Oracle® Endeca Information Discovery Integrator

Integrator ETLデザイナー・ガイド

バージョン3.1.0 • 2013年10月

ORACLE

著作権情報および免責条項

CloverETL Designer

User's Guide

Release 3.4

CloverETL

CloverETL Designer: ユーザーズ・ガイド

このユーザーズ・ガイドは、CloverETL Designer 3.4.xリリースを対象にしています。

著者: Tomas Waller、Miroslav Stys他

リリース3.4

発行日: 2013年1月

Javlin

www.cloveretl.com

www.javlininc.com

フィードバックの連絡先:

このドキュメントに関するご意見またはご提案は、docs@cloveretl.comまで電子メールでお送りください。

目次

第I部 CloverETLの概要

第1章 CloverETLの製品ファミリ

この章では、CloverETLソフトウェアの3つの製品である**CloverETL Designer**、**CloverETL Engine**および**CloverETL Server**の概要を示します。

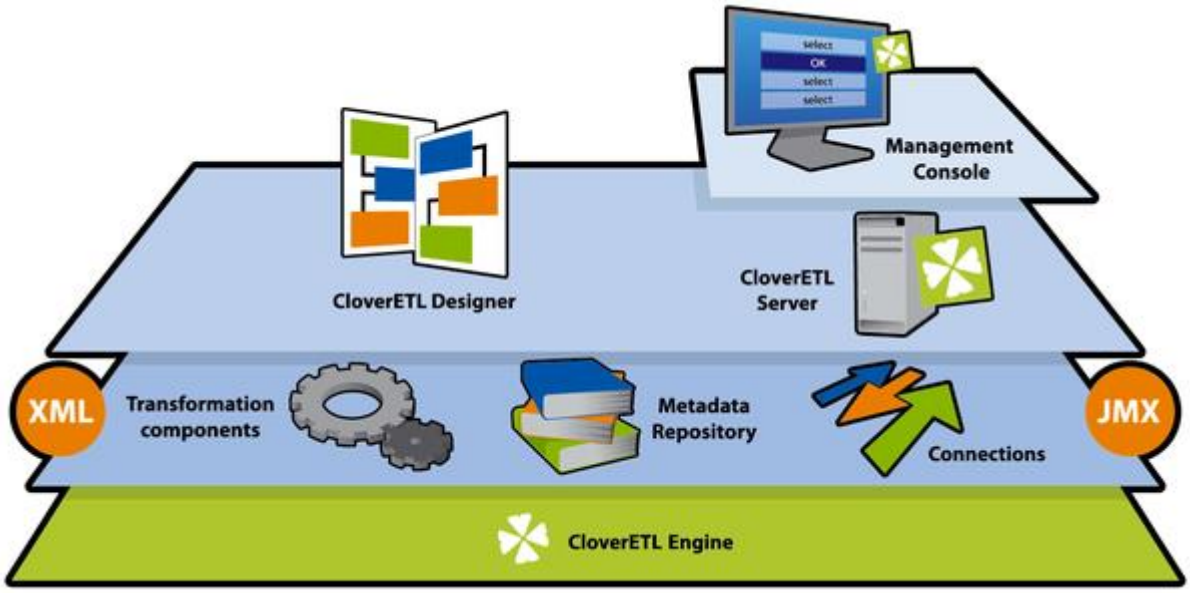


図1.1. CloverETLの製品ファミリ

CloverETL Designer

CloverETL Designerは、Javlin社が開発した**CloverETL**ソフトウェア製品ファミリの構成要素です。データを抽出、変換およびロードするための強力なJavaベースのスタンドアロン・アプリケーションです。

CloverETL Designerは、拡張可能な**Eclipse**プラットフォーム上に構築されています。www.eclipse.orgを参照してください。

CloverETL Designerを使用すると、データ解析用のコードを記述するよりも作業が大幅に簡素化されます。そのグラフィカル・ユーザー・インタフェースにより、グラフをより容易かつ快適に作成および実行できます。

また、**CloverETL Designer**は**CloverETL Server**との連携も容易です。これら2つの製品は完全に統合されています。**CloverETL Designer**を使用すると、**CloverETL Server**と接続および通信し、**CloverETL Designer**のみをローカルで使用して作業する場合と同じ方法で、プロジェクト、グラフおよびその他のすべてのリソースを**CloverETL Server**上で作成できます。

詳細は、第2章「CloverETL DesignerとCloverETL Serverの統合」(p.4)を参照してください。

CloverETL Engine

CloverETL Engineは、Javlin社が開発した**CloverETL**ソフトウェア製品ファミリの基盤となる要素です。
CloverETL Engineは、**CloverETL Designer**で作成した変換グラフを実行するランタイム・レイヤーです。

変換グラフは**CloverETL Designer**でグラフ要素から作成され、**CloverETL Engine**によって実行されます。

CloverETL Engineは、他のJavaアプリケーションに埋め込むことができるJavaライブラリです。

CloverETL Server

CloverETL Serverは、Javlin社が開発した**CloverETL**ソフトウェア製品ファミリの最も新しい構成要素です。**CloverETL Server**もJavaに基づきます。

CloverETL Designerは**CloverETL Server**と連携して使用できます。これら2つの製品は完全に統合されています。**CloverETL Designer**を使用して**CloverETL Server**と接続および通信することにより、標準の**CloverETL Designer**のみをローカルで使用して作業する場合と同じ方法で、プロジェクトやグラフをはじめとするすべてのリソースを**CloverETL Server**で作成できます。

詳細は、第2章「CloverETL DesignerとCloverETL Serverの統合」(p.4)を参照してください。

CloverETL Serverを使用して、次のことを実現できます。

- ETLジョブの集中管理
- 企業ワークフローへの統合
- マルチユーザー環境
- 複数グラフのパラレル実行
- 複数グラフの実行のトラッキング
- タスクのスケジュール
- グラフのクラスタ化および分散実行
- 起動サービス
- ロード・バランシングおよびフェイルオーバー

追加情報の入手方法

この**ユーザーズ・ガイド**以外に、次のサイトで追加情報を参照できます。

- **CloverETL Designer**の使用方法の基本を簡潔に説明する**クイック・スタート・ガイド**
  www.cloveretl.com/documentation/quickstart
- **CloverETL**製品の様々な領域に関する**FAQ**
  www.cloveretl.com/faq
- **CloverETL**の機能の詳細に関する**フォーラム**
  http://forum.cloveretl.com
- **CloverETL**製品に基づく興味深いソリューションについて説明する**ブログ**
  http://blog.cloveretl.com
- **CloverETL**ツールに関する情報が含まれている**Wiki**ページ
  http://wiki.cloveretl.com

サポート

前述の各サイトに加えて、Javlin社では充実したサポート・オプションを用意しています。この**テクニカル・サポート**は、ユーザーの時間を節約し、最高レベルのパフォーマンス、信頼性および稼働時間が実現されるように支援することを目的にしています。**CloverCare**は、主に米国およびヨーロッパ向けとなっています。

www.cloveretl.com/services/clovercare-support

第2章 CloverETL DesignerとCloverETL Serverの統合

CloverETL Designerと**CloverETL Server**が完全に統合されたため、手動で相互にコピーしなくても、**Designer**から**Server**のサンドボックスに直接アクセスできるようになりました。

すべてのデータ転送は**Designer**によって自動的に処理されるため、グラフの編集、Serverでのグラフの実行、データ・ファイルやメタデータなどの編集を直接実行できます。**Server**で実行されるグラフの実行のライブ・トラッキングを表示することもできます。

重要

この機能は、**Eclipse 3.5**以上およびJava 1.6.4以上でのみ動作します。

また、**CloverETL Designer**のバージョン3.0は**CloverETL Server**のバージョン3.0でのみ機能し、その逆も同様です。

CloverETL Serverには、**CloverETL Designer**で**CloverETL Serverプロジェクト**を作成することによって接続できます。詳細は、CloverETL Serverプロジェクト(p.27)を参照してください。

CloverETL Serverのサンドボックスと標準の**CloverETL**プロジェクトとの間でグラフやメタデータなどを交換する方法は、次のリンクを参照してください。

- CloverETL Serverサンドボックスからのインポート(p.74)
- CloverETL Serverサンドボックスへのエクスポート(p.81)

CloverETL Serverの**ユーザーズ・ガイド**は次のリンクにあります。
http://server-demo-ec2.cloveretl.com/clover/docs/index.html

CloverETL Serverプロジェクトの作成(基本原則)

1. 初めに、**CloverETL Server**にサンドボックスが存在する必要があります。**CloverETL Server**の各サンドボックスに対し、同じワークスペース内に**CloverETL Server**プロジェクトを1つのみ作成できます。1つの**CloverETL Server**サンドボックスに対して複数の**CloverETL Server**プロジェクトを作成するには、これらのプロジェクトをそれぞれ異なるワークスペースに置く必要があります。

2. **Designer**を使用すると、1つのワークスペース内に、より多くの**CloverETL Server**プロジェクトを作成できます。

   これらの**CloverETL Server**プロジェクトを、それぞれ異なる**CloverETL Server**にリンクすることもできます。

3. **CloverETL Designer**は、HTTP/HTTPSプロトコルを使用して**CloverETL Server**に接続します。これらのプロトコルは、複雑なネットワーク設定およびファイアウォールにも対応します。**CloverETL Server**へのそれぞれの接続はワークスペースに保存されます。このため、1つのワークスペースで使用できるプロトコルは1つのみとなります。ユーザーは自分のログイン名、パスワード、および指定されたユーザー権限またはキー(あるいはその両方)を持ちます。

4. 複数のユーザーが同じサンドボックスに(**Designer**を介して)アクセスしている場合は、同じリソース(グラフなど)に加えられたそれぞれの変更を上書きしないように連携する必要があります。1人が**CloverETL Server**でグラフなどのリソースを変更した場合に、他のユーザーが**Server**でそのリソースを上書きできます。ただし、警告が表示されるため、各ユーザーは**CloverETL Server**のそのようなリソースを上書きするかどうかを判断する必要があります。リモート・リソースはロックされないため、ユーザーはこのような競合が発生した場合にどうするかを判断する必要があります。

5. **CloverETL Designer**を再起動すると、すべての**CloverETL Server**プロジェクトが表示されますが、どれも閉じられています。これらを開くには2つの方法があります。

   - プロジェクトをダブルクリックします。
   - プロジェクトを右クリックし、コンテキスト・メニューから「**Open project**」を選択します。

   詳細は、「CloverETL Serverプロジェクトを開く」(p.5)を参照してください。

CloverETL Serverプロジェクトを開く

CloverETL Designerを起動すると、次のように表示されます。

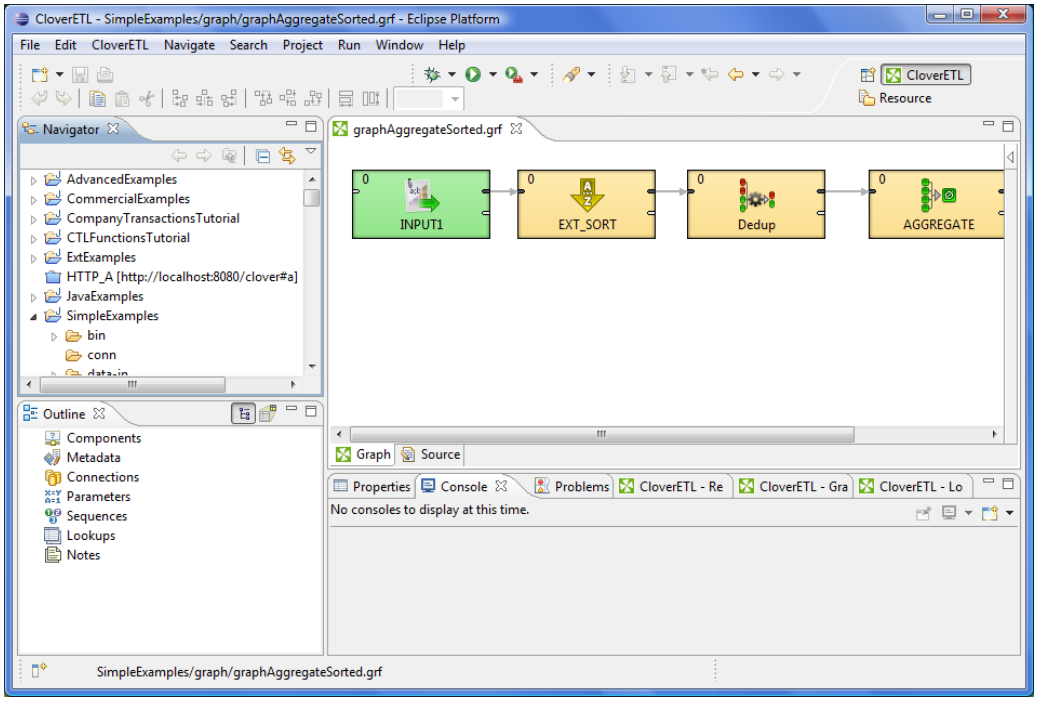


図2.1. CloverETL Designerを開いた後に表示されるCloverETL Serverプロジェクト

CloverETL Serverのプロジェクト名の横に、**Server**のURLとサンドボックスのIDがハッシュで区切って表示されます。**CloverETL Server**プロジェクトは閉じられます。これらのプロジェクトを開くには、前述の手順を実行します。つまり、プロジェクトをダブルクリックするか、コンテキスト・メニューから「**Open project**」を選択します。

ユーザーIDおよび**パスワード**を入力するプロンプトが表示される場合があります。**パスワード**を保存する必要があるかどうかも選択します。

図2.2. CloverETL Serverプロジェクトを開くためのプロンプト

その後、「**OK**」をクリックし、該当する**CloverETLServerプロジェクト**をクリックします。

図2.3. CloverETL Serverプロジェクトを開く

ウィルス対策アプリケーションによる問題が発生した場合は、HTTP/HTTPS接続の設定に例外を追加します。たとえば、`*clover*`をマスクとして使用して、**CloverETL Server**または**CloverETL**のWebページへの接続を許可できます。

HTTPを介した接続

`http`を介して接続するために、たとえば、**Tomcat**を使用して**CloverETL Server**をインストールできます。

このことを行うには、**CloverETL Server**の`clover.war`ファイルおよび`clover_license.war`ファイルを**Tomcat**の`webapps`サブディレクトリにコピーし、**Tomcat**の`bin`サブディレクトリにある`startup`スクリプトを実行して**Server**を実行します。**Tomcat**が起動すると、その`webapps`サブディレクトリ内で、これら2つのファイルが展開されます。

HTTP接続では、**CloverETL Designer**の構成は必要ありません。**CloverETL Designer**を起動するのみです。この**Designer**では、**Server**へのデフォルトの接続`http://localhost:8080/clover`を使用して、**CloverETL Server**プロジェクトを作成できます。ログイン名およびパスワードはどちらも`clover`です。

HTTPSを介した接続

`https`を介して接続するために、たとえば、**Tomcat**を使用して**CloverETL Server**をインストールできます。

最初の手順としては、**CloverETL Server**の`clover.war`ファイルおよび`clover_license.war`ファイルを**Tomcat**の`webapps`サブディレクトリにコピーし、**Tomcat**の`bin`サブディレクトリにある`startup`スクリプトを実行してServerを実行します。**Tomcat**が起動すると、その`webapps`サブディレクトリ内で、これら2つのファイルが展開されます。

Serverと**Designer**の両方を構成する(Designerに独自の証明書がある場合)か、Serverのみを構成する(Designerに証明書がない場合)必要があります。

Designerに独自の証明書がある場合

Designerに独自の証明書が必要な場合に、`https`を介して**CloverETL Server**に接続するには、クライアントおよびサーバーのキーストアおよびトラストストアを作成します。

これらのキーを生成するには、`keytool`が置かれているJDKまたはJREの`bin`サブディレクトリで、次のスクリプト(UNIX用バージョン)を実行します。

```
# SERVER
# create server key-store with private-public keys
keytool  -genkeypair -alias server -keyalg RSA -keystore ./serverKS.jks \
         -keypass semafor -storepass semafor -validity 900 \
         -dname "cn=localhost, ou=ETL, o=Javlin, c=CR"
# exports public key to separated file
keytool  -exportcert -alias server -keystore serverKS.jks \
```

```
        -storepass semafor -file server.cer

# CLIENT
# create client key-store with private-public keys
keytool  -genkeypair -alias client -keyalg RSA -keystore ./clientKS.jks \
        -keypass chodnik -storepass chodnik -validity 900 \
        -dname "cn=Key Owner, ou=ETL, o=Javlin, c=CR"
# exports public key to separated file
keytool  -exportcert -alias client -keystore clientKS.jks \
        -storepass chodnik -file client.cer

# trust stores

# imports server cert to client trust-store
keytool  -import -alias server -keystore clientTS.jks \
        -storepass chodnik -file server.cer

# imports client cert to server trust-store
keytool  -import -alias client -keystore serverTS.jks \
        -storepass semafor -file client.cer
```

(これらのコマンドに含まれるlocalhostは使用する**CloverETL Server**のデフォルト名です。他の**Server**の名前を使用するには、コマンドのlocalhostの名前を他のホスト名に置き換えます。**)**

その後、serverKS.jksファイルおよびserverTS.jksファイルを**Tomcat**のconfサブディレクトリにコピーします。

続いて、このconfサブディレクトリ内のserver.xmlファイルに次のコードをコピーします。

```
<Listener  className="org.apache.catalina.core.AprLifecycleListener"
           SSLEngine="off" />


           <Connector port="8443" maxHttpHeaderSize="7192"
                 maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
                 enableLookups="false" disableUploadTimeout="true"
                 acceptCount="100" scheme="https" secure="true"
                 clientAuth="true" sslProtocol="TLS"
                 SSLEnabled="true"
                 protocol="org.apache.coyote.http11.Http11NioProtocol"
                 keystoreFile="pathToTomcatDirectory/conf/serverKS.jks"
                 keystorePass="semafor"
                 truststoreFile="pathToTomcatDirectory/conf/serverTS.jks"
                 truststorePass="semafor"
           />
```

これで、**Tomcat**のbinサブディレクトリにあるstartupスクリプトを実行することにより、**CloverETL Server**を実行できるようになります。

CloverETL Designerの構成

次に、clientKS.jksファイルおよびclientTS.jksファイルを任意の場所にコピーする必要があります。

その後、次のコードを、eclipseディレクトリに保存されているeclipse.iniファイルの最後にコピーします。

```
-Djavax.net.ssl.keyStore=locationOfClientFiles/clientKS.jks
-Djavax.net.ssl.keyStorePassword=chodnik
-Djavax.net.ssl.trustStore=locationOfClientFiles/clientTS.jks
-Djavax.net.ssl.trustStorePassword=chodnik
```

これにより、**CloverETL Designer**を起動したときに、**Server**へのデフォルト接続`https://localhost:8443/clover`を使用して、**CloverETL Serverプロジェクト**を作成できるようになります。ログイン名およびパスワードはどちらも`clover`です。

Designerに独自の証明書がない場合

Designerに独自の証明書が必要でない場合、`https`を介して**CloverETL Server**に接続するには、サーバーのキーストアの作成のみが必要となります。

このキーを生成するには、`keytool`が置かれているJDKまたはJREの`bin`サブディレクトリで、次のスクリプト(UNIX用バージョン)を実行します。

```
keytool -genkeypair -alias server -keyalg RSA -keystore ./serverKS.jks \
        -keypass semafor -storepass semafor -validity 900 \
        -dname "cn=localhost, ou=ETL, o=Javlin, c=CR"
```

(これらのコマンドに含まれる`localhost`は使用する**CloverETL Server**のデフォルト名です。他の**Server**の名前を使用するには、コマンドの`localhost`の名前を他のホスト名に置き換えます。)

その後、`serverKS.jks`ファイルを**Tomcat**の`conf`サブディレクトリにコピーします。

続いて、この`conf`サブディレクトリ内の`server.xml`ファイルに次のコードをコピーします。

```
<Listener className="org.apache.catalina.core.AprLifecycleListener"
          SSLEngine="off" />


          <Connector port="8443" maxHttpHeaderSize="7192"
                maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
                enableLookups="false" disableUploadTimeout="true"
                acceptCount="100" scheme="https" secure="true"
                clientAuth="false" sslProtocol="SSL"
                SSLEnabled="true"
                protocol="org.apache.coyote.http11.Http11NioProtocol"
                keystoreFile="pathToTomcatDirectory/conf/serverKS.jks"
                keystorePass="semafor"
          />
```

これで、**Tomcat**の`bin`サブディレクトリにある`startup`スクリプトを実行することにより、**CloverETL Server**を実行できるようになります。

これにより、**CloverETL Designer**を起動したときに、**Server**へのデフォルト接続`https://localhost:8443/clover`を使用して、**CloverETL Serverプロジェクト**を作成できるようになります。ログイン名およびパスワードはどちらも`clover`です。

Serverの証明書を受け入れるためのプロンプトが表示されます。その後、**CloverETL Serverプロジェクト**の作成が許可されます。

プロキシ・サーバーを介した接続

プロキシ・サーバーを使用してClover Serverに接続することもできます。

重要

プロキシ・サーバーでHTTP 1.1がサポートされている必要があります。そうでない場合は、すべての接続の試行が失敗します。

接続を管理するには、「**Window**」→「**Preferences**」→「**General**」→「**Network Connections**」とナビゲートします。

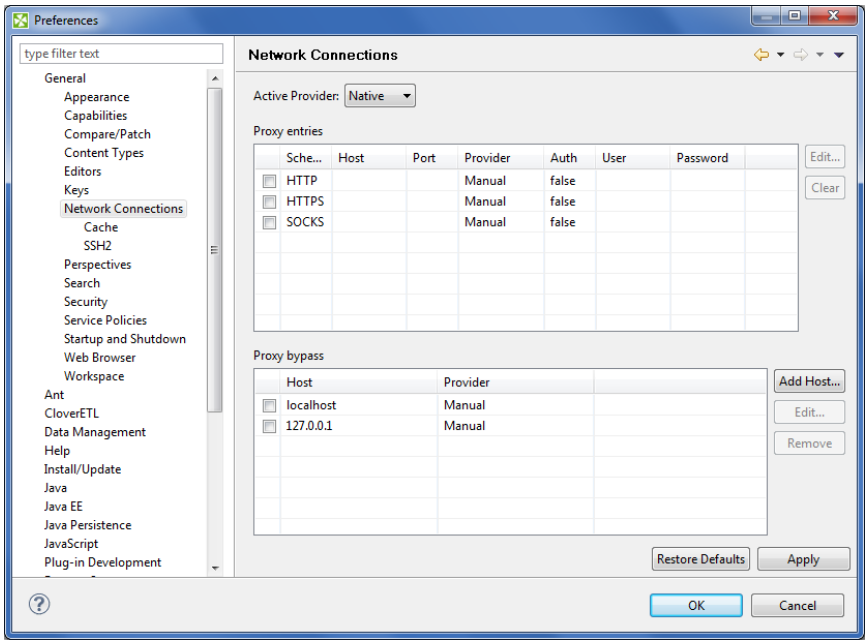


図2.4.「Network connections」ウィンドウ

プロキシ設定の操作の詳細は、EclipseのWebサイトを参照してください。

第II部　インストール手順

第3章 CloverETL Designerのシステム要件

CloverETL Designerは3つの形式で配布されます。

1. **完全インストール**は推奨手順であり、必要な環境およびアプリケーションがすべて含まれているため、必要な前提条件はありません。
2. **オンライン・プラグイン**は、次に説明するすべての前提条件がコンピュータに存在する場合に適しています。**CloverETL Designer**は、対応するCloverETL更新サイトを使用してオンラインでインストールされます。
3. **オフライン・プラグイン**は、オンライン・プラグインと同じ条件で適用可能です。ただし、この場合、**CloverETL Designer**は、以前にダウンロードされたアーカイブ・ファイルを実行することによってオフラインでインストールされます。

CloverETLを実行するには、次の要件を満たす必要があります。

- サポートされるOS: Microsoft Windows 32ビット、Microsoft Windows 64ビット、Linux 32ビット、Linux 64ビット、Mac OS X Cocoa
- 512MB以上のRAM

完全インストール

- **ソフトウェア要件:**
  - Microsoft Windows: なし。Eclipse Platform 3.6.2 for Java developersおよびRSE + GEF + Eclipse Web Tools Platformがインストーラに含まれています。
  - Mac OS XおよびLinux: Java 6 Runtime Environment以上(Java 7 Development Kitを推奨)

プラグイン・インストール

- **ソフトウェア要件:**
  - 最小要件: Eclipse Platform 3.6.2 + GEF。Java 6 Runtime Enviroment。
  - 推奨要件: Eclipse Platform 3.6.2およびRSE + GEF + Eclipse Web Tools Platform。Java 7 Development Kit。
  - Eclipse 3.7は完全にサポートされています。

重要

Mac OSユーザーの場合:

デフォルトのJavaシステムが1.7以上に設定されていることを確認してください。**「Finder」→「アプリケーション」→「ユーティリティ」→「Java」→「Java preferences」**と進み、使用可能なJavaのインストールの順序を変更して、Java 7がリストの最初に来るようにします。

関連リンク

- Eclipse Classicのダウンロード・ページ: 使用するOSに適したバージョンを選択してください。

  http://www.eclipse.org/downloads

- JDKのダウンロード・ページ: CloverではJava 1.6以上がサポートされています。

  http://www.oracle.com/technetwork/java/javase/downloads/index.html

- Java Development Kit (JDK)とJava SE Runtime Environment (JRE)の違いを理解するために役立つリンク。

  http://docs.oracle.com/javase/7/docs/

  http://www.oracle.com/technetwork/java/javase/webnotes-136672.html

第4章 CloverETLのダウンロード

CloverETL Designer Editionは2通りの方法でダウンロードできます。

1. www.cloveretl.com/userでサインインした後、自分のユーザー・アカウントまたは顧客アカウントからダウンロードします。

   推奨されるインストール方法です。

2. 使用するプラットフォームのための直接HTTPダウンロード・リンクを使用します。

   - 完全インストールのリンクは、オペレーティング・システムごとに異なります。
   - プラグイン・インストールのリンク(オンラインおよびオフラインの両方)は、各エディションにおいてサポートされているすべてのオペレーティング・システムで共通です。

CloverETL Desktop Edition

顧客アカウントへのログインには、取得したログイン・データ(電子メール・アドレスとパスワード)とともにライセンス資格証明を使用します。後の手順で、ライセンス番号とパスワードを入力してダウンロードを開始する必要があります。顧客プロファイルからのインストールの詳細は、www.cloveretl.com/resources/installation-guideを参照してください。

次の表に、直接ダウンロード用のHTTPリンクを示します。

完全インストールの入手

OS	ダウンロード・サイト
Windows 32ビット	designer.cloveretl.com/update/cloveretl-designer-win32.exe
Windows 64ビット	designer.cloveretl.com/update/cloveretl-designer-win32-x86_64.exe
Linux 32ビット	designer.cloveretl.com/update/cloveretl-designer-linux-gtk.tar.gz
Linux 64ビット	designer.cloveretl.com/update/cloveretl-designer-linux-gtk-x86_64.tar.gz
Mac OS Cocoa	designer.cloveretl.com/update/cloveretl-designer-macosx-cocoa-x86_64.dmg.zip

Eclipseプラグインの入手

プラグイン	ダウンロード・サイト
オンライン	designer.cloveretl.com/update
オフライン	designer.cloveretl.com/update/cloveretl-designer.zip

プラグインのインストール手順は、第6章「DesignerのEclipseプラグインとしてのインストール」(p.17)を参照してください。

CloverETL Desktop Trial Edition

トライアル版を入手するには、CloverETLの会社サイト(www.cloveretl.com/user/registration)でユーザー・アカウントを作成します。ログイン名とパスワードを取得して登録を確認すると、そのユーザー・アカウントに、**CloverETL Desktop Trial Edition**のダウンロードへの期限付きアクセス権が付与されます。インストールの詳細は、www.cloveretl.com/resources/installation-guideを参照してください。

完全インストールの入手

OS	ダウンロード・サイト
Windows 32ビット	designer.cloveretl.com/eval-update/cloveretl-designer-eval-win32.exe
Windows 64ビット	designer.cloveretl.com/eval-update/cloveretl-designer-eval-win32-x86_64.exe
Linux 32ビット	designer.cloveretl.com/eval-update/cloveretl-designer-eval-linux-gtk.tar.gz
Linux 64ビット	designer.cloveretl.com/eval-update/cloveretl-designer-eval-linux-gtk-x86_64.tar.gz
Mac OS Cocoa	designer.cloveretl.com/eval-update/cloveretl-designer-eval-macosx-cocoa-x86_64.dmg.zip

Eclipseプラグインの入手

プラグイン	ダウンロード・サイト
オンライン	designer.cloveretl.com/eval-update
オフライン	designer.cloveretl.com/eval-update/cloveretl-designer-eval.zip

Eclipseのソフトウェア更新サイトのメカニズムを使用して**CloverETL Designer**をインストールする方法の詳細は、第6章「DesignerのEclipseプラグインとしてのインストール」(p.17)を参照してください。

CloverETL Community Edition

CloverETL Community Editionを入手するには、www.cloveretl.com/user/registrationでユーザー・アカウントを作成します。折り返し送信されるログイン名とパスワードにより、ダウンロードおよびインストールが認可されます。

完全インストールの入手

OS	ダウンロード・サイト
Windows 32ビット	designer.cloveretl.com/community-update/cloveretl-designer-community-win32.exe
Windows 64ビット	designer.cloveretl.com/community-update/cloveretl-designer-community-win32-x86_64.exe
Linux 32ビット	designer.cloveretl.com/community-update/cloveretl-designer-community-linux-gtk.tar.gz
Linux 64ビット	designer.cloveretl.com/community-update/cloveretl-designer-community-linux-gtk-x86_64.tar.gz
Mac OS Cocoa	designer.cloveretl.com/community-update/cloveretl-designer-community-macosx-cocoa-x86_64.dmg.zip

Eclipseプラグインの入手

プラグイン	ダウンロード・サイト
オンライン	designer.cloveretl.com/community-update
オフライン	designer.cloveretl.com/community-update/cloveretl-designer-community.zip

プラグインのインストール手順は、第6章「DesignerのEclipseプラグインとしてのインストール」(p.17)を参照してください。

第5章 CloverETL Designerの起動

CloverETL Designerを起動すると、次の画面が表示されます。

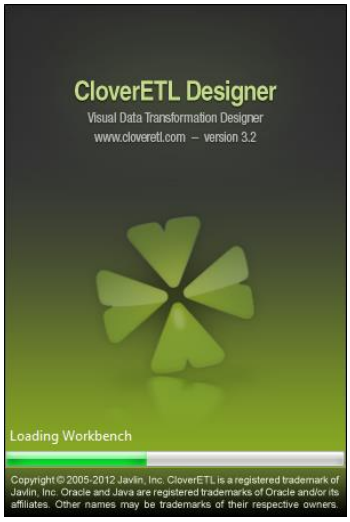


図5.1. CloverETL Designerのスプラッシュ画面

CloverETL Designerが起動した後、最初に、`workspace`フォルダの定義をプロンプトで要求されます。この場所にプロジェクトが保存されます。通常は、ユーザーの`home`ディレクトリ(`C:\Users\your_name\workspace`または `/home/your_name/CloverETL/workspace`など)内のフォルダになります。

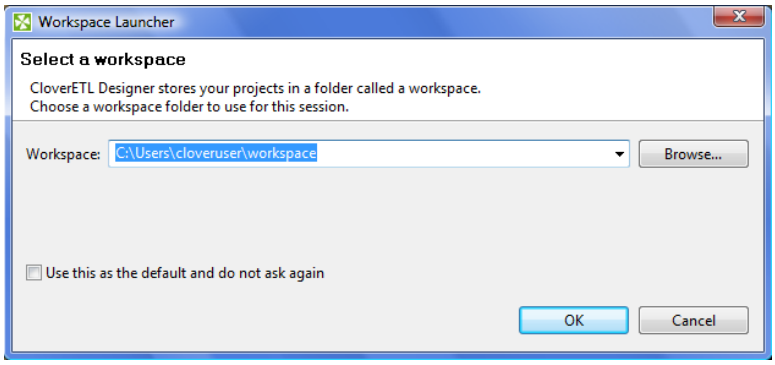


図5.2. ワークスペース選択ダイアログ

`workspace`は任意の場所に配置できます。したがって、その場所に対する適切な権限があることを確認してください。指定したフォルダが存在しない場合は作成されます。

`workspace`を設定すると、ようこそ画面が表示されます。

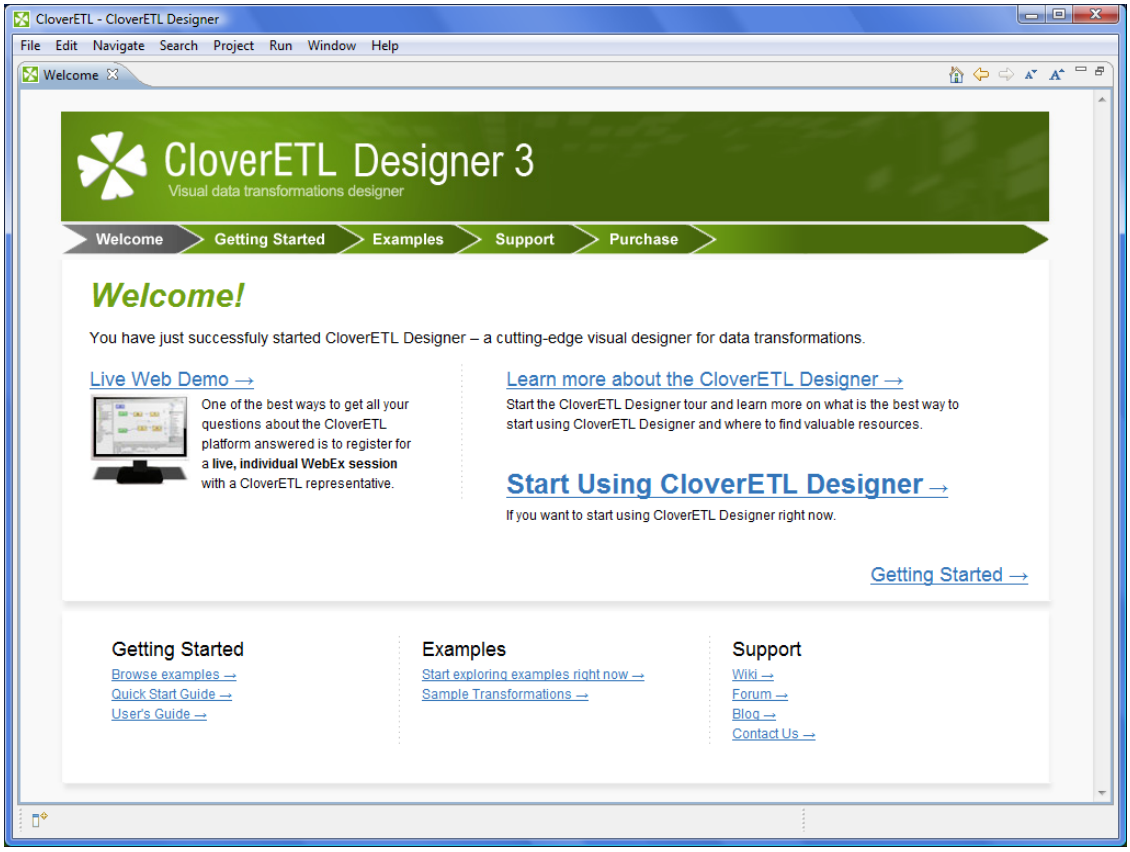


図5.3. CloverETL Designerの導入画面

CloverETL Designerで実行する最初の手順は、第8章「CloverETLプロジェクトの作成」(p.26)を参照してください。

将来、必要に応じて、製品リソースにオンラインでアクセスしたり、自分のCloverライセンスを管理する場合があります。最も簡単な方法は、次の図に示すように、ヘルプ・メニューにナビゲートすることです。

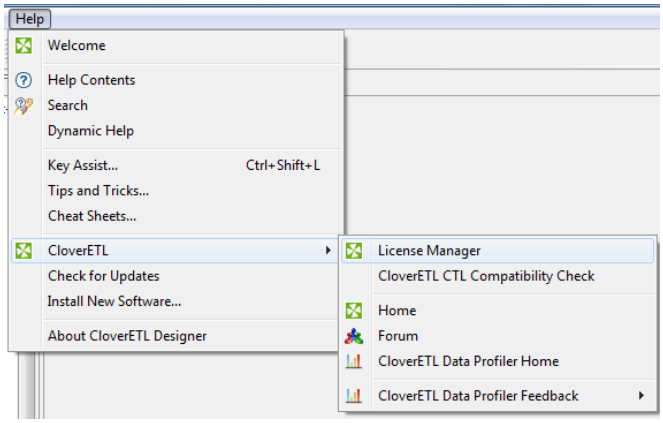


図5.4. CloverETLのヘルプ

第6章 DesignerのEclipseプラグインとしてのインストール

Eclipseでは、**CloverETL Designer**プラグインを直接インストールできます。**CloverETL Designer**プラグインのインストール時には、必要なすべてのプラグインがインストールされます。これらを個別にインストールする必要はなく、**CloverETL Designer**をダウンロードおよびインストールするのみです。

CloverETL Designerプラグインの**Eclipse**リリースへのインストールは、新しいソフトウェアのインストールとみなされます。「**Help**」→「**Install New Software...**」を選択すると、インストール可能なソフトウェア・サイトおよび項目のリストがウィザードによって表示されます。

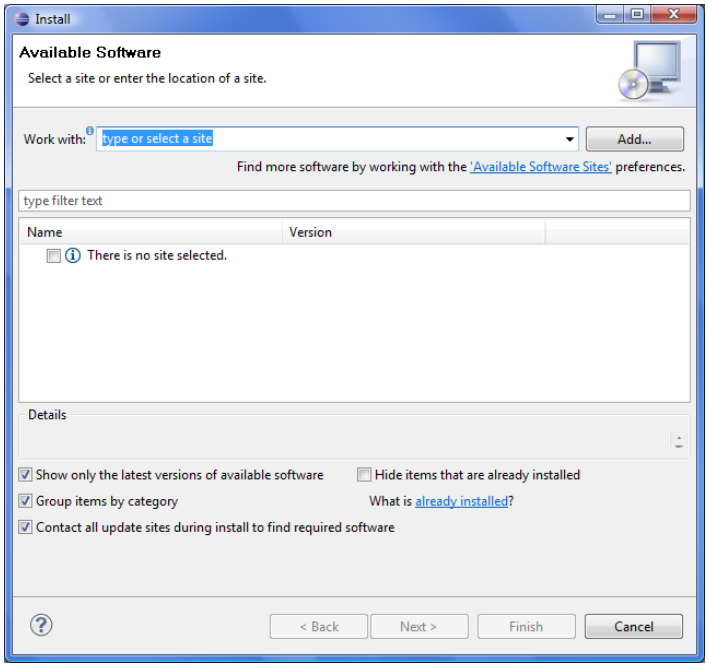


図6.1. 使用可能なソフトウェア

「**Add...**」ボタンを使用して、プラグインの場所を入力できます。**CloverETL Designer**オンライン・プラグインの更新サイトの概要を次の表に示します。

表6.1. CloverETLのサイト

CloverETL製品	更新サイト
CloverETL Desktop Edition	designer.cloveretl.com/update
CloverETL Desktop Trial Edition	designer.cloveretl.com/eval-update
CloverETL Community Edition	designer.cloveretl.com/community-update

CloverETLプラグインのインストールにアクセスするには、プロンプトにユーザー名とパスワードを入力します。**CloverETL Desktop Edition**をインストールする場合は、ユーザー名のライセンス番号を入力します。他のバージョンをインストールする場合は、ユーザー・アカウント・ログイン(自分の電子メール・アドレス)を使用します。

次の画面で**CloverETL**の項目を確認し、ライセンス契約条項に同意した後、「**Finish**」をクリックして、ダウンロードおよびインストールの続行を許可します。その後、変更を有効にするためにEclipse SDKの再起動を求められたときは「**Yes**」をクリックします。

インストールが正常に完了したことを確認するには、「**Help**」→「**About Eclipse SDK**」を選択します。**CloverETL**のロゴ が表示されます。

第III部　スタート・ガイド

第7章 ライセンス・マネージャ

この章では、CloverETL Designerで**ライセンス**を追加または削除する方法について説明します。

ライセンス・マネージャは、新しいライセンスの追加または既存のライセンスの削除や表示が容易になるように設計されています。メイン・メニューから「**Help**」→「**CloverETL**」→「**License Manager**」を選択してアクセスします。

重要

CloverETL Designer Communityでは**ライセンス・マネージャ**にアクセスできません。

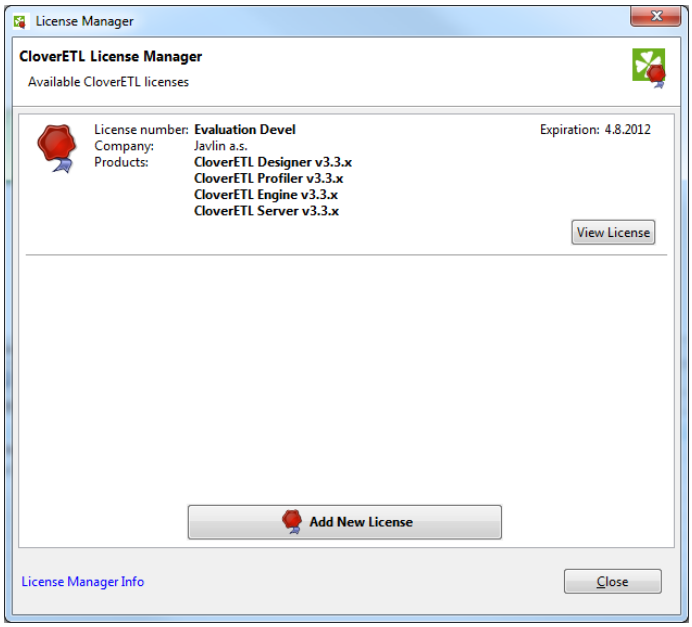


図7.1. ライセンス・マネージャによるインストール済ライセンスの表示

ライセンス・マネージャを使用して次の作業が可能です。

- 使用可能なすべてのライセンスおよびその詳細を参照します。
  - **License number**: インストールされているライセンスの番号
  - **Company**
  - **Products**: ライセンスされている製品のリスト
  - **Expiration**: ライセンスの有効期限日
- 「CloverETL License」ダイアログ(p.20)を開いて、ライセンスに関して提供されるすべての情報を表示します。
- 使用可能なライセンス・ソースを確認します。ライセンス・ソースは「**License Manager Info**」をクリックすると表示されます。
- CloverETLのライセンス・ウィザード(p.20)を開きます。このウィザードを使用して、新しいライセンスを追加できます。「**Add New License**」ボタンをクリックして、ライセンスのアクティブ化のプロセスを開始します。
- 既存のライセンスを削除します。アクティブ化されているライセンスの削除が可能な場合は、「**Remove**」ボタンが表示されます。ライセンスを削除するときには、確認を要求されます。

「CloverETL License」ダイアログ

「**CloverETL** License」ダイアログには、ライセンスに関して提供されるすべての情報が表示されます。**ライセンス条項**はここから確認できます。**ライセンス・マネージャ**(第7章「ライセンス・マネージャ」(p.19))から開くことができます。

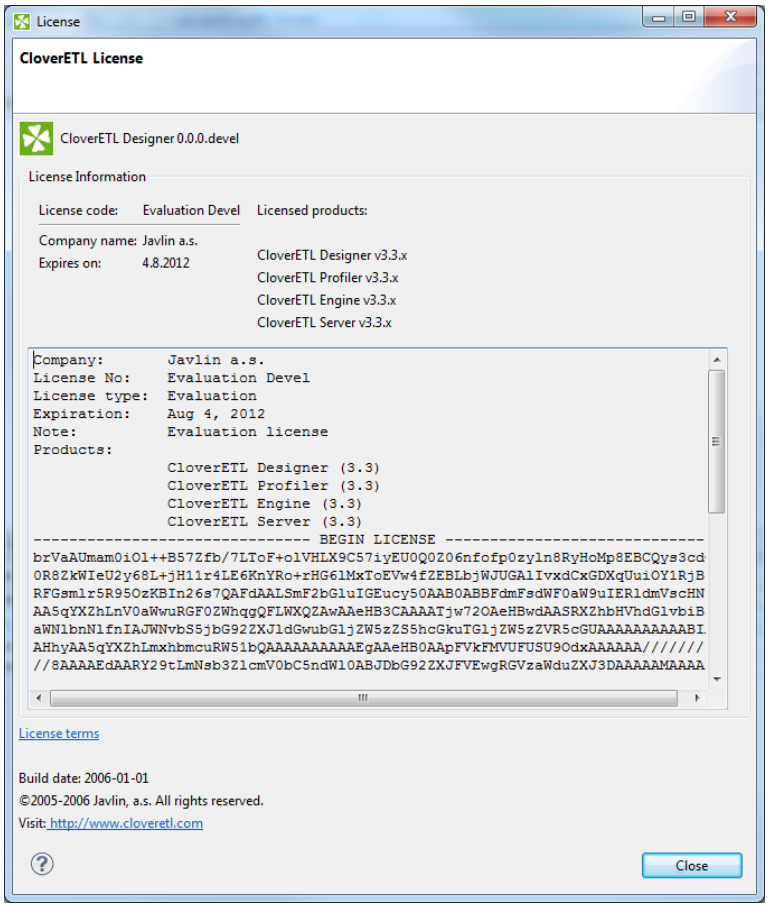


図7.2.「CloverETL License」ダイアログ

注意

ライセンスによっては、ライセンス条項へのアクセスを可能にする必要がないものがあります。

CloverETLのライセンス・ウィザード

CloverETLのライセンス・ウィザードでは、ライセンスのアクティブ化のプロセスを順を追って示します。

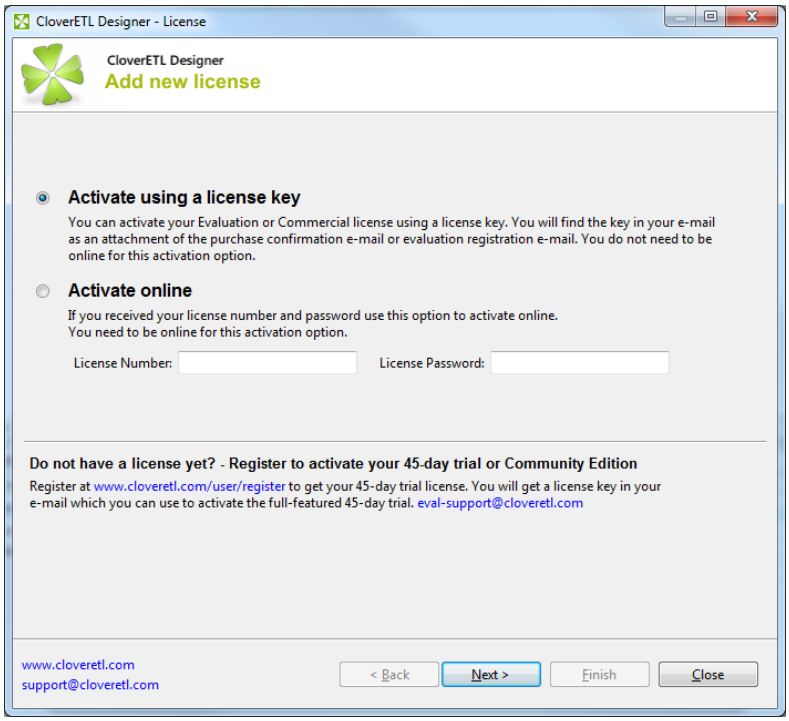


図7.3. CloverETLのライセンス・ウィザード

新しいライセンスをアクティブ化する方法は2通りあります。

- ライセンス・キーを使用したアクティブ化(p.21)

  ライセンス・キーがある場合のオフラインでのアクティブ化です。

- オンラインでのアクティブ化(p.23)

  ライセンス番号とパスワードがある場合のオンラインでのアクティブ化です。

ライセンス・キーを使用したアクティブ化

ライセンスは、ライセンス・キーを使用してアクティブ化できます。このことを選択する場合、インターネット接続は必要ありません。次の各図に、新しいライセンスのアクティブ化のプロセスを示します。

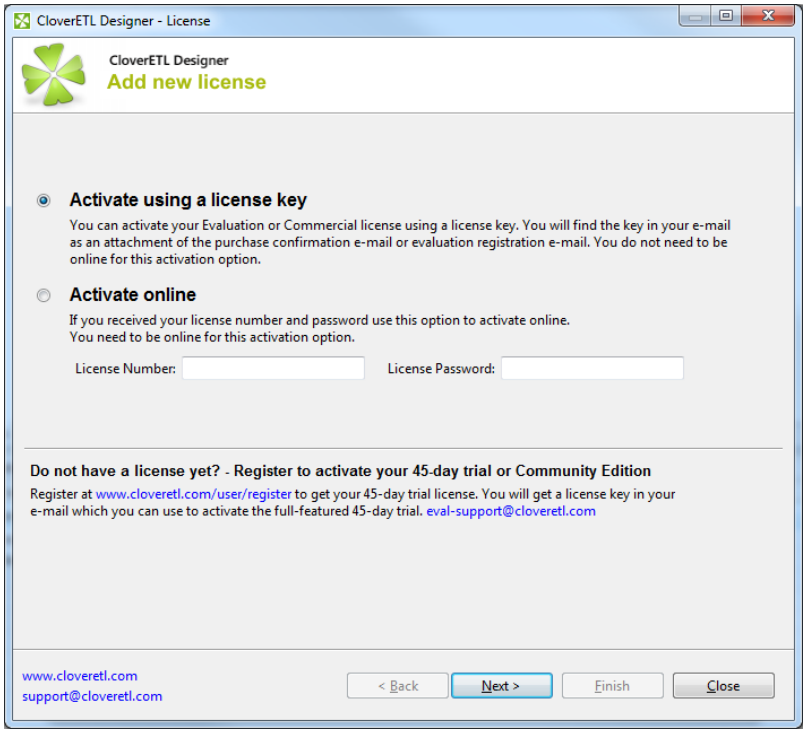


図7.4. 「Activate using license key」ラジオ・ボタンの選択および「Next」のクリック

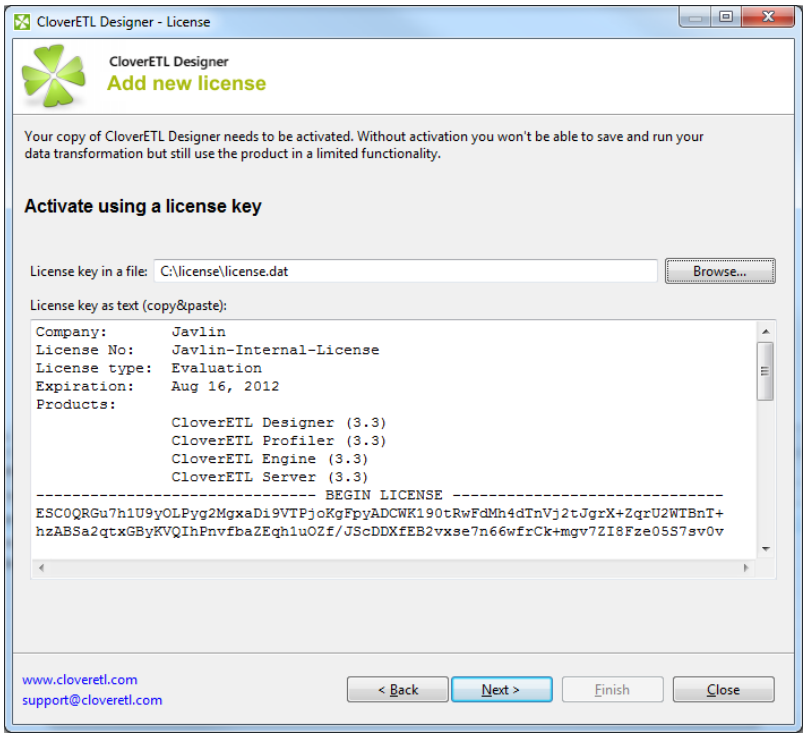


図7.5. ライセンス・ファイルのパスの入力またはライセンス・テキストのコピー・アンド・ペースト

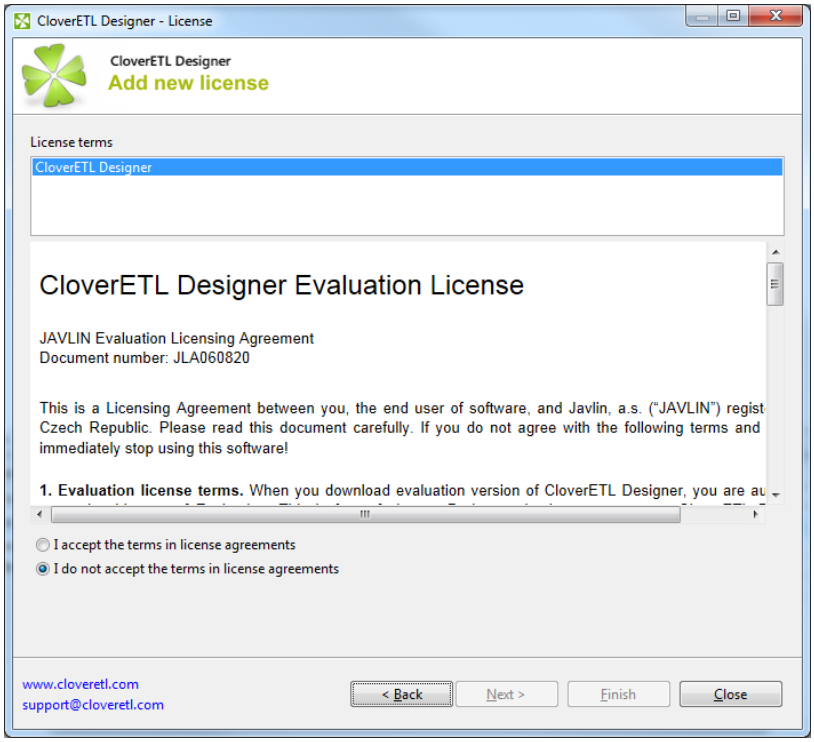


図7.6. ライセンス契約の同意確認および「Finish」ボタンのクリック

これらの手順の後、成功したライセンスのアクティブ化に関する情報ダイアログが表示されます。「OK」ボタンを押してダイアログを確認し、アクティブ化のプロセスを終了します。

注意

新しいライセンスのアクティブ化のプロセスは、**「Finish」**ボタンを押す前の任意の時点で終了できます。すでにアクティブ化されたライセンスは、**ライセンス・マネージャ**を使用して削除できます。

オンラインでのアクティブ化

新しいライセンスのオンラインでのアクティブ化には、ライセンス番号とパスワードを使用できます。この場合はインターネット接続が必要です。次の各図に、新しいライセンスのアクティブ化のプロセスを示します。

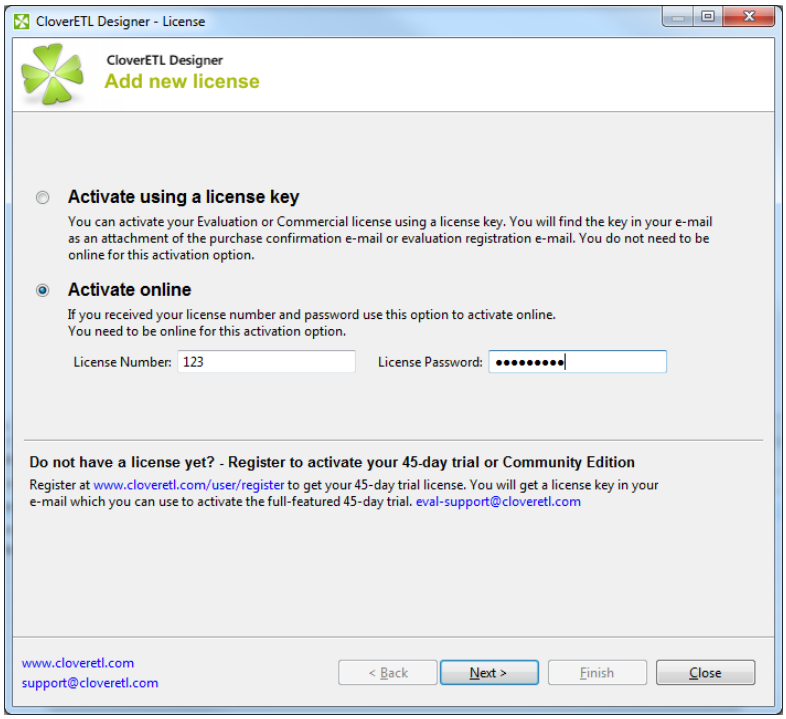


図7.7. 「Activate online」ラジオ・ボタンの選択、ライセンス番号およびパスワードの入力、「Next」のクリック

注意

入力されたパスワードまたはライセンス番号が正しくない場合はエラー・メッセージが表示されます。

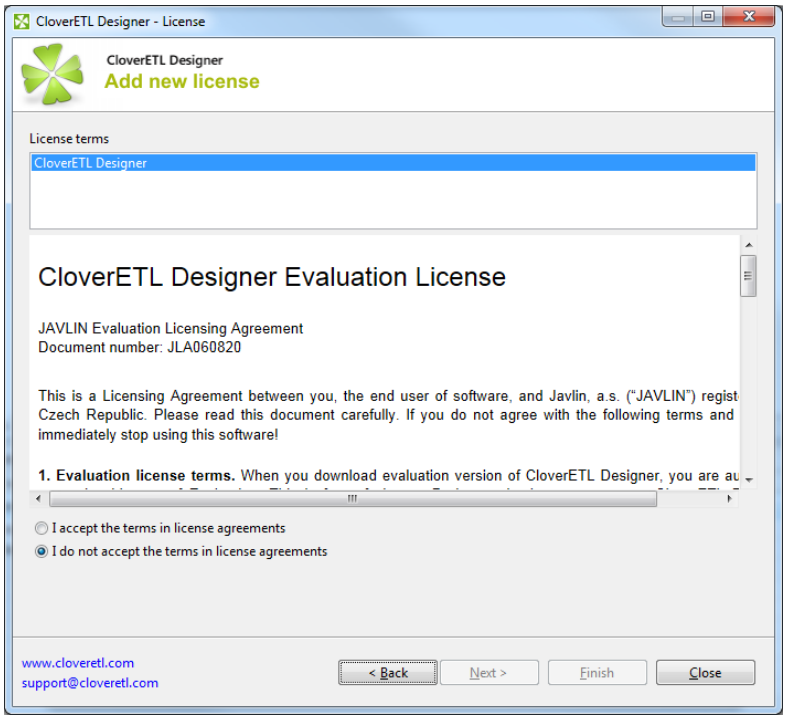


図7.8. ライセンス契約の同意確認および「Finish」ボタンのクリック

これらの手順の後、成功したライセンスのアクティブ化に関する情報ダイアログが表示されます。「OK」ボタンを押してダイアログを確認し、アクティブ化のプロセスを終了します。

注意

新しいライセンスのアクティブ化のプロセスは、**「Finish」**ボタンを押す前の任意の時点で終了できます。すでにアクティブ化されたライセンスは、**ライセンス・マネージャ**を使用して削除できます。

第8章 CloverETLプロジェクトの作成

この章では、**CloverETL**プロジェクトの作成方法を説明します。

CloverETL Designerを使用して、3種類の**CloverETL**プロジェクトを作成できます。

- CloverETLプロジェクト(p.26)

  ローカルの**CloverETL**プロジェクトです。プロジェクト構造全体がローカル・コンピュータ上に存在します。

- CloverETL Serverプロジェクト(p.27)

  CloverETL Serverのサンドボックスに対応する**CloverETL**プロジェクトです。プロジェクト構造全体が**CloverETL Server**上に存在します。

- CloverETLサンプル・プロジェクト(p.30)

  前述の2種類のプロジェクトに加えて、**CloverETL Designer**ではサンプルを含めて事前に準備された一連のローカルの**CloverETL**プロジェクトを作成できます。これらのサンプルは、**CloverETL**の機能のデモンストレーションです。

CloverETLのパースペクティブから前述のプロジェクトを作成する方法について説明します。

CloverETL Designerを完全インストールとしてインストールした場合は、すでにこのパースペクティブが示されています。

CloverETL Designerを**Eclipse**へのプラグインとしてインストールした場合は、パースペクティブを変更する必要があります。このことを行うには、右上隅のボタンをクリックして、「**Others**」メニューから「**CloverETL**」を選択します。

CloverETLプロジェクト

CloverETLパースペクティブから、「**File**」→「**New**」→「**CloverETL Project**」を選択します。

次のウィザードが開き、プロジェクト名の入力を求められます。

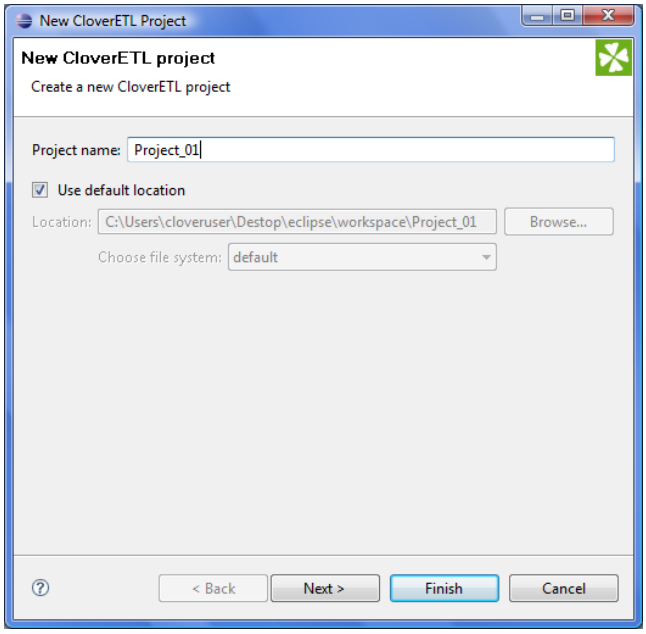


図8.1. CloverETLプロジェクトの名前付け

「**Finish**」をクリックすると、選択したローカルの**CloverETL**プロジェクトが、指定した名前で作成されます。

CloverETL Serverプロジェクト

CloverETLパースペクティブから、「**File**」→「**New**」→「**CloverETL Server Project**」を選択します。

次のウィザードが開き、**CloverETL**プロジェクトのプロパティを3つの手順で指定できます。

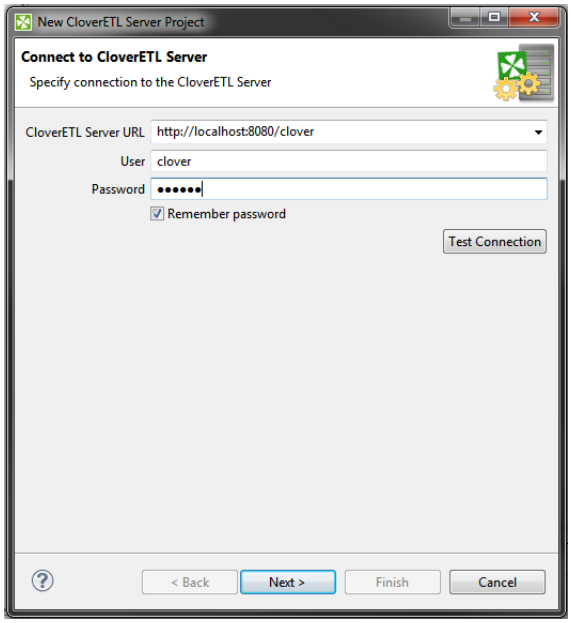


図8.2. CloverETL Serverプロジェクト・ウィザード: サーバー接続

最初の手順では、**CloverETL Server**への作業用接続を作成します。表示される「**CloverETL Server URL**」、「**User**」および「**Password**」の各テキスト・フィールドに入力します。

次に、「**Test Connection**」をクリックして、接続パラメータの妥当性を確認します。ここで「**Remember password**」チェック・ボックスを選択することによって、パスワードをPCに保存するかどうかを決定できます。**CloverETL Server**への接続が確立した後は、「**Next**」ボタンをクリックして次の手順に進むことができます。**[Return]**キーを押して、設定の確認と次の手順への移動を一度に行うこともできます。

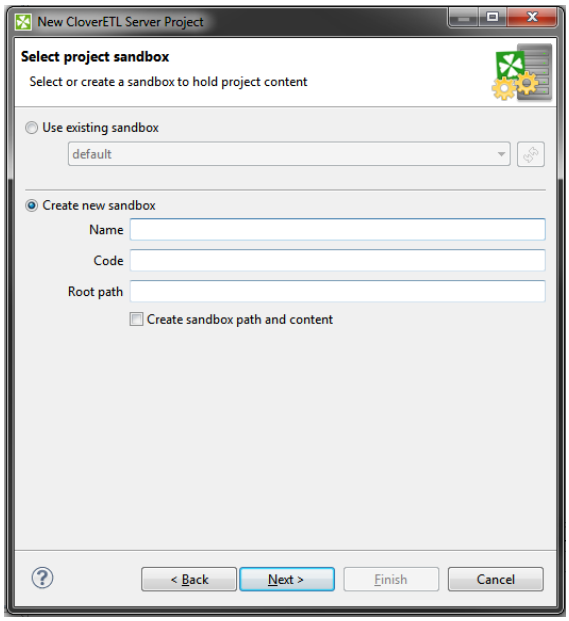


図8.3. CloverETL Serverプロジェクト・ウィザード: サンドボックスの選択

ウィザードの次の手順では、プロジェクトに対応する**CloverETL Server**のサンドボックスを選択または作成します。1つのサンドボックスは、単一のワークスペース・プロジェクトにのみ接続できます。新しいサンドボックスを作成することを選択した場合、そのフォームは**CloverETL Server**のWebインタフェースに存在するものと同様になります。サンドボックスのプロパティの詳細は、**CloverETL Server**のマニュアルを参照してください。
「**Next**」ボタンをクリックすると、ウィザードによって新しいサンドボックスが作成されます。

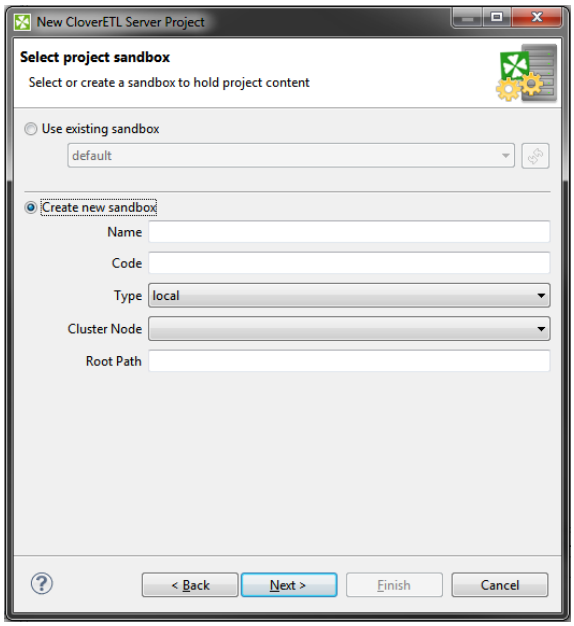


図8.4. CloverETL Serverプロジェクト・ウィザード: クラスタ化されたサンドボックスの作成

クラスタ内にデプロイされた**CloverETL Server**を使用する場合は、サンドボックス作成のフォームが異なります(CloverETL Serverプロジェクト・ウィザード: クラスタ化されたサンドボックスの作成 (p.28) 参照してください)。この場合も、サンドボックスのタイプおよび特定のプロパティの詳細は、**CloverETL Server**のマニュアルを参照してください。

注意

クラスタ化された環境では、**CloverETL Designer**プロジェクトにバインドするサンドボックスのタイプとしては、ユーザーがデータ変換を定義および実行できる`shared`が適しています。他のタイプのサンドボックスもワークスペース・プロジェクトに接続できますが、データへのアクセスおよびクラスタへの配布を目的とする場合に適しています。

最後の手順では、新しい**CloverETL Server**プロジェクトの名前を指定します。他の値(**場所**および**ファイル・システム**)は変更しないでください。

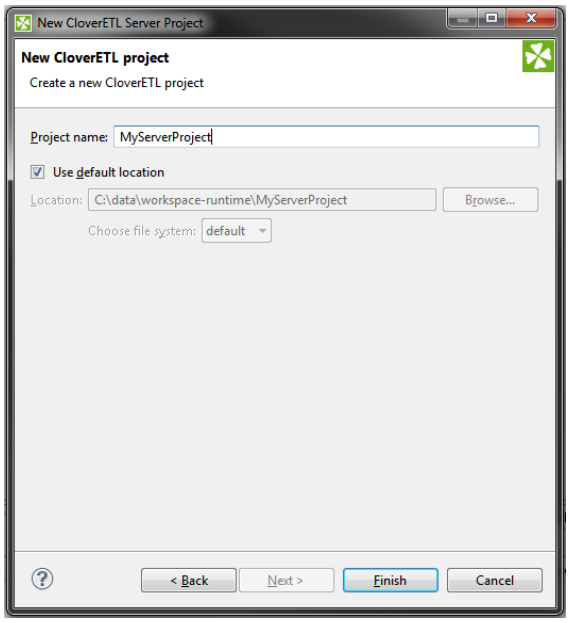


図8.5. 新しいCloverETL Serverプロジェクトの名前付け

「**Finish**」をクリックすると、**CloverETL Server**プロジェクトが作成されます。

重要

実際には、すべての**CloverETL Server**プロジェクトは、既存の**CloverETL Server**サンドボックスへの単なるリンクになります。

ただし、他のローカル・プロジェクトと同様、CloverETL Designerの「**Navigator**」ペインに表示され、ローカルで作成するかのように**CloverETL Designer**でグラフを作成できますが、存在するのは前述のサンドボックスです。また、**Designer**から実行することもできます。

CloverETL Serverプロジェクト内でグラフを作成するには、サンドボックスへの書込み権限が必要です。

CloverETLサンプル・プロジェクト

事前に準備されたサンプル・プロジェクトを作成するには、「**File**」→「**New**」→「**Others...**」を選択し、**CloverETL**カテゴリを展開して、「**CloverETL Examples Project**」を選択します。

次のウィザードが表示されます。

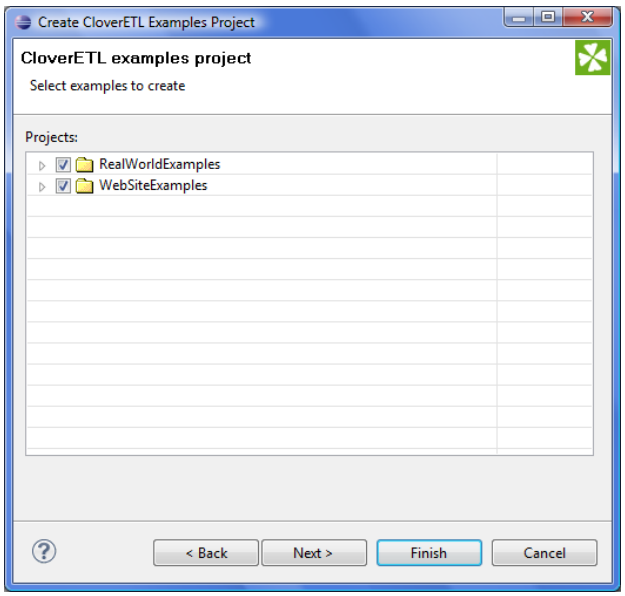


図8.6. CloverETLサンプル・プロジェクト・ウィザード

チェック・ボックスを選択することにより、任意の**CloverETL**サンプル・プロジェクトを選択できます。

「**Finish**」をクリックすると、選択したローカルの**CloverETLサンプル・**プロジェクトが作成されます。

重要

これらのプロジェクトをすでにインストールしている場合は、インストールする前に「**Next**」をクリックして名前を変更できます。その後、「**Finish**」をクリックできます。

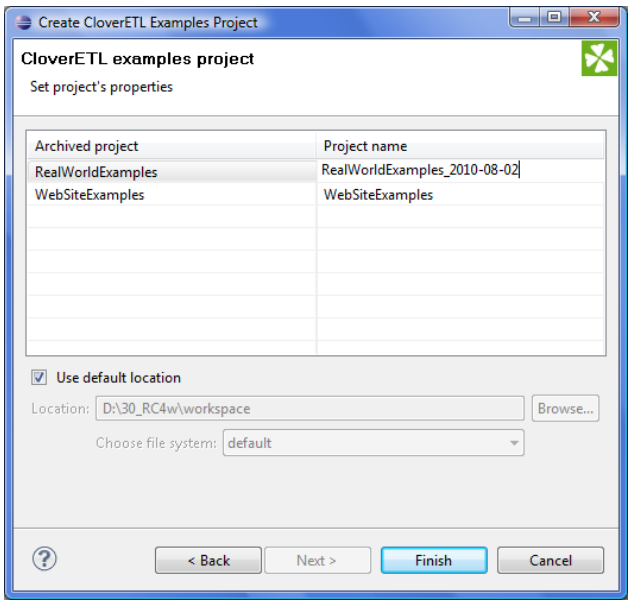


図8.7. CloverETLサンプル・プロジェクトの名前の変更

第9章 CloverETLプロジェクトの構造

この章では、**CloverETL**プロジェクトを作成するときに発生する事項の簡単な概要のみを示します。

このことは、ローカルのCloverETLプロジェクト(p.26)およびCloverETLサンプル・プロジェクト(p.30)のみでなく、CloverETL Serverプロジェクト(p.27)にも該当します。

1. すべてのCloverETLプロジェクトの標準構造(p.32)

   各**CloverETLプロジェクト**は、プロジェクトを作成するとき変更されていないかぎり、標準のプロジェクト構造になります。

2. Workspace.prmファイル(p.33)

   ローカルまたはリモート(サーバー)の各**CloverETL**プロジェクトには、プロジェクトに関する基本情報を格納した`workspace.prm`ファイル(プロジェクト・フォルダ内)が含まれています。

3. CloverETLパースペクティブを開く(p.34)

   - **CloverETL Designer**を完全インストールとしてインストールし、**Designer**を起動した場合は、**CloverETL**パースペクティブがデフォルトで開きます。
   - **CloverETL Designer**を**Eclipse**へのプラグインとしてインストールし、**Designer**を起動した場合は、基本的な**Eclipse**パースペクティブが開きます。これは**CloverETL**パースペクティブに切り替える必要があります。

すべてのCloverETLプロジェクトの標準構造

CloverETLパースペクティブでは、ウィンドウの左側に「**Navigator**」ペインがあります。このペインでは、プロジェクト・フォルダを展開できます。その後にフォルダ構造が表示されます。次のようなサブフォルダがあります。

表9.1. 標準フォルダおよびパラメータ

目的	標準フォルダ	標準パラメータ	パラメータの使用方法[1)]
すべての接続	conn	CONN_DIR	${CONN_DIR}
入力データ	data-in	DATAIN_DIR	${DATAIN_DIR}
出力データ	data-out	DATAOUT_DIR	${DATAOUT_DIR}
一時データ	data-tmp	DATATMP_DIR	${DATATMP_DIR}
グラフ	graph	GRAPH_DIR	${GRAPH_DIR}
ジョブフロー(*.jbf)	jobflow	JOBFLOW_DIR	${JOBFLOW_DIR}
参照表	lookup	LOOKUP_DIR	${LOOKUP_DIR}
メタデータ	meta	META_DIR	${META_DIR}
プロファイリング・ジョブ(*.cpj)	profile	PROFILE_DIR	${PROFILE_DIR}
シーケンス	seq	SEQ_DIR	${SEQ_DIR}
変換の定義(ソース・ファイルとクラスの両方)	trans	TRANS_DIR	${TRANS_DIR}

説明:

1): パラメータの詳細は、第29章「パラメータ」(p.217)を、その使用方法は「パラメータの使用方法」(p.225)を参照してください。

重要

CloverETLでパラメータを使用することによって、そのグラフ、メタデータまたはその他のグラフ要素を、名前を変更しないで任意の場所で使用できます。

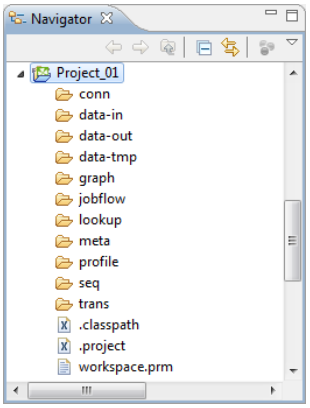


図9.1. 「Navigator」ペインのプロジェクト・フォルダ構造

Workspace.prmファイル

workspace.prmファイルは、「**Navigator**」ペインでこの項目をクリックし、右クリックしてコンテキスト・メニューから「**Open With**」→「**Text Editor**」を選択することによって表示できます。

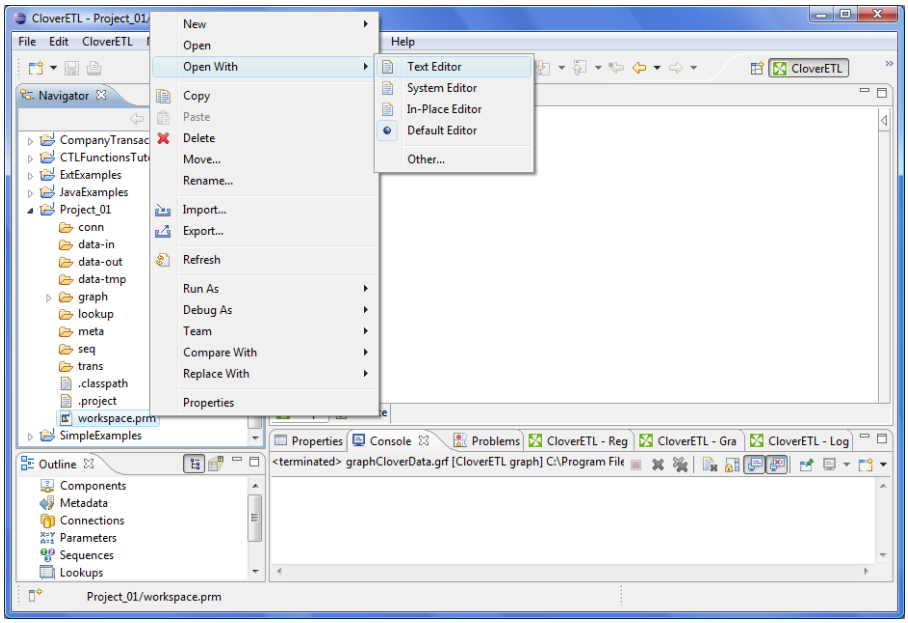


図9.2. Workspace.prmファイルを開く

新しいプロジェクトのパラメータを確認できます。

注意

インポートされたプロジェクトのパラメータは、新しいプロジェクトのデフォルトのパラメータと異なる場合があります。

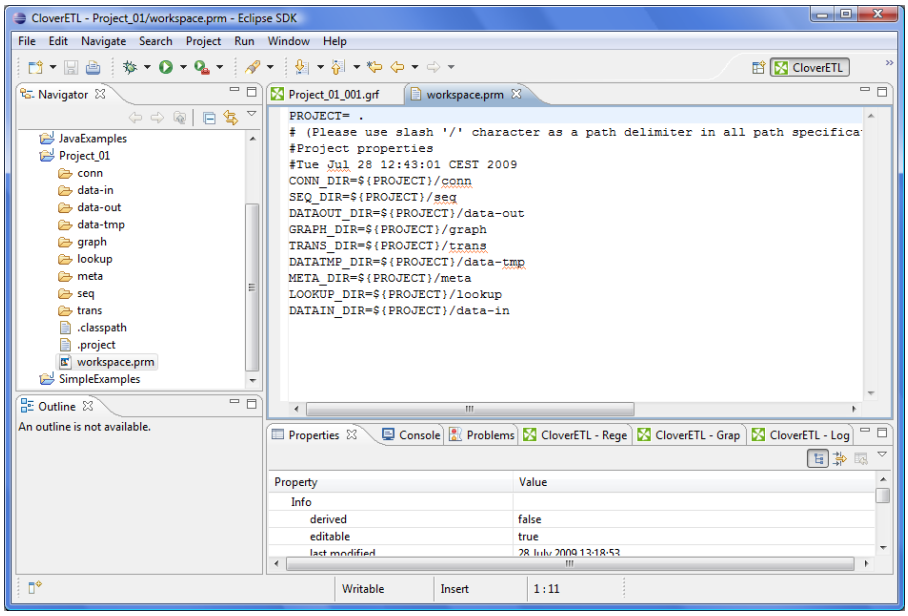


図9.3. Workspace.prmファイル

CloverETLパースペクティブを開く

前述のとおり、**CloverETL Designer**をプラグインとしてインストールした場合は、**CloverETL**パースペクティブに切り替える必要があります。

Eclipseのようこそ画面を閉じた後にパースペクティブを切り替えるには、ウィンドウ右上の「**Outline**」ペインの上にある**Java**ラベルの横のボタンをクリックして、「**Other...**」を選択します。

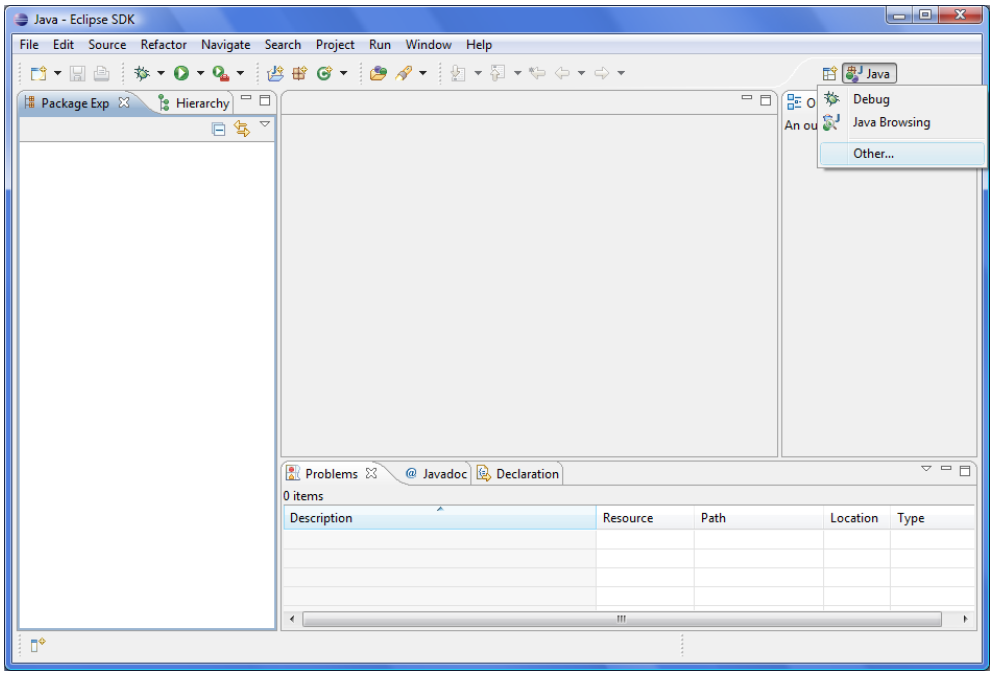


図9.4. 基本的なEclipseパースペクティブ

次に、リストから**CloverETL**の項目を選択し、「**OK**」をクリックします。

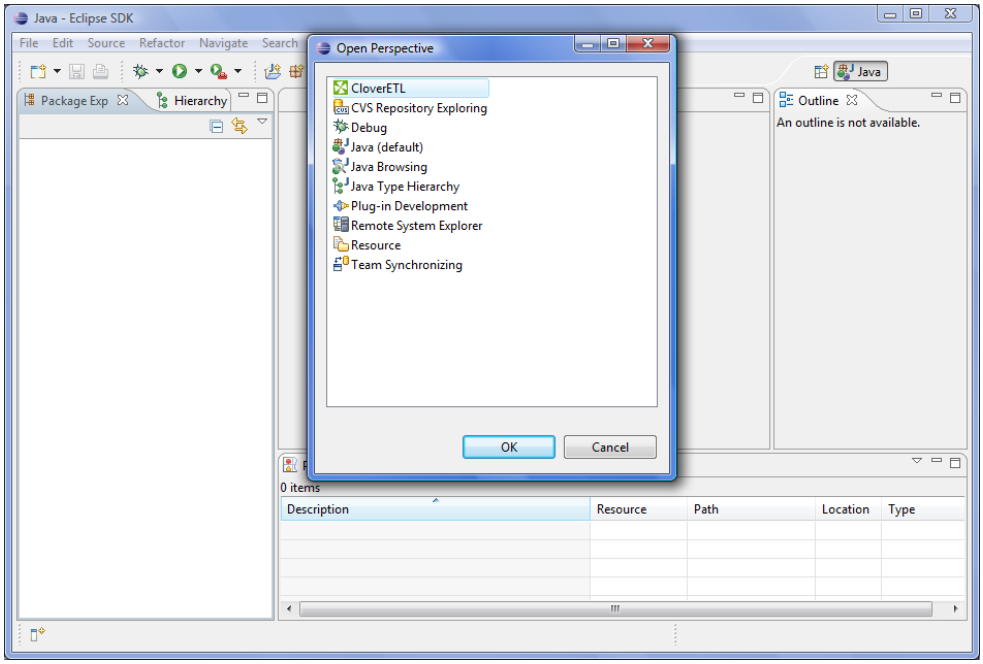


図9.5. CloverETLパースペクティブの選択

CloverETLパースペクティブが開きます。

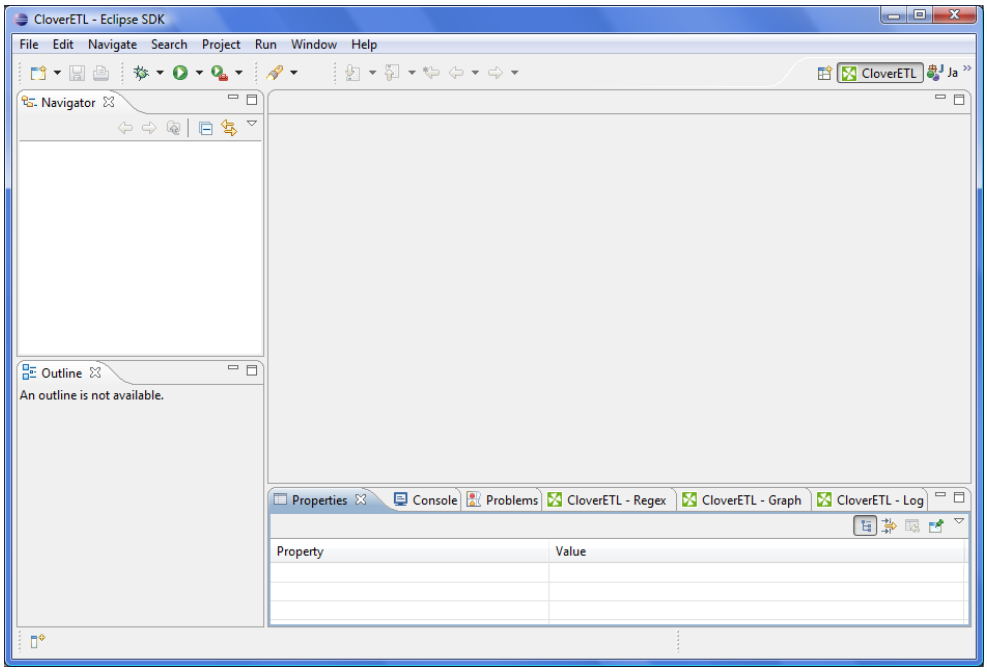


図9.6. CloverETLパースペクティブ

第10章 CloverETLパースペクティブの外観

CloverETLパースペクティブは4つのペインで構成されます。

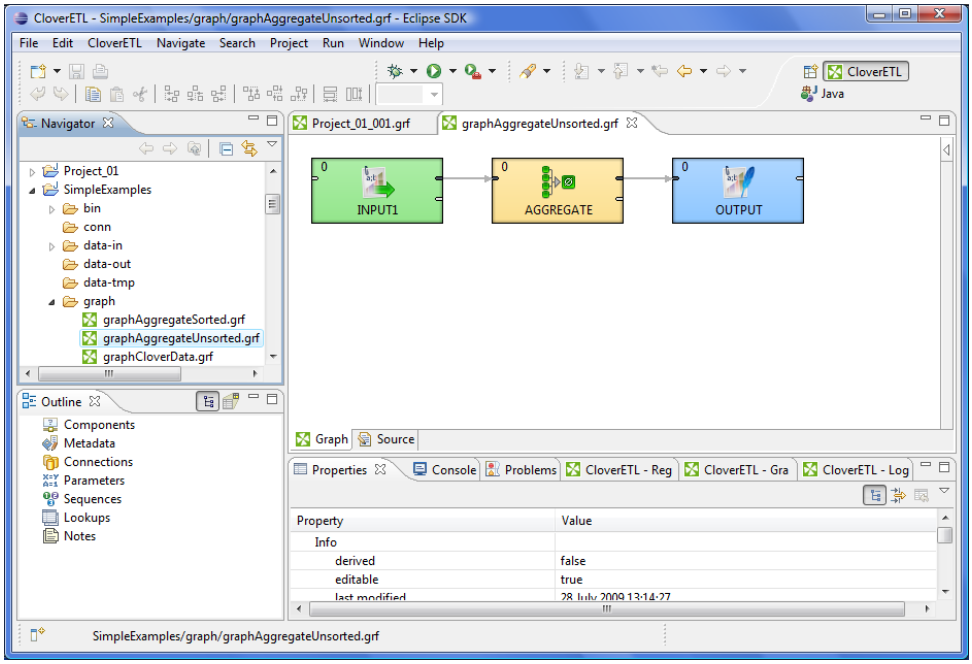


図10.1. CloverETLパースペクティブ

- **グラフ・エディタ**および**コンポーネント・パレット**は、右上のウィンドウに表示されます。

  このペインでグラフを作成できます。**コンポーネント・パレット**は、コンポーネントの選択、**グラフ・エディタ**への移動、およびエッジによる接続に使用します。このペインには2つのタブがあります。(グラフ・エディタおよびコンポーネント・パレット(p.37)を参照してください。)

- 「**Navigator**」ペインはウィンドウの左上にあります。

  このペインには、プロジェクトのフォルダとファイルが表示されます。これらは展開または縮小でき、その項目をダブルクリックして任意のグラフを開くことができます。(「Navigator」ペイン(p.41)を参照してください。)

- 「**Outline**」ペインはウィンドウの左下にあります。

  グラフ・エディタで開かれている、グラフのすべての部品が表示されます。(「Outline」ペイン(p.41)を参照してください。)

- **タブ・**ペインはウィンドウの右下にあります。

  これらのタブにはデータ解析プロセスが表示されます。(タブ・ペイン(p.43)を参照してください。)

CloverETL Designerのペイン

ここでは、各ペインについてさらに詳しく説明します。

CloverETL Designerのペインは次のとおりです。

- グラフ・エディタおよびコンポーネント・パレット(p.37)
- 「Navigator」ペイン(p.41)

- 「Outline」ペイン(p.41)
- タブ・ペイン(p.43)

グラフ・エディタおよびコンポーネント・パレット

最も重要なペインは、**グラフ・エディタ**および**コンポーネント・パレット**です。

グラフを作成するには、**パレット**・ツールで作業する必要があります。**CloverETL Designer**の起動後に開く場合と、ユーザーが**パレット**・ラベルの上にある矢印をクリックするか、**パレット**・ラベルの上にカーソルを置くことによって開く場合があります。**パレット**を再び閉じるには、同じ矢印をクリックするか、単にカーソルを**パレット**・ツールの外側に移動します。**グラフ・エディタ**内で境界線を移動して**パレット**の形を変更したり、ラベルをクリックして**グラフ・エディタ**の左側に移動することもできます。

グラフを作成したユーザーの名前および最後に変更したユーザーの名前は、「**Source**」タブに自動的に保存されます。

パレット・ツールからコンポーネントを選択し、**グラフ・エディタ**に貼り付けることができます。コンポーネントを貼り付けるには、単にコンポーネント・ラベルをクリックし、カーソルを**グラフ・エディタ**に移動して、再びクリックします。これにより、コンポーネントが**グラフ・エディタ**に表示されます。他のコンポーネントについても操作は同じです。

複数のコンポーネントを選択して**グラフ・エディタ**に貼り付けた後は、同じ**パレット**・ツールから取得したエッジで接続する必要があります。2つのコンポーネントをエッジで接続するには、**パレット**・ツールで**エッジ**・ラベルをクリックし、カーソルを1つ目のコンポーネントに移動し、クリックしてエッジをコンポーネントの出力ポートに接続し、カーソルをもう1つのコンポーネントの入力に移動して、再びクリックする必要があります。これにより、2つのコンポーネントが接続されます。エッジの作業が終了した後は、「**Palette**」ウィンドウの「**Select**」項目をクリックする必要があります。

グラフを作成または変更した後は、コンテキスト・メニューから「**Save**」項目を選択するか、メイン・メニューの「**Save**」ボタンをクリックして保存する必要があります。グラフは、そのグラフを作成したプロジェクトの一部になります。新しいグラフ名が「**Navigator**」ペインに表示されます。このグラフを**グラフ・エディタ**で開くと、グラフのすべてのコンポーネントおよびプロパティが「**Outline**」ペインに表示されます。

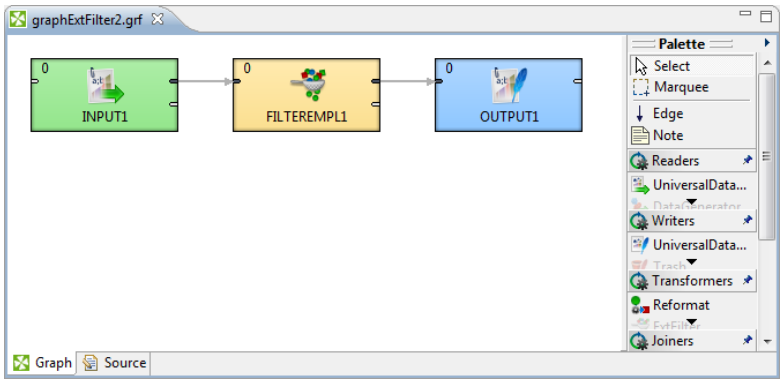


図10.2. コンポーネント・パレットを開いた状態のグラフ・エディタ

グラフ・エディタで開いているグラフを閉じるには、タブの右側の十字をクリックできますが、複数のタブを一度に閉じるには、タブの1つを右クリックし、コンテキスト・メニューから該当する項目を選択します。「**Close**」、「**Close Others**」、「**Close All**」などの項目があります。次を参照してください。

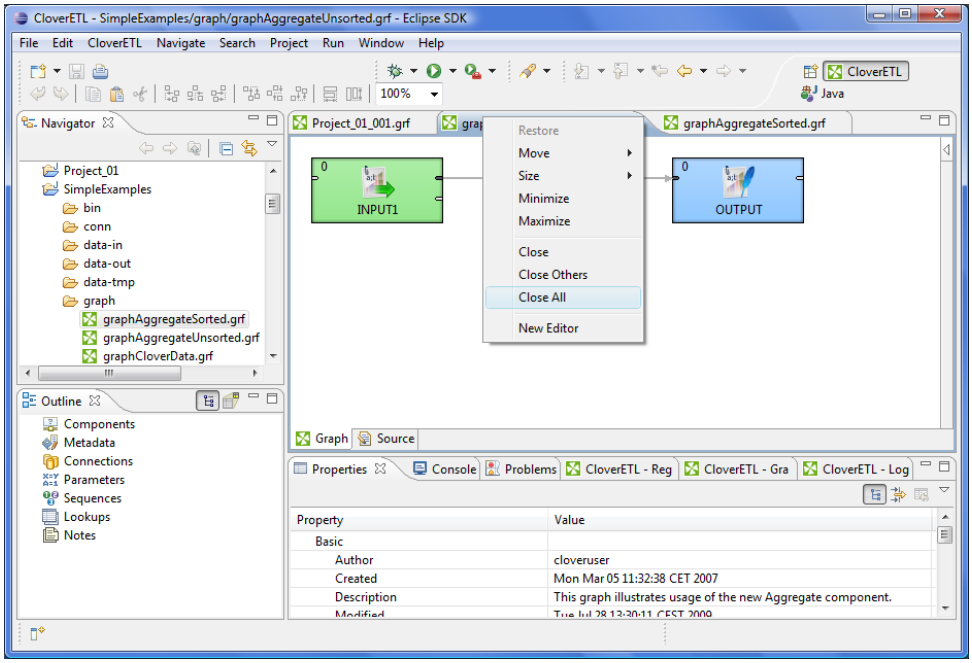


図10.3. グラフを閉じる

メイン・メニューから「**CloverETL**」項目を選択し(ただし**グラフ・エディタ**が選択されている場合のみ)、メニュー項目から「**Rulers**」オプションをオンにすることができます。

その後、水平または垂直のルーラーの任意の場所をクリックすると、それぞれ垂直または水平の直線が表示されます。続いて、任意のコンポーネントを特定の直線に貼り付けることができます。コンポーネントのいずれか1辺を貼り付けた後は、その直線を動かしてコンポーネントを移動できます。ルーラーで任意の直線をクリックすると、その直線を**グラフ・エディタ**・ペイン全体で移動できます。これにより、コンポーネントの位置合せができます。

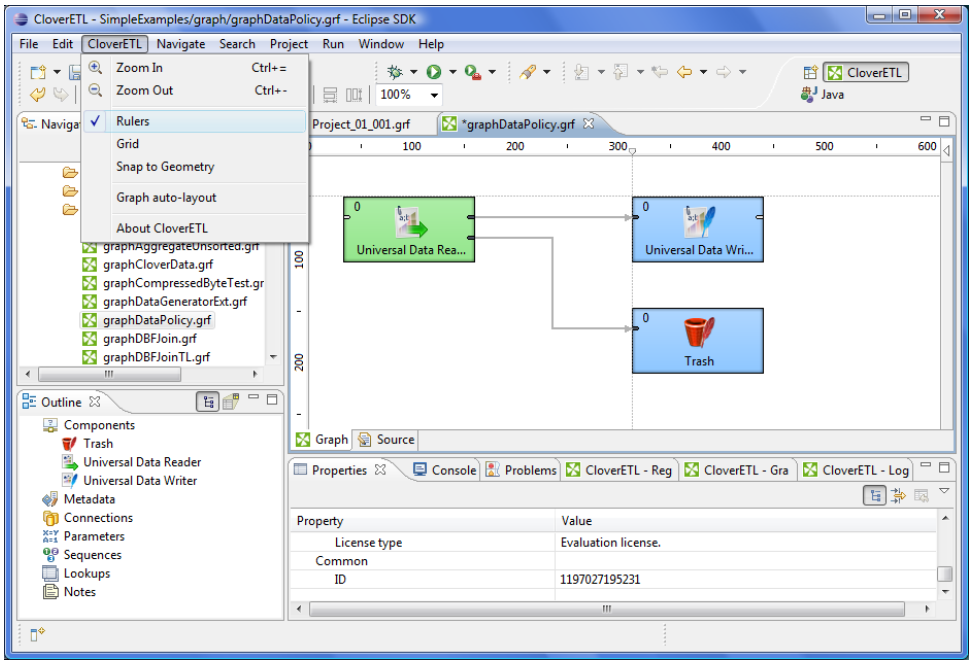


図10.4. グラフ・エディタのルーラー

メイン・メニューから「**CloverETL**」項目を選択し(ただし**グラフ・エディタ**が選択されている場合のみ)、メイン・メニューから「**Grid**」項目を選択して、**グラフ・エディタ**にグリッドを表示することもできます。

その後は、グリッドを使用してコンポーネントの位置合せができます。コンポーネントを移動すると、その上辺と左辺がグリッドの線に貼り付きます。この方法でもコンポーネントの位置合せができます。

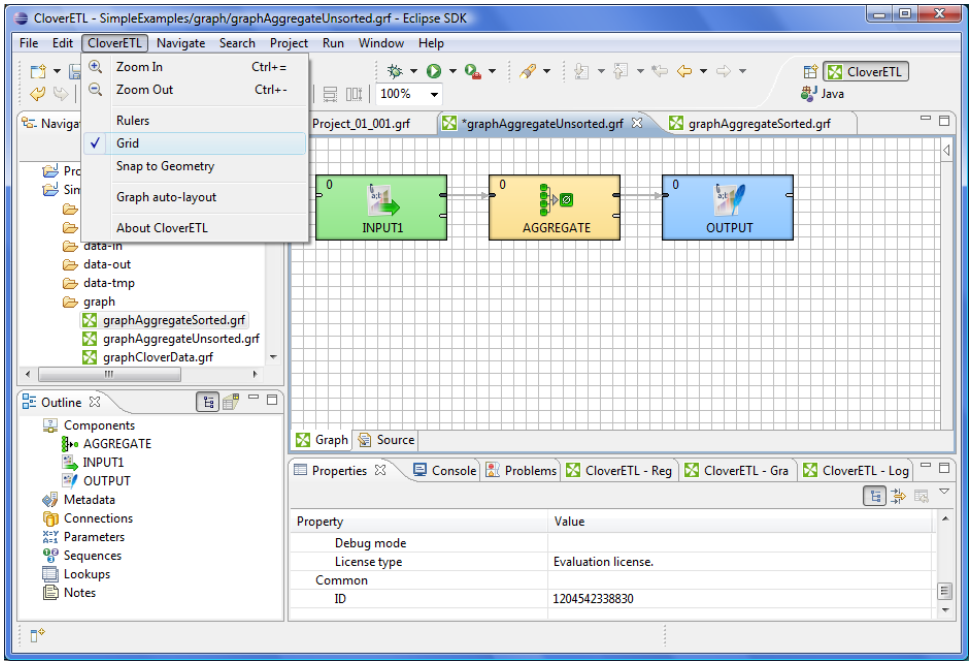


図10.5. グラフ・エディタのグリッド

「**Graph auto-layout**」項目をクリックすると、グラフのレイアウトを変更できます。`graphAggregateUnsorted.grf`を開いているときに「**Graph auto-layout**」項目を選択した場合の変化を次に示します。この項目を選択する前のグラフは次のとおりです。

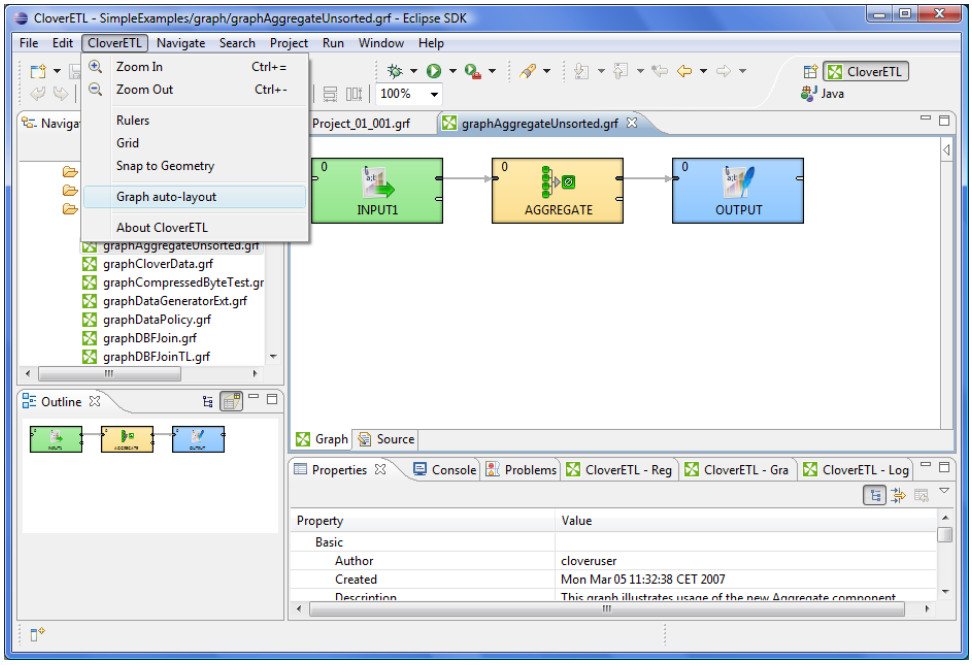


図10.6. 自動レイアウト選択前のグラフ

この項目を選択した後のグラフは次のようになります。

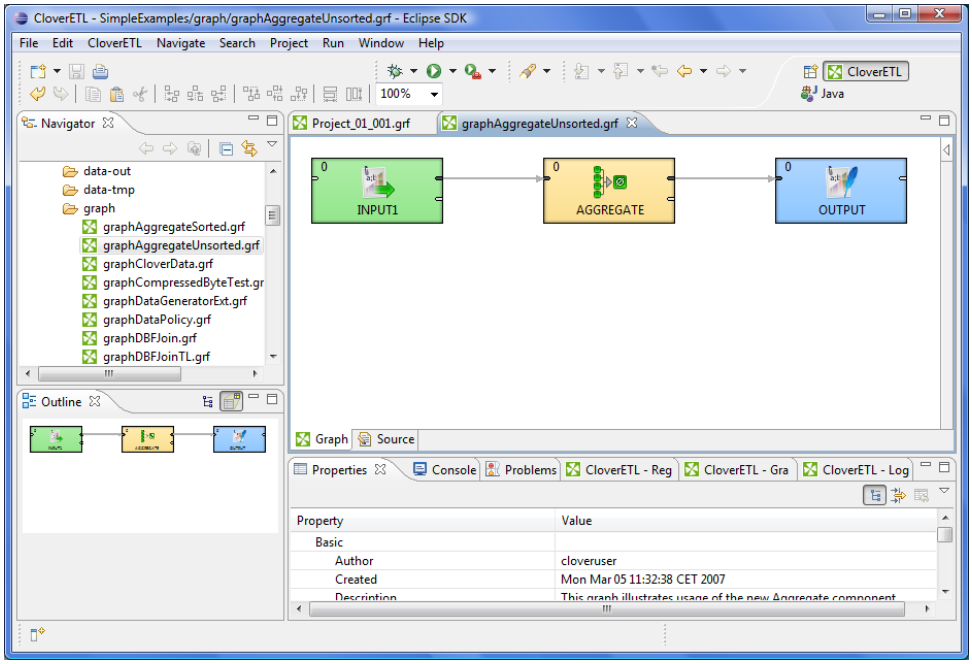


図10.7. 自動レイアウト選択後のグラフ

*グラフ・エディタ*では、他に次のような操作が可能です。

*グラフ・エディタ*内の任意の場所でマウスの左ボタンを押したまま、ペインでマウスをドラッグすると、四角形が作成されます。この方法で四角形を作成し、いくつかのグラフ・コンポーネントを囲んで最後にマウス・ボタンを放すと、これらのコンポーネントが強調表示されます。(次のグラフの左から1つ目と2つ目。)その後、*グラフ・エディタ*または「**Navigator**」ペインの上のツールバーで、6つのボタン(「**Align Left**」、「**Align Center**」、「**Align Right**」、「**Align Top**」、「**Align Middle**」および「**Align Bottom**」)が強調表示されます。(これらを使用して、選択したコンポーネントの位置を変更できます。)次を参照してください。

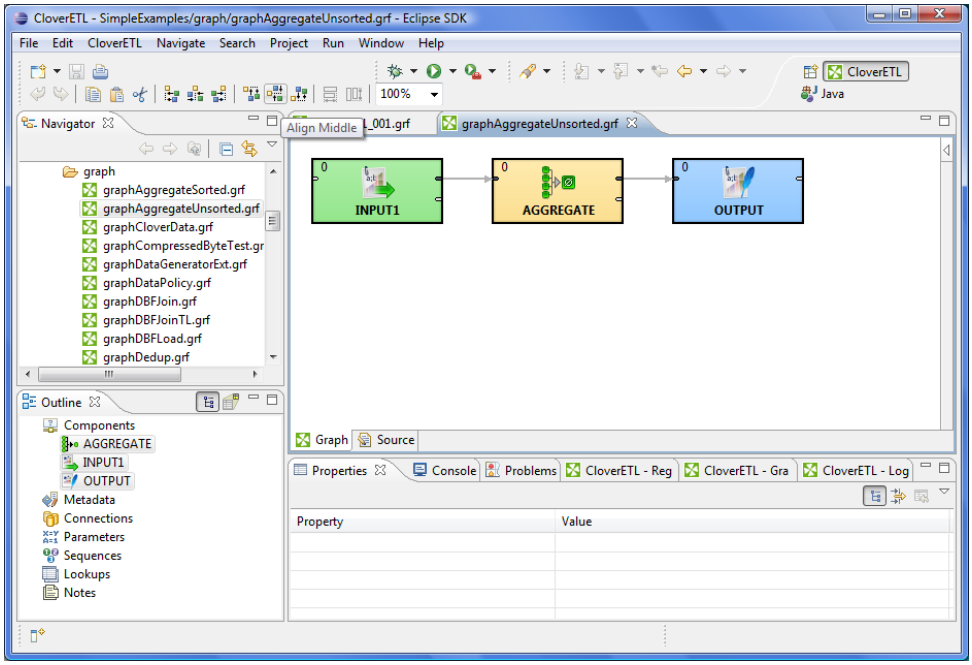


図10.8. ツールバーの新しい6つのボタンの強調表示(「Align Middle」を表示した状態)

同じ操作は、**グラフ・エディタ**内を右クリックして、コンテキスト・メニューから「**Alignments**」項目を選択することによっても可能です。前述と同じ項目のサブメニューが表示されます。

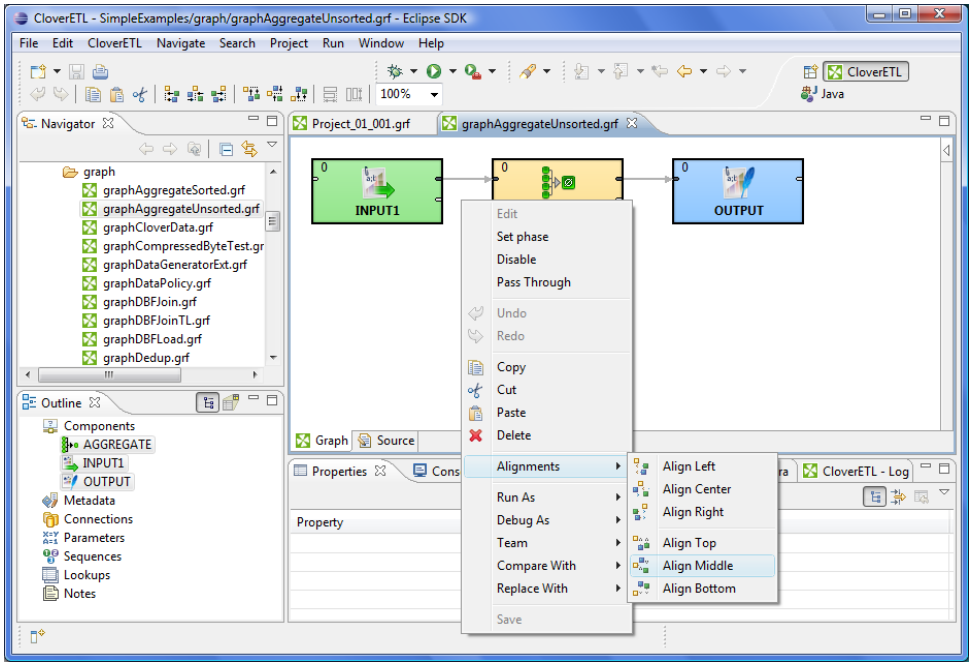


図10.9. コンテキスト・メニューの「Alignments」

グラフの強調表示された部分は、**[Ctrl]キーを押しながら[C]キー**を押し、続いて他のグラフを開いた後で**[Ctrl]キーを押しながら[V]キー**を押すことによってコピーできます。

「Navigator」ペイン

「**Navigator**」ペインには、プロジェクトおよびそのサブフォルダとファイルのリストが表示されます。これらは展開、縮小、表示および開くことができます。

プロジェクトのすべてのグラフがこのペインに配置されます。これらはグラフ項目をダブルクリックすることにより、**グラフ・エディタ**で開くことができます。

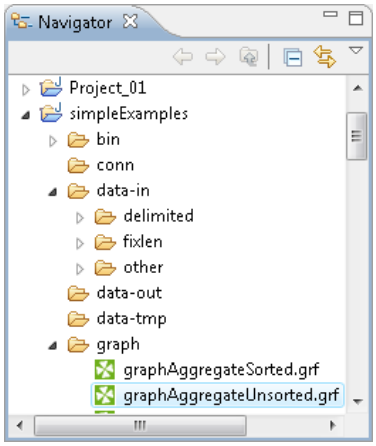


図10.10. 「Navigator」ペイン

「Outline」ペイン

「**Outline**」ペインには、選択したグラフのすべてのコンポーネントが表示されます。ここではグラフ・コンポーネントのすべてのプロパティ、エッジのメタデータ、データベース接続またはJMS接続、参照、パラメータ、シーケ

ンスおよびノートを作成または編集できます。内部プロパティの作成および外部(共有)プロパティのリンクの両方が可能です。内部プロパティはグラフに含まれているため、グラフで表示できます。内部プロパティの外部化および外部(共有)プロパティの内部化またはその両方が可能です。内部メタデータをエクスポートすることもできます。「Outline」ペインで項目(コンポーネント、接続、メタデータなど)を選択して**[Enter]**キーを押すと、そのエディタが開きます。

ヒント

右上隅にある黄色の「**Link with Editor**」アイコンをアクティブ化すると、グラフ・エディタでコンポーネントを選択するたびに「Outline」ペインでも選択されるようになります。これは小さなグラフでは便利ですが、複雑なグラフではオフにして、グラフの作業中に「Outline」ペインでコンポーネントの長いリストが何度も展開されないようにします。

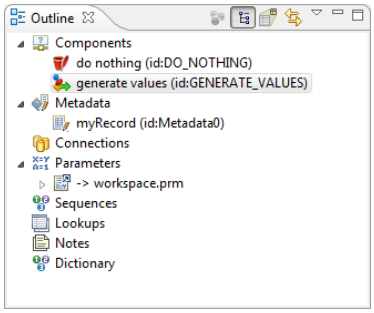


図10.11. 「Outline」ペイン

「**Outline**」ペインの右上にある2つのボタンのプロパティは次のとおりです。

デフォルトでは、「**Outline**」ペインにはコンポーネント、メタデータ、接続、パラメータ、シーケンス、参照およびノートのツリーが表示されます。「**Outline**」ペインの右上の左から2つ目のボタンをクリックすると、ペインがもう1つの表現に切り替わります。次のようになります。

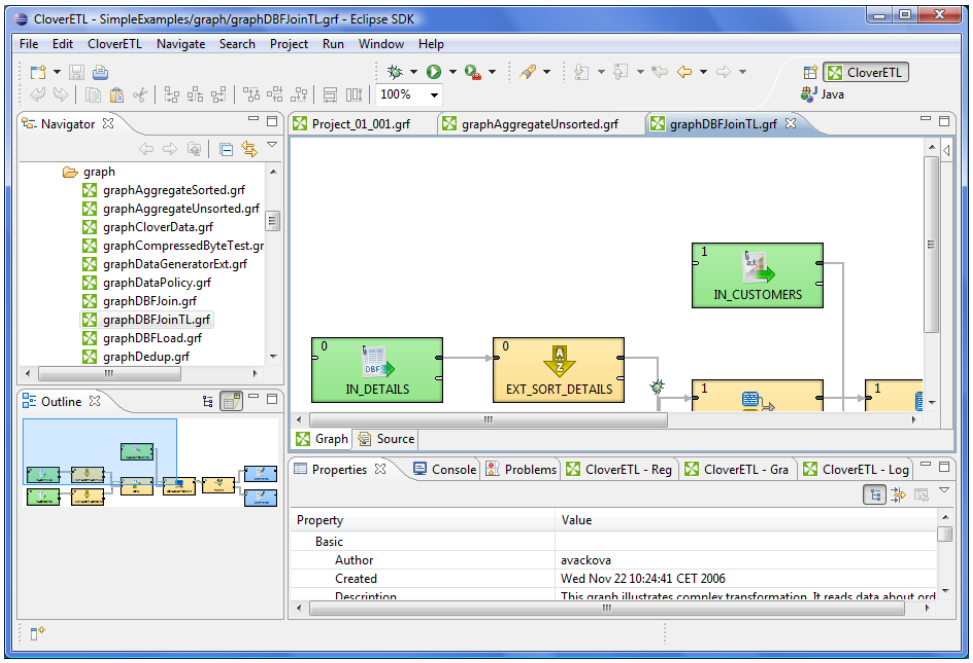


図10.12. 「Outline」ペインのもう1つの表現

グラフ・エディタでサンプル・グラフの一部が表示され、それと同じグラフ構造が「**Outline**」ペインに表示されます。さらに、「**Outline**」ペインには水色の四角形があります。***グラフ・エディタ***に表示されているグラフと同じ部分が「**Outline**」ペインの水色の四角形で囲まれていることがわかります。この四角形を「**Outline**」ペインの空間

内で移動することにより、**グラフ・エディタ**のグラフの対応する部分が四角形とともに移動します。水色の四角形と**グラフ・エディタ**内のグラフの両方が同様に移動します。

グラフ・エディタの右側と下側にあるスクロール・バーを使用して、同じ操作を実行できます。

「**Outline**」ペインのツリー表現を切り替えるには、単に「**Outline**」ペインの右上の左から1つ目のボタンをクリックします。

要素のロック

「**Outline**」では、任意の共有グラフ要素をロックできます。ロックとは、要素を変更しようとする他者に対してそれが好ましくないことを伝えるために、要素に意図的に割り当てるフラグ(およびオプションのテキスト・メッセージ)です。次のグラフ要素をロックできます。

- **メタデータ**
- **接続**
- **パラメータ**(外部のみ)
- **シーケンス**
- **参照**

これらの要素をロックするには、「**Outline**」で右クリックして「**Lock**」をクリックします。

注意

ロックはセキュリティ・ツールではありません。ロックはだれでも解除でき、ユーザーに所有されません。

様々な場所(変換エディタ(p.286)など)で、ロックされたメタデータを変更するなどしてロックされた要素にアクセスすると、警告が表示されます。

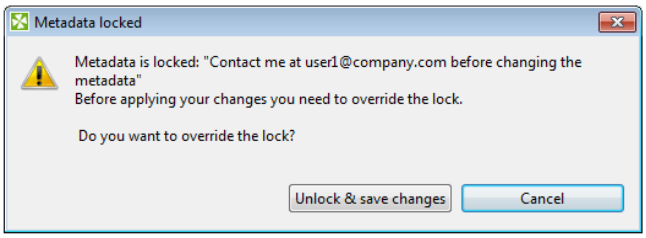


図10.13. ロックされたグラフ要素へのアクセス(ロックの説明テキストは任意に追加可能)

タブ・ペイン

ウィンドウの右下には一連のタブがあります。

注意

特定のペインのタブを拡大するには、単にそのタブをダブルクリックします。ペインがウィンドウ全体のサイズに拡大します。再びダブルクリックすると元のサイズに戻ります。

- 「**Properties**」タブ

  このタブでは、コンポーネントのプロパティを表示および編集できます。コンポーネントをクリックすると、選択したコンポーネントのプロパティ(属性)がこのタブに表示されます。

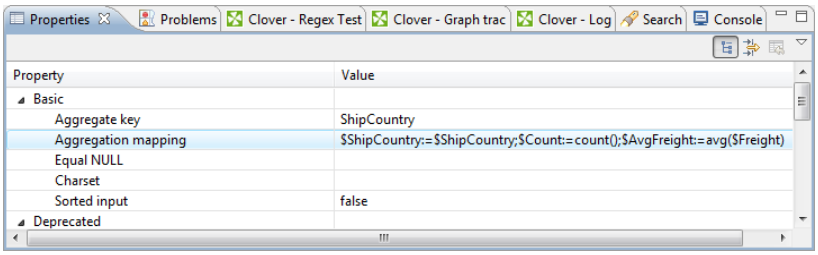


図10.14.「Properties」タブ

- 「**Console**」タブ

  このタブには、データの読取り、アンロード、変換、結合、書込みおよびロードのプロセスが表示されます。

  デフォルトでは、「**Console**」は**CloverETL**によって`stdout`または`stderr`に書き込まれるたびに開きます。

  これを変更するには、「**Window**」→「**Preferences**」を選択して表示される2つのチェック・ボックスの選択を解除し、「**Run/Debug**」カテゴリを展開し、「**Console**」項目を開きます。

  「**Console**」の動作を制御する2つのチェック・ボックスは次のとおりです。

  - **Show when program writes to standard out**
  - **Show when program writes to standard error**

  「**Console**」に保存されるバッファの文字数も制御できます。

  バッファに保存される文字数およびタブの幅のために、もう1つのチェック・ボックス(「**Limit console output**」)および2つのテキスト・フィールドがあります。

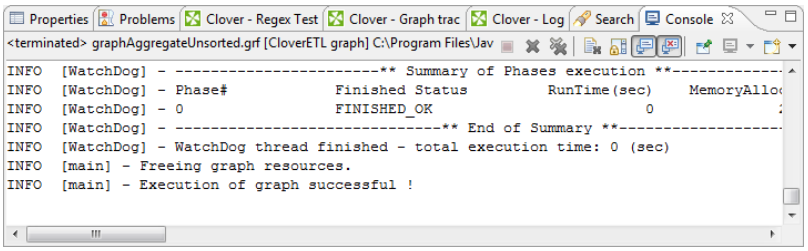


図10.15.「Console」タブ

- 「**Problems**」タブ

  このタブにはエラーメッセージや警告などが表示されます。項目の1つを展開すると、そのリソース(グラフの名前)、パス(グラフのパス)、場所(コンポーネントの名前)が表示されます。

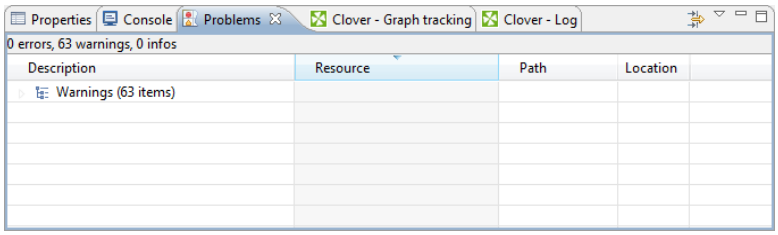


図10.16.「Problems」タブ

- 「**Clover - Regex Tester**」タブ

  このタブでは、**正規表現**を操作できます。「**Regular expression**」テキスト領域に、任意の正規表現を貼付けまたは入力できます。この領域内で**[Ctrl]キーを押しながらスペース・キー**を押すことにより、コンテンツ・アシストを呼び出すことができます。「**Regular expression**」テキスト領域の下のペインに、任意のテキストを貼付けまたは入力する必要があります。その後で、表現をテキストと比較できます。表現を変更しながらオンザフライで評価することも、「**Evaluate on the fly**」チェック・ボックスの選択を解除して、「**Go**」ボタンをクリックしたときに表現をテキストと比較することもできます。結果は右側のペインに表示されます。一部のオプションはデフォルトで選択されています。テキスト内の表現の**検索**、表現に従ったテキストの**分割**、テキストが正規表現と完全に**一致**するかどうかの識別なども選択できます。チェック・ボックスの選択は自由です。正規表現および提供されるオプションの詳細は、次のサイトを参照してください。
  http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

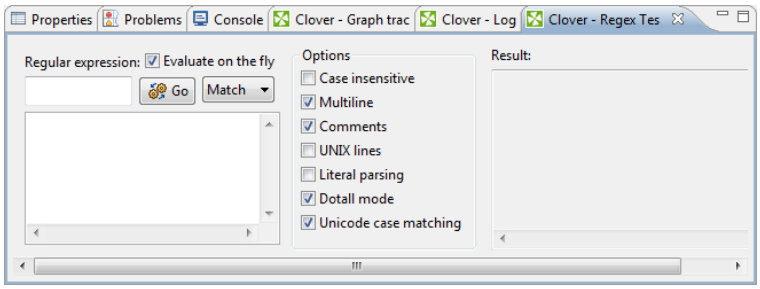


図10.17.「Clover - Regex Tester」タブ

- 「**Clover - Graph tracking**」タブ

  このタブには、正常に実行されたグラフの概要が表示されます。フェーズごと(秒単位の使用時間、パーセント単位の使用容量)にグループ化された各コンポーネントの名前、すべてのコンポーネントのステータス、使用した**CPU時間**(秒)、使用した**CPU**負荷(パーセント)、**バイト・フロー**および**バイト・フローのピーク**(1秒当たりのバイト数)、**レコード・フロー**および**レコード・フローのピーク**(1秒当たりのレコード数)、処理された**総バイト**数(バイト、KBなど)、処理された**総レコード数**(レコード数)があります。これらのプロパティはデフォルトでタブに表示されます。

  さらに、**ユーザー時間**、**ピークCPU**、**待機中のレコード数**、**待機中の平均レコード数および使用済メモリー**も表示できます。これらは「**Window**」→「**Preferences**」を選択して、既存の項目に追加する必要があります。次に**CloverETL**グループを展開し、「**Tracking**」項目を選択する必要があります。ペインには、ログ列のリストが表示されます。「**Next...**」または「**Remove**」ボタンをクリックして、一部を追加または削除できます。「**Graph tracking**」タブ内で「**Up**」または「**Down**」ボタンをクリックすることにより、列の順序を変更することもできます。

  「**Show tracking during graph run**」チェック・ボックスの選択を解除して、グラフのトラッキングをオフにすることもできます。

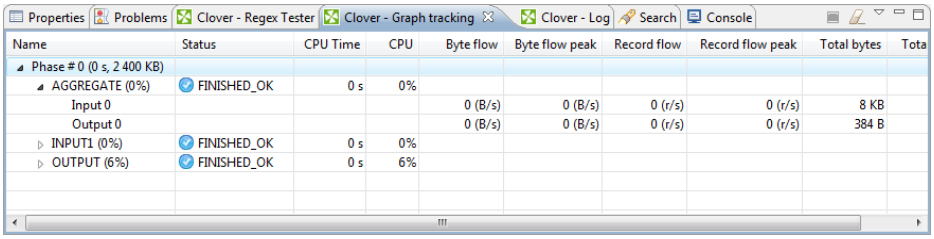


図10.18.「Clover - Graph Tracking」タブ

- 「**Clover - Log**」タブ

  このタブには、グラフの実行後に作成されたデータ解析プロセスのログ全体が表示されます。グラフの複数実行で得られた一連のログになる場合があります。

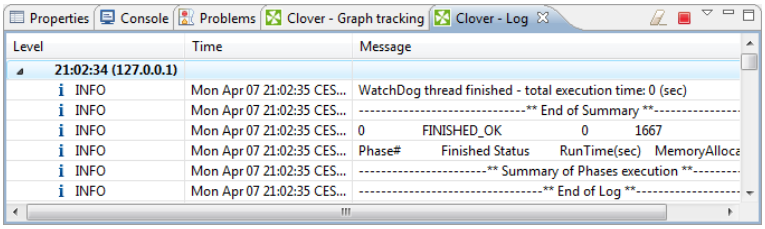


図10.19.「Clover - Log」タブ

第11章 CloverETLグラフの作成

すべての**CloverETL**プロジェクト内に**CloverETL**グラフを作成する必要があります。次の各項では、グラフの作成方法について説明します。

1. 最初の手順として、プロジェクト内に空のグラフを作成する必要があります。空のグラフの作成(p.47)を参照してください。

2. 2つ目の手順として、グラフ・コンポーネント、要素およびその他のツールを使用して、変換グラフを作成する必要があります。例については、単純な手順での単純なグラフの作成(p.51)で変換グラフの作成の例を参照してください。

注意

ワークスペース内にすでに**CloverETL**プロジェクトが存在し、**CloverETL**パースペクティブが開いている場合は、次の**CloverETL**プロジェクトを作成する方法が多少異なります。

- メイン・メニューから「**File**」→「**New**」→「**CloverETL Project**」を選択して新しい**CloverETL**プロジェクトを直接作成するか、「**File**」→「**New**」→「**Project...**」を選択してから前述の手順を実行して作成できます。
- 「**Navigator**」ペイン内で右クリックして、コンテキスト・メニューから「**New**」→「**CloverETL Project**」を直接選択するか、「**New**」→「**Project...**」を選択してから前述の手順を実行することもできます。

純粋なETLグラフを作成するときは、「**File**」→「**New**」メニューの次の2つのオプションに注意してください。

- **Jobflow**: ETLグラフに類似する`*.jbf`ファイルを作成します。ジョブ制御(p.684)コンポーネントを使用して入力できます。これらは他のグラフおよび複雑なワークフローを実行、監視および中断するためにあります。詳細は、第50章「ジョブ制御の共通プロパティ」(p.333)を参照してください。
- **Profiler Job**: 新しい`*.cpj`ファイルを作成します。これを使用してデータの統計解析を実行できます。ProfilerProbe(p.773)コンポーネントを参照してください。

空のグラフの作成

この後は、任意の**CloverETL**プロジェクトに対して**CloverETL**グラフを作成できます。たとえば、「**File**」→「**New**」→「**ETL Graph**」を選択して、`Project_01`のグラフを作成できます。「**Navigator**」ペインで目的のプロジェクトを右クリックして、コンテキスト・メニューから「**New**」→「**ETL Graph**」を選択することもできます。

注意

ジョブフローの新規作成は同様の方法で行います。**プロファイラ・ジョブ**の詳細は、ProfilerProbe(p.773)を参照してください。

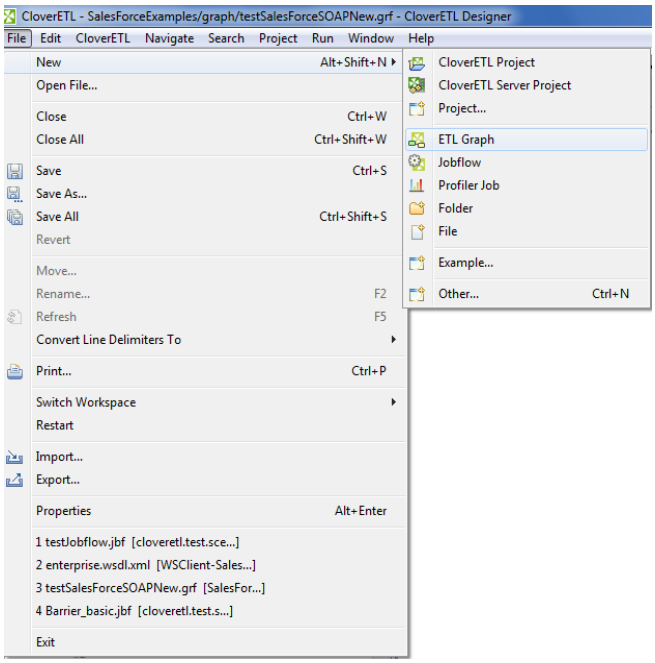


図11.1. 新規グラフの作成

項目をクリックすると、グラフに名前を付けるように要求されます。たとえば、名前も*Project_01*にすることができます。ただし、多くの場合、プロジェクトにはさらに多くのグラフが含まれるため、たとえば*Project_01_###*のような名前を付けることができます。または、グラフの目的を説明する他の名前を付けることもできます。

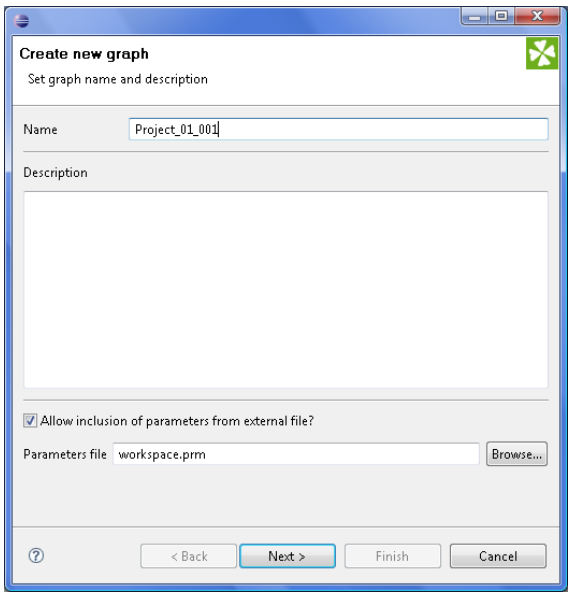


図11.2. 新しいCloverETLグラフの名前付け

グラフとともにこのプロジェクトに含める必要があるパラメータ・ファイルも決定できます。この選択は、このウィンドウ下部のテキスト領域で行います。他のファイルを見つけるには、「Browse...」ボタンをクリックして、適切なものを検索します。チェック・ボックスの選択を解除して、グラフにパラメータ・ファイルを含めないままにすることもできます。

ここでは、`workspace.prm`ファイルを含めることにします。

最後に、「**Next**」ボタンをクリックできます。その後は、選択した名前に自動的に識別子`.grf`が追加されます。

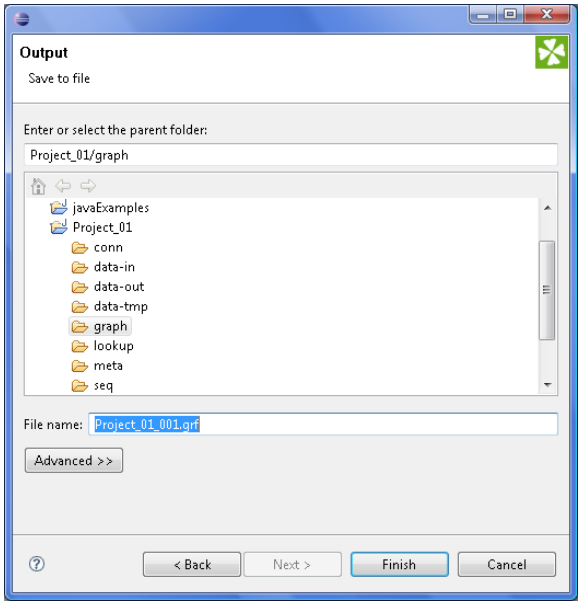


図11.3. グラフの親フォルダの選択

「**Finish**」をクリックすると、グラフが`graph`サブフォルダに保存されます。続いて、項目`Project_01_001.grf`が「**Navigator**」ペインに表示され、「**Project_01_001.grf**」という名前のタブがウィンドウに表示されます。

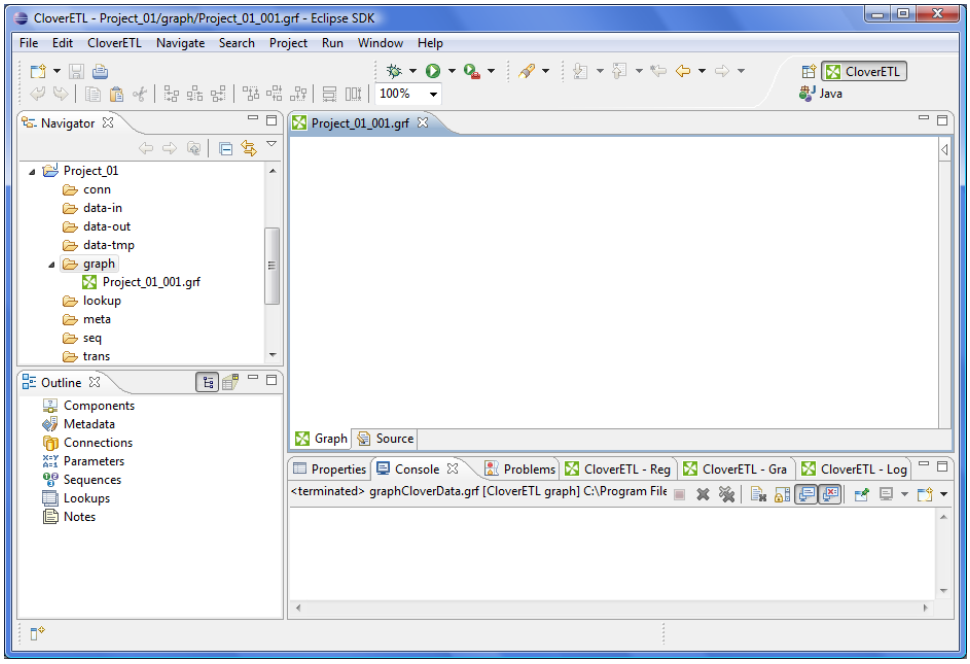


図11.4. グラフ・エディタを選択した状態のCloverETLパースペクティブ

グラフの右側にコンポーネント・パレットがあるのがわかります。このパレットは、**三角形の**ボタンをクリックすることによって開閉できます。各コンポーネントの詳細は、第VII部「コンポーネントの概要」(p.260)を参照してください。

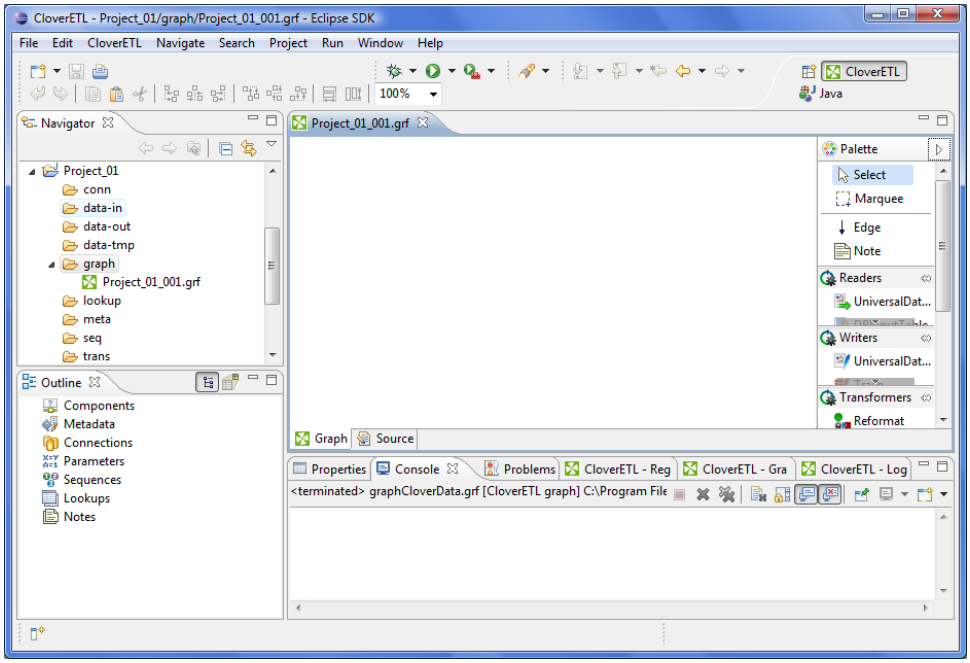


図11.5. 新しいグラフとコンポーネント・パレットを表示したグラフ・エディタ

単純な手順での単純なグラフの作成

新しい**CloverETL**グラフを作成した後のグラフは空のペインです。空ではないグラフを作成するには、空のグラフにコンポーネントおよびその他のグラフ要素を入力する必要があります。必要な作業には、グラフ・コンポーネントの選択、そのプロパティ(属性)の設定、これらのコンポーネントのエッジによる接続、読取り(アンロード)元および書込み(ロード)先にする必要があるデータ・ファイルまたはデータベース表の選択、データを説明するメタデータの作成およびエッジへの割当て、データベース接続またはJMS接続の作成、参照表の作成、シーケンスおよびパラメータの作成などがあります。すべてが完了した後は、グラフを実行できます。

エッジ、メタデータ、接続、参照表、シーケンスまたはパラメータの詳細は、第V部「グラフ要素、構造およびツール」(p.96)を参照してください。

ここでは、**CloverETL Designer**を使用して**CloverETL**変換グラフを作成する方法の単純な例を示します。できるだけわかりやすく説明します。

初めに、**コンポーネント・パレット**からコンポーネントを選択する必要があります。

コンポーネントを選択するには、**グラフ・エディタ**・ペインの右上隅にある三角形をクリックします。**コンポーネント・パレット**が開きます。目的のコンポーネントをクリックして選択し、**グラフ・エディタ**・ペインにドラッグ・アンド・ドロップします。

このデモンストレーションでは、パレットの「**Readers**」カテゴリから「**UniversalDataReader**」を選択します。さらに、「**Transformers**」カテゴリから「**ExtSort**」コンポーネント、「**Writers**」カテゴリから「**UniversalDataWriter**」も選択します。

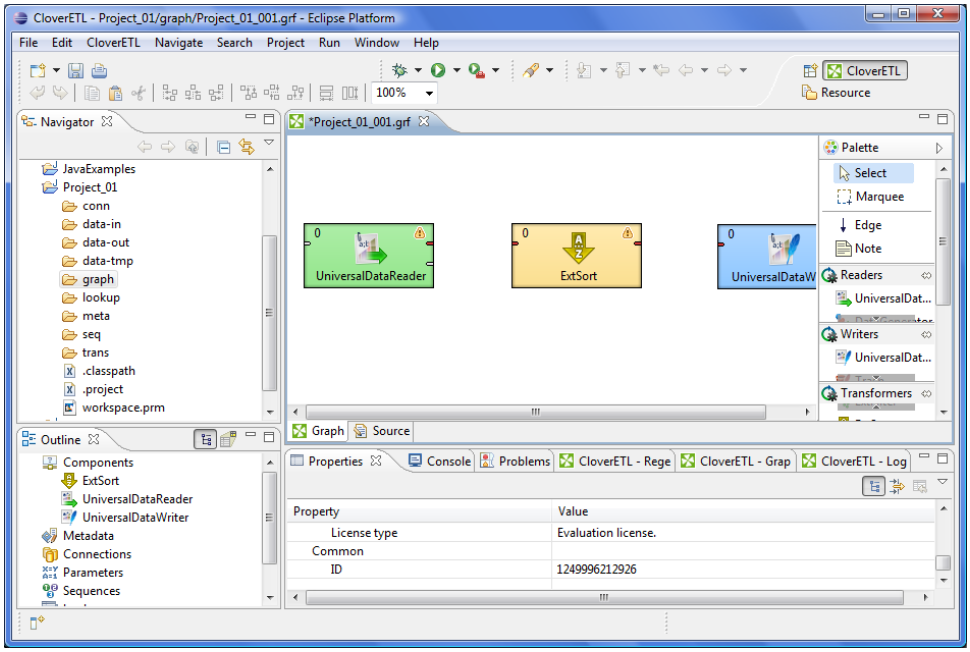


図11.6. パレットからのコンポーネントの選択

コンポーネントを**グラフ・エディタ**・ペインに挿入した後は、エッジで接続する必要があります。**パレット**の「**Edge**」ツールを選択し、1つのコンポーネントの出力ポートをクリックし、もう一度クリックして別のコンポーネントの入力ポートに接続します。選択したすべてのコンポーネントで同じ操作を実行します。新しく接続したエッジは、まだ破線の状態です。右上隅の三角形をクリックして**パレット**を閉じます。(**エッジ**の詳細は、第20章「エッジ」(p.99)を参照してください。)

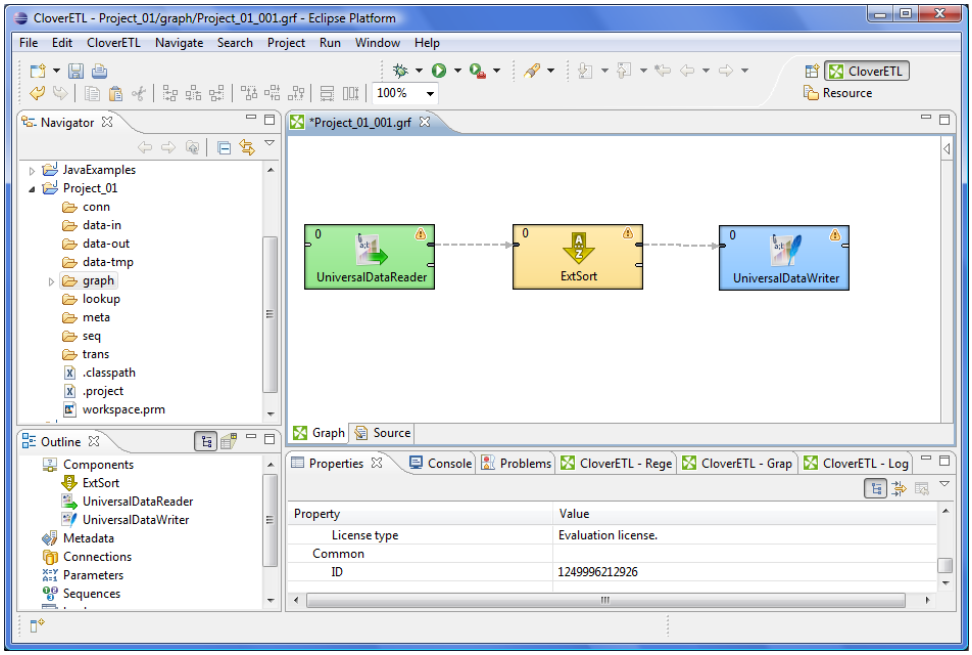


図11.7. エッジで接続されたコンポーネント

次に、入力ファイルを準備する必要があります。**Eclipse**ウィンドウの左側にある「**Navigator**」ペインに移動します。プロジェクトの`data-in`フォルダを右クリックして、「**New**」→「**File**」を選択します。

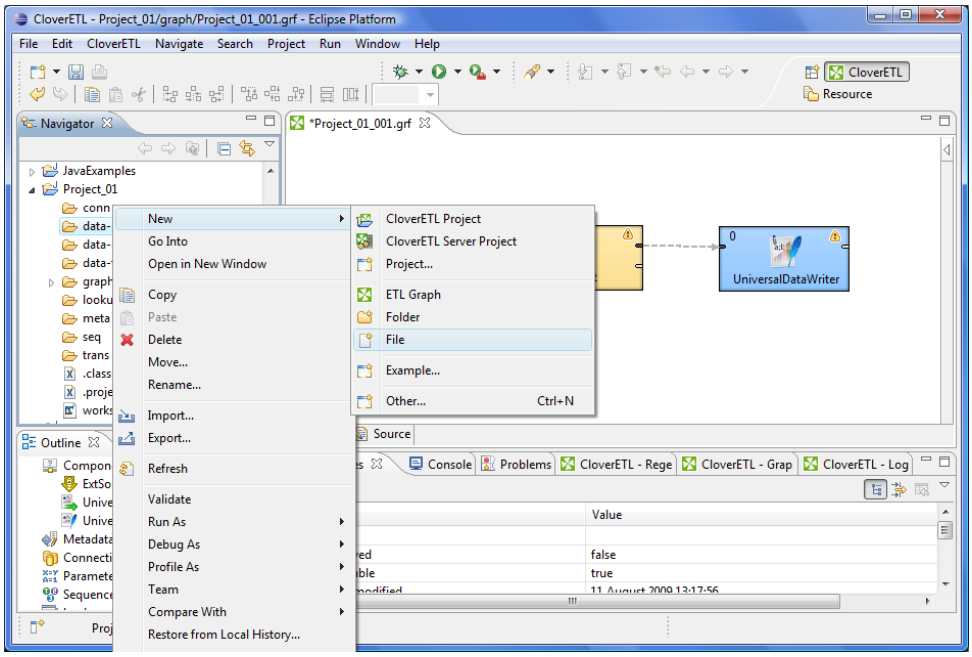


図11.8. 入力ファイルの作成

新しいウィンドウが表示された後に、このウィンドウで入力ファイルの名前を選択します。たとえば、名前は`input.txt`などになります。「**Finish**」をクリックします。ファイルが**Eclipse**ウィンドウで開きます。

このファイルにデータを入力します。たとえば、名と姓のペアを`John|Brown`のように入力できます。同様の形式でさらに多くの行を入力します。最後に新しい空の行を作成することを忘れないでください。各行(レコード)は次のようになります。

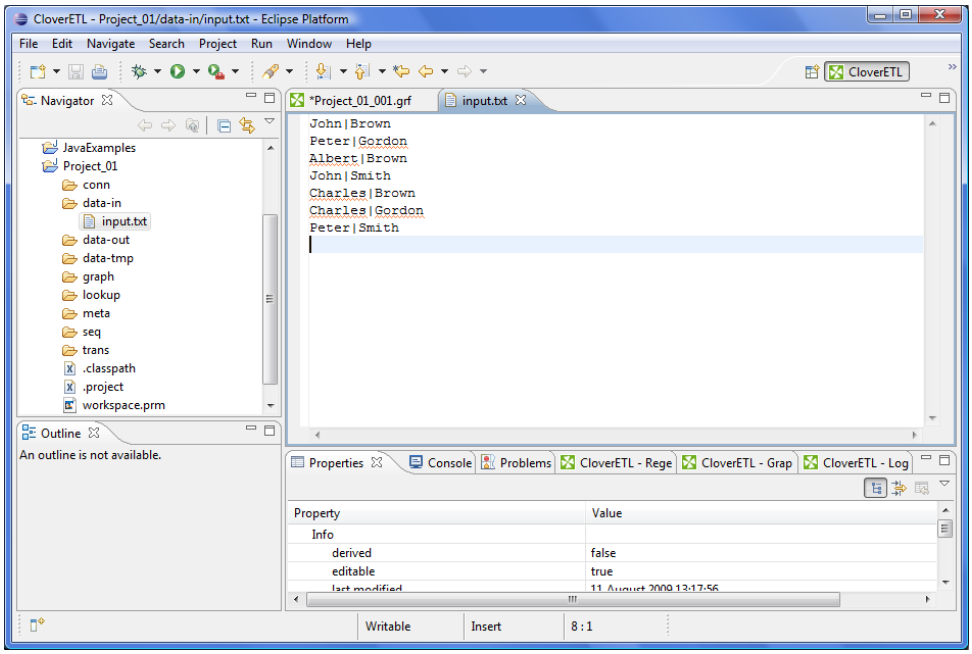


図11.9. 入力ファイルのコンテンツの作成

[Ctrl]キーを押しながら[S]キーを押して、ファイルを保存する必要があります。

次に、左から1つ目のエッジをダブルクリックして、エッジの横に表示されるメニューから「**Create metadata**」を選択します。「**Metadata editor**」で、緑色の**プラス記号**のボタンをクリックします。別の(2つ目の)フィールドが表示されます。これら2つのフィールドをクリックして名前を変更できます。クリックすると青色に変わり、名前を変更できます。その後**[Enter]**キーを押します。(**メタデータ**作成の詳細は、第21章「メタデータ」(p.110)を参照してください。)

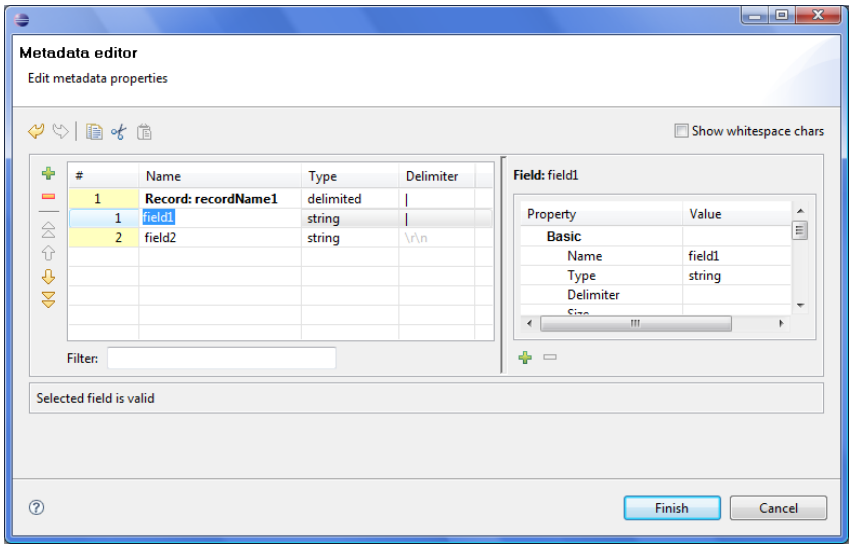


図11.10. フィールドのデフォルト名が表示された「Metadata editor」

操作を終えると、2つのフィールドの名前はそれぞれ「**Firstname**」と「**Surname**」になります。

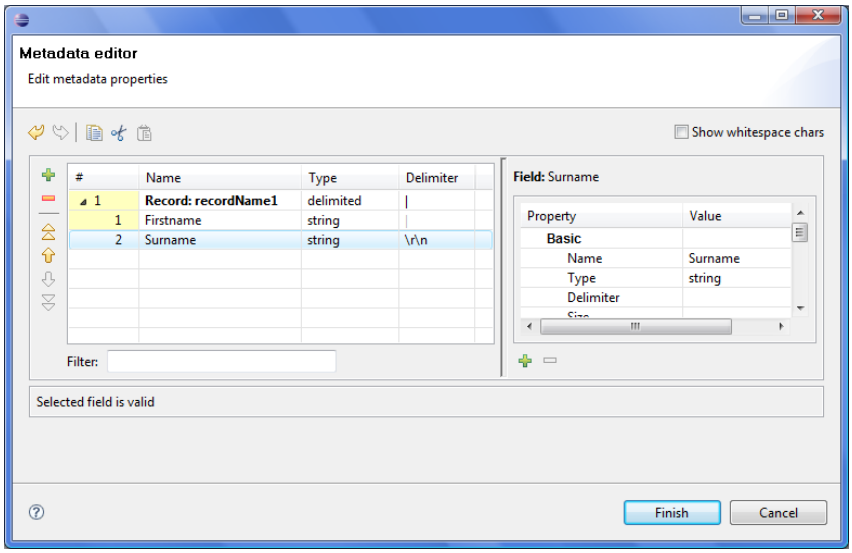


図11.11. フィールドの新しい名前が表示された「Metadata editor」

「**Finish**」をクリックするとメタデータが作成され、エッジに割り当てられます。これでエッジが実線になります。

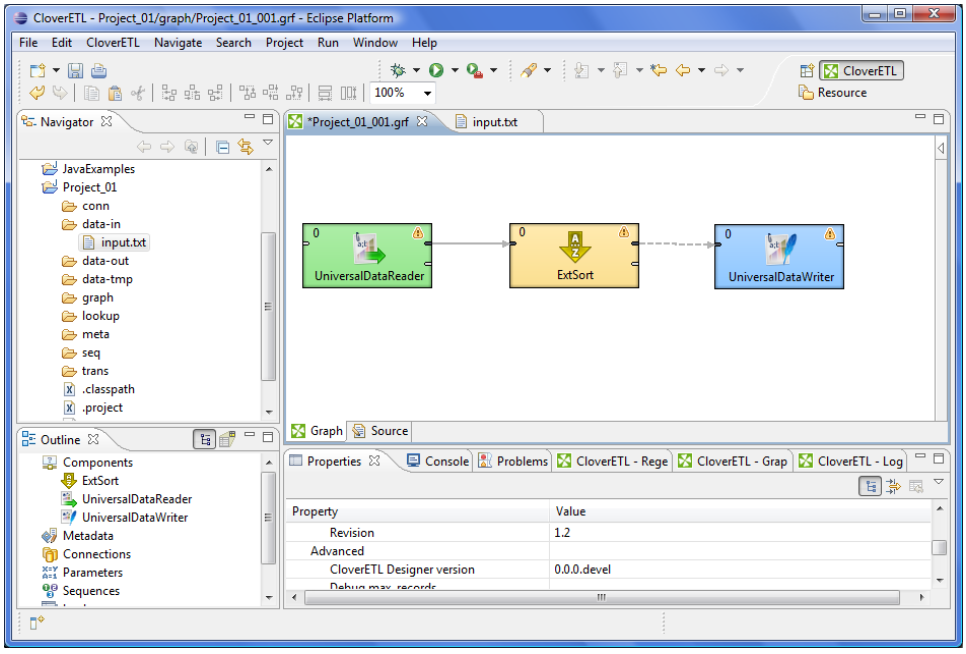


図11.12. メタデータが割り当てられたエッジ

次に、1つ目のエッジを右クリックし、コンテキスト・メニューから「**Propagate metadata**」を選択します。2つ目のエッジも実線になります。

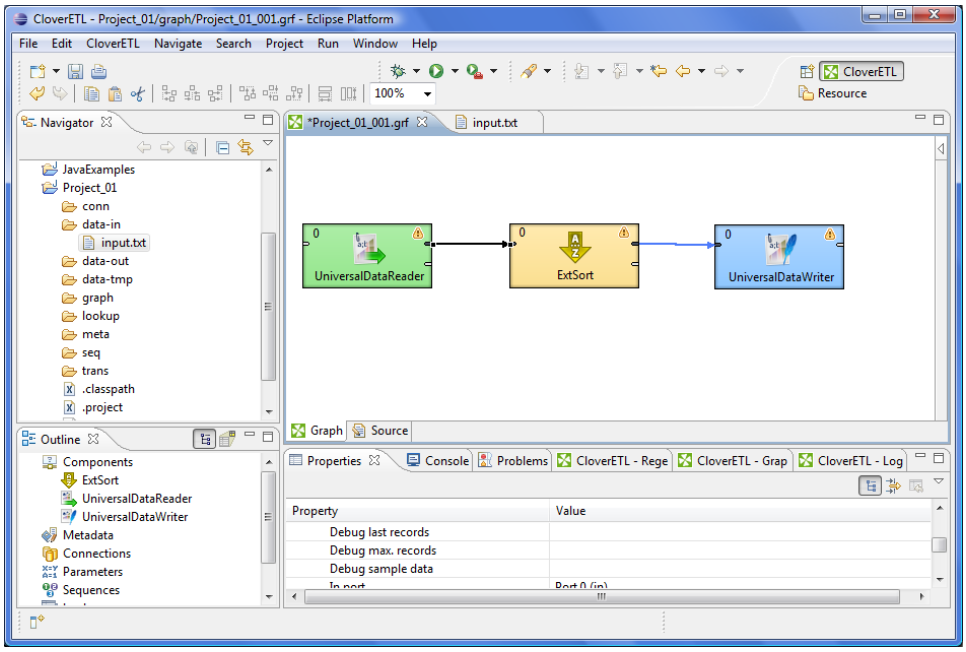


図11.13. コンポーネント全体へのメタデータの伝播

UniversalDataReaderをダブルクリックし、「**File URL**」属性行をクリックして、この「**File URL**」属性行に表示されるボタンをクリックします。

(**UniversalDataReader**の詳細は、UniversalDataReader(p.415)を参照してください。)

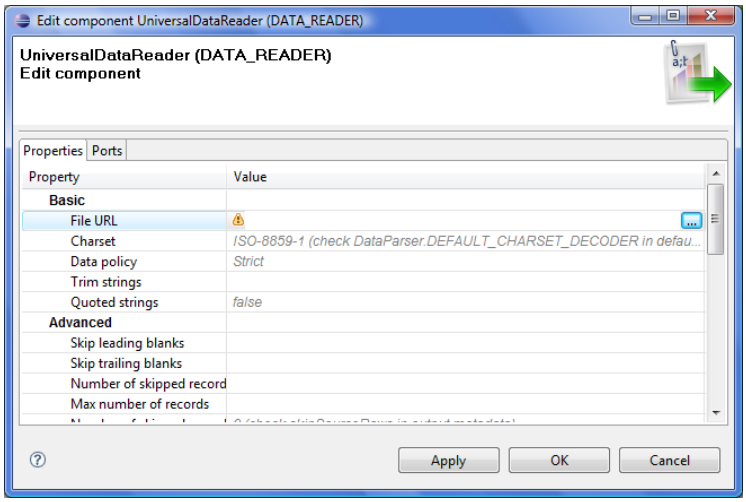


図11.14. 属性行を開く

URLファイル・ダイアログ(p.69)が開きます。`data-in`フォルダをダブルクリックし、このフォルダ内の`input.txt`ファイルをダブルクリックします。右ペインにファイル名が表示されます。

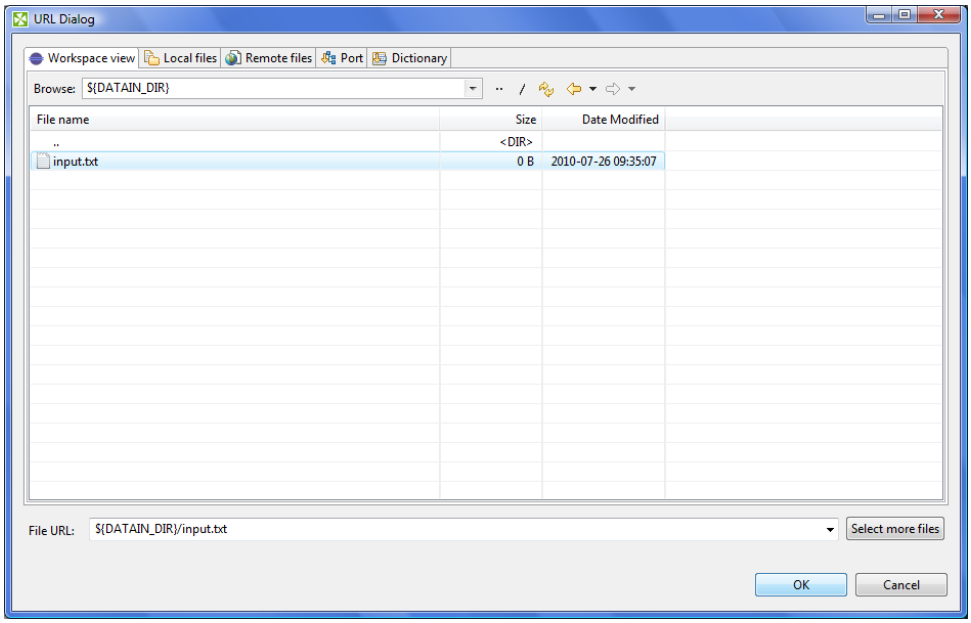


図11.15. 入力ファイルの選択

「**OK**」をクリックします。「**File URL**」属性行が次のように表示されます。

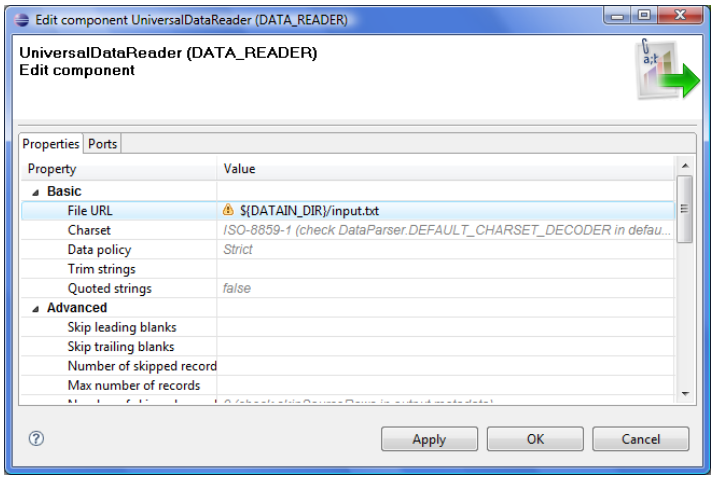


図11.16. 入力ファイルURL属性の設定完了

「**OK**」をクリックして、**UniversalDataReader**エディタを閉じます。

次に、**UniversalDataWriter**をダブルクリックします。

(**UniversalDataWriter**の詳細は、UniversalDataWriter(p.552)を参照してください。)

「**File URL**」属性行をクリックして、この「**File URL**」属性行に表示されるボタンをクリックします。URLファイル・ダイアログ(p.69)が開きます。`data-out`フォルダをダブルクリックします。「**OK**」をクリックします。「**File URL**」属性行が次のように表示されます。

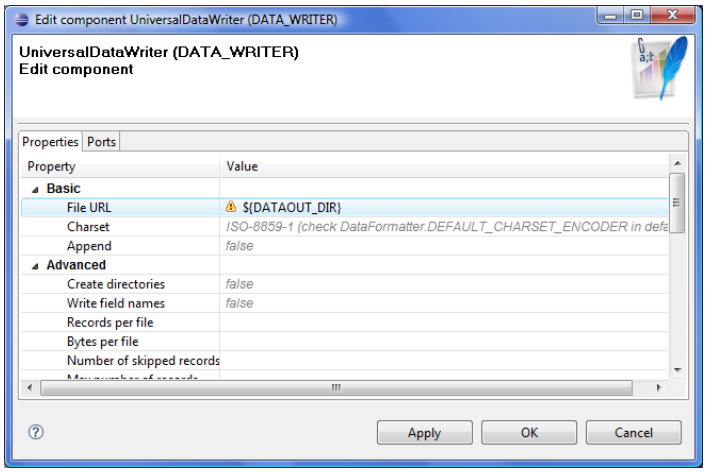


図11.17. ファイル未設定の出力ファイルURL

「**File URL**」属性行をダブルクリックして、`/output.txt`と入力します。結果は次のようになります。

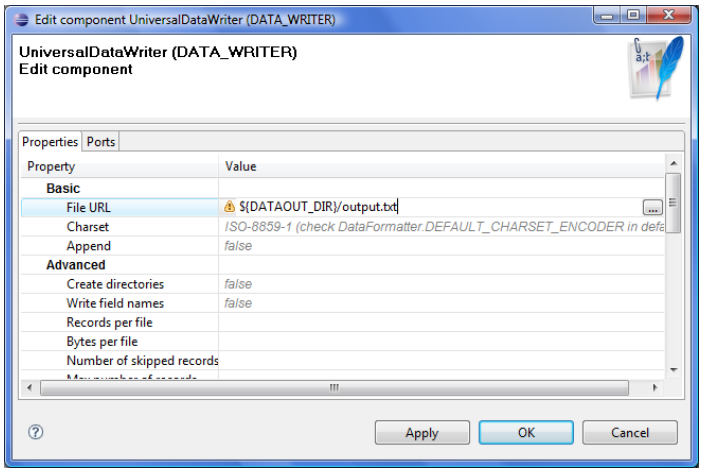


図11.18. ファイルが設定された出力ファイルURL

「**OK**」をクリックして、**UniversalDataWriter**エディタを閉じます。

後は**ExtSort**コンポーネントを設定するのみです。

(**ExtSort**の詳細は、ExtSort(p.600)を参照してください。)

コンポーネントをダブルクリックし、その「**Sort key**」属性行をダブルクリックします。続いて、2つのメタデータ・フィールドを左ペイン(「**Fields**」)から右ペイン(「**Key parts**」)に移動します。初めに「**Surname**」を移動し、次に「**Firstname**」を移動します。

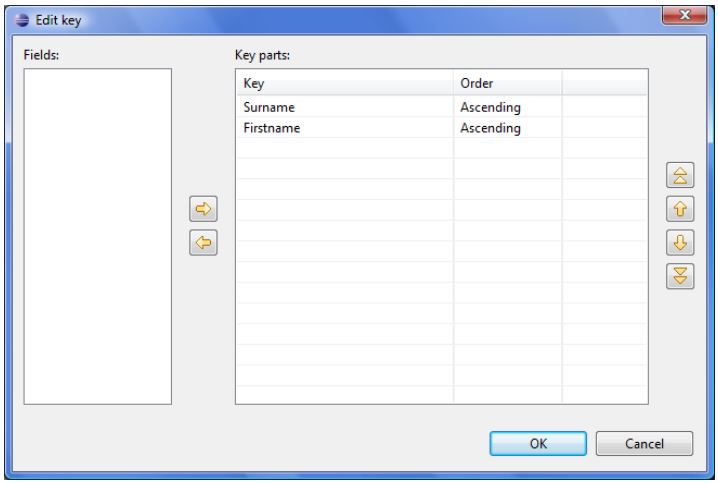


図11.19. ソート・キーの定義

「**OK**」をクリックすると、「**Sort key**」属性行が次のように表示されます。

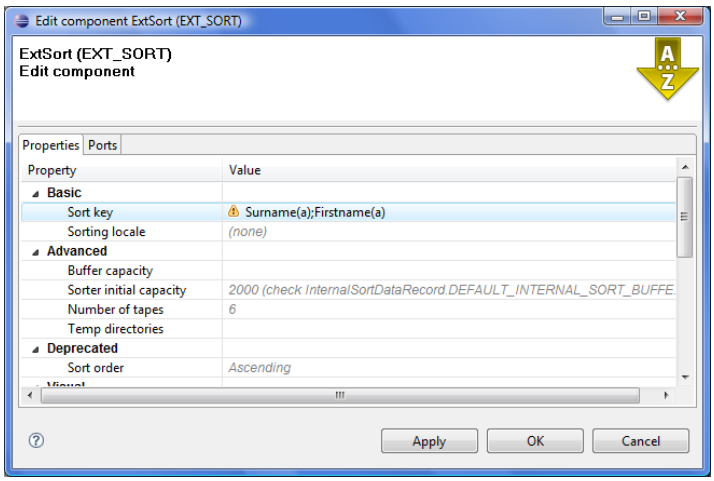


図11.20. ソート・キーの定義完了

「**OK**」をクリックして**ExtSort**エディタを閉じ、**[Ctrl]キーを押しながら[S]キー**を押してグラフを保存します。

グラフ・エディタの任意の場所(コンポーネントまたはエッジ以外)を右クリックして、「**Run As**」→「**CloverETL graph**」を選択します。

(グラフの実行方法の詳細は、第12章「CloverETLグラフの実行」(p.61)を参照してください。)

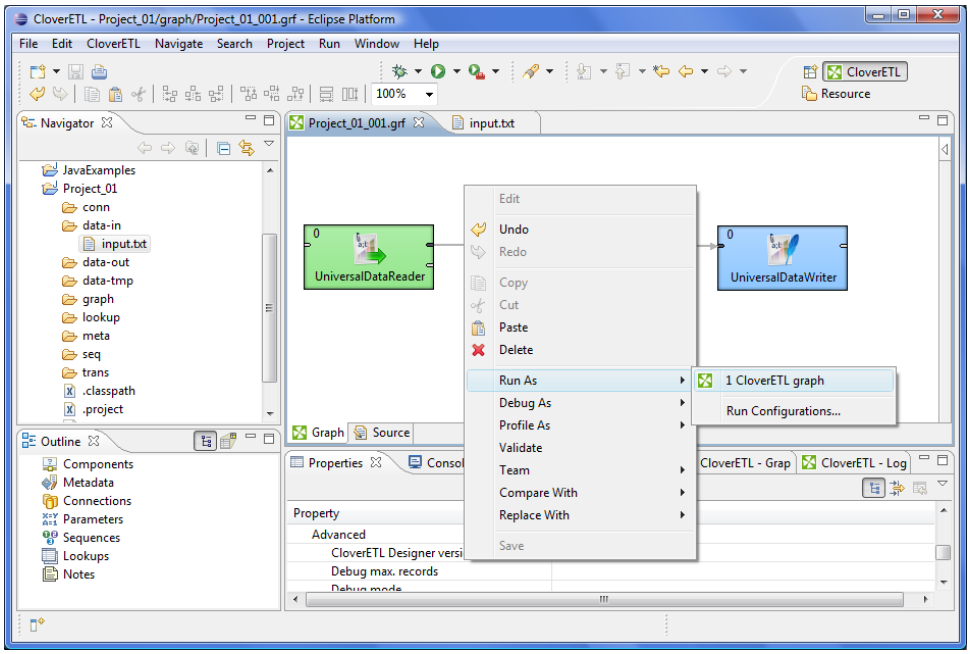


図11.21. グラフの実行

グラフが正常に実行した後は、コンポーネントに青色の円が表示され、解析されたレコードの数がエッジの下に表示されます。

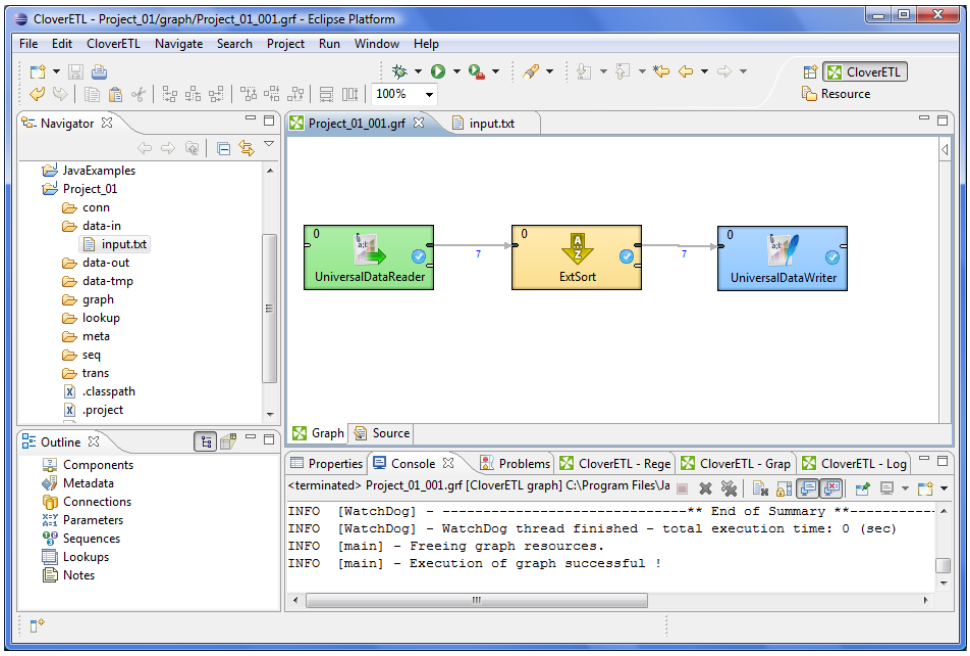


図11.22. グラフの正常な実行の結果

「**Navigator**」ペインで`data-out`フォルダを展開して出力ファイルを開くと、ファイルのコンテンツが次のように表示されます。

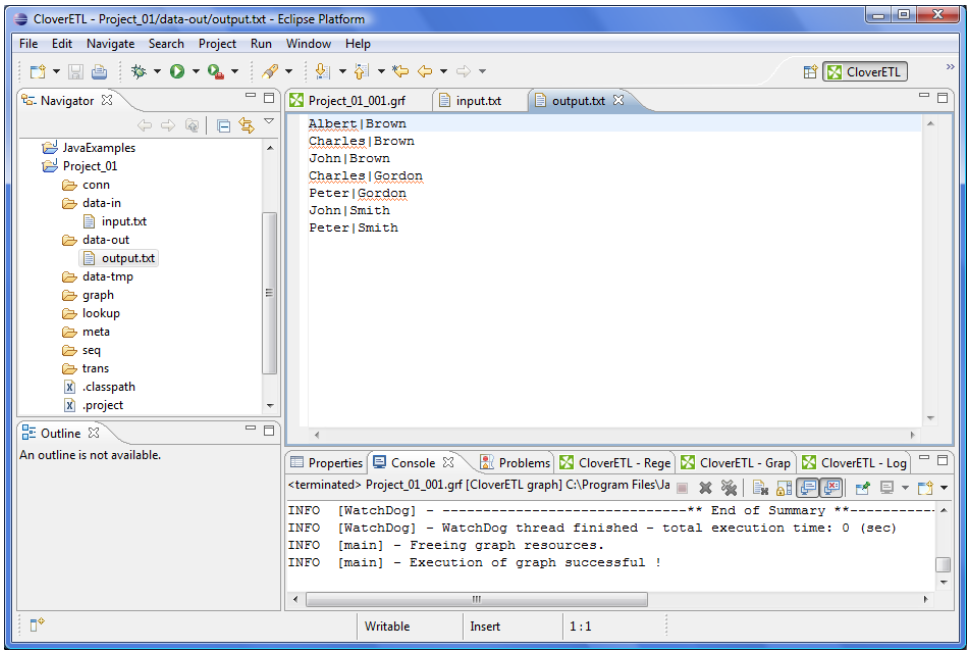


図11.23. 出力ファイルのコンテンツ

全員がアルファベット順にソートされていることがわかります。姓が先、名が後です。これで最初のグラフの作成と実行が完了しました。

第12章 CloverETLグラフの実行

前述したように、コンテキスト・メニューからグラフを実行できることに加え、他の場所からグラフを実行できます。

すでにグラフを作成しているか、プロジェクトにインポートしている場合は、様々な方法で実行できます。

最も簡単にグラフを実行する方法は4つあります。

- メイン・メニューから「**Run**」→「**Run As**」→「**CloverETL graph**」を選択します。
- または、**グラフ・エディタ**内で右クリックし、コンテキスト・メニューの「**Run As**」を選択して、「**CloverETL graph**」項目をクリックします。
- または、ウィンドウの上部にあるツールバーの白い三角形が付いた緑色の円をクリックします。

ヒント

ジョブフロー(p.684)を実行するには、同じ手順に従い、最後の手順で「**CloverETL Jobflow**」を選択します。一部のジョブ制御コンポーネントについては、Clover Server環境で作業する必要があります。つまり、プロジェクトをサーバー・サンドボックスに(p.81)エクスポートする必要がある場合があります。

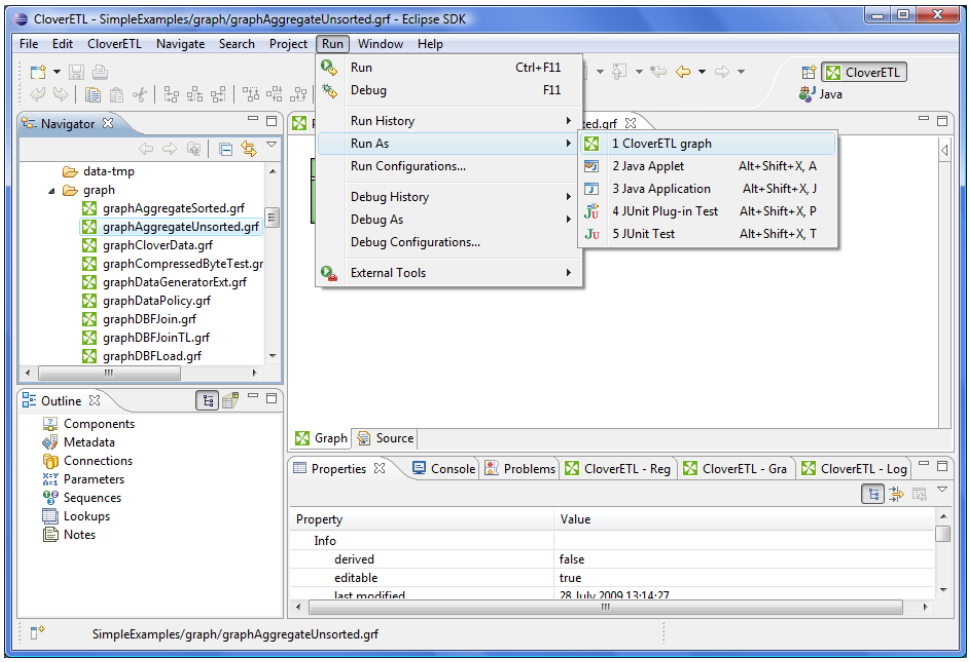


図12.1. メイン・メニューからのグラフの実行

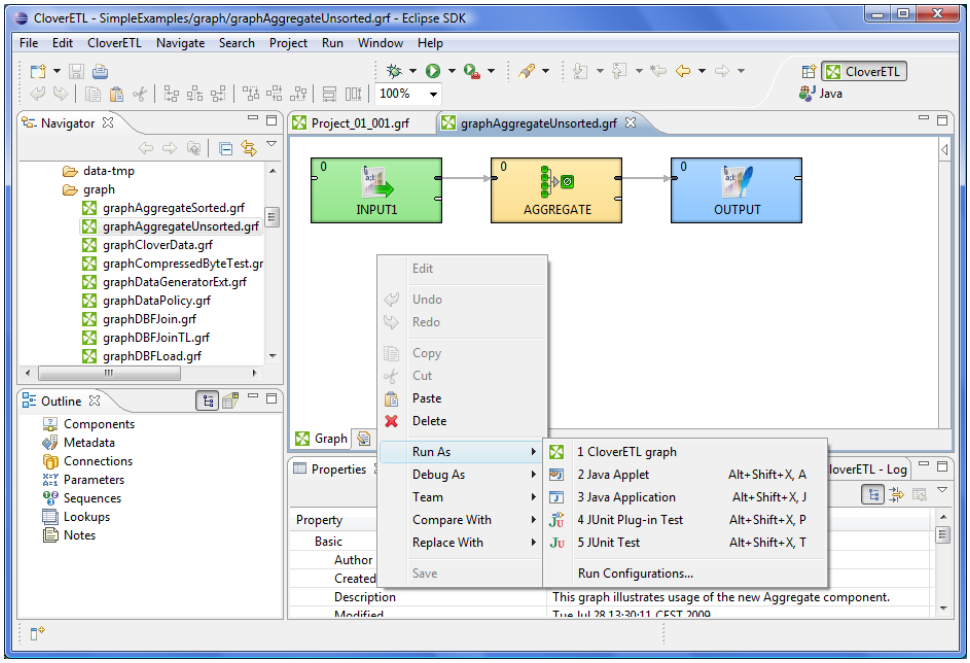


図12.2. コンテキスト・メニューからのグラフの実行

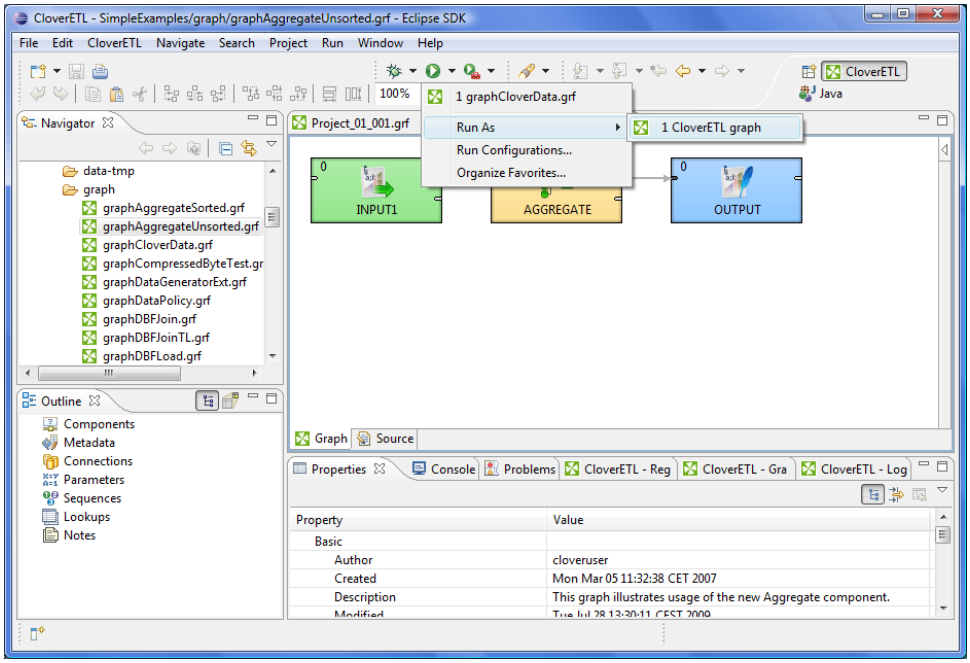


図12.3. 上部ツールバーからのグラフの実行

グラフの正常な実行

グラフの実行後は、グラフ実行のプロセスを「**Console**」タブおよびその他のタブで確認できます。(詳細は、タブ・ペイン(p.43)を参照してください。)

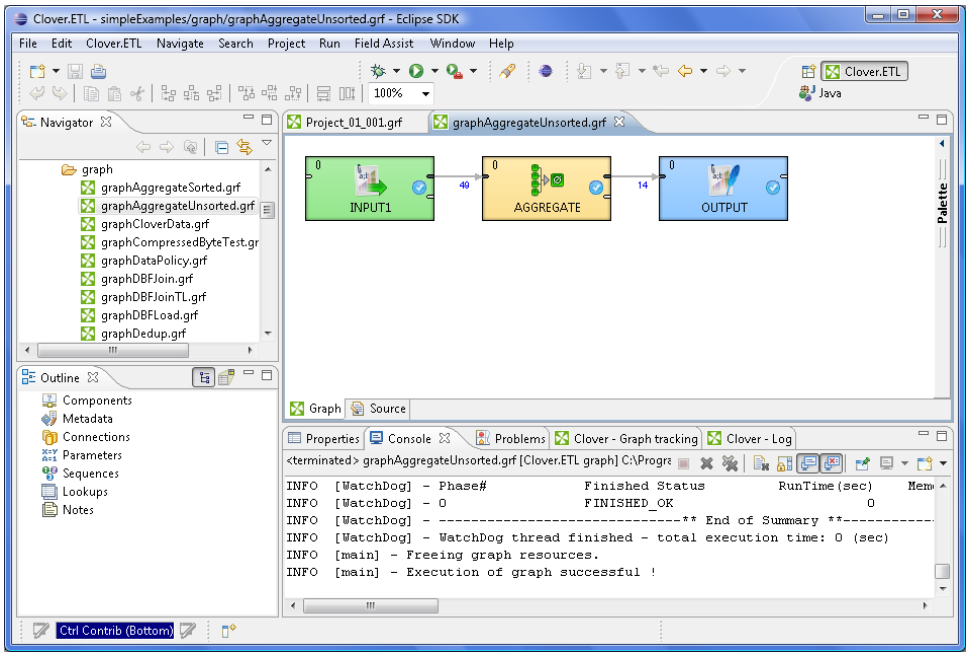


図12.4. グラフの正常な実行

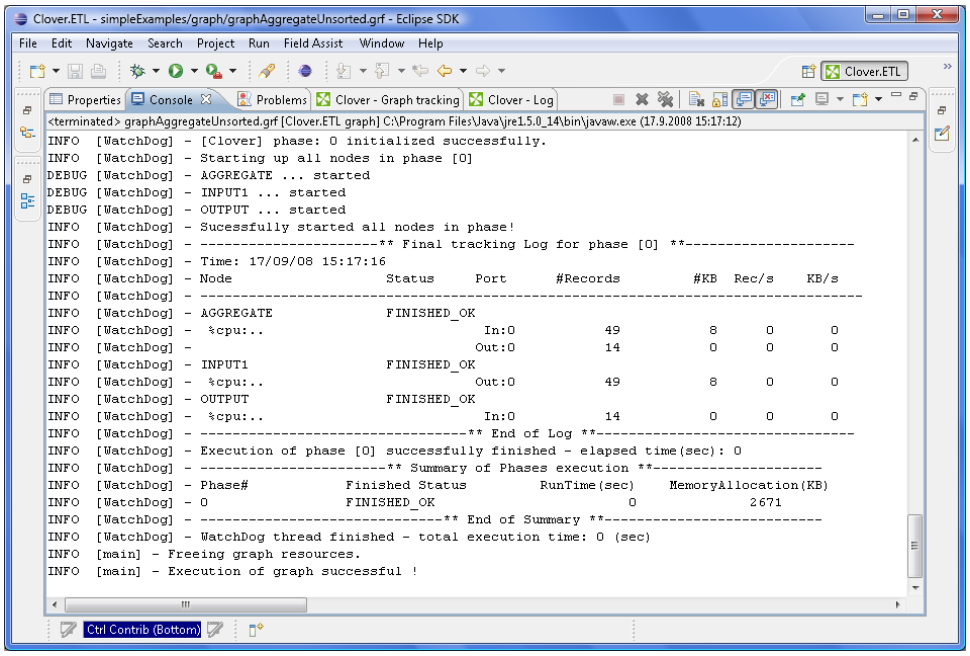


図12.5. グラフ実行の概要を表示した「Console」タブ

さらに、エッジの下には、処理されたデータのカウントが表示されます。

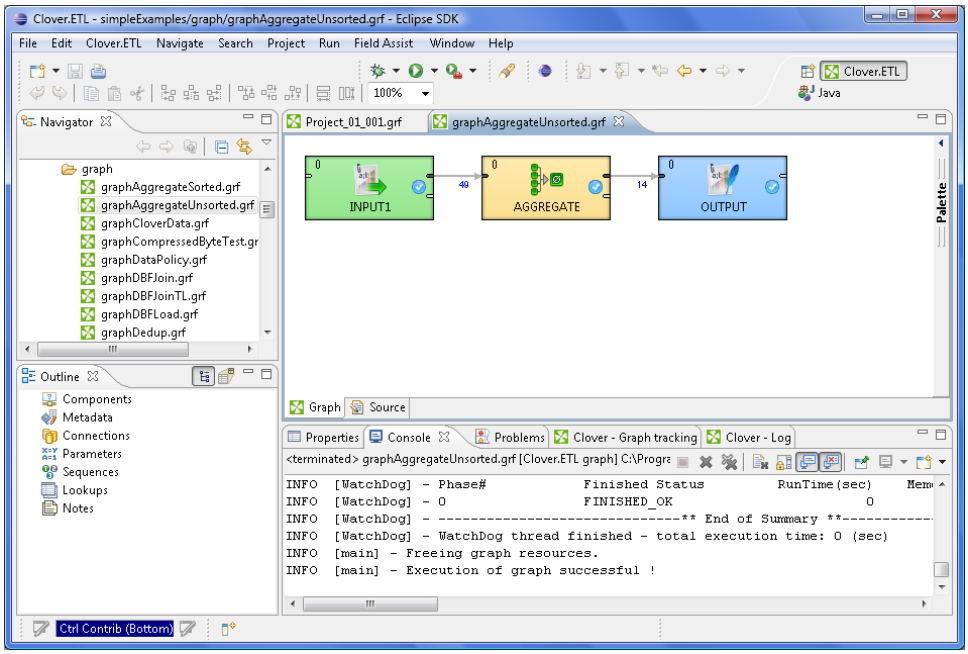


図12.6. 解析済データのカウント

「Run Configurations」ダイアログの使用方法

前述のオプションに加えて、「**Run Configurations**」ダイアログを開き、プロジェクト名、グラフ名を入力し、プログラムおよびVM引数、パラメータなどを設定して「**Run**」ボタンをクリックすることもできます。

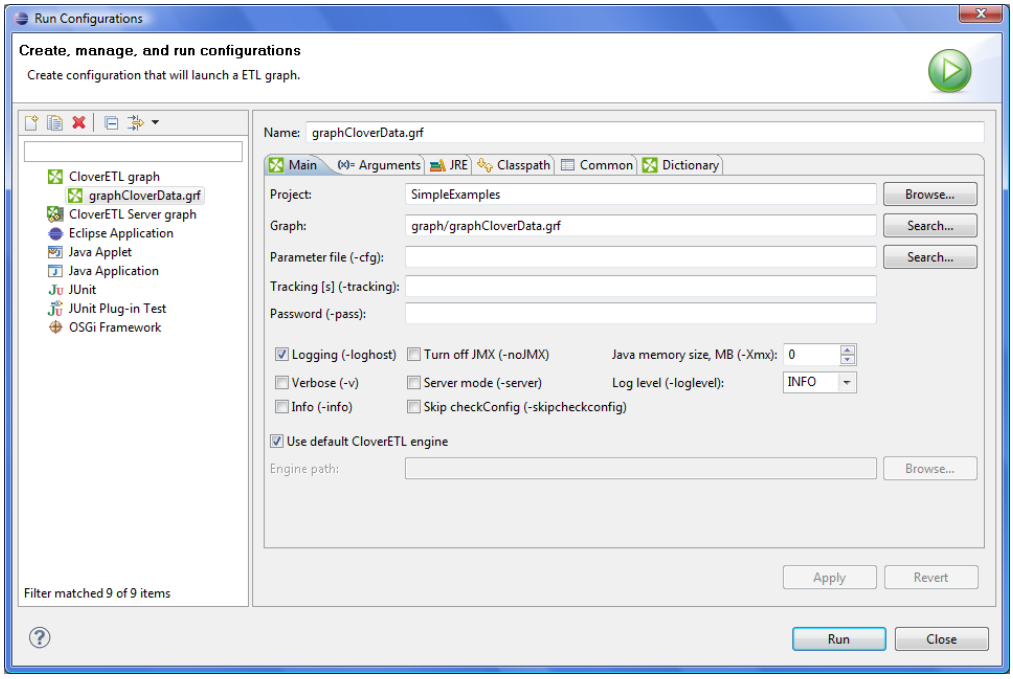


図12.7.「Run Configurations」ダイアログ

「**Run Configurations**」ダイアログの使用方法の詳細は、第18章「高度なトピック」(p.85)を参照してください。

第IV部 CloverETL Designerの操作

第13章 チート・シートの使用方法

チート・シートとは、**CloverETL Designer**をはじめとするアプリケーションの使用方法の学習、準備済のタスクの実行および新しいタスクの作成に使用する対話型のツールです。

CloverETL Designerには2種類のチート・シートが含まれています。メイン・メニューから「**Help**」→「**Cheat Sheets...**」を選択することによって表示できます。

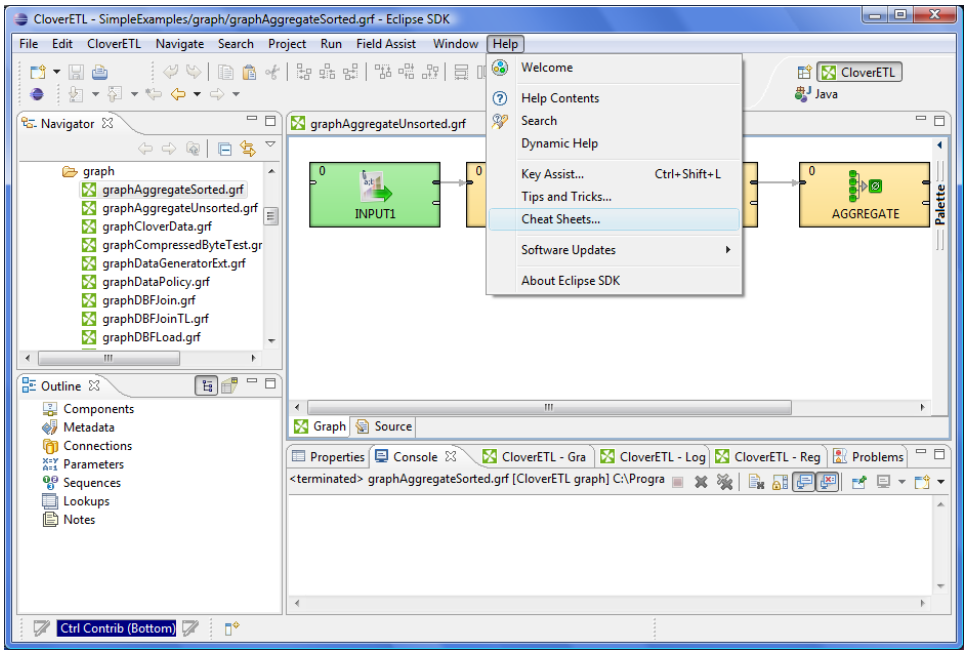


図13.1. チート・シートの選択

次のようなウィンドウが開きます。

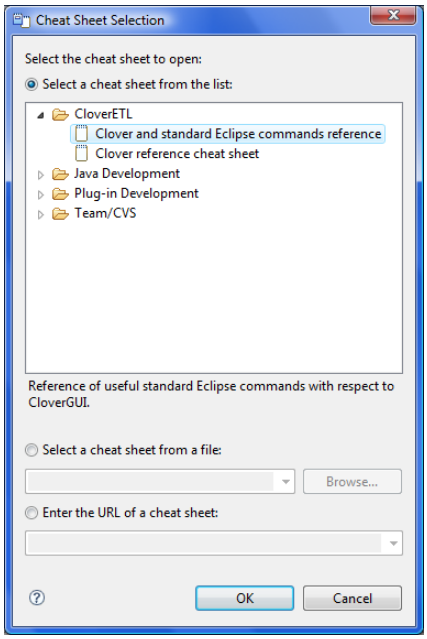


図13.2. 「Cheat Sheet Selection」ウィザード

CloverETL Designerの2種類のチート・シートは次のとおりです。

- Clover and standard Eclipse commands reference
- Clover reference cheat sheet

1つ目は、**CloverETL Designer**に関係のある、便利な標準の**Eclipse**コマンドについて説明および指導します。

2つ目は、**CloverETL Designer**のチート・シートの記述方法のガイドを表示します。

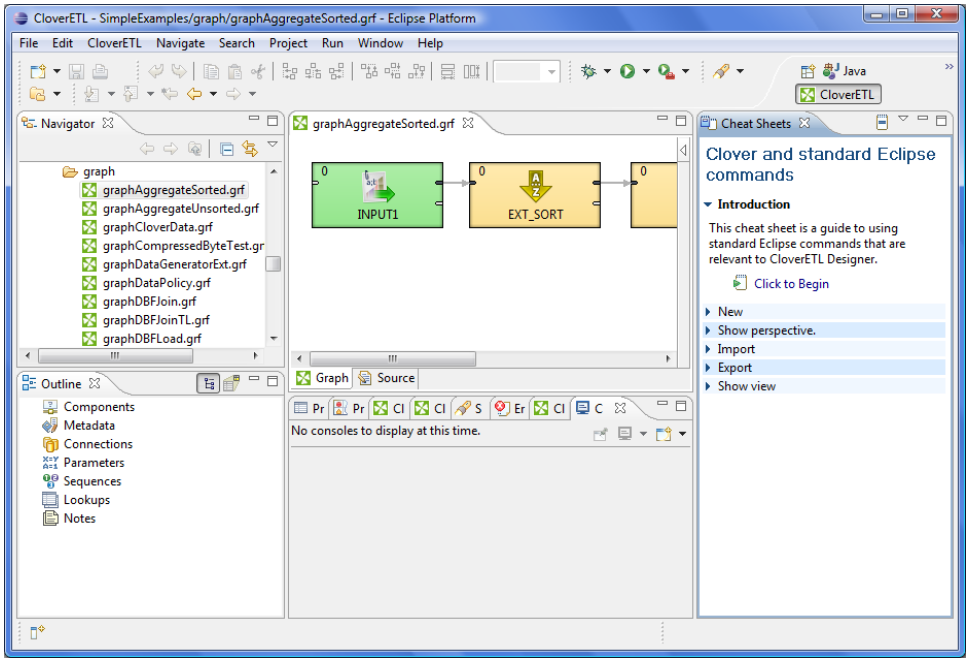


図13.3. CloverETLおよび標準のEclipseコマンド(縮小状態)

項目を展開すると、**CloverETL**および**Eclipse**コマンドの概要を表示できます。

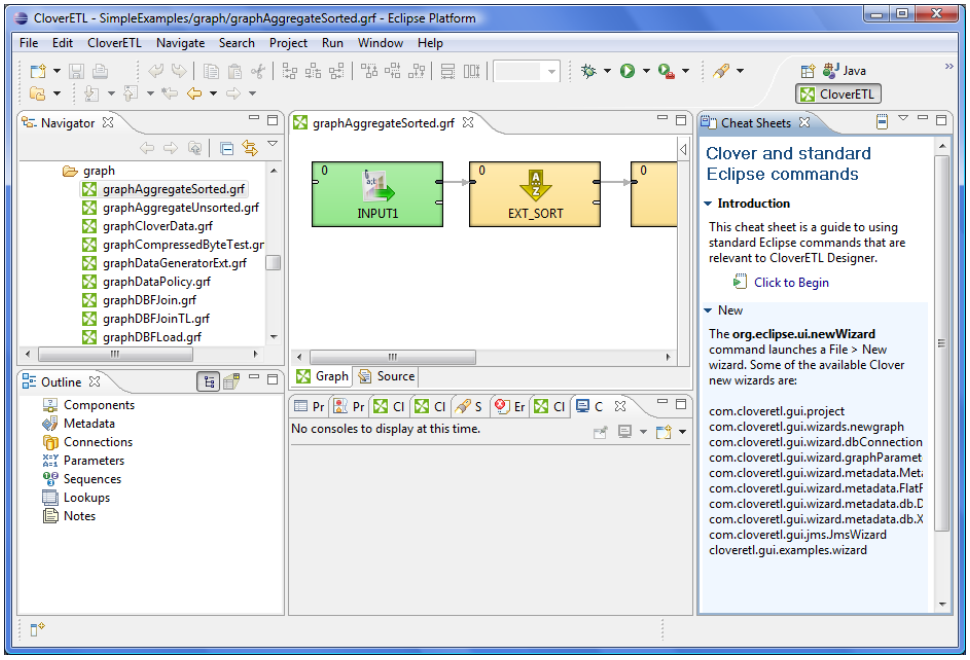


図13.4. CloverETLおよび標準のEclipseコマンド(展開状態)

2つ目のチート・シートは、**CloverETL Designer**の操作方法を指導します。指示に従って新しいプロジェクトや新しいグラフなどを作成できます。

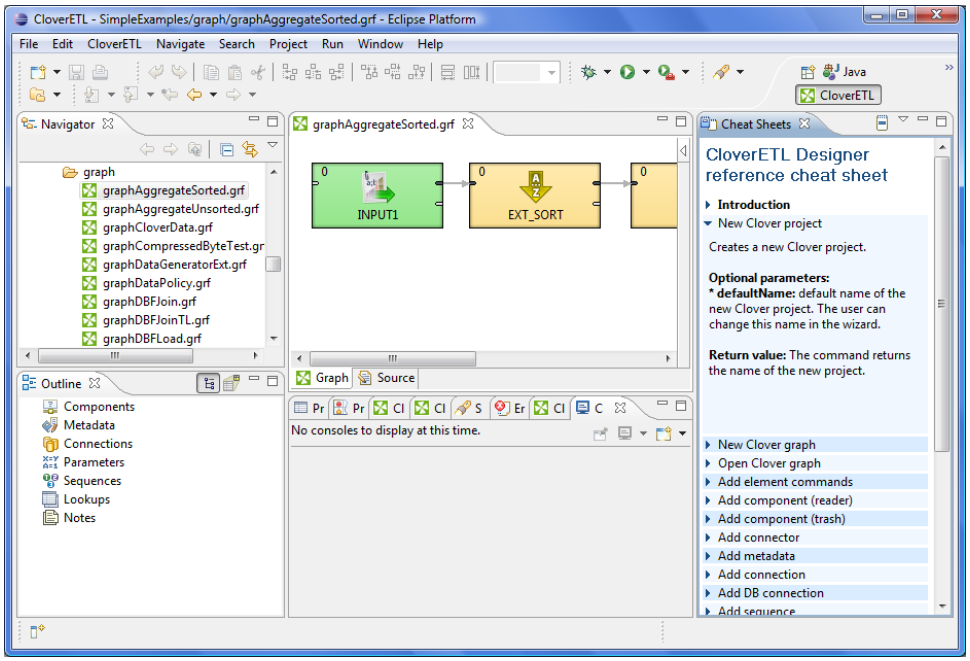


図13.5. CloverETL Designerのリファレンス・チート・シート

これら2つのチート・シートの基礎は、2つの`.xml`ファイルです。これらは**CloverETL Designer**の`plugins`ディレクトリ内にある`com.cloveretl.gui.cheatsheet_2.9.0.jar`ファイルに含まれており、この`.jar`ファイルの`cheatsheets`サブディレクトリに置かれています。それぞれ`EclipseReference.xml`ファイルおよび`Reference.xml`ファイルです。

準備済のチート・シートに加えて、これら2つのファイルをパターンとして使用して、独自の別のチート・シートを記述できます。カスタム・チート・シートの記述後は、単に「**Select a cheat sheet from a file**」ラジオ・ボタンを選択します。これにより、ディスクを参照してカスタム・チート・シートの`.xml`ファイルを見つけることができます。「**OK**」をクリックすると、自分のチート・シートが、準備済の2つのチート・シートと同様に、**CloverETL Designer**ウィンドウの右側に表示されます。

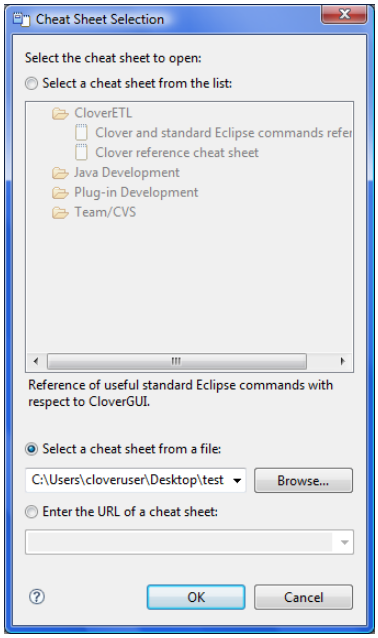


図13.6. カスタム・チート・シートの参照

第14章 共通ダイアログ

最も一般的なダイアログのリストを次に示します。

URLファイル・ダイアログ

ほとんどのコンポーネントでは、特定のファイルのURLも指定する必要があります。これらのファイルは、読み取る必要があるデータのソース、データを書き込む必要があるソース、またはコンポーネント内を流れるデータの変換に使用する必要があるファイルおよびその他のファイルURLの検索に役立ちます。このようなファイルURLを指定するために、**URLファイル・ダイアログ**を使用できます。

URLファイル・ダイアログを開くと、いくつかのタブが表示されます。

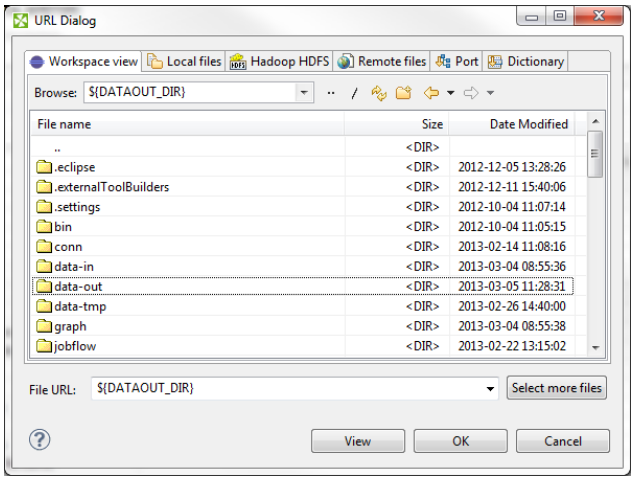


図14.1. URLファイル・ダイアログ

- **Workspace view**

  ローカル**CloverETL**プロジェクトのワークスペース内のファイルの検索に使用します。

- **Local files**

  localhost上のファイルの検索に使用します。コンボ・ボックスには各ディスクおよびパラメータが含まれています。**CloverETLプロジェクト**およびその他のローカル・ファイルの両方の指定に使用できます。

- **Clover Server**

  開いているすべての**CloverETL Serverプロジェクト**のファイルの検索に使用します。**CloverETL Server**プロジェクトのみに使用可能です。

- **Remote files**

  リモート・コンピュータまたはインターネット上のファイルの検索に使用します。接続のプロパティ、プロキシ設定およびHTTPプロパティを指定できます。

- **Port**

  ポートの読取りまたは書込みに使用するフィールドおよび処理タイプの指定に使用します。そのようなデータ・ソースまたはターゲットが許容されるコンポーネントの場合のみ開きます。

- **Dictionary**

  ディクショナリの読取りまたは書込みに使用するディクショナリ・キー値および処理タイプの指定に使用します。そのようなデータ・ソースまたはターゲットが許容されるコンポーネントの場合のみ開きます。

重要

グラフの移植性を確保するため、URL内のパスを定義する場合にはスラッシュが使用されます(Microsoft Windowsの場合でも)。

注意

「**Workspace view**」タブおよび「**Local files**」タブのツールバーでは、「**New Directory**」アクションを使用できます。このアクションのショートカットとして[F7]キーを使用できます。新しく作成されたディレクトリがダイアログで選択され、その名前をインライン編集できます。[F2]キーを使用してディレクトリの名前を変更し、[DEL]キーを使用して削除できます。

前述の各タブの**URL**の詳細は、次の項を参照してください。

- リーダーにサポートされているファイルURL形式(p.297)
- ライターにサポートされているファイルURL形式(p.310)

「Edit value」ダイアログ

「**Edit value**」ダイアログには単純なテキスト領域が表示され、**JMSReader**、**JMSWriter**および**JavaExecute**コンポーネントの変換コードを記述できます。

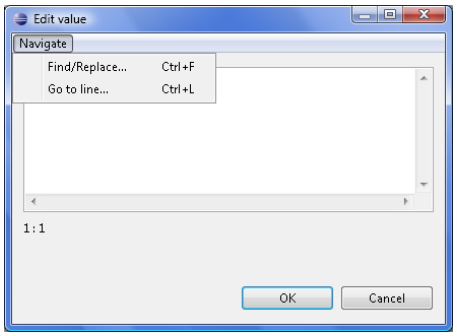


図14.2. 「Edit value」ダイアログ

左上隅の「**Navigate**」ボタンをクリックすると、選択可能なオプションのリストが表示されます。「**Find**」または「**Go to line**」を選択できます。

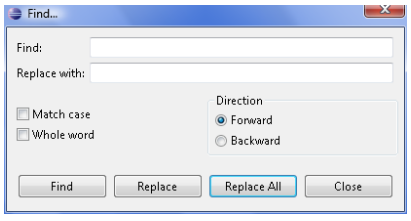


図14.3. 「Find」ウィザード

「**Find**」項目をクリックすると、別のウィザードが表示されます。ここでは、検索する式を入力し(「**Find**」テキスト領域)、完全一致のみを検索するかどうか(「**Whole word**」)、大/小文字を区別するかどうか(「**Match case**」)、および**検索方向**(下向きは「**Forward**」、上向きは「**Backward**」)を決定できます。これらのオプションを選択するには、表示されているチェック・ボックスまたはラジオ・ボタンを選択する必要があります。

「**Go to line**」項目をクリックすると、新しいウィザードが開き、移動先の行番号を入力する必要があります。

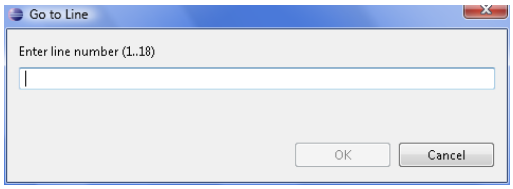


図14.4.「Go to Line」ウィザード

「Open Type」ダイアログ

このダイアログは、必要な変換を定義するクラス(**変換クラス、非正規化クラスなど**)の選択に使用します。開くときは、単にクラス名の最初の部分を入力します。最初の部分を入力することにより、入力された文字に一致するクラスがウィザードに表示され、適切なクラスを選択できます。

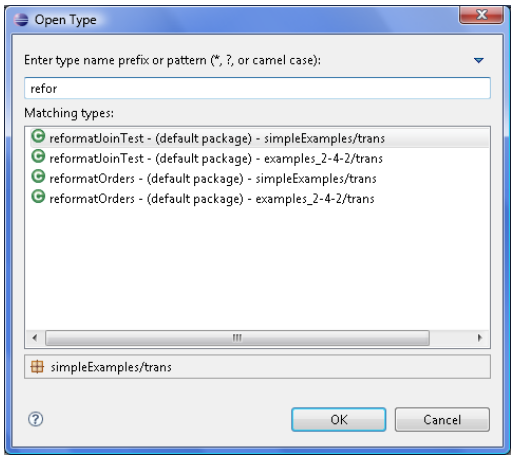


図14.5.「Open Type」ダイアログ

第15章 インポート

CloverETL Designerでは、すでに準備済の**CloverETL**のプロジェクト、グラフまたはメタデータをインポートできます。インポートする場合は、メイン・メニューから「**File**」→「**Import...**」を選択します。

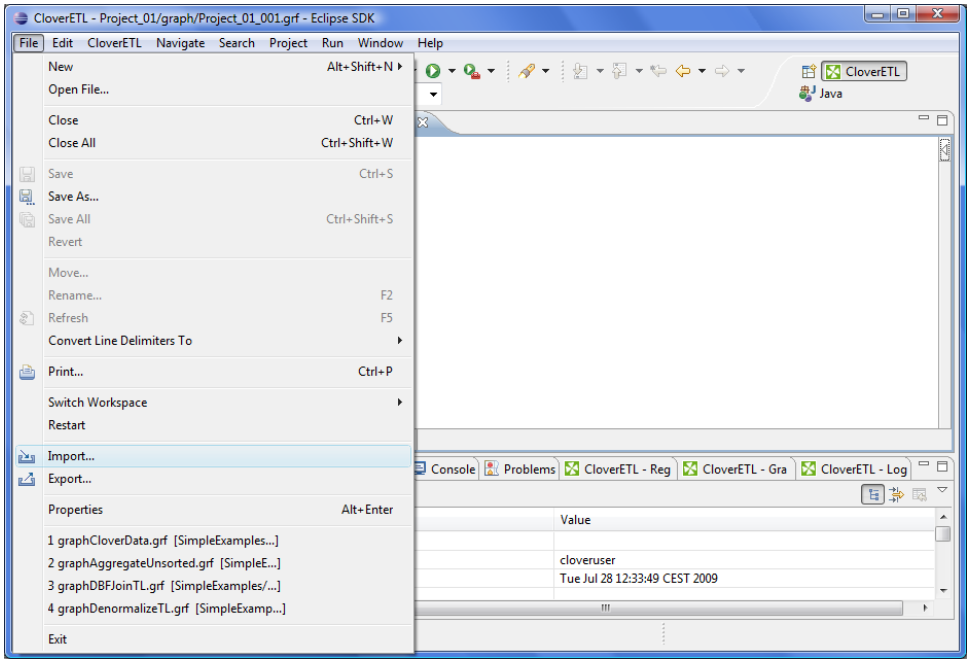


図15.1. 「Import」(メイン・メニュー)

または、「**Navigator**」ペインを右クリックし、コンテキスト・メニューから「**Item...**」を選択します。

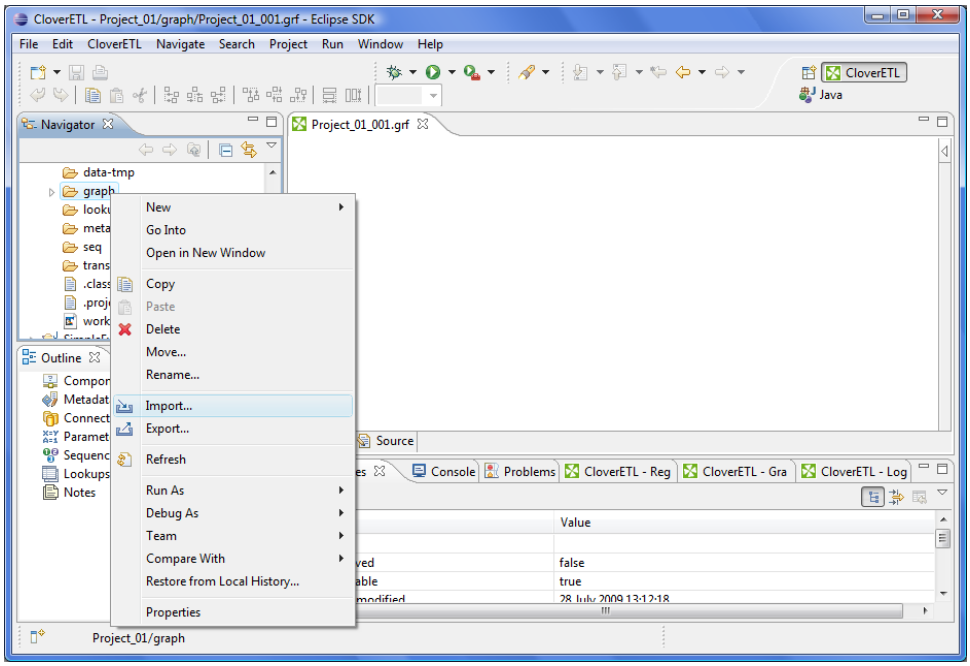


図15.2. 「Import」(コンテキスト・メニュー)

これで、次のウィンドウが開きます。**Clover ETL**のカテゴリを展開すると、ウィンドウが次のように表示されます。

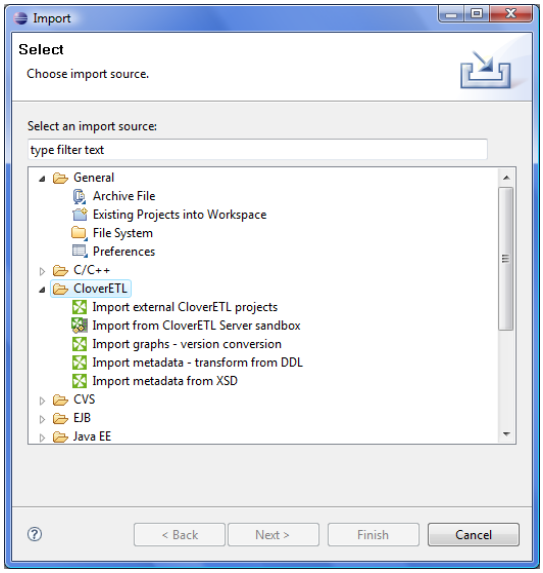


図15.3. インポート・オプション

CloverETLプロジェクトのインポート

「**Import external CloverETL projects**」項目を選択した場合、「**Next**」ボタンをクリックすると、次のウィンドウが表示されます。

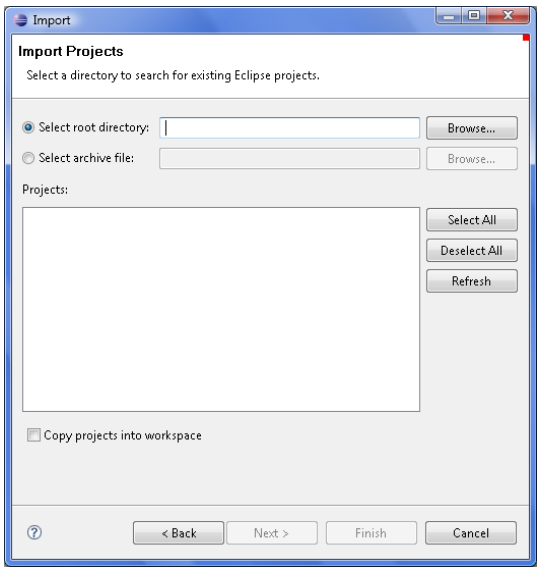


図15.4. プロジェクトのインポート

ディレクトリまたは圧縮済アーカイブ・ファイルを検索できます(ラジオ・ボタンを切り替えて、正しいオプションを選択する必要があります)。ディレクトリを検索する場合、プロジェクトをワークスペースにコピーするか、またはリンクするかも決定できます。プロジェクトのリンクのみが必要な場合は、「**Copy projects into workspace**」チェック・ボックスの選択を解除できます。このようにしなかった場合、コピーされます。リンクされたプロジェクトは、より多くのワークスペースに格納されます。アーカイブ・ファイルを選択した場合、そのアーカイブに格納されているプロジェクトのリストが「**Projects**」領域に表示されます。それらとともに表示されるチェック・ボックスを選択して、それらの一部またはすべてを選択できます。

CloverETL Serverサンドボックスからのインポート

CloverETL Designerでは、**CloverETL Server**サンドボックスの任意の部分をインポートできるようになっています。インポートするには、「**Import from CloverETL Server Sandbox**」オプションを選択します。これで、次のウィザードが開きます。

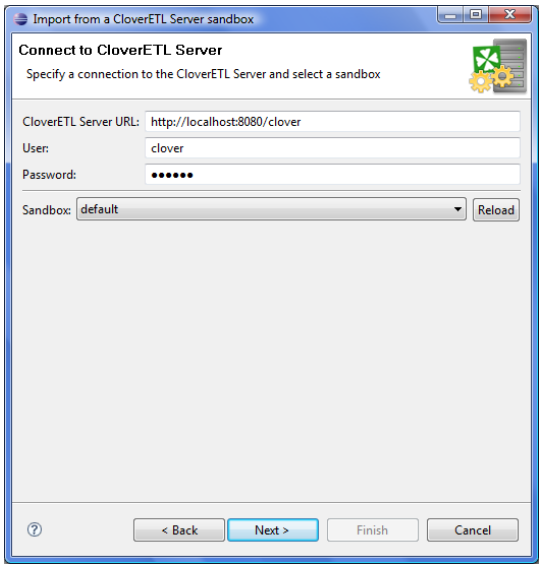


図15.5. CloverETL Serverサンドボックスからのインポートのウィザード(CloverETL Serverに接続)

CloverETL ServerのURL、**ユーザー名**および**パスワード**の3つの項目を指定します。次に、「**Reload**」をクリックします。これで、サンドボックスのリストが「**Sandbox**」メニューに表示されるようになります。いずれかを選択して、「**Next**」をクリックします。新しいウィザードが開きます。

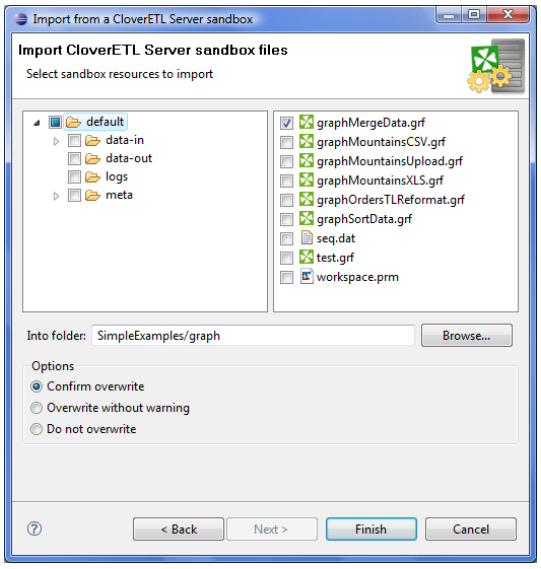


図15.6. CloverETL Serverサンドボックスからのインポートのウィザード(ファイルのリスト)

インポートするファイルまたはディレクトリ(あるいはその両方)を選択し、インポート先のフォルダを選択して、同じ名前を持つファイルやディレクトリを警告なしで上書きするか、上書きを確認するか、または同じ名前を持つファイルやディレクトリを上書きしないかを決定します。次に、「**Finish**」をクリックします。選択したファイルまたはディレクトリ(あるいはその両方)が**CloverETL Server**サンドボックスからインポートされます。

グラフのインポート

「**Import graphs - version conversion**」項目を選択した場合、「**Next**」ボタンをクリックすると、次のウィンドウが表示されます。

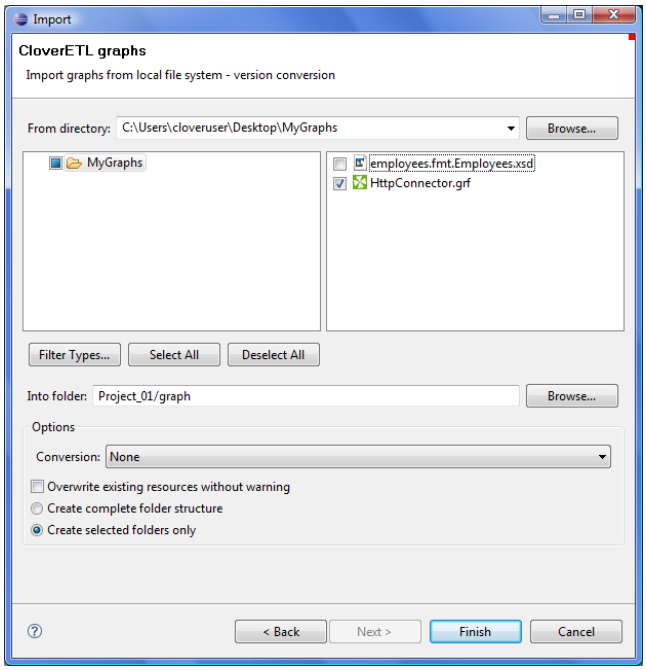


図15.7. グラフのインポート

適切なグラフを選択し、選択したグラフをどのディレクトリからどのフォルダにコピーするかを指定する必要があります。ラジオ・ボタンを切り替えて、フォルダ構造全体を作成するのか、選択したフォルダのみを作成するのかを決定します。また、警告なしで既存のソースを上書きするよう指示することもできます。

注意

CloverETL Designerの1.x.xバージョンから2.x.xバージョンへ、および**CloverETL Engine**の2.x.xバージョンから2.6.xバージョンへ、より古いグラフを変換することもできます。

メタデータのインポート

XSDまたはDDLからメタデータをインポートすることもできます。

メタデータの説明および作成方法の詳細は、第21章「メタデータ」(p.110)を参照してください。

XSDのメタデータ

「**Import metadata from XSD**」項目を選択した場合、「**Next**」ボタンをクリックすると、次のウィンドウが表示されます。

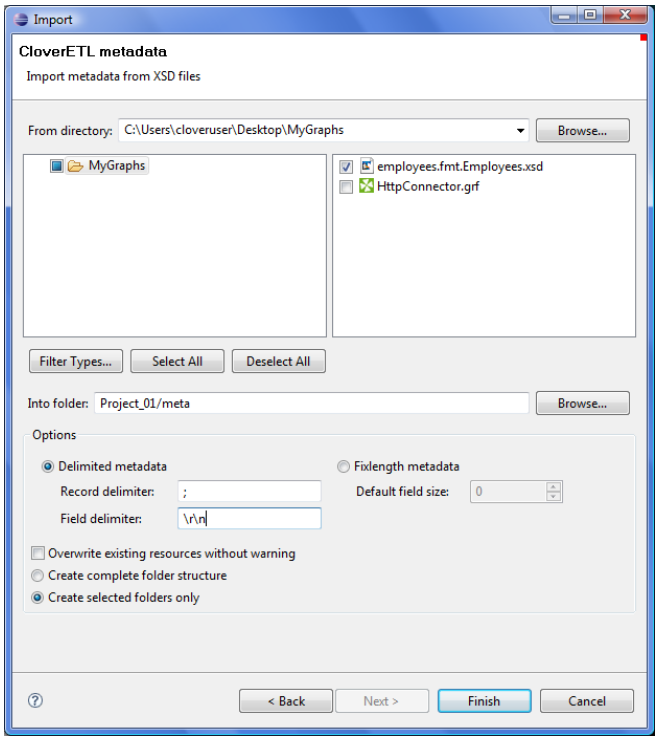


図15.8. XSDからのメタデータのインポート

適切なメタデータを選択し、選択したメタデータをどのディレクトリからどのフォルダにコピーするかを指定する必要があります。ラジオ・ボタンを切り替えて、フォルダ構造全体を作成するのか、選択したフォルダのみを作成するのかを決定します。また、警告なしで既存のソースを上書きするよう指示することもできます。デリミタまたはデフォルト・フィールド・サイズを指定できます。

DDLのメタデータ

「**Import metadata - transform from DDL**」項目を選択した場合、「**Next**」ボタンをクリックすると、次のウィンドウが表示されます。

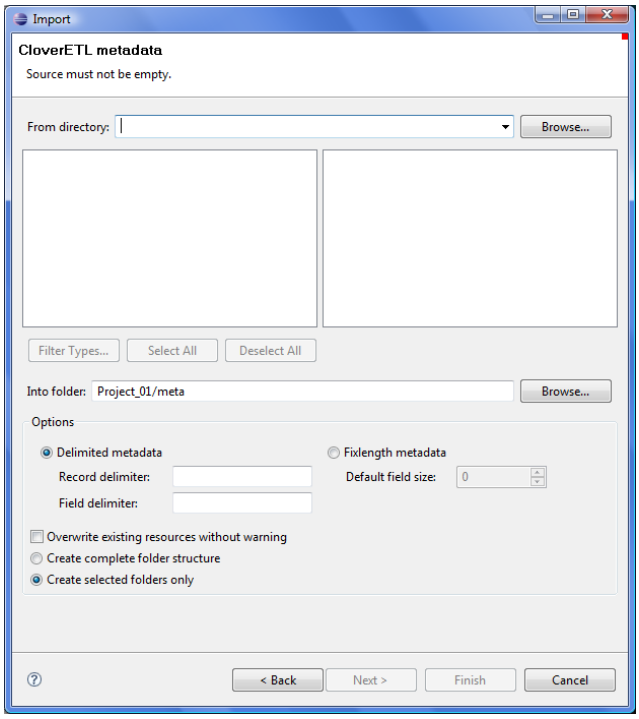


図15.9. DDLからのメタデータのインポート

適切なメタデータを選択し、選択したメタデータをどのディレクトリからどのフォルダにコピーするかを指定する必要があります。ラジオ・ボタンを切り替えて、フォルダ構造全体を作成するのか、選択したフォルダのみを作成するのかを決定します。また、警告なしで既存のソースを上書きするよう指示することもできます。デリミタを指定する必要があります。

第16章 エクスポート

CloverETL Designerでは、独自の**CloverETL**グラフまたはメタデータ(あるいはその両方)をエクスポートできます。エクスポートする場合は、メイン・メニューから「**File**」→「**Export...**」を選択します。または、「**Navigator**」ペインを右クリックし、コンテキスト・メニューから「**Item...**」を選択します。これで、次のウィンドウが開きます。「**CloverETL**」カテゴリを展開すると、ウィンドウが次のように表示されます。

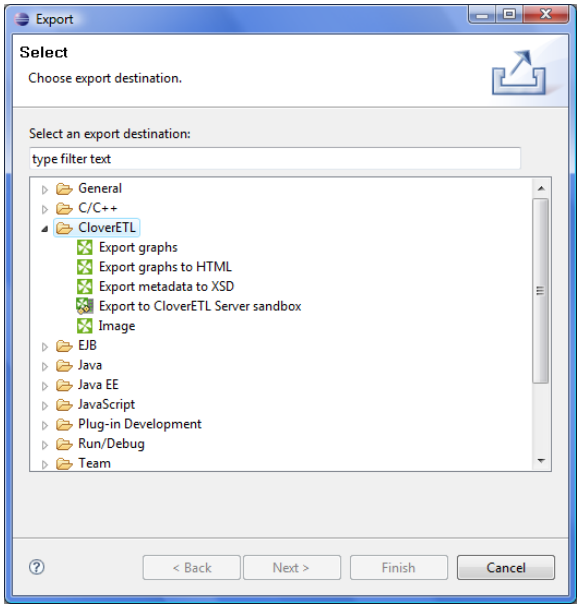


図16.1. エクスポート・オプション

グラフのエクスポート

「**Export graphs**」項目を選択した場合、「**Next**」ボタンをクリックすると、次のウィンドウが表示されます。

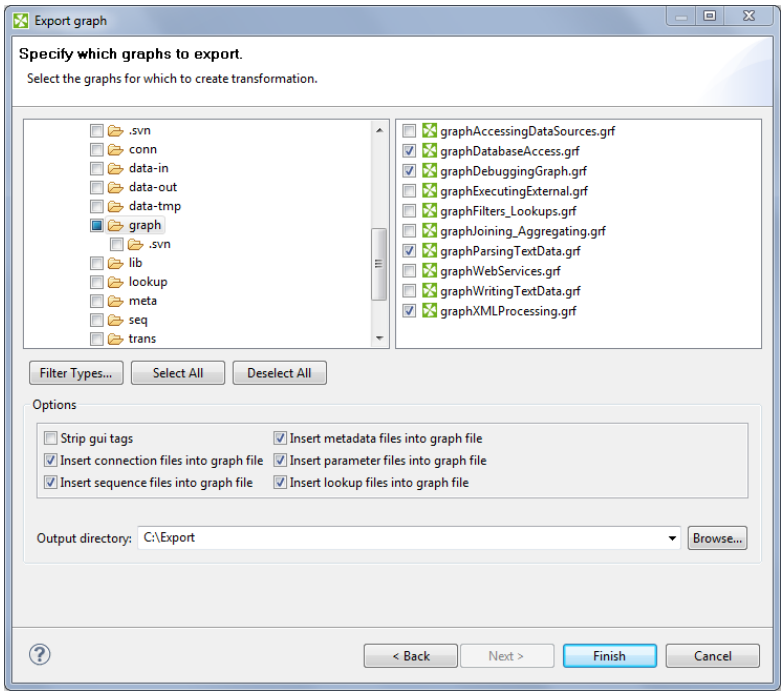


図16.2. グラフのエクスポート

右側のペインで、エクスポートするグラフを選択します。出力ディレクトリも指定する必要があります。さらに、外部(共有)メタデータ、接続、パラメータ、シーケンスおよび参照を取り込んで、グラフに挿入するかどうかも選択できます。これは、対応するチェック・ボックスを選択して行う必要があります。また、「**Strip gui tags**」チェック・ボックスの選択を解除して、出力ファイルからGUIタグを削除することもできます。

HTMLへのグラフのエクスポート

「**Export graphs to HTML**」項目を選択した場合、「**Next**」ボタンをクリックすると、次のウィンドウが表示されます。

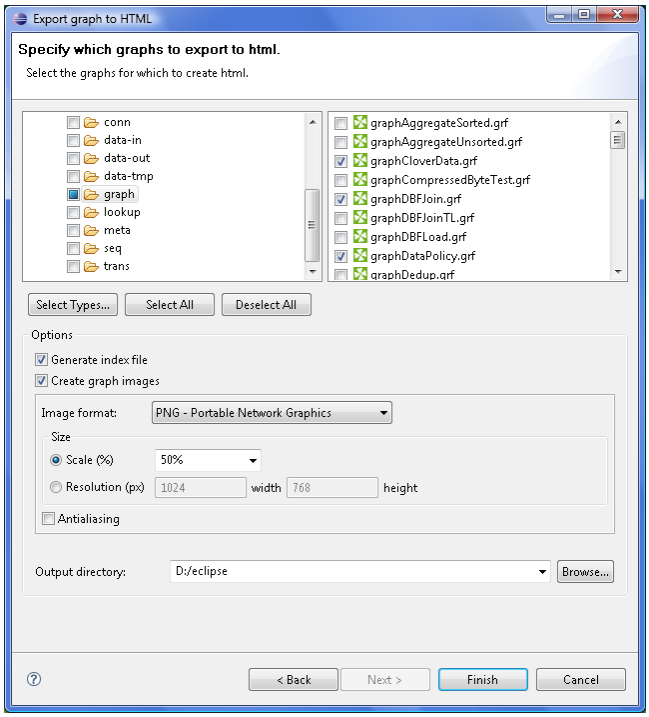


図16.3. HTMLへのグラフのエクスポート

グラフを選択し、選択したグラフをどの出力ディレクトリにエクスポートするかを指定する必要があります。エクスポートされたページおよび対応するグラフの索引ファイルや、選択したグラフのイメージを生成することもできます。ラジオ・ボタンを切り替えることで、出力イメージのスケールまたはイメージの幅と高さのいずれかを選択します。アンチエイリアスを使用するかどうかも決定できます。

XSDへのメタデータのエクスポート

「**Export metadata to XSD**」項目を選択した場合、「**Next**」ボタンをクリックすると、次のウィンドウが表示されます。

メタデータの説明および作成方法の詳細は、第21章「メタデータ」(p.110)を参照してください。

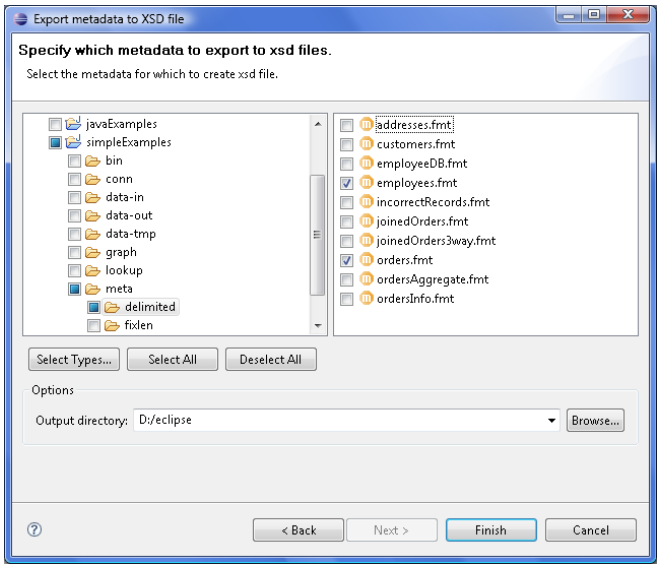


図16.4. XSDへのメタデータのエクスポート

メタデータを選択し、選択したメタデータをどの出力ディレクトリにエクスポートするかを指定する必要があります。

CloverETL Serverサンドボックスへのエクスポート

CloverETL Designerでは、プロジェクトの任意の部分を**CloverETL Server**サンドボックスにエクスポートできるようになっています。エクスポートするには、「**Export to CloverETL Server sandbox**」オプションを選択します。これで、次のウィザードが開きます。

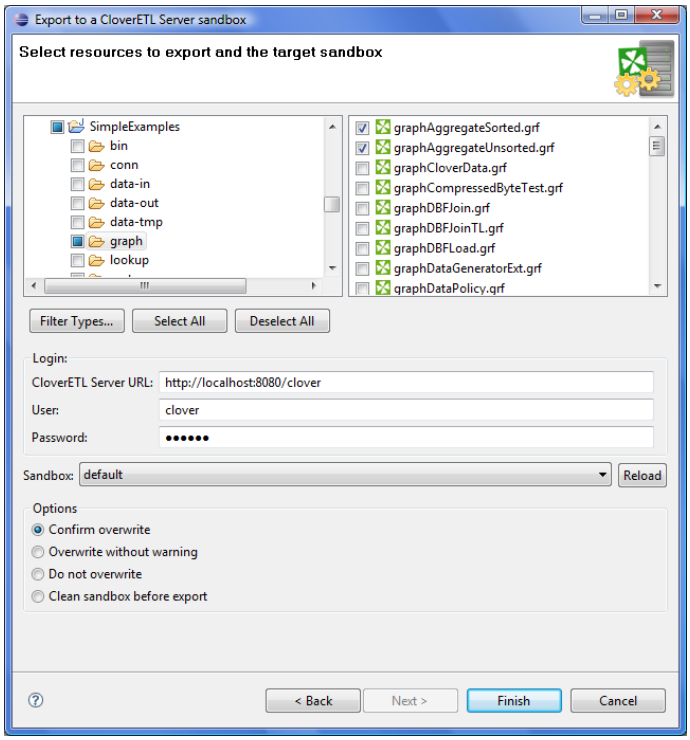


図16.5. CloverETL Serverサンドボックスへのエクスポート

エクスポートするファイルまたはディレクトリ(あるいはその両方)を選択して、同じ名前を持つファイルやディレクトリを警告なしで上書きするか、上書きを確認するか、または同じ名前を持つファイルやディレクトリを上書きしないかを決定し、さらにエクスポート前にサンドボックスを消去するかどうかも決定します。

CloverETL ServerのURL、**ユーザー名**および**パスワード**の3つの項目を指定します。次に、「**Reload**」をクリックします。これで、サンドボックスのリストが「**Sandbox**」メニューに表示されるようになります。

サンドボックスを選択します。次に、「**Finish**」をクリックします。選択したファイルまたはディレクトリ(あるいはその両方)が、選択した**CloverETL Server**サンドボックスにエクスポートされます。

注意

パーティション化されたサンドボックスへのエクスポートはサポートされていません。影響を受けるサンドボックスの場所が不明であるため、エラーが表示されます。

イメージのエクスポート

「**Export image**」項目を選択した場合、「**Next**」ボタンをクリックすると、次のウィンドウが表示されます。

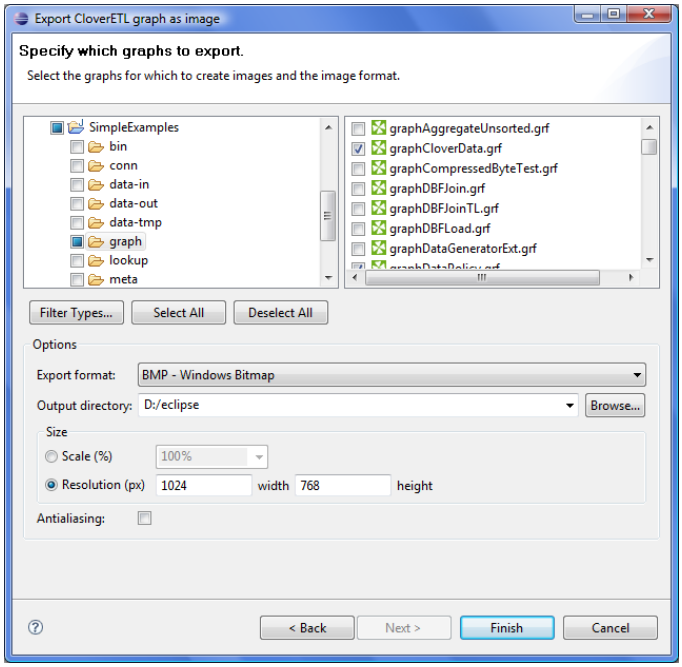


図16.6. イメージのエクスポート

このオプションを使用すると、選択したグラフのイメージのみをエクスポートできます。グラフを選択し、選択したグラフ・イメージをどの出力ディレクトリにエクスポートするかを指定する必要があります。出力ファイルの形式も指定できます(`bmp`、`jpeg`または`png`)。ラジオ・ボタンを切り替えることで、出力イメージのスケールまたはイメージの幅と高さのいずれかを選択します。アンチエイリアスを使用するかどうかも決定できます。

第17章 グラフ・トラッキング

CloverETL Engineによって、実行中のグラフに関する様々なトラッキング情報が提供されます。最も重要な情報を使用して、CloverETLパースペクティブの下部にある**トラッキング・ビュー**に値が移入されます(Designerのタブ(p.43)を参照してください)。

同じデータ・ソースが、グラフ要素のデコレーション表示に使用されます。転送されたレコードの数が、実行中のグラフのエッジに沿って表示されます。フェーズ・エッジには2つの数字が表示されます。エッジの左端にはエッジに送信されたデータ・レコードの数が表示され、エッジの右端にはフェーズ・エッジからすでに読み取られたデータ・レコードの数が表示されます。

図17.1. エッジ・トラッキングの例

CloverETL Cluster環境で、マルチワーカー割当てを使用してグラフが実行されている場合、グラフ内トラッキング情報をより詳細に示すことができます。コンポーネントごとにコンポーネント・インスタンスの数、つまりパラレル実行の数が表示されます。エッジのトラッキング情報は、低、中、高の3つの詳細レベルで表示されます。レベルは、「Window」/「Preference」/「CloverETL」/「Tracking」ページで変更できます。また、グラフ・エディタで、すべてのトラッキング詳細レベルを直接繰り返して処理するには、[D]を押します。

- **低**レベルのトラッキング詳細では、すべてのワーカー/パーティションについて転送されたレコードの合計数が表示されます。
- **中**レベルでは、転送されたレコードの合計数と、追加のドリルダウン情報(渡されたレコードの数と処理パーティションごとの偏り)が表示されます。

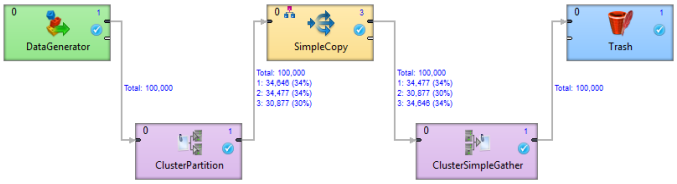


図17.2. 中レベルのトラッキング情報の例

前述の例は、中レベルのトラッキング情報を含むクラスタ化された単純なグラフを示しています。DataGeneratorおよびClusterPartitionコンポーネントは単一のワーカーで実行されているため、相互接続しているエッジは、転送されたレコードの合計数でのみデコレートされています。SimpleCopyコンポーネントは3回実行されているため、ClusterPartitionコンポーネントの出力側にはパーティション化されたエッジが示されています。このエッジの上のラベルには、データ・レコードの30%がSimpleCopyの1つのインスタンスに送信され、34%が他の2つのインスタンスに送信されていることが示されています。

- **高**レベルでは、最も詳細な情報(転送されたレコードの数とパーティションが実行されているクラスタ・ノード名(例: node1: 250 123))が表示されます。エッジがリモートであるパーティション、ソース・クラスタ・ノードおよびターゲット・クラスタ・ノード(例: node1~node2: 250 123)が表示されます。

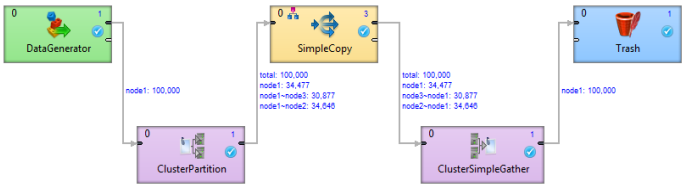


図17.3. 高レベルのトラッキング情報の例

前述の例は、高レベルのトラッキング情報を含むクラスタ化された単純なグラフを示しています。ClusterPartitionコンポーネントは、SimpleCopyコンポーネントの3つの異なるインスタンスにデータを送信します。最初のインスタンスはClusterPartitionコンポーネントと同じワーカーで実行しているため、リモート・エッジは必要ありません(34,477レコードがローカルに転送されています)。2つ目および3つ目のインスタンスはそれぞれ別々のワーカー(さらに別々のクラスタ・ノード)で実行しています。そのため、34,646レコードがnode1からnode2に移動し、30,877レコードがnode3に転送されています。

第18章 高度なトピック

すでに「Run Configurations」ダイアログの使用方法(p.64)で説明したとおり、グラフを実行するために、**「Run Configurations」**ダイアログを使用して、より詳細なオプションを設定できます。

次の場所で設定できます。

- プログラムとVM引数(p.85)
- デフォルトのCloverETL設定の変更(p.88)

処理されたレコードについて表示される数のサイズを変更することもできます。

- 表示される数のフォントの拡大(p.91)

もう1つの高度なオプションとして、Javaの使用があります。次の項で、JREおよびJDKの設定方法を参照できます。

- Javaの設定および構成(p.92)
  - Javaランタイム環境の設定(p.92)
  - Java Development Kitのインストール(p.94)

プログラムとVM引数

グラフの実行時に引数を指定する場合は、コンテキスト・メニューから**「Run Configurations」**を選択し**「Main」**タブでオプションを設定します。

次の3つの**プログラム引数**をタブに入力できます。

- **Program file** (`-cfg <filename>`)

  指定したファイルに、追加のパラメータ値を定義できます。`-P:parameter1=value1 -P:parameter2=value2 ...P:parameterN=valueN`のような一連の式を使用するかわりに(次を参照)、これらのパラメータ値を単一のファイルに定義できます。`parameterK=valueK`のように、行ごとに1つのパラメータを定義する必要があります。次に必要なことは、前述の式でファイル名を指定することのみです。

- **Priorities**

  このタブで指定されたパラメータは、**「Arguments」**タブで指定されたパラメータ、グラフにリンクされているパラメータ(外部パラメータ)またはグラフ自体に指定されているパラメータ(内部パラメータ)よりも高い優先度を持ち、環境変数を上書きすることもできます。ファイル内に任意のパラメータを2回指定した場合は、最後のパラメータのみが適用されることにも注意してください。

- **Tracking [s]** (`-tracking <filename>`)

  グラフ処理ステータスを出力する頻度を設定します。

- **Password** (`-pass`)

  暗号化された接続を復号化するためのパスワードを入力します。リンクされているすべての接続に対して同一である必要があります。

- **Checkconfig** (`-checkconfig`)

  グラフを実行しないでグラフ構成のみをチェックします。

チェック・ボックスを選択して、次の**プログラム引数**を定義することもできます。

- **Logging** (`-loghost`)

log4jのソケット・アペンダのホストおよびポートを定義します。log4jライブラリが必要です。たとえば、localhost:4445などです。

「**Windows**」→「**Preferences**」を選択し、「**CloverETL**」カテゴリで「**Logging**」項目を選択してポートを設定している場合に、ポートを指定できます。

- **Verbose** (-v)

  グラフの実行を冗長モードに切り替えます。

- **Info** (-info)

  Cloverライブラリ・バージョンに関する情報を出力します。

- **Turn off JMX** (-noJMX)

  JMX Beanを介したトラッキング情報の送信をオフにします。これで、パフォーマンスを向上させることができます。

- **Log level** (-loglevel <option>)

  ALL | TRACE | DEBUG | INFO | WARN | ERROR | FATAL | OFFのいずれかを定義します。

  デフォルトの**ログ・レベル**は、**CloverETL Designer**の場合はINFOですが、**CloverETL Engine**の場合はDEBUGです。

- **Skip checkConfig** (-skipcheckconfig)

  グラフを実行する前のグラフ構成のチェックをスキップします。

- **Delete obsolete temp files**

  注意: ジョブフロー(p.684)専用です。

  ジョブフローを実行する前に、Clover Serverにある古いジョブフロー実行からtmpファイルが削除されます。Designerからグラフ/ジョブフローを実行する場合は、常にDEBUGモードが起動されます。このため、一時ファイルがサーバー上に保持されます。

次の2つのチェック・ボックスは**VM引数**を定義します。

- **Server mode** (-server)

  短い起動時間または小さいフットプリントが必要なアプリケーションには、クライアント・システム(デフォルト)が最適です。一般的に、起動時間を最短にすることよりもプログラムの最大実行速度に到達することの方が重要な長時間実行のアプリケーションには、サーバー・モードへの切替えが有益です。サーバー・システムを実行するには、Java Development Kit (JDK)をダウンロードする必要があります。

- **Java memory size, MB** (-Xmx)

  メモリー割当てプール(グラフ実行中に使用可能なメモリー)の最大サイズを指定します。Javaヒープ領域のデフォルト値は68MBです。

これらの引数はすべて、「**Arguments**」タブの「**Program arguments**」ペインまたは「**VM arguments**」に入力するときに、前述のカッコ内の式を使用して指定することもできます。

前述の引数の他に、「**Arguments**」タブに切り替えて、「**Program arguments**」ペインで次のように入力することもできます。

- -P:<parameter>=<value>

  parameterのvalueを指定します。この式では空白を使用しないでください。

優先度

- これらの式よりもさらに多くの式をグラフ実行に使用できます。これらは、グラフにリンクされているパラメータ(外部パラメータ)や、グラフ自体に指定されているパラメータ(内部パラメータ)よりも高い優先度を持ち、環境変数を上書きすることもできます。ただし、「**Main**」タブで指定されている同じパラメータよりも優先度は低くなります。「**Arguments**」タブに任意のパラメータを2回指定した場合は、最後のパラメータのみが適用されることにも注意してください。

- `-config <filename>`

  指定されたファイルからデフォルトの**CloverETL Engine**プロパティをロードします。`defaultProperties`ファイルに含まれる同じプロパティ定義を上書きします。ファイルの名前は任意に選択でき、ファイルは選択されたデフォルト・プロパティのみを再定義します。

- `-logcfg <filename>`

  指定されたファイルから`log4j`プロパティをロードします。指定されていない場合は、`log4j.properties`がクラスパスに存在する必要があります。

  機密データ((S)FTPを介して読取り/書込みを行う場合のユーザー名およびパスワードなど)は、デフォルトではログに出力**されません**。なんらかの理由でこの種類のロギングをオンにする必要がある場合は、`log4j.logger.sensitive`を有効にするカスタムのlog4jファイルを提供する必要があります。

メモリー・サイズの設定例

「**Run Configurations**」ダイアログで、Javaメモリー・サイズ(MB)を設定できます。グラフを実行するにはJava仮想マシンにこのメモリー容量が必要になるため、メモリー・サイズの定義は重要です。適切な値を選択して、JVMの最大メモリー・サイズを定義する必要があります。

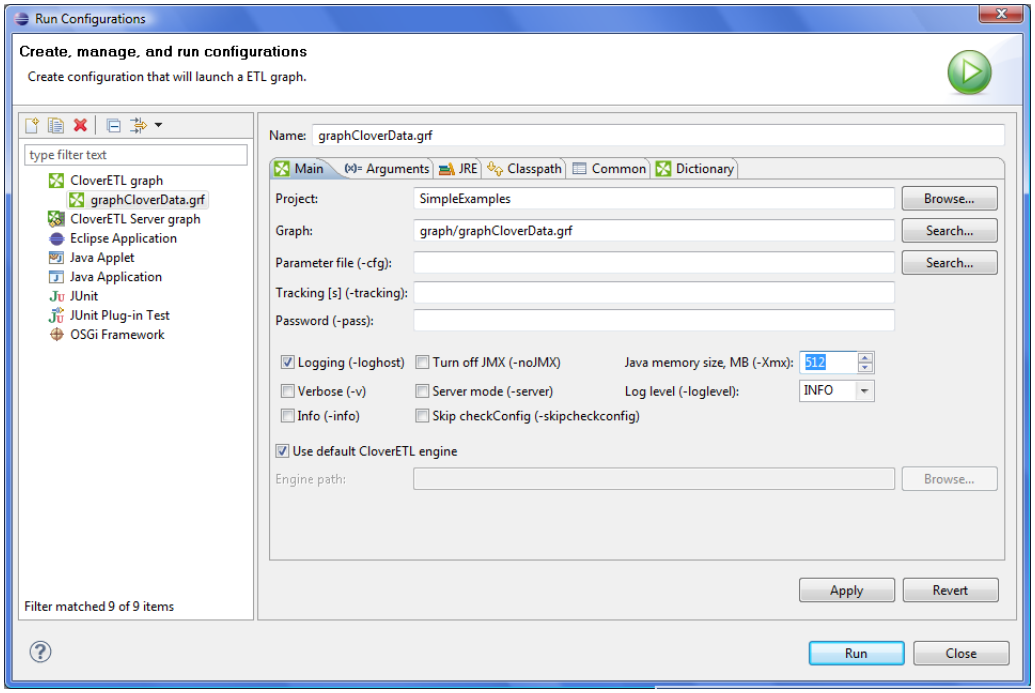


図18.1. メモリー・サイズの設定

デフォルトのCloverETL設定の変更

CloverETLの内部設定(デフォルト)は、**CloverETL** Engine (`plugins/com.cloveretl.gui/lib/lib/cloveretl.engine.jar`)の`org/jetel/data`サブフォルダにある`defaultProperties`ファイルに格納されています。このソース・ファイルには、実行時にロードされ、変換実行時に使用される様々なパラメータが含まれています。

`defaultProperties`ファイル内の値を変更した場合、この変更はすべてのグラフ実行に適用されます。

現在のグラフに対してのみ値を変更するには、オーバーライドが必要なプロパティのみを含むローカル・ファイルを作成します。そのファイルをプロジェクト・ディレクトリに配置します。このローカル・ファイルからプロパティを取得するように**CloverETL**に指示するには、`-config`スイッチを使用します。「**Run Configurations...**」→「**Arguments**」タブに移動して、「**Program arguments**」ペインで次のように入力します。`-config <file_with_overriden_properties>`スイッチを使用します。

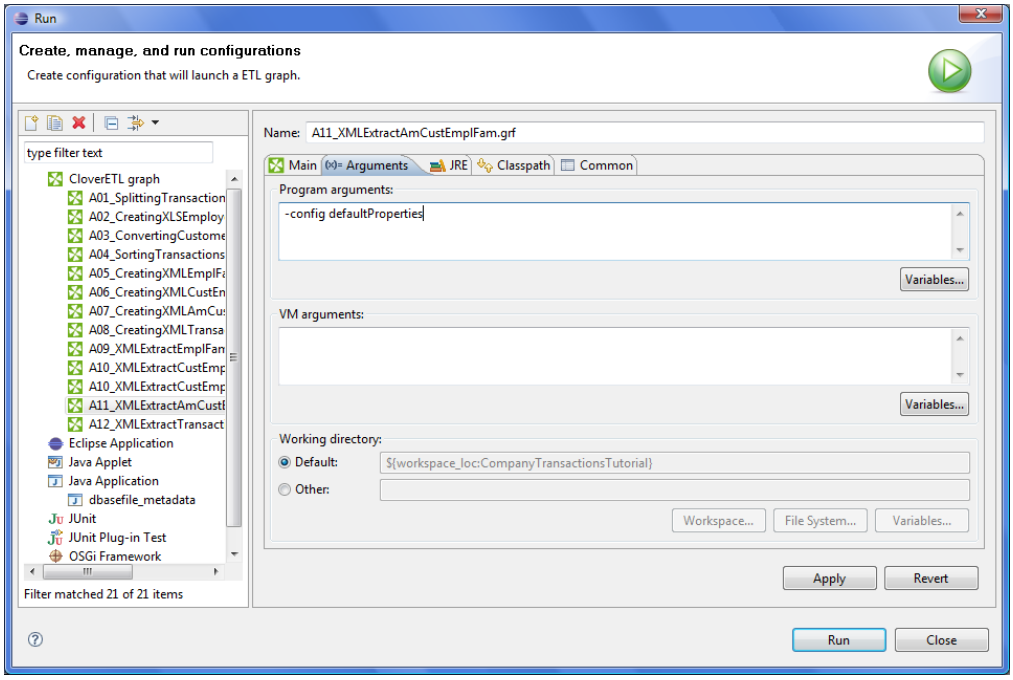


図18.2. カスタムのClover設定

次に、プロパティおよびその値の一部を`defaultProperties`ファイルに示されているとおりに示します。

- `Record.RECORD_LIMIT_SIZE = 32 MB`

  レコードの最大サイズを制限します。理論的には制限は数十MBですが、エラー検出を簡単にするためにできるかぎり小さくしておく必要があります。メモリー要求の詳細は、エッジのメモリー割当て(p.108)を参照してください。

- `Record.FIELD_LIMIT_SIZE = 64 kB`

  レコード内の1フィールドの最大サイズを制限します。メモリー要求の詳細は、エッジのメモリー割当て(p.108)を参照してください。

- `Record.RECORD_INITIAL_SIZE`

  各レコードに割り当てられるメモリーの初期容量を設定します。エッジがどの程度メモリーを必要とするかに応じて、メモリーを最大で`Record.RECORD_LIMIT_SIZE`まで動的に増やすことができます。エッジのメモリー割当て(p.108)を参照してください。

- Record.FIELD_INITIAL_SIZE

  レコード内の各フィールドに割り当てられるメモリーの初期容量を設定します。エッジがどの程度メモリーを必要とするかに応じて、メモリーを最大でRecord.FIELD_LIMIT_SIZEまで動的に増やすことができます。エッジのメモリー割当て(p.108)を参照してください。

- Record.DEFAULT_COMPRESSION_LEVEL=5

  これは、圧縮されるデータ・フィールド(cbyte)の圧縮レベルを設定します。

- DEFAULT_INTERNAL_IO_BUFFER_SIZE = 32768

  コンポーネントがI/O操作に割り当てる初期バッファ・サイズを決定します。この値を増やしても、パフォーマンスにほとんど影響しません。

- USE_DIRECT_MEMORY = true

  Cloverエンジンは、データ・レコード操作にダイレクト・メモリーを集中的に使用します。たとえば、CloverBuffer (シリアル化されたデータ・レコードのコンテナ)の基礎となるメモリーは、ダイレクト・メモリー内のJavaヒープ領域外に割り当てられます。この属性は、パフォーマンス向上のために、デフォルトでtrueに設定されています。ただし、ダイレクト・メモリーはJava仮想マシンでは制御できないため、OutOfMemory例外が発生した場合は、ダイレクト・メモリーの使用をオフにしてみてください。

- DEFAULT_DATE_FORMAT = yyyy-MM-dd
- DEFAULT_TIME_FORMAT = HH:mm:ss
- DEFAULT_DATETIME_FORMAT = yyyy-MM-dd HH:mm:ss
- DEFAULT_REGEXP_TRUE_STRING = true|T|TRUE|YES|Y|t|1|yes|y
- DEFAULT_REGEXP_FALSE_STRING = false|F|FALSE|NO|N|f|0|no|n
- DataParser.DEFAULT_CHARSET_DECODER = ISO-8859-1
- DataFormatter.DEFAULT_CHARSET_ENCODER = ISO-8859-1
- Lookup.LOOKUP_INITIAL_CAPACITY = 512

  サイズを指定しないで参照表を作成した場合の初期容量です。

- DataFieldMetadata.DECIMAL_LENGTH = 12

  小数データ・フィールドのメタデータのデフォルトの最大精度を決定します。精度は数値内の桁の数です。たとえば、数値123.45の精度は5です。

- DataFieldMetadata.DECIMAL_SCALE = 2

  小数データ・フィールドのメタデータのデフォルトの位取りを決定します。位取りは数値の小数点以下の桁数です。たとえば、数値123.45の位取りは2です。

- Record.MAX_RECORD_SIZE = 32 MB

注意

これは非推奨プロパティです。今は、Record.RECORD_LIMIT_SIZEを使用する必要があります。

レコードの最大サイズを制限します。理論的には制限は数十MBですが、エラー検出を簡単にするためにできるかぎり小さくしておく必要があります。

重要

他の多くのプロパティの中には、ロケールを定義できる別のプロパティもあります。ロケールはデフォルト・プロパティとして使用する必要があります。

設定は次のとおりです。

```
# DEFAULT_LOCALE = en.US
```

デフォルトでは、システム・ロケールが**CloverETL**によって使用されます。この行のコメントを解除すると、DEFAULT_LOCALEプロパティを、**CloverETL**でサポートされている任意のロケールに設定できます。すべてのロケールのリスト(p.126)を参照してください。

表示される数のフォントの拡大

必要な場合には、エッジを移動したレコードの数を示すためにエッジに沿って表示される数のフォントを拡大できます。これを行うには、「**Window**」→「**Preferences...**」を選択します。

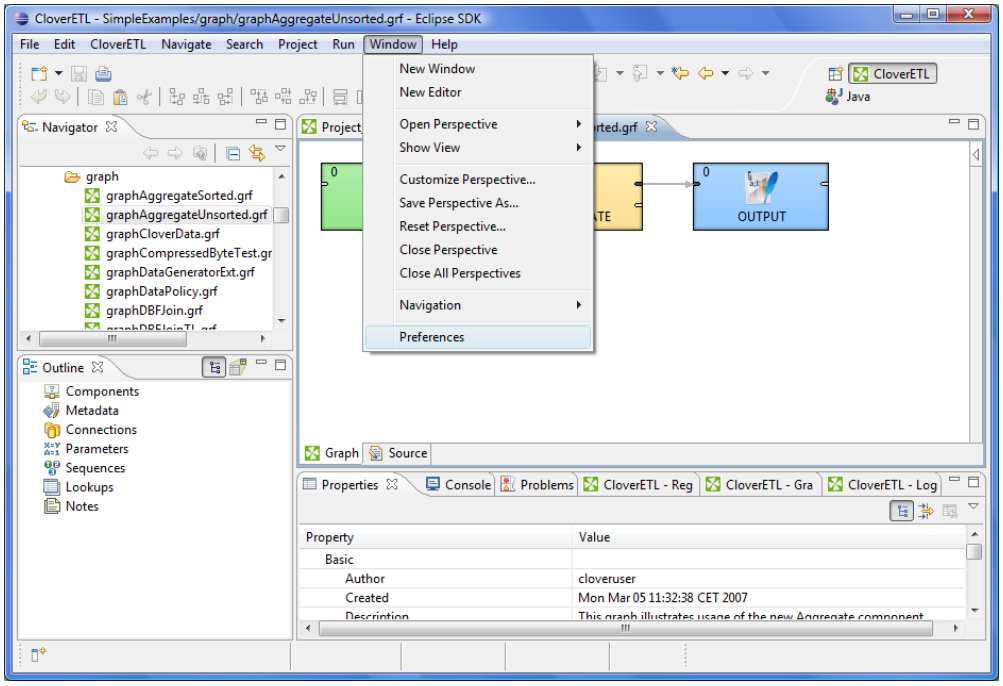


図18.3. 数のフォントの拡大

次に、「**CloverETL**」項目を展開し、「**Tracking**」を選択して、「**Record number font size**」領域で必要なフォント・サイズを選択します。デフォルトでは7に設定されています。

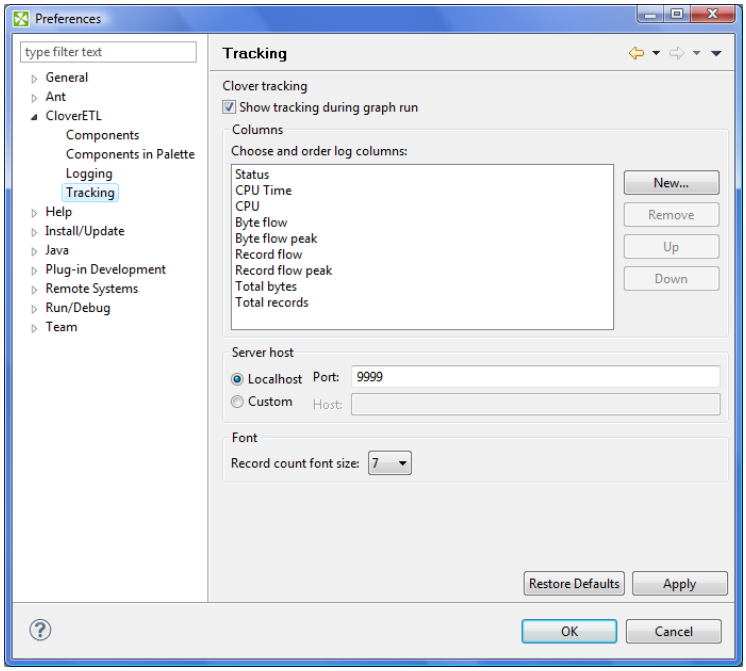


図18.4. フォント・サイズの設定

Javaの設定および構成

JREを設定したりJDKライブラリを追加する場合は、次のように実行できます。

注意

Java 1.6以上を使用してください。

- JREの設定の詳細は、Javaランタイム環境の設定(p.92)を参照してください。
- JDKのインストールの詳細は、Java Development Kitのインストール(p.94)を参照してください。

Javaランタイム環境の設定

JREを設定する場合は、「**Window**」→「**Preferences**」を選択して、これを実行できます。

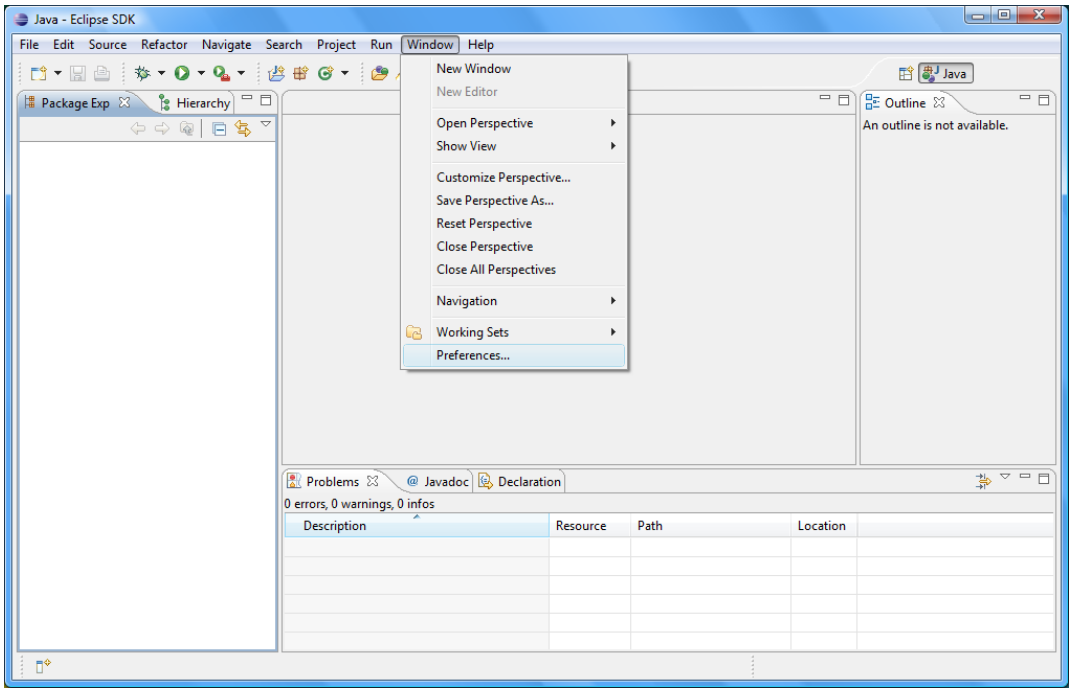


図18.5. Javaランタイム環境の設定

オプションをクリックすると、次のウィンドウが表示されます。

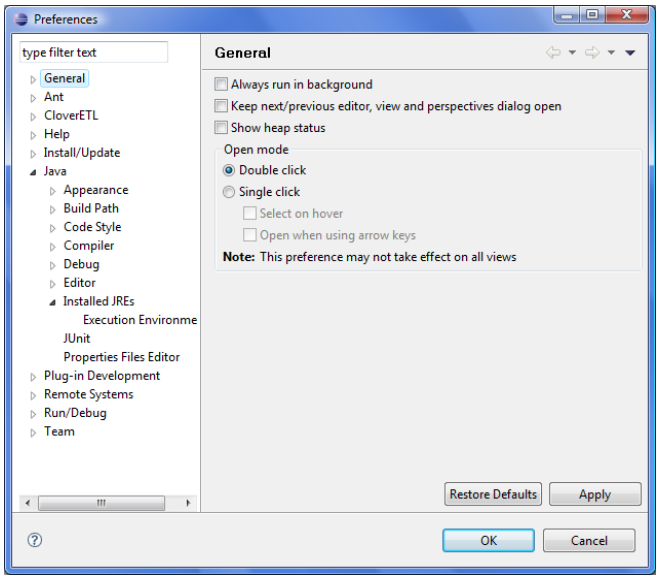


図18.6.「Preferences」ウィザード

ここで、前述のように「**Java**」項目を展開して、「**Installed JREs**」項目を選択する必要があります。すでにJRE 1.6がインストールされている場合は、次のウィンドウが表示されます。

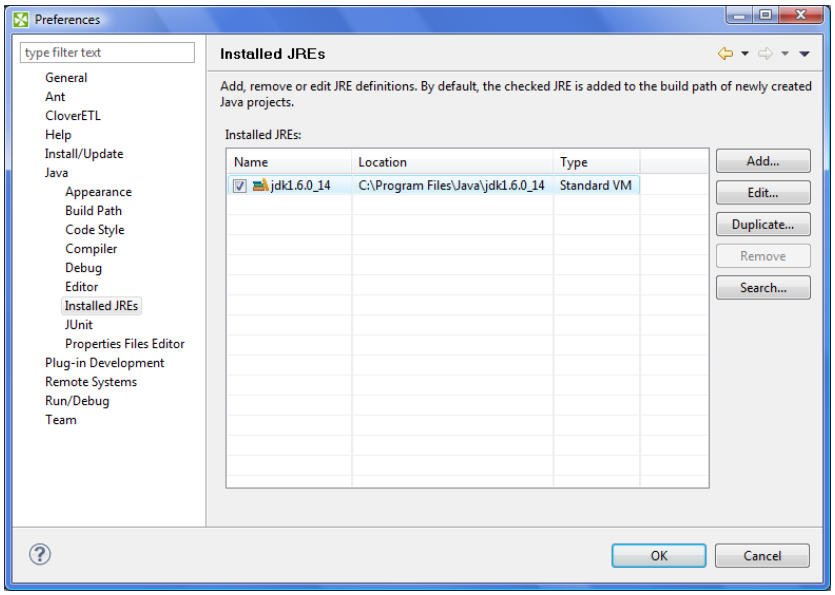


図18.7.「Installed JREs」ウィザード

Java Development Kitのインストール

Javaで変換を記述およびコンパイルする場合は、JDKをインストールしてプロジェクトに追加できます。これを行うには、プロジェクトを選択して右クリックし、コンテキスト・メニューで「**Properties**」項目を選択します。JDK 1.6以上を使用することを再度お薦めします。

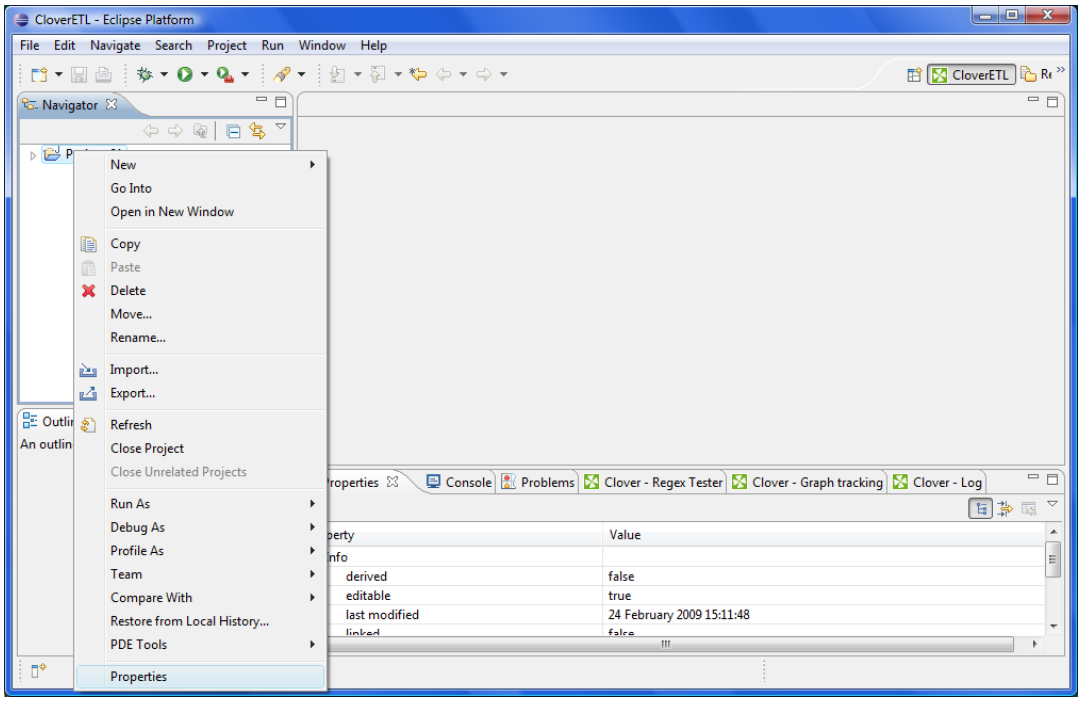


図18.8. Java Development Kitの追加

次に、「**Java Build Path**」項目を選択し、その「**Libraries**」タブを選択します。「**Add External JARs...**」ボタンをクリックします。

重要

Macユーザーの場合: インストール後、直接このメニューにナビゲートします。次に、正しいJavaバージョン(1.6以上)を検索して選択します。インストール後にデフォルトで古いJavaバージョンが選択されるという既知の問題があるため、Macではこの処理が必要です。

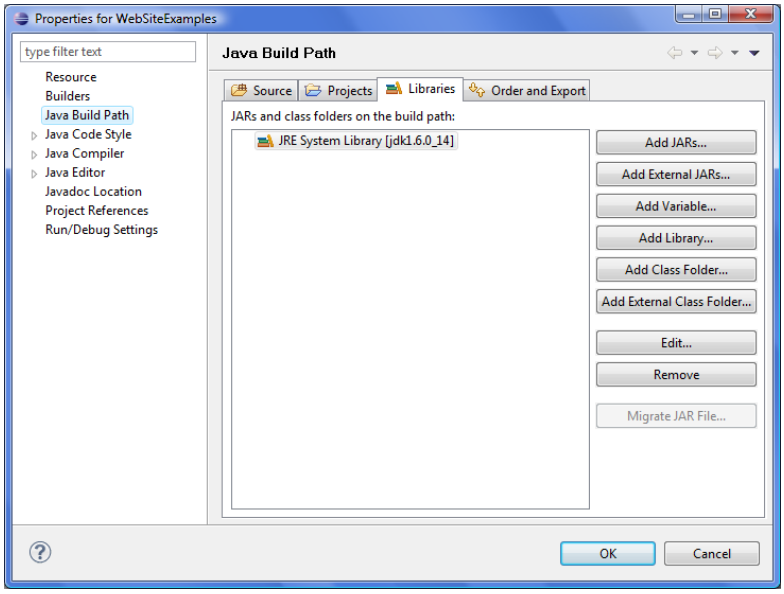


図18.9. JDK Jarの検索

選択したjdkフォルダに含まれるすべての.jarファイルを「**Libraries**」タブに追加できます。

選択を確認した後、次のように.jarファイルがプロジェクトに追加されます。

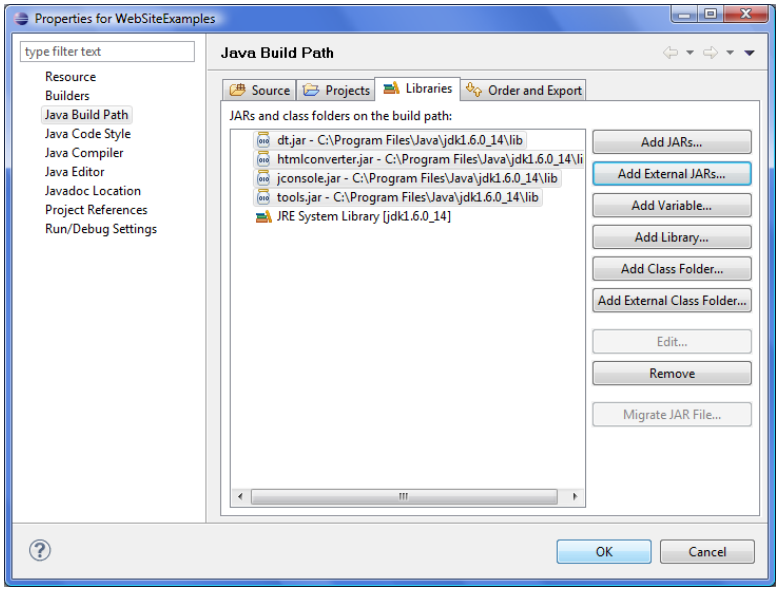


図18.10. JDK Jarの追加

第V部 グラフ要素、構造およびツール

第19章 コンポーネント

最も重要なグラフ要素はコンポーネント(ノード)です。それらはすべて、データ処理を行います。それらの大部分にはポートがあり、これを介してデータの受信または処理済データの送信(あるいはその両方)を行うことができます。大部分のコンポーネントは、これらのポートにエッジが接続されている場合にのみ動作します。ポートに接続されているグラフ内の各エッジには、メタデータが割り当てられている必要があります。メタデータは、1つのコンポーネントから別のコンポーネントへの、エッジを通過するデータ・フローの構造を示します。

コンポーネントはすべて5つのグループに分けることができます。

- リーダー(p.339)

  一般的に、これらのコンポーネントはグラフの初期ノードです。入力ファイル(ローカルまたはリモート)からのデータの読取り、接続されている入力ポートからのデータの受信、辞書からのデータの読取り、またはデータの生成を行います。このようなノードは、**リーダー**と呼ばれます。

- ライター(p.459)

  他方のコンポーネントは、グラフの終了ノードです。入力ポートを介したデータの受信とファイルへの書込み(ローカルまたはリモート)、接続されている出力ポートを介したデータの送信、電子メールの送信、辞書へのデータの書込み、または受信済データの破棄を行います。このようなノードは、**ライター**と呼ばれます。

- トランスフォーマ(p.576)

  これらのコンポーネントは、グラフの中間ノードです。データの受信と全出力ポートへのコピー、データの重複解除、フィルタ処理またはソート処理、多数のポートを介した受信済データの連結、収集またはマージと単一の出力ポートを介した送信、接続されている多数の出力ポートへのレコードの配布、2つの入力ポートを介して受信したデータの結合、新しい情報を得るためのデータの集計、またはより複雑な方法でのデータの変換を行います。このようなノードは、**トランスフォーマ**と呼ばれます。

- ジョイナ(p.653)

  ジョイナもまた、グラフの中間ノードです。2つ以上のソースからのデータの受信、特定のキーに応じたデータの結合、および出力ポートを介した結合済データの送信を行います。

- ジョブ制御(p.684)

  ジョブ制御は、様々なジョブ・タイプの実行および監視に重点を置くコンポーネントのグループです。これらのコンポーネントでは、ETLグラフ、ジョブフローおよび解析済スクリプトを実行できます。グラフおよびジョブフローは、監視および中断(オプション)できます。

ヒント

注意: このコンポーネント・カテゴリが表示されない場合は、「**Window**」→「**Preferences**」→「**CloverETL**」→「**Components in Palette**」にナビゲートし、「**Job Control**」の横にあるチェック・ボックスを両方とも選択します。

- ファイル操作(p.734)

  ファイル操作は、ファイル・システム上のファイルの操作((FTPを介して)ローカルまたはリモート)に適したコンポーネントです。これらは、Clover Serverサンドボックス内のファイルにもアクセスできます。

ヒント

注意: このコンポーネント・カテゴリが表示されない場合は、「**Window**」→「**Preferences**」→「**CloverETL**」→「**Components in Palette**」にナビゲートし、「**File Operations**」の横にあるチェック・ボックスを両方とも選択します。

- クラスタ・コンポーネント(p.750)

  2つの**クラスタ・コンポーネント**は、**CloverETL Server**インスタンスのクラスタの様々なノードにデータ・レコードを配布するか、またはこれらのレコードをまとめます。

  このようなグラフは、クラスタ内で並行して実行されます。

- データ品質(p.763)

  データ品質は、データ品質に関連する様々なタスクを実行するコンポーネントのグループで、データに関する情報の決定、問題の検出や修正などを行います。

- その他(p.778)

  その他グループは、コンポーネントの異種グループです。様々なタスク(システム、JavaまたはDBコマンドの実行、**CloverETL**グラフの実行またはサーバーへのHTTPリクエストの送信)を実行できます。このグループのその他のコンポーネントは、参照表からの読取りまたは参照表への書込み、データのキーのチェックと他のキーでの置換、シーケンスのソート順のチェック、またはコンポーネントを通過するデータ・フローの処理の減速を行うことができます。

- 一部のプロパティはすべてのコンポーネントに共通です。

  すべてのコンポーネントの共通プロパティ(p.266)

- 一部のプロパティはこれらの多くに共通です。

  多くのコンポーネントの共通プロパティ(p.275)

- その他のプロパティはそれぞれのグループに共通です。
  - リーダーの共通プロパティ(p.296)
  - ライターの共通プロパティ(p.309)
  - トランスフォーマの共通プロパティ(p.320)
  - ジョイナの共通プロパティ(p.323)
  - クラスタ・コンポーネントの共通プロパティ(p.330)
  - 「その他」の共通プロパティ(p.331)
  - データ品質の共通プロパティ(p.332)

これらの共通プロパティの詳細は、第VII部「コンポーネントの概要」(p.260)を参照してください。

個々のコンポーネントの詳細は、第VIII部「コンポーネント・リファレンス」(p.338)を参照してください。

第20章 エッジ

この章では、エッジの概要について説明します。エッジの内容、グラフのコンポーネントへのエッジの接続方法、エッジへのメタデータの割当て方法およびエッジを介した伝播方法、エッジのデバッグ方法、およびエッジを通過するデータ・フローの表示方法について説明します。

エッジとは

エッジは、1つのコンポーネントから別のコンポーネントへのデータ・フローを表します。

エッジのプロパティを次に示します。

- エッジによるコンポーネントの接続(p.99)

  各エッジは、2つのコンポーネントを接続している必要があります。

- エッジのタイプ(p.100)

  各エッジは、4つのタイプのいずれかです。

- エッジへのメタデータの割当て(p.101)

  各エッジにメタデータを割り当てて、エッジを通過するデータ・フローを示す必要があります。

- エッジを介したメタデータの伝播(p.102)

  メタデータは、一部のコンポーネントを介して入力ポートから出力ポートに伝播できます。

- エッジの色(p.102)

  各エッジは、メタデータの割当て、エッジの選択などに応じて色が変わります。

- エッジのデバッグ(p.103)

  各エッジをデバッグできます。

- エッジのメモリー割当て(p.108)

  一部のエッジは他のエッジよりも多くのメモリーを必要とします。この項にはその説明が含まれています。

エッジによるコンポーネントの接続

2つ以上のコンポーネントを選択して**グラフ・エディタ**に貼り付けた場合、**パレット**・ツールから取得したエッジでそれらを接続する必要があります。データはこのエッジを、1つのコンポーネントからもう片方のコンポーネントに流れます。そのため、エッジを流れるデータ・レコードの構造を示すメタデータが、各エッジに割り当てられている必要があります。

2つのコンポーネント間にエッジを作成する方法は2つあります。**パレット**・ツールで**エッジ**・ラベルをクリックしてから、ソース・コンポーネント(エッジを開始するコンポーネント)にカーソルを置き、左クリックしてエッジの作成を開始できます。次に、ターゲット・コンポーネント(エッジを終了するコンポーネント)にカーソルを置き、再度クリックします。これで、エッジが作成されます。2つ目の方法では、ツール選択をショートカットします。任意のコンポーネントの出力ポートにマウスを置きます。現在、**選択**ツールが選択されている場合には、Cloverによって自動的に**エッジ**・ツールに切り替わります。ここでクリックして、エッジ作成プロセスを開始できます。このプロセスは前述のように動作します。

入力ポートからのデータの受信とデータ・ソースへの書込みのみを行うコンポーネント(**Trash**などの**ライター**)もあれば、データ・ソースからのデータの読取りまたはデータの生成と、出力ポートを介した送信を行うコンポーネント(**DataGenerator**などの**リーダー**)や、データの受信と他のコンポーネントへの送信の両方を行うコンポーネント(**トランスフォーマ**および**ジョイナ**)があります。また、最後のグループのコンポーネントは、エッジに接続されている必要

があるコンポーネント(**CheckForeignKey**、**LookupTableReaderWriter**、**SequenceChecker**、**SpeedLimiter**などの非実行コンポーネント)か、または接続できるコンポーネント(**実行コンポーネント**)のいずれかになります。

前述のように、エッジをグラフに貼り付けている場合、そのエッジは常にコンポーネントのポートに結合されています。一部のコンポーネントではポートの数が厳密に指定されていますが、その他のコンポーネントではポートの数は無制限です。ポートの数が無制限の場合、新しいエッジを接続することで新しいポートが作成されます。エッジの作業が終了したら、パレット・ツールの「**Select**」項目をクリックするか、キーボードの**[Esc]**を押します。

すでにエッジで2つのコンポーネントを接続してある場合、このエッジを別のコンポーネントに移動できます。これを行うには、クリックしてエッジを強調表示してから、マウス・カーソルが矢印から十字に変わるまで、エッジの接続先のポート(入力または出力)に移動します。十字が表示されたら、任意のコンポーネントの他の空いているポートにエッジをドラッグできます。選択ツールでポートにマウスを置くと、自動的にエッジが選択されるため、単にクリックしてドラッグできます。出力ポートは別の出力ポートでのみ、入力ポートは別の入力ポートでのみ置換できます。

エッジの自動ルーティングまたは手動ルーティング

2つのコンポーネントがエッジで接続されている場合、エッジが他のコンポーネントやノートなどの他の要素と重なることがあります。このような場合、エッジのデフォルトの自動ルーティングから手動ルーティングに切り替えることができます。手動ルーティング・モードでは、エッジが表示される場所を制御できます。これを行うには、エッジを右クリックし、コンテキスト・メニューから「**Edge Autorouting**」の選択を解除します。

これで、エッジの各直線部分の中央にポイントが表示されます。

このポイントにカーソルを置くと、カーソルが水平または垂直のサイズ変更カーソルに置き換わり、該当するエッジ部分を水平または垂直方向にドラッグできるようになります。

このようにして、問題のある領域からエッジを離すことができます。

エッジを選択してから**[Ctrl] + [R]**を押して、エッジの自動ルーティングと手動モードを切り替えることもできます。

エッジのタイプ

エッジには4つのタイプがあり、そのうちの3つには内部バッファがあります。エッジを右クリックしてから「**Select edge**」項目をクリックし、表示されたタイプのいずれかをクリックすることで、エッジを選択できます。

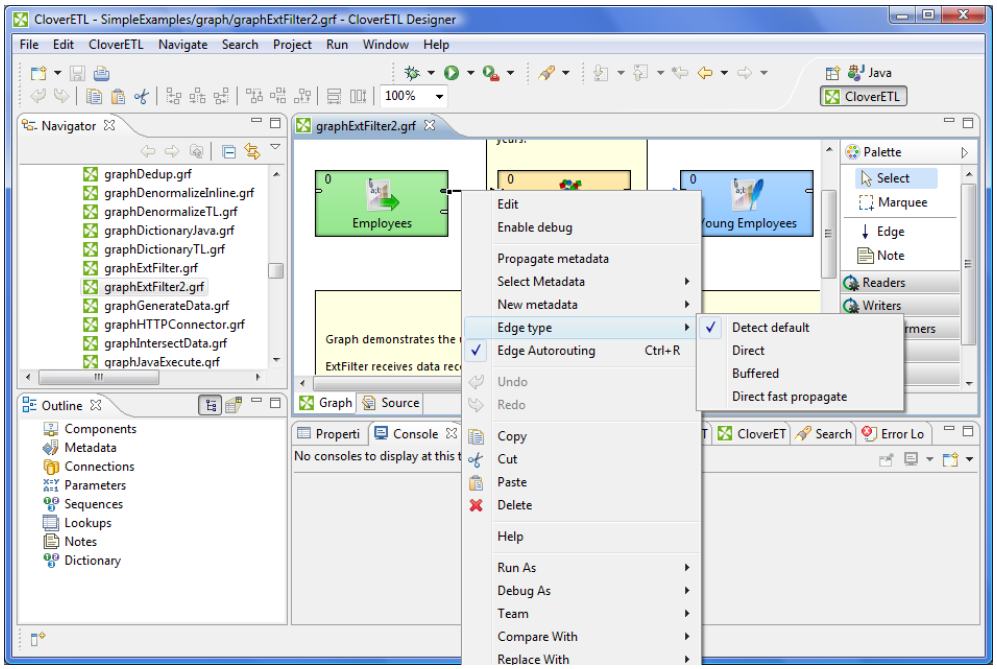


図20.1. エッジ・タイプの選択

エッジは次のタイプのいずれかに設定できます。

- **ダイレクト・エッジ**: このエッジ・タイプはメモリー内にバッファを持ち、バッファはデータ・フローの高速化に役立ちます。これは、ETLグラフのデフォルトのエッジ・タイプです。
- **バッファ付きエッジ**: このエッジ・タイプもメモリー内にバッファを持ちますが、必要に応じてディスクにもデータを格納できます。そのため、バッファ・サイズの制限はありません。2つのバッファを持ち、1つは読取り用でもう1つは書込み用です。
- **ダイレクト高速伝播エッジ**: これは、**ダイレクト・エッジ**の代替となる実装です。このエッジ・タイプはバッファを持ちませんが、それでもなお高速なデータ・フローが提供されます。各データ・レコードを受信するとすぐにこのエッジのターゲットに送信します。これは、ジョブフローのデフォルトのエッジ・タイプです。
- **フェーズ接続エッジ**: このエッジ・タイプは選択できません。これは、異なるフェーズ番号を持つ2つのコンポーネント間で自動的に作成されます。

エッジ・タイプを明確に指定しない場合は、「**Detect default**」オプションを選択すると、Cloverで決定するようにできます。

エッジへのメタデータの割当て

メタデータは、データを記述する構造体です。最初、各エッジは点線として表示されます。メタデータを作成してエッジに割り当てた後でのみ、実線になります。

次の該当する各項で示すようにメタデータを作成できますが、空の(点線の)エッジをダブルクリックしてメニューから「**Create metadata**」を選択したり、「**Link shared metadata**」を選択して既存の外部メタデータ・ファイルをリンクすることもできます。

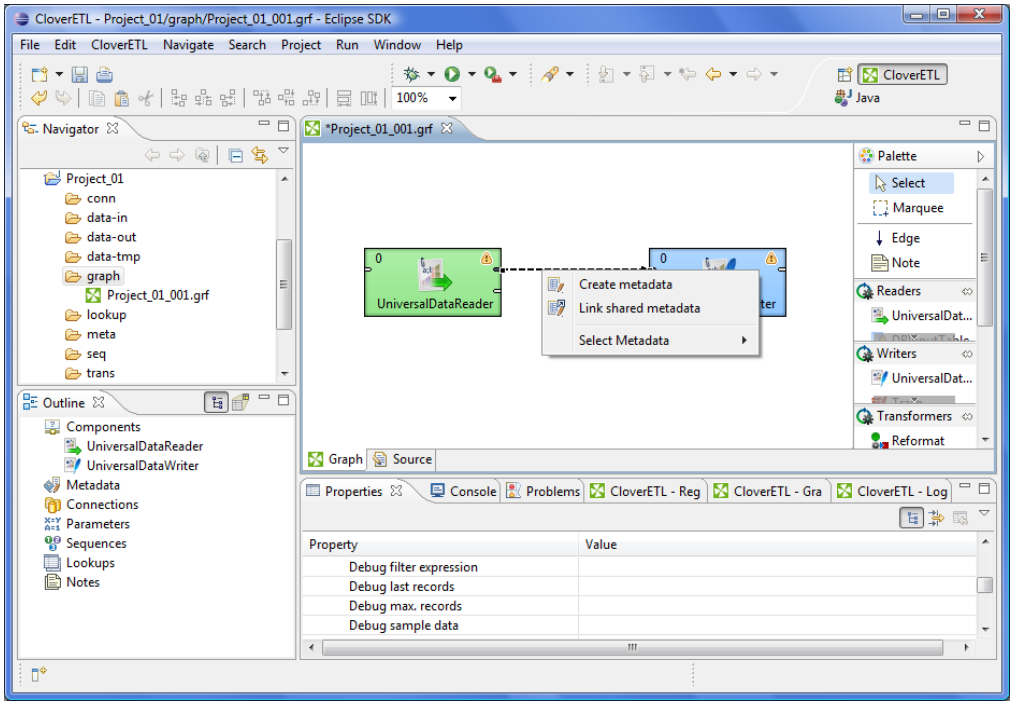


図20.2. 空のエッジでのメタデータの作成

エッジを右クリックしてコンテキスト・メニューから「**Select metadata**」項目を選択し、リストから目的のメタデータを選択しても、エッジにメタデータを割り当てることができます。また、メタデータのエントリを「**Outline**」からエッジにドラッグしても実行できます。

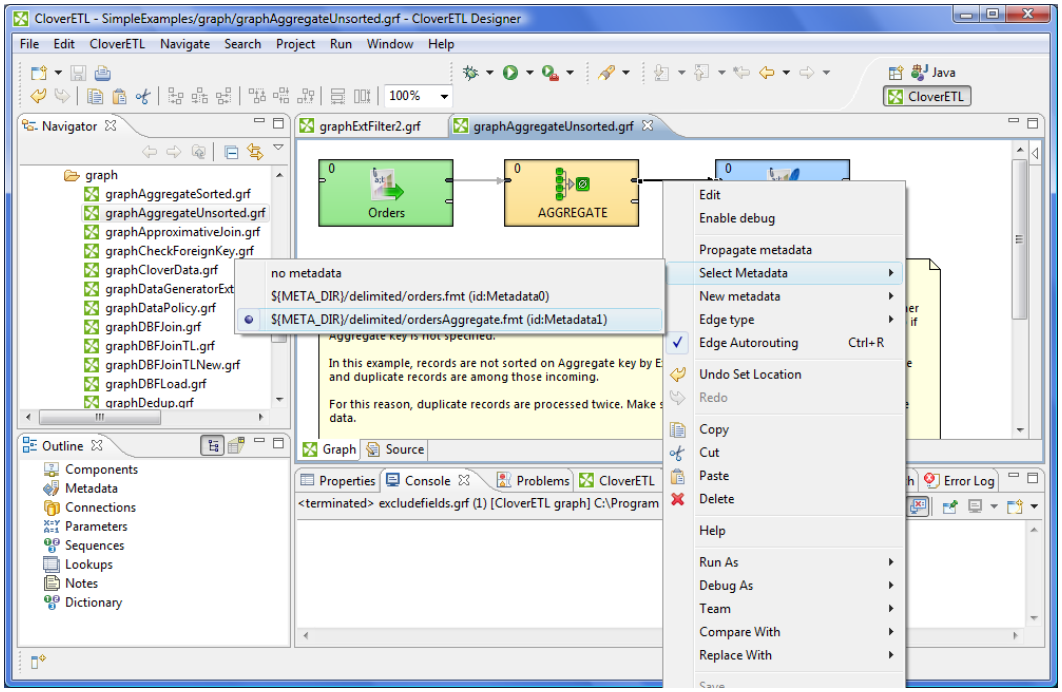


図20.3. エッジへのメタデータの割当て

エッジの作成時に自動的にエッジに適用されるメタデータを選択することもできます。これを選択するには、**パレット**でエッジ・ツールを右クリックしてから必要なメタデータを選択します。選択を削除する場合は**何も選択しません**。

エッジを介したメタデータの伝播

すでにエッジにメタデータを割り当てている場合は、コンポーネントを介して、割当て済のメタデータを他のエッジに伝播する必要があります。

メタデータを伝播するには、エッジを右クリックしてコンテキスト・メニューを開いてから「**Propagate metadata**」項目を選択する必要があります。メタデータは、メタデータを変更できるコンポーネント(**Reformat**、**ジョイナ**など)に達するまで伝播されます。

その他のエッジについては、別のメタデータを定義し、必要に応じて再度伝播する必要があります。

エッジの色

- エッジで2つのコンポーネントを接続すると、そのエッジはグレーの点線で表示されます。
- エッジにメタデータを割り当てると実線になりますが、色はグレーのままです。
- 「**Outline**」ペインでメタデータ項目をクリックすると、選択したメタデータを持つすべてのエッジが青色になります。
- **グラフ・エディタ**でエッジをクリックした場合、選択されたエッジは黒色になり、同じメタデータを持つその他のエッジはすべて青色になります。(さらに、この場合、エッジのツールチップにメタデータが表示されます。)

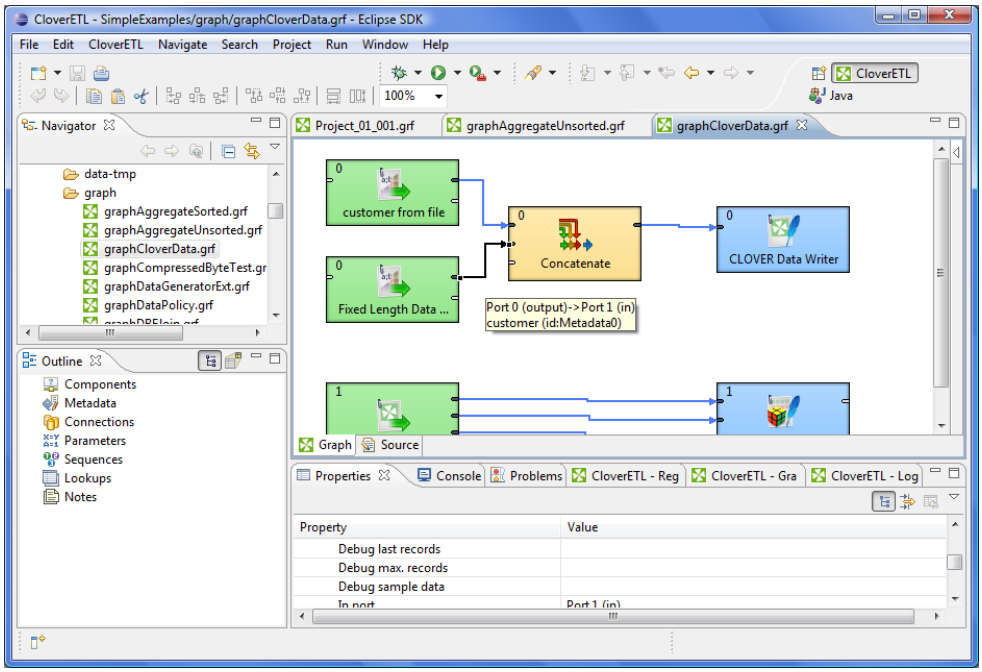


図20.4. ツールチップ内のメタデータ

エッジのデバッグ

グラフの実行中に正しくないまたは予期しない結果が発生し、発生したエラーとその発生場所の確認が必要な場合は、グラフをデバッグできます。問題の原因となっている場所を推測し、場合によっては、どのレコードをデバッグ・ファイルに保存するかを指定する必要もあります。少量のレコードを処理している場合は、デバッグ・ファイルに保存するレコードの数を制限する必要はありませんが、大量のレコードを処理している場合は、このオプションが役立つことがあります。

エッジをデバッグするには、次のように行います。

1. デバッグを有効にします。デバッグの有効化(p.103)を参照してください。
2. デバッグ・データを選択します。デバッグ・データの選択(p.104)を参照してください。
3. デバッグ・データを表示します。デバッグ・データの表示(p.106)を参照してください。
4. デバッグを終了します。デバッグの終了(p.108)を参照してください。

デバッグの有効化

- グラフをデバッグするには、疑いのあるエッジを右クリックし、コンテキスト・メニューから「**Enable debug**」オプションを選択します。これで、バグ・アイコンがエッジの上に表示され、グラフ実行時にデバッグが実行されることが示されます。
- エッジをクリックして**タブ**・ペインの「**Properties**」タブに切り替えても、同じことを実行できます。そこで必要なことは、「**Debug mode**」属性を`true`に設定することのみです。これは、デフォルトでは`false`に設定されています。再び、バグ・アイコンがエッジの上に表示されます。

グラフを実行すると、デバッグ・エッジごとに1つのデバッグ・ファイルが作成されます。この後に必要なことは、これらのデバッグ・ファイル(`.dbg`拡張子)のデータ・レコードを表示して調査することのみです。

デバッグ・データの選択

デバッグするエッジの選択以外に何も行わなかった場合、そのエッジを通過するすべてのデータ・レコードがデバッグ・ファイルに保存されます。

ただし、前述のとおり、デバッグ・ファイルに保存されるデータ・レコードを制限できます。

これは、デバッグ・エッジの「**Properties**」タブで実行するか、デバッグ・エッジを右クリックした後にコンテキスト・メニューから「**Debug properties**」を選択して実行できます。

次の4つのエッジ属性を「**Properties**」タブまたは「**Debug properties**」ウィザードで設定できます。

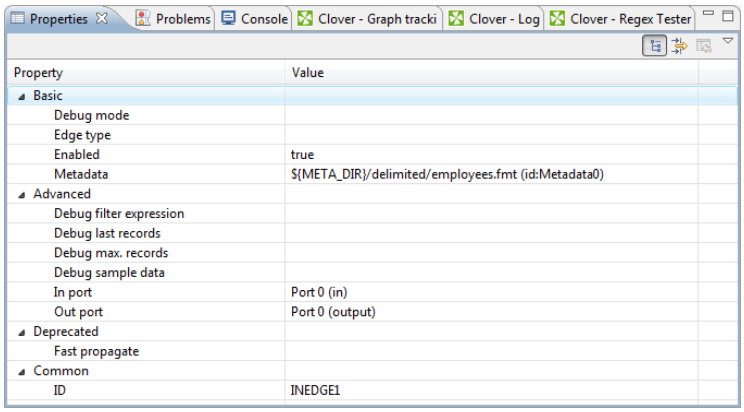


図20.5. エッジのプロパティ

- **デバッグ・フィルタ式**

  エッジに対してフィルタ式を指定すると、指定されたフィルタ式を満たすデータ・レコードがデバッグ・ファイルに保存されます。式を満たしていないその他のデータ・レコードは無視されます。

  フィルタ式が定義されている場合は、式を満たすすべてのレコードが保存されるか(「**Debug sample data**」が`false`に設定されている)、またはそれらのサンプルのみが保存される(「**Debug sample data**」が`true`に設定されている)ことにも注意してください。

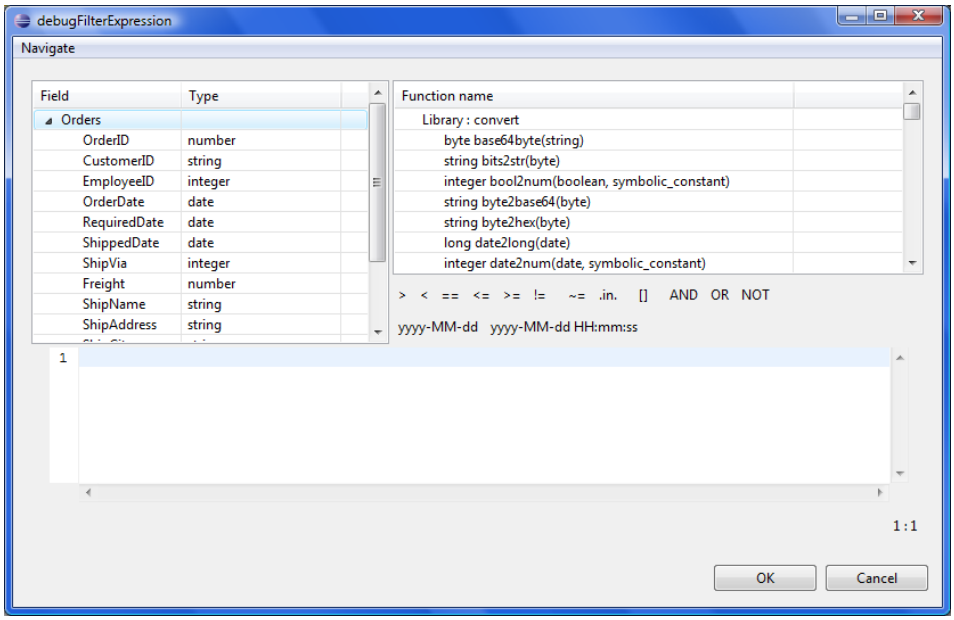


図20.6. フィルタ・エディタ・ウィザード

このウィザードは3つのペインで構成されています。左側のペインにはレコード・フィールド(その名前およびデータ型)のリストが表示されます。ダブルクリックするか、またはドラッグ・アンド・ドロップしてこれらを選択できます。これで、フィールド名が下の領域に表示されます。このとき、フィールド名の前にはドル記号とポート番号が表示されます。(`$0.employee`など。)ウィンドウの右側のペインから選択した関数を使用することもできます。このペインの下には、比較記号と論理接続の両方があります。ダブルクリックすることで、名前、関数、記号および接続を選択できます。これで、これらが下の領域に表示されます。この領域でこれらを使用して、フィルタ式の作成を完了できます。式を検証し、**「Cancel」**をクリックして作成を終了するか、または**「OK」**をクリックして式を確定できます。

重要

フィルタ・エディタではCTL1かCTL2のいずれかを使用できます。

次の2つのオプションは同等です。

1. **CTL1の場合**

```
is_integer($0.field1)
```

2. **CTL2の場合**

```
//#CTL2
isInteger($0.field1)
```

- **Debug last records**

  Debug last recordsプロパティを`false`に設定すると、最初の部分のデータ・レコードがデバッグ・ファイルに保存されます。デフォルトでは、最後の部分のレコードがデバッグ・ファイルに保存されます。**Debug last records**のデフォルト値は、`true`です。

  「Debug last records」属性を`false`に設定すると、データ・レコードは、最後の部分よりも最初の部分から、より高い頻度で選択されます。また、**「Debug last records」**属性を`true`に設定した場合または変更しなかった場合は、最初の部分からよりも最後の部分から、より高い頻度で選択されるようになります。

- **Debug max. records**

  デバッグ・ファイルに保存する最大データ・レコード数の制限を定義することもできます。これらのデータ・レコードは、最初の部分から**(Debug last records**が`false`に設定されている)または最後の部分から**(Debug last records**がデフォルト値になっているか、明示的に`true`に設定されている)取得されます。

- **Debug sample data**

  「Debug sample data」属性を`true`に設定した場合、**「Debug max. records」**属性の値は、デバッグ・ファイルに保存可能なデータ・レコード数を制限する、単なるしきい値になります。データ・レコードはランダムに保存され、それらの一部は除外され、残りがデバッグ・ファイルに保存されます。この場合、デバッグ・ファイルに保存されるデータ・レコードの数は、この制限以下になります。

  「Debug sample data」の値を設定していない場合、または明示的に`false`に設定している場合、デバッグ・ファイルに保存されるレコードの数は、**「Debug max. records」**属性の値と等しくなります(**「Debug max. records」**よりも多くのレコードがデバッグ・エッジを通過している場合)。

これらのプロパティは、コンテキスト・メニューで**「Debug properties」**オプションを選択しても定義できます。これで、次のウィザードが開きます。

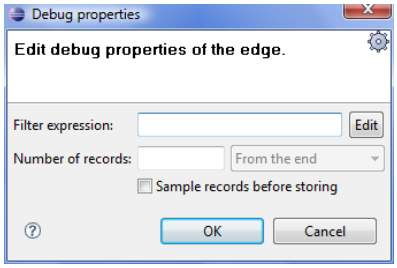


図20.7.「Debug Properties」ウィザード

デバッグ・データの表示

エッジを通過し、フィルタ式を満たして保存されたレコードを表示するには、右クリックしてコンテキスト・メニューを開く必要があります。次に「**View data**」項目をクリックする必要があります。これで、「**View data**」ダイアログが開きます。ここで前述と同じ方法でフィルタ式を作成できることに注意してください。

表示するレコードの数を選択し、「**OK**」をクリックして確定します。

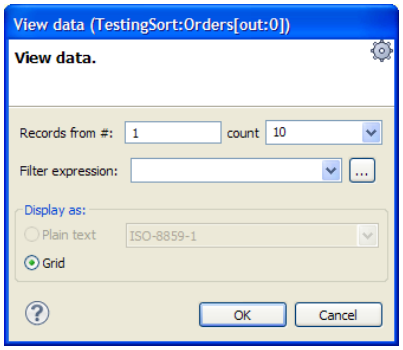


図20.8.「View Data」ダイアログ

選択したカウントは**CloverETL Designer**によって記憶され、「**View data**」ダイアログを再度開いたときに、同じカウントが表示されます。

レコードは別の「**View data**」ダイアログに表示されます。このダイアログはグリッド・モードになっています。単に列のヘッダーをクリックするのみで、その列のレコードを昇順または降順でソートできます。より多くのエッジのデータを同時に表示できます。ダイアログ・ウィンドウを区別するために、表示されているエッジの情報が、GRAPH.name:COMPONENT.name[out: PORT.id]の形式でタイトルに示されます。

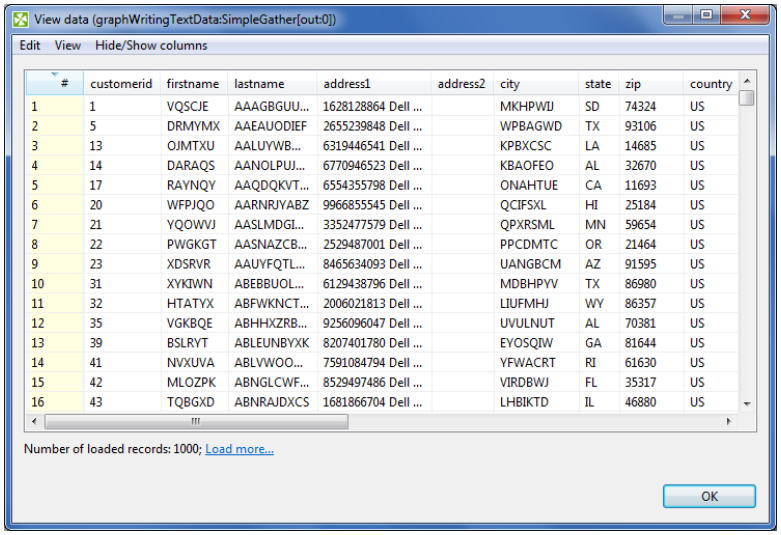


図20.9. デバッグ・エッジのデータの表示

注意

レコードが多きすぎる場合は、表示できないデータを示す[...]マークが表示されます。

レコードが多すぎる場合は、グリッドの下に「**Load more...**」という青色のテキストが表示されます。これをクリックすると、現在表示されているレコードの後に、新しいレコードのチャンクが追加されます。これは、グラフがまだ実行している間にレコードを確認する場合に特に役立ちます。レコードは、グラフの変換によって生成されているときに、クリックでロードされます。

グリッドの上には、「**Edit**」、「**View**」、「**Hide/Show columns**」の3つのラベルがあります。

「**Hide/Show columns**」ラベルをクリックすると、表示する列(すべて、なしまたは選択された列のみ)を選択できます。クリックすることで任意のオプションを選択できます。

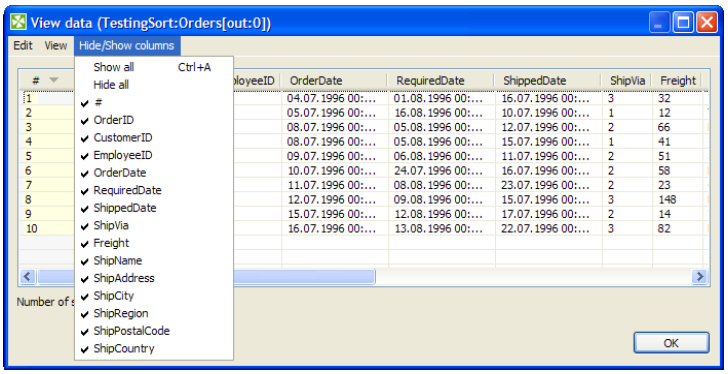


図20.10. データ表示時の「Hide/Show Columns」

「**View**」ラベルをクリックすると、2つのオプションが表示されます。印刷できない文字を表示するかどうかを決定できます。また、1つのレコードのみを別に表示するかどうかも決定できます。このようなレコードは「**View record**」ダイアログに表示されます。このダイアログの下部に複数の矢印ボタンが表示されます。これらを使用して、レコードを参照したり、順番に表示することができます。最も右側にあるボタンをクリックすると、表示されているレコードの最後のレコードが表示されますが、必ずしも最後に処理されたレコードが表示されるわけではありません。

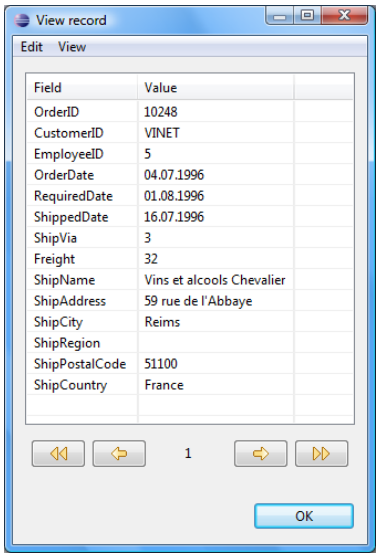


図20.11. 「View Record」ダイアログ

「**Edit**」ラベルをクリックすると、4つのオプションが表示されます。

- 表示するレコードまたは行の番号を選択できます。番号を入力して「OK」をクリックすると、そのレコードが強調表示されます。

- もう1つのオプションでは「**Find**」ダイアログが開きます。まず最初に、このウィザードには式を入力できるテキスト領域が含まれています。続いて、「**Match case**」チェック・ボックスを選択した場合、検索で大文字小文字が区別されるようになります。「**Entire cells**」チェック・ボックスを選択した場合、式を完全に満たすセルのみが強調表示されます。「**Regular expression**」チェック・ボックスを選択した場合、テキスト領域に入力した式が正規表現 (p.963)として使用されます。行または列のいずれの方向で式を検索するかも決定できます。また、検索する列(すべて、表示されている列のみ、リストの1つの列)を選択することもできます。さらに、最後のオプションとして、条件を満たすすべてのセルを検索するか、または1つのセルのみを検索するかも選択できます。

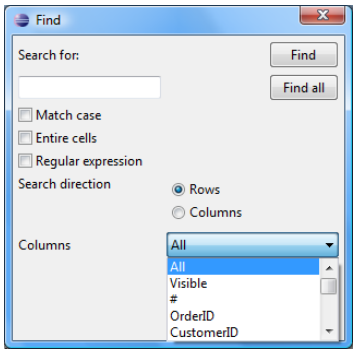


図20.12. 「Find」ダイアログ

- 最後のオプションとして、一部のレコードまたは1つのレコードの一部をコピーできます。レコード全体をコピーするか(文字列にコピー、またはレコードとしてコピー(後者の場合、デリミタも選択可能))、または一部のレコード・フィールドのみをコピーするかを選択する必要があります。選択されたオプションが有効になり、もう片方のオプションは無効になります。「**OK**」ボタンをクリックした後に必要なことは、コピー先の場所を選択し、そこに貼り付けることのみです。

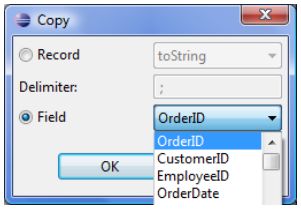


図20.13. 「Copy」ダイアログ

デバッグの終了

デバッグを終了する場合は、***グラフ・エディタ***で、コンポーネントやエッジの外側の任意の場所をクリックして、「**Properties**」タブに切り替え、「**Debug mode**」属性を`false`に設定します。これで、すべてのデバッグを一度に終了できます。

また、「**Debug max. records**」属性を定義していない場合は、この「**Properties**」タブで、すべてのデバッグ・エッジに対して一度に指定できます(「**Debug mode**」が空かまたは`true`に設定されている場合)。ただし、エッジに固有の「**Debug max. records**」属性値が定義されている場合は、この属性のグローバル値が無視され、そのエッジの属性値が適用されます。

エッジのメモリー割当て

常に問題となるのが、単一レコードに含まれる大量データの操作です。**CloverETL Designer**の場合、グラフ・エッジに沿った大量データの送信が、これを意味します。

- 2つのコンポーネント間で、数MBのデータを単一レコードで搬送する必要がある場合は、それらを接続するエッジによって、常にその容量が拡張されます。これを動的メモリー割当てと呼びます。
- いくつかの部分で非常に大きいデータを転送する複雑なETLシナリオがある場合、これらの部分に含まれるエッジでのみ動的メモリー割当てが使用されます。他のエッジでは少ないメモリー要件が維持されます。

- 以前に大きいレコードを搬送し、それ自身により多くのメモリーを割り当てたことのあるエッジは、二度と縮小しません。グラフの実行が終了するまで、より多くの量のメモリーを消費します。

デフォルトでは、エッジに沿って送信されるレコードの最大サイズは32MBです。理論的にこの値は、Record.RECORD_LIMIT_SIZEプロパティ(デフォルトのCloverETL設定の変更(p.88))を設定することで、数十MBまで増やすことができます。Record.FIELD_LIMIT_SIZEもまた、デフォルトで32MBにすることができます。全フィールドの合計としてRecord.RECORD_LIMIT_SIZEよりも多いメモリーを使用することはできません。

Record.RECORD_LIMIT_SIZEは、どのようなサイズに増やしても問題はありません。この値を小さくしておく唯一の理由は、エラーの早期検出にあります。たとえば、文字列フィールドへの追加を開始して、(各レコードの後の)レコード・リセットを忘れた場合、フィールド・サイズが制限を超す可能性があります。

注意

メモリー内で発生する内容についてもう少し詳しく見ていきましょう。最初に、レコードは64kのメモリーを割り当てられた状態で開始します。大きいデータの転送が必要な場合は、そのサイズを動的にRecord.RECORD_LIMIT_SIZEの値まで増やすことができます。レコードで消費できるメモリーの量は<64k; Record.RECORD_LIMIT_SIZE>になります。

ETLグラフでは、多くのメモリーを必要とするエッジも通常のエッジと同様に表示されます。特別な視覚的効果はありません。

エッジのメモリー要求の測定および見積り

グラフの実行中にエッジおよびコンポーネントで消費されるメモリー量の概要をリアルタイムで示すことができます。これを行うには、**「Window」→「Preferences」**にナビゲートします。次に、**「CloverETL」**を展開して**「Tracking」**を選択する必要があります。このペインで、**「New...」**をクリックし、**「Used Memory」**を選択して確定します。**「Clover - Graph Tracking」**タブに新しい列が表示されます。図10.18. 「Clover - Graph Tracking」タブ(p.46)を参照してください。

グラフを実行する前にそのグラフがメモリーをどの程度必要とするかを見積もるには、次の表を参照してください(注意: 計算は簡略化されています)。一般に、グラフのメモリー要求は、入力データ、使用されるコンポーネントおよびエッジ・タイプによって決まります。ここでは、最後の要因について説明します。グラフを実行する前に、おおよそでどの程度のメモリー量がグラフで使用されるか、およびどの程度までメモリー要求が上昇する可能性があるかについて確認してください。

表20.1. エッジ・タイプごとのメモリー要求

エッジ・タイプ	初期		最大	
ダイレクト	576 kB	9 RIS	96 MB	3 RLS
バッファ付き	1344 kB	21 RIS	96 MB	3 RLS
フェーズ	128 kB	2 RIS	64 MB	2 RLS
ダイレクト高速伝播	256 kB	4 RIS *	128 MB	4 RLS

* ... 4はバッファの数で、変更可能です。一般に、バッファのメモリーは、RLS×(バッファの数)まで上昇する可能性があります。

説明:

RIS = Record.RECORD_INITIAL_SIZE = 64 kB (デフォルト)

RLS = Record.RECORD_LIMIT_SIZE = 32 MB (デフォルト)

第21章 メタデータ

グラフのすべてのエッジでデータが搬送されます。このデータは、メタデータを使用して記述する必要があります。これらのメタデータは内部または外部(共有)のいずれかにすることができます。

メタデータで使用可能なデータ型およびレコード・タイプの詳細は、データ型およびレコード・タイプ(p.111)を参照してください。

様々なデータ型を使用する場合は、書式設定またはロケールも指定できます。次を参照してください。

- 日付と時刻の書式(p.113)
- 数値の書式(p.120)
- ロケール(p.126)

一部のコンポーネントでは、**メタデータ**で**自動入力機能**を使用することもできます。

自動入力関数(p.132)を参照してください。

作成できる各メタデータは次のとおりです。

- **内部**: 内部メタデータ(p.134)を参照してください。

  内部メタデータに対して、次の処理を実行できます。

  - **外部化**: 内部メタデータの外部化(p.135)を参照してください。
  - **エクスポート**: 内部メタデータのエクスポート(p.136)を参照してください。
- **外部(共有)**: 外部(共有)メタデータ(p.137)を参照してください。

  外部(共有)メタデータに対して、次の処理を実行できます。

  - **グラフへのリンク**: 外部(共有)メタデータのリンク(p.137)を参照してください。
  - **内部化**: 外部(共有)メタデータの内部化(p.138)を参照してください。

メタデータは次のソースから作成できます。

- **フラット・ファイル**: フラット・ファイルからのメタデータの抽出(p.139)を参照してください。
- **XLS(X)ファイル**: XLS(X)ファイルからのメタデータの抽出(p.144)を参照してください。
- **DBaseファイル**: DBaseファイルからのメタデータの抽出(p.150)を参照してください。
- **データベース**: データベースからのメタデータの抽出(p.146)を参照してください。
- **ユーザーによる**: ユーザーによるメタデータの作成(p.150)を参照してください。
- **Lotus Notes**: Lotus Notesからのメタデータの抽出(p.150)を参照してください。
- **Cobolコピーブック**
- **既存のメタデータのマージ**: 既存のメタデータのマージ(p.152)を参照してください。

メタデータは、動的に作成したり、リモート・ソースから読み取ることもできます。

- **動的メタデータ**: 動的メタデータ(p.153)を参照してください。
- **特殊なソースからの読取り**: 特殊なソースからのメタデータの読取り(p.154)を参照してください。

メタデータ・エディタについては、メタデータ・エディタ(p.158)で説明しています。

delimitedまたはmixedレコード・タイプのデリミタの変更または定義の詳細は、デリミタの変更および定義(p.165)を参照してください。

メタデータは、そのソース・コードで編集することもできます。ソース・コードのメタデータの編集(p.168)を参照してください。

メタデータは、データベース表を作成するソースとして機能できます。メタデータからのデータベース表の作成に関する項(p.155)を参照してください。

データ型およびレコード・タイプ

エッジを通過するデータ・フローは、メタデータを使用して記述されている必要があります。メタデータでは、レコード全体とすべてのレコード・フィールドの両方を記述します。

次の項で、Cloverのデータ型について説明します。

- メタデータのデータ型(p.111)
- CTLのデータ型(p.831) (CTL1の場合)
- CTL2のデータ型(p.892) (CTL2の場合)

メタデータのデータ型

メタデータで使用されるレコード・フィールドのタイプを次に示します。

表21.1. メタデータのデータ型

データ型	サイズ[5]	範囲または値	デフォルト値
boolean	1ビットを表します。サイズは正確には定義されていません。	`true|false|1|0`	`false|0`
byte	実際のデータ長によって決まります。	-128から127。	`null`
cbyte	実際のデータ長および圧縮の成功によって決まります。	-128から127。	`null`
date	64ビット[1]。	1970年1月1日00:00:00 GMTから始まり、1ms単位で増加します。	現在の日時。
decimal	`Length`および`Scale`によって決まります。(前者はすべての桁の最大数で、後者は小数点以下の桁の最大数です。デフォルト値はそれぞれ12と2です。) [2, 3]	`decimal(6,2)` (-9999.99から9999.99の値を取ることができ、長さと位取りはCTL1でのみ定義できます)	`0.00`
integer	32ビット[2]。	`Integer.MIN_VALUE`から`Integer.MAX_VALUE` (Javaのintegerデータ型に準拠): -2^{31}から$2^{31}-1$。`Integer.MIN_VALUE`は`null`として解釈されます。	0
long	64ビット[2]。	`Long.MIN_VALUE`から`Long.MAX_VALUE` (Javaのlongデータ型に準拠): -2^{63}から$2^{63}-1$。`Long.MIN_VALUE`は`null`として解釈されます。	0

データ型	サイズ[5]	範囲または値	デフォルト値
number	64ビット[2]。	負の値は-(2-2^{-52}).2^{1023}から-2^{-1074}、もう1つの値は0で、正の値は2^{-1074}から(2-2^{-52}).2^{1023}です。3つの特別な値(`NaN`、`-Infinity`および`Infinity`)が定義されています。	`0.0`
string	実際のデータ長によって決まります。各文字は16ビットで格納されます。	無限の文字列を持つことはできません。各文字列の構成可能文字数を制限するかわりに(理論的には、最大で64K)、メモリー要件を検討します。文字列は、(文字数)×2バイトのメモリーを使用します。それと同時に、どのレコードも`MAX_RECORD_SIZE`を上回るバイト数を使用することはできません。第18章「高度なトピック」(p.85)を参照してください。	`null` [4]

説明:

1): どの日付も、日付と時刻の書式パターンを使用して解析および書式設定できます。日付と時刻の書式(p.113)を参照してください。解析と書式設定は、ロケールの影響を受ける可能性もあります。ロケール(p.126)を参照してください。

2): どの数値データ型も、数値の書式パターンを使用して解析および書式設定できます。数値の書式(p.120)を参照してください。解析と書式設定は、ロケールの影響を受ける場合があります。ロケールを(p.126)参照してください。

3): `decimal`のデフォルトの*`length`*および*`scale`*は、それぞれ12と2です。DECIMAL_LENGTHおよびDECIMAL_SCALEのこれらのデフォルト値は、`org.jetel.data.defaultProperties`ファイルに格納されており、他の値への変更が可能です。

4): デフォルトでは、メタデータのstringデータ型であるフィールドが空の文字列である場合、フィールドの**Null value**プロパティを他の値に設定していないかぎり、このフィールドの値は、空の文字列(`""`)ではなく`null`に変換されます。

5): この列は実装の詳細のように見えますが、それほど正確なわけではありません。**サイズ**を使用すると、レコードで必要となる予定のメモリー量を見積もることができます。これを行うには、レコードに含まれるフィールドの数とそれらのデータ型を調べ、その結果を`MAX_RECORD_SIZE`プロパティ(レコードの最大サイズ(バイト数)、第18章「高度なトピック」(p.85)を参照)と比較します。レコードのバイト数がこれよりも多くなる可能性が高い場合は、単純に値を増やします(そうしないと、バッファ・オーバーフローが発生します)。

これらのデータ型およびClover Transformation Language (CTL)で使用されるその他のデータ型の詳細は、CTLのデータ型(p.831)(CTL1の場合)またはCTL2のデータ型(p.892)(CTL2の場合)を参照してください。

レコード・タイプ

各レコードは、次の3つのタイプのいずれかです。

- **デリミタ付き**: これは、隣接する2つのフィールドのすべてがデリミタによって互いに区切られ、レコード全体もレコード・デリミタで終了しているレコードのタイプです。
- **固定**: これは、すべてのフィールドに、指定された長さ(サイズ)があるレコードのタイプです。これは文字数でカウントされます。
- **混合**: これは、デリミタでフィールドを互いに区切ることができ、さらに指定された長さ(サイズ)もあるレコードのタイプです。サイズは文字数でカウントされます。このレコード・タイプは前述の2つのケースを混在させたものです。個々のフィールドでそれぞれ異なるプロパティを持つことができます。一部のフィールドはデリミタのみを持ち、他のフィールドは指定された値を持ち、残りのフィールドはデリミタとサイズの両方を持つというようにできます。

データ形式

書式を定義してデータ値の解析および書式設定を行う場合があります。

1. どの日付も、日付と時刻の書式パターンを使用して解析または書式設定(あるいはその両方)を行うことができます。日付と時刻の書式(p.113)を参照してください。

   解析および書式設定はロケールの影響を受ける可能性もあります(月の名前、日にちまたは月の情報の順序など)。ロケール(p.126)を参照してください。

2. どの数値データ型(`decimal`、`integer`、`long`、`number`)も、数値の書式パターンを使用して解析または書式設定(あるいはその両方)を行うことができます。数値の書式(p.120)を参照してください。

   解析および書式設定はロケールの影響を受ける可能性もあります(小数点、小数点を表すカンマなど)。ロケール(p.126)を参照してください。

3. どのブール・データ型も、ブール書式パターンを使用して解析および書式設定できます。ブール書式(p.124)を参照してください。

4. どの文字列データ型も、文字列書式パターンを使用して解析できます。文字列書式(p.125)を参照してください。

注意

日付と時刻の書式も数値の書式も、システムの**ロケール**値または`defaultProperties`ファイルで指定されている**ロケール**を使用して表示されます。ただし、明示的に別の**ロケール**を指定されている場合を除きます。

`defaultProperties`で**ロケール**を変更する方法の詳細は、デフォルトのCloverETL設定の変更(p.88)を参照してください。

日付と時刻の書式

書式設定文字列では、日付/時刻の値をどのように文字列表現(フラット・ファイル、人が判読できる出力など)から読み取るか、または文字列表現に書き込むかを記述します。

また、書式では、接頭辞を指定することで、**CloverETL**が使用するエンジンを指定することもできます(次を参照)。標準Javaとサードパーティの Joda (http://joda-time.sourceforge.net)の2つの組込み日付エンジンを使用できます。

表21.2. 使用可能な日付エンジン

日付エンジン	接頭辞	デフォルト	説明	例
Java	`java:`	はい(接頭辞が指定されていない場合)	標準Javaの日付実装。寛大で誤りは発生しやすいが、完全な機能を持った解析および書込みを提供します。適度な処理速度で、大量の日付/時刻フィールドを処理する必要がない場合に一般的に適した選択です。詳細は、Java SimpleDateFormatのドキュメントを参照してください。	`java:yyyy-MM-dd HH:mm:ss`

日付エンジン	接頭辞	デフォルト	説明	例
Joda	`joda:`		サードパーティの改良型日付ライブラリ。Jodaは解析時の入力データの精度により厳しく、タイム・ゾーンは適切に処理されません。ただし、標準Javaと比較すると、処理速度は20-30%速くなります。詳細は、http://joda-time.sourceforge.netにあるプロジェクト・サイトを参照してください。 JodaはAS/400マシンに有用です。 一方で、Jodaは、パターンに含まれる任意の数のz文字または3つ以上のz文字(あるいはその両方)で表現されるタイム・ゾーンを読み取ることができません。	`joda:yyyy-MM-dd HH:mm:ss`

JavaおよびJodaの実際の書式文字列は、互いにほぼ100%の互換性があります(下の表を参照)。

重要

この項で説明する書式パターンは、メタデータで**書式**プロパティとして使用され、またCTLでも使用されます。

最初に、Javaのパターン構文、ルールおよびその使用例のリストを示します。

表21.3. 日付書式パターンの構文(Java)

文字	日付または時刻のコンポーネント	表現	例
G	紀元前または紀元後	テキスト	AD
Y	年	年	1996; 96
M	月	月	July、Jul、VII、07、7
w	年初からの週数	数	27
W	月初からの週数	数	2
D	年初からの日数	数	189
d	月初からの日数	数	10
F	月初からのその曜日の回数	数	2
E	曜日	テキスト	Tuesday、Tue
a	Am/pmマーカー	テキスト	PM
H	時(0-23)	数	0
k	時(1-24)	数	24
K	am/pmでの時(0-11)	数	0
h	am/pmでの時(1-12)	数	12
m	分	数	30
s	秒	数	55

文字	日付または時刻のコンポーネント	表現	例
S	ミリ秒	数	970
z	タイム・ゾーン	一般的なタイム・ゾーン	Pacific Standard Time、PST、GMT-08:00
Z	タイム・ゾーン	RFC 822タイム・ゾーン	-0800
'	テキスト/ID用のエスケープ	デリミタ	(なし)
''	一重引用符	リテラル	'

指定する記号文字の数によっても書式が決まります。たとえば、zzパターンの結果がPDTの場合、zzzzパターンではPacific Daylight Timeが生成されます。次の表にこれらのルールをまとめます。

表21.4. 日付書式の使用ルール(Java)

表現	処理	パターン文字の数	形式
テキスト	書式設定	1 - 3	短い形式または省略形式(存在する場合)
テキスト	書式設定	>= 4	完全形式
テキスト	解析	>= 1	両方の形式
年	書式設定	2	2桁に切捨て
年	書式設定	1または>= 3	数として解釈
年	解析	1	リテラルに解釈
年	解析	2	`SimpleDateFormat`インスタンスが作成された日時よりも前の80年または後の20年に含まれる世紀と比較して解釈
年	解析	>= 3	リテラルに解釈
月	両方	1-2	数として解釈
月	解析	>= 3	テキストとして解釈(ローマ数字、月の省略名(ある場合)または月の完全名を使用)
月	書式設定	3	テキストとして解釈(ローマ数字または月の省略名(ある場合)を使用)
月	書式設定	>= 4	テキストとして解釈(月の完全名)
数	書式設定	最小必要桁数	短い数はゼロで埋込み
数	解析	パターン文字の数は無視(隣接する2つのフィールドを区切る必要がない場合)	任意の形式

表現	処理	パターン文字の数	形式
一般的なタイム・ゾーン	両方	1-3	短い形式または省略形式(名前がある場合)。そうでない場合は、GMTオフセット値(GMT[記号][[0]0-23]:[00-59])
一般的なタイム・ゾーン	両方	>= 4	完全形式(名前がある場合)。そうでない場合は、GMTオフセット値(GMT[記号][[0]0-23]:[00-59])
一般的なタイム・ゾーン	解析	>= 1	RFC 822タイム・ゾーン形式が可能
RFC 822タイム・ゾーン	両方	>= 1	RFC 822の4桁のタイム・ゾーン形式を使用([記号][0-23][00-59])
RFC 822タイム・ゾーン	解析	>= 1	一般的なタイム・ゾーン形式が可能

日付書式パターンと結果の日付の例を次に示します。

表21.5. 日付と時刻の書式パターンおよび結果(Java)

日付と時刻のパターン	結果
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, "yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o"clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700

説明した書式パターンは、メタデータで**書式**プロパティとして使用され、またCTLでも使用されます。

次に、Jodaの書式パターン構文のリストを示します。

表21.6. 日付書式パターンの構文(Joda)

記号	説明	表現	例
G	紀元前または紀元後	テキスト	AD
C	世紀(>=0)	数	20
Y	年(>=0)	年	1996
y	年	年	1996
x	暦週の基準年の週数	年	1996
M	月	月	July、Jul、07
w	年初からの週数	数	27
D	年初からの日数	数	189
d	月初からの日数	数	10
e	曜日	数	2
E	曜日	テキスト	Tuesday、Tue
a	AM/PM	テキスト	PM
H	時(0-23)	数	0
k	時(1-24)	数	24
K	時(AM/PM) (0-11)	数	0
h	時(AM/PM) (1-12)	数	12
m	分	数	30
s	秒	数	55
S	ミリ秒	数	970
z	タイム・ゾーン	テキスト	Pacific Standard Time、PST
Z	タイム・ゾーンのオフセット/ID	ゾーン	-0800、-08:00、America/Los_Angeles
'	テキスト/ID用のエスケープ	デリミタ	(なし)
''	一重引用符	リテラル	'

指定する記号文字の数によっても書式が決まります。次の表にこれらのルールをまとめます。

表21.7. 日付書式の使用ルール(Joda)

表現	処理	パターン文字の数	形式
テキスト	書式設定	1 - 3	短い形式または省略形式(存在する場合)
テキスト	書式設定	>= 4	完全形式
テキスト	解析	>= 1	両方の形式

表現	処理	パターン文字の数	形式
年	書式設定	2	2桁に切捨て
年	書式設定	1または>= 3	数として解釈
年	解析	>= 1	リテラルに解釈
月	両方	1-2	数として解釈
月	解析	>= 3	テキストとして解釈(ローマ数字、月の省略名(ある場合)または月の完全名を使用)
月	書式設定	3	テキストとして解釈(ローマ数字または月の省略名(ある場合)を使用)
月	書式設定	>= 4	テキストとして解釈(月の完全名)
数	書式設定	最小必要桁数	短い数はゼロで埋込み
数	解析	>= 1	任意の形式
ゾーン名	書式設定	1-3	短い形式または省略形式
ゾーン名	書式設定	>= 4	完全形式
タイム・ゾーンのオフセット/ID	書式設定	1	時と分の間にコロンがないオフセット
タイム・ゾーンのオフセット/ID	書式設定	2	時と分の間にコロンがあるオフセット
タイム・ゾーンのオフセット/ID	書式設定	>= 3	「大陸/都市」のような完全テキスト形式
タイム・ゾーンのオフセット/ID	解析	1	時と分の間にコロンがないオフセット
タイム・ゾーンのオフセット/ID	解析	2	時と分の間にコロンがあるオフセット

重要

任意の数のz文字を使用した解析は許可されていません。また、3個以上のZ文字を使用した解析も許可されていません。

メタデータとCTL1およびCTL2におけるデータ型の情報を参照してください。

- データ型およびレコード・タイプ(p.111)
- **CTL1の場合:**

  CTLのデータ型(p.831)
- **CTL2の場合:**

  CTL2のデータ型(p.892)

これらは、CTL1およびCTL2の関数でも使用されます。次を参照してください。

CTL1の場合:

- 変換関数(p.860)
- 日付関数(p.865)
- 文字列関数(p.872)

CTL2の場合:

- 変換関数(p.921)
- 日付関数(p.928)
- 文字列関数(p.934)

数値の書式

テキストを数値データ型として解析する場合、または数値データ型をテキストに書式設定する場合は、書式パターンを指定する必要があります。

解析および書式設定はロケールに依存します。

CloverETLでは、Javaの10進数書式が使用されます。

表21.8. 数値書式パターンの構文

記号	位置	ローカライズ	説明
#	数	はい	数字。ゼロの場合は表示されない。
0	数	はい	数字。
.	数	はい	小数区切りまたは金額の小数区切り。
-	数	はい	マイナス記号。
,	数	はい	グループ化セパレータ。
E	数	はい	科学表記法の仮数と指数を区切る。接頭辞や接尾辞内で引用符で囲む必要はない。
;	サブパターン境界	はい	正と負のサブパターンを区切る。
%	接頭辞または接尾辞	はい	100倍してパーセントを表す。
‰(\u2030)	接頭辞または接尾辞	はい	1000倍してパーミル値を表す。
¤ (\u00A4)	接頭辞または接尾辞	いいえ	通貨記号で置換される通過符号。2つの場合は、国際通貨記号で置換される。パターン内にある場合は、小数区切りではなく、金額の小数区切りが使用される。
'	接頭辞または接尾辞	いいえ	接頭辞や接尾辞内の特殊文字を引用符で囲む場合に使用される。たとえば、'#'#を使用すると123は#123に書式設定される。一重引用符自体を作成するには、1行に2つの引用符を使用する(# o"clock)。

- 接頭辞も接尾辞も\u0000から\uFFFDのUnicode文字で、これらの境界は含みますが、特殊文字は含みません。

次の表に示すように、書式パターンはサブパターン、接頭辞、接尾辞などで構成されます。

表21.9. BNFダイアグラム

書式	コンポーネント
パターン	subpattern{;subpattern}
サブパターン	{prefix}integer{.fraction}{suffix}
接頭辞	'\\u0000'..'\\uFFFD' - specialCharacters
接尾辞	'\\u0000'..'\\uFFFD' - specialCharacters
整数	'#'* '0'* '0'
小数	'0'* '#'*

これらの記号の説明は次のとおりです。

表21.10. 使用されている表記法

表記法	説明
X*	Xの0個以上のインスタンス
(X | Y)	XまたはYのいずれか
X..Y	XからYまでの任意の文字、包括的
S - T	Sに含まれる文字、ただしTに含まれる文字を除く
{X}	Xはオプション

重要

一般的に、グループ化セパレータは千ごとに使用されますが、万ごとに区切る国もあります。グループ・サイズとは、グループ化文字間の一定の桁数です(100,000,000の場合は3、1,0000,0000の場合は4など)。複数のグループ化文字を含むパターンを指定した場合、最後のグループ化文字と整数の終わりとの間隔が、使用されるグループ・サイズになります。そのため、"#,##,###,####" == "######,####" == "##,####,####"です。

書式設定もまたローカル依存になります。次の表を参照して、それぞれのロケールで結果がどのように異なるかを確認してください。

表21.11. ロケール依存の書式設定

パターン	ロケール	結果
###,###.###	en.US	123,456.789
###,###.###	de.DE	123.456,789
###,###.###	fr.FR	123 456,789

注意

数の処理の詳細は、公式のJavaドキュメントを参照してください。

科学表記法

科学表記法における数は、仮数と10の累乗の積で表現されます。

たとえば、1234は1.234×10^3で表すことができます。

ほとんどの場合、仮数は1.0 <= x < 10.0の範囲内になりますが、そうである必要はありません。

数値データ型は、パターンによってのみ、科学表記法を書式設定および解析するように指示できます。パターンでは、指数文字のすぐ後に1つ以上の文字を続けて、科学表記法を表します。

例: 0.###E0を使用すると、1234という数は1.234E3として書式設定されます。

数値パターンと結果の例を次に示します。

表21.12. 数値の書式パターンおよび結果

値	パターン	結果
1234	0.###E0	1.234E3
12345	##0.#####E0[1)]	12.345E3
123456	##0.#####E0[1)]	123.456E3
1234567	##0.#####E0[1)]	1.234567E6
12345	#0.#####E0[2)]	1.2345E4
123456	#0.#####E0[2)]	12.3456E4
1234567	#0.#####E0[2)]	1.234567E6
0.00123	00.###E0[3)]	12.3E-4
123456	##0.##E0[4)]	12.346E3

説明:

1): 各ケースで、整数桁の最大数は3、整数桁の最小数は1、最大は最小よりも大きい、したがって指数は3 (整数桁の最大数)の倍数になります。

2): 各ケースで、整数桁の最大数は2、整数桁の最小数は1、最大は最小よりも大きい、したがって指数は2 (整数桁の最大数)の倍数になります。

3): 整数桁の最大数は2、整数桁の最小数は2、最大と最小が等しい、したがって整数桁の最小数は指数を調整することによって得られます。

4): 整数桁の最大数は3、小数桁の最大数は2、有効桁数は整数桁の最大数と小数桁の最大数の合計、したがって有効桁数は図のとおり(5桁)になります。

バイナリ形式

次の表に、使用可能な形式のリストを示します。

表21.13. 使用可能なバイナリ形式

型	名前	形式	長さ
整数	BIG_ENDIAN	2の補数、ビッグエンディアン	可変
	LITTLE_ENDIAN	2の補数、リトルエンディアン	
	PACKED_DECIMAL	パック10進数	
浮動小数点	DOUBLE_BIG_ENDIAN	IEEE 754、ビッグエンディアン	8バイト
	DOUBLE_LITTLE_ENDIAN	IEEE 754、リトルエンディアン	
	FLOAT_BIG_ENDIAN	IEEE 754、ビッグエンディアン	4バイト
	FLOAT_LITTLE_ENDIAN	IEEE 754、リトルエンディアン	

浮動小数点形式は、numericおよびdecimalデータ型で使用できます。整数形式は、integerおよびlongデータ型で使用できます。このルールの例外はdecimalデータ型で、整数形式(BIG_ENDIAN、LITTLE_ENDIANおよびPACKED_DECIMAL)もサポートしています。整数形式がdecimalデータ型で使用されると、**位取り**属性に従って、暗黙的な小数点が設定されます。たとえば、格納されている値が123456789で**位取り**が3に設定されている場合、フィールドの値は123456.789になります。

バイナリ形式を使用するには、サポートされているいずれかのデータ型を使用してメタデータ・フィールドを作成し、**書式**属性を"BINARY:"の接頭辞を付けた書式名に設定します。たとえば、PACKED_DECIMAL形式を使用するには、10進数フィールドを作成し、その**書式**属性を、使用可能な書式のリストから選択して、"BINARY:PACKED_DECIMAL"に設定します。

固定長形式(doubleおよびfloat)では、サイズ属性も適宜に設定する必要があります。

現在、バイナリ・データ形式は、ComplexDataReader(p.343)および非推奨のFixLenDataReaderでのみ処理できます。

ブール書式

メタデータで指定されるブール・データ型の書式は、同じデリミタで互いに区切られた最大4つの部分で構成されます。

このデリミタは**書式設定**文字列の最初と最後にも指定されている必要があります。一方で、そのデリミタがブール・フィールドの値には含まれないようにする必要があります。

重要

書式設定文字列の最初と最後に同じ文字を使用しなかった場合、文字列全体が`true`値の正規表現として機能します。デフォルト値(`false|F|FALSE|NO|N|f|0|no|n`)が、`false`として解釈される唯一の値になります。

書式設定の正規表現(`true`としてのみ解釈される)にも前述の`false`のデフォルト値にも一致しない値は、エラーとして解釈されます。このような場合、グラフは失敗します。

書式を次のように象徴的に表した場合:

```
/A/B/C/D/
```

各部分の意味は次のようになります。

1. ブール・フィールドの値が最初の部分(A)のパターンと一致し、2番目の部分(B)とは一致していない場合、`true`として解釈されます。
2. ブール・フィールドの値が最初の部分(A)のパターンとは一致せず、2番目の部分(B)と一致している場合、`false`として解釈されます。
3. ブール・フィールドの値が最初の部分(A)のパターンと一致し、同時に、2番目の部分(B)のパターンとも一致している場合、`true`として解釈されます。
4. ブール・フィールドの値が最初の部分(A)のパターンとも、2番目の部分(B)のパターンとも一致していない場合、エラーとして解釈されます。このような場合、グラフは失敗します。

すべての部分がオプションですが、これらのいずれかを省略する場合、その右側にある他の部分もすべて省略する必要があります。

2番目の部分(B)を省略した場合、次のデフォルト値がブール値`false`として解析される唯一の値です。

```
false|F|FALSE|NO|N|f|0|no|n
```

書式設定がない場合、次のデフォルト値がブール値`true`として解析される唯一の値です。

```
true|T|TRUE|YES|Y|t|1|yes|y
```

- 3番目の部分(C)は、すべての一致文字列に対して、ブール値`true`を表すのに使用される書式設定文字列です。3番目の部分を省略した場合、`true`という単語が使用されるか(最初の部分(A)が複雑な正規表現である場合)、または最初の部分の最初のサブ文字列が使用されます(最初の部分がパイプで区切られた単純なサブ文字列の並び(`Iagree|sure|yes|ok`など)である場合。これらの値はすべて`Iagree`として書式設定されます)。
- 4番目の部分(D)は、すべての一致文字列に対して、ブール値`false`を表すのに使用される書式設定文字列です。4番目の部分を省略した場合、`false`という単語が使用されるか(2番目の部分(B)が複雑な正規表現である場合)、または2番目の部分の最初のサブ文字列が使用されます(2番目の部分がパイプで区切られた単純なサブ文字列の並び(`Idisagree|nope|no`など)である場合。これらの値はすべて`Idisagree`として書式設定されます)。

文字列書式

この文字列パターンは、文字列の解析を許可または禁止する正規表現(p.963)です。

例21.1. 文字列書式

入力ファイルに文字列フィールドが含まれ、このフィールドの**書式**プロパティが`\\w{4}`である場合、長さが4である文字列のみが解析されます。

このように、**書式**プロパティが文字列に対して指定されている場合、**データ・ポリシー**によってグラフが失敗することがあります(**データ・ポリシー**が`Strict`の場合)。

データ・ポリシーが`Controlled`または`Lenient`に設定されている場合、この文字列値と指定された**書式**プロパティが一致しているレコードが読み取られ、その他のレコードはスキップされます(**コンソール**または拒否ポートに送信されます)。

ロケールおよびロケール依存

ロケールのプロパティに応じて、様々なデータ型(日時、任意の数値、文字列)を様々な方法で表示、解析または書式設定できます。詳細は、ロケール(p.126)を参照してください。

文字列も**ロケール依存**の影響を受ける場合があります。ロケール依存(p.131)を参照してください。

ロケール

ロケールは、地理的、政治的または文化的に特定の地域を表します。タスクの実行に**ロケール**が必要となる操作はロケール依存と呼ばれ、ユーザーに合せて情報を調整するために**ロケール**が使用されます。たとえば、数値はユーザーの国、地域または文化の習慣や慣行に従って書式設定される必要があるため、数値の表示はロケール依存の操作です。

各ロケール・コードは、言語コードおよび国の引数で構成されます。

言語の引数は、有効なISO言語コードです。これらのコードは、ISO-639で定義されている2文字の小文字のコードです。

国の引数は、有効なISO国コードです。これらのコードは、ISO-3166で定義されている2文字の大文字のコードです。

書式パラメータを指定するかわりに(または指定するとともに)、ロケール・パラメータを指定できます。

- 文字列では、日付フィールド全体に書式を設定するかわりに、たとえばドイツ語ロケールを指定します。これにより、ドイツで使用される正しい日付書式が自動的に選択されるようになります。ロケールが指定されていない場合、Cloverでは、システム指定のデフォルトのロケールが選択されます。デフォルトのロケールの変更方法は、デフォルトのCloverETL設定の変更(p.88)を参照してください。
- 一方、数値では、書式パラメータとロケール・パラメータの両方が重要となる場合があります。10進数の書式を指定する場合、小数区切りを使用して書式/パターンを定義し、ロケールによって小数区切りがカンマであるかドットであるかが判断されます。ロケールも書式も指定されていない場合、数値は(defaultPropertiesを確認しないで)汎用的な技術を使用して文字列に変換されます。書式パラメータのみが指定された場合は、デフォルトのロケールが使用されます。

例21.2. ロケールの例

en.USまたはen.GB

ロケールが変更された場合に影響を受けるその他の書式設定の例は、ロケール依存の書式設定(p.121)を参照してください。

日付も、言語が同じで国が異なる場合を含め、異なるロケールでは書式が異なる場合があります。たとえば、`March 2, 2009` (米国)と`2 March 2009` (英国)。

すべてのロケールのリスト

CloverETLでサポートされているロケールの完全なリストを次の表に示します。前述のように、ロケールの書式は常にlanguage.COUNTRYとなります。

表21.14. すべてのロケールのリスト

ロケール・コード	説明
[システム・デフォルト]	OSで決定されるロケール
ar	アラビア語
ar.AE	アラビア語 - アラブ首長国連邦

ロケール・コード	説明
ar.BH	アラビア語 - バーレーン
ar.DZ	アラビア語 - アルジェリア
ar.EG	アラビア語 - エジプト
ar.IQ	アラビア語 - イラク
ar.JO	アラビア語 - ヨルダン
ar.KW	アラビア語 - クウェート
ar.LB	アラビア語 - レバノン
ar.LY	アラビア語 - リビア
ar.MA	アラビア語 - モロッコ
ar.OM	アラビア語 - オマーン
ar.QA	アラビア語 - カタール
ar.SA	アラビア語 - サウジアラビア
ar.SD	アラビア語 - スーダン
ar.SY	アラビア語 - シリア・アラブ共和国
ar.TN	アラビア語 - チュニジア
ar.YE	アラビア語 - イエメン
be	ベラルーシ語
be.BY	ベラルーシ語 - ベラルーシ
bg	ブルガリア語
bg.BG	ブルガリア語 - ブルガリア
ca	カタロニア語
ca.ES	カタロニア語 - スペイン
cs	チェコ語
cs.CZ	チェコ語 - チェコ共和国
da	デンマーク語
da.DK	デンマーク語 - デンマーク
de	ドイツ語
de.AT	ドイツ語 - オーストリア
de.CH	ドイツ語 - スイス
de.DE	ドイツ語 - ドイツ
de.LU	ドイツ語 - ルクセンブルグ
el	ギリシャ語
el.CY	ギリシャ語 - キプロス

ロケール・コード	説明
el.GR	ギリシャ語 - ギリシャ
en	英語
en.AU	英語 - オーストラリア
en.CA	英語 - カナダ
en.GB	英語 - 英国
en.IE	英語 - アイルランド
en.IN	英語 - インド
en.MT	英語 - マルタ
en.NZ	英語 - ニュージーランド
en.PH	英語 - フィリピン
en.SG	英語 - シンガポール
en.US	英語 - アメリカ合衆国
en.ZA	英語 - 南アフリカ
es	スペイン語
es.AR	スペイン語 - アルゼンチン
es.BO	スペイン語 - ボリビア
es.CL	スペイン語 - チリ
es.CO	スペイン語 - コロンビア
es.CR	スペイン語 - コスタリカ
es.DO	スペイン語 - ドミニカ共和国
es.EC	スペイン語 - エクアドル
es.ES	スペイン語 - スペイン
es.GT	スペイン語 - グアテマラ
es.HN	スペイン語 - ホンジュラス
es.MX	スペイン語 - メキシコ
es.NI	スペイン語 - ニカラグア
es.PA	スペイン語 - パナマ
es.PR	スペイン語 - プエルトリコ
es.PY	スペイン語 - パラグアイ
es.US	スペイン語 - アメリカ合衆国
es.UY	スペイン語 - ウルグアイ
es.VE	スペイン語 - ベネズエラ
et	エストニア語

ロケール・コード	説明
et.EE	エストニア語 - エストニア
fi	フィンランド語
fi.FI	フィンランド語 - フィンランド
fr	フランス語
fr.BE	フランス語 - ベルギー
fr.CA	フランス語 - カナダ
fr.CH	フランス語 - スイス
fr.FR	フランス語 - フランス
fr.LU	フランス語 - ルクセンブルグ
ga	アイルランド語
ga.IE	アイルランド語 - アイルランド
he	ヘブライ語
he.IL	ヘブライ語 - イスラエル
hi.IN	ヒンディー語 - インド
hr	クロアチア語
hr.HR	クロアチア語 - クロアチア
id	インドネシア語
id.ID	インドネシア語 - インドネシア
is	アイスランド語
is.IS	アイスランド語 - アイスランド
it	イタリア語
it.CH	イタリア語 - スイス
it.IT	イタリア語 - イタリア
iw	ヘブライ語
iw.IL	ヘブライ語 - イスラエル
ja	日本語
ja.JP	日本語 - 日本
ko	韓国語
ko.KR	韓国語 - 大韓民国
lt	リトアニア語
lt.LT	リトアニア語 - リトアニア
lv	ラトビア語
lv.LV	ラトビア語 - ラトビア

ロケール・コード	説明
mk	マケドニア語
mk.MK	マケドニア語 - マケドニア旧ユーゴスラビア共和国
ms	マレー語
ms.MY	マレー語 - ビルマ語
mt	マルタ語
mt.MT	マルタ語 - マルタ
nl	オランダ語
nl.BE	オランダ語 - ベルギー
nl.NL	オランダ語 - オランダ
no	ノルウェー語
no.NO	ノルウェー語 - ノルウェー
pl	ポーランド語
pl.PL	ポーランド語 - ポーランド
pt	ポルトガル語
pt.BR	ポルトガル語 - ブラジル
pt.PT	ポルトガル語 - ポルトガル
ro	ルーマニア語
ro.RO	ルーマニア語 - ルーマニア
ru	ロシア語
ru.RU	ロシア語 - ロシア連邦
sk	スロバキア語
sk.SK	スロバキア語 - スロバキア
sl	スロベニア語
sl.SI	スロベニア語 - スロベニア
sq	アルバニア語
sq.AL	アルバニア語 - アルバニア
sr	セルビア語
sr.BA	セルビア語 - ボスニア・ヘルツェゴビナ
sr.CS	セルビア語 - セルビア・モンテネグロ
sr.ME	セルビア語 - セルビア(キリル、モンテネグロ)
sr.RS	セルビア語 - セルビア(ラテン、セルビア)
sv	スウェーデン語
sv.SE	スウェーデン語 - スウェーデン

ロケール・コード	説明
th	タイ語
th.TH	タイ語 - タイ
tr	トルコ語
tr.TR	トルコ語 - トルコ
uk	ウクライナ語
uk.UA	ウクライナ語 - ウクライナ
vi.VN	ベトナム語 - ベトナム
zh	中国語
zh.CN	中国語 - 中国
zh.HK	中国語 - 香港
zh.SG	中国語 - シンガポール
zh.TW	中国語 - 台湾

ロケール依存

ロケール依存は、stringデータ型にのみ適用できます。さらに、**ロケール**はフィールドまたはレコード全体に対して指定する必要があります。

フィールド設定によって、レコード全体に対して指定した**ロケール依存**がオーバーライドされます。

ロケール依存の値を次に示します。

- `base_letter_sensitivity`

  文字の大文字/小文字を識別せず、発音区別文字の付いた文字も識別しません。

- `accent_sensitivity`

  文字の大文字/小文字を識別しません。発音区別文字の付いた文字は識別します。

- `case_sensitivity`

  文字の大文字/小文字を識別し、発音区別文字の付いた文字を識別します。文字エンコーディングは識別しません(\u00C0はA\u0300と等しくなります)。

- `identical_sensitivity`

  文字エンコーディングを識別します(\u00C0はA\u0300と等しくなります)。

自動入力関数

事前に定義された特別な値(読み取るファイルの名前、データ・ソースのサイズなど)をレコードに入力するために使用できる一連の関数があります。これらの関数は、*メタデータ・エディタ*→「**Details**」ペイン→「**Advanced Properties**」で使用できます。

次の関数は、**ParallelReader**、**QuickBaseRecordReader**および**QuickBaseQueryReader**を除くほとんどの**リーダー**でサポートされています。

ErrCode関数とErrText関数は、コンポーネント**DBExecute**、**DBOutputTable**および**XMLExtract**でのみ使用できます。

MultiLevelReaderコンポーネントに値を自動入力する特別な場合のtrueに注意してください。

- default_value: **リーダー**によって値が読み取られなかった場合は、「**Default**」プロパティとして指定された、対応するデータ型の値が設定されます。

- global_row_count。この関数は、1つの**リーダー**で読み取られるすべてのソースのレコードをカウントします。エッジ内の数値データ型の指定フィールドに整数を順次入力します。レコードは、出力ポート経由で送信されたのと同じ順序で番号が付けられます。番号付けは0から開始します。データ・レコードが複数のデータ・ソースから読み取られる場合は、すべてのデータ・ソースにわたって連続する番号が付けられます。(**XMLExtract**内などの)一部のエッジにそのようなフィールドが含まれていない場合、対応する番号はスキップされます。番号付けは続行されます。

- source_row_count。この関数は、1つの**リーダー**で読み取られる各ソースのレコードを個別にカウントします。エッジ内の数値データ型の指定フィールドに整数を順次入力します。レコードは、出力ポート経由で送信されたのと同じ順序で番号が付けられます。各ソース・ファイルのレコードの番号は、他のソースとは関係なく付けられます。各データ・ソースの番号付けは0から開始します。(**XMLExtract**内などの)一部のエッジにそのようなフィールドが含まれていない場合、対応する番号はスキップされます。番号付けは続行されます。

- metadata_row_count。この関数は、ある**リーダー**で読み取られてエッジに送信される、同じメタデータが割り当てられたすべてのソースのレコードをカウントします。エッジ内の数値データ型の指定フィールドに整数を順次入力します。レコードは、出力ポート経由で送信されたのと同じ順序で番号が付けられます。番号付けは0から開始します。データ・レコードが複数のデータ・ソースから読み取られる場合は、すべてのデータ・ソースにわたって連続する番号が付けられます。

- metadata_source_row_count。この関数は、ある**リーダー**で読み取られ、エッジに送信される、同じメタデータが割り当てられた各ソースのレコードをカウントします。エッジ内の数値データ型の指定フィールドに整数を順次入力します。レコードは、出力ポート経由で送信されたのと同じ順序で番号が付けられます。各ソース・ファイルのレコードの番号は、他のソースとは関係なく付けられます。各データ・ソースの番号付けは0から開始します。

- source_name。この関数は、文字列データ型の指定のレコード・フィールドに、レコードの読取り元のデータ・ソースの名前を入力します。

- source_timestamp。この関数は、日付データ型の指定のレコード・フィールドに、レコードの読取り元のデータ・ソースに対応するタイムスタンプを入力します。この関数は**DBInputTable**では使用できません。

- source_size。この関数は、数値データ型の指定のレコード・フィールドに、レコードの読取り元のデータ・ソースのサイズを入力します。この関数は**DBInputTable**では使用できません。

- row_timestamp。この関数は、日付データ型の指定のレコード・フィールドに、個々のレコードが読み取られた時間を入力します。

- reader_timestamp。この関数は、日付データ型の指定のレコード・フィールドに、リーダーが読取りを開始した時間を入力します。この値は、リーダーにより読み取られるすべてのレコードで同じになります。

- ErrCode。この関数は、整数データ型の指定のレコード・フィールドに、コンポーネントから返されたエラー・コードを入力します。**DBOutputTable**コンポーネントと**DBExecute**コンポーネントでのみ使用できます。

- ErrText。この関数は、文字列データ型の指定のレコード・フィールドに、コンポーネントから返されたエラー・メッセージを入力します。**DBOutputTable**コンポーネントと**DBExecute**コンポーネントでのみ使用できます。

- sheet_name。この関数は、文字列データ型の指定のレコード・フィールドに、データ・レコードの読取り元の入力XLS(X)ファイルのシート名を入力します。**SpreadsheetDataReader**コンポーネントと**XLSDataReader**コンポーネントでのみ使用できます。

内部メタデータ

前述のように、内部メタデータはグラフの一部であり、グラフに含まれ、そのソース・タブで表示できます。

内部メタデータの作成

内部メタデータは、次の方法で作成できます。

- **アウトライン**

  「**Outline**」ペインで、**メタデータ**項目を選択し、右クリックしてコンテキスト・メニューを開き、**新しいメタデータ**項目を選択します。

- **グラフ・エディタ: エッジ**

  グラフ・エディタで、任意の**エッジ**を右クリックしてコンテキスト・メニューを開く必要があります。そこに**新しいメタデータ**項目が表示されます。

- **グラフ・エディタ: コンポーネント**

  コンポーネントを使用してメタデータを作成するには、最初に必須プロパティを入力します。その後、**コンポーネント**を右クリックし、「**Extract Metadata**」を選択します。

内部メタデータの作成: アウトラインまたはエッジ

どちらの場合も、**新規メタデータ**項目を選択すると、新しいサブメニューが表示されます。そこでメタデータの定義方法を選択できます。

前述のいずれの場合も、3通りの方法があります。つまり、メタデータを自分で定義する場合は「**User defined**」項目を選択する必要があり、ファイルからメタデータを抽出する場合は「**Extract from flat file**」項目または「**Extract from xls(x) file**」項目を選択する必要があり、データベースからメタデータを抽出する場合は「**Extract from database**」項目を選択する必要があります。このようにして、内部メタデータのみを作成できます。

コンテキスト・メニューを使用してメタデータを定義する場合、メタデータは作成されるとすぐにエッジに割り当てられます。

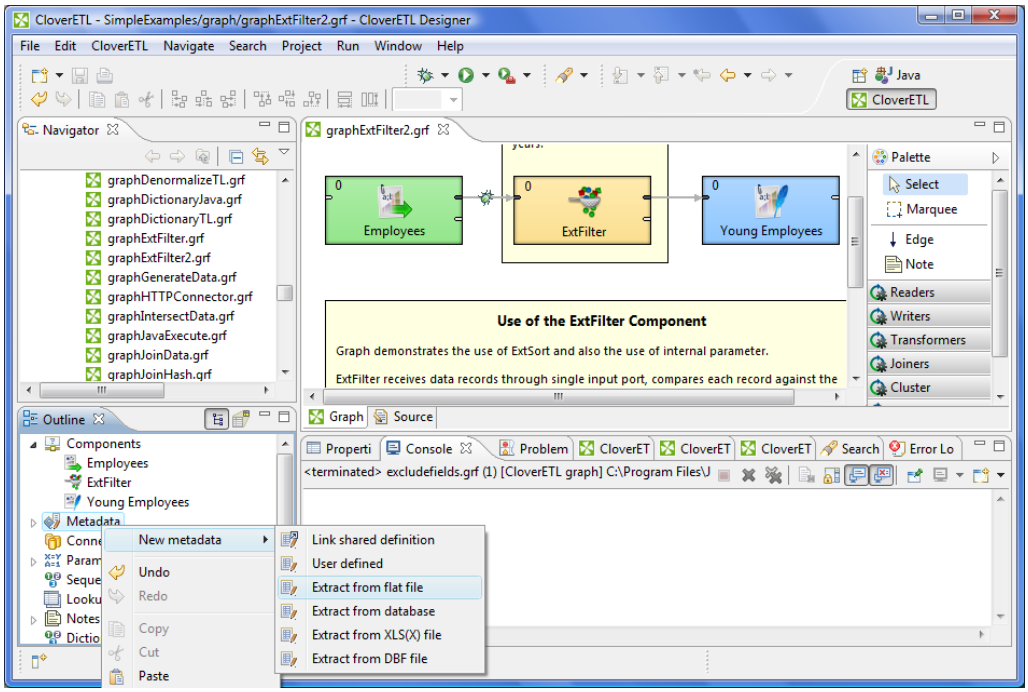


図21.1. 「Outline」ペインでの内部メタデータの作成

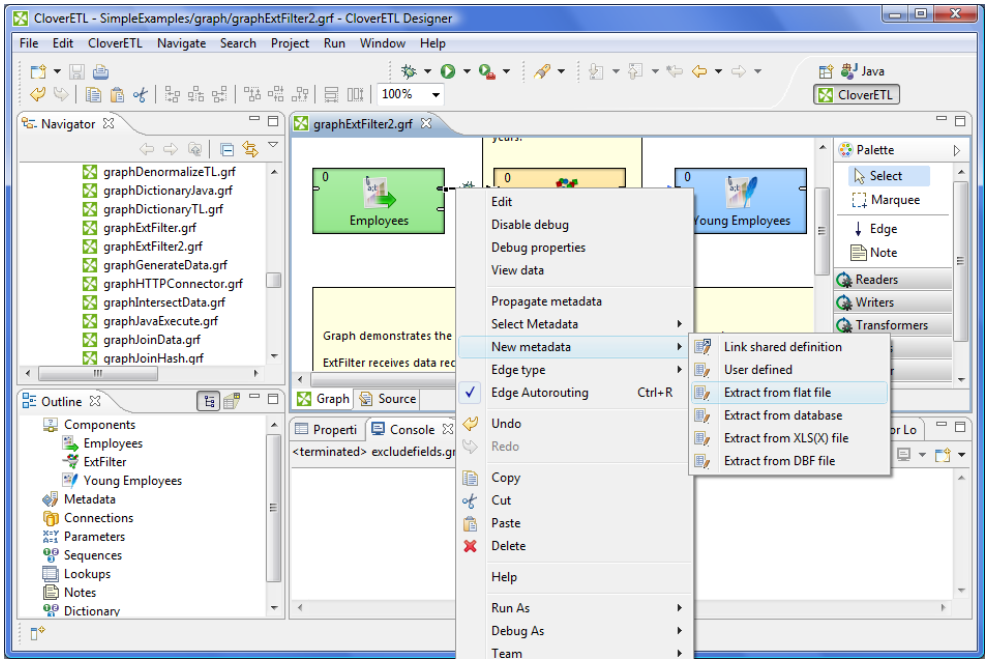


図21.2. エッジでの内部メタデータの作成

内部メタデータの作成: コンポーネント

多くのリーダーおよびライターでは、コンポーネントのプロパティを使用してメタデータを抽出できます。コンポーネントのタイプに基づいて、メタデータはファイル、データベース表またはその他のソースから抽出されます。

サポートされているコンポーネントは、UniversalDataReader、ParallelReader、DBInputTable、XLSDataReader、DBFDataReader、LotusReader、UniversalDataWriter、DBOutputTable、XLSDataWriter、DBFDataWriter、LotusWriter、DB2DataWriter、InfobrightDataWriter、InformixDataWriter、MSSQLDataWriter、MysqlDataWriter、OracleDataWriter、PostgreSQLDataWriterです。

「**Extract metadata**」コンテキスト・メニューは、そのコンポーネントに対して、必須ファイル、接続プロパティまたはデータベース・プロパティが設定されている場合にのみ使用できます。

内部メタデータの外部化

グラフの一部として内部メタデータを作成した後、外部(共有)メタデータに変換できます。そのような場合は、同じメタデータを他のグラフで使用できるようになります(他のグラフでメタデータを共有します)。

任意の内部メタデータ項目を外部化して外部(共有)ファイルにするには、「**Outline**」ペインで内部メタデータ項目を右クリックし、コンテキスト・メニューから「**Externalize metadata**」を選択します。このことを実行すると、新しいウィザードが開き、新しい外部(共有)メタデータ・ファイルの場所としてプロジェクトの`meta`フォルダが提案されるので、「**OK**」をクリックします。必要な場合は、提案されたメタデータ・ファイル名を変更できます。

その後、内部メタデータ項目は「**Outline**」ペインの「**Metadata**」グループからなくなり、同じ場所に、すでにリンクされた状態で、新しく作成された外部(共有)メタデータ・ファイルが表示されます。プロジェクトの`meta`サブフォルダに、同じメタデータ・ファイルが表示されることを「**Navigator**」ペインで確認できます。

複数の内部メタデータ項目を一度に外部化することもできます。このことを行うには、「**Outline**」ペインで内部パラメータを選択して右クリックした後、コンテキスト・メニューから「**Externalize metadata**」を選択します。このことを実行すると、新しいウィザードが開き、選択した最初の内部メタデータ項目の場所としてプロジェクトの`meta`フォルダが提案されるので、「**OK**」をクリックします。選択したメタデータ項目がすべて外部化されるまで、項目ごとに同じウィザードが開きます。必要に応じて(同じ名前のファイルがすでに存在するなど)、提案されたメタデータ・ファイル名を変更できます。

[Shift]を押しながら**下矢印**または**上矢印**キーで移動すると、隣接するメタデータ項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各メタデータ項目を**クリック**します。

内部メタデータのエクスポート

このケースは、メタデータの外部化と似ています。グラフの外部にあるメタデータ・ファイルを、外部化されたファイルのメタデータ・ファイルと同じ方法で作成しますが、そのようなファイルは元のグラフにリンクされません。メタデータ・ファイルのみが作成されます。その後、前の項で説明したように、そのようなファイルを外部(共有)メタデータ・ファイルとして他のグラフに使用できます。

内部メタデータを外部(共有)メタデータにエクスポートするには、「**Outline**」ペインの内部メタデータ項目を右クリックし、コンテキスト・メニューから「**Export Metadata**」をクリックして、メタデータを追加するプロジェクトを選択し、そのプロジェクトを展開して`meta`フォルダを選択し、必要な場合はメタデータ・ファイルの名前を変更して「**Finish**」をクリックします。

その後、「**Outline**」ペインのメタデータ・フォルダは同じままですが、「**Navigator**」ペインの`meta`フォルダには、新しく作成されたメタデータ・ファイルが表示されます。

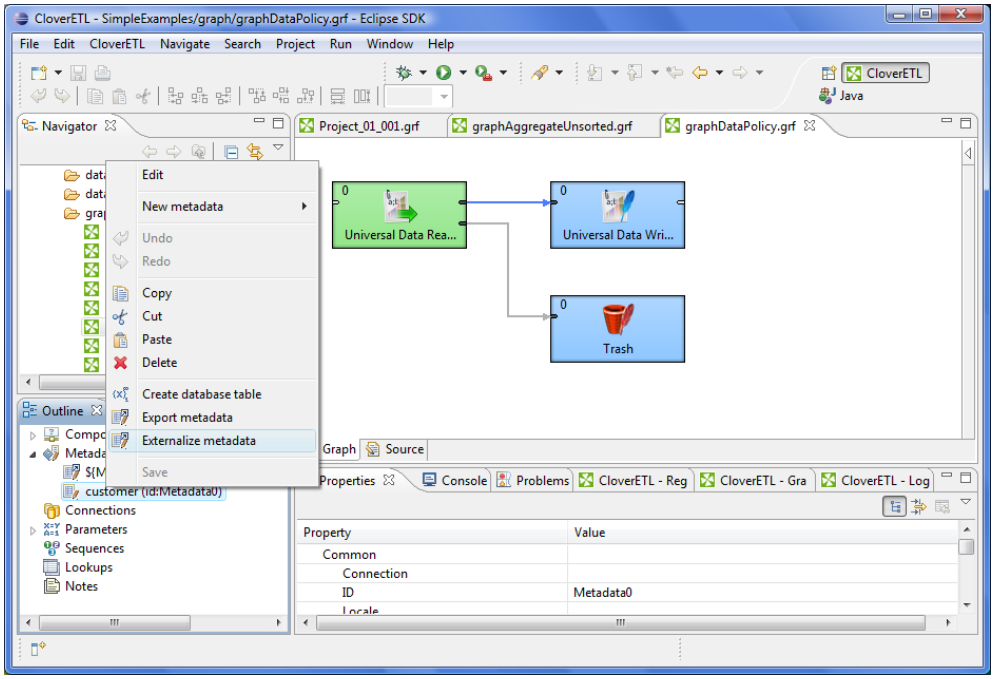


図21.3. 内部メタデータの外部化/エクスポート

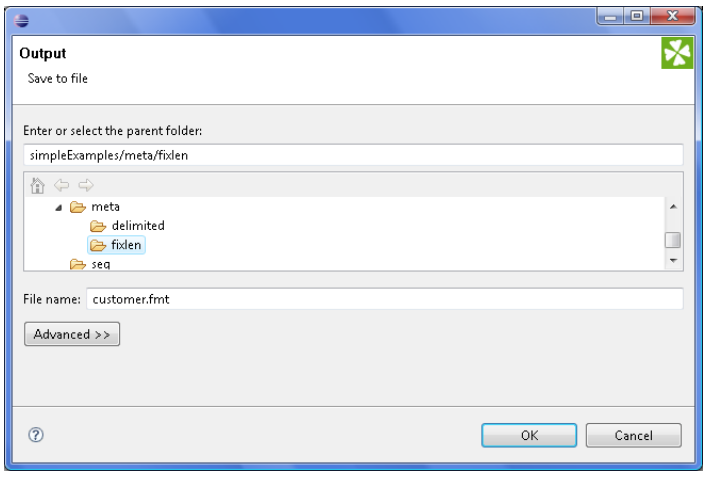


図21.4. 新しく外部化/エクスポートする内部メタデータの場所の選択

外部(共有)メタデータ

前述のように、外部(共有)メタデータは1つのみでなく複数のグラフに使用できるメタデータです。グラフの外部にあり、複数のグラフにわたって共有できます。

外部(共有)メタデータの作成

共有メタデータを作成する場合は、次の2つの方法で実行できます。

- メイン・メニューで「**File**」→「**New**」→「**Other**」を選択して、実行できます。

  外部(共有)メタデータを作成するには、「**Other**」項目をクリックした後に、「**CloverETL**」項目を選択し、それを展開して、自分でメタデータを定義するか(「**User defined**」)、ファイルから抽出するか(「**Extract from flat file**」または「**Extract from XLS file**」)、あるいはデータベースから抽出するか(「**Extract from database**」)を決定する必要があります。

- 「**Navigator**」ペインで実行できます。

  外部(共有)メタデータを作成するには、右クリックしてコンテキスト・メニューを開き、そこで「**New**」→「**Others**」を選択し、ウィザードのリストを開いた後に「**CloverETL**」項目を選択し、それを展開して、自分でメタデータを定義するか(「**User defined**」)、ファイルから抽出するか(「**Extract from flat file**」または「**Extract from XLS file**」)、あるいはデータベースから抽出するか(「**Extract from database**」)を決定する必要があります。

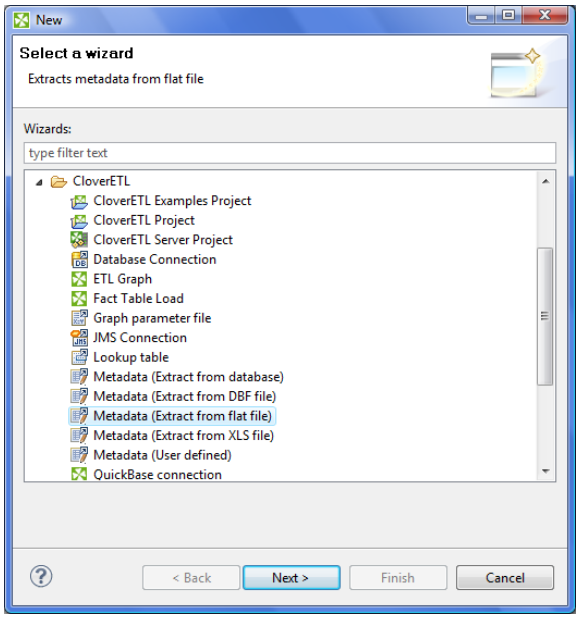


図21.5. メイン・メニュー/「Navigator」ペインでの外部(共有)メタデータの作成

外部(共有)メタデータのリンク

作成後(前の項を参照)、外部(共有)メタデータは使用される各グラフにリンクされる必要があります。「**Metadata**」グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから「**New metadata**」→「**Link shared definition**」を選択する必要があります。その後、プロジェクト・コンテンツが表示された**ファイル選択**ウィザードが開きます。このウィザードで`meta`フォルダを展開し、このウィザードに含まれているすべてのファイルから目的のメタデータ・ファイルを選択する必要があります。

複数の外部(共有)メタデータ・ファイルを一度にリンクすることもできます。このことを行うには、「**Metadata**」グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから「**New metadata**」→「**Link shared**

definition」を選択します。その後、プロジェクト・コンテンツが表示された**ファイル選択**ウィザードが開きます。このウィザードでmetaフォルダを展開し、このウィザードに含まれているすべてのファイルから目的のメタデータ・ファイルを選択する必要があります。**[Shift]**を押しながら**下矢印**または**上矢印**キーで移動すると、隣接するファイル項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各ファイル項目を**クリック**します。

外部(共有)メタデータの内部化

外部(共有)メタデータを作成してリンクした後に、それらをグラフに含める場合は、内部メタデータに変換する必要があります。このような場合、グラフ自体にその構造が表示されます。

任意のリンク済外部(共有)メタデータ・ファイルを内部化するには、「**Outline**」ペインでリンク済外部(共有)メタデータ項目を右クリックし、コンテキスト・メニューから「**Internalize metadata**」をクリックします。

複数のリンク済外部(共有)メタデータ・ファイルを一度に内部化することもできます。このことを行うには、「**Outline**」ペインで目的の外部(共有)メタデータ項目を選択します。**[Shift]**を押しながら**下矢印**または**上矢印**キーで移動すると、隣接する項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各項目を**クリック**します。

その後、選択したリンク済外部(共有)メタデータ項目は「**Outline**」ペインの「**Metadata**」グループからなくなり、同じ場所に、新しく作成された内部メタデータ項目が表示されます。

元の外部(共有)メタデータ・ファイルは、metaサブフォルダに依然として存在し、「**Navigator**」ペインで表示できます。

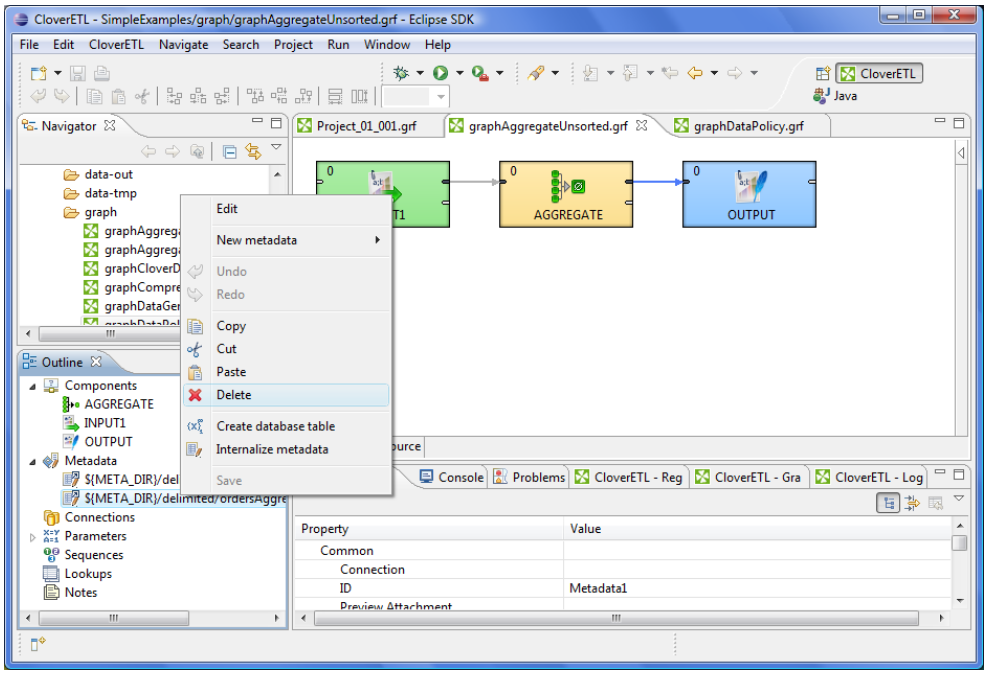


図21.6. 外部(共有)メタデータの内部化

メタデータの作成

前述のように、メタデータはデータの構造を示します。

データ自体は、フラット・ファイル、XLSファイル、DBFファイル、XMLファイルまたはデータベース表に含まれます。これらのデータ・ソースごとに異なる方法でメタデータを抽出または作成する必要があります。また、メタデータを手動で作成することもできます。

次の各説明は、内部メタデータと外部(共有)メタデータの両方に有効です。

フラット・ファイルからのメタデータの抽出

フラット・ファイルからメタデータを抽出して作成する場合は、最初に「**Extract from flat file**」をクリックします。その後、**フラット・ファイル**・ウィザードが開きます。

このウィザードで、ファイル名を入力するか、「**Browse...**」ボタンを使用して検索します。ファイルを選択すると、「**Encoding**」オプションと「**Record type**」オプションも指定できます。デフォルトの**エンコーディング**は`ISO-8859-1`で、デフォルトの**レコード・タイプ**は`delimited`です。

レコードのフィールドがデリミタによって互いに区切られている場合は、「**Record type**」オプションとしてデフォルトの「**Delimited**」を受け入れます。フィールドのサイズが定義されている場合は、「**Fixed Length**」オプションに切り替える必要があります。

ファイルを選択すると、そのコンテンツが「**Input file**」ペインに表示されます。次を参照してください。

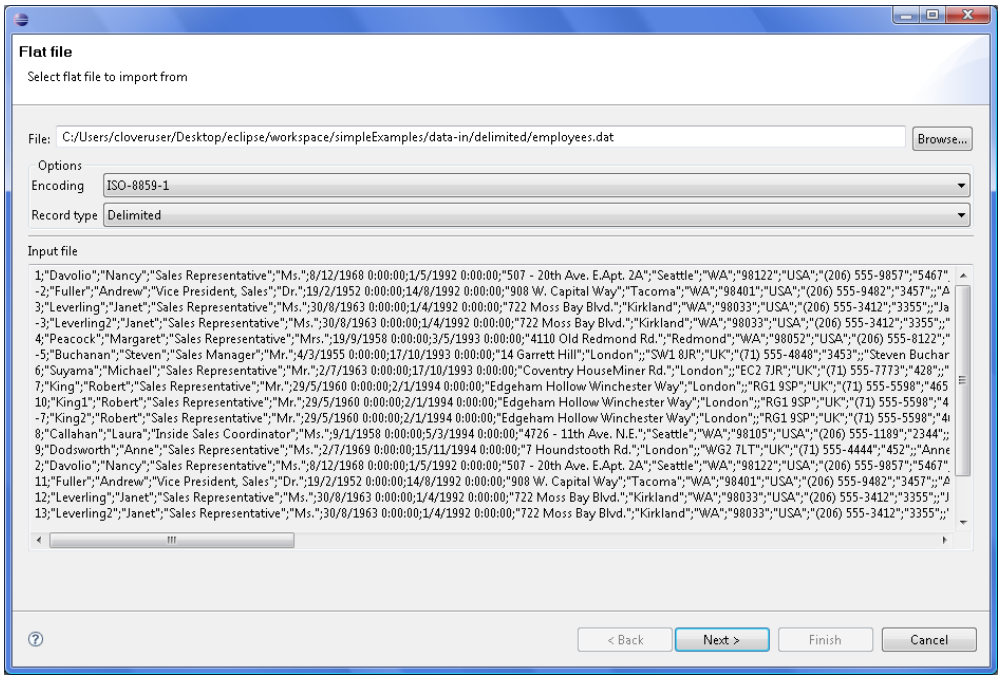


図21.7. デリミタ付きフラット・ファイルからのメタデータの抽出

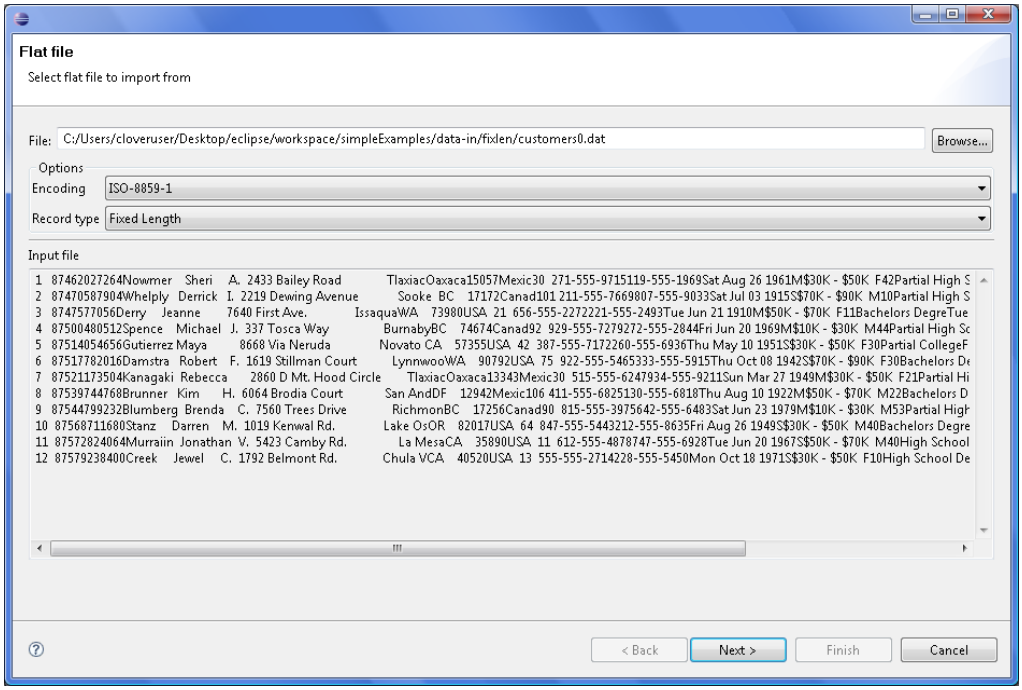


図21.8. 固定長フラット・ファイルからのメタデータの抽出

「**Next**」をクリックすると、入力ファイルのコンテンツおよびデリミタに関する詳細が「**Metadata**」ダイアログに表示されます。これは、4つのペインで構成されます。最初の2つはウィンドウの上部、3つ目は中央、4つ目は下部にあります。各ペインは、その右上隅にある対応する記号をクリックして、全画面に展開できます。

上部にある最初の2つのペインは、メタデータ・エディタ(p.158)で説明しているペインです。メタデータを設定する場合は、言及した項で詳細に説明している方法で実行できます。ペインの右上隅の記号をクリックすると、2つのペインが全画面に展開されます。左と右のペインはそれぞれ「**Record**」ペインおよび「**Details**」ペインと呼ばれます。「**Record**」ペインには、フィールドの**デリミタ**(デリミタ付きメタデータの場合)または**サイズ**(固定長メタデータの場合)、あるいはその両方(混合メタデータの場合のみ)が表示されます。

「**Record**」ペインの任意のフィールドをクリックすると、選択したフィールドに関する詳細情報、またはレコード全体が「**Details**」ペインに表示されます。

プロパティにはデフォルト値があるものとないものがあります。

このペインで、**基本的な**プロパティ(フィールドの**名前**、フィールドの**タイプ**、フィールドの後ろの**デリミタ**、フィールドの**サイズ**、「**Nullable**」、フィールドの**デフォルト**値、「**Skip source rows**」、「**Description**」)および**拡張**プロパティ(「**Format**」、「**Locale**」、「**Autofilling**」、「**Shift**」、「**EOF as delimiter**」)を表示できます。メタデータ構造の変更方法の詳細は、メタデータ・エディタ(p.158)を参照してください。

3つ目のペインの一部のメタデータ設定は変更できます。ファイルの最初の行にレコード・フィールドの名前を含めるかどうかを指定できます。その場合は、「**Extract names**」チェック・ボックスを選択する必要があります。必要な場合は、列ヘッダーをクリックして、フィールドの名前を変更するかどうか(「**Rename**」)、またはフィールドのデータ型を変更するかどうか(「**Retype**」)を決定することもできます。ファイルにフィールド名がな**い場合、CloverETL Designer**によってフィールドのデフォルト名として**Field#**という名前が付けられます。デフォルトで、すべてのレコード・フィールドのタイプは`string`に設定されます。このデータ型をその他の任意のタイプに変更する場合は、提供されるリストから適切なオプションを選択します。これらのオプションには、`boolean`、`byte`、`cbyte`、`date`、`decimal`、`integer`、`long`、`number`、`string`があります。詳細は、データ型およびレコード・タイプ(p.111)を参照してください。

3つ目のペインは、**デリミタ付き**ファイルと**固定長**ファイルとで異なります。次を参照してください。

- デリミタ付きファイルからのメタデータの抽出(p.141)
- 固定長ファイルからのメタデータの抽出(p.143)

ウィザードの下部では、4つ目のペインにファイルのコンテンツが表示されます。

内部メタデータを作成する場合は、「**Finish**」ボタンをクリックするのみです。外部(共有)メタデータを作成する場合は、表示されている「**Next**」ボタンをクリックし、フォルダ(`meta`)およびメタデータの名前を選択し、「**Finish**」をクリックする必要があります。拡張子`.fmt`がメタデータ・ファイルに自動的に追加されます。

デリミタ付きファイルからのメタデータの抽出

中央のペインをウィザード全画面に展開すると、次のように表示されます。

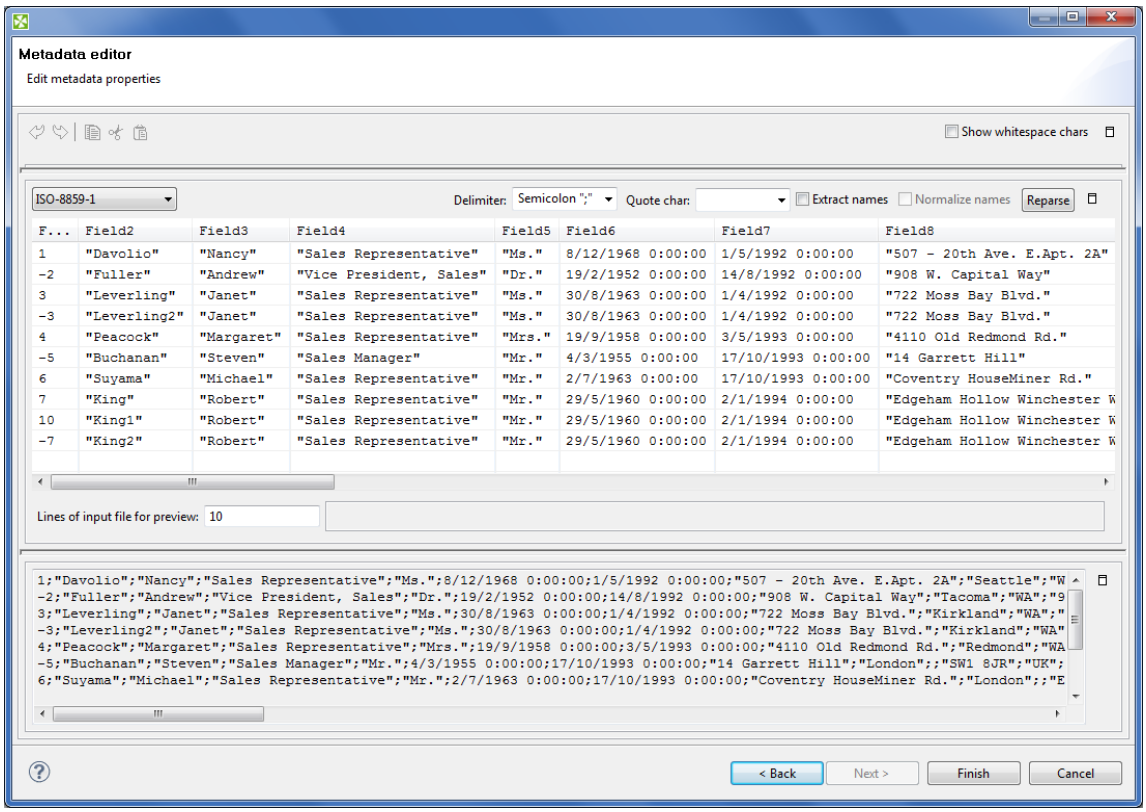


図21.9. デリミタ付きメタデータの設定

ファイルで使用するデリミタを指定する必要がある場合があります(「**Delimiter**」)。デリミタは、カンマ、コロン、セミコロン、スペース、タブまたは文字列にできます。適切なオプションを選択する必要があります。

最後に、「**Reparse**」ボタンをクリックすると、解析されたファイルが下のペインに表示されます。

「**Normalize names**」オプションを使用して、フィールド内の無効な文字を取り除くことができます。これらは、アンダースコア文字、つまり_に置き換えられます。このことは、「**Extract names**」が選択されている場合にのみ有効です。

または、「**Quote char**」コンボ・ボックスを使用して、文字列フィールドから削除する必要がある引用符の種類を選択します。「"」または「'」、あるいは「"」と「'」の両方のうち、いずれかのオプションを選択した後に、必ず「**Reparse**」をクリックしてください。引用符はペアを形成する必要があるため、1種類の引用符を選択すると、もう一方の引用符は無視されます(たとえば、「"」を選択した場合、これらが各フィールドから削除される一方で、すべての'文字は一般的な文字列として扱われます)。実際の引用符文字をフィールドで保持する必要がある場合は、エスケープする、つまり""とする必要があります(単一の"として抽出されます)。引用符で囲まれたデリ

ミタ(**「Delimiter」**で選択)は、無視されます。さらに、パイプ、つまり、|のみ(引用符で囲まない)を入力するなど、独自のデリミタを単一文字としてコンボ・ボックスに入力できます。

例:

`"person"`: `person`として抽出されます(**「Quote char」**を「"」、または**「"」と「'」の両方**に設定)。

`"address"1`: 抽出されず、フィールドにエラーが表示されます。これは、デリミタが引用符の直後に存在する必要があるためです(`"address";`の場合は、デリミタとして;があるので問題ありません)。

`first"Name"`: `first"Name"`として抽出されます。フィールドの先頭に引用符がない場合、フィールド全体が一般的な文字列とみなされます。

`"'doubleQuotes'"` (**「Quote char」**を「"」、または**「"」と「'」の両方**に設定): `'doubleQuotes'`として抽出されます。これは、外側の引用符のみが常に削除され、残りのフィールドはそのままになるためです。

`"unpaired`: 引用符はペアである必要があるため抽出されず、エラーになります。

`'delimiter;'` (**「Quote char」**を「'」、または**「"」と「'」の両方**に設定、かつ**「Delimiter」**を「;」に設定): 引用符内のデリミタは無視されるため、`delimiter;`として抽出されます。

固定長ファイルからのメタデータの抽出

中央のペインをウィザード全画面に展開すると、次のように表示されます。

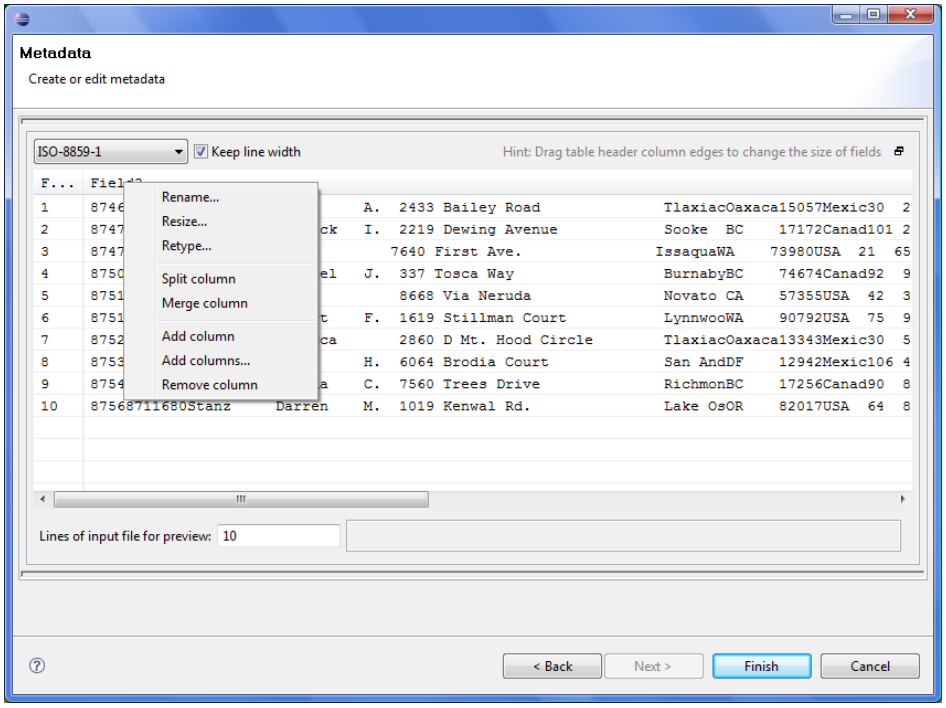


図21.10. 固定長メタデータの設定

各フィールドのサイズを指定する必要があります(「**Resize**」)。また、列の分割、列のマージ、1つ以上の列の追加、列の削除も行う場合があります。列の境界線を移動して、サイズを変更できます。

XLS(X)ファイルからのメタデータの抽出

XLS(X)ファイルからメタデータを抽出する場合は、(「**Outline**」)の「**Metadata**」をクリックし、「**New**」→「**Extract from XLS(X) file**」を選択します。

ヒント

同様に、XLSファイルを「**Navigator**」領域からドラッグし、「**Outline**」の「**Metadata**」にドロップすることもできます。これにより、次に示す抽出ウィザードも表示されます。

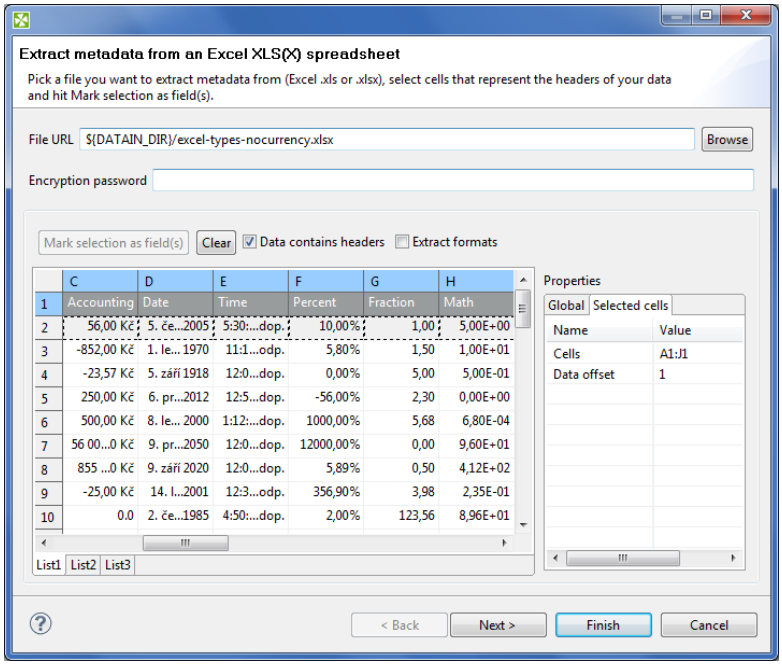
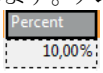


図21.11. Excelスプレッドシートからのメタデータ抽出ウィザード

このウィザードで、次を実行します。

- 目的のXLSファイルを**参照**し、「**OK**」をクリックします。
- ソース・データの方向を決定します。「**Properties**」→「**Global**」→「**Orientation**」で、**垂直**処理(行単位)または**水平**処理(列単位)を切り替えることができます。
- データのヘッダーを表すセルを選択します。このことは、Excelでの通常の操作と同様、Excelの行や列全体をクリックするか、選択領域をクリックしてドラッグするか、[Ctrl]を押しながらセルをクリックするか、または[Shift]を押しながらセルをクリックすることによって実行できます。デフォルトでは、最初の行が選択されます。
- 「**Mark selection as fields**」をクリックします。選択したセルは、色が変わり、これ以降はメタデータ・フィールドとみなされます。変更する場合は、選択したセルをクリックし、「**Clear**」をクリックして、そこからメタデータを抽出しないようにします。
- 各フィールドで、サンプル値を提供するセルを指定する必要があります。これにより、対応するメタデータ型がウィザードによってそこから導出されます。デフォルトでは、下の図のように、マークされたセルのすぐ下の(境界線が破線の)セルが選択されます。この図では、「Percent」がフィールド名になり、「10,00%」によってフィールド・タイプ(この場合は`long`)が決定されます。サンプル値が取得される領域を変更するには、**データ・オフセット**を調整します(詳細は後述します)。



色については、オレンジのセルはヘッダーを形成し、黄色のセルはデータが取得される領域の先頭を示します。

このダイアログで実行できるオプションのタスクを次に示します。

- ソース・ファイルがロックされている場合は**暗号化パスワード**を入力します。正しい大文字/小文字の区別や特殊文字を含め、パスワードは正確に入力してください。
- **データにヘッダーを含める**と、フィールド抽出の対象としてマークされたセルは、ヘッダーとみなされます。データ型とデータ形式は、現在の**データ・オフセット**を考慮して、マークされたセルの下のセルから抽出されます。
- **書式を抽出**します。各フィールドの「**Format**」プロパティは、サンプル・データに応じたパターンで入力されます。この書式パターンは、ウィザードの次の手順、「**Property**」→「**Advanced**」→「**Format**」(「#0.00%」など)で表示されます。詳細は、数値の書式(p.120)を参照してください。

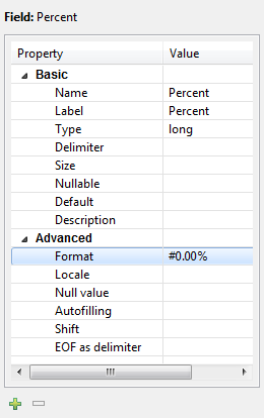


図21.12. スプレッドシートのセルから抽出される書式

警告

メタデータから抽出される書式は、SpreadsheetDataReader(p.404)の「**Format**」**フィールド**とは関係ありません。「**Format**」**フィールド**は、特定のセルのExcel書式を(文字列として)保持する追加のメタデータ・フィールドです。

- **データ・オフセット**を調整します(右側の「**Properties**」ペインの「**Selected cells**」タブ)。メタデータにおいては、データ・オフセットによってデータ型の推測元が決定されます。基本的に、その値は、省略する行(垂直モード)または列(水平モード)の数を表します。デフォルトでは、データ・オフセットは1 (次の行の最初のデータ)です。「**Value**」フィールドのスピナー  をクリックして、データ・オフセットを簡単に調整できます。データ・オフセットの変更による影響はシートのプレビューに表示され、省略された行は色が変わっていることがわかります。

最終手順として、「**OK**」をクリックするか(内部メタデータを作成する場合)、または「**Next**」をクリックして場所(デフォルトでは`meta`)を選択して名前を入力します(外部/共有メタデータを作成する場合)。拡張子`.fmt`がメタデータ名に自動的に追加されます。

データベースからのメタデータの抽出

データベースからメタデータを抽出する(「**Extract from database**」オプションを選択した)場合、メタデータを抽出する前に、データベース接続を定義しておく必要があります。

これ以外に、内部メタデータをデータベースから抽出する場合は、「**Outline**」ペインの接続項目を右クリックして、「**New metadata**」→「**Extract from database**」を選択することもできます。

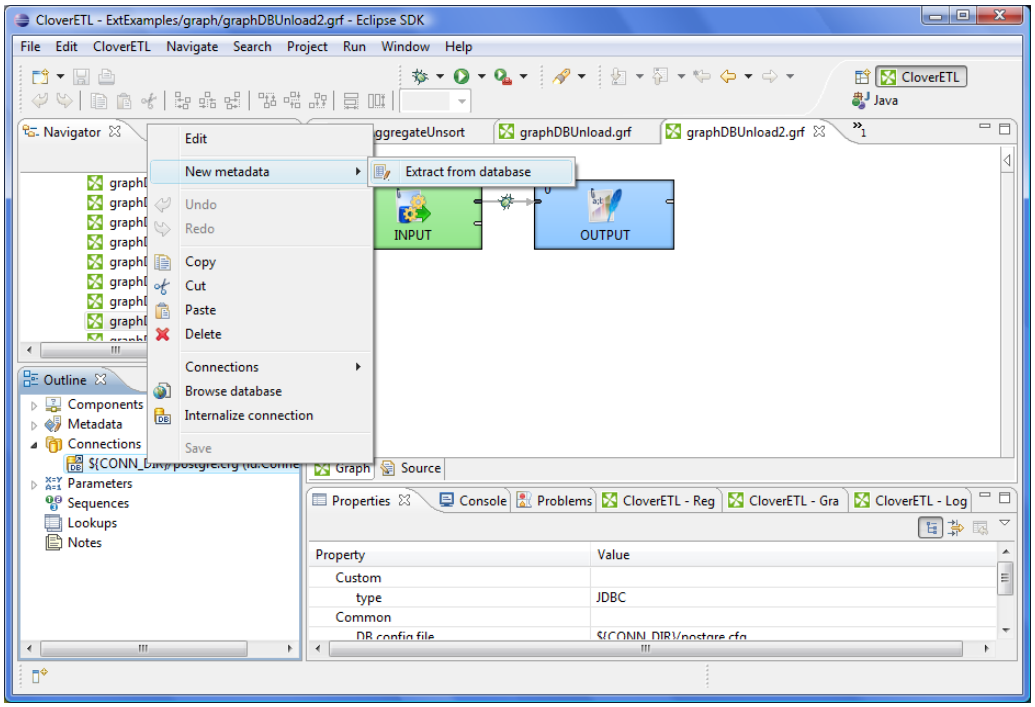


図21.13. データベースからの内部メタデータの抽出

これら3つの各オプションの後に、**データベース接続**ウィザードが開きます。

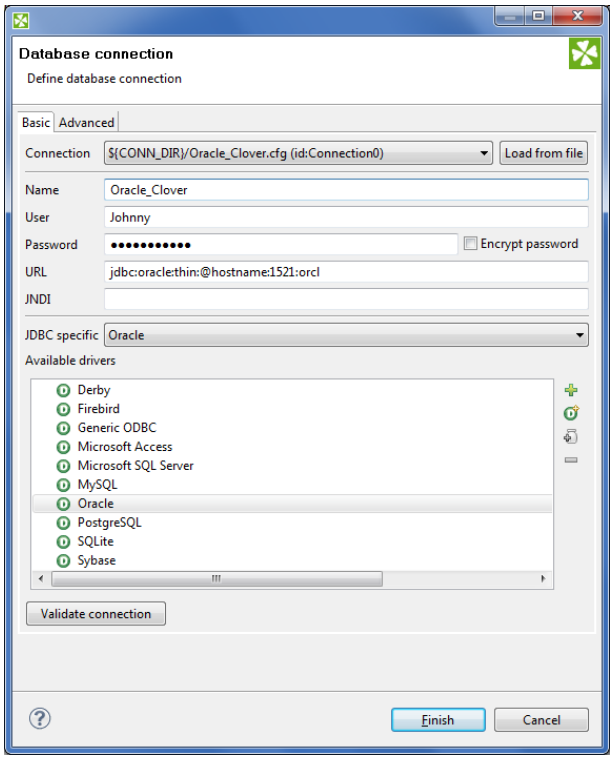


図21.14. データベース接続ウィザード

メタデータを抽出するには、(「**Connection**」メニューを使用して)最初にデータベース接続を既存の接続から選択するか、「**Load from file**」ボタンを使用してデータベース接続をロードするか、または対応する項で示しているように新しい接続を作成する必要があります。定義されると、**データベース接続**ウィザードの**名前**、**ユーザー**、**パスワード**、**URL**または**JNDI** (あるいはその両方)のフィールドが入力された状態になります。

次に、「**Next**」をクリックする必要があります。その後、データベース・スキーマを表示できます。

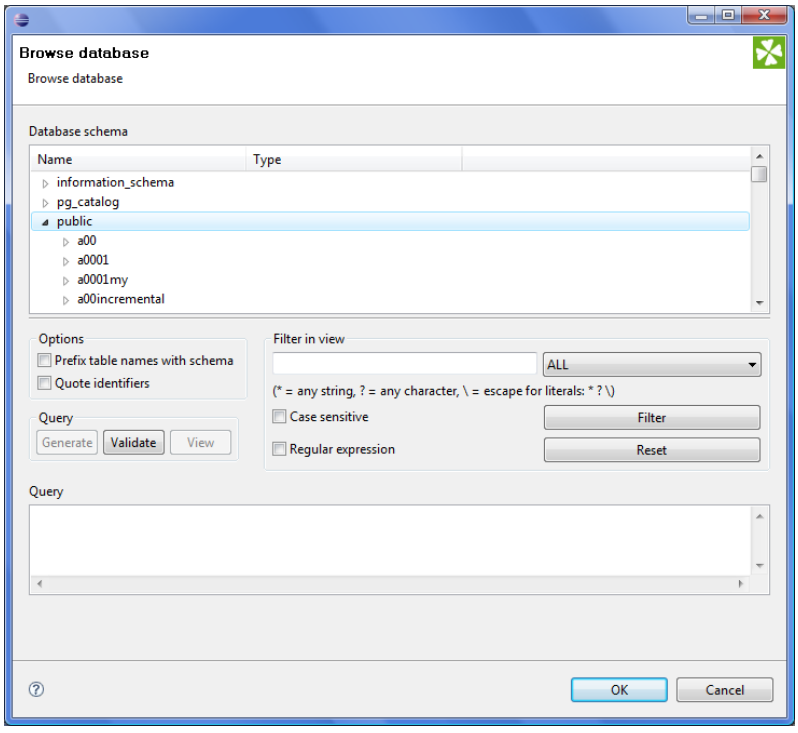


図21.15. メタデータの列の選択

これで、次の2つのことを実行できます。

問合せを直接記述するか、またはデータベース表の個々の列を選択して問合せを生成します。

問合せを生成する場合は、キーボードの**[Ctrl]**を押したまま、マウス・ボタンをクリックして個々の表のそれぞれの列を強調表示し、「**Generate**」ボタンをクリックします。問合せは自動的に生成されます。

次のウィンドウを参照してください。

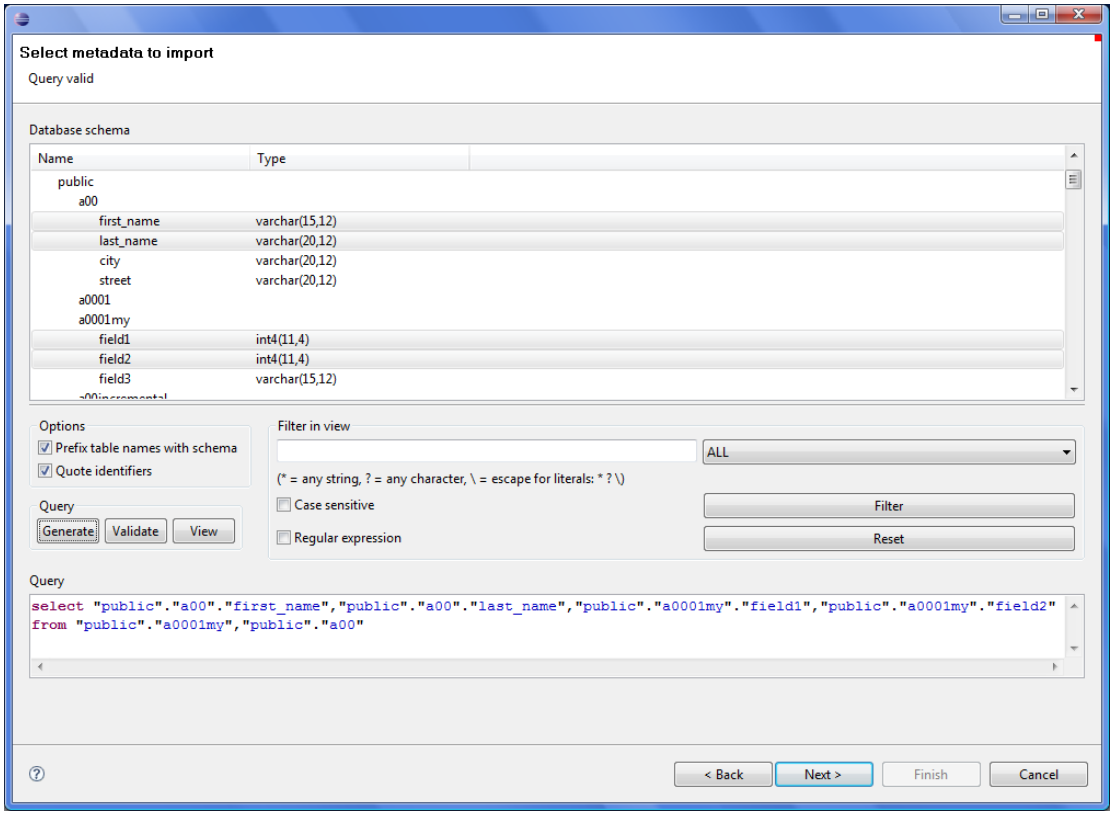


図21.16. 問合せの生成

「**Prefix table names with schema**」チェック・ボックスを選択すると、`schema.table.column`という形式になります。「**Quote identifiers**」チェック・ボックスを選択すると、`"schema"."table"."column"`のようになるか(「**Prefix table names with schema**」が選択されている場合)、または`"table"."column"`のみになります(「**Prefix table names with schema**」チェック・ボックスが選択されていない場合)。また、この問合せはデフォルトの(**汎用**) JDBC仕様を使用することによっても生成されます。ただし、引用符は含まれません。

Sybaseでは、スキーマが先頭に付いた別のタイプの問合せがあります。次のようになります。

```
"schema"."dbowner"."table"."column"
```

重要

引用識別子は、データベースによって異なる場合があります。次のようなものがあります。

- **二重引用符**

  DB2、**Informix** (**Informix**の場合、`DELIMIDENT`変数が「`yes`」に設定されている必要があり、そうでないと引用識別子は使用されません)、**Oracle**、**PostgreSQL**、**SQLite**、**Sybase**

- **バッククォート**

  Infobright

- **バックスラッシュとバッククォート**

  MySQL (バッククォートはインラインCTLの特殊文字として使用されます)

- **大カッコ**

  MSSQL 2008、**MSSQL 2000-2005**

- **引用符なし**

  デフォルト(**汎用**) JDBC仕様または**Derby**仕様が、対応するデータベースに選択されている場合、生成される問合せには引用符が使用されません。

問合せを記述または生成した後に、「**Validate**」ボタンをクリックしてその妥当性をチェックできます。

次に、「**Next**」をクリックする必要があります。その後、**メタデータ・エディタ**が開きます。ここで、メタデータの抽出を終了する必要があります。元のデータベース・フィールド長の制約(特に、`strings/varchars`に対する)を保存する必要がある場合は、**固定長**または**混合**レコード・タイプを選択します。そのようなメタデータは、データベースに表が作成(生成)されるときに使用される正確なデータベース・フィールド定義を提供します。メタデータからのデータベース表の作成に関する項(p.155)を参照してください。

- 「**Finish**」ボタンをクリックして(内部メタデータの場合)、内部メタデータを「**Outline**」ペインで取得します。
- 一方、外部(共有)メタデータを抽出する場合は、最初に「**Next**」ボタンをクリックする必要があり、クリックすると、今後のメタデータ・ファイルを格納するプロジェクトおよびサブフォルダを決定するよう求められます。プロジェクトを展開し、`meta`サブフォルダを選択し、メタデータ・ファイルの名前を指定し、「**Finish**」をクリックすると、選択した場所に保存されます。

DBaseファイルからのメタデータの抽出

DBaseファイルからメタデータを抽出する場合は、「**Extract from DBF file**」オプションを選択する必要があります。

メタデータの抽出元となるファイルを検索します。ファイルは次のエディタで開きます。

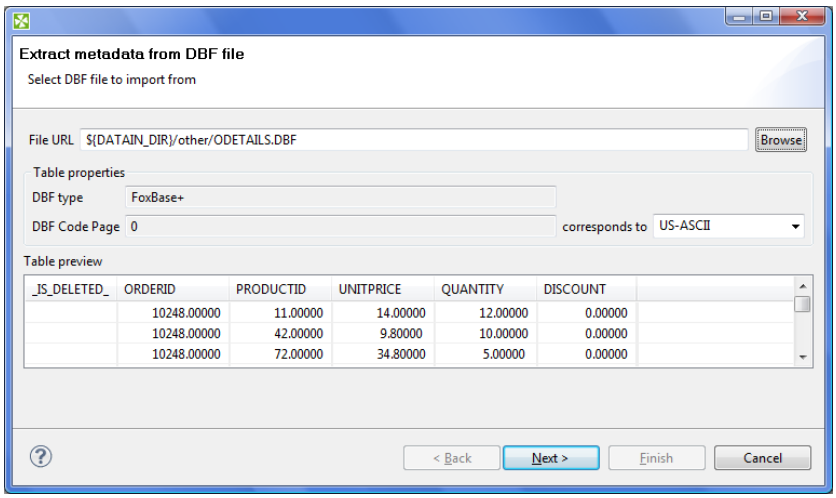


図21.17. DBFメタデータ・エディタ

DBFタイプ、**DBFコード・ページ**が自動的に選択されます。必要な内容と異なる場合は、これらの値を変更します。

「**Next**」をクリックすると、抽出されたメタデータが含まれた***メタデータ・エディタ***が開きます。デフォルトのメタデータ値およびメタデータ型を維持したまま「**Finish**」をクリックできます。

ユーザーによるメタデータの作成

自分でメタデータを作成する場合(「**User defined**」)は、次の方法で行う必要があります。

メタデータ・エディタを開いた後、プラス記号をクリックして必要な数のフィールドを追加し、それらの名前、データ型、デリミタ、サイズ、書式、および前述のすべてを設定する必要があります。

詳細は、メタデータ・エディタ(p.158)を参照してください。

このことをすべて実行した後、内部メタデータの場合は「**OK**」をクリックし、外部(共有)メタデータの場合は「**Next**」をクリックします。後者の場合は、メタデータ・ファイルの場所(デフォルトは`meta`)および名前の選択のみが必要となります。「**OK**」をクリックすると、メタデータ・ファイルが保存され、拡張子`.fmt`がファイルに自動的に追加されます。

Lotus Notesからのメタデータの抽出

Lotus Notesコンポーネントの場合(詳細は、LotusReader(p.390)、LotusWriter(p.511)を参照)、処理するLotusデータのメタデータを指定する必要があります。**LotusReader**コンポーネントでは、Lotusビューからデータを正しく読み取るためにメタデータが必要となります。メタデータによって、1つのビューに含まれる列の数が示され、名前およびタイプが列に割り当てられます。**LotusWriter**コンポーネントは、メタデータを使用して、書き込まれるデータ・フィールドのタイプを特定します。

メタデータは、内部メタデータまたは外部メタデータとしてLotusビューから取得されます。内部メタデータと外部メタデータの作成方法は、内部メタデータ(p.134)および外部(共有)メタデータ(p.137)を参照してください。

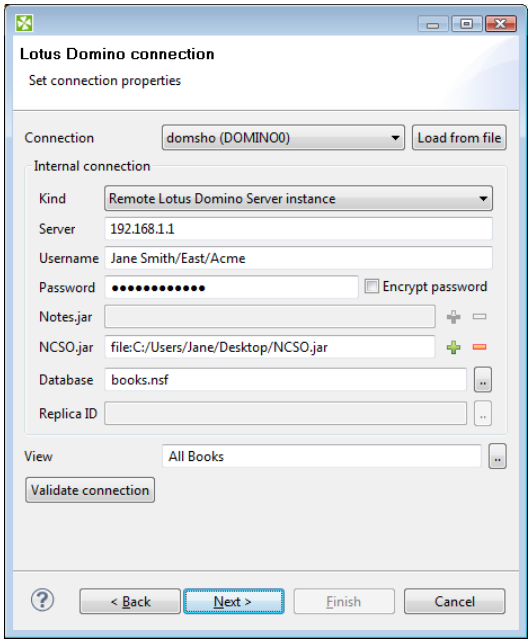


図21.18. メタデータ抽出のためのLotus Notes接続の指定

Lotus Notesメタデータ抽出ウィザードの最初のページで、Lotus NotesまたはLotus Dominoサーバーへの接続の詳細を提供するよう求められます。既存のLotus接続を選択するか、「**Load from file**」ボタンを使用して外部接続をロードするか、または**接続**メニューから「**<custom>**」を選択して新しい接続を定義できます。

接続の詳細は、第25章「Lotus接続」(p.191)を参照してください。

最後に、メタデータを抽出できるようにするために、メタデータの抽出元となる**ビュー**を指定する必要があります。

抽出プロセスによって、選択したビューの列の数と同じ数のフィールドを持つメタデータが準備されます。また、「View」列の名前に基づいて、フィールドに名前も割り当てられます。Lotusビューのすべての列には(プログラム的な)内部名があります。読みやすくするため、一部の列にはユーザー定義の名前を付けることができます。可能な場合、抽出ウィザードではユーザー定義の名前が使用され、その他の場合は、プログラム的な内部の列名が使用されます。

メタデータ抽出プロセスにより、すべてのフィールドのタイプが**文字列**に設定されます。これは、Lotusの「View」列にタイプが割り当てられていないためです。列の値には、(たとえば、特定の条件や複雑な計算の結果に基づいた)任意のタイプを設定できます。

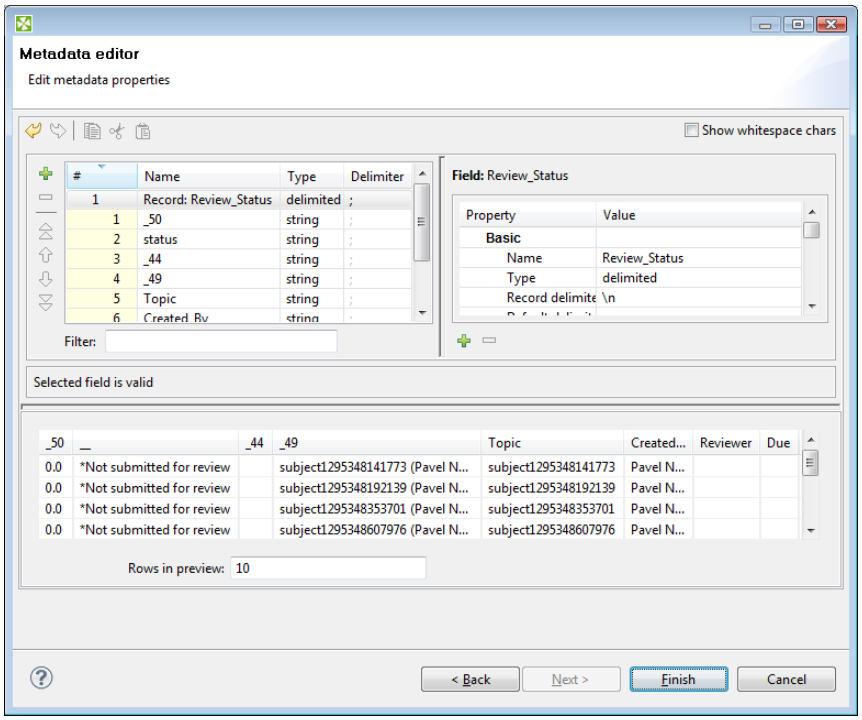


図21.19. Lotus Notesメタデータ抽出ウィザード、ページ2

Lotus Notesメタデータ抽出ウィザードの2ページ目は2つの部分に分かれています。上の部分には、メタデータ抽出の結果をカスタマイズするために使用できる標準メタデータ・エディタがあります。下の部分には、ビューに含まれているデータのプレビューが表示されます。

このページで、たとえば、フィールドの名前を変更したり、フィールドのタイプをデフォルトの文字列タイプから他の特定のタイプに変更できます。**この場合、受信データが選択したデータ型に確実に変換できることを確認する必要があります。LotusReader**コンポーネントは、Lotusデータを文字列に常に正常に変換します。ただし、無効な変換が試行された場合は失敗する可能性があります。たとえば、整数を日付データ型に変換しようとすると、データ変換例外が発生し、読取りプロセス全体が失敗します。

内部メタデータを抽出する場合は、このページがLotus Notesメタデータ抽出ウィザードの最後のページになります。「**Finish**」をクリックすると、内部メタデータが現在開かれているグラフに追加されます。外部メタデータを抽出する場合は、次のページで、抽出されたメタデータを格納する場所の指定を求められます。

既存のメタデータのマージ

2つ以上の既存のメタデータを1つの新しいメタデータ・オブジェクトにマージすることによって、新しいメタデータを作成できます。フィールドおよびその設定は、選択したソースから新しいメタデータにコピーされます。

競合するフィールド名は、次のいずれかの方法で解決されます。

- 自動。2つのオプションがあり、最初のフィールドのみが取得されるか、または重複した名前が変更されます(`field_1`、`field_2`など。)
- 手動。このウィザードの2番目の手順です。

「**Merge metadata**」ダイアログで、結果に含めるメタデータおよびフィールドを選択します。ダイアログを起動するには、次を実行します。

1. 「**Outline**」の2つ以上の既存のメタデータを右クリックします。

   または

   「**Metadata**」→「**New Metadata**」→「**Merge existing**」

または

エッジを右クリックして、「**New metadata**」をクリックします。

2. 「**Merge metadata...**」(**既存のマージ**)をクリックします。
3. 2つの手順が含まれるウィザードで続行します。最初の手順では、選択したメタデータのすべてのフィールドを管理します。最終的なマージに含める必要があるもののみを選択します(これらは太字で強調表示されます)。

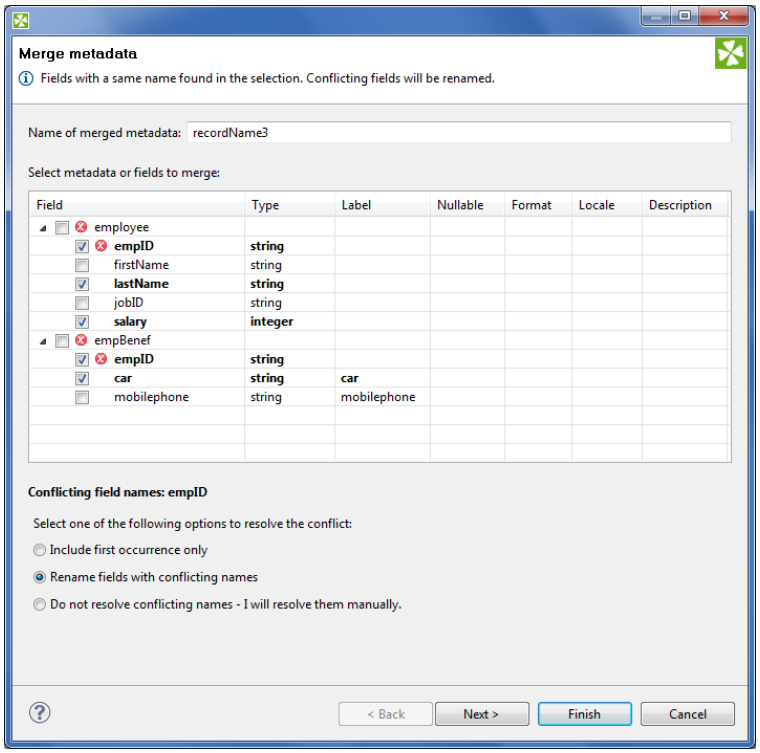


図21.20. 2つのメタデータのマージ: 競合は3つの方法(下部のラジオ・ボタン)のうちいずれかで解決できます。

4. 「**Next**」をクリックして、マージされたメタデータを確認するか、または「**Finish**」をクリックして、ただちに作成します。

動的メタデータ

CloverETL Designerを使用して作成または抽出されたメタデータに加えて、グラフ・エディタ・ペインの「**Source**」タブでメタデータ定義を記述することもできます。**CloverETL Designer**で定義されたメタデータとは異なり、「**Source**」タブで記述されたメタデータは**CloverETL Designer**で編集できません。

メタデータを「**Source**」タブで定義するには、このタブを開いて次のように記述します。

```
<Metadata id="YourMetadataId" connection="YourConnectionToDB"
sqlQuery="YourQuery"/>
```

`YourMetadataId`に一意の式(DynamicMetadata1など)を指定し、DBへの接続に使用する必要がある、前に作成したDB接続のIDを`YourConnectionToDB`として指定します。DBからのメタデータの抽出に使用する問合せを`YourQuery`として入力します(select * from myTableなど)。

メタデータ抽出を高速化するために、句`"where 1=0"`または`"and 1=0"`を問合せに追加します。前者の句はwhere条件なしで問合せに追加され、後者の句は`"where ..."`式がすでに含まれている問合せに追加される必要があります。この方法では、メタデータのみが抽出され、データは読み取られません。

このようなメタデータは、実行時にのみ動的に生成されます。そのフィールドは、**CloverETL Designer**のメタデータ・エディタで表示または変更することはできません。

注意

動的メタデータを使用する場合は、checkConfigメソッドをスキップすることをお薦めします。このことを行うには、-skipcheckconfigをプログラム引数に追加します。プログラムとVM引数(p.85)を参照してください。

特殊なソースからのメタデータの読取り

前の項で説明した動的メタデータと同様の方法で、別のメタデータ定義を**グラフ・エディタ・**ペインの「**Source**」タブで使用することもできます。

これらのすべてのメタデータは**CloverETL Designer**で編集できません。

グラフのソース・コードで外部(共有)メタデータを定義する最も簡単な形式(fileURL="${META_DIR}/metadatafile.fmt")に加えて、他の外部(共有)メタデータへのパスも定義する、より複雑なURLを「**Source**」タブで使用できます。

例:

```
<Metadata fileURL="zip:(${META_DIR}\delimited.zip)#delimited/employees.fmt"
id="Metadata0"/>
```

または:

```
<Metadata fileURL="ftp://guest:guest@localhost:21/employees.fmt"
id="Metadata0"/>
```

このような式では、外部(共有)メタデータのロード元およびグラフのリンク先となるソースを指定できます。

メタデータおよびデータベース接続からのデータベース表の作成

最後のオプションとして、メタデータ(内部および外部の両方)に基づいてデータベース表を作成することもできます。

「**Outline**」ペインや**グラフ・エディタ**から呼び出される2つのコンテキスト・メニューのそれぞれから「**Create database table**」項目を選択すると、データベース表を作成できるSQL問合せが含まれたウィザードが開きます。

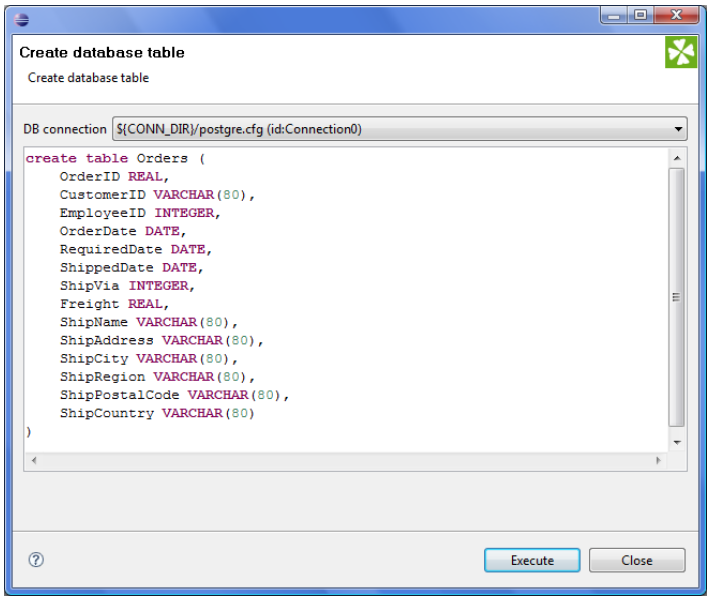


図21.21. メタデータおよびデータベース接続からのデータベース表の作成

必要な場合は、このウィンドウの内容を編集できます。

データベースへの接続を選択する場合があります。詳細は、第22章「データベース接続」(p.172)を参照してください。そのようなデータベース表が作成されます。

注意

複数のSQLタイプがリストされる場合、実際の構文は特定のメタデータ(固定長フィールドのサイズ、長さ、スケールなど)によって異なります。

表21.15. CloverETLからSQLデータ型への変換表(パートI)

DBタイプ	DB2とDerby	Firebird	Hive	Informix	MSAccess
Cloverタイプ					
boolean	SMALLINT	CHAR(1)	BOOLEAN	BOOLEAN	BIT
byte	VARCHAR(80) FOR BIT DATA	CHAR(80)	BINARY[a]	BYTE	VARBINARY(80)
	CHAR(n) FOR BIT DATA	CHAR(n)			BINARY(n)
cbyte	VARCHAR(80) FOR BIT DATA	CHAR(80)	BINARY[a]	BYTE	VARBINARY(80)
	CHAR(n) FOR BIT DATA	CHAR(n)			BINARY(n)
date	TIMESTAMP	TIMESTAMP	TIMESTAMP[a]	DATETIME YEAR TO SECOND	DATETIME
	DATE			DATE	DATE
	TIME			DATETIME HOUR TO SECOND	TIME
decimal	DECIMAL	DECIMAL	DECIMAL[b]	DECIMAL	DECIMAL
	DECIMAL(p)	DECIMAL(p)		DECIMAL(p)	DECIMAL(p)
	DECIMAL(p,s)	DECIMAL(p,s)		DECIMAL(p,s)	DECIMAL(p,s)
integer	INTEGER	INTEGER	INT	INTEGER	INT
long	BIGINT	BIGINT	BIGINT	INT8	BIGINT
number	DOUBLE	FLOAT	DOUBLE	FLOAT	FLOAT
string	VARCHAR(80)	VARCHAR(80)	STRING	VARCHAR(80)	VARCHAR(80)
	CHAR(n)	CHAR(n)		CHAR(n)	CHAR(n)

[a] Hiveのバージョン0.8.0から使用可能

[b] Hiveのバージョン0.11.0から使用可能

表21.16. CloverETLからSQLデータ型への変換表(パートII)

DBタイプ	MSSQL	MSSQL	MySQL	Oracle	Pervasive
Cloverタイプ	2000-2005	2008			
boolean	BIT	BIT	TINYINT(1)	SMALLINT	BIT
byte	VARBINARY(80)	VARBINARY(80)	VARBINARY(80)	RAW(80)	LONGVARBINARY(80)
	BINARY(n)	BINARY(n)	BINARY(n)	RAW(n)	BINARY(n)
cbyte	VARBINARY(80)	VARBINARY(80)	VARBINARY(80)	RAW(80)	LONGVARBINARY(80)
	BINARY(n)	BINARY(n)	BINARY(n)	RAW(n)	BINARY(n)

DBタイプ	MSSQL	MSSQL	MySQL	Oracle	Pervasive
Cloverタイプ	2000-2005	2008			
date	DATETIME	DATETIME	DATETIME	TIMESTAMP	TIMESTAMP
		DATE	YEAR	DATE	DATE
		TIME	DATE		TIME
			TIME		
decimal	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL
	DECIMAL(p)	DECIMAL(p)	DECIMAL(p)	DECIMAL(p)	DECIMAL(p)
	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
integer	INT	INT	INT	INTEGER	INTEGER
long	BIGINT	BIGINT	BIGINT	NUMBER(11,0)	BIGINT
number	FLOAT	FLOAT	DOUBLE	FLOAT	DOUBLE
string	VARCHAR(80)	VARCHAR(80)	VARCHAR(80)	VARCHAR2(80)	VARCHAR2(80)
	CHAR(n)	CHAR(n)	CHAR(n)	CHAR(n)	CHAR(n)

表21.17. CloverETLからSQLデータ型への変換表(パートIII)

DBタイプ	PostgreSQL	SQLite	Sybase	Generic
Cloverタイプ				
boolean	BOOLEAN	BOOLEAN	BIT	BOOLEAN
byte	BYTEA	VARBINARY(80)	VARBINARY(80)	VARBINARY(80)
		VARBINARY(80)	BINARY(n)	BINARY(n)
cbyte	BYTEA	VARBINARY(80)	VARBINARY(80)	VARBINARY(80)
		BINARY(n)	BINARY(n)	BINARY(n)
date	TIMESTAMP	TIMESTAMP	DATETIME	TIMESTAMP
	DATE	DATE	DATE	DATE
	TIME	TIME	TIME	TIME
decimal	NUMERIC	DECIMAL	DECIMAL	DECIMAL
	NUMERIC(p)	DECIMAL(p)	DECIMAL(p)	DECIMAL(p)
	NUMERIC(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
integer	INTEGER	INTEGER	INT	INTEGER
long	BIGINT	BIGINT	BIGINT	BIGINT
number	REAL	NUMERIC	FLOAT	FLOAT
string	VARCHAR(80)	VARCHAR(80)	VARCHAR(80)	VARCHAR(80)
	CHAR(n)	CHAR(n)	CHAR(n)	CHAR(n)

改訂日: 2013年2月18日

メタデータ・エディタ

*メタデータ・エディタ*はメタデータを編集するためのビジュアル・ツールです。

メタデータ・エディタを開く

*メタデータ・エディタ*は、メタデータをフラット・ファイル(2つの上部ペイン)やデータベースから作成するとき、または手動で作成するときに開きます。

また、*メタデータ・エディタ*を開いて、既存のメタデータを編集することもできます。

- エッジに割り当てられているメタデータ(内部および外部の両方)を編集する場合は、**グラフ・エディタ**・ペインで次のいずれかの方法を実行します。
  - エッジをダブルクリックします。
  - エッジを選択し、**[Enter]**を押します。
  - エッジを右クリックし、コンテキスト・メニューから**「Edit」**を選択します。
- *メタデータ*(内部および外部の両方)を編集する場合は、**「Outline」**ペインでメタデータ・カテゴリを展開した後に次を実行します。
  - メタデータ項目をダブルクリックします。
  - メタデータ項目を選択し、**[Enter]**を押します。
  - メタデータ項目を右クリックし、コンテキスト・メニューから**「Edit」**を選択します。
- 任意のプロジェクトの外部(共有)メタデータを編集する場合は、**「Navigator」**ペインで**meta**サブフォルダを展開した後に次を実行します。
  - メタデータ・ファイルをダブルクリックします。
  - メタデータ・ファイルを選択し、**[Enter]**を押します。
  - メタデータ・ファイルを右クリックし、コンテキスト・メニューから**「Open With」→「CloverETL Metadata Editor」**を選択します。

メタデータ・エディタの基本

*メタデータ・エディタ*を開く方法をすでに理解していることを想定しています。詳細は、メタデータ・エディタを開く(p.158)を参照してください。

ここでは、*メタデータ・エディタ*の外観について説明します。

このエディタには、次のように、左側のボタン、2つのペインおよび1つのフィルタ・テキスト領域があります。

- ダイアログの左側には、フィールドの追加と削除、および1つ以上のフィールドを最上位、上、下または最下位に移動するためのボタンが(上から下に) 6つあります。これらのボタンの上には、左から右に(元に戻す、および再実行ための) 2つの矢印があります。
- 左のペインは**「Record」**ペインと呼ばれます。

  詳細は、「Record」ペイン(p.160)を参照してください。
- 右のペインは**「Details」**ペインと呼ばれます。

詳細は、「Details」ペイン(p.161)を参照してください。

- 「**Filter**」テキスト領域には、「**Record**」ペインのフィールド間を検索するための式を入力できます。大文字/小文字が区別されます。

「**Record**」ペインでは、レコード全体に関する情報の概要、およびデリミタ、サイズまたはその両方が含まれたレコード・フィールドのリストを表示できます。

「**Details**」ペインの内容は、「**Record**」ペインで選択した行に応じて次のように変わります。

- 最初の行が選択された場合、レコードに関する詳細が「**Details**」ペインに表示されます。

  詳細は、レコードの詳細(p.162)を参照してください。

- 別の行が選択された場合は、選択したフィールドに関する詳細が「**Details**」ペインに表示されます。

  詳細は、フィールドの詳細(p.163)を参照してください。

注意

一部のプロパティのデフォルト値は、グレーのテキストで出力されます。

次に、デリミタ付きメタデータの例、および固定長メタデータの別の例を示します。混合メタデータは、両方の場合の組合せです。フィールド名によって、デリミタが定義されるがサイズは指定されない、サイズが定義されるがデリミタは指定されない、または両方が定義される、などの違いがある場合があります。そのようなメタデータは、手動で作成する必要があります。

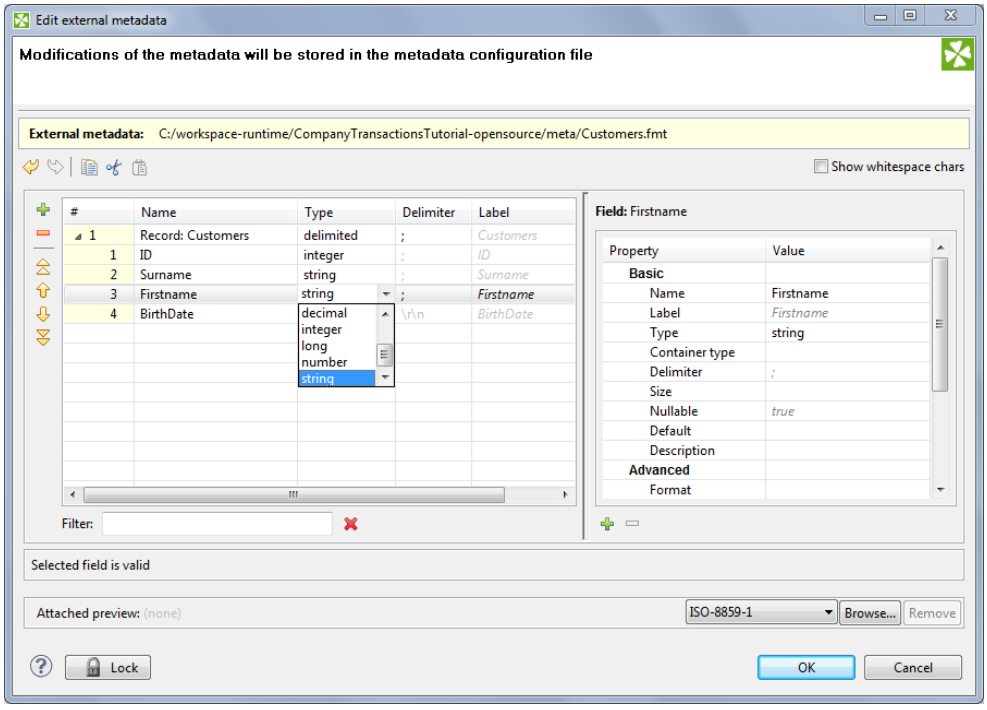


図21.22. デリミタ付きファイルでのメタデータ・エディタ

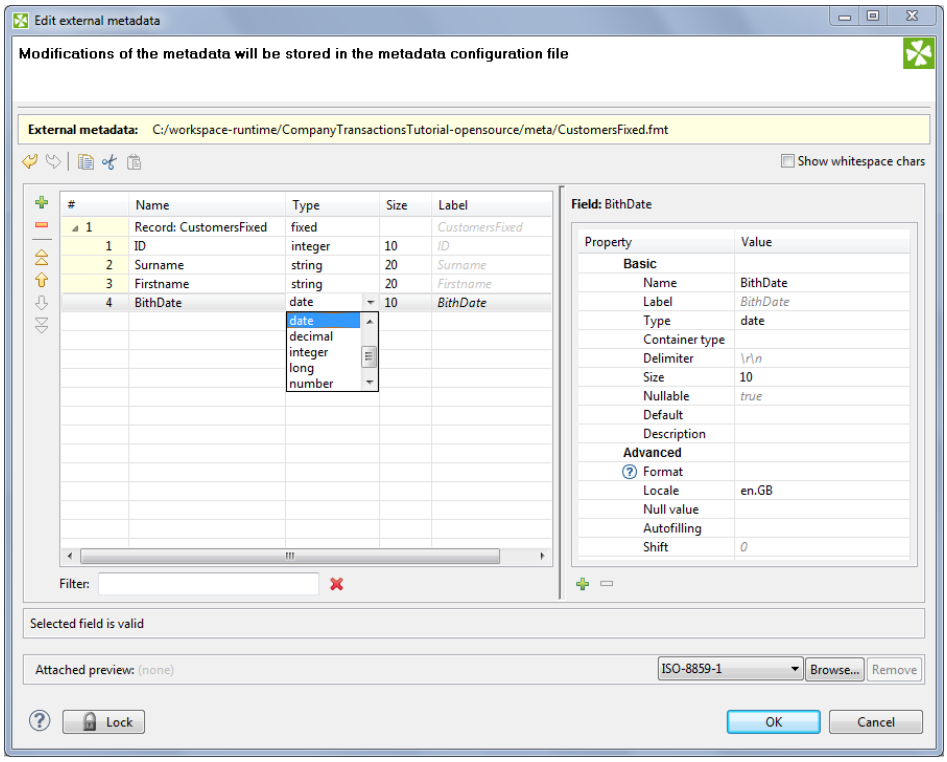


図21.23. 固定長ファイルでのメタデータ・エディタ

トラッキング可能フィールドの選択

Jobflow (p.250)では、選択したフィールドの値を追跡(p.251)できます。次に示すように、**トークン・ボタンが含まれたログ・フィールド**を使用してフィールドを選択できます。

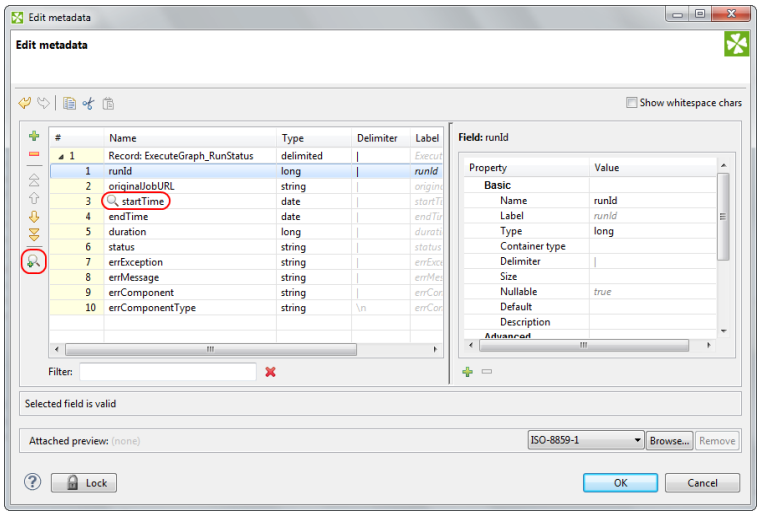


図21.24. メタデータ・エディタでのトラッキング可能フィールドの選択

「Record」ペイン

このペインには、次のようにレコード全体の概要およびそのすべてのフィールドが表示されます。

- 最初の行は、レコード全体の概要を表します。

  次の列で構成されます。

- レコードの名前は2番目の列に表示され、そこで変更できます。
- レコードのタイプは3番目の列に表示され、デリミタ付き、固定または混合として選択できます。
- 他の列もそれぞれ表示されます。たとえば、各フィールドを次のフィールド(最後のフィールドは除く)と区切るデフォルトのデリミタ、またはレコード全体のサイズ(固定長メタデータの場合)があります。
- 最後の列は常にラベルになります。フィールド名に似ていますが、関係する制約はありません。フィールド名とラベルと説明(p.161)を参照してください。

- 最後の行を除く他の行は、次のようにレコード・フィールドのリストを示します。
  - 最初の列には、フィールドの数が表示されます。フィールドは、1から番号付けされます。
  - 2番目の列には、フィールドの名前が表示されます。そこで変更できます。フィールド名には`[a-zA-Z0-9_]`の文字のみを使用することをお薦めします。
  - 3番目の列には、フィールドのデータ型が表示されます。いずれかのデータ型をメタデータに選択できます。詳細は、データ型およびレコード・タイプ(p.111)を参照してください。
  - 他の列には、行に表示されたフィールドの後のデリミタ、フィールドのサイズ、またはデリミタとサイズの両方が表示されます。デリミタがグレーで表示される場合、それはデフォルトのデリミタであり、黒の場合はデフォルト以外のデリミタです。
- 最後の行は、次のように最後のフィールドを表します。
  - 最初の3つの列は、他のフィールド行の列と同じです。
  - 他の列には、最後のフィールドの後のレコード・デリミタ(グレーで表示される場合)または最後のフィールドの後およびレコード・デリミタの前に置かれるデフォルト以外のデリミタ(黒で表示される場合)、フィールドのサイズ、またはデリミタとサイズの両方が表示されます。

デリミタの詳細は、デリミタの変更および定義(p.165)を参照してください。

フィールド名とラベルと説明

この項は、これらの基本的な違いの理解に役立ちます。

フィールド名は、メタデータがファイルから抽出されるなどの場合に使用される内部Clover表記です。フィールド名にはスペース、発音区別、およびアクセントは使用できないという制約があります。

フィールド・ラベルはフィールド名から自動的にコピーされ、制約なしで変更できます(アクセント、発音区別などを使用できます)。さらに、1つのレコード内のラベルが重複してもかまいません。通常、メタデータをCSVファイルなどから抽出する場合、フィールド名はマシン形式で取得されます。その後、任意の文字を使用してわかりやすいラベルに変更できます。最後に、Excelファイルに書き込み、これらのラベルをスプレッドシートのヘッダーにします。(一部のライターでの「**Write field names**」属性は、第54章「ライター」(p.459)を参照してください)

説明は、純粋なコメントです。これを使用して、自分、または自分のメタデータを使用して作業する他のユーザーに対するアドバイスを提供します。出力は生成されません。

「Details」ペイン

「**Details**」ペインの内容は、「**Record**」ペインで選択した行に応じて変わります。

- 最初の行を選択した場合は、レコード全体に関する詳細が表示されます。

  レコードの詳細(p.162)を参照してください。

- 別の行を選択した場合は、選択したフィールドに関する詳細が表示されます。

  フィールドの詳細(p.163)を参照してください。

注意

一部のプロパティのデフォルト値は、グレーのテキストで出力されます。

レコードの詳細

「**Details**」ペインにレコード全体に関する情報が表示されるときには、レコードのプロパティが表示されます。

基本的なプロパティを次に示します。

- 「**Name**」。レコードの名前です。そこで変更できます。
- 「**Type**」。レコードのタイプです。3つ(`delimited`、`fixed`、`mixed`)のいずれかを選択できます。詳細は、レコード・タイプ(p.112)を参照してください。
- 「**Record delimiter**」。これは、最後のフィールドの後のデリミタで、レコードの終了を意味します。そこで変更できます。「**Record**」ペインの最後の行の「**Delimiter**」列のデリミタがグレーで表示される場合、それがレコードのデリミタです。黒の場合は、デフォルト以外のその他のデリミタであり、そのデリミタの後、およびレコード・デリミタの前の最後のフィールドに対して定義されます。

  詳細は、デリミタの変更および定義 (p.165)を参照してください。
- 「**Record size**」。`fixed`または`mixed`レコード・タイプのみが表示されます。これは、文字の数でカウントされる、レコードの長さです。そこで変更できます。
- 「**Default delimiter**」。`delimited`または`mixed`レコード・タイプのみが表示されます。これは、最後のフィールドを除き、デフォルトでレコードの各フィールドの後に続くデリミタです。そこで変更できます。このデリミタは、グレーの場合、「**Record**」ペインの「**Delimiter**」列のそれぞれの行(最後の行は除く)に表示されます。黒の場合は、デフォルト以外のその他のデリミタであり、デフォルトのデリミタをオーバーライドするフィールドに対して定義され、デフォルトのデリミタのかわりに使用されます。

  詳細は、デリミタの変更および定義(p.165)を参照してください。
- 「**Skip source rows**」。これは、各入力ファイルでスキップされるレコードの数です。この属性を持つエッジが**リーダー**に接続されている場合、この値は、「**Number of skipped records per source**」属性のデフォルト値(0)をオーバーライドします。「**Number of skipped records per source**」属性が指定されていない場合は、このレコード数が各入力ファイルからスキップされます。**リーダー**の属性がなんらかの値に設定されている場合、このプロパティ値はその値によってオーバーライドされます。これらの2つの値は合計されません。
- 「**Description**」。このプロパティは、レコードの意味を示します。

拡張プロパティを次に示します。

- 「**Quoted strings**」: 特殊文字(カンマ、改行または二重引用符)を含むフィールドは、引用符で囲む必要があります。一重引用符および二重引用符のみが引用符文字として受け入れられます。「**Quoted strings**」が`true`の場合、特殊文字はデリミタとして扱われず、次のように処理されます。
  - 削除: リーダーによって入力が読み取られるとき。
  - 書出し: 出力フィールドは引用符付き文字列で囲まれます(UniversalDataWriterの属性(p.553)を参照してください)。

  コンポーネントがこの属性を持つ場合(ParallelReader、ComplexDataReader、UniversalDataReader、UniversalDataWriterなど)、その値は入力/出力ポートに関するメタデータ内の「**Quoted strings**」の設定に

従って設定されます。ただし、コンポーネントのtrue/false値は、メタデータ内の値よりも優先度が高く、オーバーライドできます。

例(**ParallelReader**の場合など): 入力データ`"25"|"John"`を読み取るには、「**Quoted strings**」を`true`に切り替え、「**Quote character**」を「"」に設定します。これにより、`25|John`という2つのフィールドが生成されます。

- 「**Quote character**」: 「**Quoted strings**」で使用される引用符の種類を指定します。コンポーネントがこの属性を持つ場合(ParallelReader、ComplexDataReader、UniversalDataReader、UniversalDataWriterなど)、その値は入力/出力ポートに関するメタデータ内の「**Quote character**」の設定に従って設定されます。ただし、コンポーネントの値は、メタデータ内の値よりも優先度が高く、オーバーライドできます。

- 「**Locale**」。これは、レコード全体に対して使用されるロケールです。このプロパティは、日付書式や小数区切りなどの場合に役立ちます。個々のフィールドに指定された**ロケール**によってオーバーライドできます。

  詳細は、ロケール(p.126)を参照してください。

- 「**Locale sensitivity**」。レコード全体に適用されます。(`string`データ型の)個々のフィールドに指定されている**ロケール依存**によってオーバーライドできます。

  詳細は、ロケール依存(p.131)を参照してください。

- 「**Null value**」。このプロパティはレコード全体に設定されます。`null`として処理されるフィールドの値を指定するために使用されます。デフォルトでは、空のフィールドまたは空の文字列(`""`)が`null`として処理されます。このプロパティ値を、`null`として解釈される必要がある任意の文字列に設定できます。他のすべての文字列値は変更されないままとなります。このプロパティを空でない文字列に設定した場合、空の文字列または空のフィールド値は空の文字列(`""`)のままとなります。

  個々のフィールドの「**Null value**」プロパティの値によってオーバーライドできます。

- 「**Preview attachment**」。これは、メタデータに添付されているファイルのファイルURLです。ここで変更するか、または「**Browse...**」ボタンを使用して検索できます。

- 「**Preview Charset**」。これは、メタデータに添付されているファイルのキャラクタ・セットです。ここで変更するか、またはコンボ・ボックスから選択できます。

- 「**Preview Attachment Metadata Row**」。これは、レコード・フィールド名が検索される添付ファイルの行の数です。

- 「**Preview Attachment Sample Data Row**」。これは、フィールドのデータ型の推測元となる添付ファイルの行の数です。

また、**プラス**記号ボタンをクリックして、「**Custom**」プロパティを定義することもできます。たとえば、これらのプロパティは次のようになります。

- 「**charset**」。レコードのキャラクタ・セットです。たとえば、メタデータがdBaseファイルから抽出される場合は、これらのプロパティが表示される場合があります。

- 「**dataOffset**」。`fixed`または`mixed`レコード・タイプのみが表示されます。

フィールドの詳細

「**Details**」ペインにフィールドに関する情報が表示されるときには、そのプロパティが表示されます。

基本的なプロパティを次に示します。

- 「**Name**」。これは、「**Record**」ペインと同じフィールド名です。

- 「**Type**」。これは、「**Record**」ペインと同じデータ型です。

  詳細は、データ型およびレコード・タイプ(p.111)を参照してください。

- 「**Container type**」: フィールドに(同じタイプの)複数の値を格納できるかどうかを決定します。**list**および**map**という2つのオプションがあります。**単一**に切り替えて戻すと、再び一般的な単一値フィールドになります。

  詳細は、複数値フィールド(p.168)を参照してください。

- 「**Delimiter**」。これは、「**Record**」ペインの場合と同様、デフォルト以外のフィールド・デリミタです。空の場合は、かわりにデフォルトのデリミタが使用されます。

  詳細は、デリミタの変更および定義(p.165)を参照してください。

- 「**Size**」。これは、「**Record**」ペインでのサイズと同じです。

- 「**Nullable**」。`true`または`false`を指定できます。デフォルト値は`true`です。この場合、フィールド値をNULLに設定できます。そうでない場合、NULL値は禁止され、NULLが設定されるとグラフは失敗します。

- 「**Default**」。フィールドのデフォルト値です。「**Autofilling**」プロパティを`default_value`に設定した場合に使用されます。

  詳細は、自動入力関数(p.132)を参照してください。

- 「**Length**」。`decimal`データ型の場合にのみ表示されます。`decimal`データ型の場合、オプションで、その長さを定義できます。その数値の最長桁数となります。デフォルト値は`12`です。

  詳細は、データ型およびレコード・タイプ(p.111)を参照してください。

- 「**Scale**」。`decimal`データ型の場合にのみ表示されます。`decimal`データ型の場合、オプションで、スケールを定義できます。小数点の後の最大桁数となります。デフォルト値は`2`です。

  詳細は、データ型およびレコード・タイプ(p.111)を参照してください。

- 「**Description**」。このプロパティは、選択したフィールドの意味を示します。

拡張プロパティを次に示します。

- 「**Format**」。書式では、`boolean`、`date`、`decimal`、`integer`、`long`、`number`および`string`データ・フィールドの解析や書式設定を定義します。

  詳細は、データ形式(p.113)を参照してください。

- 「**Locale**」。このプロパティは、日付書式や小数区切りなどの場合に役立ちます。レコード全体に対して指定した**ロケール**をオーバーライドします。

  詳細は、ロケール(p.126)を参照してください。

- 「**Locale sensitivity**」。文字列データ型の場合にのみ表示されます。**ロケール**がフィールドまたはレコード全体に指定されている場合にのみ適用されます。レコード全体に対して指定した**ロケール依存**をオーバーライドします。

  詳細は、ロケール依存(p.131)を参照してください。

- 「**Null value**」。このプロパティは、`null`として処理される必要があるフィールドの値を指定するために設定できます。デフォルトでは、空のフィールドまたは空の文字列(`""`)が`null`として処理されます。このプロパティ値を、`null`として解釈される必要がある任意の文字列に設定できます。他のすべての文字列値は変更され

ないままとなります。このプロパティを空でない文字列に設定した場合、空の文字列または空のフィールド値は空の文字列("")のままとなります。

レコード全体の「**Null value**」プロパティの値をオーバーライドします。

- 「**Autofilling**」。定義されている場合、autofillingとしてマークされたフィールドには、自動入力関数(p.132)でリストされているいずれかの関数によって値が入力されます。
- 「**Shift**」。これは、フィールドが固定または混合レコードの一部で、それらのサイズがなんらかの値に設定されている場合、あるフィールドの終了と次のフィールドの開始との間のギャップです。
- 「**EOF as delimiter**」。EOF文字がデリミタとして使用されるかどうかに応じて、trueまたはfalseに設定できます。ファイルが他のデリミタで終了しない場合に役立ちます。このプロパティをtrueに設定しないで、そのようなデータ・ファイルを使用するグラフを実行すると、失敗します(デフォルトはfalse)。デリミタ付きまたは混合データ・レコードでのみ表示されます。

デリミタの変更および定義

*メタデータ・エディタ*の「**Record**」ペインの最初の列に数値が表示されます。これらは、個々のレコード・フィールドの番号です。これらの番号に対応するフィールド名が2番目の列(「**Name**」列)に表示されます。これらのフィールドに対応するデリミタが、「**Record**」ペインの4番目の列(「**Delimiter**」列)に表示されます。

「**Record**」ペインのこの「**Delimiter**」列のデリミタがグレーで表示される場合は、デフォルトのデリミタが使用されることを示します。「**Record**」ペインの右側の「**Details**」ペインの「**Delimiter**」行を確認すると、この行が空であることがわかります。

注意

「**Record**」ペインの最初の行には、フィールドに関してではなく、レコード全体に関する情報が表示されます。フィールド番号、フィールド名、フィールドのタイプ、デリミタやサイズは2番目の行から表示されます。このため、「**Record**」ペインの最初の行をクリックすると、個々のフィールドではなく、レコード全体に関する情報が「**Details**」ペインに表示されます。

次のことを実行できます。

- **レコードのデリミタの変更**

  詳細は、レコードのデリミタの変更(p.166)を参照してください。

- **デフォルトのデリミタの変更**

  詳細は、デフォルトのデリミタの変更(p.167)を参照してください。

- **デフォルト以外のその他のデリミタの定義**

  詳細は、フィールドのデフォルト以外のデリミタの定義(p.167)を参照してください。

重要

- **複数のデリミタ**

  複数のデリミタが含まれるレコード(たとえば、`John;Smith\30000,London|Baker Street`)がある場合、デフォルトのデリミタを次のように指定できます。

  これらすべてのデリミタを`\\|`で区切ったシーケンスとして入力します。シーケンスには空白を含めません。

  前述の例では、デフォルトのデリミタを`,\\|;\\|\\|\\|\\`とします。二重バックスラッシュは、デリミタとしての単一バックスラッシュを表します。

  他のデリミタ、およびレコード・デリミタやデフォルト以外のデリミタにも、同じものを使用できます。

  たとえば、レコード・デリミタを次のようにできます。

  ```
  \n\\|\r\n
  ```

  また、**UniversalDataReader**の「**Quoted string**」属性を`true`に設定し、そのようなデリミタが含まれるフィールドを引用符で囲むと、フラット・ファイルのフィールド値の一部としてデリミタを含めることもできます。たとえば、フィールド・デリミタとしてカンマを使用するレコードがある場合、次を1つのフィールドとして処理できます。

  ```
  "John,Smith"
  ```

- **CTL式デリミタ**

  出力不可能なデリミタを使用する必要がある場合は、CTL式として記述できます。たとえば、次のシーケンスをデリミタとしてメタデータに入力できます。

  ```
  \u0014
  ```

  このような式は、引用符で囲まれていないUnicode `\uxxxx`コードで構成されます。入力データに含まれる各バックスラッシュ文字(\)は、実際に表示されるときには二重になります。このため、メタデータでは「\\」と表示されます。

重要

Java形式のUnicode式

CloverETLのバージョン3.0以降、Java形式のUnicode式も**CloverETL**で使用できます(URL属性以外)。

1つ以上のJava形式のUnicode式を`\u0014`のように使用できます。

このような式は、一連の`\uxxxx`コード文字で構成されます。

これらは、次のようにデリミタとしても機能します(前述の、引用符のないCTL式と同様)。

```
\u0014
```

レコードのデリミタの変更

レコード・デリミタを他の任意の値に変更する場合、次の方法で実行できます。

- ***メタデータ・エディタ***の「**Record**」ペインの最初の行をクリックします。

その後、「**Details**」ペインにレコードのプロパティが表示されます。それらの中に、「**Record delimiter**」プロパティがあります。このデリミタを他の任意の値に変更します。

レコード・デリミタの新しい値は、レコード・デリミタの前の値のかわりに、「**Record**」ペインの最初の行に表示されます。再びグレーで表示されます。

重要

「**Record**」ペインの最後の行に表示される値を変更することでレコード・デリミタを変更しようとしても、レコード・デリミタは変更されません。この方法では、最後のフィールドの後、およびレコード・デリミタの前の他のデリミタのみが定義されます。

デフォルトのデリミタの変更

デフォルトのデリミタを他の値に変更する場合は、次の2つの方法のいずれかで実行できます。

- ***メタデータ・エディタ***の「**Record**」ペインの最初の行の任意の列をクリックします。その後、「**Details**」ペインにレコードのプロパティが表示されます。それらの中に、「**Default delimiter**」プロパティがあります。このデリミタを他の任意の値に変更します。

  「**Record**」ペインの行で、デフォルトのデリミタが使用されていた場所に、デフォルトのデリミタの前の値のかわりに新しい値が表示されます。これらの値は、再びグレーで表示されます。

- ***メタデータ・エディタ***の「**Record**」ペインの最初の行の「**Delimiter**」列をクリックします。その後は、このセルの値を他の任意の値に置き換えるのみです。

  このデリミタを他の任意の値に変更します。

  「**Details**」ペインの「**Default delimiter**」行と「**Record**」ペインの行の両方で、デフォルトのデリミタが使用されていた場所に、デフォルト・デリミタの前の値のかわりに新しい値が表示されます。これらの値は、再びグレーで表示されます。

フィールドに対するデフォルト以外のデリミタの定義

レコード・フィールドに対するデフォルトのデリミタ値を他の値に置き換える場合は、次の2つのうち、いずれかの方法で実行できます。

- ***メタデータ・エディタ***の「**Record**」ペインのフィールドの行の任意の列をクリックします。その後、「**Details**」ペインにフィールドのプロパティが表示されます。それらの中に、「**Delimiter**」プロパティがあります。デフォルトのデリミタが使用されている場合は、空になっています。このプロパティの任意の値をそこに入力します。

  新しい文字は、デフォルトのデリミタをオーバーライドし、同じ行内のフィールドと次の行内のフィールドとの間のデリミタとして使用されます。

- 「**Record**」ペインのフィールドの行の「**Delimiter**」列をクリックし、他の文字に置き換えます。

  新しい文字は、デフォルトのデリミタをオーバーライドし、同じ行内のフィールドと次の行内のフィールドとの間のデリミタとして使用されます。デフォルト以外のデリミタは、デフォルト・デリミタが使用される場合は空になっていた「**Details**」ペインの「**Delimiter**」行にも表示されます。

重要

ここで説明した2つの方法のいずれかで最後のフィールドに他のデリミタを定義した場合、レコード・デリミタは、そのようなデフォルト以外のデリミタによってオーバーライドされません。その値はレコードの最後のフィールドにのみ追加され、最後のフィールドとレコード・デリミタの前との間に配置されます。

ソース・コードのメタデータの編集

次のようにソース・コードのメタデータを編集することもできます。

- **内部**メタデータを編集する場合、それらの定義を**グラフ・エディタ**・ペインの「**Source**」タブに表示できます。
- **外部**メタデータを編集する場合、「**Navigator**」ペインのメタデータ・ファイル項目を右クリックして、コンテキスト・メニューから「**Open With**」→「**Text Editor**」を選択します。ファイルの内容が**グラフ・エディタ**・ペインに開かれます。

複数値フィールド

各メタデータ・フィールドには、通常、1つの値のみが格納されます(1つの整数、1つの文字列、1つの日付など)。ただし、同じタイプの複数の値を格納するように1つのフィールドを設定することもできます。

注意

複数値フィールドは、Cloverバージョン3.3の時点で使用可能になった新機能です。

例21.3. 複数値フィールドによる利点がある状況の例

- 従業員のID、NameおよびAddressが含まれるレコード。従業員は転居することがあるため、現在と過去の両方について従業員のすべての住所を追跡する必要がある場合があります。従業員が転居するたびに新しいメタデータ・フィールドを作成するのではなく、すべての住所の**リスト**を1つのフィールドに格納できます。
- 含まれる列数がそれぞれ異なるCSVファイルの入力ストリームを処理する場合。通常は、ファイルごと(列数ごと)に新しいメタデータを作成することになります。かわりに、汎用マップをメタデータに定義し、フィールドが出現するたびにフィールドをその**マップ**に追加できます。

前述のように、次に示す2つのタイプの構造があります。

リスト: 指定した(任意の)データ型の要素が含まれたセットです。ソース・コードでは、リストは[]カッコによってマークされています。例:

```
integer[] list1 = [1, 367, -1, 20, 5, 0, -79]; // a list of integer elements
boolean[] list2 = [true, false, randomBoolean()]; // a list of three boolean elements
string[] list3; // a just-declared empty list to be filled by strings
```

マップ: キーとその値のペアです。キーは常に文字列である一方、値は任意のデータ型を設定できますが、データ型は混在できません(マップでは同じタイプの値が保持されます)。例:

```
map[string,date] dateMap; // declaration

// filling the map with values
dateMap["a"] = 2011-01-01;
dateMap["b"] = 2012-12-31;
dateMap["c"] = randomDate(2011-01-01,2012-12-31);
```

マップおよびリストの詳細は、CTL2のデータ型(p.892)を参照してください。

重要

フィールドを単一値から複数値に変更するには、次の手順を実行します。

1. **メタデータ・エディタ**に移動します。
2. フィールドをクリックするか、新しいフィールドを作成します。
3. 「**Property**」→「**Basic**」で、「**Container Type**」を「**list**」または「**map**」に切り替えます。(左側の「Record」ペインのフィールド・**タイプ**の横にアイコンが表示されます。)

コンポーネントでのリストおよびマップのサポート

複数値フィールドを使用できるコンポーネントのリストを次に示します。

コンポーネント	リスト	マップ
Denormalizer(p.589)	✔ ✔	✖ ✔ (マップはキーの一部ではない)
ExtFilter(p.597)	✔	✔
ExtSort(p.600)	✔ ✔	✖ ✔ (マップはキーの一部ではない)
Merge(p.607)	✔ ✔	✖ ✔ (マップはキーの一部ではない)
Normalizer(p.612)	✔ ✔	✖ ✔ (マップはキーの一部ではない)
Partition(p.619)	✖ ✔ ✔	✖ **(範囲)** ✖ **(パーティション・キー)** ✔ **(パーティション・クラス)**
Reformat(p.632)	✔	✔
Rollup(p.635)	✔ ✔ ✔	✖ (ソート済入力) ✔ (ソート済入力、マップはキーの一部ではない) ✔ (未ソート入力)
SimpleCopy(p.647)	✔	✔
SimpleGather(p.648)	✔	✔
CloverDataReader(p.341)	✔	✔
CloverDataWriter(p.461)	✔	✔
DataGenerator(p.351)	✔	✔
JavaBeanReader	✔	✖
JavaBeanWriter	✔	✖
JSONReader	✔	✖

コンポーネント	リスト	マップ
JSONWriter	✔	✖
XMLReader	✔	✖
XMLWriter(p.558)	✔	✔
JavaMapWriter	✔	✔
Concatenate(p.581)	✔	✔
DataIntersection(p.582)	✔ ✔	✖ ✔ (マップはキーの一部ではない)
Dedup(p.587)	✔ ✔	✖ ✔ (マップはキーの一部ではない)
SortWithinGroups(p.649)	✔	✖
ApproximativeJoin(p.654)	✔ ✔	✖ ✔ (マップはキーの一部ではない)
DBJoin(p.664)	✔	✔ (マップはキーの一部ではない)
ExtHashJoin(p.667)	✔ ✔	✖ ✔ (マップはキーの一部ではない)
ExtMergeJoin(p.672)	✔	✔ (マップはキーの一部ではない)
LookupJoin(p.677)	✔	✔
RelationalJoin(p.680)	✔	✔ (マップはキーの一部ではない)
ClusterSimpleGather(p.757)	✔ ✔ ✔	✔ **(ラウンド・ロビン)** ✖ **(キーによるマージ)** ✔ **(単純な収集)**
ClusterPartition(p.751)	✖ ✔ ✔	✖ **(範囲)** ✖ **(パーティション・キー)** ✔ **(パーティション・クラス)**
LookupTableReaderWriter(p.793)	✔	✔
SequenceChecker(p.799)	✔ ✔	✖ ✔ (マップはキーの一部ではない)
SpeedLimiter(p.801)	✔	✔

現在、`map`と`list`のいずれの構造もフラット・ファイルからメタデータとして抽出することはできません。

リストおよびマップでの結合(比較ルール)

他のフィールドと同様、リストまたはマップであるフィールドを**結合キー**(結合タイプ(p.324)を参照)として指定できます。どのような場合に2つのマップ(リスト)が等しいとされるかが、唯一の問題となります。

初めに、次のことを確認してください。リスト/マップは、次のようにできます。

- `null`にする: 指定しない

```
map[string,date] myMap; // a just-declared map - no keys, no values
```

- 空の要素を含める

```
string[] myList = ["hello", ""]; // a list whose second element is empty
```

- *n*個の要素を含める: 一般的なケースは複数値フィールドによる利点がある状況の例(p.168)などを参照

2つのマップ(リスト)が等しい場合とは、それら両方が`null`ではなく、データ型が同じで、要素数とすべての要素値(マップではキーと値)が等しい場合です。

2つのマップ(リスト)が等しくない場合とは、それらのいずれかが`null`である場合です。

重要

2つのリストを比較する場合、それらの要素の順序も一致している必要があります。マップには要素の順序はないため、順序を**ソート・キー**に使用することはできません。

例21.4. 等しい(等しくない)整数リスト: シンボリック表記法

```
[1,2] == [1,2]
[null] != [1,2]
[1] != [1,2]
null != null // two unspecified lists
[null] == [null] // an extra case: lists which are not empty but whose elements
are null
```

注意: マップは`LinkedHashMap`として実装されるため、それらのプロパティはそこから導出されます。

第22章 データベース接続

データを解析する場合は、データのソースが必要となります。データは、ファイルから取得する場合や、データベースやその他のデータ・ソースから取得する場合があります。

ここでは、ファイル以外のリソースを使用して作業する方法について説明します。そのように作業するには、データ・ソースへの接続を作成する必要があります。ここでは、データベースを使用して作業する方法についてのみ説明し、接続を使用する、より高度なデータ・ソースについては後で説明します。

データベースを使用して作業する場合、2通りの方法で実行できます。1つは、クライアント・ユーティリティを使用してサーバー上のデータベースと接続するクライアントがコンピュータ上にある場合です。もう1つは、JDBCドライバを使用する方法です。ここでは、JDBCドライバを使用するデータベース接続について説明します。もう1つの方法(クライアント/サーバー・アーキテクチャ)については、後でコンポーネントに言及するときに説明します。

注意

CloverETL Serverプロジェクトでデータベース接続を使用する場合、すべてのデータベース接続はサーバー側で行われます。この利点の1つは、**CloverETL Server**からアクセス可能なデータベース・サーバーを**CloverETL Designer**内からも使用できることです。

メタデータの場合と同様、データベース接続は内部または外部(共有)の場合があります。これらは2つの方法で作成できます。

作成できるデータベース接続は次のとおりです。

- **内部**: 内部データベース接続(p.172)を参照してください。

  内部データベース接続に対して次のことを実行できます。

  - **外部化**: 内部データベース接続の外部化(p.173)を参照してください。
  - **エクスポート**: 内部データベース接続のエクスポート(p.174)を参照してください。

- **外部(共有)**: 外部(共有)データベース接続(p.175)を参照してください。

  外部(共有)データベース接続に対して次のことを実行できます。

  - **グラフへのリンク**: 外部(共有)データベース接続のリンク(p.175)を参照してください。
  - **内部化**: 外部(共有)データベース接続の内部化(p.175)を参照してください。

データベース接続ウィザードについては、データベース接続ウィザード(p.176)で説明しています。

アクセス・パスワードは暗号化できます。アクセス・パスワードの暗号化(p.180)を参照してください。

データベース接続は、メタデータを作成するためのリソースとして機能します。データベースの参照およびデータベース表からのメタデータの抽出(p.181)を参照してください。

データベース表をメタデータから直接作成することもできます。メタデータおよびデータベース接続からのデータベース表の作成(p.155)を参照してください。

内部データベース接続

メタデータに関して前述したように、内部データベース接続もグラフの一部であり、グラフに含まれ、そのソース・タブで表示できます。このプロパティは、すべての内部構造に共通です。

内部データベース接続の作成

内部データベース接続を作成する場合は、「**Outline**」ペインで「**Connections**」項目を選択し、この項目を右クリックして、「**Connections**」→「**Create DB connection**」を選択する必要があります。

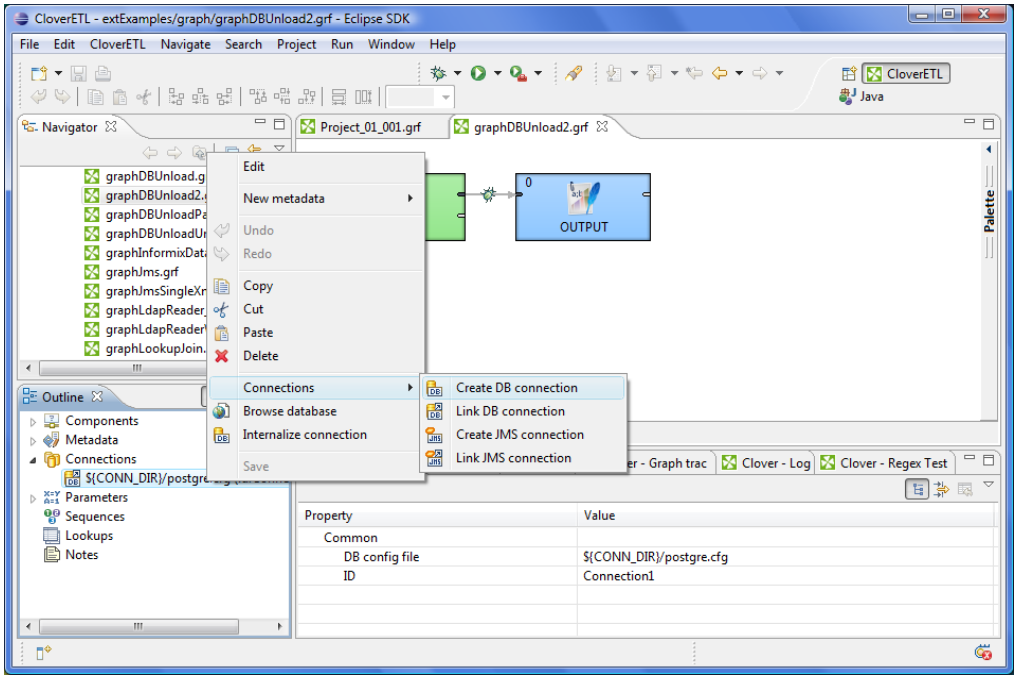


図22.1. 内部データベース接続の作成

データベース接続ウィザードが開きます。(このウィザードは、「**Outline**」ペインでDB接続項目を選択し、**[Enter]**を押した場合も開きます。)

データベース接続の作成方法の詳細は、データベース接続ウィザード(p.176)を参照してください。

接続のすべての属性を設定した後に、「**Validate connection**」ボタンをクリックして、接続を検証できます。

「**Finish**」をクリックすると、内部データベース接続が作成されます。

内部データベース接続の外部化

グラフの一部として内部データベース接続を作成した後に、それをグラフに含めることができます。グラフに含められ、表示可能となった後に、それを外部(共有)データベース接続に変換する必要がある場合があります。これにより、同じデータベース接続を複数のグラフで使用できるようになります(複数のグラフが接続を共有します)。

任意の内部接続項目を外部化して外部(共有)ファイルにするには、「**Outline**」ペインで内部接続項目を右クリックし、コンテキスト・メニューから「**Externalize connection**」を選択します。このことを実行すると、新しいウィザードが開き、新しい外部(共有)接続構成ファイルの場所としてプロジェクトのconnフォルダが提案されるので、「**OK**」をクリックします。必要に応じて(同じ名前のファイルがすでに存在するなど)、提案された接続構成ファイル名を変更できます。

その後、内部接続項目は「**Outline**」ペインの「**Connections**」グループからなくなり、同じ場所に、すでにリンクされた状態で、新しく作成された外部(共有)接続構成ファイルが表示されます。プロジェクトのconnサブフォルダに、同じ構成ファイルが表示されることを「**Navigator**」ペインで確認できます。

複数の内部接続項目を一度に外部化することもできます。このことを行うには、「**Outline**」ペインで内部パラメータを選択して右クリックした後、コンテキスト・メニューから「**Externalize connection**」を選択します。このことを実行すると、新しいウィザードが開き、選択した最初の内部接続項目の場所としてプロジェクトのconnフォル

ダが提案されるので、「**OK**」をクリックします。選択した接続項目がすべて外部化されるまで、項目ごとに同じウィザードが開きます。必要に応じて(同じ名前のファイルがすでに存在するなど)、提案された接続構成ファイル名を変更できます。

[Shift]を押しながら**下矢印**または**上矢印**キーで移動すると、隣接する接続項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各接続項目を**クリック**します。

同じことが、データベース接続とJMS接続の両方に有効です。

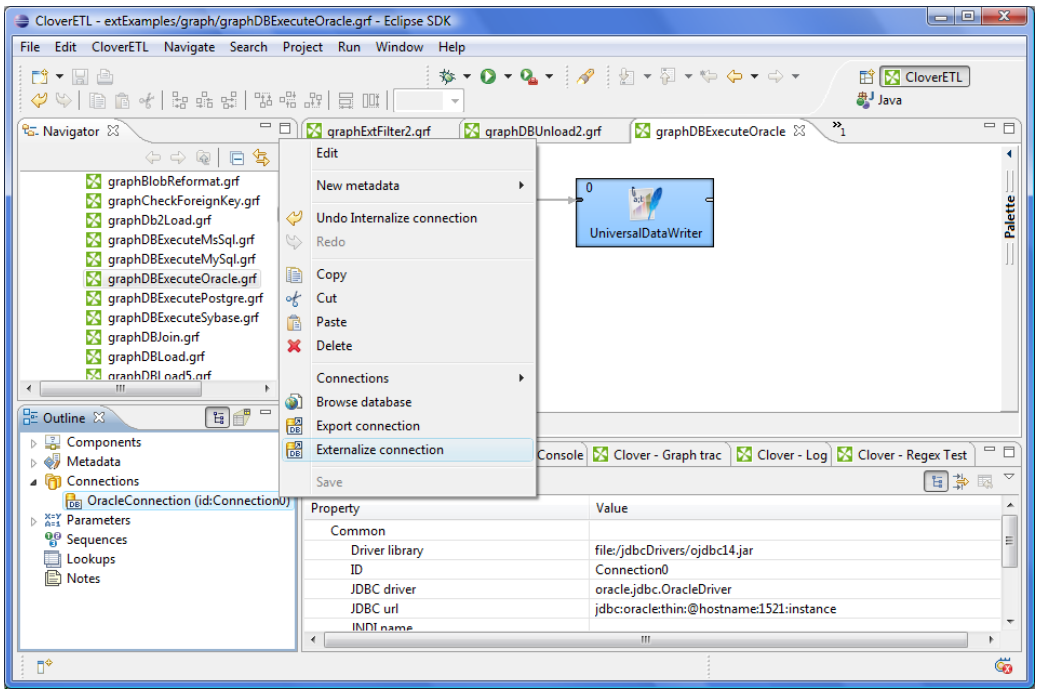


図22.2. 内部データベース接続の外部化

その後、内部ファイルは「**Outline**」ペインの「Connections」フォルダからなくなり、同じ場所に、新しく作成された構成ファイルが表示されます。

同じ構成ファイルが「**Navigator**」ペインのconnサブフォルダに表示されます。

内部データベース接続のエクスポート

このケースは、内部データベース接続の外部化と似ています。グラフの外部にある接続構成ファイルを、外部化された接続と同じ方法で作成しますが、そのファイルは元のグラフにはリンクされません。その後、前の項で説明したように、そのファイルを外部(共有)接続構成ファイルとして他のグラフで使用できます。

内部データベース接続を外部(共有)接続にエクスポートするには、「**Outline**」ペインのいずれかの内部データベース接続項目を右クリックし、コンテキスト・メニューから「**Export connection**」を選択します。新しく作成された外部ファイルに対して、対応するプロジェクトのconnフォルダが提案されます。提案されたものとは異なる名前をファイルに付けることもでき、「**Finish**」をクリックしてファイルを作成します。

その後、「**Outline**」ペインの接続フォルダは同じままですが、「**Navigator**」ペインのconnフォルダには、新しく作成された接続構成ファイルが表示されます。

外部化に関する前の項で説明しているのと同様の方法で、選択した複数の内部データベース接続をエクスポートすることもできます。

外部(共有)データベース接続

前述のように、外部(共有)データベース接続は複数のグラフで使用できる接続です。グラフの外部に格納されるため、グラフによる共有が可能となります。

外部(共有)データベース接続の作成

外部(共有)データベース接続を作成する場合はメイン・メニューから「**File**」→「**New**」→「**Other...**」を選択し、**CloverETL**カテゴリを展開し、**データベース接続**項目をクリックして「**Next**」をクリックするか、または**データベース接続**項目をダブルクリックします。**データベース接続ウィザード**が開きます。

次に、外部(共有)データベース接続のプロパティを内部データベース接続の場合と同じ方法で指定する必要があります。データベース接続の作成方法の詳細は、データベース接続ウィザード(p.176)を参照してください。

接続のすべてのプロパティを設定した後に、「**Validate connection**」ボタンをクリックして、接続を検証できます。

「**Next**」をクリックした後に、プロパティ、そのconnサブフォルダを選択し、外部データベース接続ファイルの名前を選択し、「**Finish**」をクリックします。

外部(共有)データベース接続のリンク

作成後(前の項およびデータベース接続ウィザード(p.176)を参照)、外部(共有)データベース接続を、それらが使用される各グラフにリンクできます。「**Connections**」グループまたはそのいずれかの項目を右をクリックし、コンテキスト・メニューから「**Connections**」→「**Link DB connection**」を選択する必要があります。その後、プロジェクト・コンテンツが表示された**ファイル選択**ウィザードが開きます。このウィザードでconnフォルダを展開し、このウィザードに含まれているすべてのファイルから目的の接続構成ファイルを選択する必要があります。

複数の外部(共有)接続構成ファイルを一度にリンクすることもできます。このことを行うには、「**Connections**」グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから「**Connections**」→「**Link DB connection**」を選択します。その後、プロジェクト・コンテンツが表示された**ファイル選択**ウィザードが開きます。このウィザードでconnフォルダを展開し、このウィザードに含まれているすべてのファイルから目的の接続構成ファイルを選択する必要があります。**[Shift]**を押しながら**下矢印**または**上矢印**キーで移動すると、隣接するファイル項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各ファイル項目を**クリック**します。

同じことが、データベース接続とJMS接続の両方に有効です。

外部(共有)データベース接続の内部化

外部(共有)接続を作成してリンクした後に、それをグラフに含める場合は、内部接続に変換する必要があります。このような場合、グラフ自体に接続構造が表示されます。

任意の外部(共有)接続構成ファイルを内部接続に変換するには、「**Outline**」ペインでリンク済外部(共有)接続項目を右クリックし、コンテキスト・メニューから「**Internalize connection**」をクリックします。

複数のリンク済外部(共有)接続構成ファイルを一度に内部化することもできます。このことを行うには、「**Outline**」ペインで目的のリンク済外部(共有)接続項目を選択します。**[Shift]**を押しながら**下矢印**または**上矢印**キーで移動すると、隣接する項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各項目を**クリック**します。

その後、選択したリンク済外部(共有)接続項目は「**Outline**」ペインの「**Connections**」グループからなくなり、同じ場所に、新しく作成された内部接続項目が表示されます。

ただし、元の外部(共有)接続構成ファイルはconnサブフォルダにそのまま存在し、「**Navigator**」ペインで確認できます。

同じことが、データベース接続とJMS接続の両方に有効です。

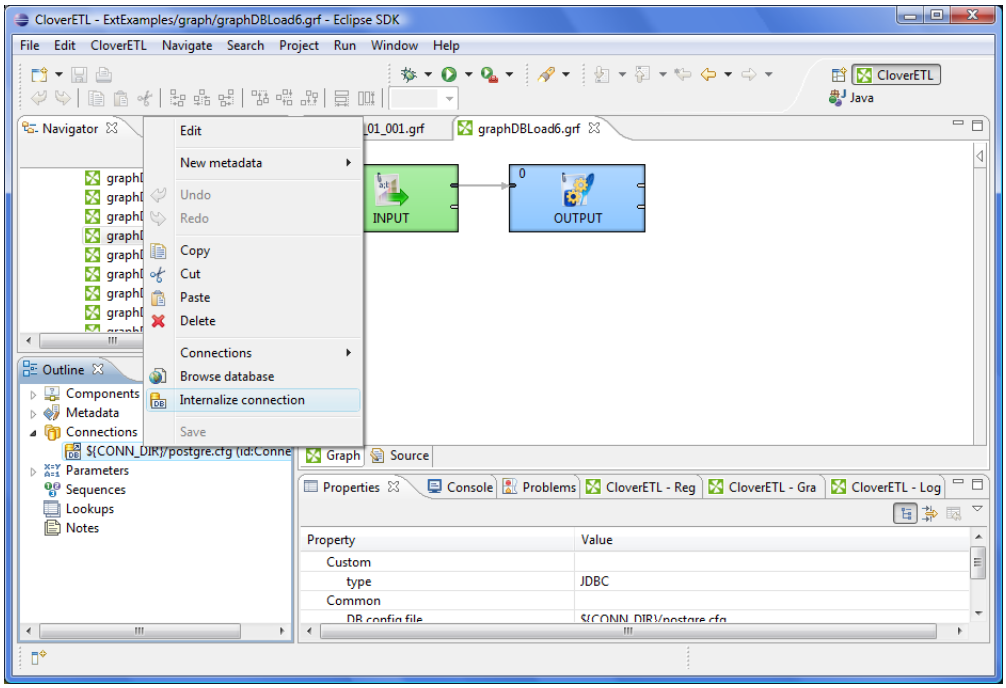


図22.3. 外部(共有)データベース接続の内部化

データベース接続ウィザード

このウィザードは、2つのタブ(「**Basic properties**」および「**Advanced properties**」)で構成されます。

データベース接続ウィザードの「**Basic properties**」タブでは、接続の名前を指定し、**ユーザー**名、アクセス・**パスワード**、およびデータベース接続の**URL** (**ホスト名**、**データベース**名またはその他のプロパティ)または**JNDI**を入力する必要があります。チェック・ボックスを選択して、アクセス・パスワードを暗号化するかどうかを決定することもできます。**JDBC固有**のプロパティを設定する必要があります。デフォルトのプロパティを使用することもできますが、実際に必要となるすべてのことが実行されるとはかぎりません。**JDBC仕様**を設定することで、異なるデータ型変換、自動生成キーの取得など、接続の動作を多少変更できます。

データベース接続は、この属性によって最適化されます。**JDBC仕様**では、指定のデータベース・タイプと最適に連携できるように接続が調整されます。

組込み接続を選択することもできます。**CloverETL**に現在組み込まれている接続は、**Derby**、**Firebird**、**Microsoft SQL Server** (**Microsoft SQL Server 2008**または**Microsoft SQL Server 2000-2005**固有)、**MySQL**、**Oracle**、**PostgreSQL**、**Sybase**および**SQLite**です。これらのいずれかを選択すると、接続コードに次のいずれかの式がそれぞれ表示されます。`database="DERBY"`、`database="FIREBIRD"`、`database="MSSQL"`、`database="MYSQL"`、`database="ORACLE"`、`database="POSTGRE"`、`database="SYBASE"`または`database="SQLITE"`。

重要

ODBCリソースに接続する必要がある場合は、**汎用ODBC**ドライバを使用します。ただし、他のダイレクトJDBCドライバが機能しない場合にのみ選択してください。さらに、使用しているClover (32または64ビット)に合った適切なODBCバージョンを使用するようにしてください。

新しいデータベース接続を作成する場合は、グラフにすでにリンクされている既存の接続(内部および外部のいずれか)を**接続**リスト・メニューから選択して、その接続を使用することを選択できます。また、「**Load from file**」ボタンをクリックして、接続構成ファイルから(リンクされていない)外部接続をロードすることもできます。

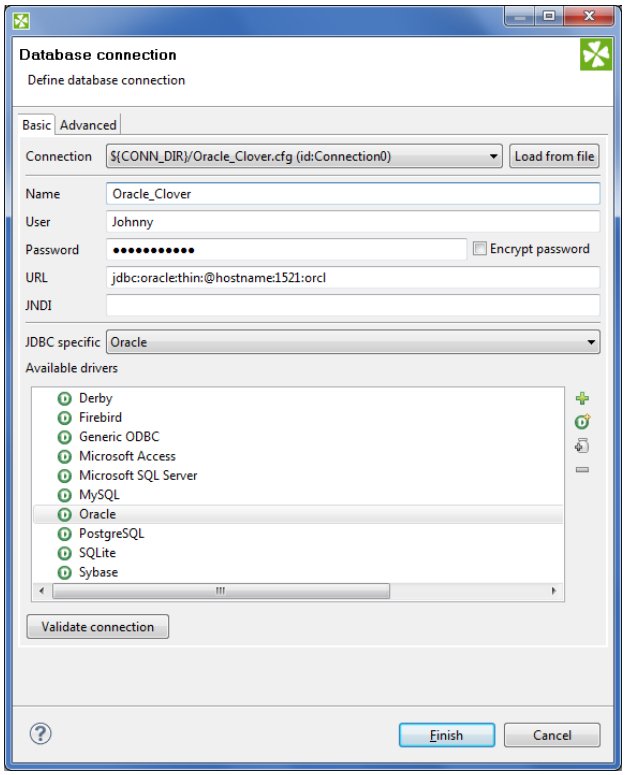


図22.4. データベース接続ウィザード

すべての属性は対応する方法で変更されます。

(組み込まれていない)他のドライバを使用する場合は、**使用可能なドライバ**のいずれかを使用できます。目的のJDBCドライバがリストにない場合は、ウィザードの右側にある**プラス**記号をクリックして追加できます(JARからのドライバのロード)。次に、ドライバを検索して選択できます。結果は次のようになります。

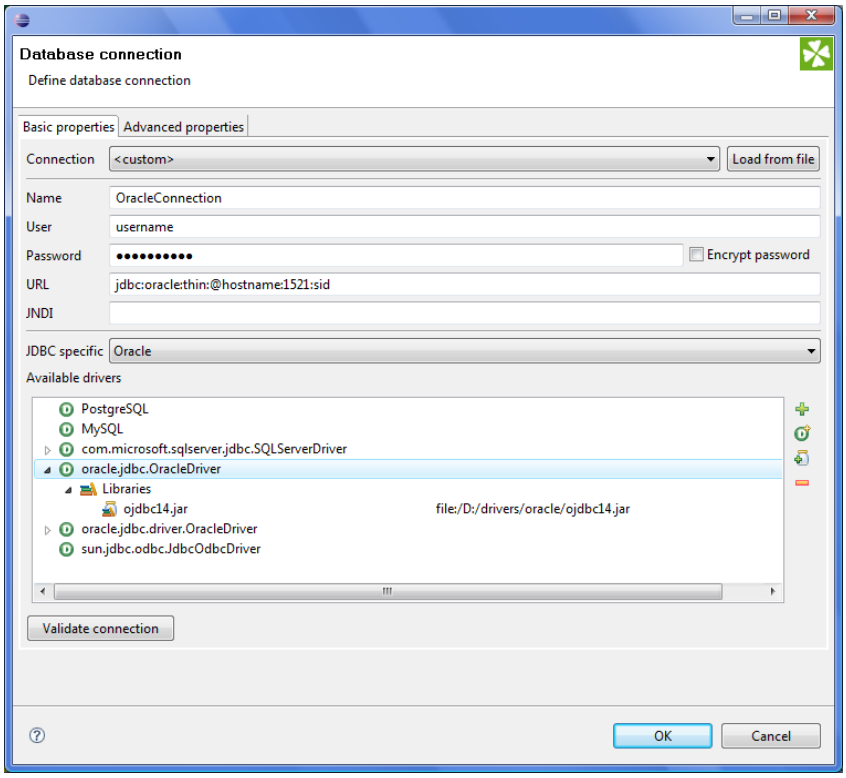


図22.5. 使用可能なドライバのリストへの新しいJDBCドライバの追加

必要な場合は、別のJARをドライバ・クラスパスに追加することもできます(**ドライバ・クラスパスへのJARの追加**)。たとえば、一部のデータベースでは、データベースのライセンスをドライバとともに追加する必要があります。

プロパティを追加することもできます(**ユーザー定義プロパティの追加**)。

マイナス記号をクリックして、リストからドライバを削除することもできます(**削除の選択**)。

前述のように、**CloverETL**にはすでに組込み**JDBC**ドライバが備えられており、使用可能なドライバのリストに表示されます。これらは、**Derby**、**Firebird**、**Microsoft SQL Server 2008**、**MySQL**、**Oracle**、**PostgreSQL**、**SQLite**および**Sybase**データベースの**JDBC**ドライバです。

使用可能なドライバのリストから任意の**JDBC**ドライバを選択できます。これらのいずれかをクリックすると、接続文字列のヒントが**URL**テキスト領域に表示されます。接続の変更のみが必要となります。**JNDI**を指定することもできます。

重要

CloverETLでは、JDBC 3以降のドライバがサポートされます。

リストからドライバを選択した後は、データベースに接続するためのユーザー名とパスワードを入力するのみです。また、ホスト名を適切な名前に変更することも必要となります。データベースによって入力された語句のかわりに、適切なデータベース名を入力する必要もあります。他のドライバでは異なるURLが提供される場合があり、そのURLは違う方法で変更する必要があります。既存のいずれかの構成ファイルから既存の接続をロードすることもできます。**JDBC固有**のプロパティを設定できます。デフォルトのプロパティも使用できますが、実際に必要となるすべてのことが実行されるとはかぎりません。**JDBC仕様**を設定することで、異なるデータ型変換、自動生成キーの取得など、選択した接続の動作を多少変更できます。

データベース接続は、この属性に基づいて最適化されます。**JDBC仕様**では、指定のデータベース・タイプと最適に連携できるように接続が調整されます。

汎用ODBC

このドライバは、**使用可能なドライバ**に直接リストされていないデータ・ソース(DBFなど)の読取りに使用されます。ODBCリソースに接続するには、次を実行します。

- **汎用ODBC**ドライバをクリックします。
- **URL**: `dsn_source`を指定します。Windowsの場合は、「**ODBC データ ソース アドミニストレータ**」→「**ユーザー DSN**」で「**名前**」として表示されます。
- **User**および**Password**: 空白のままにします。

汎用ODBCドライバを使用する場合の注意事項は次のとおりです。

- DBOutputTable(p.473)では、SQLフィールドへのメタデータ・フィールドのマッピングはチェックできません。マッピングは正しく設計してください。マッピングが無効な場合は、グラフが失敗します。
- トランザクション分離レベルは設定できません(ログへの書込みに関する警告)。

重要

汎用ODBCは、他のダイレクトJDBCドライバが機能しない場合にのみ選択してください。ODBCドライバが存在する場合でも、(MySQL ODBCドライバでのテストには成功した) Cloverで機能するとはかぎりません。さらに、使用しているClover (32または64ビット)に合った適切なODBCバージョンを使用するようにしてください。

MS Access

このドライバは、インストールされているデフォルトのMS Accessドライバを想定しています(**MS Access Database**が存在するかどうかは、**「ODBC データ ソース アドミニストレータ」→「ユーザー DSN」**で確認)。次の手順:

- **「Available drivers」**で**「Microsoft Access」**をクリックします。
- **URL**: `database_file`をMDBファイルへの絶対パスに置き換えます。

MS Accessドライバを使用する場合の注意事項を次に示します。

- DBOutputTable(p.473)では、`long`および`decimal`タイプを入力メタデータで使用できません。Reformat(p.632)をグラフで使用して、これらをその他のメタデータ型に変換することを検討してください。
- DBOutputTable(p.473)では、SQLフィールドへのメタデータ・フィールドのマッピングはチェックできません。マッピングは正しく設計してください。マッピングが無効な場合は、グラフが失敗します。
- トランザクション分離レベルは設定できません(ログへの書込みに関する警告)。
- `null`の`boolean`フィールドは、実際には`false`として書き込まれます(null値はサポートされていません)。
- `binary`フィールド: これらにも`null`を書き込むことはできません。

拡張プロパティ

前述の**「Basic properties」**タブに加えて、**データベース接続**ウィザードでは**「Advanced properties」**タブも表示されます。このタブに切り替えると、選択した接続に関して、次のようなその他のプロパティを指定できます。

- **threadSafeConnection**

  デフォルトでは、`true`に設定されています。このデフォルト設定では、各スレッドは、複数のコンポーネントがスレッド・セーフでない同じ接続オブジェクトを介してDBと対話する場合の問題を回避するために、独自の接続を取得します。

- **transactionIsolation**

  特定のトランザクション分離レベルを指定できます。詳細は、http://docs.oracle.com/javase/6/docs/api/java/sql/Connection.htmlを参照してください。この属性の可能な数値は、次のとおりです。

  - `0 (TRANSACTION_NONE)`。

    トランザクションがサポートされていないことを示す定数。

  - `1 (TRANSACTION_READ_UNCOMMITTED)`。

    ダーティ読取り、反復不可能な読取り、仮読取りが発生する可能性を示す定数。このレベルでは、あるトランザクションで変更された行が、その行の変更がコミットされる前に別のトランザクションで読み取られることを許可します(ダーティ読取り)。変更がロールバックされた場合、2番目のトランザクションで無効な行が取得されます。

    これは、**DB2**、**Derby**、**Informix**、**MySQL**、**MS SQL Server 2008**、**MS SQL Server 2000-2005**、**PostgreSQL**および**SQLite**固有のデフォルト値です。

    この値は、**汎用**と呼ばれる**JDBC仕様**が使用される場合のデフォルトとしても使用されます。

  - `2 (TRANSACTION_READ_COMMITTED)`。

    ダーティ読取りを回避し、反復不可能な読取りと仮読取りは発生する可能性があることを示す定数。このレベルでは、コミットされていない変更が含まれる行をトランザクションが読み取ることのみを禁止します。

    これは、**Oracle**および**Sybase**固有のデフォルト値です。

- 4 (TRANSACTION_REPEATABLE_READ)。

  ダーティ読取りおよび反復不可能な読取りを回避し、仮読取りは発生する可能性があることを示す定数。このレベルでは、コミットされていない変更が含まれる行をトランザクションが読み取ることを禁止し、かつ、1つのトランザクションが行を読み取り、2番目のトランザクションがその行を変更し、最初のトランザクションがその行を再読取りした場合に、異なる値を2回目に取得する状況(反復不可能な読取り)も禁止します。

- 8 (TRANSACTION_SERIALIZABLE)。

  ダーティ読取り、反復不可能な読取りおよび仮読取りが回避されることを示す定数。このレベルでは、TRANSACTION_REPEATABLE_READでの禁止内容が含まれる以外に、1つのトランザクションがあるwhere条件を満たすすべての行を読み取り、2番目のトランザクションがそのwhere条件を満たす行を挿入し、最初のトランザクションが同じ条件で再読取りし、その2番目の読取りで、追加された仮の行を取得する状況も禁止します。

- **holdability**

  Connectionを使用して作成されたResultSetオブジェクトの保持機能を指定できます。詳細は、http://docs.oracle.com/javase/6/docs/api/java/sql/ResultSet.htmlを参照してください。可能なオプションを次に示します。

  - 1 (HOLD_CURSORS_OVER_COMMIT)。

    メソッドConnection.commitが呼び出されたときにResultSetオブジェクトを閉じないことを示す定数。

    これは、**Informix**および**MS SQL Server 2008**固有のデフォルト値です。

  - 2 (CLOSE_CURSORS_AT_COMMIT)。

    メソッドConnection.commitが呼び出されたときにResultSetオブジェクトを閉じることを示す定数。

    これは、**DB2**、**Derby**、**MS SQL Server 2000-2005**、**MySQL**、**Oracle**、**PostgreSQL**、**SQLite**および**Sybase**固有のデフォルト値です。

    この値は、**汎用**と呼ばれる**JDBC仕様**が使用される場合のデフォルトとしても使用されます。

アクセス・パスワードの暗号化

アクセス・パスワードを暗号化しないと、構成ファイル(共有接続)またはグラフ自体(内部接続)に格納されて表示可能な状態のままとなります。このため、アクセス・パスワードがこれらの2つの場所のいずれかで表示される可能性があります。

このことは、グラフやコンピュータにアクセスできるユーザーが1人のみである場合は問題ありません。ただし、それ以外の場合は、パスワードを使用してそのデータベースにアクセスできないように、パスワードを暗号化することをお薦めします。

他のユーザーにグラフを提供する場合にも、データベース全体に対するアクセス・パスワードを渡す必要はありません。したがって、アクセス・パスワードを暗号化できます。このオプションがないと、データベースへの侵入や、アクセス・パスワードを入手できた人物によるその他の損害のリスクが非常に高くなります。

このため、アクセス・パスワードを暗号化してグラフを提供することが重要であり、可能となっています。このようにすると、他人がパスワードを簡単に抽出できなくなります。

アクセス・パスワードを非表示にするには、**データベース接続**ウィザードで**「Encrypt password」**チェック・ボックスを選択し、元の(暗号化対象の)アクセス・パスワードを暗号化する新しい(暗号化)パスワードを入力して、最後に**「Finish」**ボタンをクリックする必要があります。

この設定により、「**Run as**」→「**CloverETL graph**」を選択してグラフを実行できなくなります。グラフを実行するには、**実行構成**ウィザードを使用する必要があります。この場合、「**Main**」タブで、プロジェクトの名前、グラフ名およびパラメータ・ファイルを入力するか、参照して指定する必要があります。次に、「**Password**」テキスト領域に暗号化パスワードを入力します。これで、アクセス・パスワードはすでに暗号化されて読み取ることができなくなり、構成ファイルやグラフで表示されなくなります。

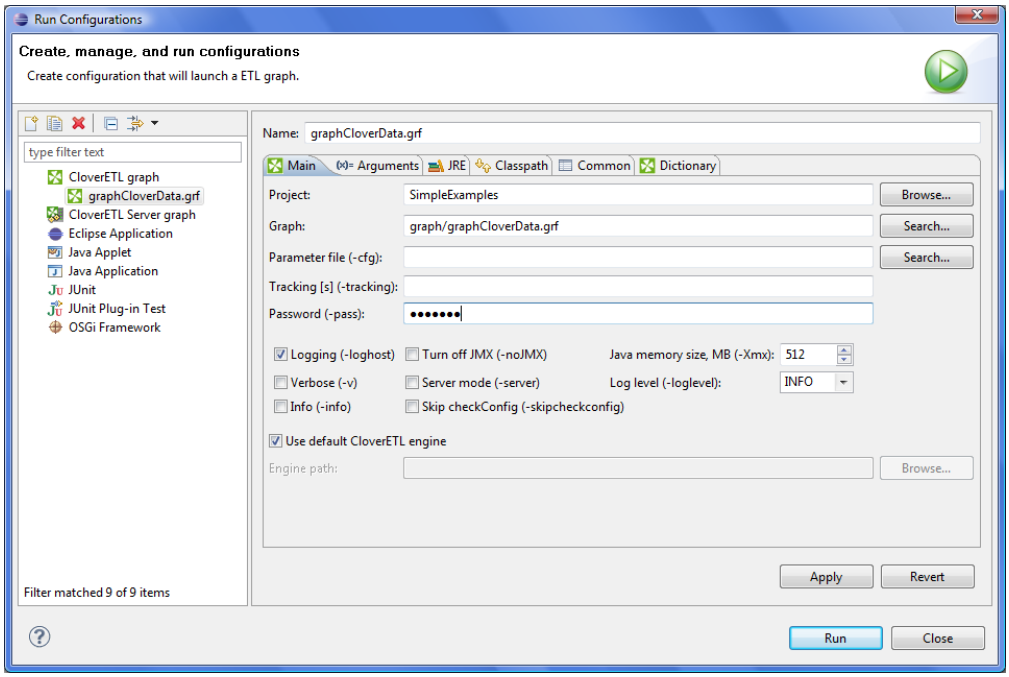


図22.6. パスワードが暗号化されたグラフの実行

アクセス・パスワードに戻す場合は、**データベース接続**ウィザードに暗号化パスワードを入力し、「**Finish**」をクリックします。

データベースの参照およびデータベース表からのメタデータの抽出

前述(内部データベース接続の外部化(p.173)および外部(共有)データベース接続の内部化(p.175)を参照)のように、これら両方の場合のコンテキスト・メニューに「**Browse database**」と「**New metadata**」という2つの有用な項目が含まれます。これらにより、データベースを参照(接続が有効な場合)したり、選択したデータベース表からメタデータを抽出できます。このようなメタデータは、内部メタデータのみですが、後で外部化またはエクスポートできます。

重要

データベース表をメタデータから直接作成することもできます。メタデータおよびデータベース接続からのデータベース表の作成(p.155)を参照してください。

Microsoft SQL ServerでのWindows認証

Windows認証とは、「**User**」と「**Password**」を空白にしたままでMicrosoft SQL Serverへのデータベース接続を作成する(下の図を参照)ことを意味します。MS SQLデータベースへのアクセスでは、JTDSドライバはWindowsアカウントを使用してログインします。これらのことをすべて有効にするには、この項で説明している手順に従います。

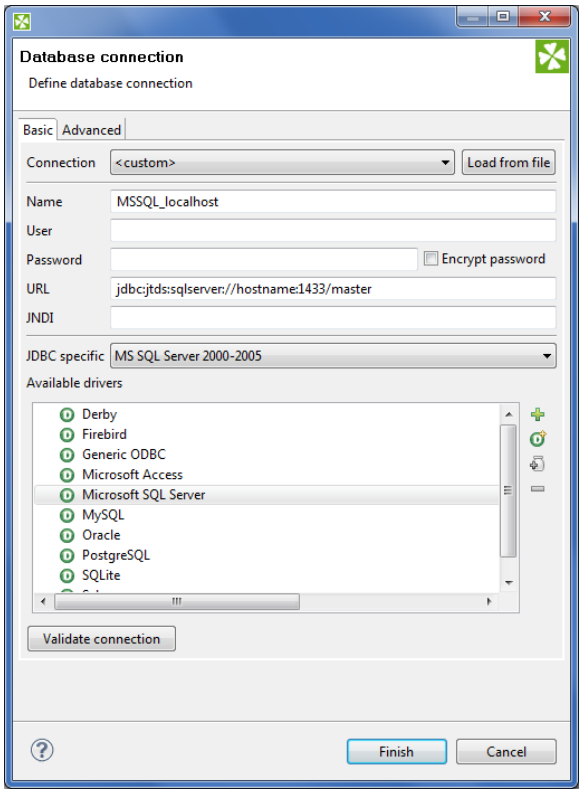


図22.7. Windows認証によるMS SQLへの接続。このようにデータベース接続を設定するのみでは、十分ではありません。この図の下で説明している追加手順を実行する必要があります。

Cloverには、JDTSのJDBCドライバが付属しています。ただし、JDTSがMircrosoft SQL ServerでWindows認証を使用して作業するために必要なネイティブ・ライブラリは提供されません。このため、ネイティブdll (`ntlmauth.dll`)をダウンロードして、追加設定を実行する必要があります。

ネイティブ・ライブラリの取得

Cloverでは、**JTDSバージョン1.2.4**がサポートされます。ダウンロードの手順は、次のとおりです。

1. 配布パッケージを取得します。
2. 内容を抽出し、フォルダ`x64\SSO`または`x86\SSO`に移動します。
3. `ntlmauth.dll`がそこに格納されています。ファイル(それぞれ、Cloverの64ビットまたは32ビット・バージョン用)をフォルダ(`C:\jtds_dll`など)にコピーします。

インストール

dllを機能させる方法は2通りあります。1つ目のものには、Windowsの`PATH`変数の変更が伴います。このことを行わない場合は、2つ目のオプションを実行します。

1. dllファイルへの絶対パス(`C:\jtds_dll`)をWindowsのPATH変数に追加します。または、PATHにすでに含まれているフォルダ(`C:\WINDOWS\system32`など)にdllファイルを配置できます。
2. 次に示す、CloverETL製品ファミリのすべてのメンバーの`java.library.path`プロパティを変更します。

   - **Designer**

`CloverETLDesigner.ini`を変更し、次のようにdllの場所へのjavaライブラリ・パスを設定する新しい行を追加します。

```
-Djava.library.path=C:\jtds_dll
```

次に、グラフの「**Run Configurations**」画面でプログラムとVM引数(p.85)を変更します(下の図を参照してください)。次の行を**VM引数**に追加します。

```
-Djava.library.path=C:\jtds_dll
```

注意

Windows認証を使用する必要があるコンポーネントのすべてのグラフのVM引数を変更する必要があります。

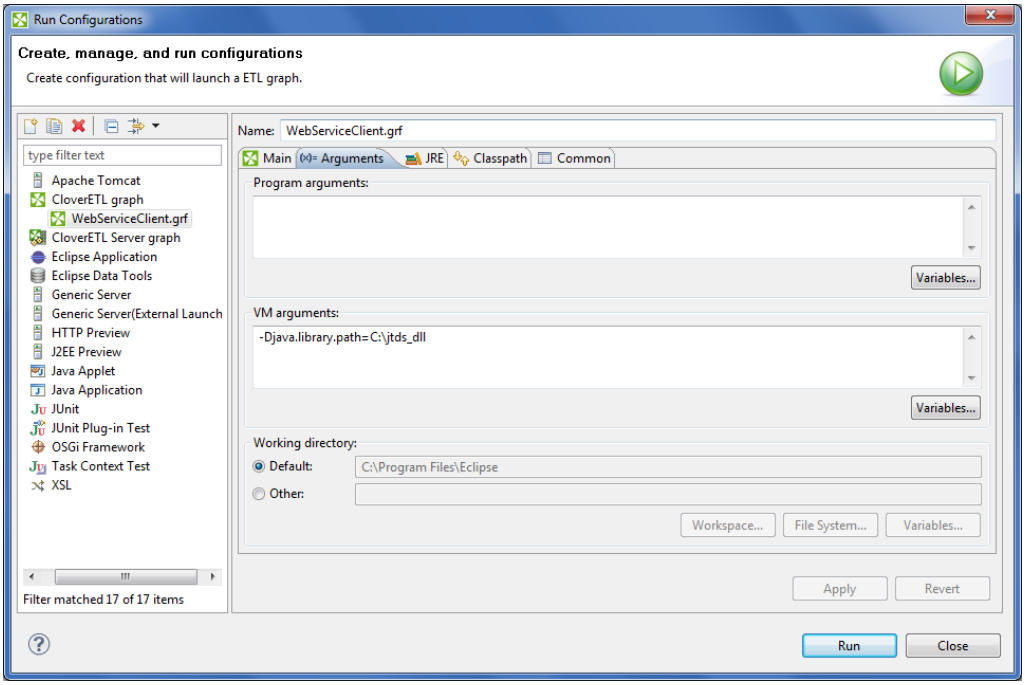


図22.8. ネイティブdllへのパスをVM引数に追加

- **Clover Server**

  Tomcatを起動するスクリプトで、`-Djava.library.path=C:\jtds_dll`オプションを`JAVA_OPT`に追加します。たとえば、次の行を`catalina.bat`の先頭に追加します。

  ```
  set JAVA_OPTS=%JAVA_OPTS% -Djava.library.path=C:\jtds_dll
  ```

3. **MS SQL Server**: 次のことを確認してください。

   - 「**SQL Server ネットワークの構成**」→「**プロトコル**」で、TCP/IPが有効化されている。
   - 「**TCP/IP のプロパティ**」→「**IP アドレス**」→「**IPAll**」で、**TCPポート**が1433に設定されている。

Hive接続

Apache Hiveへの接続は、他のDB接続(p.172)と同じ方法で作成できます。ここでは、いくつかの有用なHive固有の注釈を示します。

Hive JDBCドライバ

JDBCドライバは、Hiveリリースに含まれています。ただし、ライブラリとその依存性は、他のHiveライブラリの間で分散しています。さらに、このドライバは、Hadoop配布のもう1つのライブラリ(`hadoop-core-*.jar`または`hadoop-common-*.jar`)に依存していますが、Hadoopのバージョンによっては、常に、それらのうち1つのみが存在します。

Hiveバージョン0.8.1のHive DB接続JDBCドライバで必要な最小限のライブラリ・リストを次に示します。

```
hadoop-core-0.20.205.jar
hive-exec-0.8.1.jar
hive-jdbc-0.8.1.jar
hive-metastore-0.8.1.jar
hive-service-0.8.1.jar
libfb303-0.7.0.jar
slf4j-api-1.6.1.jar
slf4j-log4j12-1.6.1.jar
```

すべてのHive配布ライブラリおよび1つのHadoopライブラリをJDBCドライバ・クラスパスに配置できます。ただし、一部のHive配布ライブラリがすでにCloverに含まれている場合があり、その場合はクラスのロード競合が発生する可能性があります。通常、`commons-logging*`ライブラリと`log4j*`ライブラリは含まれていませんが、含まれている場合は、(無害を示す)警告がグラフ実行ログに表示されます。

Clover変換グラフでのHiveの使用方法

Hiveは通常のSQLリレーショナル・データベースではなく、QLと呼ばれる、SQLに似た独自の問合せ言語があります。Hive QLおよびHiveに関する一般的なリソースとしてはApache Hive Wikiが非常に有用です。

`INSERT INTO`文では`SELECT`問合せの結果のみを挿入できるため、DBOutputTable(p.473)コンポーネントを使用することは意味がありません。これを回避することは可能ですが、そのような`INSERT`文を使用して挿入される各Cloverデータ・レコードによって、MapReduceジョブが重くなり、コンポーネントが非常に低速になります。かわりに、`LOAD DATA` Hive QL文を使用します。

DBExecute(p.782)コンポーネントでは、「**Transaction set**」属性は常に**1つの文**に設定されます。これは、Hive JDBCドライバでは、複数のトランザクションがサポートされず、それらを使用しようとすると、自動コミット・モードを無効にできない、というエラーが発生するためです。

ファイルの追加操作は、バージョン0.21.0以降のHDFSでのみ完全にサポートされます。そのため、古いHDFS上でHiveを実行すると、データを既存の表に追加できません(`OVERWRITE`キーワードの使用が必須となります)。

第23章 JMS接続

JMSメッセージを受信するには、JMS接続が必要です。メタデータ、パラメータおよびデータベース接続と同様、内部または外部(共有)の場合があります。

作成できる各JMS接続は次のとおりです。

- **内部**: 内部JMS接続(p.185)を参照してください。

  内部JMS接続に対して次のことを実行できます。

  - **外部化**: 内部JMS接続の外部化(p.185)を参照してください。
  - **エクスポート**: 内部JMS接続のエクスポート(p.186)を参照してください。

- **外部(共有)**: 外部(共有) JMS接続(p.187)を参照してください。

  外部(共有) JMS接続に対して次のことを実行できます。

  - **グラフへのリンク**: 外部(共有) JMS接続のリンク(p.187)を参照してください。
  - **内部化**: 外部(共有) JMS接続の内部化(p.187)を参照してください。

JMS接続の編集ウィザードについては、JMS接続の編集ウィザード(p.188)で説明しています。

認証パスワードは暗号化できます。認証パスワードの暗号化(p.189)を参照してください。

内部JMS接続

他のツール(メタデータ、データベース接続およびパラメータ)について前述したように、内部JMS接続もグラフの一部であり、グラフに含まれ、そのソース・タブで表示できます。このプロパティは、すべての内部構造に共通です。

内部JMS接続の作成

内部JMS接続を作成する場合は、「**Outline**」ペインで「**Connections**」項目を選択し、この項目を右クリックして、「**Connections**」→「**Create JMS connection**」を選択する必要があります。**JMS接続の編集**ウィザードが開きます。このウィザードでJMS接続を定義できます。その外観、および接続の設定方法については、JMS接続の編集ウィザード(p.188)で説明しています。

内部JMS接続の外部化

グラフの一部として内部JMS接続を作成した後に、それを外部(共有) JMS接続に変換する必要がある場合があります。これにより、複数のグラフにわたって同じJMS接続を使用できるようになります。

任意の内部接続項目を外部化して外部(共有)ファイルにするには、「**Outline**」ペインで内部接続項目を右クリックし、コンテキスト・メニューから「**Externalize connection**」を選択します。このことを実行すると、新しいウィザードが開き、新しい外部(共有)接続構成ファイルの場所としてプロジェクトのconnフォルダが提案されるので、「**OK**」をクリックします。必要に応じて(同じ名前のファイルがすでに存在するなど)、提案された接続構成ファイル名を変更できます。

その後、内部接続項目は「**Outline**」ペインの「**Connections**」グループからなくなり、同じ場所に、すでにリンクされた状態で、新しく作成された外部(共有)接続が表示されます。プロジェクトのconnサブフォルダに、同じ構成ファイルが表示されることを「**Navigator**」ペインで確認できます。

複数の内部接続項目を一度に外部化することもできます。このことを行うには、「**Outline**」ペインで内部パラメータを選択して右クリックした後、コンテキスト・メニューから「**Externalize connection**」を選択します。このことを実行すると、新しいウィザードが開き、選択した最初の内部接続項目の場所としてプロジェクトのconnフォルダが提案されるので、「**OK**」をクリックします。選択した接続項目がすべて外部化されるまで、項目ごとに同じウィザードが開きます。必要に応じて(同じ名前のファイルがすでに存在するなど)、提案された接続構成ファイル名を変更できます。

[Shift]を押しながら**下矢印**または**上矢印**キーで移動すると、隣接する接続項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各接続項目を**クリック**します。

同じことが、データベース接続とJMS接続の両方に有効です。

内部JMS接続のエクスポート

このケースは、内部JMS接続の外部化と似ています。ただし、グラフの外部にある接続構成ファイルを外部化と同じ方法で作成する一方で、ファイルは元のグラフにリンクされません。接続構成ファイルのみが作成されます。その後、前の項で説明したように、そのファイルを外部(共有)接続構成ファイルとして他のグラフに使用できます。

内部JMS接続を外部(共有)接続にエクスポートするには、「**Outline**」ペインでいずれかの内部JMS接続項目を右クリックし、コンテキスト・メニューから「**Export connection**」を選択します。新しく作成された外部ファイルに対して、対応するプロジェクトのconnフォルダが提案されます。提案されたものとは異なる名前をファイルに付けることもでき、「**Finish**」をクリックしてファイルを作成します。

その後、「**Outline**」ペインの接続フォルダは同じままですが、「**Navigator**」ペインのconnフォルダには、新しく作成された接続構成ファイルが表示されます。

外部化に関する前の項で説明しているのと同様の方法で、選択した複数の内部JMS接続をエクスポートすることもできます。

外部(共有) JMS接続

前述のように、外部(共有) JMS接続は複数のグラフにわたって使用できる接続です。グラフの外部に格納されるため、共有可能となります。

外部(共有) JMS接続の作成

外部(共有) JMS接続を作成する場合は、**「File」→「New」→「Other...」**を選択し、**CloverETL**項目を展開し、**JMS接続**項目をクリックして**「Next」**をクリックするか、または**JMS接続**項目をダブルクリックします。**JMS接続の編集**ウィザードが開きます。JMS接続の編集ウィザード(p.188)を参照してください。

接続のすべてのプロパティを設定した後に、**「Validate connection」**ボタンをクリックして、接続を検証できます。

「Next」をクリックした後に、プロパティ、そのconnサブフォルダを選択し、外部JMS接続ファイルの名前を選択し、**「Finish」**をクリックします。

外部(共有) JMS接続のリンク

作成後(前の項およびJMS接続の編集ウィザード(p.188)を参照)、外部(共有)接続を使用される任意のグラフにリンクできます。単に**「Connections」**グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから**「Connections」→「Link JMS connection」**を選択します。その後、プロジェクト・コンテンツが表示される**ファイル選択**ウィザードが開きます。このウィザードでconnフォルダを展開し、目的の接続構成ファイルを選択する必要があります。

複数の外部(共有)接続構成ファイルを一度にリンクできます。このことを行うには、**「Connections」**グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから**「Connections」→「Link JMS connection」**を選択します。その後、プロジェクト・コンテンツが表示された**ファイル選択**ウィザードが開きます。このウィザードでconnフォルダを展開し、目的の接続構成ファイルを選択する必要があります。**[Shift]**を押しながら**下矢印**または**上矢印**キーを押すと、隣接するファイル項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各ファイル項目を**クリック**します。

同じことが、データベース接続とJMS接続の両方に有効です。

外部(共有) JMS接続の内部化

外部(共有)接続を作成してリンクした後に、それをグラフに含める場合は、内部接続に変換する必要があります。このような場合、グラフ自体に接続構造が表示されます。

任意の外部(共有)接続構成ファイルを内部化して内部接続にするには、**「Outline」**ペインでリンク済外部(共有)接続項目を右クリックし、コンテキスト・メニューから**「Internalize connection」**をクリックします。

複数のリンク済外部(共有)接続構成ファイルを一度に内部化することもできます。このことを行うには、**「Outline」**ペインで目的のリンク済外部(共有)接続項目を選択します。**[Shift]**を押しながら**下矢印**または**上矢印**キーで移動すると、隣接する項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各項目を**クリック**します。

その後、選択したリンク済外部(共有)接続項目は**「Outline」**ペインの**「Connections」**グループからなくなり、同じ場所に、新しく作成された内部接続項目が表示されます。

ただし、元の外部(共有)接続構成ファイルはconnサブフォルダにそのまま存在し、**「Navigator」**ペインで確認できます。

同じことが、データベース接続とJMS接続の両方に有効です。

「Edit JMS Connection」ウィザード

次に示すように、「**Edit JMS connection**」ウィザードには、入力が必要な8つの領域、つまり、「**Name**」、「**Initial context factory class**」(初期コンテキストを作成するファクトリ・クラスの完全修飾名)、「**Libraries**」、「**URL**」、「**Connection factory JNDI name**」(`javax.jms.ConnectionFactory`インタフェースを実装)、「**Destination JNDI**」(`javax.jms.Destination`インタフェースを実装)、「**User**」、「**Password**」(メッセージを受信または作成するためのパスワード)が含まれています。

(このウィザードは、「**Outline**」ペインでJMS接続項目を選択し、**[Enter]**を押して開くこともできます。)

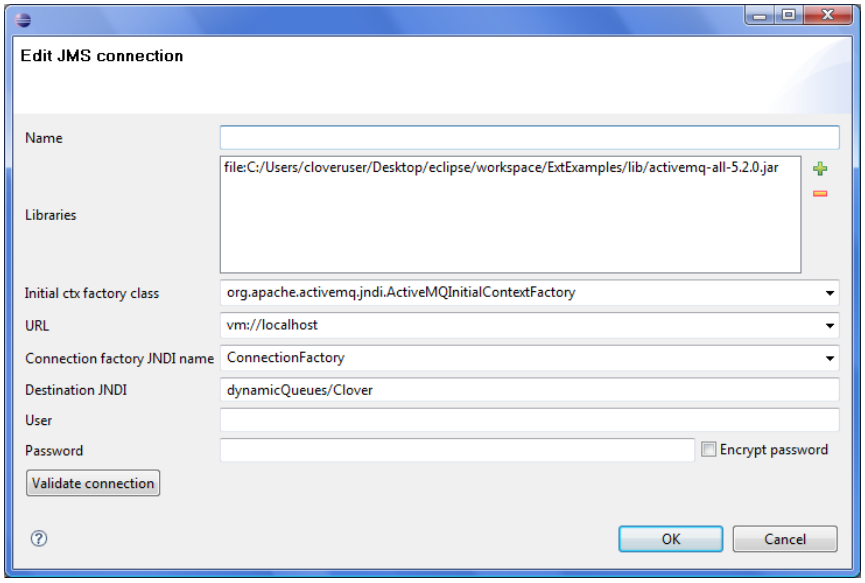


図23.1. 「Edit JMS connection」ウィザード

「**Edit JMS connection**」ウィザードで、接続の**名前**を指定し、必要な**ライブラリ**を選択し(**プラス・**ボタンをクリックしてライブラリを追加できます)、**初期コンテキスト・ファクトリ・クラス**(初期コンテキストを作成するファクトリ・クラスの完全修飾名)、接続の**URL**、**接続ファクトリJNDI名**(`javax.jms.ConnectionFactory`インタフェースを実装)、**接続先JNDI名**(`javax.jms.Destination`インタフェースを実装)、認証ユーザー名(「**User**」)および認証パスワード(「**Password**」)を指定する必要があります。また、この認証パスワードを暗号化するかどうかを決定することもできます。これを行うには、「**Encrypt password**」チェック・ボックスを選択します。外部(共有)JMS接続を作成する場合は、この外部(共有) JMS接続のファイル名とその場所を選択する必要があります。

認証パスワードの暗号化

認証パスワードを暗号化しないと、構成ファイル(共有接続)またはグラフ自体(内部接続)に格納されて表示可能な状態のままとなります。このため、これらの2つの場所のいずれかで認証パスワードを表示できます。

このことは、グラフまたはコンピュータにアクセスできるユーザーが1人のみである場合は問題ありません。ただし、それ以外の場合は、パスワードを使用してデータベースにアクセスできないように、パスワードを暗号化することをお薦めします。

他のユーザーにグラフを提供する必要がある場合にも、通常は、認証パスワードを渡しません。このため、認証パスワードの暗号化が重要となります。このようにしない場合、この認証パスワードを入手できた人物による侵入アクションやその他の損害のリスクが非常に高くなります。

このため、認証パスワードを暗号化してグラフを提供することが重要であり、可能となっています。このようにすると、管理者の許可なくメッセージを受信したり作成することができなくなります。

認証パスワードを非表示にするには、「**Edit JMS connection**」ウィザードでチェック・ボックスを選択して「**Encrypt password**」を選択し、元の(暗号化対象の)認証パスワードを暗号化する新しい(暗号化)パスワードを入力して、「**Finish**」ボタンをクリックする必要があります。

パスワードを暗号化した場合は、「**Run as**」→「**CloverETL**」を選択してグラフを実行できなくなります。かわりに、グラフを実行するには、「**Run Configurations**」ウィザードを使用する必要があります。この場合、「**Main**」タブで、プロジェクト名、グラフ名およびパラメータ・ファイルを入力するか、または参照して指定し、最も重要なこととして、「**Password**」テキスト領域に暗号化パスワードを入力する必要があります。これで、認証パスワードはすでに暗号化されて読み取ることができなくなり、構成ファイルやグラフに表示されなくなります。

認証パスワードに戻す場合は、**JMS接続**ウィザードに暗号化パスワードを入力し、「**Finish**」をクリックして戻すことができます。

第24章 QuickBase接続

QuickBaseデータベースを使用するには、「QuickBase connection」ウィザードを使用して最初に接続パラメータを定義する必要があります。

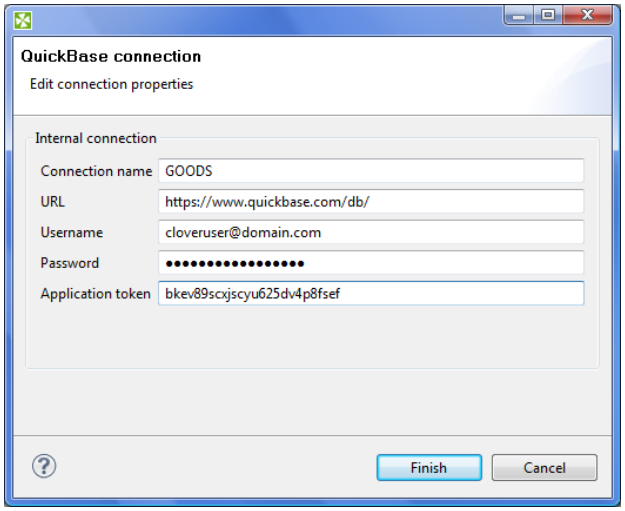


図24.1. 「QuickBase Connection」ダイアログ

接続に名前を付け(「**Connection name**」)、適切なURLを選択します。デフォルトでは、QuickBaseデータベースではAPIを介したSSLアクセスのみが許可されます。

ユーザー名として電子メール・アドレスまたはQuickBaseユーザー・プロファイルの画面名を入力します。必要な**パスワード**は、ユーザー・アカウントに関連します。

アプリケーション・トークンは、データベースに対して作成し、割り当てることができる文字列です。トークンを使用すると、認可されていないユーザーがデータベースに接続することがほぼ不可能になります。

第25章 Lotus接続

Lotusデータベースを使用するには、最初にLotus Domino接続を指定する必要があります。Lotus Domino接続は、内部と外部の両方として作成できます。これらの作成方法は、内部データベース接続の作成(p.173)および外部(共有)データベース接続の作成(p.175)を参照してください。Lotus Domino接続のプロセスは、他のデータベース接続と非常によく似ています。

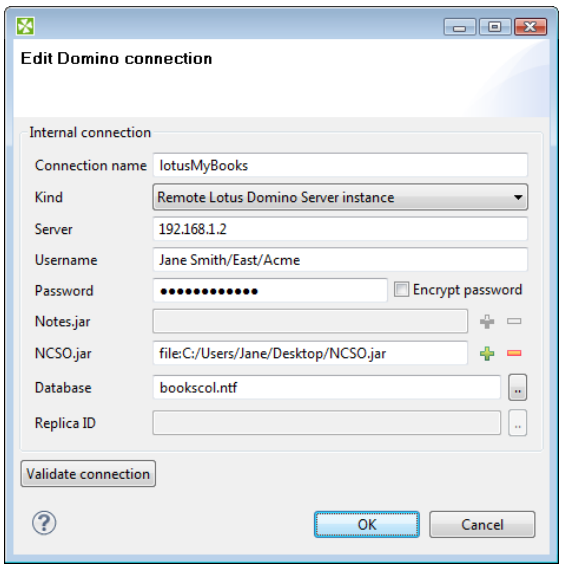


図25.1. Lotus Notes接続ダイアログ

接続に名前を付け(「**Connection name**」)、接続の**種類**を選択します。現在サポートされている**種類**としては、**リモートLotus Dominoサーバー**のみとなります。

リモートLotusサーバーに接続する場合は、「**Server**」フィールドにその場所を指定する必要があります。これは、サーバーのIPアドレスまたはネットワーク名のいずれかにできます。

いずれの種類のサーバーに接続するにも**ユーザー名**を指定する必要があります。ユーザー名は、サーバーのDomino DirectoryのPersonドキュメントと一致している必要があります。

選択したユーザーの**パスワード**も入力する必要があります。アクセス・パスワードは暗号化できます。アクセス・パスワードの暗号化(p.180)を参照してください。

接続を確立するには、Lotus Notesに接続するためのLotusライブラリを指定する必要があります。

リモートLotus Dominoサーバーに接続するには、**Notes.jar**ライブラリを使用できます。これは、Notes/Dominoインストールのプログラム・ディレクトリにあります。たとえば、**c:\lotus\domino\Notes.jar**です。かわりに、**Notes.jar**の軽量バージョンを指定できます。このバージョンには、リモート接続のサポートのみが含まれ、**NCSO.jar**というファイルに格納されています。このファイルは、Lotus Dominoサーバー・インストールにあります。たとえば、**c:\lotus\domino\data\domino\java\NCSO.jar**です。

やり取りするデータの読取り元または書込み先となるデータベースを選択するために、データベースのファイル名を「**Database**」フィールドに入力できます。該当するデータベースの**レプリカID**番号を入力することが、別のオプションとなります。

第26章 Hadoop接続

Hadoopを使用するには、最初にHadoop接続を定義する必要があります。Hadoop接続によって、CloverでHadoop分散ファイル・システム(HDFS)と通信し、HadoopクラスタでMapReduceジョブを実行できます。Hadoop接続は、内部と外部の両方として作成できます。これらの作成方法は、内部データベース接続の作成(p.173)および外部(共有)データベース接続の作成(p.175)を参照してください。Hadoop接続の定義プロセスは、Cloverにおける他の接続と同様であり、「**Create DB connection**」のかわりに「**Create Hadoop connection**」を選択するのみです。

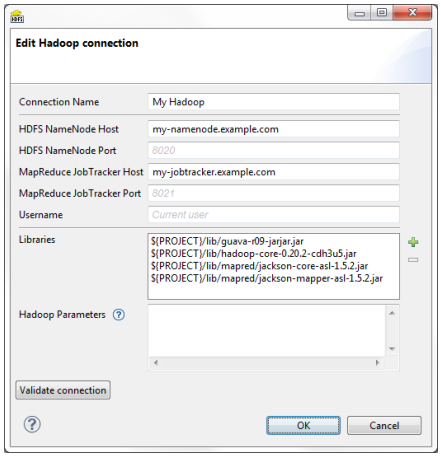


図26.1. Hadoop接続ダイアログ

Hadoop接続プロパティで、「**Connection Name**」と「**HDFS NameNode Host**」は必須です。また、「**Libraries**」もほとんどの場合に必要です。

Connection Name

このフィールドには、このHadoop接続の名前を入力します。新しい接続を作成する場合は、ここに入力する属性名が接続のIDを生成するために使用されます。接続名が単なる情報的ラベルであるのに対し、接続IDは様々なグラフ・コンポーネントからこの接続を参照するために使用されます(たとえば、リモート・ファイルの読取り(p.299)で説明されているように、ファイルURLで)。接続が作成されると、参照の偶発的な破損を避けるために、このダイアログを使用してIDを変更することはできません(すでに作成された接続のIDを変更する必要がある場合は、「Properties」ビューで変更できます)。

HDFS NameNode Host、HDFS NameNode Host Port

HDFS NameNodeのホスト名またはIPアドレスを「**HDFS NameNode Host**」フィールドに指定します。

「**HDFS NameNode Port**」フィールドを空のままにした場合、デフォルト・ポート番号の8020が使用されます。

MapReduce JobTracker Host & Port

JobTrackerのホスト名またはIPアドレスを「**MapReduce JobTracker Host**」フィールドに指定します。このフィールドはオプションです。空のままにした場合、Cloverはこの接続を使用してMapReduceジョブを実行(p.699)できません(その場合もHDFSへのアクセスは正常に機能します)。

「**MapReduce JobTracker Port**」フィールドに入力しない場合、デフォルト・ポート番号の8021が使用されます。

Username

HDFSでのファイル操作の実行と、MapReduceジョブの実行に使用するユーザーの名前です。

- HDFSは通常のUnixファイル・システムと同様に機能します(ファイル所有権、アクセス権)。ただし、HadoopクラスタでKerberosセキュリティが有効になっていない場合、これらの名前はラベルおよび予期しない

データ損失の回避策として役立ちますが、他のユーザーに偽装することが簡単になります。

- ただし、MapReduceジョブは、Cloverグラフを実行するユーザー以外のユーザーとして容易に実行することはできません。MapReduceジョブを実行する必要がある場合は、このフィールドを空のままにします。

デフォルトの**ユーザー名**は、Clover変換グラフを実行するOSアカウント名です。したがって、たとえば、ユーザー名をWindowsログインにすることができ、その場合、HDFS NameNodeを実行しているLinuxでは同じ名前のユーザーを定義する必要はありません。

Libraries

ここでは、Hadoop NameNodeサーバーおよび(オプションで) JobTrackerサーバーと通信するために必要なHadoopライブラリへのパスを指定する必要があります。Hadoopと互換性のないバージョンが多数あるため、Hadoopクラスタのバージョンと一致するライブラリを選択する必要があります。

たとえば、前に示したスクリーン・ショットは、Hadoop分散のCloudera 3更新5バージョンを使用するために必要な、ClouderaのWebサイトからダウンロードできるライブラリを示しています。HDFSを使用するには2つのライブラリ`guava-r09-jarjar.jar`および`hadoop-core-0.20.2-cdh3u5.jar`のみで十分ですが、MapReduceジョブを実行する場合は、さらに2つのライブラリ`jackson-core-asl-1.5.2.jar`および`jackson-mapper-asl-1.5.2.jar`も必要となります。

必要なライブラリを省略した場合、通常、`java.lang.NoClassDefFoundError`が発生します。

異なるバージョンのライブラリを使用するバージョンのHadoopサーバーに接続しようとすると、通常、次のようなエラーが表示されます。

```
org.apache.hadoop.ipc.RemoteException: Server IPC
version 7 cannot communicate with client version 4
```

ライブラリへのパスは、絶対またはプロジェクト相対にできます。グラフ・パラメータも使用できます。

Javaバージョン

Hadoopは、Oracle Java 1.6+での実行のみが保証されていますが、Hadoopの開発者による、Oracle/Sun固有コードの除去が進められています。Hadoop WikiのHadoop Java Versionsを参照してください。

特に、Cloudera 3 distribution of HadoopはOracle Javaでのみ動作します。

Clover Serverでの使用

Clover Serverがデプロイされているアプリケーション・サーバーのクラスパスにライブラリが存在する場合、**ライブラリ**を指定する必要はありません。たとえば、Tomcatアプリケーション・サーバーを使用しており、Hadoopライブラリが`$CATALINA_HOME/lib`ディレクトリに存在する場合です。

ライブラリ・パスを定義する場合、絶対パスはアプリケーション・サーバー上の絶対パスとなります。相対パスはサンドボックス(プロジェクト)相対であり、ライブラリが共有サンドボックスにある場合にのみ機能します。

Hadoop Parameters

このシンプル・テキスト・フィールドには、HDFS操作を微調整する様々なパラメータを指定します。通常、このフィールドは空のままにします。使用可能なプロパティとデフォルト値のリストがHadoopのバージョンの`core-default.xml`ファイルおよび`hdfs-default.xml`ファイルのドキュメントにあります。ドキュメント内でこれらを見つけるのは少し難しいため、次にリンク例を示します。最新リリースのhdfs-default.xmlおよびバージョン0.20.2のhdfs-default.xml。Hadoopクライアントに影響を及ぼすのはリストされているプロパティの一部のみで、ほとんどはサーバー側構成専用です。

ここに入力するテキストは、標準Javaプロパティ・ファイルの形式にする必要があります。マウス・ポインタを疑問符のアイコンの上に置くと、ヒントが表示されます。

Hadoop接続の設定が終了したら、「**Validate connection**」ボタンをクリックして、入力したパラメータを使用してHadoop HDFS NameNodeへの接続が正常に確立されたことをすぐに確認します。ライブラリが(リモート)Clover Serverのサンドボックスにある場合、接続検証は実行できません。

注意

Hadoopバージョン0.21.0以降、HDFSではファイルの追加操作が完全にサポートされています。

YARN (別名MapReduce 2.0、MRv2)への接続

HadoopクラスタでMapReduceフレームワークを最初に生成するかわりにYARNを実行する場合は、Clover Hadoop接続を構成するために次の手順が必要です。

1. 「**MapReduce JobTracker Host**」フィールドに任意の値を入力します。この値は使用されませんが、このHadoop接続に対してMapReduceジョブの実行が可能になるようにします。
2. 次のキーと値のペアを「**Hadoop Parameters**」に追加します。`mapreduce.framework.name=yarn`
3. 「**Hadoop Parameters**」に、`YARN ResourceManager`のホスト名とポートをコロンで区切った形式の値でキー`yarn.resourcemanager.address`を追加します(たとえば、`yarn.resourcemanager.address=my-resourcemanager.example.com:8032`)。

`yarn-default.xml`のデフォルト値が機能しない場合は、`yarn.application.classpath`パラメータも指定する必要があることがあります。この場合、エラーが発生したYARNアプリケーション・コンテナのログに`java.lang.NoClassDefFoundError`が見つかる可能性があります。

第27章 参照表

CloverETL Designerの使用中に、**参照表**を作成して使用することもできます。これらの表は、既知のキーまたはSQL問合せを使用して格納されたデータへの高速アクセスを可能にするデータ構造です。この方法で、データベースまたはデータ・ファイルを参照する必要性を削減できます。

警告

参照表は、CTLテンプレートの`init()`、`preExecute()`または`postExecute()`関数およびJavaインタフェースの同じメソッドで使用しないでください。

参照表に格納されたすべてのデータ・レコードはファイル、データベースに保存されるか、またはメモリーにキャッシュされます。

メタデータおよびデータベース接続の場合と同様、参照表も内部または外部(共有)にできます。これらは2つの方法で作成できます。

作成できる各参照表は次のとおりです。

- **内部**: 内部参照表(p.197)を参照してください。

  内部参照表は次のようにできます。

  - **外部化**: 内部参照表の外部化(p.197)を参照してください。
  - **エクスポート**: 内部参照表のエクスポート(p.199)を参照してください。
- **外部(共有)**: 外部(共有)参照表(p.200)を参照してください。

  外部(共有)参照表は次のようにできます。

  - **グラフへのリンク**: 外部(共有)参照表のリンク(p.200)を参照してください。
  - **内部化**: 外部(共有)参照表の内部化(p.201)を参照してください。

参照表のタイプは次のとおりです。

- **簡易参照表**: 簡易参照表(p.202)を参照してください。
- **データベース参照表**: データベース参照表(p.205)を参照してください。
- **範囲参照表**: 範囲参照表(p.206)を参照してください。
- **永続参照表**: 永続参照表(p.208)を参照してください。
- **Aspell参照表**: Aspell参照表(p.209)を参照してください。

CloverETL Cluster環境における参照表

クラスタ化されたグラフが処理される方法や、クラスタ化されたグラフが複数の個別のグラフに分割され、クラスタ・ノードに分配される方法を理解するには、クラスタ環境で参照表がどのように機能するかを理解する必要があります。これらの詳細は、CloverETL Serverのドキュメントの「データのパラレル処理」を参照してください。つまり、クラスタ化されたグラフは変換計画に従って複数のインスタンスで実行され、これらはワーカー・グラフと呼ばれます。変換計画は変換分析の結果であり、この分析では、コンポーネント割当て、パーティション化されたサンドボックスの使用、およびクラスタ化されたコンポーネントの出現が考慮されます。変換計画は、いくつのグラフのインスタンスが、どのクラスタ・ノードで実行されるかを示します。さらに、変換計画は、クラスタ化された

実行のためにワーカー・グラフを更新する方法、特定のワーカーで実際に実行されるコンポーネント、および削除されるコンポーネントを示します。

CloverETL Serverのクラスタ環境では、参照表に対する特別なサポートは提供されません。クラスタ化された各グラフ・インスタンスでは、独自の参照表のセットが作成されます。参照表インスタンスは互いに連携しません。このため、たとえば、SimpleLookupTableを使用する場合、クラスタ化されたグラフの各インスタンスには独自のSimpleLookupTableインスタンスがあり、これによって、指定されたデータ・ファイルから個別にデータをロードします。したがってデータ・ファイルはクラスタ化された各グラフによって読み取られ、各インスタンスは個別のキャッシュ・レコードのセットを持ちます。DBLookupTableはクラスタ環境でシームレスに機能し、データベース・レスポンスの内部キャッシュは各ワーカー・グラフによって個別に管理されます。

LookupTableReaderWriterコンポーネントを使用して参照表にデータ・レコードを書き込む場合は注意してください。参照表の更新はローカルでのみ実行されるため、どのワーカーが書込みを行うかを考慮する必要があります。このため、参照の更新が必要になるすべてのワーカーでLookupTableReaderWriterコンポーネントが実行されるようにしてください。

内部参照表

内部参照表はグラフの一部で、グラフに含まれており、そのソース・タブで確認できます。

内部参照表の作成

内部参照表を作成する場合は、「**Outline**」ペインで「**Lookups**」項目を選択し、この項目を右クリックして、「**Lookup tables**」→「**Create lookup table**」を選択する必要があります。「**Lookup table**」ウィザードが開きます。参照表のタイプ(p.202)を参照してください。参照表のタイプを選択し、「**Next**」をクリックした後、選択した参照表のプロパティを指定できます。参照表および参照表のタイプの詳細は、次の対応する各項を参照してください。

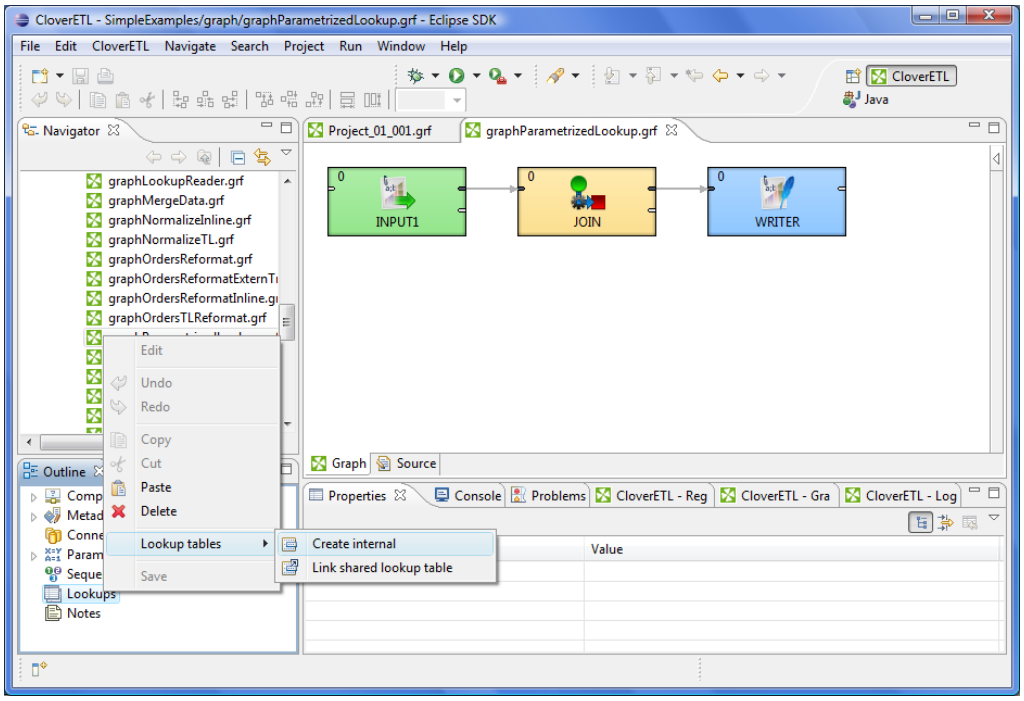


図27.1. 内部参照表の作成

内部参照表の外部化

内部参照表をグラフの一部として作成した後、内部参照表を外部(共有)参照表に変換できます。これにより、同じ参照表を他のグラフで使用できるようになります。

内部参照表を外部(共有)ファイルに外部化する場合は、「**Outline**」ペインの「**Lookups**」グループ内で目的の内部参照表項目を右クリックし、コンテキスト・メニューから「**Externalize lookup table**」をクリックします。参照表に内部メタデータが含まれている場合は、次のウィザードが表示されます。

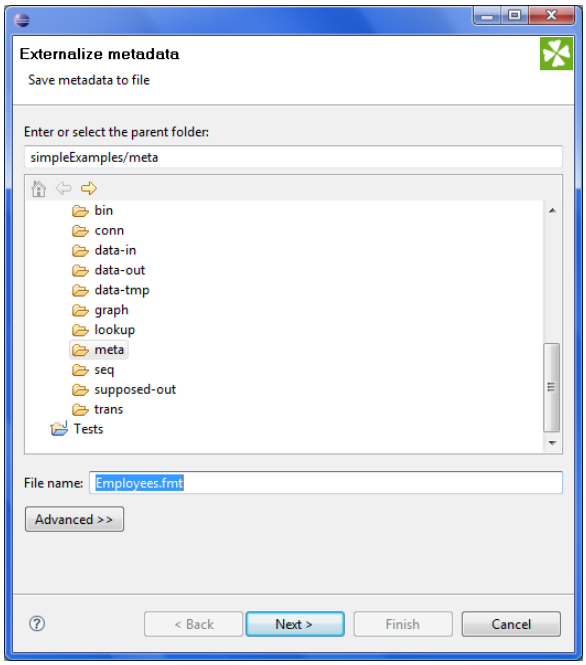


図27.2. 外部化ウィザード

このウィザードでは、プロジェクトのmetaサブフォルダ、および選択した参照表に割り当てられている内部メタデータを外部化する新しい外部(共有)メタデータ・ファイルのファイル名が提案されます。必要に応じて(同じ名前のファイルがすでに存在する場合)、外部(共有)メタデータ・ファイルの提案された名前を変更できます。「**Next**」をクリックした後、データベース接続を外部化するための同様のウィザードが開きます。メタデータと同じ手順を実行します。最後に、参照表用のウィザードが開きます。ここでは、プロジェクトのlookupフォルダがこの新しい外部(共有)参照表ファイルの場所として表示され、「**OK**」をクリックできます。必要に応じて(同じ名前のファイルがすでに存在する場合)、参照表ファイルの提案された名前を変更できます。

その後、内部メタデータ(と内部接続)および参照表項目がそれぞれ「**Outline**」ペインの「**Metadata**」(と「**Connections**」)および「**Lookups**」グループからなくなり、同じ場所に、新しく作成された外部(共有)メタデータ(と接続構成ファイル)、および対応するグループ内の参照表ファイルがすでにリンクされた状態で、新しいエントリが表示されます。プロジェクトのmeta、connおよびlookupサブフォルダに、これらの同じファイルがそれぞれ表示されることを「**Navigator**」ペインで確認できます。

参照表に外部(共有)メタデータ(および外部データベース接続)のみが含まれる場合は、(参照表を外部化するための)最後のウィザードのみが表示されます。ここでは、プロジェクトのlookupフォルダがこの新しい外部(共有)参照表ファイルの場所として表示され、「**OK**」をクリックします。必要に応じて(同じ名前のファイルがすでに存在する場合)、参照表ファイルの提案された名前を変更できます。

内部参照表が外部化された後、内部項目は「**Outline**」ペインの「**Lookups**」グループに表示されなくなりますが、同じ場所に、すでにリンク済の新しい参照表ファイル項目が表示されます。プロジェクトのlookupサブフォルダに、同じファイルが表示されることを「**Navigator**」ペインで確認できます。

複数の内部参照表項目を一度に外部化することもできます。このことを行うには、「**Outline**」ペインで項目を選択して右クリックした後、コンテキスト・メニューから「**Externalize lookup table**」を選択します。選択したすべての参照表が(必要に応じて、メタデータまたは参照表に割り当てられた接続、あるいはその両方とともに)外部化されるまで、前述のプロセスが繰り返されます。

[Shift]を押しながら**下矢印**または**上矢印**キーを押すと、隣接する参照表項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各接続項目を**クリック**します。

内部参照表のエクスポート

このケースは、グラフ外の参照表ファイルを外部化されたファイルと同じ方法で作成するときに、ファイルが元のグラフにリンクされないことを除き、内部参照表の外部化の場合と似ています。作成されるのは外部参照表ファイルのみ(場合によってはメタデータまたは接続、あるいはその両方)です。前の項で説明したように、その後に、それらのファイルを他のグラフで外部(共有)参照表ファイルとして使用できます。

内部参照表を外部(共有)参照表にエクスポートするには、「**Outline**」ペインで内部参照表項目を右クリックし、コンテキスト・メニューから「**Export lookup table**」をクリックします。新しく作成された外部ファイルに対して、対応するプロジェクトの`lookup`フォルダが提案されます。提案されたものとは異なる名前をファイルに付けることもでき、「**Finish**」をクリックすると、ファイルが作成されます。

その後、「**Outline**」ペインの`lookups`フォルダは同じままですが、「**Navigator**」ペインの`lookup`フォルダには、新しく作成された参照表ファイルが表示されます。

外部化に関する前の項で説明しているのと同様の方法で、選択した複数の内部参照表をエクスポートできます。

外部(共有)参照表

前述のように、外部(共有)参照表は複数のグラフ間で共有できます。アクセスは簡単になりますが、これにより、グラフのソースから削除されます。

外部(共有)参照表の作成

外部(共有)参照表を作成するには、「**File**」→「**New**」→「**Other...**」を選択します。

次に、**CloverETL**項目を展開し、参照表項目および「**Next**」をクリックするか、または**参照表**項目をダブルクリックする必要があります。

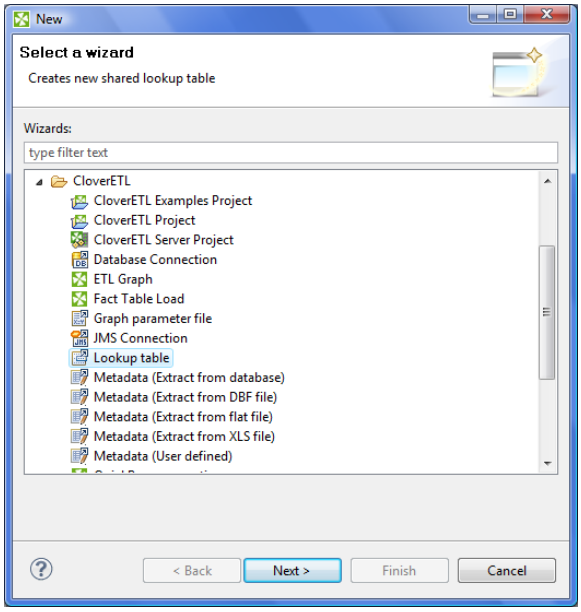


図27.3. 参照表項目の選択

その後、「**New lookup table**」ウィザードが開きます。参照表のタイプ(p.202)を参照してください。このウィザードで、目的の参照表タイプを選択し、定義して確認する必要があります。また、`lookup`フォルダ内の参照表のファイル名も選択する必要があります。「**Finish**」をクリックした後、外部(共有)データベース接続が作成されます。

外部(共有)参照表のリンク

作成した後(前の項を参照)、外部(共有)参照表を複数のグラフにリンクできます。「**Lookups**」グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから「**Lookup tables**」→「**Link shared lookup table**」を選択する必要があります。その後、プロジェクト・コンテンツが表示された**ファイル選択**ウィザードが開きます。このウィザードで`lookup`フォルダを展開し、このウィザードに含まれているすべてのファイルから目的の参照表ファイルを選択する必要があります。

複数の外部(共有)参照表ファイルを一度にリンクすることもできます。このことを行うには、「**Lookups**」グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから「**Lookup tables**」→「**Link shared lookup table**」を選択します。その後、プロジェクト・コンテンツが表示された**ファイル選択**ウィザードが開きます。このウィザードで`lookup`フォルダを展開し、このウィザードに含まれているすべてのファイルから目的の参照表ファイルを選択する必要があります。**[Shift]**を押しながら**下矢印**または**上矢印**キーを押すと、隣接するファイル項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各ファイル項目を**クリック**します。

外部(共有)参照表の内部化

外部(共有)参照表ファイルを作成してリンクした後、この参照表をグラフに挿入する場合は、内部参照表に変換する必要があります。これにより、その構造をグラフ自体で参照できるようになります。

任意のリンク済外部(共有)参照表ファイルを内部化して内部参照表にするには、「**Outline**」ペインで外部(共有)参照表項目を右クリックし、コンテキスト・メニューから「**Internalize connection**」をクリックします。

このことを行った後、次のウィザードが開き、参照表に割り当てられているメタデータまたはそのDB接続(**データベース参照表**の場合)、あるいはその両方も内部化するかどうかを決定できます。

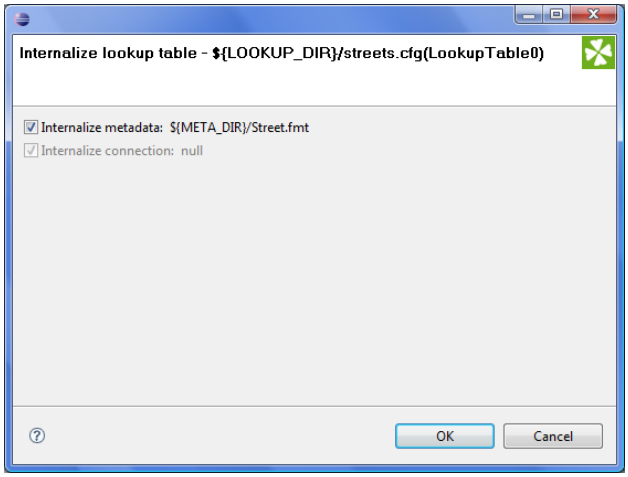


図27.4. 参照表の内部化ウィザード

チェック・ボックスを選択するか、または選択を解除したままで、「**OK**」をクリックします。

その後、選択したリンク済外部(共有)参照表項目は「**Outline**」ペインの「**Lookups**」グループからなくなり、同じ場所に、新しく作成された内部参照表項目が表示されます。参照表に割り当てられているリンク済外部(共有)メタデータも内部化することを決定した場合、それらの項目は内部メタデータ項目に変換され、「**Outline**」ペインの「**Metadata**」グループで確認できます。

ただし、元の外部(共有)参照表ファイルは`lookup`サブフォルダにそのまま存在します。これは「**Navigator**」ペインのこのフォルダで確認できます。

複数のリンク済外部(共有)参照表ファイルを一度に内部化することもできます。このことを行うには、「**Outline**」ペインで目的のリンク済外部(共有)参照表項目を選択します。その後は、選択した参照表ごとに前述のプロセスを繰り返すのみです。**[Shift]**を押しながら**下矢印**または**上矢印**キーを押すと、隣接する項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各項目を**クリック**します。

参照表のタイプ

「**New lookup table**」ウィザードが開いたら、目的の参照表タイプを選択する必要があります。ラジオ・ボタンを選択し、「**Next**」をクリックすると、対応するウィザードが開きます。

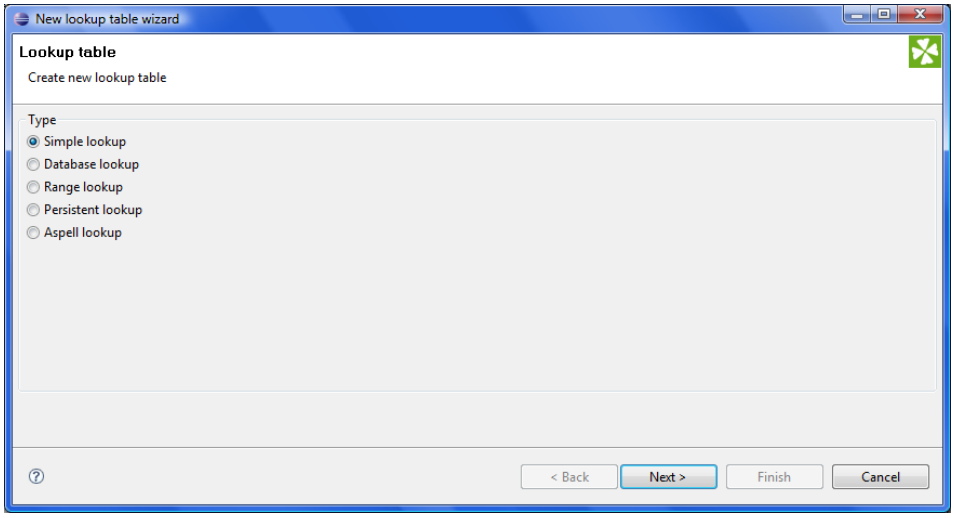


図27.5.「Lookup Table」ウィザード

簡易参照表

この参照表に格納されるすべてのデータ・レコードはメモリーに保存されます。このため、参照表のすべてのデータ・レコードを保存するには、十分なメモリーが使用可能である必要があります。データ・レコードがデータ・ファイルから簡易参照表にロードされる場合、使用可能なメモリーのサイズは、データ・ファイルのサイズのおよそ6倍以上であることが必要です。ただし、この乗数は、データ・ファイルに格納されたデータ・レコードのタイプに応じて異なります。

「**Simple lookup table**」ウィザードで、必要なプロパティを設定する必要があります。

「**Table definition**」タブで、参照表に**名前**を付け、表からデータ・レコードを検索するために使用する、対応する**メタデータ**および**キー**を選択する必要があります。参照表の**キャラクタ・セット**および**初期サイズ**(デフォルトは512)を選択することもできます。デフォルト値は、`defaultProperties`で`Lookup.LOOKUP_INITIAL_CAPACITY`値を変更することによって変更できます。

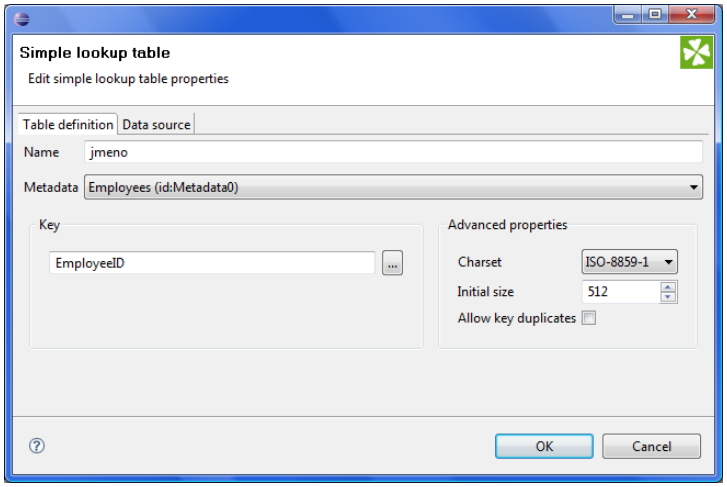


図27.6.「Simple Lookup Table」ウィザード

「**Key**」領域で右側のボタンをクリックすると、**キー**の選択に役立つ「**Edit key**」ウィザードが表示されます。

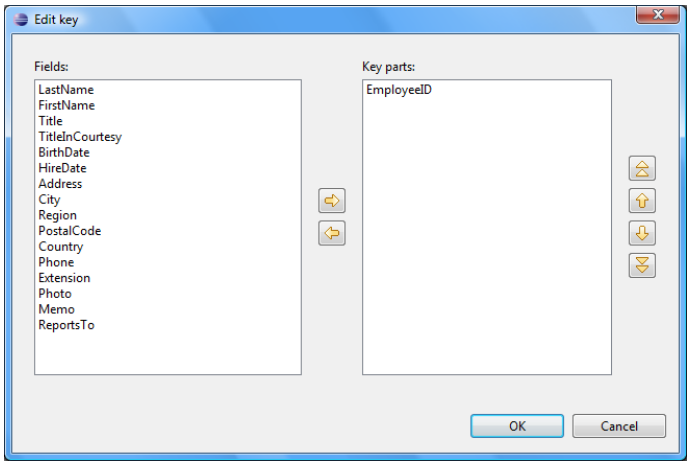


図27.7. 「Edit Key」ウィザード

「**Field**」ペインでいくつかのフィールド名を強調表示し、**右矢印**ボタンをクリックして、フィールド名を「**Key parts**」ペインに移動できます。続けてさらにフィールド名を「**Key parts**」ペインに移動できます。**上**または**下**ボタンをクリックして、「Key parts」リスト内のフィールド名の位置を変更することもできます。リスト内でより上位にあるキー部分は、優先度がより高くなります。終了したら、「**OK**」をクリックするのみです。(フィールド名を強調表示し、**左矢印**ボタンをクリックして削除することもできます。)

「**Data source**」タブで、ファイルの**URL**を検索するか、または「**Edit data**」ボタンをクリックした後にグリッドに入力できます。「**OK**」をクリックすると、「**Data text**」領域にデータが表示されます。

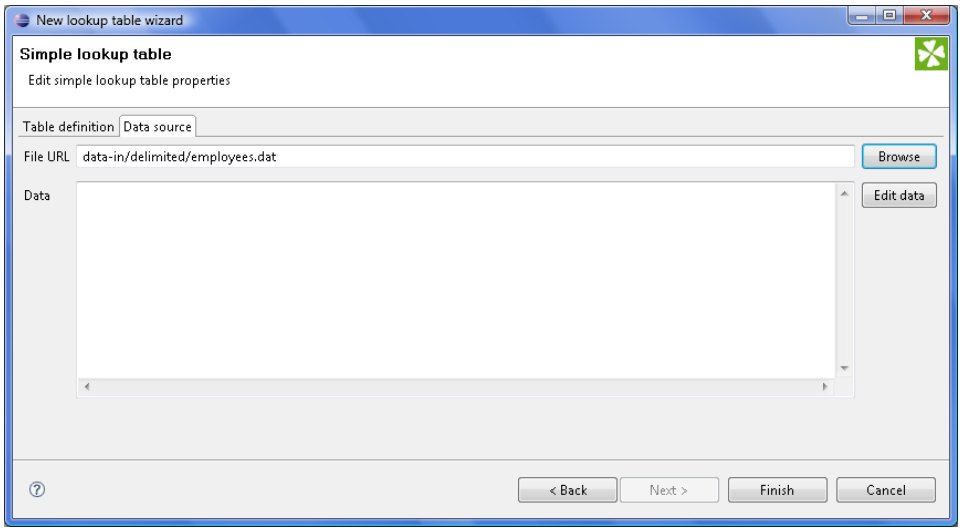


図27.8. ファイルURLが表示された「Simple Lookup Table」ウィザード

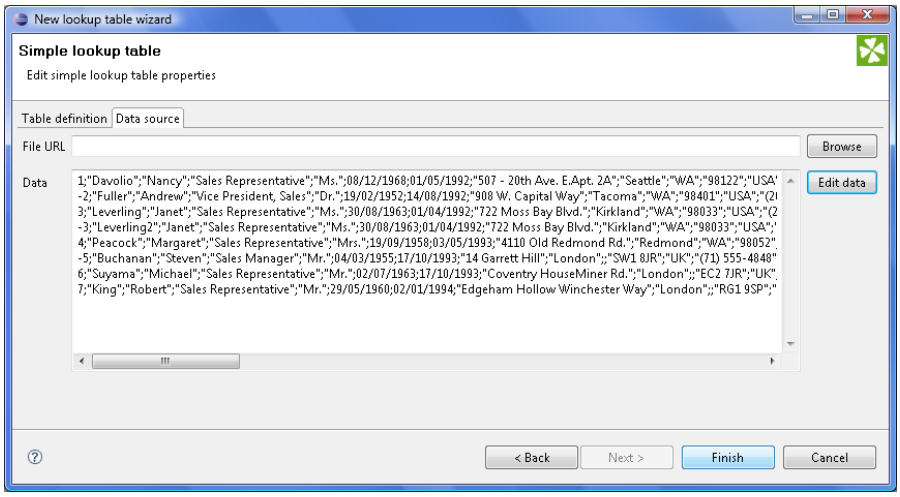


図27.9. データが表示された「Simple Lookup Table」ウィザード

「**Edit data**」ボタンをクリックした後にデータを設定または編集できます。

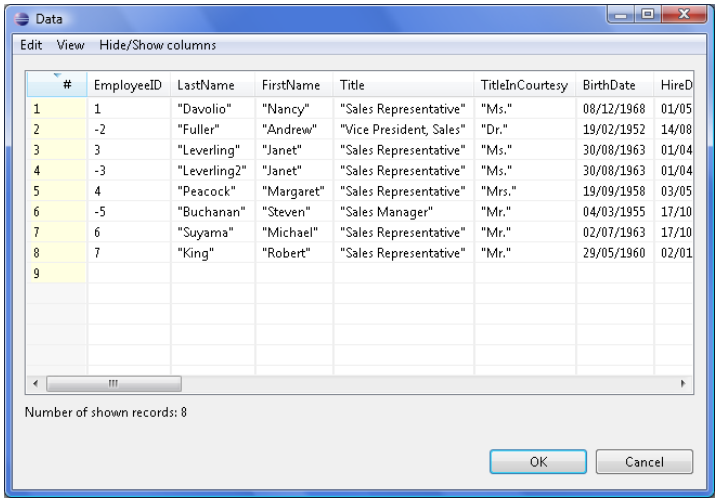


図27.10. データの変更

すべてが完了したら、「**OK**」および「**Finish**」をクリックできます。

簡易参照表には、グリッドで直接指定したデータ、ファイル内のデータまたは**LookupTableReaderWriter**を使用して読み取れるデータを含めることができます。

重要

「**Allow key duplicates**」チェック・ボックスを選択することもできます。これにより、同じキー値を持つ複数のデータ・レコード(重複レコード)を許可します。

キー値ごとに1つのレコードのみが**簡易参照表**に含められるようにする場合は、前述のチェック・ボックスを選択解除(デフォルト設定)のままにします。1つのレコードのみが選択された場合、新しいレコードによって古いレコードが上書きされます。そのような場合は、最後のレコードが**簡易参照表**に含まれる唯一のレコードとなります。

データベース参照表

このタイプの参照表はデータベースと連携し、SQL問合せを使用してデータベースからデータをアンロードします。データベース参照表は、指定されたデータベース表からデータを読み取ります。この参照表からレコードを検索するキーは、問合せの`where fieldName = ? [and ...]`部分です。データベースからアンロードされたデータ・レコードは、LRU順序(最も長い間使用されていない項目を最初に破棄)に従ってメモリーにキャッシュできます。キャッシュするには、そのようなレコードの数(**最大キャッシュ・レコード**)を指定する必要があります。あるキー値に該当するレコードがデータベースに見つからない場合は、「**Store negative key response**」チェック・ボックスを選択すると、このレスポンスを保存できます。これにより、同じキー値が再度指定された場合、参照表ではデータベース表を検索しません。**データベース参照表**では、重複レコード(同じキー値を持つ複数のレコード)を使用できます。

データベース参照表を作成または編集する場合は、「**Database lookup**」ラジオ・ボタンを選択し、「**Next**」をクリックする必要があります。(図27.5.「Lookup Table」ウィザード(p.202)を参照してください。)

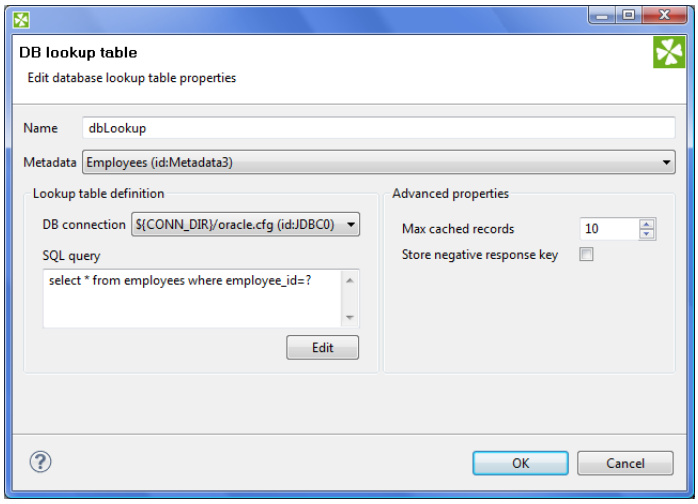


図27.11.「Database Lookup Table」ウィザード

次に、「**Database Lookup Table**」ウィザードで、選択した参照表に**名前**を付け、**メタデータ**および**DB接続**を指定する必要があります。

Javaで記述された変換には**メタデータ**定義は必要ありません。単に**no metadata**オプションを選択します。ただし、CTLでは、**メタデータ**を指定する必要があります。

また、データベースからデータ・レコードを検索する働きをするSQL問合せを入力または編集する必要もあります。問合せを編集する場合は、「**Edit**」ボタンをクリックする必要があり、データベース接続が有効で機能している場合は、「**Query**」ウィザードが表示され、そこでデータベースの参照、問合せの作成、検証および結果データの表示を行うことができます。

重要

参照表キーを指定するには、問合せの末尾に`where fieldName = ? [and ...]`文を追加し、`fieldName`は、たとえば、EMPLOYEE_IDです。指定されたキーに一致するレコードによって問合せ内の疑問符文字が置き換えられます。

その後、「**OK**」および「**Finish**」をクリックできます。データベースからのメタデータの抽出の詳細は、データベースからのメタデータの抽出(p.146)を参照してください。

範囲参照表

範囲参照表は、レコードの複数のフィールドによって範囲が作成される場合にのみ作成できます。つまり、フィールドが同じデータ型で、かつ開始と終了の両方に割り当てることができる場合です。このことは、次の例で確認できます。

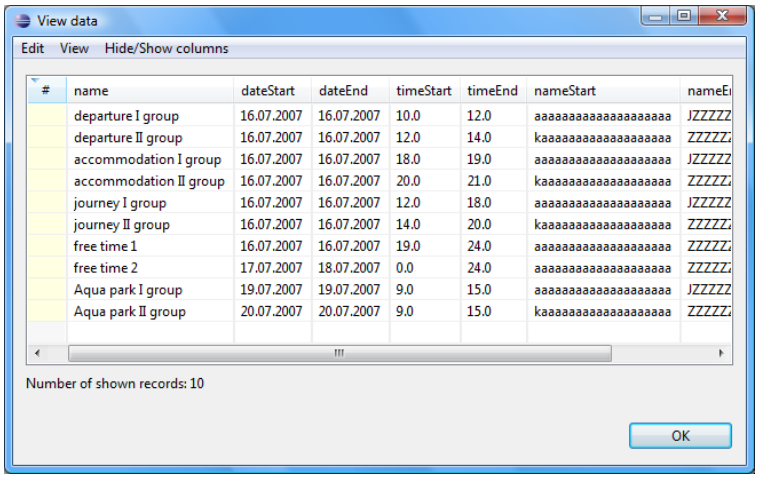


図27.12. 範囲参照表に適したデータ

範囲参照表を作成する場合は、「**Range lookup**」ラジオ・ボタンを選択し、「**Next**」をクリックする必要があります。(図27.5.「Lookup Table」ウィザード(p.202)を参照してください。)

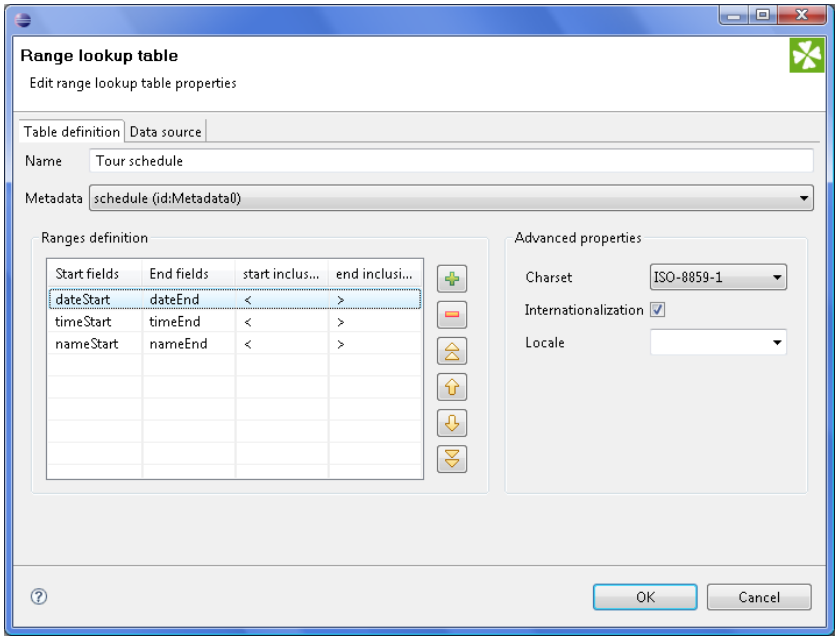


図27.13.「Range Lookup Table」ウィザード

次に、「**Range lookup table**」ウィザードで、選択した参照表に**名前**を付け、**メタデータ**を指定する必要があります。

キャラクタ・セットを選択し、**内部化**するかどうか、およびどの**ロケール**を使用するかを決定できます。

右側のボタンをクリックして、いずれかの項目を追加したり、上下に移動することができます。

これらの範囲に開始または終了フィールドを含めるかどうかも選択する必要があります。そのためには、ウィザードの対応する列でいずれかの範囲を選択し、クリックします。

「**Data source**」タブに切り替えると、ファイルを指定するか、またはグリッドにデータを直接入力できます。**LookupTableReaderWriter**を使用して参照表にデータを書き込むこともできます。

「**Edit**」ボタンをクリックすると、参照表に含まれているデータを編集できます。最後に、「**OK**」および「**Finish**」をクリックするのみです。

重要

範囲参照表には、同じキー値の最初のレコードのみが含まれます。

永続参照表

商用参照表

この参照表は商用であり、**CloverETL Designer**の商用ライセンスでのみ使用できます。

このタイプの参照表は、より多くのデータ・レコードに役立ちます。**簡易参照表**とは異なり、データは**永続参照表**のウィザードで指定したファイルに格納されます。これらのファイルは、jdbm形式(http://jdbm.sourceforge.net)となります。

「**Persistent lookup table**」ウィザードで、必要なプロパティを設定します。参照表に**名前**を付け、対応する**メタデータ**を選択し、参照表のデータ・レコードが格納される**ファイル**および表からデータ・レコードを検索するために使用される**キー**を指定します。

このファイルには、最初に作成してから使用する必要がある内部形式があります。ファイルを指定すると、2つのファイルが作成され、データが格納されます(dbおよびlg拡張子)。この表への書込みのたびに、同じキー値を持つ新しいレコードがこのファイルに格納される可能性があります。古いレコードが新しいレコードで上書きされるようにするには、「**Replace**」チェック・ボックスを選択する必要があります。

また、トランザクションを無効にするかどうかを決定することもできます(「**Disable transactions**」)。このことはグラフのパフォーマンスの向上には役立ちますが、データ損失が発生する可能性があります。

複数の**拡張プロパティ**(「**Commit interval**」、「**Page size**」および「**Cache size**」)を選択できます。「**Commit interval**」では一度にコミットされるレコードの数を定義します。「**Page size**」を指定することにより、ノード当たりのエントリ数を定義します。「**Cache size**」ではキャッシュ内のレコードの最大数を指定します。

重要

永続参照表には同じキー値を持つ複数のレコードは含まれません。このような重複は許可されません。

「**Replace**」チェック・ボックスを選択した場合、同じキー値を持つすべてのレコードの最後のレコードが参照表に含まれる唯一のレコードとなります。これに対し、このチェック・ボックスを選択解除のままにした場合は、最初のレコードが含まれる唯一のレコードとなります。

最後に、「**OK**」および「**Finish**」をクリックするのみです。

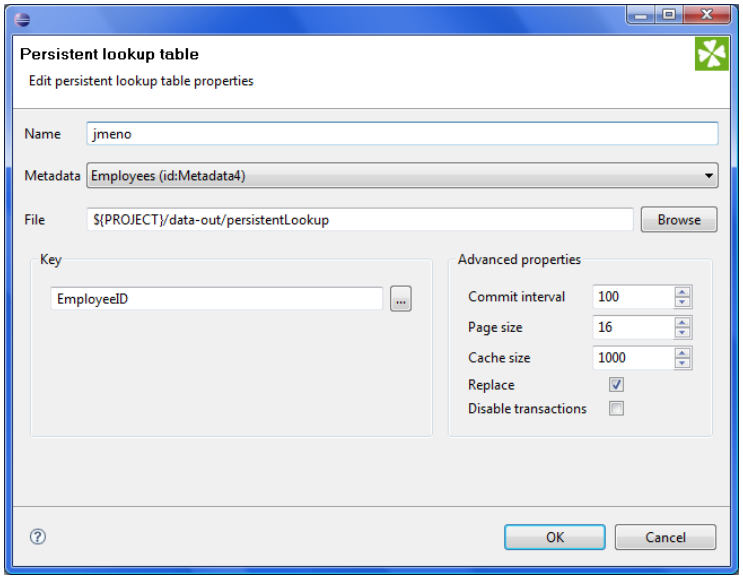


図27.14. 「Persistent Lookup Table」ウィザード

Aspell参照表

商用参照表

この参照表は商用であり、**CloverETL Designer**の商用ライセンスでのみ使用できます。

この参照表に格納されるすべてのデータ・レコードはメモリーに保存されます。このため、参照表のすべてのデータ・レコードを保存するには、十分なメモリーが使用可能である必要があります。データ・レコードがデータ・ファイルからAspell参照表にロードされる場合、使用可能なメモリーのサイズは、データ・ファイルのサイズのおよそ7倍以上であることが必要です。ただし、この乗数は、データ・ファイルに格納されたデータ・レコードのタイプに応じて異なります。

類似しているものの完全には同一でないデータ・レコードを扱っている場合は、このタイプの参照表を使用する必要があります。たとえば、**Aspell参照表**は住所に対して使用できます。

「Aspell lookup table」ウィザードで、必要なプロパティを設定します。参照表に**名前**を付け、対応する**メタデータ**を選択し、表からデータ・レコードを検索するために使用される**「Lookup key field」**(文字列データ型であることが必要です)を選択する必要があります。

参照表のデータ・レコードが格納される**データ・ファイルのURL**およびデータ・ファイルのキャラクタ・セット(**「Data file charset」**)も指定する必要があります。デフォルトのキャラクタ・セットは`ISO-8859-1`です。

参照表で使用されるしきい値を設定できます(**「Spelling threshold」**)。これは0より大きくする必要があります。しきい値が大きくなるほど、スペル・エラーに対するコンポーネントの耐性が高くなります。デフォルト値は`230`です。これは問合せから結果への`edit_distance`値です。この値が指定された制限より大きい語は、結果に含められません。

次の各操作のデフォルト・コストを変更することもできます(**「Edit costs」**)。

- **大文字/小文字コスト**

  1つの文字の大文字小文字が変更される場合に使用されます。

- **入替えコスト**

  文字列内で1つの文字が別の文字と入れ替えられる場合に使用されます。

- **削除コスト**

  文字列から1つの文字が削除される場合に使用されます。

- **挿入コスト**

  文字列に1つの文字が挿入される場合に使用されます。

- **置換コスト**

  1つの文字が別の文字に置換される場合に使用されます。

発音区別文字のある文字とない文字を同一とみなすかどうかを決定する必要があります。このことを行うには、**「Remove diacritical marks」**属性の値を設定する必要があります。`edit_distance`値の計算前に発音区別文字が削除されるようにするには、この値を`true`に設定する必要があります。これにより、発音区別文字のある文字がそれらのラテン文字と同じであるとみなされます。(デフォルト値は`false`です。デフォルトでは、発音区別文字のある文字はそれらのない文字とは異なるとみなされます。)

結果に推測を含めるには、**「Include best guesses」**を`true`に設定します。デフォルト値は`false`です。最適な推測は、`edit_distance`値が**「Spelling threshold」**より高く、かつ他により適した同様のものがない語です。

最後に必要なのは、**「OK」**および**「Finish」**をクリックするのみです。

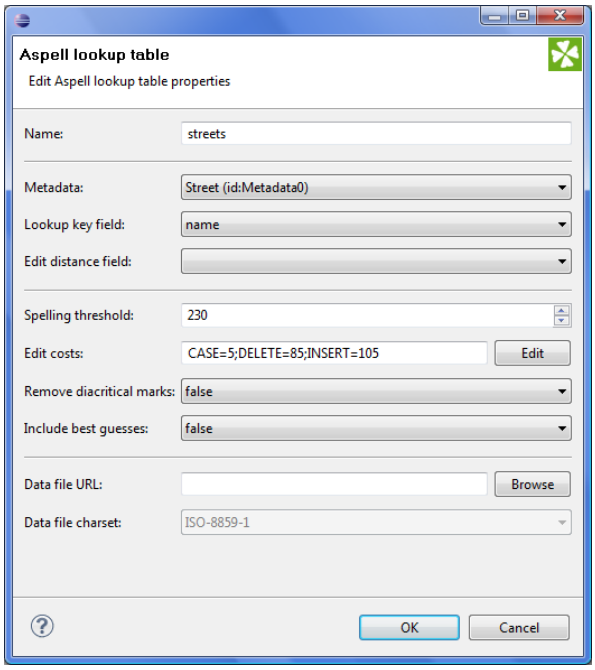


図27.15.「Aspell Lookup Table」ウィザード

重要

参照表とエッジ値との間の距離を知るには、数値タイプの別のフィールドを参照表メタデータに追加する必要があります。このフィールドを「**Autofilling**」(`default_value`)に設定します。

このフィールドを「**Edit distance field**」コンボで選択します。

Aspell参照表を**LookupJoin**で使用している場合は、この参照表フィールドを出力ポート0の対応するフィールドにマップできます。

これにより、参照表の指定した「**Edit distance field**」に格納される値が別の指定したフィールドへの出力に送信されます。

第28章 シーケンス

CloverETL Designerには、たとえば、レコードの番号付けに使用できる番号のシーケンスを作成するように設計されたツールが含まれています。レコードで、新しいフィールドが作成され、シーケンスから取得された番号が入力されます。

警告

シーケンスは、CTLテンプレートの`init()`、`preExecute()`または`postExecute()`関数およびJavaインタフェースの同じメソッドで使用しないでください。

作成できる各シーケンスは次のとおりです。

- **内部**: 内部シーケンス(p.212)を参照してください。

  内部シーケンスは次のようにできます。

  - **外部化**: 内部シーケンスの外部化(p.212)を参照してください。
  - **エクスポート**: 内部シーケンスのエクスポート(p.213)を参照してください。

- **外部(共有)**: 外部(共有)シーケンス(p.214)を参照してください。

  外部(共有)シーケンスは次のようにできます。

  - **グラフへのリンク**: 外部(共有)シーケンスのリンク(p.214)を参照してください。
  - **内部化**: 外部(共有)シーケンスの内部化(p.214)を参照してください。

シーケンスの編集ウィザードは、シーケンスの編集(p.215)で説明しています。

内部シーケンス

内部シーケンスはグラフに格納され(そのデータが格納されているファイルを除き)、グラフに表示されます。1つのシーケンスを複数のグラフで使用する場合は、外部(共有)シーケンスを使用することをお薦めします。他のユーザーにグラフを提供する場合は、内部シーケンスを作成することをお薦めします。これはメタデータ、接続およびパラメータの場合と同様です。

内部シーケンスの作成

内部シーケンスを作成する場合は、「**Outline**」ペインで「**Sequence**」項目を右クリックし、コンテキスト・メニューから「**Sequence**」→「**Create sequence**」を選択する必要があります。その後、「**Sequence**」ウィザードが表示されます。シーケンスの編集(p.215)を参照してください。

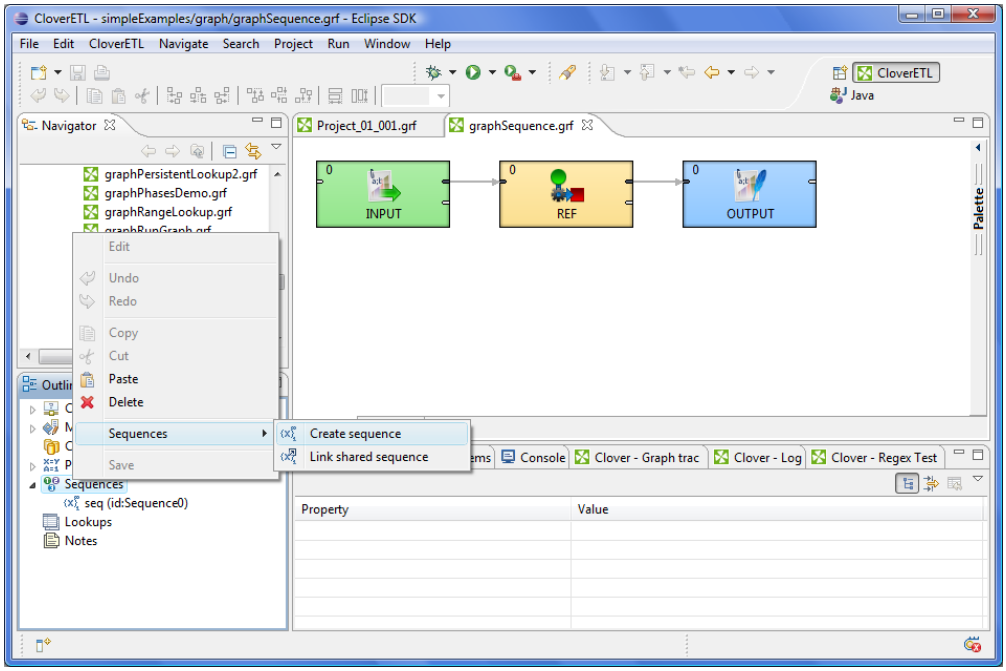


図28.1. シーケンスの作成

内部シーケンスの外部化

グラフの一部として内部シーケンスを作成すると、シーケンスはグラフに含められますが、これを外部(共有)シーケンスに変換できます。これにより、同じシーケンスを複数のグラフで使用できるようになります(複数のグラフで共有する場合)。

任意の内部シーケンス項目を外部化して外部(共有)ファイルにするには、「**Outline**」ペインで内部シーケンス項目を右クリックし、コンテキスト・メニューから「**Externalize sequence**」を選択します。その後、ワークスペースのプロジェクトのリストが表示された新しいウィザードが開き、対応するプロジェクトの`seq`フォルダがこの新しい外部(共有)シーケンス・ファイルの場所として提案されます。必要に応じて(同じ名前のファイルがすでに存在する場合)、シーケンス・ファイルの提案された名前を変更できます。次に、「**OK**」をクリックできます。

その後、内部シーケンス項目は「**Outline**」ペインの「**Sequences**」グループからなくなり、同じ場所に、すでにリンクされた状態で、新しく作成された外部(共有)シーケンス・ファイルが表示されます。選択したプロジェクトの`seq`フォルダに、同じシーケンス・ファイルが表示されることを「**Navigator**」ペインで確認できます。

複数の内部シーケンス項目を一度に外部化することもできます。このことを行うには、「**Outline**」ペインで内部シーケンス項目を選択して右クリックした後、コンテキスト・メニューから「**Externalize sequence**」を選択します。

その後、ワークスペースの対応するプロジェクトのseqフォルダが表示された新しいウィザードが開き、このフォルダがこの新しい外部(共有)シーケンス・ファイルの場所として提案されます。必要に応じて(同じ名前のファイルがすでに存在する場合)、シーケンス・ファイルの提案された名前を変更できます。次に、「**OK**」をクリックできます。

その後、選択した内部シーケンス項目は「**Outline**」ペインの「**Sequences**」グループからなくなり、同じ場所に、すでにリンクされた状態で、新しく作成された外部(共有)シーケンス・ファイルが表示されます。選択したプロジェクトに、同じシーケンス・ファイルが表示されることを「**Navigator**」ペインで確認できます。

[Shift]を押しながら**下矢印**または**上矢印**キーを押すと、隣接するシーケンス項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各シーケンス項目を**クリック**します。

内部シーケンスのエクスポート

このケースは、内部シーケンスの外部化と似ています。ただし、グラフ外部のシーケンス・ファイルを外部化されたファイルと同じ方法で作成しますが、ファイルは元のグラフにリンクされません。作成されるのは外部シーケンス・ファイルのみです。前の項で説明したように、その後に、それらのファイルを他のグラフで外部(共有)シーケンス・ファイルとして使用できます。

内部シーケンスを外部(共有)シーケンスにエクスポートするには、「**Outline**」ペインでいずれかの内部シーケンス項目を右クリックし、コンテキスト・メニューから「**Export sequence**」をクリックします。新しく作成された外部ファイルに対して、対応するプロジェクトのseqフォルダが提案されます。提案されたものとは異なる名前をファイルに付けることもでき、「**Finish**」をクリックしてファイルを作成します。

その後、「**Outline**」ペインのsequencesフォルダは同じままですが、「**Navigator**」ペインには、新しく作成されたシーケンス・ファイルが表示されます。

外部化に関する前の項で説明しているのと同様の方法で、選択した複数の内部シーケンスをエクスポートすることもできます。

外部(共有)シーケンス

外部(共有)シーケンスはグラフの外部に保存され、プロジェクト・フォルダ内の個別のファイルに格納されます。シーケンスを複数のグラフで共有する場合は、外部(共有)シーケンスを作成することをお薦めします。ただし、他のユーザーにグラフを提供する場合は、内部シーケンスを作成することをお薦めします。これはメタデータ、接続、参照表およびパラメータの場合と同様です。

外部(共有)シーケンスの作成

外部(共有)シーケンスを作成するには、メイン・メニューから**「File」→「New」→「Other」**を選択し、**CloverETL**カテゴリを展開して、**「Sequence」**項目および**「Next」**ボタンをクリックするか、**「Sequence」**項目をダブルクリックする必要があります。**「Sequence」**ウィザードが開きます。シーケンスの編集(p.215)を参照してください。

外部(共有)シーケンスを作成し、作成されたシーケンス定義ファイルを選択したプロジェクトに保存します。

外部(共有)シーケンスのリンク

作成後(前の項およびシーケンスの編集(p.215)を参照)、外部(共有)シーケンスを使用する各グラフに外部(共有)シーケンスをリンクする必要があります。**「Sequences」**グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから**「Sequences」→「Link shared sequence」**を選択する必要があります。その後、プロジェクト・コンテンツが表示された**ファイル選択**ウィザードが開きます。プロジェクトに含まれているすべてのファイルから目的のシーケンス・ファイルを見つける必要があります(シーケンス・ファイルの拡張子は`.cfg`です)。

複数の外部(共有)シーケンス・ファイルを一度にリンクすることもできます。このことを行うには、**「Sequences」**グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから**「Sequences」→「Link shared sequence」**を選択します。その後、プロジェクト・コンテンツが表示されたファイル選択ウィザードが開きます。プロジェクトに含まれているすべてのファイルから目的のシーケンス・ファイルを見つける必要があります。**[Shift]**を押しながら**下矢印**または**上矢印**キーを押すと、隣接するファイル項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各ファイル項目を**クリック**します。

外部(共有)シーケンスの内部化

外部(共有)シーケンス・ファイルを作成し、リンクした後、このシーケンスをグラフに挿入する場合は、内部シーケンスに変換する必要があります。このようにすると、それがグラフ自体に表示されます。

任意のリンク済外部(共有)シーケンス・ファイルを内部化して内部シーケンスにするには、**「Outline」**ペインで外部(共有)シーケンス項目を右クリックし、コンテキスト・メニューから**「Internalize sequence」**をクリックします。

複数のリンク済外部(共有)シーケンス・ファイルを一度に内部化することもできます。このことを行うには、**「Outline」**ペインで目的のリンク済外部(共有)シーケンス項目を選択します。**[Shift]**を押しながら**下矢印**または**上矢印**キーを押すと、隣接する項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各項目を**クリック**します。

その後、リンク済外部(共有)シーケンス項目は**「Outline」**ペインの**「Sequences」**グループからなくなり、同じ場所に、新しく作成された内部シーケンス項目が表示されます。

ただし、元の外部(共有)シーケンス・ファイルは対応するプロジェクトの`seq`フォルダにそのまま存在し、**「Navigator」**ペインで確認できます(シーケンス・ファイルの拡張子は`.cfg`です)。

シーケンスの編集

このウィザードでは、シーケンスの名前を入力し、最初の番号の値、増分ステップ(つまり、隣接する番号の各ペア間の差異)、キャッシュする計算済値の数およびオプションで、番号を格納するシーケンス・ファイルの名前を選択する必要があります。シーケンス・ファイルを指定しない場合、シーケンスは永続的でなくなり、グラフを実行するたびに値がリセットされます。名前は、`${SEQ_DIR}/sequencefile.seq`や`${SEQ_DIR}/anyothername`などにできます。ここでは`workspace.prm`ファイルで定義されている`SEQ_DIR`パラメータを使用し、その値は`${PROJECT}/seq`です。また、`PROJECT`は、ワークスペースにあるプロジェクトへのパスを定義する別のパラメータです。

既存のシーケンスを編集する場合は、「**Outline**」ペインでシーケンス名を選択し、この名前を右クリックしてコンテキスト・メニューを開き、「**Edit**」項目を選択する必要があります。「**Sequence**」ウィザードが開きます。(このウィザードは、「**Outline**」ペインでシーケンス項目を選択し、**[Enter]**を押して開くこともできます。)

これは、シーケンス番号の現在の値を含む新しいテキスト領域の点で前述のウィザードと異なります。この値はファイルから取得されています。必要に応じて、すべてのシーケンス・プロパティを変更し、ボタンをクリックして現在の値を元の値にリセットできます。

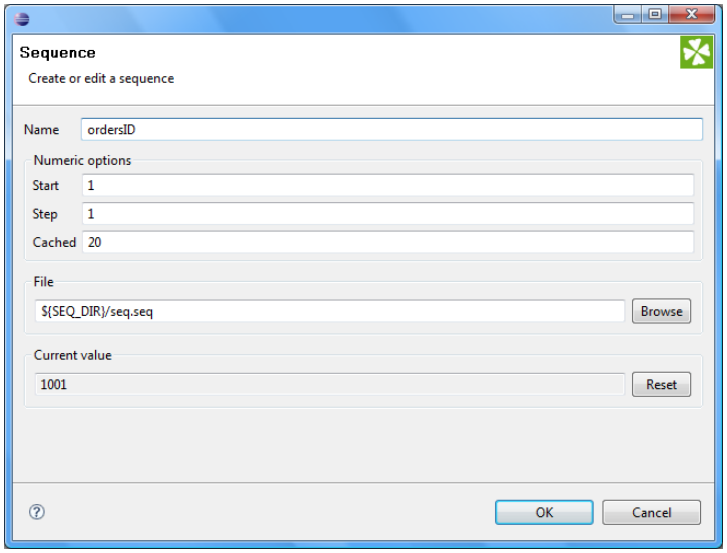


図28.2. シーケンスの編集

グラフが再度実行されると、同じシーケンスが1001から開始されます。

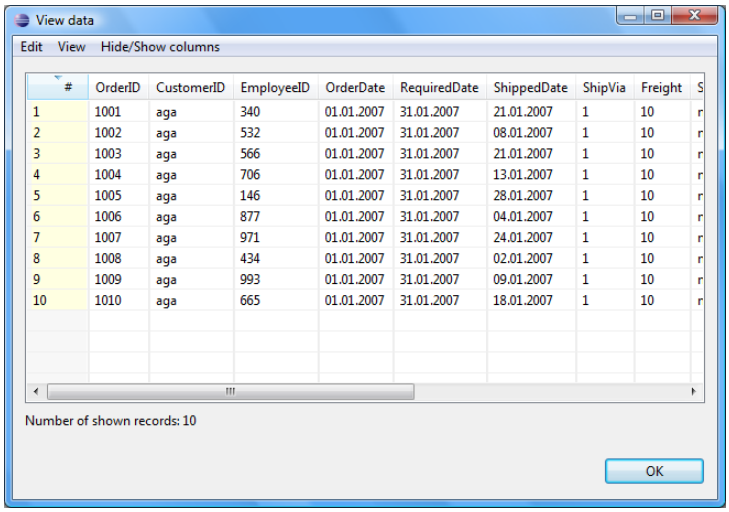


図28.3. 前のシーケンス開始値によるグラフの新規実行

シーケンス番号がいずれかのレコード・フィールドに入力される方法を確認することもできます。

第29章 パラメータ

グラフを使用するときに、パラメータの作成が必要になる場合があります。メタデータや接続と同様、パラメータは内部と外部(共有)のどちらにもできます。パラメータを使用すると、グラフ管理を簡素化できるため、パラメータを作成します。パラメータを使用して、すべての値、番号、パス、ファイル名、属性などを設定または変更できます。パラメータは名前付き定数に類似しています。これらは1箇所に保存され、いずれかの値が変更された後は、この新しい値がプログラムで使用されます。

優先度

- これらのパラメータは、**「Run Configurations...」**の**「Main」**タブまたは**「Arguments」**タブで指定されたものより優先度が低くなります。つまり、内部および外部パラメータはどちらも**「Run Configurations...」**で指定されたパラメータによって上書きされます。ただし、外部および内部パラメータはどちらもすべての環境変数より優先度が高く、これらを上書きできます。また、外部パラメータは内部パラメータを上書きできます。

CTLでパラメータを使用する場合は、`'${MyParameter}'`として入力する必要があります。これらを使用する場合は注意が必要であり、いくつかの文字を指定するためにエスケープ・シーケンスを使用することもできます。

作成できる各パラメータは次のとおりです。

- **内部**: 内部パラメータ(p.217)を参照してください。

  内部パラメータは次のようにできます。

  - **外部化**: 内部パラメータの外部化(p.218)を参照してください。
  - **エクスポート**: 内部パラメータのエクスポート(p.219)を参照してください。

- **外部(共有)**: 外部(共有)パラメータ(p.220)を参照してください。

  外部(共有)パラメータは次のようにできます。

  - **グラフへのリンク**: 外部(共有)パラメータのリンク(p.220)を参照してください。
  - **内部化**: 外部(共有)パラメータの内部化(p.220)を参照してください。

パラメータ・ウィザードはパラメータ・ウィザード(p.222)で説明されています。

内部パラメータ

内部パラメータはグラフに格納されるため、ソースに存在します。パラメータの値を変更する場合は、外部(共有)パラメータを作成することをお薦めします。他のユーザーにグラフを提供する場合は、内部パラメータを作成することをお薦めします。これはメタデータおよび接続の場合と同様です。

内部パラメータの作成

内部パラメータを作成する場合は、**「Outline」**ペインで**「Parameters」**項目を選択し、この項目を右クリックして、**「Parameters」**→**「Create internal parameter」**を選択する必要があります。**「Graph parameters」**ウィザードが表示されます。パラメータ・ウィザード(p.222)を参照してください。

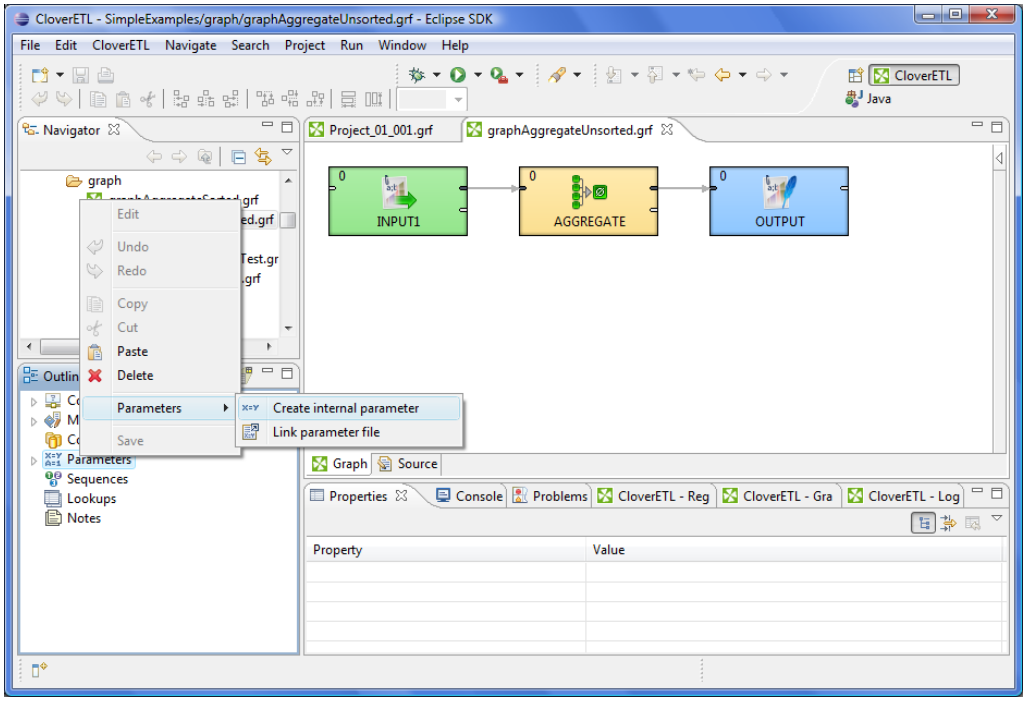


図29.1. 内部パラメータの作成

内部パラメータの外部化

グラフの一部として内部パラメータを作成すると、パラメータはグラフに含められますが、これを外部(共有)パラメータに変換できます。これにより、同じパラメータを複数のグラフで使用できるようになります。

任意の内部パラメータ項目を外部化して外部(共有)ファイルにするには、「**Outline**」ペインで内部パラメータ項目を右クリックし、コンテキスト・メニューから「**Externalize parameters**」を選択します。その後、ワークスペースのプロジェクトのリストが表示された新しいウィザードが開き、対応するプロジェクトがこの新しい外部(共有)パラメータ・ファイルの場所として提案されます。必要に応じて(同じ名前のファイルがすでに存在する場合)、パラメータ・ファイルの提案された名前を変更できます。次に、「**OK**」をクリックします。

その後、内部パラメータ項目は「**Outline**」ペインの「**Parameters**」グループからなくなり、同じ場所に、すでにリンクされた状態で、新しく作成された外部(共有)パラメータ・ファイルが表示されます。選択したプロジェクトに、同じパラメータ・ファイルが表示されることを「**Navigator**」ペインで確認できます。

複数の内部パラメータ項目を一度に外部化することもできます。このようにすると、内部パラメータは1つの外部(共有)パラメータ・ファイルに外部化されます。このことを行うには、「**Outline**」ペインで内部パラメータを選択して右クリックした後、コンテキスト・メニューから「**Externalize parameters**」を選択します。その後、ワークスペースのプロジェクトのリストが表示された新しいウィザードが開き、対応するプロジェクトがこの新しい外部(共有)パラメータ・ファイルの場所として提案されます。必要に応じて(同じ名前のファイルがすでに存在する場合)、パラメータ・ファイルの提案された名前を変更できます。次に、「**OK**」をクリックします。

その後、選択した内部パラメータ項目は「**Outline**」ペインの「**Parameters**」グループからなくなり、同じ場所に、すでにリンクされた状態で、新しく作成された外部(共有)パラメータ・ファイルが表示されます。選択したプロジェクトに、同じパラメータ・ファイルが表示されることを「**Navigator**」ペインで確認できます。このファイルには、選択したすべてのパラメータの定義が含まれています。

[Shift]を押しながら**下矢印**または**上矢印**キーを押すと、隣接するパラメータ項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各パラメータ項目を**クリック**します。

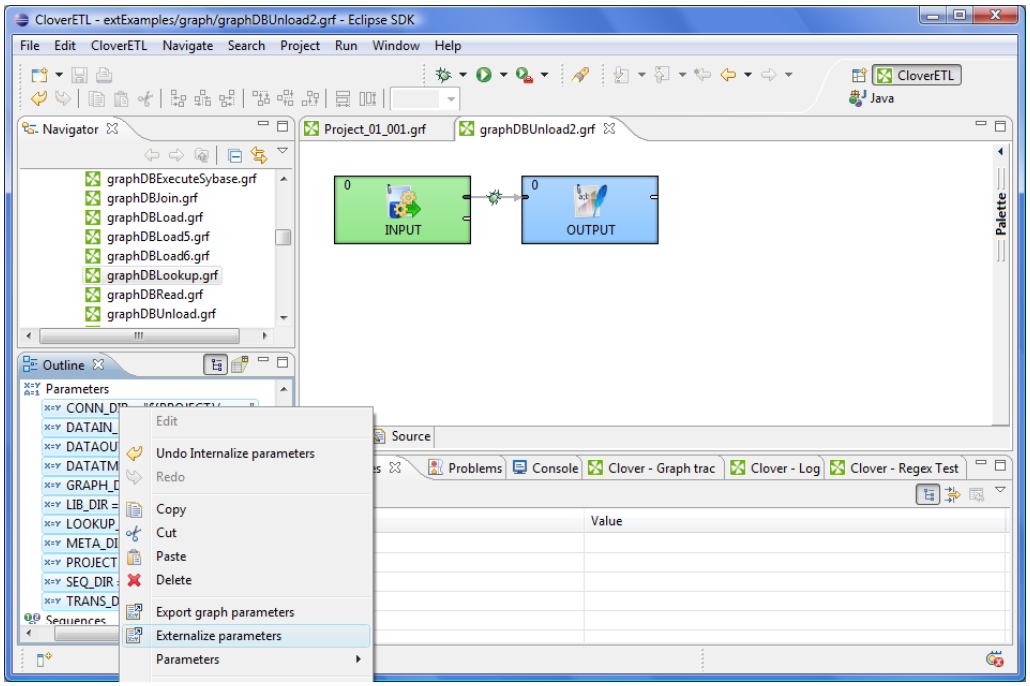


図29.2. 内部パラメータの外部化

内部パラメータのエクスポート

このケースは、内部パラメータの外部化と似ています。グラフ外部のパラメータ・ファイルを外部化されたファイルと同じ方法で作成しますが、ファイルは元のグラフにリンクされません。作成されるのは外部パラメータ・ファイルのみです。前の項で説明したように、その後に、それらのファイルを複数のグラフで外部(共有)パラメータ・ファイルとして使用できます。

内部パラメータを外部(共有)パラメータにエクスポートするには、「**Outline**」ペインで内部パラメータ項目を右クリックし、コンテキスト・メニューから「**Export parameter**」をクリックします。新しく作成された外部ファイルに対して、対応するプロジェクトが提案されます。提案されたものとは異なる名前をファイルに付けることもでき、「**Finish**」をクリックしてファイルを作成します。

その後、「**Outline**」ペインのparametersフォルダは同じままですが、「**Navigator**」ペインに新しく作成されたパラメータ・ファイルが表示されます。

外部化に関する前の項で説明しているのと同様の方法で、選択した複数の内部パラメータをエクスポートすることもできます。

外部(共有)パラメータ

外部(共有)パラメータはグラフの外部に保存され、プロジェクト・フォルダ内の個別のファイルに格納されます。パラメータの値を変更する場合は、外部(共有)パラメータを作成することをお薦めします。ただし、他のユーザーにグラフを提供する場合は、内部パラメータを作成することをお薦めします。これはメタデータおよび接続の場合と同様です。

外部(共有)パラメータの作成

外部(共有)パラメータを作成する場合は、「**Outline**」で「**Parameters**」を右クリックし、「**Parameters**」→「**Graph parameter file**」を選択します。

「**Graph parameters**」ウィザードが開きます。パラメータ・ウィザード(p.222)を参照してください。このウィザードで、名前およびパラメータの値を作成します。

外部(共有)パラメータのリンク

作成後(前の項およびパラメータ・ウィザード(p.222)を参照)、外部(共有)パラメータを使用する各グラフに外部(共有)パラメータをリンクできます。「**Parameters**」グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから「**Parameters**」→「**Link parameter file**」を選択する必要があります。その後、プロジェクト・コンテンツが表示された**ファイル選択**ウィザードが開きます。プロジェクトに含まれているすべてのファイルから目的のパラメータ・ファイルを見つける必要があります(パラメータ・ファイルの拡張子は.prmです)。

複数の外部(共有)パラメータ・ファイルを一度にリンクすることもできます。このことを行うには、「**Parameters**」グループまたはそのいずれかの項目を右クリックし、コンテキスト・メニューから「**Parameters**」→「**Link parameter file**」を選択します。その後、プロジェクト・コンテンツが表示されたファイル選択ウィザードが開きます。プロジェクトに含まれているすべてのファイルから目的のパラメータ・ファイルを見つける必要があります。**[Shift]**を押しながら**下矢印**または**上矢印**キーを押すと、隣接するファイル項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各ファイル項目を**クリック**します。

外部(共有)パラメータの内部化

外部(共有)パラメータ・ファイルを作成し、リンクした後、このパラメータをグラフに挿入する場合は、内部パラメータに変換する必要があります。このようにすると、それらがグラフ自体に表示されます。1つのパラメータ・ファイルに含まれているパラメータが多くなるほど、多数の内部パラメータが作成されます。

任意のリンク済外部(共有)パラメータ・ファイルを内部化して内部パラメータにするには、「**Outline**」ペインで外部(共有)パラメータ項目を右クリックし、コンテキスト・メニューから「**Internalize parameters**」を**クリック**します。

複数のリンク済外部(共有)パラメータ・ファイルを一度に内部化することもできます。このことを行うには、「**Outline**」ペインで目的のリンク済外部(共有)パラメータ項目を選択します。**[Shift]**を押しながら**下矢印**または**上矢印**キーを押すと、隣接する項目を選択できます。隣接していない項目を選択する場合は、かわりに**[Ctrl]**を押しながら目的の各項目を**クリック**します。

その後、リンク済外部(共有)パラメータ項目は「**Outline**」ペインの「**Parameters**」グループからなくなり、同じ場所に、新しく作成された内部パラメータ項目が表示されます。

ただし、元の外部(共有)パラメータ・ファイルは対応するプロジェクトにそのまま存在し、「**Navigator**」ペインで確認できます(パラメータ・ファイルの拡張子は.prmです)。

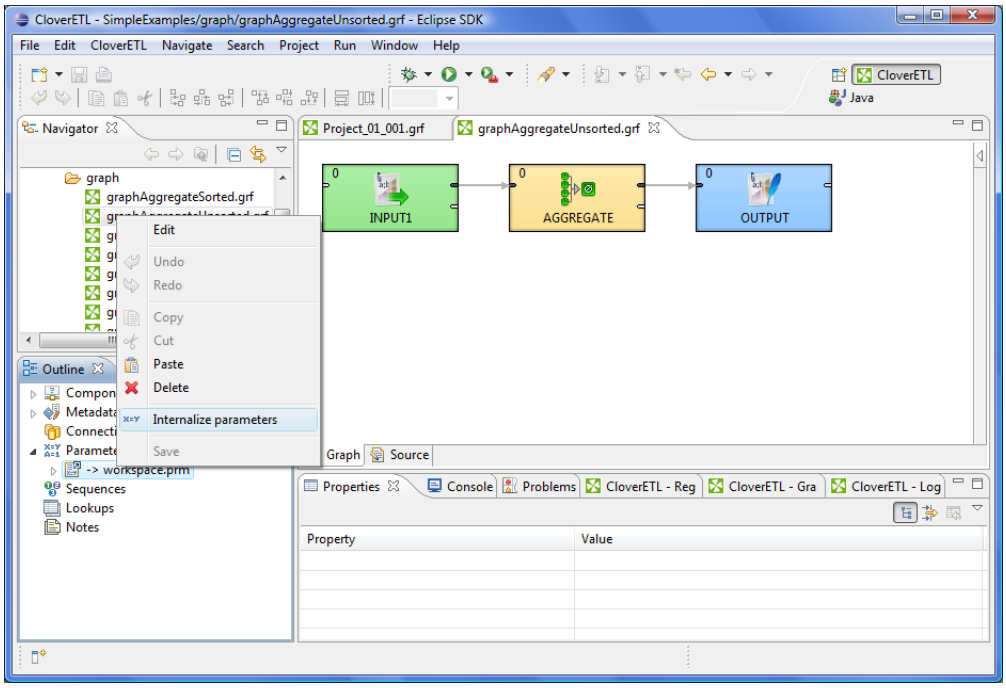


図29.3. 外部(共有)パラメータの内部化

パラメータ・ウィザード

(このウィザードは、「**Outline**」ペインでパラメータ項目を選択し、**[Enter]**を押して開くこともできます。)

右側の**プラス・**ボタンをクリックすると、**名前**と**値**の語のペアがウィザードに表示されます。**プラス・**ボタンをクリックするたびに**名前**および**値**ラベルを含む新しい行が表示され、名前と値の両方を設定する必要があります。このことを行うには、これらをクリックして強調表示し、必要な内容に変更します。すべての名前を選択し、必要なすべての値を設定したら、「**Finish**」ボタン(内部パラメータの場合)または「**Next**」ボタンをクリックし、パラメータ・ファイルの名前を入力できます。拡張子.prmがファイルに自動的に追加されます。

プロジェクト・フォルダ内のパラメータ・ファイルの場所も選択する必要があります。その後、「**Finish**」ボタンをクリックできます。その後、ファイルが保存されます。

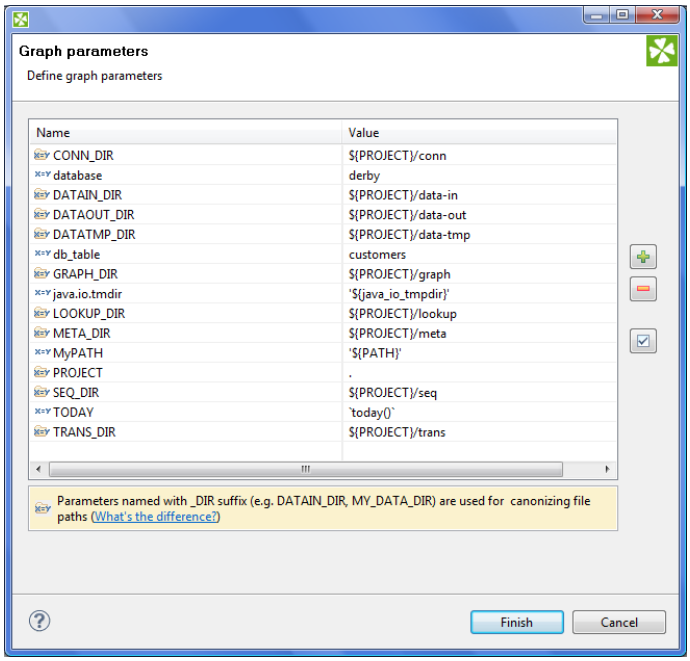


図29.4. パラメータ-値ペアの例

注意

パラメータ名の横にある2種類のアイコンに注意してください。1つはパスを正規化できるパラメータ(_DIRサフィックスの付いたパラメータおよびPROJECTパラメータ)をマークし、それ以外ではパスは正規化されません。詳細は、ファイル・パスの正規化(p.223)を参照してください。

注意

また、次のパラメータの使用方法に注意してください。

1. `TODAY`

   このパラメータではCTL1式が使用されます。詳細は、CTL式を含むパラメータ(p.223)を参照してください。

2. `java.io.tmdir, MyPATH`

   これらのパラメータは環境変数に解決されます。詳細は、環境変数(p.223)を参照してください。

3. `database, db_table`

これらは標準パラメータです。

パラメータがどのような値に解決されるかを参照するには、ダイアログの右下にあるボタンをクリックします。

CTL式を含むパラメータ

CloverETLのバージョン2.8.0以上では、パラメータおよび**CloverETL**の他の場所でもCTL式を使用できます。このようなCTL式ではCTL言語のすべての機能を使用できます。ただし、これらのCTL式はバッククォートで囲む必要があります。

たとえば、パラメータ`TODAY="`today()`"`を定義してCTLコードで使用する場合、このような`${TODAY}`式は現在の日付に解決されます。

バッククォートをそのまま表示する場合は、そのバッククォートの前にバック・スラッシュ「\`」を付けて使用する必要があります。

重要

CTL1バージョンはこのような式で使用されます。

環境変数

環境変数は、**CloverETL**では定義されず、オペレーティング・システムで定義されるパラメータです。

これらの環境変数の値は、他のすべてのパラメータに使用できる同じ式を使用して取得できます。

- `PATH`という環境変数の値を取得するには、次の式を使用します。

  '${PATH}'

重要

特に、Windowsでは、path環境変数を参照するときに一重引用符を使用します。変数の文字列値を区切る二重引用符と、値自体に含まれる可能性のある二重引用符との間の競合を回避するために、このことが必要となります。

- 名前にドットが含まれる変数(`java.io.tmpdir`など)の値を取得するには、各ドットをアンダースコア文字に置き換えて、次のように入力します。

  ```
  '${java_io_tmpdir} '
  ```

  `java.io.tmpdir`自体がバックスラッシュで終了しておりエスケープ・シーケンス(「\'」)が不要なため、終わりの一重引用符の前に空白が必要です。この空白によって、取得される末尾が「\ '」になります。

重要

Windowsパスでエスケープ・シーケンスを回避するには、一重引用符を使用します。

ファイル・パスの正規化

すべてのパラメータは次の2つのグループに分類できます。

1. `PROJECT`パラメータおよび`_DIR`がサフィックスとして使用された他のパラメータ(`DATAIN_DIR`、`CONN_DIR`、`MY_PARAM_DIR`など)。

2. その他のすべてのパラメータ。

いずれのグループも、**パラメータ・ウィザード**の対応するアイコンで識別されます。

最初のグループのパラメータは、**URLファイル・ダイアログ**および**「Outline」**ペインに表示されるファイル・パスを、次のように自動的に正規化する働きをします。

1. これらのいずれかのパラメータがパスの先頭と一致する場合は、パスの先頭の対応する部分がこのパラメータで置き換えられます。
2. 複数のパラメータが同じパスの先頭の異なる部分と一致する場合は、パスの最も長い部分を表すパラメータが選択されます。

例29.1. ファイル・パスの正規化

- 次の2つのパラメータがあるとします。

  ```
  MY_PARAM1_DIR and MY_PARAM2_DIR
  ```

  これらの値は次のとおりです。

  ```
  MY_PARAM1_DIR = "mypath/to" and MY_PARAM2_DIR = "mypath/to/some"
  ```

  パスは次のとおりであるとします。

  ```
  mypath/to/some/directory/with/the/file.txt
  ```

  このパスは次のように表示されます。

  ```
  ${MY_PARAM2_DIR}/directory/with/the/file.txt
  ```

- 次の2つのパラメータがあるとします。

  ```
  MY_PARAM1_DIR and MY_PARAM3_DIR
  ```

  値は次のとおりです。

  ```
  MY_PARAM1_DIR = "mypath/to" and MY_PARAM3_DIR = "some"
  ```

  パスは前に示したものと同じです。

  ```
  mypath/to/some/directory/with/the/file.txt
  ```

  このパスは次のように表示されます。

  ```
  ${MY_PARAM1_DIR}/some/directory/with/the/file.txt
  ```

- 次のパラメータがあるとします。

  ```
  MY_PARAM1
  ```

  値は次のとおりです。

  ```
  MY_PARAM1 = "mypath/to"
  ```

  パスは前に示したものと同じです。

  ```
  mypath/to/some/directory/with/the/file.txt
  ```

  パスは正規化されません。

  パスの先頭の同じ文字列`mypath/to`は`MY_PARAM1`というパラメータを使用して表現できますが、このパラメータはパスを正規化できるパラメータのグループに属していません。このため、このパラメータではパスは正規化されず、フルパスがそのまま表示されます。

重要

次のパスは、**URLファイル・ダイアログ**および**「Outline」**ペインに表示されません。

```
${MY_PARAM1_DIR}/${MY_PARAM3_DIR}/directory/with/the/file.txt
```

```
${MY_PARAM1}/some/directory/with/the/file.txt
```

```
mypath/to/${MY_PARAM2_DIR}/directory/with/the/file.txt
```

パラメータの使用方法

たとえば、(前述のように、)データベース表を意味するdb_tableというパラメータを定義し、employeeという名前(値)を指定した場合、このデータベース表を使用する場所では、常に、employeeではなく${db_table}のみを使用できます。

注意

CloverETLのバージョン2.8.0以上では、パラメータでCTL式を使用することもできます。このようなCTL式ではCTL言語のすべての機能を使用できます。ただし、これらのCTL式はバッククォートで囲む必要があります。

たとえば、パラメータTODAY="`today()`"を定義してCTLコード内で使用する場合、このような${TODAY}式はその日の日付に解決されます。

バッククォートをそのまま表示する場合は、そのバッククォートの前にバック・スラッシュ\`を付けて使用する必要があります。

重要

CTL1バージョンはこのような式で使用されます。

前述のように、すべてのものをパラメータを使用して表現できます。

第30章 内部/外部グラフ要素

この章は、メタデータ(p.110)、データベース接続(p.172)、JMS接続(p.185)、QuickBase接続(p.190)、参照表(p.195)、シーケンス(p.211)およびパラメータ(p.217)に適用されます。

前述のすべてのグラフ要素に共通する、複数のプロパティがあります。

これらはすべて内部または外部(共有)にできます。

内部グラフ要素

内部要素はグラフの一部となります。これらはグラフに含まれ、**グラフ・エディタ**の「**Source**」タブを参照すると確認できます。

外部(共有)グラフ要素

外部(共有)要素はグラフ外部の外部ファイル(`meta`、`conn`、`lookup`、`seq`サブフォルダ内、またはデフォルトで`project`自体)に置かれます。

「**Source**」タブを参照すると、このような外部ファイルへのリンクのみを確認できます。これらの要素はそのファイル内にあります。

グラフ要素の使用

同じデータ・ファイルまたは同じデータベース表、あるいはその他のデータ・ソースを使用する複数のグラフがあるとします。このような各グラフに対して、同じメタデータ、接続、参照表、シーケンスまたはパラメータを作成できます。これらをグラフごとに個別に定義するか、またはすべてのグラフで共有できます。

メタデータ以外に、接続(データベース接続、JMS接続およびQuickBase接続)、参照表、シーケンスおよびパラメータに同様のことが該当します。また、接続、シーケンスおよびパラメータは、内部および外部(共有)にできます。

外部(共有)グラフ要素の利点

複数のグラフに対する1つの外部(共有)定義を1箇所に置くこと、つまり、同じリソースを使用する様々なグラフにリンクされ、これらのすべてのグラフで共有される1つの外部ファイルを作成することで、利便性と単純化が促進されます。

複数のグラフ間で共有されるこれらの要素を個別に使用した場合、すべての要素に変更を加えることが困難になります。このような場合は、各グラフで同じ特性を変更する必要があります。対象となるプロパティを1箇所のみで、外部(共有)定義ファイルで変更できる方がより適切です。

外部(共有)グラフ要素は直接作成することも、内部グラフ要素をエクスポートまたは外部化することもできます。

内部グラフ要素の利点

一方、他のユーザーにグラフを提供する場合は、グラフのみでなく、すべてのリンクされた情報も提供する必要があります。この場合、これらの要素をグラフに含める方が、はるかに簡単です。

内部グラフ要素は直接作成することも、外部(共有)要素をグラフにリンクした後に内部化することもできます。

グラフ要素の形式の変更

CloverETL Designerは、内部または外部(共有)要素をいつ作成するかという問題の解決に役立ちます。

- **グラフへの外部グラフ要素のリンク**

  グラフ外部の1つまたは複数のファイルに定義された要素がある場合、これらをグラフにリンクできます。これらのリンクは、**グラフ・エディタ・**ペインの「**Source**」タブで確認できます。

- **グラフへの外部グラフ要素の内部化**

  グラフ外部にあるが、グラフにリンクされた1つまたは複数のファイルに定義された要素がある場合、これらを内部化できます。ファイルはそのまま存在しますが、新しい内部グラフ要素がグラフに表示されます。

- **グラフ内の内部グラフ要素の外部化**

  グラフに定義された要素がある場合、これらを外部化できます。これらは対応するサブディレクトリ内のファイルに変換され、元の内部グラフ要素のかわりにこれらのファイルへのリンクのみがグラフに表示されます。

- **グラフ外部への内部グラフ要素のエクスポート**

  グラフに定義された要素がある場合、これらをエクスポートできます。グラフ外部に(グラフにリンクされていない)新しいファイルが作成され、元の内部グラフ要素はグラフに残ります。

第31章 ディクショナリ

ディクショナリは、**CloverETL**のグラフの各実行に関連付けられたデータ記憶域オブジェクトです。その目的は、グラフに必要な様々なパラメータの簡素なタイプセーフの記憶域を提供することです。

これは入力または出力パラメータの格納に限定されず、1つのグラフの様々なコンポーネント間でデータを共有する方法としても使用されます。

グラフがそのXML定義ファイルからロードされると、ディクショナリがグラフ指定での定義に基づいて初期化されます。それぞれの値は、デフォルト値(デフォルト値が設定されている場合)に初期化されるか、または外部ソース(**起動サービス**など)で設定される必要があります。

重要

Clover Transformation Languageの2つのバージョンでは、ディクショナリを使用前に定義する必要かあるかどうかが異なります。

- **CTL1**

  CTL1では、ユーザーは、一連のディクショナリ関数を使用して、ディクショナリ・エントリを、前もって定義することなく作成できます。

  ディクショナリ関数(p.884)を参照してください。

- **CTL2**

  CTL1とは異なり、CTL2では、ディクショナリ・エントリを使用する前に、常に定義しておく必要があります。ユーザーは、ディクショナリを使用するための標準CTL2構文を使用する必要があります。CTL2ではディクショナリ関数は使用できません。

  CTL2のディクショナリ(p.898)を参照してください。

グラフが2回連続して実行される間に、ディクショナリは初期設定またはデフォルト設定にリセットされるため、ディクショナリの実行時変更はすべて破棄されます。このため、ディクショナリを同じグラフの異なる実行間で値を渡すために使用することはできません。

この章では、ディクショナリの作成方法および使用方法について説明します。

ディクショナリの作成

ディクショナリ指定は、グラフのインタフェースとして機能し、たとえば、グラフが**起動サービス**とともに使用される場合にも、常に必要となります。

ソース・コードでは、ディクショナリのエントリは`<Dictionary>`要素の内部に指定されます。

ディクショナリを作成するには、「**Outline**」ペインで「**Dictionary**」項目を右クリックし、コンテキスト・メニューから「**Edit**」を選択します。**ディクショナリ**・エディタが開きます。

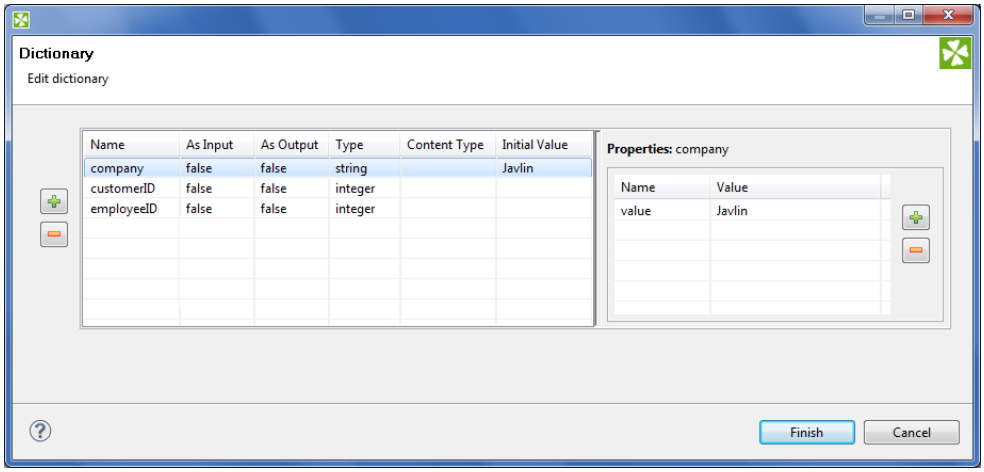


図31.1. 定義済エントリを含む「Dictionary」ダイアログ

左側の**プラス記号**ボタンをクリックして新しいディクショナリ・エントリを作成します。

その後、**名前**を指定します(名前は、大文字と小文字を区別した、ディクショナリ内で一意の有効なJava識別子にする必要があります)。エントリの他のプロパティも指定する必要があります。

1. `Name`

   ディクショナリ・エントリの名前を指定します。名前は、大文字と小文字を区別した、ディクショナリ内で一意の有効なJava識別子にする必要があります。

2. `As Input`

   ディクショナリ・エントリを入力として使用できるかどうかを指定します。この値は`true`または`false`にできます。

3. `As Output`

   ディクショナリ・エントリを出力として使用できるかどうかを指定します。この値は`true`または`false`にできます。

4. `Type`

   ディクショナリ・エントリのタイプを指定します。

   ディクショナリ・タイプは次のプリミティブCloverデータ型です。

   - `boolean`、`byte`、`date`、`decimal`、`integer`、`long`、`number`および`string`。

   CTL2でこれらのいずれかにアクセスすることもできます。詳細は、CTL2のディクショナリ(p.898)を参照してください。

   ディクショナリ・エントリには他に3つの(Javaで使用可能な)データ型があります。

   - `object`: **CloverETL Engine**で使用可能な**CloverETL**データ型。
   - `readable.channel`: 入力が**リーダー**の構成に従って**リーダー**によってエントリから直接読み取られます。このため、エントリにはデータが有効な形式で含まれている必要があります。
   - `writable.channel`: 出力が出力**ライター**(たとえば、テキスト・ファイル、XLSファイルなど)で指定された形式でこのエントリに直接書き込まれます。

5. `Content Type`

   出力エントリのコンテンツ・タイプを指定します。このコンテンツ・タイプは、たとえば、ユーザーに結果を返送するためにグラフが**起動サービス**によって起動された場合に使用されます。

6. `Initial Value`

   エントリのデフォルト値であり、ディクショナリに外部データを実際に移入することなくグラフを実行する場合に役立ちます。管理者以外はすべてのデータ型(`object`など)についてこのフィールドを編集できません。新しい**「Initial Value」**を設定すると、対応する名前/値ペアが右側の**「Properties」**ペインに作成されます。したがって、**初期値**はそのペインで作成した最初の値と同じになります。

各エントリにプロパティ(名前および値)を指定できます。これらを指定するには、右側の対応するボタンをクリックし、次の2つのプロパティを指定します。

- `Name`

  対応するエントリの値の名前を指定します。

- `Value`

  エントリに対応する名前の値を指定します。

グラフでのディクショナリの使用方法

ディクショナリには、グラフ内の様々なコンポーネントから複数の方法でアクセスできます。次のコンポーネントからアクセスできます。

リーダーおよび**ライター**。これらはどちらも、**「File URL」**属性を使用したデータ・ソースまたはデータ・ターゲットとしてディクショナリをサポートします。

ディクショナリには、変換を定義する任意のコンポーネント(すべての**ジョイナ**、**Reformat**、**Normalizer**など)でCTLまたはJavaソース・コードを使用してアクセスすることもできます。

リーダーおよびライターからのディクショナリへのアクセス

グラフ・コンポーネントの**「File URL」**属性でディクショナリ・パラメータを参照するには、この属性を`dict:<Parameter name>[:processingType]`の形式にする必要があります。ディクショナリのパラメータのタイプおよび`processingType`に応じて、値を入力または出力ファイルの名前として使用するか、あるいはデータ・ソースまたはデータ・ターゲットとして直接使用できます(つまり、データはパラメータに対して直接読取りと書込みが行われます)。

処理タイプは次のとおりです。

1. **リーダーの場合**

   - `discrete`

     これはデフォルトの処理タイプで、指定する必要はありません。

   - `source`

   リーダーのURLの詳細は、ディクショナリからの読取り(p.300)も参照してください。

2. **ライターの場合**

   - `source`

     この処理タイプはデフォルトで事前選択されます。

   - `stream`

     処理タイプを指定しない場合、**stream**が使用されます。

   - `discrete`

   ライターのURLの詳細は、ディクショナリへの書込み(p.313)も参照してください。

たとえば、dict:mountains.csvは、**リーダー**または**ライター**でそれぞれ入力または出力として使用できます(この場合、プロパティ・タイプはwritable.channelです)。

Javaを使用したディクショナリへのアクセス

グラフのコンポーネントに埋め込まれたJavaコードから値にアクセスするには、org.jetel.graph.Dictionaryクラスのメソッドを使用する必要があります。

たとえば、heightMinプロパティの値を取得するには、次のスニペットのようなコードを使用できます。

```
getGraph().getDictionary().getValue("heightMin")
```

前述のスニペットでは、TransformationGraphのインスタンスが必要であることがわかりますが、これは通常、独自のコードを挿入できる任意の場所でgetGraph()メソッドを介して利用できます。次に現在のディクショナリがgetDictionary()メソッドによって取得され、最後にgetValue(String)メソッドを呼び出すことによってプロパティの値が読み取られます。

注意

詳細は、**JavaDoc**ドキュメントを確認してください。

CTL2を使用したディクショナリへのアクセス

ディクショナリのエントリをCTL2で使用する必要がある場合は、それらをグラフで定義する必要があります。エントリの使用では、標準のCTL2構文を使用します。CTL2ではディクショナリ関数は使用できません。

詳細は、CTL2のディクショナリ(p.898)を参照してください。

CTL1を使用したディクショナリへのアクセス

stringデータ型のエントリの関数セットを使用してCTL1からディクショナリにアクセスできます。

ディクショナリのエントリをCTL1で使用する必要がある場合でも、それらをグラフで定義する必要はありません。

詳細は、ディクショナリ関数(p.884)を参照してください。

第32章 グラフ内のノート

前述の**コンポーネント・パレット**には、「**Note**」アイコンも含まれています。グラフを作成する際、**グラフ・エディタ・ペイン**に1つ以上のノートを貼り付けることができます。そのためには、「**Palette**」で「**Note**」アイコンをクリックし、**グラフ・エディタ**に移動して再度クリックします。これにより、その場所に新しいノートが表示されます。新しい**ノート**には「**New note**」ラベルが付いています。

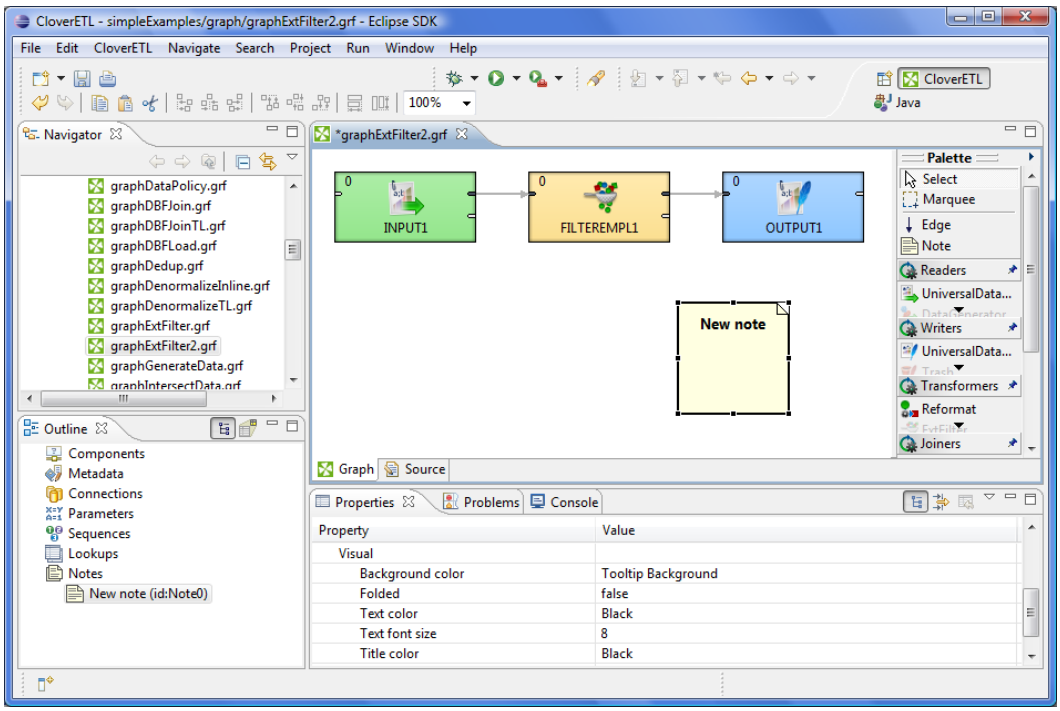


図32.1. グラフ・エディタ・ペインへのノートの貼付け

ノートをクリックし、クリック後に強調表示されたいずれかの縁をドラッグしてノートを拡大することもできます。

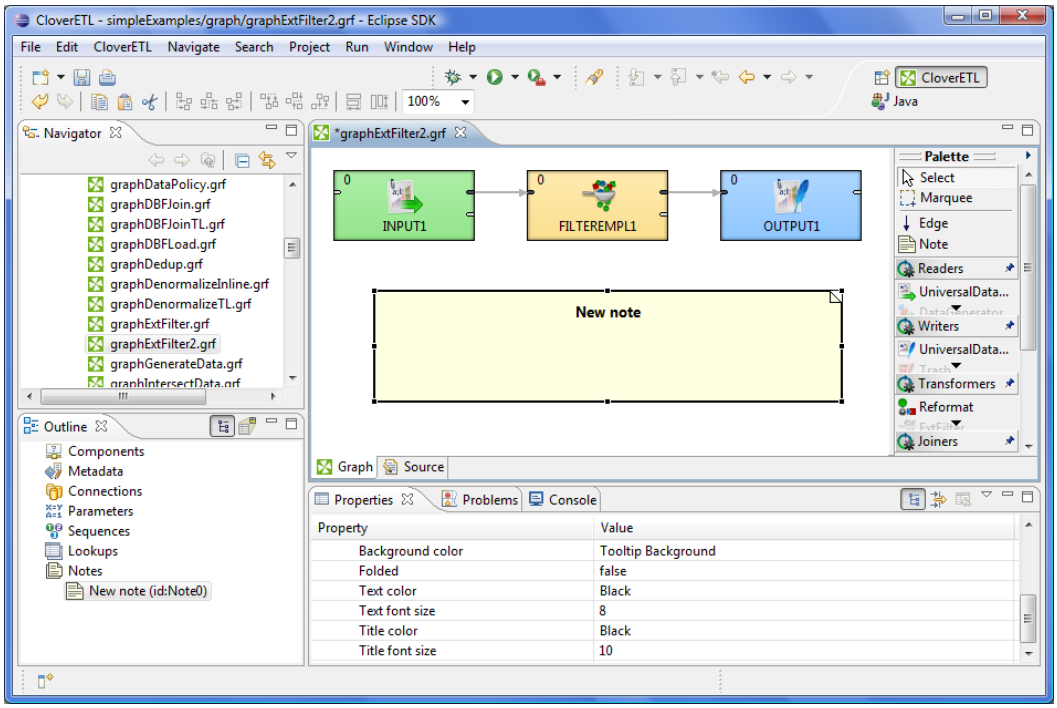


図32.2. ノートの拡大

最後に、強調表示が消えるように**ノート**以外の任意の場所をクリックします。

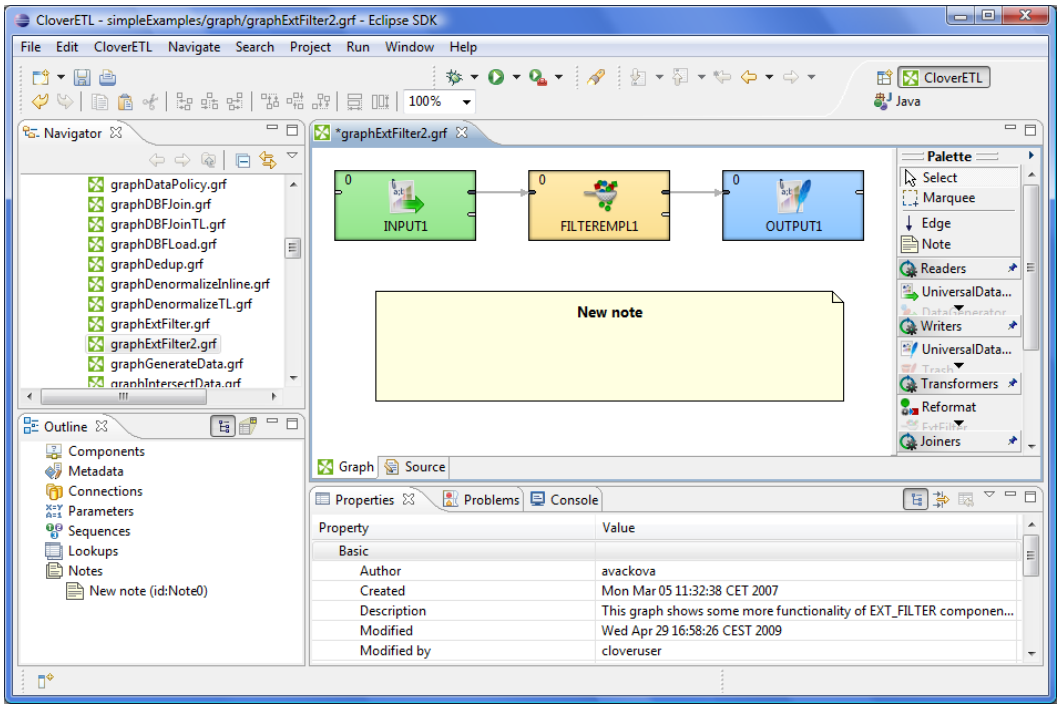


図32.3. 縁の強調表示が消えたノート

ノートにグラフの動作内容の説明を書き込む場合は、**ノート**内で2回クリックします。最初のクリック後、縁が強調表示され、2回目のクリック後、**ノート**内に白い長方形の領域が表示されます。「**New note**」ラベルでこのクリックを行うと、長方形の中にこのラベルが表示され、これを変更できます。

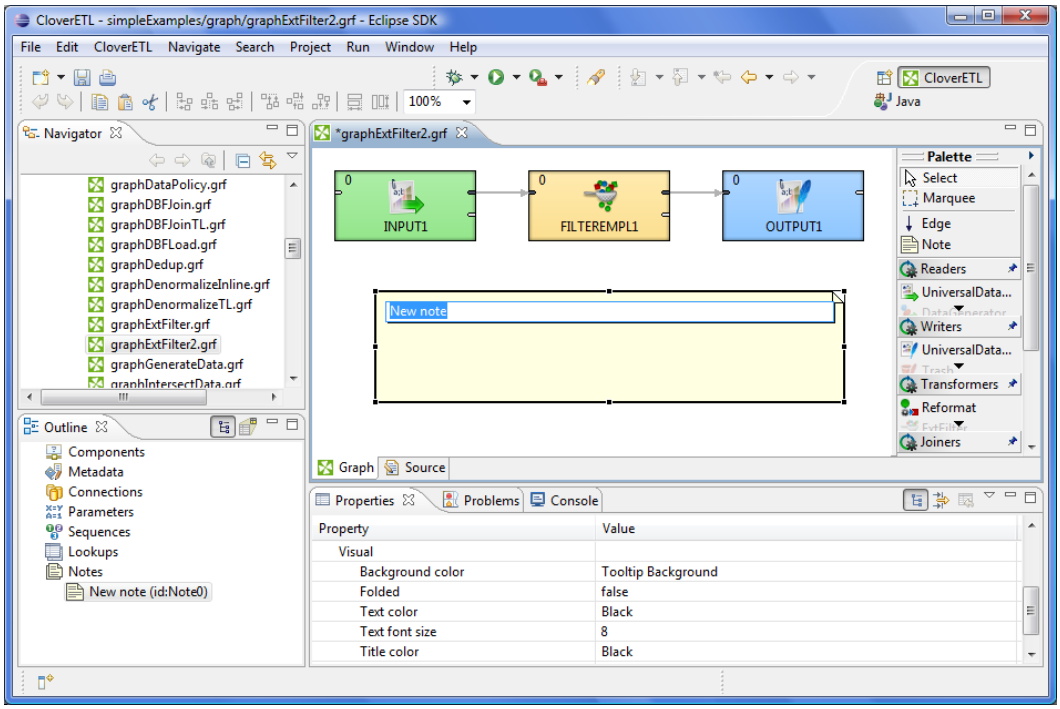


図32.4. ノート・ラベルの変更

「**New note**」ラベル以外をクリックすると、新しい長方形のスペースが表示され、その中にグラフの説明を書き込むことができます。スペースを拡大して追加のテキストを書き込むこともできます。

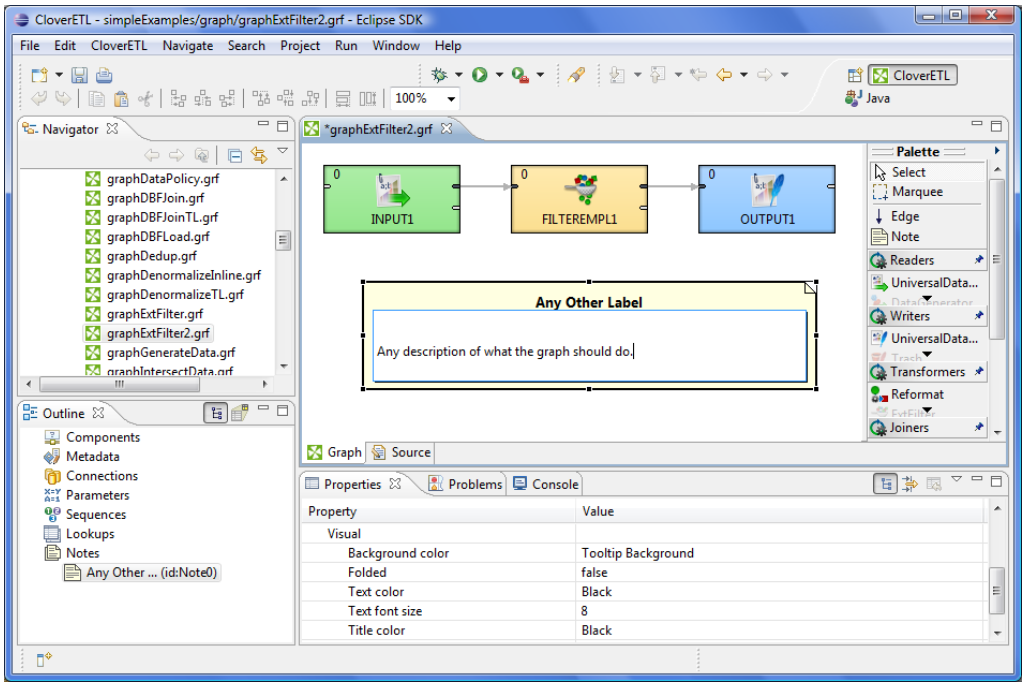


図32.5. ノートでの新しい説明の書込み

結果のノートは次のようになります。

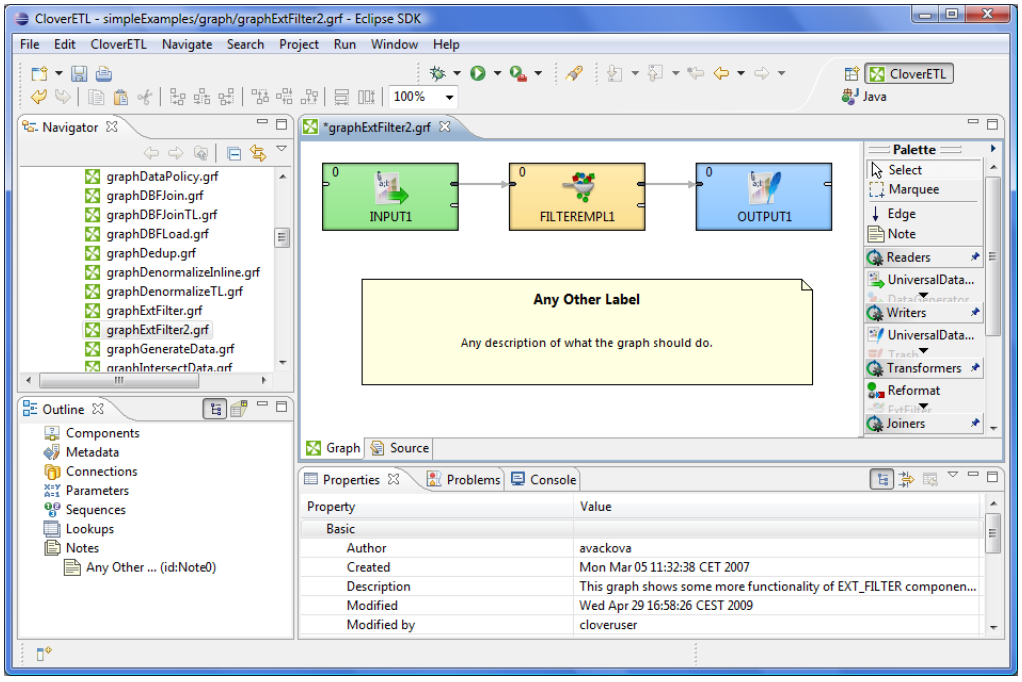


図32.6. 新しい説明を含む新しいノート

このようにして、1つのグラフに追加のノートを貼り付けることができます。

ノートにパラメータを入力した場合は、パラメータではなくその値が表示されます。

また、ノート内にある各コンポーネントは、ノートを移動すると一緒に移動します。

コンテキスト・メニューから「**Fold**」項目を選択してノートを折りたたむこともできます。結果のノートにはラベルのみが表示され、次のようになります。

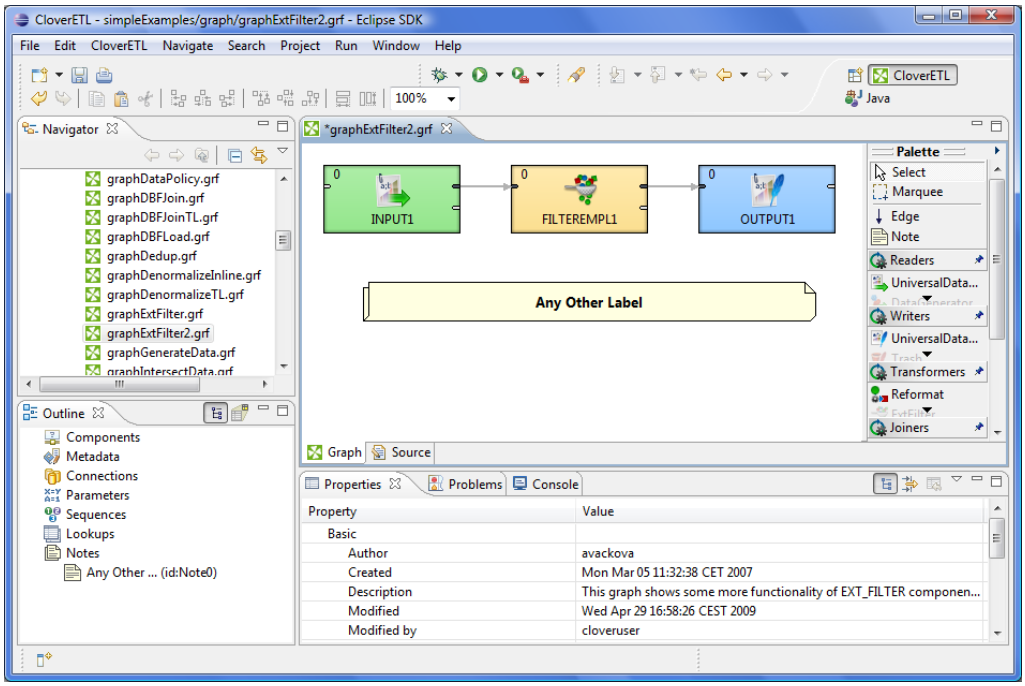


図32.7. ノートの折りたたみ

ノートをクリックし、「**Properties**」タブに切り替えて、**ノート**の多数のプロパティを設定することもできます。このタブには、**ノート**の**テキスト**と**タイトル**の両方が表示されます。これらはどちらもこのタブで変更できます。また、**テキスト**を**左**、**中央**または**右揃え**にすることを決定することもできます。**タイトル**はデフォルトで中央揃えになります。**テキスト**と**タイトル**の両方に色を指定でき、コンボ・リストから色を選択できます。デフォルトでは、これらはどちらも黒で、背景はツールチップの背景色となります。これらのプロパティをここで設定できます。デフォルトのフォント・サイズもこのタブに表示され、変更できます。**ノート**を折りたたむ場合は、「**Folded**」属性を「true」に設定します。他のグラフ・コンポーネントと同じように、各**ノート**には**ID**があります。

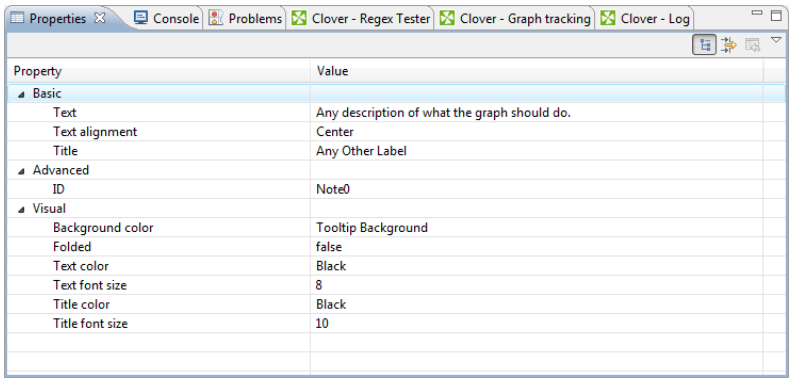


図32.8. ノートのプロパティ

第33章 検索機能

CloverETL Designerのメイン・メニューで「**Search**」→「**Search...**」を選択すると、次のタブがあるウィンドウが開きます。

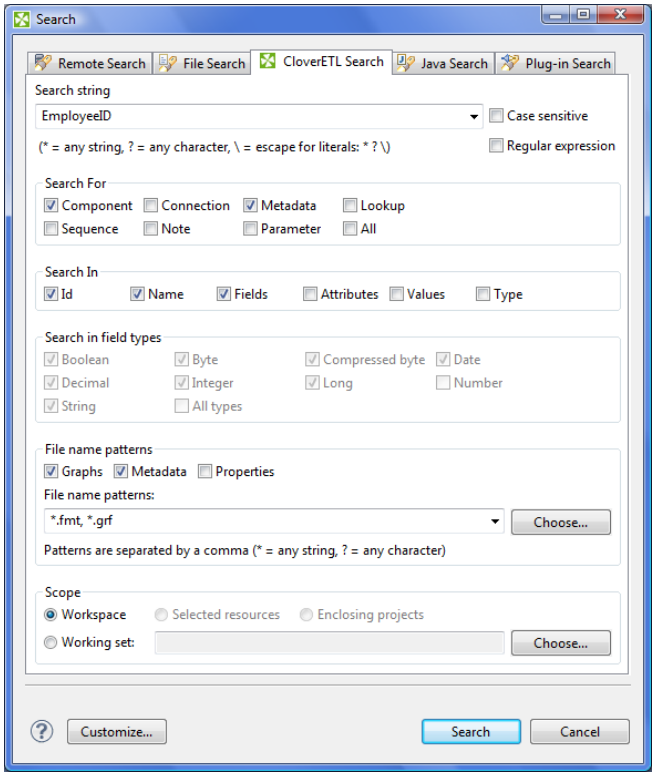


図33.1.「CloverETL Search」タブ

「**CloverETL search**」タブで、検索対象を指定する必要があります。

最初に、検索で大文字/小文字を区別するかどうかを指定できます。および「**Search string**」テキスト領域に入力される文字列を正規表現(p.963)として扱うかどうかを指定できます。

2番目に、検索対象(**コンポーネント**、**接続**、**参照**、**メタデータ**、**シーケンス**、**ノート**、**パラメータ**または**すべて**)を指定する必要があります。

3番目に、前述のオブジェクトのどの特性(**ID**、**名前**、**フィールド**、**属性**、**値**または**タイプ**)で検索を実行するかを決定する必要があります。(「**Type**」チェック・ボックスを選択した場合、使用可能なすべてのデータ型から選択できます。)

4番目に、検索を実行するファイル(**グラフ**(`*.grf`)、**メタデータ**(`*.fmt`)または**プロパティ**(パラメータを定義するファイル: `*.prm`))を決定します。入力するか、またはボタンをクリックしてファイル拡張子のリストから選択して、独自のファイルを選択することもできます。

たとえば、グラフでメタデータを検索する場合は、フィールド、属性および値を含む、内部と外部の両方のメタデータが検索されます。内部および外部の接続とパラメータについても同様です。

最後のステップとして、検索で**ワークスペース**全体を考慮するか、**選択したリソース**のみを考慮するか(**[Ctrl]を押しながらクリック**して選択)、または選択したリソースを含むプロジェクト(**包含プロジェクト**)を考慮するかを決定する必要があります。**ワーキング・セット**を定義し、検索オプションを**カスタマイズ**することもできます。

「**Search**」ボタンをクリックすると、検索の結果を含む新しいタブが**タブ・ペイン**に表示されます。カテゴリを展開し、内部の項目をダブルクリックすると、その項目がテキスト・エディタ、メタデータ・ウィザードなどで開きます。

「**Search**」タブを展開すると、検索結果を確認できます。

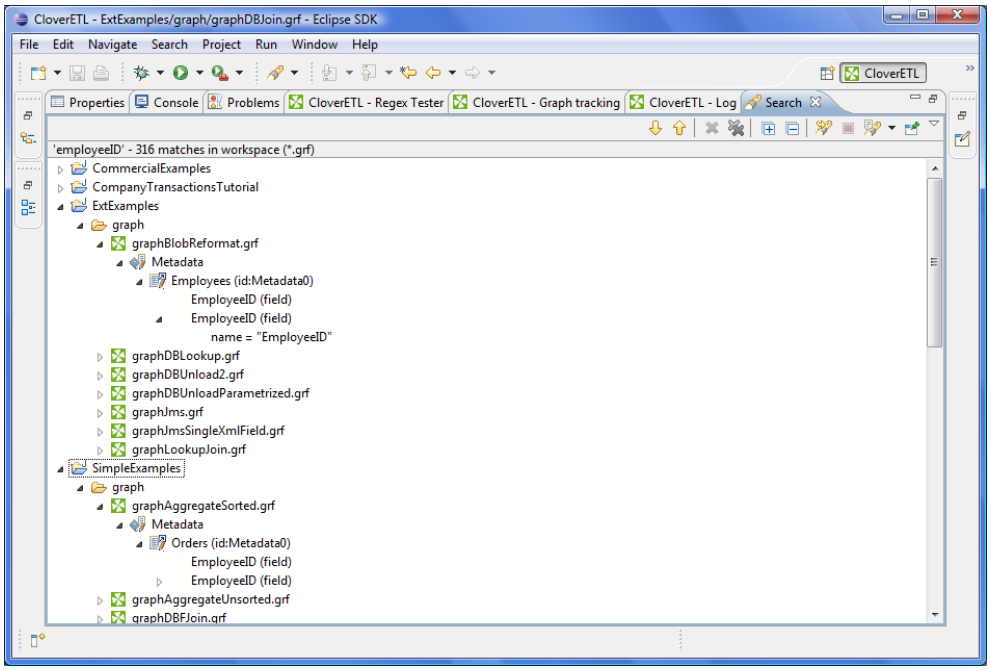


図33.2. 検索結果

第34章 変換

各変換グラフはコンポーネントで構成されています。すべてのコンポーネントはグラフの実行中にデータを処理します。一部のコンポーネントはいわゆる変換を使用してデータを処理します。

変換は、入力のデータがコンポーネントを通過するときに出力のデータに変換される方法を定義するコードです。

注意

変換グラフと変換自体は異なる概念です。変換グラフがコンポーネント、エッジ、メタデータ、接続、参照表、シーケンス、パラメータおよびノートで構成されるのに対し、変換はコンポーネントの属性として定義され、コンポーネントによって使用されます。変換グラフとは異なり、変換はグラフの実行中に実行されるコードです。

変換は、次の3つの属性のいずれかを定義することによって定義できます。

- 各変換は、コンポーネントの3つの属性のいずれかを使用して定義されます。
  - **変換**、**非正規化**、**正規化**など。
  - **変換URL**、**非正規化URL**、**正規化URL**など。
    - これらの属性のいずれかを定義する際、**変換ソース・キャラクタ・セット**、**非正規化ソース・キャラクタ・セット**、**正規化ソース・キャラクタ・セット**などのエンコーディングを指定することもできます。
  - **変換クラス**、**非正規化クラス**、**正規化クラスなど**。
- 変換は一部の変換コンポーネントでは必須であり、他の変換コンポーネントではオプションです。

  変換が許可されるコンポーネントまたは変換を必要とするコンポーネントの概要表は、変換の概要(p.282)を参照してください。
- 各変換は常にJavaで記述でき、ほとんどの変換はClover Transformation Languageでも記述できます。

  CloverETLのバージョン3.0以上のClover Transformation Language (CTL)は、CTL1とCTL2の2つのバージョンが存在します。

  Clover Transformation Languageとその各バージョンの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。

変換の詳細は、変換の定義(p.279)を参照してください。

第35章 Fact Table Loader

Fact Table Loader (FTL)は、データ・ウェアハウスのファクト表に新しいファクトを作成および挿入するために新しいデータ変換を作成する必要がある場合に、時間を短縮できるような設計になっています。

例35.1. 使用例

本番データベースからの入力データがテキスト・ファイルで保存されているとします。ユーザーはこのデータからファクトを作成し、データ・ウェアハウスのファクト表に挿入する必要があります。前述のファクトを作成するプロセスで、ユーザーはテキスト・ファイルのフィールドと選択列のフィールドが等しい行に対するディメンション・キーを検索するために、同じデータ・ウェアハウスのディメンション表を使用する必要があります。**CloverETL**を使用すると、**ExtHashJoin**および**DBLookup**コンポーネントを使用してこれを実行できます。さらに、ユーザーは4つ以上のディメンション表を処理する必要があると想定します。この機能を持つデータ変換(グラフ)を作成するには、10個以上のコンポーネントを使用して、すべての必要なオプションを設定する必要があります。これには時間がかかります。このためにFTLがあります。

FTLは、**CloverETL Designer**に統合されたウィザードです。ユーザーはこのウィザードに関連する情報を入力し、ウィザードによって必要なデータ変換を含む新しい**CloverETL**グラフが作成されます。

次の項で説明する内容は次のとおりです。

1. 「**Fact Table Loader**」ウィザードの起動方法。「Fact Table Loader」ウィザードの起動(p.239)を参照してください。
2. 「**Fact Table Loader**」ウィザードの使用方法。「Fact Table Loader」ウィザードの使用(p.241)を参照してください。
3. 作成されたグラフの外観。作成されたグラフ(p.247)を参照してください。

「Fact Table Loader」ウィザードの起動

この項では、FTLウィザードの2つの起動方法を示し、それらの相違点について説明します。相違点は、最初の方法ではグラフ・パラメータ・ファイルがウィザードで使用可能であり、2番目の方法では使用不可能であることです。プロジェクト・パラメータはファイルのパスをパラメータ化するために使用されます。

次を参照してください。

- プロジェクト・パラメータ・ファイルが有効なウィザード(p.239)
- プロジェクト・パラメータ・ファイルが無効なウィザード(p.241)

プロジェクト・パラメータ・ファイルが有効なウィザード

ウィザードでグラフ・パラメータ・ファイルを使用できるようにするには、ユーザーは既存の**CloverETLプロジェクト**、サブディレクトリまたはその中のファイルを選択する必要があります。

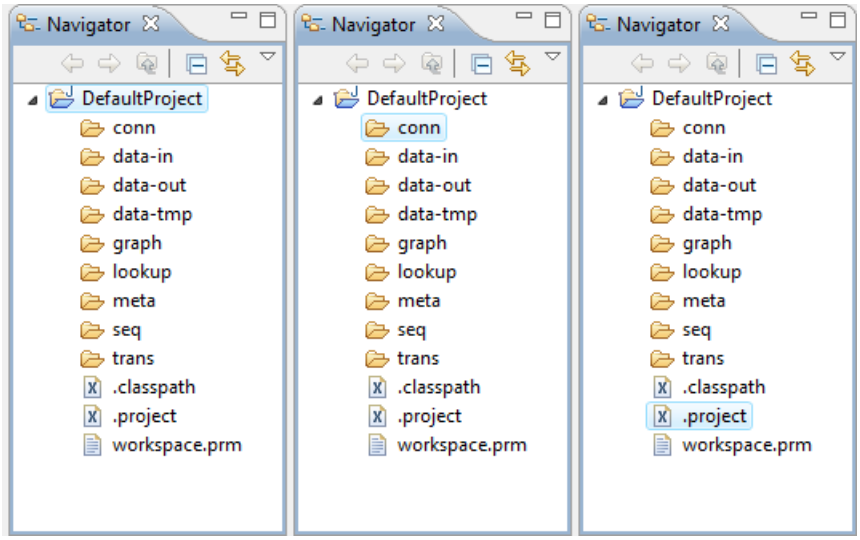


図35.1. 有効な選択

選択する際は、右クリックしてコンテキスト・メニューから「**New**」→「**Other...**」を選択するか、メイン・メニューから「**File**」→「**New**」→「**Other...**」を選択するか、または単に**[Ctrl]を押しながら[N]**を押します。

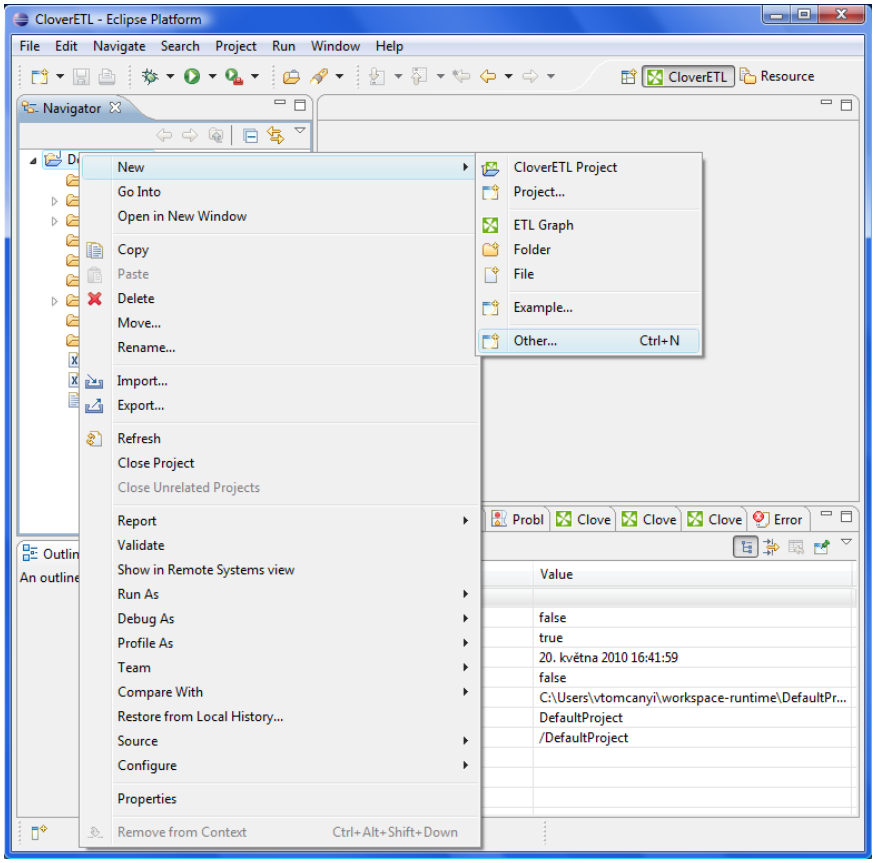


図35.2. FTLウィザードの選択方法

表示されたウィンドウのオプションから「Fact Table Load」を選択します。

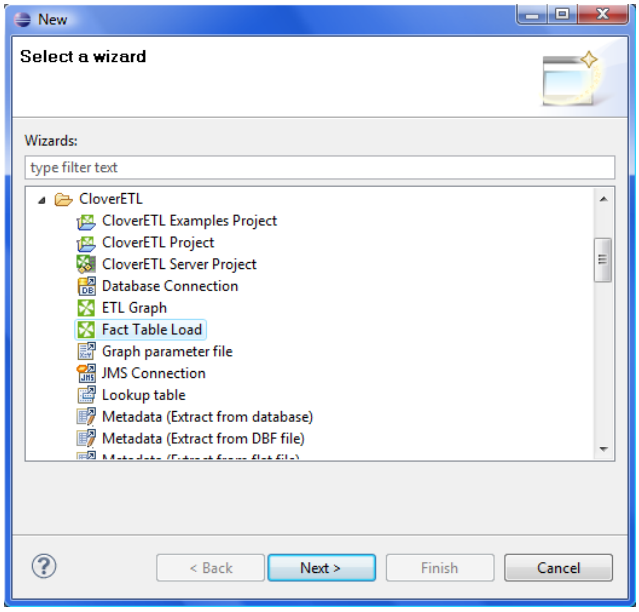


図35.3.「Select a wizard」(新しいウィザード選択ウィンドウ)

プロジェクト・パラメータ・ファイルが無効なウィザード

ウィザードでプロジェクト・パラメータを使用しないようにするには、ユーザーはすべての選択済ディレクトリ、サブディレクトリまたはプロジェクト内のファイルの選択を解除する必要があります。これを行うには、**[Crtl]**ボタンを押しながら選択済項目をクリックします。

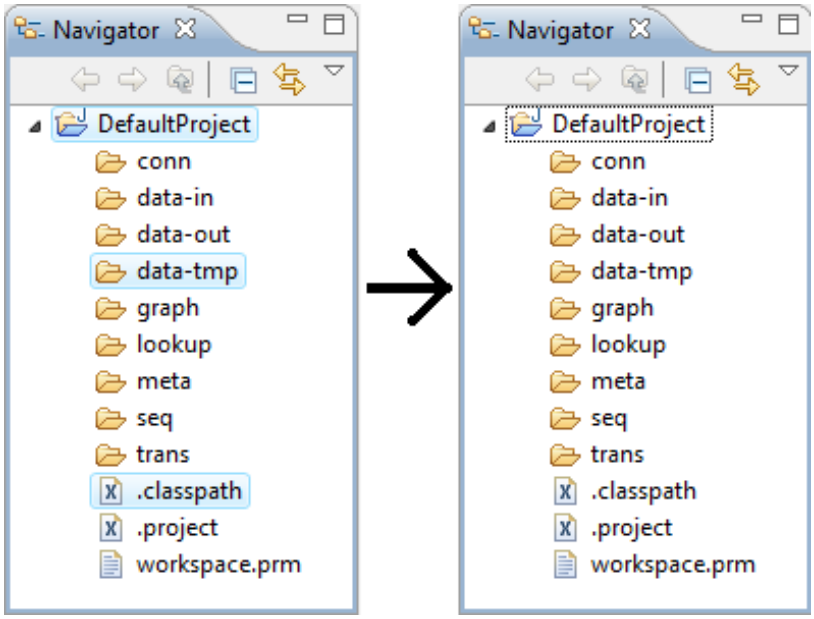


図35.4. 選択解除

選択済のものがない場合は、「**File**」→「**New**」→「**Other...**」を選択するか、単に**[Ctrl]を押しながら[N]**を押し、プロジェクト・パラメータ・ファイルが有効なウィザード(p.239)の手順に従います。

「Fact Table Loader」ウィザードの使用

新しいウィザード選択ウィンドウで「**Next**」ボタンを押すと、すぐに新しいFTLウィザードが起動します。この場合、プロジェクトが関連付けられた状態で起動するウィザードが対象となります。

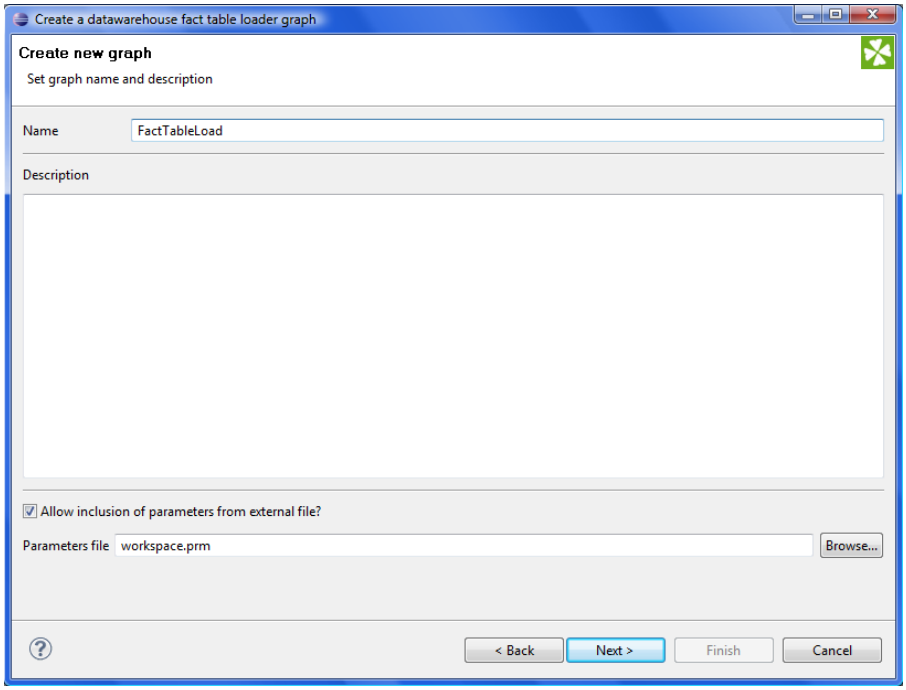


図35.5. 新しいグラフ名のページ

このページでは、ユーザーはグラフ名を入力する必要があり、ウィザードでプロジェクト・パラメータを使用できるかどうかを決定するチェック・ボックスを選択または選択解除できます。

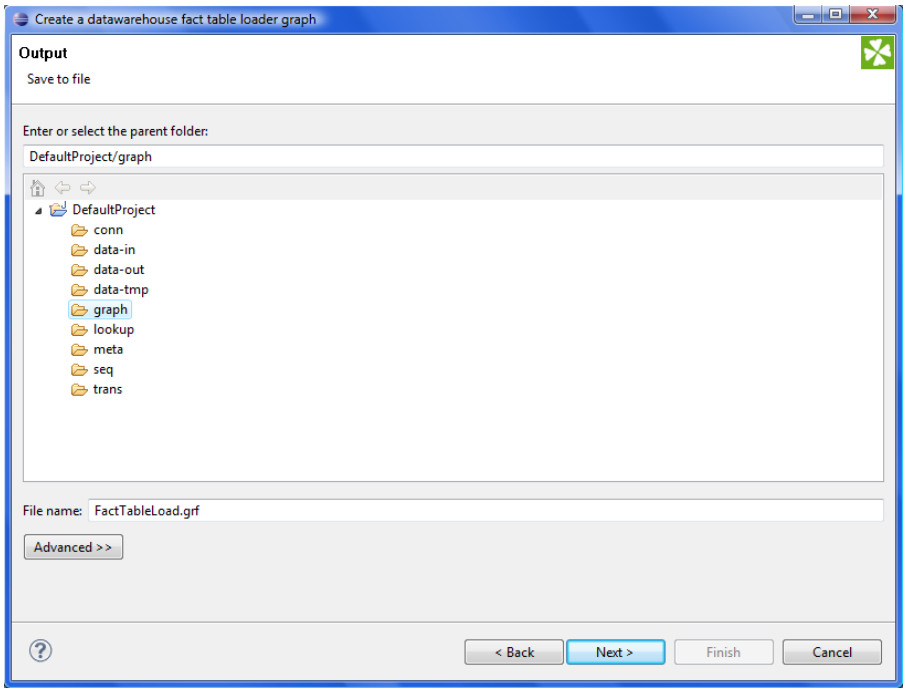


図35.6.「Output」ページ

「**Next**」ボタンをクリックすると、「**Output**」**ページ**が表示されます。このページで、ユーザーは作成したグラフを保存するディレクトリを選択できます。

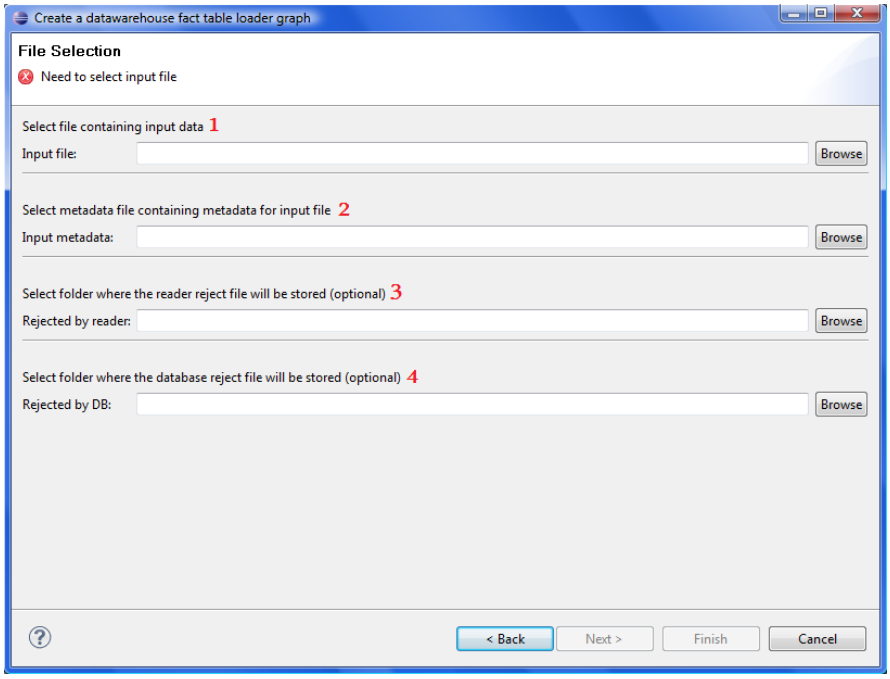


図35.7. 「File Selection」ページ

このウィンドウで、ユーザーは必要なファイルのパスを入力する必要があります。

1. 最初の行には、入力データを含むファイルのパスを入力します。
2. 2番目の行には、入力ファイル(`.fmt`ファイル)のメタデータ定義を含むファイルのパスを入力します。
3. 3番目の行には、リーダーによって拒否されたデータを含む拒否ファイルが格納されるディレクトリのパスを入力します。これはオプションで、この行が空の場合、拒否ファイルは作成されません。
4. 4番目の行には、オプションで、データベース・ライターによって拒否されたデータを含む拒否ファイルが格納されるディレクトリのパスを入力します。

ユーザーが「**Browse**」ボタンをクリックすると、特別なダイアログを使用してプロジェクトを参照できます。

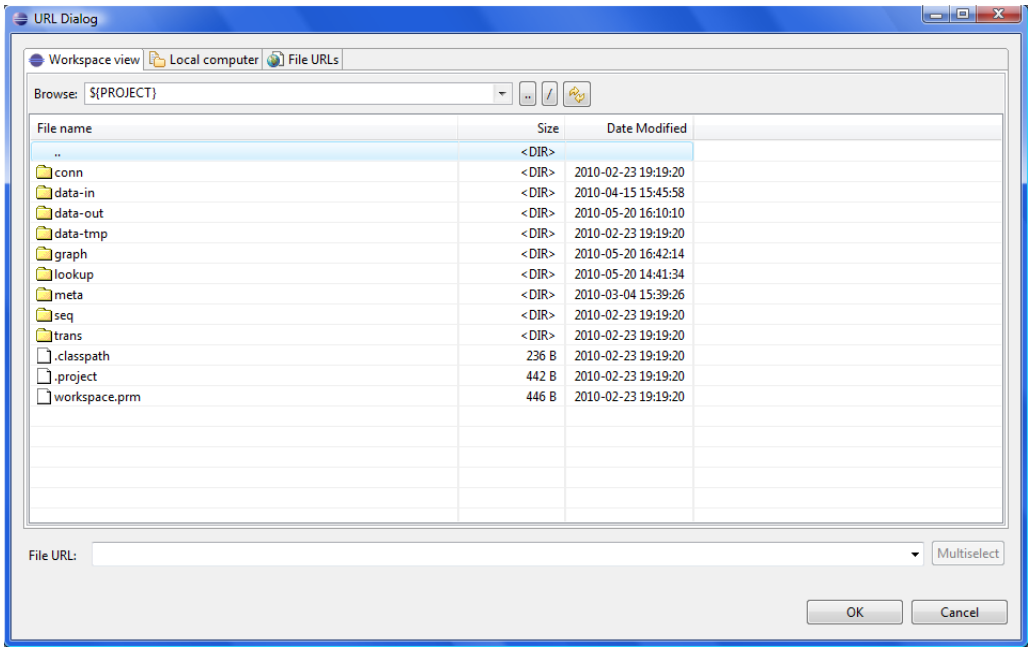


図35.8. URLダイアログ

このダイアログで、ユーザーはプロジェクトまたはファイル・システム全体を参照できます。ユーザーが「**OK**」ボタンをクリックすると、ファイルのパスがパラメータ形式で返されます(プロジェクトが関連付けられている場合)。

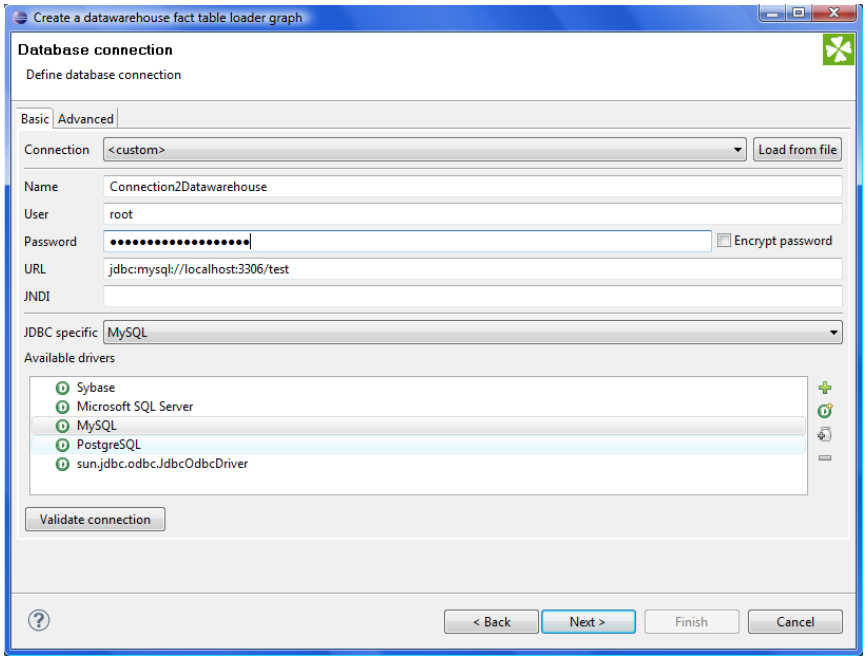


図35.9.「Database Connection」ページ

ユーザーが「**File selection**」**ページ**で「**Next**」ボタンをクリックすると、「Database connection」ページが表示されます。このページで、ユーザーはデータベースへの接続を確立するために必要な情報を指定します。作成される接続は1つのみであるため、データ・ウェアハウスのファクト表およびディメンション表は同じデータベース内にある必要があります。

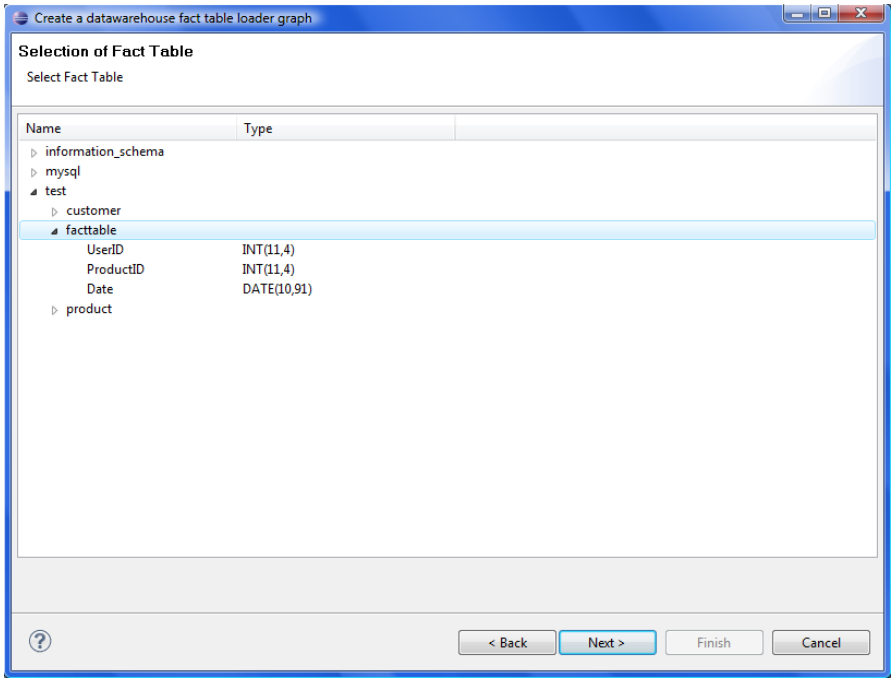


図35.10. ファクト表の選択ページ

ユーザーは、ファクト表として使用する1つの表をデータベースから選択する必要があります。選択できる表は1つのみです。作成されるファクトがこの表に挿入されます。

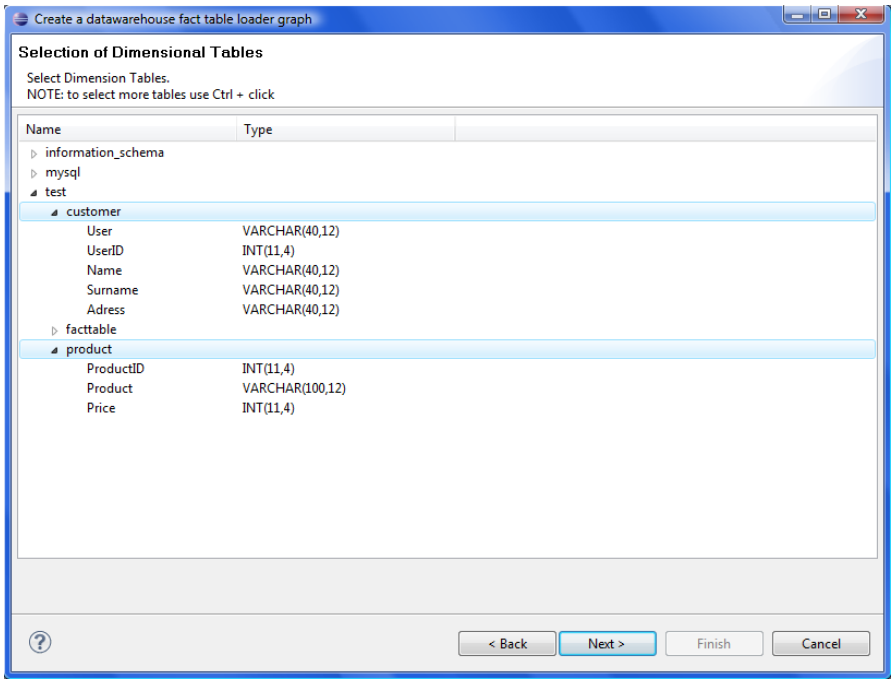


図35.11. ディメンション表の選択ページ

ユーザーは、このウィンドウでデータベースから1つ以上の表を選択する必要があります。これらの表は、データ・ウェアハウスのディメンション表とみなされます。

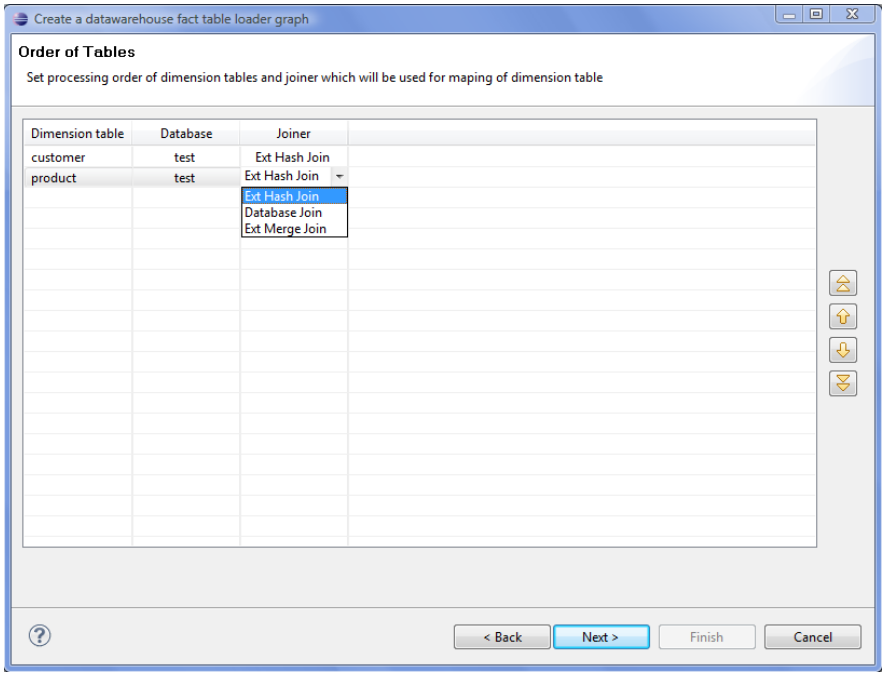


図35.12. 表の順序付けページ

このウィンドウで、ユーザーは選択したディメンション表とデータベースを確認し、マッピングに使用されるジョイナを選択し、このウィンドウに表示されるディメンション表の順序を変更できます。これは、いずれかのディメンション表のディメンション・キーが別のディメンション表内での検索に必要な場合に役立ちます。

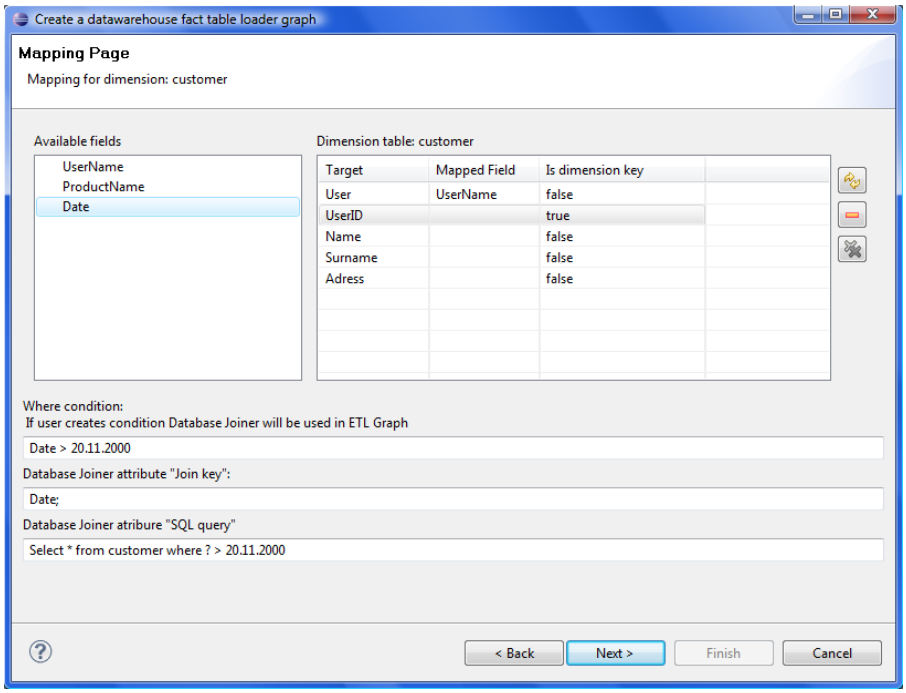


図35.13. マッピング・ページ

ユーザーは、ドラッグ・アンド・ドロップを使用して、使用可能なフィールドをディメンション表のフィールドにマップします。ユーザーは、ディメンション・キーとみなされ、ファクトに追加されるディメンション表のフィールドを選択することもできます。これを行うには、列「**Is dimension key**」のブール値を`true`に設定します。このウィンドウでSQL条件を作成することもできます。ユーザーは、他の**DBJoin**属性が生成される`where`条件を作成できます。ユーザーがディメンション表に対して`where`条件を作成し、**ExtHashJoin**または**ExtMergeJoin**を選択した場合、グラフでは**DBJoin**がかわりに使用されます。**マッピング・ページ**はディメンション表ごとに表示されます。

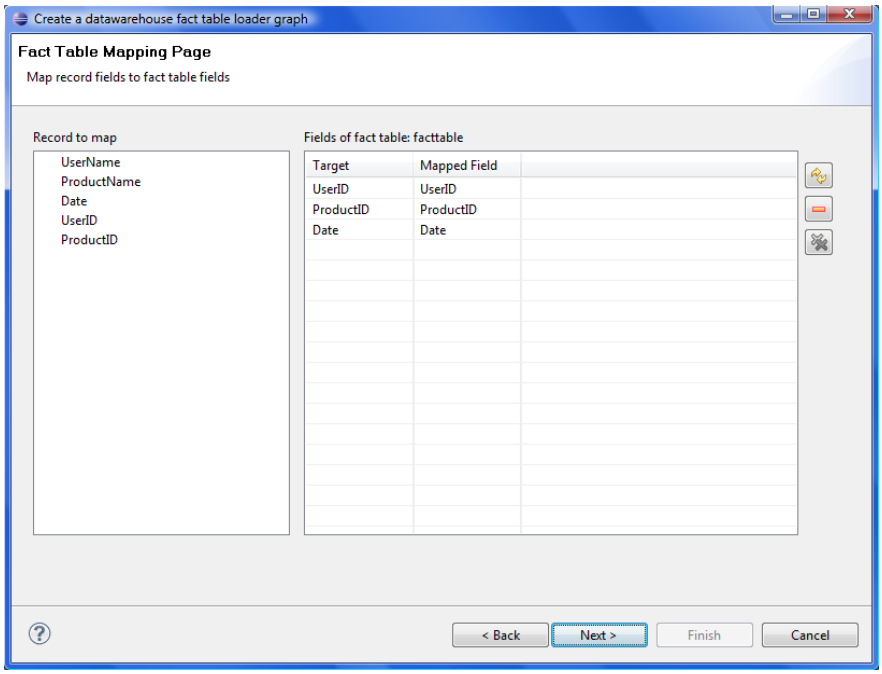


図35.14. ファクトのマッピング・ページ

ユーザーは、作成されたファクトをこのウィンドウでファクト表にマップします。エディタでは、名前が同一の場合は名前でフィールドを一致させる自動ルールが使用されます。

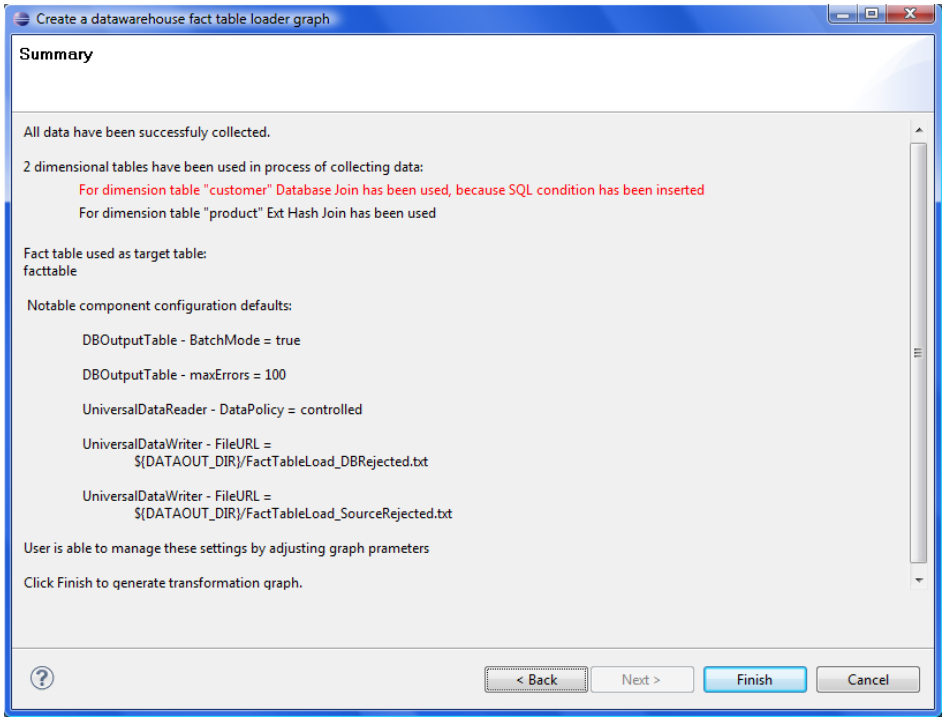


図35.15.「Summary」ページ

ウィザードの最後のページには、ウィザードからの情報が表示されます。**CloverETL**グラフで作成される新しいパラメータに関する情報も提供されます。ユーザーが拒否ファイルの出力ディレクトリに対するパスを入力しなかった場合、最後の3つのパラメータは作成されません。ユーザーにより選択されたジョイナのかわりに**DBJoin**が使用されるかどうかの警告も表示されます。

作成されたグラフ

「**Finish**」ボタンをクリックすると、ウィザードによって新しい**CloverETL**グラフが生成されます。ユーザーがこのグラフを保存すると、実行の準備ができます。

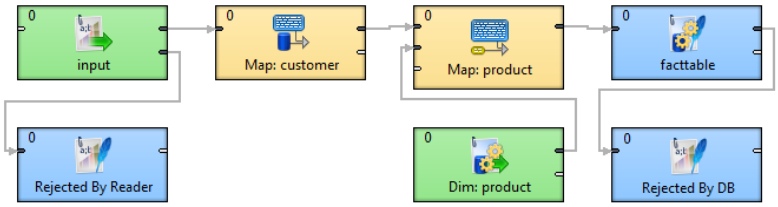


図35.16. 作成されたグラフ

このグラフには、作成されたパラメータも含まれます。

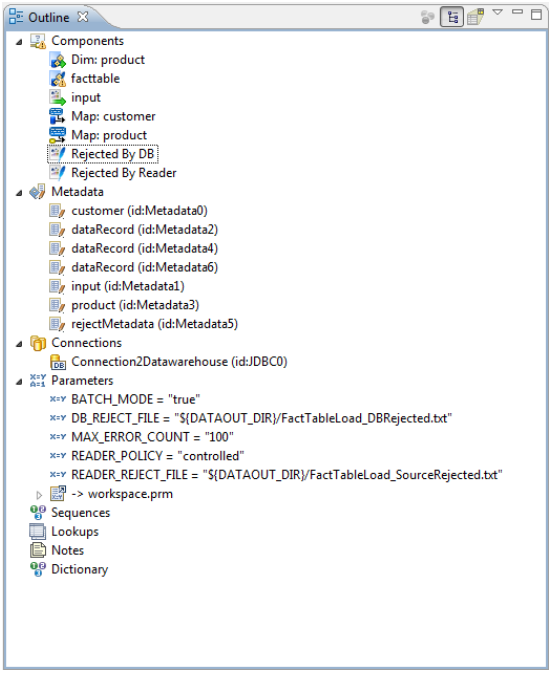


図35.17. グラフ・パラメータ

第VI部　ジョブフロー

第36章 ジョブフローの概要

概要

CloverETLジョブフローの内容

CloverETLジョブフロー・モジュールを使用すると、ETLグラフを他のアクティビティと組み合せて複雑なプロセスを作成でき、これにより、編成、条件付きジョブ実行およびエラー処理が提供されます。ジョブフローに含めることができるアクションは次のとおりです。

- CloverETL ETLグラフ
- ネイティブOSアプリケーションおよびスクリプト
- Webサービス(REST/SOAP)コール
- ローカル・ファイルおよびリモート・ファイルを使用した操作

専用のジョブフロー・コンポーネントとして使用可能な前述のアクションに加えて、ジョブフローにはETLコンポーネントを含めることもできます。これにより、ジョブフロー・ロジックをより柔軟に構成でき、外部環境からジョブフローに構成を渡すための追加オプションが提供されます。

ヒント

ジョブフローでDBInputTable ETLコンポーネントを使用して、ジョブのリストおよびそのパラメータをデータベースから読み取り、ジョブフロー・コンポーネントのExecuteGraphを使用して、リスト上の各ジョブを必要なパラメータで実行できます。最後に、EmailSenderコンポーネントをフローに追加して、実行時のエラーについて通知できます。

ジョブフローを編集する際のエディタの外観は若干変更されており、コンポーネントは異なる背景色で丸い形となり、特に、**「Palette」**のコンテンツが変わります。「Palette」からドラッグできるコンポーネントを変更するには、**「Window」→「Preferences」→「CloverETL」→「Components in Palette」**に移動します。

設計および実行

ジョブフローの実行には、CloverETL Server環境が必要です。ただし、ジョブフローは、スタンドアロンCloverETL Designerを使用してオフラインで設計できます。開発者は、CloverETL Designerで使用可能なサーバー統合モジュールを使用して、Server上で対話形式でジョブフローを構成、デプロイおよび実行できます。Server環境でジョブフロー・プロセスの実行を自動化するために、ジョブフローは、スケジューラやファイル・トリガーなどの既存の自動化や、SOAPサービスおよびRESTサービスを含むサーバーAPIに完全に統合されます。

ジョブフロー・モジュールの詳細分析

CloverETL Designerにはジョブフロー・モジュールの設計時関数要素が含まれており、一方、CloverETL Serverには必要なランタイム・サポートおよび自動化機能が含まれています。

CloverETL Designerのジョブフロー要素は次のとおりです。

- **ジョブフロー・エディタ**: ジョブフロー(*.jbf)を設計するための専用のUI。ビジュアル・エディタを開くと、エディタの外観が若干変わります。コンポーネントは異なる背景色で丸い形となり、特に、**「Palette」**のコンテンツが変わります。
- **パレットのジョブフロー・コンポーネント**: ジョブフロー関連のコンポーネントは、第57章「ジョブ制御」(p.684)および第58章「ファイル操作」(p.734)の各項で確認できます。使用できるその他のETLコンポーネントには、

第61章「その他」(p.778)カテゴリのWebServiceClientおよびHTTPConnectorがあります。第57章「ジョブ制御」(p.684)のコンポーネントの一部は、ETLのパースペクティブでも使用可能です。

- **ProfilerProbeコンポーネント**:「データ品質」パレット・カテゴリで利用可能であり、このコンポーネントでは、ETLグラフまたはジョブフローの一環としてCloverETLプロファイラ・ジョブを実行できます。
- **事前定義済メタデータ**: Designerには、ジョブフロー・コンポーネントにより提供される、予期される入力または出力を記述した事前定義済のメタデータ・テンプレートが含まれています。テンプレートによってメタデータ・インスタンスが生成され、開発者はここで変更するかどうかを決定できます。テンプレートは、グラフ・エディタのエッジ・コンテキスト・メニュー「New metadata from template」から使用できます。
- **トラッキング可能フィールド**: ジョブフロー・パースペクティブのメタデータ・エディタでは、トークン・メタデータ内で、トラッキング可能として選択済フィールドをフラグ付け(p.256)できます。トラッキング可能フィールドの値は、実行時にジョブフロー・コンポーネントによって自動的に記録されます(次のトークン中心ロギングの説明を参照してください)。この目的は、デバッグまたは事後ジョブ分析における実行フローの追跡を簡単にすることです。
- **CTL関数**: 外部環境との対話、つまり、環境変数(`getEnvironmentVariables()`)、グラフ・パラメータ(`getParamValues()`)およびJavaシステム・プロパティ(`getJavaProperties()`)へのアクセスを可能にする一連のCTL関数。
- **ログ・コンソール**: ジョブフローの実行ログおよびエラーを含むコンソール・ビュー。

CloverETL Serverのジョブフロー要素は次のとおりです。

- **実行履歴**: 実行全体およびジョブフローの個々のステップをその親子関係とともに含む階層ビュー。ステータス、パフォーマンス統計に加えて、ジョブフローのステップ間で渡される実際のパラメータおよびディクショナリの値リストを含みます。
- **自動化されたロギング**: トークン中心ロギングでは、特定のステップの実行をトリガーするトークンを追跡できます。トークンは容易に識別できるように一意に順序付けされていますが、開発者はデフォルトの数値トークン識別子とともに追加情報(ファイル名、メッセージ識別情報、セッション識別子など)を記録することもできます。
- **自動化モジュールとの統合**: 各実行でジョブフローの起動およびユーザー定義パラメータの引渡しを可能にするために、すべてのCloverETL Serverモジュールにスケジューラ、トリガーおよびSOAP APIが含まれています。

重要な概念

動的な属性設定

ジョブフロー・コンポーネント(WebServiceClientおよびHTTPConnectorコンポーネントも同様)では、コンポーネント属性を静的に設定できますが、実行時には接続済入力ポートから動的に構成することも可能です。

接続済入力ポートから受信したトークンから値が読み取られ、「Input Mapping」プロパティにより定義されたマッピングに従ってコンポーネントの属性にマッピングされます。動的に設定された属性は、任意の静的なコンポーネント構成とマージされますが、競合が発生した場合は動的な設定が静的構成をオーバーライドします。結合されたコンポーネント構成は、最終的にトークンによりトリガーされる実行に使用されます。

ヒント

動的な構成は、forループの実装に使用できます。このことは、ExecuteGraph/ExecuteJobflow/ExecuteProfilerJob内に静的構成ジョブを含め、かつ、入力マッピングを介して動的にジョブ・パラメータを渡すことによって行います。

パラメータの動的構成は、HTTPConnectorまたはWebServiceClientでも使用され、ターゲットURLを動的に構成したり、HTTP GETパラメータをURLに含めたり、接続をリダイレクトできます。

パラメータの受渡し

ExecuteGraphおよびExecuteJobflowコンポーネントを使用すると、グラフ・パラメータおよびディクショナリ値を親から子に渡すことができます。ディクショナリについては、親は子のディクショナリから値を取得することも可能です。このことは、子がその実行を終了した後にのみ可能です。

ジョブフロー内の2ステップ間でディクショナリを渡すには、次が必要です。

1. 子のディクショナリ内で必要なディクショナリ・エントリを宣言します。
2. エントリを入力(エントリ値は親が設定)または出力(親が子から値を取得)としてタグ付けします。
3. 親のExecuteGraph/ExecuteJobflow/ExecuteProfilerJobコンポーネントの「Input Mapping」または「Output Mapping」プロパティで各エントリのマッピングを定義します。
4. 子が親にエントリを渡す場合、値はSuccess、FailまたはSetJobOutputコンポーネントを使用して子の実行の間に設定できますが、CTLコードを使用しても設定できます。
5. 子グラフ内で宣言されるパラメータ(ローカルまたはパラメータ・ファイルから)は、親グラフ内のExecuteGraph/ExecuteJobflowの「Input Mapping」で設定できます。子がその実行時にパラメータ値を変更することや親がパラメータ値を子から取得することはできません。

子グラフで宣言されたパラメータおよびすべての入力/出力ディクショナリ(存在する場合)のエントリは、ExecuteGraph/ExecuteJobflowの「Input Mapping」または「Output Mapping」に適宜、自動的に表示されます。

パラメータとディクショナリは次のように使い分けます。

- **パラメータ**: 子グラフの構成(コンポーネント設定やグラフ・レイアウトなどの変更)
- **ディクショナリ**: 親に値を返すために子が処理するデータ(または値)の受渡し

ヒント

パラメータはコンポーネント属性内の任意の場所への配置や、グラフの実行前に展開されるテキスト・マクロとしての配置が可能で、これらは、実行時のグラフ・レイアウトを大幅に変更するために使用されます。これらは、グラフ内のブランチ処理の無効化、出力の特定の宛先へのルーティング、またはデータセット制限/フィルタ(営業日、製品タイプ、アクティブなユーザーなど)の受渡しによって再使用可能なロジックを作成するために使用します。これらは、集中化した方法による環境情報の受渡しにも使用されます。

ディクショナリは、結果の値を親に戻すために、または子が変数の初期値を親から受け取るために使用します。GetJobInputおよびSetJobOutputコンポーネントを使用して、ディクショナリ・エントリの受信または設定の処理を強調表示できます。

パススルー・マッピング

任意のフィールドは、ジョブフロー・コンポーネントを通過させることができます。「Output mapping」プロパティを受信トークン・フィールドのマッピングに使用して、他のコンポーネント出力値と組み合せて出力できます。

ヒント

Webサービスでは、パススルー・マッピングを使用してログイン操作を1回のみ実行して、次にセッション・トークンを複数のHTTPConnectorまたはWebServiceClientコンポーネントに渡すことができます。

実行ステータスのレポート

ジョブフロー・コンポーネントは、2つの標準出力ポートを経由して実行アクティビティの成功または失敗をレポートします。成功出力ポート(0)はデフォルトでは成功したすべての実行に関する情報を渡し、エラー・ポート(1)は失敗したすべての実行に関する情報を渡します。

開発者は、すべてのジョブフロー・コンポーネントで使用可能な「Redirect error output」属性を使用して、失敗した実行でも成功出力にリダイレクトすることを選択できます。

エラー処理

ExecuteGraph、ExecuteScript、ExecuteJobflow、ExecuteProfilerJobおよびファイル操作コンポーネントは、通常のJava try/catchブロックとして動作します。実行されたアクティビティが成功して完了した場合、例外がスローされなかった状況と同様に、結果は0番成功ポートにルーティングされます。

コンポーネントにより開始されたアクティビティが失敗した場合、エラーは補正ロジックを適用できる1番エラー・ポートにルーティングされます。この動作は、コード内のcatchブロックにより処理される例外と同様です。

エラー・ポートにエッジが接続されていない場合、コンポーネントは通常のJava例外をスローして処理を終了します。エラーが発生したジョブが親ジョブによって開始されていた場合、例外により親のExecuteでエラーが発生し、ここでさらに例外の処理またはスローが選択される場合があります。

ヒント

try/catchアプローチを使用すると、処理中の特定のエラーを処理する一方で、手動による操作が必要なエラーを意図的に処理されないままにするロジックを構築できます。捕捉されないエラーがある場合は処理が中断され、ジョブが「Server Execution History」に失敗として表示され、製品サポート・チームが処理できます。

ジョブフローまたはETLグラフ内の**Fail**コンポーネントを使用して、エラーがスローされていることを強調表示できます。これは入力データを使用した有意なエラー・メッセージの作成またはエラー・ステータスを親ジョブに示すディクショナリ値の設定に使用されます。

ジョブフロー実行モデル: 単一トークン

ジョブフロー内のアクティビティは、デフォルトではエッジ方向の順序で順次実行されます。フローの最初のコンポーネント(入力を持たない)が実行を開始し、アクションの終了後に出力トークンを生成し、フロー内の次に接続されているコンポーネントにそれを渡します。トークンは、次のコンポーネントのトリガー・イベントとして機能し、次のジョブが開始されます。

これは、ETLグラフの実行とは異なります。ETLグラフの実行では、最初のコンポーネントが2番目のコンポーネントのデータ・レコードを生成しますが、両方のコンポーネントが並行して同時に実行されます。

ジョブフロー・コンポーネント出力が(SimpleCopyなどにより)分岐されて2つの他のジョブフロー・コンポーネントの入力に接続されている場合、トークンによってこれら両方のコンポーネントがトリガーされ、並行して同時に実行が開始されます。

ETLグラフで使用可能なフェーズの概念を、ジョブフローでも使用できます。

ヒント

ブランチ処理を使用すると、処理を分岐させ、複数のパラレル・ブランチで実行できます。実行後にブランチを結合する必要がある場合は、**Combiner**、**Barrier**または**TokenGather**コンポーネントを使用します。コンポーネントは実行結果を処理する方法に応じて選択します。

ジョブフロー実行モデル: 複数トークン

基本的なシナリオでは1つのトークンのみがジョブフローを通過します。これは、ジョブフロー内の各アクションは厳密に1回のみ開始されるかまったく開始されないことを意味します。より高度なジョブフローの使用では、複数のトークンをジョブフローに投入して、特定のアクションの実行を繰り返しトリガーします。

複数トークンの概念により、開発者はforループの反復を実装でき、具体的には、CloverETLジョブフローの**パラレルforループ・**パターンをサポートできます。

パラレルforループ・パターンでは、従来のforループとは異なり、ループの各新規反復が独立して前の反復とともに並行して開始されます。ジョブフローに関しては、このことは2つのトークンがジョブフローに投入された場合、最初のトークンによりトリガーされたアクションは、後に到着した2番目のトークンによりトリガーされたアクションと同時に実行される可能性があることを意味します。

パラレル実行は望ましくない場合があるため、ループ本体またはループ直後のアクションの順次実行を強制する方法を理解することが重要です。

- **ループ本体の順次化**: ループ本体を形成する複数のステップの順次実行を強制し、このことはパラレルforループの従来のforループへの変換を意味します。

  ループ本体を順次化して従来のforループとして動作させるには、ループ本体内のアクションを、同期実行モードで実行しているExecuteJobflowコンポーネントにラップします。このことにより、コンポーネントは基本ロジックの完了を待機してから新しい反復を開始するようになります。

ヒント

ディメンション表とファクト表それぞれにロードするデータを含む2つのファイルを受信するデータ・ウェアハウスのシナリオを想定します(たとえば、dim-20120801.txtおよびfact-20120801.txtの2つのファイルを毎日受信します)。データは初めにディメンション表に挿入される必要があります。その後にのみ、ファクト表をロードできるようになります(ディメンション・キーが見つかるように)。加えて、前日のデータは、当日のデータ(ディメンションおよびファクト)をロードする前にロードされている必要があります。データ・ウェアハウスで時系列のデータ変更履歴を保持する必要があるため、これが必要となります。

  ジョブフローを使用してこれを実装するには、DataGeneratorコンポーネントを使用して1週間相当のデータを生成し、これをループの本体(1日分のウェアハウスのロード)を実装するExecuteJobflowにフィードします。具体的には、ExecuteJobflowには、LoadDimensionForDayおよびLoadFactTableForDayの2つのExecuteGraphコンポーネントが含まれます。

- **ループ後のアクション実行の順次化**: ループ直後のアクションを各新規反復によりトリガーするのではなく、アクションをすべての反復が完了した後に1回のみトリガーする必要があります。

  ループに続くアクションをすべての反復の終了後に実行するには、「All tokens form one group」オプションが有効化されたBarrierコンポーネントのアクションを事前に置きます。このモードでは、Barrierは初めにすべての受信トークンを収集し(すべての反復の終了を待機)、次に単一の出力トークンを生成します(制御フローはforループの直後にアクションに渡されます)。

ヒント

前述の例で1週間相当のデータをデータ・ウェアハウスにロードした後に、索引を再構築および有効化する必要があります。このことは、すべてのファイルが処理された後のみに実行可能です。このことを行うには、Barrierコンポーネントをジョブフローの別のExecuteGraphの後に追加します。

  Barrierによってすべてのロードが完了し、その後にのみ、最後のExecuteGraphステップで、索引を作成するために必要なSQL文を伴う1つ以上のDBExecuteコンポーネントが起動されます。

エラーによる停止

複数のトークンが1つのExecuteGraph/ExecuteJobflow/ExecuteProfilerJobコンポーネントをトリガーする場合に、基本ジョブに処理不可能なエラーが生じた場合、コンポーネントのデフォルトの動作では以降のトークンの処理を行わず、例外がスローされている間の新しいジョブの開始を回避します。

エラーに関係なく処理を続行する必要がある場合は、ExecuteGraph/ExecuteJobflow/ExecuteProfilerJobの「Stop processing on fail」属性を使用してコンポーネントの動作を変更できます。

同期または非同期実行

ExecuteGraph、ExecuteJobflowおよびExecuteProfilerJobは、デフォルトでは子ジョブを同期的に実行します。これは、これらのジョブが基本ジョブの完了を待機することを意味します。その後のみ、これらは出力トークンを生成して次のコンポーネントをトリガーします。コンポーネントがジョブの完了を待機している間、その他のトークンが到着しても新しいジョブを開始しません。

高度なユースケースでは、コンポーネントは非同期実行もサポートします。これは「Execution type」プロパティにより制御されます。実行の非同期モードでは、コンポーネントは新しい入力トークンが到着するとすぐに子ジョブを開始し、子ジョブの終了を待機しません。このような場合、ジョブ統計を使用できないことがあるため、ExecuteGraph/ExecuteJobflow/ExecuteProfilerJobコンポーネントはジョブ実行IDのみを出力します。

開発者は、**MonitorGraph**または**MonitorJobflow**コンポーネントを使用して、非同期で開始されたグラフまたはジョブフローを待機できます。

ヒント

非同期実行モードは、ジョブの合計数が未知の場合にジョブを並行して開始する場合に便利です。

ロギング

ジョブフローのコンポーネントによって生成されるログ・メッセージは、トークン中心です。トークンはジョブフロー内の基本的なトリガー・メカニズムを表し、実行中の1つのジョブフロー・インスタンス内には、1つまたは複数のトークンが存在できます。トークンによりトリガーされたアクティビティを簡単に識別できるように、トークンには自動的に番号が付与されます。

例36.1. ジョブフロー・ログの例 - サブグラフを開始しているトークン

```
2012-08-21 15:27:36,922 INFO 1310734 [EXECUTE_GRAPH0_1310734] Token [#3] started
etlGraph:1310735:sandbox://scenarios/jobflow/SubGraphFast.grf on node node01.
```

ログの形式はわかりやすく、`date time RunID [COMPONENT_NAME] Token [#number] message`となります。

すべてのジョブフロー・コンポーネントは、トークン・ライフサイクルに関する情報を自動的に記録します。重要なトークン・ライフサイクル・メッセージを次に示します。

- **Token created**: 新しいトークンがジョブフローに投入されました。
- **Token finalized**: トークン・ライフサイクルがコンポーネント内で終了しました。終端コンポーネント(Fail、Successなど)で終了したか、終端コンポーネント(SimpleCopyなど)で複数のトークンが発生しました。
- **Token sent**: トークンが接続済出力ポートから後続のコンポーネントに送信されました。
- **Token link**: 受信(終了)トークンと(Barrier、Combineなどで)結果として作成された新しいトークンとの関係が記録されています。

- **Token received**: 接続済入力ポートで、前のコンポーネントからのトークンが読み取られました。
- **Job started**: トークンによりExecuteGraph/ExecuteJobflow/ExecuteScript内のジョブの実行がトリガーされました。
- **Job finished**: 子ジョブの実行が終了し、コンポーネントは次の処理を続行します。
- **Token message**: トークンによってトリガーされて、(ユーザー定義の)ログ・メッセージがコンポーネントによって生成されました。

メタデータ・フィールドのトラッキング

さらに、開発者はトラッキング可能フィールド(p.160)の概念に基づいて、トークンのフィールドに格納されている追加情報のロギングを有効化できます。トラッキングされたフィールドはログに次のように表示されます(`fileName`フィールドの例)。

```
2012-10-08 14:00:29,948 INFO 28 [EXECUTE_JOBFLOW0_28] Token [#1
(fileURL=${DATA_IN_DIR}/inputData.txt)] received from input port 0.
```

ヒント

ファイル名、ディレクトリおよびセッション/ユーザーIDは多くの場合反復されるため、これらは有用なトラッキング可能フィールドとして機能します。

高度な概念

デーモン・ジョブ

CloverETLジョブフローでは、子ジョブがその親より長く存続することが可能です。デフォルトでは、この動作は無効化されており、これは、親ジョブが終了または中止した場合、そのすべての子ジョブも同様に終了することを意味します。この動作は、ExecuteGraph/ExecuteJobflow/ExecuteProfilerJobの「Execute as daemon」プロパティで変更できます。

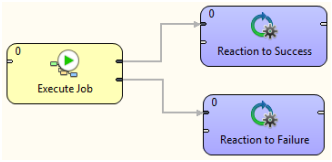
ジョブの強制終了

エラーに対してtry/catch制御を使用しているときは、ジョブフローでは開発者がKillGraphおよびKillJobflowコンポーネントを使用して強制的にジョブを終了できます。ジョブの実行は、その一意のIDを使用して強制終了でき、大量のジョブを終了する場合は、実行グループの概念を使用できます。ジョブはExecuteGraph/ExecuteJoblowコンポーネントの「Execution group」プロパティを使用して実行グループにタグ付けできます。

第37章 ジョブフローの設計パターン

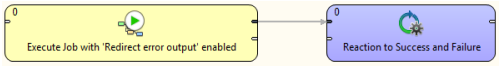
Try/Catchブロック

すべての実行コンポーネントは、単純に成功および失敗に対処することが可能です。ジョブが成功した場合、トークンは第1出力ポートに送信されます。ジョブが失敗した場合、トークンは第2出力ポートに送信されます。



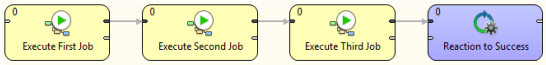
Try/Finallyブロック

すべての実行コンポーネントでは、エラー・トークンを第1出力ポートにリダイレクトすることが可能です。「Redirect error output」属性を使用して、ジョブの成功および失敗に対して一律に対処します。



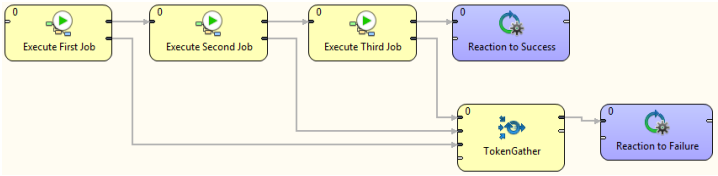
順次実行

ジョブの順次実行は、単純な実行コンポーネントのチェーンにより実行されます。いずれかのジョブでエラーが発生すると、ジョブフローは終了します。



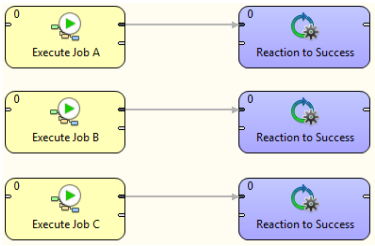
エラー処理付きの順次実行

順次ジョブ実行は、共通のジョブ失敗処理により拡張できます。TokenGatherコンポーネントは、ジョブの失敗を表すすべてのトークンの収集に適しています。



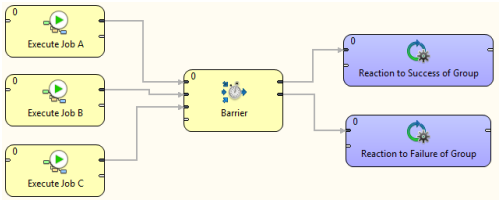
並列実行

並列ジョブ実行は、独立した一連の実行により単純に実現できます。個別の各ジョブに対して成功および失敗への対処を使用できます。



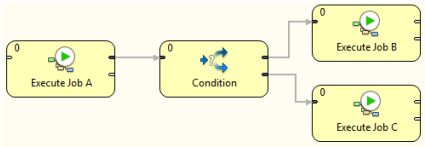
共通の成功/エラー処理付きの並行実行

Barrierコンポーネントを使用すると、並行実行ジョブの成功または失敗への対処が可能になります。デフォルトでは、並行実行ジョブのグループは、すべてのジョブが成功して終了した場合に成功とみなされます。Barrierコンポーネントの多様な設定より、この方法におけるすべてのニーズを満たすことができます。



条件付き処理

条件付き処理は、トークンのルーティングにより実現します。Conditionコンポーネントを使用して、ジョブAの結果に基づいてその後に使用する処理ブランチを決定できます。

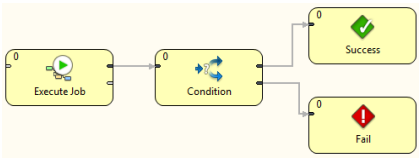


ディクショナリ駆動型ジョブフロー

親ジョブフローは、入力ディクショナリ・エントリを使用していくつかのデータを子ジョブに渡すことができ、これらのジョブ・パラメータがGetJobInputコンポーネントにより読み取られ、その後の処理に使用されます。一方、ジョブフロー結果はSetJobOutputコンポーネントを使用して出力ディクショナリ・エントリに書き込まれます。これらの結果が親ジョブフローで使用されます。

失敗制御

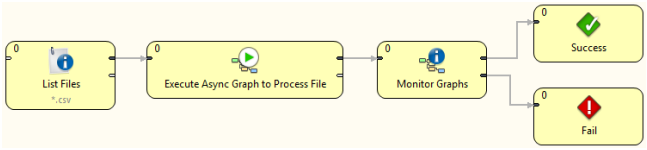
Failコンポーネントを使用して、ジョブフローの処理を意図的に停止させることができます。Failコンポーネントを使用して、ユーザー指定のメッセージをレポートできます。



非同期グラフ実行

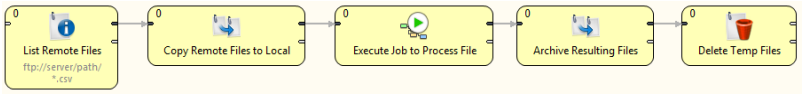
数が一定ではないジョブの並行処理は、非同期ジョブ処理により実現できます。次の例は、すべてのcsvファイルを並行して処理する方法を示しています。初めに、ListFilesコンポーネントによりすべてのファイル名がリスト化されます。ExecuteGraphコンポーネントにより、各ファイル名に対して1つのグラフが非同期で実行されます。グラフ実行ID (runId)が、すべてのグラフ結果を待機するMonitorGraphコンポーネントに送信されます。

非同期実行は、グラフおよびジョブフローのみで使用可能です。



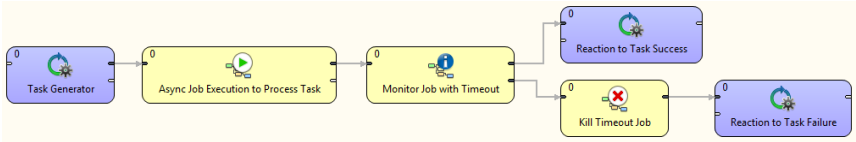
ファイル操作

ジョブフローでは、ファイルのリスト化、作成、コピー、移動および削除の、一連のファイル操作コンポーネントが提供されます。このユースケースは、一連のリモート・ファイルを処理するための、ファイル操作コンポーネントの使用方法を示しています。ファイルはリモートFTPサーバーからダウンロードされ、各ファイルがジョブにより処理され、結果が最終宛先にコピーされ、一時ファイルが作成された場合はそれらが削除されます。



グラフの中断

グラフおよびジョブフローは、それぞれKillGraphおよびKillJobflowコンポーネントを使用して明示的に中断できます。次の例は、タスクのリストを並行して処理する方法を示し、ユーザー指定のタイムアウトに達したジョブが自動的に中断されることを示しています。



第VII部 コンポーネントの概要

第38章 コンポーネントの概要

コンポーネントの基本情報は、第19章「コンポーネント」(p.97)を参照してください。

グラフ・エディタのコンポーネント・パレットでは、すべてのコンポーネントが次のグループに分類されています。リーダー(p.339)、ライター(p.459)、トランスフォーマ(p.576)、ジョイナ(p.653)、クラスタ・コンポーネント(p.750)およびその他(p.778)。各グループについて、順を追って説明します。

もう1つ、**非推奨**と呼ばれるカテゴリがあります。これは今後使用を避ける必要があるため、説明しません。

ここまで、コンポーネントをグラフに貼り付ける方法について説明しました。次に、コンポーネントのプロパティおよびその構成方法について説明します。任意のグラフ・コンポーネントのプロパティを、次の方法で構成できます。

- ***グラフ・エディタ***でコンポーネントをダブルクリックできます。
- 「**Outline**」ペインでコンポーネントまたはその項目をクリックして、「**Properties**」タブで項目を編集できます。
- 「**Outline**」ペインでコンポーネント項目を選択して**[Enter]**を押すことができます。
- ***グラフ・エディタ***または「**Outline**」ペインでコンポーネントを右クリックしてコンテキスト・メニューを開くことも可能です。次に、コンテキスト・メニューから「**Edit**」項目を選択して、**コンポーネント編集**のウィザードでコンポーネントを編集します。

第39章 コンポーネント・パレット

CloverETL Designerでは、**コンポーネント・パレット**ですべてのコンポーネントが提供されます。ただし、**パレット**に含めるものと含めないものを選択できます。一部のコンポーネントのみを選択する場合は、メイン・メニューで「**Window**」→「**Preferences...**」の順に選択します。

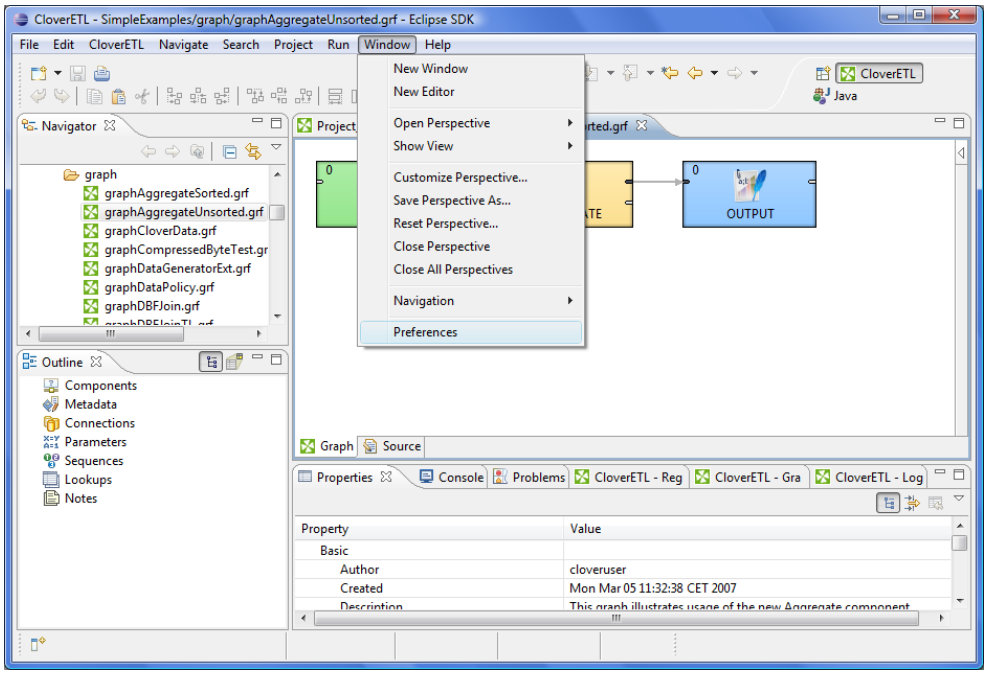


図39.1. コンポーネントの選択

次に、「**CloverETL**」項目を展開して、「**Components in Palette**」を選択します。

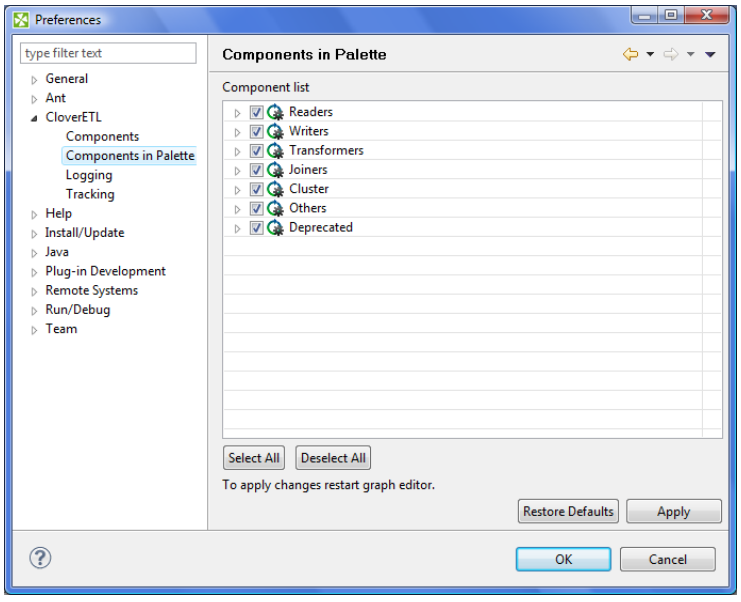


図39.2. パレット内のコンポーネント

このウィンドウに、コンポーネントのカテゴリが表示されます。必要なカテゴリを展開して、パレットから除外するコンポーネントの選択を解除します。

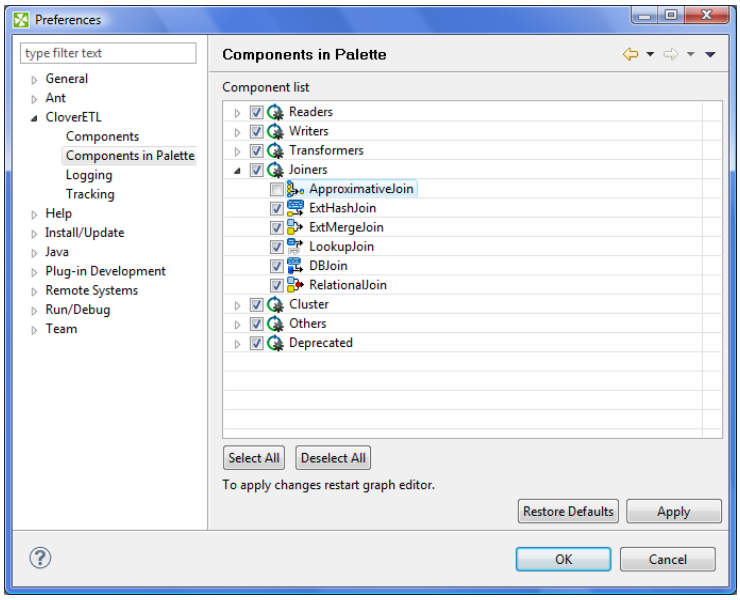


図39.3. パレットからコンポーネントを除外する

次に、グラフを閉じてから開くだけで、コンポーネントが**パレット**から除外されます。

第40章 コンポーネントの検索/追加

パレット(p.262)に加え、コンポーネントの検索または追加を行うための高度なダイアログを使用できます。

コンポーネントの検索

複雑なグラフを構築し、コンポーネントを簡単に見つけることができない場合は、**[Ctrl]キーを押しながら[O]**キーを押します。これで、**コンポーネントを検索**するダイアログが開きます。

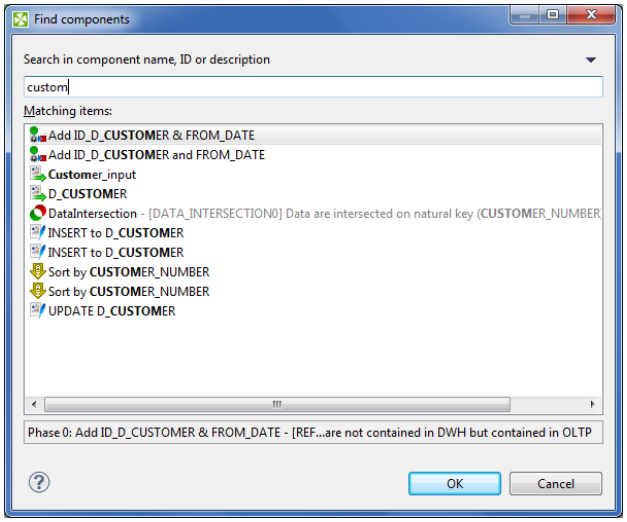


図40.1.「Find Components」ダイアログ: 検索対象テキストは、コンポーネント名および説明の両方で強調表示されます。

テキストを入力すると、次の項目に基づいてコンポーネントが検索されます。

- 名前: たとえば、**UniversalDataReader**を「read customers from text file」に名前を変更した場合、「customers」、「text file」などを入力してこのコンポーネントを検索できます。
- 説明: デフォルトの説明および自身でコンポーネントに追加したカスタムの説明も検索されます。

1. 検索結果のコンポーネントをクリックします。
2. **[Enter]**を押します。
3. コンポーネントが選択され、グラフ・レイアウト内でフォーカスされます。

コンポーネントの追加

パレットに移動せずに素早くコンポーネントを追加する必要がある場合、または使用する必要があるコンポーネントがわからない場合は、**[Shift]キーを押しながら[Space]**キーを押します。これにより、**コンポーネントを追加**するダイアログが表示されます。

高度な機能により、ここでもコンポーネントはその名前および説明(p.264)で検索されます。

例40.1. ソート・コンポーネントの検索

データをソートする必要がありますが、CloverETLでは非常に多くのソート・コンポーネントが提供されています。**[Shift]キーを押しながら[Space]**キーを押すのが、この場合の簡単な解決策です。使用可能なソーターが説明付きで表示されます。

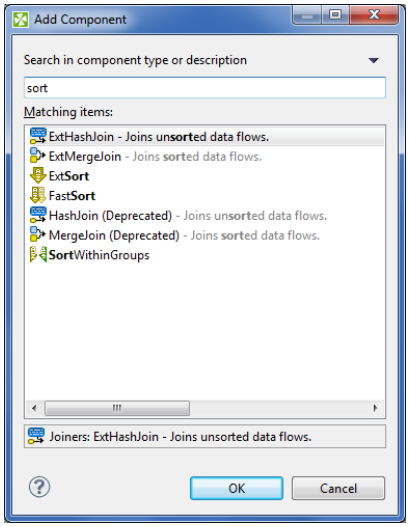


図40.2.「Add Components」ダイアログ - ソーターの検索

注意

最近検索または追加したコンポーネントはダイアログの最上部に表示されるため、簡単に見つけることができます。

第41章 すべてのコンポーネントの共通プロパティ

一部のプロパティはすべてのコンポーネントに共通です。これらについて次に示します。

- **コンポーネント・パレット**に表示するコンポーネントおよびそこから除外するコンポーネントは随時選択できます(第39章「コンポーネント・パレット」(p.262))。
- 各コンポーネントは、**「Edit Component」ダイアログ**(「Edit Component」ダイアログ(p.267))を使用して設定できます。

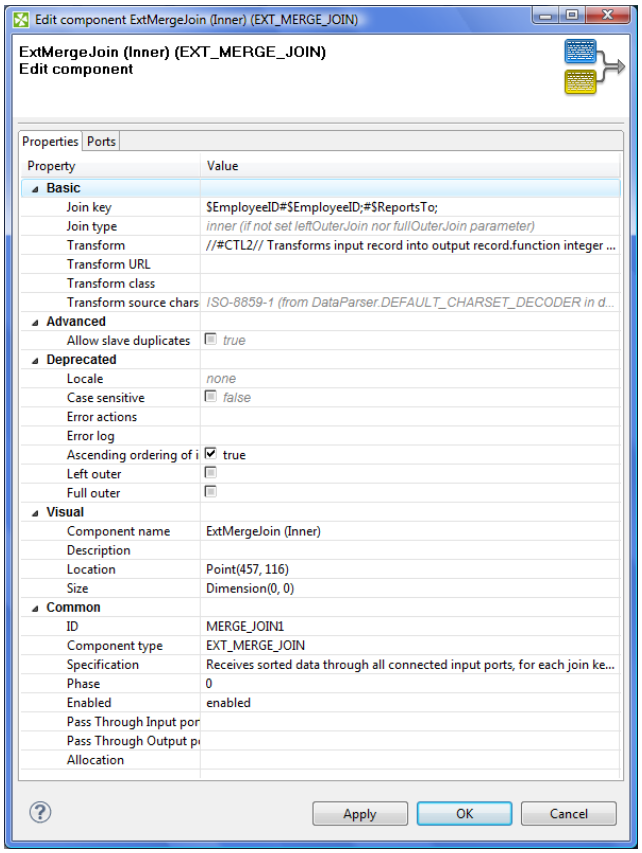


図41.1.「Edit Component」ダイアログ(「Properties」タブ)

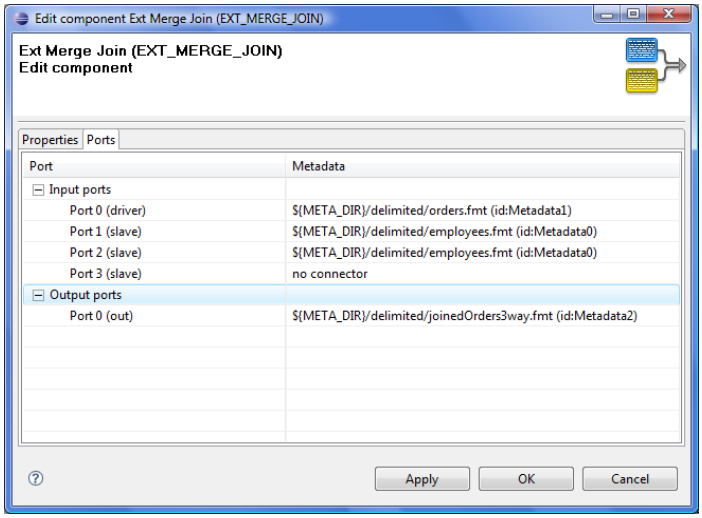


図41.2.「Edit Component」ダイアログ(「Ports」タブ)

この「**Edit Component**」**ダイアログ**で設定可能なプロパティのうち、次のものについて詳細に説明します。

- 各コンポーネントには**コンポーネント名**のラベルが付与されます(コンポーネント名(p.270))。
- 各グラフはフェーズで処理できます(フェーズ(p.271))。
- コンポーネントは無効化できます(コンポーネントの有効化/無効化(p.272))。
- コンポーネントは**パススルー・モード**に切り替え可能です(パススルー・モード(p.273))。
- コンポーネントをどのクラスタ・ノードで実行するかを指定できます(コンポーネントの割当て(p.273))。

「Edit Component」ダイアログ

「**Edit component**」ダイアログでは、各コンポーネントのコンポーネント属性を編集できます。このダイアログには、**グラフ・エディタ**・ペインに貼り付けられたコンポーネントをダブルクリックすることでアクセスできます。

このダイアログは、「**Properties**」タブと「**Ports**」タブの2つのタブで構成されています。

- 「**Properties**」タブには、設定可能なコンポーネント属性の概要が示されます。
- 「**Ports**」タブには、入力および出力の両ポートの概要および各ポートのメタデータが示されます。

「Properties」タブ

「**Properties**」ダイアログでは、コンポーネントのすべての属性が、**Basic**、**Advanced**、**Deprecated**、**Visual**および**Common**の5つのカテゴリに分類されています。

最後の2つのグループ(**Visual**および**Common**)のみが、すべてのコンポーネントに対して設定できます。

その他のグループ(**Basic**、**Advanced**および**Deprecated**)は、コンポーネントにより異なります。

ただし、ほとんどのコンポーネント、または少なくともコンポーネントの一部のカテゴリ(**リーダー**、**ライター**、**トランスフォーマ**、**ジョイナ**または**その他**)でこれらの一部は共通となる可能性があります。

- **Basic**: これらはコンポーネントの基本属性です。コンポーネントは、これらにより区別されます。必須またはオプションのいずれかとなります。

  個別のコンポーネントごと、コンポーネントのカテゴリごと、または大部分のコンポーネントで、固有となる場合があります。

  - **必須**: 必須の属性には警告マークが付けられています。これらの一部は2通り以上の方法で表現でき、2つ以上の属性が同じ目的のために機能できます。
  - **オプション**: これらは、警告マークなしで表示されます。

- **Advanced**: これらの属性では、コンポーネントのより複雑(高度)な設定を行います。これらは、コンポーネントにより異なります。

  個別のコンポーネントごと、コンポーネントのカテゴリごと、または大部分のコンポーネントで、固有となる場合があります。

- **Deprecated**: これらの属性は**CloverETL Designer**の以前のリリースで使用されていましたが、現在も残っているため使用可能です。ただし、必要でないかぎりこれらを使用しないことをお薦めします。

  個別のコンポーネントごと、コンポーネントのカテゴリごと、または大部分のコンポーネントで、固有となる場合があります。

- **Visual**: これらはグラフに表示される属性です。

  これらの属性はすべてのコンポーネントに共通です。

- **Component name**: Component nameは、各コンポーネントに表示されるラベルです。これは、コンポーネントの機能を示す必要があります。「**Edit component**」ダイアログで設定できますが、コンポーネントをダブルクリックしてデフォルトのコンポーネント名を置き換えることも可能です。

  詳細は、コンポーネント名(p.270)を参照してください。

- **Description**: この属性に、簡単な説明を記述できます。これは、コンポーネントの上にカーソルが置かれたときに、ヒントとして表示されます。コンポーネントの該当インスタンスが何を行うかを説明できます。

- **Common**:

  これらの属性も、すべてのコンポーネントに共通です。

  - **ID**: IDは、該当コンポーネントを、同じタイプのその他のすべてのコンポーネントから識別します。「**Window**」→「**Preferences**」→「**CloverETL**」の順に選択して「**Generate component ID from its name**」を選択すると、コンポーネントの名前がたとえば「Write employees to XML」である場合、ID「WRITE_EMPLOYEES_TO_XML」が自動的に取得されます。このオプションが選択されていると、コンポーネントの名前を変更するたびにIDが変更されます。

  - **Component type**: これは、コンポーネントのタイプを示します。このコンポーネント・タイプに番号を追加すると、コンポーネントIDを取得できます。この詳細は説明しません。

  - **Specification**: 該当コンポーネント・タイプの機能の説明です。これは変更できません。この詳細は説明しません。

  - **Phase**: これは、コンポーネントが属するフェーズの整数です。同じフェーズ番号のすべてのコンポーネントは並行して実行されます。次に、すべての同じフェーズ番号が順に続きます。各フェーズは、前のフェーズが成功して終了した後に開始され、それ以外の場合はデータの解析が停止します。

    詳細は、フェーズ(p.271)を参照してください。

  - **Enabled**: この属性は、コンポーネントを**有効化**または**無効化**する必要があるか、あるいは**パススルー・モード**で実行する必要があるかを指定するために機能します。これは「**Properties**」タブまたはコンテキスト・メニュー(**パススルー・モード**を除く)でも設定可能です。

    詳細は、コンポーネントの有効化/無効化(p.272)を参照してください。

  - **Pass Through Input port**: コンポーネントが**パススルー・**モードで実行している場合、データ・レコードを受信する入力ポートおよびデータ・レコードを送信する出力ポートを指定する必要があります。この属性では、すべての入力ポートのコンボ・リストから入力ポートを選択できます。

    詳細は、パススルー・モード(p.273)を参照してください。

  - **Pass Through Output port**:コンポーネントが**パススルー・**モードで実行している場合、データ・レコードを受信する入力ポートおよびデータ・レコードを送信する出力ポートを指定する必要があります。この属性では、すべての出力ポートのコンボ・リストから出力ポートを選択できます。

    詳細は、パススルー・モード(p.273)を参照してください。

  - **Allocation**: グラフが**CloverETL Server**のクラスタで実行される場合、この属性をグラフで指定する必要があります。

    詳細は、コンポーネントの割当て(p.273)を参照してください。

「Ports」タブ

このタブでは、コンポーネントのすべてのポートのリストを表示できます。2つの項目(**Input ports**、**Output ports**)のいずれかを展開して、これらの各ポートに割り当てられているメタデータを表示できます。

このタブはすべてのコンポーネントに共通です。

重要

Java形式のUnicode式

CloverETLのバージョン3.0以降、Java形式のUnicode式も**CloverETL**で使用できます(URL属性以外)。

1つ以上のJava形式のUnicode式を`\u0014`のように使用できます。

このような式は、一連の`\uxxxx`コード文字で構成されます。

これらは、次のようにデリミタとしても機能します(前述の、引用符のないCTL式と同様)。

```
\u0014
```

コンポーネント名

各コンポーネントにはラベルが付けられていますが、これは別のものに変更できます。多数のコンポーネントがグラフに存在し、これらのコンポーネントが固有の機能を持つ場合があるため、コンポーネントにはそれぞれの動作に応じた名前を付けることができます。このようにしない場合、グラフ内の多数の異なるコンポーネントに同一の名前が付けられる可能性があります。

次の4つのいずれかの方法で、任意のコンポーネントの名前を変更できます。

- 「**Edit component**」ダイアログで「**Component name**」属性を指定することで、コンポーネントの名前を変更できます。
- 「**Properties**」タブで「**Component name**」属性を指定することで、コンポーネントの名前を変更できます。
- コンポーネントを強調表示してクリックすることにより、コンポーネントの名前を変更できます。

  任意のコンポーネントを強調表示すると(コンポーネント自体をクリックするか「**Outline**」ペインでコンポーネントの項目をクリック)、コンポーネントの名前を示すヒントが表示されます。その後、強調表示されたコンポーネントをクリックすると、コンポーネントの下に四角形が表示され、青の背景色で**コンポーネント名**が示されます。この四角形に示された名前を変更でき、その後は**[Enter]**を押すことのみです。これで**コンポーネント名**が変更され、コンポーネント上に名前が表示されます。

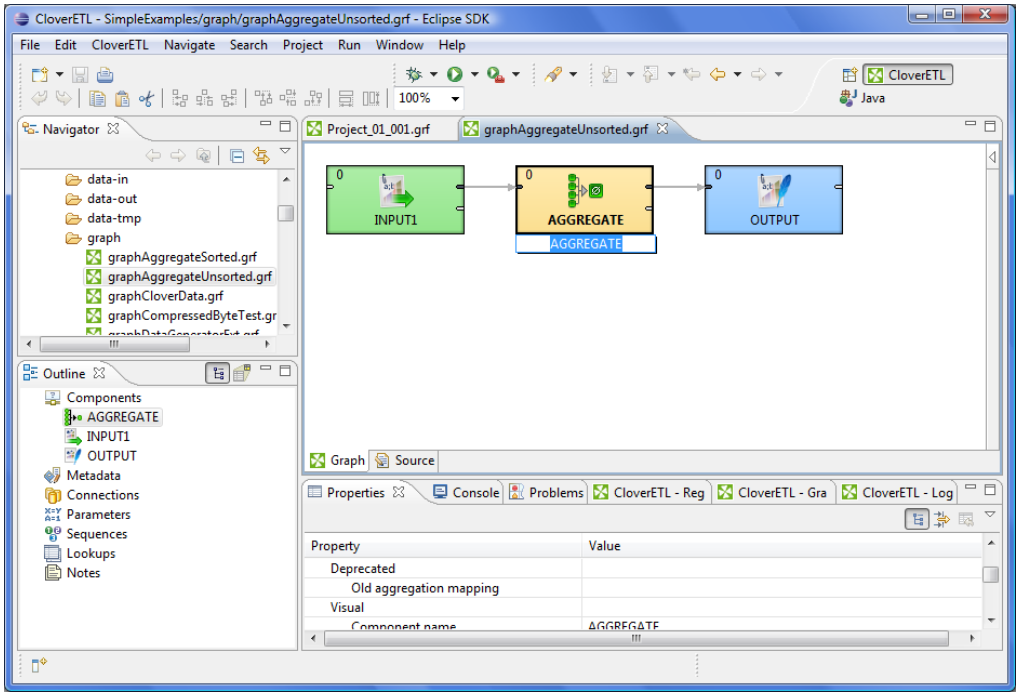


図41.3. 簡単にコンポーネントの名前を変更する方法

- コンポーネントを右クリックして、コンテキスト・メニューから「**Rename**」を選択できます。次に、コンポーネントの下に前述と同じ四角形が表示されます。前述の方法でコンポーネントの名前を変更できます。

フェーズ

各グラフは、コンポーネントにフェーズ番号を設定することでいくつかのフェーズに分割できます。このフェーズ番号は、各コンポーネントの左上隅で確認できます。

フェーズとは、各グラフが同じフェーズ番号内で並行して実行されることを意味しています。つまり、同じフェーズ番号を持つ各コンポーネントおよび各エッジは同時に実行されます。あるフェーズでプロセスが停止した場合、それ以降のフェーズは開始されません。あるフェーズ内のすべてのプロセスが成功して終了した場合のみ、次のフェーズが開始されます。

これが、グラフの実行中に同じフェーズを維持する必要がある理由です。これらを繰り下げることはできません。

したがって、任意のグラフ・コンポーネントのフェーズ番号を繰り上げると、グラフ内の後続の同じフェーズ番号を持つすべてのコンポーネントのフェーズが(より大きいフェーズ番号のコンポーネントとは異なり)新しい値に自動的に変更されます。

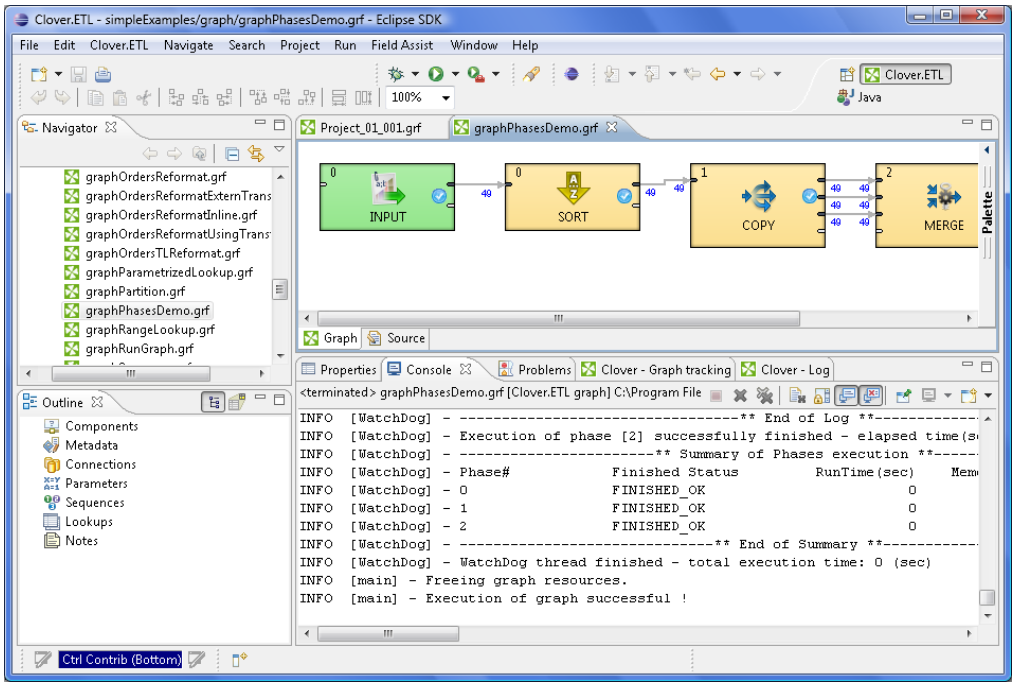


図41.4. 複数のフェーズを含むグラフの実行

複数のコンポーネントを選択して、そのフェーズ番号を設定できます。選択したすべてのコンポーネントに同じフェーズ番号を設定したり、それぞれの個別コンポーネントのフェーズ番号の増分または減分ステップを選択できます。

このことを行うには、次の**フェーズ設定**ウィザードを使用します。

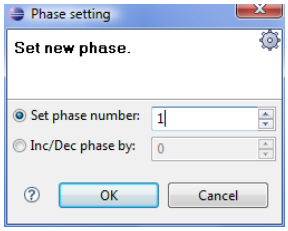


図41.5. 複数のコンポーネントのフェーズ設定

コンポーネントの有効化/無効化

デフォルトでは、すべてのコンポーネントが有効化されています。これらが構成されると、データを解析できます。一方、任意のグラフの任意のコンポーネントのグループをオフにできます。各コンポーネントは無効化できます。一部のコンポーネントを無効化すると、グレー表示となり、プロセスの開始時にデータを解析しなくなります。さらに、グラフの後続のコンポーネントもデータを解析しなくなります。グラフの後続のブランチに接続する有効化されたコンポーネントが別に存在する場合のみ、データは有効化されているコンポーネントを介してブランチに渡されます。一方、無効化されたコンポーネントにデータを渡すコンポーネント、または無効化されたコンポーネントからデータを受信するコンポーネントが、その無効化さたコンポーネントを使用しないとデータを解析できない場合、グラフはエラーで終了します。あるコンポーネントで解析されたデータは別のコンポーネントに送られる必要があり、それが不可能な場合は、解析もできなくなります。無効化は、コンテキスト・メニューまたは「**Properties**」タブで実行できます。コンポーネントが無効化されていても解析が可能な例を次に示します。

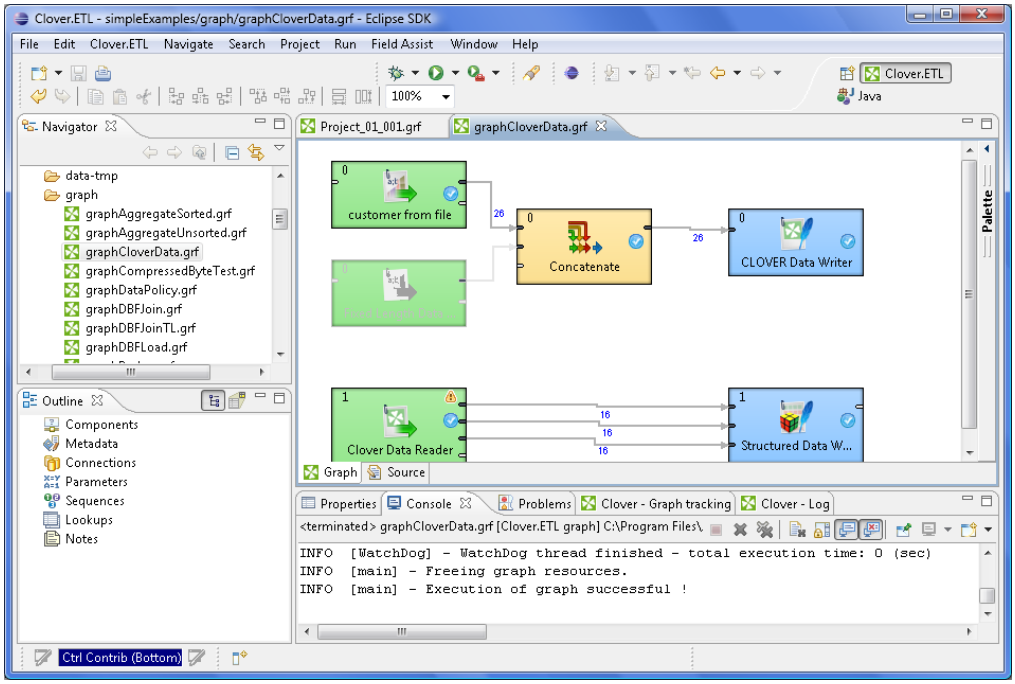


図41.6. 無効化されたコンポーネントを含むグラフの実行

図からわかるとおり、無効化されたコンポーネントからのデータ・レコードは**Concatenate**コンポーネントには必須ではなく、このことから解析は可能です。一方、**Concatenate**コンポーネントを無効化した場合、このコンポーネントより前のリーダーはそのデータ・レコードを自由に送信できるコンポーネントが存在しなくなるため、グラフはエラーで終了します。

パススルー・モード

前の項(コンポーネントの有効化/無効化(p.272))で説明したように、あるコンポーネントをオフにした状態(グラフに存在しない場合と同じ状態)でグラフを処理する必要がある場合、そのコンポーネントを**パススルー・モード**に設定することでこれを実現できます。この場合、データ・レコードは、コンポーネントの入力ポートから出力ポートに直接渡され、コンポーネントはデータを変更しません。このモードはコンテキスト・メニューまたは「**Properties**」タブからも選択できます。

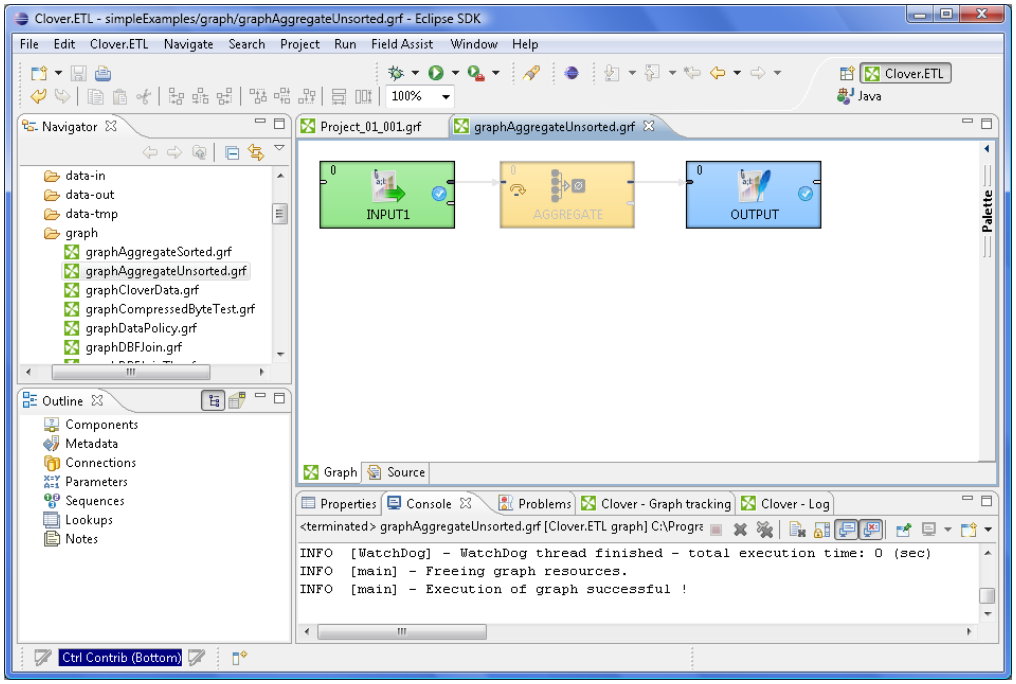


図41.7. パススルー・モードのコンポーネントを含むグラフの実行

注意

複数の入力/出力ポートを持つ一部のコンポーネントでは、データの入/出力に使用するポートをそれぞれ指定する必要があります。これらでは、「Properties」タブ(p.267)で説明したように、**パススルー入力ポート**または**パススルー出力ポート**、またはその両方を設定する必要があります。

コンポーネントの割当て

この属性は、**CloverETL Cluster**環境でのみ考慮されます。

「**Allocation**」属性は、すべてのETLコンポーネントに共通です。この属性はクラスタ・グラフ処理で使用され、実行されるコンポーネントのインスタンス数および実行先のクラスタ・ノードを計画します。割当ては、データ処理のパラレル化およびクラスタ・ノード間のデータ・ルーティングの基本概念です。

割当ては、次の3つの異なる方法で指定できます。

- ワーカー数に基づく割当て: コンポーネントは、**CloverETL Cluster**で優先的に使用されるいくつかのクラスタ・ノード上のリクエストされたインスタンスで実行されます。
- パーティション化されたサンドボックスの参照に基づく割当て: コンポーネントは、パーティション化されたサンドボックスが存在するすべてのクラスタ・ノード上で実行されます。

注意

この割当てタイプは、パーティション化されたサンドボックス内のファイルを参照する大部分のデータ・リーダーおよびデータ・ライターで、デフォルトとして透過的に使用されます。

- クラスタ・ノードIDのリストにより定義される割当て(1つのクラスタ・ノードを複数回使用可能)。

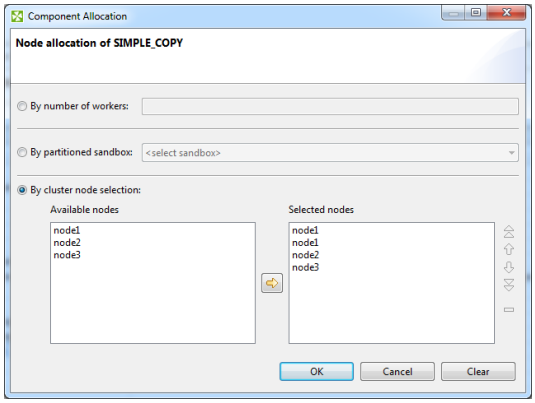


図41.8. コンポーネントの割当てダイアログ

割当ては、隣接コンポーネントから自動的に継承します。したがって、継続的なグラフでは1つのコンポーネントで割当てが設定されていれば十分であり、この割当てがその他のすべてのコンポーネントでも同様に使用されます。クラスタ化されたグラフのすべてのコンポーネントは、コンポーネントが最終的に実行されるインスタンス数(x3など)で修飾され、これは割当てカーディナリティと呼ばれます。この注釈は、グラフの保存操作時に更新されます。近接から派生した割当てカーディナリティは、グレーの斜体フォントで表記され、そのコンポーネントの割当て定義から派生したカーディナリティは正体フォントで表記されます。

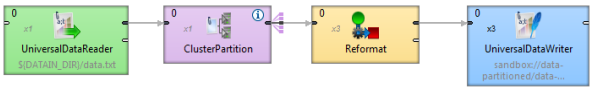


図41.9. 割当てカーディナリティ・デコレータ

2つの相互接続コンポーネントの割当てには互換性が必要であり、指定されたワーカー数は同じである必要があります。この規則の唯一の例外はクラスタ・コンポーネントであり、これはパラレル化レベルの変更専用のコンポーネントです。**クラスタ・パーティショナ**は、単一ワーカー割当てを複数ワーカー割当てに変更します。一方、**クラスタ・ギャザー**は、複数ワーカー割当てを単一ワーカー割当てに変更します。

クラスタ化グラフ処理の詳細は、**CloverETL Cluster**のマニュアルを参照してください。

第42章 多くのコンポーネントの共通プロパティ

ここでは、様々なコンポーネントのグループに共通なプロパティの概要を簡単に示します。

対応する項へのリンクを次に示します。

- コンポーネント内でファイルを指定する必要がある場合、URLファイル・ダイアログ(p.69)を使用する必要があります。
- 一部のコンポーネントでは、固有のメタデータ構造をそのポートで使用します。接続済エッジには、事前定義テンプレートからメタデータを簡単に割り当てることができます。メタデータ・テンプレート(p.275)を参照してください。
- 一部のコンポーネントでは、時間間隔(通常は遅延またはタイムアウト)を構成できます。時間間隔の指定の構文の概要は、時間間隔(p.275)を参照してください。
- 一部のコンポーネントでは、グループ・キーの値を基準にレコードをグループ化する必要があります。このキーでは、キー・フィールドの順序およびソート順のいずれも重要ではありません。グループ・キー(p.276)を参照してください。
- 一部のコンポーネントでは、ソート・キーの値を基準にレコードをグループ化してソートする必要があります。このキーでは、キー・フィールドの順序およびソート順の両方が重要となります。ソート・キー(p.277)を参照してください。
- 異なるコンポーネント・グループからの多くのコンポーネントでは、変換の定義が可能かまたは変換の定義が必要です。変換の定義(p.279)を参照してください。

メタデータ・テンプレート

一部のコンポーネントでは、そのポートのメタデータが固有の構造を持つ必要があります。例として、UniversalDataReaderのエラー・メタデータ(p.416)を参照してください。第58章「ファイル操作」(p.734)などのその他のコンポーネントの一部では、メタデータ構造は必須ではありませんが、使用することをお薦めします。両方の場合で、事前定義のメタデータ・テンプレートを活用できます。

推奨される構造の新しいメタデータを作成するには、テンプレートが定義されているポートに接続されているエッジを右クリックして、コンテキスト・メニューから「**New metadata from template**」を選択し、サブメニューからテンプレートを選択します。

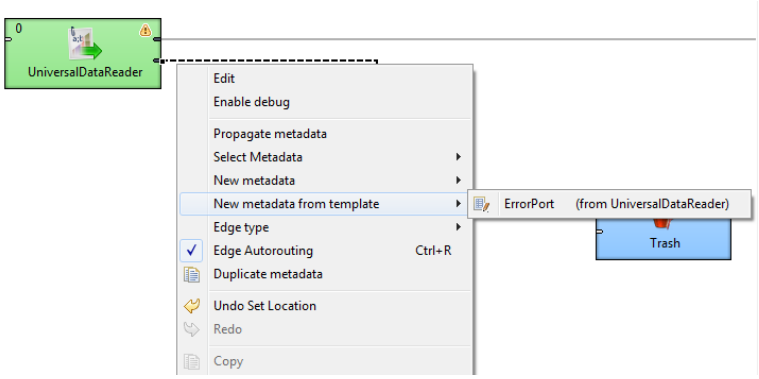


図42.1. テンプレートからのメタデータの作成

時間間隔

時間間隔を指定するときに、次の時間単位を使用できます。

`w` 週(7日)
`d` 日(24時間)
`h` 時間(60分)
`m` 分(60秒)
`s` 秒(1000ミリ秒)
`ms` ミリ秒

単位は任意の組合せが可能ですが、その順序は最大のものから最小のものとなる必要があります。

例42.1. 時間間隔の指定

`1w 2d 5h 30m 5s 100ms` = 797405100ミリ秒

`1h 30m` = 5400000ミリ秒

`120s` = 120000ミリ秒

時間単位が指定されていない場合、数値は、コンポーネント固有(通常はミリ秒)のデフォルトの単位で示されているとみなされます。

グループ・キー

グループ・キーを作成するフィールドの選択が必要となる場合があります。このことは、「**Edit key**」ダイアログで実行できます。ダイアログを開いた後に、グループ・キーを作成するフィールドを選択する必要があります。

必要なフィールドを選択して、選択した各キー・フィールドを右の「**Key parts**」ペインにドラッグ・アンド・ドロップします。(矢印ボタンも使用できます。)

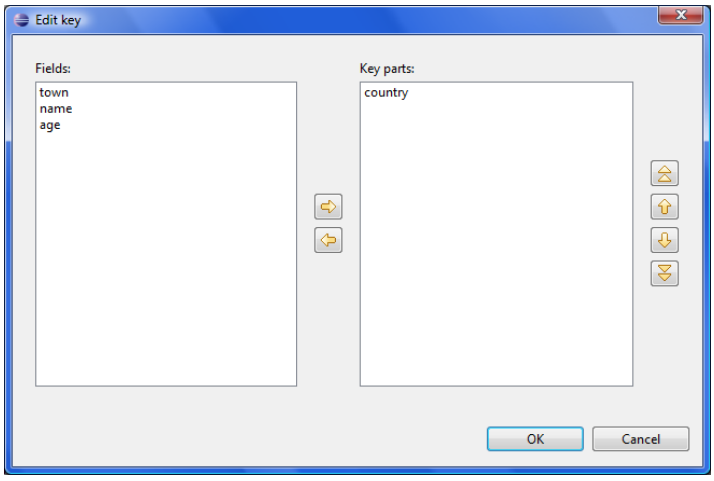


図42.2. グループ・キーの定義

フィールドの選択後に「**OK**」ボタンをクリックすると、選択したフィールドの名前が、セミコロンで区切られた同じフィールド名のシーケンスとなります。このシーケンスは、対応する属性行に表示されます。

結果のグループ・キーは、セミコロンで区切られたフィールド名のシーケンスです。これは、次のように表示されます。`FieldM;...FieldN`.

この種類のキーでは、**ソート・キー**とは異なりソート順は表示されません。デフォルトでは、順序はすべてのフィールドで昇順となり、これらのフィールドの優先度はダイアログ・ペインで上位から下げられ、属性行では左から右に移動します。詳細は、ソート・キー(p.277)を参照してください。

キーが定義されてコンポーネントで使用されると、入力レコードは等しいキー値を持つレコードのグループにまとめられます。

グループ・キーは次のコンポーネントで使用されます。

- SortWithinGroups (p.649)の**Group key**
- Merge (p.607)の**Merge key**
- Partition (p.619)およびClusterPartition (p.751)の**Partition key**
- Aggregate (p.578)の**Aggregate key**
- Denormalizer (p.589)の**Key**
- Rollup (p.635)の**Group key**
- ApproximativeJoin (p.654)の**Matching key**
- 異なる出力ポート(**クラスタ・パーティション**の場合はクラスタ・ノード)にデータ・レコードを分配する機能を持つ**パーティション・キー**も、このタイプです。異なる出力ファイルへの出力のパーティション(p.318)を参照してください。

ソート・キー

一部のコンポーネントでは、ソート・キーを定義する必要があります。グループ・キーと同様に、このソート・キーは「**Edit key**」ダイアログを使用してキー・フィールドを選択することで定義できます。ここでは、選択された各フィールドで使用されるソート順を定義することも可能です。

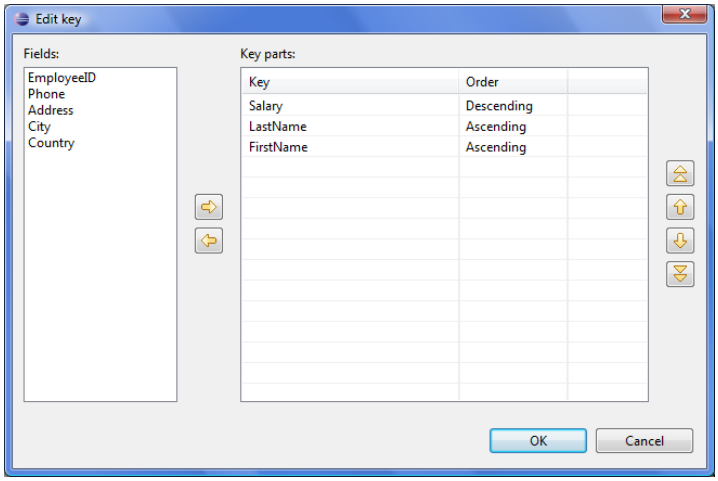


図42.3. ソート・キーおよびソート順の定義

「**Edit key**」ダイアログで必要なフィールドを選択して、選択した各キー・フィールドを右の「**Key parts**」ペインの「**Key**」列にドラッグ・アンド・ドロップします。(**矢印**ボタンも使用できます。)

グループ・キーの場合とは異なり、すべてのソート・キーではフィールドが選択される順序が重要となります。

各ソート・キーで、最上部のフィールドは、ソート優先度が最も高くなります。ソート優先度は上から下に低くなります。最下部のフィールドは、優先度が最も低くなります。

「**OK**」ボタンをクリックすると、選択されたフィールドは、セミコロンで区切られた同じフィールド名およびカッコ内の文字aまたはd (それぞれ昇順 (`ascending`) または降順 (`descending`) の意味) のシーケンスとなります。

これは、次のように表示されます。`FieldM(a);...FieldN(d).`

このシーケンスは、対応する属性行に表示されます。(最も高いソート優先度がシーケンスの最初のフィールドとなります。優先度は、シーケンスの終わりに向かって低くなります。)

図からわかるとおり、この種類のキーは、ソート順は各キー・フィールドとは別に表現されます(**昇順**または**降順**)。デフォルトのソート順は**昇順**です。デフォルトのソート順は、「**Key parts**」ペインの「**Order**」列で変更することもできます。

重要

ASCII順とアルファベット順

文字列データ・フィールドはASCII順(0、1、11、12、2、21、22 ...A、B、C、D ... a、b、c、d、e、f ...)でソートされるのに対し、その他のフィールドはアルファベット順(0、1、2、11、12、21、22 ...A、a、B、b、C、c、D、d ...)でソートされます。

例42.2. ソート

ソート・キーが`Salary(d);LastName(a);FirstName(a)`である場合について考えます。レコードは、`Salary`値の降順でソートされ、次に、レコードは同じ`Salary`値内で`LastName`に基づいてソートされ、次に同じ`LastName`と同じ`Salary`値内で`FirstName`に基づいてソートされます(いずれも昇順)。

したがって、`Salary`が`25000`のすべての人物は、給与が`28000`のその他のすべての人物の後に処理されます。さらに、同じ`Salary`では、すべての`Brown`が、すべての`Smith`より先に処理されます。さらに、同じ給与では、すべての`John Smith`がすべての`Peter Smith`より先に処理されます。最高の優先度は`Salary`であり、最低の優先度は`FirstName`です。

ソート・キーは次の場合に使用されます。

- ExtSort (p.600)の**Sort key**
- FastSort (p.602)の**Sort key**
- SortWithinGroups (p.649)の**Sort key**
- Dedup (p.587)の**Dedup key**
- SequenceChecker (p.799)の**Sort key**

変換の定義

変換の基本情報は、第34章「変換」(p.238)を参照してください。

ここでは、一部のコンポーネントを通過するデータを変更する変換の作成方法について説明します。

変換の概略を示した表は、変換の概要(p.282)を参照してください。

以降では次について説明します。

1. 変換が可能なコンポーネント

   変換が可能なコンポーネント(p.279)

2. 変換の記述に使用可能な言語

   JavaまたはCTL(p.280)

3. 定義を内部で行うか外部で行うかの判断

   内部または外部の定義(p.280)

4. 変換の戻り値

   変換の戻り値(p.283)

5. エラー発生時の対処方法

   「Error Actions」および「Error Log」(3.0以降非推奨)(p.285)

6. 変換エディタとその操作方法

   変換エディタ(p.286)

7. 変換が可能な多くのコンポーネントの共通インタフェース

   共通Javaインタフェース(p.295)

変換が可能なコンポーネント

変換は、次のコンポーネントで定義できます。

- **DataGenerator**、**Reformat**および**Rollup**

  これらのコンポーネントでは変換は必須です。

  変換は、JavaまたはClover Transformation Languageで定義できます。

  これらのコンポーネントでは、変換の戻り値を使用して、異なるデータ・レコードを異なる出力ポートに送信できます。

  異なるレコードを異なる出力ポートに送信するには、レコードから対応する出力ポートへのマッピングを作成することと、対応する整数値を返すことの両方が必要です。

- **Partition**または**ClusterPartition**

  Partitionまたは**ClusterPartition**コンポーネントでは、変換はオプションです。これは、「**Ranges**」と「**Partition key**」のいずれの属性も定義されていない場合のみ必要です。

変換は、JavaまたはClover Transformation Languageで定義できます。

Partitionでは、変換の戻り値を使用して、異なるデータ・レコードを異なる出力ポートに送信できます。

ClusterPartitionでも、変換の戻り値を使用して、異なるデータ・レコードを異なるクラスタ・ノードに送信できます(仮想出力ポートを経由)。

異なるレコードを異なる出力ポートまたはクラスタ・ノードに送信するには、対応する整数値を返す必要があります。ただし、すべてのレコードは自動的に送信されるため、このコンポーネントでマッピングを記述する必要はありません。

- **DataIntersection**、**Denormalizer**、**Normalizer**、**Pivot**、**ApproximativeJoin**、**ExtHashJoin**、**ExtMergeJoin**、**LookupJoin**、**DBJoin**および**RelationalJoin**

  これらのコンポーネントでは変換は必須です。

  変換は、JavaまたはClover Transformation Languageで定義できます。

  Pivotでは、変換はキーまたはグループ・サイズのいずれかの属性を設定することで定義できます。これもJavaまたはCTLで記述できます。

- **MultiLevelReader**および**JavaExecute**

  これらのコンポーネントでは変換は必須です。

  これは、Javaでのみ記述できます。

- **JMSReader**および**JMSWriter**

  これらのコンポーネントでは、変換はオプションです。

  定義する場合は、すべてJavaで記述する必要があります。

JavaまたはCTL

変換は、JavaまたはClover Transformation Language (CTL)で記述できます。

- Javaはすべてのコンポーネントで使用できます。

  Javaで実行される変換はCTLで記述された変換よりも高速です。変換は常にJavaで記述できます。

- CTLは、**JMSReader**、**JMSWriter**、**JavaExecute**および**MultiLevelReader**では使用できません。

  ただし、CTLは非常にシンプルなスクリプト言語であり、大部分の変換コンポーネントで使用できます。Javaに関する知識がなくても、CTLを使用できます。CTLでは、Javaの知識は一切必要ありません。

内部または外部の定義

各変換は、内部または外部として定義できます。

- **内部変換**

  変換、**非正規化**などの属性を定義する必要があります。

  このような場合では、コードの一部はグラフに直接記述され、グラフ内で確認できます。

- **外部変換**

次の2種類の属性のいずれかを定義できます。

- JavaとCTL両方の**変換URL**、**非正規化URL**など

  コードは外部ファイルに記述されます。このような外部ファイルのキャラクタ・セットも指定できます(**変換ソース・キャラクタ・セット**、**非正規化ソース・キャラクタ・セット**など)。

  Javaで記述された変換の場合、変換が正常に実行されるように、変換ソース・コードのフォルダをJavaコンパイラのソースとして指定する必要があります。

- **変換クラス、非正規化クラスなど**

  これはコンパイル済のJavaクラスです。

  変換を正常に実行するには、クラスがクラスパス内に存在する必要があります。

概要を次に示します。

- **変換、非正規化など**

  グラフ自体に変換を定義するには、**変換エディタ**(**JMSReader**、**JMSWriter**および**JavaExecute**の各コンポーネントの場合は「**Edit value**」ダイアログ)を使用する必要があります。これらでは、グラフ自体に存在し表示される変換を定義できます。変換の記述に使用可能な言語は、前述の説明を参照してください(JavaまたはCTL)。

  エディタまたはダイアログの詳細は、変換エディタ(p.286)または「Edit Value」ダイアログ(p.70)を参照してください。

- **変換URL、非正規化など**

  グラフ外のソース・ファイルで定義された変換も使用できます。変換ソース・ファイルを特定するには、URLファイル・ダイアログ(p.69)を使用します。示されている各コンポーネントでは、この変換定義を使用できます。このファイルには、JavaまたはCTLで記述された変換の定義が含まれている必要があります。この場合、変換はグラフ外に存在します。

  詳細は、URLファイル・ダイアログ(p.69)を参照してください。

- **変換クラス、非正規化クラスなど**

  すべての変換コンポーネントで、コンパイル済変換クラスを使用できます。このことを行うには、**Open Type**ウィザードを使用します。この場合、変換はグラフ外に存在します。

  詳細は、「Open Type」ダイアログ(p.71)を参照してください。

変換の定義方法の詳細は、対応するコンポーネントに関する項に記載されています。そこでは、CTLテンプレートおよびJavaインタフェースの両方の変換機能(必須およびオプション)について説明されています。

変換が可能なコンポーネントの概要の簡単な表を次に示します。

表42.1. 変換の概要

コンポーネント	変換が必須	Java	CTL	全出力に送信[1]	各出力に送信[2]	CTL テンプレート	Java インタフェース
リーダー							
DataGenerator (p.351)	✔	✔	✔	✖	✔	(p.354)	(p.357)
JMSReader (p.377)	✖	✔	✖	✔	✖	-	(p.379)
MultiLevelReader (p.392)	✔	✔	✖	✖	✔	-	(p.394)
ライター							
JMSWriter (p.501)	✖	✔	✖	-	-	-	(p.503)
トランスフォーマ							
Partition (p.619)	✖	✔	✔	✖	✔	(p.621)	(p.625)
DataIntersection (p.582)	✔	✔	✔	-	-	(p.584)	(p.584)
Reformat (p.632)	✔	✔	✔	✖	✔	(p.633)	(p.634)
Denormalizer (p.589)	✔	✔	✔	-	-	(p.591)	(p.596)
Pivot (p.628)	✔	✔	✔	-	-	(p.631)	(p.631)
Normalizer (p.612)	✔	✔	✔	-	-	(p.613)	(p.618)
MetaPivot (p.609)	✖	✖	✖	-	-	-	-
Rollup (p.635)	✔	✔	✔	✖	✔	(p.637)	(p.645)
DataSampler (p.585)	✔	✖	✖	-	-	-	-
ジョイナ							
ApproximativeJoin(p.654)	✔	✔	✔	-	-	(p.325)	(p.328)
ExtHashJoin(p.667)	✔	✔	✔	-	-	(p.325)	(p.328)
ExtMergeJoin(p.672)	✔	✔	✔	-	-	(p.325)	(p.328)
LookupJoin(p.677)	✔	✔	✔	-	-	(p.325)	(p.328)
DBJoin(p.664)	✔	✔	✔	-	-	(p.325)	(p.328)
RelationalJoin(p.680)	✔	✔	✔	-	-	(p.325)	(p.328)
クラスタ・コンポーネント							
ClusterPartition(p.751)	✖	✔	✔	✖	✔	???	???
その他							
JavaExecute(p.791)	✔	✔	✖	-	-	-	(p.792)

説明

1): 該当する場合、各データ・レコードは常に接続済のすべての出力ポートを介して送信されます。

2): 該当する場合、各データ・レコードは、変換によって返された番号の接続済出力ポートを介して送信されます。詳細は、変換の戻り値(p.283)を参照してください。

変換の戻り値

変換が定義されているコンポーネントでは、戻り値もいくつか定義できます。これらは、1以上、または0以下の整数です。

注意

DBExecuteでは、SQL例外の形式で0未満の整数値を返すこともできます。

- **正またはゼロの戻り値**
  - **ALL = Integer.MAX_VALUE**

    この場合、レコードはすべての出力ポートを介して送信されます。この変数は、使用する前に宣言する必要はありません。CTLでは、ALLは2147483647に等しく、つまり、Integer.MAX_VALUEです。ALLおよび2147483647の両方を使用できます。
  - **OK = 0**

    この場合、レコードは単一の出力ポートまたは出力ポート0 (**Reformat**、**Rollup**など、コンポーネントに複数のポートがある場合)を介して送信されます。この変数は、使用前に宣言する必要はありません。
  - **0以上のその他の整数**

    この場合、レコードはこの戻り値と等しい番号の出力ポートを介して送信されます。これらの値は**マッピング・コード**と呼ばれます。
- **負の戻り値**
  - **SKIP = - 1**

    この値は、エラーが発生したが正しくないレコードはスキップされ処理が続行されることを示すために使用されます。この変数は、使用する前に宣言する必要はありません。SKIPおよび-1の両方を使用できます。

    この戻り値は、「**Error actions**」属性のCONTINUE (**CloverETL**のリリース3.0では非推奨)の設定と同じ意味を持ちます。
  - **STOP = - 2**

    この値は、エラーが発生して処理を停止する必要があることを示すために使用されます。この変数は、使用する前に宣言する必要はありません。STOPおよび-2の両方を使用できます。

    この戻り値は、「**Error actions**」属性のSTOP (**CloverETL**のリリース3.0では非推奨)の設定と同じ意味を持ちます。

重要

CTL1での同じ戻り値はERRORです。CTL2では、STOPを使用できます。

- **-1以下の任意の整数**

  これらの値は、後述するようにユーザーが定義する必要があります。これらは致命的エラーを意味します。これらの値は**エラー・コード**と呼ばれます。これらは、一部のコンポーネントのError actions(p.285)の定義に使用されます(この属性および「**Error log**」は**CloverETL**のリリース3.0以降では非推奨です)。

重要

1. **0以上の値**

   DataGenerator、**Partition**、**Reformat**および**Rollup**の場合のみ、0以上のすべての戻り値で、同じデータ・レコードを指定された出力ポートへ送信することが可能になります。**DataGenerator**、**Reformat**および**Rollup**では、接続済のこのような各出力ポートのマッピングを忘れずに定義してください。**Partition** (および**clusterpartition**)では、マッピングは自動的に行われます。その他のコンポーネントでは、これは意味を持ちません。これらは、一意の出力ポートを持っているか、その出力ポートが明示的な出力用に厳密に定義されているかのいずれかです。一方、**CloverDataReader**、**XLSDataReader**および**DBFDataReader**では、各データ・レコードは常に接続済のすべての出力ポートに送信されます。

2. **-1未満の値**

   CTLテンプレートの対応するオプションの`OnError()`関数を呼び出すときに、これらの戻り値を使用しないでください。いずれかのオプションの`<required function>OnError()`を呼び出すには、たとえば次の関数を使用できます。

   ```
   raiseError(string Arg)
   ```

   これは、このような`<required function>OnError()`、たとえば`transformOnError()`などの呼出しが可能な例外をスローします。任意の`<required function>()`関数によりスローされるその他の例外は、呼出しが定義されている場合は、対応する`<required function>OnError()`を呼び出します。

3. **-2以下の値**

   整数値を返す任意の関数が-2以下の値(`STOP`を含む)を返す場合、`getMessage()`関数が呼び出されます(定義されている場合)。

   したがって、この関数の呼出しを可能にするには、-2以下の値の`return`文を、整数を返す関数に追加する必要があります。たとえば、`transform()`、`append()`または`count()`などのいずれかの関数が-2を返す場合、`getMessage()`が呼び出されてメッセージがコンソールに書き出されます。

重要

グラフが例外により、または-1未満の負の値を返して失敗した場合、出力ファイルにはレコードは書き込まれません。

以前に処理されたレコードが出力に書き込まれるようにするには、SKIP (-1)を返す必要があります。この方法で、このようなレコードはスキップされ、グラフは失敗せずに少なくとも一部のレコードは出力に書き込まれます。

「Error Actions」および「Error Log」(3.0以降非推奨)

重要

リリース3.0以降の**CloverETL**では、これらの属性は非推奨です。処理を続行または停止する必要がある場合は、これらを`SKIP`または`STOP`戻り値に置き換える必要があります。

一部のコンポーネントでは、**エラー・コード**を使用して次の2つの属性を定義できます。

- **Error actions**

  値はすべて致命的エラーが発生したことを意味し、処理を停止する必要があるか続行する必要があるかをユーザーが判断します。レコードへの対処を定義するには、「**Error actions**」属性の行をクリックして、表示されたボタンをクリックし、次のダイアログでアクションを指定します。**プラス記号**のボタンをクリックすると、このダイアログ・ペインに行が追加されます。「**Error action**」列でSTOPまたはCONTINUEを選択します。「**Error code**」列に整数を入力します。左列で`MIN_INT`値をそのまま変更しない場合は、このダイアログで明示的に指定されていないその他のすべての整数値に対してアクションが適用されることを意味します。

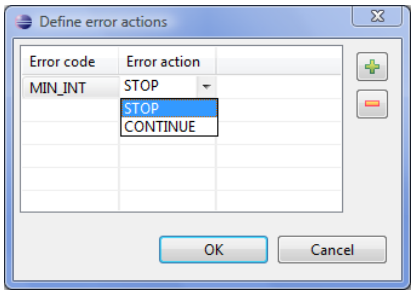


図42.4. 「Define Error Actions」ダイアログ

「**Error actions**」属性は、割当て(`errorCode=someAction`)が相互にセミコロンで区切られたシーケンス形式となります。

- 左側は、`MIN_INT`または変換の定義で戻り値として指定された0未満の任意の整数にすることが可能です。

  `errorCode`が`MIN_INT`である場合、これはシーケンスで指定されていないすべての値に対して、指定されたアクションが実行されることを意味します。

- 右側の割当ては、`STOP`または`CONTINUE`とすることが可能です。

  `someAction`が`STOP`である場合、対応する`errorCode`が返されると、`TransformExceptions`がスローされてグラフが停止します。

  `someAction`が`CONTINUE`である場合、対応する`errorCode`が返されるとエラー・メッセージが**コンソール**または「**Error log**」属性で指定されたファイルに書き出され、グラフは次のレコードの処理を続行します。

例42.3. 「Error actions」属性の例

`-1=CONTINUE;-3=CONTINUE;MIN_INT=STOP`: この場合、変換が`-1`または`-3`を返すと処理は続行し、その他の負の値(`-2`を含む)を返すと処理は停止します。

- **Error log**

  この属性では、エラー・メッセージが**コンソール**または指定されたファイルに書き出されるかどうかを指定できます。ファイルは、URLファイル・ダイアログ(p.69)を使用して定義する必要があります。

変換エディタ

一部のコンポーネントでは**変換エディタ**が提供され、ここで変換を定義できます。

変換エディタを開くと、「**Transformations**」、「**Source**」および「**Regex tester**」のタブが表示されます。

Transformations

「**Transformations**」タブは次のように表示されます。

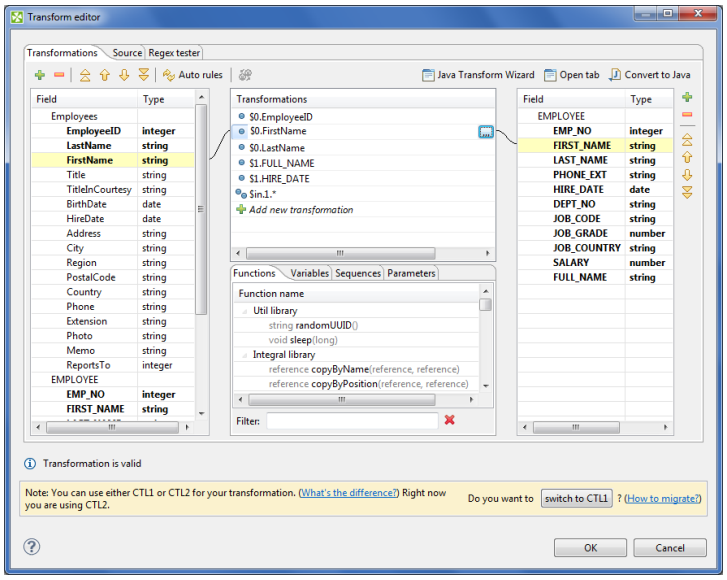


図42.5. 変換エディタの「Transformations」タブ

「**Transformations**」タブでは、入力から出力への簡単なマッピングを使用して変換を定義できます。初めに、入力と出力両方のメタデータを定義して割り当てる必要があります。この後にのみ、必要なマッピングを定義できます。

変換エディタを開くと、その中にいくつかのペインとタブが表示されます。すべての入力ポートの入力フィールドおよびそのデータ型が左ペインに表示されます。すべての出力ポートの出力フィールドおよびそのデータ型が右ペインに表示されます。中央の下部領域には、「**Functions**」、「**Variables**」、「**Sequences**」、「**Parameters**」の各タブが表示されます。

マッピングを定義する場合、入力フィールドをいくつか選択して、その上で左マウス・ボタンを押し、ボタンを押し続けながら中央の「**Transformations**」ペインにドラッグして、ボタンを放します。その後、選択されたフィールド名が「**Transformations**」ペインに表示されます。ここで定義された変換は、左マウス・クリックのドラッグ・アンド・ドロップまたは左上隅のツールバーのボタンで調節できます。

結果として、式の形式は`$portnumber.fieldname`となります。

この後に、別のいくつかの入力フィールドで同じ操作を行うことができます。複数のフィールド(**ジョイナ**および**DataIntersection**コンポーネントの場合、異なる入力ポートでも可)の値を連結する場合、選択したすべてのフィールドを「**Transformations**」ペインの同じ行に移動することが可能で、その後に`$portnumber1.fieldnameA+$portnumber2.fieldnameB`のような式が表示されます。

ポート番号は同じものにも異なるものにもできます。`portnumber1`および`portnumber2`は、**0**または**1**、またはその他の任意の整数とすることができます。(すべてのコンポーネントで、入力ポートおよび出力ポートのいずれも**0**から始まる番号が付けられます。)このようにして、変換の定義の一部を完了しました。残りの必要な作業は、これらの式を出力フィールドに割り当てることだけです。

これらの式を出力に割り当てるには、中央の「**Transformations**」ペインで任意の項目を選択して、その上でマウスの左ボタンを押して、ボタンを押したまま右ペインの必要なフィールドにドラッグして、ボタンを放します。

多数のフィールドのメタデータにフィルタを使用して、使用するフィールドを簡単に見つけることができます。右ペインの出力フィールドは太字で表示されます。

ヒント

左側のペインから右側のペインにフィールドをドラッグするだけで、簡単に変換を設計できます。中央ペインの変換スタブは、自動的に作成されます。フィールドをドロップするときには注意が必要です。これを出力フィールドの上にドロップすると、マッピングが作成されます。これを右側ペインの空白スペース(2つのフィールド間)にドロップすると、入力のメタデータのみが出力にコピーされます。メタデータのコピーは、単一のポート内のみで動作する機能です。

また、これらの各式の左(「**Transformations**」ペイン内)には空白の円が表示されます。マッピングが作成されるたびに、対応する円が青で塗られます。この方法で、「**Transformations**」ペイン内のすべての式が青色になるまで、「**Transformations**」ペイン内のすべての式を出力フィールドにマッピングします。これで、変換の定義は完了します。

左ペインの入力項目を右クリックして「**Copy fields to...**」を選択し、出力メタデータの名前を選択しても、任意の入力フィールドを出力にコピーできます。

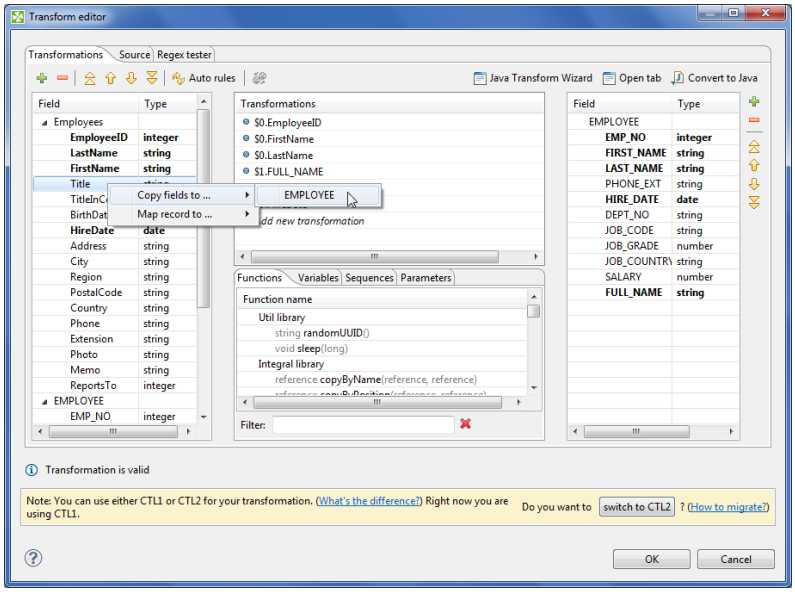


図42.6. 入力フィールドの出力へのコピー

変換を定義する前に出力メタデータを定義していない場合、右クリックを使用してコピーと名前変更を行うことで、ここでもこれらのパラメータを定義できます。ただし、変換の定義前に新しいメタデータを定義しておく方がはるかに簡単です。この**変換エディタ**を使用して出力メタデータを定義した場合、出力レコードが未知であることが通知され、さらに、このエラーで変換を確認して、(その後に)メタデータ・エディタでデリミタを指定する必要があることが通知される場合があります。

注意

出力メタデータのフィールドは、左マウス・ボタンを使用してドラッグ・アンド・ドロップ操作で簡単に並べ替えることができます。

簡単なマッピングの結果は、次のように表示されます。

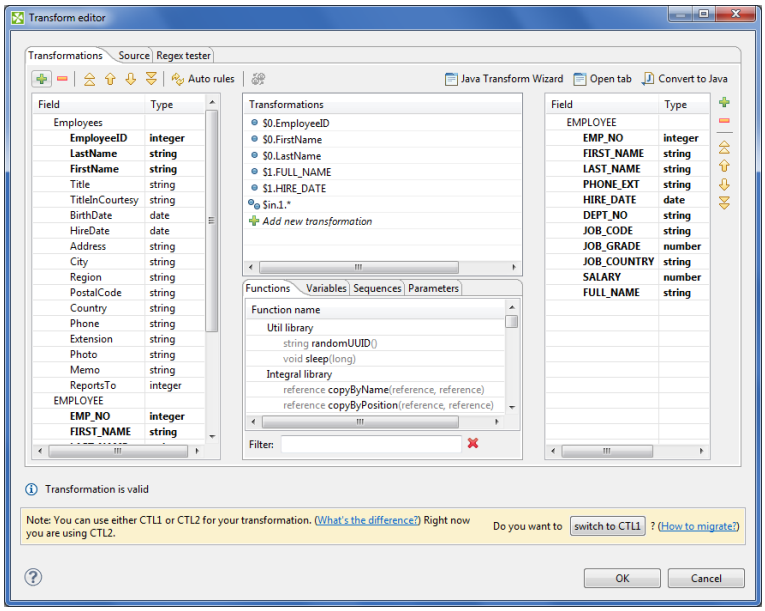


図42.7. CTLでの変換定義(「Transformations」タブ)

左、中央または右ペインで任意の項目を選択すると、対応する項目が線で接続されます。次の例を参照してください。

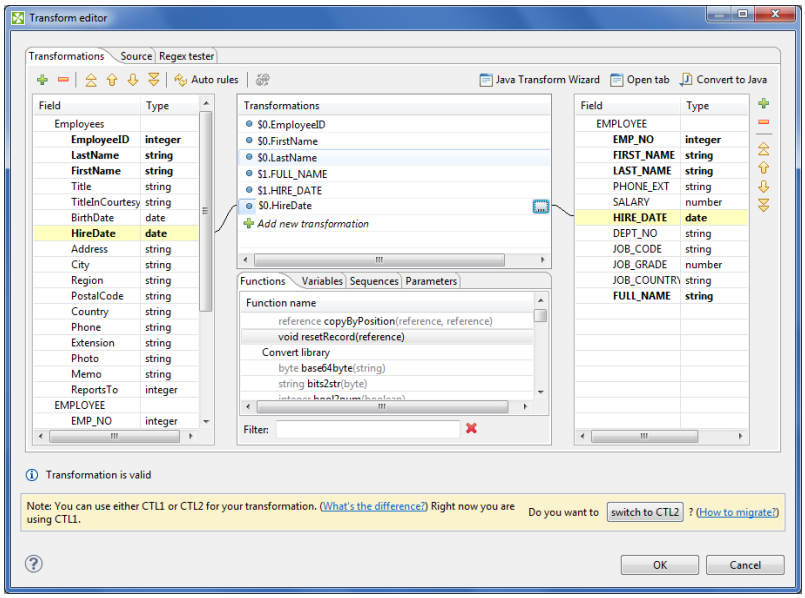


図42.8. 入力から出力へのマッピング(接続線)

必要な変換は、次のように記述することが可能です。

- 「**Transformations**」ペインの各行に記述します。必要な任意の関数を下の「**Functions**」タブからドラッグして(**Variables**、**Sequences** または**Parameters**も同様)、これらをこのペインにドロップすることも可能です。**フィルタ**を使用して、探している関数に素早くジャンプできます。

- 「**Transformations**」ペイン内の行を選択した後に表示される「...」ボタンをクリックします。変換を定義するエディタが開きます。これにはフィールドのリスト、関数および演算子が含まれ、ヒントも提供されます。次を参照してください。

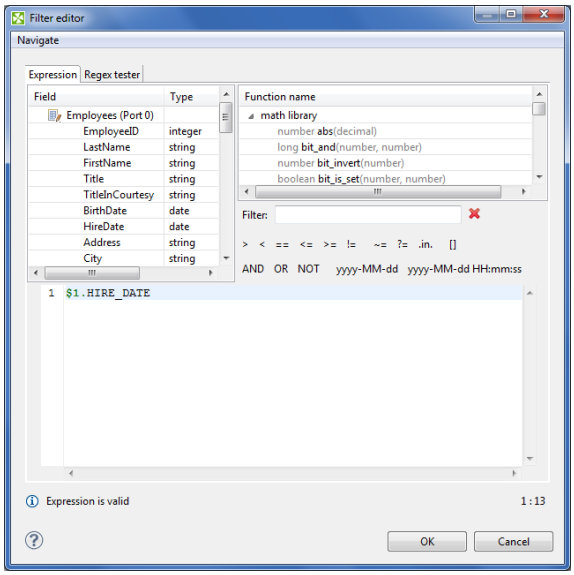


図42.9. フィールドおよび関数を使用するエディタ

変換エディタでは、マッピングでワイルドカードがサポートされます。任意のレコードまたはいずれかのフィールドを右クリックして「**Map record to**」をクリックしてレコードを選択すると、`$out.0.* = $in.1.*;`のような変換が作成されます(「**Source**」タブに表示されます)。これは、レコード番号0のすべての出力フィールドが、レコード番号1のすべての入力フィールドにマッピングされていることを意味しています。「**Transformations**」では、ワイルドカード・マッピングは次のように表示されます。

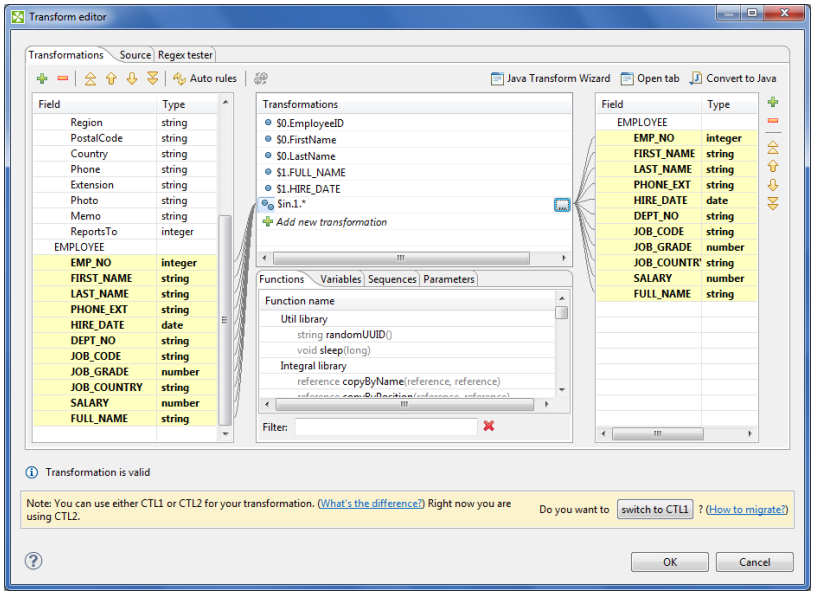


図42.10. ワイルドカードを使用して出力レコードにマッピングされた入力レコード

Source

変換は、非常に複雑となり「**Transformations**」タブでは定義できない場合があります。かわりに「Source」タブを使用できます。

(個別のコンポーネントの「**Source**」タブは、これらのコンポーネントを説明している対応する項で示されています。)

次に、上で定義した変換の「Source」タブを示します。これは、Clover Transformation Language (第66章 「CTL2」(p.889))で記述されています。

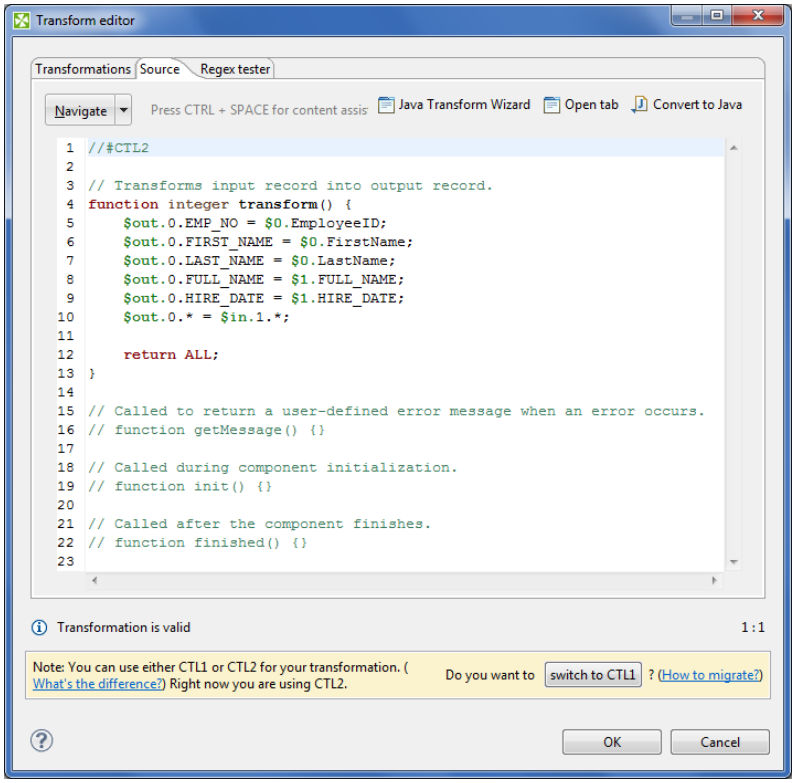


図42.11. CTLでの変換定義(「Source」タブ)

どちらのタブの右上隅にも3つのボタンが表示され、それぞれ新しいJava変換クラスを作成するウィザードの起動(「**Java Transform Wizard**」ボタン)、**グラフ・エディタ**での新規タブの作成(「**Open tab**」ボタン)および定義済変換のJavaへの変換(「**Convert to Java**」ボタン)を行います。

新しいJava変換クラスを作成する場合、「**Java Transform Wizard**」ボタンを押します。次のダイアログが表示されます。

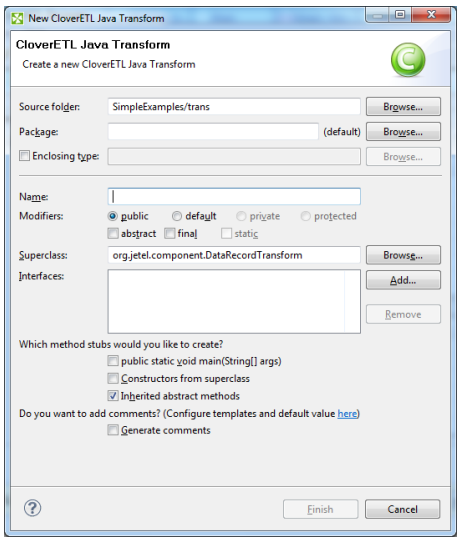


図42.12. Java変換ウィザードのダイアログ

「**Source folder**」フィールドは、プロジェクト`${TRANS_DIR}`にマッピングされます(例: `SimpleExamples/trans`)。「**Superclass**」フィールドの値は、ターゲット・コンポーネントにより異なります。これは、必要なインタフェースを実装している適切な抽象クラスに設定されます。詳細は、変換の概要(p.282)を

参照してください。新しい変換クラスは、クラスの**名前**を入力して、オプションで**パッケージ**を含めて「**Finish**」ボタンを押すことで作成できます。新しく作成されたクラスは**Sourceフォルダ**内に配置されます。

変換エディタの右上隅の2番目のボタン「**Open tab**」ボタンをクリックすると、変換のCTLソース・コードを含む新しいタブが**グラフ・エディタ**で開きます。このことは、次のメッセージで確認されます。

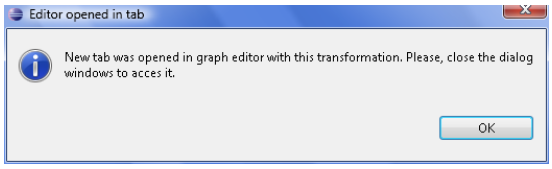


図42.13. 確認メッセージ

タブは次のように表示されます。

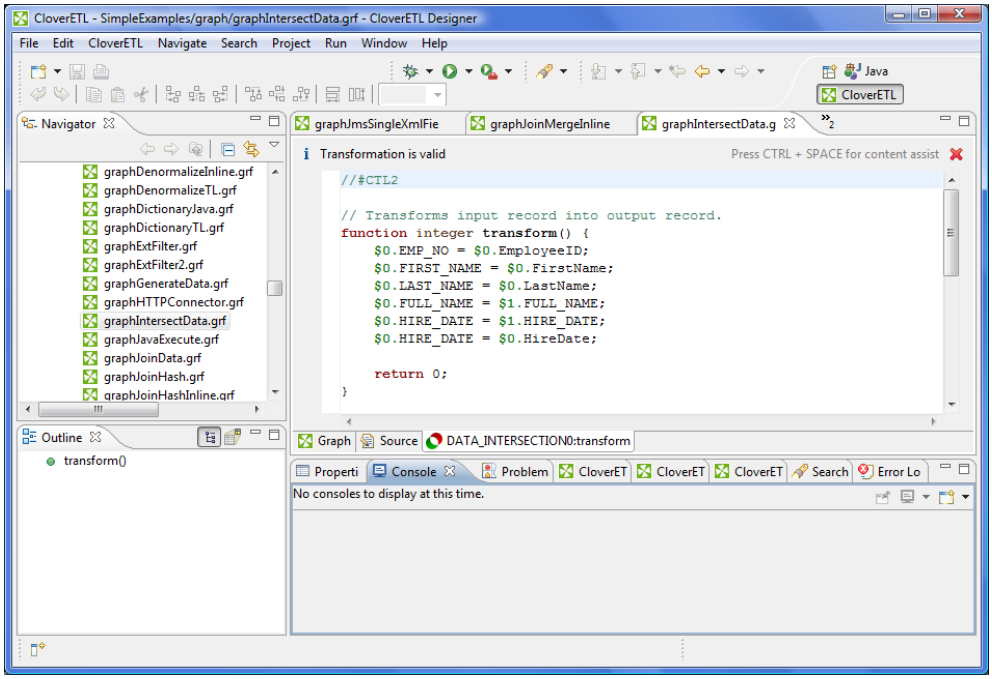


図42.14. CTLでの変換定義(グラフ・エディタの変換タブ)

このタブに切り替えると、宣言された変数および関数を「**Outline**」ペインで確認できます。(タブは、タブ右上隅の赤のX記号をクリックすると閉じます。)

「**Outline**」ペインは次のように表示されます。

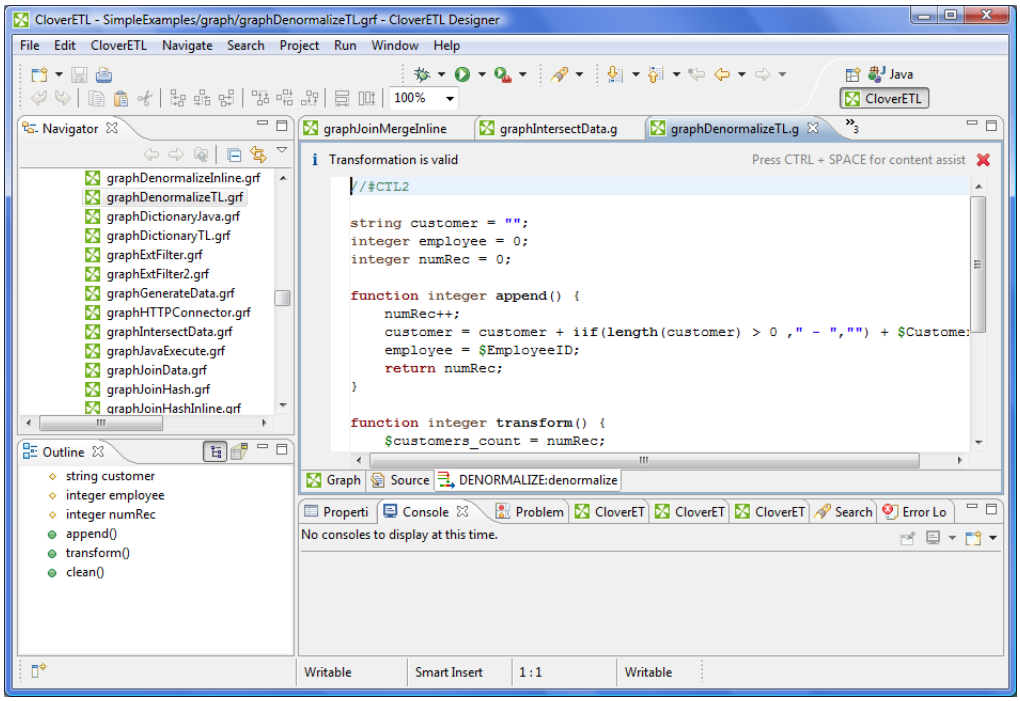


図42.15. 変数と関数を表示する「Outline」ペイン

[Ctrl]キーを押しながら**[Space]**キーを押すと、コンテンツ・アシストを使用できます。

この2つのキーを任意の式の内側で押すと、変換を定義するために必要な記述内容がヘルプによりアドバイスされます。

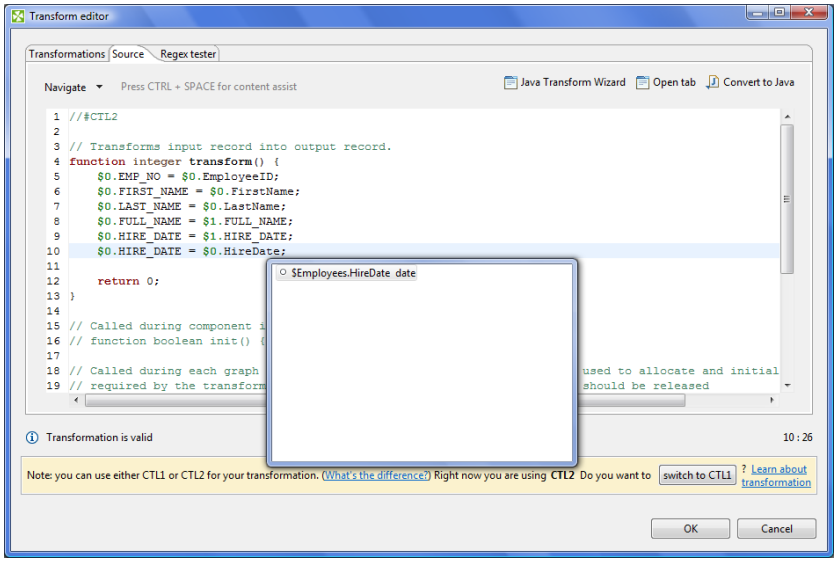


図42.16. コンテンツ・アシスト(レコードおよびフィールド名)

この2つのキーを任意の式の外側で押すと、変換を定義するために使用可能な関数のリストがヘルプにより提供されます。

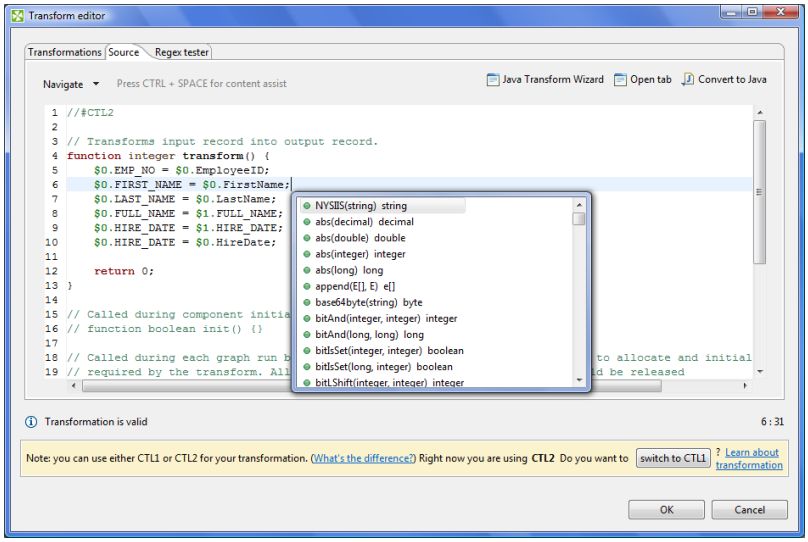


図42.17. コンテンツ・アシスト(CTL関数のリスト)

ヒント

[Shift]キーと**[Space]**キーを同時に押すと、**使用可能なCTL関数を示すダイアログ**が表示されます。

定義にエラーがある場合、行が赤色の円とその中の白色のX記号で強調表示され、左下隅にはこのエラーに関する詳細情報が表示されます。

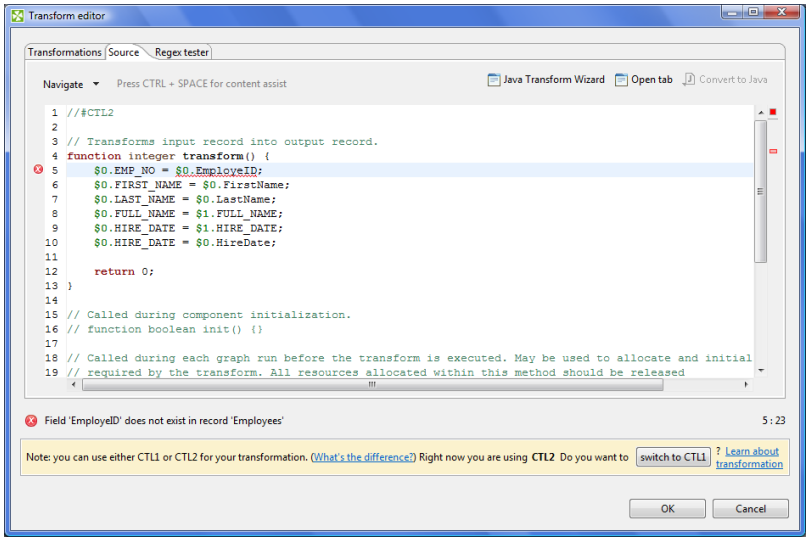


図42.18. 変換のエラー

変換コードをJava言語に変換する場合は、「**Convert to Java**」ボタンをクリックして、cloverプリプロセッサ・マクロを使用するかどうかを選択します。

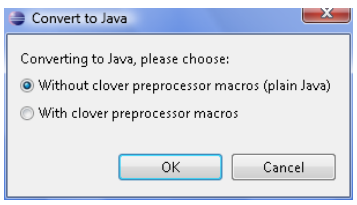


図42.19. 変換のJavaへの変換

選択後に「**OK**」をクリックすると、変換は、次の形式に変換されます。

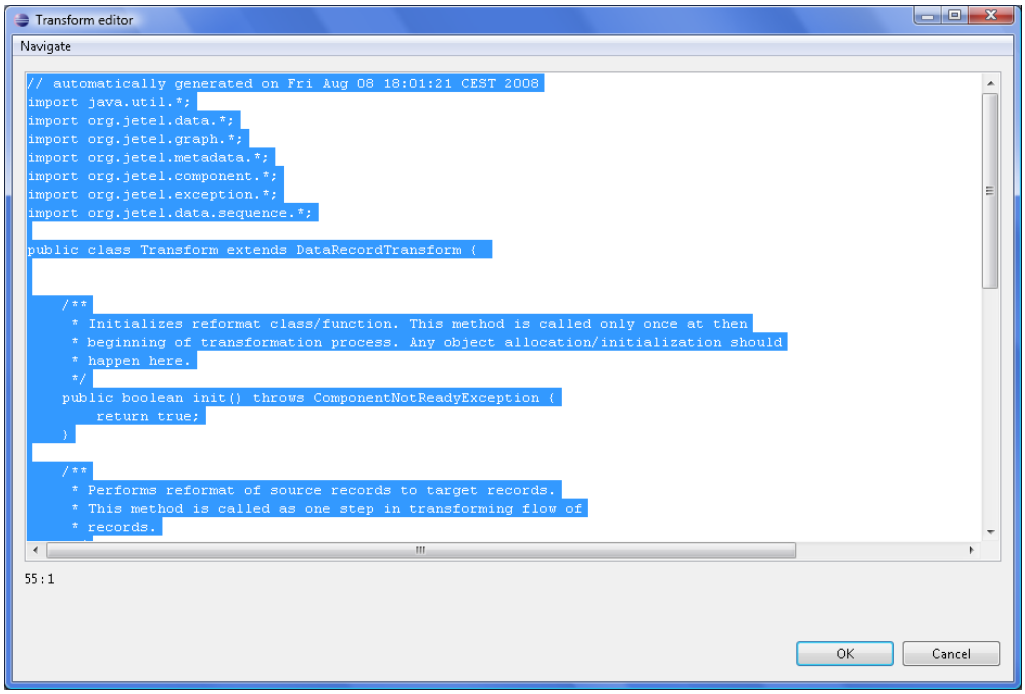


図42.20. Javaでの変換定義

最後に関数`getMessage()`を定義することで、独自のエラー・メッセージも定義できます。これは、コンソールに書き込まれる文字列を返します。各コンポーネント内の変換の詳細は、これらのコンポーネントについて説明されている項に記載されています。

重要

`getMessage()`関数は、整数データ型を返す関数内からのみ呼び出されます。

この関数の呼出しを可能にするには、-2以下の値の`return`文を、整数を返す関数に追加する必要があります。たとえば、`transform()`、`append()`または`count()`などのいずれかの関数が-2を返す場合、`getMessage()`が呼び出されてメッセージがコンソールに書き出されます。

Regex Tester

これは、変換エディタの最後のタブで、タブ・ペイン(p.43)で説明されています。

共通Javaインタフェース

共通のTransformインタフェースのメソッドを次に示します。

- `void setNode(Node node)`

  グラフ・ノードとこの変換を関連付けます。

- `Node getNode()`

  この変換と関連付けられているグラフ・ノードを返すか、グラフ・ノードが関連付けられていない場合はnullを返します。

- `TransformationGraph getGraph()`

  この変換と関連付けられているTransformationGraphを返すか、グラフが関連付けられていない場合はnullを返します。

- `void preExecute()`

  変換が実行される前の各グラフの実行中に呼び出されます。変換で必要なリソースの割当てと初期化に使用できます。このメソッド内に割り当てられたすべてのリソースは、postExecute()メソッドによって解放される必要があります。

- `void postExecute(TransactionMethod transactionMethod)`

  変換全体が実行された後の各グラフの実行中に呼び出されます。preExecute()メソッド内に割り当てられたすべてのリソースを解放するために使用する必要があります。

- `String getMessage()`

  変換中にエラーが発生して、変換により-2以下の値が返された場合に、任意のユーザー定義エラー・メッセージをレポートするために呼び出されます。これは、append()、count()、generate()、getOutputPort()、transform()またはupdateTansform()のいずれか、またはそれらに対応するいずれかのOnError()が-2以下の値を返した場合に呼び出されます。

- `void finished()` (非推奨)

  すべての入力レコードが処理された後の変換の最後に呼び出されます。

- `void reset()` (非推奨)

  変換を初期状態にリセットします(別の実行用)。このメソッドは、変換が正常に初期化された後にのみ呼び出されます。

第43章 リーダーの共通プロパティ

リーダーはグラフの最初のコンポーネントです。これらは、データ・ソースからデータを読み取り、別のグラフ・コンポーネントにそのデータを送信します。これが、各リーダーにはデータを出力する少なくとも1つの出力ポートが必要である理由です。**リーダー**は、ディスク上に存在するファイルまたはデータベースからデータを読み取ることができます。これらは、FTP、LDAPまたはJMSを使用した接続からもデータを受信できます。一部の**リーダー**では、エラーに関する情報を記録できます。リーダーの1つである**Data Generator**コンポーネントは、指定されたパターンに従ってデータを生成します。また、一部の**リーダー**にはオプションの入力ポートがあり、ここからもデータを受信できます。また、これらはディクショナリからデータを読み取ることができます。

リーダーを右クリックして「**View data**」オプションを選択すると、入力データの一部を確認できます。その後、エッジをデバッグするときと同じ「**View data**」ダイアログが表示されます。詳細は、デバッグ・データの表示(p.106)を参照してください。このダイアログでは、読み取られたデータを表示できます(グラフの実行前でも使用可能)。

これらのオプションへのリンクの概要を次に示します。

- ローカルおよびリモートのファイルからの読取り、プロキシを介した読取り、コンソール、入力ポートおよびディクショナリからの読取りを行うための「**File URL**」属性のいくつかの例

  リーダーにサポートされているファイルURL形式(p.297)

- リーダー上のデータの表示(p.301)
- 入力ポートの読取り(p.303)
- 増分読取り(p.304)
- 入力レコードの選択(p.305)
- データ・ポリシー(p.306)
- XML機能(p.307)
- 変換の定義(p.279)で示されているように、一部の**リーダー**では変換の定義が可能であるか、またはそれが必須である場合があります。ローカルおよびリモートのファイルからの読取り、プロキシを介した読取り、コンソール、入力ポートおよびディクショナリからの読取りを行うための属性のいくつかの例も示します。CTLで記述された変換の変換テンプレートの詳細は、次を参照してください。

  リーダー用のCTLテンプレート(p.308)

- 変換の定義(p.279)で示されているように、一部の**リーダー**では変換の定義が可能であるか、またはそれが必須である場合があります。ローカルおよびリモートのファイルからの読取り、プロキシを介した読取り、コンソール、入力ポートおよびディクショナリからの読取りを行うための属性のいくつかの例も示します。Javaで記述された変換に実装する必要がある変換インタフェースの詳細は、次を参照してください。

  リーダー用Javaインタフェース(p.308)

すべての**リーダー**の概要を次に示します。

表43.1. リーダーの比較

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
DataGenerator(p.351)	なし	0	1-n	✗	✓	✓	はい[3]	✓	✓
UniversalDataReader(p.415)	フラット・ファイル	0-1	1-2	✗	✗	✗	✗	✗	✗
ParallelReader(p.396)	フラット・ファイル	0	1	✗	✗	✗	✗	✗	✗
CloverDataReader(p.341)	cloverバイナリ・ファイル	0	1-n	✓	✗	✗	✗	✗	✗
SpreadsheetDataReader(p.404)	XLS(X)ファイル	0-1	1-2	✗	✗	✗	✗	✗	✗
XLSDataReader(p.421)	XLS(X)ファイル	0-1	1-n	✓	✗	✗	✗	✗	✗
DBFDataReader(p.359)	dBaseファイル	0-1	1-n	✓	✗	✗	✗	✗	✗
DBInputTable(p.361)	データベース	0	1-n	✓	✗	✗	✗	✗	✗
XMLExtract(p.426)	XMLファイル	0-1	1-n	✗	✓	✗	✗	✗	✗
XMLXPathReader(p.452)	XMLファイル	0-1	1-n	✗	✓	✗	✗	✗	✗
JMSReader(p.377)	jmsメッセージ	0	1	-	-	✓	✗	✓	✗
EmailReader(p.365)	電子メール・メッセージ	0	1	-	-	✓	✗	✓	✗
LDAPReader(p.387)	LDAPディレクトリ・ツリー	0	1-n	✗	✗	✗	✗	✗	✗
MultiLevelReader(p.392)	フラット・ファイル	1	1-n	✗	✓	✓	✓	✓	✗
ComplexDataReader(p.343)	フラット・ファイル	1	1-n	✗	✓	✓	✓	✓	✓
QuickBaseRecordReader(p.400)	QuickBase	0-1	1-2	✗	✗	✗	✗	✗	✗
QuickBaseQueryReader(p.402)	QuickBase	0	1	✗	✗	✗	✗	✗	✗
LotusReader(p.390)	Lotus Notes	0	1	✗	✗	✗	✗	✗	✗
HadoopReader(p.375)	Hadoopシーケンス・ファイル	0	1	✗	✗	✗	✗	✗	✗

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します(**DataGenerator**および**MultiLevelReader**)。詳細は、変換の戻り値(p.283)を参照してください。**XMLExtract**および**XMLXPathReader**は、それぞれの「**Mapping**」属性または「**Mapping URL**」属性で定義されたとおりにデータをポートに送信します。

リーダーにサポートされているファイルURL形式

「**File URL**」属性は、URLファイル・ダイアログ(p.69)を使用して定義できます。

重要

グラフの移植性を確保するため、URL内のパスを定義する場合にはフォワード・スラッシュを使用する必要があります(Microsoft Windowsの場合でも)。

リーダーで使用可能なURLの例を次に示します。

ローカル・ファイルの読取り

- `/path/filename.txt`

  指定されたファイルを読み取ります。

- `/path1/filename1.txt;/path2/filename2.txt`

  指定された2つのファイルを読み取ります。

- `/path/filename?.txt`

  マスクを満たすすべてのファイルを読み取ります。

- `/path/*`

  指定されたディレクトリ内のすべてのファイルを読み取ります。

- `zip:(/path/file.zip)`

  `file.zip`ファイル内に圧縮されている最初のファイルを読み取ります。

- `zip:(/path/file.zip)#innerfolder/filename.txt`

  `file.zip`ファイル内に圧縮されている指定されたファイルを読み取ります。

- `gzip:(/path/file.gz)`

  `file.gz`ファイル内に圧縮されている最初のファイルを読み取ります。

- `tar:(/path/file.tar)#innerfolder/filename.txt`

  `file.tar`ファイル内にアーカイブされている指定されたファイルを読み取ります。

- `zip:(/path/file??.zip)#innerfolder?/filename.*`

  指定されたマスクを満たす、zip圧縮ファイル内のすべてのファイルを読み取ります。圧縮ファイル名、内部フォルダ名および内部ファイル名には、ワイルドカード(`?`および`*`)を使用できます。

- `tar:(/path/file????.tar)#innerfolder??/filename*.txt`

  指定されたマスクを満たす、アーカイブ・ファイル内のすべてのファイルを読み取ります。圧縮ファイル名、内部フォルダ名および内部ファイル名には、ワイルドカード(`?`および`*`)を使用できます。

- `gzip:(/path/file*.gz)`

  指定されたマスクを満たす、ファイル内にgzip圧縮されているすべてのファイルを読み取ります。圧縮ファイル名にはワイルドカードを使用できます。

- `tar:(gzip:/path/file.tar.gz)#innerfolder/filename.txt`

  `file.tar.gz`ファイル内に圧縮されている指定されたファイルを読み取ります。**CloverETL**では`.tar`ファイル内のデータを読み取ることができますが、`.tar`ファイルへの書込みはサポートされていません。

- `tar:(gzip:/path/file??.tar.gz)#innerfolder?/filename*.txt`

  指定されたマスクを満たす、gzip圧縮された`tar`アーカイブ内のすべてのファイルを読み取ります。圧縮ファイル名、内部フォルダ名および内部ファイル名には、ワイルドカード(`?`および`*`)を使用できます。

- `zip:(zip:(/path/name?.zip)#innerfolder/file.zip)#innermostfolder?/filename*.txt`

  指定されたzipマスクを満たすすべての圧縮ファイルから、指定されたパス・マスクを満たすすべてのパスに存在する、ファイル・マスクを満たすすべてのファイルを読み取ります。外部圧縮ファイル名、最下層フォルダ名および最下層ファイル名には、ワイルドカード(?および*)を使用できます。これらは、内部フォルダ名および内部zipファイル名には使用できません。

リモート・ファイルの読取り

- `ftp://username:password@server/path/filename.txt`

  ユーザー名とパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバー上の指定された`filename.txt`ファイルを読み取ります。

- `sftp://username:password@server/path/filename.txt`

  ユーザー名とパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバー上の指定された`filename.txt`ファイルを読み取ります。

- `http://server/path/filename.txt`

  httpプロトコルを介して接続したリモート・サーバー上の指定された`filename.txt`ファイルを読み取ります。

- `https://server/path/filename.txt`

  httpsプロトコルを介して接続したリモート・サーバー上の指定された`filename.txt`ファイルを読み取ります。

- `zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt`

  ユーザー名とパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバー上の`file.zip`ファイル内に圧縮されている、指定された`filename.txt`ファイルを読み取ります。

- `zip:(http://server/path/file.zip)#innerfolder/filename.txt`

  httpプロトコルを介して接続したリモート・サーバー上の`file.zip`ファイル内に圧縮されている、指定された`filename.txt`ファイルを読み取ります。

- `tar:(ftp://username:password@server/path/file.tar)#innerfolder/filename.txt`

  ユーザー名とパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバー上の`file.tar`ファイル内にアーカイブされている、指定された`filename.txt`ファイルを読み取ります。

- `zip:(zip:(ftp://username:password@server/path/name.zip)#innerfolder/file.zip)#innermostfolder/filename.txt`

  ユーザー名とパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバー上の、`name.zip`ファイル内で圧縮されている、`file.zip`ファイル内でさらに圧縮されている、指定された`filename.txt`ファイルを読み取ります。

- `gzip:(http://server/path/file.gz)`

  httpプロトコルを介して接続したリモート・サーバー上の`file.gz`ファイル内に圧縮されている最初のファイルを読み取ります。

- `http://server/filename*.dat`

  指定されたマスク(*のみがサポートされます)を満たす、WebDAVサーバー上のすべてのファイルを読み取ります。

- `http://access_key_id:secret_access_key@bucketname.s3.amazonaws.com/filename*.out`

アクセス・キーIDとシークレット・アクセス・キーを使用して、Amazon S3 Webストレージ・サービスの指定されたバケットから、指定されたマスク(*のみがサポートされます)を満たすすべてのファイルを読み取ります。

- `hdfs://CONN_ID/path/filename.dat`

  Hadoop分散ファイル・システム(HDFS)上のファイルを読み取ります。接続先のHDFS NameNodeは、IDが`CONN_ID`であるHadoop接続(p.192)で定義されます。この例のファイルURLでは、絶対HDFSパス`/path/filename.dat`のファイルが読み取られます。

入力ファイルからの読取り

- `port:$0.FieldName:discrete`

  入力ポートの読取りのために選択された各レコード・フィールドのデータが、単一の入力ファイルとして読み取られます。

- `port:$0.FieldName:source`

  URLアドレス、つまり入力ポートの読取りのために選択されたフィールドの値がロードされて解析されます。

- `port:$0.FieldName:stream`

  入力ポート・フィールド値は連結され、1つまたは複数の入力ファイルとして処理されます。`null`値は、`eof`に置き換えられます。

コンソールからの読取り

- `-`

  グラフの開始後に、`stdin`からデータを読み取ります。読取りを停止する場合は、**[Ctrl]**を押しながら**[Z]**を押します。

読取りにおけるプロキシの使用方法

- `http:(direct:)//seznam.cz`

  プロキシを使用しません。

- `http:(proxy://user:password@212.93.193.82:443)//seznam.cz`

  httpプロトコルのプロキシ設定です。

- `ftp:(proxy://user:password@proxyserver:1234)//seznam.cz`

  ftpプロトコルのプロキシ設定です。

- `sftp:(proxy://66.11.122.193:443)//user:password@server/path/file.dat`

  sftpプロトコルのプロキシ設定です。

ディクショナリからの読取り

- `dict:keyName:discrete`[1)]

  ディクショナリからデータを読み取ります。

- `dict:keyName:source`[1)]

  `discrete`処理タイプと同様にディクショナリからデータを読み取りますが、ディクショナリ値が入力ファイルのURLであると想定します。この入力からのデータが**リーダー**に渡されます。

説明:

1): **リーダー**はディクショナリからのソース値の型を特定して、パーサー用の読取り可能チャネルを作成します。**リーダー**では次の型のソースがサポートされます。`InputStream`, `byte[]`、`ReadableByteChannel`、`CharSequence`、`CharSequence[]`、`List<CharSequence>`、`List<byte[]>`、`ByteArrayOutputStream`。

データ・ソースとしてのサンドボックス・リソース

サンドボックス・リソースは、それが共有、ローカルまたはパーティション化されたサンドボックスのいずれであるかに関係なく、グラフの「fileURL」属性で、次のようなサンドボックスURLとして指定します。

```
sandbox://data/path/to/file/file.dat
```

ここで、dataはサンドボックスのコード、path/to/file/file.datはサンドボックス・ルートからリソースへのパスです。URLはCloverETL Serverによってグラフの実行中に評価され、コンポーネント(リーダーまたはライター)は開かれたストリームをサーバーから取得します。これは、ローカル・ファイルまたは別のリモート・リソースへのストリームにできます。したがって、グラフはリソースにローカル・アクセスできるノード上で実行する必要はありません。グラフでは複数のサンドボックス・リソースを使用でき、各リソースは異なるノード上に存在できます。このような場合、CloverETL Serverでは、リモート・ストリームを最小化するために、最も多くのローカル・リソースが存在するノードを選択します。

サンドボックスURLには、データのパラレル処理のための、特別な使用方法があります。パーティション化されたサンドボックス内のリソースのサンドボックスURLが使用される場合、この部分のグラフ/フェーズは、パーティション化されたサンドボックスのロケーション・リストによって指定されたノード割当てに従い、並行して実行されます。したがって、各ワーカーに、独自のローカル・サンドボックス・リソースがあります。CloverETL Serverは各ワーカー上のサンドボックスURLを評価して、ローカル・リソースへの開かれたストリームをコンポーネントに提供します。

リーダー上のデータの表示

リーダー上のデータは、コンテキスト・メニューを使用して表示できます。このことを行うには、対象のコンポーネントを右クリックして、コンテキスト・メニューから「**View data**」を選択します。

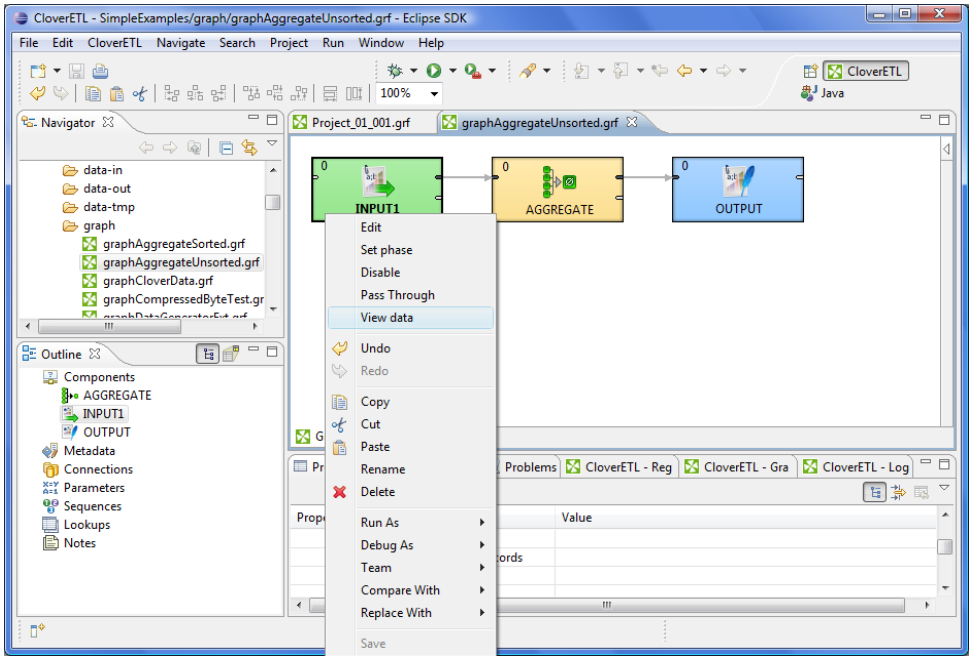


図43.1. コンポーネント上のデータの表示

その後、データをプレーン・テキストとして表示するか、グリッド(解析済データのプレビュー)として表示するかを選択できます。**プレーン・テキスト**・オプションを選択した場合、**キャラクタ・セット**を選択できますが、フィルタ式は選択できません。複数のコンポーネントのデータを同時に表示できます。結果を区別するために、ウィンドウ・タイトルには、表示しているエッジの情報が、GRAPH.name:COMPONENT.nameの形式で表示されます。

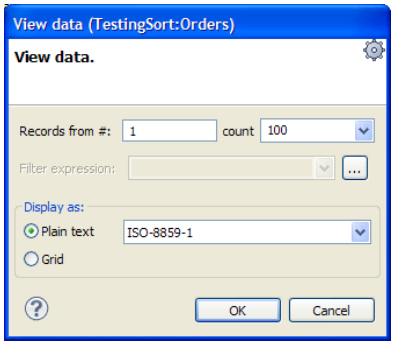


図43.2. プレーン・テキストとしてのデータの表示

一方、**グリッド**・オプションを選択した場合は、**フィルタ式**を選択できますが、**キャラクタ・セット**は選択できません。

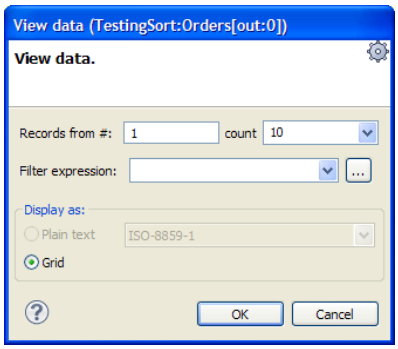


図43.3. データをグリッドとして表示

プレーン・テキスト・モードでは、結果は次のように表示されます。

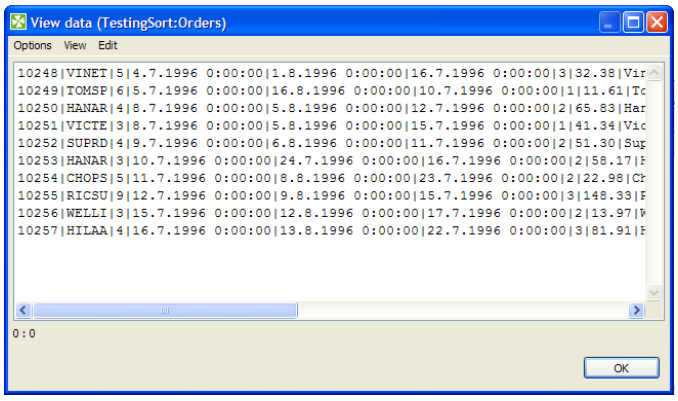
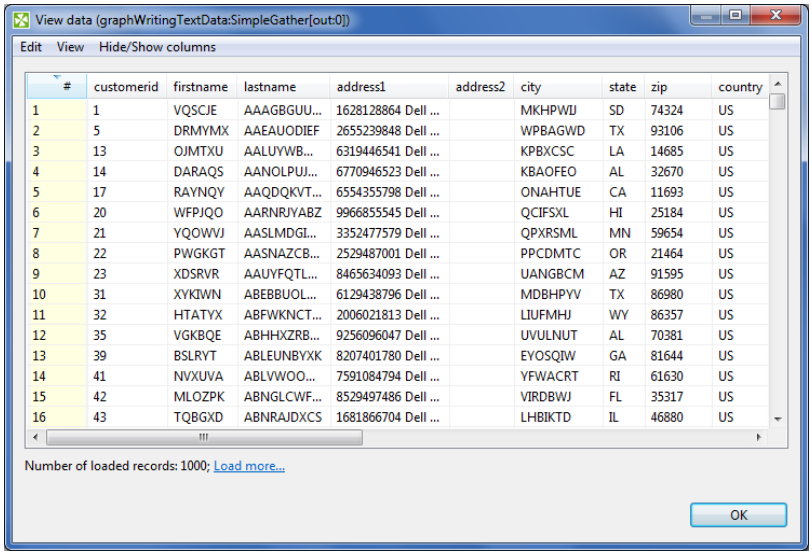


図43.4. プレーン・テキスト・データ表示

グリッド・モードでは、次のように表示されます。



#	customerid	firstname	lastname	address1	address2	city	state	zip	country
1	1	VQSCJE	AAAGBGUU...	1628128864 Dell ...		MKHPWIJ	SD	74324	US
2	5	DRMYMX	AAEAUODIEF	2655239848 Dell ...		WPBAGWD	TX	93106	US
3	13	OJMTXU	AALUYWB...	6319446541 Dell ...		KPBXCSC	LA	14685	US
4	14	DARAQS	AANOLPUJ...	6770946523 Dell ...		KBAOFEO	AL	32670	US
5	17	RAYNQY	AAQDQKVT...	6554355798 Dell ...		ONAHTUE	CA	11693	US
6	20	WFPJQO	AARNRJYABZ	9966855545 Dell ...		QCIFSXL	HI	25184	US
7	21	YQOWVJ	AASLMDGI...	3352477579 Dell ...		QPXRSML	MN	59654	US
8	22	PWGKGT	AASNAZCB...	2529487001 Dell ...		PPCDMTC	OR	21464	US
9	23	XDSRVR	AAUYFQTL...	8465634093 Dell ...		UANGBCM	AZ	91595	US
10	31	XYKIWN	ABEBBUOL...	6129438796 Dell ...		MDBHPYV	TX	86980	US
11	32	HTATYX	ABFWKNCT...	2006021813 Dell ...		LIUFMHJ	WY	86357	US
12	35	VGKBQE	ABHHXZRB...	9256096047 Dell ...		UVULNUT	AL	70381	US
13	39	BSLRYT	ABLEUNBYXK	8207401780 Dell ...		EYOSQIW	GA	81644	US
14	41	NVXUVA	ABLVWOO...	7591084794 Dell ...		YFWACRT	RI	61630	US
15	42	MLOZPK	ABNGLCWF...	8529497486 Dell ...		VIRDBWJ	FL	35317	US
16	43	TQBGXD	ABNRAJDXCS	1681866704 Dell ...		LHBIKTD	IL	46880	US

図43.5. グリッド・データ表示

注意

表示対象のデータが多すぎる場合は、ビューの下に「**Load more...**」という青色のテキストが表示されます。これをクリックすると、現在表示されているレコードの後に、新しいレコードのチャンクが追加されます。**プレーン・ビュー**でも、ビューの最下部までスクロール・ダウンすると(または**[Page Down]**を押すと)、レコードがロードされます。

一部の**ライター**でも、同じことを実行できます。ライター上のデータの表示(p.314)を参照してください。ただし、出力ファイルの作成が完了している必要があります。

入力ポートの読取り

一部の**リーダー**では、オプションの入力ポートからデータを読み取ることができます。

入力ポートの読取りは、次の**リーダー**でサポートされています。

- **UniversalDataReader**
- **XLSDataReader**
- **SpreadsheetDataReader**
- **DBFDataReader**
- **XMLExtract**
- **XMLXPathReader**
- **MultiLevelReader (商用コンポーネント)**

重要

ポート読取りは、**DBExecute**によるSQLコマンドの受信にも使用できます。**問合せURL**は、$0.fieldName:discreteのようになります。SQLコマンドは、ファイルから読み取ることもできます。この場合は、その名前がパスを含めて入力ポートからDBExecuteに渡され、「**Query URL**」属性はport:$0.fieldName:sourceである必要があります。

これらのいずれかの**リーダー**のオプションの入力ポートをエッジに接続する場合、このエッジのもう一端をデータ・ソースに接続する必要があります。フィールド・マッピングのプロトコルを定義するには、データ読取り元の

フィールドを**リーダー**の「**File URL**」属性で設定する必要があります。FieldName入力フィールドの型は、入力エッジ・メタデータの定義と同様に、string、byteまたはcbyteのいずれかである必要があります。

プロトコルの構文は、port:$0.FieldName[:processingType]です。

ここで、processingTypeはオプションであり、データがプレーン・データとして処理されるか、またはURLアドレスとして処理されるかを定義します。これは、source、discreteまたはstreamのいずれかとなります。明示的に設定されてない場合、デフォルトでdiscreteが適用されます。

URLファイル・ダイアログ(p.69)を使用して、入力ポートの読取りの属性を定義することもできます。

グラフが実行されると、データが元のデータ・ソースから読み取られ(**リーダー**のオプション入力ポートに接続しているエッジのメタデータに従う)、オプション入力ポートを介して**リーダー**に受信されます。各レコードは他のレコードから独立して読み取られます。各レコードの指定されたフィールドが、出力メタデータに従って**リーダー**により処理されます。

- discrete

  入力ポートからの各データ・レコード・フィールドは、特定のデータ・ソースを表します。

- source

  入力ポートからの各データ・レコード・フィールドは、ロードして解析されるURLを表します。

- stream

  入力ポートからのすべてのデータ・フィールドは、連結されて1つの入力ファイルとして処理されます。このフィールドの値がnullである場合、eofに置き換えられます。以降のデータ・レコード・フィールドは、別の入力ファイルとして同じ方法で、つまり、null値を受信するまで解析されます。リーダーは、ポートに最初のバイトが到達すると、即座にデータの解析を開始して、eofが出現するまで順次データを処理します。stream処理タイプでの書込みの詳細は、出力ポート書込み(p.316)を参照してください。

増分読取り

一部の**リーダー**では、増分読取りを使用できます。グラフが同じファイルまたはファイルの集合を複数回読み取る場合、最後のグラフ実行以降に追加されたレコードまたはファイルのみが考慮されます。

次の4つの**リーダー**では、**増分ファイル**および**増分キー**属性を設定できます。**増分キー**は、読み取られたレコード/ファイルに関する情報を保持する文字列です。このキーは、**増分ファイル**に保存されます。このようにすると、**増分ファイル**にマークされていないレコードまたはファイルのみがコンポーネントによって読み取られるようになります。

増分読取りが可能な**リーダー**は次のとおりです。

- **UniversalDataReader**
- **XLSDataReader**
- **DBFDataReader**

データベースからデータを読み取るコンポーネントは、この増分読取りを異なる方法で実行します。

- **DBInputTable**

他の増分リーダーとは異なり、このデータベース・コンポーネントでは、より多くのデータベース列をキー・フィールドとして評価および使用できます。**増分キー**は、個別の式keyname=FUNCTIONNAME(db_field)[!InitialValue]がセミコロンで区切られたシーケンスです。たとえば、key01=MAX(EmployeeID);key02=FIRST(CustomerID)!20という**増分キー**を設定できます。選択できる関数は、FIRST、LAST、MIN、MAXの4つです。同時に、**増分キー**を定義する場合は、これらのキー部分を**問合せ**に追加することも必要となります。問合せでは、where文の部分が、たとえば、where

db_field1 > #key01 and db_field2 < #key02のようになります。これにより、次回に読み取るレコードを制限できます。このことは、これらのdb_field1およびdb_field2フィールドの値に依存します。問合せで指定された条件を満たすレコードのみが読み取られます。これらのキー・フィールド値は**増分ファイル**に保存されます。**増分キー**を定義するには、この属性行をクリックし、**「Define incremental key」**ダイアログの**プラス**または**マイナス・**ボタンをクリックしてキー名を追加または削除して、DBフィールド名および関数名を選択します。最後の2つは、いずれも可能な値のコンボ・リストから選択します。

注意

バージョン2.8.1以降の**CloverETL Designer**では、各キーに対して初期値を定義することもできます。このようにすると、指定された値を設定して新規の**増分ファイル**を作成できます。

入力レコードの選択

リーダーを設定するときは、読み取られるレコードを制限する必要がある場合があります。

一部の**リーダー**では、同時に複数のファイルを読み取ることができます。これらの**リーダー**では、読み取られるレコード数を各入力ファイルに対して個別に定義でき、またすべての入力ファイルに対する合計でも定義できます。

これらの**リーダー**では、**「Number of skipped records」**または**「Max number of records」**、あるいはその両方の属性を定義できます。前者の属性は、スキップするレコード数を指定し、後者は読み取るレコード数を定義します。これらのレコードのスキップまたは読取り(あるいはその両方)は、すべての入力ファイルにわたって連続的に行われます。これらのレコードのスキップまたは読取り(あるいはその両方)は、次に示す類似の2つの属性の値に基づき、独立して実行されます。

これらのコンポーネントでは、各入力ファイルでスキップまたは読取り(あるいはその両方)が行われるレコード数を指定することも可能です。このことを行うには、**「Number of skipped records per source」**または**「Max number of records per source」**、あるいはその両方の属性を設定します。

したがって、スキップされるレコード数の合計は、**「Number of skipped records per source」**とソース・ファイル数を乗算して**「Number of skipped records」**を加算した数と等しくなります。

また、読み取られるレコード数の合計は、**「Max number of records per source」**とソース・ファイル数を乗算して**「Max number of records」**を加算した数と等しくなります。

個別の入力ファイルおよびすべての入力ファイル合計の両方でレコード数を制限できる**リーダー**は次のとおりです。

- **UniversalDataReader**
- **XLSDataReader**
- **SpreadsheetDataReader**
- **DBFDataReader**
- **MultiLevelReader (商用コンポーネント)**

CloverDataReaderは前述のコンポーネントとは異なり、すべての入力ファイルの合計レコード数のみを制限できます。

- **CloverDataReader**では、前述のコンポーネントと同様、**「Number of skipped records」**または**「Max number of records」**、あるいはその両方の属性を使用して、合計レコード数を制限することのみができます。

次の2つの**リーダー**では、**「Number of skipped mappings」**(スキップされるマッピング数)または**「Max number of mappings」**(マッピング最大数)、あるいはその両方の属性を使用してレコードの合計数を制限できます。ここで**マッピング**と呼ばれるものは、マッピングされて出力ポートから送信される必要があるサブツリーです。

- **XMLExtract**。前述の内容に加え、このコンポーネントでは個別のXML要素の「skipRows」または「numRecords」(あるいはその両方の)属性を使用できます。
- **XMLXPathReader**。前述の内容に加え、このコンポーネントでは、XPath言語を使用してマッピングされるXML構造の数を制限できます。

次の**リーダー**では、異なる方法で数を制限できます。

- **JMSReader**では、「**Max msg count**」属性、またはコンポーネント・インタフェースのendOfInput()メソッドのfalse戻り値(あるいはその両方)を使用して、受信されて処理されるメッセージ数を制限できます。
- **QuickBaseRecordReader (商用コンポーネント)**。このコンポーネントでは、「**Records list**」属性を使用して読み取られるレコード数を指定します。
- **QuickBaseQueryReader (商用コンポーネント)**。このコンポーネントでは、「**Query**」または「**Options**」属性を使用してレコード数を制限できます。
- **DBInputTable**。このコンポーネントでは「**SQL query**」または「**Query URL**」属性を使用してレコード数を制限できます。

次の**リーダー**では、読み取られるレコード数を制限できません(すべて読み取られます)。

- **LDAPReader**
- **ParallelReader (商用コンポーネント)**

データ・ポリシー

一部の**リーダー**では、データ・ポリシーを設定できます。そのリストを次に示します。

- **UniversalDataReader**
- **ParallelReader (商用コンポーネント)**
- **XLSDataReader**
- **DBFDataReader**
- **DBInputTable**
- **XMLXPathReader**
- **MultiLevelReader (商用コンポーネント)**
- **SpreadsheetDataReader (商用コンポーネント)**

これらのコンポーネントを構成するときには、初めに、正しくない、または不完全なレコードが解析された場合の対応を決定する必要があります。このことは、「**Data Policy**」属性を利用して指定できます。選択するデータ・ポリシーに応じて、3つのオプションがあります。

- **Strict**。このデータ・ポリシーがデフォルトで設定されます。これは、値や形式が正しくないレコード・フィールドが読み取られたときに、データ解析が停止することを意味します。後続の処理は中断されます。
- **Controlled**。このデータ・ポリシーでは、すべてのエラーが記録されますが、正しくないレコードはスキップされてデータ解析は続行します。通常は、正しくないレコードとエラー情報がstdoutに記録されます。これらは、**UniversalDataReader**および**SpreadsheetDataReader**でのみ、オプションの2つ目のポートを介して送信できます。

重要

UniversalDataReaderで「**Data policy**」属性を「controlled」に設定した場合、情報の処理または単なる書込みを実行するコンポーネントを選択する必要があります。エッジを選択して、**UniversalDataReader** (「data policy」属性を「controlled」に設定済)のエラー・ポートを、選択したライターの入力ポートに接続するか(書込みのみを行う場合)、またはその他の処理コンポーネントの入力ポートに接続する必要があります。また、このエッジにはメタデータを割り当てる必要があります。メタデータは手動で作成する必要があります。これらは、number of incorrect record(正しくないレコードの番号)、number of incorrect field(正しくないフィールドの番号)、incorrect record(正しくないレコード)、error message(エラー・メッセージ)の4つのフィールドで構成されます。最初の2つのフィールドのデータ型はintegerで、その他の2つのデータ型はstringsです。ユーザーによるメタデータの作成方法の詳細は、ユーザーによるメタデータの作成(p.150)を参照してください。

SpreadsheetDataReaderのエラー・ポート・メタデータの形式の詳細は、**SpreadsheetDataReader**のドキュメントを参照してください。

- **Lenient**。このデータ・ポリシーは、正しくないレコードは単にスキップされ、データ解析が続行することを意味します。

XML機能

XMLExtract (p.426)およびXMLXPathReader (p.452)では、「**Xml features**」属性を指定することで入力XMLファイルの検証を構成できます。「Xml features」では、特定のチェックを有効化または無効化することによって、XMLの詳細な検証を構成します(『Parser Features』を参照してください)。これは、nameM:=trueまたはnameN:=false (ここで、各nameMは検証の対象とするXML機能)のいずれかの形式の、個別の式のシーケンスとして表現します。これらの式は、セミコロンで区切ります。

検証のオプションは次のとおりです。

- **Custom parser setting**
- **Default parser setting**
- **No validations**
- **All validations**

この属性は、次のダイアログを使用して定義できます。

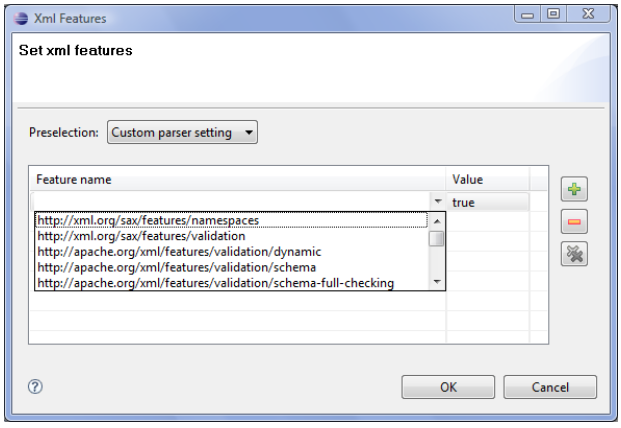


図43.6.「XML Features」ダイアログ

このダイアログでは、**プラス**・ボタンによる機能の追加や、trueまたはfalse値の選択などが可能です。

リーダー用CTLテンプレート

- DataGenerator (p.351)には、変換が必要です(CTLおよびJavaのどちらでも記述できます)。

  変換テンプレートの詳細は、DataGenerator用CTLテンプレート(p.354)を参照してください。

  このコンポーネントでは、変換によって返された値と番号が等しい接続済出力ポートを経由して各レコードを送信できます(変換の戻り値(p.283))。このようなポートには、マッピングを定義する必要があります。

リーダーのJavaインタフェース

- DataGenerator (p.351)には、変換が必要です(CTLおよびJavaのどちらでも記述できます)。

  インタフェースの詳細は、DataGenerator用Javaインタフェース(p.357)を参照してください。

  このコンポーネントでは、変換によって返された値と番号が等しい接続済出力ポートを経由して各レコードを送信できます(変換の戻り値(p.283))。このようなポートには、マッピングを定義する必要があります。

- JMSReader (p.377)では、オプションで、Javaでのみ記述できる変換を使用できます。

  インタフェースの詳細は、JMSReader用Javaインタフェース(p.379)を参照してください。

  このコンポーネントでは、すべての接続済出力ポートを介して各レコードを送信します。マッピングを定義する必要はありません。

- MultiLevelReader (p.392)では、Javaでのみ記述できる変換が必要です。

  詳細は、MultiLevelReader用Javaインタフェース(p.394)を参照してください。

第44章 ライターの共通プロパティ

ライターは変換グラフの最後のコンポーネントです。各ライターには、他のコンポーネントからこのグラフ・コンポーネントへのデータ・フローが経由する、少なくとも1つの入力ポートが必要です。ライターでは、ディスク上に存在するファイルやデータベース表にデータを書き込むか、またはFTP、LDAPやJMS接続を使用してデータを送信します。ライターの1つである**Trash**コンポーネントは、受信したすべてのレコードを(デバッグ・ファイルに保存するように設定されている場合を除き)破棄します。

すべてのライターで、データを既存のファイル、シートまたはデータベース表に追加するか(ファイルに対する「**Append**」属性など)、または既存のファイル、シートまたはデータベース表を新しいものに置き換えるかを決定することが重要です。「**Append**」属性は、デフォルトでfalseに設定されています。このことは、データを追加しないで置き換えることを意味します。

同じグラフの複数のライターによって、1つのファイルまたは1つのデータベース表にデータを書き込むことができますが、このような場合、異なるライターによるデータの書込みは異なるフェーズで行う必要があることを理解することが重要です。

(大部分のライターでは)ライターを右クリックして「**View data**」オプションを選択すると、結果データの一部を確認できます。その後、エッジをデバッグするときと同じ「View data」ダイアログが表示されます。詳細は、デバッグ・データの表示(p.106)を参照してください。このダイアログでは、書き込まれたデータを表示できます(グラフの実行後のみ使用可能)。

これらのオプションへのリンクの概要を次に示します。

- ローカルおよびリモートのファイルへの書込み、プロキシを介した書込み、コンソール、出力ポートおよびディクショナリへの書込みを行うための「**File URL**」属性のいくつかの例

  ライターにサポートされているファイルURL形式(p.310)

- ライター上のデータの表示(p.314)
- 出力ポート書込み(p.316)
- データの書込み方法および書込み先(p.316)
- 出力レコードの選択(p.317)
- 異なる出力ファイルへの出力のパーティション(p.318)
- 変換の定義(p.279)で示されているように、一部の**ライター**では変換の定義が可能であるか、またはそれが必須である場合があります。Javaで記述された変換に実装する必要がある変換インタフェースの詳細は、次を参照してください。

  ライター用Javaインタフェース(p.319)

すべての**ライター**の概要を次に示します。

表44.1. ライターの比較

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
Trash(p.550)	なし	1	0	✖	✖	✖	✖
UniversalDataWriter(p.552)	フラット・ファイル	1	0-1	✖	✖	✖	✖
CloverDataWriter(p.461)	Cloverバイナリ・ファイル	1	0	✖	✖	✖	✖
SpreadsheetDataWriter(p.532)	XLS(X)ファイル	1	0-1	✖	✖	✖	✖
XLSDataWriter(p.555)	XLS(X)ファイル	1	0-1	✖	✖	✖	✖
StructuredDataWriter(p.546)	構造化フラット・ファイル	1-3	0-1	✖	✖	✖	✖
EmailSender(p.481)	電子メール	1	0-2	✖	✖	✖	✖
DBOutputTable(p.473)	データベース	1	0-2	✖	✖	✖	✖
DB2DataWriter(p.464)	データベース	0-1	0-1	✖	✖	✖	✖
InfobrightDataWriter(p.487)	データベース	1	0-1	✖	✖	✖	✖
InformixDataWriter(p.489)	データベース	0-1	0-1	✖	✖	✖	✖
MSSQLDataWriter(p.514)	データベース	0-1	0-1	✖	✖	✖	✖
MySQLDataWriter(p.518)	データベース	0-1	0-1	✖	✖	✖	✖
OracleDataWriter(p.521)	データベース	0-1	0-1	✖	✖	✖	✖
PostgreSQLDataWriter(p.525)	データベース	0-1	0	✖	✖	✖	✖
XMLWriter(p.558)	XMLファイル	1-n	0-1	✖	✖	✖	✖
JMSWriter(p.501)	JMSメッセージ	1	0	✔	✖	✔	✖
LDAPWriter(p.509)	LDAPディレクトリ・ツリー	1	0-1	✖	✖	✖	✖
QuickBaseRecordWriter(p.530)	QuickBase	1	0-1	✖	✖	✖	✖
QuickBaseImportCSV(p.528)	QuickBase	1	0-2	✖	✖	✖	✖
LotusWriter(p.511)	Lotus Notes	1	0-1	✖	✖	✖	✖
DBFDataWriter(p.470)	.dbfファイル	1	0	✖	✖	✖	✖
HadoopWriter(p.485)	Hadoopシーケンス・ファイル	1	0	✖	✖	✖	✖

ライターにサポートされているファイルURL形式

「**File URL**」属性は、URLファイル・ダイアログ(p.69)を使用して定義できます。

次に示すURLには、ドル記号またはハッシュ記号のプレースホルダも含めることができます。

重要

ドル記号とハッシュ記号の使用は区別する必要があります。

- **ドル記号**は、「**Records per file**」属性に基づいて、指定された数のレコードのみが複数の出力ファイルのそれぞれに含まれている場合に使用します。
- **ハッシュ記号**は、指定された「**Partition key**」の値に対応するレコードが、複数の出力ファイルのそれぞれに含まれている場合にのみ使用します。

注意

この項のURLの例では、ハッシュ記号は、圧縮ファイル(`zip`、`gz`)とその内容を区別するために使用されています。これらは、プレースホルダではありません。

重要

グラフの移植性を確保するため、URL内のパスを定義する場合にはフォワード・スラッシュを使用する必要があります(Microsoft Windowsの場合でも)。

ライターで使用可能なURLの例を次に示します。

ローカル・ファイルへの書込み

- `/path/filename.out`

  指定されたファイルをディスク上のファイルに書き込みます。

- `/path1/filename1.out;/path2/filename2.out`

  指定された2つのファイルをディスク上のファイルに書き込みます。

- `/path/filename$.out`

  複数のファイルをディスクに書き込みます。ドル記号は1桁の数字を表します。したがって、出力ファイルの名前は`filename0.out`から`filename9.out`までにできます。ドル記号は、「**Records per file**」が設定されている場合に使用します。

- `/path/filename$$.out`

  複数のファイルをディスクに書き込みます。2つのドル記号は、2桁の数字を表します。したがって、出力ファイルの名前は`filename00.out`から`filename99.out`までにできます。ドル記号は、「**Records per file**」が設定されている場合に使用します。

- `zip:(/path/file$.zip)`

  複数の圧縮ファイルをディスクに書き込みます。ドル記号は1桁の数字を表します。したがって、圧縮出力ファイルの名前は`file0.zip`から`file9.zip`までにできます。ドル記号は、「**Records per file**」が設定されている場合に使用します。

- `zip:(/path/file$.zip)#innerfolder/filename.out`

  ディスク上の圧縮ファイルに、指定されたファイルが書き込まれます。ドル記号は1桁の数字を表します。したがって、指定されたfilename.outファイルを含む圧縮出力ファイルの名前は、`file0.zip`から`file9.zip`までにできます。ドル記号は、「**Records per file**」が設定されている場合に使用します。

- `gzip:(/path/file$.gz)`

  複数の圧縮ファイルをディスクに書き込みます。ドル記号は1桁の数字を表します。したがって、圧縮出力ファイルの名前は`file0.gz`から`file9.gz`までにできます。ドル記号は、「**Records per file**」が設定されている場合に使用します。

注意

CloverETLでは`.tar`ファイル内のデータを読み取ることができますが、`.tar`ファイルへの書込みはサポートされていません。

リモート・ファイルへの書込み

- `ftp://user:password@server/path/filename.out`

  ユーザー名とパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバーに、指定された`filename.out`ファイルを書き込みます。

- `sftp://user:password@server/path/filename.out`

  ユーザー名とパスワードを使用し、sftpプロトコルを介して接続したリモート・サーバーに、指定された`filename.out`ファイルを書き込みます。

- `zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt`

  ユーザー名とパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバーに、指定された`filename.txt`ファイルを`file.zip`ファイル内に圧縮して書き込みます。

- `zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt`

  ftpプロトコルを介して接続したリモート・サーバーに、指定された`filename.txt`ファイルを`file.zip`ファイル内に圧縮して書き込みます。

- `zip:(zip:(ftp://username:password@server/path/name.zip)#innerfolder/file.zip)#innermostfolder/filename.txt`

  ユーザー名とパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバーに、指定された`filename.txt`ファイルを、`file.zip`内で圧縮されている`name.zip`圧縮ファイルに、さらに圧縮して書き込みます。

- `gzip:(ftp://username:password@server/path/file.gz)`

  ftpプロトコルを介して接続したリモート・サーバーに、`file.gz`ファイル内の最初のファイルを圧縮して書き込みます。

- `http://username:password@server/filename.out`

  ユーザー名とパスワードを使用し、WebDAVプロトコルを介して接続したリモート・サーバーに、指定された`filename.out`ファイルを書き込みます。

- `http://access_key_id:secret_access_key@bucketname.s3.amazonaws.com/filename.out`

  アクセス・キーのIDとして`access_key_id`を、個人アクセス・キーとして`secret_access_key`を使用して、Amazon S3 Webストレージ・サービスのバケット`bucketname`に、指定された`filename.out`ファイルを書き込みます。

- `hdfs://CONN_ID/path/filename.dat`

  Hadoop分散ファイル・システム(HDFS)にファイルを書き込みます。接続先のHDFS NameNodeは、IDが`CONN_ID`であるHadoop接続(p.192)で定義されます。この例のファイルURLでは、絶対HDFSパス`/path/filename.dat`のファイルが書き込まれます。

出力ポートへの書込み

- `port:$0.FieldName:discrete`

  このURLを使用する場合は、**ライター**の出力ポートを別のコンポーネントに接続する必要があります。出力メタデータには、データ型が`string`、`byte`または`cbyte`の`FieldName`が含まれている必要があります。**ライター**によって入力ポートを介して受信された各データ・レコードは、入力メタデータに従って処理され、オプションの出力ポートから送信されて、出力エッジのメタデータで指定されたフィールドの値として書き込まれます。次のレコードは、ここでの説明と同じ方法で解析されます。

コンソールへの書込み

- -

  データをstdoutに書き込みます。

書込みにおけるプロキシの使用方法

- `http:(direct:)//seznam.cz`

  プロキシを使用しません。

- `http:(proxy://user:password@212.93.193.82:443)//seznam.cz`

  httpプロトコルのプロキシ設定です。

- `ftp:(proxy://user:password@proxyserver:1234)//seznam.cz`

  ftpプロトコルのプロキシ設定です。

- `ftp:(proxy://proxyserver:443)//server/path/file.dat`

  ftpプロトコルのプロキシ設定です。

- `sftp:(proxy://66.11.122.193:443)//user:password@server/path/file.dat`

  sftpプロトコルのプロキシ設定です。

ディクショナリへの書込み

- `dict:keyName:source`

  ディクショナリで指定されているファイルURLにデータを書き込みます。ターゲット・ファイルURLは、指定されたディクショナリ・エントリから取得されます。

- `dict:keyName:discrete`[1)]

  データをディクショナリに書き込みます。`ArrayList<byte[]>`が作成されます。

- `dict:keyName:stream`[2)]

  データをディクショナリに書き込みます。`WritableByteChannel`が作成されます。

説明:

1): `discrete`処理タイプでは、バイト配列を使用してデータを格納します。

2): `stream`処理タイプでは、グラフの実行前(Javaコードから)に作成する必要がある、出力ストリームを使用します。

データ・ソースとしてのサンドボックス・リソース

サンドボックス・リソースは、それが共有、ローカルまたはパーティション化されたサンドボックスのいずれであるかに関係なく、グラフの「fileURL」属性で、次のようなサンドボックスURLとして指定します。

```
sandbox://data/path/to/file/file.dat
```

ここで、dataはサンドボックスのコード、path/to/file/file.datはサンドボックス・ルートからリソースへのパスです。URLはCloverETL Serverによってグラフの実行中に評価され、コンポーネント(リーダーまたはライター)は開かれたストリームをサーバーから取得します。これは、ローカル・ファイルまたは別のリモート・リソースへのストリームにできます。したがって、グラフはリソースにローカル・アクセスできるノード上で実行する必要はありません。

グラフでは複数のサンドボックス・リソースを使用でき、各リソースは異なるノード上に存在できます。このような場合、CloverETL Serverでは、リモート・ストリームを最小化するために、最も多くのローカル・リソースが存在するノードを選択します。

サンドボックスURLには、データのパラレル処理のための、特別な使用方法があります。パーティション化されたサンドボックス内のリソースのサンドボックスURLが使用される場合、この部分のグラフ/フェーズは、パーティション化されたサンドボックスのロケーション・リストによって指定されたノード割当てに従い、並行して実行されます。したがって、各ワーカーに、独自のローカル・サンドボックス・リソースがあります。CloverETL Serverは各ワーカー上のサンドボックスURLを評価して、ローカル・リソースへの開かれたストリームをコンポーネントに提供します。

ライター上のデータの表示

出力ファイルが作成された後、コンテキスト・メニューを使用して**ライター**上で出力ファイル・データを確認できます。このことを行うには、対象のコンポーネントを右クリックして、コンテキスト・メニューから「**View data**」を選択します。

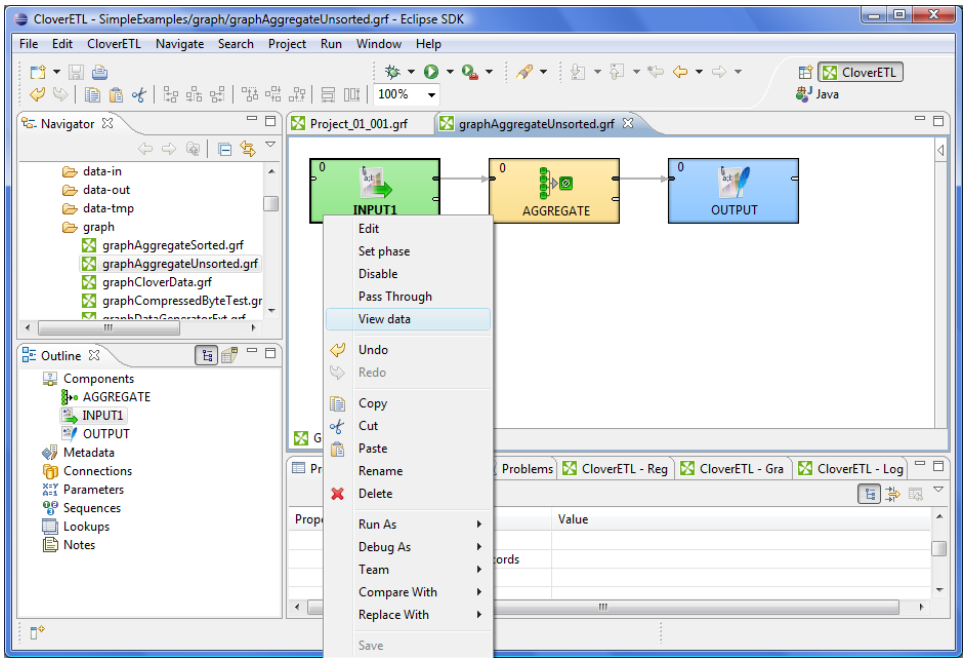


図44.1. コンポーネント上のデータの表示

ここで、データをプレーン・テキストで表示するかグリッドで表示するかを選択する必要があります。**プレーン・テキスト**・オプションを選択した場合、**キャラクタ・セット**を選択できますが、フィルタ式は選択できません。複数のコンポーネントのデータを同時に表示できます。結果を区別するために、ウィンドウ・タイトルには、表示しているエッジの情報が、GRAPH.name:COMPONENT.nameの形式で表示されます。

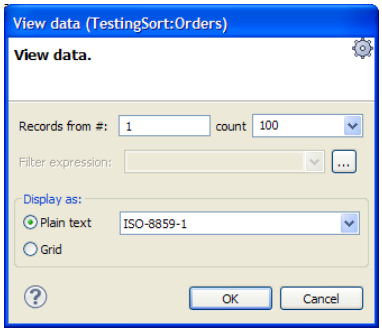


図44.2. プレーン・テキストとしてのデータの表示

一方、**グリッド・**オプションを選択した場合は、**フィルタ式**を選択できますが、**キャラクタ・セット**は選択できません。

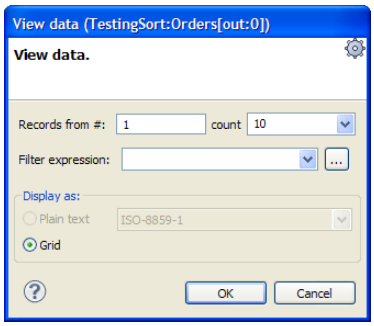


図44.3. データをグリッドとして表示

プレーン・テキスト・モードでは、結果は次のようになります。

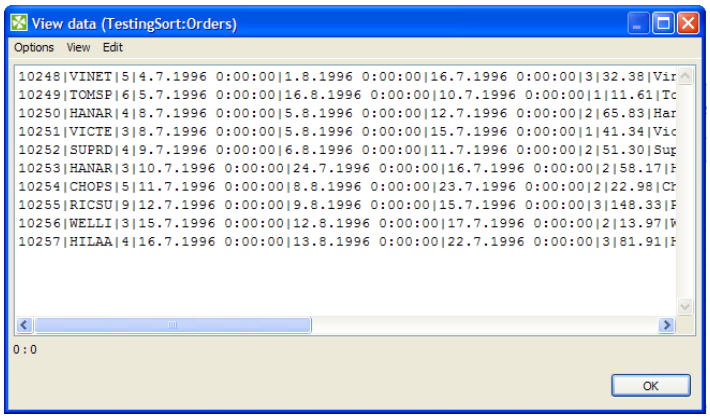


図44.4. プレーン・テキスト・データ表示

グリッド・モードでは、次のように表示されます。

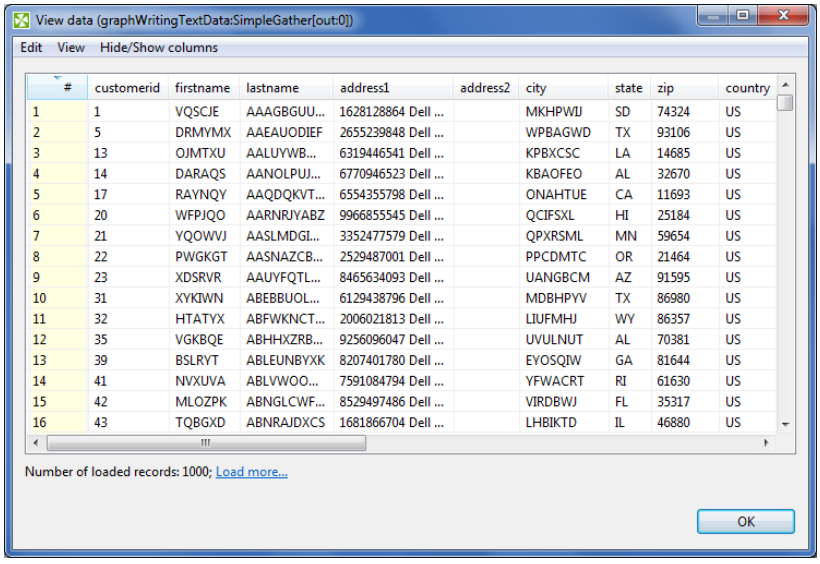


図44.5. グリッド・データ表示

注意

表示対象のデータが多すぎる場合は、ビューの下に「**Load more...**」という青色のテキストが表示されます。これをクリックすると、現在表示されているレコードの後に、新しいレコードのチャ

ンクが追加されます。**プレーン・ビュー**でも、ビューの最下部までスクロール・ダウンすると(または**[Page Down]**を押すと)、レコードがロードされます。

一部のリーダーでも、同じことを実行できます。リーダー上のデータの表示(p.301)を参照してください。

出力ポート書込み

一部の**ライター**では、オプションの出力ポートにデータを書き込むことができます。

出力ポート書込みが可能な**ライター**のリストを次に示します。

- **UniversalDataWriter**
- **XLSDataWriter**
- **XMLWriter**
- **StructuredDataWriter**

これらのコンポーネントの出力ポート書込みの属性は、URLファイル・ダイアログ(p.69)を使用して定義できます。

これらのいずれかの**ライター**のオプションの出力ポートをエッジに接続する場合、このエッジのもう一端を別のコンポーネントに接続する必要があります。このエッジのメタデータには、データ型が`string`、`byte`または`cbyte`である`FieldName`が指定されて含まれている必要があります。

次に、このような**ライター**の**「File URL」**属性を`port:$0.FieldName[:processingType]`に設定します。

ここで、`processingType`はオプションであり、`discrete`または`stream`のいずれかに設定できます。明示的に設定されてない場合、デフォルトで`discrete`となります。

グラフが実行されると、データは入力メタデータに従って読み取られ、指定された処理タイプに従って**ライター**により処理され、**ライター**のオプションの出力ポートを介して後続の別のコンポーネントに送信されます。

- `discrete`

  入力ポートを介して受信された各データ・レコードは、入力メタデータに従って処理され、オプションの出力ポートから送信されて、出力エッジのメタデータで指定されたフィールドの値として書き込まれます。次のレコードは、ここでの説明と同じ方法で解析されます。

- `stream`

  入力ポートを介して受信された各データ・レコードは`discrete`処理タイプの場合と同じ方法で処理されますが、出力の最後に`null`値を含む別のフィールドが追加されます。このような`null`値は、`stream`処理タイプを使用して複数のファイルが入力ポートから再び読み取られる場合に`eof`を意味します。`stream`処理タイプでの書込みの詳細は、入力ポートの読取り(p.303)を参照してください。

データの書込み方法および書込み先

ファイルURLを指定する場合は、次の属性の設定方法も決定する必要があります。

- **Append**

  既存のファイルにレコードを追加するか(**追加**)、そのファイルを置き換えるかを決定することは非常に重要です。この属性は、デフォルトで`false` (ファイルを追加せずに置き換える)に設定されています。

  ローカルの(リモートでない) zipアーカイブ内のファイルにデータを追加することもできます。この場合、サーバー環境では`use_local_context_url`を**`true`**に設定する必要があります。

  この属性は、次の**ライター**で使用可能です。

- **Trash (Debug追加属性)**
- **UniversalDataWriter**
- **CloverDataWriter**
- **XLSDataWriter (シート属性に対する追加)**
- **StructuredDataWriter**
- **XMLWriter**

- **Create directories**

  まだ存在しないディレクトリを「**File URL**」で指定する場合は、「**Create directories**」属性を`true`に設定する必要があります。このようなディレクトリが作成されます。設定しなかった場合はグラフが失敗します。「**Create directories**」のデフォルト値は`false`です。

  この属性は、次の**ライター**で使用可能です。

  - **Trash**
  - **UniversalDataWriter**
  - **CloverDataWriter**
  - **XLSDataWriter**
  - **StructuredDataWriter**
  - **XMLWriter**

- **Exclude fields**

  この属性を使用すると、一部のフィールドの値を書込みから除外できます。この属性はキー・ウィザードを使用して作成する必要があり、出力に書き込まないフィールドを指定する場合に使用します。その形式は、一連のフィールド名をセミコロンで区切るものとします。たとえば、「**Partition key**」を使用して複数のファイルに出力を分割する場合は、出力ファイルに値が書き込まれないフィールドとして同じフィールドを指定できます。

  - **UniversalDataWriter**
  - **XLSDataWriter**

出力レコードの選択

ライターを設定する場合は、書込み対象のレコードを制限できます。

次の**ライター**では、「**Number of skipped records**」属性または「**Max number of records**」属性(あるいはその両方)を使用して、書き込まれるレコードの数を制限できます。後述の説明における**マッピング**とは、入力ポートからマッピングされ、出力ファイルに書き込まれるサブツリーのことです。

- **UniversalDataWriter**
- **CloverDataWriter**
- **XLSDataWriter**
- **StructuredDataWriter**
- **XMLWriter (「Number of skipped mappings」属性および「Max number of mappings」属性)**

異なる出力ファイルへの出力のパーティション

3つのコンポーネントにより、受信データ・フローを分割し、複数の出力ファイルにレコードを分散できます。これらのコンポーネントは、**UniversalDataWriter**、**XLSDataWriter**および**StructuredDataWriter**です。

データ・フローを分割し、キー値に応じて異なる出力ファイルにレコードを書き込む場合は、そのような分割のためのキー(「**Partition key**」)を指定する必要があります。これは、一連の受信レコードのフィールド名をセミコロンで区切った形式をしています。

この他にも、一部の受信レコードのみを選択することもできます。これは、参照表(**パーティション参照表**)を使用して実現されます。「**Partition key**」が参照表のキー値に等しいレコードは、指定された出力ファイルに保存されます。キー値が参照表のキー値に等しくないレコードは、「**Partition unassigned file name**」属性に指定されたファイルに保存されるか、破棄されます(「**Partition unassigned file name**」が指定されていない場合)。

すべての受信レコードが参照表の値に割り当てられている場合、未割当てレコードのファイルは空になります(定義されている場合でも)。

このような参照表は、選択したデータ・レコードを複数の出力ファイルにグループ化し、それらに名前を付けるためにも使用します。「**Partition output fields**」属性が指定されている必要があります。これは、一連の参照表フィールドをセミコロンで区切ったものです。

「**File URL**」値は、出力ファイル名のベース名としての役割のみを果たします。このベース名は、識別名または番号を連結したものです。パーティションが指定されている場合(「**Partition key**」が定義されている場合)、「**File URL**」では識別名または番号用のプレースホルダとしてハッシュ記号を使用できます。このようなハッシュ記号は、「**File URL**」のファイル名部分でのみ使用する必要があります。

重要

ハッシュ記号とドル記号の使用方法は区別する必要があります。

- **ハッシュ記号**

  ハッシュ記号は、指定された「**Partition key**」の値に対応するレコードが、複数の出力ファイルのそれぞれに含まれている場合にのみ使用します。

- **ドル記号**

  ドル記号は、「**Records per file**」属性に基づいて、指定された数のレコードのみが複数の出力ファイルのそれぞれに含まれている場合に使用します。

ハッシュは、「**File URL**」のファイル部分の任意の位置(中間部分を含む)に配置できます。たとえば、`path/output#.xls`などとなります(出力XLSファイルの場合)。「**File URL**」にハッシュが含まれていない場合は、ファイルのベース名の末尾に識別名または番号が追加されます。

「**Partition file tag**」が`Number file tag`に設定されている場合は、出力ファイルが番号付けされ、「**File URL**」で使用されているハッシュの数がこれらの識別番号の桁数を表します。これは、「**Partition file tag**」のデフォルト値です。したがって、`###`の範囲は`000`から`999`です。

「**Partition file tag**」が`Key file tag`に設定されている場合、「**File URL**」で使用するハッシュは1つのみです。識別名が使用されます。

これらの識別名は、次のように作成されます。

「**Partition key**」属性(または「**Partition output fields**」属性)が`field1;field2;...;fieldN`という形式で、これらのフィールドの値が`valueofthefield1`、`valueofthefield2`...`valueofthefieldN`の場合は、フィールドのすべての値が文字列に変換されて連結されます。生成される文字列の形式は、`valueofthefield1valueofthefield2...valueofthefieldN`となります。生成された文字列は識

別名として使用され、「**File URL**」のハッシュでマークされた位置にそれぞれが挿入されます。また、「**File URL**」でハッシュが使用されていない場合は、「**File URL**」の末尾に追加されます。

たとえば、`firstname;lastname`が「**Partition key**」(または「**Partition output fields**」)の場合、出力ファイルは次のようになります。

- `path/outjohnsmith.xls`、`path/outmarksmith.xls`、`path/outmichaelgordon.xls`など(「**File URL**」が`path/out#.xls`、「**Partition file tag**」が`Key file tag`の場合)
- `path/out01.xls`、`path/out02.xls`など(「**File URL**」が`path/out##.xls`、「**Partition file tag**」が`Number file tag`に設定されている場合)

XLSDataWriterおよび**UniversalDataWriter**には、「**Exclude fields**」という別の属性があります。

これは、出力に書き込まれない一連のフィールド名をセミコロンで区切ったものです。同じフィールドが、「**Partition key**」の一部として機能する場合に使用できます。

これらの2つのライターのいずれかを使用してデータをパーティションし、「**Partition file tag**」を**Key file tag**に設定すると、「**Partition key**」の値はこれらのファイルに書き込まれます。同時に、対応する出力ファイルにも同じ値が書き込まれます。

名前に同じ値が含まれているファイルに書き込まれないようにするために、そのファイルへの書込み時に除外されるフィールドを選択できます。「**Exclude fields**」属性を選択する必要があります。

これらのフィールドはファイル名またはシート名にのみ含まれ、ファイルのコンテンツには書き込まれません。

その後に、これらのファイルを読み取ると、自動入力関数を使用して(**UniversalaDataReader**または**XLSDataReader**の`source_name`、あるいは**XLSDataReader**の`sheet_name`)、ファイル名またはシート名から値を取得できます(シート名があらかじめ`$<field name>`に設定してある場合)。

つまり、`City`に設定された「**Partition key**」を使用してファイルを作成し、出力ファイルが`London.txt`、`Stockholm.txt`などの場合は、これらの名前から値(`London`、`Stockholm`など)を取得できます。`City`フィールド値がこれらのファイルに含まれている必要はありません。

注意

既存のファイルへのパスとしてフィールドの値を使用する場合は、**ライター**の「**File URL**」属性として次を入力します。

```
//#
```

このとき、パーティションに使用するフィールドの値が`path/to/my/file/filename.txt`の場合、出力ファイルにはこの値が名前として割り当てられます。このため、出力ファイルは`path/to/my/file`に格納され、その名前は`filename.txt`となります。

ライター用Javaインタフェース

- JMSWriter(p.501)は、オプションで変換を実行できますが、これはJavaのみで記述できます。

  インタフェースの詳細は、JMSWriter用Javaインタフェース(p.503)を参照してください。

第45章 トランスフォーマの共通プロパティ

これらのコンポーネントには、入力ポートと出力ポートの両方があります。同じメタデータを持つ複数のデータ・フローの結合(**Concatenate**、**SimpleGather**および**Merge**)、重複レコードの削除(**Dedup**)、データ・レコードのフィルタ(**ExtFilter**および**EmailFilter**)、入力レコードからのサンプルの作成(**DataSampler**)、データ・レコードのソート(**ExtSort**、**FastSort**および**SortWithinGroups**)、既存のデータ・フローの複製(**SimpleCopy**)、1つのデータ・フローの複数データ・フローへの分割(すべての場合に**Partition**、オプションで**Dedup**、**ExtFilter**および**Reformat**も)、2つのデータ・フローの交差(入力のメタデータが異なる場合でも) (**DataIntersection**)、データ情報の集計(**Aggregate**)、およびより複雑なデータ・フロー変換の実行(**Reformat**、**Denormalizer**、**Pivot**、**Normalizer**、**MetaPivot**、**Rollup**および**XLSTransformer**)を実行できます。

メタデータは、これらのトランスフォーマのいくつかを使用して伝播できますが、より複雑な方法によるデータ・フローの変換はそれらのコンポーネントでは実現できません。これらのコンポーネントを構成する前に、出力メタデータを定義しておく必要があります。

このように一部のトランスフォーマでは、前述の変換が使用されます。変換の定義方法の詳細は、変換の定義(p.279)を参照してください。

- 一部の**トランスフォーマ**には変換属性を定義できます。オプションの場合と必須の場合があります。CTLで記述された変換の変換テンプレートの詳細は、次を参照してください。

  トランスフォーマ用CTLテンプレート(p.321)

- 一部の**トランスフォーマ**には変換属性を定義できます。オプションの場合と必須の場合があります。Javaで記述された変換に実装する必要がある変換インタフェースの詳細は、次を参照してください。

  トランスフォーマ用Javaインタフェース(p.322)

すべての**トランスフォーマ**の概要を次に示します。

表45.1. トランスフォーマの比較

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
SimpleCopy(p.647)	-	✖	1	1-n	-	-
ExtSort(p.600)	-	✖	1	1-n	-	-
FastSort(p.602)	-	✖	1	1-n	-	-
SortWithinGroups(p.649)	-	✔	1	1-n	-	-
Dedup(p.587)	-	✔	1	1-2	-	-
ExtFilter(p.597)	-	✖	1	1-2	-	-
Concatenate(p.581)	✔	✖	1-n	1	-	-
SimpleGather(p.648)	✔	✖	1-n	1	-	-
Merge(p.607)	✔	✔	2-n	1	-	-
Partition(p.619)	-	✖	1	1-n	はい/いいえ[1)]	はい/いいえ[1)]
LoadBalancingPartition(p.626)	-	✖	1	1-n	-	-
DataIntersection(p.582)	✖	✔	2	3	✔	✔
Aggregate(p.578)	-	✖	1	1	-	-

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Reformat(p.632)	-	✗	1	1-n	✓	✓
Denormalizer(p.589)	-	✗	1	1	✓	✓
Pivot(p.628)	-	✗	1	1	✓	✓
Normalizer(p.612)	-	✗	1	1	✓	✓
MetaPivot(p.609)	-	✗	1	1	-	-
Rollup(p.635)	-	✗	1	1-n	✓	✓
DataSampler(p.585)	-	✗	1	n	-	-
XSLTransformer(p.651)	-	✗	1	1	-	-

説明

1) **Partition**では、変換または他の2つの属性(「**Ranges**」または「**Partition key**」)のいずれかを使用できます。これらのどちらかが指定されない場合は、変換を定義する必要があります。

トランスフォーマ用CTLテンプレート

- Partition(p.619)は、「**Partition key**」または「**Ranges**」が定義されていない場合に変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  変換テンプレートの詳細は、Partition (およびclusterpartition)用Javaインタフェース(p.625)を参照してください。

  このコンポーネントは、変換によって返された値と番号が等しい接続済出力ポートを経由して各レコードを送信します(変換の戻り値(p.283))。マッピングの操作は不要です。レコードは自動的にマッピングされます。

- DataIntersection(p.582)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  変換テンプレートの詳細は、DataIntersection用CTLテンプレート(p.584)を参照してください。

- Reformat(p.632)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  変換テンプレートの詳細は、Reformat用CTLテンプレート(p.633)を参照してください。

  このコンポーネントは、変換によって返された値と番号が等しい接続済出力ポートを経由して各レコードを送信します(変換の戻り値(p.283))。このようなポートには、マッピングを定義する必要があります。

- Denormalizer(p.589)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  変換テンプレートの詳細は、Denormalizer用CTLテンプレート(p.591)を参照してください。

- Normalizer(p.612)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  変換テンプレートの詳細は、Normalizer用CTLテンプレート(p.613)を参照してください。

- Rollup(p.635)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  変換テンプレートの詳細は、Rollup用CTLテンプレート(p.637)を参照してください。

  このコンポーネントは、変換によって返された値と番号が等しい接続済出力ポートを経由して各レコードを送信します(変換の戻り値(p.283))。このようなポートには、マッピングを定義する必要があります。

トランスフォーマ用Javaインタフェース

- Partition(p.619)は、「**Partition key**」または「**Ranges**」が定義されていない場合に変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  インタフェースの詳細は、Partition (およびclusterpartition)用Javaインタフェース(p.625)を参照してください。

  このコンポーネントは、変換によって返された値と番号が等しい接続済出力ポートを経由して各レコードを送信します(変換の戻り値(p.283))。マッピングの操作は不要です。レコードは自動的にマッピングされます。

- DataIntersection(p.582)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  インタフェースの詳細は、DataIntersection用Javaインタフェース(p.584)を参照してください。

- Reformat(p.632)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  インタフェースの詳細は、Reformat用Javaインタフェース(p.634)を参照してください。

  このコンポーネントは、変換によって返された値と番号が等しい接続済出力ポートを経由して各レコードを送信します(変換の戻り値(p.283))。このようなポートには、マッピングを定義する必要があります。

- Denormalizer(p.589)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  インタフェースの詳細は、Denormalizer用Javaインタフェース(p.596)を参照してください。

- Normalizer(p.612)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  インタフェースの詳細は、Normalizer用Javaインタフェース(p.618)を参照してください。

- Rollup(p.635)には変換が必要です(CTLおよびJavaのどちらでも書込み可能)。

  インタフェースの詳細は、Rollup用Javaインタフェース(p.645)を参照してください。

  このコンポーネントは、変換によって返された値と番号が等しい接続済出力ポートを経由して各レコードを送信します(変換の戻り値(p.283))。このようなポートには、マッピングを定義する必要があります。

第46章 ジョイナの共通プロパティ

これらのコンポーネントには、入力ポートと出力ポートの両方があります。これらは、指定されたキーおよび指定された変換に従って、潜在的に異なるメタデータを持つレコードを結合するために使用します。

最初の入力ポートをマスター(ドライバ)、それ以外をスレーブと呼びます。

これらは、2つの入力ポート(**ApproximativeJoin**)または2つ以上の入力ポート(**ExtHashJoin**、**ExtMergeJoin**および**RelationalJoin**)を使用して、受信するレコードを結合できます。その他には、参照表からのレコードを含む単一の入力ポートを使用して受信するレコードを結合したり(**LookupJoin**)、データベース表からのレコードを結合できます(**DBJoin**)。これらでは、スレーブ・データ・レコードは2番目の仮想入力ポートを使用して受信するものとみなされます。

前述の**ApproximativeJoin**、**ExtMergeJoin**および**RelationalJoin**の3つのジョイナでは、受信データがソートされている必要があります。

他のすべてのジョイナと異なり、**RelationalJoin**は非同等性条件に基づいてデータ・レコードを結合します。他のすべてのジョイナでレコードを結合するには、レコードを結合する場合のキー・フィールドに同じ値が設定されている必要があります。

ApproximativeJoin、**DBJoin**および**LookupJoin**には、一致のないマスター・データ・レコード用のオプションの出力ポートがあります。**ApproximativeJoin**には、一致のないマスターおよびスレーブのデータ・レコード用のオプションの出力ポートがあります。

メタデータは、これらのコンポーネントを通じて伝播できません。最初に正しいメタデータを選択するか、必要な結果に応じて手動で作成する必要があります。変換は、その場合にのみ定義できます。一部の出力エッジの場合は、入力のメタデータも選択できますが、これらのメタデータもコンポーネントを介して伝播できません。

これらのコンポーネントは、トランスフォーマに関する項で説明している変換を使用します。変換の定義方法の詳細は、変換の定義(p.279)を参照してください。ジョイナでのすべての変換では、共通の変換テンプレート(ジョイナ用CTLテンプレート(p.325))および共通のJavaインタフェース(ジョイナ用Javaインタフェース(p.328))を使用します。

これらのオプションへのリンクの概要を次に示します。

- 結合タイプ(p.324)
- スレーブの重複(p.325)
- ジョイナ用CTLテンプレート(p.325)
- ジョイナ用Javaインタフェース(p.328)

すべての**ジョイナ**の概要を次に示します。

表46.1. ジョイナの比較

コンポーネント	同じ入力メタデータ	ソート済入力	スレーブ入力	出力	スレーブのないドライバの出力	ドライバのないスレーブの出力	同等性に基づく結合
ApproximativeJoin(p.654)	✖	✔	1	2-4	✔	✔	✔
ExtHashJoin(p.667)	✖	✖	1-n	1	✖	✖	✔
ExtMergeJoin(p.672)	✖	✔	1-n	1	✖	✖	✔

コンポーネント	同じ入力メタデータ	ソート済入力	スレーブ入力	出力	スレーブのないドライバの出力	ドライバのないスレーブの出力	同等性に基づく結合
LookupJoin(p.677)	✗	✗	1 (仮想)	1-2	✓	✗	✓
DBJoin(p.664)	✗	✗	1 (仮想)	1-2	✓	✗	✓
RelationalJoin(p.680)	✗	✓	1	1	✗	✗	✗

結合タイプ

これらのコンポーネントは、次の3つの処理モードで動作します。

- **内部結合**

  この処理モードでは、**結合キー**・フィールドの値がスレーブの対応する値に等しいマスター・レコードのみが処理され、結合済レコードの出力ポートを通じて送信されます。

  ApproximativeJoinの場合、この属性の名前は「**Matching key**」になります。

  一致しないマスター・レコードは、スレーブを持たないマスター・レコードのオプションの出力ポートを通じて送信できます(**ApproximativeJoin**、**LookupJoin**または**DBJoin**)。

  一致しないスレーブ・レコードは、マスターを持たないスレーブ・レコードのオプションの出力ポートを通じて送信できます(**ApproximativeJoin**のみ)。

- **左側外部結合**

  この処理モードでは、**結合キー**・フィールドの値がスレーブの対応する値に等しくないマスター・レコードのみが処理され、結合済レコードの出力ポートを通じて送信されます。

  ApproximativeJoinの場合、この属性の名前は「**Matching key**」になります。

  一致しないスレーブ・レコードは、マスターを持たないスレーブ・レコードのオプションの出力ポートを通じて送信できます(**ApproximativeJoin**のみ)。

- **完全外部結合**

  この処理モードでは、**結合キー**・フィールドの値がスレーブの対応する値に等しいかどうかに関係なく、マスターおよびスレーブのすべてのレコードが処理され、結合済レコードの出力ポートを通じて送信されます。

  ApproximativeJoinの場合、この属性の名前は「**Matching key**」になります。

重要

LookupJoinおよび**DBJoin**では**完全外部結合**モードは許可されません。

注意

ジョイナは、**結合キー**属性の同じフィールドに`null`値があるレコードを、その`null`が異なるものとしてレコードの各ペア(マスターおよびスレーブ)を解析します。したがって、それらのフィールドではレコードが互いに一致しないため、レコードは結合されません。

スレーブの重複

ジョイナでは、複数のスレーブ・レコードに、**結合キー**に対応するフィールドの同じ値が設定されている場合があります(**ApproximativeJoin**では「**Matching key**」)。このようなスレーブを重複と呼びます。重複スレーブ・レコードが許可されている場合は、すべてのレコードが解析され、一致するレコードがあるとマスター・レコードと結合されます。重複が許可されていない場合は、レコードの1つまたはレコードの一部のみが解析され(そのレコードがマスター・レコードに一致する場合)、その他は破棄されます。

ジョイナの違いによって、スレーブの重複を処理する方法も異なります。ここでは、このような重複の解析方法、および当該コンポーネントまたはその他のツールで設定可能な項目の概要について説明します。

- **ApproximativeJoin**

  重複するスレーブを含むすべてのレコード(「**Matching key**」と同じ値)が常に処理されます。

- 「**Allow slave duplicates**」属性は、次の**ジョイナ**に含まれています(`true`または`false`に設定できます)。

  - **ExtHashJoin**

    デフォルトは`false`です。**最初の**レコードのみが処理され、その他は破棄されます。

  - **ExtMergeJoin**

    デフォルトは`true`です。`false`に切り替えると**最後の**レコードのみが処理され、その他は破棄されます。

  - **RelationalJoin**

    デフォルトは`false`です。**最初の**レコードのみが処理され、その他は破棄されます。

- **SQL問合せ**属性は**DBJoin**に含まれています。**SQL問合せ**を使用すると、スレーブの重複の正確な数を明示的に指定できます。

- **LookupJoin**は、使用中の参照表の設定に従って、スレーブの重複を次のように解析します。

  - **単純な参照表**には、**キーの重複を許可**属性も含まれています。デフォルト値は`true`です。このチェック・ボックスの選択を解除すると**最後の**レコードのみが処理され、その他は破棄されます。

  - **DB参照表**を使用すると、スレーブの重複の正確な数を明示的に指定できます。

  - **範囲参照表**でスレーブの重複は許可されません。**最初の**スレーブ・レコードのみが使用され、その他は破棄されます。

  - **永続参照表**でスレーブの重複は許可されません。ただし、これには**置換**属性があります。デフォルトでは、新しいスレーブ・レコードによって古いレコードが上書きされ、古いレコードは破棄されます。デフォルトでは、**最後の**スレーブ・レコードが維持され、その他は破棄されます。このチェック・ボックスの選択を解除すると**最初の**レコードが維持され、その他は破棄されます。

  - **Aspell参照表**では、すべてのスレーブの重複の使用が許可されます。重複の数は制限できません。

ジョイナ用CTLテンプレート

この変換テンプレートは、すべての**ジョイナ**および**Reformat**と**DataIntersection**で使用されます。

CTLで変換を定義するための「**Source**」タブの表示例を次に示します。

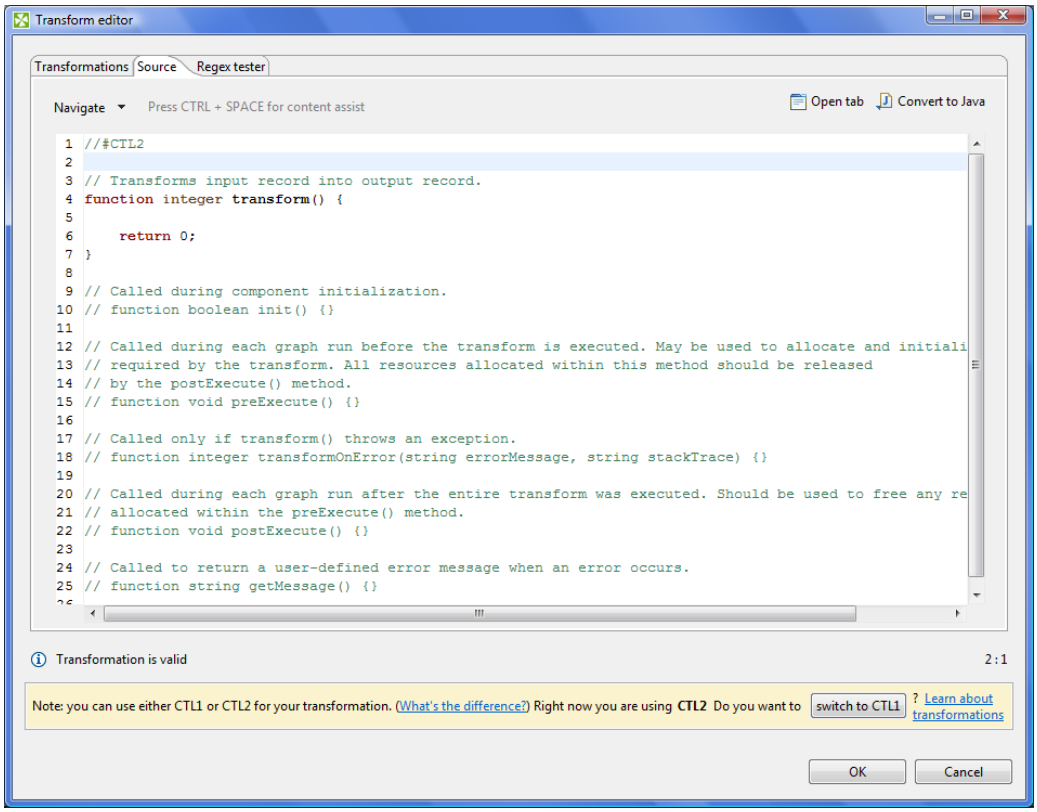


図46.1. ジョイナの変換エディタの「Source」タブ

表46.2. ジョイナ、DataIntersectionおよびReformatの機能

CTLテンプレート関数	
boolean init()	
必須	いいえ
説明	コンポーネントを初期化し、環境およびグローバル変数を設定します。
起動	1つ目のレコードを処理する前に呼び出されます。
戻り値	`true` | `false` (`false`グラフの失敗時)
integer transform()	
必須	はい
入力パラメータ	なし
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	結合済または交差済入力レコードのセットごと(**ジョイナ**および**DataIntersection**)、および入力レコードごと(**Reformat**)に繰り返し呼び出されます

CTLテンプレート関数	
説明	スクリプトを使用して、入力フィールドを出力フィールドにマップできます。特定の出力レコードに対するtransform()関数の一部が原因でtransform()関数が失敗し、ユーザーが別の関数(transformOnError())を定義している場合は、transform()が失敗した場所で、このtransformOnError()に処理が引き継がれます。transform()が失敗し、ユーザーがtransformOnError()を定義していない場合は、グラフ全体が失敗します。transformOnError()関数は、以前に処理が成功したコードから取得した、transform()によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースがtransformOnError()に渡されます。

例	`function integer transform() {`
	`  $0.name = $0.name;`
	`  $0.address = $city + $0.street + $0.zip;`
	`  $0.country = toUpper($0.country);`
	`  return ALL;`
	`}`

integer transformOnError(string errorMessage, string stackTrace, integer idx)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	transform()が例外をスローすると呼び出されます。
説明	出力レコードを作成します。特定の出力レコードに対するtransform()関数の一部が原因でtransform()関数が失敗し、ユーザーが別の関数(transformOnError())を定義している場合は、transform()が失敗した場所で、このtransformOnError()に処理が引き継がれます。transform()が失敗し、ユーザーがtransformOnError()を定義していない場合は、グラフ全体が失敗します。transformOnError()関数は、以前に処理が成功したコードから取得した、transform()によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースがtransformOnError()に渡されます。

例	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  $0.name = $0.name;`
	`  $0.address = $city + $0.street + $0.zip;`
	`  $0.country = "country was empty";`
	`  printErr(stackTrace);`
	`  return ALL;`
	`}`

string getMessage()	
必須	いいえ
説明	指定されたエラー・メッセージを出力し、ユーザーにより呼び出されます。

CTLテンプレート関数	
起動	ユーザーにより任意の時点で呼び出されます(`transform()`が-2以下の値を返した場合にのみ呼び出されます)。
戻り値	`string`
void preExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	変換で必要なリソースの割当てと初期化に使用できます。この関数内に割り当てられたすべてのリソースは、`postExecute()`関数によって解放される必要があります。
起動	変換が実行される前の各グラフの実行中に呼び出されます。
void postExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	`preExecute()`関数内に割り当てられたすべてのリソースを解放するために使用する必要があります。
起動	変換全体が実行された後の各グラフの実行中に呼び出されます。

重要

- **入力レコードまたはフィールド、および出力レコードまたはフィールド**

  入力および出力には、`transform()`関数および`transformOnError()`関数内からのみアクセスできます。

- その他すべてのCTLテンプレート関数では入力にも出力にもアクセスできません。

警告

これらのルールに従わない場合はNPEがスローされます。

ジョイナ用Javaインタフェース

これは、すべての**ジョイナ**および**Reformat**と**DataIntersection**で使用されます。

この変換は、`RecordTransform`インタフェースのメソッドを実装し、`Transform`インタフェースからその他の共通メソッドを継承しています。共通Javaインタフェース(p.295)を参照してください。

`RecordTransform`インタフェースのメソッドを次に示します。

- `boolean init(Properties parameters, DataRecordMetadata[] sourcesMetadata, DataRecordMetadata[] targetMetadata)`

  再フォーマット用クラスまたは関数を初期化します。このメソッドは、変換処理の開始時に1回のみ呼び出されます。すべてのオブジェクト割当ておよび初期化はここで実行する必要があります。

- `int transform(DataRecord[] sources, DataRecord[] target)`

  ソース・レコードをターゲット・レコードに再フォーマットします。このメソッドは、レコードの変換フローの1手順として呼び出されます。戻り値およびその意味の詳細は、変換の戻り値(p.283)を参照してください。

- `int transformOnError(Exception exception, DataRecord[] sources, DataRecord[] target)`

  ソース・レコードをターゲット・レコードに再フォーマットします。このメソッドは、レコードの変換フローの1手順として呼び出されます。戻り値およびその意味の詳細は、変換の戻り値(p.283)を参照してください。
  `transform(DataRecord[], DataRecord[])`が例外をスローした場合にのみ呼び出されます。

- `void signal(Object signalObject)`

  外部で何かが発生したことを変換に通知するために使用できるメソッド。(たとえば、集計コンポーネントでのキー変更など。)

- `Object getSemiResult()`

  変換から中間結果を取得するために使用できるメソッド。実装される場合と実装されない場合があります。

第47章 クラスタ・コンポーネントの共通プロパティ

これらのコンポーネントは、CloverETL Cluster環境でのデータ・フロー管理専用です。

第59章「クラスタ・コンポーネント」(p.750)の概要を次に示します。

表47.1. クラスタ・コンポーネントの比較

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
ClusterPartition(p.751)	✔	✖	1	1[1)]	はい/いいえ[2)]	はい/いいえ[2)]
ClusterLoadBalancingPartition(p.753)	✔	✖	1	1[1)]	いいえ	いいえ
ClusterSimpleGather(p.757)	✔	✖	1[3)]	1	いいえ	いいえ
ClusterMerge(p.759)	✔	✔	1[3)]	1	いいえ	いいえ
ClusterRepartition(p.761)	✔	✖	1[3)]	1[1)]	はい/いいえ[2)]	はい/いいえ[2)]

説明

1) 1つの出力ポートが複数の仮想出力ポートを表します。

2) **ClusterPartition**および**ClusterRepartition**では、変換または他の2つの属性(「**Ranges**」または「**Partition key**」)のいずれかを使用できます。これらのどちらかが指定されない場合は、変換を定義する必要があります。

3) 1つの入力ポートが複数の仮想入力ポートを表します。

第48章「その他」の共通プロパティ

これらのコンポーネントは、これまでに説明していないタスクを実行するために使用します。ここではこれらについて説明します。これらは異種グループであるため、共通のプロパティはありません。

JavaExecute(p.791)のみ、このコンポーネントに変換が定義されている必要があります。次のインタフェースを実装する必要があります。

JavaExecute用Javaインタフェース(p.792)

すべての**「その他」**の概要を次に示します。

表48.1.「その他」の比較

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1]	Java	CTL
SystemExecute(p.803)	-	✗	0-1	0-1	-	✗	✗
JavaExecute(p.791)	-	-	0	0	-	✓	✗
DBExecute(p.782)	-	✗	0-1	0-2	-	✗	✗
RunGraph(p.795)	-	✗	0-1	1-2	-	✗	✗
HTTPConnector(p.786)	-	✗	0-1	0-1	-	✗	✗
WebServiceClient(p.806)	-	✗	0-1	0-N	いいえ[2]	✗	✗
CheckForeignKey(p.779)	✗	✗	2	1-2	-	✗	✗
SequenceChecker(p.799)	-	✗	1	1-n	✓	✗	✗
LookupTableReaderWriter(p.793)	-	✗	0-1	0-n	✓	✗	✗
SpeedLimiter(p.801)	-	✗	1	1-n	✓	✗	✗

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、処理済データ・レコードをマッピングで指定された接続済出力ポートに送信します。

第61章「その他」(p.778)を参照してください。

第49章 データ品質の共通プロパティ

データ品質は、データ品質に関連する様々なタスクを実行するコンポーネントのグループで、データに関する情報の決定、問題の検出や修正などを行います。これらのコンポーネントは様々なタスクを実行するため、共通のプロパティがありません。

すべての**データ品質**コンポーネントの概要を次に示します。

表49.1. データ品質の比較

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1)]	Java	CTL
Address Doctor 5(p.764)	-	✗	1	1-2	-	-	
EmailFilter(p.768)	-	✗	1	0-2	-	-	
ProfilerProbe(p.773)	-	✗	1	1-n	✓	✗	✓

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

第60章「データ品質」(p.763)を参照してください。

第50章 ジョブ制御の共通プロパティ

ジョブ制御は、ETLグラフ、ジョブフローおよび解釈済スクリプトの実行、監視および中断(オプション)など、様々なジョブのタイプを管理するコンポーネントのグループです。これらのコンポーネントのほとんどは、ジョブフロー(p.250)と密接に関連しています。

すべての実行コンポーネント(**ExecuteGraph**、**ExecuteJobflow**、**ExecuteProfilerJob**、**ExecuteScript**)およびその他のいくつかのジョブ制御コンポーネントでは、ジョブ実行管理にほぼ同じアプローチを採用しています。それぞれのコンポーネントには、オプションの入力ポートがあります。このポートから受信する各トークンは実行コンポーネントによって解釈され、それぞれのジョブが開始されます。デフォルトの実行設定は、各種のコンポーネント属性で直接指定されます。このデフォルト設定は、受信トークンの値でオーバーライドできます。オーバーライドは、**入力マッピング**属性で指定します。成功したジョブの結果は最初の出力ポートに送信され、失敗したジョブの実行は2番目の出力ポートに送信されます。これらの出力トークンのコンテンツは、**出力マッピング**および**エラー・マッピング**に定義されています。

入力ポートが接続されていない場合は、コンポーネントの属性で実行設定が直接指定された状態で1つのジョブのみが開始されます。最初の出力ポートが接続されていない場合、ジョブの結果はログに出力されます。最後に、2番目の出力ポートが接続されていない場合は、最初に失敗したジョブによってジョブフロー全体が失敗します。

「**Redirect error output**」属性は、成功したジョブの結果と失敗したジョブの結果すべてを最初の出力ポートに送信するために使用できます。すべてのジョブ実行に対して、**出力マッピング**が使用されます。

すべての**ジョブ制御**コンポーネントの概要を次に示します。

表50.1. ジョブ制御の比較

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL	
Barrier(p.685)	✖	✖	1-n	1-n	-	-	
Condition(p.688)	-	✖	1	1-2	-	-	
ExecuteGraph(p.690)	-	✖	0-1	0-2	✖	✔	
ExecuteJobflow(p.697)	-	✖	0-1	0-2	✖	✔	
ExecuteProfilerJob(p.709)	-	✖	0-1	0-2	✖	✔	
ExecuteScript(p.712)	-	✖	0-1	0-2	✖	✔	
Fail(p.717)	-	✖	0-1	0	✖	✔	
GetJobInput(p.719)	-	✖	0	1	✖	✔	
KillGraph(p.721)	-	✖	0-1	0-1	✖	✔	
KillJobflow(p.724)	-	✖	0-1	0-1	✖	✔	
MonitorGraph(p.725)	-	✖	0-1	0-2	✖	✔	
MonitorJobflow(p.728)	-	✖	0-1	0-2	✖	✔	
SetJobOutput(p.729)	-	✖	1	0	✖	✔	
Success(p.731)	✖	✖	0-n	0	-	-	
TokenGather(p.733)	✖	✖	1-n	1-n	-	-	

第57章「ジョブ制御」(p.684)を参照してください。

第51章 ファイル操作の共通プロパティ

ファイル操作コンポーネントは、ファイルおよびディレクトリを操作します。

すべての**ファイル操作**コンポーネントの概要を次に示します。

表51.1. ファイル操作の比較

コンポーネント	入力	出力
CopyFiles(p.735)	0-1	0-2
CreateFiles(p.738)	0-1	0-2
DeleteFiles(p.741)	0-1	0-2
MoveFiles(p.747)	0-1	0-2
ListFiles(p.744)	0-1	1-2

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

ファイル操作コンポーネントの共通属性

ファイル操作にサポートされているURL形式の概要は、ファイル操作にサポートされているURL形式(p.335)を参照してください。

属性	必須	説明	可能な値
Input mapping	1)	コンポーネント属性への入力レコードのマッピングを定義します。	
Output mapping	1)	標準出力ポートへの結果のマッピングを定義します。	
Error mapping	1)	エラー出力ポートへのエラーのマッピングを定義します。	
Redirect error output	いいえ	有効な場合、エラーはエラー・ポートではなく出力ポートに送信されます。	false (デフォルト) | true

説明

1) マッピングを省略すると、同じ名前に基づいてデフォルト・マッピングが使用されます。

入力マッピング

操作は入力レコードごとに実行されます。入力エッジが接続されていない場合、操作は1回のみ実行されます。

コンポーネントの属性は、**入力マッピング**で指定されたとおり、入力ポートから読み取られた値によって上書きされる可能性があります。

出力マッピング

出力マッピングおよび**エラー・マッピング**・エディタの左側にあるレコードの意味を理解することが不可欠です。1つまたは2つのレコードが表示されている場合があります。

最初のレコードは入力エッジから読み取られた実際の入力レコードであるため、このレコードはコンポーネントに入力エッジが接続されている場合にのみ表示されます。このレコードには、**「Type」**列に表示される**ポート0**があります。

左側にある**「Result」**という名前の他のレコードは常に表示されますが、これはコンポーネントによって生成された結果レコードです。

エラー処理

デフォルトでは、コンポーネントが操作の実行に失敗すると、グラフが失敗します。エラー・ポートを接続すると、これを防止できます。エラー・ポートが接続されている場合、エラーはエラー・ポートに送信され、コンポーネントは継続されます。**「Redirect error output」**オプションが有効化されている場合は、標準出力ポートをエラー処理に使用することもできます。

エラーが発生した場合、**「Stop processing on fail」**オプションが無効化されていないかぎり、コンポーネントは以降の操作を実行しません。スキップした操作に関する情報は、エラー出力ポートに送信されます。

第58章「ファイル操作」(p.734)を参照してください。

ファイル操作にサポートされているURL形式

URL属性は、URLファイル・ダイアログ(p.69)を使用して定義できます。

特に明記しないかぎり、**ファイル操作**コンポーネントのURL属性は、セミコロン(;)で区切られた複数のURLを受け入れます。

重要

グラフの移植性を確保するため、URL内のパスを定義する場合にはフォワード・スラッシュを使用する必要があります(Microsoft Windowsの場合でも)。

ほとんどのプロトコルはワイルドカードをサポートします。? (疑問符)は任意の1文字に一致し、* (アスタリスク)は任意の数の任意の文字に一致します。ワイルドカードのサポートおよびその構文はプロトコルによって異なります。

ファイル操作に指定可能なURLの例を次に示します。

ローカル・ファイル

- `/path/filename.txt`

  指定された1つのファイル。

- `/path1/filename1.txt;/path2/filename2.txt`

  指定された2つのファイル。

- `/path/filename?.txt`

  マスクを満たすすべてのファイル。

- `/path/*`

  指定されたディレクトリ内のすべてのファイル。

- `/path?/*.txt`

  `path?`マスクを満たすディレクトリ内のすべての`.txt`ファイル。

リモート・ファイル

- `ftp://username:password@server/path/filename.txt`

  ユーザー名およびパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバー上の`filename.txt`ファイルを指します。

- `ftp://username:password@server/dir/*.txt`

  ユーザー名およびパスワードを使用し、ftpプロトコルを介して接続したリモート・サーバー上にある、マスクを満たすすべてのファイルを指します。

- `sftp://username:password@server/path/filename.txt`

  ユーザー名およびパスワードを使用し、sftpプロトコルを介して接続したリモート・サーバー上の`filename.txt`ファイルを指します。

- `sftp://username:password@server/path?/filename.txt`

  ユーザー名およびパスワードを使用し、sftpプロトコルを介して接続したリモート・サーバー上にある、マスクを満たすディレクトリ内のすべてのファイル`filename.txt`を指します。

- `http://server/path/filename.txt`

  httpプロトコルを介して接続したリモート・サーバー上の`filename.txt`ファイルを指します。

- `https://server/path/filename.txt`

  httpsプロトコルを介して接続したリモート・サーバー上の`filename.txt`ファイルを指します。

- `hdfs://CONNECTION_ID/path/filename.txt`

  Hadoop HDFS上の`filename.txt`ファイルを指します。`CONNECTION_ID`は、グラフに定義されたHadoop接続のIDを示しています。

サンドボックス・リソース

サンドボックス・リソースは、それが共有、ローカルまたはパーティション化されたサンドボックスのいずれであるかに関係なく、グラフの「fileURL」属性で、次のようなサンドボックスURLとして指定します。

```
sandbox://data/path/to/file/file.dat
```

ここで、dataはサンドボックスのコード、path/to/file/file.datはサンドボックス・ルートからリソースへのパスです。グラフは、リソースにローカル・アクセスできるノードで実行する必要はありません。

第52章 カスタム・コンポーネント

デフォルトで**CloverETL**によって提供されるコンポーネントの他にも、独自のコンポーネントを記述できます。詳細な手順は、ドキュメント・ページの**カスタム・コンポーネントの作成**に関する項を参照してください。

第VIII部 コンポーネント・リファレンス

第53章 リーダー

コンポーネントとは何かを理解していることを想定しています。概要は、第19章「コンポーネント」(p.97)を参照してください。

グラフ内の一部のコンポーネントのみが初期ノードとなります。これらを**リーダー**と呼びます。

リーダーは、入力ファイル(ローカルおよびリモート)からデータを読み取ったり、オプションの接続済入力ポートからデータを受信したり、ディクショナリからデータを読み取ることができます。データを生成するコンポーネントは1つのみです。これは初期ノードでもあるため、ここではそのコンポーネントについて説明します。

コンポーネントには様々なプロパティを設定できます。ただし、一部のものが共通する場合があります。すべてのコンポーネントに共通のプロパティもあれば、ほとんどのコンポーネントに共通のプロパティもあり、**リーダー**のみに共通のプロパティもあります。次のことを学習している必要があります。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

リーダーは、読取り可能なデータに従って区別できます。

- データを生成するコンポーネントは1つのみです。
  - DataGenerator(p.351)はデータを生成します。

その他の**リーダー**はファイルからデータを読み取ります。

- フラット・ファイル:
  - UniversalDataReader(p.415)は、フラット・ファイル(デリミタ付きまたは固定長)からデータを読み取ります。
  - ParallelReader(p.396)は、より多くのスレッドを使用して、デリミタ付きフラット・ファイルからデータを読み取ります。
  - ComplexDataReader(p.343)では、構造が不均一であるか、相互依存しているために整っていないフラット・ファイルから、簡単なGUIを使用してデータを読み取ります。
  - MultiLevelReader(p.392)は、不均一な構造を持つフラット・ファイルからデータを読み取ります。
- その他のファイル:
  - CloverDataReader(p.341)は、Cloverバイナリ形式のファイルからデータを読み取ります。
  - SpreadsheetDataReader(p.404)は、XLSファイルまたはXLSXファイルからデータを読み取ります。
  - XLSDataReader(p.421)は、XLSファイルまたはXLSXファイルからデータを読み取ります。
  - DBFDataReader(p.359)は、dBaseファイルからデータを読み取ります。
  - XMLExtract(p.426)は、SAXテクノロジを使用してXMLファイルからデータを読み取ります。
  - XMLXPathReader(p.452)は、XPath問合せを使用してXMLファイルからデータを読み取ります。
  - HadoopReader(p.375)は、Hadoopシーケンス・ファイルからデータを読み取ります。

その他の**リーダー**はデータベースからデータをアンロードします。

- データベース:
  - DBInputTable(p.361)は、JDBCドライバを使用してデータベースからデータをアンロードします。
  - QuickBaseRecordWriter(p.530)は、**QuickBase**オンライン・データベースからデータを読み取ります。
  - QuickBaseImportCSV(p.528)は、問合せを使用して**QuickBase**オンライン・データベースからデータを読み取ります。
  - LotusReader(p.390)は、**Lotus Notes**または**Lotus Domino**データベースからデータを読み取ります。

その他の**リーダー**はJMSメッセージを受信するか、またはディレクトリ構造を読み取ります。

- JMSメッセージ:
  - JMSReader(p.377)は、JMSメッセージをデータ・レコードに変換します。
- ディレクトリ構造:
  - LDAPReader(p.387)は、ディレクトリ構造をデータ・レコードに変換します。
- 電子メール・メッセージ:
  - EmailReader(p.365)は、電子メール・メッセージを読み取ります。

CloverDataReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

CloverDataReaderは、内部のバイナリCloverデータ形式ファイルに保存されているデータを読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1)]	各出力に送信[2)]	変換	変換が必要	Java	CTL
CloverDataReader	cloverバイナリ・ファイル	0	1-n	はい	いいえ	いいえ	いいえ	いいえ	いいえ

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します。詳細は、変換の戻り値(p.283)を参照してください。

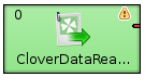
概要

CloverDataReaderは、内部のバイナリCloverデータ形式ファイルに保存されているデータを読み取ります。圧縮ファイル、コンソールまたはディクショナリからもデータを読み取ることができます。

注意

CloverETLのバージョン2.9以上では、**CloverDataWriter**は出力ファイルにバージョン番号を含むヘッダーも書き込みます。このため、**CloverDataReader**は、Cloverバイナリ形式のファイルにこのようなバージョン番号付きのヘッダーが含まれることを予期します。
CloverDataReader 2.9で、より古いバージョンの**CloverETL**によって書き込まれたファイルを読み取ることはできず、また、これらのより古いバージョンで、**CloverDataWriter** 2.9によって書き込まれたデータを読み取ることはできません。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	はい	正しいデータ・レコード用。	任意[1)]
	1-n	いいえ	正しいデータ・レコード用。	出力0

説明:

1): メタデータは自動入力関数(p.132)を使用できます。

CloverDataReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	読み取るデータ・ソースを指定する属性(フラット・ファイル、コンソール、入力ポート、ディクショナリ)。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Index file URL[1)]		索引ファイルの名前(パスを含む)。リーダーにサポートされているファイルURL形式(p.297)を参照してください。また、索引ファイル名の詳細は、出力ファイル構造(p.462)も参照してください。	
Advanced			
Number of skipped records		スキップするレコード数。入力レコードの選択(p.305)を参照してください。	0-N
Max number of records		読み取る最大レコード数。入力レコードの選択(p.305)を参照してください。	0-N
Deprecated			
Start record		すでに読み取られた最初のレコードの直前のレコード(そのレコードは含まれません)。「**Number of skipped records**」よりも優先度は低くなります。	0 (デフォルト) | 1-n
Final record		最後に読み取るレコード(そのレコードが含まれます)。「**Max number of records**」よりも優先度は低くなります。	all (デフォルト) | 1-n

説明:

1) これは**非推奨**の属性です。これを指定しない場合、すべてのレコードを読み取る必要があります。

ComplexDataReader

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適切なリーダーを見つけるには、リーダーの比較(p.297)を参照してください。

要約

ComplexDataReaderは、ファイルから不均一なデータを読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信	各出力に送信	変換	変換が必要	Java	CTL
ComplexDataReader	フラット・ファイル	1	1-n	いいえ	はい	はい	はい	はい	はい

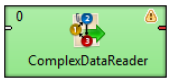
概要

ComplexDataReaderは、状態と遷移、およびオプションの先読み(セレクタ)の概念を使用して、複数のメタデータを含むファイルから不均一なデータを読み取ります。

ユーザー定義の状態およびその遷移により、ファイルの解析に使用するメタデータの順序が決定し、ほとんどの場合はファイルの構造に従います。

このコンポーネントは、データ・ポリシー(p.306)で説明する「**Data policy**」属性を使用します。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	ポート読取りに使用。入力ポートからの読取り(p.300)を参照してください。	1つのフィールド(byte、cbyte、string)。
出力	0	はい	正しいデータ・レコード用。	任意(Out0)[1)]
	1-N	いいえ	正しいデータ・レコード用。	任意(Out1からOutN)

説明:

1): 出力ポート上のメタデータは、自動入力関数(p.132)を使用できます。注意: source_timestamp関数およびsource_size関数は、ファイルから直接読み取る場合のみ機能します(ファイルがアーカイブである場合またはリモートの場所に保存されている場合は、タイムスタンプが空になり、サイズは0になります)。

ComplexDataReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	**ComplexDataReader**が読み取るデータ・ソース。ソースには、フラット・ファイル、コンソール、入力ポートまたはディクショナリを指定できます。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Transform		読取りを実行するステート・マシンの定義。設定ダイアログは、詳細説明(p.345)で説明する別ウィンドウで開きます。	
Charset		読み取られたレコードのエンコーディング。	ISO-8859-1 (デフォルト) | <any encoding>
Data policy		エラーの発生時に実行される処理を指定します。詳細は、データ・ポリシー(p.306)を参照してください。他のリーダーとは異なり、Controlledの**データ・ポリシー**は実装されていません。Lenientを使用すると、レコード・デリミタを含む(ただし正しくない行ではない)メタデータ内の冗長な列をスキップできます。	Strict (デフォルト) | Lenient
Trim strings		文字列をデータ・フィールドに挿入する前に、先頭および末尾のスペースを削除するかどうかを指定します。データのトリミング(p.418)を参照してください。	false (デフォルト) | true
Quoted strings		特殊文字(カンマ、改行または二重引用符)が含まれるフィールドは、引用符で囲む必要があります。一重引用符および二重引用符のみが引用符文字として受け入れられます。trueの場合、コンポーネントによる読取り時に特殊文字が削除されます(デリミタとして扱われません)。 例: 入力データ"25"|"John"を読み取るには、「**Quoted strings**」をtrueに切り替えて「**Quote character**」を「"」に設定します。これにより、25|Johnという2つのフィールドが生成されます。 デフォルトでは、この属性の値は出力ポート0のメタデータから継承されます。レコード詳細(p.162)も参照してください。	false | true

属性	必須	説明	可能な値
Quote character		「**Quoted strings**」で許可される引用符の種類を指定します。デフォルトでは、この属性の値は出力ポート0のメタデータから継承されます。レコード詳細(p.162)も参照してください。	both | " | '
Advanced			
Skip leading blanks		入力文字列をデータ・フィールドに挿入する前に、先頭のスペース文字(空白など)をスキップするかどうかを指定します。デフォルトのままにした場合は、「**Trim strings**」の値が使用されます。データのトリミング(p.418)を参照してください。	false (デフォルト) | true
Skip trailing blanks		入力文字列をデータ・フィールドに挿入する前に、末尾のスペース文字(空白など)をスキップするかどうかを指定します。デフォルトのままにした場合は、「**Trim strings**」の値が使用されます。データのトリミング(p.418)を参照してください。	false (デフォルト) | true
Max error count		入力で許容されるエラー・レコードの最大数。この属性は、**データ・ポリシー**で`Controlled`が使用されている場合にのみ適用可能です。	0 (デフォルト) - N
Treat multiple delimiters as one		この属性を`true`に設定すると、フィールドが複数のデリミタ文字で区切られている場合に単一のデリミタとして解釈されます。	false (デフォルト) | true
Verbose		デフォルトでは、エラー通知が簡略化され、パフォーマンスがかなり高くなります。`true`に切り替えた場合は、情報が詳細になり、パフォーマンスが低下します。	false (デフォルト) | true
Selector code	1)	セレクタを使用する場合は、ここにJavaのコードを記述します。セレクタは、変換における唯一のオプション機能です。データ・ファイルを先読みする必要がある場合の意思決定をサポートします。セレクタ(p.349)を参照してください。	
Selector URL	1)	Javaで記述されたセレクタ・コードを含む外部ファイルの名前およびパス。セレクタの詳細は、詳細説明(p.345)を参照してください。	
Selector class	1)	セレクタを含む外部クラスの名前。セレクタの詳細は、詳細説明(p.345)を参照してください。	
Transform URL		ステート・マシンの状態遷移を定義する外部ファイルへのパス。	
Transform class		ステート・マシンの状態遷移を定義するJavaクラスの名前。	
Selector properties		「**State transitions**」ウィンドウ内の現在のセレクタを簡単に編集できます。	
State metadata		「**State transitions**」ウィンドウ内でメタデータおよびメタデータに割当て済の状態を簡単に編集できます。	

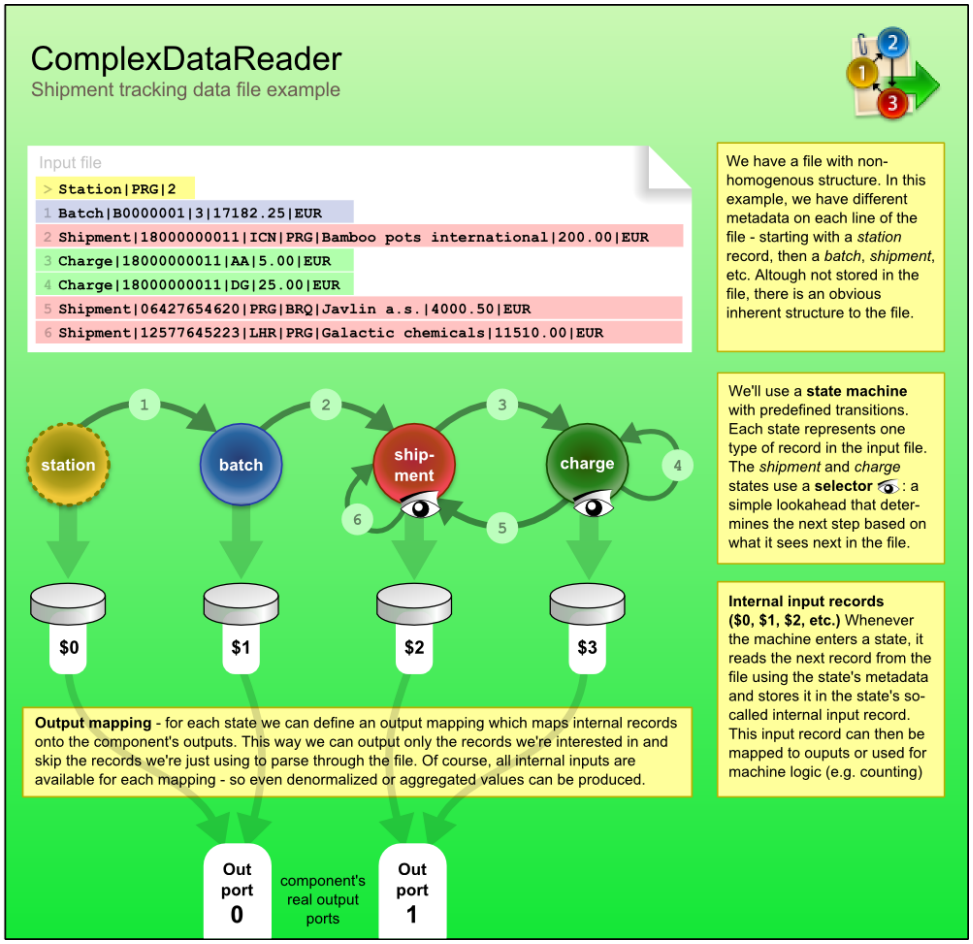
説明:

1)この3つの属性を定義しなかった場合は、デフォルトの「**Selector class**」(`PrefixInputMetadataSelector`)が使用されます。

詳細説明

通常、不均一なデータの読取りは簡単な作業ではありません。データには、様々なデータ形式、デリミタ、フィールドおよびレコード・タイプが混在している可能性があります。さらに、レコードおよびそのセマンティクスが相互に依存している場合があります。たとえば、タイプ`address`のレコードは、直前のレコードが`person`の場合は個人の住所を表しますが、`address`が`company`の後にくる場合は企業の住所を表します。

MultiLevelReaderおよび**ComplexDataReader**は、処理内容の点で非常に似ているコンポーネントです。**MultiLevelReader**では、ロジック全体をJavaの変換として(AbstractMultiLevelSelectorの拡張という形で)プログラムする必要がありましたが、ComplexDataReaderでは、相当複雑なデータ構造であっても強力なGUIを使用して構成できます。**ComplexDataReader**には状態および遷移という新しい概念が導入され、解析ロジックがシンプルなCTL2スクリプトとして実装されています。



状態間の遷移は、明示的に指定するか(状態3は常に状態2の後とするなど)、CTLで計算(グループ内のエントリ数を数えるなど)できます。また、遷移を選択するために支援ツールを参考にできます。このツールは**セレクタ**と呼ばれ、受信データを解析することなく、そのデータを先読みする拡大鏡であると理解できます。

セレクタは、Javaでカスタム実装したり、デフォルトのセレクタを使用できます。デフォルトのセレクタでは、接頭辞およびそれに対応する遷移の表を使用します。特定の接頭辞を検出すると、すべての遷移を評価し、最初に一致するターゲット状態を返します。

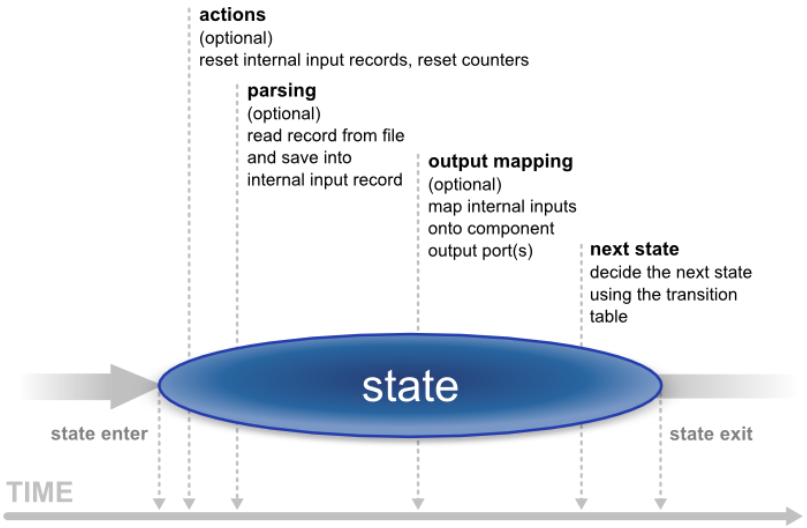
それでは、状態の変化について少し詳しく見てみます(次の図を参照してください)。ある状態が開始されると、その**アクション**が実行されます。使用可能なアクションは次のとおりです。

- **カウンタのリセット**: 状態へのアクセス回数を格納するカウンタをリセットします。
- **レコードのリセット**: 内部ストレージに格納されているレコードの数をリセットします。これにより、様々なデータの読取りで相互に混在することがなくなります。

次に、入力データの**解析**が実行されます。これにより、ファイルからレコードが読み取られ、状態の内部入力に格納されます。

その後に**出力**が行われ、内部入力がコンポーネントの出力ポートにマッピングされます。これは、データがコンポーネントから送信される唯一の手順です。

最後に、マシンを次の状態に変更する方法を定義する**遷移**があります。



最後に付け加えると、CTLの読取りロジック全体を書き込むこともできます。詳細は、ComplexDataReaderのCTL(p.348)を参照してください。

動画: ComplexDataReaderの使用方法

大量の資料を読むかわりに、動画による手順の説明を見てください。これを見ると、**ComplexDataReader**コンポーネントの使用方法および構成方法が明確になります。

ComplexDataReaderの次のサンプル動画を参照してください。

http://www.cloveretl.com/resources/repository/complexdatareader-shipments

ステート・マシンの設計

マシンの設計を開始するには、**変換**属性を編集します。新しいウィンドウが開き、「**States**」、「**Overview**」、「**Selector**」、「**Source**」の各タブおよび「**$stateNo stateLabel**」とラベル付けされた、状態を表すその他のタブ(「$0 myFirstState」など)が表示されます。

「**States**」タブの左側には、グラフで操作するすべての**使用可能なメタデータ**を含むペインが表示されます。このタブでは、右側の「States」ペインにメタデータをドラッグして新しい状態を設計します。下の部分では、**初期状態**(最初の状態)および**最終状態**(マシンは実行終了の直前または**Flush and finish**を呼び出した場合にこの状態に切り替わります)を設定できます。最終状態はオートマトンが終了する前に、出力に対するデータのマッピングに使用できます(特に、入力の最後のレコードの処理に有効)。最後に、中央には**[Ctrl]キーと[Space]キー**を同時に押すコンテンツ・アシストをサポートし、コードを直接編集できる「**Expression editor**」ペインがあります。

「**Overview**」タブでは、マシンがグラフィカルに表示されます。ここでは、外部ファイルに対する「**Export Image**」の実行や、「**Cycle View Modes**」による同じマシンの他のグラフィック表示ができます。「**Undock**」をクリックすると、ビュー全体が別ウィンドウで開きます。このウィンドウは定期的にリフレッシュされます。

状態のタブ(「$0 firstState」など)では、「**Output ports**」ペインで出力を定義します。「**Output**」**フィールド**には実際の(固定済)出力メタデータが表示されます。次に、「**Actions**」を定義し、タブの下部にあるペインで「**Transition**」**表**を操作します。この表内には、上から下へと評価される「**Conditions**」およびそれらに割り当てられた「**Target states**」があります。「**Target states**」の値を次に示します。

- **Let selector decide**: 次の移動先となる状態をセレクタが決定します。
- **Flush and finish**: マシンの動作を通常どおり終了します。
- **Fail**: マシンでエラーが発生して実行を終了します。(無効なレコードを検出した場合など)
- マシンが変化する特定の状態。

「**Selector**」タブを使用すると、独自のセレクタを実装したり、外部ファイルまたはJavaクラスでそのセレクタを指定できます。

最後に、「**Source**」タブにはマシンが実行するコードが表示されます。詳細は、ComplexDataReaderのCTL(p.348)を参照してください。

ComplexDataReaderのCTL

マシンは3通りの方法で指定できます。最初に、GUIを使用して、全体としてマシンを設計できます。2番目に、マシンを記述するJavaクラスを作成できます。3番目に、「**Transform**」の「**Source**」タブに切り替えて、GUI内でCTLのコードを記述できます。このタブには、マシンが実行するソース・コードが表示されます。

重要

ソース・コードの処理は必要ありません。このウィンドウの別グラフィック・タブで、マシンの全体を構成できます。

「**Source**」で加えた変更は、「**Refresh states**」をクリックするとそれ以外のタブで有効になります。ソース・コードを状態構成と同期化する場合は、「**Refresh source**」をクリックします。

コードの重要な要素の概要を次で説明します。

カウンタ

状態へのアクセス回数を格納するcounterStateNo変数があります。状態ごとに変数が1つあり、その番号は0から始まります。そのため、たとえばcounter2には状態$2へのアクセス回数が格納されます。カウンタは「**Actions**」でリセットできます。

初期状態関数

`integer initialState()`: オートマトンの状態が開始時の最初の状態かを決定します。ALLが返された場合は、**Let selector decide**を表しています。したがって、次の状態を決定するセレクタに現在の状態を渡します(これを実行しない場合、マシンでエラーが発生します)。

最終状態関数

`integer finalState(integer lastState)`: オートマトンの最後の状態を指定します。STOPが返された場合は、最終状態が定義されていないことを表しています。

すべての状態での関数

各状態には、その状態を記述する2つの主な関数が用意されています。

- `nextState`
- `nextOutput`

`integer nextState_stateNo()`は、現在の状態の後続の状態を示す番号を返します(stateNo)。ALLが返された場合は、**Let selector decide**を表しています。STOPが返された場合は、**Flush and finish**を表しています。

例53.1. 状態関数の例

```
nextState_0() {
 if(counter0 > 5) {
  return 1; // if state 0 has been accessed more than five times since
            // the last counter reset, go to state 1
 }
 return 0;  // otherwise stay in state 0
}
```

`nextOutput_stateNo(integer seq)`: 特定の状態(`stateNo`)のメイン出力関数。`seq`の値に従って、個々の`nextOutput_stateNo_seq()`サービス関数を呼び出します。この`seq`は、`nextOutput_stateNo`関数がこれまでに呼び出された回数を格納するカウンタです。最後に、`nextOutput_stateNo_default(integer seq)`が呼び出されます。これは通常、すべてが出力に送信され、オートマトンを次の状態に変更できることを示す`STOP`を返します。

`integer nextOutput_stateNo_seq()`: データを出力ポートにマップします。特にこの関数は、`integer nextOutput_1_0()`などとなることがあります。これは、状態$1のマッピングを定義し、`seq`が0に等しいことを示します(この関数が今回初めて呼び出された場合など)。この関数は数値を返します。この数値により、この関数で使用されているポートが示されます。

グローバル次状態関数

`integer nextState(integer state)`: 現在の状態に従って、個々の`nextState()`関数を呼び出します。

グローバル次出力関数

`integer nextOutput(integer state, integer seq)`: 現在の状態および`seq`の値に従って、個々の`nextOutput()`関数を呼び出します。

セレクタ

デフォルトでは、セレクタは読み取るデータの最初の部分(接頭辞)を取得し、ステート・マシンの次の状態を決定します。これは、`PrefixInputMetadataSelector`として実装されています。

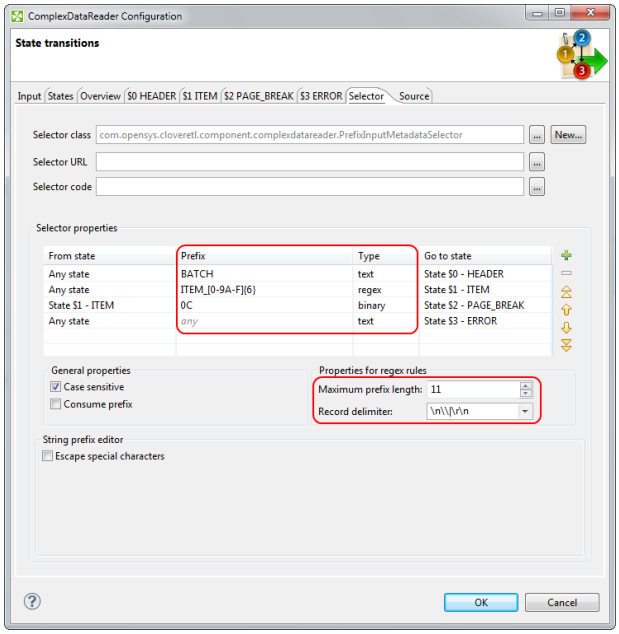


図53.1.ComplexDataReaderでの接頭辞セレクタの構成。ルールは「Selector properties」ペインで定義します。正規表現用に2つの追加属性があります。

セレクタは、ルールのリストを作成して構成できます。すべてのルールは、次の要素から構成されています。

1. 適用対象の状態(「**From state**」)。
2. 「**Prefix**」およびその「**Type**」の指定。接頭辞は、プレーン・テキスト、または16進数コードで記述されたバイト・シーケンスとして指定するか、正規表現を使用して指定できます。これらはルールの**タイプ**となります。接頭辞は空にできますが、その場合は入力に関係なく常にルールが適用されます。
3. オートマトンの次の状態(「**Go to state**」)。

セレクタが呼び出されると、ルールのリストが(上から下に)処理され、最初に適用可能なルールが検索されます。成功すると、オートマトンは選択したルールのターゲット状態に切り替わります。

警告

注意: 残りのルールはチェックされません。したがって、ルールの順番をよく検討する必要があります。空の接頭辞を持つルールがリスト内にある場合、セレクタはそのルールより下のルールに到達できません。通常は、最も一般的なルールをリストの最後に置く必要があります。次の例を参照してください。

例53.2.

次の2つのルールがあります。どちらも任意の状態に適用可能であると仮定します。

- `.{1,3}PATH` (正規表現)
- `APATHY`

ルールがこの順番どおりに定義されている場合、正規表現は「APATHY」の最初の5文字にも一致するため、**後者が適用されることは絶対にありません**。

注意

一部の正規表現は任意の長さの文字シーケンスに一致する可能性があるため、`PrefixInputMetadataSelector`によって入力全体が読み取られないように、2つの新しい属性が導入されました。これらの属性はオプションですが、少なくともそのどちらかを使用することをお薦めします。使用しない場合は、正規表現を使用するルールがあるとセレクタが常に入力全体を読み取ります。この2つの属性は、「**Maximum prefix length**」および「**Record delimiter**」です。正規表現に一致すると、セレクタは文字の「**Maximum prefix length**」までを先に読み取ります(0は無制限を表します)。この読取りは、「**Record delimiter**」が検出された場合にも終了します。

DataGenerator

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

DataGeneratorはデータを生成します。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1)]	各出力に送信[2)]	変換	変換が必要	Java	CTL
DataGenerator	生成済	0	1-N	いいえ	はい	はい	1)	はい	はい

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します。詳細は、変換の戻り値(p.283)を参照してください。

概要

DataGeneratorは、ファイル、データベースまたは他のデータ・ソースからデータを読み取るかわりに、一定のパターンに従ってデータを生成します。データを生成するために、生成変換を定義できます。この変換は**DataGenerator**用CTLテンプレートを使用するか、`RecordGenerate`インタフェースを実装します。そのメソッドを次に示します。コンポーネントは、変換の戻り値(p.283)を使用して、各レコードを異なる出力ポートに送信できます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	はい	生成されたデータ・レコードに使用	任意[1)]
	1-N	いいえ	生成されたデータ・レコードに使用	出力0

説明:

1): すべての出力ポート上のメタデータは、自動入力関数(p.132)を使用できます。

DataGeneratorの属性

属性	必須	説明	可能な値
Basic			
Generator	1)	CTLまたはJavaでグラフに記述された、生成が必要なレコードの定義。	
Generator URL	1)	CTLまたはJavaで記述されたレコードの生成方法の定義を含む、外部ファイルの名前(パスを含む)。	
Generator class	1)	レコードの生成方法を定義する外部クラスの名前。	
Number of records to generate	はい	生成するレコード数。負の数は、設計時に数が不明であることを示します。可変数のレコードの生成(p.358)を参照してください。	
Deprecated			
Record pattern	2)	生成された定数レコードの全フィールドで構成される文字列。ランダム・フィールドやシーケンス・フィールドの値は含まれません。詳細は、レコード・パターン(p.353)を参照してください。ユーザーは、最初にランダム・フィールドとシーケンス・フィールドを定義する必要があります。詳細は、ランダム・フィールド(p.354)およびシーケンス・フィールド(p.353)を参照してください。	
Random fields	2)	セミコロンで区切られた個別フィールド範囲のシーケンス。個別範囲は、その最小値と最大値によって定義されます。最小値は範囲に含まれ、最大値は範囲から除外されます。数値データ型は、ランダムに生成された最小値以上かつ最大値未満の数字を表します。最小値と最大値の両方に同じ値が定義されている場合、これらのフィールドはそのように指定された値と等しくなります。文字列およびバイトのデータ型のフィールドは、その最小長と最大長を指定して定義します。詳細は、ランダム・フィールド(p.354)を参照してください。1つの個別フィールド範囲の例は、`$salary:=random("0","20000")`です。	
Sequence fields	2)	シーケンスにより生成されるフィールド。これらは、セミコロンで区切られた個別フィールド・マッピング(`$field:=IdOfTheSequence`)のシーケンスとして定義します。シーケンスIDは重複して使用でき、同時に複数のフィールドに対して使用できます。詳細は、シーケンス・フィールド(p.353)を参照してください。	
Random seed	2)	1つの`long`シードを使用して、このランダム数ジェネレータのシードを設定します。各グラフの実行中、すべてのフィールドの値は常に安定します。	0-N

説明:

1): 2のマークが付けられた非推奨属性のかわりに、これらの変換属性の1つを指定する必要があります。ただし、これらの新規属性はオプションです。これらの変換属性はいずれも、**DataGenerator**用CTLテンプレートを使用するか、`RecordGenerate`インタフェースを実装する必要があります。

詳細は、CTLスクリプトの詳細(p.354)またはDataGenerator用Javaインタフェース(p.357)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

2): これらの属性は現在では非推奨です。かわりに、1のマークが付けられた変換属性の1つを定義してください。

詳細説明

DataGeneratorの非推奨属性

これら3つの属性のいずれも定義しない場合は、かわりに、ランダムに生成されるフィールド(「**Random fields**」)、シーケンスにより生成されるフィールド(「**Sequence fields**」)、および定数のその他のフィールド(「**Record pattern**」)を定義する必要があります。

- **Record Pattern**

  「Record pattern」は、生成済レコードのすべての定数フィールド(ランダム・フィールドと順序フィールドを除くすべてのフィールド)を含む文字列であり、区切り形式(出力ポートのメタデータに定義されているデリミタを使用)または固定長(出力ポートのメタデータに定義されているサイズを使用)のいずれかです。

- **Sequence Fields**

  「Sequence fields」は、「**Sequence fields**」属性をクリックすると開くダイアログで定義できます。「**Sequences**」ダイアログは次のようなダイアログです。

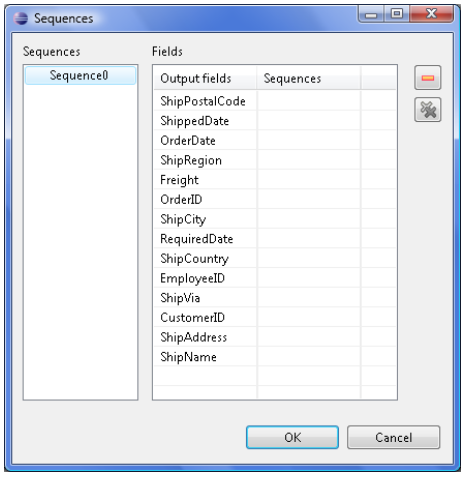


図53.2. 「Sequences」ダイアログ

このダイアログは2つのペインで構成されます。左側にすべてのグラフのシーケンスが表示され、右側にすべてのcloverフィールド(メタデータ内のフィールドの名前)が表示されます。左側で目的のシーケンスを選択して、右側のペインの目的のフィールドにドラッグ・アンド・ドロップします。

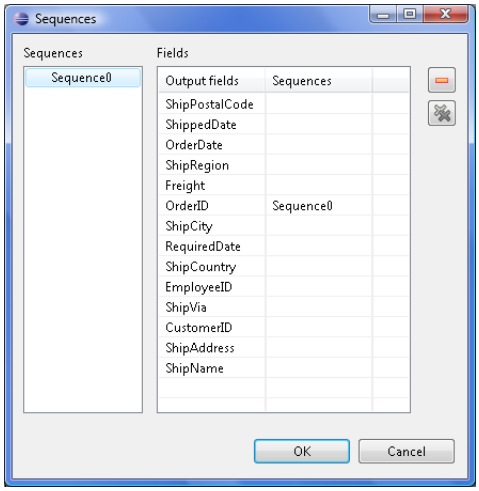


図53.3. 割り当てられたシーケンス

同じシーケンスを複数の異なるcloverフィールドに割り当てる必要はありません。ただし、このように割り当てることは可能です。割り当てるかどうかはユーザー自身が決めます。このダイアログの右側にはボタンが2つあります。選択した割当て済マッピングを取り消すボタンとすべての割当て済マッピングを取り消すボタンです。

- **Random Fields**

この属性は、値がランダムに生成されるすべてのフィールドの値を定義します。それぞれのフィールドに範囲を定義できます。(範囲は最小値と最大値で定義します。)これらの値のデータ型は、メタデータに定義されているデータ型に対応したものになります。「**Random fields**」属性をクリックすると開く「**Edit key**」ダイアログで、ランダム・フィールドを割り当てることができます。

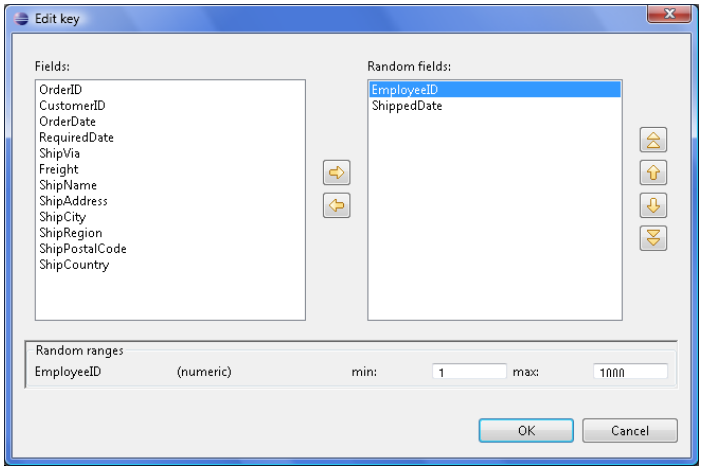


図53.4.「Edit Key」ダイアログ

左側に「**Fields**」ペイン、右側に「**Random fields**」、下部に「**Random ranges**」ペインがあります。「Random ranges」ペインでは、選択したランダム・フィールドの範囲を指定できます。このペインで、特定の値を入力できます。前述のように、**左矢印**ボタンと**右矢印**ボタンをクリックして、「**Fields**」ペインと「**Random fields**」ペインの間でフィールドを移動できます。

CTLスクリプトの詳細

この3つの新しい変換属性のいずれかを定義する場合、値を出力フィールドに割り当てるための変換を指定する必要があります。これを行うには、**変換エディタ**の「**Transformations**」タブを使用します。ただし、この最も簡単なアプローチでは、より詳細な変換を指定できないこともあります。この場合は、CTLスクリプトを使用する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム・フィールド・マッピングを指定できます。

DataGeneratorは、次の変換テンプレートを使用します。

DataGenerator用CTLテンプレート

この変換テンプレートは、**DataGenerator**でのみ使用されます。

CTLで変換を記述した後、タブの右上隅にある対応するボタンをクリックすることによってJava言語コードに変換することもできます。

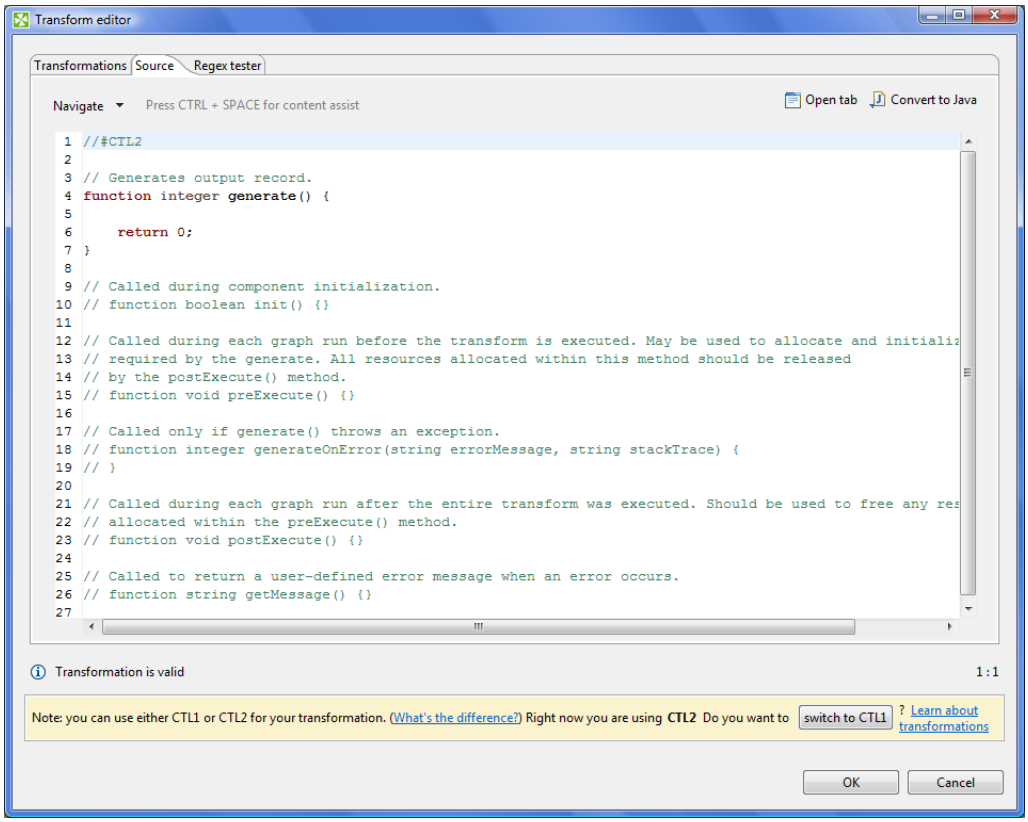


図53.5. DataGeneratorの変換エディタの「Source」タブ

表53.1. DataGeneratorの関数

CTLテンプレート関数	
boolean init()	
必須	いいえ
説明	コンポーネントを初期化し、環境およびグローバル変数を設定します。
起動	1つ目のレコードを処理する前に呼び出されます。
戻り値	`true` | `false` (`false`グラフの失敗時)
integer generate()	
必須	はい
入力パラメータ	なし
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。可変数のレコードの生成(p.358)では、STOPはエラーを表すためではく、生成を正常終了するために使用されています。
起動	出力レコードごとに繰り返し呼び出されます。

CTLテンプレート関数	
説明	出力レコードのすべてのフィールドの構造および値を定義します。特定の出力レコードに対するgenerate()関数の一部が原因でgenerate()関数が失敗し、ユーザーが別の関数(generateOnError())を定義している場合は、generate()が失敗した場所で、このgenerateOnError()内で処理が続行されます。generate()が失敗し、ユーザーがgenerateOnError()を定義していない場合は、グラフ全体が失敗します。generateOnError()関数は、以前に処理が成功したコードから取得した、generate()によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースがgenerateOnError()に渡されます。

例	`function integer generate() {`
	`  myTestString = iif(randomBool(),"1","abc");`
	`  $0.name = randomString(3,5) + " " randomString(5,7);`
	`  $0.salary = randomInteger(20000,40000);`
	`  $0.testValue = str2integer(myTestString);`
	`  return ALL;`
	`}`

integer generateOnError(string errorMessage, string stackTrace)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	generate()が例外をスローすると呼び出されます。
説明	出力レコードのすべてのフィールドの構造および値を定義します。特定の出力レコードに対するgenerate()関数の一部が原因でgenerate()関数が失敗し、ユーザーが別の関数(generateOnError())を定義している場合は、generate()が失敗した場所で、このgenerateOnError()内で処理が続行されます。generate()が失敗し、ユーザーがgenerateOnError()を定義していない場合は、グラフ全体が失敗します。generateOnError()関数は、以前に処理が成功したコードから取得した、generate()によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースがgenerateOnError()に渡されます。

例	`function integer generateOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  $0.name = randomString(3,5) + " " randomString(5,7);`
	`  $0.salary = randomInteger(20000,40000);`
	`  $0.stringTestValue = "myTestString is abc";`
	`  return ALL;`
	`}`

string getMessage()	
必須	いいえ
説明	ユーザーが指定および起動したエラー・メッセージを出力します(generate()またはgenerateOnError()が-2以下の値を返した場合にのみ呼び出されます)。
起動	ユーザーが指定したときに呼び出されます。
戻り値	string

CTLテンプレート関数	
void preExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	生成で必要なリソースを割り当てて初期化するために使用できます。この関数内に割り当てられたすべてのリソースは、`postExecute()`関数によって解放される必要があります。
起動	変換が実行される前の各グラフの実行中に呼び出されます。
void postExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	`preExecute()`関数内に割り当てられたすべてのリソースを解放するために使用する必要があります。
起動	変換全体が実行された後の各グラフの実行中に呼び出されます。

重要

- **出力レコードまたはフィールド**

  出力レコードまたはフィールドには、`generate()`関数および`generateOnError()`関数内からのみアクセスできます。

- CTLテンプレート関数の他のすべての関数では、出力にアクセスできません。

警告

これらのルールに従わない場合はNPEがスローされます。

DataGenerator用Javaインタフェース

変換は、`RecordGenerate`インタフェースのメソッドを実装し、`Transform`インタフェースから他の共通メソッドを継承します。共通Javaインタフェース(p.295)を参照してください。

`RecordGenerate`インタフェースのメソッドを次に示します。

- `boolean init(Properties parameters, DataRecordMetadata[] targetMetadata)`

  生成クラス/関数を初期化します。このメソッドは、生成プロセスの開始時に1回のみ呼び出されます。すべてのオブジェクト割当ておよび初期化はここで実行する必要があります。

- `int generate(DataRecord[] target)`

  ターゲット・レコードのジェネレータを実行します。このメソッドは、レコードの生成フローの1手順として呼び出されます。

注意

このメソッドを使用すると、返された値に応じて、各レコードを異なる接続済出力ポートに配布できます。戻り値およびその意味の詳細は、変換の戻り値(p.283)を参照してください。

- `int generateOnError(Exception exception, DataRecord[] target)`

  ターゲット・レコードのジェネレータを実行します。このメソッドは、レコードの生成フローの1手順として呼び出されます。`generate(DataRecord[])`が例外をスローした場合にのみ呼び出されます。

- `void signal(Object signalObject)`

  外部で何かが発生したことをジェネレータに通知するために使用できるメソッド。

- `Object getSemiResult()`

  生成から中間結果を取得するために使用できるメソッド。実装される場合と実装されない場合があります。

可変数のレコードの生成

設計時に、生成するレコード数が不明な場合があります。このような場合、**生成するレコード数**の属性の値を負の数に設定します。これにより、このコンポーネントは`generate()`関数が`STOP`を返すまでレコードを生成します(この場合、これはエラーとはみなされません)。これは、JavaとCTLのいずれで定義された変換でも機能します。

警告

最後の繰り返しで`STOP`が返されると、レコードはいずれの出力ポートにも送信されません。

例53.3. CTLでの可変数のレコードの生成

```
integer total = randomInteger(1, 100);
integer counter = 0;

// Generates output record.
function integer generate() {
    counter++;

    if (counter > total) {
        printLog(info, "Terminating");
        return STOP;
    }

    if ((counter % 10) == 0) {
        printLog(info, "Skipping record # " + counter);
        return SKIP;
    }

    $out.0.value = "Record # " + counter;

    return OK;
}
```

DBFDataReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

DBFDataReaderは、固定長dbaseファイルからデータを読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
DBFDataReader	dBaseファイル	0-1	1-n	はい	いいえ	いいえ	いいえ	いいえ	いいえ

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します。詳細は、変換の戻り値(p.283)を参照してください。

概要

DBFDataReaderは、固定dbaseファイル(ローカルまたはリモート)からデータを読み取ります。圧縮ファイル、コンソール、入力ポートまたはディクショナリからもデータを読み取ることができます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	ポート読取りに使用。入力ポートからの読取り(p.300)を参照してください。	1つのフィールド(`byte`、`cbyte`、`string`)。
出力	0	はい	正しいデータ・レコード用。	任意[1]
	1-n	いいえ	正しいデータ・レコード用。	出力0

説明:

1): 出力ポート上のメタデータは、自動入力関数(p.132)を使用できます。注意: `source_timestamp`関数および`source_size`関数は、ファイルから直接読み取る場合のみ機能します(ファイルがアーカイブである場合またはリモートの場所に保存されている場合は、タイムスタンプが空になり、サイズは0になります)。

DBFDataReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	読み取るデータ・ソースを指定する属性(dbaseファイル、コンソール、入力ポート、ディクショナリ)。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Charset		読み取られたレコードのエンコーディング。	IBM850 (デフォルト) | <other encodings>
Data policy		エラーの発生時に必要な処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Controlled | Lenient
Advanced			
Number of skipped records		すべてのソース・ファイルにわたって継続的にスキップされるレコード数。入力レコードの選択(p.305)を参照してください。	0-N
Max number of records		すべてのソース・ファイルにわたって継続的に読み取られる最大レコード数。入力レコードの選択(p.305)を参照してください。	0-N
Number of skipped records per source		各ソース・ファイルからスキップするレコード数。入力レコードの選択(p.305)を参照してください。	メタデータ内と同じ(デフォルト) | 0-N
Max number of records per source		各ソース・ファイルから読み取るレコードの最大数。入力レコードの選択(p.305)を参照してください。	0-N
Incremental file	1)	パスを含む、増分キーを格納しているファイルの名前。増分読取り(p.304)を参照してください。	
Incremental key	1)	最後に読み取られたレコードの位置を格納している変数。増分読取り(p.304)を参照してください。	

説明:

1) これらの属性は両方とも指定するか、いずれも指定しないかどちらかにしてください。

DBInputTable

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

DBInputTableは、JDBCドライバを使用してデータベースからデータをアンロードします。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
DBInputTable	データベース	0-1	1-n	✔	✖	✖	✖	✖	✖

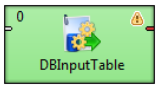
[1] 各データ・レコードをすべての接続済出力ポートに送信します。

[2] 変換の戻り値(p.283)に従って、データ・レコードを出力ポートに送信します。

概要

DBInputTableは、SQL問合せを使用するか、データベース表を指定してデータベース列からCloverフィールドへのマッピングを定義することにより、データベース表からデータをアンロードします。アンロードしたレコードを、すべての接続済出力ポートに送信できます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0-1	✖	**SQL問合せ属性**で使用される受信問合せ。入力ポートが接続されている場合、**問合せURL**を、たとえば `port:$0.fieldName:discrete`として指定する必要があります。入力ポートからの読取り(p.300)を参照してください。	
出力	0	✔	正しいデータ・レコード用。	メタデータと同じ[1]
	1-n	✖	正しいデータ・レコード用。	

1) 出力メタデータは自動入力関数(p.132)を使用できます。

DBInputTableの属性

属性	必須	説明	可能な値
Basic			
DB connection	✔	データベースへのアクセスに使用されるデータベース接続のID。	
Query URL	[1]	パスを含む、SQL問合せを定義する外部ファイルの名前。	
SQL query	[1]	グラフで定義されているSQL問合せ。詳細は、SQL問合せエディタ(p.363)を参照してください。	
Query source charset		SQL問合せを定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Data policy		エラーの発生時に必要な処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Controlled | Lenient
Advanced			
Fetch size		データベースから一度にフェッチする必要があるレコード数を指定します。	20 | 1-N
Incremental file	[2]	パスを含む、増分キーを格納しているファイルの名前。増分読取り(p.304)を参照してください。	
Incremental key	[2]	最後に読み取られたレコードの位置を格納している変数。増分読取り(p.304)を参照してください。	
Auto commit		デフォルトでは、SQL問合せは即時にコミットされます。1つのトランザクション内でさらに操作を実行する必要がある場合は、この属性を`false`に切り替えます。	true (デフォルト) | false

[1]これらの属性の少なくとも1つを指定する必要があります。両方を定義した場合、問合せURLのみが適用されます。

[2]これらの属性は両方とも指定するか、いずれも指定しないかどちらかにしてください。

詳細説明

問合せ属性の定義

- **マッピングを使用しない問合せ文**

  CloverETLメタデータ・フィールドとselect文のデータベース列の順序が同じであり、データ型に互換性がある場合、位置指定マッピングを実行する暗黙的マッピングを使用できます。標準SQL問合せ構文を使用する必要があります。

  - `select * from table [where dbfieldJ = ? and dbfieldK = somevalue]`
  - `select column3, column1, column2, ... from table [where dbfieldJ = ? and dbfieldK = somevalue]`

  SQL問合せの定義方法の詳細は、SQL問合せエディタ(p.363)を参照してください。

- **マッピングを使用する問合せ文**

  複数の表に対してもデータベース・フィールドをcloverフィールドにマップする場合、問合せは次のようになります。

```
select $cloverfieldA:=table1.dbfieldP, $cloverfieldC:=table1.dbfieldS, ...,
$cloverfieldM:=table2.dbfieldU, $cloverfieldM:=table3.dbfieldV from table1,
table2, table3 [where table1.dbfieldJ = ? and table2.dbfieldU = somevalue]
```

SQL問合せの定義方法の詳細は、SQL問合せエディタ(p.363)を参照してください。

DB表名のドル記号

- データベース表の名前に含まれるドル記号は、生成される問合せでは二重ドル記号に変換されます。したがって、各問合せのdb表には(隣接するドル記号のペアで構成される)偶数個のドル記号が含まれるはずです。db表の名前に含まれる単一ドル記号は、問合せのdb表名では二重ドル記号に置換されます。

重要

また、MS SQL Serverに接続する場合は、jTDS http://jtds.sourceforge.netドライバを使用することをお薦めします。これは、Microsoft SQL ServerおよびSybase向けのオープン・ソースの100% Pure Java JDBCドライバです。Microsoftのドライバよりも高速です。

SQL問合せエディタ

SQL問合せ属性を定義するには、**SQL問合せエディタ**を使用できます。

このエディタは、「**SQL query**」属性行をクリックすると開きます。

左側には、これらの列のスキーマ、表、列およびデータ型に関する情報を含む「**Database schema**」ペインがあります。

表示されたスキーマ、表および列は、「**ALL**」コンボや「**Filter in view**」テキスト領域の値、「**Filter**」ボタンおよび「**Reset**」ボタンなどを使用してフィルタできます。

列を選択するには、スキーマ、表を展開して、目的の列で**[Ctrl]を押しながらクリック**します。

隣接する複数の列を選択するには、**[Shift]**を押しながら最初の項目とリスト項目を**クリック**します。

次に、「**Generate**」をクリックすると、問合せが「**Query**」ペインに表示されます。

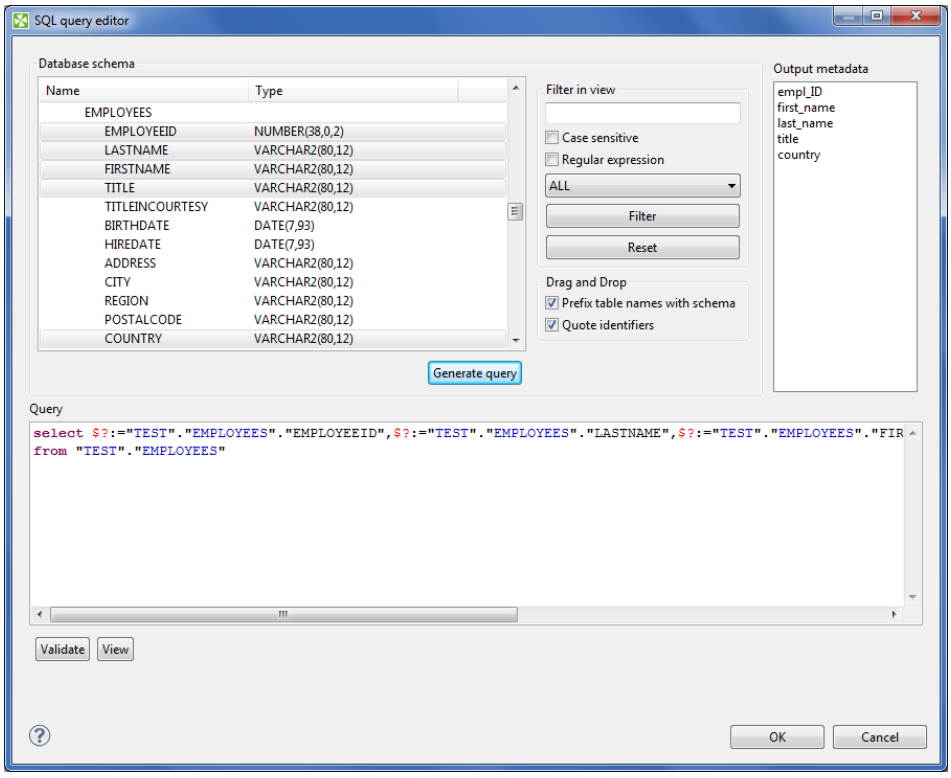


図53.6 疑問符を含む生成済問合せ

出力メタデータ・フィールドと異なるdb列があると、問合せに疑問符が含まれることがあります。出力メタデータは、右側の「**Output metadata**」ペインに表示されます。

「**Output metadata**」ペインから「**Query**」ペインの対応する場所にフィールドをドラッグ・アンド・ドロップして、手動で"$:="という文字を削除します。次の図を参照してください。

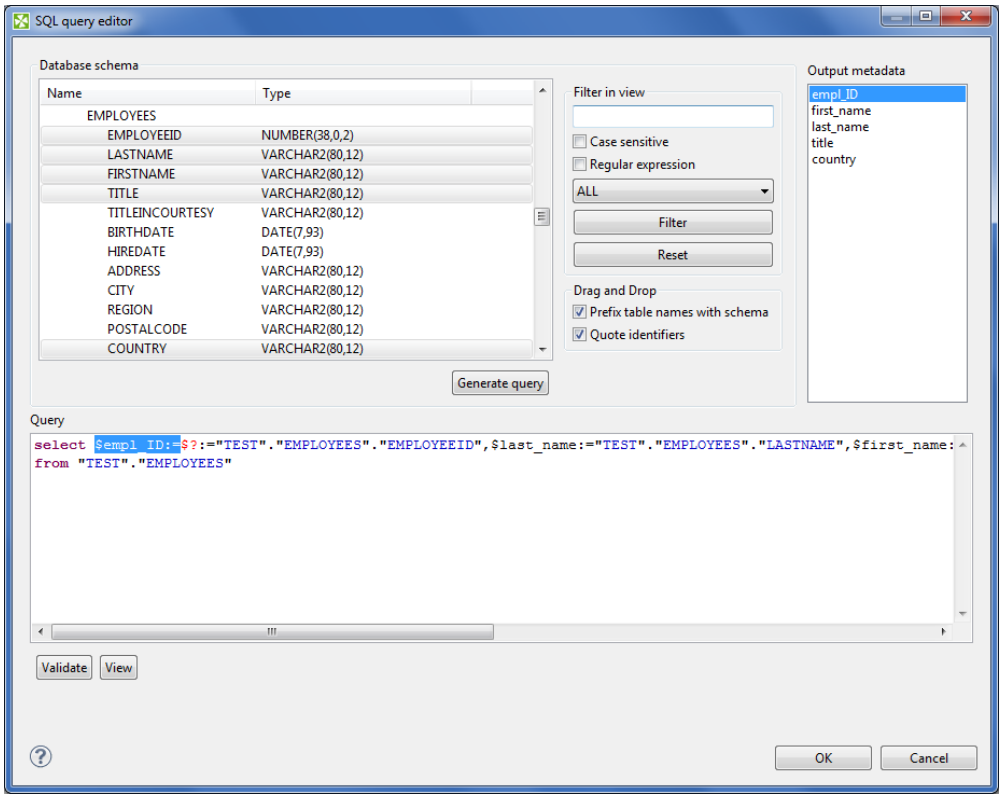


図53.7 出力フィールドを含む生成済問合せ

問合せにwhere文を入力することもできます。

下にある2つのボタンを使用して、問合せを検証(「**Validate**」)したり、表のデータを表示(「**View**」)できます。

EmailReader

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

EmailReaderは、電子メール・メッセージのストアをデリミタ付きフラット・ファイルからローカルで、または外部サーバーで読み取ります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
EmailReader	✖	✖	1	2	-	-

概要

EmailReaderは、オンラインまたはローカルの電子メール・メッセージの読取りを可能にするコンポーネントです。

このコンポーネントは、電子メール・メッセージを解析して、その属性を2つの接続済出力ポートに書き出します。1つ目のポートはコンテンツ・ポートであり、電子メールおよび本文の関連情報を出力します。2つ目のポートは添付ファイル・ポートであり、電子メールに含まれる添付ファイルに関連する情報を書き込みます。

コンテンツ・ポートは、電子メールごとに1つずつレコードを書き込みます。添付ファイル・ポートは、電子メール・メッセージごとに複数のレコード、つまり検出された添付ファイルごとに1つずつレコードを書き込むことができます。

アイコン

ポート

ポートを参照する際は、ユースケース・シナリオを理解している必要があります。このコンポーネントには、ローカル・ソースまたは外部サーバーからデータを読み取る機能があります。このコンポーネントは、1つの入力ポートに接続されているエッジがあるかどうかに基づいて、どのケースを使用するかを決定します。

ケース1: エッジが入力ポートに接続されている場合、コンポーネントはデータをローカルに読み取ることを想定します。多くの場合、このエッジは**UniversalDataReader**が起点となります。この場合、1つのファイルに、選択したデリミタで区切られた複数の電子メール・メッセージ本文が含まれることがあります。また、各メッセージは、解析および処理のために1つずつ**EmailReader**に渡されます。

ケース2: エッジが入力ポートに接続されていない場合、コンポーネントは外部サーバーからメッセージを読み取ることを想定します。この場合、ユーザーは、サーバー・ホストやプロトコルのパラメータなどの関連属性と、関連するユーザー名やパスワードを入力する必要があります。

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✖	フラット・ファイルからの電子メール・メッセージの入力に使用	文字列フィールド
出力	0	✖	コンテンツ・ポート	任意
	1	✖	添付ファイル・ポート	任意

EmailReaderの属性

多くの属性が必須かどうかは、コンポーネントの構成によってのみ決まります。ポート(p.365)を参照してください。エッジが入力ポートに接続されていないケース2では、外部サーバーに接続するために多くの属性が必須となります。ユーザーは、少なくとも、プロトコルを選択し、サーバーのホスト名を入力する必要があります。通常は、ユーザー名とパスワードも必要です。

属性	必須	説明	可能な値
Basic			
Server Type		メール・サーバーへの接続に使用するプロトコル。オプションは、POP3とIMAPです。ほとんどの場合、可能であればIMAPを選択する必要がありますが、これは、IMAPがPOP3を改良したものであるためです。	POP3、IMAP
Server Name		サーバーのホスト名。	例: imap.google.com
Server Port		外部サーバーへの接続に使用するポートを指定します。空白のままにした場合、デフォルトのポートが使用されます。	整数
Security		サーバーへの接続に使用するセキュリティ・プロトコルを指定します。	NONE、SSL、STARTTLS、SSL+STARTTLS
User Name		サーバーに接続するためのユーザー名(権限が必要な場合)	
Password		サーバーに接続するためのパスワード(権限が必要な場合)	
Fetch Messages		メッセージのステータスに基づいてメッセージをフィルタします。オプションALLを指定すると、メッセージのステータスに関係なく、サーバーに存在するすべてのメッセージが読み取られます。NEWを指定すると、まだ読み取られていないメッセージのみがフェッチされます。	NEW、ALL
Field Mapping	はい	電子メールの一部(標準およびユーザー定義)をCloverフィールドにマップする方法を定義します。フィールドのマッピング(p.367)を参照してください。	
Advanced			
POP3 Cache File		どのメッセージが読み取られたかを追跡するために使用するファイルのURLを指定します。POP3サーバーには、デフォルトでは、読み取られたメッセージと読み取られていないメッセージを追跡する方法はありません。読み取られてないメッセージのみをフェッチするに	

属性	必須	説明	可能な値
		は、サーバーからすべてのメッセージIDをダウンロードして、すでに読み取られたメッセージIDのリストと比較する必要があります。この方法を使用すると、このリストに含まれていないメッセージのみが実際にダウンロードされるため、帯域幅を節約できます。このファイルは、すでに読み取られたメッセージの一意のメッセージIDを格納するデリミタ付きテキスト・ファイルにすぎません。ALLメッセージを選択した場合でも、読み取られるメッセージが移入されるため、ユーザーはキャッシュ・ファイルを指定する必要があります。注意: popキャッシュ・ファイルは汎用的です。多くの受信ボックス間で共有でき、ユーザーは各メール・ボックスに個別のキャッシュを管理するように選択できます。	

詳細説明

フィールドのマッピング

フィールド・マッピング属性を編集すると、次の単純なダイアログが表示されます。

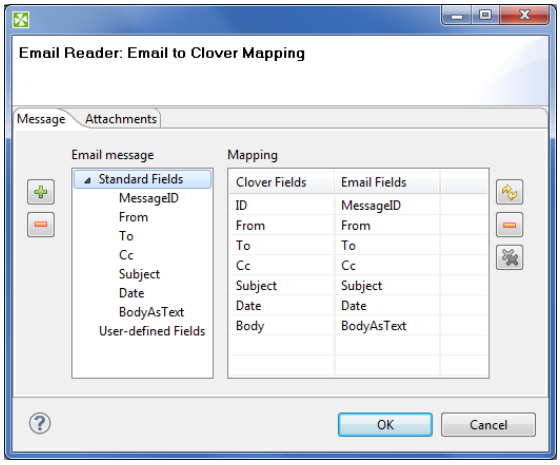


図53.8. EmailReaderでのCloverフィールドへのマッピング

このダイアログの2つのタブ(「**Message**」と「**Attachments**」)では、ドラッグ・アンド・ドロップのみで受信電子メール・フィールドをCloverフィールドにマッピングします。右側のボタンをクリックすると、**すべてのマッピングを取り消す**ことができます。このウィンドウを初めて開くと、**自動マッピング**が実行されます。また、2つ目の出力ポートを使用している場合は、「**Attachments**」にメタデータ・フィールドのみが表示されます(その理由は、ポート (p.365)を参照してください)。

注意

ユーザー定義フィールドを使用すると、**標準**のフィールド以外のすべてのフィールドを処理できます。たとえば、電子メール・ヘッダー内のカスタム・フィールドなどです。

ヒントおよびポイント

- 必ず、Java仮想マシン(JVM)に専用のメモリーが十分にあることを確認してください。メッセージ添付ファイルのサイズによっては(添付ファイルを読み取るように選択した場合)、CloverETLがデータを効率的に処理できるように最大512Mを**CloverETL**に割り当てる必要があります。

パフォーマンスのボトルネック

- 外部サーバーから処理するメッセージの量: **EmailReader**は外部サーバーに接続する必要があるため、帯域幅の制限に達することがあります。サイズの大きい添付ファイルを含む大量のメッセージを処理すると、ダウンロードするコンテンツを待機して、アプリケーションにボトルネックが発生する可能性があります。可能であればNEWオプションを使用し、POP3プロトコルを使用する場合はPOP3キャッシュを管理してください。

JavaBeanReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

JavaBeanReaderは、ディクショナリに保存されているJavaBeans階層構造を読み取ります。これにより、Cloverグラフとクラウドなどの外部環境との間で動的なデータ交換が可能になります。読み取るディクショナリは、外部世界(クラウドなど)と内部世界(Clover)の間のインタフェースとして機能します。

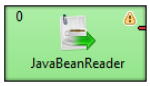
コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1)]	各出力に送信[2)]	変換	変換が必要	Java	CTL
JavaBeanReader	ディクショナリ	0	1-n	いいえ	はい	いいえ	いいえ	いいえ	いいえ

概要

JavaBeanReaderは、ディクショナリを介してJavaBeansからデータを読み取ります。定義したマッピングに基づいて、Java属性/コレクション要素が出力レコードにマップされます。入力ファイル全体をマップする必要はなく、XPath式を使用して必要なデータのみを選択します。このコンポーネントは、マッピングによる定義に従って、様々な接続済出力レコードにデータを送信します。

マッピング・プロセスは、XMLXpathReader (p.452)でのマッピング・プロセスに類似しています。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	はい	正常に読み取られたレコード。	任意
	1-n	マッピングで必要な場合、他の出力ポートを接続します。	正常に読み取られたレコード。	任意。ポートごとに異なるメタデータを保持できます。

JavaBeanReaderの属性

属性	必須	説明	可能な値
Basic			
Dictionary source	はい	JavaBeansを読み取るディクショナリ。	すでに定義したディクショナリの名前。
Data policy		読取りエラーの発生時に必要な処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Controlled | Lenient
Mapping	1)	入力JavaBeans構造から出力ポートへのマッピング。詳細は、JavaBeanReaderのマッピング定義(p.371)を参照してください。	
Mapping URL	1)	マッピング定義が含まれる外部テキスト・ファイル。	
Implicit mapping		デフォルトでは、同じ名前のCloverフィールドにも入力要素を手動でマップする必要があります。trueに切り替えると、一致する名前についてJSONからCloverへのマッピングが自動的に実行されます。これにより、適切に構造化された長いJSONファイルでの作業が大幅に軽減されます。JSONマッピング: 詳細(p.383)を参照してください。	false (デフォルト) | true

説明:

1) これらのうちどちらかを指定する必要があります。両方が指定された場合は、「**Mapping URL**」の優先度が高くなります。

詳細説明

JavaBeanReaderのマッピング定義

1. **マッピング**定義はそれぞれ<Context>タグで構成されており、ここにも属性が含まれ、要素名をCloverフィールドにマッピングできます。<Context>タグのネストした構造は、入力JavaBeansの要素のネストした構造に似ています。
2. 各<Context>タグで、一連のネストされた<Mapping>タグを囲むことができます。これらにより、JavaBeans要素の名前をCloverフィールドに変更できます。ただし、Mappingでは入力構造全体をコピーする必要はなく、**Mapping**はツリー内の任意の深さから開始できます。
3. これらの各<Context>タグにはいくつかのJavaBeanReaderのコンテキスト・タグの属性(p.372)が含まれ、各<Mapping>タグにはいくつかのJavaBeanReaderのマッピング・タグの属性(p.373)が含まれます。

例53.4. JavaBeanReaderでのMappingの例

```
<Context xpath="/employees" outPort="0" sequenceId="empSeq" sequenceField="id">
<Mapping xpath="firstName" cloverField="firstName"/>
<Mapping xpath="lastName" cloverField="lastName"/>
<Mapping xpath="salary" cloverField="salary"/>
<Mapping xpath="jobTitle" cloverField="jobTitle"/>
<Context xpath="children" outPort="1" parentKey="id" generatedKey="empID">
 <Mapping xpath="name" cloverField="chname"/>
 <Mapping xpath="age" cloverField="age"/>
</Context>
<Context xpath="benefits" outPort="2" parentKey="id" generatedKey="empID">
 <Mapping xpath="car" cloverField="car"/>
 <Mapping xpath="cellPhone" cloverField="mobilephone"/>
 <Mapping xpath="monthlyBonus" cloverField="monthlyBonus"/>
 <Mapping xpath="yearlyBonus" cloverField="yearlyBonus"/>
</Context>
<Context xpath="projects" outPort="3" parentKey="id" generatedKey="empID">
 <Mapping xpath="name" cloverField="projName"/>
 <Mapping xpath="manager" cloverField="projManager"/>
 <Mapping xpath="start" cloverField="Start"/>
 <Mapping xpath="end" cloverField="End"/>
 <Mapping xpath="customers" cloverField="customers"/>
</Context>
</Context>
```

重要

「**Implicit mapping**」をtrueに切り替えると、要素(salaryなど)が同じ名前(salary)のフィールドに自動的にマップされ、次のように記述する必要は**ありません**。

```
<Mapping xpath="salary" cloverField="salary"/>
```

明示的にマップするのは、フィールドに固有要素からのデータを移入する場合のみです。

4. **JavaBeanReaderのコンテキスト・タグおよびマッピング・タグ**

- **空のコンテキスト・タグ(子なし)**

<Context xpath="xpathexpression" JavaBeanReaderのコンテキスト・タグの属性(p.372) />

- **空ではないコンテキスト・タグ(子がある親)**

  ```
  <Context xpath="xpathexpression" JavaBeanReaderのコンテキスト・タグの属性(p.372) >
  (nested Context and Mapping elements (only children, parents with one or more children, etc.)
  </Context>
  ```

- **空のマッピング・タグ(名前変更タグ)**

  - xpathを使用:

    ```
    <Mapping xpath="xpathexpression" JavaBeanReaderのマッピング・タグの属性(p.373) />
    ```

  - nodeNameを使用:

    ```
    <Mapping nodeName="elementname" JavaBeanReaderのマッピング・タグの属性(p.373) />
    ```

5. **JavaBeanReaderのコンテキスト・タグおよびマッピング・タグの属性**

   1) **JavaBeanReaderのコンテキスト・タグの属性**

- xpath

  必須

  xpath式は、任意のXPath問合せにできます。

  例: `xpath="/tagA/.../tagJ"`

- outPort

  オプション

  データの送信先となる出力ポートの数。定義されない場合、このレベルの**マッピング**からのデータは、このレベルの**マッピング**を使用して送信されません。

  例: `outPort="2"`

- parentKey

  parentKeyおよびgeneratedKeyの両方を指定する必要があります。

  セミコロン、コロンまたはパイプで区切られた、次の親レベルのメタデータ・フィールドのシーケンス。これらすべてのフィールドの数およびデータ型は、generatedKey属性内で同じにする必要があります。同じでない場合は、すべての値が連結されて、一意の文字列値が作成されます。この場合、キーのフィールドは1つのみになります。

  例: `parentKey="first_name;last_name"`

  これらの属性の値を等しくすることにより、今後のこのようなレコードを確実に結合できます。

- generatedKey

  parentKeyおよびgeneratedKeyの両方を指定する必要があります。

  セミコロン、コロンまたはパイプで区切られた、指定されたレベルのメタデータ・フィールドのシーケンス。これらすべてのフィールドの数およびデータ型は、parentKey属性内で同じにする必要があります。同じでない場合は、すべての値が連結されて、一意の文字列値が作成されます。この場合、キーのフィールドは1つのみになります。

  例: `generatedKey="f_name;l_name"`

これらの属性の値を等しくすることにより、今後のこのようなレコードを確実に結合できます。

- `sequenceId`

  `parentKey`と`generatedKey`のペアによってレコードの一意の識別が保証されない場合は、シーケンスを定義して使用できます。

  シーケンスのID。

  例: `sequenceId="Sequence0"`

- `sequenceField`

  `parentKey`と`generatedKey`のペアによってレコードの一意の識別が保証されない場合は、シーケンスを定義して使用できます。

  シーケンスの値が書き込まれる、指定されたレベルのメタデータ・フィールド。次にネストされたレベルの`parentKey`として機能できます。

  例: `sequenceField="sequenceKey"`

2) **JavaBeanReaderのマッピング・タグの属性**

- `xpath`

  `xpath`または`nodeName`のいずれかを`<Mapping>`タグ内に指定する必要があります。

  XPathの問合せ。

  例: `xpath="tagA/.../salary"`

- `nodeName`

  `xpath`または`nodeName`のいずれかを`<Mapping>`タグ内に指定する必要があります。`nodeName`を使用する方が、`xpath`を使用するよりも速くなります。

  Cloverフィールドにマップする必要があるJavaBeansノード。

  例: `nodeName="salary"`

- `cloverField`

  必須

  JavaBeansノードをマップする必要があるCloverフィールド。

  対応するレベルにおけるフィールドの名前。

  例: `cloverFields="SALARY"`

複数値フィールドの読取り

Clover 3.3の時点では、複数値フィールドの読取りがサポートされています。ただし、リストの読取りのみが可能です(複数値フィールド(p.168)を参照してください)。

注意

マップの読取りは、(マップの値としてのすべてのデータ型について)純粋な`string`の読取りとして扱われます。

例53.5. JavaBeanReaderによるリストの読取り

たとえば、3つの要素(John、Vicky、Brian)のリストを含む入力ファイルがあるとします。

次のマッピングを使用すると、コンポーネントでリード・バックできます。

```
<Mapping xpath="attendees" cloverField="attendanceList"/>
```

ここで、attendanceListは、使用するメタデータのフィールドです。このメタデータは、コンポーネントの出力エッジに割り当てる必要があります。グラフの実行後、フィールドに次のようなデータが移入されます(これは**「View data」**に表示されます)。

```
[John,Vicky,Brian]
```

HadoopReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

HadoopReaderは、Hadoopシーケンス・ファイルを読み取ります。

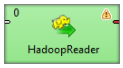
コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
HadoopReader	Hadoopシーケンス・ファイル	0–1	1	✖	✖	✖	✖	✖	✖

1) 各データ・レコードをすべての接続済出力ポートに送信します。
2) 変換の戻り値(p.283)に従って、データ・レコードを出力ポートに送信します。

概要

HadoopReaderは、特殊なHadoopシーケンス・ファイル(`org.apache.hadoop.io.SequenceFile`)からデータを読み取ります。これらのファイルはキーと値のペアを含み、MapReduceジョブで入力/出力ファイル形式として使用されます。このコンポーネントは、単一ファイルに加え、HDFSまたはローカル・ファイル・システムに配置する必要のあるファイルのコレクションを読み取ることができます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✖	入力ポートの読取り(p.303)用。`source`モードのみがサポートされています。	任意
出力	0	✔	読取りデータ・レコード用。	任意

HadoopReaderの属性

属性	必須	説明	可能な値
Basic			
Hadoop connection		Hadoopシーケンス・ファイル・パーサー実装を含むHadoopライブラリとのHadoop接続(p.192)。「**File URL**」属性の `hdfs://` URLでHadoop接続IDが指定されている場合、この属性の値は無視されます。	Hadoop接続ID
File URL	✔	HDFSまたはローカル・ファイル・システム上のファイルのURL。 プロトコルを含まないURL (実際には絶対パスまたは相対パス)、または`file://`プロトコルを含むURLは、ローカル・ファイル・システムに配置されているとみなされます。 読み取るファイルがHDFSに配置されている場合は、`hdfs://ConnID/path/to/file`という形式のURLを使用してください。ここで、`ConnID`はHadoop接続(p.192)のIDであり(「**Hadoop connection**」コンポーネント属性は無視されます)、`/path/to/myfile`は、対応するHDFS上の`myfile`という名前のファイルの絶対パスです。	
Key field	✔	キーと値の各ペアのキーが格納される出力エッジ・レコード・フィールドの名前。	
Value field	✔	キーと値の各ペアの値が格納される出力エッジ・レコード・フィールドの名前。	

詳細説明

HadoopReaderコンポーネントによりサポートされるファイル形式の正確なバージョンは、「**File URL**」属性から参照される「**Hadoop connection**」で指定したHadoopライブラリによって決まります。一般に、あるHadoopバージョンで作成されたシーケンス・ファイルを他のバージョンで読み取ることはできません。

Hadoopシーケンス・ファイルには圧縮データが含まれることがあります。**HadoopReader**は、これを自動的に検出してデータを解凍します。サポートされる圧縮コーデックも、「**Hadoop connection**」で指定したライブラリによって決まります。

Hadoopシーケンス・ファイルの技術的な詳細は、Apache Hadoop Wikiを参照してください。

JMSReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

JMSReaderは、JMSメッセージをCloverデータ・レコードに変換します。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
JMSReader	jmsメッセージ	0	1	はい	いいえ	はい	いいえ	はい	いいえ

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します。詳細は、変換の戻り値(p.283)を参照してください。

概要

JMSReaderはJMSメッセージを受信してCloverデータ・レコードに変換し、これらのレコードを接続済出力ポートに送信します。コンポーネントは、`JmsMsg2DataRecord`インタフェースを実装するプロセッサ変換か、`JmsMsg2DataRecordBase`スーパークラスから継承するプロセッサ変換を使用します。`JmsMsg2DataRecord`インタフェースのメソッドについては後述します。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	はい	正しいデータ・レコード用。	任意[1]

説明:

1): 出力ポートのメタデータには、「**Message body field**」属性で指定されたフィールドが含まれることがあります。メタデータは自動入力関数(p.132)を使用することもできます。

JMSReaderの属性

属性	必須	説明	可能な値
Basic			
JMS connection	はい	使用されるJMS接続のID。	
Processor code	1)	JMSメッセージから、Javaでグラフに記述されたレコードへの変換。	
Processor URL	1)	JMSメッセージからJavaで記述されたレコードへの変換が含まれる外部ファイルの名前(パスを含む)。	
Processor class	1)	JMSメッセージからレコードへの変換を定義する外部クラスの名前。ほとんどの場合、デフォルトのプロセッサ値で間に合います。`javax.jms.TextMessage`と`javax.jms.BytesMessage`の両方を処理できます。	JmsMsg2DataRecordProperties (デフォルト) | 他のクラス
JMS message selector		処理が必要なJMSメッセージをフィルタするために使用する標準JMX問合せ。実際には、メッセージ・プロパティおよびSQL式のサブセットである構文を使用する文字列問合せです。詳細は、http://docs.oracle.com/javaee/1.4/api/javax/jms/Message.htmlを参照してください。	
Processor source charset		Javaでの変換を含む外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト) | その他のエンコーディング
Message charset		JMSメッセージ・コンテンツのエンコーディング。この属性は、デフォルトのプロセッサ実装(`JmsMsg2DataRecordProperties`)でも使用されます。これは`javax.jms.BytesMessage`のみに使用されます。	ISO-8859-1 (デフォルト) | その他のエンコーディング
Advanced			
Max msg count		受信する最大メッセージ数。0は制限がないことを意味します。詳細は、実行の制限(p.379)を参照してください。	0 (デフォルト) | 1-N
Timeout		メッセージを受信する最大時間(ミリ秒)。0は制限がないことを意味します。詳細は、実行の制限(p.379)を参照してください。	0 (デフォルト) | 1-N
Message body field		メッセージ本文を書き込む必要があるフィールドの名前。この属性は、デフォルトのプロセッサ実装(`JmsMsg2DataRecordProperties`)で	bodyField (デフォルト) | 他の名前

属性	必須	説明	可能な値
		使用されます。「**Message body field**」が指定されていない場合、bodyFieldという名前のフィールドにメッセージ本文が入力されます。メッセージ本文用のフィールドがメタデータに含まれていない場合、本文はどのフィールドにも書き込まれません。	

説明:

1) これらのいずれかを設定できます。これらの変換属性はいずれも、JmsMsg2DataRecordインタフェースを実装します。

詳細は、JMSReader用Javaインタフェース(p.379)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

詳細説明

実行の制限

受信するメッセージ数または処理時間(あるいはその両方)を制限するかどうかを決定することも重要です。これを行うには、次の設定を使用します。

- **制限付き実行**

  最大メッセージ数(「**Max msg count**」)またはタイムアウト(「**Timeout**」)、あるいはその両方を指定した場合、処理はメッセージ数または処理時間(あるいはこれらの属性の両方)により制限されます。これらを正の値に設定する必要があります。

  指定したメッセージ数が受信された時点、またはプロセスが定義済の時間続いた時点で、プロセスは停止します。いずれの値に先に達するかに関係なく、このような属性が適用されます。

  #### 注意

  グラフの実行は、JmsMsg2DataReaderインタフェースのendOfInput()メソッドを使用して制限することもできます。このメソッドはブール値を返し、グラフの実行を制限することもできます。falseが返されるたびに、処理は停止します。

- **制限なし実行**

  一方、この2つの属性をいずれも指定しない場合、処理が停止することはありません。デフォルトではどちらの属性も0に設定されています。したがって、処理はメッセージ数によっても経過時間によっても制限されません。これは**JMSReader**のデフォルト設定です。

JMSReader用Javaインタフェース

変換は、JmsMsg2DataRecordインタフェースのメソッドを実装し、Transformインタフェースから他の共通メソッドを継承します。共通Javaインタフェース(p.295)を参照してください。

JmsMsg2DataRecordインタフェースのメソッドを次に示します。

- `void init(DataRecordMetadata metadata, Properties props)`

  プロセッサを初期化します。

- `boolean endOfInput()`

  `false`が返された場合に入力JMSメッセージの処理を終了するために使用できます。詳細は、実行の制限 (p.379)を参照してください。

- `DataRecord extractRecord(Message msg)`

  JMSメッセージをデータ・レコードに変換します。`null`は、メッセージがプロセッサにより受け入れられないことを示します。

- `String getErrorMsg()`

  エラー・メッセージを返します。

JSONReader

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

JSONReaderは、通常は`*.json`ファイルに格納されているJava Script Object Notation (JSON形式)のデータを読み取ります。JSONは階層テキスト形式で、読み取る値が名前と値のペアまたは配列に格納されています。マッピングにおいては配列に注意する必要があります(配列の処理(p.385)を参照してください)。多くの場合、JSONオブジェクトは繰り返されるため、通常は複数の出力ポートにマップします。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信	各出力に送信	変換	変換が必要	Java	CTL
JSONReader	JSONファイル	0-1	1-n	いいえ	はい	いいえ	いいえ	いいえ	いいえ

概要

JSONReaderは、入力JSONを取得して内部的にDOMに変換します。その後、XPath式を使用してDOMツリーを走査し、Cloverレコードにマップされるデータ構造を選択します。

DOMには要素のみが含まれ、属性は含まれません。したがって、XPath式に@が含まれることは**ありません**。

入力全体がメモリーに格納されるため、このコンポーネントのメモリー使用率は大きくなることがあります。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	オプション。ポート読取りに使用。	1つのフィールドのみ(byte、cbyteまたはstring)が使用されます。フィールド名は「**File URL**」で使用され、入力レコードの処理方法(discreteモード、sourceモード、streamモードのいずれか)を決定します。入力ポートからの読取り(p.300)を参照してください。
出力	0	はい	正常に読み取られたレコード。	任意。
	1-n	いいえ(マッピングで必要な場合は追加の出力ポートを接続)	正常に読み取られたレコード。	任意。出力ポートごとに異なるメタデータを保持できます。

JSONReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	読み取られるデータ・ソース(JSONファイル、ディクショナリまたはポート)を指定します。リーダーにサポートされているファイルURL形式(p.297)および注意および制限(p.386)を参照してください。	
Charset		読み取られたレコードのエンコーディング。JSONは自動的にUTF-*エンコーディングのファミリを認識します(Auto)。入力で別のキャラクタ・セットを使用する場合は、この属性で手動でそのキャラクタ・セットを明示的に指定します。	Auto (デフォルト) | <other encodings>
Data policy		エラーの発生時に必要な処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Controlled | Lenient
Mapping URL	1)	マッピング定義が含まれる外部テキスト・ファイル。	
Mapping	1)	入力JSON構造から出力ポートへのマッピング。詳細説明(p.383)を参照してください。	
Implicit mapping		デフォルトでは、同じ名前のCloverフィールドにもJSON要素を手動でマップする必要があります。trueに切り替えると、一致する名前についてJSONからCloverへのマッピングが自動的に実行されます。これにより、適切に構造化された長いJSONファイルでの作業が大幅に軽減されます。JSONマッピング: 詳細(p.383)を参照してください。	false (デフォルト) | true

説明:

1) これらのうちどちらかを指定する必要があります。両方が指定された場合は、「**Mapping URL**」の優先度が高くなります。

詳細説明

JSONはツリー・データの表現であり、各JSONオブジェクトはネストした他のJSONオブジェクトを含むことができます。したがって、**JSONReader**マッピングの作成方法は、XMLおよび他のツリー形式を読み取る場合と同様です。JSONReaderの構成は、XMLXPathReader(p.452)の構成と似ています。マッピングの基本は次のとおりです。

- `<Context>`要素は、マップするJSON構造内の要素を選択します。
- `<Mapping>`要素は、(`<Context>`で選択された)これらのJSON要素をCloverフィールドにマップします。
- どちらもXPath式(p.384)を使用します。

初めてマッピング属性を編集するときは、**マッピング**の説明および例が表示されます。

JSONマッピング：詳細

重要

マッピングの最初の`<Context>`要素は固定形式です。コンポーネントが適切に機能するようにxpathを設定する方法は、次の2つのみです。

`xpath="/root/object"` (JSON構造内のルートがオブジェクトの場合)

`xpath="/root/array"` (JSON構造内のルートが配列の場合)

JSONの例:

```
[
 { "value" : 1},
 { "value" : 2}
]
```

JSONReaderマッピング:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context outPort="0" xpath="/root/array">
   <Mapping cloverField="cloverValue" xpath="value"/>
</Context>
```

(`cloverValue`は出力エッジに割り当てられたメタデータ内のフィールドとみなします)

- 通常の名前と値のペアからデータを読み取るには、必ず、最初にJSON構造内の位置を適切な深さに設定してください。例: `<Context xpath="zoo/animals/tiger">`。

  オプションで、`<Context>`のサブツリーを出力ポートにマップできます。例: `<Context xpath="childObjects" outPort="2">`。

  `<Mapping>`を実行します。xpathで名前と値のペアを選択します。次に、cloverFieldを使用して値をCloverに送信します。例: `<Mapping cloverField="id" xpath="nestedObject">`。

  JSONの例:

```
{
  "property" : 1,
  "innerObject" : {
     "property" : 2
  }
}
```

JSONReaderマッピング:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context outPort="0" xpath="/root/object">
   <Mapping cloverField="property" xpath="property"/>
   <Context xpath="innerObject">
      <Mapping cloverField="propertyOfInnerObject" xpath="property"/>
   </Context>
</Context>
```

- XPath式: @記号は存在しないため、属性へのアクセスに@記号を使用しないでください。特定の値を含むオブジェクトを選択するには、次のような方法でマッピングを記述します。

```
<Context xpath="//website[uri='http://www.w3.org/']" outPort="1">
   <Mapping cloverField="dateUpdated" xpath="dateUpdated" />
   <Mapping cloverField="title" xpath="title"/>
</Context>
```

例に示されたXPathは、uriが指定した文字列と一致するすべての要素websiteを選択します(JSON内での深さは関係ありません)。次に、その2つの要素(dateUpdatedおよびtitle)をポート1の各メタデータ・フィールドに送信します。

前述のように、JSONは内部的にXML DOMに変換されます。有効なXML要素名でないJSON名もあるため、名前はエンコードされます。無効な文字は、_xHHHHという形式(HHHHは16進数のUnicodeコード・ポイント)のエスケープ・シーケンスに置き換えられます。したがって、これらのシーケンスをJSONReaderのXPath式でも使用する必要があります。

XPathのname()関数は、JSONオブジェクトのプロパティの名前を読み取るためにも使用できます(ノードのXPath関数の詳細は、http://www.w3schools.com/xpath/xpath_functions.asp#nodeを参照してください)。ただし、前述のように、名前にエスケープ・シーケンスが含まれる場合もあります。JSONReaderには、これらを処理する2つの関数が用意されています。これらの関数は、http://www.cloveretl.com/ns/TagNameEncoder名前空間から使用可能です。この名前空間は、後で示すようにnamespacePaths属性を使用して宣言する必要があります。一方はdecode(string)関数であり、_xHHHHエスケープ・シーケンスをデコードするために使用できます。もう一方はencode(string)関数であり、無効な文字をエスケープします。

たとえば、次の構造を処理してみます。

```
{"map" : { "0" : 2 , "7" : 1 , "16" : 1 , "26" : 3 , "38" : 1 }}
```

適切なマッピングは次のようになります。

```
 <Context xpath="/root/object/map/*" outPort="0" namespacePaths='
tag="http://www.cloveretl.com/ns/TagNameEncoder"'>
   <Mapping cloverField="key" xpath="tag:decode(name())" />
   <Mapping cloverField="value" xpath="."/>
 </Context>
```

このマッピングでは、"map"のプロパティの名前("0"、"7"、"16"、"26"および"38")をフィールド"key"にマップし、それらの値(それぞれ2、1、1、3および1)をフィールド"value"にマップします。

- **Implicit mapping**: コンポーネントの属性を`true`に切り替えると、JSON構造から同じ名前のフィールドにマッピングするため多くの領域を節約できます。

```
<Mapping cloverField="salary" xpath="salary"/>
```

自動的に実行されます(つまり、前述のマッピング・コードは記述しません)。

配列の処理

- 繰り返しとなりますが、JSON構造はオブジェクトまたは配列によってラップされます。したがって、マッピングは2つの方法のいずれかで開始する必要があります(JSONマッピング: 詳細(p.383)を参照してください)。

```
<Context xpath="/root/object">
```

```
<Context xpath="/root/array">
```

- ネストした配列: 複数の配列が相互にネストされている場合、(最上位の配列の) 1つの名前を繰り返し使用することにより、内側の配列の値に到達できます。したがって、XPathでは、ネストした配列の数に応じて、`arrayName/arrayName/.../arrayName`のような構成メンバーを記述します。例:

  JSON:

```
{
  "commonArray" : [ "hello" , "hi" , "howdy" ],
  "arrayOfArrays" : [ [ "val1", "val2", "val3"] , [""], [ "val5", "val6" ] ]
}
```

JSONReaderマッピング:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context xpath="root/object">

   <Context xpath="commonArray" outPort="0">
      <Mapping xpath="." cloverField="field1"/>
   </Context>

   <Context xpath="arrayOfArrays/arrayOfArrays" outPort="1">
      <Mapping xpath="." cloverField="field2"/>
   </Context>

</Context>
```

マッピングでのドット(p.437)の使用方法に注意してください。このマッピングでのみ、ポート1で予期した結果を生成できます。

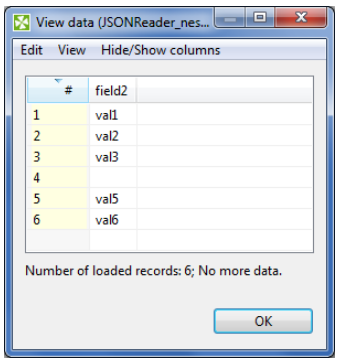


図53.9. ネストした配列のマッピングの例: 結果

- 配列内のNullおよび空の要素: 図53.9「ネストした配列のマッピングの例: 結果」(p.386)では、配列内の空の文字列([""])によって、フィールドに空の文字列が移入されています(図のレコード4)。

  一方、Null値([])は完全にスキップされます。**JSONReader**は、これらがソース内にないものとして処理します。

注意および制限

- **JSONReader**は、ファイル、ディクショナリまたはポートに含まれるJSONからデータを読み取ります。**ポートまたはディクショナリから読み取る**場合は、常に明示的に**キャラクタ・セット**を設定します(そうしないとエラーが発生します)。自動検出は行われません。
- メタデータにアンダースコア「_」が含まれる場合、警告が表示されます。JSONReader**マッピング**では、アンダースコアは無効な文字です。次のいずれかを行う必要があります。

  a) この文字を削除します。

  b) ダッシュ「-」などで置き換えます。

  c) アンダースコアを対応するUnicode表記(_x005f)で置き換えます。

LDAPReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

LDAPReaderは、LDAPディレクトリから情報を読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
LDAPReader	LDAPディレクトリ・ツリー	0	1-n	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します。詳細は、変換の戻り値(p.283)を参照してください。

概要

LDAPReaderは、LDAPディレクトリから情報を読み取ってCloverデータ・レコードに変換します。検索結果を抽出するためのロジックを提供し、検索結果をCloverデータ・レコードに変換します。検索結果には同じ`objectClass`が含まれる必要があります。

文字列およびバイトのCloverデータ・フィールドのみがサポートされています。文字列は、ldapの通常のタイプほとんどと互換性があり、バイトは、`userPassword`のldapタイプの読取りなどの場合に必要です。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	はい	正しいデータ・レコード用。	任意[1]
	1-n	いいえ	正しいデータ・レコード用。	出力0

説明:

1): 出力のメタデータは、読取りオブジェクトの構造を正確に記述している必要があります。メタデータは自動入力関数(p.132)を使用できます。

LDAPReaderの属性

属性	必須	説明	可能な値
Basic			
LDAP URL	はい	ディレクトリのLDAP URL。	`ldap://host:port/`
Base DN	はい	ベース識別名(LDAPツリーのルート)。そのディレクトリを含む場所を参照する属性=値のペアのカンマ区切りリストです。たとえば、`ou=Humans,dc=example,dc=com`が検索対象のサブツリーのルートである場合、*example.com*ドメインからの人を表すエントリが検索されます。	
Filter	はい	検索のフィルタ条件としての属性=値ペア。フィルタと一致するすべてのエントリが返されます。たとえば、`mail=*`では、電子メール・アドレスを持つすべてのエントリが返されます。すべてのエントリには*objectclass*の値が含まれるため、`objectclass=*`は特定のベースおよびスコープと一致するすべてのエントリを返すための標準の方法です。	
Scope		検索リクエストのスコープ。デフォルトでは、1つのみのオブジェクトが検索されます。`onelevel`の場合は識別名のすぐ下のレベルが検索され、`subtree`の場合は識別名の下のサブツリー全体が検索されます。	object (デフォルト) | onelevel | subtree
User		LDAPディレクトリへの接続時に使用されるユーザーDN。`cn=john.smith,dc=example,dc=com`のようになります。	
Password		LDAPディレクトリへの接続時に使用されるパスワード。	
Advanced			
Multi-value separator		**LDAPReader**は、複数値を含むキーを処理できます。これらは、この文字列または文字によって区切られます。<none>は、この機能をオフにする特殊なエスケープ値であり、指定すると最初の値のみが読み取られます。この属性は、文字列データ型のみに使用できます。バイト型が使用される場合、最初の値のみが読み取られます。	"|" (デフォルト) | その他の文字または文字列
Alias handling		別名(名前空間内の別のオブジェクトを指すリーフ・エントリ)の逆参照方法を制御します。	always | never | finding (デフォルト)| searching
Referral handling		デフォルトでは、他のサーバーへのリンクは無視されます。`follow`の場合、参照が処理されます。	ignore (デフォルト) | follow

詳細説明

- **Alias handling**

  別名エントリが指すエントリを検索することを、別名の逆参照と呼びます。「**Alias handling**」属性を設定すると、エントリをどの程度まで検索するかを制御できます。

  - `always`: 常に別名を逆参照します。
  - `never`: 別名を逆参照しません。
  - `finding`: 検索ベースを見つける場合は別名を逆参照しますが、ベースの下位部分を検索する場合は逆参照しません。
  - `searching`: ベースの下位部分を検索する場合は別名を逆参照しますが、ベースを見つける場合は逆参照しません。

ヒントおよびポイント

- 検索パフォーマンスの改善: LDAPディレクトリ内に逆参照を必要とする別名エントリが存在しない場合、**別名処理**の`never`オプションを選択します。

LotusReader

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、第53章「リーダー」(p.339)を参照してください。

要約

LotusReaderは、**Lotus Dominoサーバー**からデータを読み取ります。データは**ビュー**から読み取られます。このとき、各ビュー・エントリが単一のデータ・レコードとして読み取られます。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
LotusReader	Lotus Domino	0	1	✖	✖	✖	✖	✖	✖

1)各データ・レコードをすべての接続済出力ポートに送信します。
2)変換の戻り値(p.283)に従って、データ・レコードを出力ポートに送信します。

概要

LotusReaderは、Lotusデータベースからデータ・レコードを読み取ることができるコンポーネントです。読取りは、**Lotus Dominoサーバー**に格納されているデータベースに接続することによって行われます。

データは、Lotusで**ビュー**と呼ばれるものから読み取られます。ビューにより、Lotusデータベース内のデータが表構造となっています。ビューの各行は、**LotusReader**コンポーネントにより単一のデータ・レコードとして読み取られます。

このコンポーネントのユーザーは、Lotusに接続するためのJavaライブラリを指定する必要があります。このライブラリは、Lotus NotesおよびLotus Dominoのインストール内にあります。**LotusReader**コンポーネントは、このライブラリへのパスが指定されていない場合、またはライブラリがユーザーのクラスパスに配置されていない場合はLotusと通信できません。ライブラリへのパスは、Lotus接続の詳細で指定できます(第25章「Lotus接続」(p.191)を参照してください)。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	✔	読取りデータ・レコード用	

LotusReaderの属性

属性	必須	説明	可能な値
Basic			
Domino connection	✔	Lotus Dominoデータベースへの接続のID。	
View	✔	読み取るデータ・レコードが含まれるLotusデータベース内のビューの名前。	
Advanced			
Multi-value read mode	✘	Lotus複数値フィールドの読取りに使用される読取り方式。複数値の最初のフィールドのみが読み取られるか、すべての値が読み取られてから、ユーザー指定のセパレータにより区切られます。	Read all values (デフォルト) | Read first value only
Multi-value separator	✘	複数値Lotusフィールドからの値を区切るために使用される文字列。	";" (デフォルト) | "," | ":" | "|" | "\t" | その他の文字または文字列

MultiLevelReader

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

MultiLevelReaderは、不均一な構造を持つフラット・ファイルからデータを読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1)]	各出力に送信[2)]	変換	変換が必要	Java	CTL
MultiLevelReader	フラット・ファイル	1	1-n	いいえ	はい	はい	はい	はい	いいえ

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します。詳細は、変換の戻り値(p.283)を参照してください。

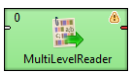
概要

MultiLevelReaderは、不均一で複雑な構造を持つフラット・ファイル(デリミタ付き、固定長または混合であるローカルまたはリモートのファイル)から情報を読み取ります。圧縮フラット・ファイル、コンソール、入力ポートまたはディクショナリからもデータを読み取ることができます。

UniversalDataReaderや他の2つの非推奨リーダー(**DelimitedDataReader**および**FixLenDataReader**)とは異なり、**MultiLevelReader**は、デリミタ付きデータ・レコードと固定長データ・レコードの両方、異なる数のフィールド、異なるデータ型など、構造が異なるフラット・ファイルからデータを読み取ることができます。異なるタイプのデータ・レコードを区別し、異なる接続済出力ポートを介して送信できます。入力ファイルには非レコード・データが含まれることもあります。

このコンポーネントは、「**Data policy**」オプションも使用します。詳細は、データ・ポリシー(p.306)を参照してください。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	ポート読取りに使用。入力ポートからの読取り(p.300)を参照してください。	1つのフィールド(byte、cbyte、string)。
出力	0	はい	正しいデータ・レコード用。	任意(Out0)[1)]
	1-N	いいえ	正しいデータ・レコード用。	任意(Out1-OutN)[1)]

説明:

1) すべての出力ポート上のメタデータは、自動入力関数(p.132)を使用できます。注意: source_timestamp関数およびsource_size関数は、ファイルから直接読み取る場合のみ機能します(ファイルがアーカイブである場合またはリモートの場所に保存されている場合は、タイムスタンプが空になり、サイズは0になります)。

MultiLevelReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	読み取るデータ・ソースを指定する属性(フラット・ファイル、コンソール、入力ポート、ディクショナリ)。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Charset		読み取られたレコードのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Data policy		エラーの発生時に必要な処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Lenient
Selector code	1)	Javaでグラフに記述された、入力データ・ファイルの行からデータ・レコードへの変換。	
Selector URL	1)	Javaで記述された入力データ・ファイルの行からデータ・レコードへの変換を定義する外部ファイルの名前(パスを含む)。	
Selector class	1)	入力データ・ファイルの行からデータ・レコードへの変換を定義する外部クラスの名前。	PrefixMultiLevelSelector (デフォルト) | 他のクラス
Selector properties		全体を中カッコで囲む場合に、セミコロンで区切られるkey=valueの式のリスト。各値は、データ・レコードの送信に使用するポートの番号です。各キーはフラット・ファイルに含まれる行の先頭からの一連の文字であり、レコードのグループの区別を可能にします。	
Advanced			
Number of skipped records		すべてのソース・ファイルにわたって継続的にスキップされるレコード数。入力レコードの選択(p.305)を参照してください。	0-N
Max number of records		すべてのソース・ファイルにわたって継続的に読み取られる最大レコード数。入力レコードの選択(p.305)を参照してください。	0-N

属性	必須	説明	可能な値
Number of skipped records per source		各ソース・ファイルからスキップするレコード数。入力レコードの選択(p.305)を参照してください。	メタデータ内と同じ(デフォルト) | 0-N
Max number of records per source		各ソース・ファイルから読み取るレコードの最大数。入力レコードの選択(p.305)を参照してください。	0-N

説明:

1): この3つの属性を定義しなかった場合は、デフォルトの「Selector class」(`PrefixMultiLevelSelector`)が使用されます。

`PrefixMultiLevelSelector`クラスは`MultiLevelSelector`インタフェースを実装します。インタフェース・メソッドは次の場所で確認できます。

詳細は、MultiLevelReader用Javaインタフェース(p.394)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

詳細説明

Selector Properties

使用する必要のある一連のパラメータを設定する必要もあります(**「Selector properties」**)。これらは、個々のデータ・レコード・タイプを出力ポートにマップします。すべてのプロパティは、セミコロンで区切られた`key=value`式のリストにする必要があります。シーケンス全体を中カッコで囲みます。**「Selector properties」**を指定するには、この属性行内のボタンをクリックすると開くダイアログを使用できます。このダイアログで**プラス・**ボタンをクリックすることによって、新しいキーと値のペアを追加できます。後は、デフォルト名とデフォルト値の両方を変更するだけです。各値は、データ・レコードの送信に使用するポートの番号にする必要があります。各キーはフラット・ファイルに含まれる行の先頭からの一連の文字であり、レコードのグループの区別を可能にします。

MultiLevelReader用Javaインタフェース

`MultiLevelSelector`インタフェースのメソッドを次に示します。

- `int choose(CharBuffer data, DataRecord[] lastParsedRecords)`

  このメソッドは、読み取るレコードのメタデータを特定して、`init()`メソッドで指定されたメタデータ・プールに索引を返すことができるまで、または`MultiLevelSelector.MORE_DATA`を返すデータがなくなるまで`CharBuffer`を検索します。

- `void finished()`

  すべての入力データ・レコードが処理された後のセレクタ処理の最後に呼び出されます。

- `void init(DataRecordMetadata[] metadata, Properties properties)`

  このセレクタを初期化します。

- `int lookAheadCharacters()`

  (次の)レコード・タイプを決定するために必要な文字数を返します。通常は任意の固定文字数にできますが、前のレコード・タイプによっては動的先読みサイズがサポートされ、可能な場合には推奨されます。

- `int nextRecordOffset()`

  `choose()`の各コールは、`choose()`メソッドで返されたメタデータに従ってレコードを解析する前に特定の文字数をスキップするように親に指示できます。

- `void postProcess(int metadataIndex, DataRecord[] records)`

  このメソッドでは、セレクタは解析済レコードを変更してから、対応する出力ポートに送信できます。

- `int recoverToNextRecord(CharBuffer data)`

  このメソッドは、解析できる可能性のある次のレコードのオフセットを見つけるようにセレクタに指示します。

- `void reset()`

  このセレクタを完全にリセットします。このメソッドは、グラフの各実行前に1回呼び出されます。

- `void resetRecord()`

  セレクタの内部状態(ある場合)をリセットします。このメソッドは、新しい選択を行う必要があるたびに呼び出されます。

ParallelReader

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

ParallelReaderは、複数のスレッドを使用してフラット・ファイルからデータを読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1)]	各出力に送信[2)]	変換	変換が必要	Java	CTL
ParallelReader	フラット・ファイル	0	1-2	✖	✖	✖	✖	✖	✖

1)コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2)コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します。詳細は、変換の戻り値(p.283)を参照してください。

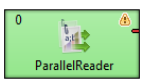
概要

ParallelReaderは、CSVやタブ区切りなどのデリミタ付きフラット・ファイル、固定長ファイルまたは混合テキスト・ファイルを読み取ります。複数のパラレル・スレッドで行われるため、読取りがより高速になります。入力ファイルはチャンク・セットに分割され、各読取りスレッドはファイルのこの部分からのレコードのみを解析します。このコンポーネントは、単一ファイルに加え、ローカル・ディスクまたはリモートに置かれたファイルのコレクションを読取り可能です。リモート・ファイルにはFTPプロトコルを介してアクセスできます。

コンポーネント設定およびデータ構造に応じて、高速の簡易パーサー(`SimpleDataParser`)または堅牢なパーサー(`CharByteDataParser`)が使用されます。

解析されたデータ・レコードは1つ目の出力ポートに送信されます。このコンポーネントには、正しくないレコードに関する詳細情報を取得するための、オプションの出力ロギング・ポートがあります。データ・ポリシー(p.306)が`controlled`に設定され、適切な**ライター**(**Trash**または**UniversalDataWriter**)がポート1に接続されている場合にのみ、すべての正しくないレコードが、正しくない値、場所およびエラー・メッセージに関する情報とともに、このエラー・ポートを介して送信されます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	✔	正しいデータ・レコード用。	任意[1)]
	1	✖	正しくないデータ・レコードに使用	特定の構造、下の表を参照

1)出力ポート上のメタデータは、自動入力関数(p.132)を使用できます。

表53.2. Parallel Readerのエラー・メタデータ

フィールド番号	フィールドの内容	データ型	説明
0	レコード番号	integer	データセット内のエラーのあるレコードの場所(レコード番号は1から始まります)
1	フィールド番号	integer	レコード内のエラーのあるフィールドの場所(1は最初のフィールド、つまりインデックス0のフィールドを表します)
2	RAWレコード	string	エラーのあるRAW形式のレコード(デリミタを含む)
3	エラー・メッセージ	string	エラー・メッセージ: このエラーに関する詳細情報
4	最初のレコード・オフセット	long	解析スレッドの初期ファイル・オフセットを示します。

ParallelReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	✔	読み取られるデータ・ソースを指定する属性。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Charset		読み取られたレコードのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Data policy		エラーの発生時に必要な処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Controlled | Lenient
Trim strings		文字列をデータ・フィールドに設定する前に、先頭および末尾のスペースを削除するかどうかを指定します。データのトリミング(p.418)を参照してください。`true`の場合は、堅牢なパーサーが強制的に使用されます。	false (デフォルト) | true
Quoted strings		特殊文字(カンマ、改行または二重引用符)が含まれるフィールドは、引用符で囲む必要があります。一重引用符および二重引用符のみが引用符文字として受け入れられます。`true`の場合、コンポーネントによる読取り時に特殊文字が削除されます(デリミタとして扱われません)。 例: 入力データ`"25"|"John"`を読み取るには、「**Quoted strings**」を`true`に切り替えて「**Quote character**」を「"」に設定	false | true

属性	必須	説明	可能な値
		します。これにより、`25|John`という2つのフィールドが生成されます。 デフォルトでは、この属性の値は出力ポート0のメタデータから継承されます。レコード詳細(p.162)も参照してください。	
Quote character		「**Quoted strings**」で許可される引用符の種類を指定します。デフォルトでは、この属性の値は出力ポート0のメタデータから継承されます。レコード詳細(p.162)も参照してください。	both | " | '
Advanced			
Skip leading blanks		入力文字列をデータ・フィールドに設定する前に、先頭のスペース(空白など)をスキップするかどうかを指定します。明示的に設定されない場合(デフォルト値のまま)は、「**Trim strings**」属性の値が使用されます。データのトリミング(p.418)を参照してください。`true`の場合は、堅牢なパーサーが強制的に使用されます。	false (デフォルト) | true
Skip trailing blanks		入力文字列をデータ・フィールドに設定する前に、末尾のスペース(空白など)をスキップするかどうかを指定します。明示的に設定されない場合(デフォルト値のまま)は、「**Trim strings**」属性の値が使用されます。データのトリミング(p.418)を参照してください。`true`の場合は、堅牢なパーサーが強制的に使用されます。	false (デフォルト) | true
Max error count		入力ファイル内で許容されるエラー・レコードの最大数。「**Data policy**」が`Controlled`に設定されている場合のみ適用可能です。	0 (デフォルト) - N
Treat multiple delimiters as one		`true`に設定すると、フィールドが複数のデリミタ文字で区切られている場合に単一のデリミタとして解釈されます。	false (デフォルト) | true
Verbose		デフォルトでは、エラー通知が簡略化され、パフォーマンスが多少高くなります。`true`に切り替えた場合は、情報が詳細になり、パフォーマンスが低下します。	false (デフォルト) | true
Level of parallelism		入力データ・ファイルの読取りに使用されるスレッド数。これが2以上の場合、レコードの順序は保持されません。ファイルが小さすぎる場合、この値は自動的に1に切り替わります。	2 (デフォルト) | 1-n
Distributed file segment reading		コンポーネントが**CloverETL Server**の**クラスタ**環境で実行されている場合、共有ファイルが読み取られると、各コンポーネントのインスタンスによってファイルの適切な部分が処理されます。**CloverETL Server**によってファイル全体がセグメントに分割され、各クラスタ・ワーカーによってファイルの適切な部分が1つのみ処理されます。デフォルトでは、このオプションはオフです。パーティション・ファイルの場合、この属性は無視されます。	false (デフォルト) | true
Parser		デフォルトでは、最も適切なパーサーが適用されます。さらに、データ処理用のパーサーを明示的に設定できます。不適切なものが設定された場合は、例外がスローされ、グラフが失敗します。データ・パーサー(p.418)を参照してください。	auto (デフォルト) | *<other>*

詳細説明

- **Quoted strings**

この属性では、データの解析方法が大幅に変わります。これを`true`に設定した場合、引用符付き文字列内のすべてのフィールド・デリミタは無視されます(最初の**引用符文字**が実際に読み取られた後)。引用符文字はフィールドから削除されます。

入力例:

```
1;"lastname;firstname";gender
```

「Quoted strings」が「true」の場合の出力:

{1}, {`lastname;firstname`}, {`gender`}

「Quoted strings」が「false」の場合の出力:

{1}, {`"lastname`}, {`firstname";gender`}

QuickBaseRecordReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

QuickBaseRecordReaderは、**QuickBase**オンライン・データベースからデータを読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
QuickBaseRecordReader	QuickBase	0-1	1-2	✖	✖	✖	✖	✖	✖

1) 各データ・レコードをすべての接続済出力ポートに送信します。
2) 変換の戻り値(p.283)に従って、データ・レコードを出力ポートに送信します。

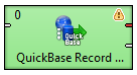
概要

QuickBaseRecordReaderは、**QuickBase**オンライン・データベース(http://quickbase.intuit.com)からデータを読み取ります。IDが「**Records list**」コンポーネント属性に指定されているレコードが最初に読み取られます。その後、入力にIDが指定されているレコードが読み取られます。

最初に接続された出力ポートを介して、読み取られたレコードが送信されます。レコードにエラーがある場合(データベース表に存在しないなど)は、オプションの2つ目のポートを介して送信できます(そのポートが接続済の場合)。

このコンポーネントは、`API_GetRecordInfo` HTTP対話をラップします(http://www.quickbase.com/api-guide/getrecordinfo.html)。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✖	読み取るアプリケーション表レコードIDの取得用。	最初のフィールド: `integer | long`
出力	0	✔	正しいデータ・レコード用。	フィールドのデータ型および位置は、表フィールド・タイプに適合している必要があります[1]。
	1	✖	拒否されたレコードに関する情報。	表53.3 QuickBaseRecordReaderのエラー・メタデータ(p.401)[2]

1レコードIDを返すsource_row_count自動入力関数のみを使用できます。
2エラー・メタデータは自動入力関数(p.132)を使用できません。

表53.3 QuickBaseRecordReaderのエラー・メタデータ

フィールド番号	フィールド名	データ型	説明
0	<any_name1>	integer | long	エラーのあるレコードのID
1	<any_name2>	integer | long	エラー・コード
2	<any_name3>	string	エラー・メッセージ

QuickBaseRecordReaderの属性

属性	必須	説明	可能な値
Basic			
QuickBase connection	✔	QuickBaseオンライン・データベースへの接続のID。第24章「QuickBase接続」(p.190)を参照してください。	
Table ID	✔	データ・レコードが読み取られる、QuickBaseアプリケーションの表のID (表IDを取得するには`application_stats`を確認してください)。	
Records list		指定したデータベース表から読み取られる(セミコロンで区切られた)レコードIDのリスト。入力データに指定されたレコードよりも前に、まずこれらのレコードが読み取られます。	

QuickBaseQueryReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

QuickBaseQueryReaderは、与えられた条件を満たすレコードを**QuickBase**オンライン・データベース表から取得します。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
QuickBaseQueryReader	QuickBase	0-1	1-2	✖	✖	✖	✖	✖	✖

1) 各データ・レコードをすべての接続済出力ポートに送信します。

2) 変換の戻り値(p.283)に従って、データ・レコードを出力ポートに送信します。

概要

QuickBaseQueryReaderは、**QuickBase**オンライン・データベース(http://quickbase.intuit.com)からデータを取得します。コンポーネント属性を使用して、どの列を返すか、レコードをいくつ返しどのようにソートするか、およびQuickBaseで構造化データを返すかどうかを定義できます。要件を満たすレコードが、接続済出力ポートを介して送信されます。

このコンポーネントは、`API_DoQuery` HTTP対話をラップします(http://www.quickbase.com/api-guide/do_query.html)。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	✔	正しいデータ・レコード用。	任意[1]

[1]メタデータは自動入力関数(p.132)を使用できません。

QuickBaseQueryReaderの属性

属性	必須	説明	可能な値
Basic			
QuickBase connection	✔	QuickBaseオンライン・データベースへの接続のID。第24章「QuickBase接続」(p.190)を参照してください。	
Table ID	✔	データ・レコードの取得元となる、QuickBaseアプリケーションの表のID (表IDを取得するには`application_stats`を確認してください)。	
Query		{<*field_id*>.<*operator*>.'<*matching_value*>'}という形式を使用して、どのレコードを返すかを決定します(デフォルトではすべて)。	
CList		*列リスト*には、返される各レコードにどの列を含め、返されたレコードの集計にその列をどの順序で並べるかを指定します。ピリオドで区切られた*field_id*を使用します。	
SList		ソート・リストでは、返されたレコードの表示順序を決定します。ピリオドで区切られた*field_id*を使用します。	
Options		読み取られるデータ・レコードに使用されるオプション。詳細はオプション(p.403)を参照してください。	

詳細説明

Options

オプションの属性は次のとおりです。

- `skp-n`

  先頭からnレコードをスキップすることを指定します。

- `num-n`

  nレコードを読み取ることを指定します。

- `sortorder-A`

  ソート順序を昇順として指定します。

- `sortorder-D`

  ソート順序を降順として指定します。

- `onlynew`

  匿名ユーザーではこのパラメータを使用できません。このコンポーネントは新しいレコードのみを読み取ります。結果はユーザーによって異なることがあります。

SpreadsheetDataReader

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

SpreadsheetDataReaderは、Excelスプレッドシート(XLSファイルまたはXLSXファイル)からデータを読み取ります。**SpreadsheetDataReader**は元の**XLSDataReader**に代わるものであり、より多くの新機能と読取りモードを備え、パフォーマンスが改善されています。(**XLSDataReader**には引き続き下位互換性があり、**Community**エディションで使用可能です)

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信	各出力に送信	変換	変換が必要	Java	CTL
SpreadsheetDataReader	XLS(X)ファイル	0–1	1–2	いいえ	はい	いいえ	いいえ	いいえ	いいえ

概要

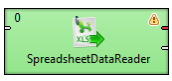
SpreadsheetDataReaderは、XLSファイルまたはXLSXファイルの指定のシートからデータを読み取ります。複雑なデータ・マッピングが可能です(フォーム、表、複数行レコードなど)。他のリーダーと同様に、すべての標準入力オプション(ローカル・ファイルとリモート・ファイル、zipアーカイブ、入力ポートまたはディクショナリ)を使用できます。

サポートされているファイル形式は次のとおりです。

- XLS: Excel 97/2003 XLSファイルのみがサポートされています(BIFF8)。
- XLSX: Microsoft Excel 2007以降のオープンなドキュメント形式です。

XLSXでは1,048,576行を超えるファイルも読み取ることができますが、公式にはサポートされていません(Excelでは2^20を超えた行は表示されません)。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	オプションのポート読取りに使用。入力ポートからの読取り(p.300)を参照してください。	1つのフィールド(`byte`、`cbyte`、`string`)。
出力	0	はい	正常に読み取られたレコード。	任意[1)]
	1	いいえ	エラー・レコード。	ポート0からの固定デフォルト・フィールド+オプションのフィールド。[2)]

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

説明

1) メタデータは自動入力関数(p.132)を使用できます。注意: `source_timestamp`関数および`source_size`関数は、ファイルから直接読み取る場合のみ機能します(ファイルがアーカイブである場合またはリモートの場所に保存されている場合は、タイムスタンプが空になり、サイズは0になります)。

2) データ・ポリシー(p.306)が**Controlled**に設定され、ポートにエッジが接続されている場合(エッジがないとメッセージがコンソールに記録されます)、正しく読み取ることができなかったレコードが出力ポート1に送信されます。レコードの障害の原因となったエラーの理由と位置について説明する一連の固定フィールドがあります。また、ポート0から任意のフィールドをマップすることもできます。**入力のエラーごとに1つのエラー・レコードが生成されます。つまり、1つのレコードに複数のエラーがある場合、複数のエラー・レコードが生成されます。それらをたとえば先頭の整数フィールドでグループ化できます。**

表53.4. エラー・ポート・メタデータ: 先頭10個のフィールドに必須のタイプあり、名前は任意

フィールド番号	フィールド名	データ型	説明
0	recordNo	`integer`	誤って読み取られたレコードのインデックス(レコード番号は1から開始)
1	fileName	`string`	エラーが発生したファイルの名前(利用可能な場合)
2	sheetName	`string`	エラーが発生したシートの名前
3	fieldNo	`integer`	データを読み込むことができなかったフィールドのインデックス(ゼロベース)
4	fieldName	`string`	データを読み込むことができなかったフィールドの名前。例: "CustomerID"
5	cellCoords	`string`	読取りのエラーが発生したソース・スプレッドシートのセルの座標。例: "D7"
6	cellValue	`string`	読取りのエラーが発生したセルの値。例: "-5.12"
7	cellType	`string`	読取りのエラーが発生したセルのExcelタイプ。例: "String"
8	cellFormat	`string`	読取りのエラーが発生したセルのExcel書式文字列。例: "#,##0"
9	errMsg	`string`	判読可能な形式のエラー・メッセージ。例: "Cannot get Date value from cell of type String in C1"

SpreadsheetDataReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	読み取られるデータソースを指定します。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Sheet		読み取るシートの名前または番号(ゼロベース)。複数のシートをセミコロン(;)で区切って指定できます。また、?および*のワイルドカードを使用して複数のシートを指定することもできます。シートは、その後、同じマッピングを使用して順に読み取られます。	0 (最初のシートの読取り)
Mapping	1)	ビジュアル・マッピング・エディタでスプレッドシート・セルをCloverフィールドにマップします。詳細説明(p.407)を参照してください。	
Mapping URL	1)	**マッピング**定義が含まれたXMLファイルへのパス。単一のマッピングを複数のグラフで共有する場合は、マッピングを外部ファイルに置きます。	
Data policy		エラーの発生時に実行する処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Controlled | Lenient
Advanced			
Read mode		入力ファイルからのデータの読取り方法を決定します。メモリー内モードでは入力ファイル全体がメモリーに格納されるため、読取り速度が向上します。より小さいファイルに適しています。ストリーミング・モードでは、ファイルはメモリーに格納されずに直接読み取られます。このため、ストリーミングを使用すると、メモリーが不足することなくより大きいファイルを読み取ることができます。ストリーミングでは、XLSとXLSXの両方がサポートされています。	メモリー内(デフォルト) | ストリーミング
Number of skipped records		すべてのソース・ファイルにわたってスキップされるレコードの合計数。入力レコードの選択(p.305)を参照してください。	0-N
Max number of records		すべてのソース・ファイルにわたって読み取られるレコードの合計数。入力レコードの選択(p.305)を参照してください。	0-N
Number of skipped records per source		各ソース・ファイルでスキップするレコード数。入力レコードの選択(p.305)を参照してください。	メタデータ内と同じ(デフォルト) | 0-N
Max number of records per source		各ソース・ファイルから読み取るレコードの最大数。入力レコードの選択(p.305)を参照してください。	0-N
Number of skipped records per spreadsheet		各Excelシートでスキップするレコード数。	
Max number of records per spreadsheet		各Excelシートから読み取るレコードの最大数。	
Max error count		グラフが失敗となるまでに許容されるエラーの最大数。「**Data Policy**」の`Controlled`値に適用されます。	0 (デフォルト) | 1-N

属性	必須	説明	可能な値
Incremental file	2)	増分キーを格納しているファイルの名前(パスを含む)。増分読取り(p.304)を参照してください。	
Incremental key	2)	最後に読み取られたレコードの位置を保存します。増分読取り(p.304)を参照してください。	
Encryption password		ソース・スプレッドシートのデータが暗号化されている場合は、ここにパスワードを入力します。大/小文字、特殊文字、アクセント記号付き文字など、すべての文字を正確に入力してください。	

説明:

1) マッピングを定義するには、この2つのいずれかを指定する必要があります。

2) これらの属性の両方を指定するか、両方とも指定しないでください。

詳細説明

スプレッドシート・マッピングの概要

マッピングは、コンポーネントに対して、Excelスプレッドシートを読み取る方法を指定する汎用的なパターンです。マッピング・エディタは常に1つのファイルのスプレッドシートをプレビューしますが、マッピングは類似したファイルのグループ全体に適用できます。

各セルは、次のいずれかのモードでCloverフィールドにマップできます。

- **順序別マップ**は、最も簡単なアプローチです。スプレッドシート・セルは、入力と同じ順序で1つずつ出力フィールドにマップされます。別のメタデータを選択すると、セルは新しいフィールドに自動的に再マップされます。

- **名前別マップ**: マップされた先頭セルごとに、コンポーネントはその内容を読み取り、同じ名前またはラベルを持つ、一致するフィールドを見つけようとします(フィールド名とラベルと説明(p.161)を参照してください)。現在のファイルにマップできなかったフィールドは**unresolved**とマークされます。これらを明示的にマップまたはアンマップするか、または出力メタデータを変更できます。類似した入力ファイルのグループを読み取り、各ファイルに可能な列のサブセットが含まれている場合などには、セルが未解決でも問題ありません。マッピングに未解決のセルがあっても、実行時にグラフは失敗しません。

注意

順序別マップと**名前別マップ**のいずれのモードでも、入力ファイルの内容を出力メタデータに自動的にマップすることが試行されます。このため、これらのモードは、複数のソース・ファイルを読み取り、かつ、単独ですべてに対応する汎用的なマッピングを1つのみ設計するときに役立ちます。

- **明示**: いつでも、任意のフィールドにセルをマップできます。このようにして、たとえば、1つのみのセル(マッピングに適合しない)と順序別マップされたシート全体を、正しいフィールドに明示的にマップできます。単に**選択したセル**に移動し、**フィールド名またはインデックス**に目的のフィールドを入力します。セルがまだマップされていない場合は、まず**マッピング・モード**を**明示**に切り替えることが必要になる場合があります。また、メタデータ・ビューアからセルにフィールドをドラッグして、セルをフィールドに明示的にマップすることもできます。反対方向(フィールドにセルをドラッグ)も機能しますが、選択したセルのみをドラッグできるため、まずセルをクリックして選択する必要があります。一度に複数のフィールド/セルをドラッグ・アンド・ドロップできます。

- **暗黙**: すべての**マッピング**・コンポーネント・プロパティを完全に空白のままにしておく特別なケース。コンポーネントは失敗しませんが、そのかわり最初のスプレッドシート行全体が名前別にマップされ、データ・オフセットが1に等しくなります。マッピング・エディタでどのセルもマップしないときでも、「OK」ボタンをクリックしてマッピングを確定すると、別のタイプの暗黙的マッピングが作成されます。その後、基本的なマッピング・

プロパティのみが「**Mapping**」属性に格納されます。このようにして、前述の基本的な暗黙的マッピングで使用されるデフォルトの「**Rows per record**」または**データ・オフセット**を変更できます(デフォルトのオフセットが0に設定されている場合、名前別マップではなく名前別マップが使用されます)。また、読取りの「**Orientation**」プロパティを切り替えると、先頭の行ではなく先頭の列が暗黙的にマップされます。

スプレッドシート・マッピング・エディタの色

- オレンジのセルは**先頭セル**と呼ばれ、ヘッダーとなります。数多くのマッピング設定を行える場所です。詳細なマッピング・オプション(p.410)を参照してください。
- 黄色のセルは、最初のレコードの先頭を示しています。
- 破線の境界線内のセル(先頭セルの選択後に表示)は、データが取得される領域を示しています。

マッピング・エディタ

マッピングを開始するには、データが含まれているファイルおよびシート名を「**File URL**」属性と「**Sheet**」属性にそれぞれ入力します。その後、**マッピング**を編集してビジュアル・マッピング・エディタを開きます。選択したシートが、次のようにプレビューされます。

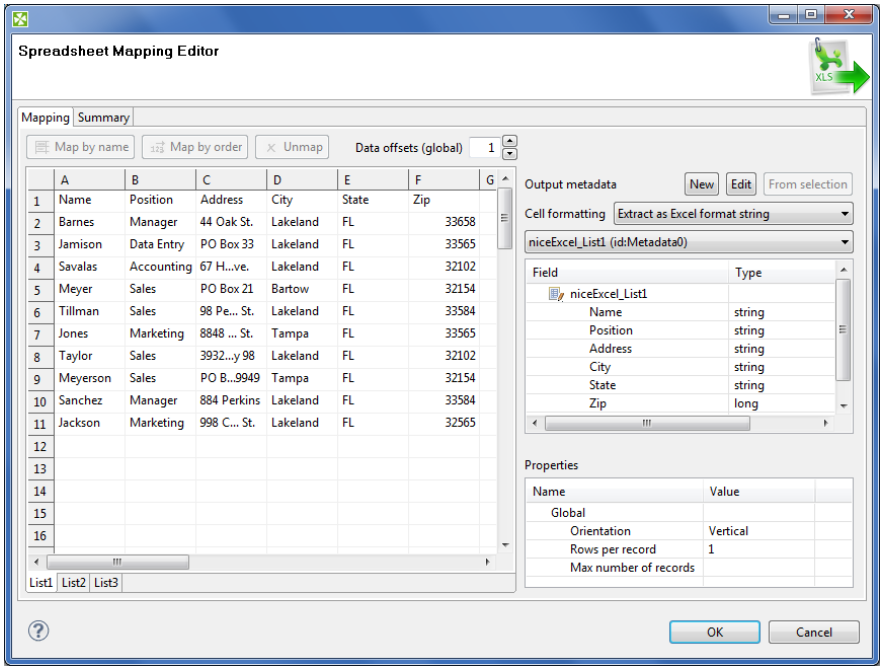


図53.10. SpreadsheetDataReaderマッピング・エディタ

このように、エディタは次の要素で構成されています。

- ツールバー: Excelデータの**マップ**方法(**順序別**または**名前別**)を制御するボタンおよびグローバルな**データ・オフセット**・コントロール(データ・オフセットの詳細は、詳細なマッピング・オプション(p.410)を参照してください)。
- シートのプレビュー領域: ソース・ファイルのすべてのマッピングを実行および参照します。
- 「Output metadata」: Excelセルのマップ先となるCloverフィールドです。
- 「Properties」: ソース・ファイル全体(**Global**)用、または**選択したセル**に関するもののみとなります。
- 「**Summary**」タブ: これまでに実行したスプレッドシートとCloverとのマッピング全体を簡潔に確認できます。

メタデータ

スプレッドシートの読取りを開始する前に、そのメタデータをCloverフィールドとして抽出することが必要になる場合があります(XLS(X)ファイルからのメタデータの抽出(p.144)を参照してください)。抽出ウィザードは、ここで説明したスプレッドシート・**マッピング**・エディタに似ており、同じ原則を使用します。

注意

マッピング・エディタを使用すると、メタデータ抽出ウィザード(スプレッドシート・メタデータのみを取得する必要がある場合に適する)に移動する必要なく、適切な場所にメタデータを抽出できます。

「**Output metadata**」領域では、送信エッジに割り当てられたメタデータを編集できます。出力エッジを接続しなかった場合でも(フィールドを作成すると自動的に作成されます)、マッピング・エディタから適切にメタデータを作成し、操作できます。使用可能な操作は次のとおりです。

- 「Output metadata」コンボを使用して、グラフで既存のメタデータを選択します。
- 「Output metadata」コンボの<new metadata>オプションを使用して、新しいメタデータを作成します。
- 「**Field**」をダブルクリックして名前を変更します。
- コンボ・ボックスを使用してデータ**型**を変更します。
- 出力メタデータをさらに操作するには、「**Edit**」ボタンを使用します。
- メタデータを作成するには、スプレッドシートのプレビュー領域からセルをドラッグし、出力メタデータ・フィールド間にドロップします。

基本マッピングの例

通常、Excelデータは先頭行にヘッダーを含んでおり、このため容易にマップできます。この項では、その方法について説明します。

- まず、「**Properties**」→「**Global**」→「**Orientation**」で「**Vertical**」モードが設定済であることを確認します。これにより、SpreadsheetDataReaderは(列ごとに詳細を読み取る「**Horizontal**」方向とは反対に)行ごとに入力を処理します。
- オプションで(XLS(X)ファイルからのメタデータの抽出(p.144)での説明に従ってメタデータを抽出しなかった場合には)、先頭行を選択し、そのフィールドを「**Output metadata**」ペインにドラッグします。これにより、選択したすべてのセルに対してフィールドが作成されます。タイプは自動的に推測されますが、後で自分で確認することをお薦めします。
- 先頭行全体を選択し(「1」行ヘッダーをクリック)、「**Map by order**」または「**Map by name**」をクリックします(詳細は、スプレッドシート・マッピングの概要(p.407)を参照してください)。

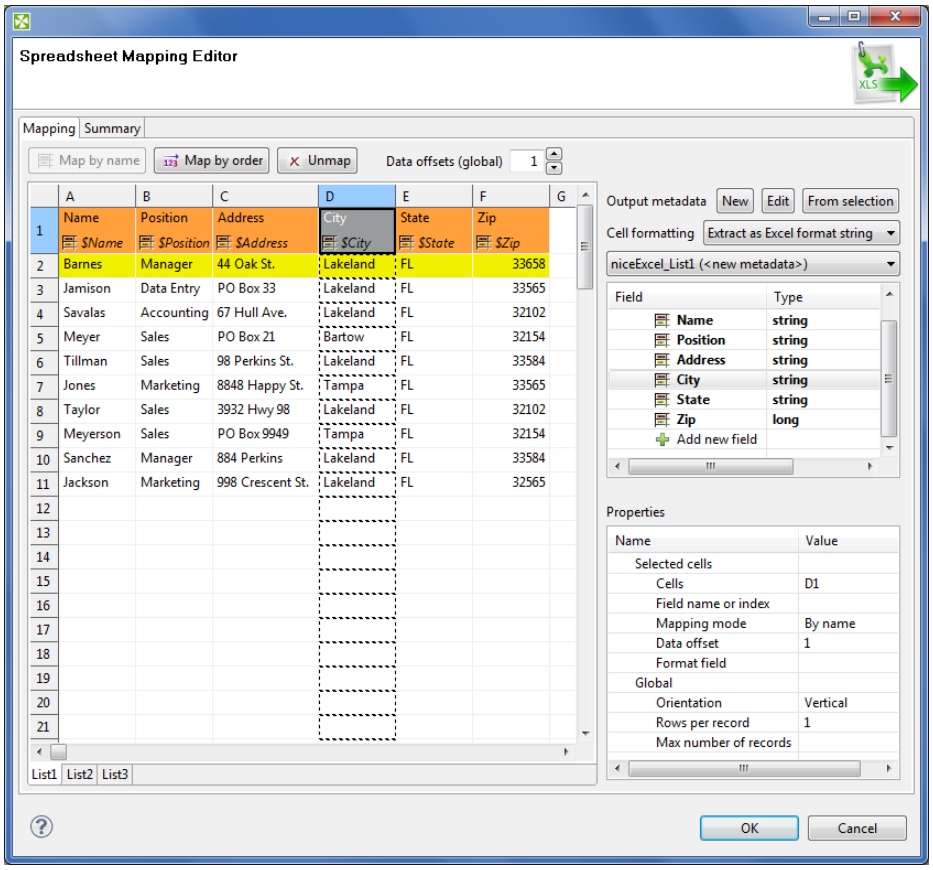


図53.11. 基本マッピング: データの取得先となる領域をマークしている先頭セルおよび破線の境界線を確認してください。

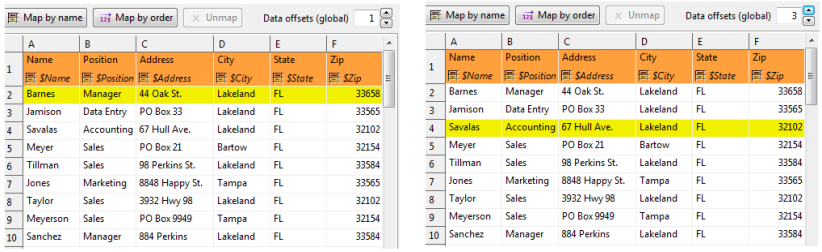
詳細なマッピング・オプション

この項では、基本マッピングの例(p.409)を拡張するその他いくつかの概念について説明します。

- **Data offsets (global)**: データの取得先を決定します。基本的に、その値は、先頭セル(オレンジ)を基準にして、省略する行(垂直モード)または列(水平モード)の数を表します。スプレッドシート全体のデータ・オフセットを調整するには、右上隅の矢印ボタンをクリックします。また、各先頭セル(オレンジ)の「**Selected cells**」領域でスピナー をクリックして、データ・オフセットをローカルに、つまり特定の列のみを調整することもできます。シート・プレビューでデータ・オフセットの変更がどのように可視化されるかを確認してください。つまり、省略した行では色が変化します。先頭セルをクリックすると表示される破線のセルに従って、レコードの開始位置を素早く示すことができます。

ヒント

「**Data offsets (global)**」の矢印ボタンは、各セルのデータ・オフセット・プロパティを上下にシフトするのみです。混合オフセットが保持されているため、必要に応じてシフトします。すべてのデータ・オフセットを1つの値に設定するには、「Data offsets (global)」の数値フィールドに値を入力します。いくつかの混合オフセットが存在する場合は、値がグレーで表示されます。



	A	B	C	D	E	F
1	Name $Name	Position $Position	Address $Address	City $City	State $State	Zip $Zip
2	Barnes	Manager	44 Oak St.	Lakeland	FL	33658
3	Jamison	Data Entry	PO Box 33	Lakeland	FL	33565
4	Savalas	Accounting	67 Hull Ave.	Lakeland	FL	32102
5	Meyer	Sales	PO Box 21	Bartow	FL	32154
6	Tillman	Sales	98 Perkins St.	Lakeland	FL	33584
7	Jones	Marketing	8848 Happy St.	Tampa	FL	33565
8	Taylor	Sales	3932 Hwy 98	Lakeland	FL	32102
9	Meyerson	Sales	PO Box 9949	Tampa	FL	32154
10	Sanchez	Manager	884 Perkins	Lakeland	FL	33584

図53.12. グローバル・データ・オフセットが1 (デフォルト)の場合と3の場合の違い。右側の図では、読取りが行4で開始されます(行2および3のデータは無視されます)。

	A	B	C
1	Name $Name	Position $Position	Address $Address
2	Barnes	Manager	44 Oak St.
3	Jamison	Data Entry	PO Box 33
4	Savalas	Accounting	67 Hull Ave.
5	Meyer	Sales	PO Box 21
6	Tillman	Sales	98 Perkins St.
7	Jones	Marketing	8848 Happy St.
8	Taylor	Sales	3932 Hwy 98
9	Meyerson	Sales	PO Box 9949
10	Sanchez	Manager	884 Perkins

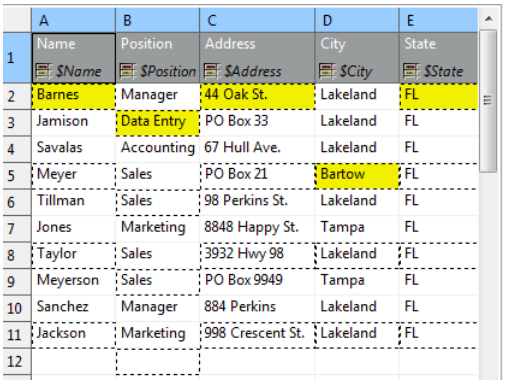
図53.13. グローバル・データ・オフセットがすべての列に対して1に設定されています。3番目の列では、ローカルに3に変更されています。

- **Rows per record**: 1つのレコードを形成する行数を指定する**グローバル・**プロパティ。次の図のような場合の最適な値を推測します。

	A	B	C	D	E	F
1	Name $Name	Position $Position	Address $Address	City $City	State $State	Zip $Zip
2	Barnes	Manager	44 Oak St.	Lakeland	FL	33658
3	Jamison	Data Entry	PO Box 33	Lakeland	FL	33565
4	Savalas	Accounting	67 Hull Ave.	Lakeland	FL	32102
5	Meyer	Sales	PO Box 21	Bartow	FL	32154
6	Tillman	Sales	98 Perkins St.	Lakeland	FL	33584
7	Jones	Marketing	8848 Happy St.	Tampa	FL	33565
8	Taylor	Sales	3932 Hwy 98	Lakeland	FL	32102
9	Meyerson	Sales	PO Box 9949	Tampa	FL	32154
10	Sanchez	Manager	884 Perkins	Lakeland	FL	33584

図53.14.「Rows per record」が4に設定されています。これにより、SpredsheetDataReaderではExcelの4行を取得し、それらのセルから1つのレコードを作成します。実際にレコードのフィールドになるセルは破線の境界線でマークされるため、レコードにはすべてのデータが移入されません。どのセルにレコードが移入されるかは、データ・オフセット設定によっても決定されます。次の箇条書きを参照してください。

- **データ・オフセット**(グローバルおよびローカル)と**「Rows per record」**の組合せ: 前の箇条書きで説明した設定を組み合せることができます。次の例を参照してください。



	A	B	C	D	E
1	Name $Name	Position $Position	Address $Address	City $City	State $State
2	Barnes	Manager	44 Oak St.	Lakeland	FL
3	Jamison	Data Entry	PO Box 33	Lakeland	FL
4	Savalas	Accounting	67 Hull Ave.	Lakeland	FL
5	Meyer	Sales	PO Box 21	Bartow	FL
6	Tillman	Sales	98 Perkins St.	Lakeland	FL
7	Jones	Marketing	8848 Happy St.	Tampa	FL
8	Taylor	Sales	3932 Hwy 98	Lakeland	FL
9	Meyerson	Sales	PO Box 9949	Tampa	FL
10	Sanchez	Manager	884 Perkins	Lakeland	FL
11	Jackson	Marketing	998 Crescent St.	Lakeland	FL
12					

図53.15. 「Rows per record」は3に設定されています。最初の列および3番目の列はそれぞれの先頭行がレコードに反映されます(グローバル・データ・オフセットが1であるため)。2番目の列および4番目の列は、(ローカルな)データ・オフセットがそれぞれ2と4です。このため、最初のレコードはジグザグになったセルで形成されます(この概念をわかりやすくするために黄色で示されています)。

- **Max number of records**: これもコンポーネント属性を介して指定できる**グローバル・プロパティ**です(SpreadsheetDataReaderの属性(p.406)を参照してください)。これを小さくすると、スプレッドシート・プレビューの破線のセルの数も減ります(実際にレコードにマップされるセルのみが強調表示されます)。

- **Format Field**: 読み取ったセルからExcel書式(Excelの右クリック・メニューの「セルの書式設定」と同じ)を取得できます。先頭セルを選択し、(「**Selected cells**」で)「**Format Field**」プロパティを、書式パターンの読取り先となるターゲット・フィールドとして指定します。ターゲット・フィールドは`string`である必要があります。読み取ったデータ・セルに様々な書式(たとえば、様々な通貨)がある場合にも、このアプローチを使用できます。

注意

Excelセルに`General`書式がある場合(一般的なケース)、その書式はExcel内部書式であるためCloverに転送できません。そのかわり、ターゲット・フィールドには文字列Generalが付きます。

Date	Special
2005-06-05	[$-405]d\.\ mmmm\ yyyy;@
1970-01-01	[$-405]d\.\ mmmm\ yyyy;@
1918-09-05	[$-405]d\.\ mmmm\ yyyy;@
2012-12-06	[$-405]d\.\ mmmm\ yyyy;@
2000-01-08	[$-405]d\.\ mmmm\ yyyy;@
2050-12-09	[$-405]d\.\ mmmm\ yyyy;@
2020-09-09	[$-405]d\.\ mmmm\ yyyy;@
2001-01-14	[$-405]d\.\ mmmm\ yyyy;@
1985-06-02	[$-405]d\.\ mmmm\ yyyy;@
1999-01-01	[$-405]d\.\ mmmm\ yyyy;@

図53.16. 日付フィールドからの書式の取得。「Format Field」は、ターゲットとして「Special」フィールドに設定されました。

書式は、1回かぎりのメタデータ抽出プロセスの間に抽出することもできます。メタデータでは、書式はサンプル値としてメタデータ抽出ウィザードに指定した単一のセルから取得されます。XLS(X)ファイルからのメタデータの抽出(p.144)を参照してください。

セルの書式がExcel書式文字列(`excel:`)で指定されている場合、**SpreadsheetDataReader**はそれを読み直すことができます。その他のリーダーはそれを無視します。書式文字列の読取りの詳細は、セルの書式設定(Format Field)(p.540)を参照してください。

- 1列に複数の先頭セル: スプレッドシートによっては、1つの列で混在してしまったデータをすべて1つのレコードとして処理することが必要な場合があります。たとえば、奇数行に名、偶数行に姓が順に含まれている列があるとします。その場合、名と姓の両方を読み取ることができるように、それぞれに対応する2つの先頭セルを作成します。すべての先頭セルに同じデータを読み取らないように、必ず、「**Rows per record**」を適切な値(この例では2)に設定してください。また、実際の場所でデータの読取りが始まるように、上の先頭セルに**データ・オフセット**を設定してください。次の図を参照してください。

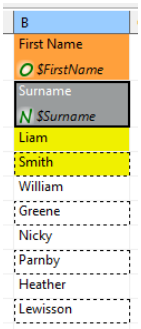


図53.17. 列当たり2つの先頭セルを使用した混合データの読取り。3番目の行で読取りを開始する必要がある最初の先頭セルを反映し、「Rows per record」は2、データ・オフセットは2に引き上げる必要があります。

注意および制限

- **無効なマッピング**: マッピング・エディタを使用して無効なマッピングを作成できます。無効なマッピングによって**SpredsheetDataReader**が失敗します。たとえば、1つのメタデータ・フィールドが複数のセルにマップされたときや、自動入力されたフィールドがマップされたときに(自動入力関数(p.132)を参照)、このようなマッピングが発生します。セル(1つ以上)を複数のメタデータ・フィールドに読み込もうとすると、別の無効なマッピングが発生します。

  なんらかの方法でマッピングを変更すると、検証プロセスが自動的に実行され、マッピングを無効にしたセルやメタデータ・フィールドに警告アイコンが表示されます。このようなセルまたはフィールドにマウスを移動すると、ツールチップに検証の問題に関する情報が表示されます。また、警告検証メッセージの1つがエディタの最上部(白のヘッダー領域)に表示されます。

  前述のように、名前/順序別にマップされたセルに起因する警告が、必ずしもコンポーネントの失敗を示すわけではありません。

- **文字列としての日付の読取り**: SpreadsheetDataReaderでは、`string`フィールドに読み込まれた日付がMS Excelでの表示と同じように表示されることを保証できません。これは、Cloverではセルに格納されている書式文字列の解釈方法が、Excelとは異なり、ロケールに従うためです。

重要

日付を`date`フィールドに読み込み、CTL(p.889)変換を使用して`string`に変換することをお薦めします。

組込みのExcel書式は、次の表に従って解釈されます。

表53.5. 書式文字列

Excelセルに格納された書式インデックス	書式文字列
0	"General"
1	"0"
2	"0.00"
3	"#,##0"
4	"#,##0.00"
5	"$#,##0_);($#,##0)"
6	"$#,##0_);[Red]($#,##0)"
7	"$#,##0.00);($#,##0.00)"

Excelセルに格納された書式インデックス	書式文字列
8	"$#,##0.00_);[Red]($#,##0.00)"
9	"0%"
0xa	"0.00%"
0xb	"0.00E+00"
0xc	"# ?/?"
0xd	"# ??/??"
0xe	"m/d/yy"
0xf	"d-mmm-yy"
0x10	"d-mmm"
0x11	"mmm-yy"
0x12	"h:mm AM/PM"
0x13	"h:mm:ss AM/PM"
0x14	"h:mm"
0x15	"h:mm:ss"
0x16	"m/d/yy h:mm"
0x25	"#,##0_);(#,##0)"
0x26	"#,##0_);[Red](#,##0)"
0x27	"#,##0.00_);(#,##0.00)"
0x28	"#,##0.00_);[Red](#,##0.00)"
0x29	"_(*#,##0_);_(*(#,##0);_(* \"-\"_);_(@_)"
0x2a	"_($*#,##0_);_($*(#,##0);_($* \"-\"_);_(@_)"
0x2b	"_(*#,##0.00_);_(*(#,##0.00);_(*\"-\"??_);_(@_)"
0x2c	"_($*#,##0.00_);_($*(#,##0.00);_($*\"-\"??_);_(@_)"
0x2d	"mm:ss"
0x2e	"[h]:mm:ss"
0x2f	"mm:ss.0"
0x30	"##0.0E+0"
0x31	"@" (テキスト形式)

カスタム書式文字列は、Excelでの定義に従って読み取られます。小数点はロケールに従って変更されます。二重引用符などの特殊文字は解釈されません。

組込みの書式とカスタム書式のいずれの場合も、結果が、Excelでの表示方法とは異なる場合があります。

UniversalDataReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

UniversalDataReaderは、フラット・ファイルからデータを読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
UniversalDataReader	フラット・ファイル	0-1	1-2	✖	✖	✖	✖	✖	✖

1) 各データ・レコードをすべての接続済出力ポートに送信します。
2) 変換の戻り値(p.283)に従って、データ・レコードを出力ポートに送信します。

概要

UniversalDataReaderは、CSV (カンマ区切り値)ファイルやデリミタ付きファイル、固定長ファイルまたは混合テキスト・ファイルなど、フラット・ファイルからデータを読み取ります。このコンポーネントは、単一ファイルに加え、ローカル・ディスクまたはリモートに置かれたファイルのコレクションを読取り可能です。リモート・ファイルは、HTTP、HTTPS、FTP、SFTPのいずれかのプロトコルを介してアクセスできます。このコンポーネントを使用すると、フラット・ファイルのZIPアーカイブおよびTARアーカイブを読み取ることができます。また、stdin (コンソール)、入力ポートまたはディクショナリからのデータ読取りもサポートされています。

解析されたデータ・レコードは1つ目の出力ポートに送信されます。このコンポーネントには、正しくないレコードに関する詳細情報を取得するための、オプションの出力ロギング・ポートがあります。データ・ポリシー(p.306)がcontrolledに設定され、適切なライター(TrashまたはUniversalDataWriter)がポート1に接続されている場合にのみ、すべての正しくないレコードが、正しくない値、場所およびエラー・メッセージに関する情報とともに、このエラー・ポートを介して送信されます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✖	入力ポートの読取り(p.303)用	特定のbyte、cbyte、stringフィールドを含みます
出力	0	✔	正しいデータ・レコード用。	任意[1)]
	1	✖	正しくないデータ・レコードに使用	特定の構造、次の表を参照

1)出力ポート0のメタデータでは、自動入力関数(p.132)を使用できます。注意: source_timestamp関数およびsource_size関数は、ファイルから直接読み取る場合のみ機能します(ファイルがアーカイブである場合またはリモートの場所に保存されている場合は、タイムスタンプが空になり、サイズは0になります)。

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

正しくないレコード用のオプションのロギング・ポートでは、次のメタデータ構造を定義する必要があります。レコードには、特定の型の、正確に5つのフィールド(名前は任意)が次の順序で含まれています。

表53.6 UniversalDataReaderのエラー・メタデータ

フィールド番号	フィールド名	データ型	説明
0	recordNo	long	データセット内のエラーのあるレコードの場所(レコード番号は1から始まります)
1	fieldNo	integer	レコード内のエラーのあるフィールドの場所(1は最初のフィールド、つまりインデックス0のフィールドを表します)
2	originalData	string | byte | cbyte	RAW形式のエラーのあるレコード(すべてのフィールド・デリミタおよびレコード・デリミタを含みます)
3	errorMessage	string | byte | cbyte	エラー・メッセージ: このエラーに関する詳細情報
4	fileURL	string	エラーが発生したソース・ファイル

UniversalDataReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	✔	指定された読取り対象のデータソース(フラット・ファイル、コンソール、入力ポート、ディクショナリ)へのパス、リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Charset		入力レコードの文字エンコーディング(レコード・タイプがfixedである場合、文字エンコーディングはバイト・フィールドに適用されません)	ISO-8859-1 (デフォルト) | <other encodings>
Data policy		書式に誤りがあるデータまたは正しくないデータの処理方法を指定します。データ・ポリシー(p.306)を参照してください	strict (デフォルト) | controlled | lenient

属性	必須	説明	可能な値
Trim strings		文字列をデータ・フィールドに設定する前に先頭および末尾のスペースを削除するかどうかを指定します。次のデータのトリミング(p.418)を参照してください	default | true | false
Quoted strings		特殊文字(カンマ、改行または二重引用符)が含まれるフィールドは、引用符で囲む必要があります。一重引用符および二重引用符のみが引用符文字として受け入れられます。`true`の場合、コンポーネントによる読取り時に特殊文字が削除されます(デリミタとして扱われません)。 例: 入力データ`"25"|"John"`を読み取るには、「**Quoted strings**」を`true`に切り替えて「**Quote character**」を「"」に設定します。これにより、`25|John`という2つのフィールドが生成されます。 デフォルトでは、この属性の値は出力ポート0のメタデータから継承されます。レコード詳細(p.162)も参照してください。	false | true
Quote character		「**Quoted strings**」で許可される引用符の種類を指定します。デフォルトでは、この属性の値は出力ポート0のメタデータから継承されます。レコード詳細(p.162)も参照してください。	both | " | '
Advanced			
Skip leading blanks		入力文字列をデータ・フィールドに設定する前に、先頭のスペース(空白など)をスキップするかどうかを指定します。明示的に設定されない場合(デフォルト値のまま)は、「**Trim strings**」属性の値が使用されます。データのトリミング(p.418)を参照してください。	default | true | false
Skip trailing blanks		入力文字列をデータ・フィールドに設定する前に、末尾のスペース(空白など)をスキップするかどうかを指定します。明示的に設定されない場合(デフォルト値のまま)は、「**Trim strings**」属性の値が使用されます。データのトリミング(p.418)を参照してください。	default | true | false
Number of skipped records		ソース・ファイルからスキップするレコード/行数。入力レコードの選択(p.305)を参照してください。	0 (デフォルト) - N
Max number of records		ソース・ファイルから読み取るレコード数。デフォルトではすべてのレコードが読み取られます。入力レコードの選択(p.305)を参照してください。	1 - N
Number of skipped records per source		各ソース・ファイルからスキップするレコード/行数。デフォルトでは、出力ポート0メタデータの、「**Skip source rows record**」プロパティの値が使用されます。メタデータの値がこの属性の値と異なる場合は、「**Number of skipped records per source**」の値が優先的に適用されます。入力レコードの選択(p.305)を参照してください。	0 (デフォルト) - N
Max number of records per source		各ソース・ファイルから読み取られたレコード/行数。デフォルトでは、各ファイルからすべてのレコードが読み取られます。入力レコードの選択(p.305)を参照してください。	1 - N
Max error count		入力ファイル内で許容されるエラー・レコードの最大数。「**Data policy**」が`Controlled`に設定されている場合のみ適用可能です。	0 (デフォルト) - N

属性	必須	説明	可能な値
Treat multiple delimiters as one		`true`に設定すると、フィールドが複数のデリミタ文字で区切られている場合に単一のデリミタとして解釈されます。	false (デフォルト) | true
Incremental file	[1)]	パスを含む、増分キーを格納しているファイルの名前。増分読取り(p.304)を参照してください。	
Incremental key	[1)]	最後に読み取られたレコードの位置を格納している変数。増分読取り(p.304)を参照してください。	
Verbose		デフォルトでは、エラー通知が簡略化され、パフォーマンスが多少高くなります。`true`に切り替えた場合は、情報が詳細になり、パフォーマンスが低下します。	false (デフォルト) | true
Parser		デフォルトでは、最も適切なパーサーが適用されます。さらに、データ処理用のパーサーを明示的に設定できます。不適切なものが設定された場合は、例外がスローされ、グラフが失敗します。データ・パーサー(p.418)を参照してください。	auto (デフォルト) | *<other>*
Deprecated			
Skip first line		デフォルトでは、最初の行はスキップされません。`true`に切り替えた場合(最初の行にヘッダーが含まれている場合)は、最初の行がスキップされます。	false (デフォルト) | true

[1)]これらの属性を両方指定するか、または両方とも指定しないでください。

詳細説明

- **データのトリミング**

1. 入力文字列は、次のようにフィールドのデータ型に従って値に変換する前に、暗黙的に(つまり、「**Trim strings**」属性がdefault値のまま)処理されます。

   - `boolean`、`date`、`decimal`、`integer`、`long`または`number`の場合、先頭と末尾の両方からスペースが削除されます。

   - `byte`、`cbyte`または`string`の場合、入力文字列は先頭と末尾のスペースを含むフィールドに設定されます。

2. 「**Trim strings**」属性が`true`に設定されている場合は、先頭と末尾のすべてのスペース文字が削除されます。スペースのみで構成されるフィールドは、null (長さがゼロの文字列)に変換されます。`false`値は、先頭と末尾のすべてのスペース文字を保持することを意味します。数値データ型またはブールを表す入力文字列は、スペースが含まれていると解析できません。このため、`false`値は慎重に使用してください。

3. 「**Skip leading blanks**」属性と「**Skip trailing blanks**」属性の両方が、「**Trim strings**」よりも優先されます。そのため、入力文字列のトリミングは、「**Trim strings**」の値に関係なく、これらの属性の`true`値または`false`値によって決定されます。

- **データ・パーサー**

1. `org.jetel.data.parser.SimpleDataParser`: 検証、エラー処理および機能が制限された、非常に簡易的ながら高速のパーサーです。次の属性はサポートされていません。

   - **Trim strings**

- **Skip leading blanks**
- **Skip trailling blanks**
- **Incremental reading**
- **Number of skipped records**
- **Max number of records**
- **Quoted strings**
- **Treat multiple delimiters as one**
- **Skip rows**
- **Verbose**

さらに、これらのいずれかの属性を持つフィールドが1つ以上含まれているメタデータは使用できません。

- フィールドが固定長です。
- フィールドにデリミタがないか、または逆により多くのデリミタがあります。
- 「**Shift**」がnullではありません(「Details」ペイン(p.161)を参照してください)。
- 「**Autofilling**」が`true`に設定されています。
- フィールドがバイトベースです。

2. `org.jetel.data.parser.DataParser`: どのようなリーダー設定でも機能する汎用的なパーサーです。
3. `org.jetel.data.parser.CharByteDataParser`: メタデータにバイトベースのフィールドと文字ベースのフィールドが混在している場合に使用できます。バイトベースのフィールドは、`byte`または`cbyte`のタイプのフィールドか、またはそれ以外の`format`プロパティが`BINARY:`という接頭辞で始まるフィールドです。バイナリ形式(p.123)を参照してください。
4. `org.jetel.data.parser.FixLenByteDataParser`: バイトベースのフィールドのみのメタデータに使用されます。固定数バイトで構成される一連のレコードを解析します。

注意

「**Quoted strings**」の使用中に`org.jetel.data.parser.SimpleDataParser`を選択すると、「**Quoted strings**」属性は無視されます。

ヒントおよびポイント

- データ・フィールドのサイズが大きなレコードの処理: フィールドおよびレコードのバッファ・サイズを調整した場合、**UniversalDataReader**は数百または数千もの文字の入力文字列を処理できます。単に、ニーズに従って、次のプロパティを増やします。レコード・シリアライズ用のRecord.MAX_RECORD_SIZE、解析用のDataParser.FIELD_BUFFER_LENGTHおよび書式設定用のDataFormatter.FIELD_BUFFER_LENGTH。最後に、必ず、DEFAULT_INTERNAL_IO_BUFFER_SIZE変数を2*MAX_RECORD_SIZE以上に増やします。これらのプロパティ変数の変更方法は、デフォルトのCloverETL設定の変更(p.88)を参照してください。

一般的な例

- ヘッダー付きのファイルの処理: 入力ファイルの最初の行が実際のデータではなくフィールド・ラベルを表す場合、「**Number of skipped records**」属性を設定します。ヘッダー付き入力ファイルのコレクションを読み取る場合は、「**Number of skipped records per source**」を設定します。
- 入力ファイルを手動で作成したときのタイピストのエラーの処理: デリミタの偶発的なエラー(入力ファイルの手動入力時にメタデータで定義されているようにセミコロンが1つではなく2つであるなど)を無視する場合は、「**Treat multiple delimiters as one**」属性を`true`に設定します。すべての冗長なデリミタ文字は適切な文字に置き換えられます。

XLSDataReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

XLSDataReaderは、XLSファイルまたはXLSXファイルからデータを読み取ります。

重要

Clover 3.3.0以降、スプレッドシートの読取り/書込みに使用可能な新しい強力なコンポーネントがあります。SpreadsheetDataReader(p.404)およびSpreadsheetDataWriter(p.532)を参照してください。ただし、前のXLSコンポーネント(XLSDataReader(p.421)およびXLSDataWriter(p.555))には引き続き互換性があります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
XLSDataReader	XLS(X)ファイル	0-1	1-n	はい	いいえ	いいえ	いいえ	いいえ	いいえ

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します。詳細は、変換の戻り値(p.283)を参照してください。

概要

XLSDataReaderは、XLSファイルまたはXLSXファイルの指定したシートからデータを読み取ります(ローカルまたはリモート)。圧縮ファイル、コンソール、入力ポートまたはディクショナリからもデータを読み取ることができます。

注意

XLSDataReaderは、すべてのデータをメモリーに格納し、メモリー要件が高くなっています。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	ポート読取りに使用。「入力ポートからの読取り」の項(p.300)を参照してください。	1つのフィールド(byte、cbyte、string)。
出力	0	はい	正しいデータ・レコード用。	任意[1)]
	1-n	いいえ	正しいデータ・レコード用。	出力0

説明:

1) メタデータは自動入力関数(p.132)を使用できます。注意: source_timestamp関数およびsource_size関数は、ファイルから直接読み取る場合のみ機能します(ファイルがアーカイブである場合またはリモートの場所に保存されている場合は、タイムスタンプが空になり、サイズは0になります)。

XLSDataReaderの属性

属性	必須	説明	可能な値
Basic			
Type of parser		使用するパーサーを指定します。デフォルトでは、拡張子(XLSまたはXLSX)に従ってコンポーネントが推測されます。	Auto (デフォルト) | XLS | XLSX
File URL	はい	読み取るデータ・ソースを指定する属性(入力ファイル、コンソール、入力ポート、ディクショナリ)。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Sheet name	1)	読み取るシートの名前。名前にワイルドカード?および*を使用できます。	
Sheet number	1)	読み取るシートの番号。番号は0から始まります。番号のシーケンスはカンマで区切られたり、ハイフンでまとめられます。number、minNumber-maxNumber、*-maxNumber、minNumber-*の各パターンを使用できます。例: *-5,9-11,17-*。	
Charset		読み取られたレコードのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Data policy		エラーの発生時に必要な処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Controlled | Lenient
Metadata row		列の名前が含まれている行の番号。デフォルトでは、シートのヘッダーがメタデータ行として使用されます。詳細は、マッピングとメタデータ(p.423)を参照してください。	0 (デフォルト) | 1-N

属性	必須	説明	可能な値
Field mapping		CloverフィールドへのXLSフィールドのマッピング。Cloverフィールドの各マッピングをセミコロンで区切ったシーケンスとして表されます。各マッピングは、`$CloverField:=#XLSColumnCode`または`$CloverField:=XLSColumnName`のようになります。詳細は、マッピングとメタデータ(p.423)を参照してください。	
Advanced			
Number of skipped records		すべてのソース・ファイルにわたって継続的にスキップされるレコード数。入力レコードの選択(p.305)を参照してください。	0-N
Max number of records		すべてのソース・ファイルにわたって継続的に読み取られる最大レコード数。入力レコードの選択(p.305)を参照してください。	0-N
Number of skipped records per source		各ソース・ファイルからスキップするレコード数。入力レコードの選択(p.305)を参照してください。	メタデータ内と同じ(デフォルト) | 0-N
Max number of records per source		各ソース・ファイルから読み取るレコードの最大数。入力レコードの選択(p.305)を参照してください。	0-N
Max error count		グラフが失敗する前に「**Data Policy**」の`Controlled`値に対して許容されるエラーの最大数。	0 (デフォルト) | 1-N
Incremental file	2)	パスを含む、増分キーを格納しているファイルの名前。増分読取り(p.304)を参照してください。	
Incremental key	2)	最後に読み取られたレコードの位置を格納している変数。増分読取り(p.304)を参照してください。	
Deprecated			
Start row		読み取る最初の行(その行が含まれます)。「**Number of skipped records**」よりも優先度は低くなります。	0 (デフォルト) | 1-n
Final row		読取り済の最後の行に続くまだ読み取られない最初の行(その行は含まれません)。「**Max number of records**」よりも優先度は低くなります。	all (デフォルト) | 1-n

説明:

1) これらのいずれかの属性を指定する必要があります。「**Sheet name**」の方が優先されます。

2) これらの属性を両方指定するか、または両方とも指定しないでください。

詳細説明

マッピングとメタデータ

いくつかのマッピング(**フィールド・マッピング**)を指定する場合は、この属性の行をクリックします。その後、そこにボタンが表示され、このボタンをクリックすると次のダイアログが開きます。

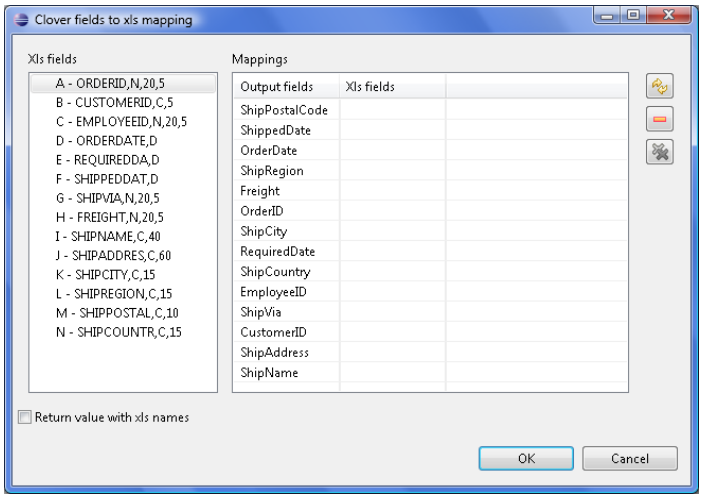


図53.18. XLSマッピングのダイアログ

このダイアログは2つのペインで構成されています。左側の**XLSフィールド**と右側の**マッピング**です。このダイアログの右側にボタンが3つあります。自動マッピング用、選択した1つのマッピングの取消用およびすべてのマッピングの取消用です。左側のペインからxlsフィールドを選択し、左マウス・ボタンを押し、右ペイン(**XLSフィールド**列)にドラッグしてボタンを放す必要があります。このように、選択したxlsフィールドは出力cloverフィールドの1つにマップされます。他のxlsフィールドにも同じことを繰り返します。(または「**Auto mapping**」ボタンをクリックできます。)

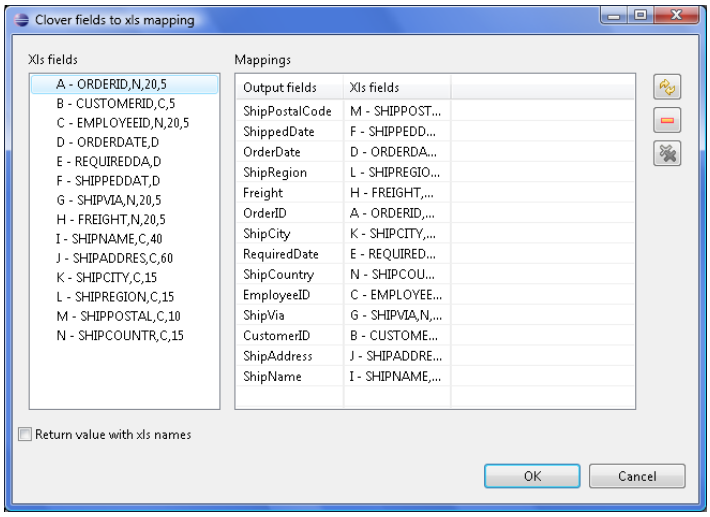


図53.19. CloverフィールドにマップされたXLSフィールド

xlsフィールドは、XLSファイルからメタデータを抽出するときにxls列名から自動的に派生します。

「**OK**」をクリックしてマッピングを確定すると、「**Field mapping**」属性が(たとえば)`$OrderDate:=#D;$OrderID:=#A`のようになります。

他方、XLSマッピングのダイアログで「**Return value with xls names**」チェック・ボックスを選択した場合は、同じマッピングが`$OrderDate:=ORDERDATE,D;$OrderID:=ORDERID,N,20,5`のようになります。

「**Field mapping**」属性は、単一のマッピングをセミコロンで区切ったシーケンスです。

それぞれの単一のマッピングは、cloverフィールド名とxlsフィールドの割当てで構成されています。Cloverフィールドは割当ての左側で、ドル記号に続けて指定します。xlsフィールドは割当ての右側で、xls列のコードをハッシュに続けて指定するか、または「**Xls fields**」ペインに示されたxlsフィールドとします。

すべてのxls列を読み取って送信する必要はなく、その一部のみを読み取って送信できます。

例53.6. XLSDataReaderのフィールド・マッピング

- **列コードを使用したマッピング**

```
$first_name:=#B;$last_name:=#D;$country:=#E
```

- **列名(XLSフィールド)を使用したマッピング**

```
$first_name:=f_name;$last_name:=l_name;$country:=country
```

XMLExtract

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

XMLExtractは、XMLファイルからデータを読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1)]	各出力に送信[2)]	変換	変換が必要	Java	CTL
XMLExtract	XMLファイル	0-1	1-n	いいえ	はい	いいえ	いいえ	いいえ	いいえ

説明

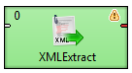
1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します(**DataGenerator**および**MultiLevelReader**)。詳細は、変換の戻り値(p.283)を参照してください。**XMLExtract**および**XMLXPathReader**は、それぞれの「**Mapping**」属性または「**Mapping URL**」属性で定義されたとおりにデータをポートに送信します。

概要

XMLExtractは、SAXテクノロジを使用してXMLファイルからデータを読み取ります。圧縮ファイル、コンソール、入力ポートおよびディクショナリからもデータを読み取ることができます。このコンポーネントは、XMLファイルの読取りも可能な**XMLXPathReader**よりも高速です。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	ポート読取りに使用。入力ポートからの読取り(p.300)を参照してください。	コンポーネントの入力を指定する1つのフィールド(`byte`、`cbyte`、`string`)。入力フィールドは出力にマップできま

ポート・タイプ	番号	必須	説明	メタデータ
				す。詳細は、XMLExtractのマッピング定義(p.429)を参照してください。
出力	0	はい	正しいデータ・レコード用。	任意[1)]
	1-n	2)	正しいデータ・レコード用。	任意[1)](ポートごとに異なるメタデータを保持できます)

説明:

1): 各出力ポート上のメタデータは同じである必要はありません。各メタデータは自動入力関数(p.132)を使用できます。

2): マッピングによっては他の出力ポートが必要になる場合があります。

XMLExtractの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	読み取るデータ・ソースを指定する属性(XMLファイル、コンソール、入力ポート、ディクショナリ)。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Charset		読み取られたレコードのエンコーディング。	任意のエンコーディング、デフォルトではデフォルト・システム・エンコーディング
Mapping	1)	出力ポートへの入力XML構造のマッピング。詳細は、XMLExtractのマッピング定義(p.429)を参照してください。	
Mapping URL	1)	出力ポートへの入力XML構造のマッピングを定義するパスを含めた、外部ファイルの名前。詳細は、XMLExtractのマッピング定義(p.429)を参照してください。	
Namespace Bindings		「**Mapping**」に任意の名前空間接頭辞を使用できるようにします。名前空間(p.441)を参照してください。	
XML Schema		**マッピング**定義の作成に使用するファイルのURL。詳細は、XMLExtractマッピング・エディタおよびXSDスキーマ(p.434)を参照してください。	
Use nested nodes		デフォルトでは、ネストした要素も自動的に出力ポートにマップされます。`false`に設定した場合、このようなネストした各要素に対して明示的な`<Mapping>`タグを作成する必要があります。	true (デフォルト) | false
Trim strings		デフォルトでは、要素の値の先頭から末尾までにある空白が削除されます。`false`に設定した場合は削除されません。	true (デフォルト) | false

属性	必須	説明	可能な値
Advanced			
Validate		XMLの(スキーマに対する)検証を有効化/無効化します。	true | false (デフォルト)
XML features		`nameM:=true`または`nameN:=false`(ここで、各`nameM`は検証の対象とするXML機能)のいずれかの形式の個別の式のシーケンス。これらの式は、セミコロンで区切ります。詳細は、XML機能(p.307)を参照してください。	
Number of skipped mappings		すべてのソース・ファイルにわたって継続的にスキップされるマッピング数。入力レコードの選択(p.305)を参照してください。	0-N
Max number of mappings		すべてのソース・ファイルにわたって継続的に読み取られる最大レコード数。入力レコードの選択(p.305)を参照してください。	0-N

説明:

1) これらのいずれかを指定する必要があります。両方が指定された場合は、**マッピングURL**が優先されます。

詳細説明

例53.7. XMLExtractのマッピング

```
<Mappings>
    <TypeOverride elementPath="/employee/child" overridingType="boy" />
    <Mapping element="employee" outPort="0" implicit="false" xmlFields="salary"
cloverFields="basic_salary">
        <Mapping element="child" outPort="1" parentKey="empID" generatedKey="parentID"/>
        <Mapping element="benefits" outPort="2"
                                    parentKey="empID;jobID" generatedKey="empID;jobID"
                                    sequenceField="seqKey" sequenceId="Sequence0">
            <Mapping element="financial" outPort="3" parentKey="seqKey"
generatedKey="seqKey"/>
        </Mapping>
        <Mapping element="project" outPort="4" parentKey="empID;jobID"
generatedKey="empID;jobID">
            <Mapping element="customer" outPort="5"
                                    parentKey="projName;projManager;inProjectID;Start"
                                    generatedKey="joinedKey"/>
        </Mapping>
    </Mapping>
</Mappings>
```

XMLExtractのマッピング定義

1. **マッピング**定義(「**Mapping URL**」属性と「**Mapping**」属性で指定されるファイルの内容の両方)は、それぞれ<Mappings>の開始タグと終了タグのペアで構成されています。<Mappings>の開始タグと終了タグの両方とも空であり、他の属性もありません。

2. <Mappings>タグのこのペアで、ネストしたすべての<Mapping>タグおよび<TypeOverride>タグを囲みます。これらの各<Mapping>タグには、いくつかのXMLExtractのマッピング・タグの属性(p.431)が含まれます。詳細は、XMLExtractのタイプ・オーバーライド・タグ(p.429)またはXMLExtractのマッピング・タグ(p.430)も参照してください。

3. **XMLExtractのタイプ・オーバーライド・タグ**

   タイプ・オーバーライド・タグを使用すると、指定のパス上の要素を、タイプが実際にはoverridingTypeであるかのように処理する必要があることをマッピング・エディタに通知できます。このタグは、実行時にXMLファイルの実際の処理に影響を及ぼしません。

   例:

   ```
   <TypeOverride elementPath="/employee/child" overridingType="boy" />
   ```

   - elementPath

     必須

     各タイプ・オーバーライド・タグには、1つの「elementPath」属性が含まれている必要があります。この要素の値は、入力XML構造のルートからノードへのパスである必要があります。

     ```
     elementPath="/[prefix:]parent/.../[prefix]nodeName"
     ```

   - overridingType

     必須

     各タイプ・オーバーライド・タグには、1つの「overridingType」属性が含まれている必要があります。この要素の値は、参照先のXMLスキーマのタイプである必要があります。

     ```
     overridingType="[prefix:]typeName"
     ```

4. **XMLExtractのマッピング・タグ**

   - **空のマッピング・タグ(子なし)**

     `<Mapping element="[prefix:]nameOfElement"` XMLExtractのマッピング・タグの属性(p.431) `/>`

     これは、XML構造の次のノードに対応します。

     `<[prefix:]nameOfElement>ValueOfTheElement</[prefix:]nameOfElement>`

   - **空ではないマッピング・タグ(子がある親)**

     `<Mapping element="[prefix:]nameOfElement"` XMLExtractのマッピング・タグの属性(p.431) `>`

     `(nested Mapping elements (only children, parents with one or more children, etc.)`

     `</Mapping>`

     これは、次のXML構造に対応します。

     `<[prefix:]nameOfElement elementAttributes>`

     `(nested elements (only children, parents with one or more children, etc.)`

     `</[prefix:]nameOfElement>`

     ネストした`<Mapping>`要素の他に、Mappingに`<FieldMapping>`要素を含めて入力レコードから出力レコードにフィールドをマップできます。詳細は、XMLExtractのフィールド・マッピング・タグ(p.430)を参照してください。

5. **XMLExtractのフィールド・マッピング・タグ**

   フィールド・マッピング・タグを使用すると、入力レコードから親マッピング要素の出力レコードにフィールドをマップできます。

   例:

   `<FieldMapping inputField="sessionID" outputField="sessionID" />`

   - `inputField`

     必須

     出力レコードにマップする、入力レコードからのフィールドを指定します。

     `inputField="fieldName"`

   - `outputField`

     必須

     入力フィールドからの値を格納するフィールドを指定します。

     `outputField="fieldName"`

6. `<Mapping>`タグのネストした構造は、入力XMLファイル内のXML要素のネストした構造をコピーします。次の例を参照してください。

例53.8. XML構造からマッピング構造へ

- **XML構造が次のようになっている場合**

```
<[prefix:]nameOfElement>
   <[prefix1:]nameOfElement1>ValueOfTheElement11</[prefix1:]nameOfElement1>
   ...
   <[prefixK:]nameOfElementM>ValueOfTheElementKM</[prefixK:]nameOfElementM>
   <[prefixL:]nameOfElementN>
      <[prefixA:]nameOfElementE>ValueOfTheElementAE</[prefixA:]nameOfElementE>
      ...
      <[prefixR:]nameOfElementG>ValueOfTheElementRG</[prefixR:]nameOfElementG>
   </[prefixK:]nameOfElementN>
</[prefix:]nameOfElement>
```

- **マッピングは次のようになります。**

```
<Mappings>
   <Mapping element="[prefix:]nameOfElement" attributes>
      <Mapping element="[prefix1:]nameOfElement1" attributes11/>
      ...
      <Mapping element="[prefixK:]nameOfElementM" attributesKM/>
      <Mapping element="[prefixL:]nameOfElementN" attributesLN>
         <Mapping element="[prefixA:]nameOfElementE" attributesAE/>
         ...
         <Mapping element="[prefixR:]nameOfElementG" attributesRG/>
      </Mapping>
   </Mapping>
</Mappings>
```

ただし、「**Mapping**」ではすべてのXML構造をコピーする必要はなく、XMLファイル内の指定されたレベルから開始できます。また、「**Use nested nodes**」属性のデフォルト設定が使用されている(`true`)場合、子`<Mapping>`タグを別途作成することなく、より深いノードをマッピングすることもできます。

重要

ネストしたノードは、それらの名前が親内で一意であり、混乱の可能性がない場合にのみ、マッピングできます。

7. **XMLExtractのマッピング・タグの属性**

- `element`

  必須

  各マッピング・タグには、1つの「`element`」属性が含まれている必要があります。この要素の値は、最終的に接頭辞(名前空間)が付く、入力XML構造のノードである必要があります。

  `element="[prefix:]name"`

- `outPort`

  オプション

  データの送信先となる出力ポートの数。定義されない場合、このレベルの**マッピング**からのデータは、このレベルの**マッピング**を使用して送信されません。

  `<Mapping>`タグは、「`outPort`」属性が含まれていない場合には、より深いXMLノードの位置を特定するためにのみ使用します。

  例: `outPort="2"`

重要

上位の親<Mapping>タグを使用して任意のレベルからの値を送信することもできます(「**Use nested nodes**」のデフォルト設定が使用され、その識別情報が一意であり、混乱の可能性がない場合)。

- useParentRecord

  オプション

  trueの場合、マッピングはマップされた値をoutPortが指定された至近の親マッピング要素によって生成されたレコードに割り当てます。この属性のデフォルト値はfalseです。

  useParentRecord="false|true"

- implicit

  オプション

  falseの場合、マッピングはXMLフィールドを同じ名前のレコード・フィールドに自動的にはマップしません。この属性のデフォルト値はtrueです。

  implicit="false|true"

- parentKey

  「parentKey」属性は、子の親を識別する役割を果たします。

  このため、parentKeyはセミコロン、コロンまたはパイプで区切られた、次の親レベルのメタデータ・フィールドのシーケンスです。

  これらのフィールドはこのような上位要素に対して指定されたポート上のメタデータに使用され、対応する値が入力されます。この属性(parentKey)は、識別情報としてどのフィールドが親レベルから子レベルにコピーされるかを伝えるのみです。

  このため、これらのメタデータ・フィールドの数およびデータ型は、「generatedKey」属性内で同じにする必要があります。同じでない場合は、すべての値が連結されて、一意の文字列値が作成されます。この場合、キーのフィールドは1つのみになります。

  例: parentKey="first_name;last_name"

  これらの親cloverフィールドの値は、「generatedKey」属性に指定されたcloverフィールドにコピーされます。

- generatedKey

  「generatedKey」属性には、親要素から取得された値が入力されます。これは、子の親を指定します。

  このため、generatedKeyはセミコロン、コロンまたはパイプで区切られた、指定の子レベルのメタデータ・フィールドのシーケンスです。

  これらのメタデータ・フィールドはこの子要素に指定されたポートで使用され、親レベルから取得された値が入力されます(親レベルでは、これらのメタデータ・フィールドはこの子レベルに指定された「parentKey」属性のメタデータ・フィールドに送信されます)。この属性は、識別情報としてどのフィールドが親レベルから子レベルにコピーされるかを伝えるのみです。

  このため、これらのメタデータ・フィールドの数およびデータ型は、「parentKey」属性内で同じにする必要があります。同じでない場合は、すべての値が連結されて、一意の文字列値が作成されます。この場合、キーのフィールドは1つのみになります。

  例: generatedKey="f_name;l_name"

これらのcloverフィールドの値は、「parentKey」属性に指定されたcloverフィールドから取得されます。

- sequenceField

  parentKeyとgeneratedKeyのペアでもレコードの一意の識別情報(親子関係)が保証されない場合があります。これは、1つの親に同じ要素名の子が複数存在する場合です。

  この場合、これらの子に識別情報として番号を付与できます。

  デフォルトでは(作成済シーケンスでも定義されない場合)、子には1から始まる整数が1刻みで付けられます。

  この属性は、識別番号が書き込まれる、指定されたレベルのメタデータ・フィールドの名前です。

  次にネストされたレベルのparentKeyとして機能できます。

  例: sequenceField="sequenceKey"

- sequenceId

  オプション

  parentKeyとgeneratedKeyのペアでもレコードの一意の識別情報(親子関係)が保証されない場合があります。これは、1つの親に同じ要素名の子が複数存在する場合です。

  この場合、これらの子に識別情報として番号を付与できます。

  このシーケンスが定義されている場合、そのシーケンスを使用してこれらの子要素にも開始値および刻みを変えて番号を付けることができます。また、以後のグラフ実行間の値を保持することもできます。

  シーケンスのID。

  例: sequenceId="Sequence0"

重要

同じ要素名の子を複数持つ親が存在する場合があります。このような場合、parentKeyからgeneratedKeyにコピーされる親情報を使用して、これらの子を識別することはできません。このような情報のみでは不十分です。そのため、シーケンスを定義してその複数の子要素に識別番号を付けます。

- xmlFields

  XMLノードまたは属性の名前を変更する場合は、「xmlFields」属性と「cloverFields」属性のペアを使用して行う必要があります。

  指定されたレベルの要素名または属性名のシーケンスは、セミコロン、コロンまたはパイプで区切ることができます。

  同じ数のこれらの名前を「cloverFields」属性に指定する必要があります。

  値は指定のデータ型に対応している必要があります。

  例: xmlFields="salary;spouse"

  また、現行のレベルのXML要素およびその属性以外にも到達できます。"この要素の親"を参照するには、"../"文字列を使用します。詳細は、「Source」タブ(p.436)を参照してください。

重要

デフォルトでは、XML名(エレメント名および属性名)はそれぞれの名前でメタデータ・フィールドにマッピングされます。

- `cloverFields`

  XMLノードまたは属性の名前を変更する場合は、「`xmlFields`」属性と「`cloverFields`」属性のペアを使用して行う必要があります。

  セミコロン、コロンまたはパイプで区切られた、指定されたレベルのメタデータ・フィールド名のシーケンス。

  これらの名前の数は、「`xmlFields`」属性の値と同じである必要があります。

  また、値は指定のデータ型に対応している必要があります。

  例: `cloverFields="SALARY;SPOUSE"`

重要

デフォルトでは、XML名(エレメント名および属性名)はそれぞれの名前でメタデータ・フィールドにマッピングされます。

- `skipRows`

  オプション

  スキップする必要がある要素の数。デフォルトでは何もスキップされません。

  例: `skipRows="5"`

重要

親がスキップされると、そのネストされた(子)要素もスキップされます。

- `numRecords`

  オプション

  読み取られる要素の数。デフォルトではすべてが読み取られます。

  例: `numRecords="100"`

XMLExtractマッピング・エディタおよびXSDスキーマ

マッピング・コードを手動で記述する他に、「**XML Schema**」属性を設定できます。これは、**マッピング**定義の作成に使用可能なXSDスキーマが含まれているファイルのURLです。

XSDを使用するときは、**マッピング**・ダイアログで視覚的にマッピングを実行できます。これは、「**Mapping**」タブと「**Source**」タブの2つのタブで構成されています。「**Source**」タブでは「**Mapping**」属性を定義でき、「**Mapping**」タブでは**XML Schema**を操作できます。

注意

ソースXMLに有効なXSDスキーマを保有していない場合は、「**Mapping**」タブに切り替え、「**Genereate XML Schema**」をクリックして、XMLからXSD構造を推測してみることができます。

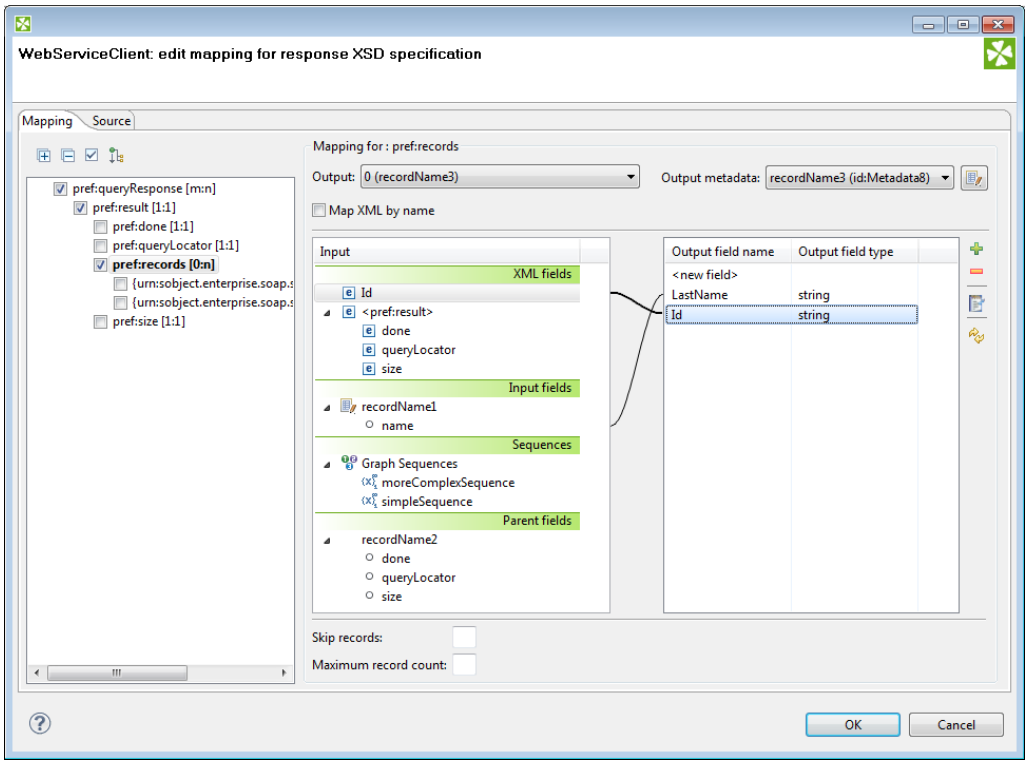


図53.20. XMLExtractのマッピング・ダイアログ

「**Mapping**」タブの左側のペインで、XMLのツリー構造を参照できます。どの要素も、ソース・ファイルでの出現の数(たとえば、[0:n])を示します。このペインでは、出力ポートにマップする要素を確認する必要があります。

最上部で、ドロップダウン・リストから選択することで、選択した要素ごとに**出力**を指定します。有効な値は次のとおりです。

- **Not mapped**: マッピングはレコードを生成しません。このようなマッピング要素を使用して、パーサーがまずこの要素を検出した場合にのみ、任意の子マッピングが処理されるようにできます。

  Parent record: マッピングはレコードを生成しませんが、マップされた値を親レコードに入力します。

  portNumber(metadata): マッピングはレコードを生成し、選択した出力ポートに書き込みます。

次に、`portNumber(metadata)`のラベルが付いたメタデータのリストから、たとえば「3(customer)」を選択できます。

右側では、マッピングの**入力フィールド**および**出力フィールド**を参照できます。これらを(「**Map XML by name**」チェック・ボックスを選択して)それぞれの名前に従って互いにマップするか、または手動で明示的にマップします。「**Input - XML fields**」では、要素だけでなくその親要素も(親にフィールドがある場合)参照可能であり、マップできます。前の図では、「pref:records」要素が選択されていますが、「size」フィールドが実際にマップされているその親要素「pref:result」を飛び越えることができます。したがって、「**Parent key**」プロパティおよび「**Generated key**」プロパティを使用する場合よりもはるかに容易にマッピング全体を作成できます。

また、入力フィールド(「**Input fields**」セクション)や親マッピングによって生成されたレコードからのフィールド(「**Parent fields**」セクション)をマップしたり、「**Sequences**」セクションからいずれかの出力フィールドにシーケンスをマップしてレコードの一意のidを生成することもできます。

注意

いくつかのシーケンスが出力メタデータ・フィールドにマップされている場合は、**sequenceId**および**sequenceField**が設定されます。ただし、**sequenceField**のみを設定できます。この場合、新しいシーケンスが作成され、メタデータ・フィールドにマップされます。マッピングは有効で

すが、メタデータ・フィールドが存在しないシーケンスにマップされたとの警告が**マッピング・ダイアログ**に表示されます。

「Source」タブ

すべての要素を定義し、出力ポート、マッピングおよびその他のプロパティを指定した後、「Source」タブに切り替えることができます。ここには、マッピング・コードが表示されます。その構造は、これまでの項での説明と同じです。

注意

ソースXMLに有効なXSDスキーマを保有していない場合は、要素を視覚的にマップできなくなり、ここの**「Source」**で行う必要があります。

注意

スキーマにない属性または要素をマップできます。検証警告は発生せず、マッピングは**「Mapping」**タブに視覚化されます。スキーマにないマップ済の要素および属性は、斜体フォントを使用して表示されます。

要素をその親のXMLフィールドにマップする場合は、フィールド名の前に(ファイル・システムの場合と同様) ../ 文字列を使用します。どの../も"この要素の親"を表すため、../../は要素の親の親であることを意味します。次に示す例を検討してください。../../empIDは、現在選択している要素「customer」で利用可能な「employee」のフィールドです。

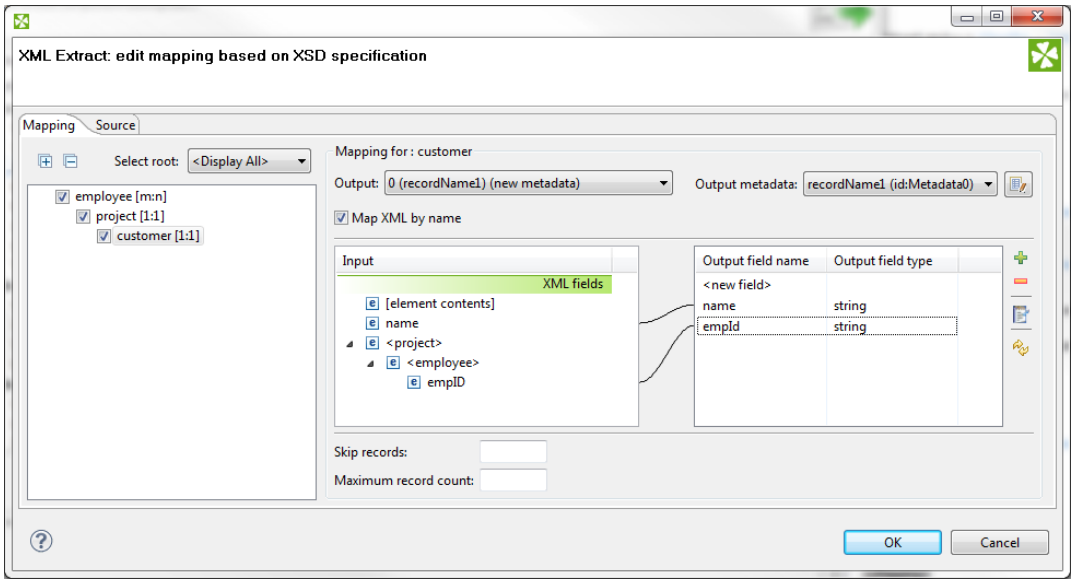


図53.21. 親要素

```
<Mapping element="employee">
 <Mapping element="project">
  <Mapping element="customer" outPort="0"
   xmlFields="name;../../empID"
   cloverFields="name;empId"/>
 </Mapping>
</Mapping>
```

特に、**「Use nested nodes」**プロパティを`true`に設定することによって親要素を参照する場合には、1つのことに注意が必要です。つまり、../を使用して1つの親レベルを参照することは、実際には、XML内で、../親参照によって`<Mapping>`の直接の親`<Mapping>`に定義されている祖先要素(親より上位)を参照することを意味します。

わかりやすくするために、例を示します。直前の例に示したマッピングを思い出してください。その<Mapping>要素の1つを省略しますが、それに伴って親フィールド参照をどのように変える必要があったかを確認してください。

```
<Mapping element="employee">
 <Mapping element="customer" outPort="0"
  xmlFields="name;../empID"
  cloverFields="name;empId"/>
</Mapping>
```

マッピングでのドットの使用

'.'ドット構文を使用して要素の値をマップできます。**ドットは、要素自体(その名前)を意味します。** それ以外に要素の名前がマッピングに(「customer」というようにテキストとして)出現した場合は、要素のサブ要素または属性を表します。(注意: Clover v. 3.1.0以降で使用可能)

ドットは、他のXML要素/属性名と同様に、xmlFields属性で使用できます。ビジュアル・マッピング・エディタでは、ドットは「XML Fields」ツリーに要素の内容として表されます。

次のコード・チャンクは、メタデータ・フィールドcustomerValue上で要素customerの値をマップします。次に、projectValueフィールド上でproject (つまり、customerの親要素、したがって../.を使用)がマップされます。

```
<Mapping element="project">
 <Mapping element="customer" outPort="0"
  xmlFields=".;../."
  cloverFields="customerValue;projectValue"/>
</Mapping>
```

要素値は、子要素がない場合にのみ、要素の開始タグと終了タグで囲まれたテキストで構成されます。要素に子要素がある場合、要素の値は要素の開始タグとその最初の子要素の開始タグで囲まれたテキストで構成されます。

重要

要素値は、それぞれの名前でCloverフィールドにマップされます。このため、前述の<customer>要素はcustomerという名前のCloverフィールドに自動的にマップされます(暗黙的マッピング)。

ただし、別の名前のCloverフィールドに<customer>要素の名前を変更する場合は(明示的マッピング)、次の構成が必要です。

```
   <Mapping ... xmlFields="customer" cloverFields="newFieldName"   />
```

また、XMLファイルに同じ名前の要素および属性が含まれているときは

```
   <customer customer="JohnSmithComp">
   ...
</customer>
```

要素と属性値の両方を2つの異なるフィールドにマップできます。

```
<Mapping element="customer" outPort="2"
   xmlFields=".;customer"
   cloverFields="customerElement;customerAttribute"/>
</Mapping>
```

例に示された明示的マッピング(フィールドの名前変更)は、暗黙的マッピングよりも優先されます。暗黙的マッピングは、対応するMapping要素の「implicit」属性をfalseに設定してオフにできます。

要素の属性およびサブ要素がすべて同じ名前という、前述の例からさらに複雑な状況が生じることもあります。唯一必要な操作は、構成の最後に別のマッピングを追加することです。オプションでサブ要素をその親とは別の出力ポートに送信することもできます。その他のオプションとしてマッピングを空白にしておくこともありますが、なんらかの方法でサブ要素を処理する必要があります。

```
<Mapping element="customer" outPort="2"
  xmlFields=".;customer"
  cloverFields="customerElement;customerAttribute"/>
  <Mapping element="customer" outPort="4" /> // customer's subelement called 'customer' as
well
</Mapping>
```

要素の内容(テキストおよび子要素)マッピング

要素の内容をフィールドにマップできます。その場合、要素のサブツリー全体が出力ポートに送信されます。要素内容をマップするには、+文字または-文字を使用します。+ (プラス)と- (マイナス)のマッピングは、+は要素の内容とその包含する要素をマップし、-は要素自体ではなく要素の内容をマップすることが異なります。

xmlがあり

```
<customers>
  <customer>
    <firstname>John</firstname>
    <lastname>Smith</lastname>
    <city>Smith</city>
  </customer>
</customers>
```

要素customerで+マッピングを使用した場合

```
<customer>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
  <city>Smith</city>
</customer>
```

という出力が得られます。

customer要素で-マッピングを使用した場合

```
    <firstname>John</firstname>
    <lastname>Smith</lastname>
    <city>Smith</city>
```

という出力が得られます。

重要

要素の内容のマッピングでは、非常に大量のデータが生成される場合があります。これは、処理速度に大きな影響を及ぼすことがあります。

「useParentRecord」属性の使用

ネストした要素からの値をマップするものの、親およびネストした要素用に別のレコードを作成しない場合は、Mapping要素の「useParentRecord」属性の使用を検討してください。属性をtrueに設定することで、Mapping要素によってマップされた値は新しいレコードに割り当てられませんが、親レコードに設定されます。(注意: Clover v. 3.3.0-M3以降で使用可能)

次のコード・チャンクは、メタデータ・フィールドprojectValue上で要素projectの値をマップし、メタデータ・フィールドcustomerValue上でcustomer要素の値をマップします。customerValueフィールドは、projectValueと同じレコードに設定されます。

```
<Mapping element="project" outPort="0" xmlFields="." cloverFields="projectValue">
 <Mapping element="customer" useParentRecord="true" xmlFields="."
cloverFields="customerValue" />
</Mapping>
```

テンプレート

テンプレートは「**Source**」タブでのみ使用できます。テンプレートは、一般的に、数多くのネストした要素または再帰的データを読み取る場合に役立ちます。

テンプレートは、宣言とボディで構成されています。ボディは宣言の後(最大でテンプレート参照まで、次を参照)にあり、任意のマッピングを含めることができます。宣言は、「templateId」属性が含まれている要素です。テンプレートの宣言の例を参照してください。

```
<Mapping element="category"  templateId="myTemplate">
 <Mapping element="subCategory"
  xmlFields="name"
  cloverFields="subCategoryName"/>
</Mapping>
```

テンプレートを使用するには、「templateRef」属性に既存のtemplateIdを入力します。テンプレートを参照するには、まずテンプレートを宣言する必要があります。テンプレートを使用することによって、宣言で始まるマッピング全体がテンプレート参照の記述先にコピーされます。その利点は、頻繁に繰り返されるコードの変更が必要になった場合に、変更を加える場所が常に1つ、つまりテンプレートのみとなることです。マッピングでテンプレートを参照する方法の基本的な例を次に示します。

```
<Mapping templateRef="myTemplate" />
```

また、テンプレート参照はテンプレート宣言内に記述できます。参照は、宣言の最後の要素として配置してください。宣言するものと同じテンプレートを参照する場合は、再帰的テンプレートを作成します。

ソースXMLの外観に常に注意してください。nレベルのネストしたデータに対しては、「nestedDepth」属性をnに設定する必要があります。次の例を参照してください。

```
<Mapping element="myElement"  templateId="nestedTempl">
  <!-- ... some mapping ...-->
 <Mapping templateRef="nestedTempl" nestedDepth="3"/>
</Mapping> <!-- template declaration ends here -->
```

注意

次のコード・チャンクは

```
 <Mapping templateRef="unnestedTempl" nestedDepth="3" />
```

次のように考えることができます。

```
 <Mapping templateRef="unnestedTempl">
  <Mapping templateRef="unnestedTempl">
   <Mapping templateRef="unnestedTempl">
   </Mapping>
  </Mapping>
 </Mapping>
```

また、参照をネストする両方の方法を使用できます。ただし、3つのネストされた参照のある後者の方法をテンプレート宣言内で使用すると、予期しない結果が発生する場合があります。深さが増すたびに、各templateRefがそのテンプレート・コードをコピーします。ただし、たとえば3番目の参照がアクティブである場合は、まず、より上位の2つの参照のコードをコピーし、次にそれ自体のコードをコピーする必要があります。このように、ツリーの深さは急速に(指数関数的に)増大します。このような混乱を避けるために、要素とともに宣言をラップし、ネストされた参照を宣言の外部で使用できます。次に示す例を参照してください。テンプレートを参照から切り離すためにwrap要素が効果的に使用されています。その場合、3つの参照は3つのレベルのネストしたデータを参照します。

```
<Mapping element="wrap">
 <Mapping element="realElement" templateId="unnestedTempl"

 <!-- ... some mapping ...-->

 </Mapping>  <!-- template declaration ends here -->
</Mapping> <!-- end of wrap -->
```

```
<Mapping templateRef="unnestedTempl">
 <Mapping templateRef="unnestedTempl">
  <Mapping templateRef="unnestedTempl">
  </Mapping>
 </Mapping>
</Mapping>
```

要約すると、ネストしたテンプレート参照ではなくnestedDepthを使用すると、常に結果が透過的になります。これを使用することをお薦めします。

名前空間

名前空間がある**XMLスキーマ**を指定した場合、名前空間が**Namespace Bindings**に自動的に抽出され、**名前**が付与されます。ただし、**「Name」**は表記にすぎないため、入力スキーマの名前空間接頭辞と正確に一致する必要はありません。次のように**「Namespace Bindings」**属性でいつでも編集できます。

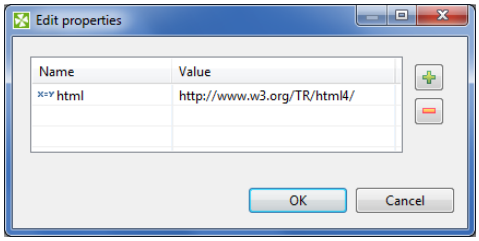


図53.22. XMLExtractのNamespace Bindingsの編集

マッピングを開くと、要素および属性名の前に名前空間接頭辞が表示されます。**「Name」**が空白のままである場合は、かわりに名前空間URIが表示されます。

注意

XSDに2つ以上の名前空間が含まれている場合は、ビジュアル・エディタでの出力への要素のマップがサポートされません。**「Source」**タブに切り替え、名前空間を手動で処理する必要があります。**Namespace Bindings**の「Add」ボタンを使用して、名前空間の前準備を実施します。次に、ソース・コードで次のように使用します。

Name = `myNs`

Value = `http://www.w3c.org/foo`

次のように記述できます。

```
myNs:element1
```

次のかわりに使用します。

```
{http://www.w3c.org/foo}element1
```

サブタイプの選択

スキーマによって、汎用的なタイプとなる要素が定義されることもありますが、実際に要素の特定のタイプとなるものは、処理されるXMLに含められます。スキーマに、汎用タイプのサブタイプも定義されている場合は、**「Select subtype」**アクションを使用することもできます。この場合は、次に示すダイアログが開きます。サブタイプを選択すると、スキーマ・ツリーの要素は選択したタイプであるかのように処理されます。このように、マッピング・エディタを使用してこの要素のマッピングを定義できます。この情報はマッピング・ソースにも格納されます。タイプ・オーバーライド・タグ(p.429)を参照してください。

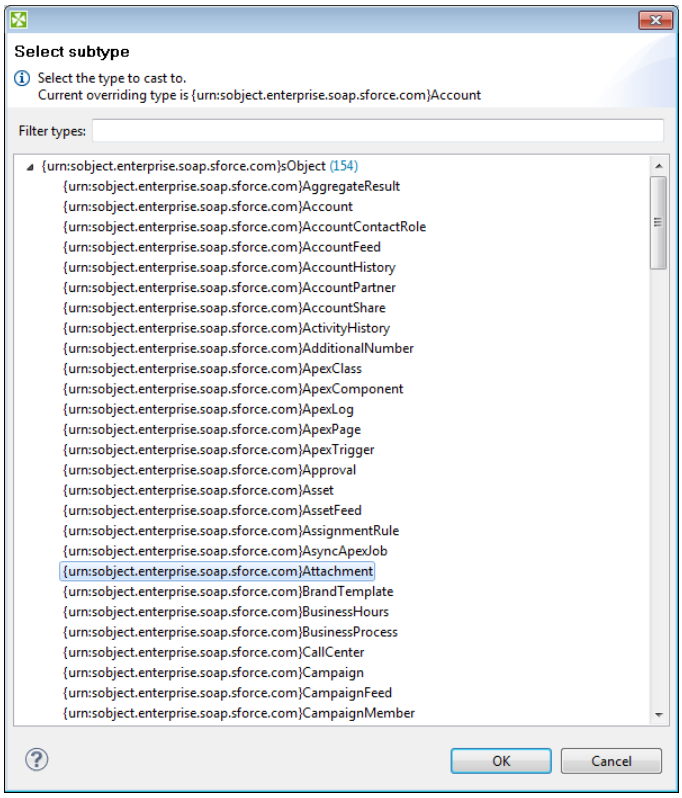


図53.23. XMLExtractでのサブタイプの選択

注意

次のXMLファイルを検討してください。

```
<customer name="attribute_value">
<name>element_value</name>
</customer>
```

この場合、要素customerには「name」属性および同じ名前の子要素があります。「name」属性と要素nameの両方を出力メタデータにマップする必要がある場合、次のマッピングは正しくありません。

```
<Mappings>
 <Mapping element="customer" outPort="0"
 xmlFields="{}name"
 cloverFields="field1">
 <Mapping element="name" useParentRecord="true">
 </Mapping>
 </Mapping>
</Mappings>
```

このマッピングの結果、field1とfield2の両方に要素nameの値が含まれます。「name」属性の値をいくつかの出力メタデータ・フィールドに読み取る必要がある場合は、次のマッピングを使用します。

```
<Mappings>
 <Mapping element="customer" outPort="0"
 xmlFields="{}name"
 cloverFields="field2">
 <Mapping element="name" useParentRecord="true"
  xmlFields="../{}name"
  cloverFields="field1">
 </Mapping>
   </Mapping>
</Mappings>
```

XMLReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

XMLReaderは、XMLファイルからデータを読み取ります。これは、元の**XMLXPathReader**および**XMLExtract**に代わる、強力な新しいコンポーネントです。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
XMLReader	XMLファイル	0-1	1-n	いいえ	はい	いいえ	いいえ	いいえ	いいえ

説明

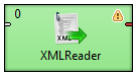
1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します(**DataGenerator**および**MultiLevelReader**)。詳細は、変換の戻り値(p.283)を参照してください。**XMLReader**、**XMLExtract**および**XMLXPathReader**は、それぞれの「**Mapping**」属性または「**Mapping URL**」属性で定義されているようにデータをポートに送信します。

概要

XMLReaderは、XMLファイルからデータを読み取ります。圧縮ファイル、コンソール、入力ポートおよびディクショナリからもデータを読み取ることができます。このコンポーネントは、XMLファイルの読取りも可能な**XMLExtract**よりも低速です。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	ポート読取りに使用。「入力ポートからの読取り」の項(p.300)を参照してください。	1つのフィールド(`byte`、`cbyte`、`string`)。

ポート・タイプ	番号	必須	説明	メタデータ
出力	0 ... n-1	はい	正しいデータ・レコード用。マッピングによっては複数の出力ポートへの接続が必要になる場合があります。	任意[1)]
	n	いいえ	エラー・ポート	制限付き書式[2)]

説明:

1) 各出力ポート上のメタデータは同じである必要はありません。これらの各メタデータは自動入力関数(p.132)を使用できます。

2) 最後の出力ポートをエラー・ロギングに使用する場合は、メタデータを固定形式にする必要があります。フィールド名は任意で、フィールド・タイプは次のとおりです。

- `integer`: エラーが発生した出力ポートの番号。
- `integer`: レコード番号(ソースおよびポートごと)
- `integer`: フィールド番号
- `string`: フィールド名
- `string`: エラーの原因となった値
- `string`: エラー・メッセージ
- オプションのフィールド: `string`: ソース名

XMLReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	読み取るデータ・ソース(XMLファイル、コンソール、入力ポート、ディクショナリ)を指定します。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Charset		読み取られたレコードのエンコーディング。ファイルから読み取るときは、キャラクタ・セットが(手動で指定しないかぎり)自動的に検出されます。 **重要** ポートまたはディクショナリから読み取る場合は、常に明示的に**キャラクタ・セット**を設定します(そうしないとエラーが発生します)。ファイルからの読取りのときのような自動検出はありません。	ISO-8859-1 (デフォルト) | <other encodings>
Data policy		エラーの発生時に必要な処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Controlled | Lenient
Mapping URL	1)	マッピング定義が含まれる外部テキスト・ファイル。詳細は、XMLReaderのマッピング定義(p.454)を参照してください。	

属性	必須	説明	可能な値
Mapping	1)	出力ポートへの入力XML構造のマッピング。詳細は、XMLReaderのマッピング定義(p.454)を参照してください。	
Advanced			
XML features		検証するXML機能に関連する個々のtrue/false式のシーケンス。これらの式はそれぞれセミコロンで区切られています。詳細は、XML機能(p.307)を参照してください。	

説明:

1) これらのうちどちらかを指定する必要があります。両方が指定された場合は、**マッピングURL**が優先されます。

詳細説明

例53.9. XMLReaderでのマッピング

```
<Context xpath="/employees/employee" outPort="0">
    <Mapping nodeName="salary" cloverField="basic_salary"/>
    <Mapping xpath="name/firstname" cloverField="firstname"/>
    <Mapping xpath="name/surname" cloverField="surname"/>
    <Context xpath="child" outPort="1" parentKey="empID" generatedKey="parentID"/>
    <Context xpath="benefits" outPort="2" parentKey="empID;jobID" generatedKey="empID;jobID"
                              sequenceField="seqKey" sequenceId="Sequence0">
        <Context xpath="financial" outPort="3" parentKey="seqKey" generatedKey="seqKey"/>
    </Context>
    <Context xpath="project" outPort="4" parentKey="empID;jobID" generatedKey="empID;jobID">
       <Context xpath="customer" outPort="5" parentKey="projName;projManager;
inProjectID;Start"
                                 generatedKey="joinedKey"/>
    </Context>
</Context>
```

注意

<Context>タグのネストした構造は、入力XMLファイル内のXML要素のネストした構造に似ています。

ただし、「**Mapping**」属性ではすべてのXML構造をコピーする必要はなく、XMLファイル全体における指定されたレベルから開始できます。

XMLReaderのマッピング定義

1. **マッピング**定義(「**Mapping URL**」属性と「**Mapping**」属性で指定されるファイルの内容の両方)は、それぞれ<Context>タグで構成されており、ここにも属性が含まれ、要素名をCloverフィールドにマッピングできます。
2. 各<Context>タグで、一連のネストされた<Mapping>タグを囲むことができます。これらにより、XML要素の名前をCloverフィールドに変更できます。
3. これらの各<Context>タグにはいくつかのXMLReaderのコンテキスト・タグの属性(p.448)が含まれ、<Mapping>タグにはXMLReaderのマッピング・タグの属性(p.449)が含まれます。

重要

デフォルトでは、マッピング定義は**暗黙的**です。要素(たとえばsalary)は同じ名前(salary)のフィールドに対して、次のように自動マップされるため、自分で記述する必要は**ありません**。

```
    <Mapping xpath="salary" cloverField="salary"/>
```

このため、明示的マッピングはフィールドに固有要素からのデータを移入する場合にのみ使用してください。

4. **XMLReaderコンテキスト・タグおよびマッピング・タグ**
   - **空のコンテキスト・タグ(子なし)**

     <Context xpath="xpathexpression" XMLReaderのコンテキスト・タグの属性(p.448) />
   - **空ではないコンテキスト・タグ(子がある親)**

     <Context xpath="xpathexpression" XMLReaderのコンテキスト・タグの属性(p.448) >

```
(nested Context and Mapping elements (only children, parents with one or
more children, etc.)

</Context>
```

- **空のマッピング・タグ(名前変更タグ)**
  - xpathを使用:

    ```
    <Mapping xpath="xpathexpression" XMLReaderのマッピング・タグの属性(p.449) />
    ```

  - nodeNameを使用:

    ```
    <Mapping nodeName="elementname" XMLReaderのマッピング・タグの属性(p.449) />
    ```

5. **XMLReaderのコンテキスト・タグおよびマッピング・タグの属性**

1) **XMLReaderのコンテキスト・タグの属性**

- xpath

  必須

  xpath式は、任意のXPath問合せにできます。

  例: `xpath="/tagA/.../tagJ"`

- outPort

  オプション

  データの送信先となる出力ポートの数。定義されない場合、このレベルの**マッピング**からのデータは、このレベルの**マッピング**を使用して送信されません。

  例: `outPort="2"`

- parentKey

  parentKeyおよびgeneratedKeyの両方を指定する必要があります。

  セミコロン、コロンまたはパイプで区切られた、次の親レベルのメタデータ・フィールドのシーケンス。これらすべてのフィールドの数およびデータ型は、generatedKey属性内で同じにする必要があります。同じでない場合は、すべての値が連結されて、一意の文字列値が作成されます。この場合、キーのフィールドは1つのみになります。

  例: `parentKey="first_name;last_name"`

  これらの属性の値を等しくすることにより、今後のこのようなレコードを確実に結合できます。

- generatedKey

  parentKeyおよびgeneratedKeyの両方を指定する必要があります。

  セミコロン、コロンまたはパイプで区切られた、指定されたレベルのメタデータ・フィールドのシーケンス。これらすべてのフィールドの数およびデータ型は、parentKey属性内で同じにする必要があります。同じでない場合は、すべての値が連結されて、一意の文字列値が作成されます。この場合、キーのフィールドは1つのみになります。

  例: `generatedKey="f_name;l_name"`

  これらの属性の値を等しくすることにより、今後のこのようなレコードを確実に結合できます。

- sequenceId

  parentKeyとgeneratedKeyのペアによってレコードの一意の識別が保証されない場合は、シーケンスを定義して使用できます。

  シーケンスのID。

  例: sequenceId="Sequence0"

- sequenceField

  parentKeyとgeneratedKeyのペアによってレコードの一意の識別が保証されない場合は、シーケンスを定義して使用できます。

  シーケンスの値が書き込まれる、指定されたレベルのメタデータ・フィールド。次にネストされたレベルのparentKeyとして機能できます。

  例: sequenceField="sequenceKey"

- namespacePaths

  オプション

  <Context>タグに指定された「xpath」属性に使用されるデフォルトの名前空間。

  パターン: namespacePaths='prefix1="URI1";...;prefixN="URIN"'

  例:
  namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'

注意

入力されたXMLファイルにデフォルトの名前空間が含まれている場合は、**マッピング**属性の対応する場所に、このnamespacePathsを指定する必要があることを覚えておいてください。さらに、namespacePathsは<Context>要素から継承され、<Mapping>要素によって使用されます。

2) **XMLReaderのマッピング・タグの属性**

- xpath

  xpathまたはnodeNameのいずれかを<Mapping>タグ内に指定する必要があります。

  XPathの問合せ。

  例: xpath="tagA/.../salary"

- nodeName

  xpathまたはnodeNameのいずれかを<Mapping>タグ内に指定する必要があります。nodeNameを使用する方が、xpathを使用するよりも速くなります。

  Cloverフィールドにマップされる XMLノード。

  例: nodeName="salary"

- cloverField

  必須

  XMLノードのマップ先となるCloverフィールド。

対応するレベルにおけるフィールドの名前。

例: `cloverFields="SALARY"`

- `trim`

  オプション

  先頭と末尾の空白を削除するかどうかを指定します。デフォルトでは、先頭と末尾の空白の両方が削除されます。

  例: `trim="false"` (空白は削除されません)

- `namespacePaths`

  オプション

  <Mapping>タグに指定された「xpath」属性に使用されるデフォルトの名前空間。

  パターン: `namespacePaths='prefix1="URI1";...;prefixN="URIN"'`

  例:
  `namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'`

注意

入力されたXMLファイルにデフォルトの名前空間が含まれている場合は、**マッピング**属性の対応する場所に、このnamespacePathsを指定する必要があることを覚えておいてください。さらに、namespacePathsは<Context>要素から継承され、<Mapping>要素によって使用されます。

複数値フィールドの読取り

Clover 3.3の時点では、複数値フィールドの読取りがサポートされています。ただし、リストの読取りのみが可能です(複数値フィールド(p.168)を参照してください)。

注意

マップの読取りは、(マップの値としてのすべてのデータ型について)純粋なstringの読取りとして扱われます。

例53.10. XMLReaderを使用したリストの読取り

次の要素が含まれているサンプルの入力ファイル(コードは一部抜粋):

```
...
<attendees>John</attendees>
<attendees>Vicky</attendees>
<attendees>Brian</attendees>
...
```

次のマッピングを使用すると、コンポーネントでリード・バックできます。

```
<Mapping xpath="attendees" cloverField="attendanceList"/>
```

ここで、attendanceListは、使用するメタデータのフィールドです。このメタデータは、コンポーネントの出力エッジに割り当てる必要があります。グラフの実行後、フィールドにはこのようなXMLデータが移入されます(それが「**View data**」に表示されるデータです)。

```
[John,Vicky,Brian]
```

XMLXPathReader

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第43章「リーダーの共通プロパティ」(p.296)

目的に適した**リーダー**を見つけるには、リーダーの比較(p.297)を参照してください。

要約

XMLXPathReaderは、XMLファイルからデータを読み取ります。

コンポーネント	データ・ソース	入力ポート	出力ポート	全出力に送信[1]	各出力に送信[2]	変換	変換が必要	Java	CTL
XMLXPathReader	XMLファイル	0-1	1-n	いいえ	はい	いいえ	いいえ	いいえ	いいえ

説明

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

2) コンポーネントは、変換の戻り値を使用して、各データ・レコードを異なる出力ポートに送信します(**DataGenerator**および**MultiLevelReader**)。詳細は、変換の戻り値(p.283)を参照してください。**XMLExtract**および**XMLXPathReader**は、それぞれの「**Mapping**」属性または「**Mapping URL**」属性で定義されたとおりにデータをポートに送信します。

概要

XMLXPathReaderは、(DOMパーサーを使用して) XMLファイルからデータを読み取ります。圧縮ファイル、コンソール、入力ポートおよびディクショナリからもデータを読み取ることができます。このコンポーネントは、XMLファイルの読取りも可能な**XMLExtract**よりも低速です。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	ポート読取りに使用。入力ポートからの読取り(p.300)を参照してください。	1つのフィールド(`byte`、`cbyte`、`string`)。

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	はい	正しいデータ・レコード用。	任意[1)]
	1-n	2)	正しいデータ・レコード用。	任意[1)](ポートごとに異なるメタデータを保持できます)

説明:

1) 各出力ポートのメタデータが同じである必要はありません。メタデータは自動入力関数(p.132)を使用できます。注意: `source_timestamp`関数および`source_size`関数は、ファイルから直接読み取る場合のみ機能します(ファイルがアーカイブである場合またはリモートの場所に保存されている場合は、タイムスタンプが空になり、サイズは0になります)。

2) その他の出力ポートは、マッピングでそのポートを必要とする場合に必要です。

XMLXPathReaderの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	読み取るデータ・ソース(XMLファイル、コンソール、入力ポート、ディクショナリ)を指定します。リーダーにサポートされているファイルURL形式(p.297)を参照してください。	
Charset		読み取られたレコードのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Data policy		エラーの発生時に必要な処理を決定します。詳細は、データ・ポリシー(p.306)を参照してください。	Strict (デフォルト) | Controlled | Lenient
Mapping URL	1)	マッピング定義が含まれる外部テキスト・ファイル。詳細は、XMLXPathReaderのマッピング定義(p.454)を参照してください。	
Mapping	1)	出力ポートへの入力XML構造のマッピング。詳細は、XMLXPathReaderのマッピング定義(p.454)を参照してください。	
Advanced			
XML features		検証するXML機能に関連する個々の`true/false`式のシーケンス。これらの式はそれぞれセミコロンで区切られています。詳細は、XML機能(p.307)を参照してください。	
Number of skipped mappings		すべてのソース・ファイルにわたって継続的にスキップされるマッピング数。入力レコードの選択(p.305)を参照してください。	0-N
Max number of mappings		すべてのソース・ファイルにわたって継続的に読み取られる最大レコード数。入力レコードの選択(p.305)を参照してください。	0-N

説明:

1) これらのうちどちらかを指定する必要があります。両方が指定された場合は、**マッピングURL**が優先されます。

詳細説明

例53.11. XMLXPathReaderでのマッピング

```
<Context xpath="/employees/employee" outPort="0">
    <Mapping nodeName="salary" cloverField="basic_salary"/>
    <Mapping xpath="name/firstname" cloverField="firstname"/>
    <Mapping xpath="name/surname" cloverField="surname"/>
    <Context xpath="child" outPort="1" parentKey="empID" generatedKey="parentID"/>
    <Context xpath="benefits" outPort="2" parentKey="empID;jobID" generatedKey="empID;jobID"
                              sequenceField="seqKey" sequenceId="Sequence0">
        <Context xpath="financial" outPort="3" parentKey="seqKey" generatedKey="seqKey"/>
    </Context>
    <Context xpath="project" outPort="4" parentKey="empID;jobID" generatedKey="empID;jobID">
        <Context xpath="customer" outPort="5" parentKey="projName;projManager;
inProjectID;Start"
                                  generatedKey="joinedKey"/>
    </Context>
</Context>
```

注意

<Context>タグのネストした構造は、入力XMLファイル内のXML要素のネストした構造に似ています。

ただし、「**Mapping**」属性ではすべてのXML構造をコピーする必要はなく、XMLファイル全体における指定されたレベルから開始できます。

XMLXPathReaderのマッピング定義

1. **マッピング**定義(「**Mapping URL**」属性と「**Mapping**」属性で指定されるファイルの内容の両方)は、それぞれ<Context>タグで構成されており、ここにも属性が含まれ、要素名をCloverフィールドにマッピングできます。
2. 各<Context>タグで、一連のネストされた<Mapping>タグを囲むことができます。これらにより、XML要素の名前をCloverフィールドに変更できます。
3. これらの<Context>および<Mapping>タグにはそれぞれ、XMLXPathReaderのコンテキスト・タグの属性(p.455)およびXMLXPathReaderのマッピング・タグの属性(p.456)が含まれます。

重要

デフォルトでは、マッピング定義は**暗黙的**です。このため、要素(たとえばsalary)は同じ名前(salary)のフィールドに自動的にマップされ、次のように記述する必要は**ありません**。

```
    <Mapping xpath="salary" cloverField="salary"/>
```

このため、明示的マッピングはフィールドに別の名前の要素からのデータを移入する場合にのみ使用します。

4. **XMLXPathReaderのコンテキスト・タグおよびマッピング・タグ**
   - **空のコンテキスト・タグ(子なし)**

     <Context xpath="xpathexpression" XMLXPathReaderのコンテキスト・タグの属性(p.455) />
   - **空ではないコンテキスト・タグ(子がある親)**

     <Context xpath="xpathexpression" XMLXPathReaderのコンテキスト・タグの属性(p.455) >

```
(nested Context and Mapping elements (only children, parents with one or
more children, etc.)

</Context>
```

- **空のマッピング・タグ(名前変更タグ)**
  - xpathを使用:

    ```
    <Mapping xpath="xpathexpression"
    ```
    XMLXPathReaderのマッピング・タグの属性(p.456)
    ```
    />
    ```
  - nodeNameを使用:

    ```
    <Mapping nodeName="elementname"
    ```
    XMLXPathReaderのマッピング・タグの属性(p.456)
    ```
    />
    ```

5. **XMLXPathReaderのコンテキスト・タグおよびマッピング・タグの属性**

1) **XMLXPathReaderのコンテキスト・タグの属性**

- xpath

  必須

  xpath式は、任意のXPath問合せにできます。

  例: `xpath="/tagA/.../tagJ"`

- outPort

  オプション

  データの送信先となる出力ポートの数。定義されない場合、このレベルの**マッピング**からのデータは、このレベルの**マッピング**を使用して送信されません。

  例: `outPort="2"`

- parentKey

  parentKeyおよびgeneratedKeyの両方を指定する必要があります。

  セミコロン、コロンまたはパイプで区切られた、次の親レベルのメタデータ・フィールドのシーケンス。これらすべてのフィールドの数およびデータ型は、generatedKey属性内で同じにする必要があります。同じでない場合は、すべての値が連結されて、一意の文字列値が作成されます。この場合、キーのフィールドは1つのみになります。

  例: `parentKey="first_name;last_name"`

  これらの属性の値を等しくすることにより、今後のこのようなレコードを確実に結合できます。

- generatedKey

  parentKeyおよびgeneratedKeyの両方を指定する必要があります。

  セミコロン、コロンまたはパイプで区切られた、指定されたレベルのメタデータ・フィールドのシーケンス。これらすべてのフィールドの数およびデータ型は、parentKey属性内で同じにする必要があります。同じでない場合は、すべての値が連結されて、一意の文字列値が作成されます。この場合、キーのフィールドは1つのみになります。

  例: `generatedKey="f_name;l_name"`

  これらの属性の値を等しくすることにより、今後のこのようなレコードを確実に結合できます。

- sequenceId

  parentKeyとgeneratedKeyのペアによってレコードの一意の識別が保証されない場合は、シーケンスを定義して使用できます。

  シーケンスのID。

  例: sequenceId="Sequence0"

- sequenceField

  parentKeyとgeneratedKeyのペアによってレコードの一意の識別が保証されない場合は、シーケンスを定義して使用できます。

  シーケンスの値が書き込まれる、指定されたレベルのメタデータ・フィールド。次にネストされたレベルのparentKeyとして機能できます。

  例: sequenceField="sequenceKey"

- namespacePaths

  オプション

  <Context>タグに指定されたxpath属性に使用されるデフォルトの名前空間。

  パターン: namespacePaths='prefix1="URI1";...;prefixN="URIN"'

  例:
  namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'

注意

入力されたXMLファイルにデフォルトの名前空間が含まれている場合は、**マッピング**属性の対応する場所に、このnamespacePathsを指定する必要があることを覚えておいてください。さらに、namespacePathsは<Context>要素から継承され、<Mapping>要素によって使用されます。

2) **XMLXPathReaderのマッピング・タグの属性**

- xpath

  xpathまたはnodeNameのいずれかを<Mapping>タグ内に指定する必要があります。

  XPathの問合せ。

  例: xpath="tagA/.../salary"

- nodeName

  xpathまたはnodeNameのいずれかを<Mapping>タグ内に指定する必要があります。nodeNameを使用する方が、xpathを使用するよりも速くなります。

  Cloverフィールドにマップされる XMLノード。

  例: nodeName="salary"

- cloverField

  必須

  XMLノードのマップ先となるCloverフィールド。

対応するレベルにおけるフィールドの名前。

例: `cloverFields="SALARY"`

- `trim`

  オプション

  先頭と末尾の空白を削除するかどうかを指定します。デフォルトでは、先頭と末尾の空白の両方が削除されます。

  例: `trim="false"` (空白は削除されません)

- `namespacePaths`

  オプション

  `<Mapping>`タグに指定された`xpath`属性に使用されるデフォルトの名前空間。

  パターン: `namespacePaths='prefix1="URI1";...;prefixN="URIN"'`

  例:
  `namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'`

注意

入力されたXMLファイルにデフォルトの名前空間が含まれている場合は、**マッピング**属性の対応する場所に、この`namespacePaths`を指定する必要があることを覚えておいてください。さらに、`namespacePaths`は`<Context>`要素から継承され、`<Mapping>`要素によって使用されます。

複数値フィールドの読取り

Clover 3.3の時点では、複数値フィールドの読取りがサポートされています。ただし、リストの読取りのみが可能です(複数値フィールド(p.168)を参照してください)。

注意

マップの読取りは、(マップの値としてのすべてのデータ型について)純粋な`string`の読取りとして扱われます。

例53.12. XMLXpathReaderによるリストの読取り

次の要素が含まれているサンプルの入力ファイル(コードは一部抜粋):

```
...
<attendees>John</attendees>
<attendees>Vicky</attendees>
<attendees>Brian</attendees>
...
```

次のマッピングを使用すると、コンポーネントでリード・バックできます。

```
<Mapping xpath="attendees" cloverField="attendanceList"/>
```

ここで、attendanceListは、使用するメタデータのフィールドです。このメタデータは、コンポーネントの出力エッジに割り当てる必要があります。グラフの実行後、フィールドにはこのようなXMLデータが移入されます(それが「**View data**」に表示されるデータです)。

```
[John,Vicky,Brian]
```

第54章 ライター

コンポーネントとは何かを理解していることを想定しています。概要は、第19章「コンポーネント」(p.97)を参照してください。

グラフ内の一部のコンポーネントのみが端末ノードです。これらは**ライター**と呼ばれます。

ライターは、データを出力ファイル(ローカルおよびリモートの両方)に書き込むか、接続済のオプションの出力ポートを介して送信するか、ディクショナリに書き込みます。コンポーネントの1つはデータの破棄のみを行います。これも端末ノードであるため、ここではそれについて説明します。

コンポーネントには様々なプロパティを設定できます。ただし、一部のものが共通する場合があります。すべてのコンポーネントに共通のプロパティもあれば、ほとんどのコンポーネントに共通のプロパティもあり、**ライター**のみに共通のプロパティもあります。次のことを学習している必要があります。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

書き込む内容によって、**ライター**を区別できます。

- コンポーネントの1つはデータを破棄します。
  - Trash(p.550)はデータを破棄します。

その他の**ライター**はファイルにデータを書き込みます。

- フラット・ファイル:
  - UniversalDataWriter(p.552)は、フラット・ファイル(デリミタ付きまたは固定長)にデータを書き込みます。
- その他のファイル:
  - CloverDataWriter(p.461)は、Cloverバイナリ形式でファイルにデータを書き込みます。
  - XLSDataWriter(p.555)は、XLSまたはXLSXファイルにデータを書き込みます。
  - StructuredDataWriter(p.546)は、ユーザー定義の構造でファイルにデータを書き込みます。
  - XMLWriter(p.558)は、入力データ・レコードからXMLファイルを作成します。
  - DBFDataWriter(p.470)は、dbaseファイルにデータを書き込みます。
  - HadoopWriter(p.485)は、Hadoopシーケンス・ファイルにデータを書き込みます。

その他の**ライター**はデータベースにデータをロードします。

- データベース・ライター:
  - DBOutputTable(p.473)は、JDBCドライバを使用してデータベースにデータをロードします。
  - QuickBaseRecordWriter(p.530)は、**QuickBase**オンライン・データベースにデータを書き込みます。
  - QuickBaseImportCSV(p.528)は、**QuickBase**オンライン・データベースにデータを書き込みます。
  - LotusWriter(p.511)は、**Lotus Notes**および**Lotus Domino**データベースにデータを書き込みます。

- 高速データベース固有のライター(バルク・ローダー):
  - DB2DataWriter(p.464)は、DB2クライアントを使用してDB2データベースにデータをロードします。
  - InfobrightDataWriter(p.487)は、Infobrightクライアントを使用してInfobrightデータベースにデータをロードします。
  - InformixDataWriter(p.489)は、Informixクライアントを使用してInformixデータベースにデータをロードします。
  - MSSQLDataWriter(p.514)は、MSSQLクライアントを使用してMSSQLデータベースにデータをロードします。
  - MySQLDataWriter(p.518)は、MYSQLクライアントを使用してMYSQLデータベースにデータをロードします。
  - OracleDataWriter(p.521)は、Oracleクライアントを使用してOracleデータベースにデータをロードします。
  - PostgreSQLDataWriter(p.525)は、PostgreSQLクライアントを使用してPostgreSQLデータベースにデータをロードします。

その他の**ライター**は、電子メールまたはJMSメッセージを送信するか、ディレクトリ構造を書き込みます。

- 電子メール:
  - EmailSender(p.481)は、データ・レコードを電子メールに変換します。
- JMSメッセージ:
  - JMSWriter(p.501)は、データ・レコードをJMSメッセージに変換します。
- ディレクトリ構造:
  - LDAPWriter(p.509)は、データ・レコードをディレクトリ構造に変換します。

CloverDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

CloverDataWriterは、内部のバイナリCloverデータ形式でファイルにデータを書き込みます。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
CloverDataWriter	Cloverバイナリ・ファイル	1	0	いいえ	いいえ	いいえ	いいえ

概要

CloverDataWriterは、内部のバイナリCloverデータ形式で(ローカルまたはリモートの)ファイルにデータを書き込みます。出力ファイルの圧縮、およびコンソールまたはディクショナリへのデータの書込みも実行できます。

注意

CloverETLのバージョン2.9以上では、**CloverDataWriter**は出力ファイルにバージョン番号を含むヘッダーも書き込みます。このため、**CloverDataReader**は、Cloverバイナリ形式のファイルにこのようなバージョン番号付きのヘッダーが含まれることを予期します。
CloverDataReader 2.9で、より古いバージョンの**CloverETL**によって書き込まれたファイルを読み取ることはできず、また、これらのより古いバージョンで、**CloverDataWriter** 2.9によって書き込まれたデータを読み取ることはできません。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	受信したデータ・レコード用	任意

CloverDataWriterの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	受信したデータの書込み先(Cloverデータ・ファイル、コンソール、ディクショナリ)を指定する属性。ライターにサポートされているファイルURL形式(p.310)を参照してください。詳細は、出力ファイル構造(p.462)も参照してください。	
Append		デフォルトでは、新しいレコードで古いレコードが上書きされます。`true`に設定された場合、出力ファイルに保存されている古いレコードに新しいレコードが追加されます。	false (デフォルト) | true
Save metadata		デフォルトでは、メタデータ定義付きで保存されるファイルはありません。`true`に設定すると、メタデータがメタデータ・ファイルに保存されます。詳細は、出力ファイル構造(p.462)を参照してください。	false (デフォルト) | true
Save index [1)]		デフォルトでは、レコードのインデックス付きで保存されるファイルはありません。`true`に設定すると、レコードのインデックスがインデックス・ファイルに保存されます。詳細は、出力ファイル構造(p.462)を参照してください。	false (デフォルト) | true
Advanced			
Create directories		デフォルトでは、存在しないディレクトリは作成されません。`true`に設定すると、これらが作成されます。	false (デフォルト) | true
Compress level		圧縮レベルを設定します。デフォルトでは、zip圧縮レベルが使用されます。レベル`0`は、圧縮なしのアーカイブを意味します。	-1 (デフォルト) | 0-9
Number of skipped records		スキップするレコード数。出力レコードの選択(p.317)を参照してください。	0-N
Max number of records		出力ファイルに書き込まれるレコードの最大数。出力レコードの選択(p.317)を参照してください。	0-N

説明:

1) これは**非推奨**の属性です。

詳細説明

出力ファイル構造

- **アーカイブされない出力ファイル**

  作成されたファイルをアーカイブまたは圧縮(あるいはその両方)しない場合、出力ファイルは`filename` (データが含まれるファイルの場合)、`filename.idx` (インデックスが含まれるファイルの場合)および`filename.fmt` (メタデータが含まれるファイルの場合)という名前で個別に保存されます。作成されるこれら3つのすべてのファイル名について、`filename`にはファイルの拡張子(ある場合)が含まれます。

- **アーカイブされる出力ファイル**

  出力ファイルがアーカイブまたは圧縮(あるいはその両方)される場合、ファイルのタイプに関係なく、ファイルは`DATA/filename`、`INDEX/filename.idx`および`META/filename.fmt`という内部構造を持ちます。ここで、これら3つの名前すべてについて、`filename`にはファイルの拡張子(ある場合)が含まれます。

例54.1. アーカイブされる出力ファイルの内部構造

```
DATA/employees.clv, INDEX/employees.clv.idx, META/employees.clv.fmt.
```

DB2DataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

DB2DataWriterは、DB2データベースにデータをロードします。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
DB2DataWriter	データベース	0-1	0-1	いいえ	いいえ	いいえ	いいえ

概要

DB2DataWriterは、DB2データベース・クライアントを使用してデータベースにデータをロードします。データは、入力ポートを介してまたは入力ファイルから読み取ることができます。入力ポートが他のコンポーネントに接続されていない場合、コンポーネントで指定される入力ファイルにデータが含まれている必要があります。その他のコンポーネントをオプションの出力ポートに接続すると、そのコンポーネントによりエラーに関する情報を記録できます。DB2データベース・クライアントがlocalhostにインストールおよび構成されている必要があります。サーバーおよびデータベースがカタログ化されている必要もあります。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	1)	データベースにロードされるレコード	任意
出力	0	いいえ	正しくないレコードに関する情報	DB2DataWriterのエラー・メタデータ(p.465)[2]

説明:

1): ロードするデータを含むファイル(**Loader input file**)が指定されていない場合は、入力ポートが接続されている必要があります。

2): **エラー・メタデータ**は自動入力関数(p.132)を使用できません。

表54.1. DB2DataWriterのエラー・メタデータ

フィールド番号	フィールド名	データ型	説明
0	<any_name1>	integer	正しくないレコードの番号(レコードには1から始まる番号が付けられます)
1	<any_name2>	integer	(デリミタ付きレコードの)正しくないフィールドの数。フィールドには1から始まる番号が付けられます。| (固定長レコードの)正しくないフィールドのオフセット
2	<any_name3>	string	エラー・メッセージ

DB2DataWriterの属性

属性	必須	説明	可能な値
Basic			
File metadata		外部ファイルのメタデータ。デリミタ付きである必要があります。最後の列以外のすべての列には、末尾に同一の1文字のデリミタが付いています。最後の列の末尾に付く最後のデリミタは\nです。デリミタをフィールド値に含めることはできません。	
Database	はい	レコードがロードされるデータベースの名前。	
Database table	はい	レコードがロードされるデータベース表の名前。	
User name	はい	データベース・ユーザー。	
Password	はい	データベース・ユーザーのパスワード。	
Load mode		データのロード時に実行されるアクションのモード。詳細は、Load mode(p.468)を参照してください。	insert (デフォルト) | replace | restart | terminate
Field mapping	1)	セミコロン、コロンまたはパイプで区切られる個々のマッピングのシーケンス($CloverField:=DBField)。詳細は、DBフィールドへのCloverフィールドのマッピング(p.468)を参照してください。	
Clover fields	1)	セミコロン、コロンまたはパイプで区切られるCloverフィールドのシーケンス。詳細は、DBフィールドへのCloverフィールドのマッピング(p.468)を参照してください。	
DB fields	1)	セミコロン、コロンまたはパイプで区切られるDBフィールドのシーケンス。詳細は、DBフィールドへのCloverフィールドのマッピング(p.468)を参照してください。	
Advanced			
Loader input file	2)	パスを含む、ロードする入力ファイルの名前。詳細は、Loader input file(p.469)を参照してください。	

属性	必須	説明	可能な値
Parameters		loadメソッドでパラメータとして使用できるすべてのパラメータ。これらの値は、key=valueという形式のペアまたはkeyのみ(key値がブールのtrueの場合)のシーケンスに、セミコロン、コロンまたはパイプで区切られて含まれています。パラメータの値にデリミタが含まれる場合は、その値を二重引用符で囲む必要があります。	
Rejected records URL (on server)		拒否されたレコードが保存される、DB2サーバー上のファイルのパスを含む名前。データベース・ユーザーが所有するディレクトリ内の場所である必要があります。	
Batch file URL		db2ロード・ユーティリティのconnect、loadおよびdisconnectコマンドが保存されるファイルのURL。通常、バッチ・ファイルは自動的に生成され、現行ディレクトリに保存され、ロードが終了すると削除されます。**「Batch file URL」**が指定されている場合、コンポーネントはそのファイルをそのまま使用しようとし(ファイルが存在しないか長さが0の場合にのみ生成し)、ロードが終了してもファイルを削除しません(バッチ・ファイルには、明示的に提供されない場合はランダムに生成される一時データ・ファイルの名前が含まれるため、この属性は**「Loader input file」**属性と組み合せて使用することが適切です)。パスに空白を含めることはできません。	
DB2 command interpreter		DB2コマンド(connect、load、disconnect)でスクリプトを実行する必要があるインタプリタ。interpreterName [parameters] ${} [parameters]という形式にする必要があります。この ${} 式は、このスクリプト・ファイルの名前で置き換えます。	
Use pipe transfer		デフォルトでは、入力ポートから受信したデータは一時ファイルに書き込まれた後、コンポーネントにより読み取られます。trueに設定すると、UNIXでは、入力ポートから受信したデータ・レコードが一時ファイルではなくパイプに送信されます。	false (デフォルト) | true
Column delimiter		**「File metadata」**または入力エッジ上のメタデータ(**「File metadata」**が指定されていない場合)の最初の1文字のフィールド・デリミタ。データ・ファイルで各列のデリミタとして使用される文字です。デリミタをフィールド値に含めることはできません。**「Parameters」**属性のcoldelパラメータの値を指定することでも、同じデリミタを設定できます。**「Column delimiter」**が設定されている場合、**「Parameters」**のcoldelは無視されます。	
Number of skipped records		スキップするレコード数。デフォルトでは、スキップされるレコードはありません。この属性は、データが入力ポートを介して受信される場合にのみ適用されます。それ以外の場合は無視されます。	0 (デフォルト) | 1-N
Max number of records		データベースにロードされるレコードの最大数。**「Parameters」**属性のrowcountパラメータの値を指定することによって、同じことを設定できます。**「Parameters」**でrowcountが設定されている場合、**「Max number of records」**属性は無視されます。	all (デフォルト) | 0-N
Max error count		その数に達するとロードが停止するレコードの最大数。数値が明示的に設定され、その数値に達した場合、プロセスは	all (デフォルト) | 0-N

属性	必須	説明	可能な値
		RESTARTモードで続行できます。REPLACEモードでは、プロセスは最初から続行されます。「**Parameters**」属性の`warningcount`を使用しても、同じ数値を指定できます。`warningcount`が指定されている場合、「**Max error count**」は無視されます。	
Max warning count		出力されるエラー・メッセージまたは警告(あるいはその両方)の最大数。	999 (デフォルト) | 0-N
Fail on warnings		デフォルトでは、コンポーネントはエラーが発生したときに失敗します。属性を`true`に変更すると、警告が発生したときにコンポーネントが失敗するようにできます。背景: 基礎となる一括ローダー・ユーティリティが警告で終了した場合、その警告は単にコンソールに記録されます。基礎となる一括ローダーからの警告は以降の処理に深刻な影響を及ぼす可能性があるため、この動作は適切でない場合があります。たとえば、表領域を拡張できないと、すべてのデータ・レコードがデータベースにロードされず、想定したタスクが正常に完了しないことがあります。	false (デフォルト) | true

説明:

1) これらの関係の詳細は、DBフィールドへのCloverフィールドのマッピング(p.468)を参照してください。

2) 入力ポートが接続されていない場合、「**Loader input file**」が指定され、データが含まれている必要があります。詳細は、Loader input file(p.469)を参照してください。

詳細説明

DBフィールドへのCloverフィールドのマッピング

- **フィールド・マッピングが定義されている場合**

  フィールド・マッピングが定義されている場合、この属性で指定された各Cloverフィールドの値が、**フィールド・マッピング**属性でこのCloverフィールドが割り当てられた名前を持つDBフィールドに挿入されます。

- **CloverフィールドとDBフィールドの両方が定義されている場合**

  Cloverフィールドと**DBフィールド**の両方が定義されている(が**フィールド・マッピング**は定義されていない)場合、Cloverフィールド属性で指定された各Cloverフィールドの値が、DBフィールド属性の同じ位置にある**DBフィールド**に挿入されます。

  これら両方の属性のCloverフィールドおよびDBフィールドの数は、互いに等しい必要があります。いずれの数も、他の方法(ドル記号の接頭辞を付けたCloverフィールドの指定、DB関数の指定または問合せでの定数の指定)では定義されていないDBフィールドの数に等しい必要があります。

  Cloverフィールドのパターン:

  ```
  CloverFieldA;...;CloverFieldM
  ```

  DBフィールドのパターン:

  ```
  DBFieldA;...;DBFieldM
  ```

- **Cloverフィールドのみが定義されている場合**

  Cloverフィールド属性のみが定義されている(が**フィールド・マッピング**または**DBフィールド**(あるいはその両方)は定義されていない)場合、**Cloverフィールド**属性で指定された各Cloverフィールドの値が、DB表での位置が等しいDBフィールドに挿入されます。

  Cloverフィールド属性で指定されたCloverフィールドの数は、他の方法(ドル記号の接頭辞を付けたCloverフィールドの指定、DB関数の指定または問合せでの定数の指定)では定義されていないDB表内のDBフィールドの数に等しい必要があります。

  Cloverフィールドのパターン:

  ```
  CloverFieldA;...;CloverFieldM
  ```

- **マッピングが自動的に実行される場合**

  フィールド・マッピング、**Cloverフィールド**または**DBフィールド**のいずれも定義されていない場合、マッピング全体が自動的に実行されます。メタデータの各Cloverフィールドの値が、DB表内の同じ位置に挿入されます。

  すべてのCloverフィールドの数は、他の方法(ドル記号の接頭辞を付けたCloverフィールドの指定、DB関数の指定または問合せでの定数の指定)では定義されていないDB表内のDBフィールドの数に等しい必要があります。

Load mode

- `insert`

  ロードされたデータは、既存の表コンテンツを削除または変更せずにデータベース表に追加されます。

- `replace`

  データベース表内の既存データがすべて削除され、新しくロードされたデータが表に挿入されます。表定義およびインデックス定義はいずれも変更されません。

- `restart`

  以前に中断されたロード操作が再開されます。ロード操作は、ロード、ビルドまたは削除フェーズの最後の整合性ポイントから自動的に続行されます。

- `terminate`

  以前に中断されたロード操作が終了され、整合性ポイントをすぎていても、操作が開始した時点にロールバックされます。

Loader input file

「**Loader input file**」は、ロードするデータが含まれる入力ファイルのパスを含む名前です。名前付きパイプがかわりに使用されないかぎり、通常、このファイルがdbloadユーティリティに渡されるデータの一時記憶域となります。DB2クライアントはローカルホストでインストールおよび構成されている必要があります(『DB2 client setup overview』を参照してください)。サーバーおよびデータベースがカタログ化されている必要もあります。

- これが設定されていない場合、ローダー・ファイルがCloverまたはOSの一時ディレクトリに作成されるか(Windowsの場合)、一時ファイルのかわりに名前付きパイプが使用されます(Unixの場合)。ロードが終了するとファイルは削除されます。

- これが設定されている場合、指定したファイルが作成されます。これはデータがロードされた後も削除されず、グラフが実行されるたびに上書きされます。

- 入力ポートが接続されていない場合、このファイルが存在し、指定され、データベースにロードされるデータを含んでいる必要があります。これは削除も上書きもされません。

DBFDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

DBFDataWriterは、データをdbaseファイルに書き込みます。`Character/Number/Logical/Date` dBaseデータ型を処理します。バイナリ・データを書き込むため、入力メタデータは固定長である必要があります。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
DBFDataWriter	.dbfファイル	1	0	✖	✖	✖	✖

概要

DBFDataWriterは、データをdbaseファイルに書き込みます。

このコンポーネントは、単一のファイルまたはパーティション化されたファイルのコレクションを作成できます。

重要

出力データはローカルでのみ保存できます。リモート転送プロトコルを介したアップロードおよびZIPやTARアーカイブの作成はサポートされません。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	受信データ・レコード	固定長

DBFDataWriterの属性

属性	必須	説明	可能な値
Basic			
File URL	✔	データの書込み先(.dbfファイルへのパス)を指定します。ライターにサポートされているファイルURL形式(p.310)を参照してください。	
Charset		出力に書き込まれるレコードの文字エンコード。	ISO-8859-1 (デフォルト) | その他の8ビット固定幅エンコード
Append		デフォルト(false)では、レコードが空でないファイルに書き込まれると、以前のコンテンツがそのレコードで置き換えられます。trueに設定された場合、既存の出力ファイルの末尾に新しいレコードが追加されます。	false (デフォルト) | true
DBF type		作成されるDBFファイルのタイプ(ファイル・ヘッダーの最初のバイトによって決定されます)。選択するタイプが不明である場合、この属性はデフォルトのままにします。	0x03 FoxBASE+ (デフォルト) | Dbase III plus, no memo |その他のdbfタイプのバイト
Advanced			
Create directories		trueの場合、「**File URL**」のパスに含まれる存在しないディレクトリが自動的に作成されます。	false (デフォルト) | true
Records per file		各出力ファイルに書き込まれるレコードの最大数。指定する場合、ファイル名マスクにドル記号$ (桁数のプレースホルダ)を含める必要があります。ライターにサポートされているファイルURL形式(p.310)を参照してください。	1 - N
Number of skipped records		最初のレコードを出力ファイルに書き込む前にスキップするレコード数または行数。出力レコードの選択(p.317)を参照してください。	0 (デフォルト) - N
Max number of records		すべての出力ファイルに書き込まれるレコード/行の集計数。出力レコードの選択(p.317)を参照してください。	0-N
Exclude fields		出力に書き込まれないフィールド名のシーケンス(セミコロンで区切ります)。同じフィールドが「**Partition key**」の一部となる場合に使用できます。	

属性	必須	説明	可能な値
Partition key	[2]	複数の出力ファイル間でのレコードの分配を定義するフィールド名のシーケンス。同じ**パーティション・キー**を持つレコードは同じ出力ファイルに書き込まれます。フィールド名のセパレータとして、セミコロン(;)を使用します。選択した**パーティション・ファイル・タグ**に応じて、適切なプレースホルダ($または#)をファイル名マスクで使用します。異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition lookup table	[1]	出力ファイルに書き込まれるレコードを選択するために使用する参照表のID。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition file tag	[2] [2]	デフォルトでは、パーティション化された出力ファイルには番号が付けられます。この属性が`Key file tag`に設定されている場合、出力ファイルには**パーティション・キー・フィールド**または**パーティション出力フィールド**の値に基づいて名前が付けられます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	`Number file tag` (デフォルト) | `Key file tag`
Partition output fields	[1] [1]	出力ファイル名として使用される値が含まれる、**パーティション参照表**のフィールド。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition unassigned file name		未割当てのレコードがある場合に、それらのレコードを書き込むファイルの名前。指定されない場合、**パーティション参照表**にキー値が含まれていないデータ・レコードは破棄されます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	

[2]これら2つの属性を両方指定するか、または両方とも指定しないでください。

[1]これら2つの属性を両方指定するか、または両方とも指定しないでください。

DBOutputTable

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

DBOutputTableは、JDBCドライバを使用してデータベースにデータをロードします。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
DBOutputTable	データベース	1	0-2	✖	✖	✖	✖

概要

DBOutputTableは、JDBCドライバを使用してデータベースにデータをロードします。拒否されたレコードを送信して、使用可能なデータベースの一部に自動生成列を生成することもできます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	データベースにロードされるレコード	任意
出力	0	✖	拒否されたレコード用	入力0に基づく[1)]
	1	✖	返される値用	任意[2)]

説明:

1): 出力ポート0のメタデータには、入力からの任意の数のフィールド(同じ名前および型)とエラー情報用の最大2つの追加フィールドが含まれることがあります。入力メタデータは、その名前と型に基づいて自動的にマッ

プされます。2つのエラー・フィールドには任意の名前を付けることができ、自動入力関数(p.132)の`ErrCode`および`ErrText`に設定する必要があります。

2): 出力ポート1のメタデータには、少なくとも、問合せに指定される`returning`文によって返されるフィールドが含まれている必要があります(たとえば、`returning $outField1:=$inFieldA,$outField2:=update_count,$outField3:=$inFieldB`)。フィールドが自動的に名前別にマップされることはありません。`returning`文には、常に、マッピングを指定する必要があります。返されるレコードの数は、受信レコードの数と等しくなります。

DBOutputTableの属性

属性	必須	説明	可能な値
Basic			
DB connection	✔	使用されるDB接続のID。	
Query URL	1)	パスを含む、SQL問合せを定義する外部ファイルの名前。詳細は、問合せまたはDB表が定義されている場合(p.475)を参照してください。	
SQL query	1)	グラフで定義されているSQL問合せ詳細は、問合せまたはDB表が定義されている場合(p.475)を参照してください。SQL問合せエディタ(p.478)も参照してください。	
DB table	1)	DB表の名前。詳細は、問合せまたはDB表が定義されている場合(p.475)を参照してください。	
Field mapping	2)	セミコロン、コロンまたはパイプで区切られる個々のマッピングのシーケンス(`$CloverField:=DBField`)。詳細は、DBフィールドへのCloverフィールドのマッピング(p.477)を参照してください。	
Clover fields	2)	セミコロン、コロンまたはパイプで区切られるCloverフィールドのシーケンス。詳細は、DBフィールドへのCloverフィールドのマッピング(p.477)を参照してください。	
DB fields	2)	セミコロン、コロンまたはパイプで区切られるDBフィールドのシーケンス。詳細は、DBフィールドへのCloverフィールドのマッピング(p.477)を参照してください。	
Query source charset		SQL問合せを定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Batch mode		デフォルトでは、バッチ・モードは使用されません。`true`に設定すると、バッチ・モードが有効になります。一部のデータベースでのみサポートされます。詳細は、バッチ・モードおよびバッチ・サイズ(p.477)を参照してください。	false (デフォルト) | true
Advanced			
Batch size		1回のバッチ更新でデータベースに送信できるレコード数。詳細は、バッチ・モードおよびバッチ・サイズ(p.477)を参照してください。	25 (デフォルト) | 1-N

属性	必須	説明	可能な値
Commit		何件の(エラーなしの)レコードごとにコミットが実行されるかを定義します。MAX_INTに設定すると、コンポーネントではコミットは実行されません。つまり、グラフの解放中に接続がクローズされるまでは実行されません。「**Atomic SQL query**」が定義されている場合、この属性は無視されます。	100 (デフォルト) | 1-MAX_INT
Max error count		許可されるレコードの最大数。この数を超えるとグラフが失敗します。デフォルトでは、エラーは許可されません。-1に設定すると、すべてのエラーが許可されます。詳細は、エラー(p.478)を参照してください。	0 (デフォルト) | 1-N | -1
Action on error		デフォルトでは、エラーの数が「**Max error count**」を超えると、正しいレコードがデータベースにコミットされます。ROLLBACKに設定すると、現行バッチのコミットは実行されません。詳細は、エラー(p.478)を参照してください。	COMMIT (デフォルト) | ROLLBACK
Atomic SQL query		SQL問合せ実行のアトミック性を設定します。trueに設定すると、1つのレコードに対するすべてのSQL問合せがアトミック操作として実行されますが、「**Commit**」属性の値は無視され、レコードごとにコミットが実行されます。詳細は、Atomic SQL Query(p.478)を参照してください。	false (デフォルト) | true

[1)]これらの属性のいずれか1つを指定する必要があります。それより多く定義された場合、「**Query URL**」の優先度が最も高く、「**DB table**」の優先度が最も低くなります。詳細は、問合せまたはDB表が定義されている場合(p.475)を参照してください。

[2)]これらの関係の詳細は、DBフィールドへのCloverフィールドのマッピング(p.477)を参照してください。

詳細説明

問合せまたはDB表が定義されている場合

- **問合せが定義されている場合(「SQL Query」または「Query URL」)**
  - **問合せにCloverフィールドが含まれている場合**

    Cloverフィールドは、DB表の指定位置に挿入されます。

    これが、CloverおよびDBフィールドのマッピングを定義する最も簡単で明示的な方法です。その他の属性は定義できません。

    SQL問合せエディタ(p.478)も参照してください。
  - **問合せに疑問符が含まれる場合**

    疑問符は、次に示すいずれかの方法でCloverフィールド値のプレースホルダとして使用されます。詳細は、DBフィールドへのCloverフィールドのマッピング(p.477)を参照してください。

    SQL問合せエディタ(p.478)も参照してください。

例54.2. 問合せの例

文	形式
Derby、Infobright、Informix、MSSQL2008、MSSQL2000-2005、MySQL[1)]	
insert (Cloverフィールドを指定)	insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm) [returning $out1field1 := $in0field3[, $out1field2 := auto_generated][, $out1field3 := $in0field7]]

文	形式
insert (疑問符を指定)	insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?)[returning $out1field1 := $in0field3[, $out1field2 := auto_generated][, $out1field3 := $in0field7]]
DB2、Oracle[2)]	
insert (Cloverフィールドを指定)	insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm) [returning $out1field1 := dbf3[, $out1field3 := dbf7]]
insert (疑問符を指定)	insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?)[returning $out1field1 := dbf3[, $out1field3 := dbf7]]
PostgreSQL、SQLite、Sybase[3)]	
insert (Cloverフィールドを指定)	insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm)
insert (疑問符を指定)	insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?)
すべてのデータベース[4)]	
update	update mytable set dbf1 = $in0field1, ..., dbfn = $in0fieldn [returning $out1field1 := $in0field3[, $out1field2 := update_count][, $out1field3 := $in0field7]]
delete	delete from mytable where dbf1 = $in0field1 and ... and dbfj = ? and dbfn = $in0fieldn

説明:

1) これらのデータベースは`auto_generated`という名前の仮想フィールドを生成し、それを`insert`文で指定される出力メタデータ・フィールドの1つにマップします。

2) これらのデータベースは複数のデータベース・フィールドを返し、それらを`insert`文で指定される出力メタデータ・フィールドにマップします。

3) これらのデータベースは、`insert`文で何も返しません。

4) `update`文では、すべてのデータベースで、`update_count`仮想フィールドの値とともに任意の数の入力メタデータ・フィールドを出力メタデータ・フィールドにマップできます。

重要

デフォルトの(**汎用の**) JDBC定義では、自動生成キーはサポートされません。

- **DB表が定義されている場合**

  DBフィールドへのCloverフィールドのマッピングは、次に示すように定義されます。詳細は、DBフィールドへのCloverフィールドのマッピング(p.477)を参照してください。

DB表名のドル記号

- データベース表の名前に含まれるドル記号は、生成される問合せでは二重ドル記号に変換されます。したがって、各問合せではdb表にドル記号が偶数含まれている必要があります(隣接するドル記号のペアで構成されます)。db表の名前に含まれる一重のドル記号は、問合せのdb表の名前では、二重のドル記号で置換されます。

  `my$table$`という名前の表は、問合せでは`my$$table$$`に変換されます。

DBフィールドへのCloverフィールドのマッピング

- **フィールド・マッピングが定義されている場合**

  フィールド・マッピングが定義されている場合、この属性で指定された各Cloverフィールドの値が、**フィールド・マッピング**属性でこのCloverフィールドが割り当てられた名前を持つDBフィールドに挿入されます。

  フィールド・マッピングのパターン:

  ```
  $CloverFieldA:=DBFieldA;...;$CloverFieldM:=DBFieldM
  ```

- **CloverフィールドとDBフィールドの両方が定義されている場合**

  Cloverフィールドと**DBフィールド**の両方が定義されている(が**フィールド・マッピング**は定義されていない)場合、**Cloverフィールド**属性で指定された各Cloverフィールドの値が、**DBフィールド**属性の同じ位置にあるDBフィールドに挿入されます。

  これら両方の属性のCloverフィールドおよびDBフィールドの数は、互いに等しい必要があります。いずれの数も、他の方法(ドル記号の接頭辞を付けたCloverフィールドの指定、DB関数の指定または問合せでの定数の指定)では定義されていないDBフィールドの数に等しい必要があります。

  Cloverフィールドのパターン:

  ```
  CloverFieldA;...;CloverFieldM
  ```

  DBフィールドのパターン:

  ```
  DBFieldA;...;DBFieldM
  ```

- **Cloverフィールドのみが定義されている場合**

  Cloverフィールド属性のみが定義されている(が**フィールド・マッピング**または**DBフィールド**(あるいはその両方)は定義されていない)場合、Cloverフィールド属性で指定された各Cloverフィールドの値が、DB表での位置が等しいDBフィールドに挿入されます。

  Cloverフィールド属性で指定された**Cloverフィールド**の数は、他の方法(ドル記号の接頭辞を付けたCloverフィールドの指定、DB関数の指定または問合せでの定数の指定)では定義されていないDB表内のDBフィールドの数に等しい必要があります。

  Cloverフィールドのパターン:

  ```
  CloverFieldA;...;CloverFieldM
  ```

- **マッピングが自動的に実行される場合**

  フィールド・マッピング、**Cloverフィールド**または**DBフィールド**のいずれも定義されていない場合、マッピング全体が自動的に実行されます。メタデータの各Cloverフィールドの値が、DB表内の同じ位置に挿入されます。

  すべてのCloverフィールドの数は、他の方法(ドル記号の接頭辞を付けたCloverフィールドの指定、DB関数の指定または問合せでの定数の指定)では定義されていないDB表内のDBフィールドの数に等しい必要があります。

バッチ・モードおよびバッチ・サイズ

1. **Batch Mode**

   「Batch mode」により、データベースへのデータのロードが高速化されます。

2. **Batch Size**

   データベースによっては、拒否されたものとして、実際の数より多くのレコードが返されます。これらのデータベースでは、データベースに正常にロードされたレコードも返し、それらを出力ポート0経由で送信します(接続されている場合)。

エラー

1. **Max error count**

   許容されるエラーの最大数を指定します。指定した数を超えると、グラフの実行が停止します。その後、定義された「**Action on Error**」が実行されます。

2. **Action on Error**

   `COMMIT`

   デフォルトでは、エラーの最大数を超えた場合に、一部のデータベースでのみ正しいレコードに対するコミットが実行されます。その他のデータベースでは、かわりにロールバックが実行されます。その後、グラフが停止します。

   `ROLLBACK`

   一方、ロールバックは、エラーの最大数を超えた場合にすべてのデータベースで実行されますが、その対象は最後のコミットされていないレコードのみとなります。その後、グラフが停止します。コミットされたレコードはいずれもロールバックできません。

Atomic SQL Query

- 「**Atomic SQL query**」は、単一のレコードに関する複数の副問合せで構成される問合せの処理方法を指定します。

  デフォルトでは、各副問合せは個別に考慮され、それらの一部が失敗すると、それ以前の副問合せはデータベースに応じてコミットされるかロールバックされます。

  「**Atomic SQL query**」属性が`true`に設定されている場合、コミットまたはロールバックについて、すべての副問合せで実行されるか、またはどの副問合せでも実行されません。これにより、すべてのデータベースが同じ動作をすることが保証されます。

重要

また、MS SQL Serverに接続する場合は、jTDS (http://jtds.sourceforge.net)ドライバを使用すると便利です。これは、Microsoft SQL ServerおよびSybase向けのオープン・ソースの100% Pure Java JDBCドライバです。Microsoftのドライバよりも高速です。

SQL問合せエディタ

SQL問合せ属性を定義するには、**SQL問合せエディタ**を使用できます。

このエディタは、「**SQL query**」属性行をクリックすると開きます。

左側には、これらの列のスキーマ、表、列およびデータ型に関する情報を含む「**Database schema**」ペインがあります。

表示されたスキーマ、表および列は、「**ALL**」コンボや「**Filter in view**」テキスト領域の値、「**Filter**」ボタンおよび「**Reset**」ボタンなどを使用してフィルタできます。

列を選択するには、スキーマ、表を展開して、目的の列で**[Ctrl]**を押しながら**クリック**します。

隣接する複数の列を選択するには、**[Shift]**を押しながら最初の項目と最後の項目を**クリック**します。

コンボから、「**insert**」、「**update**」、「**delete**」のいずれかの文を選択します。

次に、「**Generate**」をクリックする必要があります。これにより、問合せが「**Query**」ペインに表示されます。

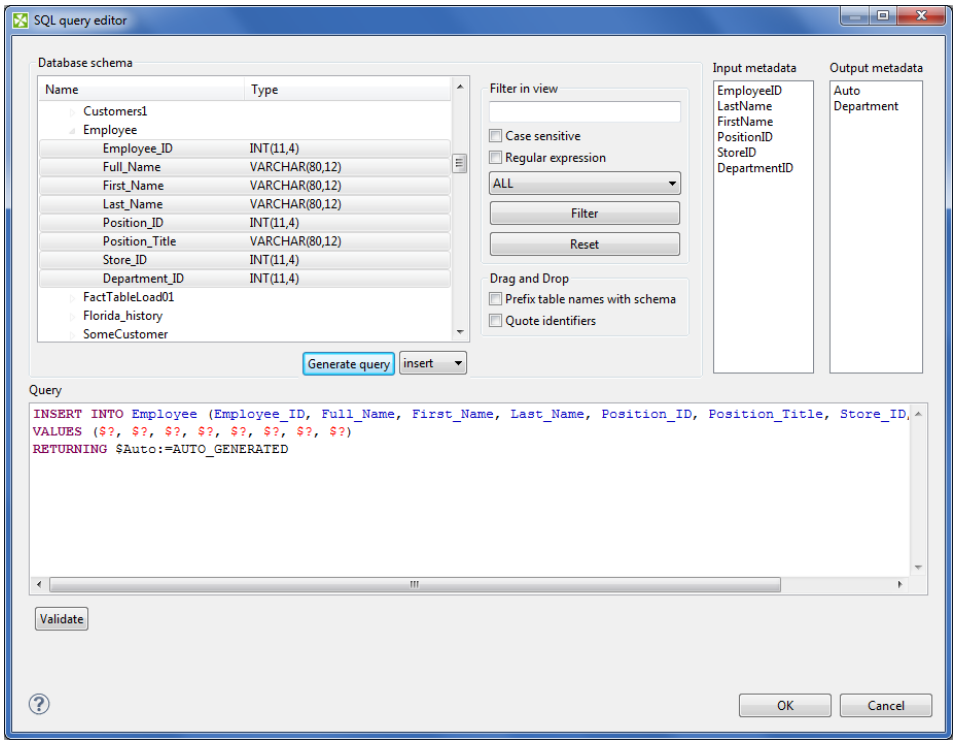


図54.1. 疑問符を含む生成済問合せ

入力メタデータ・フィールドと異なるdb列があると、問合せに疑問符が含まれることがあります。入力メタデータは、右側の「**Input metadata**」ペインに表示されます。

「**Input metadata**」ペインから「**Query**」ペインの対応する場所にフィールドをドラッグ・アンド・ドロップして、手動で"$?"という文字を削除します。次の図を参照してください。

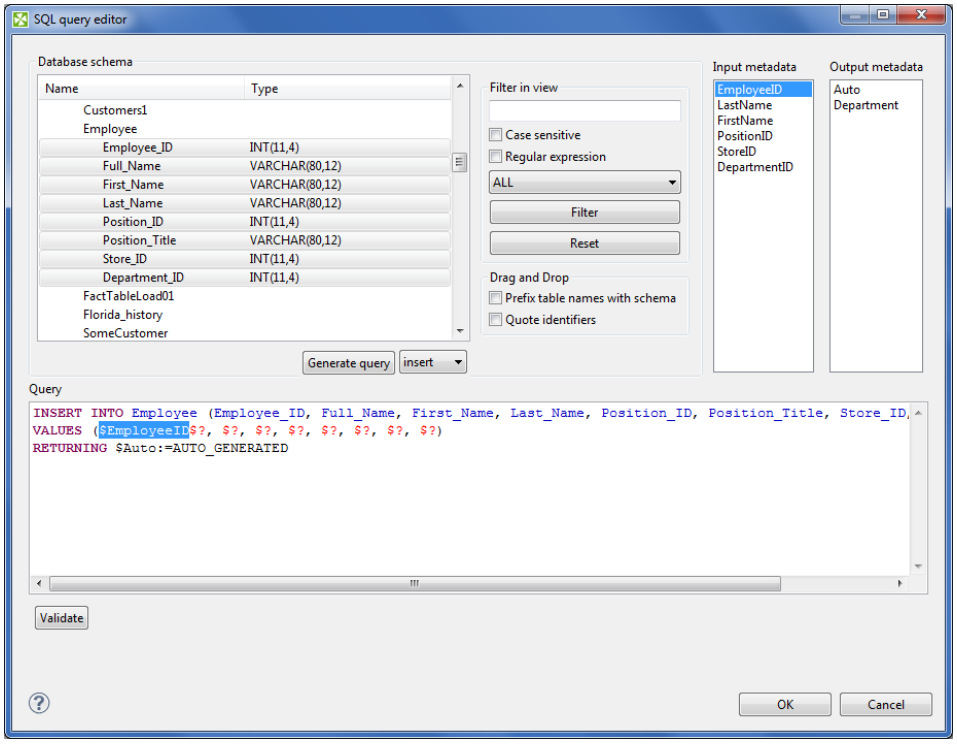


図54.2. 入力フィールドを含む生成済問合せ

2番目の出力ポートに接続されたエッジがある場合、自動生成された列および返されたフィールドを返すことができます。

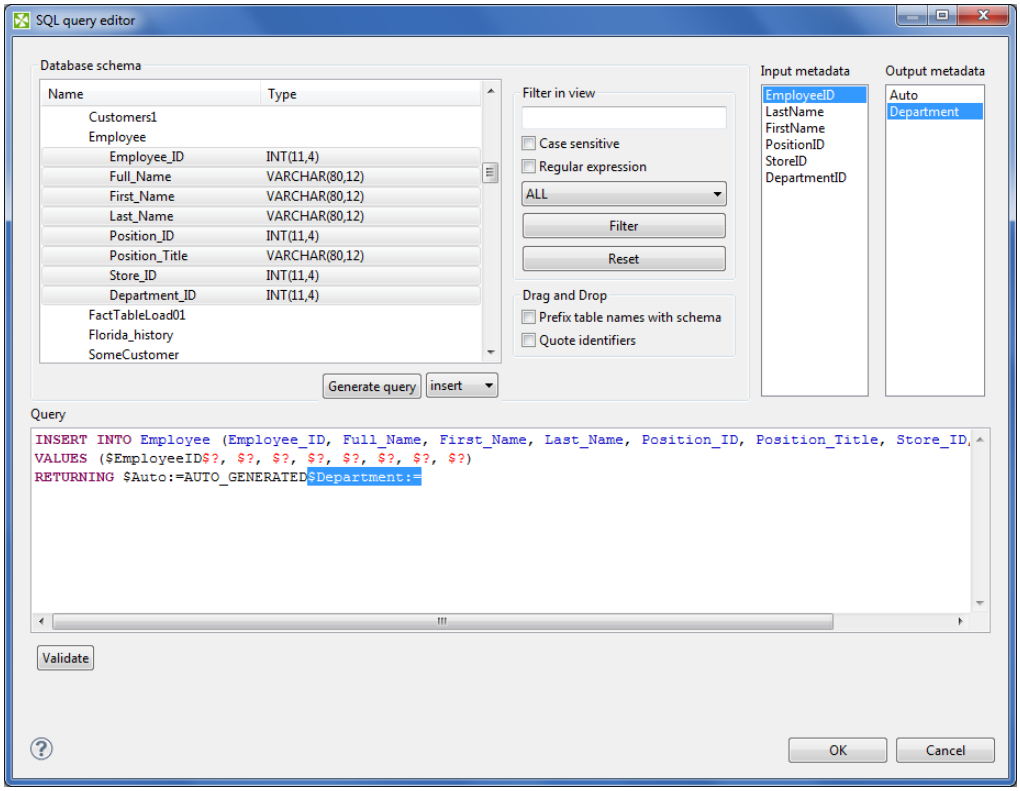


図54.3. 返されたフィールドを含む生成済問合せ

2つのボタンを使用して、問合せを検証(「**Validate**」)したり、表にデータを表示(「**View**」)できます。

EmailSender

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

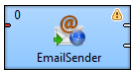
EmailSenderは電子メールを送信します。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
EmailSender	フラット・ファイル	1	0-1	いいえ	いいえ	いいえ	いいえ

概要

EmailSenderは、データ・レコードを電子メールに変換します。入力データを使用して、電子メールの差出人と受取人、電子メールの件名、メッセージ・ボディおよび添付ファイルを作成できます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	電子メールのデータ用。	任意
出力	0	いいえ	送信に成功した電子メール用。	入力0
	1	いいえ	拒否された電子メール用。	入力0および`errorMessage`という名前のフィールド[1)]

説明:

1): レコードが拒否され電子メールが送信されない場合、エラー・メッセージが作成されて出力1のメタデータの`errorMessage`フィールドに送信されます(出力1にこのフィールドが含まれている場合)。

EmailSenderの属性

属性	必須	説明	可能な値
Basic			
SMTP server	はい	送信電子メール用のSMTPサーバーの名前。	
SMTP user		認証済SMTPサーバーのユーザーの名前。	
SMTP password		認証済SMTPサーバーのユーザーのパスワード。	
Use TLS		デフォルトでは、TLSは使用されません。`true`に設定すると、TLSが有効になります。	false (デフォルト) | true
Use SSL		デフォルトでは、SSLは使用されません。`true`に設定すると、SSLが有効になります。	false (デフォルト) | true
Message	はい	メッセージ・ヘッダーおよびメッセージ本文を定義する一連のプロパティ。詳細は、電子メール・メッセージ(p.482)を参照してください。	
Attachments		メッセージの添付ファイルを定義する一連のプロパティ。詳細は、電子メールの添付ファイル(p.483)を参照してください。	
Advanced			
SMTP port		SMTPサーバーへの接続に使用されるポートの番号。	25 (デフォルト) | その他のポート
Trust invalid SMTP server certificate		デフォルトでは、無効なSMTPサーバー証明書は受け入れられません。`true`に設定すると、無効なSMTPサーバー証明書(名前が異なる、有効期限が切れているなど)が受け入れられます。	false (デフォルト) | true
Ignore send fail		デフォルトでは、電子メールが正常に送信されない場合にグラフが失敗します。`true`に設定すると、正常に送信されたメールがない場合でも、グラフの実行が停止します。	false (デフォルト) | true

詳細説明

- **電子メール・メッセージ**

  「**Message**」属性を定義するには、次のウィザードを使用します。

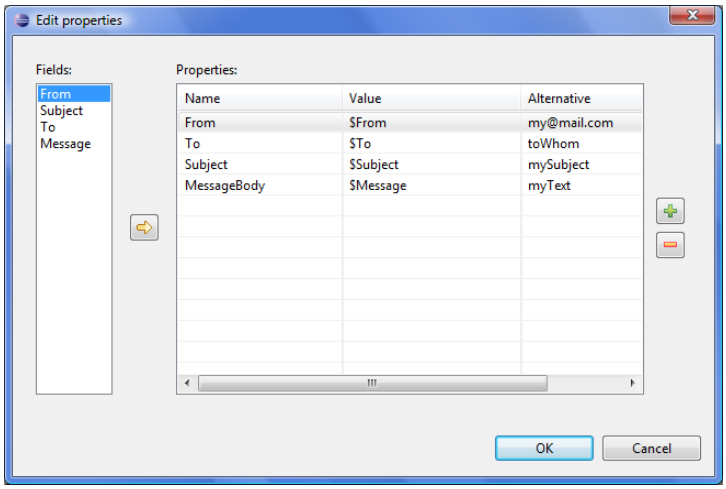


図54.4. EmailSenderメッセージ・ウィザード

このウィザードでは、左側の「**Fields**」ペインから右側の「**Properties**」ペインの「**Value**」列にフィールドをコピーする必要があります。**右矢印**ボタンを使用するか、選択したフィールドを「**Value**」列にドラッグ・アンド・ドロップします。また、これらの属性の代替値を指定することもできます(「**Alternative**」列)。空であるかNULL値を持つフィールドがあった場合、フィールド値のかわりにその**代替値**が使用されます。

生成される「**Message**」属性の値は次のようになります。

```
From=$From|my@mail.com;Subject=$Subject|mySubject;To=$To|toWhom;MessageBody=$Message|myText
```

ヒント

複数の受信者に電子メールを送信する場合、受信者のアドレスをカンマ(,)で区切ります。必要に応じて、CcおよびBccフィールドで同じデリミタを使用します。

- **電子メールの添付ファイル**

設定可能な属性の1つに「**Attachments**」があります。これは、個別の添付ファイルをセミコロンで区切ったシーケンスとして指定できます。各添付ファイルはパスを含むファイル名にするか、この(パスを含む)ファイル名をいくつかの入力フィールドの値を使用して指定することもできます。各添付ファイルを、フィールド名、添付ファイルの名前およびそのMIMEタイプの3つの組合せとして指定することもできます。これらは明示的に指定する(`[$fieldName, FileName, MimeType]`)ことも、フィールド値`[$fieldNameWithFileContents, $fieldWithFileName, $fieldWithMimeType]`を使用して指定することもできます。この3つの組合せの各部分は、静的な式を使用して指定することもできます。添付ファイルは、次の「**Edit attachments**」ウィザードを使用して電子メールに追加する必要があります。

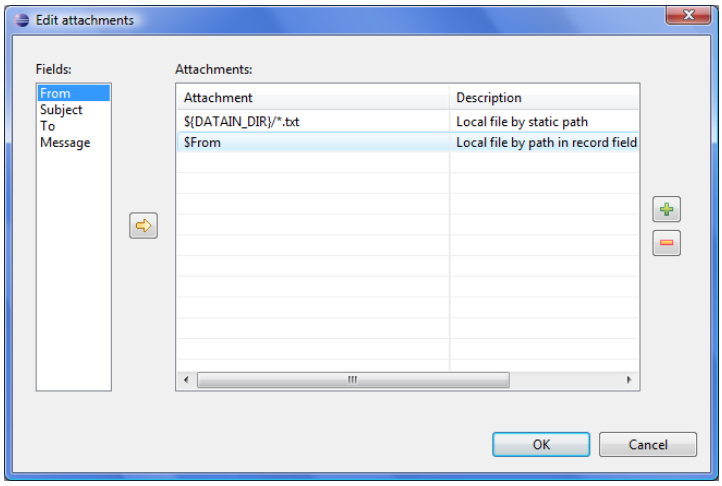


図54.5. 「Edit Attachments」ウィザード

項目を追加するには**プラス記号**ボタンをクリックし、削除するには**マイナス記号**ボタンをクリックします。入力フィールドは「**Attachments**」ペインの「**Attachment**」列にドラッグするか、矢印ボタンを使用して移動します。添付ファイル定義を編集する場合は、「**Attachment**」列で該当する行をクリックすると、次の属性が開きます。

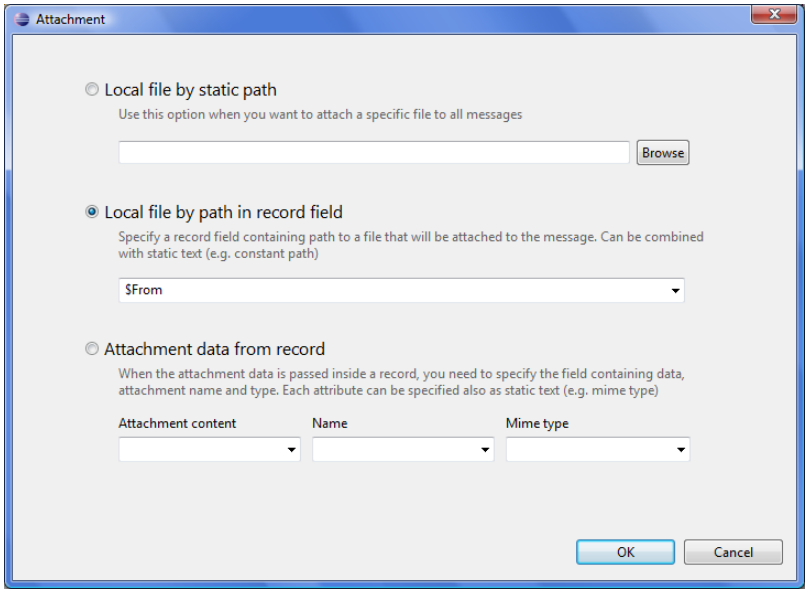


図54.6.「Attachment」ウィザード

このウィザードでは、ファイルを検索し、フィールド名または前述の3つの組合せを使用してファイルを指定する必要があります。**「OK」**をクリックすると、添付ファイルが定義されます。

HadoopWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、第54章「ライター」(p.459)を参照してください。

要約

HadoopWriterは、Hadoopシーケンス・ファイルにデータを書き込みます。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
HadoopWriter	Hadoopシーケンス・ファイル	1	0	✖	✖	✖	✖

概要

HadoopWriterは、特別なHadoopシーケンス・ファイル(`org.apache.hadoop.io.SequenceFile`)にデータを書き込みます。これらのファイルは、キーと値のペアを含んでおり、MapReduceジョブで入力/出力ファイル形式として使用されます。このコンポーネントは、HDFSまたはローカル・ファイル・システム上に存在する必要がある単一のファイルおよびパーティション化されたファイルを書き込むことができます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	入力データ・レコード用	任意

HadoopWriterの属性

属性	必須	説明	可能な値
Basic			
Hadoop connection		Hadoopシーケンス・ファイル・ライター実装を含むHadoopライブラリとのHadoop接続(p.192)。**「File URL」**属性の`hdfs://`URLでHadoop接続IDが指定されている場合、この属性の値は無視されます。	Hadoop接続ID
File URL	✔	HDFSまたはローカル・ファイル・システム上の出力ファイルのURL。 プロトコルを含まないURL (実際には絶対パスまたは相対パス)、または`file://`プロトコルを含むURLは、ローカル・ファイル・システムに配置されているとみなされます。 出力ファイルがHDFS上に配置されている必要がある場合は、`hdfs://ConnID/path/to/file`という形式のURLを使用してください。ここで、`ConnID`はHadoop接続(p.192)のIDであり(**「Hadoop connection」**コンポーネント属性は無視されます)、`/path/to/myfile`は、対応するHDFS上の`myfile`という名前のファイルの絶対パスです。	
Key field	✔	書き込まれた各キー値ペアのキーが含まれる入力レコード・フィールドの名前。	
Value field	✔	書き込まれた各キー値ペアの値が含まれる入力レコード・フィールドの名前。	

詳細説明

HadoopWriterコンポーネントにより作成されるファイル形式の正確なバージョンは、**「File URL」**属性から参照される**「Hadoop connection」**で指定したHadoopライブラリによって異なります。一般に、あるHadoopバージョンで作成されたシーケンス・ファイルを他のバージョンで読み取ることはできません。

現時点では、圧縮データの書込みはサポートされていません。

ローカル・ファイル・システムに書き込む場合、デフォルト設定の「Hadoop connection」を使用しないと、追加の`.crc`ファイルが作成されます。これは、Hadoopがローカル・ファイル・システムと対話する際に、デフォルトでは、書き込まれた各ファイルのチェックサム・ファイルを作成する`org.apache.hadoop.fs.LocalFileSystem`を使用するためです。そのようなファイルを読み取る場合、チェックサムが検証されます。Hadoop接続(p.192)の**Hadoopパラメータ**に`fs.file.impl=org.apache.hadoop.fs.RawLocalFileSystem`というキー値ペアを追加すると、チェックサムの作成および検証を無効にできます。

Hadoopシーケンス・ファイルの技術的な詳細は、Apache Hadoop Wikiを参照してください。

InfobrightDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

InfobrightDataWriterは、Infobrightデータベースにデータをロードします。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
InfobrightDataWriter	データベース	1	0-1	いいえ	いいえ	いいえ	いいえ

概要

InfobrightDataWriterは、Infobrightデータベースにデータをロードします。rootユーザーのみがこのコンポーネントを使用してデータベースにデータを挿入できます。Windowsでこのコンポーネントを実行するには、Javaライブラリ・パスに`infobright_jni.dll`が存在する必要があります。(www.infobright.org/downloads/contributions/infobright-core-2_7.zipでダウンロードできます。)

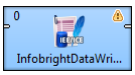
ホスト名が`localhost`または`127.0.0.1`の場合、ロードはローカル・パイプを使用して実行されます。これ以外の場合はリモート・パイプを使用します。サーバーの外部IPアドレスは、ローカル・サーバーとして認識されません。

リモート・サーバーにロードするには、Infobrightが実行されているサーバーでInfobrightリモート・ロード・エージェントを起動する必要があります。これには`java -jar infobright-core-3.0-remote.jar [-p PortNumber] [-l all | debug | error | info]`コマンドを実行する必要があります。出力はログ・ファイルにリダイレクトできます。デフォルトでは、サーバーはポート5555でリスニングします。`infobright-core-3.0-remote.jar`は**CloverETL**に付属しているか、Infobrightサイト(www.infobright.org)でダウンロードできます。

デフォルトでは、`root`には、`localhost`からの接続のみが許可されます。その他のホストから接続するには、別のユーザー`root@%`を追加する必要があります。データのロード用に別のユーザー(`root`以外)を作成することをお薦めします。データをロードまたはコネクタを使用できるようにするために、このユーザーには`FILE`権限が必要です。

```
grant FILE on *.* to 'user'@'%';
```

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	データベースにロードされるレコード	任意
出力	0	いいえ	データベースにロードされる際のレコード用	入力0のメタデータの対応する部分[1)]

説明:

1): マップ済のCloverフィールド値のみをオプションの出力ポート経由で送信できます。カンマを各フィールドのデリミタとして設定する必要があり、`System.getProperty("line.separator")` (UNIXの場合は"\n"、Windowsの場合は"\r\n")をレコード・デリミタとして設定する必要があります。日付フィールドでは、日付書式には`yyyy-MM-dd`を使用し、時刻が含まれる日付書式には`yyyy-MM-dd HH:mm:ss`を使用する必要があります。

InfobrightDataWriterの属性

属性	必須	説明	可能な値
Basic			
DB connection	はい	データベースにアクセスするためのDB接続オブジェクトのID。	
Database table	はい	データがロードされるDB表の名前。	
Charset		VAR、VARCHARの各列タイプへの文字列値のエンコーディングに使用されるキャラクタ・セット。	ISO-8859-1 (デフォルト) | その他のエンコーディング
Data format		Infobrightによりサポートされる`bh_dataformat`。オプションは、`txt_variable`または`binary`です(binaryの方が高速ですが、IEEでのみ機能します)。	Text (デフォルト) | Binary
Advanced			
Clover fields		セミコロンで区切られたCloverフィールドのシーケンス。この属性にリストされているCloverフィールドのみがデータベース列にロードされます。Cloverフィールドおよびデータベース列の位置はどちらも同じになります。これらの数はデータベース列の数と等しい必要があります。	
Log file		パスを含む、データベースにロードされるレコードのファイル。この属性を指定すると、出力ポートが接続されている場合でも、出力ポートにデータが送信されません。	
Append data to log file		デフォルトでは、新しいレコードで古いレコードが上書きされます。`true`に設定すると、新しいレコードが古いレコードに追加されます。	false (デフォルト) | true
Execution timeout		loadコマンドのタイムアウト(秒単位)。Windowsプラットフォームでのみ機能します。	15 (デフォルト) | 1-N
Check string's and binary's sizes		デフォルトでは、データベースにデータが渡される前にサイズは確認されません。`true`に設定するとサイズが確認されます。デバッグがサポートされている場合は、`true`に設定する必要があります。	false (デフォルト) | true
Remote agent port		サーバーへの接続時に使用するポート。	5555 (デフォルト) | その他のポート番号

InformixDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

InformixDataWriterは、Informixデータベースにデータをロードします。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
InformixDataWriter	データベース	0-1	0-1	いいえ	いいえ	いいえ	いいえ

概要

InformixDataWriterは、Informixデータベース・クライアント(`dbload`ユーティリティ)または`load2`無料ライブラリを使用して、データベースにデータをロードします。

データベースがあるサーバーを`dbload`データベース・ユーティリティおよび**CloverETL**の両方と同じコンピュータ上に配置することが重要であり、rootユーザーとしてログインする必要があります。Informixサーバーは、**Clover**が実行されるマシンと同じマシンにインストールおよび構成する必要があり、ユーザーはrootとしてログインする必要があります。Dbloadコマンドライン・ツールも使用可能である必要があります。

InformixDataWriterは、入力ポートまたはファイルからデータを読み取ります。入力ポートが他のコンポーネントに接続されていない場合、コンポーネントで指定される別のファイルにデータが含まれている必要があります。

オプションの出力ポートにコンポーネントを接続すると、拒否されたレコードがエラーに関する情報とともにそのポートに送信されます。

`dbload`ユーティリティのかわりの別のツールは`load2`無料ライブラリです。`load2`無料ライブラリは、サーバーがリモート・コンピュータに配置されている場合でも使用できます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	1)	データベースにロードされるレコード	任意
出力	0	いいえ	正しくないレコードに関する情報	入力0 (さらにInformixDataWriterのエラー・フィールド(p.490)の2つ[2)])

説明:

1): ロードするデータを含むファイル(**Loader input file**)が指定されていない場合は、入力ポートが接続されている必要があります。

2): 出力ポート0のメタデータには、末尾に`number of row`および`error message`の2つの追加フィールドが含まれています。

表54.2. InformixDataWriterのエラー・フィールド

フィールド番号	フィールド名	データ型	説明
LastInputField + 1	<anyname1>	integer	行数
LastInputField + 2	<anyname2>	string	エラー・メッセージ

InformixDataWriterの属性

属性	必須	説明	可能な値
Basic			
Path to dbload utility	はい	パスを含む、dbloadユーティリティの名前。Informixサーバーは、Cloverが実行されるマシンと同じマシンにインストールおよび構成する必要があり、ユーザーはrootとしてログインする必要があります。dbloadコマンドライン・ツールも使用可能である必要があります。	
Host		データベース・サーバーがあるホスト。	
Database	はい	レコードがロードされるデータベースの名前。	
Database table	はい	レコードがロードされるデータベース表の名前。	
Advanced			
Control script		dbloadユーティリティにより使用される制御スクリプト。これが設定されない場合、かわりにデフォルトの制御スクリプトが使用されます。これは、「**Use load utility**」属性が`false`に設定されている場合にのみ使用されます。	
Error log URL		パスを含む、エラー・ログ・ファイルの名前。設定されない場合、かわりにデフォルトのエラー・ログ・ファイルが使用されます。	./error.log
Max error count		許可されるレコードの最大数。この数を超えるとグラフが失敗します。	10 (デフォルト) | 0-N
Ignore rows		スキップする行数。これは、「**Use load utility**」属性が`true`に設定されている場合にのみ使用されます。	0 (デフォルト) | 1-N
Commit interval		行数で示されるコミット間隔。	100 (デフォルト) | 1-N

属性	必須	説明	可能な値
Column delimiter		データの各列で使用される1文字のデリミタ。フィールド値にこのデリミタを含めることはできません。これは、**「Use load utility」**属性が`false`に設定されている場合にのみ使用されます。	"|" (デフォルト) | その他の文字
Loader input file		パスを含む、ロードする入力ファイルの名前。名前付きパイプがかわりに使用されないかぎり、通常、このファイルがdbloadユーティリティに渡されるデータの一時記憶域となります。	
Use load utility		デフォルトでは、データベースにデータをロードするためにはdbloadユーティリティが使用されます。`true`に設定すると、dbloadのかわりにload2ユーティリティが使用されます。load2ユーティリティが使用可能である必要があります。	false (デフォルト) | true
User name		データベースへの接続時に使用するユーザー名。これは、**「Use load utility」**属性が`true`に設定されている場合にのみ使用されます。	
Password		データベースへの接続時に使用するパスワード。これは、**「Use load utility」**属性が`true`に設定されている場合にのみ使用されます。	
Ignore unique key violation		デフォルトでは、一意キーの制約違反は無視されません。キー値が一意でない場合、グラフが失敗します。`true`に設定すると、一意キーの制約違反が無視されます。これは、**「Use load utility」**属性が`true`に設定されている場合にのみ使用されます。	false (デフォルト) | true
Use insert cursor		デフォルトでは、挿入カーソルが使用されます。挿入カーソルを使用すると、転送パフォーマンスが2倍になります。これは、**「Use load utility」**属性が`true`に設定されている場合にのみ使用されます。`false`に設定すると、これを無効にできます。	true (デフォルト) | false

詳細説明

Loader input file

パスを含む、ロードする入力ファイルの名前。名前付きパイプがかわりに使用されないかぎり、通常、このファイルがdbloadユーティリティに渡されるデータの一時記憶域となります。

- これが設定されない場合、ローダー・ファイルはCloverまたはOSの一時ディレクトリに作成されます。ロードが終了するとファイルは削除されます。
- これが設定されている場合、指定したファイルが作成されます。これはデータがロードされた後も削除されず、グラフが実行されるたびに上書きされます。
- 入力ポートが接続されていない場合、このファイルが存在し、指定され、データベースにロードされるデータを含んでいる必要があります。これは削除も上書きもされません。

JavaBeanWriter

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

JavaBeanWriterは、階層構造をJavaBeansとしてディクショナリに書き込みます。書込みでは複数のクラスがサポートされています。これにより、Cloverグラフとクラウドなどの外部環境との間で動的なデータ交換が可能になります。どのJavaBeanを選択するかによって、出力がある程度定義されます。これが、入力を事前設定済だがカスタマイズ可能なツリー構造にマップする理由です。また、Javaコレクション(リスト、マップ)にデータを書き込むこともできます。書込みの際、**JavaBeanWriter**はBeanのクラスパスを調べて書き込むデータ型を決定します。つまり、メタデータ・フィールドの型とJavaBeansの型の間で型変換を実行します。変換が失敗した場合は、書込み時にエラーが発生します。

データ型に関する制約が少なく、外部クラスパスを必要としない、より柔軟なコンポーネントが必要な場合は、**JavaMapWriter**を選択してください。

コンポーネント	データ出力	入力ポート	出力ポート	全出力に送信	各出力に送信	変換	変換が必要	Java	CTL
JavaBeanWriter	ディクショナリ	1-n	0	はい	いいえ	いいえ	いいえ	いいえ	いいえ

概要

JavaBeanWriterは、接続されているすべての入力ポートからデータを受信し、ユーザーが定義したマッピングに基づいてCloverレコードをJavaBeanプロパティに変換します。最後に、生成されたツリー構造がディクショナリ(p.228)に書き込まれます(これが唯一可能な出力です)。このコンポーネントはファイルには書き込むことができません。

マッピングのロジックはXMLWriter(p.560)と似ています。XMLWriterのマッピング・エディタの使用に慣れている場合は、このコンポーネントで問題なく出力ツリーを設計できます。違いは次のとおりです。

- 入力を出力に自由にマップできません。マッピング・エディタに表示されるツリー構造の設計は、使用するJavaBeanによって決まります。
- **JavaBeanWriter**を使用すると、Bean、そのプロパティまたはコレクション(リスト、マップ)にマップできます。
- XMLの場合のような属性、ワイルドカード属性およびワイルドカード要素はありません。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0-N	1つ以上	結合されJavaBeansにマップされる入力レコード。	任意(ポートごとに異なるメタデータを保持できます)

JavaBeanWriterの属性

属性	必須	説明	可能な値
Basic			
Dictionary target	はい	JavaBeansの書込み先となるディクショナリ。	以前に定義したディクショナリの名前。
Bean structure		「...」ボタンをクリックして、カスタム・クラス、オブジェクト、コレクションまたはマップで構成される出力JavaBeanの構造を設計します。	Bean構造の定義 (p.493)を参照してください。
Mapping	1)	入力データを出力JavaBeansにどのようにマップするかを定義します。	マッピング・エディタ (p.494)を参照してください。
Mapping URL	1)	マッピング定義が含まれる外部テキスト・ファイル。	

説明:

1) これらのうちどちらかを指定する必要があります。両方が指定された場合は、「**Mapping URL**」の優先度が高くなります。

詳細説明

Bean構造の定義

マッピングを開始する前に、出力JavaBeanのコンテンツを定義する必要があります。まず、「**Bean structure**」属性を編集します。これにより次のダイアログが開きます。

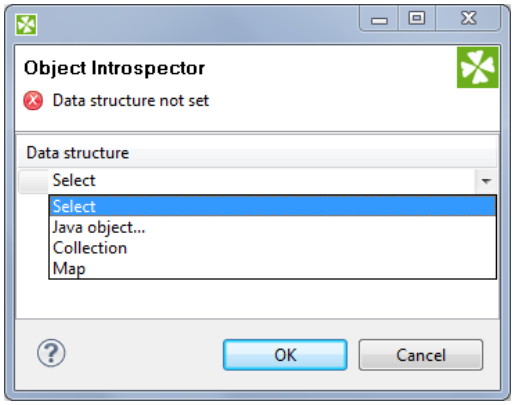


図54.7. Bean構造の定義: 「Select」コンボ・ボックスをクリックして開始

- **Java object**: クリックすると、Javaクラスを選択するダイアログが開きます。**重要:** カスタムJavaBeansクラスを使用する場合は、そのクラスを`trans`フォルダに配置してください。これにより、このダイアログでそのクラスを選択できるようになります。
- **Collection**: その他のオブジェクト、マップまたはその他のコレクションで構成されるリストを追加します。
- **Map**: キーと値のマップを追加します。

マッピング・エディタ

Bean構造を定義したら、入力レコードを出力JavaBeansにマップします。これは、XMLWriter(p.560)ですでに学習した方法と非常によく似た方法で実行します。いずれのコンポーネントのマッピング・エディタも類似のロジックを使用しています。

マッピングの基本は次のとおりです。

- コンポーネントの「**Mapping**」属性を編集します。これにより視覚的なマッピング・エディタが開きます。

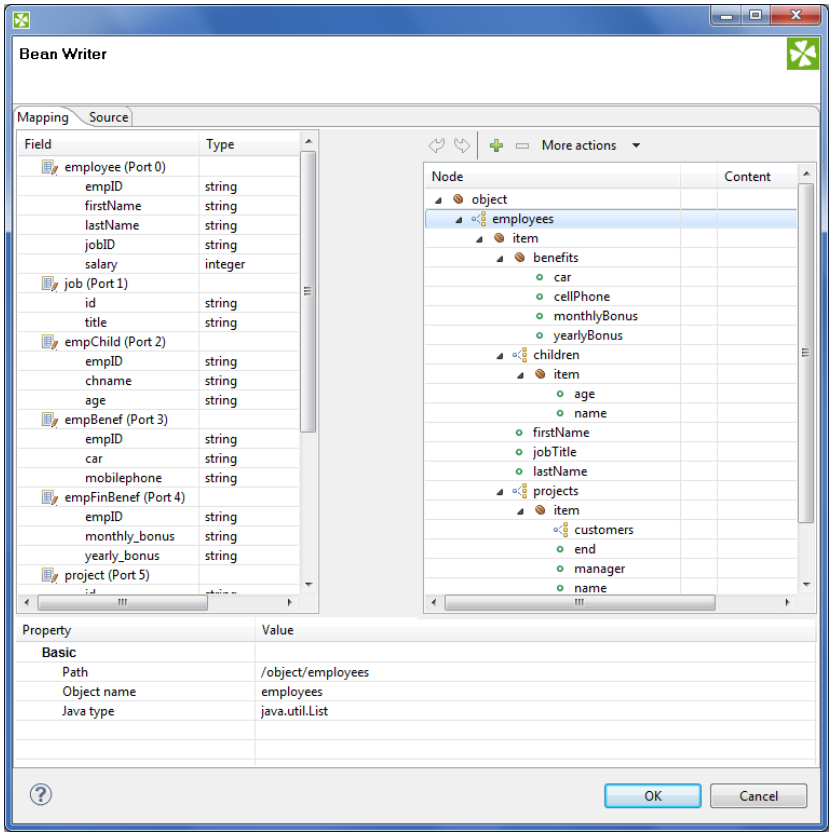


図54.8. 最初に開いた際のJavaBeanWriterのマッピング・エディタ。入力エッジのメタデータが左側に表示されます。右側のペインでは必要な出力ツリーを設計します。これはBeanの構造により事前定義されています(注意: この例では、Beanには従業員とその担当プロジェクトが含まれています)。マッピングを実行するには、メタデータを左から右にドラッグします(また、次に説明する追加タスクを実行します)。

- 右側のペインでは、入力メタデータを次のものにマップできます。
  - Beans
  - Beanプロパティ
  - リスト
  - マップ

緑色の+記号をクリックして**エントリを追加**します。これにより、ツリーに新規項目が追加されます。その型はコンテキスト(選択したノード)によって異なります。出力構造はBean構造(p.493)によって決まるため、このボタンは常に使用できるわけではありません。

- 入力レコードを出力ノードに接続して、バインディング(p.569)を作成します。

例54.3. バインディングの作成

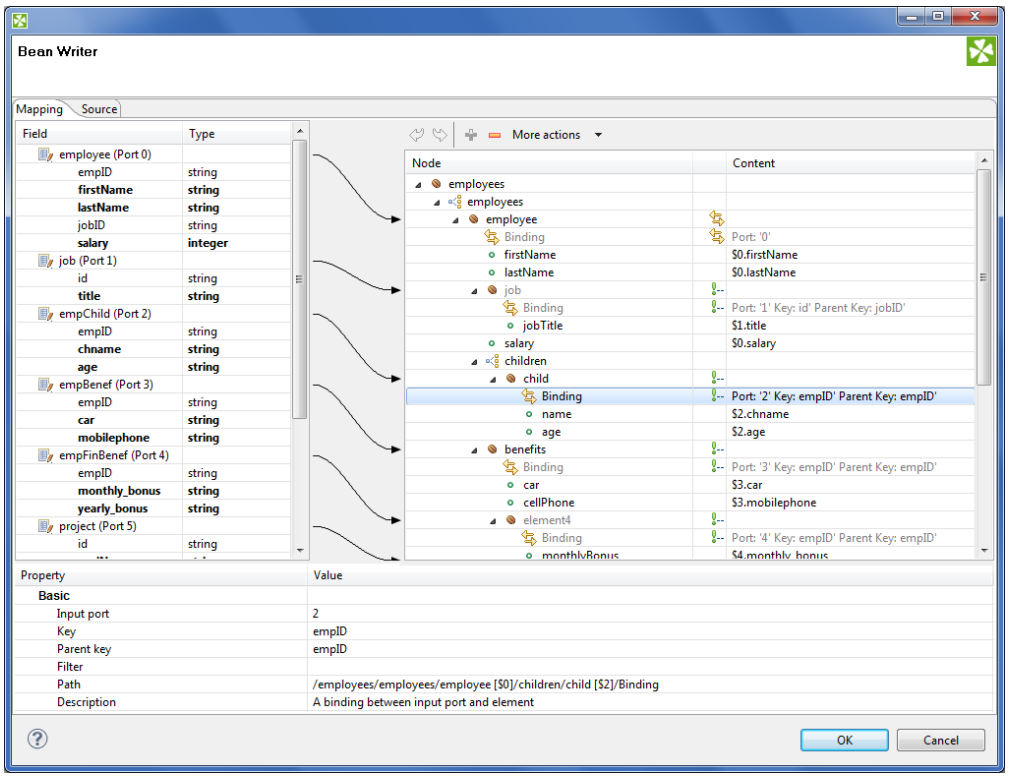


図54.9. JavaBeanWriterでのマッピングの例: 従業員はそれぞれの担当プロジェクトに結合されます。太字で示すフィールド(そのコンテンツ)は出力ディクショナリに出力され、マッピングで使用されます。

- いつでも「Source」タブ(p.572)に切り替えて、コードで独自のマッピングを記述して確認できます。
- ここに示されている基本的な説明では不十分な場合は、マッピング・プロセス全体について詳細に説明している、XMLWriterの詳細説明(p.560)を参照してください。

JavaMapWriter

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

JavaMapWriterは、JavaBeans (`HashMaps`として表されます)をディクショナリに書き込みます。これにより、Cloverグラフとクラウドなどの外部環境との間で動的なデータ交換が可能になります。このコンポーネントは、より容易なマッピングを可能にする**JavaBeanWriter**の特別な実装です。マップはBeanより制約が少なく、データ型および型変換がありません。このため**JavaMapWriter**の柔軟性が高まり、常にメタデータで定義されたものと同じデータ型をマップに書き込みます。オーバーヘッドが小さくなりますが、これには、目的のデータ型とは異なるデータ型を誤って読み取るという可能性が伴います(たとえば、`string`をマップに書き込み、それを**JavaBeanReader**で`integer`として読み込んだ場合は、グラフが失敗します)。

コンポーネント	データ出力	入力ポート	出力ポート	全出力に送信	各出力に送信	変換	変換が必要	Java	CTL
JavaMapWriter	ディクショナリ	1-n	0	はい	いいえ	いいえ	いいえ	いいえ	いいえ

概要

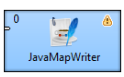
JavaMapWriterコンポーネントは、接続されているすべての入力ポートからデータを受信し、ユーザーが定義したマッピングに基づいてデータ・レコードをJava `HashMaps`に変換します。最後に、コンポーネントは生成された要素のツリー構造をマップに書き込みます。

マッピングのロジックはXMLWriter(p.560)と似ています。XMLWriterのマッピング・エディタの使用に慣れている場合は、このコンポーネントで問題なく出力ツリーを設計できます。違いは次のとおりです。

- **JavaMapWriter**では配列をマップできます。
- XMLにあるような属性はありません。

このコンポーネントはファイルに書き込むことができません。ディクショナリ(p.228)が唯一可能な出力です。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0-N	1つ以上	結合されJavaマップにマップされる入力レコード。	任意(ポートごとに異なるメタデータを保持できます)。

JavaMapWriterの属性

属性	必須	説明	可能な値
Basic			
Dictionary target	はい	Javaマップの書込み先となるディクショナリ。	
Mapping	1)	入力データを出力Javaマップにどのようにマップするかを定義します。詳細説明(p.498)を参照してください。	
Mapping URL	1)	マッピング定義が含まれる外部テキスト・ファイル。	

説明:

1) これらのうちどちらかを指定する必要があります。両方が指定された場合は、「**Mapping URL**」の優先度が高くなります。

詳細説明

XMLWriter(p.560)ですでに学習した方法と非常によく似た方法で、入力レコードを出力ディクショナリにマップします。いずれのコンポーネントのマッピング・エディタも類似のロジックを使用しています。マッピングの基本は次のとおりです。

- 入力エッジを**JavaMapWriter**に接続し、コンポーネントの「**Mapping**」属性を編集します。これにより視覚的なマッピング・エディタが開きます。

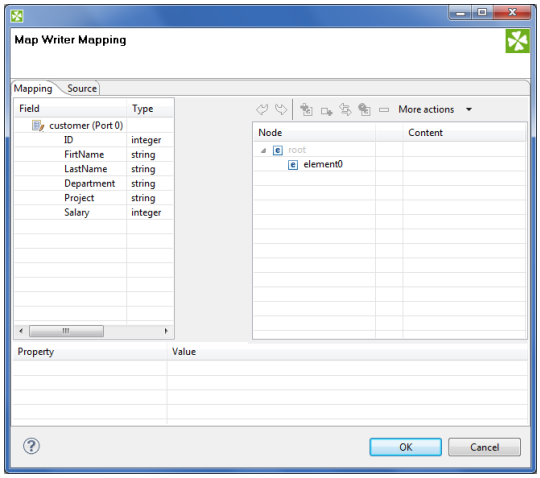


図54.10. 最初に開いた際のJavaMapWriterのマッピング・エディタ。入力エッジのメタデータが左側に表示されます。右側のペインでは、目的の出力ツリーを設計します。マッピングを実行するには、メタデータを左から右にドラッグします(また、次に説明する追加タスクを実行します)。

- 右側のペインでは、次のもので構成される出力ツリー構造を設計します。
  - 要素(p.564)

重要

XMLWriterとは異なり、メタデータを属性にマップしません。

  - **配列**: 配列は、順序付けられた値のセットです。これらのマップ方法は、配列の書込み(p.499)を参照してください。
  - ワイルドカード要素(p.566): 要素を明示的にマップするための別のオプションです。**含める**パターンと**除外**パターンを使用して、該当するメタデータから要素名を生成します。
- 入力レコードを出力(ワイルドカード)要素に接続してバインディング(p.569)を作成します。

例54.4. バインディングの作成

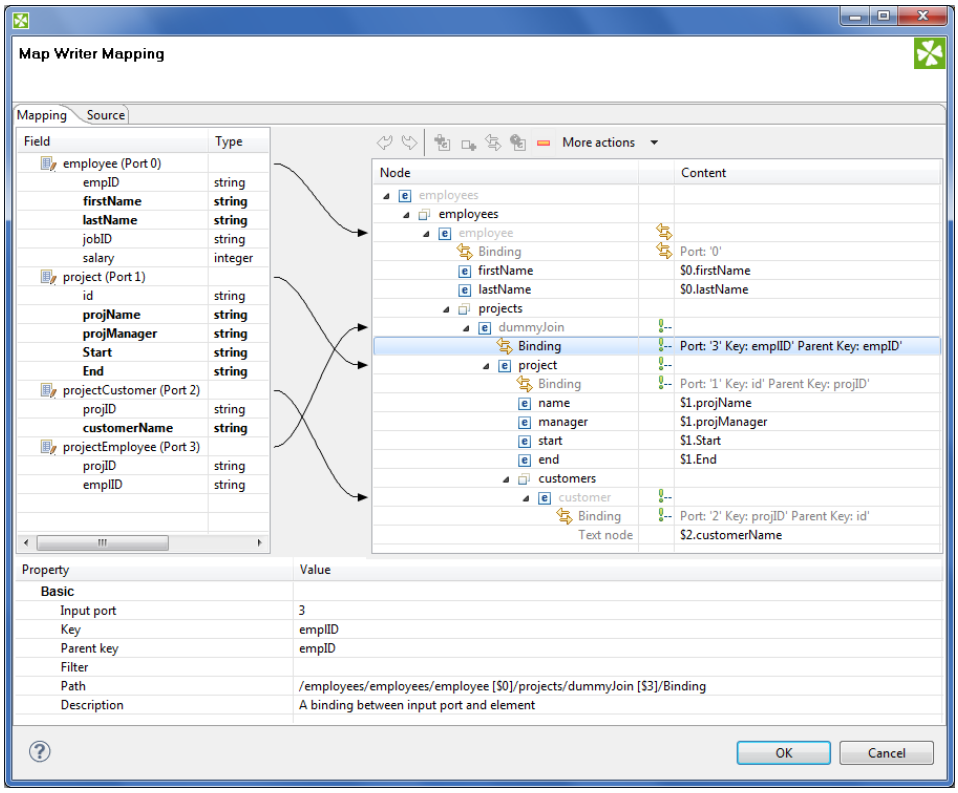


図54.11. JavaMapWriterでのマッピングの例: 従業員はそれぞれの担当プロジェクトに結合されます。太字で示すフィールド(そのコンテンツ)は出力ディクショナリに出力されます。

注意

グラフを拡張し、出力ディクショナリがコンソールに書き出されるようにした場合は、このような構造になります。この抜粋は、前述の図でマップされたJavaマップが内部的にどのように保存されるかを単に示すためのものです。

```
[{employees=[{projects=[{manager=John Major, start=01062005,
name=Data warehouse, customers=[Hanuman, Weblea, SomeBank], end=31052006}],
lastName=Fish, firstName=Mark}, {projects=[{manager=John Smith, start=06062006,
name=JSP, customers=[Sunny, Weblea], end=in progress}, {manager=Raymond Brown,
start=11052004, name=OLAP, customers=[Sunny], end=31012006}], lastName=Simson,
firstName=Jane}, {projects=[{manager=John Major, start=01062005,
name=Data warehouse, customers=[Hanuman, Weblea, SomeBank], end=31052006},
{manager=Raymond Brown, start=11052004, name=OLAP, customers=[Sunny], end=31012006},
{manager=Brandon Morrison, start=01032006, name=Javascripting,
customers=[Nestele, Traincorp, AnotherBank, Intershop], end=in progress}],
lastName=Morrison, firstName=Brandon}]}]
```

例54.5. 配列の書込み

次のような、俳優に関する情報を含む入力ファイルのマッピングがあるとします。説明のために、俳優の個人データを出身国と分けます。次に、すべての国の要約が配列に書き込まれます。

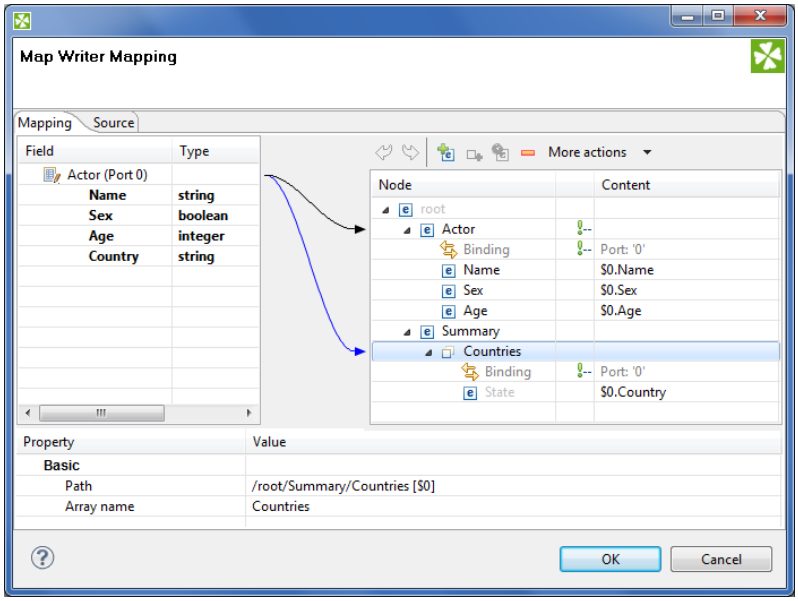


図54.12. JavaMapWriterでの配列のマッピング：配列には、入力フィールドをバインドするダミー要素のStateが含まれています。

配列は、次のようにマップに書き込まれます。

```
[ ...
Summary={Countries=[USA, ESP, ENG]}}]
```

- いつでも「Source」タブ(p.572)に切り替えて、コードで独自のマッピングを記述して確認できます。
- ここに示されている基本的な説明では不十分な場合は、マッピング・プロセス全体について詳細に説明している、XMLWriterの詳細説明(p.560)を参照してください。

JMSWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

JMSWriterは、Cloverデータ・レコードをJMSメッセージに変換します。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
JMSWriter	jmsメッセージ	1	0	はい	いいえ	はい	いいえ

概要

JMSWriterはCloverデータ・レコードを受信し、それらをJMSメッセージに変換して、メッセージを送信します。
コンポーネントは、`DataRecord2JmsMsg`インタフェースを実装するプロセッサ変換か、
`DataRecord2JmsMsgBase`スーパークラスから継承するプロセッサ変換を使用します。
`DataRecord2JmsMsg`インタフェースのメソッドについては後述します。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	データ・レコード用。	任意[1)]

説明:

1): 入力ポートのメタデータには、「**Message body field**」属性で指定されたフィールドが含まれることがあります。

JMSWriterの属性

属性	必須	説明	可能な値
Basic			
JMS connection	はい	使用されるJMS接続のID。	
Processor code		レコードから、Javaでグラフに記述されたJMSメッセージへの変換。	
Processor URL		レコードからJavaで記述されたJMSメッセージへの変換が含まれる外部ファイルの名前(パスを含む)。	
Processor class		レコードからJMSメッセージへの変換を定義する外部クラスの名前。ほとんどの場合、デフォルトのプロセッサ値(`org.jetel.component.jms.DataRecord2JmsMsgProperties`)で間に合います。これは`javax.jms.TextMessage`を生成します。	DataRecord2JmsMsgProperties (デフォルト) | 他のクラス
Processor source charset		Javaでの変換を含む外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト) | その他のエンコーディング
Message charset		JMSメッセージ・コンテンツのエンコーディング。この属性は、デフォルトのプロセッサ実装(`JmsMsg2DataRecordProperties`)でも使用されます。これは`javax.jms.BytesMessage`のみに使用されます。	ISO-8859-1 (デフォルト) | その他のエンコーディング
Advanced			
Message body field		メッセージ本文の取得元および送信元にするメタデータのフィールド名。この属性は、デフォルトのプロセッサ実装(`JmsMsg2DataRecordProperties`)で使用されます。「**Message body field**」が指定されない場合、`bodyField`という名前のフィールドがメッセージ・コンテンツの本文のリソースとして使用されます。メッセージ本文用のフィールド(明示的に指定されたフィールドまたはデフォルトのフィールド)がメタデータに含まれていない場合、プロセッサはbodyFieldという名前のフィールドを設定しようとしますが、出力レコード・メタデータにそのフィールドが存在しない場合、これは内部で無視されます。メタデータのその他のフィールドは、フィールド名と同じ名前を持つメッセージ・プロパティのリソースとして使用されます。	bodyField (デフォルト) | 他の名前

説明:

1) これらのいずれかを設定できます。これらの変換属性はいずれも`DataRecord2JmsMsg`インタフェースを実装します。

詳細は、JMSWriter用Javaインタフェース(p.503)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

JMSWriter用Javaインタフェース

変換は、DataRecord2JmsMsgインタフェースのメソッドを実装し、Transformインタフェースから他の共通メソッドを継承します。共通Javaインタフェース(p.295)を参照してください。

DataRecord2JmsMsgインタフェースのメソッドを次に示します。

- `void init(DataRecordMetadata metadata, Session session, Properties props)`

  プロセッサを初期化します。

- `void preExecute(Session session)`

  これも初期化メソッドであり、各グラフ実行の前に呼び出されます。init()プロシージャとは異なり、ここではこのグラフ実行用のリソースのみを割り当てる必要があります。ここで割り当てられたすべてのリソースは、`postExecute()`メソッドで解放する必要があります。

  `session`はJMSメッセージの作成に使用できます。各グラフ実行で個別のセッションが開きます。したがって、`init()`メソッドで設定されたセッションは、グラフ・インスタンスの最初の実行中にのみ使用できます。

- `Message createMsg(DataRecord record)`

  データ・レコードをJMSメッセージに変換します。すべてのデータ・レコードに対して呼び出されます。

- `Message createLastMsg()`

  このメソッドは最後のレコードの後に呼び出され、JMS出力を終了するメッセージを返すものです。NULLを返した場合、終了メッセージは送信されません。2.8以上。

- `String getErrorMsg()`

  エラー・メッセージを返します。

- `Message createLastMsg(DataRecord record)` (非推奨)

  このメソッドは、2.8以上では明示的に呼び出されません。かわりに`createLastMsg()`を使用してください。

JSONWriter

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

JSONWriterは、JSON形式でデータを書き込みます。

コンポーネント	データ出力	入力ポート	出力ポート	全出力に送信	各出力に送信	変換	変換が必要	Java	CTL
JSONWriter	JSONファイル	1-n	0-1	はい	いいえ	いいえ	いいえ	いいえ	いいえ

概要

JSONWriterは、接続されているすべての入力ポートからデータを受信し、ユーザーが定義したマッピングに基づいてレコードをJSONオブジェクトに変換します。最後に、コンポーネントは生成された要素のツリー構造をJSONファイル、ポートまたはディクショナリの出力に書き込みます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0-N	1つ以上	結合されJSON構造にマップされる入力レコード。	任意(ポートごとに異なるメタデータを保持できます)。
出力	0	いいえ	オプション。ポート書込み用。	1つのフィールドのみ(`byte`、`cbyte`または`string`)が使用されます。フィールド名は「**File URL**」で使用され、出力レコードの処理方法を管理します。出力ポートへの書込み(p.312)を参照してください。

JSONWriterの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	出力JSONのターゲット・ファイル。ライターにサポートされているファイルURL形式(p.310)を参照してください。	
Charset		**JSONWriter**により生成される出力ファイルのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Mapping	1)	入力データを出力JSONにどのようにマップするかを定義します。詳細説明(p.506)を参照してください。	
Mapping URL	1)	マッピング定義が含まれる外部テキスト・ファイル。	

説明:

1) これらのうちどちらかを指定する必要があります。両方が指定された場合は、「**Mapping URL**」の優先度が高くなります。

詳細説明

各JSONオブジェクトには、その他のJSONオブジェクトをネストして含めることができます。そのため、JSON形式はXMLおよび類似のツリー形式に似ています。

その結果、XMLWriter(p.560)ですでに学習した方法と非常によく似た方法で、入力レコードを出力ファイルにマップします。いずれのコンポーネントのマッピング・エディタも類似のロジックを使用しています。マッピングの基本は次のとおりです。

- 入力エッジを**JSONWriter**に接続し、コンポーネントの「**Mapping**」属性を編集します。これにより視覚的なマッピング・エディタが開きます。

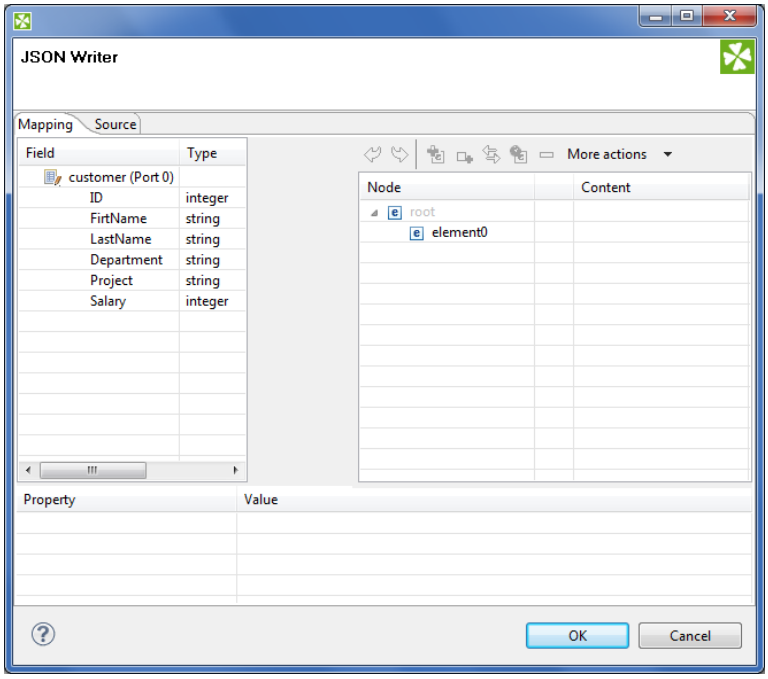


図54.13. 最初に開いた際のJSONWriterのマッピング・エディタ。入力エッジのメタデータが左側に表示されます。右側のペインでは、目的のJSONツリーを設計します。マッピングを実行するには、メタデータを左から右にドラッグします(また、次に説明する追加タスクを実行します)。

- 右側のペインでは、次のもので構成されるJSONツリーを設計します。
  - 要素(p.564)

重要

XMLWriterとは異なり、メタデータを属性にマップしません。

  - **配列**: 配列は、[カッコと]カッコで囲まれた、順序付けられたJSON形式の値のセットです。**JSONWriter**でのこれらのマップ方法は、配列の書込み(p.508)を参照してください。
  - ワイルドカード要素(p.566): 要素を明示的にマップするための別のオプションです。**含める**パターンと**除外**パターンを使用して、該当するメタデータから要素名を生成します。
- 入力レコードを出力(ワイルドカード)要素に接続してバインディング(p.569)を作成します。

例54.6. バインディングの作成

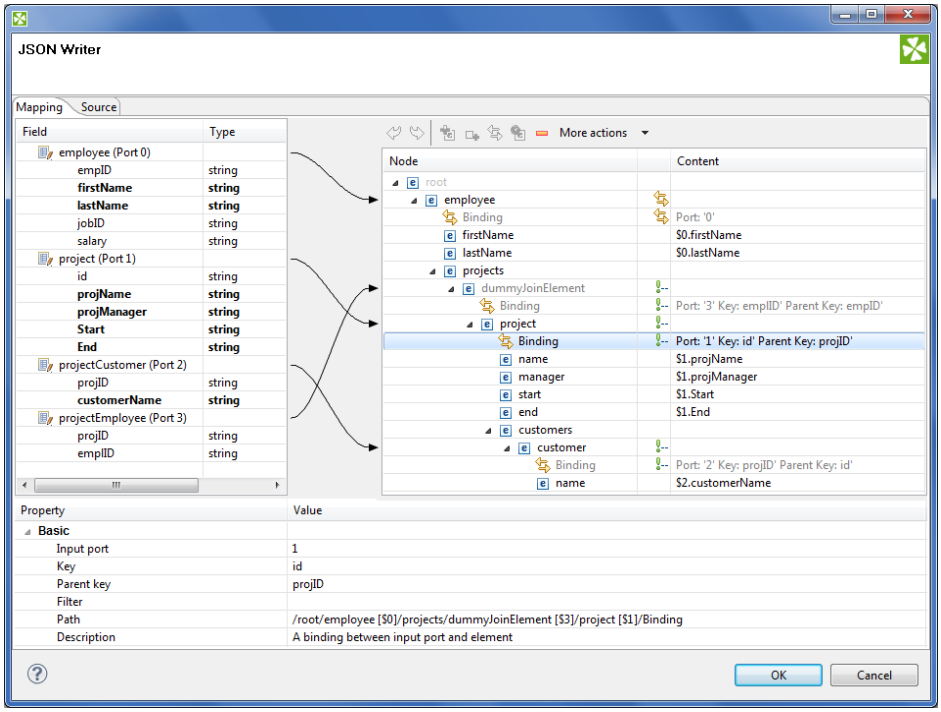


図54.14. JSONWriterでのマッピングの例: 従業員はそれぞれの担当プロジェクトに結合されます。太字で示すフィールド(そのコンテンツ)は出力ファイルに出力されます(次を参照してください)。

前述の図(p.507)に関連する出力ファイルの抜粋(JSONとして記述されている1人の従業員の例):

```
  "employee" : {
  "firstName" : "Jane",
  "lastName" : "Simson",
  "projects" : {
    "project" : {
      "name" : "JSP",
      "manager" : "John Smith",
      "start" : "06062006",
      "end" : "in progress",
      "customers" : {
        "customer" : {
          "name" : "Sunny"
        },
        "customer" : {
          "name" : "Weblea"
        }
      }
    },
    "project" : {
      "name" : "OLAP",
      "manager" : "Raymond Brown",
      "start" : "11052004",
      "end" : "31012006",
      "customers" : {
        "customer" : {
          "name" : "Sunny"
        }
      }
    }
  }
},
```

例54.7. 配列の書込み

次のような、俳優に関する情報を含む入力ファイルのマッピングがあるとします。説明のために、俳優の個人データを出身国と分けます。次に、すべての国の要約が配列に書き込まれます。

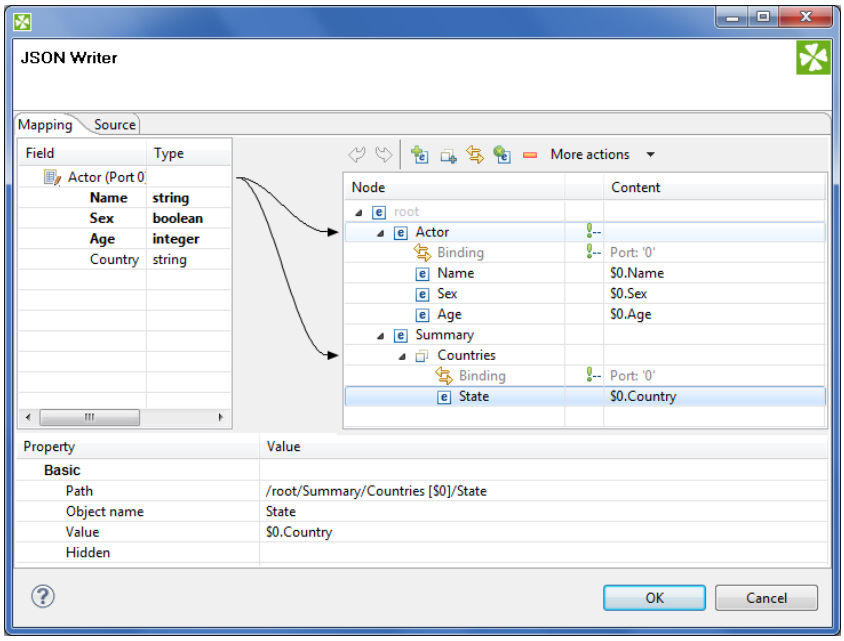


図54.15. JSONWriterでの配列のマッピング：配列には、入力フィールドをバインドするダミー要素のStateが含まれています。

出力JSON:

```
{
  "Actor" : {
    "Name" : "John Malkovich",
    "Sex" : true,
    "Age" : 50
  },
  "Actor" : {
    "Name" : "Liz Hurley",
    "Sex" : false,
    "Age" : 42
  },
  "Actor" : {
    "Name" : "Antonio Banderas",
    "Sex" : true,
    "Age" : 33
  },
  "Summary" : {
    "Countries" : [ "USA", "ENG", "ESP" ]
  }
}
```

- いつでも「Source」タブ(p.572)に切り替えて、コードで独自のマッピングを記述して確認できます。
- ここに示されている基本的な説明では不十分な場合は、マッピング・プロセス全体について詳細に説明している、XMLWriterの詳細説明(p.560)を参照してください。

LDAPWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

LDAPWriterは、LDAPディレクトリに情報を書き込みます。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
LDAPWriter	LDAPディレクトリ・ツリー	1	0-1	いいえ	いいえ	いいえ	いいえ

概要

LDAPWriterは、LDAPディレクトリに情報を書き込みます。これにより、LDAPディレクトリに関する情報を更新するためのロジックが提供されます。更新とは、エントリの追加/削除および属性の追加/置換/削除です。メタデータはLDAPオブジェクト属性名に一致する必要があります。DNメタデータ属性は必須です。

`string`、`byte`および`cbyte`のメタデータ型のみがサポートされます。ほとんどのLDAP型はClover文字列と互換性がありますが、たとえば、バイト・データ・フィールドからデータを移入するためには`userPassword` LDAP型が必要です。LDAPルールが適用されます。エントリを追加するには、メタデータに必須属性(およびオブジェクト・クラス)が必要です。

注意

LDAP属性には複数値を指定できます。デフォルトの値セパレータは|で、これが適切なのは文字列データ・フィールドに対してのみです。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	1	はい	正しいデータ・レコード用	任意[1)]
出力	0-1	いいえ	拒否されたレコード用	入力0

説明:

1): 入力のメタデータは、LDAPオブジェクト属性名と正確に一致している必要があります。識別名メタデータ属性は必須です。LDAP属性は複数値を取るため、これらの値はパイプまたは指定されたセパレータで区切ることができます。サポートされるメタデータ型は文字列およびバイトのみです。

LDAPWriterの属性

属性	必須	説明	可能な値
Basic			
LDAP URL	はい	ディレクトリのLDAP URL。パイプで区切られたURLのリストにできます。	パターン: `ldap://host:port/`
Action		エントリについて実行されるアクションを定義します。	replace_attributes (デフォルト) | add_entry | remove_entry | remove_attributes
User		LDAPディレクトリへの接続時に使用されるユーザーDN。`cn=john.smith,dc=example,dc=com`のようになります。	
Password		LDAPディレクトリへの接続時に使用されるパスワード。	
Advanced			
Multi-value separator		**LDAPWriter**は、複数値を持つキーを処理できます。これらは、この文字列または文字によって区切られます。<none>は、この機能をオフにする特殊なエスケープ値であり、指定する最初の値のみが書き込まれます。この属性は、文字列データ型のみに使用できます。バイト型が使用される場合、最初の値のみが書き込まれます。	"|" (デフォルト) | その他の文字または文字列

LotusWriter

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、第54章「ライター」(p.459)を参照してください。

要約

LotusWriterは、**Lotus Domino**データベースにデータを書き込みます。データ・レコードは、データベースにLotusドキュメントとして格納されます。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
LotusWriter	Lotus Notes	1	0-1	✖	✖	✖	✖

概要

LotusWriterは、Lotusデータベースにデータ・レコードを書き込むことができるコンポーネントです。書込みは、**Lotus Dominoサーバー**に接続することによって行われます。

データ・レコードは、**ドキュメント**としてLotusデータベースに書き込まれます。Lotusのドキュメントは、キーと値のペアのリストです。書き込まれるデータ・レコードのフィールドごとに、キーと値のペアが1つ生成され、キーにはフィールド名、値にはフィールドの値が設定されます。

このコンポーネントのユーザーは、Lotusに接続するためのJavaライブラリを提供する必要があります。ライブラリは、Lotus NotesまたはLotus Dominoのインストール先にあります。このライブラリに至るパスが提供されているか、またはライブラリがユーザーのクラスパスに配置されないかぎり、**LotusWriter**コンポーネントはLotusと通信できません。ライブラリへのパスは、Lotus接続の詳細で指定できます(第25章「Lotus接続」(p.191)を参照してください)。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	入力データ・レコード用	
出力	0	✖	無効なデータ・レコード用	

LotusWriterの属性

属性	必須	説明	可能な値
Basic			
Domino connection	✔	Lotus Dominoデータベースへの接続のID。	
Mode	✖	書込みモード。挿入モードは常にデータベースに新しいドキュメントを作成します。更新モードではビューを指定する必要があります。更新操作ではまず、受信したデータ・レコードと同じキー値を持つすべてのドキュメントがビューで検索されます。その後、見つかったすべてのドキュメントが更新されるか、または最初に見つかったドキュメントのみが更新されます。	"insert" (デフォルト) | "update"
View	✖	データ・レコードが更新される、Lotusデータベースのビューの名前。	
Advanced			
Mapping	✖	マッピングが提供されていない場合、新しいドキュメントは入力ポートから取得したものとまったく同じ一連のフィールドを取得します。マッピングを使用すると、Lotusドキュメントにどのフィールドを書き込むかをカスタマイズできます。フィールドは名前や順序を変えて書き込むことができ、一部のフィールドをスキップしたり、名前を変えて複数回書き込むことができます。多くの場合、**Document** Lotusフォームからフィールドを操作することをお薦めします。Documentフォーム・フィールドへの入力ポート・フィールドのマッピングは、この属性で確立できます。	docFieldX := inFieldY; ...
Compute with form	✖	有効にすると、ドキュメントが新しく作成されたときに計算が開始されます。計算は、通常、**Document**フォームに定義します。このフォームは、Lotus Notesのユーザーが新しいドキュメントを作成する場合に使用します。計算によって、たとえば、空のフィールドにデフォルト値を入力したり、データ変換を実行します。	true (デフォルト) | false
Skip invalid documents	✖	有効にすると、Lotusによって無効とマークされたドキュメントはLotusデータベースに保存されません。この設定を使用するには、「**Compute with form**」属性を有効にする必要があり、そのようにしない場合、検証は実行されません。検証は、Documentフォームの計算アクションによって実行されます。	true | false (デフォルト)
Update mode	✖	更新対象として複数のドキュメントが見つかったときの遅延更新モードの使用および動作を切り替えます。遅延更新モードは値が実際に変更されたときにドキュメントを更新するのみであり、書き込まれる値(オプションの計算後)は元の値とは異なります。更新対象として複数のドキュメントが見つかったときは、最初に見つかったドキュメントのみを更新するか、またはすべてのドキュメントを更新できます。	"Lazy, first match" (デフォルト) | "Lazy, all matches" | "Eager, first match" | "Eager, all matches"
Multi-value fields	✖	複数値フィールドとして処理する入力フィールドを指定します。複数値フィールドは、「**Multi-value separator**」属性に指定されたセパレータを使用して複数の文字列に分割されま	入力フィールドのセミコロン区切りリスト

属性	必須	説明	可能な値
		す。結果として得られる値の配列が複数値ベクターとしてLotusデータベースに格納されます。	
Multi-value separator	✖	複数値Lotusフィールドからの値を区切るために使用される文字列。	";" (デフォルト) | "," | ":" | "|" | "\t" | その他の文字または文字列

MSSQLDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

MSSQLDataWriterは、MSSQLデータベースにデータをロードします。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
MSSQLDataWriter	データベース	0-1	0-1	いいえ	いいえ	いいえ	いいえ

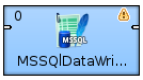
概要

MSSQLDataWriterは、MSSQLデータベース・クライアントを使用してデータベースにデータをロードします。これは、入力ポートを介してまたはファイルからデータを読み取ります。入力ポートが他のコンポーネントに接続されていない場合、コンポーネントで指定される別のファイルにデータが含まれている必要があります。

オプションの出力ポートに他のコンポーネントを接続すると、拒否されたレコードおよびエラーに関する情報を記録できます。このエラー・ポート上のメタデータでは、メタデータ・フィールドが入力レコードおよび末尾にある`number of incorrect row (integer)`、`number of incorrect column (integer)`、`error message (string)`の3つの追加フィールドと同じである必要があります。

SQL Serverのクライアント接続コンポーネントは、**CloverETL**が実行されるものと同じマシンにインストールして構成する必要があります。bcpコマンドライン・ツールが使用可能である必要があります。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	1)	データベースにロードされるレコード	任意
出力	0	いいえ	正しくないレコードに関する情報	入力0 (さらにMSSQLDataWriterのエラー・フィールド(p.515)の3つ[2])

説明:

1): ロードするデータを含むファイル(**Loader input file**)が指定されていない場合は、入力ポートが接続されている必要があります。

2): 出力ポート0上のメタデータには、末尾に`number of incorrect row`、`number of incorrect column`、`error message`の3つの追加フィールドが含まれています。

表54.3. MSSQLDataWriterのエラー・フィールド

フィールド番号	フィールド名	データ型	説明
LastInputField + 1	<anyname1>	integer	正しくない行の数
LastInputField + 2	<anyname2>	integer	正しくない列の数
LastInputField + 3	<anyname3>	string	エラー・メッセージ

MSSQLDataWriterの属性

属性	必須	説明	可能な値
Basic			
Path to bcp utility	はい	パスを含む、bcpユーティリティの名前。SQL Serverのクライアント接続コンポーネントは、Cloverが実行されるものと同じマシンにインストールして構成する必要があります。bcpコマンドライン・ツールが使用可能である必要があります。	
Database	はい	宛先の表またはビューが存在するデータベースの名前。	
Server name		bcpユーティリティの接続先のサーバーの名前。bcpユーティリティがサーバーのローカルな名前付きインスタンスまたはリモートの名前付きインスタンスに接続する場合、**Server name**は`serverName\instanceName`に設定する必要があります。bcpユーティリティがサーバーのデフォルト・インスタンスまたはリモート・インスタンスに接続する場合、**Server name**は`serverName`に設定する必要があります。このように設定されていない場合、bcpはlocalhostのサーバーのローカルなデフォルト・インスタンスに接続します。同じ意味が、「**Parameters**」に設定できる`serverName`にも該当します。ただし、「**Server name**」属性と「`serverName`」パラメータの両方が設定されている場合、「**Parameters**」の`serverName`は無視されます。	
Database table	1)	宛先の表の名前。	
Database view	1)	宛先のビューの名前。ビューのすべての列は、同じ表を参照している必要があります。	
Database owner		表またはビューの所有者。操作を実行するユーザーが所有者である場合は指定する必要がありません。ユーザーが所有者でない場合に設定されていないと、SQL Serverはエラー・メッセージを返し、プロセスは取り消されます。	
User name	はい	サーバーへの接続時に使用されるログインID。「**Parameters**」属性に「`userName`」パラメータの値を指定して、同じものを設定できます。設定した場合、「**Parameters**」の`userName`は無視されます。	

属性	必須	説明	可能な値
Password	はい	サーバーへの接続時に使用されるログインIDのパスワード。「**Parameters**」属性に「password」パラメータの値を指定して、同じものを設定できます。設定した場合、「**Parameters**」のpasswordは無視されます。	
Advanced			
Column delimiter		データの各列で使用されるデリミタ。フィールド値内にデリミタは指定できません。「**Parameters**」属性に「fieldTerminator」パラメータの値を指定して、同じものを設定できます。設定した場合、「**Parameters**」のfieldTerminatorは無視されます。	\t (デフォルト) | その他の任意の文字
Loader input file		パスを含む、ロードする入力ファイルの名前。詳細は、Loader input file(p.516)を参照してください。	
Parameters		bcpユーティリティでパラメータとして使用できるすべてのパラメータ。これらの値は、key=valueという形式のペアまたはkeyのみ(key値がブールのtrueの場合)のシーケンスに、セミコロンで区切られて含まれています。パラメータの値にセミコロンが含まれる場合は、その値を二重引用符で囲む必要があります。詳細は、Parameters(p.516)を参照してください。	

説明:

1) これらのいずれかを指定する必要があります。

2) 入力ポートが接続されていない場合、「**Loader input file**」が指定され、データが含まれている必要があります。詳細は、Loader input file(p.516)を参照してください。

詳細説明

Loader input file

入力ポートに接続されているエッジによっては、別の属性(**Loader input file**)を指定できるか、または指定する必要があります。これは、データがロードされる入力ファイルの名前(パスを含む)です。

- これが設定されていない場合、ローダー・ファイルがCloverまたはOSの一時ディレクトリに作成されるか(Windowsの場合)、一時ファイルのかわりに名前付きパイプが使用されます(Unixの場合)。ロードが終了するとファイルは削除されます。
- これが設定されている場合、指定したファイルが作成されます。これはデータがロードされた後も削除されず、グラフが実行されるたびに上書きされます。
- 入力ポートが接続されていない場合、このファイルは存在する必要があり、このファイルを指定する必要があり、ロードされるデータがこのファイルに含まれている必要があります。これは削除も上書きもされません。

Parameters

MSSQLを操作するときに使用できる一連のパラメータ(**Parameters**)を設定することもお薦めします。たとえば、ポートの数などを設定できます。すべてのパラメータがkey=valueまたはkeyのみという形式である必要があります(その値がtrueである場合)。個々のパラメータはそれぞれコロン、セミコロンまたはパイプで区切られている必要があります。コロン、セミコロンまたはパイプもパラメータ値に含めることができますが、その場合値を二重引用符で囲む必要があります。

オプションのパラメータの中には、**「User name」**属性、**「Password」**属性または**「Column delimiter」**属性にそれぞれuserName、passwordまたはfieldTerminatorを設定できるものがあります。この3つの属性(**User name**、**Password**または**Column delimiter**)のいくつかが設定された場合、対応するパラメータ値は無視されます。

サーバー名を指定する必要もある場合は、パラメータ内で指定してください。パターンは serverName=[msServerHost]:[msServerPort]です。たとえば、サーバー名とユーザー名の両方を指定する場合は、serverName=msDbServer:1433|userName=msUserとします。

MySQLDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

MySQLDataWriterは高速のMySQL表ローダーです。MySQLネイティブ・クライアントを使用します。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
MySQLDataWriter	データベース	0-1	0-1	いいえ	いいえ	いいえ	いいえ

概要

MySQLDataWriterは、ネイティブのMySQLデータベース・クライアントを使用して、データを迅速にデータベース表にロードします。

これは、入力ポートまたは入力ファイルからデータを読み取ることができます。

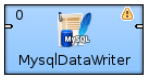
オプションの出力ポートを接続し、拒否と報告されたレコードを読み取ることができます。

入力ポートからの読取り(接続済の入力ポート)は、データのダンプを一時ファイルに生成します(ファイルはその後`mysql`ユーティリティによって使用されます)。**「Loader input file」**属性を設定するか、または空白のままにしてデフォルトを使用して、一時ファイルを明示的に設定できます。

ファイルからの読取り(接続済入力なし)は、「Loader input file」属性をデータ・ファイルへのパスとして使用します。この場合この属性は必須です。ファイルは、`mysql`ユーティリティで認識される形式である必要があります(『MySQL LOAD DATA』を参照してください)。

このコンポーネントは、MySQLネイティブ・コマンドライン・クライアント(`bin/mysql`または`bin/mysql.exe`)を実行します。クライアントは、グラフが実行されているのと同じマシンにインストールする必要があります。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	1)	データベースにロードされるレコード	任意
出力	0	いいえ	正しくないレコードに関する情報	MySQLDataWriterのエラー・メタデータ(p.519)[2)]

説明:

1): ロードするデータを含むファイル(**Loader input file**)が指定されていない場合は、入力ポートが接続されている必要があります。

2): **エラー・メタデータ**は自動入力関数(p.132)を使用できません。

表54.4. MySQLDataWriterのエラー・メタデータ

フィールド番号	フィールド名	データ型	説明
0	<any_name1>	integer	正しくないレコードの番号(レコードには1から始まる番号が付けられます)
1	<any_name2>	integer	正しくない列の数
2	<any_name3>	string	エラー・メッセージ

MySQLDataWriterの属性

属性	必須	説明	可能な値
Basic			
Path to mysql utility	はい	パスを含む、mysqlユーティリティの名前。Cloverが実行されるのと同じマシンにインストールして構成する必要があります。mysqlコマンドライン・ツールが使用可能である必要があります。	
Host		データベース・サーバーがあるホスト。	localhost (デフォルト) | その他のホスト
Database	はい	レコードがロードされるデータベースの名前。	
Database table	はい	レコードがロードされるデータベース表の名前。	
User name	はい	データベース・ユーザー。	
Password	はい	データベース・ユーザーのパスワード。	
Advanced			
Path to control script		LOAD DATA INFILE文が含まれているコマンド・ファイルの、パスを含む名前。詳細は、Path to Control Script(p.520)を参照してください。	

属性	必須	説明	可能な値
Lock database table		デフォルトでは、データベースはロックされず、マルチユーザー・アクセスが許可されます。trueに設定した場合、データベース表はロックされて排他的アクセスが確保され、ロード速度が向上する可能性があります。	false (デフォルト) | true
Ignore rows		スキップするデータ・ファイルの行の数。デフォルトでは、スキップされるレコードはありません。データがある入力ファイルに対してのみ有効。	0 (デフォルト) | 1-N
Column delimiter		データの各列で使用されるデリミタ。フィールド値にこのデリミタを含めることはできません。デフォルトではタブが使用されます。	\t (デフォルト) | その他の文字
Loader input file	1)	パスを含む、ロードする入力ファイルの名前。詳細は、Loader input file(p.520)を参照してください。	
Parameters		loadメソッドでパラメータとして使用できるすべてのパラメータ。これらの値は、key=valueという形式のペアまたはkeyのみ(key値がブールのtrueの場合)のシーケンスに、セミコロン、コロンまたはパイプで区切られて含まれています。パラメータの値にデリミタが含まれる場合は、その値を二重引用符で囲む必要があります。	

説明:

1) 入力ポートが接続されていない場合、「**Loader input file**」が指定され、データが含まれている必要があります。詳細は、Loader input file(p.520)を参照してください。

詳細説明

Path to Control Script

LOAD DATA INFILE文が含まれているコマンド・ファイルの、パスを含む名前(『MySQL LOAD DATA』を参照してください)。

- これが設定されていない場合、コマンド・ファイルはClover一時ディレクトリに作成され、ロードが終了すると削除されます。
- これが設定され、指定されたコマンド・ファイルが存在しない場合、指定された名前およびパスで一時的なコマンド・ファイルが作成され、ロードが終了しても削除されません。
- これが設定され、指定されたコマンド・ファイルが存在する場合、Cloverによって作成されたコマンド・ファイルではなく、このファイルが使用されます。

Loader input file

パスを含む、ロードする入力ファイルの名前。

- これが設定されていない場合、ローダー・ファイルがCloverまたはOSの一時ディレクトリに作成されるか(Windowsの場合)、一時ファイルのかわりにstdinが使用されます(Unixの場合)。ロードが終了するとファイルは削除されます。
- これが設定されている場合、指定したファイルが作成されます。これはデータがロードされた後も削除されず、グラフが実行されるたびに上書きされます。
- 入力ポートが接続されていない場合、このファイルが指定され、存在し、データベースにロードされるデータを含んでいる必要があります。このファイルは削除も上書きもされません。

OracleDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

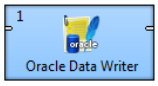
OracleDataWriterは、データをOracleデータベースにロードします。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
OracleDataWriter	データベース	0-1	0-1	いいえ	いいえ	いいえ	いいえ

概要

OracleDataWriterは、Oracleデータベース・クライアントを使用してデータをデータベースにロードします。データは、入力ポートを介してまたは入力ファイルから読み取ることができます。入力ポートが他のコンポーネントに接続されていない場合、コンポーネントで指定される入力ファイルにデータが含まれている必要があります。オプションの出力ポートに他のコンポーネントを接続した場合、拒否されたレコードがそのポートに送信されます。Oracle sqlldrデータベース・ユーティリティは、**CloverETL**が実行されるコンピュータにインストールする必要があります。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	1)	データベースにロードされるレコード	任意
出力	0	いいえ	拒否されたレコード	入力0

説明:

1): ロードするデータを含むファイル(**Loader input file**)が指定されていない場合は、入力ポートが接続されている必要があります。

OracleDataWriterの属性

属性	必須	説明	可能な値
Basic			
Path to sqlldr utility	はい	パスを含む、sqlldrユーティリティの名前。	
TNS name	はい	TNS名識別子。	
User name	はい	Oracleデータベースへの接続時に使用されるユーザー名。	
Password	はい	Oracleデータベースへの接続時に使用されるパスワード。	
Oracle table	はい	レコードがロードされるデータベース表の名前。	
Advanced			
Control script		sqlldrユーティリティの制御スクリプト。詳細は、制御スクリプト(p.523)を参照してください。	
Append		データベース表で実施する操作を指定します。詳細は、「Append」属性(p.524)を参照してください。	append (デフォルト) | insert | replace | truncate
Log file name		プロセスが記録されるファイルの名前。	${PROJECT}/loaderinputfile.log
Bad file name		エラーが発生したレコードが書き込まれるファイルの名前。	${PROJECT}/loaderinputfile.bad
Discard file name		選択基準を満たさないレコードが書き込まれるファイルの名前。	${PROJECT}/loaderinputfile.dis
DB column names		データベース表内のすべての列の名前。	
Loader input file		パスを含む、ロードする入力ファイルの名前。詳細は、Loader Input File(p.524)を参照してください。	
Max error count		許可される挿入エラーの最大数。この数を超えるとグラフが失敗します。エラーを許可しない場合は、この属性を0に設定します。すべてのエラーを許可するには、この属性を非常に大きな数値に設定します。	50 (デフォルト) | 0-N
Max discard count		グラフが停止する前に、破棄できるレコードの数。1に設定すると、レコードが1回破棄されてもグラフの実行が停止します。	all (デフォルト) | 1-N
Ignore rows		データをデータベースにロードするときにスキップするデータ・ファイルの行の数。	0 (デフォルト) | 1-N
Commit interval		従来型パスによるロードのみ: **Commit interval**には、バインド配列の行数を指定します。ダイレクト・パス・ロードのみ: **Commit interval**には、データを保存する前にデータ・ファイルから読み取る行数を指定します。デフォルトでは、すべての行が読み取られ、ロードの最後にすべてのデータが一度にまとめて保存されます。	64 (従来型パスのデフォルト) | すべての (ダイレクト・パスのデフォルト) | 1-N

属性	必須	説明	可能な値
Use file for exchange		デフォルトでは、Unixではパイプ転送が使用されます。これがtrueに設定され、「**Loader input file**」が設定されていない場合は、一時ファイルが作成されてデータソースとして使用されます。デフォルトでは、Windowsでは一時ファイルが作成されてデータソースとして使用されます。ただし、クライアントによっては一時データ・ファイルの作成は必要なく、このようなクライアントに対してはこの属性をfalseに設定できます。	false (Unixのデフォルト) | true (Windowsのデフォルト)
Parameters		sqlldrユーティリティでパラメータとして使用できるすべてのパラメータ。これらの値は、key=valueという形式のペアまたはkeyのみ(key値がブールのtrueの場合)のシーケンスに、セミコロン、コロンまたはパイプで区切られて含まれています。パラメータの値にセミコロン、コロンまたはパイプが含まれる場合は、その値を二重引用符で囲む必要があります。	
Fail on warnings		デフォルトでは、コンポーネントはエラーが発生したときに失敗します。属性をtrueに変更すると、警告が発生したときにコンポーネントが失敗するようにできます。背景: 基礎となる一括ローダー・ユーティリティが警告で終了した場合、その警告は単にコンソールに記録されます。基礎となる一括ローダーからの警告は以降の処理に深刻な影響を及ぼす可能性があるため、この動作は適切でない場合があります。たとえば、表領域を拡張できないと、すべてのデータ・レコードがデータベースにロードされず、想定したタスクが正常に完了しないことがあります。	false (デフォルト) | true

詳細説明

Control Script

sqlldrユーティリティの制御スクリプト。

- 指定した場合、「**Oracle table**」属性と「**Append**」属性の両方が無視されます。入力ポートが接続されていない場合は、指定する必要があります。このような場合は、「**Loader input file**」も定義する必要があります。
- **Control script**が設定されていない場合は、デフォルトの制御スクリプトが使用されます。

例54.8. 制御スクリプトの例

```
LOAD DATA
INFILE *
INTO TABLE test
append
(
    name TERMINATED BY ';',
    value TERMINATED BY '\n'
)
```

「Append」属性

- **Append (デフォルト)**

  データが単に表に追加されることを指定します。既存の空き領域は使用されません。

- **Insert**

  INSERT文で表/ビューに新しい行を追加します。OracleのINSERT文は、表、ビューのベース表、パーティション化された表のパーティションまたはコンポジット・パーティション化された表のサブパーティション、またはオブジェクト表またはオブジェクト・ビューのベース表に行を追加する場合に使用します。

  VALUES句を指定したINSERT文は、VALUES句に指定された値が含まれている単一行を表に追加します。

  VALUES句ではなく副問合せを指定したINSERT文は、副問合せから返されたすべての行を表に追加します。Oracleは、副問合せを処理し、返されたそれぞれの行を表に挿入します。副問合せによって行が選択されなかった場合、Oracleは行を表に挿入しません。副問合せは、INSERT文のターゲット表を含め、任意の表、ビューまたはスナップショットを参照できます。

- **Update**

  表またはビューのベース表の既存の値を変更します。

- **Truncate**

  表またはクラスタからすべての行を削除し、STORAGEパラメータを表またはクラスタの作成時の値にリセットします。

Loader Input File

パスを含む、ロードする入力ファイルの名前。

- これが設定されていない場合、(「**Use file for exchange**」がfalseに設定されている場合を除いて)ローダー・ファイルがCloverまたはOSの一時ディレクトリに作成されるか(Windowsの場合)、一時ファイルのかわりに名前付きパイプが使用されます(Unixの場合)。ロードが終了すると、作成されたファイルは削除されます。
- これが設定されている場合、指定したファイルが作成されます。作成されたファイルはデータがロードされた後も削除されず、グラフが実行されるたびに上書きされます。
- 入力ポートが接続されていない場合、このファイルが指定され、存在し、データベースにロードされるデータを含んでいる必要があります。同時に、**Control script**を指定する必要があります。このファイルは削除も上書きもされません。

PostgreSQLDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

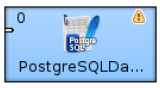
PostgreSQLDataWriterは、PostgreSQLデータベースにデータをロードします。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
PostgreSQLDataWriter	データベース	0-1	0	いいえ	いいえ	いいえ	いいえ

概要

PostgreSQLDataWriterは、PostgreSQLデータベース・クライアントを使用してデータをデータベースにロードします。データは、入力ポートを介してまたは入力ファイルから読み取ることができます。入力ポートが他のコンポーネントに接続されていない場合、コンポーネントで指定される入力ファイルにデータが含まれている必要があります。PostgreSQLクライアント・ユーティリティ(`psql`)は、**CloverETL**が実行されるのと同じマシンにインストールして構成する必要があります。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	1)	データベースにロードされるレコード	任意

説明:

1): ロードするデータを含むファイル(**Loader input file**)が指定されていない場合は、入力ポートが接続されている必要があります。

PostgreSQLDataWriterの属性

属性	必須	説明	可能な値
Basic			
Path to psql utility	はい	パスを含む、psqlユーティリティの名前。**CloverETL**が実行されるのと同じマシンにインストールして構成する必要があります。psqlコマンドライン・ツールが使用可能である必要があります。	
Host		データベース・サーバーがあるホスト。	localhost (デフォルト) | その他のホスト
Database	はい	レコードがロードされるデータベースの名前。	
Database table		レコードがロードされるデータベース表の名前。	
User name		サーバーへの接続時に使用されるPostgreSQLユーザー名。	
Advanced			
Fail on error		デフォルトでは、エラーのたびにグラフが失敗します。PostgreSQLデータベースの標準の動作にする場合は、この属性を`false`に切り替える必要があります。`false`に設定した場合、一部のエラーがPostgreSQLデータベースで発生しても、グラフは正常に実行されます。	true (デフォルト) | false
Path to control script		`\copy`文が含まれているコマンド・ファイルの、パスを含む名前。詳細は、Path to Control Script(p.526)を参照してください。	
Column delimiter		データの各列で使用されるデリミタ。フィールド値にこのデリミタを含めることはできません。	タブ文字(テキスト・モードのデフォルト) | カンマ(CVSモードのデフォルト)
Loader input file		パスを含む、ロードする入力ファイルの名前。詳細は、Loader Input File(p.527)を参照してください。	
Parameters		psqlユーティリティまたは`\copy`文でパラメータとして使用可能なすべてのパラメータ。これらの値は、`key=value`という形式のペアまたは`key`のみ(`key`値がブールの`true`の場合)のシーケンスに、セミコロン、コロンまたはパイプで区切られて含まれています。パラメータの値にセミコロン、コロンまたはパイプが含まれる場合は、その値を二重引用符で囲む必要があります。	

詳細説明

Path to Control Script

`\copy`文が含まれているコマンド・ファイルの、パスを含む名前。

- これが設定されていない場合、コマンド・ファイルはClover一時ディレクトリに作成され、ロードが終了すると削除されます。
- これが設定されているものの、指定のコマンド・ファイルが存在しない場合、一時ファイルが指定の名前およびパスで作成され、ロードが終了しても削除されません。

- これが設定され、指定のコマンド・ファイルが存在する場合、Cloverによって作成されたファイルではなく、このファイルが使用されます。ロードが終了してもファイルは削除されません。

Loader Input File

パスを含む、ロードする入力ファイルの名前。

- 入力ポートが接続され、このファイルが設定されていない場合、一時ファイルは作成されません。データはエッジから読み取られてデータベースに直接ロードされます。
- これが設定されている場合、指定したファイルが作成されます。これはデータがロードされた後も削除されず、グラフが実行されるたびに上書きされます。
- 入力ポートが接続されていない場合、このファイルが存在し、データベースにロードされるデータを含んでいる必要があります。これは削除されず、別のグラフ実行で上書きされることもありません。

QuickBaseImportCSV

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

QuickBaseImportCSVは、**QuickBase**オンライン・データベース表レコードを追加および更新します。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
QuickBaseImportCSV	QuickBase	1	0-2	✖	✖	✖	✖

概要

QuickBaseImportCSVは、入力ポートを介してデータ・レコードを受信し、**QuickBase**オンライン・データベース(http://quickbase.intuit.com)に書き込みます。正常に書き込まれたレコードのレコードIDを生成し、最初のオプションの出力ポートを介して送信します(接続済の場合)。この出力ポート上の最初のフィールドは、stringデータ型である必要があります。このフィールドに、生成されたレコードIDが書き込まれます。拒否されたデータ・レコードに関する情報は、オプションの2番目のポートを介して送信できます(接続済の場合)。

このコンポーネントは、`API_ImportFromCSV` HTTP対話をラップします(http://www.quickbase.com/api-guide/importfromcsv.html)。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	入力データ・レコード用	任意
出力	0	✖	受け入れられたデータ・レコード用	インポート済レコードの*Record ID#*フィールド値表の文字列フィールド
出力	1	✖	拒否されたデータ・レコード用	QuickBaseImportCSVのエラー・フィールド(p.529)の最大3つによって拡張された入力メタデータ

表54.5. QuickBaseImportCSVのエラー・フィールド

フィールド番号	フィールド名	データ型	説明
オプション[1]	「**Error code output field**」で指定します。	integer | long	エラー・コード
オプション[1]	「**Error message output field**」で指定します。	string	エラー・メッセージ
オプション[1]	「**Batch number output field**」で指定します。	integer | long	失敗したバッチの(1から始まる)インデックス

[1]エラー・フィールドは入力フィールドの後ろに置く必要があります。

QuickBaseImportCSVの属性

属性	必須	説明	可能な値
Basic			
QuickBase connection	✔	QuickBaseオンライン・データベースへの接続のID。第24章「QuickBase接続」(p.190)を参照してください。	
Table ID	✔	データ・レコードが書き込まれる、QuickBaseアプリケーション内の表のID (表IDのを取得するには、`application_stats`を確認してください)。	
Batch size		1つのバッチ内のレコードの最大数。	100 (デフォルト) | 1-N
Clist		入力データ列のマップ先となる表の*field_id*のピリオド区切りリスト。順序は保持されます。このため、インポート対象外の列に対しては0を入力してください。指定しない場合、データベースは一意のレコードを追加しようとします。レコードを編集する場合、この設定は必須です。入力データには、更新する各レコードのレコードIDが含まれている列を含める必要があります。	
Error code output field		エラー・コードを格納するためのエラー・メタデータ・フィールドの名前。QuickBaseImportCSVのエラー・フィールド(p.529)を参照してください。	
Error message output field		エラー・メッセージを格納するためのエラー・メタデータ・フィールドの名前。QuickBaseImportCSVのエラー・フィールド(p.529)を参照してください。	
Batch number output field		破損したバッチのインデックスを格納するためのエラー・メタデータ・フィールドの名前。QuickBaseImportCSVのエラー・フィールド(p.529)を参照してください。	

QuickBaseRecordWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

QuickBaseRecordWriterは、データを**QuickBase**オンライン・データベースに書き込みます。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
QuickBaseRecordWriter	QuickBase	1	0-1	✖	✖	✖	✖

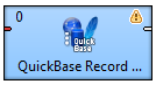
概要

QuickBaseRecordWriterは、入力ポートを介してデータ・レコードを受信して、**QuickBase**オンライン・データベース(http://quickbase.intuit.com)に書き込みます。

このコンポーネントは、API_AddRecord HTTP対話をラップします(http://www.quickbase.com/api-guide/add_record.html)。

オプションの出力ポートが接続されている場合、拒否されたレコードがエラーに関する情報とともにそのポートを介して送信されます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	入力データ・レコード用	任意
出力	0	✖	拒否されたデータ・レコード用	QuickBaseRecordWriterのエラー・フィールド(p.531)の最大2つによって拡張された入力メタデータ

表54.6. QuickBaseRecordWriterのエラー・フィールド

フィールド番号	フィールド名	データ型	説明
オプション[1]	「**Error code output field**」で指定します。	integer | long	エラー・コード
オプション[1]	「**Error message output field**」で指定します。	string	エラー・メッセージ

[1]エラー・フィールドは入力フィールドの後ろに置く必要があります。

QuickBaseRecordWriterの属性

属性	必須	説明	可能な値
Basic			
QuickBase connection	✔	QuickBaseオンライン・データベースへの接続のID。第24章「QuickBase接続」(p.190)を参照してください。	
Table ID	✔	データ・レコードが書き込まれる、QuickBaseアプリケーション内の表のID (表IDのを取得するには、`application_stats`を確認してください)。	
Mapping	✔	メタデータ・フィールド値が書き込まれる、セミコロンで区切られたデータベース表のfield_idのリスト。	
Error code output field		エラー・コードが格納されるフィールドの名前。QuickBaseRecordWriterのエラー・フィールド(p.531)を参照してください。	
Error message output field		エラー・メッセージが格納されるフィールドの名前。QuickBaseRecordWriterのエラー・フィールド(p.531)を参照してください。	

SpreadsheetDataWriter

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

SpreadsheetDataWriterは、データをスプレッドシート(XLSファイルまたはXLSXファイル)に書き込みます。**SpreadsheetDataWriter**は元の**XLSDataWriter**に代わるものであり、より多くの新機能と書込みモードを備え、パフォーマンスが改善されています。(**XLSDataWriter**には引き続き下位互換性があり、**Community**エディションで使用可能です)

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
SpreadsheetDataWriter	XLS(X)ファイル	1	0-1	いいえ	いいえ	いいえ	いいえ

概要

SpreadsheetDataWriterは、データをXLSファイルまたはXLSXファイルに書き込みます。スプレッドシートを作成するための、次の高度な機能があります。

- 挿入/上書き/追加モード
- 複雑なスプレッドシート用の強力な視覚的マッピング
  - 明示的に定義されたマッピングまたは動的自動マッピング
  - フォーム書込み
  - 複数行レコード
- 垂直/水平書込み
- セル書式設定のサポート
- パフォーマンスおよび大量データ・ロード用のストリーミング・モード
- 動的ファイル/シートのパーティション化
- テンプレートのサポート

サポートされているファイル形式は次のとおりです。

- XLS: Excel 97/2003 XLSファイルのみ(BIFF8)
- XLSX: Microsoft Excel 2007以降のオープンなドキュメント形式

サポートされる出力は次のとおりです。

- ローカルまたはリモート(FTP、HTTP、**CloverETL Server**のサンドボックスなど。SpreadsheetDataWriterの属性(p.533)の**File URL**を参照してください)
- 出力ポート
- コンソール
- ディクショナリ

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	スプレッドシートに書き込まれる受信レコード。	任意
出力	0	いいえ	ポート書込み用。出力ポートへの書込み(p.312)を参照してください。	1つのフィールド(`byte`、`cbyte`、`string`)。

SpreadsheetDataWriterの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	データの書込み先を指定します。XLSファイル、XLSXファイル、コンソール、出力ポート、ディクショナリのいずれかになります。ライターにサポートされているファイルURL形式(p.310)を参照してください。	
Sheet		書込み先のシートの名前または数値(ゼロベース)。設定しない場合、シートがデフォルト名で作成され、既存のすべてのシートの後ろに挿入されます。複数のシートをセミコロン(;)で区切って指定できます。パーティション化の詳細は、特定のユースケースに対する書込みの手法とヒント(p.542)を参照してください。	0-N
Mapping	1)	出力スプレッドシートへの入力データのマップ方法を定義するためのビジュアル・エディタ。詳細は、詳細説明(p.535)を参照してください。	
Mapping URL	1)	マッピング定義が含まれる外部ファイル。	

属性	必須	説明	可能な値
Write mode		出力スプレッドシートへのデータの書込み方法を指定します。可能な値は次のとおりです。 • **Overwrite in sheet (in-memory)**: 既存のセルがある場合は上書きします。 • Insert into sheet (in-memory): 新しいデータをマップ領域に挿入し、既存のセルがある場合は切り替えます。 • **Append to sheet (in-memory)**: 既存のデータ列/行の最後にデータを追加します。 • **Create new file (streaming: XLSX only)**: ストリーミング・モードを可能にする新規作成ファイルに既存のファイルを置き換えます。 • **Create new file (in-memory)**: 既存のファイルを置き換え、メモリー内モードで機能します。 メモリー内書込みモードでは、すべての値がメモリーに格納されるため、読取り速度が向上します。より小さいファイルに適しています。ストリーミング・モード(XLSXでのみ使用可能)では、ファイルはメモリーに格納されずに直接書き込まれます。このため、ストリーミングを使用すると、メモリーが不足することなくより大きいファイルを書き込むことができます。	「説明」を参照
Actions on existing sheets		指定した**シート**がターゲットのスプレッドシートにすでに存在する場合に実行するアクションを定義します。この属性は**書込みモード**に従って機能します。使用可能なオプションは次のとおりです。 • **Do nothing, keep existing sheets and data**: これがデフォルトのオプションです。書込み前に操作は実行されません。挿入/上書き/追加モードが適用されます。 • **Clear target sheet(s)**: 書込み前に指定の**シート**が消去されます。**書込みモード**設定は無視されます。 • **Replace all existing sheets**: ファイルに書き込む前にすべてのシートが削除されます。**書込みモード**の「`Create new file`」オプションと同等です。	「説明」を参照
Advanced			
Template File URL		出力ファイルに複製され、定義済のマッピングに従ってデータが移入されるテンプレート・スプレッドシート・ファイル。テンプレートは任意のスプレッドシートにでき、通常は、ヘッダー、フッター、データの各セクション(書込み中にレプリケートされる1つの空の行)が含まれています。より多くのヒントは、特定のユースケースに対する書込みの手法とヒント(p.542)を参照してください。出力ファイルおよびテンプレート・ファイルの形式が一致する必要があります。XLTXファイルの使用は制限されており(注意および制限(p.545)を参照)、テンプレートとしてXLTXではなくXLSXファイルを使用してください。	
Create directories		`true`に設定した場合、「**File URL**」パスに含まれている存在しないディレクトリが自動的に作成されます。	false (デフォルト) | true
Records per file		1つのファイルに書き込まれるレコードの最大数。異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	1-N

属性	必須	説明	可能な値
Number of skipped records		すべての出力ファイルにわたってスキップされるレコードの合計数。入力レコードの選択(p.305)を参照してください。	0-N
Max number of records		すべての出力ファイルにわたって書き込まれるレコードの合計数。入力レコードの選択(p.305)を参照してください。	0-N
Partition key		複数の出力ファイルへのレコードの分配を制御する値を持つキー。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition lookup table	2)	参照表のID。この表を使用して、出力ファイルに書き込まれるレコードを選択します。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition file tag		デフォルトでは、出力ファイルには番号が付けられます。この属性が`Key file tag`に設定されている場合、出力ファイルには**パーティション・キー・フィールド**または**パーティション出力フィールド**の値に基づいて名前が付けられます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	Number file tag (デフォルト) | Key file tag
Partition output fields	2)	出力ファイルに名前を付けるために使用する値が含まれる、**パーティション参照表**のフィールド。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition unassigned file name		未割当てのレコードがある場合に、それらのレコードを書き込むファイルの名前。指定されない場合、**パーティション参照表**にキー値が含まれていないデータ・レコードは破棄されます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Type of formatter		使用するフォーマッタを指定します。デフォルトでは、コンポーネントは出力ファイル拡張子`XLS`または`XLSX`に従って推測されます。	Auto (デフォルト) | XLS | XLSX

説明:

1) 2つのマッピング属性は相互に排他的です。**Mapping**にマッピングを手動で指定するか、または**Mapping URL**を介して外部ファイルを指定します。3つ目のオプションは、すべてのマッピングを空白にすることです。

2) これらの属性を両方指定するか、または両方とも指定しないでください。

詳細説明

スプレッドシート・マッピングの概要

マッピングは、Cloverレコードをスプレッドシートに書き込む方法をコンポーネントに通知します。マッピングでは、メタデータ情報、データ、書式、書込み方向などの格納先を定義します。

マッピングでは、Cloverフィールドと先頭セルの間のバインディングを定義します。そのフィールドのデータは、先頭セル位置から下向き(垂直方向。デフォルト)または右(水平)向きに、スプレッドシートに書き込まれます。

各先頭セルとフィールドのバインディングは相互に独立しています。これを使用して複雑なマッピングを作成できます(たとえば、1つのレコードを複数行にマップできます「**Rows per record**」グローバル・マッピング・プロパティを参照してください)。

各Cloverフィールドは、次のいずれかのマッピング・モードでスプレッドシート・セルにマップできます。

- **明示**: 自分で設定した固定先頭セルにフィールドを静的にマップします。通常、ライターに最も使用されるマッピング・モード(基本マッピングの例(p.537)を参照してください)。明示モードは他のマッピング・モードと組み合せることができます。

ヒント

フィールド(またはレコード全体)を明示的にマップするには、単にフィールド(レコード)をスプレッドシート・プレビュー領域にドラッグして、目的の位置にドロップします。複数のフィールドを選択できます。

- **順序別マップ**: 動的マッピング・モード。順序別モードのセルには、入力メタデータでのフィールドの出現順に左から右、上から下の方向に入力レコード・フィールドが入力されます。明示的にマップされず、名前でもマップされないフィールドのみが考慮されます。

- **名前別マップ**: このモードは、すでに存在するシートに書き込む場合にのみ適用されます。名前別でマップされたセルは、実際のコンテンツ(ヘッダーから推定可能)に従って実行時に遅延バインディングを使用して入力フィールドにバインドされます。コンポーネントは、セルの内容を入力メタデータからのフィールド名またはラベルと照合します(フィールド名とラベルと説明(p.161)を参照してください)。このような一致が見つかった場合、マップされたセルは対応する入力フィールドにバインドされます。セルの一致がない(つまり、セルの内容がフィールド名/ラベルではない)場合、セルは**解決されません**。入力フィールドを割り当てることはできません。類似したテンプレートのグループに書き込み、各テンプレートに、入力メタデータのフィールドのサブセットが含まれている場合などには、セルが未解決でも問題ありません。マッピングに未解決のセルがあっても、実行時にグラフは失敗しません。

このモードは、事前定義テンプレート(「**Template file URL**」属性)を使用して書き込む場合に役立ちます。特定のユースケースに対する書込みの手法とヒント(p.542)を参照してください。

注意

順序別マップと**名前別マップ**のいずれのモードでも、出力ファイルのコンテンツを出力メタデータに自動的にマップすることが試行されます。このため、これらのモードは、複数のファイルに書き込み、かつ、(通常は複数のテンプレート用に)単独ですべてに対応する汎用的なマッピングを1つのみ設計するときに役立ちます。入力メタデータを別のものと置き換える場合に、マッピングを変更する必要はなく、マッピング・ロジックに応じて再計算されます。

- **暗黙**: マッピングが空白である場合のデフォルトです。コンポーネントは「**Write header**」が`true`であることを前提とし、左上隅から順序に従ってすべての入力フィールドをマップします。

スプレッドシート・マッピング・エディタ

スプレッドシート・マッピング・エディタでは、マッピングとそのプロパティを定義します。マッピング・エディタには、出力ファイルのシートがある場合はプレビューが表示され、ない場合は空白のスプレッドシートが表示されます。ただし、出力ファイルやシートのグループ全体に同じマッピングが適用されます(たとえば、複数のシートまたはファイルにパーティション化する場合)。

マッピングを開始するには、書込み先のファイル(およびシート名)を「**File URL**」属性および(オプションの)「**Sheet**」属性にそれぞれ入力します。その後、**マッピング**を編集してスプレッドシート・マッピング・エディタを開きます。新しい(空の)スプレッドシートに書き込むと、マッピング・エディタが次のように空白で表示されます。

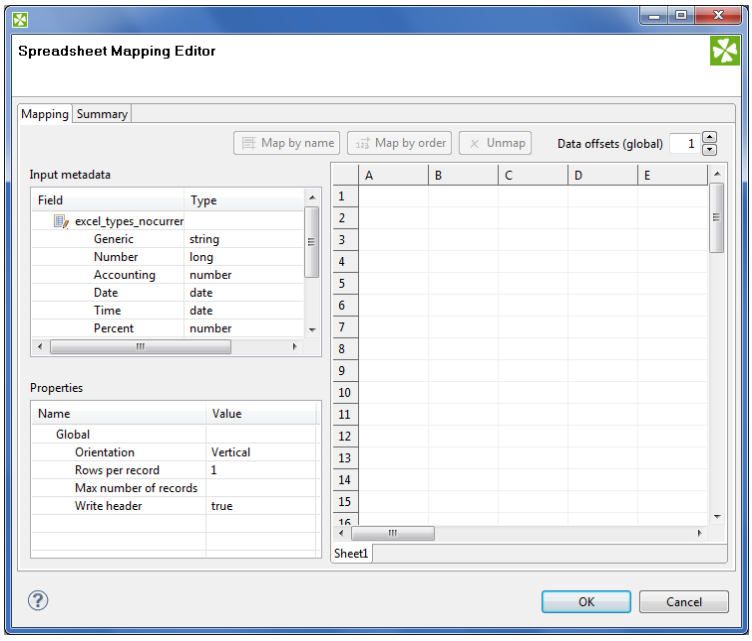


図54.16. スプレッドシート・マッピング・エディタ

エディタでは、左側の入力フィールドを右側のスプレッドシートにマップします。スプレッドシートの先頭セルを作成するには、マウスのドラッグ・アンド・ドロップか、「**Map by name**」ボタンまたは「**Map by order**」ボタンを使用します。

エディタには次の要素があります。

- ツールバー: スプレッドシート・データへのCloverフィールドの**マップ**方法(**順序別**または**名前別**)を制御するボタンおよびグローバルな**データ・オフセット**・コントロール(データ・オフセットの詳細は、詳細なマッピング・オプション(p.538)を参照してください)。
- シート・プレビュー領域: ここで出力ファイルのすべてのマッピングを作成および変更します。
- **Input metadata**: スプレッドシート・セルにマップできるCloverフィールド。これは、入力エッジに割り当てられたメタデータです。(編集**できません**。)
- **Properties**: マップされたセルおよび**グローバル**なマッピング属性のプロパティを制御します。単一のセルまたはセルのグループに一度に適用できます。
- 「**Summary**」タブ: これまでに実行したCloverとスプレッドシートとのマッピングを簡潔に確認できます。

スプレッドシート・マッピング・エディタの色

マップするかどうかとその方法を識別できるように様々な色で強調表示されたプレビュー領域のセル。

- オレンジは**先頭セル**であり、ヘッダーとなります。オレンジの各セルでプロパティを調整して複雑なマッピングを作成できます。詳細なマッピング・オプション(p.538)を参照してください。
- 破線の境界線内のセル(先頭セルの選択時にのみ表示)はデータ領域を示しています。
- 黄色のセルは、最初に書き込まれるレコードを示します。

基本マッピングの例

SpreadsheetDataWriterで行う操作の典型的な例は、空のスプレッドシートへの書込みです。この項では、数ステップで簡単にこのことを行う方法について説明します。

- 「**Mapping**」属性を編集して、**スプレッドシート・マッピング・エディタ**を開きます。
- **Input metadata**でレコード全体(次の例では`excel_types_nocurrency`)をクリックし、スプレッドシート・プレビュー領域のセルA1までドラッグし、ドロップします。入力レコードのフィールドごとに先頭セルが作成さ

れ、デフォルトの明示的マッピングが生成されます(スプレッドシート・マッピングの概要(p.535)を参照してください)。図54.17「レコード全体の明示的マッピング」(p.538)を参照してください。

- 「**Properties**」(左下隅)で、「**Write header**」が`true`に設定されていることを確認します。これにより、まず先頭セルにフィールド名(実際にはラベル)が書き込まれ、その後、実際のデータが書き込まれます。ヘッダーを出力するときは、常にこれを使用します。
- また、**Properties**では**Orientation**が**Vertical**になっています。これにより、コンポーネントは(列ごとに詳細を書き込む「**Horizontal**」方向とは反対に)行ごとに出力を生成します。
- 「**Data offsets (global)**」は1に設定されています。つまり、データは、ヘッダー・セルの領域を確保するために、先頭セルの1行下に書き込まれます。

注意

実際には、マッピングを空白のままにしても(暗黙的マッピング)同じ結果になります。その場合、先頭行は順序別にマップされます。

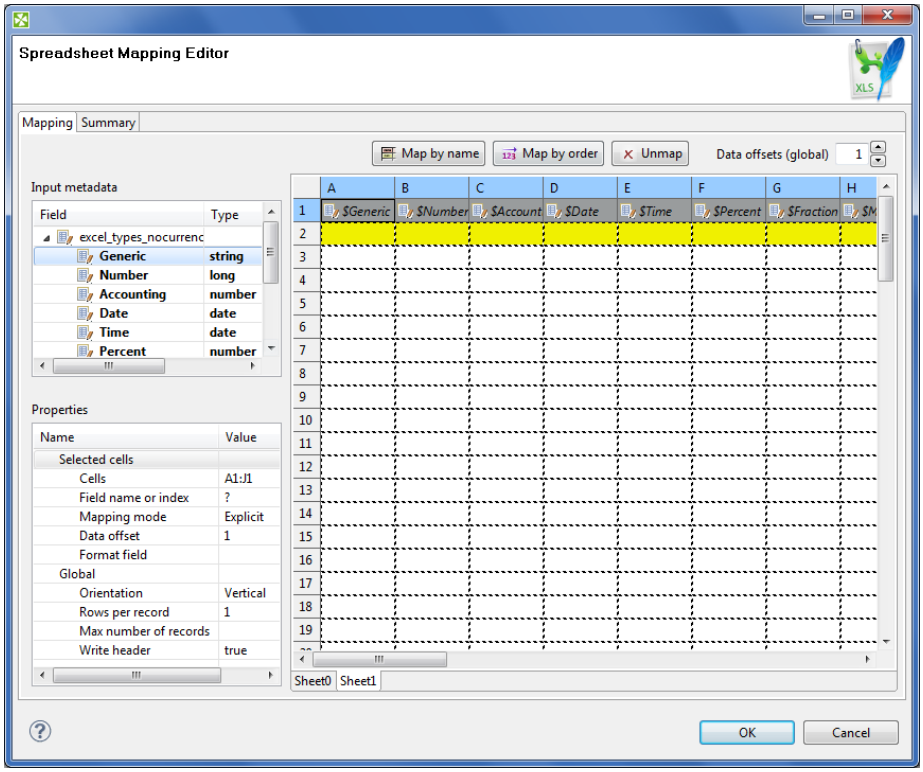


図54.17. レコード全体の明示的マッピング

詳細なマッピング・オプション

この項では、基本マッピングの例(p.537)を基にした高度な概念について説明します。

- **データ・オフセット**: 先頭セルの位置を基準にしてデータの書込み先を特定します。

  基本的に、その値は(先頭セルを基準にして)最初のレコードが書き込まれる前にスキップする行数(垂直モード)または列(水平モード)を表します。

  データ・オフセット0は何もスキップせず、データは先頭セル位置の右に書き込まれます(この設定では「**Write header**」オプションは機能しません)。

  データ・オフセット1は通常、ヘッダーが先頭セル位置に書き込まれるときに使用されます。そのため、実際のデータを1行下(または右側の列)に切り替える必要があります。

スプレッドシート全体のデータ・オフセットを調整するには、「**Data offsets (global)**」コントロールの矢印ボタンをクリックします。

また、各先頭セル(オレンジ)の「**Properties**」→「**Selected cells**」→「**Data offset**」でスピナー  を使用して、データ・オフセットをローカルに、つまり特定の列のみを調整することもできます。シート・プレビューでデータ・オフセットの変更がどのように可視化されるかを確認してください。つまり、省略した行では色が変化します。先頭セルをクリックすると表示される破線のセルに従って、レコードの書込み先を素早く確認できます。

ヒント

「**Data offsets (global)**」の矢印ボタンは、各セルのデータ・オフセット・プロパティを上下にシフトするのみです。混合オフセットが保持されているため、必要に応じてシフトします。すべてのデータ・オフセットを1つの値に設定するには、「Data offsets (global)」の数値フィールドに値を入力します。いくつかの混合オフセットが存在する場合は、値がグレーで表示されます。

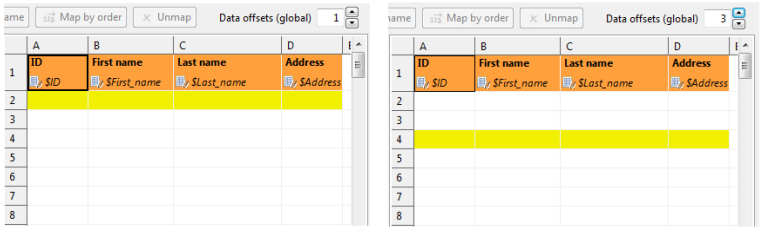


図54.18. グローバル・データ・オフセットが1 (デフォルト)の場合と3の場合の違い。右側の図では、書込みが行4で開始され、行2および3にはデータは書き込まれません。

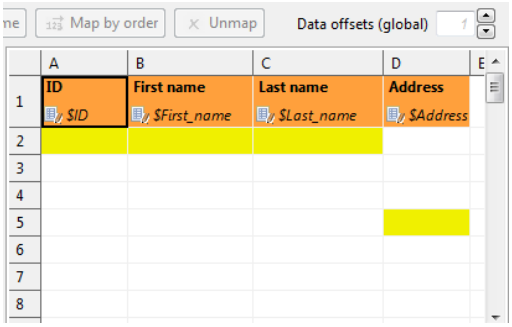


図54.19. グローバル・データ・オフセットは1に設定されています。最後の列では、ローカルに4に変更されています。出力ファイルでは、この列の最初の行は空白となり、データはD5から始まります。

- **Rows per record**: 行間のギャップを指定する**グローバル**なプロパティです。デフォルト値は1です(つまり、ギャップはありません)。複数のセルを互いの上位にマップする場合(単一レコード用)、またはデータの間に空白行を出力する必要がある場合に便利です。次の図のような場合の最適な値を推測します。

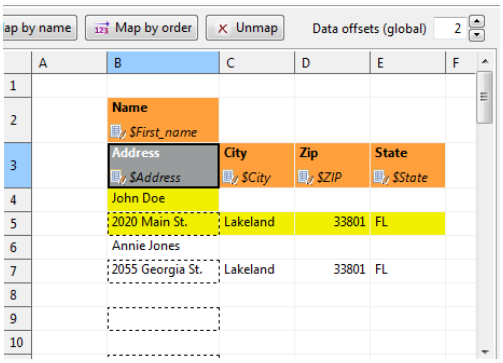


図54.20. 先頭セル「Name」および「Address」で「Rows per record」を2に設定すると、コンポーネントは常に1つのデータ行を書き込み、1行スキップし、その後再び書き込みます。このようにして、様々なデータの混在(他のデータによる上書き)を防ぎます。適切に出力するために、データ・オフセットが2に設定されていることを確認します。

- **データ・オフセット**(グローバルおよびローカル)と**「Rows per record」**の組合せ: 前の箇条書きで説明した設定を組み合せることができます。次の例を参照してください。

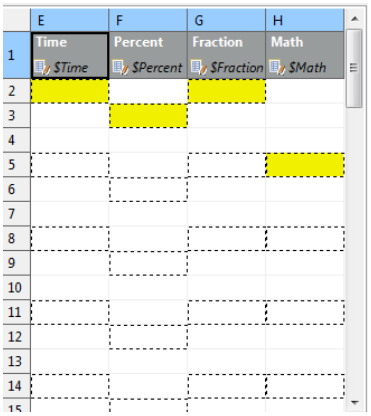


図54.21. 「Rows per record」は3に設定されています。最初の列および3つ目の列のデータはそれぞれの先頭行で始まります(それぞれのデータ・オフセットが1で始まるため)。2つ目の列および4つ目の列はそれぞれデータ・オフセットが2および4になっています。このため、出力は、ジグザグになったセル(破線部分。これに従って、この概念の理解を確認してください)で形成されます。

- **Max number of records**: これもコンポーネント属性を介して指定できる**グローバル**・プロパティです(SpreadsheetDataWriterの属性(p.533)を参照してください)。これを小さくすると、スプレッドシート・プレビューの破線のセルの数も減ります(実際に書き込まれるセルのみが強調表示されます)。

- **セルの書式設定(Format Field)**: スプレッドシートでは、単一のセルごとに独自の書式を設定できます(Excelでは、セルの右クリック→「セルの書式設定」の「表示形式」タブ)。この書式は書式文字列(Clover書式文字列ではなくExcel固有の書式文字列)で表されます。Cloverの書式はレコードごとではなくメタデータのフィールドに対してグローバルに定義されるため、Excelへの書式の書込みには注意が必要です。**SpreadsheetDataWriter**では、2つの方法でExcel固有の書式をセルに書き込むことができます。

  ケース1:

  メタデータ・フィールドの書式を指定できます(メタデータのその**「Format」**プロパティ)。つまり、シートに書き込まれたフィールドのすべての値に、指定した書式が設定されます。コンポーネントが標準の書式文字列を無視するため(CloverとExcelとの書式変換が不可能であるため)、メタデータの**「Format」**には`excel:`の接頭辞を付ける必要があります(たとえば、小数点3桁のパーセントの場合は`excel:0.000%`)。

  ケース2:

  単一のセルに2つの入力フィールドを提供します。1つはセル値を指定するためのもので、もう1つはその書式を定義するためのものです。

注意

- これは、書式が列ごとではなくセルごとに設定されるExcelの機能を最大限に引き出します。
- データの書式は追加の文字列値として渡します。
- 書式はCloverではなくExcel用語で指定されます。
- 書式(文字列)が含まれる入力フィールドを指定するには、先頭(オレンジ)セルの「**Properties**」→「**Selected cells**」の「**Format field**」を使用します。

両方が設定されている場合に、どの書式が使用されるか

- 「**Format field**」プロパティによって書式をマップしている場合。コンポーネントはそれを使用します。
- 「**Format field**」が指定されていないか、その特定のフィールドの値が空(nullまたは空の文字列)である場合。メタデータ・フィールドの**書式**を使用します(`excel:`接頭辞付きで設定されている場合)。フィールドの詳細 (p.163)も参照してください。

「**Format field**」またはメタデータの「**Format**」で`excel:General`書式を使用できます。出力は一般的な書式(Excel用語)に設定されます。

例54.9. Excel書式の書込み

`fieldValue (integer)`および`fieldFormat (string)`の2つのフィールドを(1つは値として、もう1つは「**Format field**」として)セルA1にマップするとします。受信レコードは次のとおりです。

- (100, ”#00,0”)
  - 値`100`および書式”#00,0”をセルA1に書き込みます。
- (100, “General”)
  - 値`100`をセルA1に書き込み、その書式を`General`に設定します。
- (100, ””)または(100, null)
  - 値`100`をセルA1に書き込みます。`fieldFormat`が空であるため、(`fieldFormat`ではなく)`fieldValue`の「**Format**」メタデータ属性が調べられます。
    1. 書式がない場合は、`General` を使用します。
    2. “excel:XYZ”書式文字列がある場合は、書式 XYZ をセルに適用します。
    3. (`excel:`の接頭辞が付いていない)別の書式がある場合は、General を使用します(Clover と Excel との書式変換は実行されません)。

注意

Excel書式を「**Metadata**」→「**Format**」で指定する場合は、書式文字列がExcel専用であることをCloverが認識できるように、接頭辞`excel:`を付ける必要があります。例: `"excel:0.000%"`

前述の`fieldFormat`としてExcel書式がデータに渡される場合、接頭辞は付けません。例: `"0.000%"`

`excel:`書式文字列は、**SpreadsheetDataReader**または**XLSDataReader**のスプレッドシート・リーダーで出力を読み取る場合に重要になります。一般的なリーダー(**UniversalDataReader**など)は`excel:`を完全に無視します。これは空の書式文字列とみなされます。

特定のユースケースに対する書込みの手法とヒント

- **テンプレートを使用した書込み**

  静的ヘッダーやフッターなどを含めて適切に書式設定したテンプレートを事前にExcelに準備し、Cloverを使用して自動的にデータを入力するようにした方がよい場合があります。また、テンプレートを上書きせずに再利用することもできます。

  このような場合に**SpreadsheetDataWriter**テンプレート機能が役立ちます。コンポーネントは、以前に設計されたテンプレートExcelファイルを取得し(SpreadsheetDataWriterの属性(p.533)の**Template File URL**を参照)、そのコピーを指定の出力ファイルに作成し(**File URL**を参照)、そのファイルにデータを書き込み、テンプレートの残りを保持します。

  テンプレートは任意のExcelファイルであり、通常はヘッダー、データ用の1つのテンプレート行、フッターの3つのセクションが含まれています。

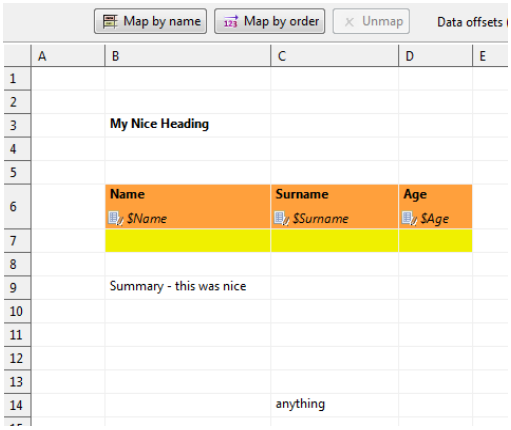


図54.22. テンプレートへの書込み。元のコンテンツは影響を受けず、データは「Name」、「Surname」、「Age」の各フィールドに書き込まれます。

テンプレート行に注意してください。これも他のものと同様に行ですが、マッピング・エディタでは、マップされたデータの先頭行として指定されます。コンポーネントは、新しいデータを書き込むたびにその行を複製します。このように、このデータ行に任意の書式や色などを割り当てることができ、書き込まれるすべての行にそれが適用されます。

テンプレート・ファイルは、それ以外の方法で変更または影響されることはありません。

重要

テンプレートを使用する場合に適正な設定は1つのみですが、他のすべてのモードも想定どおりに機能します(ただし、結果は想定どおりになりません)。設定は次のとおりです。

- **Sheet**: テンプレートからシートを選択します(新しいシートは作成しないで、番号または名前で選択します)。
- **Mapping**: これは、名前別マップが適するケースの1つです。必要に応じてテンプレートのヘッダーを使用します。通常どおりフィールドをマップできます。
- **Write mode**: `Insert`
- **Actions on existing sheets**: `Do nothing, keep existing sheets and data`

- **フォームの入力**: コンポーネントを使用して、元のボックスに影響を及ぼすことなくフォームに書き込むことができます。次の設定を使用します。

  すべてのフォーム値が含まれているコンポーネントの入力に1つの入力レコードのみを送信します。**File URL**を入力対象のフォーム・ファイルに設定します。次に、プレビュー・シートを使用して、入力フィールドを対応するフォーム・セルに明示的に1つずつマップします。

次に、次の設定を使用します。

- **Write header**: `false`
- **Data offsets (global)**: 0 (データは、マップ済の先頭セル(オレンジ)の右に書き込まれます)。

- **チャートおよび式**: 挿入、追加、上書きのいずれかのモードを使用した場合、Cloverに書き込まれたデータ領域を操作する式およびチャートはExcelで表示されるときに正しく更新されます。

注意

式、チャートやその他のExcelオブジェクトの生成は、現時点ではサポートされていません。

- **複数の書込みが1つのシートに渡されます。**

  単一シートへの複数の順次書込みを使用して、複雑なパターンを生成できます。そのためには、同じファイル/シートを書き込む複数の**SpreadsheetDataWriter**コンポーネントを設定し、それらを様々な入力にフィードします。

重要

同じファイルに書き込む複数のコンポーネントは、必ず異なるフェーズに配置します。そうしないと、グラフは書込み競合で失敗します。

  通常、シーケンス内のすべてのコンポーネントに対して`Overwrite in sheet (in-memory)`書込みモードを使用します。

- **パーティション化**: ある特定のキー・フィールド(またはより多くのフィールド)の値に従って個々のシートにパーティション化することは効果的な手法です。このため、たとえば国ごとに異なるシートにデータを書き込むことができます。単にパーティション化キーとして「`Country`」を選択します。これを行うには、「**Sheet**」属性を編集し、「**Partition data into sheets by data fields**」に切り替え、1つのフィールド(あるいは**[Ctrl]**を押しながら**クリック**か、**[Shift]**を押しながら**クリック**を使用して複数のフィールド)を選択します。

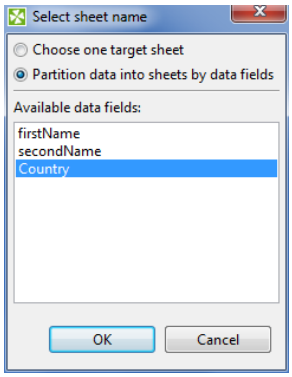


図54.23. 1つのデータ・フィールドによるパーティション化

複数のフィールドに従ってパーティション化できます。その場合、出力シート名は選択したフィールド名の複合です。**例:** 顧客の注文が1つのCSVファイルに格納されています。それらを、たとえば店や都市の名前別のシートに分けるとします。2つのフィールドに従ってパーティション化する間、**SpreadsheetDataWriter**を新規ファイル作成モードで使用します。次のようなシートを生成します。

```
Pete's Grocery,New York

Hoboken Deli,New Jersey

Al's Hardware,New York
```

これらのそれぞれに1つの店のデータのみが含まれています。

異なる出力ファイルへの出力のパーティション(p.318)も参照してください。

- **大規模ファイルの書込み**

  Excel書式は主として大規模データ・ロード向けに設計されたものではありませんが、高いメモリー要件にあわせて処理を容易に拡大できます。

  書式自体に、いくつかの制限があります。

  - Excel 97/2003: XLS
    - シート当たり最大65,535行および256列
    - シートの最大数: 255
  - Excel 2007以降: XLSX
    - 最大行数: 無制限(ただし、Excel自体ではファイルに含まれる最初の1,048,576行のみが処理されます)。すべてのデータは、**SpreadsheetDataReader**または大規模ファイルをサポートするその他のツールでリード・バックされます。
    - 最大列数: 16,384
    - 最大シート数: 無制限(メモリーがあるかぎり)

ヒント

大規模スプレッドシートの操作ではメモリー消費が大きいため、コンポーネントではそのメモリー・フットプリントを最大限最適化するものの、次のヒントを考慮してください。

- スプレッドシート・マッピング・エディタでマップするときは、Designerのメモリー消費が一時的に1GBを超えるメモリーにランプアップする場合があるため、Designer自体に対して十分なヒープ領域を設定してください(プログラムとVM引数(p.85)を参照してください)。
- メモリー消費はExcel内部でのファイル編成方法の影響を受けるため、同じデータ量の2つのファイルでもメモリー要件が大幅に異なる場合があります。
- 可能な場合は、常にストリーミング・モードを使用してください。ストリーミング・モードがオンかオフかを検出するには、グラフの**Run Configurations**で`DEBUG`モードに切り替えます。方法は、プログラムとVM引数(p.85)を参照してください。

通常、新規ファイル作成 (ストリーミング： `XLSX`のみ) 書込みモードを使用します。その他の書込みモードはストリーミングをサポートしていません。

- **マッピングの確認**

  数多くのメタデータ・フィールドのある複雑なマッピングでは、すべてのマッピングが正しいことを確認することをお薦めします。**スプレッドシート・マッピング・エディタ**での作業中、いつでも、「**Summary**」タブに切り替えて次のような先頭セルおよびマッピングの概要を確認できます。

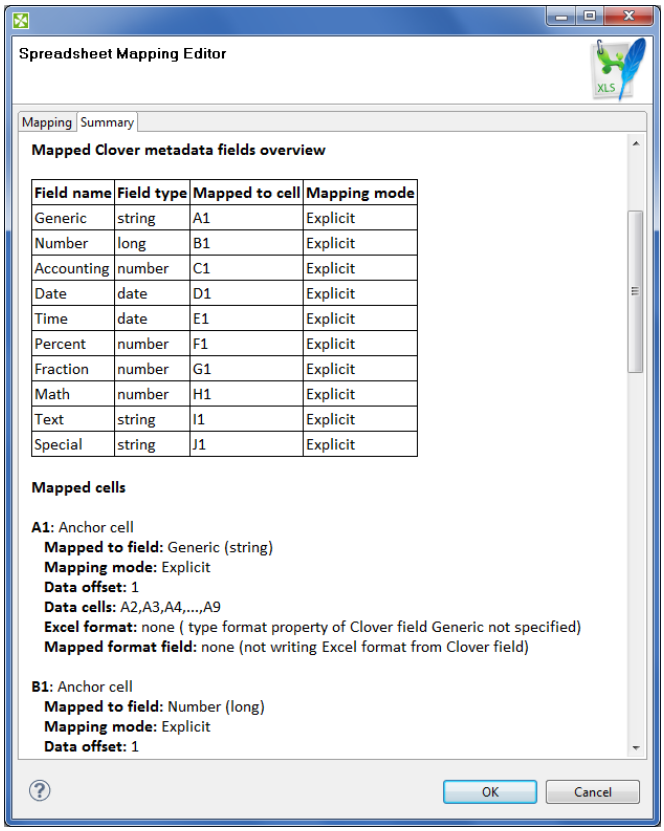


図54.24. マッピングの概要

注意および制限

- **暗号化**: 暗号化されたXLSファイルまたはXLSXファイルの書込みはサポートされていません(暗号化ファイルを読み取ることができるSpreadsheetDataReader(p.404)とは異なります)。
- **XLTXとXLSXのテンプレート**: 技術的な理由から、現時点でXLSX出力にXLTXテンプレートは使用できません。それでも、XLTXファイルとXLSXファイルとの違いは最小限に抑えられています。このため、テンプレートおよび出力ファイルの両方の書式としてXLSXを使用することをお薦めします。XLSファイルおよびXLTファイルには、このような制限がありません。
- **サーバー・ファイルでのマッピング・エディタ**: **File URL**にワイルドカード文字が含まれている場合を除き、サーバー・ファイルでのスプレッドシート・マッピング・エディタは通常どおり動作します。その場合、CloverETL Designerは一致するサーバー・ファイルを見つけることができず、マッピング・エディタのスプレッドシート・プレビューにはデータが表示されません。このことは今後のリリースで修正される予定です。
- **エラー・レポート**: コンポーネントにはエラー・ポートがありません。仕様により、コンポーネント構成は有効であり、ファイルへのレコードの書込みは正常に完了するか、または致命的なエラー(無効な構成やデバイスの空き領域なしなど)で失敗します。入力レコードごとにエラーは生成されません。
- **列の幅**: **SpreadsheetDataWriter**は、新規作成のシートまたはまず消去される既存のシート(つまり、「**Actions on existing sheets**」が「**Clear target sheet(s)**」に設定されている)に書き込むとき、各特定の列で最も幅の広いセルの内容の幅に一致するように列の幅を自動的に調整します。テンプレートが使用される場合または既存の(最初に消去されない)シートに書き込むときは、列幅は調整されません。つまり、テンプレートからの列幅が保持されます。また、すでに存在するシートの列幅は、そのシートのデータを追加/挿入/上書きするときにも保持されます。

StructuredDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

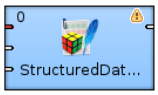
StructuredDataWriterは、ユーザー定義構造のファイルにデータを書き込みます。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
StructuredDataWriter	構造化フラット・ファイル	1-3	0-1	いいえ	いいえ	いいえ	いいえ

概要

StructuredDataWriterは、ユーザー定義構造のファイル(ローカルまたはリモート、デリミタ付き、固定長または混合)にデータを書き込みます。出力ファイルの圧縮、およびコンソール、出力ポートまたはディクショナリへのデータの書込みも実行できます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	ボディ用のレコード。	任意
	1	いいえ	ヘッダー用のレコード。	任意
	2	いいえ	フッター用のレコード。	任意
出力	0	いいえ	ポート書込み用。出力ポートへの書込み(p.312)を参照してください。	1つのフィールド(`byte`、`cbyte`、`string`)。

StructuredDataWriterの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	受信したデータの書込み先(フラット・ファイル、コンソール、出力ポート、ディクショナリ)を指定する属性。ライターにサポートされているファイルURL形式(p310.)を参照してください。	
Charset		出力に書き込まれるレコードのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Append		デフォルトでは、新しいレコードで古いレコードが上書きされます。`true`に設定された場合、出力ファイルに保存されている古いレコードに新しいレコードが追加されます。	false (デフォルト) | true
Body mask		出力ファイルのボディの書込みに使用されるマスク。最初の入力ポートを介して受信したレコードに基づくことができます。**ボディ・マスク**およびその結果生成される出力構造の定義の詳細は、マスクおよび出力ファイル構造(p.548)を参照してください。	Default Body Structure (p.549) (デフォルト) | ユーザー定義
Header mask	1)	出力ファイルのヘッダーの書込みに使用されるマスク。2番目の入力ポートを介して受信したレコードに基づくことができます。**ヘッダー・マスク**およびその結果生成される出力構造の定義の詳細は、マスクおよび出力ファイル構造(p.548)を参照してください。	empty (デフォルト) | user-defined
Footer mask	2)	出力ファイルのフッターの書込みに使用されるマスク。3番目の入力ポートを介して受信したレコードに基づくことができます。**フッター・マスク**およびその結果生成される出力構造の定義の詳細は、マスクおよび出力ファイル構造(p.548)を参照してください。	empty (デフォルト) | user-defined
Advanced			
Create directories		デフォルトでは、存在しないディレクトリは作成されません。`true`に設定すると、これらが作成されます。	false (デフォルト) | true
Records per file		1つの出力ファイルに書き込まれるレコードの最大数。	1-N
Bytes per file		1つの出力ファイルの最大サイズ(バイト単位)。	1-N
Number of skipped records		スキップするレコード数。出力レコードの選択(p.317)を参照してください。	0-N
Max number of records		すべての出力ファイルに書き込まれるレコードの最大数。出力レコードの選択(p.317)を参照してください。	0-N
Partition key		複数の出力ファイルへのレコードの分配を定義する値を持つキー。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition lookup table	1)	出力ファイルに書き込まれるレコードを選択するために使用する参照表のID。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition file tag		デフォルトでは、出力ファイルには番号が付けられます。これが`Key file tag`に設定された場合、出力ファイルには**パーティション・キー・フィールド**または**パーティション出力フィールド**の値に基づく名前が付けられます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	Number file tag (デフォルト) | Key file tag

属性	必須	説明	可能な値
Partition output fields	1)	出力ファイルに名前を付けるために使用する値が含まれる、**パーティション参照表**のフィールド。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition unassigned file name		未割当てのレコードがある場合に、それらのレコードを書き込むファイルの名前。指定されない場合、**パーティション参照表**にキー値が含まれていないデータ・レコードは破棄されます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	

説明:

1) 2番目の入力ポートが接続されている場合は指定する必要があります。ただし、入力データ・レコードに基づく必要はありません。

2) 3番目の入力ポートが接続されている場合は指定する必要があります。ただし、入力データ・レコードに基づく必要はありません。

詳細説明

マスクおよび出力ファイル構造

- **出力ファイル構造**

  1. 出力ファイルではヘッダー、ボディおよびフッターがこの順序で構成されています。
  2. いずれも、対応するマスクを指定することで定義します。
  3. マスクの定義後、マスクのコンテンツが繰り返し書き込まれ、受信したレコードごとに1つのマスクが書き込まれます。
  4. ただし、**「Records per file」**属性が定義されている場合、出力構造は様々な出力ファイルに分散されますが、この属性は**ボディ・マスク**に対してのみ適用されます。ヘッダーおよびフッターはすべての出力ファイルで同じです。

- **マスクの定義**

  ボディ・マスク、**ヘッダー・マスク**および**フッター・マスク**は、**マスク・**ダイアログで定義できます。このダイアログは、対応する属性行をクリックすると開きます。そのウィンドウには、**「Metadata」**ペインと**「Mask」**ペインが表示されます。最下部に**「Auto XML」**ボタンがあります。

  フィールド値なしまたはフィールド値ありのマスクを定義できます。

  フィールド値は、ドル記号に続けてフィールド名を指定することで表されます。

  「Auto XML」ボタンをクリックすると、簡単なXML構造が**「Mask」**ペインに表示されます。

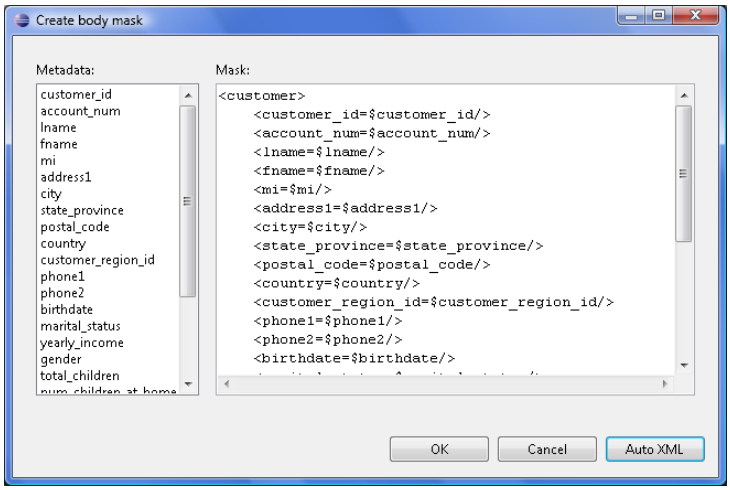


図54.25. マスク作成のダイアログ

出力ファイルに保存しないフィールドの削除のみが必要で、一致の左側に提案された名前を変更することもできます。これらには、<sometag=$metadatafield/>のような一致の書式があります。デフォルトでは、「**Auto XML**」ボタンをクリックすると、<metadatafield=$metadatafield/>のような式が含まれているXML構造が取得されます。これらの一致の左側は他の任意のものに置き換えることができますが、右側は同じままにする必要があります。一致の右側にドル記号に続けて指定するフィールド名は変更しないでください。これらはフィールドの値を表します。

マスクとして任意のXMLファイルを使用する必要はありません。マスクはその他の任意の構造にできます。

- **デフォルト・マスク**

1. デフォルト・**ヘッダー・マスク**は空です。ただし、2番目の入力ポートが接続されている場合は定義する必要があります。

2. デフォルト・**フッター・マスク**は空です。ただし、3番目の入力ポートが接続されている場合は定義する必要があります。

3. デフォルト・**ボディ・マスク**は空です。ただし、生成されるデフォルト・ボディ構造は次のようになります。

   ```
   < recordName field1name=field1value field2name=field2value ...
   fieldNname=fieldNvalue />
   ```

   この構造は、すべてのレコードの出力ファイルに書き込まれます。

   Records per fileが設定されている場合は、最大でも指定した数のレコードのみが各出力ファイルのボディに使用されます。

Trash

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

Trashではデータを破棄します。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
Trash	なし	1-n	0	いいえ	いいえ	いいえ	いいえ

概要

Trashではデータを破棄します。デバッグ目的で、そのデータをファイル(ローカルまたはリモート)またはコンソールに書き込むことができます。グラフの可読性を高めるために複数の入力を接続できます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	1-n	はい	受信したデータ・レコード用	任意

Trashの属性

属性	必須	説明	可能な値
Basic			
Debug print		デフォルトでは、すべてのレコードが破棄されます。`true`に設定した場合、すべてのレコードがデバッグ・ファイル(指定した場合)またはコンソールに書き込まれます。**ログ・レベル**をデフォルト値(`INFO`)から変更する必要はありません。このモード	false (デフォルト) | true

属性	必須	説明	可能な値
		は、単一の入力ポートが接続されているときにのみサポートされます。	
Debug file URL		デバッグ出力ファイルを指定する属性。ライターにサポートされているファイルURL形式(p.310)を参照してください。パスを指定していない場合、ファイルは`${PROJECT}`ディレクトリに保存されます。**ログ・レベル**をデフォルト値(`INFO`)から変更する必要はありません。	
Debug append		デフォルトでは、新しいレコードで古いレコードが上書きされます。`true`に設定された場合、出力ファイルに保存されている古いレコードに新しいレコードが追加されます。	false (デフォルト) | true
Charset		デバッグ出力のエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Advanced			
Print trash ID		デフォルトでは、TrashのIDはデバッグ出力に書き込まれません。`true`に設定した場合、**Trash**のIDはデバッグ・ファイルまたはコンソールに書き込まれます。**ログ・レベル**をデフォルト値(`INFO`)から変更する必要はありません。	false (デフォルト) | true
Create directories		デフォルトでは、存在しないディレクトリは作成されません。`true`に設定すると、これらが作成されます。	false (デフォルト) | true
Mode		Trashは、PerformaceモードまたはValidate recordsモードで実行できます。PerformanceモードではRAWデータが破棄され、Validate recordsではTrashはライターをシミュレートして入力のデシリアライズを試行します。	Performance (デフォルト) | Validate records

UniversalDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

UniversalDataWriterは、フラット・ファイルにデータを書き込む最後のコンポーネントです。

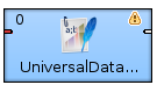
コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
UniversalDataWriter	フラット・ファイル	1	0-1	✖	✖	✖	✖

概要

UniversalDataWriterは、入力ポートからのすべてのレコードにデリミタ付き、固定長または混合の書式を設定し、それらをCSV (カンマ区切り値)またはテキスト・ファイルなど指定のフラット・ファイルに書き込みます。出力データをローカルに格納したり、リモート転送プロトコルを介してアップロードできます。また、ZIPアーカイブおよびTARアーカイブの書込みがサポートされています。

コンポーネントは、単一のファイルまたはパーティション化されたファイルのコレクションを書き込むことができます。書式のタイプは、入力ポート・データ・フローのメタデータに指定されます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	受信したデータ・レコード用。	任意
出力	0	✖	ポート書込み用。出力ポートへの書込み(p.312)を参照してください。	特定のbyte/cbyte/stringのフィールドを含みます。

UniversalDataWriterの属性

属性	必須	説明	可能な値
Basic			
File URL	✔	受信データの書込み先(フラット・ファイル、コンソール、出力ポート、ディクショナリ)を指定します。ライターにサポートされているファイルURL形式(p.310)を参照してください。	
Charset		出力に書き込まれるレコードの文字エンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Append		レコードが既存の空ではないファイルに出力される場合、デフォルト(`false`)では古いレコードが置き換えられます。`true`に設定した場合、新しいレコードは既存の出力ファイルの内容の最後に追加されます。	false (デフォルト) | true
Quoted strings		`true`に切り替えると、(`byte`および`cbyte`を除いて)すべてのフィールド値が引用符で囲まれます。この属性を設定しない場合、その値は入力ポートのメタデータから継承されます(およびグレーにフェードしたテキストで表示されます。レコード詳細(p.162)も参照してください)。	false | true
Quote character		出力フィールドを囲む引用符の種類を指定します。**Quoted strings**が`true`の場合にのみ適用されます。デフォルトでは、この属性の値は入力ポート上のメタデータから継承されます。レコード詳細(p.162)も参照してください。	" | '
Advanced			
Create directories		`true`に設定した場合、「**File URL**」属性パスに存在しないディレクトリは作成されます。	false (デフォルト) | true
Write field names		フィールド・ラベルはデフォルトでは出力ファイルに書き込まれません。`true`に設定した場合、個々のフィールドのラベルが出力に出力されます。フィールド・ラベルはフィールド名とは異なります。ラベルは重複可能で、任意の文字(たとえば、アクセントや発音区別文字)を使用できます。「Record」ペイン(p.160)を参照してください。	false (デフォルト) | true
Records per file		各出力ファイルに書き込まれるレコードの最大数。指定した場合、ドル記号$ (桁数プレースホルダ)をファイル名マスクに含める必要があります。ライターにサポートされているファイルURL形式(p.310)を参照してください。	1 - N
Bytes per file		各出力ファイルの最大サイズ(バイト単位)。指定した場合、ドル記号$ (桁数プレースホルダ)をファイル名マスクに含める必要があります。ライターにサポートされているファイルURL形式(p.310)を参照してください。レコードが2つのファイルに分割されないように、最大サイズを多少高く設定できます。	1 - N
Number of skipped records		出力ファイルに最初のレコードを書き込む前にスキップするレコード/行数。出力レコードの選択(p.317)を参照してください。	0 (デフォルト) - N
Max number of records		すべての出力ファイルに書き込むレコード/行数。出力レコードの選択(p.317)を参照してください。	0-N

属性	必須	説明	可能な値
Exclude fields		セミコロンで区切られた、出力に書き込まれないフィールド名のシーケンス。同じフィールドが「**Partition key**」の一部となる場合に使用できます。	
Partition key	[2]	様々な出力ファイルへのレコードの分配を定義する、セミコロンで区切られたフィールド名のシーケンス。同じ**パーティション・キー**を持つレコードは同じ出力ファイルに書き込まれます。選択した**パーティション・ファイル・タグ**に従って、ファイル名マスクに適切なプレースホルダ($または#)を使用してください。異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition lookup table	[1]	出力ファイルに書き込まれるレコードを選択するために使用する参照表のID。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition file tag	[2] [2]	デフォルトでは、出力ファイルには番号が付けられます。これがKey file tagに設定された場合、出力ファイルには**パーティション・キー・フィールド**または**パーティション出力フィールド**の値に基づく名前が付けられます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	Number file tag (デフォルト) | Key file tag
Partition output fields	[1] [1]	出力ファイルに名前を付けるために使用する値が含まれる、**パーティション参照表**のフィールド。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition unassigned file name		未割当てのレコードがある場合に、それらのレコードを書き込むファイルの名前。指定されない場合、**パーティション参照表**にキー値が含まれていないデータ・レコードは破棄されます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	

[2]これらの属性を両方指定するか、または両方とも指定しないでください。

[1]これらの属性を両方指定するか、または両方とも指定しないでください。

ヒントおよびポイント

- フィールド・サイズ制限1: **UniversalDataWriter**は、最大4KBのサイズのフィールドを書き込むことができます。より大きいフィールドをファイルに書き込むことができるようにするには、「DataFormatter.FIELD_BUFFER_LENGTH」プロパティを増やします。デフォルトのCloverETL設定の変更(p.88)を参照してください。このバッファを大きくしても、グラフ・メモリー消費は大きく増えません。

- フィールド・サイズ制限2: 書き込まれるフィールドが大きいという問題を解決する別の方法としては、大きなフィールドをいくつかのレコードに分割できるNormalizer(p.612)コンポーネントの利用があります。

XLSDataWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

XLSDataWriterは、データをXLSファイルまたはXLSXファイルに書き込みます。

重要

Clover 3.3.以降、スプレッドシートの読取り/書込みに使用可能な新しい強力なコンポーネントがあります。SpreadsheetDataReader(p.404)およびSpreadsheetDataWriter(p.532)を参照してください。ただし、前のXLSコンポーネント(XLSDataReader(p.421)およびXLSDataWriter(p.555))には引き続き互換性があります。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
XLSDataWriter	XLS(X)ファイル	1	0-1	いいえ	いいえ	いいえ	いいえ

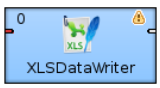
概要

XLSDataWriterは、データをXLSファイルまたはXLSXファイルに書き込みます(ローカルまたはリモート)。出力ファイルの圧縮、およびコンソール、出力ポートまたはディクショナリへのデータの書込みも実行できます。

注意

XLSDataWriterはメモリー要件が高く、メモリーにデータを格納する場合があります(「**Disable temporary files (inMemory mode)**」属性を参照してください)。XLSXファイルを操作するときは、すべてのデータがメモリーに格納されます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	受信したデータ・レコード用	任意
出力	0	いいえ	ポート書込み用。出力ポートへの書込み(p.312)を参照してください。	1つのフィールド(`byte`、`cbyte`、`string`)。

XLSDataWriterの属性

属性	必須	説明	可能な値
Basic			
Type of parser		使用するフォーマッタを指定します。デフォルトでは、拡張子(XLSまたはXLSX)に従ってコンポーネントが推測されます。	Auto (デフォルト) | XLS | XLSX
File URL	はい	受信したデータの書込み先(XLSファイルまたはXLSXファイル、コンソール、出力ポート、ディクショナリ)を指定する属性。ライターにサポートされているファイルURL形式(p.310)を参照してください。	
Sheet name	1)	レコードが書き込まれるシートの名前。設定しないと、シートがデフォルト名で作成され、すべてのシートの最後に挿入されます。フィールド名のシーケンスにし、それぞれの名前にドル記号の接頭辞を付けてセミコロン、コロンまたはパイプで区切ることもできます。このようにして、該当シーケンスの値ごとに異なるシートが作成されます。たとえば、`$Country;$City`などです。この例では、国と都市の組合せごとに異なるシートが作成されます。シートはレコード値の順に作成されます。このため、まずこのフィールドでレコードをソートし、その後にのみ、出力XLS(X)ファイルにそのレコードを書き込む必要があります。	Sheet[0-9]+ (名前の数値は、同じ名前構造Sheet[0-9]+を持つ最後のシートの番号です。)
Sheet number	1)	レコードが書き込まれるシートの番号。設定しないと、シートがデフォルト名で作成され、すべてのシートの最後に挿入されます。	0-N
Charset		出力に書き込まれるレコードのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Append to the sheet		デフォルトでは、1つのシートで古いレコードが上書きされます。`true`に設定された場合、シートに保存されている古いレコードに新しいレコードが追加されます。	false (デフォルト) | true
Metadata row		フィールド名が書き込まれる行の番号。デフォルトでは、フィールド名はシートのヘッダーに書き込まれます。	0 (デフォルト) | 1-N
Advanced			
Create directories		デフォルトでは、存在しないディレクトリは作成されません。`true`に設定すると、これらが作成されます。	false (デフォルト) | true
Disable temporary files (inMemory mode)		書込み中に作成される一時ファイルは、デフォルトではディスクに格納されます。この属性を`true`に設定した場合、これらのファイルはメモリーに格納されます。注意: これは`xls`ファイルにのみ適用できます。	false

属性	必須	説明	可能な値
Start row		レコードの書込みが始まるシートの行。デフォルトでは、レコードは先頭行から始まるシートに書き込まれます。	1 (デフォルト) | 2-N
Start column		レコードの書込みが始まるシートの列。デフォルトでは、レコードは先頭列から始まるシートに書き込まれます。	A (デフォルト) | B-*
Records per file		1つの出力ファイルに書き込まれるレコードの最大数。	1-N
Number of skipped records		スキップするレコード数。出力レコードの選択(p.317)を参照してください。	0-N
Max number of records		すべての出力ファイルに書き込まれるレコードの最大数。出力レコードの選択(p.317)を参照してください。	0-N
Exclude fields		出力に書き込まれない、セミコロンで区切られたフィールド名のシーケンス。同じフィールドが**パーティション・キー**の一部となる場合またはフィールドが前述の**Sheet name**として選択されている場合に使用できます。	
Partition key		複数の出力ファイルへのレコードの分配を定義する値を持つキー。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition lookup table	2)	出力ファイルに書き込まれるレコードを選択するために使用する参照表のID。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition file tag		デフォルトでは、出力ファイルには番号が付けられます。これが`Key file tag`に設定された場合、出力ファイルには**パーティション・キー・フィールド**または**パーティション出力フィールド**の値に基づく名前が付けられます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	Number file tag (デフォルト) | Key file tag
Partition output fields	2)	出力ファイルに名前を付けるために使用する値が含まれる、**パーティション参照表**のフィールド。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition unassigned file name		未割当てのレコードがある場合に、それらのレコードを書き込むファイルの名前。指定されない場合、**パーティション参照表**にキー値が含まれていないデータ・レコードは破棄されます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	

説明:

1) これらのいずれかの属性を指定できます。「**Sheet name**」の方が優先されます。出力ファイルの作成前に、**Sheet name**のみを設定できます。これらのいずれも指定されていない場合、グラフ実行のたびに新しいシートが作成されます。

2) これらの属性を両方指定するか、または両方とも指定しないでください。

重要

同じファイルの複数のシートにデータを書き込む場合は、各シートに別のフェーズでデータを書き込む必要があります。

XMLWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第44章「ライターの共通プロパティ」(p.309)

目的に適した**ライター**を見つけるには、ライターの比較(p.310)を参照してください。

要約

XMLWriterは、レコードをXMLファイルにフォーマットします。

コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
XMLWriter	XMLファイル	1-n	0-1	いいえ	いいえ	いいえ	いいえ

概要

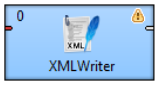
XMLWriterは、入力データ・レコードを受信して結合し、ユーザー定義のXML構造にフォーマットします。複雑なマッピングも可能であるため、コンポーネントでは任意のネストされたXML構造を作成できます。

XMLWriterは、XML構造の複雑さに応じて、ストリーム・データ処理とキャッシュ・データ処理を組み合せます。これにより、ほとんどの場合、任意のサイズのXMLファイルを生成できます。ただし、出力を複数のチャンクにパーティション化できます。つまり、大きいために処理が困難なXMLファイルを簡単に複数の小さいチャンクに分割できます。

標準の出力オプション(ファイル、圧縮ファイル、コンソール、出力ポートまたはディクショナリ)を使用できます。

このコンポーネントの実行には、Eclipseバージョン3.6以上が必要です。

アイコン



ポート

ポート・タイプ	ポート番号	必須	説明	メタデータ
入力	0-N	1つ以上	結合されXMLファイルにマップされる入力レコード。	任意(ポートごとに異なるメタデータを保持できます)。

ポート・タイプ	ポート番号	必須	説明	メタデータ
出力	0	いいえ	ポートへの書込みの詳細は、出力ポートへの書込み(p.312)を参照してください。	1つのフィールド(byte、cbyte、string)。

XMLWriterの属性

属性	必須	説明	可能な値
Basic			
File URL	はい	出力XML用のターゲット・ファイル。ライターにサポートされているファイルURL形式(p.310)を参照してください。	
Charset		XMLWriterにより生成された出力ファイルのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Mapping	1)	入力データが出力XMLにマップされる方法の定義。詳細は、詳細説明(p.560)を参照してください。	
Mapping URL	1)	マッピング定義が含まれる外部テキスト・ファイル。マッピング・ファイル形式は、マッピングの作成: ポートとフィールドのマッピング(p.569)およびマッピングの作成:「Source」タブ(p.572)を参照してください。単一のマッピングを複数のグラフで共有する場合は、マッピングを外部ファイルに置きます。	
XML Schema		XSDスキーマへのパス。**XMLスキーマ**が設定されている場合は、マッピング全体がスキーマから自動的に事前生成されます。その方法は、マッピングの作成: 既存のXSDスキーマの使用(p.572)を参照してください。スキーマは、metaフォルダに配置される必要があります。	none (デフォルト) | 任意の有効なXSDスキーマ
Advanced			
Create directories		trueの場合、**File URL**パスに含まれている既存のディレクトリは自動的に作成されません。	false (デフォルト) | true
Omit new lines wherever possible		デフォルトでは、各要素は個別の行に書き込まれます。trueに設定すると、データを出力XML構造に書き込むときに改行が省略されます。このため、すべてのXMLタグが1行上にのみ存在します。	false (デフォルト) | true
Cache size		ポートから要素へのデータのキャッシュ時(データは最初に処理されてから書き込まれます)に使用されるデータベースのサイズ。データが大きくなるほど、高速な処理を維持するためにより大規模なキャッシュが必要となります。	デフォルト: auto | 300MB、1GBなど。
Sorted input		入力データがソートされているかどうかをXMLWriterに通知します。この属性をtrueに設定することで、「**Sort keys**」(次を参照)で定義したソート順序を使用することを宣言します。	false (デフォルト) | true
Sort keys		入力データのソート方法をXMLWriterに通知することによってストリーミングを可能にします(マッピングの作成: ポートとフィールドのマッピング(p.569)を参照してください)。フィールドのソート順序は、個別タブのポートごとに指定できます。ソート・キーの操作については、ソート・キー(p.277)で説明しています。	

属性	必須	説明	可能な値
Records per file		1つのファイルに書き込まれるレコードの最大数。異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	1-N
Max number of records		すべての出力ファイルに書き込まれるレコードの最大数。出力レコードの選択(p.317)を参照してください。	0-N
Partition key		複数の出力ファイルへのレコードの分配を制御する値を持つキー。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition lookup table		参照表のID。この表を使用して、出力ファイルに書き込まれるレコードを選択します。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition file tag		デフォルトでは、出力ファイルには番号が付けられます。この属性がKey file tagに設定されている場合、出力ファイルには**「Partition key」フィールド**または**「Partition output fields」フィールド**の値に基づいて名前が付けられます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	Number file tag (デフォルト) | Key file tag
Partition output fields		出力ファイルに名前を付けるために使用する値が含まれる、**パーティション参照表**のフィールド。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	
Partition unassigned file name		未割当てのレコードがある場合に、それらのレコードを書き込むファイルの名前。指定されない場合、**パーティション参照表**にキー値が含まれていないデータ・レコードは破棄されます。詳細は、異なる出力ファイルへの出力のパーティション(p.318)を参照してください。	

説明:

1) これらの属性のいずれかを指定する必要があります。両方が定義された場合は、**「Mapping URL」**が優先されます。

詳細説明

XMLWriterのコア部分はマッピング・エディタであり、これを使用して、入力データ・レコードをXMLツリー構造に視覚的にマップできます(図54.26「マッピング・エディタ」(p.561)を参照してください)。入力ポートまたはフィールドをマップ対象のXML要素および属性にドラッグすることによって、XML構造にデータを効率的に移入します。

さらに、このエディタでは、出力XMLファイルをテキストとして視覚的に編集できるマッピング・ソースに直接アクセスできます。特殊なディレクティブを使用して、そこにあるCloverETLデータをXMLに移入します(図54.34「マッピング・エディタの「Source」タブ」(p.572)を参照してください)。

XML構造はXSDスキーマとして指定するか(**「XML Schema」**属性を参照)、または構造を最初から手動で定義できます。

「Mapping」属性の「...」ボタンをクリックして、視覚的なマッピング・エディタにアクセスできます。

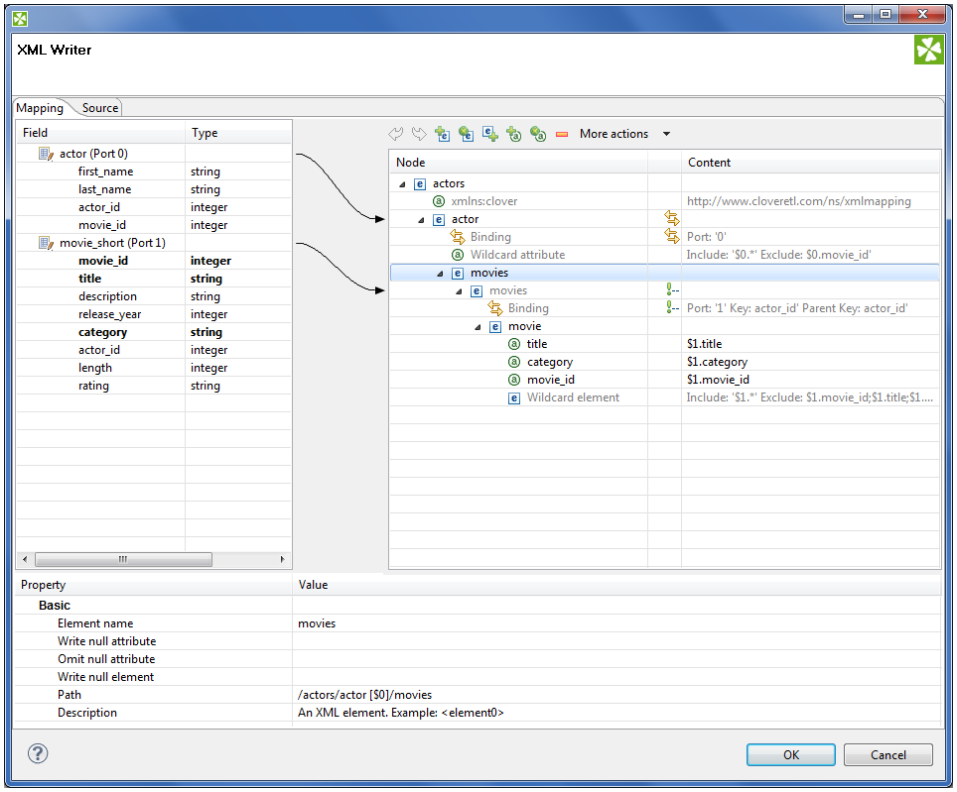


図54.26. マッピング・エディタ

エディタ内には、ウィンドウの左上隅に次の2つのメイン・タブが表示されます。

- **Mapping**: 視覚的環境で出力XMLを設計できます。
- **Source**: XMLマッピングのソース・コードを直接編集できます。

「Mapping」タブでの変更内容は「Source」タブにすぐに反映され、その逆も同じです。つまり、両方のエディタ・タブで同じ変更を加えることができます。

「**Mapping**」タブに切り替えると、ウィンドウの次の3つの基本部分が表示されます。

1. 「**Field**」列と「**Type**」列が含まれる左側の部分: 入力データのポートを表します。ポートは、「**Field**」列にシンボリック名で表されます。シンボリック名以外に、ポートには$0 (リスト内の最初のポート用)から始まる番号が付けられます。各ポートの下には、そのすべてのフィールドとそれらのデータ型のリストがあります。ポート名、フィールド名、およびそれらのデータ型はいずれもこのセクションでは編集できません。これらはすべて、XMLWriterの入力エッジのメタデータにのみ依存します。

2. 「**Node**」列と「**Content**」列が含まれる右側部分: 出力要素、属性、ワイルドカード要素またはワイルドカード属性と名前空間を定義します。このセクションでは、「**Node**」列または「**Content**」列のセルをダブルクリックして、データを変更できます。もう1つのオプションとして、行をクリックし、ウィンドウの下部セクションでその**プロパティ**を参照します。

3. 「**Property**」列と「**Value**」列が含まれる下の部分: 選択した各ノードのプロパティが表示され、変更はここで加えます。

マッピングの作成: 新しいXML構造の設計

マッピング・エディタでは、完全に空白のマッピングから開始できます(最初に出力XML構造を設計し、次に入力データをその構造にマッピングします)。もう1つのオプションは、独自のXSDスキーマを使用することです。マッピングの作成: 既存のXSDスキーマの使用(p.572)を参照してください。

空白のマッピング・エディタに入力すると、入力ポートが左側に表示され、ルート要素が右側に表示されます。マッピングのポイントは、最初に右側で出力XML構造を設計することです(データ出力先)。2番目に、左側のポート・フィールド(データ・ソース)をこれらの事前準備済XMLノードに接続する必要があります(マッピングの作成: ポートとフィールドのマッピング(p.569)を参照してください)。

次に、入力データのフロー先となるノードのツリーを作成する方法を確認します。ノードを追加するには、次のように要素を右クリックし、「**Add Child**」または「**Add Property**」をクリックし、使用可能ないずれかのオプションを選択します。

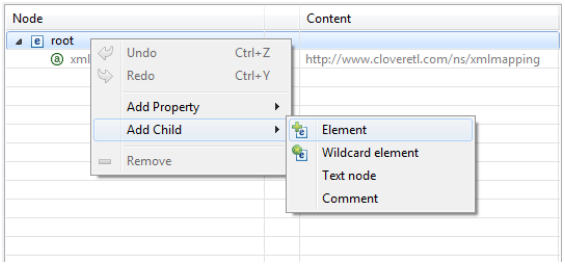


図54.27. ルート要素への子の追加

重要

ノードの追加、ノードの操作およびマウスによる高度なドラッグ・アンド・ドロップの手法の詳細は、ノードの操作(p.567)を参照してください。

名前空間

選択した要素の新しい`xmlns:prefix`属性として**名前空間**を追加します。名前空間を宣言すると、独自のXMLタグを使用できます。各名前空間は、接頭辞およびURIで構成されます。XMLWriterマッピングの場合、ルート要素は、URIが`http://www.cloveretl.com/ns/xmlmapping`の`clover`名前空間を宣言する必要があります。これにより、すべての特別なXMLマッピング・タグへのアクセスが付与されます。「**Source**」タブに切り替えると、これらのタグは明確に`clover:`で始まる(`clover:inport="2"`など)ため、簡単に識別できます。`clover:`接頭辞で始まるXMLタグは実際には出力XMLに書き込まれないことに注意してください。

ワイルドカード属性

属性を明示的にマッピングするかわりに、「**Include**」や「**Exclude**」のワイルドカード・パターンに基づいて属性を要素に移入する、特別なディレクティブを追加します。この機能は、メタデータの独立性を保持する必要がある場合に役立ちます。

属性名は、各メタデータのフィールド名から生成されます。構文: フィールドの指定には「`$portNumber.field`」または「`$portName.field`」を使用し、任意の文字列のフィールド名には「`*`」を使用します。複数のパターンを指定する場合は「`;`」を使用します。

例54.10. ポートおよびフィールドの式の使用方法

`$0.*`: ポート0上のすべてのフィールド

`$0.*;$1.*`: ポート0およびポート1上のすべてのフィールドの組合せ

`$0.address*`: address接頭辞で始まるすべてのフィールド($0.addressState、$0.addressCityなど)

`$child.*`: ポート`child`上のすべてのフィールド(ポートを明示的な番号ではなく名前で指定)

ワイルドカード属性には2つの主要プロパティがあります。次に示すこれらのうち少なくとも1つを常に設定する必要があります。

- **Include:** 含めるパターン(自動生成されたリストに含める必要があるフィールドなど)を定義します。これは、構文が`$port.field`の式で定義されます。前述の式をここで活用できます。「**Exclude**」が設定されている場合は

「**Include**」を空白にできます(その逆も同じです)。「**Include**」が空白である場合は、XMLWriterで、現在の要素の上位のノード(つまり、そのすべての親)または要素自体に接続されたすべてのポートを使用できます。

- **Exclude:** 自動生成されたリストに明示的に含めないフィールドを指定できます。ここでは、「**Include**」を使用する場合と同じ方法で式を使用できます。

例54.11. 「Include」および「Exclude」プロパティの例

1. **Include** = `$0.i*`

Exclude = `$0.index`

「Include」では、iの文字で始まるポート$0のすべてのフィールドを取得します。「Exclude」では、同じポートの`index`フィールドを削除します。

2. **Include** = (空白)

Exclude = `$1.*;$0.id`

「Include」が指定されていないため、そのノードまたは上位のノードに接続されているすべてのポートが考慮されます。「Exclude」では、ポート$1のすべてのフィールド、およびポート$0の`id`フィールドが削除されます。
条件: ポート$0とポート$1は要素またはその親に接続されています。

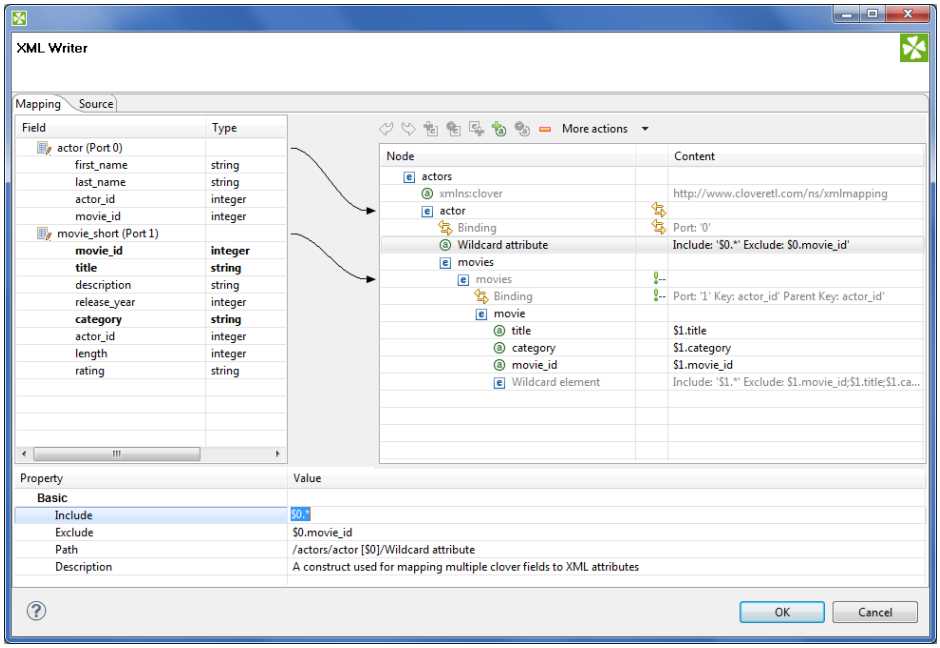


図54.28. ワイルドカード属性およびそのプロパティ

属性

選択した要素に単一の属性を追加します。これを実行すると、属性名をダブルクリックするか、または下部にある**属性名**を編集して、属性名を変更できます。属性**値**は、固定文字列、または属性値にマップしたフィールド値のいずれかです。静的テキストおよび複数のフィールド・マッピングを組み合せることもできます。次の例を参照してください。

例54.12. 属性値の例

`Film`: 属性の値は、リテラル文字列「Film」に設定されます。

`$1.category`: ポート$1の`category`フィールドが属性値になります。

`ID: '{$1.movie_id}'`: ポート$1上の`movie_id`フィールド値535と536に関して「ID: '535'」、「ID: '536'」を生成します。ここでは、フィールド識別子を区切ることができる、オプションの中カッコを使用しています。

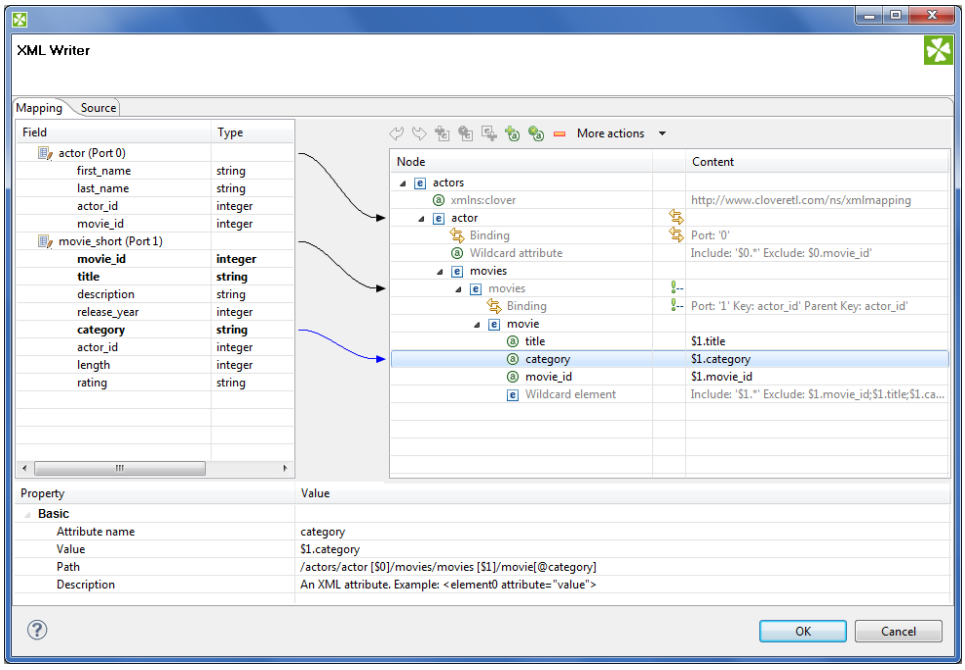


図54.29. 属性およびそのプロパティ

パスおよび**説明**は、ほとんどのノードに共通するプロパティです。これらの両方によって、ノードの概要を把握しやすくなります。**パス**では、XMLツリー内でノードが位置する深さがわかります。

要素

出力XMLツリーの基本部分として要素を追加します。

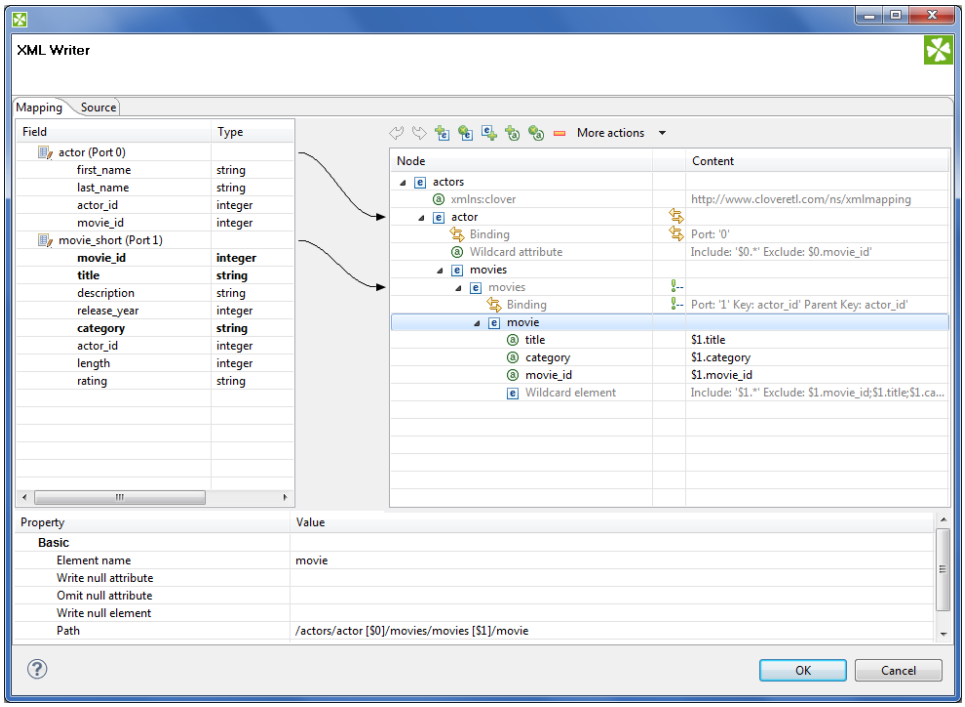


図54.30. 要素およびそのプロパティ

ツリー内での要素の位置と要素に接続されているポートに応じて、次のプロパティを要素に指定できます。

- **Element name**: 出力XMLに表示される要素の名前。
- **Value**: 要素値。フィールドを要素にマップすると、フィールドの値が移入されます。一方、ポートを要素にマップすると、**バインディング**が作成されます(マッピングの作成: ポートとフィールドのマッピング(p.569)を参照してください)。**値**が存在しない場合は、要素を右クリックし、**「Add Child」→「Text node」**を選択します。これで、要素はそのテキスト値を表す新しいフィールドを取得します。新しく作成した「Text node」は空白のままにできません。
- **Write null attribute**: デフォルトでは、NULLにマップされた値を持つ属性は出力に書き込まれません。ただし、ここでは、出力に常に表示される属性の名前を明示的にリストできます。

例54.13. null属性の書込み

要素<date>とその属性「time」が、入力ポート0、フィールド`time`にマップされる(つまり、<date time="$0.time"/>)とします。`time`フィールドが空(null)のレコードの場合、デフォルト出力は次のようになります。

```
<date/>
```

「Write null attribute」を`time`に設定すると、次が生成されます。

```
<date time="" />
```

- **Omit null attribute**: **「Write null attribute」**とは対照的に、現在の要素の属性の値がnullの場合は、その値が書き込まれないことを指定します。この動作がデフォルトです。**「Omit null attribute」**の実際の目的は、ワイルドカード式で**「Write null attribute」**と組み合せることにあります。

例54.14. Null属性の省略

ワイルドカード属性を持つ要素があるとします。要素はポート2に接続され、そのフィールドは**「Include=$2.*」**などのワイルドカード属性にマップされています。一部のフィールドにはデータが含まれないことがわかっています。ここで、「`height`」および「`width`」など一部の空フィールドに書き込もうとしています。これを実行するには、要素をクリックして次を設定します。

Write null attribute=$2.*: nullであってもすべての属性の書込みを強制します。

Omit null attribute=$2.height;$2.width: 該当の属性のみが書き込まれなくなります。

- **Hide**: ポートが接続されている要素で**「Hide」**を`true`に設定し、選択した要素は出力XMLに書き込まれないが選択した要素の子はすべて書き込まれるという動作を強制します。デフォルトでは、このプロパティは`false`に設定されます。非表示の要素は、グレーのフォントでマッピング・エディタに表示されます。

例54.15. 非表示の要素

次のようなXMLがあるとします。

```
<address>
 <city>Atlanta</city>
 <state>Georgia</state>
</address>
<address>
 <city>Los Angeles</city>
 <state>California</state>
</address>
```

address要素を非表示にすると、次が生成されます。

```
<city>Atlanta</city>
<state>Georgia</state>
<city>Los Angeles</city>
<state>California</state>
```

- **Partition**: デフォルトでは、ポートに接続されている最初かつ最上位の要素に応じて、パーティション化が実行されます。そのような要素が複数ある場合は、パーティションを管理する要素を識別するために、それらのいずれかで「**Partition**」をtrueに設定します。パーティションは1回のみ設定できます。つまり、ある要素の「**Partition**」をtrueに設定した場合、そのサブ要素のいずれにもそれを設定しないでください(そのようにしないとグラフが失敗します)。パーティションの詳細は、異なる出力ファイルへの出力のパーティション (p.318)を参照してください。

例54.16. 任意の要素に応じたパーティション

次のマッピング・スニペットでは、<invoice>要素で「**Partition**」をtrueに設定すると、次の動作が生成されます。

<person>は、各ファイルで繰り返されます。

<invoice>は、複数のファイルに分割(パーティション化)されます。

```
<person clover:inPort="0">
 <firstname> </firstname>
 <surname> </surname>
</person>

<invoice clover:inPort="1" clover:partition="true""">
 <customer> </customer>
 <total> </total>
</invoice>
```

ワイルドカード要素

一連の要素を追加します。「**Include**」プロパティと「**Exclude**」プロパティは、どの要素が追加され、どの要素が追加されないかに影響します。$port.field構文の使用方法は、ワイルドカード属性(p.562)を参照してください。そこで説明されているルールと例は、**ワイルドカード要素**にも適用されます。さらに、**ワイルドカード要素**には2つの追加プロパティが付属し、それらの意味は、次のように「**Write null attribute**」と「**Omit null attribute**」のいずれかと密接に関連します。

- **Write null element**: $port.field構文を使用して、どの要素が、コンテンツがない場合にも出力に書き込まれるかを決定します。デフォルトでは、要素に値がない場合、その要素は書き込まれません。「**Omit null element**」を指定する場合は、「**Write null element**」を入力する必要はありません。この場合は、「**Include**」および「**Exclude**」の場合と同様、要素またはその上位に接続されているすべてのポートを使用できます。次の例を参照してください。

- **Omit null element**: $port.field構文を使用して、空白の要素をスキップします。デフォルトではこれらは書き込まれませんが、「**Write null element**」でその前に強制的に書き込まれた空白の要素をスキップするために「**Omit null element**」を使用する場合があります。または、「**Omit null element**」のみを使用することもできます。これは、要素またはその上位に接続されているすべてのポートから受信した空白要素を除外することを意味します。

例54.17. 空白の要素の書込みおよび省略

次のようなXMLファイルを作成するとします。

```
<person>
 <firstname>William</firstname>
 <middlename>Makepeace</middlename>
 <surname>Thackeray</surname>
</person>
```

ただし、ミドル・ネームがない人については、ミドル・ネームを表す要素を書き込む必要はありません。**ワイルドカード要素**を作成し、人に関するデータが含まれるポート(「middle」フィールドが含まれるポート$0など)にそれを接続し、「**Include**」プロパティを入力して、最後に次のように設定する必要があります。

Write null element = `$0.*`

Omit null element = `$0.middle`

結果として、名と姓は空白の場合でも常に書き込まれます。ミドル・ネーム要素は、「middle」フィールドにデータが含まれない場合は書き込まれません。

テキスト・ノード

要素のコンテンツを追加します。これは、縮小されていない要素の最も端に表示されます。つまり、存在する可能性のあるバインディング、ワイルドカード属性または属性の後ろに常に置かれます。ここでも、その値は、固定文字列、ポートのフィールド、またはそれらの組合せのいずれかです。

コメント

コメントを追加します。これにより、XMLツリー内のすべてのノードにコメントして、マッピングを明確かつ判読しやすくできます。追加したすべてのコメントは、マッピング・エディタにのみ表示されます。さらに、コメントの「**Write to the output**」をtrueに設定して、出力XMLファイルにコメントが書き込まれるようにできます。「**Source**」タブを調べて、コメントが存在することを確認します。次に例を示します。

```
<!-- clover:write This is my comment in the Source tab.It will be written to the output
XML because I set its 'Write to output' to true.There is no need to worry about the
"clover:write" directive at the beginning as no attribute/element starting with
the "clover" prefix is put to the output.
-->
```

ノードの操作

最初の要素を追加すると、ルートを除くすべての要素に「**Add Child**」(および「**Add Property**」)以外のその他のオプションも含まれていることがわかります。要素を右クリックして、「**Add Sibling Before**」または「**Add Sibling After**」をさらに選択します。これらを使用すると、現在選択している要素の前または後ろに兄弟を追加できます。

右クリックによるコンテキスト・メニュー以外に、XMLツリー・ビューの上にあるツールバー・アイコンを使用できます。

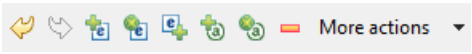


図54.31. マッピング・エディタのツールバー

ツールバー・アイコンは、ツリーで選択したノードに応じてアクティブになります。実行できるアクションは次のとおりです。

- 実行した最後のアクションを**元に戻したり再実行します。**
- 選択した要素の下に**子要素を追加します。**
- 選択した要素の下に**(子)ワイルドカード要素を追加します。**

- 選択した要素の**後ろに兄弟要素を追加します。**
- 選択した要素に**子属性を追加します。**
- 選択した要素に**ワイルドカード属性を追加します。**
- 選択したノードを**削除します。**
- **その他のアクション**: 前述のアクションの他に、特に、兄弟を**前または後ろに追加できます。**

XMLツリーを最初から作成する場合は(マッピングの作成: 新しいXML構造の設計(p.561)を参照)、マウスのクリック回数の削減と作業の高速化に、次のヒントが役立ちます。

- ポートをドラッグして要素の上にドロップする: **バインディング**が作成されます(マッピングの作成: ポートとフィールドのマッピング(p.569)を参照してください)。
- フィールドをドラッグして要素の上にドロップする: 同じ名前の子要素がフィールドとして追加されます。
- 使用可能な1つのフィールド(または複数のフィールド)を要素上にドラッグ: 名前がフィールド名であるサブ要素が作成されます。同時に、要素のコンテンツが`$portNumber.fieldName`に設定されます。
- 使用可能な1つ以上のポートをドラッグして、**バインディング**のある要素の上にドロップ: **「Include」**が`$portNumber.*`に設定された**ワイルドカード要素**が作成されます。
- 前述の2つの組合せ、つまり、ポートおよびフィールド(別のポートのフィールドも含む)を**バインディング**のある要素の上にドラッグ: ポートは**ワイルドカード要素**になり(**Include**=`$portNumber.*`)、その一方でフィールドはコンテンツが`$portNumber.fieldName`のサブ要素になります。
- 使用可能なポートまたはフィールドをドラッグして、ワイルドカード要素や属性の上にドロップ: ポートまたはフィールドがワイルドカード要素や属性の**インクルード・ディレクティブ**に追加されます。ポートの場合は、$0.* (ポート0の場合の例)として追加されます。フィールドの場合は、$0.priceTotal (ポート0、フィールドpriceTotalの場合の例)として追加されます。
- ポートまたはフィールドをドラッグして、**「Include」**や**「Exclude」**などのプロパティ(または**「Input in Binding」**を除くその他のプロパティ)の上にドロップ。このことは、**「Content」**ペインまたは**「Property」**ペインのいずれかで実行でき、結果として、プロパティはポートまたはフィールドの値を受信します。フィールドの複数選択およびそれらのドラッグも機能します。さらに、**[Ctrl]**を押しながらドラッグすると、ポートまたはフィールド値が、(置き換えるのではなく)プロパティの最後に追加されます。**「Include」**プロパティに、現在、たとえば`$0.*`が含まれているとします。**[Ctrl]**を押しながら「`port $1`」の「`field1`」をドラッグして**「Include」**の上にドロップすると、「`$0.*;$1.field1`」というコンテンツが生成されます。

追加した各ノードは、マウスの左ボタンを使用して単にドラッグ・アンド・ドロップすることで、ツリーに後で移動できます。このようにして、XMLツリーを目的に応じて再配置できます。実行できるアクションは次のとおりです。

- (ワイルドカード)要素をドラッグして別の要素の上にドロップ: (ワイルドカード)要素がサブ要素になります。
- (ワイルドカード)属性をドラッグして要素の上にドロップ: 要素に(ワイルドカード)属性が含まれるようになります。
- テキスト・ノードをドラッグして要素の上にドロップ: 要素の値がテキスト・ノードになります。
- 名前空間をドラッグして要素の上にドロップ: 要素に名前空間が含まれるようになります。

マッピング・エディタでのノード(要素や属性など)の削除は、[Delete]を押すか、またはノードを右クリックして**「Remove」**を選択しても実行できます。複数のノードを一度に選択するには、**[Ctrl]**を押しながら**クリックする**か、または**[Shift]**を押しながら**クリックします**。

マッピング・エディタでの操作時、いつでも**[Ctrl]**を押しながら**[Z]**をクリックして、最後に実行したアクションを元に戻したり、**[Ctrl]**を押しながら**[Y]**をクリックして、そのアクションを再実行できます。

マッピングの作成: ポートとフィールドのマッピング

マッピングの作成: 新しいXML構造の設計(p.561)では、データのフロー先となる出力XML構造の設計方法について学習しました。マッピング・エディタでの2番目の作業手順は、データ・ソースを要素と属性に接続することです。データ・ソースは、マッピング・エディタ・ウィンドウの左側のポートおよびフィールドで表されます。「**Field**」列と「**Type**」列は、XMLWriterの入力ポートのメタデータに依存しているため、変更できません。

フィールドをXMLノードに接続するには、「**Field**」列のフィールドをクリックし、それをウィンドウの右側部分にドラッグしてXMLノードの上にドロップします。このアクションの結果は、ノード・タイプによって次のように異なります。

- 要素: フィールドは要素値のデータを提供します。
- 属性: フィールドは属性値のデータを提供します。
- テキスト・ノード: フィールドはテキスト・ノードのデータを提供します。
- マウスによる高度なドラッグ・アンド・ドロップの手法については後述します。

新しく作成された接続は、ポートまたはフィールドからノードを指す矢印として表示されます。

ポートをマップするには、マッピング・エディタの左側にあるポートをクリックし、それをウィンドウの右側にドラッグします。フィールドの操作とは異なり、ポートは要素の上にドロップされるのみです。ポートを要素の上にドラッグすることによって、そのデータをマップするのではなく、そのポートでの受信レコードごとにその要素自体を繰り返すことを要素に指示します。結果として、新しい**バインディング**擬似要素が作成されます。この後に示す図を参照してください。

注意

ルート要素への入力ポートのバインディングにはいくつか制限があります。ルートは、次の方法でのみバインドできます。

- 入力ポートには必ず1つのレコードのみが到達するようにします。これで、パーティションを指定する必要がなくなります(ただし、警告メッセージは表示されます)。
- 複数のレコードが入力ポートに到達する場合は、パーティションを指定する必要があります。そうしない場合、XMLWriterが(複数のルート要素が含まれた)無効なXMLファイルを作成します。

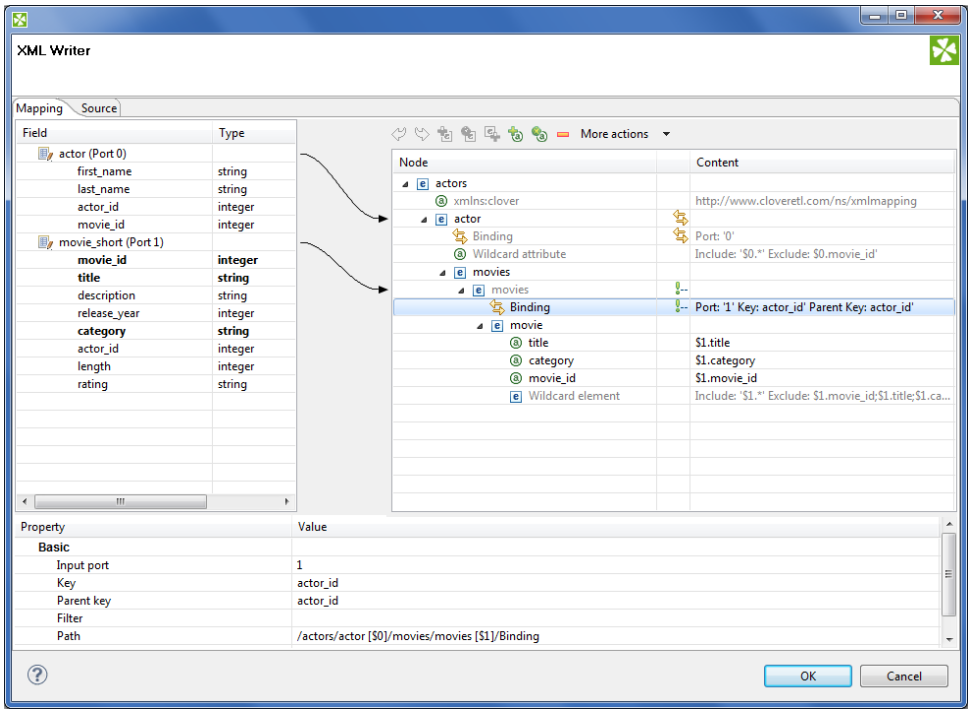


図54.32. ポートおよび要素のバインディング

バインディングでは、要素への入力ポートのマッピングを指定します。このバインディングにより、受信レコードごとに要素自体が繰り返されるようになります。

「Binding」上にマウスを置くと、ツールチップが表示されます。ツールチップでは、ポート・データがキャッシュされるかストリームされるか(全体のパフォーマンスに影響します)、およびどのポートから開始されるかを知らせます。さらに、キャッシュの場合は、ストリームを可能にするにはどのようにデータをソートする必要があるかがわかります。

各**バインディング**には、次に示す一連のプロパティがあります。

- **Input port**: データのフロー元となるポートの番号。この横の矢印を見ると、ノードが接続しているポートを常に簡単にチェックできます。
- **Key**および**Parent key**: このキー・ペアにより、受信データが結合される方法が決まります。**「Key」**に、現在の要素の使用可能なフィールドの名前を入力します。**「Parent key」**に、要素の直接的な親として使用可能なフィールドの名前を入力します。この結果、受信キーの値が等しい場合はデータが結合されます。キーのペアの一方を指定した場合は、もう一方のキーも入力する必要があります。どのフィールドが使用可能であるかを確認するには、キー値領域の右側にある「...」ボタンをクリックします。**「Edit key」**ウィンドウが開き、キーのペアを**「Key parts」**リストに追加することで、キーのペアを効率よく選択できます。parentKeyと正確に同じ数のkeyがある必要があり、そうでない場合はエラーが発生します。

  keyおよび**parentKey**のフィールドに数値が含まれている場合、これらは、データ型に関係なく比較されます。このため、たとえば1.00 (倍精度浮動小数点)は1 (整数)と等しいとみなされ、これらの2つのフィールドは結合されます。

注意

キーは必須プロパティではありません。設定しないと、バインドされているポートから受信するレコードごとに要素が繰り返されます。キーを使用して、これらのレコードの一部のみを実際に選択します。

- **Filter**: どのレコードが出力に書き込まれ、どのレコードが書き込まれないかを選択するCTL式。参考のために、詳細説明(p.598)を参照してください。

バインディングを削除するには、クリックして[Delete]を押します(または、右クリックして**「Remove」**を選択するか、ツールバーにあるこのオプションを見つけます)。

最後に、**バインディング**では、入力ポートとその親ノード(入力ポートにバインドされている最も近い親ノードを意味します)との間の結合をXML構造で指定できます。入力はそれ自体と結合できます。つまり、要素と、同じポートから導出されたその親を結合できます。ただし、これにより、キャッシュが行われて処理速度が低下します。次の例を参照してください。

例54.18. 結合として機能するバインディング

次の2つの入力ポートがあるとします。

0: customers (id, name, address)

1: orders (order_id, customer_id, product, total)

次のような出力が必要です。

```
<customer id="1">
  <name>John Smith</name>
  <address>35 Bowens Rd, Edenton, NC (North Carolina)</address>
  <order>
    <product>Towel</product>
    <total>3.99</total>
  </order>
  <order>
    <product>Pillow</product>
    <total>7.99</total>
  </order>
</customer>

<customer id="2">
  <name>Peter Jones</name>
  <address>332 Brixton Rd, Louisville, KY (Kentucky)</address>
  <order>
    <product>Rug</product>
    <total>10.99</total>
  </order>
</customer>
</programlisting>
```

ordersとcustomerを(orders.customer_id = customers.id)で結合する必要があることがわかります。ポート0 (customers)が<customer>要素にバインドされ、ポート1 (orders)が<order>要素にバインドされます。これで、ネストされたorder要素の**バインディング**擬似属性での設定が非常に簡単になります。「**Key**」をcustomer_idに、「**Parent key**」をidに設定するのは、適切な方法です。

複数値フィールド

Cloverバージョン3.3の時点で、XMLWriterはメタデータの複数値フィールドをサポートします。これには、出力XMLへのリストとマップのマッピングが含まれます。詳細は、複数値フィールド(p.168)およびCTL2のデータ型(p.892)を参照してください。

XMLWriterでは、出力ファイルでリストとマップがどのように表示されるかにのみ注意する必要があります。マップは(中カッコ{ }の間の)単一タグに書き込まれ、一方、リストは*n* (*n*はリストの要素数)個のタグに分けられます。例:

```
        <canadianMap>{ot=Ontario, bc=British_Columbia, at=Alberta,
nt=Northern_Territory}</canadianMap> <!--  map with four key-value pairs -->

        <valueList>-65.25</valueList> <!-- a three-element list -->
     <valueList>71.49</valueList>
     <valueList>-35.02</valueList>
```

マッピングの作成: 既存のXSDスキーマの使用

XSDスキーマをすでに保持している場合は、XML構造を最初から作成する必要はありません。この場合、スキーマを使用して、XMLツリーを事前に生成できます。残っている作業は、XMLノードへのポートのマッピングのみです。マッピングの作成: ポートとフィールドのマッピング(p.569)を参照してください。

まず、スキーマの場所を示すことから開始します。XSDへのフルパスは、**XMLスキーマ**属性で設定する必要があります。2番目に、「**Mapping**」をクリックしてマッピング・エディタを開きます。3番目に、エディタ内でXSDからルート要素を選択し、最後に「**Change root element**」をクリックします(次の図を参照してください)。これで、XMLツリーが自動的に生成されます。プロセスが適切に機能するには、依然として`clover`名前空間を使用する必要があります。

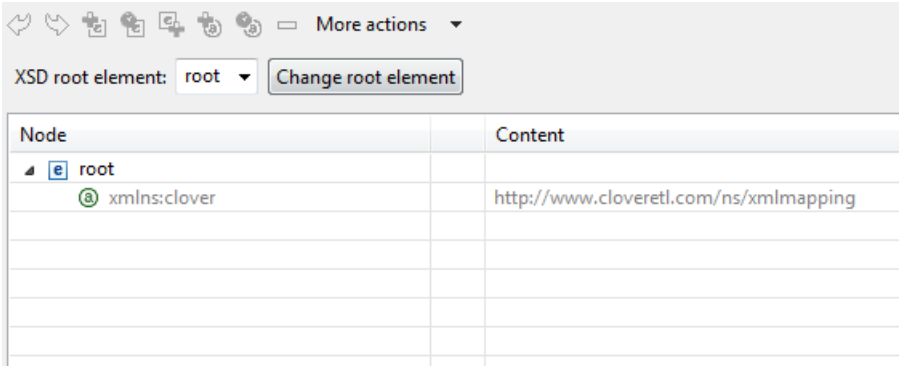


図54.33. XSDルート要素からのXMLの生成

マッピングの作成: 「Source」タブ

マッピング・エディタの「**Source**」タブで、XML構造およびデータ・マッピングを直接編集できます。その概念は、次のように非常に簡単です。

1) 目的のXMLデータを書き込むか、貼り付けます。

2) データ・フィールド・プレースホルダ(`$0.field`など)を、ソースの、要素または属性に入力データを移入する場所に配置します。

3) ポート・バインディングおよび(結合)関係: **入力ポート**、**キー**、**親キー**を作成します。

図54.34. マッピング・エディタの「Source」タブ

実験的に使用できるように、上の図の場合と同じコードを次に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<actors xmlns:clover="http://www.cloveretl.com/ns/xmlmapping">
  <actor clover:inPort="0" clover:include="$0.*" clover:exclude="$0.movie_id">
    <movies>
      <movies clover:inPort="1" clover:key="actor_id" clover:parentKey="actor_id"
         clover:hide="true">
        <movie title="$1.title" category="$1.category" movie_id="$1.movie_id">
          <clover:elements clover:include="$1.*"
            clover:exclude="$1.movie_id;$1.title;$1.category;$1.actor_id"/>
        </movie>
      </movies>
    </movies>
  </actor>
</actors>
```

いずれかのタブで行った変更内容は、もう一方のタブにただちに反映されます。たとえば、ポート**$1**を「**Mapping**」で`invoice`と呼ばれる要素に接続し、「**Source**」に切り替えると、要素が`<invoice clover:inPort="1">`に変更されたことがわかります。

「**Source**」タブでは、タブの左側にあるポートとフィールド両方に対するドラッグ・アンド・ドロップをサポートします。ポート($0)などをソース・コードの任意の場所にドラッグすると、`$0.*`が挿入され、そのすべてのフィールドが使用されることを示します。フィールドのドラッグも同様に機能します。たとえば、ポート**$2**のフィールド`id`をドラッグすると、コード`$2.id`が生成されます。

「**Source**」タブには有用なキーボード・ショートカットがあります。**[Ctrl]**を押しながら**[F]**を押すと、「**Find/Replace**」ダイアログが表示されます。**[Ctrl]**を押しながら**[L]**を押すと、入力する行に素早くジャンプできます。さらに、**[Ctrl]**を押しながら**[Space]**を押すと、高度にインタラクティブなコンテンツ・アシストを使用できます。使用可能なオプションの範囲は、XMLでのカーソル位置によって次のように異なります。

I. 要素タグ内: コンテンツ・アシストを使用すると、「**Write attributes when null**」、「**Omit attributes when null**」、「**Select input data**」、「**Exclude attributes**」、「**Filter input data**」、「**Hide this element**」、「**Include attributes**」、「**Define key**」、「**Omit when null**」、「**Define parent key**」または「**Partition**」のコードを自動的に挿入できます。次の図で、コンテンツ・アシストが機能するように、要素名の後ろに追加のスペースを挿入する必要があることを確認してください。

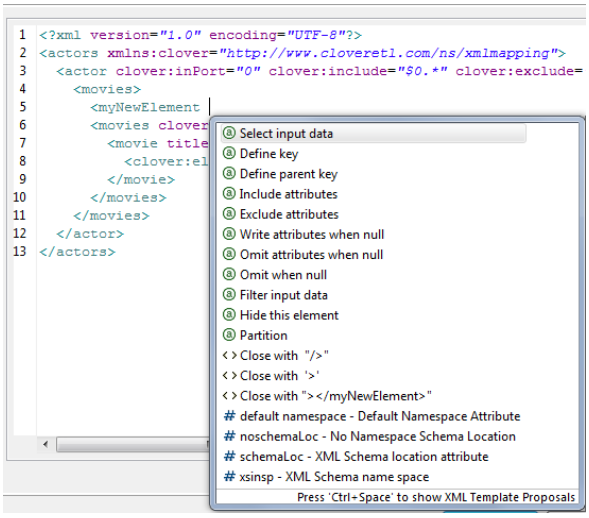


図54.35. 要素内でのコンテンツ・アシスト

挿入されたコードは、マッピングの作成: 新しいXML構造の設計(p.561)で説明しているように、ノードおよびそのプロパティに対応します。

II. 「""」の引用符内: コンテンツ・アシストにより、ノード・プロパティの値(「**Include**」および「**Exclude**」の特定のポートやフィールドなど)を簡単に選択でき、デリミタを追加することもできます。デリミタを使用して、複数の式を互いに分離します。

III. 2つの要素間の空き領域: 選択したポートやフィールドの挿入とは別に、**ワイルドカード要素**を(マッピングの作成: 新しいXML構造の設計(p.561)で説明したように)追加したり、次のように**テンプレートの挿入**または**テンプレートの宣言**を行うことができます。

例54.19. 「Source」タブでのワイルドカード属性の挿入

最初に、要素を作成します。次に、要素タグ内をクリックし、[Space]を押し、次に**[Ctrl]**を押しながら**[Space]**を押して、**「Include attributes」**を選択します。コード: `clover:include=""`が挿入されます。その後、属性の受信元となるポートおよびフィールドを決定する必要があります(つまり、「Mapping」タブでの**「Include」**プロパティの設定と同じアクティビティです)。たとえば、「`$1.id`」を手動で入力するかわりに、コンテンツ・アシストを再度使用します。「""」のカッコ内をクリックし、**[Ctrl]**を押しながら**[Space]**を押すと、使用可能なすべてのポートのリストが表示されます。いずれかを選択し、**[Ctrl]**を押しながら**[Space]**を再び押します。

これで`include`が終了したので、[Space]を押し、次に再度[Ctrl]を押しながら[Space]を押します。実行内容および実行場所に応じてコンテンツ・アシストが実行されたことがわかります。新しいオプション**「Exclude attributes」**が表示されます。それを選択し、「`clover:exclude=""`」を挿入します。その値の指定は、「Mapping」での**「Exclude」**プロパティの入力に相当します。

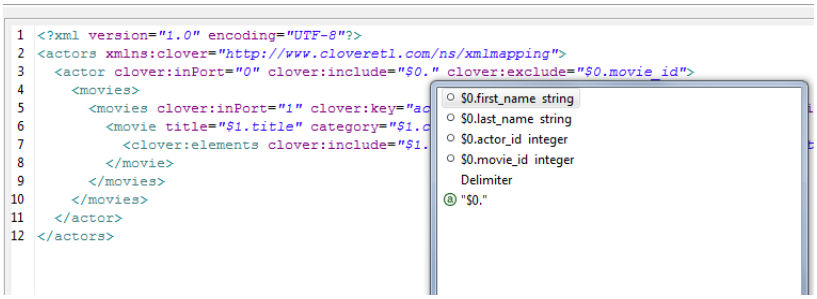


図54.36. ポートおよびフィールドのコンテンツ・アシスト

「Source」タブに関する最後の説明です。`$port.field`構文をもう少し操作する必要がある場合があります。ポート$0があり、その`price`フィールドがあるとします。目的は、これらの価格を要素(`subsidy`など)に送信することです。最初に、ポートと要素との間に接続を確立します。次に、USドル通貨を`price`の数値の直後に追加します。これを行うには、次のようなソース・コードを単に編集します(同じ変更を「Mapping」でも実行できます)。

```
<subsidy>$0.price USD</subsidy>
```

ただし、理由があってUSD文字列を価格に付加する必要がある場合は、{ }カッコを使用して、次のように`$port.field`構文を追加文字列と分離します。

```
<subsidy>{$0.price}USD</subsidy>
```

ドルのプレースホルダを抑制する必要がある場合は、2回入力します。たとえば、通常はポート0からのフィールド・データをマップする「$0.field」を文字列として出力する場合は、「$$0.field」と入力します。このようにすると、次のように出力されます。

```
<element attribute="$0.field">
```

テンプレートおよび再帰

テンプレートは、別の(より大きい)コード・ブロックに挿入するために使用されるコードです。テンプレートは他のテンプレートに挿入できるため、再帰的テンプレートが作成されます。

前述のように、「**Source**」タブのコンテンツ・アシストにより、独自のテンプレートを簡単に宣言および使用できます。このオプションを使用するには、2つの要素間の空き領域で**[Ctrl]**を押しながら**[Space]**を押します。その後、「**Declare template**」または「**Insert template**」を選択します。

「**Declare template**」では、テンプレート・ヘッダーが挿入されます。最初に、テンプレート名を入力する必要があります。2番目に、独自のコードを入力します。テンプレートの例を次に示します。

```
<clover:template clover:name="templCustomer">
<customer>
 <name>$0.name</name>
 <city>$0.city</city>
 <state>$0.state</state>
</customer>
</clover:template>
```

このテンプレートをいずれかの要素の下に挿入するには、**[Ctrl]**を押しながら**[Space]**を押して、「**Insert template**」を選択します。最後に、次のようにテンプレート名を入力します。

```
<clover:insertTemplate clover:name="templCustomer"/>
```

再帰的テンプレートでは、`insertTemplate`タグが、テンプレート内の潜在的データの後ろに表示されます。再帰的構造を作成する場合は、キーおよび親キーを定義することが重要です。これで、`key`と`parentKey`のペアが一致するかぎり、再帰が続行されます。つまり、再帰の深さは入力データによって異なります。`filter`を使用すると、書き込む必要のないレコードを取り除く場合に役立ちます。

第55章 トランスフォーマ

コンポーネントとは何かを理解していることを想定しています。概要は、第19章「コンポーネント」(p.97)を参照してください。

一部のコンポーネントは、グラフの中間ノードです。これらは、**トランスフォーマ**または**ジョイナ**と呼ばれます。

ジョイナの詳細は、第56章「ジョイナ」(p.653)を参照してください。ここでは、**トランスフォーマ**について説明します。

トランスフォーマは、接続済入力ポートを介してデータを受信し、それをユーザー指定の方法で処理し、接続済出力ポートを介して送信します。

コンポーネントには様々なプロパティを設定できます。ただし、一部のものが共通する場合があります。すべてのコンポーネントに共通のプロパティもあれば、ほとんどのコンポーネントに共通のプロパティや、**トランスフォーマ**のみに共通のプロパティもあります。次のことを学習している必要があります。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

トランスフォーマは、トランスフォーマが実行できる内容に応じて識別できます。

- ある**トランスフォーマ**は、各入力データをすべての接続済出力に単にコピーします。
  - SimpleCopy(p.647)は、各入力データ・レコードをすべての接続済出力ポートにコピーします。
- ある**トランスフォーマ**は、一部の入力レコードのみを出力に渡します。
  - DataSampler(p.585)は、一部の入力レコードを、選択されたいずれかのフィルタリング方針に基づいて出力に渡します。
- ある**トランスフォーマ**は、重複データ・レコードを削除します。
  - Dedup(p.587)は、重複データを削除します。重複データは、オプションの2番目の出力ポートを介して送信できます。
- 他のコンポーネントは、次のようにユーザー定義の条件に応じてデータをフィルタします。
  - ExtFilter(p.597)は、ユーザー定義の条件を使用してデータを比較し、この条件と一致するレコードを送信します。条件に一致しなかったデータ・レコードは、オプションの2番目の出力ポートを介して送信できます。
- その他の**トランスフォーマ**は、次のようにそれぞれ様々な方法でデータをソートします。
  - ExtSort(p.600)は、入力データをソートします。
  - FastSort(p.602)は、**ExtSort**よりも高速に入力データをソートします。
  - SortWithinGroups(p.649)は、ソート済データのグループ内の入力データをソートします。
- ある**トランスフォーマ**は、次のようにデータに関する情報を集約できます。
  - Aggregate(p.578)は、入力データ・レコードに関する情報を集約します。
- ある**トランスフォーマ**は、接続済出力ポートに入力レコードを次のように分配します。
  - Partition(p.619)は、様々な接続済出力ポートに個々の入力データ・レコードを分配します。

  - LoadBalancingPartition(p.626)は、ダウンストリーム・コンポーネントのワークロードに応じて、様々な出力ポートに受信入力データ・レコードを分配します。

- あるトランスフォーマは、2つの入力ポートを介してデータを受信し、3つの出力ポートを介してデータを送信します。最初のポートのみ、両方のポート、または2番目のポートに含まれるデータは、それぞれ対応する出力ポートに送信されます。

  - DataIntersection(p582.)は、2つの入力からのソート済データを交差し、交差によって定義される3つの接続済出力ポートを経由して送信します。

- その他のトランスフォーマでは、複数の入力ポートからデータ・レコードを受信し、それらをすべて一意の出力ポートを介して送信できます。

  - Concatenate(p581.)は、メタデータが同じデータ・レコードを1つ以上の入力ポートから受信し、それらをまとめてから、一意の出力ポートを介して送信します。各入力ポートからのデータ・レコードは、前の入力ポートからのすべてのデータ・レコードの後に送信されます。

  - SimpleGather(p.648)は、メタデータが同じデータ・レコードを1つ以上の入力ポートから受信し、それらをまとめてから、一意の出力ポートを介して可能なかぎり速く送信します。

  - Merge(p.607)は、2つ以上の入力ポートからの、メタデータが同じソート済データ・レコードを受信し、それらをすべてソートし、一意の出力ポートを介して送信します。

- その他のトランスフォーマは、接続済入力ポートを介してデータを受信し、それをユーザー定義の方法で処理し、接続済出力ポートを介して送信します。

  - Denormalizer(p.589)は、入力データ・レコードのグループから単一の出力データ・レコードを作成します。

  - Pivot(p.628)は、入力レコードが要約されるピボット・テーブルを作成するDenormalizerの単純な形式です。

  - Normalizer(p.612)は、単一の入力データ・レコードから1つ以上の出力データ・レコードを作成します。

  - MetaPivot(p.609)は、Normalizerと同様に機能しますが、常に同じ変換を実行し、出力メタデータのデータ型は固定されます。

  - Reformat(p.632)は、ユーザー定義の方法で入力データを処理します。異なる出力ポートまたはすべて接続された出力ポートに、ユーザー定義の方法で出力データ・レコードを分配できます。

  - Rollup(p.635)は、ユーザー定義の方法で入力データを処理します。複数の出力レコードを、その数とは異なる数の入力レコードから作成できます。異なる出力ポートまたはすべて接続された出力ポートに、ユーザー定義の方法で出力データ・レコードを分配できます。

  - DataSampler(p.585)は、一部の入力レコードのみを出力に渡します。使用可能なフィルタリング方針から、ニーズに合うものを選択できます。

- あるトランスフォーマは、スタイルシートを使用して入力データを変換できます。

  - XSLTransformer(p.651)は、スタイルシートを使用して入力データを変換します。

Aggregate

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Aggregateは、入力データ・レコードに関する統計情報を算出します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Aggregate	-	いいえ	1	1-n	いいえ	いいえ

概要

Aggregateは、単一の入力ポートを通してデータ・レコードを受信し、入力データ・レコードに関する統計情報を算出し、それらをすべての出力ポートに送信します。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意1
出力	1-n	はい	統計情報用	任意2

Aggregateの属性

属性	必須	説明	可能な値
Basic			
Aggregation mapping		セミコロンで区切られる、出力フィールド名の個々のマッピングのシーケンス。各マッピングは、`$outputField:=constant`や`$outputField:=$inputField` (これは、「**Aggregate key**」からのフィールド名である必要があります)または`$outputField:=somefunction($inputField)`の形式になります。	
Aggregate key		それに基づいてレコードをグループ化するキー。詳細は、グループ・キー(p.276)を参照してください。	
Charset		受信データ・レコードのエンコーディング。	ISO-8859-1 (デフォルト) | その他のエンコーディング
Sorted input		デフォルトでは、入力データ・レコードは「**Aggregate key**」に応じてソートされる必要があります。指定されたとおりにソートされない場合は、この値を`false`に切り替えます。	true (デフォルト) | false
Equal NULL		デフォルトでは、null値を持つレコードは異なるとみなされます。`true`に設定すると、null値を持つレコードは等しいとみなされます。	false (デフォルト) | true
Deprecated			
Old aggregation mapping		古いバージョンのCloverETLで使用されたマッピング。現在、その使用は非推奨になっています。	

詳細説明

集約マッピング

「**Aggregation mapping**」属性行をクリックすると、集約マッピング・ウィザードが開きます。そこで、マッピングと集約の両方を定義できます。このウィザードは2つのペインで構成されます。左には「**Input field**」ペインが、右には「**Aggregation mapping**」ペインが表示されます。

1. 出力にマップする必要がある各フィールドを選択するには、それをクリックして、右のペインの、目的の出力フィールド名の行にある「**Mapping**」列にドラッグ・アンド・ドロップします。その後、選択した入力フィールドが「**Mapping**」列に表示されます。このようにして、目的のすべての入力フィールドを出力フィールドにマップできます。

2. これに加えて、「**Aggregate key**」に含まれていないフィールドについては、集約関数も定義する必要があります。

   「**Aggregate key**」のフィールドは、関数を使用せずに(または関数を使用して)出力フィールドにマップできる唯一のフィールドです。

   このため、キー・フィールドにはマッピング: `$outField=$keyField`のみを実行できます。

   一方、キーに含まれていないフィールドに対してこのようなマッピングは使用できません。これらには、関数を常に定義する必要があります。

   (キーに含まれている、またはキーに含まれていない)フィールドの関数を定義するには、「**Function**」列の行をクリックし、コンボ・リストから関数を選択します。目的の関数を選択したら、**[Enter]**をクリックします。

3. 各出力フィールドでは、定数が割り当てられている場合もあります。

例55.1. 集約マッピング

```
$Count=count();$AvgWeight:=avg($weight);$OutFieldK:=$KeyFieldM;
$SomeDate:=2008-08-28
```

内容を次に示します。

1. 出力フィールドは「Count」、「AvgWeight」、「OutFieldK」および「SomeDate」です。出力メタデータにはその他のフィールドを含めることもできます。

2. 入力フィールドは「weight」および「KeyFieldM」です。入力メタデータにはその他のフィールドを含めることもできます。

3. 「KeyFieldM」は、**「Aggregate key」**からのフィールドである必要があります。このキーは、他のフィールドで構成される場合もあります。

   「weight」は、**「Aggregate key」**からのフィールドではありません。

   2008-08-28は、出力の日付フィールドに割り当てられた日付定数です。

   count()およびavg()は、入力に適用できる関数です。前者には引数は必要なく、後者には1つの引数、つまり各入力レコードの「weight」フィールドの値が必要です。

Concatenate

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Concatenateは、複数の入力からデータ・レコードを収集します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Concatenate	はい	いいえ	1-n	1	-	-

概要

Concatenateは、ソートされていない可能性のあるデータ・レコードを1つ以上の入力ポートを通して受信します。(すべての入力ポートのメタデータが同じである必要があります。)**Concatenate**は、すべてのレコードを入力ポート順に収集し、それらを単一の出力ポートに送信します。入力レコードの収集を最初の入力ポートから開始し、次の入力ポートで続行し、最後のポートで終了します。各入力ポート内で、レコードの順序は保持されます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
	1-n	いいえ	入力データ・レコード用	入力0[1)]
出力	0	はい	収集されたデータ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。すべての出力メタデータが同じである必要があります。

DataIntersection

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

DataIntersectionは、2つの入力からのデータを交差します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
DataIntersection	いいえ	はい	2	3	はい	はい

概要

DataIntersectionは、2つの入力からソート済データを受信し、それら両方の**結合キー**値を比較し、次の方法でレコードを処理します。

入力ポート0と入力ポート1の両方にある入力レコードは、ユーザー定義の変換に従って処理され、結果は出力ポート1に送信されます。入力ポート0にのみ存在する入力レコードは、変更されずに出力ポート0に送信されます。入力ポート1にのみ存在する入力レコードは、変更されずに出力ポート2に送信されます。

レコードは、すべての「**Join key**」フィールドの値が両方のレコードで等しい場合、両方のポートに存在するとみなされます。そうでない場合は、入力ポート0または入力ポート1にのみ存在するレコードであるとみなされます。

変換を定義する必要があります。変換では、**DataIntersection**用のCTLテンプレートが使用され、`RecordTransform`インタフェースを実装するか、または`DataRecordTransform`スーパークラスから継承します。インタフェース・メソッドをこの後に示します。

注意

このコンポーネントは**ジョイナ**と同様、入力のメタデータが同一である必要はなく、**結合キー**が等しいレコードを処理します。また、重複レコードを変換に送信できるようにするかどうかを選択できます(「**Allow key duplicates**」)。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用(データ・フローA)	任意(In0)[1)]
	1	はい	入力データ・レコード用(データ・フローB)	任意(In1)[1)]
出力	0	はい	変更されない出力データ・レコード用(フローAにのみ含まれる)	入力0[2)]
	1	はい	変更された出力データ・レコード用(両方の入力フローに含まれる)	任意(Out1)
	2	はい	変更されない出力データ・レコード用(フローBにのみ含まれる)	入力1[2)]

説明:

1): これらの一部が同等かつ比較可能である必要があります(**結合キー**)。

2): このコンポーネントを介してメタデータを伝播することはできません。

DataIntersectionの属性

属性	必須	説明	可能な値
Basic			
Join key	はい	入力ポートからのデータ・レコードを比較するキー。この属性の値が等しいレコード(各入力からのレコード)のペアのみが、変換に送信されます。詳細は、結合キー(p.584)を参照してください。適正な結果を得るために、レコードを昇順でソートする必要があります。	
Transform	1)	CTLまたはJavaでグラフに記述された、レコードの交差方法の定義。	
Transform URL	1)	CTLまたはJavaで記述された、レコードの交差方法の定義が含まれる、外部ファイルの名前(パスを含む)。	
Transform class	1)	レコードを交差する方法を定義する外部クラスの名前。	
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Equal NULL		デフォルトでは、キー・フィールドの値がnullのレコードは等しいとみなされます。`false`に設定すると、それらは互いに異なるとみなされます。	true (デフォルト) | false
Advanced			
Allow key duplicates		デフォルトでは、入力のすべての重複が許可されます。`false`に切り替えると、重複するキー値を持つレコードは許可されません。`false`の場合、結合には**最初の**レコードのみが使用されます。	true (デフォルト) | false
Deprecated			
Error actions		指定した変換が**エラー・コード**を返した場合に実行する必要があるアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、**コンソール**に書き出されます。	

属性	必須	説明	可能な値
Slave override key		古い形式の**結合キー**。2番目の入力ポートのフィールドのみが含まれます。この属性は現在非推奨になっており、現在の形式の「**Join key**」属性を使用することをお薦めします。	

説明:

1): これらのいずれかを指定する必要があります。これらの変換属性はいずれも**DataIntersection**用のCTLテンプレートを使用するか、`RecordTransform`インタフェースを実装します。

詳細は、CTLスクリプトの詳細(p.584)またはDataIntersection用Javaインタフェース(p.584)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

詳細説明

- **Join key**

  各副次式をセミコロンで区切ったシーケンスとして表されます。各副次式は、左側にある(ドル記号の接頭辞を付けた)最初の入力ポートからのフィールド名と、右側にある(ドル記号の接頭辞を付けた) 2番目の入力ポートからのフィールド名の割当てです。

例55.2. DataIntersectionの結合キー

```
$first_name=$fname;$last_name=$lname
```

この**結合キー**では、`first_name`および`last_name`は、最初の入力ポートのメタデータのフィールドで、`fname`および`lname`は、2番目の入力ポートのメタデータのフィールドです。

両方の入力ポートにおいてこのキーに同じ値が含まれているレコードのペアは、変換され、2番目の出力ポートに送信されます。2番目の入力ポートには対応するレコードがなく、最初の入力ポートを通して受信するレコードは、変更されずに最初の出力ポートに送信されます。最初の入力ポートには対応するレコードがなく、2番目の入力ポートを通して受信するレコードは、変更されずに3番目の出力ポートに送信されます。

CTLスクリプトの詳細

3つの変換属性のいずれを定義する場合も、いくつかの出力ポートを各入力レコードに割り当てる変換を指定する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム変換を指定できます。

DataIntersection用CTLテンプレート

DataIntersectionは、**Reformat**および**ジョイナ**と同じ変換テンプレートを使用します。詳細は、ジョイナ用CTLテンプレート(p.325)を参照してください。

DataIntersection用Javaインタフェース

DataIntersectionは、**Reformat**および**ジョイナ**と同じインタフェースを実装します。詳細は、ジョイナ用Javaインタフェース(p.328)を参照してください。

DataSampler

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

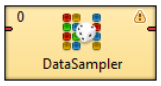
DataSamplerは、特定の入力レコードのみを出力に渡します。変換を制御するために選択できる各種のフィルタリング方針があります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
DataSampler	-	いいえ	1	1-N	いいえ	いいえ

概要

DataSamplerは、その単一の入力エッジでデータを受信します。次に、入力レコードをフィルタリングして、その一部のみを出力に渡します。**サンプリング方法**と呼ばれるフィルタリング方針の1つを選択することにより、渡される入力レコードを制御できます。入力と出力のメタデータが相互に一致する必要があります。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	サンプリングされたデータ・レコード用	Input0

DataSamplerの属性

属性	必須	説明	可能な値
Basic			
Sampling method	はい	どのレコードを出力に渡すかを決定するフィルタリング方針。選択可能な個々の方針は、詳細説明(p.586)を参照してください。	Simple|Systematic|Stratified|PPS
Required sample size	はい	入力に対する割合で表した、出力データに要求されるサイズ。出力を、たとえば入力の15% (概算)にする場合は、この属性を0.15に設定します。	(0; 1)
Sampling key	1)	「**Sampling method**」で階層を定義するために使用されるフィールド名。複数のフィールド名をコロン、セミコロンまたはパイプで区切られたシーケンスに結合できます。各フィールドの後に、カッコで囲んだ順序インジケータを追加できます(昇順は「**a**」、降順は「**d**」、無視は「**i**」、自動評価は「**r**」)。	例: Surname(a); FirstName(i); Salary(d)
Advanced			
Random seed		ランダム・ジェネレータで使用される`long`数。すべてのグラフの実行において、結果がランダムであってもその一意性が維持されることを保証します。	<0; N>

説明:

1) この属性は、「**Simple**」を除くすべてのサンプリング方法に必須です。

詳細説明

DataSamperの一般的なユースケースとして、次のような場合が考えられます。データ変換が適切に機能するかどうかを確認するとします。数百万のレコードを処理している場合に、数千のレコードのみを取得して調べると便利です。そのため、このコンポーネントを使用して、データ・サンプルを作成します。

DataSamplerには、データ・セット全体を代表するサンプルを作成するための4つの**サンプリング方法**が用意されています。

- **Simple**: 選択される可能性は、すべてのレコードについて同等です。フィルタリングは、<0.0d; 1.0d)の間隔から選択された`double`値(近似、均一)に基づきます。抽出数が「**Required sample size**」よりも小さい場合にレコードが選択されます。
- **Systematic**: ランダムに開始されます。その後は、順序付きリストのk番目ごとの要素を選択して続行されます。最初の要素および間隔は「**Required sample size**」から導出されます。この方法は、「**Sampling key**」で指定したソート順に並べられたデータ・セットに依存します(結果を典型的なものにするため)。ソートされていない入力のサンプルが必要になる場合もあります。「**Sampling key**」の指定は必須ですが、順序インジケータをi、つまり無視に設定することによってソート順を抑制できます。これにより、データ・セットのソート順は考慮されなくなります。キー設定は、たとえばInvoiceNumber(i)のようになります。
- **Stratified**: データ・セットに多くの個別カテゴリが含まれる場合は、セットをこれらのカテゴリ別の階層に構成できます。次に、各階層を個別の部分母集団として、そこから個別要素をランダムに選択してサンプリングします。各階層から1つ以上のレコードが選択されます。
- **PPS** (確率比例サンプリング): 各レコードの確率が、その階層のサイズ(最大1)に比例して設定されます。階層は「**Sampling key**」で選択したフィールドの値で定義されます。その後は、レコードの各グループに対して**Systematic**サンプリングが使用されます。

各方法を比較すると、**Simple**のランダムなサンプリングが最も単純で高速です。これはほとんどのケースに対応します。ソート順を指定しない**Systematic**サンプリングは**Simple**と同程度に高速であり、強力な代表データ・プローブも生成されます。**Stratified**サンプリングには最も注意が必要です。データ・セットを適正なサイズの個別グループに分割できるときにのみ役立ちます。そうでない場合は、データ・プローブが要求よりもかなり大きくなります。統計におけるサンプリング方法の詳細は、Wikipediaを参照してください。

Dedup

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Dedupは、重複するレコードを削除します。

コンポーネント	同じ入力メタデータ	ソート済入力 [1)]	入力	出力	Java	CTL
Dedup	-	✔	1	0-1	-	-

[1)] 入力レコードは部分的にのみソートできます。つまり、同じ**重複除外キー**の値を持つレコードがまとめてグループ化されますが、これらのグループはソートされません。

概要

Dedupは、**重複除外キー**の同じ値でグループ化したレコードのデータ・フローを読み取ります。キーは、入力レコードからの単数または複数のフィールド名で形成されます。キーが指定されない場合、このコンポーネントはUNIXの`head`コマンドまたは`tail`コマンドと同様に動作します。グループに順序を付ける必要はありません。

このコンポーネントは、グループまたは入力全体の最初または最後から指定された数のレコードを選択できます。重複のないレコードのみを選択することもできます。

重複のあるレコードは、出力ポート0に送られます。重複するレコードは、出力ポート1に送ることができます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	入力データ・レコード用	任意
出力	0	✔	重複のあるデータ・レコード用	同等の入力メタデータ [1)]
	1	✖	重複するデータ・レコード用	

[1)]メタデータはこのコンポーネントを介して伝播できます。

Dedupの属性

属性	必須	説明	可能な値
Basic			
Dedup key		レコードを重複除外する基準となるキー。デフォルト、つまり「**Dedup key**」が設定されない場合は、すべての入力レコードの最初または最後から、「**Number of duplicates**」属性によって指定された数のレコードが保存され、その他は削除されます。「**Dedup key**」が設定された場合は、**重複除外キー**として指定されたフィールドと同じ値を持つレコードが、指定された数のみ選択されます。Dedup key(p.588)を参照してください。	
Keep		保存するレコードを定義します。`First`に設定すると、最初からです。`Last`に設定すると、最後からです。レコードは、グループまたは入力全体から選択されます。`Unique`に設定した場合は、重複のないレコードのみが選択されます。	First (デフォルト) | Last | Unique
Equal NULL		デフォルトでは、キー・フィールドの値がnullのレコードは等しいとみなされます。`false`の場合、これらは異なるとみなされます。	true (デフォルト) | false
Number of duplicates		等しいキー値を持つ隣接するレコードの各グループから選択される重複レコードの最大数。キーが設定されていない場合は、すべてのレコードの最初または最後からのレコードの最大数。`Unique`オプションが選択された場合は無視されます。	1 (デフォルト) | 1-N

詳細説明

- **Dedup key**

  このコンポーネントは、ソートされた入力データおよび部分的にソートされたデータの両方を処理できます。**重複除外キー**を構成するフィールドを設定するときは、適切な「**Order**」属性を選択してください。

  1. Ascending: 同じキー・フィールド値を持つ入力レコードのグループを昇順でソートする場合。
  2. Descending: 同じキー・フィールド値を持つ入力レコードのグループを降順でソートする場合。
  3. Auto: 入力レコードのグループのソート順は、異なるキー・フィールド値を持つ最初の2つのレコード(つまり、最初の2つのグループの最初のレコード)から推測されます。
  4. Ignore: 同じキー・フィールド値を持つ入力レコードのグループをソートしない場合。

Denormalizer

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Denormalizerは、1つ以上の入力レコードから単一の出力レコードを作成します。

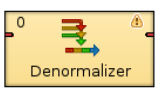
コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Denormalizer	-	✗	1	1	✓	✓

概要

Denormalizerは、単一の入力ポートを介してソート済データを受信し、「**Key**」の値をチェックし、同じ「**Key**」の値を持つ1つ以上の隣接する入力レコードから1つの出力レコードを作成します。

変換を定義する必要があります。変換は**Denormalizer**用のCTLテンプレートを使用するか、`RecordDenormalize`インタフェースを実装するか、`DataRecordDenormalize`スーパークラスから継承します。インタフェース・メソッドをこの後に示します。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✓	入力データ・レコード用	任意
出力	0	✓	非正規化データ・レコード用	任意

Denormalizerの属性

属性	必須	説明	可能な値
Basic			
Key	[1]	入力データ・レコードのグループを作成する基準となる値を持つキー。「**Key**」と同じ値を持つ隣接する入力レコードは、1つのグループの構成要素とみなされます。そのようなグループの構成要素から1つの出力レコードが作成されます。詳細は、Key(p.590)を参照してください。	
Group size	[1]	グループは、その構成要素の正確な数で定義できます。たとえば、5つのレコードごとに1つのグループが形成されます。入力レコード数は必ず`group size`の倍数になります。これは「`key`」属性と相互に排他的です。	数値
Denormalize	[2]	レコードを非正規化する方法の定義。グラフにCTLで記述します。	
Denormalize URL	[2]	パスを含めた外部ファイルの名前。このファイルには、CTLまたはJavaで記述された、レコードを非正規化する方法の定義が含まれています。	
Denormalize class	[2]	レコードを非正規化する方法の定義。グラフにJavaで記述します。	
Sort order		入力レコードのグループのソートに使用される順序。Sort order(p.591)を参照してください。	Auto (デフォルト) | Ascending | Descending | Ignore
Equal NULL		デフォルトでは、キー・フィールドの値がnullのレコードは等しいとみなされます。`false`の場合、これらは異なるとみなされます。	true (デフォルト) | false
Denormalize source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Deprecated			
Error actions		指定した変換がエラー・コードを返した場合に実行する必要があるアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、**コンソール**に書き出されます。	

[1] `group size`の優先度はkeyよりも高くなります。どちらの属性も指定されない場合は、すべてのレコードで1つのグループが形成されます。

[2] これらのいずれかを指定する必要があります。これらの変換属性はいずれも**Denormalizer**用のCTLテンプレートを使用するか、`RecordDenormalize`インタフェースを実装します。

詳細は、CTLスクリプトの詳細(p.591)またはDenormalizer用Javaインタフェース(p.596)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

詳細説明

- **Key**

  フィールド名をセミコロン、コロンまたはパイプで区切ったシーケンスとして表されます。

例55.3. Denormalizerのキー

```
first_name;last_name
```

この**キー**の「`first_name`」および「`last_name`」は、入力ポートでのメタデータのフィールドです。

- **Sort order**

  レコードを「**Group size**」ではなく「**Key**」で非正規化する場合、入力レコードを「**Key**」フィールドの値に従ってグループ化する必要があります。次に、グループのソート順に応じて適切な**ソート順**を選択します。

  1. Auto: 入力レコードのグループのソート順は、異なるキー・フィールド値を持つ最初の2つのレコード(つまり、最初の2つのグループの最初のレコード)から推測されます。
  2. Ascending: 同じキー・フィールド値を持つ入力レコードのグループを昇順でソートする場合。
  3. Descending: 同じキー・フィールド値を持つ入力レコードのグループを降順でソートする場合。
  4. Ignore: 同じキー・フィールド値を持つ入力レコードのグループをソートしない場合。

CTLスクリプトの詳細

3つの変換属性のいずれを定義する場合も、入力を出力に変換する方法を指定する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム変換を指定できます。

変換を作成したら、タブの右上隅にある該当するボタンをクリックすることで、それをJava言語コードに変換することもできます。

タブの右上隅にある該当するボタンをクリックすると、変換定義を(グラフ・エディタの「**Graph**」タブおよび「**Source**」タブとは)別のグラフのタブで開くことができます。

Denormalizer用CTLテンプレート

変換の定義に使用する「**Source**」タブの例を示します。

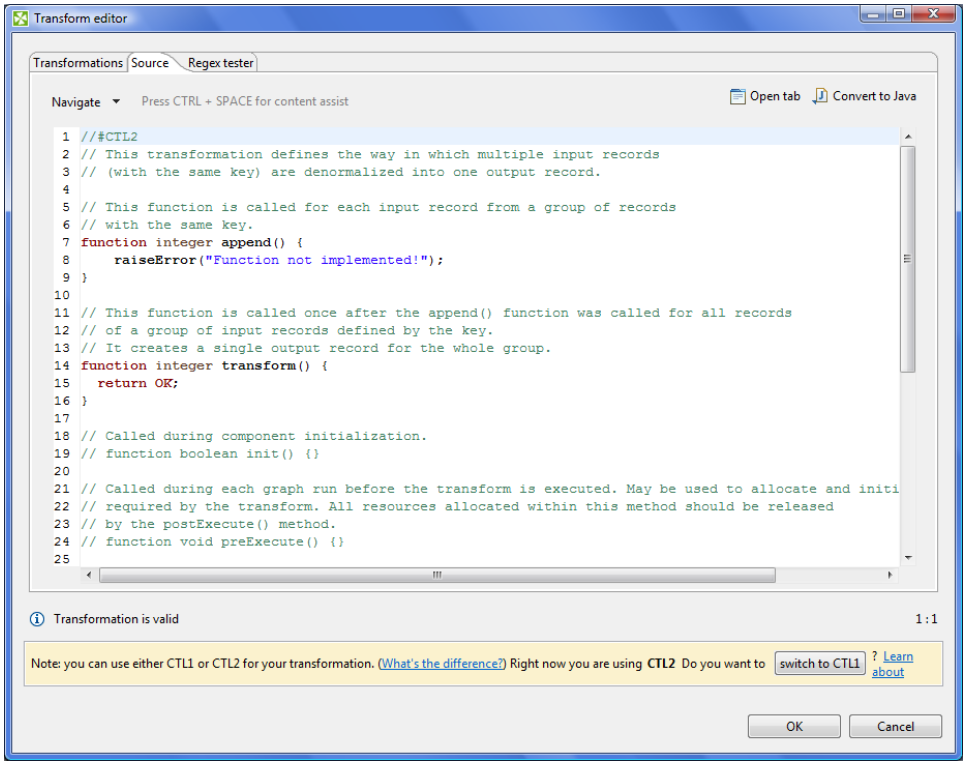


図55.1. Denormalizerコンポーネントの変換エディタの「Source」タブ(I)

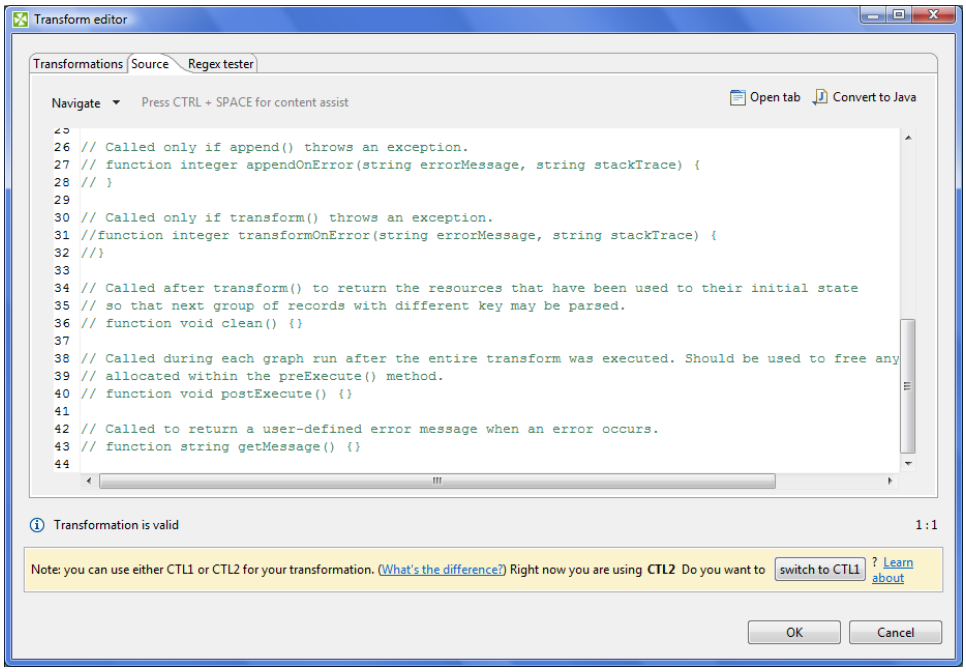


図55.2. Denormalizerコンポーネントの変換エディタの「Source」タブ(II)

表55.1. Denormalizerの関数

CTLテンプレート関数	
boolean init()	
必須	いいえ
説明	コンポーネントを初期化し、環境およびグローバル変数を設定します。
起動	1つ目のレコードを処理する前に呼び出されます。
戻り値	`true` | `false` (`false`グラフの失敗時)
integer append()	
必須	はい
入力パラメータ	なし
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	入力レコードごとに1回、繰り返し呼び出されます。
説明	同じ**キー**の値を持つ隣接する入力レコードのグループに対し、結果となる出力レコードの作成に使用する情報を追加します。入力レコードのいずれかが原因で`append()`関数が失敗し、ユーザーが別の関数(`appendOnError()`)を定義している場合、`append()`が失敗した場所で、この`appendOnError()`に処理が引き継がれます。`append()`が失敗し、ユーザーが`appendOnError()`を定義していない場合は、グラフ全体が失敗します。`append()`から`appendOnError()`にエラー・メッセージとスタック・トレースが引数として渡されます。

CTLテンプレート関数	

例	`function integer append() {`
	`  CustomersInGroup++;`
	`  myLength = length(errorCustomers);`
	`  if(!isInteger($0.OneCustomer)) {`
	`  errorCustomers = errorCustomers`
	`  + iif(myLength > 0 ,"-","")`
	`  + $0.OneCustomer;`
	`  }`
	`  customers = customers`
	`  + iif(length(customers) > 0 ," - ","")`
	`  + $0.OneCustomer;`
	`  groupNo = $GroupNo;`
	`  return CustomersInGroup;`
	`}`

integer transform()	
必須	はい
入力パラメータ	なし
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	出力レコードごとに1回、繰り返し呼び出されます。
説明	出力レコードを作成します。特定の出力レコードに対する`transform()`関数の一部が原因で`transform()`関数が失敗し、ユーザーが別の関数(`transformOnError()`)を定義している場合は、`transform()`が失敗した場所で、この`transformOnError()`に処理が引き継がれます。`transform()`が失敗し、ユーザーが`transformOnError()`を定義していない場合は、グラフ全体が失敗します。`transformOnError()`関数は、以前に処理が成功したコードから取得した、`transform()`によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースが`transformOnError()`に渡されます。

例	`function integer transform() {`
	`  $0.CustomersInGroup = CustomersInGroup;`
	`  $0.CustomersOnError = errorCustomers;`
	`  $0.Customers = customers;`
	`  $0.GroupNo = groupNo;`
	`  return OK;`
	`}`

void clean()	
必須	いいえ
入力パラメータ	なし
戻り値	void
起動	出力レコードごとに1回、繰り返し呼び出されます(**transform()**関数によって作成された後)。
説明	コンポーネントを初期設定に戻します。

CTLテンプレート関数	

例	`function void clean(){`
	`  customers = "";`
	`  errorCustomers = "";`
	`  groupNo = 0;`
	`  CustomersInGroup = 0;`
	`}`

integer appendOnError(string errorMessage, string stackTrace)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
戻り値	整数。正の整数は無視されます。0および負の値の意味は、変換の戻り値(p.283)を参照してください。
起動	`append()`が例外をスローすると呼び出されます。同じ**キー**の値を持つレコードのグループ全体について繰り返し呼び出されます。
説明	同じ**キー**の値を持つ隣接する入力レコードのグループに対し、結果となる出力レコードの作成に使用する情報を追加します。入力レコードのいずれかが原因で`append()`関数が失敗し、ユーザーが別の関数(`appendOnError()`)を定義している場合、`append()`が失敗した場所で、この`appendOnError()`に処理が引き継がれます。`append()`が失敗し、ユーザーが`appendOnError()`を定義していない場合は、グラフ全体が失敗します。`appendOnError()`関数は、以前に処理が成功した入力レコードから取得した、`append()`によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースが`appendOnError()`に渡されます。

例	`function integer appendOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  printErr(errorMessage);`
	`  return CustomersInGroup;`
	`}`

integer transformOnError(Exception exception, stackTrace)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	`transform()`が例外をスローすると呼び出されます。
説明	出力レコードを作成します。特定の出力レコードに対する`transform()`関数の一部が原因で`transform()`関数が失敗し、ユーザーが別の関数(`transformOnError()`)を定義している場合は、`transform()`が失敗した場所で、この`transformOnError()`に処理が引き継がれます。`transform()`が失敗し、ユーザーが`transformOnError()`を定義していない場合は、グラフ全体が失敗します。`transformOnError()`関数は、以前に処理が成功したコードから取得した、`transform()`によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースが`transformOnError()`に渡されます。

CTLテンプレート関数	

例	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  $0.CustomersInGroup = CustomersInGroup;`
	`  $0.ErrorFieldForTransform = errorCustomers;`
	`  $0.CustomersOnError = errorCustomers;`
	`  $0.Customers = customers;`
	`  $0.GroupNo = groupNo;`
	`  return OK;`
	`}`

string getMessage()	
必須	いいえ
説明	指定されたエラー・メッセージを出力し、ユーザーにより呼び出されます。
起動	ユーザーが指定したときに呼び出されます(`append()`、`transform()`、`appendOnError()`または`transformOnError()`のいずれかの戻り値が-2以下の場合のみ)。
戻り値	`string`
void preExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	変換で必要なリソースの割当てと初期化に使用できます。この関数内に割り当てられたすべてのリソースは、`postExecute()`関数によって解放される必要があります。
起動	変換が実行される前の各グラフの実行中に呼び出されます。
void postExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	`preExecute()`関数内に割り当てられたすべてのリソースを解放するために使用する必要があります。
起動	変換全体が実行された後の各グラフの実行中に呼び出されます。

重要

- **入力レコードまたはフィールド**

  入力レコードまたはフィールドには、`append()`関数および`appendOnError()`関数内からのみアクセスできます。

- **出力レコードまたはフィールド**

  出力レコードまたはフィールドには、`transform()`関数および`transformOnError()`関数内からのみアクセスできます。

- その他すべてのCTLテンプレート関数では入力にも出力にもアクセスできません。

警告

これらのルールに従わない場合はNPEがスローされます。

Denormalizer用Javaインタフェース

この変換では、RecordDenormalizeインタフェースのメソッドを実装し、その他の共通メソッドをTransformインタフェースから継承します。共通Javaインタフェース(p.295)を参照してください。

RecordDenormalizeインタフェースのメソッドを次に示します。

- `boolean init(Properties parameters, DataRecordMetadata sourceMetadata, DataRecordMetadata targetMetadata)`

  非正規化クラス/関数を初期化します。このメソッドは、非正規化プロセスの開始時に1回のみ呼び出されます。すべてのオブジェクト割当ておよび初期化はここで実行する必要があります。

- `int append(DataRecord inRecord)`

  1つの入力レコードを、作成中のクラスに渡します。

- `int appendOnError(Exception exception, DataRecord inRecord)`

  1つの入力レコードを、作成中のクラスに渡します。`append(DataRecord)`が例外をスローする場合にのみ呼び出されます。

- `int transform(DataRecord outRecord)`

  作成された出力レコードを取得します。戻り値およびその意味の詳細は、変換の戻り値(p.283)を参照してください。**Denormalizer**では、ALL、0、SKIPおよび**エラー・コード**のみに意味があります。

- `int transformOnError(Exception exception, DataRecord outRecord)`

  作成された出力レコードを取得します。`transform(DataRecord)`が例外をスローする場合にのみ呼び出されます。

- `void clean()`

  現在のラウンドまたは現在のラウンド後の消去をファイナライズします。入力レコードに対して変換メソッドが呼び出された後に呼び出されます。

ExtFilter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

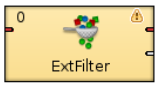
ExtFilterは、指定された条件に従って、入力レコードをフィルタリングします。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
ExtFilter	-	いいえ	1	1-2	-	-

概要

ExtFilterは、単一の入力ポートを介してデータ・レコードを受信し、指定されたフィルタ式と比較して、この式に一致したものを最初の出力ポートに送信します。拒否されたレコードは、オプションの2番目の出力ポートに送信されます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	許可されたデータ・レコード用	入力0[1)]
	1	いいえ	拒否されたデータ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。すべての出力メタデータが同じである必要があります。

ExtFilterの属性

属性	必須	説明	可能な値
Basic			
Filter expression	1)	レコードをフィルタリングする基準となる式。各入力フィールドの各式をセミコロンで区切ったシーケンスとして表されます。	
Advanced			
Filter class	1)	フィルタを通るレコードを定義する外部クラスの名前。	

説明:

1): これらの属性のいずれかを指定する必要があります。「**Filter expression**」と「**Filter class**」の両方を指定した場合は、「Filter expression」が使用されます。「**Filter class**」属性によって参照されるJavaクラスは、`RecordFilter`インタフェースを実装することが予想されます。

詳細説明

フィルタ式

このコンポーネントを選択した場合は、フィルタリングを実行する基準となる式(**フィルタ式**)を指定する必要があります。フィルタ式は、論理演算子(論理`and`および論理`or`)と優先度を表すカッコで接続した複数の副次式で構成されます。これらの副次式に対して使用できる一連の関数および比較演算子(次より大きい、次以上、次より小さい、次以下、次と等しい、次と等しくない)があります。後者は「**Filter editor**」ダイアログで数学比較記号(>、>=、<、<=、==、!=)として選択する以外に、テキスト省略形(`.gt.`、`.ge.`、`.lt.`、`.le.`、`.eq.`、`.ne.`)も使用できます。レコード・フィールドのすべての値は、それぞれのポート番号の前にドル記号を付け、ドットおよび名前で表現する必要があります。たとえば、`$0.employeeid`のようになります。

注意

ExtFilterのかわりにPartition(p.619)コンポーネントをフィルタとして使用することもできます。Partition(p.619)コンポーネントを使用すると、さらに高度なフィルタ式を定義して、より多くの出力ポートにデータ・レコードを分配できます。

あるいはReformat(p.632)コンポーネントをフィルタとして使用できます。

重要

フィルタ・エディタではCTL1かCTL2のいずれかを使用できます。

次の2つのオプションは同等です。

1. **CTL1の場合**

```
is_integer($0.field1)
```

2. **CTL2の場合**

```
//#CTL2
isInteger($0.field1)
```

ExtFilter用Javaインタフェース

フィルタ式に加えて、org.jetel.component.RecordFilterインタフェースを実装するJavaクラスによってフィルタリングを定義できます。クラスには、デフォルト(引数なし)のコンストラクタが必要です。このインタフェースは、次のメソッドで構成されます。

- `void init()`

  `isValid()`が使用される前に呼び出されます。

- `void setTransformationGraph(TranformationGraph)`

  変換グラフをフィルタ・クラス・インスタンスに関連付けます。

- `boolean isValid(DataRecord)`

  各受信データ・レコードに対して呼び出されます。この実装は、レコードがフィルタを通る場合は`true`、それ以外の場合は`false`を回答します。

ExtSort

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

ExtSortは、入力レコードをソート・キーに従ってソートします。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
ExtSort	-	✖	1	1-N	-	-

概要

ExtSortは、レコードがグラフを通過する順序を変更します。2つのレコードの比較方法は、ソート・キーによって指定されます。

ソート・キーは、1つ以上の入力フィールドおよび各フィールドのソート順(昇順または降順)で定義します。結果として得られるシーケンスは、キー・フィールドのタイプによっても異なります。「`string`」フィールドはASCIIコード順にソートされ、それ以外のフィールドはアルファベット順にソートされます。

このコンポーネントは、単一の入力ポートを介してデータ・レコードを受信し、指定されたソート・キーに従ってソートし、接続されているすべての出力ポートにそれぞれをコピーします。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	入力データ・レコード用	同じ入力および出力メタデータ [1)]
出力	0	✔	ソート済データ・レコード用	
	1-N	✖	ソート済データ・レコード用	

[1)]すべての出力メタデータが入力メタデータと同じである必要があるため、これらはこのコンポーネントを介して伝播できます。

ExtSortの属性

属性	必須	説明	可能な値
Basic			
Sort key	✔	レコードをソートする基準となるキー。詳細は、ソート・キー (p.277)を参照してください。	
Advanced			
Buffer capacity		メモリー内の解析済レコードの最大数。この数より多くの入力レコードがある場合、外部ソートが実行されます。	8000 (デフォルト) | 1-N
Number of tapes		外部ソートを実行するために使用される一時ファイルの数。2より大きい偶数です。	6 (デフォルト) | 2*(1-N)
Deprecated			
Sort order		ソート順序(`Ascending`または`Descending`)。最初の文字(`A`または`D`)のみで指定できます。すべてのキー・フィールドに対して同様です。	Ascending (デフォルト) | Descending
Sorting locale		ソートに使用する必要があるロケール。	none (デフォルト) | 任意のロケール
Case sensitive		「**Case sensitive**」のデフォルト設定(`true`)では、大文字と小文字が異なる文字としてソートされます。小文字はその大文字よりも先になります。「**Case sensitive**」を`false`に設定すると、大文字と小文字は同一の文字とみなしてソートされます。	true (デフォルト) | false
Sorter initial capacity		「**Buffer capacity**」と同じです。	8000 (デフォルト) | 1-N

詳細説明

null値のソート

ExtSortでは、「**Sort key**」属性の同じフィールドの値が`null`であるレコードを、その`null`が等しいとみなして処理します。

FastSort

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

FastSortは、ソート・キーを使用して入力レコードをソートします。**FastSort**は**ExtSort**よりも高速ですが、より多くのシステム・リソースを必要とします。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
FastSort	-	✖	1	1-N	-	-

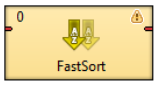
概要

FastSortはパフォーマンスの高いソート・コンポーネントであり、十分なシステム・リソースを使用できる場合に、最適な効率を実現します。**FastSort**は**ExtSort**よりも最大で2.5倍高速ですが、はるかに多くのメモリーおよび一時ディスク領域を消費します。

このコンポーネントは、入力レコードを取得して、ソート・キー(単一フィールドまたは一連のフィールド)を使用してソートします。キーの各フィールドのソート順序は個別に指定できます。ソートされた出力は、接続されているすべてのポートに送信されます。

デフォルト設定(ソート・キーの指定のみ)でも非常に良好な結果が得られます。一方で、最大のパフォーマンスを実現するために、多数のパラメータを微調整できます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	入力データ・レコード用	同じ入力および出力メタデータ [1)]
出力	0	✔	ソート済データ・レコード用	
	1-N	✖	ソート済データ・レコード用	

[1)]すべての出力メタデータが入力メタデータと同じである必要があるため、これらはこのコンポーネントを介して伝播できます。

FastSortの属性

属性	必須	説明	可能な値
Basic			
Sort key	✔	データ・レコードのソートの基準となるフィールドのリスト(セミコロンで区切られます)。各データ・フィールドを個別にソートする順序も含まれます。ソート・キー(p.277)を参照してください。	
Estimated record count	1)	ソート対象の入力レコードの見積り数。桁数の大まかな推測で十分です。Estimated Record Count(p.604)を参照してください。	auto (デフォルト) | 1-N
In memory only		`true`の場合は強制的に内部ソートが実行され、「**Sort key**」および「**Run size**」以外のすべての属性が無視されます。	false (デフォルト) | true
Advanced			
Run size (records)	1) 2)	メモリー内で一度にソートされるレコード数。多くの場合、速度およびメモリー要件に影響を与えます。Run Size(p.604)を参照してください。	auto (「**Estimated record count**」が設定されている場合にその値より算出) | 20,000 (デフォルト) | 1000 - N
Max open files		ソート中に作成可能な一時ファイルの数を制限します。数が小さすぎる(500以下)場合は、パフォーマンスが大幅に低減します。Max Open Files(p.604)を参照してください。	unlimited (デフォルト) | 1-N
Concurrency (threads)		ジョブを処理するワーカー・スレッドの数。デフォルト値で最適な結果が得られる一方で、デフォルトをオーバーライドするとグラフの実行が逆に遅くなる可能性があります。Concurrency(p.605)を参照してください。	auto (デフォルト) | 1-N
Number of read buffers	2)	メモリー内に一度に保持されるデータ・チャンクの数。Number of Read Buffers(p.605)を参照してください。	auto (デフォルト) | 1-N
Average record size (bytes)	2)	レコードの平均サイズのバイト数での推測値。Average Record Size(p.605)を参照してください。	auto (デフォルト) | 1-N
Maximum memory (MB, GB)	2)	使用可能な最大メモリーの大まかな見積り。Maximum Memory(p.605)を参照してください。	auto (デフォルト) | 1-N
Tape buffer (bytes)		ワーカーが出力を入力するために使用するバッファ。パフォーマンスに多少の影響を与えます。Tape Buffer(p.605)を参照してください。	8192 (デフォルト) | 1-N
Compress temporary files		trueの場合、一時ファイルが圧縮されます。詳細は、Compress Temporary Files(p.605)を参照してください。	false (デフォルト) | true
Deprecated			
Sorting locale		ソート順序の修正に使用されるロケール。	none (デフォルト) | 任意のロケール
Case sensitive		デフォルトでは(「**Sorting locale**」が`none`の場合)、大文字が区別してソートされ、同様に区別してソートされた小文字よりも先になります。「**Sorting locale**」が設定されている場合は、大文字	false (デフォルト) | true

属性	必須	説明	可能な値
		と小文字が一緒にソートされます。「**Case sensitive**」がtrueの場合は小文字がその大文字よりも先になり、falseの場合はデータ文字列が入力に出現した順序が保持されます。	

[1]「**Estimated record count**」は、**実行サイズ**を見積りレコード数の0.66乗の概数として(かなり不自然に)自動的に計算するヘルパー属性です。**実行サイズ**を明示的に設定すると、「**Estimated record count**」は無視されます。

適正な**実行サイズ**は、レコード・サイズおよびレコードの合計数に基づいて、5,000から200,000の範囲になります。

[2]これらの属性は**実行サイズ**の自動推測に影響を与えます。通常は次の式が成立する必要があります。

Number of read buffers * Average record size < Maximum memory

詳細説明

null値のソート

FastSortでは、「**Sort key**」属性の同じフィールドの値がnullであるレコードを、そのnullが等しいとみなして処理します。

FastSortの微調整

これらすべての属性の設定は基本的に不要ですが、設定することによってパフォーマンスを向上できる場合があります。メモリーに制限があったり、または大量のレコードや非常に大きなレコードをソートする必要がある場合があります。このような場合に、**FastSort**をその必要に合せることができます。

1. **Estimated Record Count**

   処理する必要があるレコードの概数を**FastSort**に通知する「Basic」属性です。この属性は「**Run size**」の補足であるため、「**Run size**」を指定した場合、設定は不要です。一方で、属性の設定に時間をかけたくない場合は、レコードの概数を指定することにより、グラフ実行中に割り当てられるメモリーを節約できます。この数に基づいて**最大メモリー**、レコード・サイズ、**実行サイズ**などが決定されます。

2. **Run Size**

   FastSortのコア属性であり、実行を形成するレコード数(一時ファイル内のソート済レコードの集まり)を決定します。**実行サイズ**が小さいほど多くの一時ファイルが作成され、メモリーの使用が少なくなり、速度が速くなります。一方で、値が大きくなるとメモリーの問題が発生する可能性があります。最大のパフォーマンスを得るために**実行サイズ**を大きくするか小さくするかについての一般的なルールはありません。通常は、ソートするレコードの数が多くなるほど、**実行サイズ**を大きくすることをお薦めします。**実行サイズ**を概算する式は、「**Estimated record count**^0.66」です。メモリー使用量は「**Number of read buffers**」および「**Concurrency**」とともに増加します。したがって、**実行サイズ**が大きくなるほどメモリー・フットプリントが大きくなります。

3. **Max Open Files**

   FastSortの処理中は、かなり多くの一時ファイルが使用されます。割当てまたはOS固有の制限に達した場合は、作成されるファイルの最大数を制限できます。次の表が参考になります。

データ・セット・サイズ	一時ファイルの数	デフォルトの「Run size」	注意
1,000,000	~100	~10,000	
10,000,000	~250	~45,000	
1,000,000,000	20,000-2,000	50,000-500,000	使用可能なメモリーに依存

この表の数字は厳密ではなく、システムによって異なる可能性があります。ただし、場合によっては、このような多数のファイルによってユーザー割当てなどのランタイム制限に達する問題が発生します。この問題の解決方法は、パフォーマンスのボトルネック(p.605)を参照してください。

4. **Concurrency**

   一度にいくつの実行(チャンク)を並行してソートするかを**FastSort**に通知します。デフォルトでは、システム内のCPUコアの数に基づいて1または2に自動的に設定されます。システムに多数のCPUコアがあり、使用するディスク・パフォーマンスで非常に多くのパラレル・データ・ストリームを処理できると思われる場合、この値のオーバーライドは有効です。

5. **Maximum Memory**

   単一のコンポーネント専用に割り当てるメモリーの最大量を設定できます。これは、**FastSort**の**実行サイズ**を計算するときの目安になります。**実行サイズ**が明示的に設定された場合、この設定は無視されます。200MB、1gbのように単位を指定する必要があります。

6. **Average Record Size**

   「**Average record size**」をバイト単位で設定できます。設定を省略した場合は、初めから1000個分の解析済レコードの平均レコード・サイズが計算されます。

7. **Number of Read Buffers**

   この設定はスレッド数(「**Concurrency**」)と緊密に関係します。「**Concurrency**」の値以上にする必要があります。読取りバッファが多くなるほど、ワーカーによって相互にブロックされる変更が少なくなります。デフォルトは「**Concurrency + 2**」です。

8. **Compress Temporary Files**

   このオプションを**一時ファイル・キャラクタ・セット**とともに使用して、一時ファイルに必要になる領域を減少できます。圧縮によって多くの領域を節約できますが、パフォーマンスが最大30%低下する影響を与えるため、この設定には注意してください。

9. **Tape Buffer**

   ファイル出力バッファのサイズ(バイト)。デフォルト値は8KBです。この値を小さくすることにより、多数の実行に使用されるメモリーの不足を回避できる場合があります(レコードの合計数に比べて**実行サイズ**が非常に小さい場合など)。ただし、この設定の影響はかなり小さなものです。

ヒントおよびポイント

- 必ず、Java仮想マシン(JVM)に専用のメモリーが十分にあることを確認してください。使用可能なメモリーが十分にある場合、**FastSort**は強力な処理能力を発揮します。デフォルトである64MBのJVMヒープ領域では、**FastSort**がクラッシュする可能性があります。メモリーの値を最大2GBまで増加してください(ただし、オペレーティング・システム用に一部のメモリーを残します)。これは価値があります。JVMの設定方法は、プログラムとVM引数(p.85)の項を参照してください。

パフォーマンスのボトルネック

- 大きなレコードのソート(長い文字列フィールド、数十または数百のフィールドなど): **FastSort**はメモリーとCPUコアの両方を膨大に消費します。これらの両方がシステムに十分ではない場合、メモリー不足で**FastSort**が簡単にクラッシュする可能性があります。この場合は、かわりに**ExtSort**コンポーネントを使用してください。

- 2個を超えるCPUコアの使用: 非常に高速なディスク・ドライブを使用できる場合を除き、「**Concurrency**」のデフォルト値を2よりも大きなスレッド数にオーバーライドしても有効でない場合があります。スレッドが追加されるたびに余分なメモリーがロードされるため、逆にプロセスが少し遅くなる場合があります。

- 割当ておよびその他のランタイム制限の取扱い: 複数のパラレル・ソートがある複雑なグラフで、他のグラフ・コンポーネントでも膨大な数のファイルが開いている場合は、`Too many open files`エラーが発生し、グラフの実行が失敗する可能性があります。この問題に対処できる2つの解決策は次のとおりです。

  1. 制限(割当て)を増加します。

     速度を犠牲にしないために、本番システムにお薦めするオプションです。通常は、UNIXシステムで制限を大きな数字に設定します。

  2. 一時ファイルの数を特定の制限未満に強制的に維持するように**FastSort**を設定します。

     大規模なサーバーで作業する一般ユーザーは割当ての増加を選択できません。このため、**「Max open files」**を適正な値に設定する必要があります。これにより、**FastSort**によって一時ファイルが中間マージされ、その数が制限以下に維持されます。ただし、そのようなマージが不可避となる値に**「Max open files」**を設定すると、多くの場合はパフォーマンスが大幅に低下します。そのため、可能な範囲で最も大きい値を維持してください。データ・セットが大きくても**FastSort**の一時ファイルを100未満に制限する必要がある場合は、パフォーマンスのためにテープの数を制限するように設計されている**ExtSort**をかわりに使用することを検討してください。

Merge

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Mergeは、2つ以上の入力からのデータ・レコードをマージおよびソートします。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Merge	はい	はい	2-n	1	-	-

概要

Mergeは、2つ以上の入力ポートを介して、ソート済のデータ・レコードを受信します。(すべての入力ポートのメタデータが同じである必要があります。)すべての入力レコードを収集し、出力で同様にソートします。

重要

すべてのキー・フィールドを昇順でソートする必要があります。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0-1	はい	入力データ・レコード用	任意
	2-n	いいえ	入力データ・レコード用	入力0[1)]
出力	0	はい	マージされたデータ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。すべての出力メタデータが同じである必要があります。

Mergeの属性

属性	必須	説明	可能な値
Basic			
Merge key	はい	ソート済のレコードをマージする基準となるキー。(すべてのキー・フィールドを昇順でソートする必要があります。)詳細は、グループ・キー(p.276)を参照してください。[1)]	
Equal NULL		デフォルトでは、キー・フィールドの値がnullのレコードは異なるとみなされます。`true`に設定すると、等しいとみなされます。	false (デフォルト) | true

説明:

1): メタデータはこのコンポーネントを介して伝播できます。すべての出力メタデータが同じである必要があります。

MetaPivot

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

MetaPivotは、すべての受信レコードを、それぞれが入力の単一フィールドを表す複数の出力レコードに変換します。

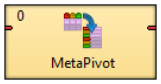
コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
MetaPivot	-	いいえ	1	1	いいえ	いいえ

概要

MetaPivotは、その単一の入力ポートでソートが不要なデータを受信します。入力レコードの各フィールドは、出力で新しい行として書き込まれます。メタデータはデータ型を表し、固定形式に制限されます。詳細説明(p.610)を参照してください。一般的に、**MetaPivot**は、レコードをわかりやすいデータ依存構造に効果的に変換するために使用できます。

MetaPivotの派生元であるNormalizer(p.612)とは異なり、変換の定義はありません。**MetaPivot**では、常に同じ変換が行われます。つまり、入力レコードを取得し、項目を回転させることによって、入力列が出力行になります。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意1
出力	0	はい	変換されたデータ・レコード用	任意2

MetaPivotの属性

MetaPivotには、コンポーネント固有の属性はありません。

詳細説明

重要

MetaPivotの操作には、固定形式の出力メタデータを使用する必要があります。各メタデータ・フィールドは特定のデータ型を表します。フィールド名およびデータ型は、正確に次のとおりに設定する必要があります(そうしないと、予期しない`BadDataFormatException`が発生します)。

`[recordNo long]`: レコードのシリアル番号(後で出力をこの番号ごとにグループ化できます)。同じレコードのフィールドは同じ番号を共有します(図55.4「MetaPivot出力の例」(p.611)を参照してください)。

`[fieldNo integer]`: 現在のフィールド番号(0...n-1)。nは入力メタデータのフィールド数です。

`[fieldName string]`: 入力に表示されたフィールド名。

`[fieldType string]`: フィールドのタイプ。「string」、「date」、「decimal」など。

`[valueBoolean boolean]`: フィールドのブール値。

`[valueByte byte]`: フィールドのバイト値。

`[valueDate date]`: フィールドの日付値。

`[valueDecimal decimal]` : フィールドの小数値。

`[valueInteger integer]`: フィールドの整数値。

`[valueLong long]`: フィールドのLong値。

`[valueNumber number]`: フィールドの数値。

`[valueString string]`: フィールドの文字列値。

MetaPivotによって生成される出力レコードの合計数は、(入力レコードの数) * (入力フィールドの数)に等しくなります。

フィールドの中には、単に出力の外観を調整しているものがあります。これらのフィールドは、必要に応じて省略できます。出力メタデータに含める必要がないフィールドは、「`recordNo`」、「`fieldNo`」および「`fieldType`」の3つのみです。

例55.4. MetaPivot変換の例

MetaPivotがデータをどのように処理するかを次に示します。様々なデータ型のデータが含まれている、デリミタ付きファイルがあるとします。レコードは2つのみです。

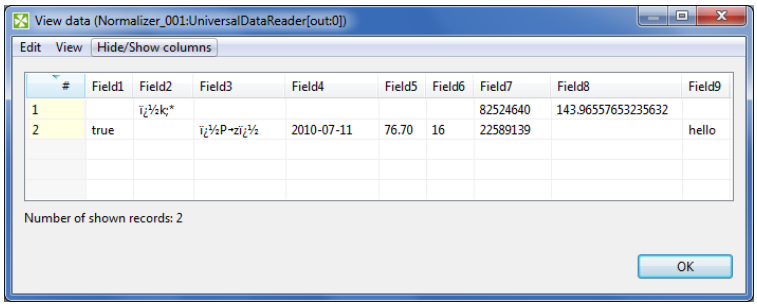
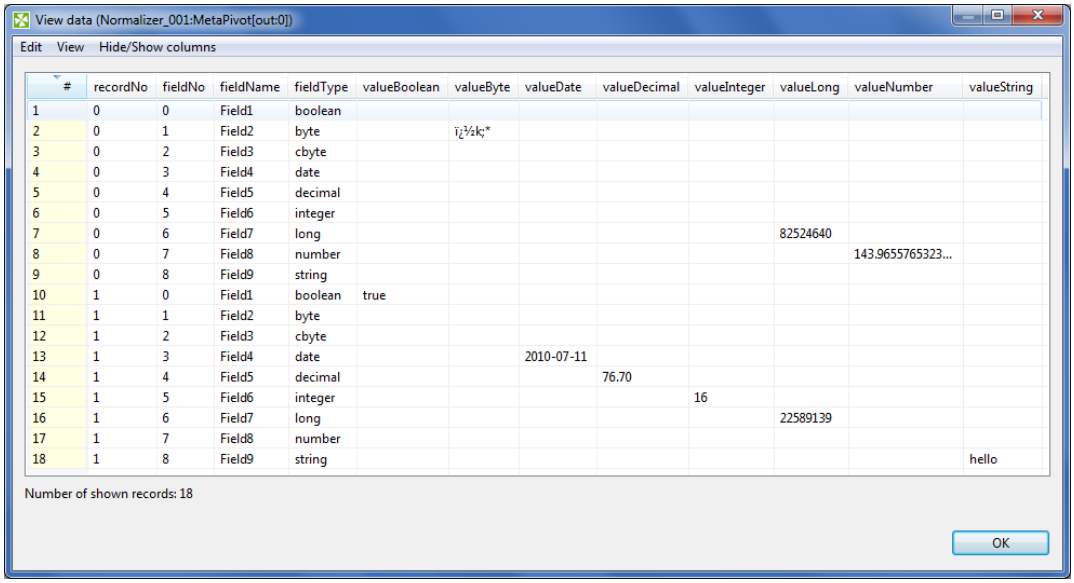


図55.3. MetaPivot入力の例

これらのデータを**MetaPivot**に送信すると、それぞれのデータ型に従ってデータが出力フィールドに分類されます。



#	recordNo	fieldNo	fieldName	fieldType	valueBoolean	valueByte	valueDate	valueDecimal	valueInteger	valueLong	valueNumber	valueString
1	0	0	Field1	boolean								
2	0	1	Field2	byte		ï¿½k; *						
3	0	2	Field3	cbyte								
4	0	3	Field4	date								
5	0	4	Field5	decimal								
6	0	5	Field6	integer								
7	0	6	Field7	long						82524640		
8	0	7	Field8	number							143.9655765323...	
9	0	8	Field9	string								
10	1	0	Field1	boolean	true							
11	1	1	Field2	byte								
12	1	2	Field3	cbyte								
13	1	3	Field4	date			2010-07-11					
14	1	4	Field5	decimal				76.70				
15	1	5	Field6	integer					16			
16	1	6	Field7	long						22589139		
17	1	7	Field8	number								
18	1	8	Field9	string								hello

図55.4. MetaPivot出力の例

このように、たとえば、「**valueString**」フィールドにhello、「**valueDecimal**」に76.70が配置されます。入力には2レコードと9フィールドが存在したため、出力は18レコードになります。

Normalizer

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Normalizerは、単一の入力レコードのそれぞれから1つ以上の出力レコードを作成します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Normalizer	-	いいえ	1	1	はい	はい

概要

Normalizerは、ソートされていない可能性のあるデータを単一の入力ポートを介して受信し、入力データ・レコードを分解して、各入力レコードから1つ以上の出力レコードを作成します。

変換を定義する必要があります。変換は**Normalizer**用のCTLテンプレートを使用するか、`RecordNormalize`インタフェースを実装するか、`DataRecordNormalize`スーパークラスから継承します。インタフェース・メソッドをこの後に示します。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意1
出力	0	はい	正規化済データ・レコード用	任意2

Normalizerの属性

属性	必須	説明	可能な値
Basic			
Normalize	1)	レコードを正規化する方法の定義。グラフにCTLまたはJavaで記述します。	
Normalize URL	1)	CTLまたはJavaで記述された、レコードの正規化方法の定義が含まれる、外部ファイルの名前(パスを含む)。	
Normalize class	1)	レコードを正規化する方法を定義する外部クラスの名前。	
Normalize source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Deprecated			
Error actions		指定した変換が**エラー・コード**を返した場合に実行する必要があるアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、**コンソール**に書き出されます。	

説明:

1): これらのいずれかを指定する必要があります。これらの変換属性はいずれも**Normalizer**用のCTLテンプレートを使用するか、`RecordNormalize`インタフェースを実装します。

詳細は、CTLスクリプトの詳細(p.613)またはNormalizer用Javaインタフェース(p.618)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

CTLスクリプトの詳細

3つの変換属性のいずれを定義する場合も、入力を出力に変換する方法を指定する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム変換を指定できます。

変換を作成したら、タブの右上隅にある該当するボタンをクリックすることで、それをJava言語コードに変換することもできます。

Normalizer用CTLテンプレート

変換を定義する「**Source**」タブは次のようになります。

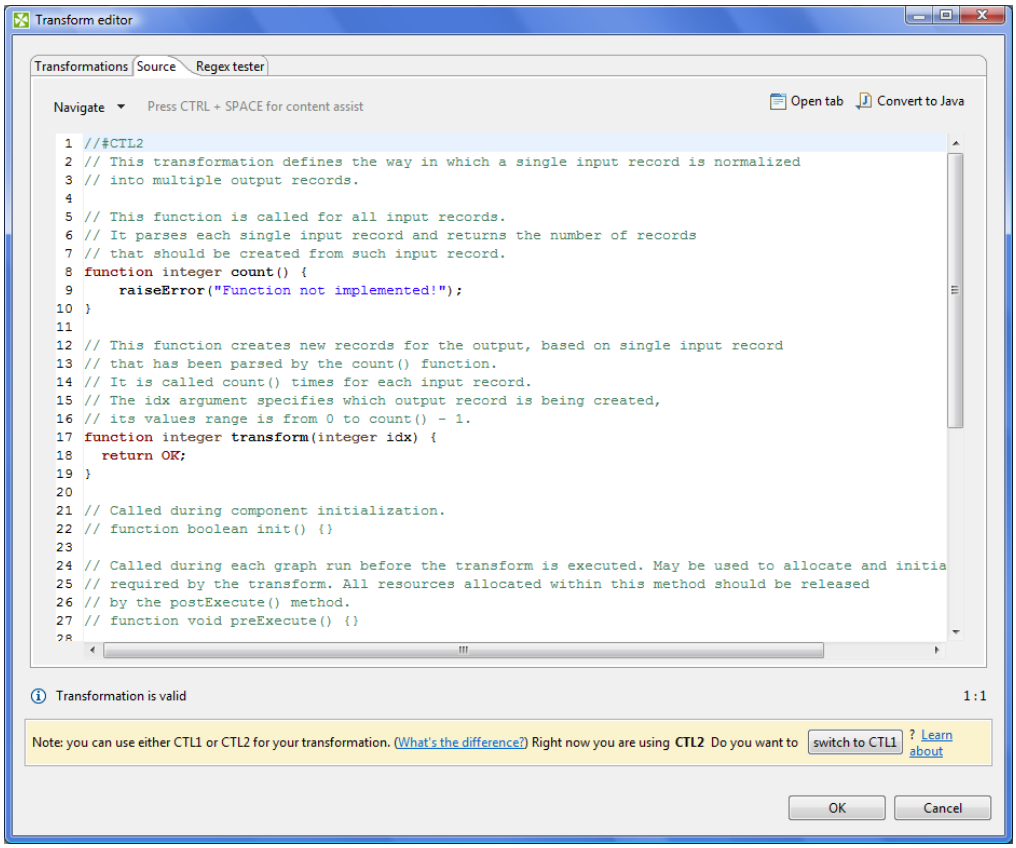


図55.5. Normalizerコンポーネントの変換エディタの「Source」タブ(I)

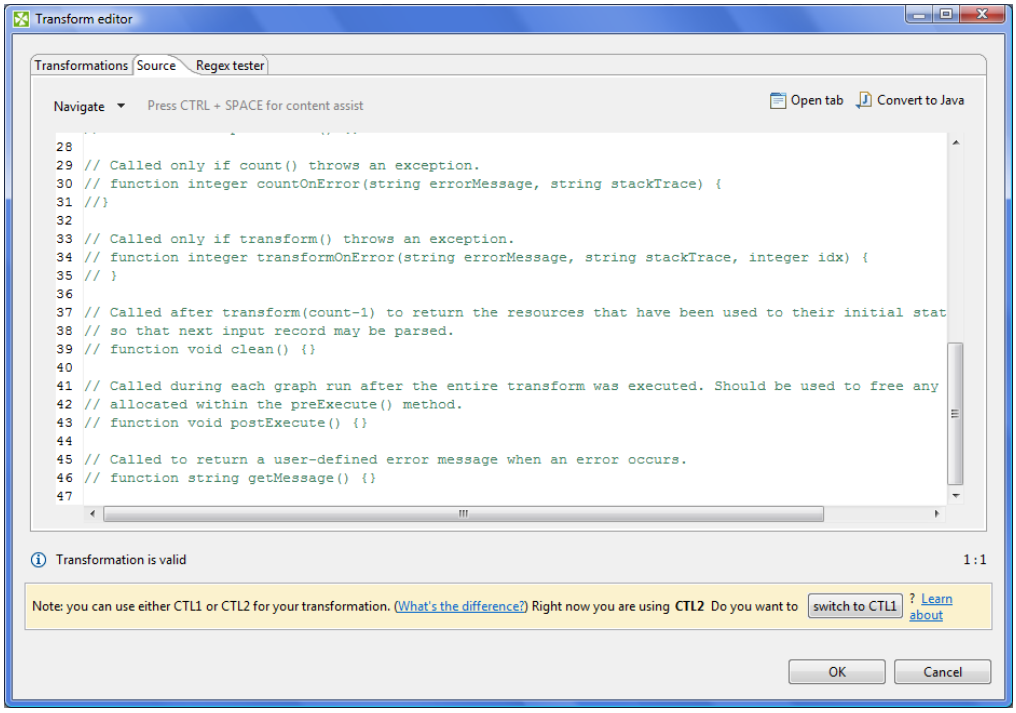


図55.6. Normalizerコンポーネントの変換エディタの「Source」タブ(II)

表55.2. Normalizerの関数

CTLテンプレート関数	
boolean init()	
必須	いいえ
説明	コンポーネントを初期化し、環境およびグローバル変数を設定します。
起動	1つ目のレコードを処理する前に呼び出されます。
戻り値	`true` | `false` (`false`グラフの失敗時)
integer count()	
必須	はい
入力パラメータ	なし
戻り値	入力レコードごとに0より大きい整数を1つ返します。返される数値は、**transform()**関数により作成される新規出力レコードの数と同じです。
起動	入力レコードごとに1回、繰り返し呼び出されます。
説明	入力レコードごとに、この入力から作成される出力レコードの数を生成します。入力レコードのいずれかが原因で`count()`関数が失敗し、ユーザーが別の関数(`countOnError()`)を定義している場合、`count()`が失敗した場所で、この`countOnError()`に処理が引き継がれます。`count()`が失敗し、ユーザーが`countOnError()`を定義していない場合は、グラフ全体が失敗します。`countOnError()`関数は、以前に処理が成功した入力レコードから取得した、`count()`によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースが`countOnError()`に渡されます。

例	`function integer count() {`
	`  customers = split($0.customers,"-");`
	`  return length(customers);`
	`}`

integer transform(integer idx)	
必須	はい
入力パラメータ	`integer idx`は、0から`count-1`の整数(`count`は**transform()**関数によって返された数です。)
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	出力レコードごとに1回、繰り返し呼び出されます。
説明	出力レコードを作成します。特定の出力レコードに対する`transform()`関数の一部が原因で`transform()`関数が失敗し、ユーザーが別の関数(`transformOnError()`)を定義している場合は、`transform()`が失敗した場所で、この`transformOnError()`に処理が引き継がれます。`transform()`が失敗し、ユーザーが`transformOnError()`を定義していない場合は、グラフ全体が失敗します。`transformOnError()`関数は、以前に処理が成功したコードから取得した、`transform()`によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースが`transformOnError()`に渡されます。

CTLテンプレート関数	

例	`function integer transform(integer idx) {`
	`  myString = customers[idx];`
	`  $0.OneCustomer = str2integer(myString);`
	`  $0.RecordNo = $0.recordNo;`
	`  $0.OrderWithinRecord = idx;`
	`  return OK;`
	`}`

void clean()	
必須	いいえ
入力パラメータ	なし
戻り値	void
起動	入力レコードごとに1回、繰り返し呼び出されます(入力レコードから最後の出力レコードが作成された後)。
説明	コンポーネントを初期設定に戻します。

例	`function void clean() {`
	`  clear(customers);`
	`}`

integer countOnError(string errorMessage, string stackTrace)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
戻り値	入力レコードごとに0より大きい整数を1つ返します。返される数値は、**transform()**関数により作成される新規出力レコードの数と同じです。
起動	`count()`が例外をスローすると呼び出されます。
説明	入力レコードごとに、この入力から作成される出力レコードの数を生成します。入力レコードのいずれかが原因で`count()`関数が失敗し、ユーザーが別の関数(`countOnError()`)を定義している場合、`count()`が失敗した場所で、この`countOnError()`に処理が引き継がれます。`count()`が失敗し、ユーザーが`countOnError()`を定義していない場合は、グラフ全体が失敗します。`countOnError()`関数は、以前に処理が成功した入力レコードから取得した、`count()`によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースが`countOnError()`に渡されます。

例	`function integer countOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  printErr(errorMessage);`
	`  return 1;`
	`}`

integer transformOnError(string errorMessage, string stackTrace, integer idx)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
	`integer idx`

CTLテンプレート関数	
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	transform()が例外をスローすると呼び出されます。
説明	出力レコードを作成します。特定の出力レコードに対するtransform()関数の一部が原因でtransform()関数が失敗し、ユーザーが別の関数(transformOnError())を定義している場合は、transform()が失敗した場所で、このtransformOnError()に処理が引き継がれます。transform()が失敗し、ユーザーがtransformOnError()を定義していない場合は、グラフ全体が失敗します。transformOnError()関数は、以前に処理が成功したコードから取得した、transform()によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースがtransformOnError()に渡されます。

例	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  integer idx) {`
	`  printErr(errorMessage);`
	`  printErr(stackTrace);`
	`  $0.OneCustomerOnError = customers[idx];`
	`  $0.RecordNo = $recordNo;`
	`  $0.OrderWithinRecord = idx;`
	`  return OK;`
	`}`

string getMessage()	
必須	いいえ
説明	指定されたエラー・メッセージを出力し、ユーザーにより呼び出されます。
起動	ユーザーが指定したときに呼び出されます(count()、transform()、countOnError()またはtransformOnError()のいずれかの戻り値が-2以下の場合のみ)。
戻り値	string
void preExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	void
説明	変換で必要なリソースの割当てと初期化に使用できます。この関数内に割り当てられたすべてのリソースは、postExecute()関数によって解放される必要があります。
起動	変換が実行される前の各グラフの実行中に呼び出されます。
void postExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	void

CTLテンプレート関数	
説明	preExecute()関数内に割り当てられたすべてのリソースを解放するために使用する必要があります。
起動	変換全体が実行された後の各グラフの実行中に呼び出されます。

重要

- **入力レコードまたはフィールド**

  入力レコードまたはフィールドには、count()関数およびcountOnError()関数内からのみアクセスできます。

- **出力レコードまたはフィールド**

  出力レコードまたはフィールドには、transform()関数およびtransformOnError()関数内からのみアクセスできます。

- その他すべてのCTLテンプレート関数では入力にも出力にもアクセスできません。

警告

これらのルールに従わない場合はNPEがスローされます。

Normalizer用Javaインタフェース

この変換では、RecordNormalizeインタフェースのメソッドを実装し、その他の共通メソッドをTransformインタフェースから継承します。共通Javaインタフェース(p.295)を参照してください。

RecordNormalizeインタフェースのメソッドを次に示します。

- `boolean init(Properties parameters, DataRecordMetadata sourceMetadata, DataRecordMetadata targetMetadata)`

  正規化クラス/関数を初期化します。このメソッドは、正規化プロセスの開始時に1回のみ呼び出されます。すべてのオブジェクト割当ておよび初期化はここで実行する必要があります。

- `int count(DataRecord source)`

  指定された入力レコードから作成される出力レコードの数を返します。

- `int countOnError(Exception exception, DataRecord source)`

  count(DataRecord)が例外をスローするときのみ呼び出されます。

- `int transform(DataRecord source, DataRecord target, int idx)`

  idxは、出力レコードの順序番号です(0から開始)。戻り値およびその意味の詳細は、変換の戻り値(p.283)を参照してください。**Normalizer**では、ALL、0、SKIPおよび**エラー・コード**のみに意味があります。

- `int transformOnError(Exception exception, DataRecord source, DataRecord target, int idx)`

  transform(DataRecord, DataRecord, int)が例外をスローするときのみ呼び出されます。

- `void clean()`

  現在のラウンドまたは現在のラウンド後の消去をファイナライズします。入力レコードに対して変換メソッドが呼び出された後に呼び出されます。

Partition

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Partitionは、異なる出力ポートに各入力データ・レコードを分配します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Partition	-	いいえ	1	1-n	はい/いいえ[1)]	はい/いいえ[1)]

説明

1) **Partition**では、変換またはその他の2つの属性(「**Ranges**」および「**Partition key**」)のいずれかを使用できます。これらの属性が1つ以上指定されていない場合は、変換を定義する必要があります。

概要

Partitionは、異なる出力ポートに各入力データ・レコードを分配します。

データ・レコードの分配には、ユーザー定義の変換、「**Partition key**」の範囲またはラウンドロビン・アルゴリズムを使用できます。**Partition**用のCTLテンプレートを使用するか、`PartitionFunction`インタフェースを実装します。そのメソッドを次に示します。このコンポーネントでは入力データ・レコードが変更されないため、マッピングは定義しません。変更を加えずに出力ポート間で分配するのみです。

ヒント

Partitionコンポーネントは**ExtFilter**と同様のフィルタとして使用できます。**Partition**コンポーネントを使用すると、さらに高度なフィルタ式を定義して、2個を超える出力ポート間で入力データ・レコードを分配できます。

Partitionでも**ExtFilter**でもレコードは変更できません。

重要

Partitionはパフォーマンスの高いコンポーネントであるため、入力レコードおよび出力レコードの変更はできません。変更するとエラーが発生します。必要な場合は、かわりに**Reformat**を使用することを検討してください。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	出力データ・レコード用	入力0[1)]
	1-N	いいえ	出力データ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。

Partitionの属性

属性	必須	説明	可能な値
Basic			
Partition	1)	CTLまたはJavaでグラフに記述された、レコードを出力ポート間で分配する方法の定義。	
Partition URL	1)	CTLまたはJavaで記述された、レコードを出力ポート間で分配する方法の定義が含まれる、外部ファイルの名前(パスを含む)。	
Partition class	1)	レコードを出力ポート間で分配する方法を定義する外部クラスの名前。	
Ranges	1),2)	個々の範囲をセミコロンで区切ったシーケンスとして表される範囲。個々の範囲は、デリミタなしで相互に隣接する一連のフィールドに対する間隔のシーケンスです。また、その境界が間隔に含まれるか含まれないかについても、山カッコと丸カッコによってそれぞれ表現します。「**Ranges**」の例: `<1,9)(,31.12.2008);<1,9)<31.12.2008,);<9,)(,31.12.2008); <9,)<31.12.2008).`	
Partition key	1),2)	入力レコードを異なる出力ポート間で分配する基準となるキー。各入力フィールド名をセミコロンで区切ったシーケンスとして表されます。「**Partition key**」の例: `first_name;last_name.`	
Advanced			
Partition source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト) | その他のエンコーディング

属性	必須	説明	可能な値
Deprecated			
Locale		国際化がtrueに設定されている場合に使用される**ロケール**。デフォルトでは、defaultPropertiesファイルに指定されている「**Locale**」の値がコメント解除され、必要なロケールに設定されていないかぎり、システム値が使用されます。defaultPropertiesで**ロケール**を変更する方法の詳細は、デフォルトのCloverETL設定の変更(p.88)を参照してください。	システム値または指定されたデフォルト値 (デフォルト) | その他のロケール
Use internationalization		デフォルトでは、国際化は使用されません。trueに設定すると、各国のプロパティに応じたソートが実行されます。	false (デフォルト) | true

説明:

1): これらの変換属性の1つを指定した場合、「**Ranges**」と「**Partition key**」はどちらも優先度が低いため無視されます。これらの変換属性はいずれも**Partition**用のCTLテンプレートを使用するか、PartitionFunctionインタフェースを実装する必要があります。

詳細は、CTLスクリプトの詳細(p.621)またはPartition (およびclusterpartition)用Javaインタフェース(p.625)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

2): 変換属性を定義しない場合は、「**Ranges**」および「**Partition key**」を次の3つの方法のいずれかで使用します。

- 「**Ranges**」と「**Partition key**」の両方を設定します。

  フィールドの値が、指定された範囲の境界内にあるレコードが、同じ出力ポートに送信されます。出力ポートの番号は、フィールドのすべての値内の範囲の順序に一致します。

- 「**Ranges**」は定義しません。「**Partition key**」のみを設定します。

  「**Partition key**」フィールドの値が同じであるすべてのレコードが同じポートに送信されるように、出力ポート間でレコードが分配されます。

  出力ポート番号は、出力ポートの数を法とするキー・フィールドから計算したハッシュ値として決定されます。

- 「**Ranges**」も「**Partition key**」も定義しません。

  レコードは、ラウンドロビン・アルゴリズムを使用して出力ポート間に分配されます。

CTLスクリプトの詳細

オプションである3つの変換属性のいずれかを定義する場合は、出力ポート番号を各入力レコードに割り当てる変換を指定する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム変換を指定できます。

Partitionでは、次のような変換テンプレートが使用されます。

Partition (またはclusterpartition)用CTLテンプレート

次の変換テンプレートは、**Partition**および**clusterpartition**で使用されます。

CTLで変換を記述した後、タブの右上隅にある対応するボタンをクリックすることによってJava言語コードに変換することもできます。

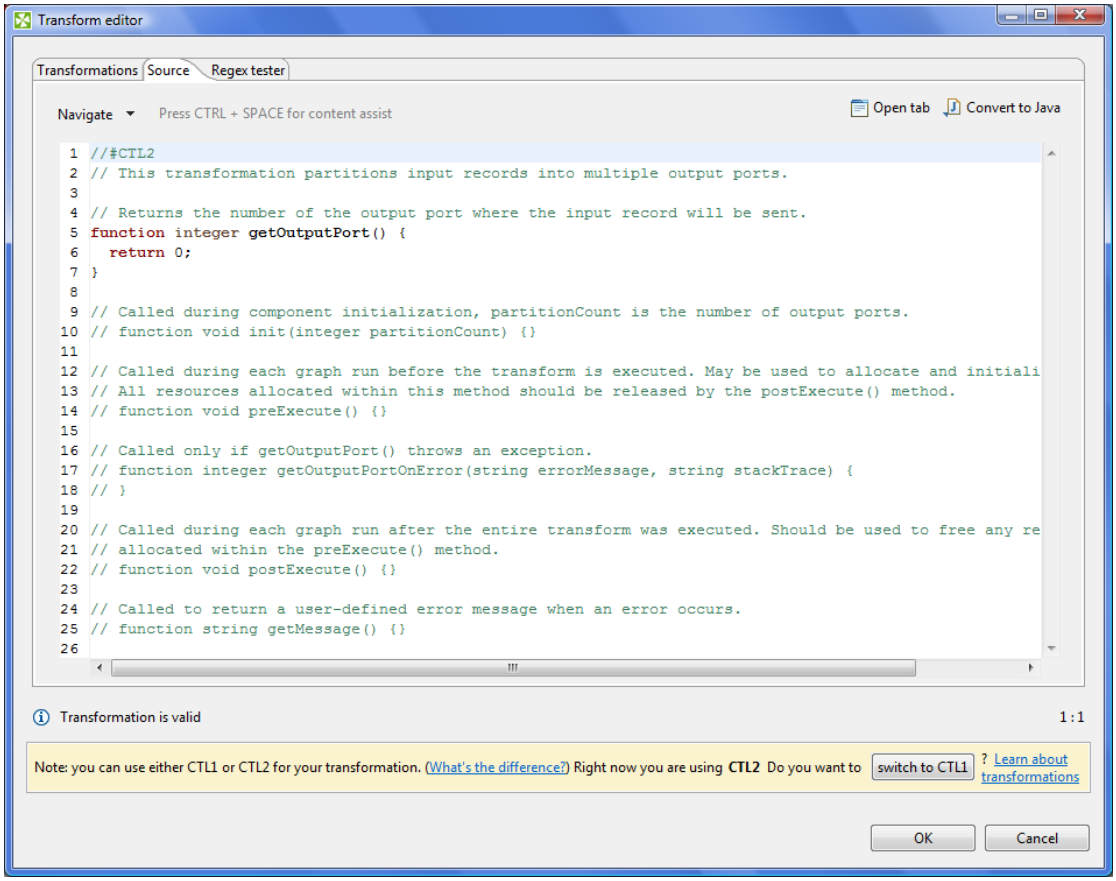


図55.7. Partitionコンポーネントの変換エディタの「Source」タブ

タブの右上隅にある該当するボタンをクリックすると、変換定義を(**グラフ・エディタ**の「**Graph**」タブおよび「**Source**」タブとは)別のグラフのタブで開くことができます。

表55.3. Partition (またはclusterpartition)の関数

CTLテンプレート関数	
void init()	
必須	いいえ
説明	コンポーネントを初期化し、環境およびグローバル変数を設定します。
起動	1つ目のレコードを処理する前に呼び出されます。
戻り値	`void`
integer getOutputPort()	
必須	はい
入力パラメータ	なし
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	入力レコードごとに繰り返し呼び出されます。

CTLテンプレート関数	
説明	レコードの変換、変更および削除は行わず、整数値を返すのみです。返される数値はそれぞれ、個々のレコードを送信する出力ポートの番号です。**clusterpartition**では、これらは仮想ポート、つまりクラスタ・ノードです。特定の出力レコードに対する`getOutputPort()`関数の一部が原因で`getOutputPort()`関数が失敗し、ユーザーが別の関数(`getOutputPortOnError()`)を定義している場合は、`getOutputPort()`が失敗した場所で、この`getOutputPortOnError()`に処理が引き継がれます。`getOutputPort()`が失敗し、ユーザーが`getOutputPortOnError()`を定義していない場合は、グラフ全体が失敗します。`getOutputPortOnError()`関数は、以前に処理が成功したコードから取得した、`getOutputPort()`によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースが`getOutputPortOnError()`に渡されます。
例	`function integer getOutputPort() { switch (expression) { case const0 : return 0; break; case const1 : return 1; break; ... case constN : return N; break; [default : return N+1;] } }`
integer getOutputPortOnError(string errorMessage, string stackTrace)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	`getOutputPort()`が例外をスローすると呼び出されます。
説明	レコードの変換、変更および削除は行わず、整数値を返すのみです。返される数値はそれぞれ、個々のレコードを送信する出力ポートの番号です。**clusterpartition**では、これらは仮想ポート、つまりクラスタ・ノードです。特定の出力レコードに対する`getOutputPort()`関数の一部が原因で`getOutputPort()`関数が失敗し、ユーザーが別の関数(`getOutputPortOnError()`)を定義している場合は、`getOutputPort()`が失敗した場所で、この`getOutputPortOnError()`に処理が引き継がれます。`getOutputPort()`が失敗し、ユーザーが`getOutputPortOnError()`を定義していない場合は、グラフ全体が失敗します。`getOutputPortOnError()`関数は、以前に処理が成功したコードから取得した、`getOutputPort()`によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースが`getOutputPortOnError()`に渡されます。

CTLテンプレート関数	

例	`function integer getOutputPortOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  printErr(errorMessage);`
	`  printErr(stackTrace);`
	`}`

string getMessage()	
必須	いいえ
説明	指定されたエラー・メッセージを出力し、ユーザーにより呼び出されます。
起動	ユーザーが指定したときに呼び出されます(`getOutputPort()`または`getOutputPotOnError()`のいずれかの戻り値が-2以下の場合のみ)。
戻り値	`string`
void preExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	リソースの割当てと初期化に使用できます。この関数内に割り当てられたすべてのリソースは、`postExecute()`関数によって解放される必要があります。
起動	変換が実行される前の各グラフの実行中に呼び出されます。
void postExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	`preExecute()`関数内に割り当てられたすべてのリソースを解放するために使用する必要があります。
起動	変換全体が実行された後の各グラフの実行中に呼び出されます。

重要

- **入力レコードまたはフィールド**

  入力レコードまたはフィールドには、`getOutputPort()`関数および`getOutputPortOnError()`関数内からのみアクセスできます。

- **出力レコードまたはフィールド**

  レコードは変更およびマッピングなしで出力にマッピングされるため、出力レコードおよびフィールドにはアクセスできません。

- その他すべてのCTLテンプレート関数では入力にも出力にもアクセスできません。

警告

これらのルールに従わない場合はNPEがスローされます。

Partition (およびclusterpartition)用Javaインタフェース

この変換では、PartitionFunctionインタフェースのメソッドを実装し、その他の共通メソッドをTransformインタフェースから継承します。共通Javaインタフェース(p.295)を参照してください。

PartitionFunctionインタフェースのメソッドを次に示します。

- `void init(int numPartitions,RecordKey partitionKey)`

  getOutputPort()が使用される前に呼び出されます。numPartitions引数は、作成する必要があるパーティションの数を指定します。RecordKey引数は、パーティションを決定する基準となるキーを作成する一連のフィールドです。

- `boolean supportsDirectRecord()`

  シリアライズされたレコード、つまり直接レコードに対する操作がパーティション関数でサポートされるかどうかを示します。getOutputPort(ByteBuffer)メソッドを呼び出せる場合はtrueを返します。

- `int getOutputPort(DataRecord record)`

  データの送信に使用するポート番号を返します。戻り値およびその意味の詳細は、変換の戻り値(p.283)を参照してください。

- `int getOutputPortOnError(Exception exception, DataRecord record)`

  データの送信に使用するポート番号を返します。getOutputPort(DataRecord)が例外をスローするときのみ呼び出されます。

- `int getOutputPort(ByteBuffer directRecord)`

  データの送信に使用するポート番号を返します。戻り値およびその意味の詳細は、変換の戻り値(p.283)を参照してください。

- `int getOutputPortOnError(Exception exception, ByteBuffer directRecord)`

  データの送信に使用するポート番号を返します。getOutputPort(ByteBuffer)が例外をスローする場合にのみ呼び出されます。

LoadBalancingPartition

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

LoadBalancingPartitionは、ダウンストリーム・コンポーネントのワークロードに従って、受信入力データ・レコードを複数の出力ポートに分配します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
LoadBalancingPartition	-	いいえ	1	1-n	いいえ	いいえ

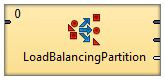
概要

LoadBalancingPartitionは、すべての接続済出力コンポーネントのワークロードに従って、受信入力データ・レコードを複数の出力ポートに分配します。

各受信レコードは、接続済出力ポートのいずれかに送信されます。出力ポートは、接続済コンポーネントの速度に従って選択されます。実際、コンポーネントは出力ポートごとに個別の作業スレッドを起動し、そのスレッドが単一の入力ポートから受信データ・レコードを同時に読み取り、専用の出力ポートに送信します。

説明済のアルゴリズムについて、異なるエッジの実装およびその結果を検討します。たとえば、ダイレクト・エッジ実装には数百から数千レコード用のキャッシュがあるため、少量のデータ・レコードのみを処理する変換では、すべての受信レコードを単一の出力ブランチに送信できます。システム・スレッド・スケジューラによって、すべてのデータが単一スレッドで処理されます。通常、このコンポーネントは大量のデータ・レコードの場合にのみ効果があります。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	出力データ・レコード用	入力0[1)]
	1-N	いいえ	出力データ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。

Pivot

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Pivotは、ピボット・テーブルを生成します。このコンポーネントは、入力レコードのグループごとにデータ・サマリー・レコードを作成します。グループは、キーまたはサイズのいずれかによって識別できます。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Pivot	-	いいえ	1	1	はい	はい

注意: Key属性を使用する場合は、入力レコードをソートしておく必要があります。詳細説明(p.630)を参照してください。

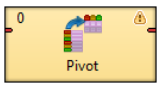
概要

このコンポーネントは入力レコードを読み取り、それらをグループとして処理します。1つのグループは、キー、またはそのグループを形成する多数のレコードで定義されます。**Pivot**は、グループから1つのレコードを生成します。つまり、このコンポーネントによってピボット・テーブルが作成されます。

Pivotには、一部の入力値を出力フィールド名、その他の入力を出力値として処理するよう指示する2つの主要な属性があります。

このコンポーネントは、Denormalizer(p.589)の単純形です。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意1
出力	0	はい	サマリー・データ・レコード用	任意2

Pivotの属性

属性	必須	説明	可能な値
Basic			
Key	1)	キーは、入力レコードのグループを識別するために使用する一連のフィールドです(複数のフィールドで1つのキーを形成できます)。グループは、同一のキー値を持つレコードのシーケンスで形成されます。	任意の入力フィールド
Group size	1)	1つのグループを形成する入力レコードの数。「Group size」を使用する場合、入力データをソートする必要はありません。**Pivot**は、複数のレコードを読み取り、それらを1つのグループに変換します。この数は単なる「Group size」の値です。	<1; n>
Field defining output field name	2)	出力のフィールド名にマップされる値を持つ入力フィールド。	
Field defining output field value	2)	出力のフィールド値にマップされる値を持つ入力フィールド。	
Sort order		入力レコードのグループは、ここで定義された順序でソートされます。意味はDenormalizerでの意味と同じです。Sort order(p.591)を参照してください。Pivotでは、入力がソートされていない場合、これを`Ignore`に設定すると予期しない結果が生成される可能性があります。	Auto (デフォルト) | Ascending | Descending | Ignore
Equal NULL		null値を含む2つのフィールドが一致とみなされるかどうかを指定します。	true (デフォルト) | false
Advanced			
Pivot transformation	3)	CTLまたはJavaを使用して、ここに独自のレコード変換を記述できます。	
Pivot transformation URL	3)	レコードを変換する方法を定義する外部ファイルへのパス。変換はCTLまたはJavaで記述できます。	
Pivot transformation class	3)	データ変換に使用されるクラスの名前。これはJavaで記述できます。	
Pivot transformation source charset		データ変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト) | 任意
Deprecated			
Error actions		指定した変換が**エラー・コード**を返した場合に実行する必要があるアクションを定義します。変換の戻り値(p.283)を参照してください。	
Error log		エラー・メッセージを書き込むファイルのURL。これらのメッセージは、**エラー・アクション**中に生成されます(前述)。この属性が設定されていない場合、メッセージは**コンソール**に書き出されます。	

説明:

1): 「**Key**」属性または「**Group size**」属性のいずれかは常に設定する必要があります。

2): これらの2つの値は、属性として指定するか独自の変換内で指定できます。

3): 「**Field defining output field name**」および「**Field defining output field value**」を使用して変換を制御しない場合は、これらの属性のいずれかを設定する必要があります。

詳細説明

データ変換は、次の2つの方法で定義できます。

1) 「**Key**」属性または「**Group size**」属性を設定します。属性の設定によるデータのグループ化(p.630)を参照してください。

2) CTLまたはJavaで独自の変換を記述するか、それを外部ファイルまたはJavaクラスで指定します。独自変換の定義: Java/CTL(p.631)を参照してください。

属性の設定によるデータのグループ化

「**Key**」属性を使用してデータをグループ化する場合は、**Key**値に基づいて入力をソートする必要があります。入力のソート方法をコンポーネントに通知するには、「**Sort order**」を指定します。出力メタデータにも「**Key**」のフィールドがある場合、**Key**値は自動的にコピーされます。

「**Group size**」属性を使用してグループ化する場合、コンポーネントはデータ自体を無視し、たとえば3つのレコードを取得して(Group size = 3の場合)、それらを1つのグループとして処理します。このため、十分な数の入力レコードを用意しておく必要があり、そうでない場合は読取りでエラーが発生します。この数値は「**Group size**」の倍数とする必要があります(Group size = 3の場合は3、6、9など)。

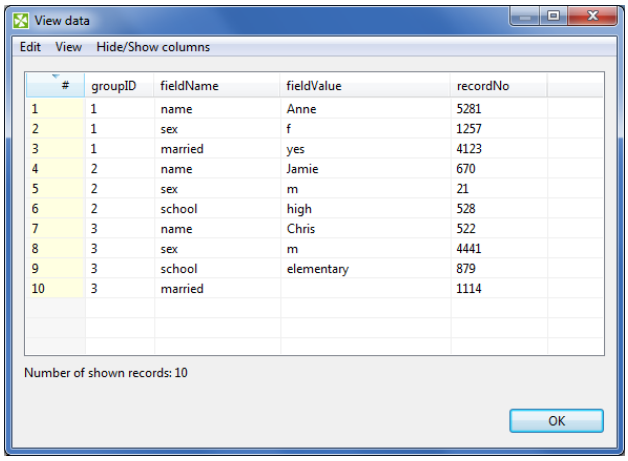
さらに、マッピングを記述する2つの主要な属性があります。これらは次のとおりです。

- 入力フィールドの値が出力フィールドを指定する場合: **Field defining output field name**
- 入力フィールドの値がそのフィールドの値として使用される場合: **Field defining output field value**

出力メタデータについては任意ですが、フィールド名に固定されています。入力データに追加フィールドがある場合、それらは単に無視されます(値/名前として定義されたフィールドのみ)。対応する入力レコードのない出力フィールドと同様、nullになります。

例55.5. Pivotによるデータ変換: キーの使用

カンマ区切り値を含む次のような入力テキスト・ファイルがあるとします。



#	groupID	fieldName	fieldValue	recordNo
1	1	name	Anne	5281
2	1	sex	f	1257
3	1	married	yes	4123
4	2	name	Jamie	670
5	2	sex	m	21
6	2	school	high	528
7	3	name	Chris	522
8	3	sex	m	4441
9	3	school	elementary	879
10	3	married		1114

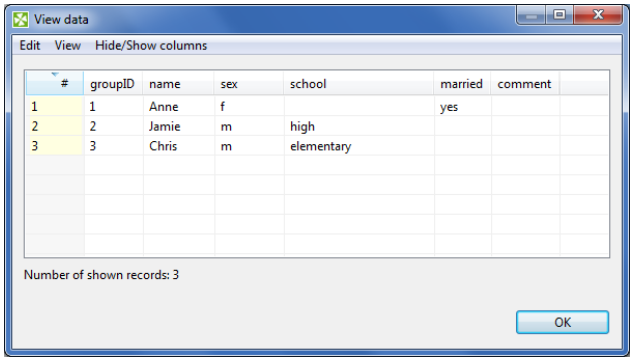
`groupID`フィールドに基づいてデータをグループ化するため、入力をソートする必要があります(`groupID`の昇順)。**Pivot**コンポーネントでは、次のような設定を行います。

Key = `groupID` (同じgroupIDですべての入力レコードをグループ化するため)

Field defining output field name = `fieldName` (この入力フィールドから出力フィールドの名前を取得するように設定するため)

Field defining output field value = `fieldValue` (この入力フィールドから出力フィールドの値を取得するように設定するため)

Pivotを使用してデータを処理すると、次の出力が生成されます。



入力のrecordNoフィールドは無視されています。同様に、出力のcommentには、対応する入力のフィールドがありません。このため、これはnullのままです。groupIDは出力メタデータに含まれるため、自動的にコピーされています。

注意

入力がソートされていない場合は(例とは異なり)、レコードを数に基づいてグループ化すると特に有効です。「**Key**」を省略してかわりに「**Group size**」を設定し、必要なレコード数と同数のレコードのシーケンスを読み取ります。

独自変換の定義: Java/CTL

Pivotでは、独自の変換関数を記述できます。これは、CTLまたはJavaで行うことができます。Pivotの属性(p.629)の「Advanced属性」を参照してください。

変換を記述する前に、次の内容に関連するいくつかの項を参照することをお薦めします。

- 変換の定義(p.279)
- **Denormalizer**での変換の記述。コンポーネント**Pivot**は、CTLスクリプトの詳細(p.591)およびDenormalizer用Javaインタフェース(p.596)から派生しています。

Java

Denormalizerと比較すると、**Pivot**コンポーネントには新しい重要な属性であるnameFieldおよびvalueFieldがあります。これらは、属性として(前述)、またはメソッドで定義できます。変換が定義されていない場合、このコンポーネントはvalueFieldからnameFieldに値をコピーするcom.opensys.cloveretl.component.pivot.DataRecordPivotTransformを使用します。

Javaでは、DataRecordPivotTransformをオーバーライドする独自のPivotTransformを実装できます。ただし、オーバーライドできるのは1つのメソッドのみ(getOutputFieldValue、getOutputFieldIndex、またはPivotTransform (RecordDenormalizeを拡張する)のその他のメソッド)です。

CTL

CTL1/2でも、属性のいずれかを設定して、メソッドでもう一方の属性を実装できます。このため、たとえばvalueFieldを設定して、getOutputFieldIndexを実装することなどができます。また、たとえばnameFieldを設定して、getOutputFieldValueを実装することなどができます。

コンパイル・モードでは、getOutputFieldValueメソッドおよびgetOutputFieldValueonErrorメソッドはオーバーライドできません。変換がCTLで記述されている場合、デフォルトのappendメソッドおよびtransformメソッドは、常にユーザー定義メソッドよりも前に実行されます。

さらに理解を深めるには、**変換エディタ**でメソッドのドキュメントを直接確認してください。

Reformat

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Reformatは、レコードの構造またはコンテンツを操作します。

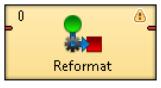
コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Reformat	-	✖	1	1-N	✔	✔

概要

Reformatは、単一の入力ポートを介してソートされていない可能性のあるデータを受信し、ユーザーが指定した方法に従って各データを変換すると、生成されるレコードをユーザーが指定したポートに送信します。この変換の戻り値は、データ・レコードが送信される出力ポートの数です。

変換を定義する必要があります。この変換では、**Reformat**用のCTLテンプレートを使用し、`RecordTransform`インタフェースを実装するか、`DataRecordTransform`スーパークラスから継承します。インタフェース・メソッドをこの後に示します。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	✔	入力データ・レコード用	任意(In0)
出力	0	✔	変換されたデータ・レコード用	任意(Out0)
	1-n	✖	変換されたデータ・レコード用	任意(OutPortNo)

Reformatの属性

属性	必須	説明	可能な値
Basic			
Transform	1)	レコードを再フォーマットする方法の定義。グラフのソースにCTLまたはJavaで書き込まれます。	
Transform URL	1)	CTLまたはJavaで記述されたレコードの再フォーマット方法の定義を含む、外部ファイルの名前(パスを含む)。	
Transform class	1)	レコードを再フォーマットする方法を定義する外部クラスの名前。	
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Deprecated			
Error actions		指定した変換が**エラー・コード**を返した場合に実行する必要があるアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、コンソールに書き出されます。	

説明:

1): これらのいずれかを指定する必要があります。これらの変換属性はいずれも**Reformat**用のCTLテンプレートを使用するか、`RecordTransform`インタフェースを実装します。

詳細は、CTLスクリプトの詳細(p.633)またはReformat用Javaインタフェース(p.634)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

Reformatの使用目的

- 不要なフィールドの削除
- 関数または正規表現(p.963)を使用するフィールドの検証
- 新しいフィールドの計算または既存のフィールドの変更
- データ型の変換

CTLスクリプトの詳細

3つの変換属性のいずれを定義する場合も、いくつかの出力ポートを各入力レコードに割り当てる変換を指定する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム変換を指定できます。

Reformat用CTLテンプレート

Reformatは、**DataIntersection**および**ジョイナ**と同じ変換テンプレートを使用します。詳細は、ジョイナ用CTLテンプレート(p.325)を参照してください。

Reformat用Javaインタフェース

Reformatは、**DataIntersection**および**ジョイナ**と同じインタフェースを実装します。詳細は、ジョイナ用Javaインタフェース(p.328)を参照してください。

Rollup

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

Rollupは、1つ以上の入力レコードから1つ以上の出力レコードを作成します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Rollup	-	いいえ	1	1-N	はい	はい

概要

Rollupは、単一の入力ポートを介してソートされていない可能性のあるデータを受信し、それらを変換して、1つ以上の入力レコードから1つ以上の出力レコードを作成します。

コンポーネントは、ユーザーが指定した複数の出力ポートに複数のレコードを送信できます。

変換を定義する必要があります。この変換では、**Rollup**用のCTLテンプレートを使用し、`RecordRollup`インタフェースを実装するか、`DataRecordRollup`スーパークラスから継承します。インタフェース・メソッドをこの後に示します。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意(In0)
出力	0	はい	出力データ・レコード用	任意(Out0)
	1-N	いいえ	出力データ・レコード用	任意(Out1-N)

Rollupの属性

属性	必須	説明	可能な値
Basic			
Group key		レコードが1つのグループに含まれているとみなす基準となるキー。各入力フィールド名をセミコロンで区切ったシーケンスとして表されます。詳細は、グループ・キー(p.276)を参照してください。指定しなかった場合は、すべてのレコードが単一グループのメンバーとみなされます。	
Group accumulator		グループ・アキュムレータの作成に使用するメタデータのID。メタデータは、データ・レコードの個々のグループの変換に使用する値を格納するために使用します。	no metadata (デフォルト) | 任意のメタデータ
Transform	1)	CTLまたはJavaでグラフに記述される変換の定義。	
Transform URL	1)	CTLまたはJavaで記述される変換の定義が格納されている外部ファイルの名前(パスを含む)。	
Transform class	1)	変換を定義する外部クラスの名前。	
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Sorted input		デフォルトでは、レコードはソート済とみなされます。昇順または降順のいずれかです。フィールドが異なると、ソート順も異なる場合があります。レコードがソートされていない場合は、この属性を`false`に切り替えます。	true (デフォルト) | false
Equal NULL		デフォルトでは、キー・フィールドの値がnullのレコードは等しいとみなされます。`false`に設定すると、それらは互いに異なるとみなされます。	true (デフォルト) | false

説明:

1): これらのいずれかを指定する必要があります。これらの変換属性はいずれも**Rollup**用のCTLテンプレートを使用するか、`RecordRollup`インタフェースを実装します。

詳細は、CTLスクリプトの詳細(p.636)またはRollup用Javaインタフェース(p.645)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

CTLスクリプトの詳細

3つの変換属性のいずれを定義する場合も、入力を出力に変換する方法を指定する必要があります。

変換は、`RecordRollup`インタフェースを実装するか、`DataRecordRollup`スーパークラスから継承します。`RecordRollup`インタフェースのメソッドのリストを次に示します。このインタフェースの詳細は、Rollup用Javaインタフェース(p.645)を参照してください。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム変換を指定できます。

変換を作成したら、タブの右上隅にある該当するボタンをクリックすることで、それをJava言語コードに変換することもできます。

タブの右上隅にある該当するボタンをクリックすると、変換定義を(**グラフ・エディタ**の「**Graph**」タブおよび「**Source**」タブとは)別のグラフのタブで開くことができます。

Rollup用CTLテンプレート

変換を定義するための「**Source**」タブの表示例を次に示します。

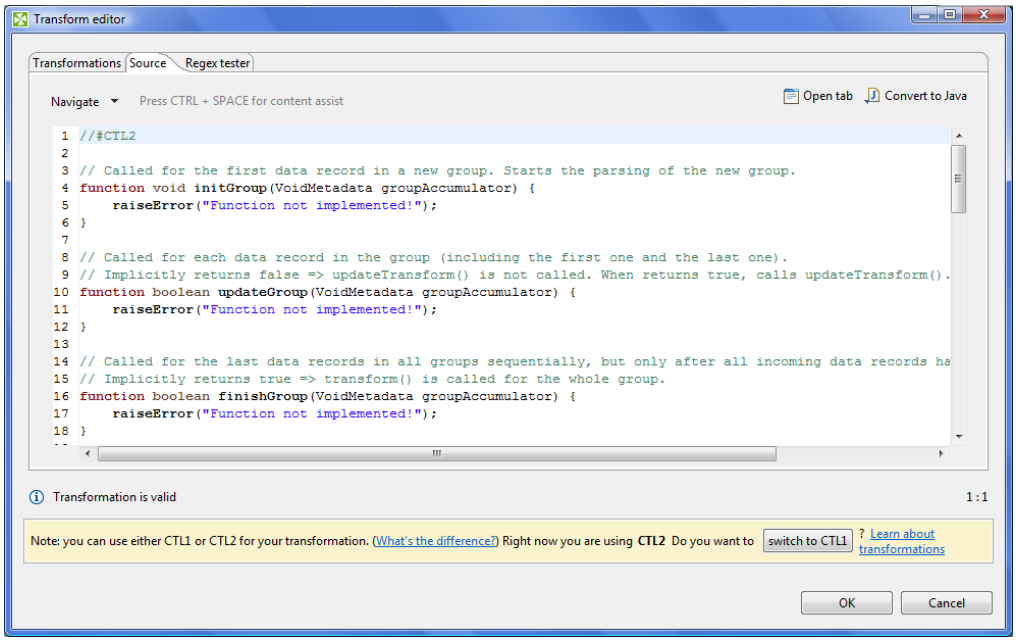


図55.8. Rollupコンポーネントの変換エディタの「Source」タブ(I)

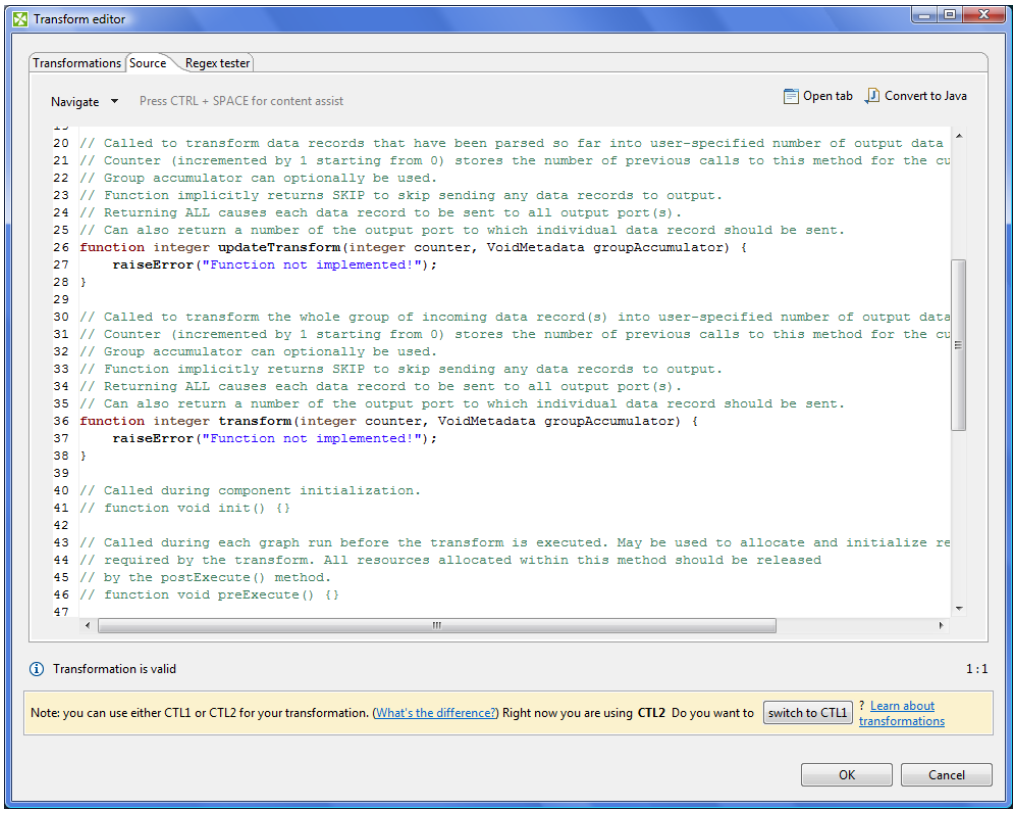


図55.9. Rollupコンポーネントの変換エディタの「Source」タブ(II)

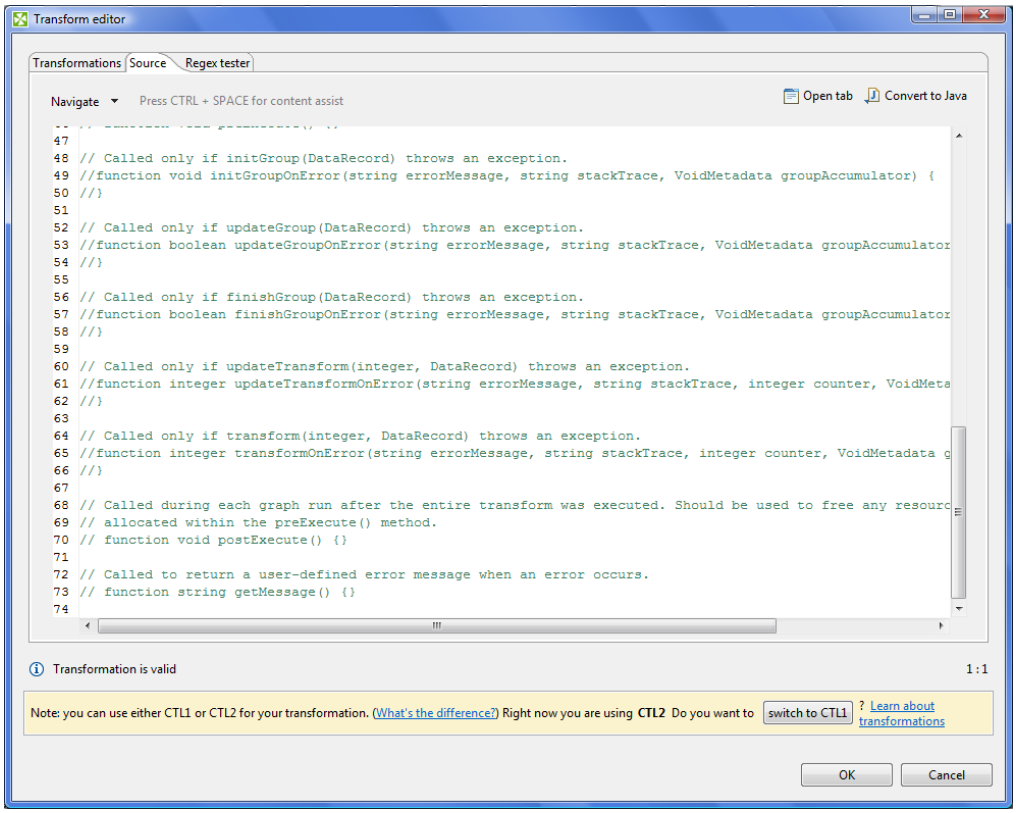


図55.10. Rollupコンポーネントの変換エディタの「Source」タブ(III)

表55.4. Rollupの関数

CTLテンプレート関数	
void init()	
必須	いいえ
説明	コンポーネントを初期化し、環境およびグローバル変数を設定します。
起動	1つ目のレコードを処理する前に呼び出されます。
戻り値	`void`
void initGroup(<metadata name> groupAccumulator)	
必須	はい
入力パラメータ	`<metadata name> groupAccumulator` (ユーザーにより指定されるメタデータ)。`groupAccumulator`が定義されていない場合、`VoidMetadata Accumulator`が関数の署名で使用されます。
戻り値	void
起動	各グループの最初の入力レコードごとに1回、繰り返し呼び出されます。**updateGroup(groupAccumulator)**より前に呼び出されます。
説明	特定のグループの情報を初期化します。

例	`function void initGroup(`
	`  companyCustomers groupAccumulator) {`
	`  groupAccumulator.count = 0;`
	`  groupAccumulator.totalFreight = 0;`
	`}`

CTLテンプレート関数	
boolean updateGroup(<metadata name> groupAccumulator)	
必須	はい
入力パラメータ	<metadata name> groupAccumulator (ユーザーにより指定されるメタデータ)。groupAccumulatorが定義されていない場合、VoidMetadata Accumulatorが関数の署名で使用されます。
戻り値	false (**updateTransform(counter,groupAccumulator)**が呼び出されない場合) | true (**updateTransform(counter,groupAccumulator)**が呼び出された場合)
起動	グループ全体に対して**initGroup(groupAccumulator)**関数が呼び出された後、繰り返し呼び出されます(最初および最後のレコードを含め、グループの入力レコードごとに1回)。
説明	特定のグループの情報を更新します。入力レコードのいずれかが原因でupdateGroup()関数が失敗し、ユーザーが別の関数(updateGroupOnError())を定義している場合、updateGroup()が失敗した場所で、このupdateGroupOnError()に処理が引き継がれます。updateGroup()が失敗し、ユーザーがupdateGroupOnError()を定義していない場合は、グラフ全体が失敗します。updateGroup()は、エラー・メッセージおよびスタック・トレースを引数としてupdateGroupOnError()に渡します。

例	`function boolean updateGroup(`
	`  companyCustomers groupAccumulator) {`
	`  groupAccumulator.count++;`
	`  groupAccumulator.totalFreight =`
	`  groupAccumulator.totalFreight`
	`  + $0.Freight;`
	`  return true;`
	`}`

boolean finishGroup(<metadata name> groupAccumulator)	
必須	はい
入力パラメータ	<metadata name> groupAccumulator (ユーザーにより指定されるメタデータ)。groupAccumulatorが定義されていない場合、VoidMetadata Accumulatorが関数の署名で使用されます。
戻り値	true (**transform(counter,groupAccumulator)**が呼び出された場合) | false (**transform(counter,groupAccumulator)**が呼び出されない場合)
起動	各グループの最後の入力レコードごとに1回、繰り返し呼び出されます。グループのすべての入力レコードに対して**updateGroup(groupAccumulator)**が呼び出された後、呼び出されます。
説明	グループ情報をファイナライズします。入力レコードのいずれかが原因でfinishGroup()関数が失敗し、ユーザーが別の関数(finishGroupOnError())を定義している場合、finishGroup()が失敗した場所で、このfinishGroupOnError()に処理が引き継がれます。finishGroup()が失敗し、ユーザーがfinishGroupOnError()を定義していない場合は、グラフ全体が失敗します。finishGroup()は、エラー・メッセージおよびスタック・トレースを引数としてfinishGroupOnError()に渡します。

CTLテンプレート関数	

例	`function boolean finishGroup(`
	`  companyCustomers groupAccumulator) {`
	`  groupAccumulator.avgFreight =`
	`  groupAccumulator.totalFreight`
	`  / groupAccumulator.count;`
	`  return true;`
	`}`

integer updateTransform(integer counter, <metadata name> groupAccumulator)	
必須	はい
入力パラメータ	`integer counter` (0から始まり、作成されたレコードの数を示します。次の例に示すように終了する必要があります。`SKIP`が返されると関数の呼出しが終了します。)
	`<metadata name> groupAccumulator` (ユーザーにより指定されるメタデータ)。`groupAccumulator`が定義されていない場合、`VoidMetadata Accumulator`が関数の署名で使用されます。
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	ユーザーに指定されたときに繰り返し呼び出されます。**updateGroup(groupAccumulator)**が`true`を返した後に呼び出されます。関数は`SKIP`が返されるまで呼び出されます。
説明	これは、個々のレコード情報に基づいて出力レコードを作成します。特定の出力レコードに対する`transform()`関数の一部が原因で`updateTransform()`関数が失敗し、ユーザーが別の関数(`updateTransformOnError()`)を定義している場合は、`updateTransform()`が失敗した場所で、この`updateTransformOnError()`に処理が引き継がれます。`updateTransform()`が失敗し、ユーザーが`updateTransformOnError()`を定義していない場合は、グラフ全体が失敗します。`updateTransformOnError()`関数は、以前に処理が成功したコードから取得される`updateTransform()`によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースが`updateTransformOnError()`に渡されます。

例	`function integer updateTransform(`
	`  integer counter,`
	`  companyCustomers groupAccumulator) {`
	`  if (counter >= Length) {`
	`  clear(customers);`
	`  return SKIP;`
	`  }`
	`  $0.customers = customers[counter];`
	`  $0.EmployeeID = $0.EmployeeID;`
	`  return ALL;`
	`}`

integer transform(integer counter, <metadata name> groupAccumulator)	
必須	はい

CTLテンプレート関数	
入力パラメータ	integer counter (0から始まり、作成されたレコードの数を示します。次の例に示すように終了する必要があります。SKIPが返されると関数の呼出しが終了します。)
	<metadata name> groupAccumulator (ユーザーにより指定されるメタデータ)。groupAccumulatorが定義されていない場合、VoidMetadata Accumulatorが関数の署名で使用されます。
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	ユーザーに指定されたときに繰り返し呼び出されます。**finishGroup(groupAccumulator)**がtrueを返した後に呼び出されます。関数はSKIPが返されるまで呼び出されます。
説明	これは、グループ全体のすべてのレコードに基づいて出力レコードを作成します。特定の出力レコードに対するtransform()関数の一部が原因でtransform()関数が失敗し、ユーザーが別の関数(transformOnError())を定義している場合は、transform()が失敗した場所で、このtransformOnError()に処理が引き継がれます。transform()が失敗し、ユーザーがtransformOnError()を定義していない場合は、グラフ全体が失敗します。transformOnError()関数は、以前に処理が成功したコードから取得した、transform()によって収集された情報を取得します。さらに、エラー・メッセージとスタック・トレースがtransformOnError()に渡されます。

例	`function integer transform(`
	`  integer counter,`
	`  companyCustomers groupAccumulator) {`
	`  if (counter > 0) return SKIP;`
	`  $0.ShipCountry = $0.ShipCountry;`
	`  $0.Count = groupAccumulator.count;`
	`  $0.AvgFreight =`
	`  groupAccumulator.avgFreight;`
	`  return ALL;`
	`}`

void initGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)	
必須	いいえ
入力パラメータ	string errorMessage
	string stackTrace
	<metadata name> groupAccumulator (ユーザーにより指定されるメタデータ)。groupAccumulatorが定義されていない場合、VoidMetadata Accumulatorが関数の署名で使用されます。
戻り値	void
起動	initGroup()が例外をスローすると呼び出されます。
説明	特定のグループの情報を初期化します。

例	`function void initGroupOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  companyCustomers groupAccumulator)`
	`  printErr(errorMessage);`
	`}`

CTLテンプレート関数	
boolean updateGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
	`<metadata name> groupAccumulator` (ユーザーにより指定されるメタデータ)。`groupAccumulator`が定義されていない場合、`VoidMetadata Accumulator`が関数の署名で使用されます。
戻り値	false (**updateTransform(counter,groupAccumulator)**が呼び出されない場合) | true (**updateTransform(counter,groupAccumulator)**が呼び出された場合)
起動	グループのレコードに対して`updateGroup()`が例外をスローした場合に呼び出されます。繰り返し呼び出されます(グループの他の入力レコードごとに1回)。
説明	特定のグループの情報を更新します。

例	`function boolean updateGroupOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  companyCustomers groupAccumulator) {`
	`  printErr(errorMessage);`
	`  return true;`
	`}`

boolean finishGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
	`<metadata name> groupAccumulator` (ユーザーにより指定されるメタデータ)。`groupAccumulator`が定義されていない場合、`VoidMetadata Accumulator`が関数の署名で使用されます。
戻り値	true (**transform(counter,groupAccumulator)**が呼び出された場合) | false (**transform(counter,groupAccumulator)**が呼び出されない場合)
起動	`finishGroup()`が例外をスローすると呼び出されます。
説明	グループ情報をファイナライズします。

例	`function boolean finishGroupOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  companyCustomers groupAccumulator) {`
	`  printErr(errorMessage);`
	`  return true;`
	`}`

integer updateTransformOnError(string errorMessage, string stackTrace, integer counter, <metadata name> groupAccumulator)	
必須	はい

CTLテンプレート関数	
入力パラメータ	`string errorMessage`
	`string stackTrace`
	`integer counter` (0から始まり、作成されたレコードの数を示します。次の例に示すように終了する必要があります。`SKIP`が返されると関数の呼出しが終了します。)
	`<metadata name> groupAccumulator` (ユーザーにより指定されるメタデータ)。`groupAccumulator`が定義されていない場合、`VoidMetadata Accumulator`が関数の署名で使用されます。
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	`updateTransform()`が例外をスローすると呼び出されます。
説明	これは、個々のレコード情報に基づいて出力レコードを作成します。

例	`function integer updateTransformOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  integer counter,`
	`  companyCustomers groupAccumulator) {`
	`  if (counter >= 0) {`
	`  return SKIP;`
	`  }`
	`  printErr(errorMessage);`
	`  return ALL;`
	`}`

integer transformOnError(string errorMessage, string stackTrace, integer counter, <metadata name> groupAccumulator)	
必須	いいえ
入力パラメータ	`string errorMessage`
	`string stackTrace`
	`integer counter` (0から始まり、作成されたレコードの数を示します。次の例に示すように終了する必要があります。`SKIP`が返されると関数の呼出しが終了します。)
	`<metadata name> groupAccumulator` (ユーザーにより指定されるメタデータ)。`groupAccumulator`が定義されていない場合、`VoidMetadata Accumulator`が関数の署名で使用されます。
戻り値	整数。詳細は、変換の戻り値(p.283)を参照してください。
起動	`transform()`が例外をスローすると呼び出されます。
説明	これは、グループ全体のすべてのレコードに基づいて出力レコードを作成します。

CTLテンプレート関数	

例	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  integer counter,`
	`  companyCustomers groupAccumulator) {`
	`  if (counter >= 0) {`
	`  return SKIP;`
	`  }`
	`  printErr(errorMessage);`
	`  return ALL;`
	`}`

string getMessage()	
必須	いいえ
説明	指定されたエラー・メッセージを出力し、ユーザーにより呼び出されます。
起動	ユーザーにより任意の時点で呼び出されます (`updateTransform()`、`transform()`、`updateTransformOnError()`または`transformOnError()`のいずれかが-2以下の値を返した場合にのみ呼び出されます)。
戻り値	`string`
void preExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	変換で必要なリソースの割当てと初期化に使用できます。この関数内に割り当てられたすべてのリソースは、`postExecute()`関数によって解放される必要があります。
起動	変換が実行される前の各グラフの実行中に呼び出されます。
void postExecute()	
必須	いいえ
入力パラメータ	なし
戻り値	`void`
説明	`preExecute()`関数内に割り当てられたすべてのリソースを解放するために使用する必要があります。
起動	変換全体が実行された後の各グラフの実行中に呼び出されます。

重要

- **入力レコードまたはフィールド**

  入力レコードまたはフィールドには、`initGroup()`、`updateGroup()`、`finishGroup()`、`initGroupOnError()`、`updateGroupOnError()`および`finishGroupOnError()`の各関数内からアクセスできます。

  これらには、`updateTransform()`、`transform()`、`updateTansformOnError()`および`transformOnError()`の各関数内からもアクセスできます。

- **出力レコードまたはフィールド**

  出力レコードまたはフィールドには、updateTransform()、transform()、updateTansformOnError()およびtransformOnError()の各関数内からアクセスできます。

- **Group accumulator**

  Group accumulatorには、initGroup()、updateGroup()、finishGroup()、initGroupOnError()、updateGroupOnError()およびfinishGroupOnError()の各関数内からアクセスできます。

  これには、updateTransform()、transform()、updateTansformOnError()およびtransformOnError()の各関数内からもアクセスできます。

- その他のすべてのCTLテンプレート関数では入力にも出力にもgroupAccumulatorにもアクセスできません。

警告

これらのルールに従わない場合はNPEがスローされます。

Rollup用Javaインタフェース

この変換は、RecordRollupインタフェースのメソッドを実装し、Transformインタフェースからその他の共通メソッドを継承します。共通Javaインタフェース(p.295)を参照してください。

RecordRollupインタフェースのメソッドを次に示します。

- `void init(Properties parameters, DataRecordMetadata inputMetadata, DataRecordMetadata accumulatorMetadata, DataRecordMetadata[] outputMetadata)`

  ロールアップ変換を初期化します。このメソッドは、ロールアップ変換のライフ・サイクルの開始時に1回のみ呼び出されます。すべての内部割当ておよび初期化コードはここに配置する必要があります。

- `void initGroup(DataRecord inputRecord, DataRecord groupAccumulator)`

  このメソッドは、グループ内の最初のデータ・レコードに対して呼び出されます。グループ・アキュムレータのすべての初期化はここに配置する必要があります。

- `void initGroupOnError(Exception exception, DataRecord inputRecord, DataRecord groupAccumulator)`

  このメソッドは、グループ内の最初のデータ・レコードに対して呼び出されます。グループ・アキュムレータのすべての初期化はここに配置する必要があります。initGroup(DataRecord, DataRecord)が例外をスローした場合にのみ呼び出されます。

- `boolean updateGroup(DataRecord inputRecord, DataRecord groupAccumulator)`

  このメソッドは、グループ・アキュムレータを更新するために、グループ内のデータ・レコードごと(最初のレコードおよび最後のレコードを含む)に呼び出されます。

- `boolean updateGroupOnError(Exception exception, DataRecord inputRecord, DataRecord groupAccumulator)`

  このメソッドは、グループ・アキュムレータを更新するために、グループ内のデータ・レコードごと(最初のレコードおよび最後のレコードを含む)に呼び出されます。updateGroup(DataRecord, DataRecord)が例外をスローした場合にのみ呼び出されます。

- `boolean finishGroup(DataRecord inputRecord, DataRecord groupAccumulator)`

  このメソッドは、グループの処理を終了するために、グループ内の最後のデータ・レコードに対して呼び出されます。

- `boolean finishGroupOnError(Exception exception, DataRecord inputRecord, DataRecord groupAccumulator)`

  このメソッドは、グループの処理を終了するために、グループ内の最後のデータ・レコードに対して呼び出されます。`finishGroup(DataRecord, DataRecord)`が例外をスローした場合にのみ呼び出されます。

- `int updateTransform(int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  このメソッドは、入力データ・レコードおよびグループ・アキュムレータのコンテンツ(リクエストされた場合)に基づいて出力データ・レコードを生成するために使用されます。このメソッドの終了時に出力データ・レコードが出力に送信されます。このメソッドは、`boolean updateGroup(DataRecord, DataRecord)`メソッドが`true`を返す場合に必ず呼び出されます。`counter`引数は、現在のグループ更新でこのメソッドがこれまでに呼び出された回数です。戻り値およびその意味の詳細は、変換の戻り値(p.283)を参照してください。

- `int updateTransformOnError(Exception exception, int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  このメソッドは、入力データ・レコードおよびグループ・アキュムレータのコンテンツ(リクエストされた場合)に基づいて出力データ・レコードを生成するために使用されます。`updateTransform(int, DataRecord, DataRecord)`が例外をスローした場合にのみ呼び出されます。

- `int transform(int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  このメソッドは、入力データ・レコードおよびグループ・アキュムレータのコンテンツ(リクエストされた場合)に基づいて出力データ・レコードを生成するために使用されます。このメソッドの終了時に出力データ・レコードが出力に送信されます。このメソッドは、`boolean finishGroup(DataRecord, DataRecord)`メソッドが`true`を返す場合に必ず呼び出されます。`counter`引数は、現在のグループでこのメソッドがこれまでに呼び出された回数です。戻り値およびその意味の詳細は、変換の戻り値(p.283)を参照してください。

- `int transformOnError(Exception exception, int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  このメソッドは、入力データ・レコードおよびグループ・アキュムレータのコンテンツ(リクエストされた場合)に基づいて出力データ・レコードを生成するために使用されます。`transform(int, DataRecord, DataRecord)`が例外をスローした場合にのみ呼び出されます。

SimpleCopy

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

SimpleCopyは、すべての接続済出力ポートにデータをコピーします。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
SimpleCopy	-	いいえ	1	1-n	-	-

概要

SimpleCopyは、単一の入力ポートを介してデータ・レコードを受信し、すべての接続済出力ポートに各データをコピーします。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	コピーされたデータ・レコード用	入力0[1)]
	1-n	いいえ	コピーされたデータ・レコード用	出力0[1)]

説明:

1): 入力のメタデータがデリミタ付きであっても、出力ポートのメタデータには固定長または混合を指定できます。逆の場合も同様です。メタデータはこのコンポーネントを介して伝播できます。すべての出力メタデータが同じである必要があります。

SimpleGather

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

SimpleGatherは、複数の入力からデータ・レコードを収集します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
SimpleGather	はい	いいえ	1-n	1	-	-

概要

SimpleGatherは、1つ以上の入力ポートを介してデータ・レコードを受信します。**SimpleGather**はできるかぎりの速度ですべてのレコードを収集(多重分離)し、そのすべてを単一の出力ポートに送信します。(すべての入力ポートおよび出力ポートのメタデータが同じである必要があります。)

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
	1-n	いいえ	入力データ・レコード用	入力0[1)]
出力	0	はい	収集されたデータ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。すべての出力メタデータが同じである必要があります。

SortWithinGroups

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

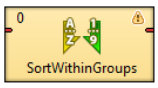
SortWithinGroupsは、ソート・キーに基づいてレコードのグループ内の入力レコードをソートします。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
SortWithinGroups	-	はい	1	1-n	-	-

概要

SortWithinGroupsは、単一の入力ポートを介してデータ・レコード(グループ・キーに基づいてグループ化済)を受信し、隣接レコードの各グループ内の個別のソート・キーに基づいてレコードをソートして、すべての接続済出力ポートに各レコードをコピーします。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	ソート済データ・レコード用	入力0[1)]
	1-n	いいえ	ソート済データ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。すべての出力メタデータが同じである必要があります。

SortWithinGroupsの属性

属性	必須	説明	可能な値
Basic			
Group key	はい	レコードのグループを定義するキー。同じキー値を持っていても、隣接していないレコードは異なるグループに属するとみなされ、異なるグループはそれぞれ他と独立して個別に処理されます。詳細は、グループ・キー(p.276)を参照してください。	
Sort key	はい	隣接するレコードの各グループ内でレコードをソートする基準となるキー。詳細は、ソート・キー(p.277)を参照してください。	
Advanced			
Buffer capacity		メモリー内の解析済レコードの最大数。この数より多くの入力レコードがある場合、外部ソートが実行されます。	10485760 (デフォルト) | 1-N
Number of tapes		外部ソートを実行するために使用される一時ファイルの数。2より大きい偶数です。	8 (デフォルト) | 2*(1-N)

詳細説明

null値のソート

SortWithinGroupsでは、「**Sort key**」属性の同じフィールドに`null`値があるレコードを、その`null`が等しいものとして処理します。

XSLTransformer

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第45章「トランスフォーマの共通プロパティ」(p.320)

目的に適した**トランスフォーマ**を見つけるには、トランスフォーマの比較(p.320)を参照してください。

要約

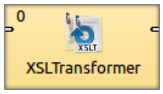
XSLTransformerは、XSL変換を使用して入力データ・レコードを変換します(XSLT 1.0およびXSLT 2.0がサポートされています)。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
XSLTransformer	-	いいえ	1	1	-	-

概要

XSLTransformerコンポーネントは、入力のXSL変換を行い、変換結果を出力に書き込みます。入力および出力は、ファイルURL、ディクショナリまたはフィールドで指定できます。XSL変換は外部ファイルからロードしたり、コンポーネントに定義できます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	入力データ・レコード用	任意1
出力	0	いいえ	変換されたデータ・レコード用	任意2

XSLTransformerの属性

属性	必須	説明	可能な値
Basic			
XSLT file	1)	XSL変換を定義する外部ファイル。	

属性	必須	説明	可能な値
XSLT	1)	グラフで定義されているXSL変換。	
Mapping	2)	互いにセミコロンで区切られた、出力フィールドの個々のマッピングのシーケンス。各マッピングの書式は、`$outputField:=transform($inputField)` (XSL変換に基づいて`inputField`を変換する必要がある場合)、または`$outputField:=$inputField` (`inputField`を変換する必要がある場合)となります。	
XML input file or field	2),3)	入力として使用するファイル、ディクショナリまたはフィールドのURL。	
XML output file or field	2),3)	出力として使用するファイル、ディクショナリまたはフィールドのURL。	
Advanced			
Output charset		出力の文字エンコーディング。	UTF-8 (デフォルト) | その他のエンコーディング

説明:

1): これらの属性のいずれかを設定する必要があります。両方が設定された場合は、**XSLTファイル**が優先されます。

2): これらの属性のいずれかを設定する必要があります。複数設定された場合は、**Mapping**が優先されます。

3): どちらも設定するか、どちらも設定しないかのいずれかとする必要があります。**Mapping**が定義されている場合、これらは無視されます。

詳細説明

マッピング

マッピングは、次のウィザードを使用して定義できます。

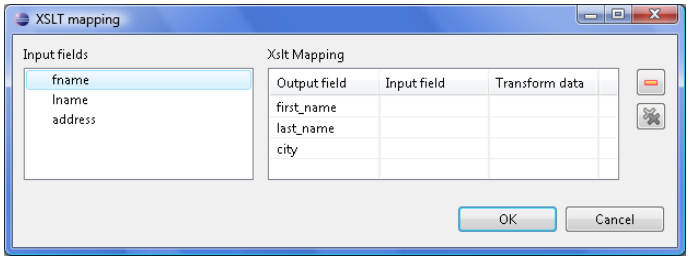


図55.11. XSLT mapping

右側のペインの「**Input field**」列に入力フィールドをドラッグ・アンド・ドロップして、左側の「**Input fields**」ペインから出力フィールドに入力フィールドを割り当てます。「**Transform data**」オプションをtrueに設定して、変換が必要なフィールドを選択します。デフォルトでは、変換されるフィールドはありません。

生成される**マッピング**は次のようになります。

```
$0.first_name:=transform($0.fname);$0.last_name:=$0.lname;$0.city:=$0.address;
```

図55.12. マッピングの例

「**Mapping**」属性、または「**XML input file or field**」および「**XML output file or field**」の2つの属性ペアのうち、いずれかを設定する必要があります。これらにより、入力ファイルと出力ファイル、ディクショナリまたはフィールドが定義されます。「**Mapping**」を設定した場合、他の2つの属性は設定されていても無視されます。

第56章 ジョイナ

コンポーネントとは何かを理解していることを想定しています。概要は、第19章「コンポーネント」(p.97)を参照してください。

一部のコンポーネントはグラフの中間ノードとなります。これらは**ジョイナ**または**トランスフォーマ**と呼ばれます。

トランスフォーマの詳細は、第55章「トランスフォーマ」(p.576)を参照してください。ここでは**ジョイナ**について説明します。

ジョイナは、キー値に基づいて複数のデータ・ソースからデータを結合するために使用します。

コンポーネントには様々なプロパティを設定できます。ただし、一部のものが共通する場合があります。すべてのコンポーネントに共通のプロパティもあれば、ほとんどのコンポーネントに共通のプロパティもあり、**ジョイナ**のみに共通のプロパティもあります。次のことを学習している必要があります。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第46章「ジョイナの共通プロパティ」(p.323)

ジョイナは、データの処理方法に基づいて識別できます。ほとんどの**ジョイナ**はキー値を使用して動作します。

- **ジョイナ**には、2つ以上の入力ポートからデータを読み取り、キー値の同等性に基づいてデータを結合するものがいくつかあります。
  - ExtHashJoin (p.667)は、キー値の同等性に基づいて2つ以上のデータ入力を結合します。
  - ExtMergeJoin(p.672)は、キー値の同等性に基づいて2つ以上のソート済データ入力を結合します。
- **ジョイナ**には、1つの入力ポートおよび別のデータ・ソースからデータを読み取り、キー値の同等性に基づいてデータを結合するものがいくつかあります。
  - DBJoin(p.664)は、キー値の同等性に基づいて1つの入力データ・ソースとデータベースを結合します。
  - LookupJoin(p.677)は、キー値の同等性に基づいて1つの入力データ・ソースと参照表を結合します。
- **ジョイナ**には、キー値の一致度レベルに基づいてデータを結合するものがあります。
  - ApproximativeJoin(p.654)は、キー値の一致度レベルに基づいて2つのソート済入力を結合します。
- **ジョイナ**には、ユーザーが定義したキー値の関係に基づいてデータを結合するものがあります。
  - RelationalJoin(p.680)は、ユーザーが定義したキー値の関係(!=、>、>=、<、<=)に基づいて2つ以上のソート済データ入力を結合します。
- Combine(p.662)は、タプルによってデータ・フローを結合します。

ApproximativeJoin

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第46章「ジョイナの共通プロパティ」(p.323)

目的に適した**ジョイナ**を見つけるには、ジョイナの比較(p.323)を参照してください。

要約

ApproximativeJoinは、2つのデータ・ソースからのソート済データを共通の一致キーでマージします。その後、ユーザーが指定した**一致度制限**に基づいてレコードを出力に分配します。

コンポーネント	同じ入力メタデータ	ソート済入力	スレーブ入力	出力	スレーブのないドライバの出力	ドライバのないスレーブの出力	同等性に基づく結合
ApproximativeJoin	いいえ	はい	1	2-4	はい	はい	はい

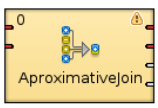
概要

ApproximativeJoinは、通常は非常に特殊な状況で使用される、柔軟性のあるジョイナです。入力をソートする必要があり、データをメモリーで処理するために非常に高速です。ただし、メモリー要件が入力のサイズに比例するため、大量の入力の場合は使用しないでください。

最初の入力ポートに接続されるデータは**マスター**と呼ばれます(他のジョイナでも同様)。2番目の入力ポートは**スレーブ**と呼ばれます。

他のジョイナとは異なり、このコンポーネントは結合に2つのキーを使用します。まず、**一致キー**を使用して、標準の方法でレコードが照合されます。その後、これらの一致レコードの各ペアが再度確認され、**結合キー**およびユーザー定義のアルゴリズムを使用して2つのレコードの一致度(類似度)が計算されます。次に、一致度レベルが**一致度制限**と比較され、各レコードが最初(一致度が高い)または2番目の(一致度が低い)出力ポートに送信されます。残りのレコードは、3番目と4番目の出力ポートに送信されます。

アイコン



ポート

ApproximativeJoinは、2つの入力ポートを介してデータを受信します。入力ポートのそれぞれは、異なるメタデータ構造を持っています。

一致度は、一致したデータ・レコードに対して計算されます。一致度の高いレコードは最初の出力ポートに送信されます。一致度の低いレコードは2番目の出力ポートに送信されます。オプションで、3番目の出力ポートを使用して一致のないマスター・レコードを取得できます。オプションで、4番目の出力ポートを使用して一致のないスレーブ・レコードを取得できます。

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	マスター入力ポート	任意
	1	はい	スレーブ入力ポート	任意
出力	0	はい	一致度の高い結合されたデータ用の出力ポート	任意。オプションで`_total_conformity_`および`_keyName_conformity_`の追加フィールドを含みます。追加フィールド(p.660)を参照してください。
	1	はい	一致度の低い結合されたデータ用の出力ポート	任意。オプションで`_total_conformity_`および`_keyName_conformity_`の追加フィールドを含みます。追加フィールド(p.660)を参照してください。
	2	いいえ	スレーブ一致のないマスター・データ・レコード用のオプションの出力ポート	入力0
	3	いいえ	マスター一致のないスレーブ・データ・レコード用のオプションの出力ポート	入力1

ApproximativeJoinの属性

属性	必須	説明	可能な値
Basic			
Join key	はい	**一致キー**と同じ値を持つ受信データ・フローを比較し、最初と2番目の出力ポートに分配する基準とするキー。指定された**一致度制限**によって異なります。Join key(p.658)を参照してください。	
Matching key	はい	このキーは、マスターおよびスレーブのレコードを照合するときに使用します。	
Transform	1)	一致度の高いレコードについて、グラフに定義されているCTLまたはJavaでの変換。	
Transform URL	1)	一致度の高いレコードについて、CTLまたはJavaでの変換を定義する外部ファイル。	
Transform class	1)	一致度の高いレコードの外部変換クラス。	
Transform for suspicious	2)	一致度の低いレコードについて、グラフに定義されているCTLまたはJavaでの変換。	
Transform URL for suspicious	2)	一致度の低いレコードについて、CTLまたはJavaでの変換を定義する外部ファイル。	

属性	必須	説明	可能な値
Transform class for suspicious	2)	一致度の低いレコードの外部変換クラス。	
Conformity limit (0,1)		この属性は、レコードのペアに対する一致度の制限を定義します。この値より高い一致度を持つレコードには変換が適用され、この値より低い一致度を持つレコードには疑わしい場合の変換が適用されます。	0.75 (デフォルト) | 0から1
Advanced			
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Deprecated			
Locale		国際化が使用されている場合に使用されるロケール。	
Case sensitive		`true`に設定すると、大文字と小文字が区別されます。デフォルトでは、これらは同一であるかのように処理されます。	false (デフォルト) | true
Error actions		指定した変換が**エラー・コード**を返した場合に実行する必要があるアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、**コンソール**に書き出されます。	
Slave override key		**CloverETL**の古いバージョンでの**結合キー**のスレーブ部分。**結合キー**は、個々の式のシーケンスとして定義されていました。この式の構成は、マスター・フィールド名と、それぞれの名前の後に6つのパラメータ(後述)を囲むカッコが続くというものでした。これらの個々の式はセミコロンで区切られていました。**スレーブ・オーバーライド・キー**は、マスターの**結合キー**・フィールドに対応するスレーブ側のシーケンスです。したがって、前述の場合では、「**Slave override key**」が`fname;lname`となり、「**Join key**」が`first_name(3 0.8 true false false false);last_name(4 0.2 true false false false)`となります。	
Slave override matching key		**CloverETL**の古いバージョンでの**一致キー**のスレーブ部分。**一致キー**はマスター・フィールド名として定義されていました。スレーブ・オーバーライド一致キーはこれに対応するスレーブ側のキーです。したがって、前述の場合(`$masterField=$slaveField`)、「**Slave override matching key**」はこの`slaveField`のみとなります。また、「**Matching key**」はこの`masterField`です。	

説明:

1) これらのいずれかを設定する必要があります。これらの変換属性は指定する必要があります。これらの変換属性はいずれも**ジョイナ**の共通CTLテンプレートを使用するか、`RecordTransform`インタフェースを実装する必要があります。

2) これらのいずれかを設定する必要があります。これらの変換属性は指定する必要があります。これらの変換属性はいずれも**ジョイナ**の共通CTLテンプレートを使用するか、`RecordTransform`インタフェースを実装する必要があります。

詳細は、CTLスクリプトの詳細(p.661)またはJavaインタフェース(p.661)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

詳細説明

- **Matching key**

  「**Matching key**」ウィザードを使用して**一致キー**を定義できます。左側の「**Master key**」ペインで必要なマスター(ドライバ)・フィールドを選択し、それを「**Master key**」タブの右側の「**Master key**」ペインにドラッグ・アンド・ドロップするのみです。(用意されているボタンを使用することもできます。)

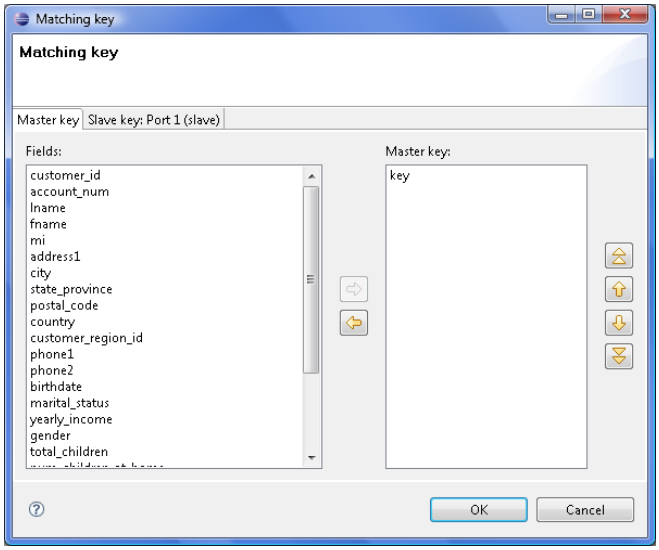


図56.1.「Matching Key」ウィザード(「Master Key」タブ)

「**Slave key**」タブでは、左側の「**Fields**」ペインにあるスレーブ・フィールドのいずれかを選択し、それを「**Key mapping**」ペインの「**Master key field**」列(「**Master key**」タブのマスター・フィールドを含む)から右側の「**Slave key field**」列にドラッグ・アンド・ドロップする必要があります。

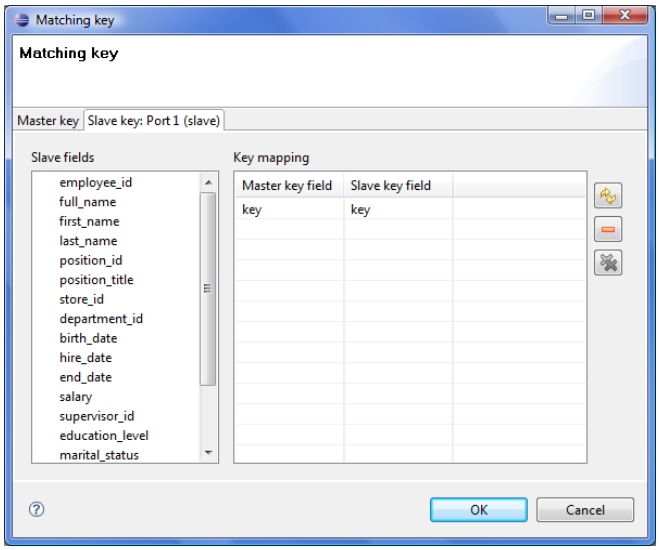


図56.2.「Matching Key」ウィザード(「Slave Key」タブ)

例56.1.一致キー

一致キーは次のようになります。

```
$master_field=$slave_field
```

- **Conformity limit**

  一致度の制限を定義する必要があります(**Conformity limit (0,1)**)。定義された値により、その一致度に基づいて受信レコードが分配されます。一致度は、指定した制限より高くなることや低くなることがあります。どちらのグループに対しても変換を定義する必要があります。一致度の低いレコードは疑わしいとマークされてポート1に送信されますが、一致度の高いレコードはポート0に送信されます(良好な一致)。

  一致度の計算は難しい問題であるため、基本的な用語のみを説明します。まず、**一致キー**に基づいてレコードのグループが作成されます。その後、**結合キー**の指定に基づいて、1つのグループ内のすべてのレコードが互いに比較されます。特定の**結合キー・**フィールドで選択された比較の強度により、使用するペナルティ文字が決定されます(比較の強度は、結合キー(p.658)を参照してください)。

  - **Identical**: 文字単位の比較です。異なる文字単位でペナルティが付与されます(`String.equals()`と同様)。
  - **Tertiary**: この比較強度のみがアクティブ化されている場合は、大文字小文字の差異を無視します(`String.equalsIgnoreCase()`と同様)。**Identical**と同時にアクティブ化された場合、発音区別文字の差異(cとčなど)はペナルティが最大ですが、大文字小文字に違い(aとAなど)はペナルティが半分になります。
  - **Secondary**: 通常の文字および同じ言語で発音区別文字の付いた派生文字は一致とみなされます。比較で使用する言語はフィールドのメタデータから取得されます。フィールドにメタデータが設定されていない場合は`en`として処理され、**Primary**と同じように機能します(派生文字は等しいとして処理されます)。

    例:

    language=`sk`: a、á、äはすべてスロバキア語の文字であるため、これらは等しくなります。

    language= `sk`: ąはポーランド語の文字であるため(スロバキア語ではない)、a、ąは異なります。
  - **Primary**: 言語設定に関係なく、発音区別文字の付いたすべての派生文字が一致とみなされます。

    例:

    language=任意: a、á、ä、ąはすべてaの派生文字であるため、これらは等しくなります。

  最後の手順として、フィールドの一致度を重み付けした平均として、総合一致度が計算されます。

- **Join key**

  結合キーは「**Join key**」ウィザードを利用して定義できます。「**Join key**」ウィザードを開くと、「**Master key**」タブおよび「**Slave key**」タブの2つのタブが表示されます。

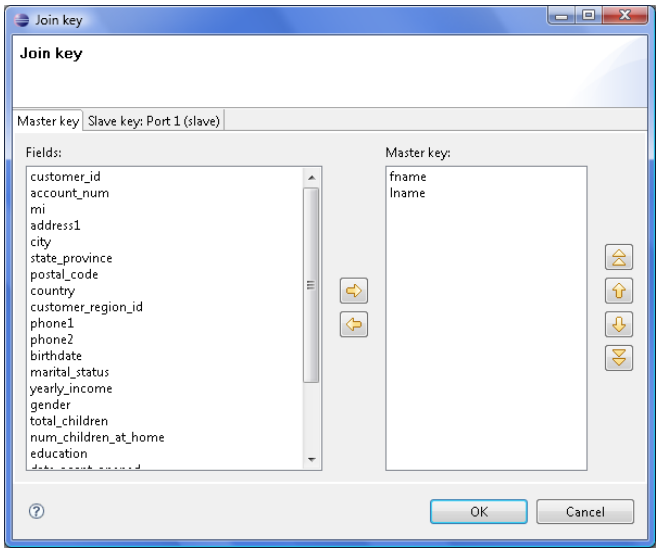


図56.3.「Join Key」ウィザード(「Master Key」タブ)

「**Master key**」タブでは、左側の「**Fields**」ペインにあるドライバ(マスター)・フィールドを選択し、それらを右側の「**Master key**」ペインにドラッグ・アンド・ドロップする必要があります。(ボタンを使用することもできます。)

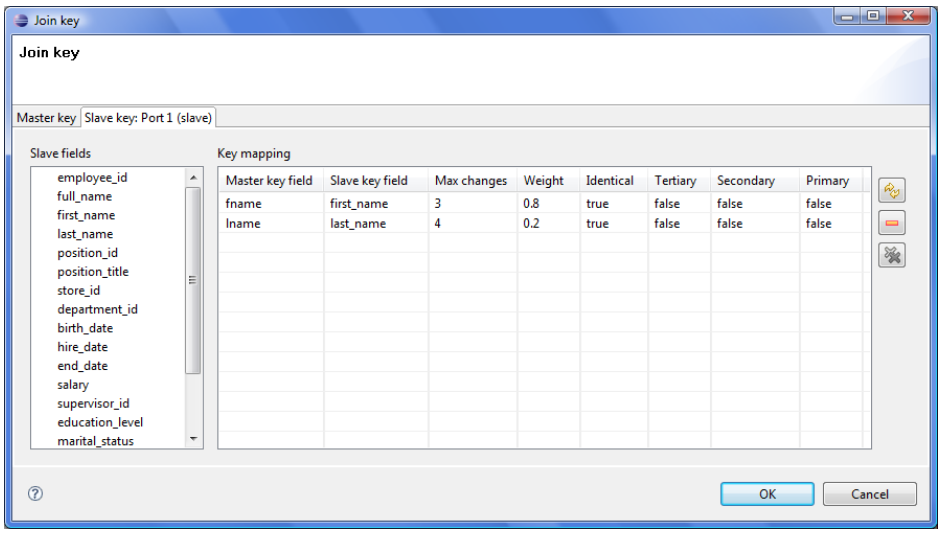


図56.4.「Join Key」ウィザード(「Slave Key」タブ)

「**Slave key**」タブには、左側に「**Fields**」ペイン(すべてのスレーブ・フィールドを含む)および右側に「**Key mapping**」ペインが表示されます。

これらのスレーブ・フィールドのいくつかを選択し、それらを「**Master key field**」列(最初の手順の「**Master key**」タブで選択したマスター・フィールドを含む)から右側の「**Slave key field**」列にドラッグ・アンド・ドロップする必要があります。これらの2つの列の他にも、「Maximum changes」、「Weight」および比較の強度を表す残り4つの列の6つの列があります。

「Maximum changes」プロパティには、1つのデータ値を別の値に変換するために変更する必要のある文字数に等しい整数が格納されます。「Maximum changes」プロパティは一致度を計算するために使用します。ある文字列を別の文字列に変換するために変更する文字が多い場合、2つの文字列間の一致度は0となります。

「Weight」プロパティは、類似性を計算する場合のフィールドの重み付けを定義します。各フィールドの重み付けの差異は、ユーザーが定義した重み付けとユーザーが定義した重み付けの合計の比率として計算されます。

比較の強度にはidentical、tertiary、secondaryまたはprimaryを設定できます。

- **identical**

  同一の文字のみが一致とみなされます。

- **tertiary**

  大文字および小文字が一致とみなされます。

- **secondary**

  発音区別文字の付いた文字およびそのラテン語相当文字が一致とみなされます。

- **primary**

  ペダンクル、ピンク、リングなどの特性が付加された文字と、そのラテン語相当文字が一致とみなされます。

ウィザードでは、単純にクリックするとこれらの列のブール値を変更できます。これにより、trueがfalseに切り替わり、その逆の操作もできます。また、クリックして必要な値を入力すると、任意の数値を変更できます。

「**OK**」をクリックすると、先頭にドル記号が付けられ、セミコロンで区切られたドライバ(マスター)・フィールドとスレーブ・フィールドの割当てのシーケンスが表示されます。各スレーブ・フィールドの後には、空白で区切られた前述の6つのパラメータを含むカッコが続きます。そのシーケンスは次のようになります。

```
$driver_field1=$slave_field1(parameters);...;$driver_fieldN=$slave_fieldN(parameters)
```

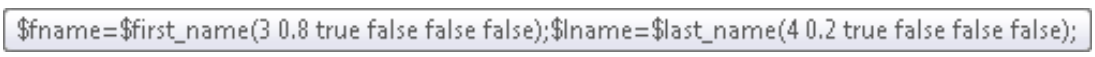


図56.5. ApproximativeJoinコンポーネントの「Join Key」属性の例

例56.2. ApproximativeJoinの結合キー

`$first_name=$fname(3 0.8 true false false false);$last_name=$lname(4 0.2 true false false false)`。この**結合キー**では、first_nameおよびlast_nameは最初(マスター)のデータ・フローからのフィールドであり、fnameおよびlnameは2番目(スレーブ)のデータ・フローからのフィールドです。

- **追加フィールド**

最初および2番目の出力ポートのメタデータには、数値データ型の追加フィールドを含めることができます。それらの名前は、"_total_conformity_"と、いくつかの"_keyName_conformity_"フィールドにする必要があります。最後のフィールド名では、これらの追加フィールド名のkeyNameとして「**Join key**」属性のフィールド名を使用する必要があります。これらの追加フィールドには、計算された一致度の値(合計またはkeyNameの値)が書き込まれます。

CTLスクリプトの詳細

結合属性を定義する場合は、入力データ・ソースのフィールドを出力にマップする変換を指定する必要があります。これを行うには、**変換エディタ**の「**Transformations**」タブを使用します。ただし、この最も簡単なアプローチを使用すると、より詳細な変換を指定できないことがあります。この場合は、CTLスクリプトを使用する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム・フィールド・マッピングを指定できます。

すべての**ジョイナ**は、ジョイナ用CTLテンプレート(p.325)で説明する同じ変換テンプレートを共有します。

Javaインタフェース

Javaで変換を定義する場合は、すべての**ジョイナ**に共通の次のインタフェースを変換で実装する必要があります。

ジョイナ用Javaインタフェース(p.328)

Combine

Jobflowコンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

コンポーネントは「**Palette**」→「**Joiners**」にあります。

要約

Combineは、各入力ポートから1つのレコードを取得し、指定された変換に基づいてそれらを結合して、生成されるレコードを1つ以上のポートに送信します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	変換	変換が必要	Java	CTL
Combine	いいえ	いいえ	1-n	1-n	はい	はい	はい	はい

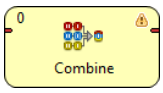
概要

各手順において、**Combine**コンポーネントはすべての入力ポートから1つのレコードを取得して単一の出力レコードを作成し、指定された変換に基づいて、この出力レコードのフィールドに入力レコードのデータ(またはその他のデータ)を入力します。

結合変換を定義する最も簡単な方法は、**Transform**コンポーネント属性で使用可能な変換エディタ(p.286)を使用することです。ここでは、各入力ポートのメタデータが左側に、1つの出力ポートのメタデータが右側に表示されます。単に左側から右側のフィールドに、フィールドをドラッグ・アンド・ドロップすると、必要な結合変換が作成されます。

デフォルト設定では、このコンポーネントは各入力ポートで同じ数のレコードを受信することを想定しているため、一部の入力エッジにレコードが含まれている場合にある入力エッジが空になると、コンポーネントが失敗します。このエラーは、「**Allow incomplete tuples**」属性をtrueに設定すると回避できます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	1-n	はい	結合される入力レコード	任意
出力	0	はい	結合結果の出力レコード	任意

Combineの属性

属性	必須	説明	可能な値
Basic			
Transform	1)	入力レコードを出力レコードに結合する方法の定義。グラフのソースにCTLまたはJavaで書き込まれます。	
Transform URL	1)	レコードを結合する方法の定義が格納されている外部ファイルの名前(パスを含む)。CTLまたはJavaで記述されます。	
Transform class	1)	レコードを結合する方法を定義する外部クラスの名前。	
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Allow incomplete tuples		各入力ポートで、出力レコードごとにレコードを提供する必要があるかどうか。	true (デフォルト) | false

説明:

1): これらのいずれかを指定する必要があります。これらの変換属性はいずれも**Reformat**用のCTLテンプレートを使用するか、`RecordTransform`インタフェースを実装します。

変換の詳細は、変換の定義(p.279)も参照してください。

DBJoin

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第46章「ジョイナの共通プロパティ」(p.323)

目的に適した**ジョイナ**を見つけるには、ジョイナの比較(p.323)を参照してください。

要約

DBJoinは1つの入力ポートを介してデータを受信し、それをデータベース表のデータと結合します。これらの2つのデータ・ソースは、異なるメタデータ構造を持っている可能性があります。

コンポーネント	同じ入力メタデータ	ソート済入力	スレーブ入力	出力	スレーブのないドライバの出力	ドライバのないスレーブの出力	同等性に基づく結合
DBJoin	いいえ	いいえ	1 (仮想)	1-2	はい	いいえ	はい

概要

DBJoinは1つの入力ポートを介してデータを受信し、それをデータベース表のデータと結合します。これらの2つのデータ・ソースは、異なるメタデータ構造を持っている可能性があります。これは、ほとんどの一般的な状況で使用される汎用のジョイナです。入力をソートする必要はなく、データがメモリーで処理されるために非常に高速です。

最初の入力ポートに接続されるデータは**マスター**と呼ばれ、2番目のデータ・ソースは**スレーブ**と呼ばれます。そのデータは、2番目(仮想)の入力ポートを介して受信するものとみなされます。各マスター・レコードは、結合キーと呼ばれる1つ以上のフィールドで、スレーブ・レコードと照合されます。出力は、結合された入力を出力にマップする変換を適用することにより生成されます。

アイコン

ポート

DBJoinは1つの入力ポートを介してデータを受信し、それをデータベース表のデータと結合します。これらの2つのデータ・ソースは、異なるメタデータ構造を持っている可能性があります。

結合されたデータは最初の出力ポートに送信されます。オプションで、2番目の出力ポートを使用して一致のないマスター・レコードを取得できます。

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	マスター入力ポート。	任意
	1 (仮想)	はい	スレーブ入力ポート。	任意
出力	0	はい	結合されたデータ用の出力ポート。	任意
	1	いいえ	スレーブ一致のないマスター・データ・レコード用のオプションの出力ポート。(「**Join type**」属性が`Inner join`に設定されている場合のみ。)これは**LookupJoin**および**DBJoin**にのみ適用されます。	入力0

DBJoinの属性

属性	必須	説明	可能な値
Basic			
Join key	はい	それに基づいて受信データ・フローを結合するキー。Join key(p.666)を参照してください。	
Left outer join		`true`に設定すると、対応するスレーブがないドライバ・レコードも解析されます。そうでない場合は内部結合が実行されます。	false (デフォルト) | true
DB connection	はい	スレーブ・レコードのリソースとして使用されるDB接続のID。	
DB metadata		使用されるDBメタデータのID。設定されていない場合は、**SQL問合せ**を使用してデータベースからメタデータが抽出されます。	
Query URL	3)	パスを含む、SQL問合せを定義する外部ファイルの名前。	
SQL query	3)	グラフで定義されているSQL問合せ。	
Transform	1), 2)	グラフに定義されているCTLまたはJavaでの変換。	
Transform URL	1), 2)	CTLまたはJavaでの変換を定義する外部ファイル。	
Transform class	1), 2)	外部変換クラス。	
Cache size		メモリーに格納できる、異なるキー値を持つレコードの最大数。	100 (デフォルト)
Advanced			
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Deprecated			
Error actions		指定した変換が**エラー・コード**を返した場合に実行する必要があるアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、**コンソール**に書き出されます。	

説明:

1) これらの変換属性のいずれかを設定する必要があります。これらはいずれも**ジョイナ**の共通CTLテンプレートを使用するか、`RecordTransform`インタフェースを実装する必要があります。

詳細は、CTLスクリプトの詳細(p.666)またはJavaインタフェース(p.666)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

2) 唯一の例外は、これらの3つの属性のいずれも指定されていない場合ですが、「**SQL query**」属性にはDB表から読み取られるレコードが定義されます。入力レコードに含まれる**結合キー**の値は、DB表からレコードを選択するために使用します。これらはアンロードされ、変換されることなく出力ポートにそのまま送信されます。

3) これらの属性のいずれかを指定する必要があります。両方が定義されている場合は、「**Query URL**」が優先されます。

詳細説明

- **Join key**

  結合キーは、セミコロン、コロンまたはパイプによって互いに区切られた、マスター・データ・ソースのフィールド名のシーケンスです。「**Edit key**」ウィザードでキーを定義できます。

  これらのフィールド名の順序は、データベース表のキー・フィールドの順序(およびそのデータ型)に対応する必要があります。**結合キー**のスレーブ部分は「**SQL query**」属性で定義する必要があります。

  問合せ属性のいずれかには、`... where field_K=? and field_L=?`という書式の式が含まれている必要があります。

例56.3. DBJoinの結合キー

```
$first_name;$last_name
```

これは、マスター・レコードをスレーブ・レコードと結合する役割を果たすフィールドの主要部分です。

SQL問合せには、次のような式が含まれている必要があります。

```
... where fname=? and lname=?
```

対応するフィールドが比較され、一致する値によりマスターおよびスレーブ・レコードが結合されます。

CTLスクリプトの詳細

結合属性を定義する場合は、入力データ・ソースのフィールドを出力にマップする変換を指定する必要があります。これを行うには、**変換エディタ**の「**Transformations**」タブを使用します。ただし、この最も簡単なアプローチを使用すると、より詳細な変換を指定できないことがあります。この場合は、CTLスクリプトを使用する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム・フィールド・マッピングを指定できます。

すべての**ジョイナ**は、ジョイナ用CTLテンプレート(p.325)で説明する同じ変換テンプレートを共有します。

Javaインタフェース

Javaで変換を定義する場合は、すべての**ジョイナ**に共通の次のインタフェースを変換で実装する必要があります。

ジョイナ用Javaインタフェース(p.328)

ExtHashJoin

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第46章「ジョイナの共通プロパティ」(p.323)

目的に適した**ジョイナ**を見つけるには、ジョイナの比較(p.323)を参照してください。

要約

2つ以上のデータ・ソースのソートされていない可能性のあるデータを共通キーに基づいてマージする汎用のジョイナです。

コンポーネント	同じ入力メタデータ	ソート済入力	スレーブ入力	出力	スレーブのないドライバの出力	ドライバのないスレーブの出力	同等性に基づく結合
ExtHashJoin	いいえ	いいえ	1-n	1	いいえ	いいえ	はい

概要

これは、ほとんどの一般的な状況で使用される汎用のジョイナです。入力をソートする必要がなく、メモリー内で処理されるため非常に高速です。

最初の入力ポートに接続されるデータは**マスター**と呼ばれます(他の**ジョイナ**でも同様)。接続されているその他すべての入力ポートは**スレーブ**と呼ばれます。各マスター・レコードは、**結合キー**と呼ばれる1つ以上のフィールドで、すべてのスレーブ・レコードと照合されます。出力は、結合された入力を出力にマップする変換を適用することにより生成されます。詳細は、結合方法(p.671)を参照してください。

スレーブ・ポートに大量の入力がある場合は、このジョイナを使用しないでください。これは、スレーブ・データがメモリーにキャッシュされるためです。

ヒント

データの量が多い場合は、**ExtMergeJoin**コンポーネントの使用を考慮してください。データ・ソースがソートされていない場合は、最初にソート・コンポーネント(**ExtSort**、**FastSort**または**SortWithinGroups**)を使用してください。

アイコン

ポート

ExtHashJoinは、それぞれ異なるメタデータ構造を持つ可能性のある2つ以上の入力ポートを介してデータを受信します。

結合されたデータは1つの出力ポートに送信されます。

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	マスター入力ポート	任意
	1	はい	スレーブ入力ポート	任意
	2-n	いいえ	オプションのスレーブ入力ポート	任意
出力	0	はい	結合されたデータ用の出力ポート	任意

ExtHashJoinの属性

属性	必須	説明	可能な値
Basic			
Join key	はい	それに基づいて受信データ・フローを結合するキー。Join key (p.669)を参照してください。	
Join type		結合のタイプ。結合タイプ(p.324)を参照してください。	Inner (デフォルト) | Left outer | Full outer
Transform	1)	グラフに定義されているCTLまたはJavaでの変換。	
Transform URL	1)	CTLまたはJavaでの変換を定義する外部ファイル。	
Transform class	1)	外部変換クラス。	
Allow slave duplicates		`true`に設定すると、重複するキー値を持つレコードが許可されます。`false`の場合、結合には**最初の**レコードのみが使用されます。	false (デフォルト) | true
Advanced			
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Hash table size		データ・フローの結合時に使用するハッシュ表の初期サイズ。結合する必要のあるレコードがより多い場合はハッシュ表を再ハッシュできますが、これにより解析プロセスの速度は遅くなります。詳細は、ハッシュ表(p.671)を参照してください。	512 (デフォルト)
Deprecated			
Error actions		指定した変換が**エラー・コード**を返した場合に実行する必要があるアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、**コンソール**に書き出されます。	

属性	必須	説明	可能な値
Left outer		trueに設定すると、左外部結合が実行されます。デフォルトはfalseです。ただし、この属性よりも「**Join type**」が優先されます。両方設定した場合は、「**Join type**」のみが適用されます。	false (デフォルト) | true
Full outer		trueに設定すると、完全外部結合が実行されます。デフォルトはfalseです。ただし、この属性よりも「**Join type**」が優先されます。両方設定した場合は、「**Join type**」のみが適用されます。	false (デフォルト) | true

説明:

1) これらのいずれかを設定する必要があります。これらの変換属性は指定する必要があります。これらの変換属性はいずれも**ジョイナ**の共通CTLテンプレートを使用するか、RecordTransformインタフェースを実装する必要があります。

詳細は、CTLスクリプトの詳細(p.671)またはJavaインタフェース(p.671)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

詳細説明

- **Join key**

  「**Join key**」属性は、すべてのスレーブのマッピング式をハッシュで区切ったシーケンスです。これらの各マッピング式は、等号を使用して結合されたマスター・レコードとスレーブ・レコード(この順序)のフィールド名をセミコロン、コロンまたはパイプで区切ったシーケンスです。

  ```
  $CUSTOMERID=$CUSTOMERID#$ORDERID=$ORDERID;$PRODUCTID=$PRODUCTID
  ```

  図56.6. ExtHashJoinコンポーネントの「Join Key」属性の例

  これらのマッピングの順序は、スレーブ入力ポートの順序と一致している必要があります。これらのマッピングの一部が空であるか欠落しているスレーブ入力ポートがある場合、最初のスレーブ入力ポートのマッピングがかわりに使用されます。

  ### 注意

  各マスター・フィールドを使用して、異なるスレーブをマスターと結合できます。

  例56.4. ExtHashJoinの結合キーのスレーブ部分

```
$master_field1=$slave_field1;$master_field2=$slave_field2;...;$master_fieldN=$slave_fieldN
```

  - 欠落している$slave_fieldJがある場合(つまり、副次式が$master_fieldJ=のような場合)、これは$master_fieldJと同じになります。
  - 欠落している$master_fieldKがある場合、最初のポートの$master_fieldKが使用されます。

  例56.5. ExtHashJoinの結合キー

```
$first_name=$fname;$last_name=$lname#=$lname;$salary=;$hire_date=$hdate
```

  - 最初のスレーブ・データ・ソース(入力ポート1)の**結合キー**の一部を次に示します。

    ```
    $first_name=$fname;$last_name=$lname
    ```

- この場合、マスター・データ・フローの次の2つのフィールドが最初のスレーブ・データ・ソースとの結合に使用されます。

  `$first_name`および`$last_name`

- これらは、この最初のスレーブ・データ・ソースの次の2つのフィールドと結合されます。

  それぞれ`$fname`および`$lname`

- 2番目のスレーブ・データ・ソース(入力ポート2)の**結合キー**の一部を次に示します。

  `=$lname;$salary=;$hire_date=$hdate`

- この場合、マスター・データ・フローの次の3つのフィールドが2番目のスレーブ・データ・ソースとの結合に使用されます。

  `$last_name` (これが最初のスレーブ・データ・ソースの`$lname`と結合されるフィールドであるため)、`$salary`および`$hire_date`

- これらは、この2番目のスレーブ・データ・ソースの次の3つのフィールドと結合されます。

  それぞれ`$lname`、`$salary`および`$hdate`。(このスレーブ`$salary`フィールドは、同じ名前のマスター・フィールドを使用して表現されます。)

「**Join key**」属性を作成するには、「**Hash Join key**」ウィザードを使用する必要があります。「**Join key**」属性行をクリックすると、この行にボタンが表示されます。このボタンをクリックすると、前述のウィザードを開くことができます。

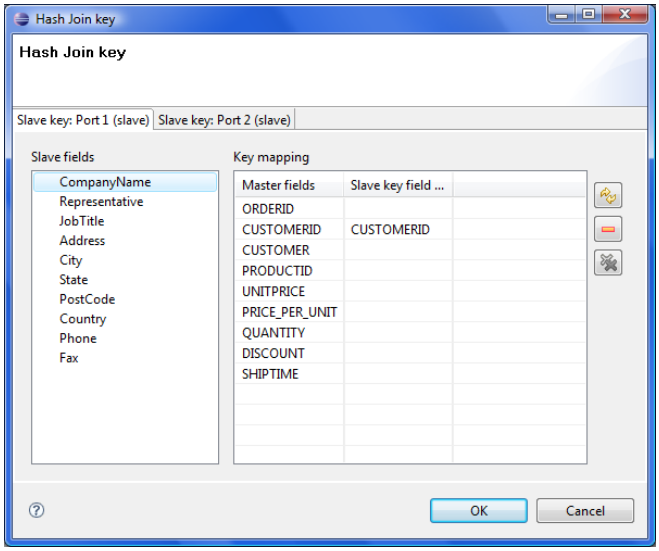


図56.7.「Hash Join Key」ウィザード

ここには、すべてのスレーブ入力ポート用のタブが表示されます。各スレーブ・タブには2つのペインがあります。左側の「**Slave fields**」ペインと右側の「**Key mapping**」ペインです。左側のペインには、すべてのスレーブ・フィールド名のリストが表示されます。右側のペインには、「**Master fields**」および「**Slave key field mapped**」の2つの列が表示されます。左側の列には、ドライバ入力ポートのすべてのフィールド名が含まれます。スレーブ・フィールドをドライバ(マスター)フィールドにマップする場合は、該当する各スレーブ・フィールドを、左側のペインでその項目をクリックすることによって選択し、それを右側のペインの「**Slave key field mapped**」列のマップ先ドライバ(マスター)フィールドの行にドラッグ・アンド・ドロップする必要があります。選択したスレーブ・フィールドについてこれを行う必要があります。また、すべてのスレーブ・タブについて同じプロセスを繰り返す必要があります。各タブで「**Auto mapping**」ボタンなどのボタンを使用することもできます。この場合、スレーブ・フィールドがその名前に基づいてドライバ(マスター)フィールドにマップされます。各スレーブは、異なる数のスレーブ・フィールドを異なる数のドライバ(マスター)・フィールドにマップできます。

- **ハッシュ表**

  このコンポーネントは、最初にスレーブ入力ポートを介してレコードを受信して読み取り、これらのレコードからハッシュ表を作成します。これらのハッシュ表は十分に小さい必要があります。その後、ドライバ入力ポートを介して受信したドライバ・レコードごとに、コンポーネントはこれらのハッシュ表内の対応するレコードを参照します。スレーブ入力ポートごとに1つのハッシュ表が作成されます。入力ポートのレコードをソートする必要はありません。このようなレコードが見つかると、ハッシュ表のドライバ・レコードとスレーブ・レコードのタプルが変換クラスに送信されます。マスターのタプルおよびそれに対応するスレーブ・レコードごとに、変換メソッドが呼び出されます。

  受信レコードをソートする必要はありませんが、ハッシュ表の初期化には時間がかかるため、ハッシュ表に保存できるレコード数を指定すると有用な場合があります。この属性を指定する場合は、必要な値よりも少し大きめに設定することをお薦めします。ただし、レコード・セットが小さい場合、デフォルト値を変更する必要はありません。

結合方法

すべてのスレーブ入力データはメモリーに保存されます。ただし、マスター・データはメモリーに保存されません。そのため、メモリー所要量ではスレーブ・データのサイズのみを考慮してください。したがって、大量のデータは必ずマスターに設定し、スレーブには少量の入力を設定してください。**ExtHashJoin**は、スレーブ・レコードの保存にメモリー内ハッシュ表を使用します。

重要

異なる数の各種マスター・フィールドを使用して、各スレーブ・ポートをマスターと結合できます。

CTLスクリプトの詳細

結合属性を定義する場合は、入力データ・ソースのフィールドを出力にマップする変換を指定する必要があります。これを行うには、**変換エディタ**の「**Transformations**」タブを使用します。ただし、この最も簡単なアプローチを使用すると、より詳細な変換を指定できないことがあります。この場合は、CTLスクリプトを使用する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム・フィールド・マッピングを指定できます。

すべての**ジョイナ**は、ジョイナ用CTLテンプレート(p.325)で説明する同じ変換テンプレートを共有します。

Javaインタフェース

Javaで変換を定義する場合は、すべての**ジョイナ**に共通の次のインタフェースを変換で実装する必要があります。

ジョイナ用Javaインタフェース(p.328)

ExtMergeJoin

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第46章「ジョイナの共通プロパティ」(p.323)

目的に適した**ジョイナ**を見つけるには、ジョイナの比較(p.323)を参照してください。

要約

2つ以上のデータ・ソースのソート済データを共通キーに基づいてマージする汎用のジョイナです。

コンポーネント	同じ入力メタデータ	ソート済入力	スレーブ入力	出力	スレーブのないドライバの出力	ドライバのないスレーブの出力	同等性に基づく結合
ExtMergeJoin	いいえ	はい	1-n	1	いいえ	いいえ	はい

概要

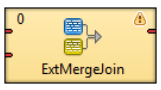
これは、ほとんどの一般的な状況で使用される汎用のジョイナです。入力をソートする必要があり、(**ExtHashJoin**とは異なり)キャッシュが行われないため非常に高速です。

最初の入力ポートに接続されるデータは**マスター**と呼ばれます(他の**ジョイナ**でも同様)。接続されているその他すべての入力ポートは**スレーブ**と呼ばれます。各マスター・レコードは、**結合キー**と呼ばれる1つ以上のフィールドで、すべてのスレーブ・レコードと照合されます。データのマージ方法の詳細は、データ・マージ(p.676)を参照してください。

ヒント

各種のキー・フィールドを含むキーに基づいて異なるスレーブをマスターと結合する場合は、かわりに**ExtHashJoin**を使用してください。ただし、スレーブ・データ・ソースは十分に小さい必要があります。

アイコン



ポート

ExtMergeJoinは、それぞれ個別のメタデータ構造を持つ可能性のある2つ以上の入力ポートを介してデータを受信します。

結合されたデータは1つの出力ポートに送信されます。

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	マスター入力ポート	任意
	1	はい	スレーブ入力ポート	任意
	2-n	いいえ	オプションのスレーブ入力ポート	任意
出力	0	はい	結合されたデータ用の出力ポート	任意

ExtMergeJoinの属性

属性	必須	説明	可能な値
Basic			
Join key	はい	それに基づいて受信データ・フローを結合するキー。Join key(p.674)を参照してください。	
Join type		結合のタイプ。結合タイプ(p.324)を参照してください。	Inner (デフォルト) | Left outer | Full outer
Transform	1)	グラフに定義されているCTLまたはJavaでの変換。	
Transform URL	1)	CTLまたはJavaでの変換を定義する外部ファイル。	
Transform class	1)	外部変換クラス。	
Allow slave duplicates		`true`に設定すると、重複するキー値を持つレコードが許可されます。`false`の場合、結合には**最初の**レコードのみが使用されます。	true (デフォルト) | false
Advanced			
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Ascending ordering of inputs		`true`に設定すると、受信レコードは昇順でソートされます。`false`に設定すると、降順になります。	true (デフォルト) | false
Deprecated			
Locale		国際化が使用されている場合に使用されるロケール。	
Case sensitive		`true`に設定すると、大文字と小文字が区別されます。デフォルトでは、これらは同一であるかのように処理されます。	false (デフォルト) | true
Error actions		指定した変換が**エラー・コード**を返した場合に実行する必要があるアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、**コンソール**に書き出されます。	
Left outer		`true`に設定すると、左外部結合が実行されます。デフォルトは`false`です。ただし、この属性よりも「**Join type**」が優先されます。両方設定した場合は、「**Join type**」のみが適用されます。	false (デフォルト) | true

属性	必須	説明	可能な値
Full outer		`true`に設定すると、完全外部結合が実行されます。デフォルトは`false`です。ただし、この属性よりも「**Join type**」が優先されます。両方設定した場合は、「**Join type**」のみが適用されます。	false (デフォルト) | true

説明:

1) これらのいずれかを設定する必要があります。これらの変換属性は指定する必要があります。これらの変換属性はいずれも**ジョイナ**の共通CTLテンプレートを使用するか、`RecordTransform`インタフェースを実装する必要があります。

詳細は、CTLスクリプトの詳細(p.676)またはJavaインタフェース(p.676)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

詳細説明

- **Join key**

  レコードを結合するために使用する必要があるキー(**結合キー**)を定義する必要があります。入力ポートのレコードは、「**Join key**」属性の該当部分に基づいてソートする必要があります。**結合キー**は「**Join key**」ウィザードで定義できます。

  「**Join key**」属性は、マスターおよびすべてのスレーブの各キー式をハッシュで区切ったシーケンスです。これらの式の順序は、最初がマスターでその後にスレーブが続く、入力ポートの順序に対応している必要があります。ドライバ(マスター)キーは、ドライバ(マスター)フィールド名(それぞれ先頭にドル記号が必要)をコロン、セミコロンまたはパイプで区切ったシーケンスです。各スレーブ・キーは、スレーブ・フィールド名(それぞれ先頭にドル記号が必要)をコロン、セミコロンまたはパイプで区切ったシーケンスです。

```
$EmployeeID;$CustomerID#$EmployeeID;$CustomerID#$ReportsTo;$CustomerID
```

図56.8. ExtMergeJoinコンポーネントの「Join Key」属性の例

この「**Join key**」ウィザードを使用できます。「**Join key**」属性行をクリックすると、そこにボタンが表示されます。このボタンをクリックすると、前述のウィザードを開くことができます。

そこに、ドライバ用のタブ(「**Master key**」タブ)とすべてのスレーブ入力ポート用のタブ(「**Slave key**」タブ)が表示されます。

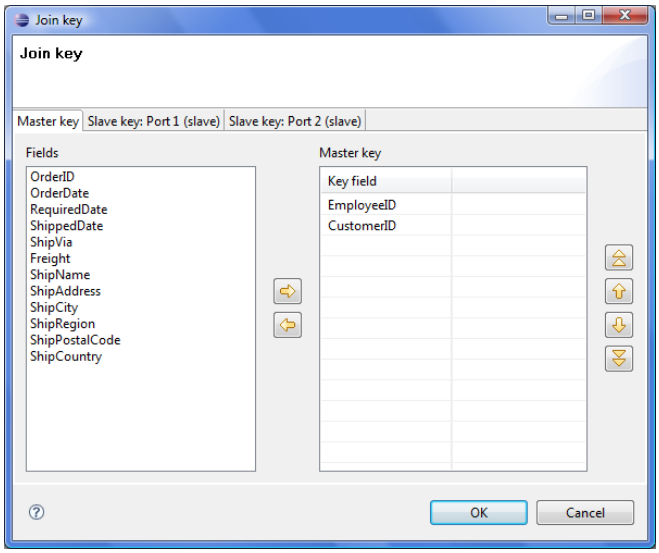


図56.9. 「Join Key」ウィザード(「Master Key」タブ)

ドライバ・タブには2つのペインがあります。左側の「**Fields**」ペインと右側の「**Master key**」ペインです。左側の「**Fields**」ペインでフィールドを選択し、**右矢印**ボタンを使用してそれを右側の「**Master key**」ペインに移動して、ドライバ式を選択する必要があります。選択された**マスター・キー**・フィールドに対し、各スレーブ内で同じ数のフィールドをマップする必要があります。したがって、すべての入力ポート(マスターと各スレーブの両方)で、キー・フィールドの数が同じになります。さらに、ドライバ(マスター)キーがすべてのスレーブで共通である必要があります。

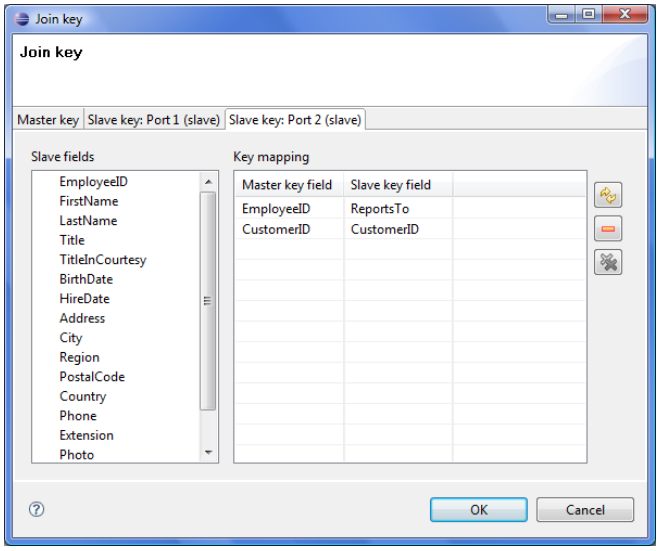


図56.10. 「Join Key」ウィザード(「Slave Key」タブ)

各スレーブ・タブには2つのペインがあります。左側の「**Fields**」ペインと右側の「**Key mapping**」ペインです。左側のペインには、スレーブ・フィールド名のリストが表示されます。右側のペインには、「**Master key field**」および「**Slave key field**」の2つの列が表示されます。左側の列には、選択されたドライバ入力ポートのフィールド名が含まれます。ドライバ・フィールドをスレーブ・フィールドにマップする場合は、スレーブ・フィールドを、左ペインでその項目クリックすることによって選択する必要があります。左マウス・ボタンを押し、右ペインの「**Slave key field**」列にドラッグしてボタンを放すと、スレーブ・フィールドをこの列に移動できます。スレーブごとに同じ操作を実行する必要があります。各タブで「**Auto mapping**」ボタンなどのボタンを使用することもできます。

例56.6. ExtMergeJoinの結合キー

```
$first_name;$last_name#$fname;$lname#$f_name;$l_name
```

マスター・データ・ソース(入力ポート0)の**結合キー**の一部を次に示します。

```
$first_name;$last_name
```

- したがって、これらのフィールドは最初のスレーブ・データ・ソース(入力ポート1)の2つのフィールドと結合されます。

  それぞれ`$fname`および`$lname`

- また、これらのフィールドは2番目のスレーブ・データ・ソース(入力ポート2)の2つのフィールドとも結合されます。

  それぞれ、`$f_name`および`$l_name`

データ・マージ

ExtMergeJoinでは、次の方法でデータの結合が行われます。最初にマスターとスレーブの両方のデータをソートする必要があります。

このコンポーネントは、マスターの最初のレコードを取得して、(**結合キー**に基づいて)スレーブの最初のレコードと比較します。次の3つの比較結果が考えられます。

- マスターとスレーブが等しい: レコードは結合されます。
- "slave.key < master.key": コンポーネントは次のスレーブ・レコードを参照します。つまり、現在のマスターと一致するスレーブを取得するために1ステップ・シフトが実行されます。
- "slave.key > master.key": コンポーネントは次のマスター・レコードを参照します。つまり、マスターで通常の1ステップ・シフトが実行されます。

同じ値のシーケンスが含まれる入力データもあります。この場合、これらはスレーブで1単位として処理されます(スレーブ・レコードは次のレコードの値を認識します)。これは、「**Allow slave duplicates**」が`true`に設定されている場合にのみ起こります。さらに、同じ値を含む単位はメモリーに保存されます。マスターでは、マスター・レコードが順番にスレーブと比較されて、同様にマージが行われます。

注意

スレーブに重複する値が多数存在する場合、それらはディスクに保存されます。

CTLスクリプトの詳細

結合属性を定義する場合は、入力データ・ソースのフィールドを出力にマップする変換を指定する必要があります。これを行うには、**変換エディタ**の「**Transformations**」タブを使用します。ただし、この最も簡単なアプローチを使用すると、より詳細な変換を指定できないことがあります。この場合は、CTLスクリプトを使用する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム・フィールド・マッピングを指定できます。

すべての**ジョイナ**は、ジョイナ用CTLテンプレート(p.325)で説明する同じ変換テンプレートを共有します。

Javaインタフェース

Javaで変換を定義する場合は、すべての**ジョイナ**に共通の次のインタフェースを変換で実装する必要があります。

ジョイナ用Javaインタフェース(p.328)

LookupJoin

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第46章「ジョイナの共通プロパティ」(p.323)

目的に適した**ジョイナ**を見つけるには、ジョイナの比較(p.323)を参照してください。

参照表の詳細は、第27章「参照表」(p.195)を参照してください。

要約

1つの入力ポートを介して受信したあるデータ・ソースのソートされていない可能性のあるデータを参照表の別のデータ・ソースと共通キーに基づいてマージする、汎用のジョイナです。

コンポーネント	同じ入力メタデータ	ソート済入力	スレーブ入力	出力	スレーブのないドライバの出力	ドライバのないスレーブの出力	同等性に基づく結合
LookupJoin	いいえ	いいえ	1 (仮想)	1-2	はい	いいえ	はい

概要

これは、ほとんどの一般的な状況で使用される汎用のジョイナです。入力をソートする必要がなく、メモリー内で処理されるため非常に高速です。

最初の入力ポートに接続されるデータは**マスター**と呼ばれ、2番目のデータ・ソースは**スレーブ**と呼ばれます。そのデータは、2番目(仮想)の入力ポートを介して受信したデータとみなされます。各マスター・レコードは、**結合キー**と呼ばれる1つ以上のフィールドで、スレーブ・レコードと照合されます。出力は、結合された入力を出力にマップする変換を適用することにより生成されます。

スレーブ・データは参照表から取得されるため、参照表によってはデータがメモリー内でソートされることがあります。また、参照表のタイプによっても異なります。たとえば、**データベース参照**の場合、すでに問合せされた値のみが保存されます。マスター・データはメモリーに保存されません。

アイコン

ポート

LookupJoinは1つの入力ポートを介してデータを受信し、それを参照表のデータと結合します。いずれかのデータ・ソースに異なるメタデータ構造が含まれる可能性があります。

結合されたデータは最初の出力ポートに送信されます。オプションで、2番目の出力ポートを使用して一致のないマスター・レコードを取得できます。

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	マスター入力ポート。	任意
	1 (仮想)	はい	スレーブ入力ポート。	任意
出力	0	はい	結合されたデータ用の出力ポート。	任意
	1	いいえ	スレーブ一致のないマスター・データ・レコード用のオプションの出力ポート。(「**Join type**」属性が`Inner join`に設定されている場合のみ。)これは**LookupJoin**および**DBJoin**にのみ適用されます。	入力0

LookupJoinの属性

属性	必須	説明	可能な値
Basic			
Join key	はい	それに基づいて受信データ・フローを結合するキー。Join key(p.679)を参照してください。	
Left outer join		`true`に設定すると、対応するスレーブがないドライバ・レコードも解析されます。そうでない場合は内部結合が実行されます。	false (デフォルト) | true
Lookup table	はい	スレーブ・レコードのリソースとして使用される参照表のID。参照キーのフィールドとそのデータ型の数は、**結合キー**のものと同じである必要があります。これらのフィールド値が比較され、一致したレコードが結合されます。	
Transform	1)	グラフに定義されているCTLまたはJavaでの変換。	
Transform URL	1)	CTLまたはJavaでの変換を定義する外部ファイル。	
Transform class	1)	外部変換クラス。	
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)
Advanced			
Free lookup table after finishing		`true`に設定すると、解析終了後に参照表は空になります。	false (デフォルト) | true
Deprecated			
Error actions		指定した変換が**エラー・コード**を返した場合に実行する必要があるアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、**コンソール**に書き出されます。	

説明:

1) これらのいずれかを設定する必要があります。これらの変換属性は指定する必要があります。これらの変換属性はいずれも**ジョイナ**の共通CTLテンプレートを使用するか、`RecordTransform`インタフェースを実装する必要があります。

詳細は、CTLスクリプトの詳細(p.679)またはJavaインタフェース(p.679)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

詳細説明

- **Join key**

  レコードを結合するために使用する必要があるキー(**結合キー**)を定義する必要があります。これは、入力メタデータのフィールド名をセミコロン、コロンまたはパイプで区切ったシーケンスです。「**Edit key**」ウィザードでキーを定義できます。

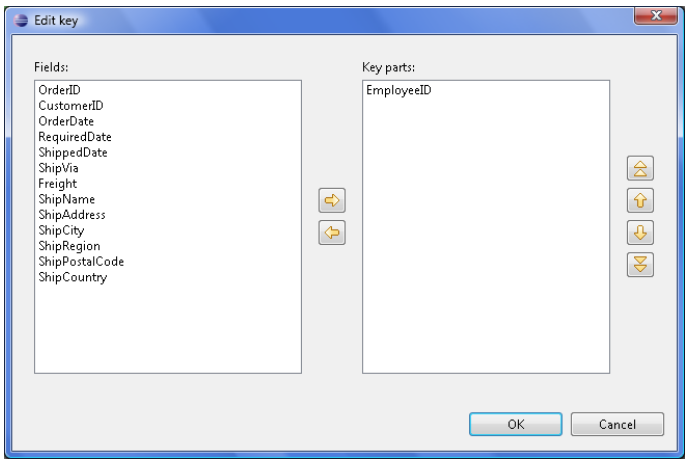


図56.11. 「Edit Key」ウィザード

入力メタデータのこの**結合キー**に対応するのは、**参照表の参照表キー**です。これは、参照表自体に指定されます。

例56.7. LookupJoinの結合キー

```
$first name;$last name
```

これは、マスター・レコードをスレーブ・レコードと結合する役割を果たすフィールドの主要部分です。

参照キーは次のようになります。

```
$fname;$lname
```

対応するフィールドが比較され、一致する値によりマスターおよびスレーブ・レコードが結合されます。

CTLスクリプトの詳細

結合属性を定義する場合は、入力データ・ソースのフィールドを出力にマップする変換を指定する必要があります。これを行うには、**変換エディタ**の「**Transformations**」タブを使用します。ただし、この最も簡単なアプローチを使用すると、より詳細な変換を指定できないことがあります。この場合は、CTLスクリプトを使用する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム・フィールド・マッピングを指定できます。

すべての**ジョイナ**は、ジョイナ用CTLテンプレート(p.325)で説明する同じ変換テンプレートを共有します。

Javaインタフェース

Javaで変換を定義する場合は、すべての**ジョイナ**に共通の次のインタフェースを変換で実装する必要があります。

ジョイナ用Javaインタフェース(p.328)

RelationalJoin

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第46章「ジョイナの共通プロパティ」(p.323)

目的に適した**ジョイナ**を見つけるには、ジョイナの比較(p.323)を参照してください。

要約

2つ以上のデータ・ソースのソート済データを共通キーに基づいてマージするジョイナです。これらのデータ・ソース内の値は異なっている必要があります。

コンポーネント	同じ入力メタデータ	ソート済入力	スレーブ入力	出力	スレーブのないドライバの出力	ドライバのないスレーブの出力	同等性に基づく結合
RelationalJoin	いいえ	はい	1	1	いいえ	いいえ	いいえ

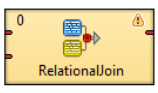
概要

これは、異なるフィールド値を含むデータ・レコードを結合する必要がある場合に役立つジョイナです。入力をソートする必要があり、メモリー内で処理されるため非常に高速です。

他の**ジョイナ**の場合と同様に、最初の入力ポートに接続されるデータは**マスター**と呼ばれます。接続されるもう一方の入力ポートは**スレーブ**と呼ばれます。各マスター・レコードは、結合キーと呼ばれる1つ以上のフィールドで、すべてのスレーブ・レコードと照合されます。この結合キーの値が対応するスレーブと異なるスレーブ・レコードは、このようなスレーブと結合されます。出力は、結合された入力を出力にマップする変換を適用することにより生成されます。

すべてのスレーブ入力データはメモリーに保存されますが、マスター・データはメモリーに保存されません。したがって、メモリー所要量にはスレーブ・データのサイズのみを考慮してください。

アイコン



ポート

RelationalJoinは、それぞれ個別のメタデータ構造を持つ可能性のある2つの入力ポートを介してデータを受信します。

結合されたデータは1つの出力ポートに送信されます。

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	マスター入力ポート	任意
	1	はい	スレーブ入力ポート	任意
出力	0	はい	結合されたデータ用の出力ポート	任意

RelationalJoinの属性

属性	必須	説明	可能な値
Basic			
Join key	はい	それに基づいて受信データ・フローを結合するキー。Join key(p.681)を参照してください。	
Join relation	はい	ドライバ(マスター)レコードとスレーブ・レコードの結合方法を定義します。Join relation(p.683)を参照してください。	`master != slave` | `master(D) < slave(D)` | `master(D) <= slave(D)` | `master(A) > slave(A)` | `master(A) >= slave(A)`
Join type		結合のタイプ。結合タイプ(p.324)を参照してください。	Inner (デフォルト) | Left outer | Full outer
Transform	1)	グラフに定義されているCTLまたはJavaでの変換。	
Transform URL	1)	CTLまたはJavaでの変換を定義する外部ファイル。	
Transform class	1)	外部変換クラス。	
Transform source charset		変換を定義する外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト)

説明:

1) これらのいずれかを設定する必要があります。これらの変換属性は指定する必要があります。これらの変換属性はいずれも**ジョイナ**の共通CTLテンプレートを使用するか、`RecordTransform`インタフェースを実装する必要があります。

詳細は、CTLスクリプトの詳細(p.683)またはJavaインタフェース(p.683)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

詳細説明

- **Join key**

  レコードを結合するために使用する必要があるキー(**結合キー**)を定義する必要があります。入力ポートのレコードは、「**Join key**」属性の該当部分に基づいてソートする必要があります。**結合キー**は「**Join key**」ウィザードで定義できます。

  「**Join key**」属性は、マスターおよびすべてのスレーブの各キー式をハッシュで区切ったシーケンスです。これらの式の順序は、最初がマスターでその後にスレーブが続く、入力ポートの順序に対応している必要があります。ドライバ(マスター)キーは、ドライバ(マスター)フィールド名(それぞれ先頭にドル記号が必要)をコロン、セミコロンまたはパイプで区切ったシーケンスです。各スレーブ・キーは、スレーブ・フィールド名(それぞれ先頭にドル記号が必要)をコロン、セミコロンまたはパイプで区切ったシーケンスです。

```
$EmployeeID;$CustomerID#$EmployeeID;$CustomerID#$ReportsTo;$CustomerID
```

図56.12. RelationalJoinコンポーネントの「Join Key」属性の例

この「**Join key**」ウィザードを使用できます。「**Join key**」属性行をクリックすると、そこにボタンが表示されます。このボタンをクリックすると、前述のウィザードを開くことができます。

そこに、ドライバ用のタブ(「**Master key**」タブ)とすべてのスレーブ入力ポート用のタブ(「**Slave key**」タブ)が表示されます。

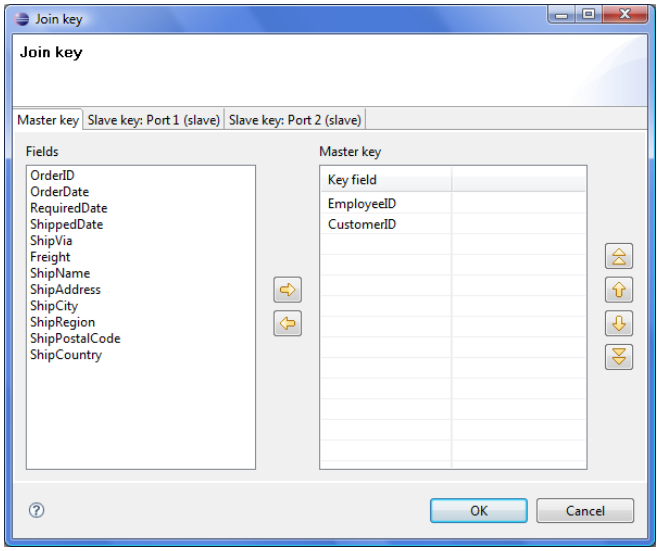


図56.13. 「Join Key」ウィザード(「Master Key」タブ)

ドライバ・タブには2つのペインがあります。左側の「**Fields**」ペインと右側の「**Master key**」ペインです。左側の「**Fields**」ペインでフィールドを選択し、**右矢印**ボタンを使用してそれを右側の「**Master key**」ペインに移動して、ドライバ式を選択する必要があります。選択された**マスター・キー**・フィールドに対し、各スレーブ内で同じ数のフィールドをマップする必要があります。したがって、すべての入力ポート(マスターと各スレーブの両方)で、キー・フィールドの数が同じになります。さらに、ドライバ(マスター)キーがすべてのスレーブで共通である必要があります。

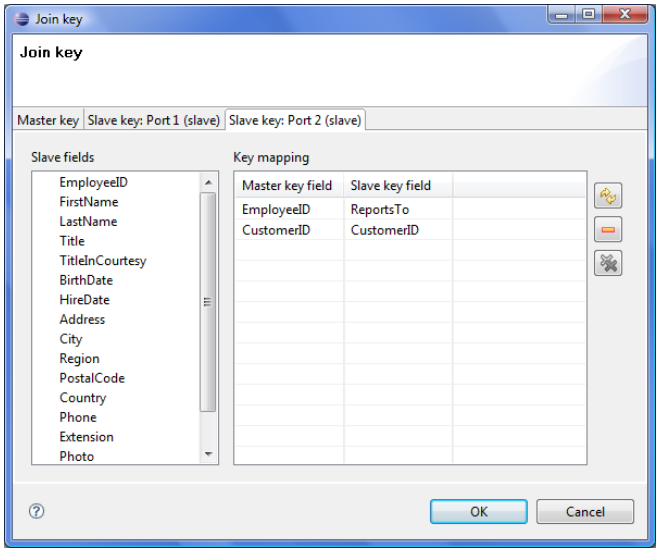


図56.14. 「Join Key」ウィザード(「Slave Key」タブ)

各スレーブ・タブには2つのペインがあります。左側の「**Fields**」ペインと右側の「**Key mapping**」ペインです。左側のペインには、スレーブ・フィールド名のリストが表示されます。右側のペインには、「**Master key field**」および「**Slave key field**」の2つの列が表示されます。左側の列には、選択されたドライバ入力ポートのフィールド名が含まれます。ドライバ・フィールドをスレーブ・フィールドにマップする場合は、スレーブ・フィールドを、左ペインでその項目クリックすることによって選択する必要があります。左マウス・ボタンを押し、右ペインの「**Slave key field**」列にドラッグしてボタンを放すと、スレーブ・フィールドをこの列に移動できます。スレーブごとに同じ操作を実行する必要があります。各タブで「**Auto mapping**」ボタンなどのボタンを使用することもできます。

例56.8. RelationalJoinの結合キー

```
$first_name;$last_name#$fname;$lname#$f_name;$l_name
```

マスター・データ・ソース(入力ポート0)の**結合キー**の一部を次に示します。

`$first_name=$fname;$last_name=$lname`

- したがって、これらのフィールドは最初のスレーブ・データ・ソース(入力ポート1)の2つのフィールドと結合されます。

  それぞれ`$fname`および`$lname`

- また、これらのフィールドは2番目のスレーブ・データ・ソース(入力ポート2)の2つのフィールドとも結合されます。

  それぞれ、`$f_name`および`$l_name`

- **Join relation**
  - 両方の入力ポートが降順でソートされているデータ・レコードを受信した場合、ドライバ(マスター)データ・レコード以上のスレーブ・データ・レコードのみがドライバ・データ・レコードと結合され、出力ポートを介して送信されます。対応する**結合関係**は、`master(D) < slave (D)` (スレーブがマスターより大きい)と`master(D) <= slave(D)` (スレーブがマスター以上)の2つのうちいずれかです。
  - 両方の入力ポートが昇順でソートされているデータ・レコードを受信した場合、ドライバ(マスター)データ・レコード以下のスレーブ・データ・レコードのみがドライバ・データ・レコードと結合され、出力ポートを介して送信されます。対応する**結合関係**は、`master(A) > slave (A)` (スレーブがドライバより小さい)と`master(A) >= slave(A)` (スレーブがドライバ以下)の2つのうちいずれかです。
  - 両方の入力ポートがソートされていないデータ・レコードを受信した場合、ドライバ(マスター)データ・レコードと異なるスレーブ・データ・レコードのみがドライバ・データ・レコードと結合され、出力ポートを介して送信されます。対応する**結合関係**は、`master != slave` (スレーブはドライバと異なる)です。
  - ソートされた順序と**結合関係**のこれ以外の組合せでは、グラフは失敗します。

CTLスクリプトの詳細

結合属性を定義する場合は、入力データ・ソースのフィールドを出力にマップする変換を指定する必要があります。これを行うには、**変換エディタ**の「**Transformations**」タブを使用します。ただし、この最も簡単なアプローチを使用すると、より詳細な変換を指定できないことがあります。この場合は、CTLスクリプトを使用する必要があります。

Clover Transformation Languageの詳細は、第IX部「CTL: CloverETL Transformation Language」(p.811)を参照してください。(CTLは本格的でありながら単純な言語であり、考えられるほぼすべての変換を実行できます。)

CTLスクリプトでは、単純なCTLスクリプト言語を使用してカスタム・フィールド・マッピングを指定できます。

すべての**ジョイナ**は、ジョイナ用CTLテンプレート(p.325)で説明する同じ変換テンプレートを共有します。

Javaインタフェース

Javaで変換を定義する場合は、すべての**ジョイナ**に共通の次のインタフェースを変換で実装する必要があります。

ジョイナ用Javaインタフェース(p.328)

第57章 ジョブ制御

コンポーネントとは何かを理解していることを想定しています。概要は、第19章「コンポーネント」(p.97)を参照してください。

一部のコンポーネントは、各種のジョブ・タイプの実行と監視に焦点を当てています。このコンポーネント・グループを**ジョブ制御**と呼びます。

ジョブ制御コンポーネントは、通常はジョブフロー(p.250)と緊密にバインドされています。ただし、通常のETLグラフでも、使用できるのはそのうちわずかです。

これらのコンポーネントでは、ETLグラフ、ジョブフローおよび解析済スクリプトを実行できます。グラフおよびジョブフローは、監視および中断(オプション)できます。

コンポーネントには様々なプロパティを設定できます。ただし、一部のものが共通する場合があります。すべてのコンポーネントに共通のプロパティもあれば、ほとんどのコンポーネントに共通のプロパティもあります。次のことを学習している必要があります。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

ジョブ制御グループの各コンポーネントは、その実行対象のタスクに従って区別できます。

- Barrier(p.685)は、並行して実行されるジョブの結果を待ち、集約された結果を出力ポートに送信します。
- Condition(p.688)は、指定された条件の結果に基づいて受信トークンをその出力ポートの1つにルーティングします。
- ExecuteGraph(p.690)は、ユーザー指定の設定でサブグラフを実行します。
- ExecuteJobflow(p.697)は、ユーザー指定の設定でジョブフローを実行します。
- ExecuteProfilerJob(p.709)は、ユーザー指定の設定でプロファイラ・ジョブを実行します。
- ExecuteScript(p.712)は、シェル・スクリプト、または選択したインタプリタにより解釈されたスクリプトを実行します。
- Fail(p.717)は親ジョブを中断します。
- GetJobInput(p.719)は、ディクショナリ・コンテンツにより移入された1つのレコードを生成します。
- KillGraph(p.721)は、指定されたグラフを中断します。
- KillJobflow(p.724)は、指定されたジョブフローを中断します。
- MonitorGraph(p.725)は、実行中のグラフを監視します。
- MonitorJobflow(p.728)は、実行中のジョブフローを監視します。
- SetJobOutput(p.729)は、受信値をディクショナリ・コンテンツに設定します。
- Success(p.731)は、成功したとみなされるすべての受信トークンまたはレコードを消費します。
- TokenGather(p.733)は、任意の入力ポートから受信したトークンをすべての出力ポートにコピーします。

Barrier

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

要約

Barrierを使用すると、パラレルで実行されているジョブの結果を待機し、簡単な方法でジョブ・グループの成功または失敗に対応できます。

Barrierは、グループに属するすべての入力トークンを待機して、このグループを評価し、結果に基づいて出力トークンを最初の出力ポートまたは2番目の出力ポートに送信します。

注意

このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。また、プロジェクトがClover Serverで実行されている必要もあります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
Barrier	いいえ	いいえ	1-n	1-2	いいえ	いいえ	いいえ

概要

Barrierは主に、パラレルで実行されているジョブの管理に使用されます。Barrierは、ジョブ結果の情報を格納するすべての受信トークンを受信し、それらを論理ジョブ・グループに分割します。各ジョブ・グループは個別に評価されます。成功と評価されたグループの結果が最初の出力ポートに送信されます。失敗したグループの結果は2番目の出力ポートに送信されます。

受信レコードの論理グループ

コンポーネント属性「Input grouping」には、受信トークンを論理ジョブ・グループに分割する方法を指定する2つのオプションがあります。

- **All**: コンポーネントではすべての受信トークンが1つのグループとみなされ、1つのグループのみが処理されます。これはほとんどの一般的なシナリオ(前のジョブがすべて成功しているかチェックする場合など)で使用できます。

- **Tuple**: すべての入力ポートからの1つのトークンで構成されるグループ。つまり、入力ポートからのトークンの波によってこれらのグループが作成されます。各入力ポートから最初に到着したトークンが最初の論理グループを構成し、各入力ポートからの2番目のトークンが2番目の論理グループを構成します(つまり、すべての入力ポートからのn番目の入力トークンがn番目のグループを構成します)。この設定は、一連のパラレル・ジョブの結果をチェックする場合に使用できます。

グループ評価

グループ内の各トークンは、コンポーネント属性「Token evaluation expression」のCTLブール式により評価されます。これをジョブ・ステータスと呼びます。論理演算ANDまたはORにより結合されたジョブ・ステータス(コンポーネント属性「Group evaluation criteria」)がtrueである場合にのみ、グループは成功したとみなされます。したがって、AND演算の場合、グループ全部が成功するためにはすべての受信トークンが成功する必要があります。これに対し、OR演算の場合は、グループの少なくとも1つのトークンが成功すればグループは成功します。

出力トークンの生成

成功したグループはその結果を最初の出力ポートに送信し、失敗したグループはその結果を2番目の出力ポートに送信します。出力トークンの数およびコンテンツは、コンポーネント属性「Output」によって指定します。

- **Single token**: 各グループにつき1つのみのトークンが出力ポートに送信され、トークンはすべてのグループ・トークンにより移入されます。フィールドは、フィールド名に基づいてコピーされます(コピーされる入力トークン順序は保証されません)。
- **All tokens**: 互換性のないメタデータ・フィールドがフィールド名に基づいてコピーされる場合、グループからの各受信トークンは専用の出力ポートに送信されます。

注意

出力ポートは必要ありません。存在しないエッジにルーティングされたトークンは、出力なしで破棄されます。

使用例

簡単な使用例を示します。

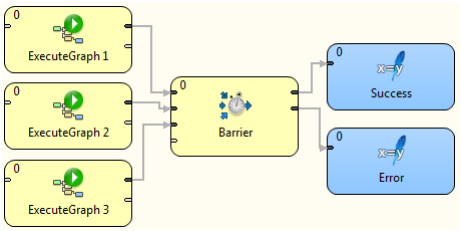


図57.1. Barrierコンポーネントの典型的な使用例

この例では、3つの異なるグラフが3つのExecuteGraphコンポーネントによって同期的に実行されます。3つのグラフはすべて並行して実行されています。Barrierはグラフ実行結果の収集ポイントであるため、すべてのグラフが完了するのを待ってから次の手順に進みます。すべてのグラフが正常に完了すると、出力トークンが最初の出力ポートに送信されます。これに対し、1つ以上のグラフが失敗した場合、出力トークンは2番目の出力ポートに送信されます。このコンポーネントを使用すると、ジョブ・グループ全部のステータスの簡略的な評価が可能になります。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	ジョブ結果を含む入力トークン	任意1)
	1-n	いいえ	ジョブ結果を含む入力トークン	任意1)
出力	0	いいえ	成功したジョブ・グループのトークン	任意2)
	1	いいえ	失敗したジョブ・グループのトークン	任意2)

説明:

1): 任意のメタデータが使用可能です。「Token evaluation expression」属性では、デフォルトで「status」フィールドのみが予期されます($status == "FINISHED_OK")。

2): 任意のメタデータが使用可能ですが、出力ポートに送信されたトークンは入力ポートのフィールド名に基づいてコピーされるため、等しい名前を含むフィールドのみが移入されます。

Barrierの属性

属性	必須	説明	可能な値
Basic			
Input grouping	いいえ	受信トークンを個別評価されるジョブ・グループに分割する方法を指定するアルゴリズムのタイプ。受信レコードの論理グループ(p.685)を参照してください。	Tuple (デフォルト) | All
Token evaluation expression	いいえ	受信トークンがジョブ実行の成功または失敗(最終ジョブ・ステータス)のどちらを表すかを判別するために各トークンに適用されるCTLのブール式。グループ評価(p.686)を参照してください。	default CTL expression '$status == "FINISHED_OK"'
Group evaluation criteria	いいえ	ジョブ・グループが成功したか失敗したかを判別するためにジョブ・ステータスに適用される論理演算(属性「Token evaluation expression」を参照してください)。グループ評価(p.686)を参照してください。	AND (デフォルト) | OR
Output	いいえ	各ジョブ・グループの出力トークンの数を決定します。出力トークンの生成(p.686)を参照してください。	Single token (デフォルト) | All tokens

Condition

Jobflowコンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

コンポーネントは「**Palette**」→「**Job Control**」にあります。

要約

Conditionコンポーネントは、指定された条件の結果に基づいて受信トークンをその出力ポートの1つにルーティングします。これは、プログラミング言語における`if`文と同様です。

注意

このコンポーネントを使用するには、ライセンスでJobflowがサポートされている必要があります。また、プロジェクトがClover Serverで実行されている必要もあります。

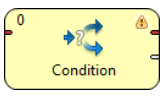
コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
Condition	–	いいえ	1	1–2	いいえ	–	–

概要

受信トークンごとに、**Condition**コンポーネントは指定されたブール条件を評価します。結果がtrueの場合、トークンは最初の出力ポートに送信され、それ以外の場合、2番目の(オプションの)出力ポートに送信されます。

Conditionは、**ExtFilter**(p.597)コンポーネントと同様に機能します。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力トークン用	任意
出力	0	はい	条件と互換性のあるトークン用	入力0[1)]
	1	いいえ	条件を満たしていないトークン用	入力0[1)]

説明:

[1)]メタデータはこのコンポーネントを介して伝播できます。すべての出力メタデータが同じである必要があります。

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

Conditionの属性

属性	必須	説明	可能な値
Basic			
Condition	はい	トークンのフィルタに使用するブール式。各入力フィールドの各式をセミコロンで区切ったシーケンスとして表されます。	

詳細説明

Condition属性の詳細は、**ExtFilter**コンポーネントのフィルタ式(p.598)を参照してください。

ExecuteGraph

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

ヒント

`*.grf`ファイルをドラッグして「jobflow」ペインにドロップすると、**ExecuteGraph**コンポーネントが追加されます。

要約

ExecuteGraphを使用すると、ユーザー指定の設定でサブグラフを実行でき、実行結果およびトラッキング詳細が出力ポートに送信されます。

注意

このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。また、プロジェクトがClover Serverで実行されている必要もあります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
ExecuteGraph	いいえ	いいえ	0-1	0-2	はい	いいえ	はい

概要

ExecuteGraphコンポーネントは、特定の設定でサブグラフを実行し、グラフの終了を待ってからグラフ結果およびトラッキング情報を監視します。

このコンポーネントには、1つの入力ポートと2つの出力ポートを接続できます。このコンポーネントは入力トークンを読み取って、受信データ値に基づいてサブグラフを実行します。次に、サブグラフの終了を待ってから、成功したサブグラフの結果を最初の出力ポートに送信し、失敗したサブグラフの結果を2番目の出力ポート(エラー・ポート)に送信します。グラフ実行が成功した場合、コンポーネントは引き続き次の入力トークンを処理します。それ以外の場合、コンポーネントは他のグラフの実行を停止します。それ以降はすべての受信トークンが無視され、無視されたトークンの情報がエラー出力ポートに送信されます。この動作は、**「Stop processing on fail」**属性によって変更できます。

入力ポートが接続されていない場合、コンポーネントの属性で指定されているデフォルト設定で1つのグラフのみが実行されます。最初の出力ポートが接続されていない場合、コンポーネントはサブグラフ結果のログへの出力のみを行います。2番目の出力ポート(エラー・ポート)が接続されていない場合、最初に失敗したサブグラフによって親ジョブが中断されます。

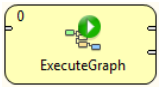
グラフを実行するには、サブグラフの場所および実行タイプを指定する必要があります。また、オプションで、グラフ・パラメータとディクショナリ・コンテンツの初期値、タイムアウト、実行グループおよびその他のいくつかの属性も指定できます。これらの実行設定のほとんどは、後述する各種のコンポーネント属性によってコンポーネントに指定できます。これらの設定は、デフォルトの実行設定とみなすことができます。ただし、これらのデフォルト実行設定は、受信トークンのデータに基づいて、グラフ実行ごとに動的に変更できます。このオーバーライドは、「Input mapping」属性で定義します。

サブグラフの終了後、結果を出力ポートにマップできます。「Output mapping」属性および「Error mapping」属性は、出力トークンの移入方法を定義します。グラフ結果で使用可能な情報は主に、一般的なランタイム情報、最終ディクショナリ・コンテンツおよびトラッキング情報で構成されます。

ヒント

コンポーネントを右クリックして「**Open Graph**」をクリックすると、実行対象のグラフにアクセスできます。同様に、**ExecuteJobflow**コンポーネントと**ExecuteProfilerJob**コンポーネントには、「**Open Jobflow**」オプションと「**Open Profiler Job**」オプションがあります。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	グラフ実行設定を含む入力トークン	任意
出力	0	いいえ	成功したサブグラフの実行情報	任意
	1	いいえ	失敗したサブグラフの実行情報	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

ExecuteGraphの属性

属性	必須	説明	可能な値
Basic			
Graph URL	はい	実行対象のサブグラフのパス。この属性では、1つのグラフのみを指定できます。入力トークンの値でオーバーライドできます。「Input mapping」属性を参照してください。この属性により参照されるグラフはすべてのマッピング・ダイアログでも使用されます。マッピング・ダイアログには、このグラフに基づくディクショナリ・エントリおよびトラッキング情報が表示されます。	
Execution type	いいえ	実行のタイプとして、同期(順序)または非同期(パラレル)実行モデルのいずれかを指定します。入力トークンの値でオーバーライドできます。「Input mapping」属性を参照してください。実行タイプ(p.693)を参照してください。	synchronous (デフォルト) | asynchronous

属性	必須	説明	可能な値
Timeout	いいえ	サブグラフ実行専用の最大時間。デフォルトではミリ秒ですが、他の時間単位(p.275)も使用できます。このタイムアウト間隔が経過すると、サブグラフは中断されます。入力トークンの値でオーバーライドできます。「Input mapping」属性を参照してください。 非同期実行タイプの場合、「Timeout」属性は無視されます。実行中のグラフを監視するには、MonitorGraphコンポーネントを使用します。	0 (無制限) | 正数
Input mapping	いいえ	「Input mapping」は、受信トークンのデータでデフォルトの実行設定をオーバーライドする方法を定義します。入力マッピング(p.693)を参照してください。	CTL変換
Output mapping	いいえ	「Output mapping」は、成功したグラフの結果を最初の出力ポートにマップします。出力マッピング(p.694)を参照してください。	CTL変換
Error mapping	いいえ	「Error mapping」は、失敗したグラフの結果を2番目の出力ポートにマップします。エラー・マッピング(p.696)を参照してください。	CTL変換
Redirect error output	いいえ	デフォルトでは、失敗したグラフの結果は2番目の出力ポート(エラー・ポート)に送信されます。これをtrueに変更した場合、失敗したグラフの結果が成功したグラフと同様に最初の出力ポートに送信されます。	false (デフォルト) | true
Advanced			
Execution group	いいえ	実行対象のサブグラフが属する実行グループの名前。KillGraphコンポーネントは、中断されるグラフ・セットの名前付きハンドラとして実行グループを使用できます。	string
Preferred cluster node ID	いいえ	サブグラフの実行で優先されるクラスタ・ノードID。クラスタ・ノードは作業環境であり、保証はされません。	string
Execute graph as daemon	いいえ	デフォルトでは、すべてのサブグラフが非デーモン・モードで実行されるため、親のグラフよりも長く存続できるサブグラフはありません。Clover serverは、完了したジョブについて、まだ完了していないすべての非デーモン・サブグラフを中断することを自動的に保証します。親グラフよりも長く存続できるサブグラフを開始する場合は、このスイッチをtrueに設定します。	false (デフォルト) | true
Skip checkConfig	いいえ	デフォルトでは、親のジョブに対して実行前構成チェックが実行された場合にのみ、サブグラフの実行前構成チェックが実行されます。この属性によって、チェックを明示的に有効化または無効化できます。	boolean (デフォルトは親ジョブから継承)
Stop processing on fail	いいえ	デフォルトでは、失敗したサブグラフがあると、コンポーネントは他のサブグラフの実行を停止し、スキップされたトークンに関する情報がエラー出力ポートに送信されます。この属性で、この動作をオフにできます。	true (デフォルト) | false
Number of executors	いいえ	デフォルトでは、同期モードで実行されるサブグラフが順番にトリガーされます。次のグラフは、その前のグラフが完了すると実行されます。このオプションを指定すると、同時に実行されるグラフの数を増やすことができます。「Number of executors」属性は、一度に実行できるサブグラフの数を定義します。それらはすべて監視され、実行中のサブグラフのうち1つの処理が完了すると、別のサブグラフが実行されます。このオプションは、同期モードで実行されるサブグラフにのみ適用されます。	正数(1がデフォルト)

実行タイプ

このコンポーネントは、グラフの同期(順序)実行および非同期(パラレル)実行をサポートします。

- 同期実行モード(デフォルト): コンポーネントはグラフの処理が完了するまでブロックするため、グラフは実行終了までこのコンポーネントにより監視されます。
- 非同期実行モード: コンポーネントはグラフを起動し、即時にステータス情報を出力に送信します。このコンポーネントでは、グラフは起動されるのみです。実行対象のグラフを非同期的に監視する場合は、コンポーネントMonitorGraphを使用してください。

入力マッピング

「Input mapping」属性を使用すると、受信トークンのデータに基づいてコンポーネントの設定をオーバーライドできます。さらに、入力マッピングでは実行対象のグラフの初期ディクショナリ・コンテンツおよびグラフ・パラメータを変更できます。

入力マッピングは、各サブグラフ実行の前に実行される通常のCTL変換です。このマッピングの入力は、入力トークン(ある場合)のみです。出力は、3つのレコード(RunConfig、JobParametersおよびDictionary)に基づきます。

- **RunConfig**レコードは実行設定を表します。このマッピングによってレコードのフィールドが移入されない場合、コンポーネントの該当する属性のデフォルト値がかわりに使用されます。

フィールド名	型	説明
jobURL	string	コンポーネント属性「**Graph URL**」をオーバーライドします。
executionType	string	コンポーネント属性「**Execution type**」をオーバーライドします。
timeout	long	コンポーネント属性「**Timeout**」をオーバーライドします。
executionGroup	string	コンポーネント属性「**Execution group**」をオーバーライドします。
clusterNodeId	string	コンポーネント属性「**Preferred cluster node ID**」をオーバーライドします。
daemon	boolean	コンポーネント属性「**Execute graph as daemon**」をオーバーライドします。
skipCheckConfig	boolean	コンポーネント属性「**Skip checkConfig**」をオーバーライドします。
jobParameters	map[string, string]	実行対象のグラフに渡されるグラフ・パラメータ。グラフ・パラメータの主要な定義方法は、入力マッピング・ダイアログで使用可能なJobParametersレコードに直接マッピングすることです。この方法では、実行対象のグラフから抽出された準備済グラフ・パラメータを簡単に移入できます。このマップを介して定義したグラフ・パラメータが最も優先されます。ジョブフローの設計時にグラフ・パラメータのセットが使用できない場合、このマップを使用できます。

- **JobParameters**レコードは、トリガーされたグラフの内部および外部のすべてのグラフ・パラメータを表します。
- **Dictionary**レコードは、トリガーされたグラフの入力ディクショナリ・エントリを表します。

注意

変換ダイアログでJobParametersレコードとDictionaryレコードが使用可能になるのは、コンポーネント属性「**Graph URL**」が、グラフ・パラメータとディクショナリ構造の抽出用のテンプレートとして使用される既存のグラフに関連付けられている場合のみです。実行時に実際にどのグラフが実行されるかに関係なく、このグラフのグラフ・パラメータおよび入力ディクショナリ・エントリのみを入力マッピングにより移入できます。

出力マッピング

出力マッピングは、最初の出力ポートに渡されたトークンを移入するために使用される標準的なCTL変換です。このマッピングは、成功したグラフに対して実行されます。このマッピングには最大4つの入力データ・レコードを使用できます。

- グラフの実行の基礎となった入力トークン(入力コネクタがないとコンポーネント使用には使用できません)。これは、入力トークンのいくつかのフィールドを出力トークンに渡す場合に非常に便利です。このレコードには、「**Type**」列に表示される**ポート0**があります。
- RunStatusレコードは、グラフ実行に関する情報を提供します。

フィールド名	型	説明
runId	long	サブグラフ実行の一意の識別子。非同期実行タイプの場合、グラフの監視または中断にこの値を使用できます。
orignalJobURL	string	実行対象のサブグラフのパス。
startTime	date	グラフが実行される時刻。
endTime	date	グラフの終了時刻(非同期実行の場合はnull)。
duration	long	ミリ秒単位のグラフ実行時間。
status	string	最終グラフ実行ステータス(非同期実行の場合はFINISHED_OK | ERROR | ABORTED | TIMEOUT | RUNNING)。
errException	string	失敗したグラフに対してのみ例外を発生させます。
errMessage	string	失敗したグラフのみのエラー・メッセージ。
errComponent	string	グラフの失敗の原因となったコンポーネントID。
errComponentType	string	グラフの失敗の原因となったコンポーネントのタイプ。

- Dictionaryレコードは、サブグラフの出力ディクショナリ・エントリのコンテンツを提供します。このレコードをマッピングに使用できるのは、「Graph URL」属性が出力ディクショナリ・エントリを含むサブグラフ・インスタンスを参照している場合のみです。
- Trackingレコードは、通常は実行中のグラフのJMXインタフェースでのみ使用可能なトラッキング情報を提供します。このレコードをマッピングに使用できるのは、「Graph URL」属性がサブグラフ・インスタンスを参照している場合のみです。
  - グラフ全体に使用可能なトラッキング・フィールド:

フィールド名	型	説明
startTime	date	グラフが実行される時刻。
endTime	date	グラフの終了時刻(実行中のグラフの場合はnull)。
executionTime	long	ミリ秒単位のグラフ実行時間。
graphName	string	実行対象のグラフの名前。
result	string	グラフ実行ステータス(FINISHED_OK | ERROR | ABORTED | TIMEOUT | RUNNING)。
runningPhase	integer	実行中フェーズのインデックス(グラフがすでに終了している場合はnull)。
usedMemory	integer	実行対象のグラフのメモリー・フットプリント(バイト単位)。

- グラフ・フェーズで使用可能なトラッキング・フィールド:

フィールド名	型	説明
startTime	date	フェーズが実行される時刻。
endTime	date	フェーズの終了時刻(実行中のフェーズの場合はnull)。
executionTime	long	ミリ秒単位のフェーズ実行時間。
memoryUtilization	long	グラフ・メモリー・フットプリント(バイト単位)。
result	string	フェーズ実行ステータス(FINISHED_OK | ERROR | ABORTED | RUNNING)。

- コンポーネントに使用可能なトラッキング・フィールド:

フィールド名	型	説明
name	string	コンポーネントの名前。
usageCPU	number	コンポーネントにより使用される実際のCPU時間を間隔からの数値で表したもの(0, 1)。(0はコンポーネントによりCPUの0%が使用されていることを示し、1はコンポーネントによりCPUの100%が使用されていることを示します。)
usageUser	number	ユーザー・モードでコンポーネントにより使用される実際のCPU時間を間隔からの数値で表したもの(0, 1)。(0はコンポーネントによりCPUの0%が使用されていることを示し、1はコンポーネントによりCPUの100%が使用されていることを示します。)
peakUsageCPU	number	コンポーネントにより使用される最大CPU時間を間隔からの数値で表したもの(0, 1)。(0はコンポーネントによりCPUの0%が使用されていることを示し、1はコンポーネントによりCPUの100%が使用されていることを示します。)
peakUsageUser	number	ユーザー・モードでコンポーネントにより使用される最大CPU時間を間隔からの数値で表したもの(0, 1)。(0はコンポーネントによりCPUの0%が使用されていることを示し、1はコンポーネントによりCPUの100%が使用されていることを示します。)
totalCPUTime	long	このコンポーネントにより使用されるCPU時間(ミリ秒単位)。
totalUsetTime	number	ユーザー・モードでこのコンポーネントにより使用されるCPU時間(ミリ秒単位)。
memoryUtilization	long	グラフ・メモリー・フットプリント(バイト単位)。
result	string	コンポーネント実行ステータス(FINISHED_OK | ERROR | ABORTED | RUNNING)。
usedMemory	integer	このコンポーネントのメモリー・フットプリント(バイト単位)。単なる実験的な実装です。

- 入力ポートまたは出力ポートに使用可能なトラッキング・フィールド:

フィールド名	型	説明
byteFlow	integer	このポートを介して渡される1秒当たりのバイト数。
bytePeak	integer	このポートに登録されている最大**byteFlow**。

フィールド名	型	説明
totalBytes	long	このポートを介して渡されるバイト数。
recordFlow	integer	このポートを介して渡される1秒当たりのレコード数。
recordPeak	integer	このポートに登録されている最大**recordFlow**。
totalRecords	integer	このポートを介して渡されるレコード数。
waitingRecords	integer	ポートに接続されているエッジでキャッシュされるレコード数。
averageWaitingRecords	integer	ポートに接続されたエッジでキャッシュされる平均レコード数。
usedMemory	integer	接続されたエッジのメモリー・フットプリント(バイト単位)。

エラー・マッピング

エラー・マッピングの属性は、出力マッピングとほぼ同じです。出力マッピングとは異なり、このエラー・マッピングは、グラフが失敗して終了し、最初の出力ポートではなく2番目の出力ポートが移入された場合にのみ使用されます。

ExecuteJobflow

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは「**Palette**」→「**Job Control**」にあります。

ヒント

`*.jbf`ファイルをドラッグして「jobflow」ペインにドロップすると、**ExecuteJobflow**コンポーネントが追加されます。

要約

ExecuteJobflowを使用すると、ユーザー指定の設定でジョブフローを実行でき、実行結果およびトラッキング詳細が出力ポートに送信されます。

注意

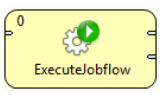
このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。また、プロジェクトがClover Serverで実行されている必要もあります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
ExecuteJobflow	いいえ	いいえ	0-1	0-2	はい	いいえ	はい

概要

このコンポーネントは**ExecuteGraph**と同様に動作します。ExecuteGraph(p.690)コンポーネントの説明を参照してください。

アイコン



ポート

ExecuteGraphの「ポート」(p.691)を参照してください。

ExecuteJobflowの属性

ExecuteGraphの属性(p.691)を参照してください。

ExecuteMapReduce

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

要約

ExecuteMapReduceは、指定されたMapReduceジョブをHadoopクラスタ上で実行します。

注意

このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
ExecuteMapReduce	いいえ	いいえ	0-1	0-2	いいえ	いいえ	はい

概要

ExecuteMapReduceコンポーネントは、提供された`.jar`ファイルで指定されているクラスを使用して実装されたHadoop MapReduceジョブを実行します。このコンポーネントは、定期的にHadoopクラスタにジョブ実行ステータスを問い合せ、この情報をグラフ・ログに出力します。

MapReduceジョブ・クラスは、Hadoop MapReduceジョブの新しいAPIおよび古いAPIの両方を使用して実装できます。新しいAPIを使用して実装すると、ジョブ・クラスは`org.apache.hadoop.mapreduce`パッケージから適切なクラスを拡張します。これに対し、古いAPIを使用したジョブ・クラスは`org.apache.hadoop.mapred`パッケージから適切なインタフェースを実装します。デフォルトでは、`ExecuteMapReduce`コンポーネントは新しいジョブAPIを想定します。古いAPIを使用してジョブが実装されている場合、**「Job implementation API version」**属性を明示的に設定する必要があります(次を参照してください)。

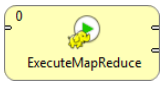
一般的なジョブ制御コンポーネントとして、**ExecuteMapReduce**には1つの入力ポートと2つの出力ポートを接続できます。このコンポーネントは入力トークンを読み取って、受信データ値に基づいてMapReduceジョブを実行します。次に、ジョブの終了を待ってから、成功したジョブの結果を最初の出力ポートに送信し、失敗したジョブの結果を2番目の出力ポート(エラー・ポート)に送信します。ジョブの実行が成功した場合、コンポーネントは引き続き次の入力トークンを処理します。それ以外の場合、コンポーネントは他のジョブの実行を停止します。それ以降はすべての受信トークンが無視され、無視されたトークンの情報がエラー出力ポートに送信されます。この動作は、**「Stop processing on fail」**属性によって変更できます。

入力ポートが接続されていない場合、コンポーネントの属性で指定されているデフォルト設定で1つのMapReduceジョブのみが実行されます。どちらの出力ポートもオプションです。

MapReduceジョブの実行では、少なくとも、Hadoop接続(p.192)、MapReduceジョブを実装するクラスを含む.jarファイルの場所、選択したHadoop接続により決定されたHDFSに配置されている入力ファイルと出力ディレクトリ、および出力キー/値クラスを指定する必要があります。これらの設定および他の(オプション)設定は、デフォルトの実行設定とみなすことができます。ただし、これらのデフォルト実行設定は、受信トークンのデータに基づいて、ジョブ実行ごとに動的に変更できます。このオーバーライドは、「**Input mapping**」属性で定義します。

MapReduceジョブの終了後、結果を出力ポートにマップできます。「Output mapping」属性および「Error mapping」属性は、出力トークンの移入方法を定義します。ジョブ結果で使用可能な情報は主に、一般的な実行時情報およびジョブ・カウンタ情報で構成されます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	MapReduceジョブ実行設定を含む入力トークン	任意
出力	0	いいえ	成功したジョブの実行情報	任意
	1	いいえ	失敗したジョブの実行情報	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

ExecuteMapReduceの属性

属性	必須	説明	可能な値
Basic			
Hadoop connection	はい	HDFSサーバー(NameNode)への接続とMapReduceサーバー(JobTracker)への接続の両方を定義するHadoop接続(p.192)。	
Job name	いいえ	ジョブ実行の任意のラベル。デフォルト値は、指定されたMapReduce .jarファイルの名前です。	任意の文字列
URL of JAR file with job classes	はい	MapReduceジョブを含む.jarファイルのパス。このファイルは、ローカル・ファイル・システムに存在する必要があります。	
Timeout (ms)	いいえ	ジョブ実行の制限時間(ミリ秒単位)。ジョブ実行時間がこの制限を超えると、ジョブは強制終了されます。制限を設定しない場合は、0 (デフォルト)に設定します。	0 (無制限) | 正数
Input mapping	いいえ	「Input mapping」は、受信トークンのデータでデフォルトの実行設定をオーバーライドする方法を定義します。	CTL変換
Output mapping	いいえ	「Output mapping」は、成功したMapReduceジョブの結果を最初の出力ポートにマップします。出力およびエラー・マッピング(p.707)を参照してください。	CTL変換
Error mapping	いいえ	「Error mapping」は、失敗したジョブの結果を2番目の出力ポートにマップします。出力およびエラー・マッピング(p.707)を参照してください。	CTL変換

属性	必須	説明	可能な値
Redirect error output	いいえ	デフォルトでは、失敗したジョブの結果は2番目の出力ポート(エラー・ポート)に送信されます。このスイッチをtrueにした場合、失敗したジョブの結果が成功したジョブと同様に最初の出力ポートに送信されます。	false (デフォルト) | true
Job folders			
Input files	はい	HDFSに配置されている入力ファイルの1つ以上のパス。パスは、HDFS URLの形式で指定できます(例: hdfs://CONN_ID/path/to/inputfile)。ここで、Hadoop接続ID CONN_IDは、「**Hadoop connection**」属性で指定されている接続のIDと一致している必要があります。あるいは、単にHDFS上の絶対パス(例: /path/to/inputfile)にしたり、ジョブの作業ディレクトリを基準とした相対パス(例: relative/path/to/inputfile)にすることもできます。	
Output directory	はい	HDFSに配置されている出力ディレクトリのパス。このディレクトリが存在しない場合は、作成されます(「**Clear output directory before execution**」属性を参照してください)。HDFS URLまたはHDFS上の絶対パスや作業ディレクトリを基準にした相対パスは、「**Input files**」属性と同様にここで指定できます(例: hdfs://CONN_ID/path/to/outputdir、/path/to/outputdir、relative/path/to/outputdir)。	
Job's working directory	いいえ	HDFS上のMapReduceジョブの作業ディレクトリの場所。これは、HDFS URL (例: hdfs://CONN_ID/path/to/workdir)にすることも、HDFS上の絶対パス(例: /path/to/workdir)にすることもできます。	
Clear output directory before execution	いいえ	ジョブの開始前に出力ディレクトリを削除する必要があるかどうかを示します。これをfalseに設定した場合、ジョブの実行前に出力ディレクトリが存在すると、出力ディレクトリがすでに存在することを示すエラーでジョブの開始に失敗します。	false (デフォルト) | true
Classes			
Job implementation API version	はい	MapReduceジョブの実装に使用されるAPIのバージョン。「**New API**」を選択した場合(デフォルト)、ジョブを実装するクラスはorg.apache.hadoop.mapreduceパッケージからクラスを拡張する必要があります。「**Old API**」を選択した場合、ジョブを実装するクラスはorg.apache.hadoop.mapredパッケージからクラス/インタフェースを拡張/実装します。	mapreduce (デフォルト) | mapred

<table>
<tr><th>属性</th><th>必須</th><th>説明</th><th>可能な値</th></tr>
<tr>
<td>Mapper class</td>
<td>いいえ</td>
<td>ジョブのマッパーとして使用されるJavaクラスの完全修飾名。クラスの定義は通常、ジョブJARファイル内にあります。
選択した「Job implementation API version」に応じて、クラスは次の表からクラス/インタフェースを拡張/実装する必要があります。
<table>
<tr><th colspan="2">拡張/実装</th></tr>
<tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.Mapper</td></tr>
<tr><td>古いAPI</td><td>org.apache.hadoop.mapred.Mapper</td></tr>
</table>
次の表に、このコンポーネント属性の設定に対応するジョブ構成パラメータおよびHadoop APIメソッドを示します。ExecuteMapReduceコンポーネントは常に、選択された「Job implementation API version」に従ってジョブ構成パラメータを直接設定します。(リストされたJavaメソッドが呼び出されることはありません。比較のためのみにリストしています。)
<table>
<tr><th colspan="3">ジョブ構成</th></tr>
<tr><td rowspan="2">パラメータ</td><td>新しいAPI</td><td>mapreduce.map.class</td></tr>
<tr><td>古いAPI</td><td>mapred.mapper.class</td></tr>
<tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td>Job.setMapperClass(Class)</td></tr>
<tr><td>古いAPI</td><td>JobConf.setMapperClass(Class)</td></tr>
</table>
</td>
<td>完全修飾クラス名
(例: com.acme.MyMap)
<table>
<tr><th colspan="2">デフォルト</th></tr>
<tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.Mapper</td></tr>
<tr><td>古いAPI</td><td>org.apache.hadoop.mapred.lib.IdentityReducer</td></tr>
</table>
</td>
</tr>
<tr>
<td>Combiner class</td>
<td>いいえ</td>
<td>ジョブのコンバイナとして使用されるJavaクラスの完全修飾名。クラスの定義は通常、ジョブJARファイル内にあります。
<table>
<tr><th colspan="2">拡張/実装</th></tr>
<tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.Reducer</td></tr>
<tr><td>古いAPI</td><td>org.apache.hadoop.mapred.Reducer</td></tr>
</table>
<table>
<tr><th colspan="3">ジョブ構成</th></tr>
<tr><td rowspan="2">パラメータ</td><td>新しいAPI</td><td>mapreduce.combine.class</td></tr>
<tr><td>古いAPI</td><td>mapred.combiner.class</td></tr>
<tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td>Job.setCombinerClass(Class)</td></tr>
<tr><td>古いAPI</td><td>JobConf.setCombinerClass(Class)</td></tr>
</table>
</td>
<td>完全修飾クラス名
(例: com.acme.MyReduce | No combiner (デフォルト))</td>
</tr>
</table>

<table>
<tr><th>属性</th><th>必須</th><th>説明</th><th>可能な値</th></tr>
<tr><td>Partitioner class</td><td>いいえ</td><td>ジョブのパーティショナとして使用されるJavaクラスの完全修飾名。クラスの定義は通常、ジョブJARファイル内にあります。
<table><tr><th colspan="2">拡張/実装</th></tr><tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.Partitioner</td></tr><tr><td>古いAPI</td><td>org.apache.hadoop.mapred.Partitioner</td></tr></table>
<table><tr><th colspan="3">ジョブ構成</th></tr><tr><td rowspan="2">パラメータ</td><td>新しいAPI</td><td>mapreduce.partitioner.class</td></tr><tr><td>古いAPI</td><td>mapred.partitioner.class</td></tr><tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td>Job.setPartitionerClass(Class)</td></tr><tr><td>古いAPI</td><td>JobConf.setPartitionerClass(Class)</td></tr></table></td><td>完全修飾クラス名
(例: com.acme.MyPartitioner)

<table><tr><th colspan="2">デフォルト</th></tr><tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.lib.partition.HashPartitioner</td></tr><tr><td>古いAPI</td><td>org.apache.hadoop.mapred.lib.HashPartitioner</td></tr></table></td></tr>
<tr><td>Reducer class</td><td>いいえ</td><td>ジョブのレデューサとして使用されるJavaクラスの完全修飾名。クラスの定義は通常、ジョブJARファイル内にあります。
<table><tr><th colspan="2">拡張/実装</th></tr><tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.Reducer</td></tr><tr><td>古いAPI</td><td>org.apache.hadoop.mapred.Reducer</td></tr></table>
<table><tr><th colspan="3">ジョブ構成</th></tr><tr><td rowspan="2">パラメータ</td><td>新しいAPI</td><td>mapreduce.reduce.class</td></tr><tr><td>古いAPI</td><td>mapred.reducer.class</td></tr><tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td>Job.setReducerClass(Class)</td></tr><tr><td>古いAPI</td><td>JobConf.setReducerClass(Class)</td></tr></table></td><td>完全修飾クラス名
(例: com.acme.MyReduce)

<table><tr><th colspan="2">デフォルト</th></tr><tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.Reducer</td></tr><tr><td>古いAPI</td><td>org.apache.hadoop.mapred.lib.IdentityReducer</td></tr></table></td></tr>
</table>

<table>
<tr><th>属性</th><th>必須</th><th>説明</th><th>可能な値</th></tr>
<tr><td>Mapper output key class</td><td>いいえ</td><td>インスタンスがマッパー出力レコードのキーであるJavaクラスの完全修飾名。マッパー出力キー・クラスが最終出力値クラスと異なる場合にのみ指定する必要があります。
<table>
<tr><th colspan="3">ジョブ構成</th></tr>
<tr><td colspan="2">パラメータ</td><td>mapred.mapoutput.key.class</td></tr>
<tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td>Job.setMapOutputKeyClass(Class)</td></tr>
<tr><td>古いAPI</td><td>JobConf.setMapOutputKeyClass(Class)</td></tr>
</table></td><td>完全修飾クラス名(例: org.apache.hadoop.io.Text) | デフォルトは「Output key class」属性の値</td></tr>
<tr><td>Mapper output value class</td><td>いいえ</td><td>インスタンスがマッパー出力レコードの値であるJavaクラスの完全修飾名。マッパー出力値クラスが最終出力値クラスと異なる場合にのみ指定する必要があります。
<table>
<tr><th colspan="3">ジョブ構成</th></tr>
<tr><td colspan="2">パラメータ</td><td>mapred.mapoutput.value.class</td></tr>
<tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td>Job.setMapOutputValueClass(Class)</td></tr>
<tr><td>古いAPI</td><td>JobConf.setMapOutputValueClass(Class)</td></tr>
</table></td><td>完全修飾クラス名(例: org.apache.hadoop.io.Text) | デフォルトは「Output value class」属性の値</td></tr>
<tr><td>Grouping comparator</td><td>いいえ</td><td>レデューサのreduceメソッドへの1つの呼出しに対してどのキーをグループ化するかを決定するJavaクラス実装コンパレータの完全修飾名。クラスはorg.apache.hadoop.io.RawComparatorインタフェースを実装する必要があります(またはベース実装のorg.apache.hadoop.io.WritableComparatorを拡張する必要があります)。
<table>
<tr><th colspan="3">ジョブ構成</th></tr>
<tr><td colspan="2">パラメータ</td><td>mapred.output.value.groupfn.class</td></tr>
<tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td>Job.setGroupingComparatorClass(Class)</td></tr>
<tr><td>古いAPI</td><td>JobConf.setOutputValueGroupingComparator(Class)</td></tr>
</table></td><td>完全修飾クラス名(例: com.acme.MyGroupingComp) | デフォルト・クラスは次の手順で導出されます。1) 「Sorting comparator」属性のクラス名値が指定されている場合、その値を取得します。2) そうでない場合、org.apache.hadoop.io.WritableComparableの実装が「Mapper output key class」のコンパレータとして登録されていれば、その実装を取得します。3) そうでない場合、汎用実装(org.apache.hadoop.io.WritableComparator)を取得します。</td></tr>
</table>

<table>
<tr><th>属性</th><th>必須</th><th>説明</th><th>可能な値</th></tr>
<tr><td>Sorting comparator</td><td>いいえ</td><td>キーをレデューサに渡す前にソートする方法を制御するJavaクラス実装コンパレータの完全修飾名。クラスは<code>org.apache.hadoop.io.RawComparator</code>インタフェースを実装する必要があります(またはベース実装の<code>org.apache.hadoop.io.WritableComparator</code>を拡張する必要があります)。
<table>
<tr><td colspan="3">ジョブ構成</td></tr>
<tr><td colspan="2">パラメータ</td><td><code>mapred.output.key.comparator.class</code></td></tr>
<tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td><code>Job.setSortComparatorClass(Class)</code></td></tr>
<tr><td>古いAPI</td><td><code>JobConf.setOutputKeyComparatorClass(Class)</code></td></tr>
</table></td><td>完全修飾クラス名(例: <code>com.acme.MySorter</code>)｜デフォルト・クラスは、<code>org.apache.hadoop.io.WritableComparable</code>の実装が「Mapper output key class」のコンパレータとして登録されていれば、その実装になります。そうでない場合は、汎用実装<code>org.apache.hadoop.io.WritableComparator</code>が使用されます。</td></tr>
<tr><td>Output key class</td><td>はい</td><td>インスタンスがジョブの出力レコード(つまり、レデューサの出力)のキーであるJavaクラスの完全修飾名。
<table>
<tr><td colspan="3">ジョブ構成</td></tr>
<tr><td colspan="2">パラメータ</td><td><code>mapred.output.key.class</code></td></tr>
<tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td><code>Job.setOutputKeyClass(Class)</code></td></tr>
<tr><td>古いAPI</td><td><code>JobConf.setOutputKeyClass(Class)</code></td></tr>
</table></td><td>完全修飾クラス名(例: <code>org.apache.hadoop.io.Text</code>)｜<code>org.apache.hadoop.io.LongWritable</code> (デフォルト)</td></tr>
<tr><td>Output value class</td><td>はい</td><td>インスタンスがジョブの出力レコード(つまり、レデューサの出力)の値であるJavaクラスの完全修飾名。
<table>
<tr><td colspan="3">ジョブ構成</td></tr>
<tr><td colspan="2">パラメータ</td><td><code>mapred.output.value.class</code></td></tr>
<tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td><code>Job.setOutputValueClass(Class)</code></td></tr>
<tr><td>古いAPI</td><td><code>JobConf.setOutputValueClass(Class)</code></td></tr>
</table></td><td>完全修飾クラス名(例: <code>org.apache.hadoop.io.IntWritable</code>)｜<code>org.apache.hadoop.io.Text</code> (デフォルト)</td></tr>
</table>

<table>
<tr><th>属性</th><th>必須</th><th>説明</th><th>可能な値</th></tr>
<tr><td>Input format</td><td>いいえ</td><td>ジョブの入力書式として使用されるJavaクラスの完全修飾名。このクラスは、入力ファイルの解析を実装し、マッパーの入力となるキーと値のペアを生成します。
<table><tr><th colspan="2">拡張/実装</th></tr><tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.InputFormat</td></tr><tr><td>古いAPI</td><td>org.apache.hadoop.mapred.InputFormat</td></tr></table>
<table><tr><th colspan="3">ジョブ構成</th></tr><tr><td rowspan="2">パラメータ</td><td>新しいAPI</td><td>mapreduce.inputformat.class</td></tr><tr><td>古いAPI</td><td>mapred.input.format.class</td></tr><tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td>Job.setInputFormatClass(Class)</td></tr><tr><td>古いAPI</td><td>JobConf.setInputFormat(Class)</td></tr></table></td><td>完全修飾クラス名
<table><tr><th colspan="2">デフォルト</th></tr><tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.lib.input.TextInputFormat</td></tr><tr><td>古いAPI</td><td>org.apache.hadoop.mapred.TextInputFormat</td></tr></table></td></tr>
<tr><td>Output format</td><td>いいえ</td><td>ジョブの出力書式として使用されるJavaクラスの完全修飾名。このクラス実装は、レデューサにより生成されたキーと値のペアを取得して出力ファイルに書き込みます。
<table><tr><th colspan="2">拡張/実装</th></tr><tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.OutputFormat</td></tr><tr><td>古いAPI</td><td>org.apache.hadoop.mapred.OutputFormat</td></tr></table>
<table><tr><th colspan="3">ジョブ構成</th></tr><tr><td rowspan="2">パラメータ</td><td>新しいAPI</td><td>mapreduce.outputformat.class</td></tr><tr><td>古いAPI</td><td>mapred.output.format.class</td></tr><tr><td rowspan="2">メソッド</td><td>新しいAPI</td><td>Job.setOutputFormatClass(Class)</td></tr><tr><td>古いAPI</td><td>JobConf.setOutputFormat(Class)</td></tr></table></td><td>完全修飾クラス名(例: org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat)。この場合、HadoopReader (p.375)コンポーネントを使用して出力ファイルを読み取ることができます。
<table><tr><th colspan="2">デフォルト</th></tr><tr><td>新しいAPI</td><td>org.apache.hadoop.mapreduce.lib.input.TextOutputFormat</td></tr><tr><td>古いAPI</td><td>org.apache.hadoop.mapred.TextOutputFormat</td></tr></table></td></tr>
<tr><td colspan="4">Advanced</td></tr>
<tr><td>Number of mappers</td><td>いいえ</td><td>ジョブを実行するためにHadoopで実行する必要のあるマッパー・タスクの数。これは単なるヒントです。生成されるマップ・タスクの実際の数は、入力書式クラス実装によって異なります。</td><td>ゼロより大きい整数</td></tr>
</table>

属性	必須	説明	可能な値
Number of reducers	いいえ	ジョブを実行するためにHadoopにより実行される必要なレデューサ・タスクの数。レデューサの数としてゼロを指定することもできます。この場合、レデューサは実行されず、マッパーの出力は**出力ディレクトリ**に直接送信されます。	ゼロ以上の整数
Execute job as daemon	いいえ	デフォルトでは、これは`false`であり、**ExecuteMapReduce**コンポーネントはMapReduceジョブを同期的に実行します。つまり、ジョブを開始して、ジョブが終了するまで待ってから、次の入力トークンにより定義された別のジョブを開始します(実行するジョブがこれ以上ない場合は終了します)。 `true`に設定すると、ジョブは非同期的に実行されます。つまり、コンポーネントはジョブを開始し、待機することなく、次の入力トークンにより定義された別のジョブを開始します(実行するジョブがこれ以上ない場合は終了します)。また、この場合、ジョブ実行は監視されません(ジョブ実行ステータスはグラフ実行ログに出力されません)。	`false` (デフォルト) | `true`
Stop processing on fail	いいえ	デフォルトでは、失敗したMapReduceジョブがあると、コンポーネントは他のジョブの実行を停止し、スキップされたトークンに関する情報がエラー出力ポートに送信されます。この属性で、この動作をオフにできます。	`true` (デフォルト) | `false`
Additional job settings	いいえ	設定する必要のあるジョブの他のプロパティは、ここでキーと値のペアとして指定できます。キーはプロパティのHadoop固有名であり(使用されているHadoopバージョンで有効な名前にする必要があります)、値は名前付きプロパティの新しい値です。コンポーネント属性値の優先度は、ここで指定した対応するプロパティの値よりも高くなります。 このフィールドの値は、Javaプロパティ・ファイルの形式にする必要があります。 実行対象のジョブごとに、「JobTracker HTML status」ページにすべてのジョブ設定の概要(job.xmlファイル)が表示されます(デフォルトではポート50030で実行)。	

注意

前述のコンポーネントの属性はすべて、入力トークンのデータを使用して構成することもできます。「**Input mapping**」のCTL変換は、入力トークンのデータ・フィールドからMapReduceジョブ実行構成へのマッピングを定義します。

ヒント

ExecutemapReduceコンポーネントがジョブ構成を作成すると、各パラメータの設定に関する情報が**DEBUG**ログ・レベルでグラフ実行ログに出力されます。さらに、完全な最終ジョブ構成XMLが**TRACE**ログ・レベルで出力されます。

出力およびエラー・マッピング

どちらのマッピングも、通常のCTL変換です。出力マッピングは、最初の出力ポートに渡されたトークンを移入するために使用されます。このマッピングは、成功したMapReduceジョブに対して実行されます。エラー・マッピングは、ジョブが失敗で終了し、最初の出力ポートではなく2番目の出力ポートが移入された場合にのみ使用されます。

どちらのマッピングでも、入力データ・レコードは同じです。2つまたは3つのレコードが使用可能です。

- ジョブ実行をトリガーした入力トークン(入力コネクタがないとコンポーネント使用には使用できません)。これは、入力トークンのいくつかのフィールドを出力トークンに渡す必要がある場合に便利です。このレコードには、「**Type**」列に表示される**ポート0**があります。

- JobResultsレコードは、ジョブ実行に関する情報を提供します。

フィールド名	型	説明
jobID	string	JobTrackerによってジョブに提供された一意の識別情報。ジョブがJobTrackerに接続しようとして開始前に失敗した場合、この値は設定されないことがあります。
startTime	date	ジョブの開始日時。これはCloverETLによりローカルに測定されるもので、JobTrackerにより測定されたジョブ開始時刻とはわずかに異なることがあります。常に設定されます。
endTime	date	ジョブの終了日時。これはCloverETLによりローカルに測定されるもので、JobTrackerにより測定されたジョブ終了時刻とはわずかに異なることがあります。常に設定されます。
duration	long	ジョブの期間(ミリ秒単位)。これは、**endTime**と**startTime**の差分(ミリ秒単位)です。ジョブのタイムアウト値(設定されている場合)を超えることはできません。この値は常に設定されます。
state	string	ジョブの実行終了時の状態。 有効なフィールド値は次のとおりです。 • **SUCCEEDED**: ジョブ正常に実行された場合 • **FAILED**: ジョブの実行が失敗した場合 • **TIMEOUT**: ジョブ実行時間が指定されたタイムアウトを超えたためにジョブが強制終了された場合
clusterErrMessage	string	JobTrackerから取得されたエラー・メッセージ文字列。
errException	string	JobTrackerで発生した、またはJobTrackerとの通信中に発生した例外の完全スタック・トレースのテキスト表現。例外が発生していない場合、この値は設定されません。
lastMapReduce Phase	string	ジョブの終了時に進行していた最後のMapReduceジョブ・フェーズ。値は、文字列Setup、Map、ReduceまたはCleanupです。**wasJobSuccessful**が`true`の場合、値はCleanupになります。ジョブが失敗した場合、JobTrackerとの通信に長い遅延があると、この値が正確でなくなる可能性があります。実際の値は常に、JobTracker管理サイトを使用して取得できます。この値は常に設定されます。
lastMapReduce PhaseProgress	number	ジョブの終了時に実行されていた最後のMapReduceフェーズの進行状況。値は、0から1までの(それらを含む)間隔内の浮動小数点数です。**wasJobSuccessful**が`true`の場合、値は1になります。ジョブが失敗した場合、特にJobTrackerとの通信に大幅な遅延があると、値が正確でなくなる可能性があります。常に設定されます。

- ジョブにより処理されたカウンタの値。

フィールド名	型	説明
allCounters	map[string,long]	ジョブに使用可能なすべてのカウンタの名前と値のペアを含むマップ。
*	long	その他のフィールドはすべて、各ジョブについてHadoopにより自動的に収集される事前定義済(デフォルト)カウンタの名前です。カウンタのリストは、使用されているHadoopバージョンによって異なる場合があります。

ExecuteProfilerJob

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

ヒント

`*.cpj`ファイルをドラッグして「jobflow」ペインにドロップすると、**ExecuteProfilerJob**コンポーネントが追加されます。

要約

ExecuteProfilerJobを使用すると、ユーザー指定の設定でプロファイラ・ジョブを実行でき、実行結果が出力ポートに送信されます。

注意

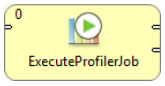
このコンポーネントを使用するには、プロファイラおよびジョブフローのライセンスが必要です。
また、プロジェクトがClover Serverで実行されている必要もあります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
ExecuteProfilerJob	いいえ	いいえ	0-1	0-2	はい	いいえ	はい

概要

このコンポーネントは**ExecuteGraph**と同様に動作します。ExecuteGraph(p.690)コンポーネントの説明を参照してください。この2つのコンポーネントの主な違いのリストは、ExecuteProfilerJobの属性(p.710)を参照してください。

アイコン



ポート

ExecuteGraphのポート(p.691)を参照してください。

ExecuteProfilerJobの属性

属性の詳細は、ExecuteGraphの属性(p.691)を参照してください。**ExecuteGraph**とは異なり、**ExecuteProfilerJob**コンポーネントには「**Execution group**」属性および「**Skip checkConfig**」属性がありません。

また、属性Input mapping(p.710)およびOutput mapping(p.711)で、これとは少し異なるプロファイラ・ジョブ固有の構成について説明されています。

入力マッピング

「Input mapping」属性を使用すると、受信トークンのデータに基づいてコンポーネントの設定をオーバーライドできます。さらに、入力マッピングでは実行対象のプロファイラ・ジョブのジョブ・パラメータを変更できます。

入力マッピングは、各プロファイラ・ジョブ実行の前に実行される通常のCTL変換です。このマッピングの入力は、入力トークン(ある場合)のみです。変換の出力は、2つのレコード(RunConfおよびParameters)に基づきます。

- **RunConf**レコードは実行設定を表します。このマッピングによってレコードのフィールドが移入されない場合、コンポーネントの該当する属性のデフォルト値がかわりに使用されます。

フィールド名	型	説明
jobURL	string	コンポーネント属性「**Profiler Job URL**」をオーバーライドします。
source	string	プロファイル対象のデータ・ソースをオーバーライドします。ファイルまたはXLSスプレッドシートをプロファイルする場合、プロファイルされるファイルが変更されます。DB表ジョブの場合、データの取得元の表がオーバーライドされます。
charset	string	ファイルおよびXLSプロファイラ・ジョブのプロファイル対象データの入力エンコーディング(charset)をオーバーライドします。
executionType	string	コンポーネント属性「**Execution type**」をオーバーライドします。
timeout	long	コンポーネント属性「**Timeout**」をオーバーライドします。
clusterNodeId	string	コンポーネント属性「**Preferred cluster node ID**」をオーバーライドします。
daemon	boolean	コンポーネント属性「**Execute profiler job as daemon**」をオーバーライドします。

- **Parameters**レコードは、トリガーされるプロファイラ・ジョブのすべての外部プロファイラ・ジョブ・パラメータを表します。

注意

変換ダイアログでParametersレコードが使用可能になるのは、コンポーネント属性「**Profiler Job URL**」が、パラメータ構造の抽出用のテンプレートとして使用される既存のプロファイラ・ジョブに関連付けられている場合のみです。実行時に実際にどのプロファイラ・ジョブが実行されるかに関係なく、このプロファイラ・ジョブのパラメータのみを入力マッピングにより移入できます。

出力マッピング

出力マッピングは、最初の出力ポートに渡されたトークンを移入するために使用される標準的なCTL変換です。このマッピングは、成功したプロファイラ・ジョブ実行に対して実行されます。このマッピングには最大4つの入力データ・レコードを使用できます。

- プロファイラ・ジョブの実行の基礎となった入力**RunConf**トークン(入力コネクタがないとコンポーネント使用には使用できません)。これは、入力トークンのいくつかのフィールドを出力トークンに渡す場合に非常に便利です。このレコードには、「**Type**」列に表示される**ポート0**があります。
- **RunStatus**レコードは、プロファイラ・ジョブ実行に関する情報を提供します。

フィールド名	型	説明
runId	long	プロファイラ・ジョブ実行の一意の識別子。
orignalJobURL	string	実行対象のプロファイラ・ジョブのパス。
startTime	date	ジョブの実行時刻。
endTime	date	ジョブの終了時刻(非同期実行の場合はnull)。
duration	long	ミリ秒単位のジョブ実行時間。
status	string	最終ジョブ実行ステータス(非同期実行の場合はFINISHED_OK | ERROR | ABORTED | TIMEOUT | RUNNING)。
errException	string	失敗したジョブに対してのみ例外を発生させます。
errMessage	string	失敗したジョブのみのエラー・メッセージ。

- **RunInfo**は、プロファイラ・ジョブに固有のジョブ実行に関する追加情報を提供します。

フィールド名	型	説明
inputRecordCount	long	プロファイルされたレコード数
rejectedRecordCount	long	解析エラーなどによりプロファイルを拒否されたレコード数

- **RunResults**レコードは、プロファイリング結果(プロファイルされるフィールドで有効化されているメトリックの出力値)を提供します。プロファイリングの結果が使用可能になるのは、同期実行で、現在のユーザーがプロファイリング結果を読み取るためのサンドボックス権限を持っている場合のみです。

  構造化された結果を含むメトリックは、戻り値を複数値フィールド(p.168)として返します。これには、チャートや書式カウント・メトリックなどが含まれます。

ExecuteScript

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

要約

ExecuteScriptは、シェル・スクリプト、または選択したインタプリタにより解釈されたスクリプトを実行するコンポーネントです。スクリプトが1回のみ実行されるか、受信レコードごとにスクリプトが繰り返し実行されます。各受信レコードは、スクリプト自体を含むほぼすべての実行パラメータを再定義できます。

概要

ExecuteScriptは、特定のインタプリタ(デフォルトではデフォルトのシステム・シェル)を使用してスクリプトを実行します。

入力ポートに接続されているエッジがない場合、コンポーネントはスクリプトを1回のみ実行します。この場合、1つの出力レコードが生成されます。

入力ポートに送信されたレコードがある場合、各レコードにつき1回スクリプト実行が行われ、各スクリプト実行につき1つの出力レコードが生成されます。

スクリプトが成功した場合、コンポーネントは引き続き次の入力トークンを処理します。それ以外の場合、コンポーネントは他のスクリプトの実行を停止します。それ以降はすべての受信トークンが無視され、無視されたトークンの情報がエラー出力ポートに送信されます。この動作は、「**Stop processing on fail**」パラメータで変更できます。

出力レコードには、スクリプト実行に関するすべての重要な情報(時間、終了値、エラー・レポートおよび標準出力)が含まれます。ユーザー定義出力メタデータへのこれらの値のマッピングは、「**Output Mapping**」属性および「**Error Mapping**」属性で定義できます。

すべてのスクリプト実行パラメータは、「**Input Mapping**」属性を使用して入力レコードを介して設定できます。マッピングにより、入力のどの値をスクリプト実行パラメータとして使用するかを設定します。入力マッピングと出力マッピングは、ジョブ制御(p.684)コンポーネントに共通です。

1回のスクリプト実行は、次のようにして実行されます。

- スクリプト・コードが一時バッチ・ファイルにコピーされます。
- 文字列${}が一時ファイル名に置き換えられて、インタプリタが実行されます。
- スクリプトが終了すると、出力レコードが生成されます。実行が成功した場合は最初の出力ポートに送信され、実行が失敗した場合は2番目の出力ポートに送信されます。

詳細は、属性の説明(p.713)を参照してください。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	スクリプト実行のパラメータ(必要に応じてスクリプト自体を含める)。	任意
出力	0	いいえ	コンポーネント入力およびスクリプト実行結果。	任意
エラー	1	いいえ	コンポーネント入力およびスクリプト実行結果。実行できないスクリプトまたは1を返すスクリプト用のレコード。	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

ExecuteScriptの属性

属性	必須	説明	可能な値
Basic			
Script	いいえ	実行するスクリプトのコード。インタプリタ属性値をデフォルトのままにする場合、スクリプトはシェル・スクリプトのコードである必要があります。そのため、1つ以上のシステム・コマンドを実行するために簡単に使用できます。それ以外の場合、コード形式は選択したインタプリタによって異なります。	
Script URL	いいえ	実行するスクリプトのURL。インタプリタ属性値をデフォルトのままにする場合、スクリプトはシェル・スクリプトのコードである必要があります。そのため、1つ以上のシステム・コマンドを実行するために簡単に使用できます。それ以外の場合、コード形式は選択したインタプリタによって異なります。スクリプト属性とスクリプトURL属性の両方を指定した場合、スクリプトURLのみが使用されます。	
Script charset	いいえ	この文字エンコーディングが実行対象のスクリプト・ファイルに使用されます。スクリプト・ファイル参照は「Script URL」属性で指定します。指定しない場合、「Script」属性から一時バッチ・ファイルが自動的に作成されます。	ISO-8859-1 (デフォルト) | <other encodings>
Working directory	いいえ	実行対象スクリプトの作業ディレクトリ。スクリプト内部で使用されているすべての相対パスは、このディレクトリを基準にして解釈されます。デフォルトでは、グラフを含むCloverETLプロジェクトのルートに設定されます。	
Timeout	いいえ	スクリプト実行の制限(ミリ秒単位)。スクリプトの実行時間がこの制限を超えると、コンポーネントはスクリプトを強制終了します。この場合、出力レコードのフィールドは次のように設定されます。exitValueは1、reachedTimeoutはtrue、期間はタイムアウト以上に設定されます。	0 (無制限) | 正数
Input Mapping	いいえ	「Input mapping」は、受信トークンのデータでデフォルトのコンポーネント設定をオーバーライドする方法を定義します。入力マッピング・フィールドの説明(p.715)を参照してください。	CTL変換

属性	必須	説明	可能な値
Output Mapping	いいえ	「Output mapping」は、成功したスクリプト実行の結果を最初の出力ポートにマップします。出力マッピング・フィールドの説明(p.716)を参照してください。	CTL変換
Error Mapping	いいえ	「Error mapping」は、失敗したスクリプトの結果を2番目の出力ポートにマップします。出力マッピング・フィールドの説明(p.716)を参照してください。	CTL変換
Redirect Error Output	いいえ	デフォルトでは、失敗したスクリプトの結果は2番目の出力ポート(エラー・ポート)に送信されます。このスイッチをtrueにした場合、失敗したスクリプトの結果が成功したスクリプトと同様に最初の出力ポートに送信されます。	false (デフォルト) | true
Interpreter	いいえ	スクリプトの実行に使用するインタプリタを設定します。インタプリタが実行されると、`${}`がスクリプトのコピーを含む一時バッチ・ファイル名で置き換えられます。インタプリタでスクリプト・ファイルの拡張子が重要となる場合は、一時ファイルが正しい拡張子を持つように**「Batch file extension」**プロパティを設定する必要があります。	`${}`に続くインタプリタのパス。デフォルトでは、スクリプトはシステム・シェル(Windowsの場合は`cmd`、Linuxの場合は`sh`など)によって解釈されます。
Environment variables	いいえ	スクリプト内の環境変数の値を設定します。これを使用すると、値の設定または追加が可能です。存在しない変数に追加すると、変数が定義されてその値が設定されます。変数の値はスクリプト内部でのみ参照できます。つまり、インタプリタのパスを設定するために`PATH`の設定を使用することはできません。このプロパティで設定された変数の値は、入力マッピング・ダイアログで入力を**EnvironmentVariables**メタデータにマップすることによりオーバーライドできます。	
Standard input	いいえ	スクリプトに送信される標準入力のコンテンツ。スクリプトで使用可能な行より多くの入力行が予期される場合、スクリプトがハングする可能性があることに注意してください。	string
Standard input file URL	いいえ	スクリプトに送信される標準入力のコンテンツのファイルURL。スクリプトで使用可能な行より多くの入力行が予期される場合、スクリプトがハングする可能性があることに注意してください。	
Standard output file URL	いいえ	スクリプトの標準出力を保存するファイルのファイルURL。**追加**フラグに応じて、ファイル・コンテンツは再書込みまたは追加されます。	
Standard error file URL	いいえ	スクリプトのエラー出力を保存するファイルのファイルURL。**追加**フラグに応じて、ファイル・コンテンツは再書込みまたは追加されます。	
Append	いいえ	ファイルに書き込まれた標準出力とエラー出力(**「Standard output file URL」**属性と**「Error output file URL」**属性)によって既存のコンテンツを再書込みするか、既存のコンテンツにそれを追加するかを設定します。	false (デフォルト) | true
Data charset	いいえ	入力ポートから渡された標準入力をエンコードして、出力ポートに渡す標準出力とエラー出力をデコードするために使用する文字エンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>

属性	必須	説明	可能な値
Batch file extension	いいえ	インタプリタに渡されるバッチ・ファイルの拡張子を設定します(インタプリタ設定で`${}`がその名前に置き換わります)。	`bat` (デフォルト) | string
Stop processing on fail	いいえ	デフォルトでは、失敗したスクリプトがあると、コンポーネントは他のスクリプトの実行を停止し、スキップされたトークンに関する情報がエラー出力ポートに送信されます。この属性で、この動作をオフにできます。	true (デフォルト) | false

注意

`script`属性のコンテンツは一時バッチ・ファイルにコピーされます。Microsoft Windowsでは、実行されたコマンドのエコーを無効にするために`@echo off`を使用してスクリプトを起動すると有用な場合があります。

入力マッピング・フィールドの説明

入力レコードは2つの異なるメタデータ(**RunConfig**および**EnvironmentVariables**)にマップできます。**RunConfig**のフィールドの機能は次のとおりです。

フィールド	説明
script	コンポーネント属性「**Script**」をオーバーライドします。
scriptURL	コンポーネント属性「**Script URL**」をオーバーライドします。
scriptCharset	コンポーネント属性「**Script charset**」をオーバーライドします。
interpreter	コンポーネント属性「**Interpreter**」をオーバーライドします。
workingDirectory	コンポーネント属性「**Working Directory**」をオーバーライドします。
timeout	コンポーネント属性「**Timeout**」をオーバーライドします。
environmentVariables	コンポーネント属性「**Environment Variables**」をオーバーライドします。値には、";"で区切られた変数割当てのリストが含まれる必要があります。単なる"="記号を使用して割り当てると、値は割当て済の環境変数に割り当てられます。"+="記号を使用して割り当てると、値は割当て済の環境変数に追加されます。
stdIn	コンポーネント属性「**Standard Input**」をオーバーライドします。
stdInFileURL	コンポーネント属性「**Standard Input File URL**」をオーバーライドします。
stdOutFileURL	コンポーネント属性「**Standard Output File URL**」をオーバーライドします。
errOutFileURL	コンポーネント属性「**Error Output File URL**」をオーバーライドします。
append	コンポーネント属性「**Append**」をオーバーライドします。
dataCharset	コンポーネント属性「**Data charset**」をオーバーライドします。
batchFileExtension	コンポーネント属性「**Batch File Extension**」をオーバーライドします。

注意

「**Input mapping**」では、`$out.0.script`フィールドを使用して動的コマンドラインを作成できます。スクリプトおよびそのパラメータをフィールドにマップするのみです。例:

```
$out.0.script = "md5.exe " + $in.0.filePath;
```

注意

実行対象のスクリプトに提供する環境変数は、3つの異なる方法で定義できます。

1. 環境変数を静的に定義する場合は、コンポーネントの属性「**Environmental variables**」を使用します。変数名および値は、すべてのスクリプト実行に対して、1回定義します。

2. 環境変数の2番目の定義方法は、「**Input mapping**」で出力レコード **EnvironmentVariables**を移入することです。この方法でもレコード構造によって変数名のセットが静的に定義されますが、変数の値は入力トークンから導出できます。

3. 環境変数の定義方法として最も複雑な方法は、「**Input mapping**」で出力レコード **RunConfig**の**environmentVariables**フィールドを移入することです。このフィールドの値の構文と意味は、コンポーネント属性「**Environment variables**」と同様です。この場合、変数のセットと変数値はどちらも完全に動的に定義できます。

注意

「**Input mapping**」で文字列を環境変数に追加する場合は、**getEnvironmentVariables()** CTL関数を使用してください。例:

```
$out.1.PATH = getEnvironmentVariables()["PATH"] + ":" +
$in.0.additionalPath;
```

出力マッピング・フィールドの説明

フィールド	説明
stdOut	スクリプトの標準出力。
errOut	スクリプトのエラー出力。
startTime	スクリプトの開始時刻。
stopTime	スクリプトの終了時刻。
duration	スクリプトの期間。(duration = stopTime - startTime)
exitValue	スクリプトによって返された値。通常、0はエラーがないことを示し、ゼロ以外の値はエラーがあることを示します。
reachedTimeout	スクリプトがタイムアウトに達したかどうかを判断するブール値。
errException	スクリプトの呼出しがエラーで終了した場合、スクリプトにエラーの原因となった例外が含まれている可能性があります。これは、スクリプトが起動されなかった(スクリプトのパスが無効である)場合か、その実行が中断された(タイムアウトに達した)場合にのみ発生します。
errMessage	errExceptionの例外によってレポートされたメッセージ。

Fail

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

要約

Failコンポーネントは、ユーザー指定のエラー・メッセージを使用して親ジョブ(ジョブフローまたはグラフ)を中断します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
Fail	いいえ	いいえ	0-1	0	いいえ	いいえ	はい

概要

Failは、親ジョブ(ジョブフローまたはグラフ)を中断します。最初に、コンポーネントが受信したトークンによって、ユーザー定義のエラー・メッセージで例外(org.jetel.exception.UserAbortException)がスローされます。このジョブは、最終ステータスERRORで即時に終了します。さらに、ジョブの中断前にディクショナリ・コンテンツを変更できます。通常、このコンポーネントを使用すると、ジョブを中断すると同時に、ディクショナリを介して結果を返すことができます。

コンポーネントFailは、入力ポートが接続されていない場合でも機能します。この場合、Failコンポーネントのあるフェーズが開始されるとすぐにジョブが中断されます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	このポートからの入力トークンはジョブフロー(またはグラフ)を中断します。	任意1)

説明:

1): ユーザー固有のエラー・メッセージには'errorMessage'という入力フィールドが自動的に使用されます。マッピングで他の指定がない場合、これによりジョブが中断されます。

Failの属性

属性	必須	説明	可能な値
Basic			
Error message	いいえ	ジョブが中断された場合、このエラー・メッセージとともに例外がスローされます。エラー・メッセージはマッピングで動的に変更できます。	"user abort" (デフォルト) | テキスト
Mapping	いいえ	ジョブが中断される場合にスローされるエラー・メッセージを動的にアセンブルするには、マッピングを使用します。さらに、中断されるジョブのディクショナリ・コンテンツも同様に変更できます。詳細説明(p.718)を参照してください。	

詳細説明

- **マッピングの詳細**

  Failコンポーネントでのマッピングは、通常、次の2つの目的に使用されます。

  - 受信レコードからのエラー・メッセージのアセンブル
  - 受信レコードからのディクショナリ・コンテンツの移入。

注意

変更できるのは出力ディクショナリ・エントリのみです。

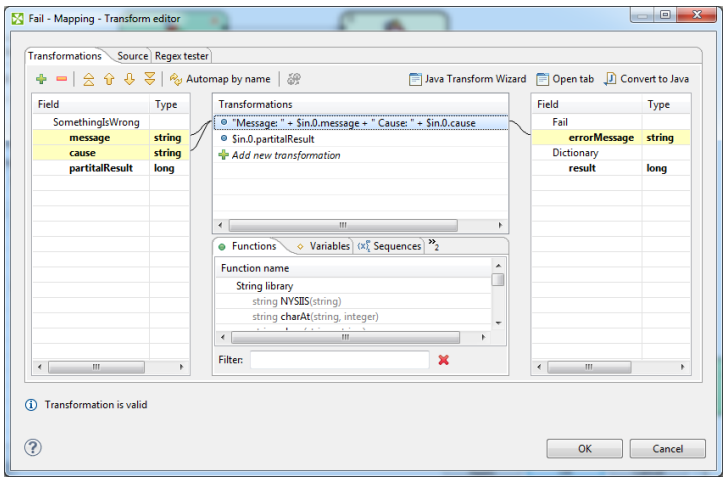


図57.2. Failコンポーネントのマッピングの例

マッピングによってコンパイルされたエラー・メッセージが最も優先されます。マッピングにより'errorMessage'が設定されていない場合、コンポーネント属性からのエラー・メッセージがかわりに使用されます。この属性も設定されていない場合は、事前定義済のテキスト"user abort"がかわりに使用されます。

GetJobInput

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

要約

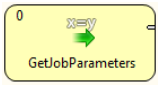
GetJobInputは、ディクショナリ・コンテンツまたはグラフ・パラメータ(あるいはその両方)により移入される1つのレコードを生成します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
GetJobInput	-	-	0	1	-	いいえ	はい

概要

コンポーネントGetJobInputは、リクエストされたジョブ・パラメータを取得して出力ポートに送信します。このコンポーネントは、マッピングにより移入される1つの出力レコードを生成します。マッピングが指定されていない場合は、自動マッピングが適用されます。たとえば、入力ディクショナリ・エントリが、同じ名前の出力フィールドに自動的にコピーされます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
出力	0	はい	ジョブ入力のあるレコード用	任意1)

説明:

1): 入力ディクショナリ・エントリを含む同じ名前のフィールドが自動的に移入されます。

GetJobInputの属性

属性	必須	説明	可能な値
Basic			
Mapping	いいえ	マッピングにより、コンポーネントの出力レコードが移入されます。マッピングの入力値は、入力ディクショナリ・エントリおよびグラフ・パラメータです。実際に、マッピング属性は、入力ディクショナリ・エントリを表すレコードから出力メタデータ構造を持つレコードへの通常のCTL変換です。マッピングは1回のみ呼び出されます。	

KillGraph

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

要約

KillGraphは、指定されたグラフを中断し、その最終ステータスを出力ポートに渡します。

注意

このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。また、プロジェクトがClover Serverで実行されている必要もあります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
KillGraph	いいえ	いいえ	0-1	0-1	はい	いいえ	はい

概要

KillGraphコンポーネントは、実行IDまたは実行グループにより指定されたグラフを中断します(実行グループに属するすべてのグラフが中断されます)。中断されたグラフの最終実行ステータスは、出力ポートに渡されるか、単にログに出力されます。さらに、中断されたグラフのデーモンの子も中断するかどうかを選択できます(デーモンでない子はいずれの場合も中断されます)。ExecuteGraph(p.690)の**「Execute graph as daemon」**属性を参照してください。

コンポーネントは入力トークンを読み取って、受信データから実行IDまたは実行グループを抽出し(**「Input mapping」**属性を参照してください)、リクエストされたグラフを中断し、中断されたグラフの最終ステータスを出力ポートに書き込みます(**「Output mapping」**属性を参照してください)。

入力ポートが接続されていない場合、「Run ID」属性または「Execution group」属性で指定されたグラフのみが中断されます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	中断されるグラフの識別情報を含む入力トークン	任意
出力	0	いいえ	最終グラフ実行ステータス	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

KillGraphの属性

属性	必須	説明	可能な値
Basic			
Run ID	いいえ	中断されるグラフの実行IDを指定します。「Execution group」属性よりも優先度は高くなります。この属性は入力マッピングでオーバーライドできます。	long
Execution group	いいえ	指定した実行グループに属するすべてのグラフが中断されます。「Run ID」属性の方が優先されます。この属性は入力マッピングでオーバーライドできます。	string
Kill daemon children	いいえ	デーモンの子も中断するかどうかを指定します。デーモン以外の子はいずれの場合も中断されます。この属性は入力マッピングでオーバーライドできます。	false (デフォルト) | true
Input mapping	いいえ	「Input mapping」は、中断する実行IDまたは実行グループを受信トークンから抽出する方法を定義します。入力マッピング(p.722)を参照してください。	CTL変換
Output mapping	いいえ	「Output mapping」は、中断されたグラフの最終グラフ・ステータスにより出力トークンを移入する方法を定義します。出力マッピング(p.722)を参照してください。	CTL変換

入力マッピング

入力マッピングは、中断する実行IDまたは実行グループを抽出するために入力トークンごとに実行される通常のCTL変換です。出力レコードの構造は次のとおりです。

フィールド名	型	説明
runId	long	コンポーネント属性「**Run ID**」をオーバーライドします。
executionGroup	string	コンポーネント属性「**Execution group**」をオーバーライドします。
killDaemonChildren	boolean	コンポーネント属性「**Kill daemon children**」をオーバーライドします。

出力マッピング

出力マッピングは、出力トークンを移入するために中断されたグラフに対して実行される通常のCTL変換です。使用可能な入力データの構造は次のとおりです。

フィールド名	型	説明
runId	long	中断されるグラフの実行ID。
originalJobURL	string	中断されるグラフのパス。

フィールド名	型	説明
version	string	中断されるグラフのバージョン。
startTime	date	グラフの実行時刻。
endTime	date	グラフの終了時刻。
duration	long	ミリ秒単位のグラフ実行時間(endTime - startTime)。
status	string	最終グラフ実行ステータス(FINISHED_OK | ERROR | ABORTED | TIMEOUT)。
errException	string	失敗したグラフに対して例外を発生させます。
errMessage	string	失敗したグラフのエラー・メッセージ。
errComponent	string	グラフの失敗の原因となったコンポーネントID。
errComponentType	string	グラフの失敗の原因となったコンポーネントのタイプ。

KillJobflow

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

要約

KillJobflowは、指定されたジョブフローを中断し、その最終ステータスを出力ポートに渡します。

注意

このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。また、プロジェクトがClover Serverで実行されている必要もあります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
KillJobflow	いいえ	いいえ	0-1	0-1	はい	いいえ	はい

概要

このコンポーネントは**KillGraph**と同様に動作します。KillGraph(p.721)コンポーネントの説明を参照してください。

アイコン

ポート

KillGraphの「ポート」(p.722)を参照してください。

KillJobflowの属性

KillGraphの属性(p.722)を参照してください。

MonitorGraph

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは「**Palette**」→「**Job Control**」にあります。

要約

MonitorGraphを使用すると、実行中のグラフを監視できます。コンポーネントは、最終的な実行ステータスを待機するか、現在の実行ステータスを定期的に監視できます。

注意

このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。また、プロジェクトがClover Serverで実行されている必要もあります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
MonitorGraph	いいえ	いいえ	0-1	0-2	はい	いいえ	はい

概要

MonitorGraphコンポーネントを使用すると、実行中のグラフを監視できます。各受信トークンは、そのトークンから抽出された実行IDにより指定されたグラフの新しいモニターをトリガーします。複数のグラフを一度に監視することが可能です。

1つのグラフ・モニターがグラフを監視し、その終了を待機します。グラフの実行が終了すると、**ExecuteGraph**コンポーネントの場合と同様にグラフ結果が出力ポートに送信されます。成功したグラフの結果は最初の出力ポートに送信され、失敗したグラフは2番目の出力ポートに送信されます。さらに、「monitoring interval」属性で指定された時間が経過すると、グラフ・モニターは、まだ実行されているグラフについても現在のグラフ・ステータス情報を送信します。

入力ポートが接続されていない場合、コンポーネントの属性で指定されている設定で1つのグラフのみが監視されます。最初の出力ポートが接続されていない場合、コンポーネントはサブグラフ結果のログへの出力のみを行います。2番目の出力ポート(エラー・ポート)が接続されていない場合、最初に失敗したサブグラフによって親ジョブが中断されます。

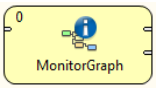
入力マッピングは、受信トークンからグラフ・モニターの実行IDおよびその他の設定を抽出する方法を定義します。グラフ結果または実際のグラフ・ステータスを出力ポートにマップすることが必要になった場合は、常に、出

力マッピングおよびエラー・マッピングの属性が、出力トークンへの移入に使用されます。グラフ結果で使用可能な情報は主に、一般的なランタイム情報、ディクショナリ・コンテンツおよびトラッキング情報で構成されます。

注意

MonitorGraphコンポーネントにより監視できるのは、現在のジョブフローにより実行されるグラフ(直接の子)のみです。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	監視対象グラフの識別情報を含む入力トークン	任意
出力	0	いいえ	成功したグラフの実行情報	任意
	1	いいえ	失敗したグラフの実行情報	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

MonitorGraphの属性

属性	必須	説明	可能な値
Basic			
Graph URL	いいえ	通常の監視対象グラフを表すグラフのパス。この属性により参照されるグラフはすべてのマッピング・ダイアログでも使用されます。マッピング・ダイアログには、このグラフに基づくディクショナリ・エントリおよびトラッキング情報が表示されます。	
Timeout	いいえ	グラフ実行専用の最大時間。デフォルトではミリ秒ですが、他の時間単位(p.275)も使用できます。グラフの実行時間がこの属性で指定されている時間を超えると、現在のグラフ情報がTIMEOUTステータスとともにエラー出力ポートに送信されます。 これはすべてのグラフ・モニターのデフォルト値にすぎません。各グラフ・モニターについて、入力マッピングで個別にオーバーライドできます。	0 (無制限) | 正数
Monitoring interval	いいえ	この属性で指定されている時間が経過するたびに、グラフ・モニターは実際のグラフ・ステータス情報を最初の出力ポートに送信します。デフォルトの間隔はミリ秒単位ですが、他の時間単位(p.275)も使用できます。 デフォルトでは、最終グラフ結果のみが出力ポートに送信されます。 これはすべてのグラフ・モニターのデフォルト値にすぎません。各グラフ・モニターについて、入力マッピングで個別にオーバーライドできます。	none (デフォルト) | 正数

属性	必須	説明	可能な値
Input mapping	いいえ	「Input mapping」は、受信トークンから実行IDおよび他のグラフ・モニター設定を抽出する方法を定義します。入力マッピング(p.727)を参照してください。	CTL変換
Output mapping	いいえ	「Output mapping」は、成功したグラフの結果を最初の出力ポートにマップします。監視間隔が指定されている場合、出力マッピングは現在のステータスを送信するためにも使用されます。出力マッピング(p.727)を参照してください。	CTL変換
Error mapping	いいえ	「Error mapping」は、失敗したグラフの結果を2番目の出力ポートにマップします。エラー・マッピング(p.727)を参照してください。	CTL変換
Redirect error output	いいえ	デフォルトでは、失敗したグラフの結果は2番目の出力ポート(エラー・ポート)に送信されます。これをtrueに変更した場合、失敗したグラフの結果が成功したグラフと同様に最初の出力ポートに送信されます。	false (デフォルト) | true
Advanced			
Run ID	いいえ	静的に定義された監視対象グラフの実行ID。この属性は、通常、入力マッピングにおいて受信トークンのデータでオーバーライドされます。	string

入力マッピング

入力マッピングは、監視対象グラフの実行IDおよび各グラフ・モニターの設定を指定するために受信トークンごとに実行される通常のCTL変換です。使用可能な出力フィールドは次のとおりです。

フィールド名	型	説明
runId	long	監視対象グラフの実行ID。コンポーネント属性「**Run ID**」をオーバーライドします。
timeout	long	コンポーネント属性「**Timeout**」をオーバーライドします。
monitoringInterval	long	コンポーネント属性「**Monitoring interval**」をオーバーライドします。

出力マッピング

出力マッピングは、最初の出力ポートに渡されたトークンを移入するために使用される標準的なCTL変換です。このマッピングは、成功したグラフまたはまだ実行中のグラフの現在のステータスに対して実行され、監視間隔が指定されている場合に送信されます。この出力マッピングの入力レコードの詳細は、ExecuteGraph(p.690)コンポーネントの説明を参照してください。

グラフが完了するかタイムアウトが経過すると、グラフ・モニターはグラフの監視を終了します。グラフの監視を停止する別の方法として、出力マッピングでSTOP定数を返す方法があります。

エラー・マッピング

エラー・マッピングは、出力マッピングとほぼ同じです。このエラー・マッピングは、グラフが失敗で終了した場合またはタイムアウトが経過した場合にのみ使用されます。2番目の出力ポートがエラー・マッピングにより移入されます。

MonitorJobflow

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

要約

MonitorJobflowを使用すると、実行中のジョブフローを監視できます。コンポーネントは、最終的な実行ステータスを待機するか、現在の実行ステータスを定期的に監視できます。

注意

このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。また、プロジェクトがClover Serverで実行されている必要もあります。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
MonitorJobflow	いいえ	いいえ	0-1	0-2	はい	いいえ	はい

概要

このコンポーネントは**MonitorGraph**と同様に動作します。MonitorGraph(p.725)コンポーネントの説明を参照してください。

アイコン

ポート

MonitorGraphの「ポート」(p.726)を参照してください。

MonitorJobflowの属性

MonitorGraphの属性(p.726)を参照してください。

SetJobOutput

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**コンポーネントを見つけるには、ジョブ制御の比較(p.333)を参照してください。

コンポーネントは**「Palette」→「Job Control」**にあります。

要約

SetJobOutputは入力レコードを受信し、受信値をディクショナリ・コンテンツに設定します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
SetJobOutput	-	-	1	0	-	いいえ	はい

概要

コンポーネントSetJobOutputは、受信レコードを出力ディクショナリ・エントリに書き込みます。マッピングに基づいて出力ディクショナリ・エントリが移入されます。マッピングが指定されていない場合、フィールド値は同じ名前のディクショナリ・エントリに設定されます。

最初に、入力レコードによりディクショナリ・エントリの値が設定され、後続の入力レコードにより既存の値がオーバーライドされます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	ディクショナリに書き込まれるレコード用	任意1)

説明:

1): 出力ディクショナリ・エントリを含む同じ名前のフィールド値が自動的に書き込まれます。

SetJobOutputの属性

属性	必須	説明	可能な値
Basic			
Mapping	いいえ	この属性は、入力レコード・メタデータから出力ディクショナリ・エントリへのマッピングを指定します。各受信レコードはこのマッピングにより処理され、その値がディクショナリにマップされます。実際に、マッピング属性は、入力メタデータ構造から出力ディクショナリ・エントリを表すレコードへの通常のCTL変換です。	

Success

Jobflowコンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

コンポーネントは**「Palette」→「Job Control」**にあります。

要約

Successは、ジョブフローでの成功したエンドポイントです。

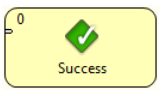
コンポーネント	データ出力	入力ポート	出力ポート	変換	変換が必要	Java	CTL
Success	なし	0-1	0	いいえ	いいえ	いいえ	はい

概要

Successは、ジョブフローでの成功したエンドポイントです。このコンポーネントで受信したトークンは、これ以上処理されません。これらは、ジョブフロー内で正常に処理されたとみなされます。このコンポーネントは、ジョブフローでの成功の視覚的なマーカーとして機能します。

このコンポーネントではメッセージを記録し、ディクショナリのコンテンツを設定できます。Fail(p.717)コンポーネントに似ています。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0-1	はい	受信したトークン用	任意

Successの属性

属性	必須	説明	可能な値
Basic			
Message	いいえ	各受信トークンに関して記録されるテキスト・メッセージ。	テキスト
Mapping	いいえ	Mappingは、ログ・メッセージの動的アセンブルに使用されます。さらに、ディクショナリ・コンテンツも変更できます。詳細説明(p.732)を参照してください。	

詳細説明

- **マッピングの詳細**

  Successコンポーネントでのマッピングは、通常、次の2つの目的に使用されます。

  - 受信レコードからのログ・メッセージのアセンブル。
  - 受信レコードからのディクショナリ・コンテンツの移入。

  ### 注意

  変更できるのは出力ディクショナリ・エントリのみです。

  マッピングによりコンパイルされたログ・メッセージは、最も高い優先度を持ちます。マッピングでメッセージが設定されていない場合は、コンポーネントの属性からのメッセージがかわりに使用されます。メッセージが属性またはマッピングにより設定されていない場合は、何も記録されません。

TokenGather

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第50章「ジョブ制御の共通プロパティ」(p.333)

目的に適した**ジョブ制御**を見つけるには、ジョブ制御の比較(p.333)を参照してください。

要約

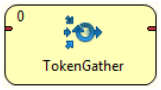
TokenGatherは、任意の入力ポートからの各受信トークンをすべての接続済出力ポートにコピーします。入力メタデータが出力メタデータと異なる場合は、フィールド名に基づくコピーが使用されます。このコンポーネントは、通常、複数のパラレル実行分岐からのすべてのトークンを収集し、統合された単一の出力に送信するために使用されます。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
TokenGather	いいえ	いいえ	1-n	1-n	-	-

概要

TokenGatherコンポーネントは、任意の入力ポートから受信トークンを受信し、それらをすべての接続済出力ポートにコピーします。各受信トークンは、すべての出力ポートにコピーされます。入力ポートおよび出力ポートは、任意のメタデータを持つことができます。入力メタデータから出力メタデータへのコピーは、フィールド名に基づいて行われます。フィールド値は、出力トークンに同一の名前のフィールドがある場合にのみ、出力トークンに移動されます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0-n	1つ以上	受信トークン用	任意
出力	0-n	1つ以上	収集されたトークン用	任意1)

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

説明:

1): 入力フィールドと同じ名前のフィールドのみが移入されます。

第58章 ファイル操作

コンポーネントとは何かを理解していることを想定しています。概要は、第19章「コンポーネント」(p.97)を参照してください。

ファイル・システム操作用に設計されたコンポーネントのグループは、**ファイル操作**と呼ばれます。

ファイル操作コンポーネントは、ファイルとディレクトリを作成、コピー、移動および削除し、ディレクトリをリストし、ファイル属性を読み込むことができます。これらのコンポーネントは、FTPまたはApache Hadoop HDFSを介してローカル・ファイルおよびリモート・ファイルを操作できます。サーバー上で実行される場合は、サンドボックスへのアクセスもサポートされます。他のプロトコルでの制限付きサポートも提供します(HTTPプロトコルを使用したWebからのファイルのコピーなど)。ただし、アーカイブされたコンテンツの操作はサポートされません(zip、gzip、tarプロトコルなど)。

リモート・ファイルを操作する場合は、サーバーとクライアントが同期されている必要があります。

コンポーネントには様々なプロパティを設定できます。ただし、一部のものが共通する場合があります。すべてのコンポーネントに共通のプロパティもあれば、ほとんどのコンポーネントに共通のプロパティもあります。次のことを学習している必要があります。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第51章「ファイル操作の共通プロパティ」(p.334)

CopyFiles

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第51章「ファイル操作の共通プロパティ」(p.334)

目的に適した**ファイル操作**コンポーネントを見つけるには、ファイル操作の比較(p.334)を参照してください。

コンポーネントは「**Palette**」→「**File Operations**」にあります。

要約

CopyFilesは、ファイルおよびディレクトリをコピーするために使用できます。

注意

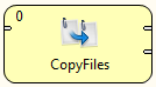
このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。

コンポーネント	入力	出力
CopyFiles	0-1	0-2

概要

CopyFilesは、複数のソースを1つの宛先ディレクトリにコピーしたり、1つの標準ソース・ファイルを1つのターゲット・ファイルにコピーできます。ディレクトリは、再帰的にコピーできます。オプションで、既存のファイルをその変更日に基づいてスキップまたは更新できます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	コンポーネント属性にマップする入力データ・レコード。	任意
出力	0	いいえ	結果	任意
	1	いいえ	エラー	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

CopyFilesの属性

属性	必須	説明	可能な値
Basic			
Source file URL	はい1)	ソース・ファイルまたはディレクトリへのパス(ファイル操作にサポートされているURL形式(p.335)を参照してください)。	
Target file URL	はい1)	宛先ファイルまたはディレクトリへのパス(ファイル操作にサポートされているURL形式(p.335)を参照してください)。ディレクトリを指している場合、ソースはそのディレクトリにコピーされます。 これは単一のファイルまたはディレクトリへのパスである必要があります。	
Recursive	いいえ	ディレクトリを再帰的にコピーします。	false (デフォルト) | true
Overwrite	いいえ	既存ファイルを上書きするかどうかを指定します。更新モードでは、ソース・ファイルが宛先ファイルより新しい場合にのみターゲットが上書きされます。	always (デフォルト) | update | never
Create parent directories	いいえ	存在しない親ディレクトリの作成を試みます。 「**Create parent directories**」オプションが有効で、「**Target file URL**」がスラッシュ(/)で終了している場合は、親ディレクトリとして扱われます。つまり、ソース・ディレクトリまたはファイルは、ターゲット・ディレクトリにコピーされます。	false (デフォルト) | true
Input mapping	2)	コンポーネント属性への入力レコードのマッピングを定義します。	
Output mapping	2)	標準出力ポートへの結果のマッピングを定義します。	
Error mapping	2)	エラー出力ポートへのエラーのマッピングを定義します。	
Redirect error output	いいえ	有効な場合、エラーはエラー・ポートではなく出力ポートに送信されます。	false (デフォルト) | true
Verbose output	いいえ	有効な場合、1つの入力レコードが原因で複数のレコードが出力に送信されることがあります(たとえば、ワイルドカード拡張の結果として)。そうでない場合、入力レコードごとに累積出力レコードが1つのみ生成されます。	false (デフォルト) | true
Advanced			
Stop processing on fail	いいえ	デフォルトでは、失敗するとコンポーネントは以降の操作をすべてスキップし、スキップした操作に関する情報をエラー出力ポートに送信します。この属性で、この動作をオフにできます。	true (デフォルト) | false

説明

1) **入力マッピング**で指定されていないかぎり、この属性は必須です。

2) 対応するエッジが接続されている場合に必須です。

詳細説明

入力、**出力**または**エラー・マッピング**のいずれかを編集すると、変換エディタ(p.286)が開きます。

入力マッピング: エディタを使用して、コンポーネントの選択した属性を入力フィールドの値でオーバーライドできます。

フィールド名	属性	型	可能な値
sourceURL	Source file URL	string	
targetURL	Target file URL	string	
recursive	Recursive	boolean	true | false
overwrite	Overwrite	string	"always" | "update" | "never"
makeParentDirs	Create parent directories	boolean	true | false

出力マッピング: エディタを使用して、結果および入力データを出力ポートにマップできます。

フィールド名	型	説明
sourceURL	string	ソース・ファイルのURL。
targetURL	string	宛先のURL。
resultURL	string	正常にコピーされたファイルの新しいURL。「**Verbose output**」モードでのみ設定されます。
result	boolean	操作が成功した場合はtrue (「**Redirect error output**」が有効な場合はfalseになることがあります)。
errorMessage	string	操作が失敗した場合、このフィールドにエラー・メッセージが含まれます (「**Redirect error output**」が有効な場合に使用されます)。
stackTrace	string	操作が失敗した場合、このフィールドにエラーのスタック・トレースが含まれます (「**Redirect error output**」が有効な場合に使用されます)。

エラー・マッピング: エディタを使用して、エラーおよび入力データをエラー・ポートにマップできます。

フィールド名	型	説明
result	boolean	常にfalseに設定されます。
errorMessage	string	エラー・メッセージ。
stackTrace	string	エラーのスタック・トレース。
sourceURL	string	ソース・ファイルのURL。
targetURL	string	宛先のURL。

CreateFiles

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第51章「ファイル操作の共通プロパティ」(p.334)

目的に適した**ファイル操作**コンポーネントを見つけるには、ファイル操作の比較(p.334)を参照してください。

コンポーネントは「**Palette**」→「**File Operations**」にあります。

要約

CreateFilesは、ファイルとディレクトリの作成、およびこれらの変更日を設定するために使用できます。

注意

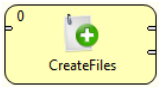
このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。

コンポーネント	入力	出力
CreateFiles	0-1	0-2

概要

CreateFilesは、ファイルとディレクトリを作成できます。また、既存および新規作成のファイルとディレクトリの両方について、変更日を設定できます。オプションで、存在しない親ディレクトリも作成できます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	コンポーネント属性にマップする入力データ・レコード。	任意
出力	0	いいえ	結果	任意
	1	いいえ	エラー	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

CreateFilesの属性

属性	必須	説明	可能な値
Basic			
File URL	はい1)	作成するファイルまたはディレクトリへのパス(ファイル操作にサポートされているURL形式(p.335)を参照してください)。スラッシュ(/)で終了する場合は、ディレクトリを作成する必要があることを示します。これは、「**Create as directory**」属性を使用して指定することもできます。	
Create as directory	いいえ	標準のファイルではなくディレクトリを作成する必要があることを指定します。	false (デフォルト) | true
Create parent directories	いいえ	存在しない親ディレクトリの作成を試みます。	false (デフォルト) | true
Last modified date	いいえ	既存および新規作成のファイルの最終更新日を、指定した値に設定します。日付の書式は、DEFAULT_DATETIME_FORMATプロパティ(デフォルトのCloverETL設定の変更(p.88))で定義されています。	
Input mapping	2)	コンポーネント属性への入力レコードのマッピングを定義します。	
Output mapping	2)	標準出力ポートへの結果のマッピングを定義します。	
Error mapping	2)	エラー出力ポートへのエラーのマッピングを定義します。	
Redirect error output	いいえ	有効な場合、エラーはエラー・ポートではなく標準出力ポートに送信されます。	false (デフォルト) | true
Verbose output	いいえ	有効な場合、1つの入力レコードが原因で複数のレコードが出力に送信されることがあります(たとえば、ワイルドカード拡張の結果として)。そうでない場合、入力レコードごとに累積出力レコードが1つのみ生成されます。	false (デフォルト) | true
Advanced			
Stop processing on fail	いいえ	デフォルトでは、失敗するとコンポーネントは以降の操作をすべてスキップし、スキップした操作に関する情報をエラー出力ポートに送信します。この属性で、この動作をオフにできます。	true (デフォルト) | false

説明

1) **入力マッピング**で指定されていないかぎり、この属性は必須です。

2) 対応するエッジが接続されている場合に必須です。

詳細説明

入力、**出力**または**エラー・マッピング**のいずれかを編集すると、変換エディタ(p.286)が開きます。

入力マッピング: エディタを使用して、コンポーネントの選択した属性を入力フィールドの値でオーバーライドできます。

フィールド名	属性	型	可能な値
fileURL	File URL	string	
directory	Create as directory	boolean	true | false
makeParentDirs	Create parent directories	boolean	true | false
modifiedDate	Last modified date	date	

出力マッピング: エディタを使用して、結果および入力データを出力ポートにマップできます。

フィールド名	型	説明
fileURL	string	ターゲット・ファイルまたはディレクトリのURL。
result	boolean	操作が成功した場合はtrue (「**Redirect error output**」が有効な場合はfalseになることがあります)。
errorMessage	string	操作が失敗した場合、このフィールドにエラー・メッセージが含まれます (「**Redirect error output**」が有効な場合に使用されます)。
stackTrace	string	操作が失敗した場合、このフィールドにエラーのスタック・トレースが含まれます (「**Redirect error output**」が有効な場合に使用されます)。

エラー・マッピング: エディタを使用して、エラーおよび入力データをエラー・ポートにマップできます。

フィールド名	型	説明
result	boolean	常にfalseに設定されます。
errorMessage	string	エラー・メッセージ。
stackTrace	string	エラーのスタック・トレース。
fileURL	string	ターゲット・ファイルまたはディレクトリのURL。

DeleteFiles

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第51章「ファイル操作の共通プロパティ」(p.334)

目的に適した**ファイル操作**コンポーネントを見つけるには、ファイル操作の比較(p.334)を参照してください。

コンポーネントは「**Palette**」→「**File Operations**」にあります。

要約

DeleteFilesは、ファイルおよびディレクトリを削除するために使用できます。

注意

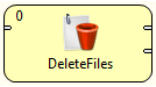
このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。

コンポーネント	入力	出力
DeleteFiles	0-1	0-2

概要

DeleteFilesは、ファイルおよびディレクトリを削除するために使用できます(再帰的実行も可能)。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	コンポーネント属性にマップする入力データ・レコード。	任意
出力	0	いいえ	結果	任意
	1	いいえ	エラー	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

DeleteFilesの属性

属性	必須	説明	可能な値
Basic			
File URL	はい1)	削除するファイルまたはディレクトリへのパス(ファイル操作にサポートされているURL形式(p.335)を参照してください)。	
Recursive	いいえ	ディレクトリを再帰的に削除します。	false (デフォルト) | true
Input mapping	2)	コンポーネント属性への入力レコードのマッピングを定義します。	
Output mapping	2)	標準出力ポートへの結果のマッピングを定義します。	
Error mapping	2)	エラー出力ポートへのエラーのマッピングを定義します。	
Redirect error output	いいえ	有効な場合、エラーはエラー・ポートではなく標準出力ポートに送信されます。	false (デフォルト) | true
Verbose output	いいえ	有効な場合、1つの入力レコードが原因で複数のレコードが出力に送信されることがあります(たとえば、ワイルドカード拡張の結果として)。そうでない場合、入力レコードごとに累積出力レコードが1つのみ生成されます。	false (デフォルト) | true
Advanced			
Stop processing on fail	いいえ	デフォルトでは、失敗するとコンポーネントは以降の操作をすべてスキップし、スキップした操作に関する情報をエラー出力ポートに送信します。この属性で、この動作をオフにできます。	true (デフォルト) | false

説明

1) **入力マッピング**で指定されていないかぎり、この属性は必須です。

2) 対応するエッジが接続されている場合に必須です。

詳細説明

入力、**出力**または**エラー・マッピング**のいずれかを編集すると、変換エディタ(p.286)が開きます。

入力マッピング: エディタを使用して、コンポーネントの選択した属性を入力フィールドの値でオーバーライドできます。

フィールド名	属性	型	可能な値
fileURL	File URL	string	
recursive	Recursive	boolean	true | false

出力マッピング: エディタを使用して、結果および入力データを出力ポートにマップできます。

フィールド名	型	説明
fileURL	string	削除されたファイルまたはディレクトリへのパス。
result	boolean	操作が成功した場合はtrue (「**Redirect error output**」が有効な場合はfalseになることがあります)。

フィールド名	型	説明
errorMessage	string	操作が失敗した場合、このフィールドにエラー・メッセージが含まれます (「**Redirect error output**」が有効な場合に使用されます)。
stackTrace	string	操作が失敗した場合、このフィールドにエラーのスタック・トレースが含まれます (「**Redirect error output**」が有効な場合に使用されます)。

エラー・マッピング: エディタを使用して、エラーおよび入力データをエラー・ポートにマップできます。

フィールド名	型	説明
result	boolean	常にfalseに設定されます。
errorMessage	string	エラー・メッセージ。
stackTrace	string	エラーのスタック・トレース。
fileURL	string	削除されたファイルまたはディレクトリのURL。

ListFiles

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第51章「ファイル操作の共通プロパティ」(p.334)

目的に適した**ファイル操作**コンポーネントを見つけるには、ファイル操作の比較(p.334)を参照してください。

コンポーネントは「**Palette**」→「**File Operations**」にあります。

要約

ListFilesは、ディレクトリ・コンテンツをリストしたり、ファイル属性(サイズや変更日など)を取得するために使用できます。

注意

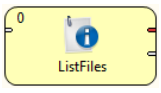
このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。

コンポーネント	入力	出力
ListFiles	0-1	1-2

概要

ListFilesは、個々のファイルの詳細情報を含む、ディレクトリ・コンテンツをリストします。サブディレクトリは、再帰的にリストできます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	コンポーネント属性にマップする入力データ・レコード。	任意
出力	0	はい	ターゲット・ディレクトリのエントリごとに1つのレコード	任意
	1	いいえ	エラー	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

ListFilesの属性

属性	必須	説明	可能な値
Basic			
File URL	はい1)	リストするファイルまたはディレクトリへのパス(ファイル操作にサポートされているURL形式(p.335)を参照してください)。	
Recursive	いいえ	サブディレクトリを再帰的にリストします。	false (デフォルト) | true
Input mapping	2)	コンポーネント属性への入力レコードのマッピングを定義します。	
Output mapping	2)	標準出力ポートへの結果のマッピングを定義します。	
Error mapping	2)	エラー出力ポートへのエラーのマッピングを定義します。	
Redirect error output	いいえ	有効な場合、エラーはエラー・ポートではなく標準出力ポートに送信されます。	false (デフォルト) | true
Advanced			
Stop processing on fail	いいえ	デフォルトでは、失敗するとコンポーネントは以降の操作をすべてスキップし、スキップした操作に関する情報をエラー出力ポートに送信します。この属性で、この動作をオフにできます。	true (デフォルト) | false

説明

1) **入力マッピング**で指定されていないかぎり、この属性は必須です。

2) 対応するエッジが接続されている場合に必須です。

詳細説明

入力、**出力**または**エラー・マッピング**のいずれかを編集すると、変換エディタ(p.286)が開きます。

入力マッピング: エディタを使用して、コンポーネントの選択した属性を入力フィールドの値でオーバーライドできます。

フィールド名	属性	型	可能な値
fileURL	File URL	string	
recursive	Recursive	boolean	true | false

出力マッピング: エディタを使用して、結果および入力データを出力ポートにマップできます。

フィールド名	型	説明
URL	string	ファイルまたはディレクトリのURL。
name	string	ファイル名。
canRead	boolean	ファイルを読み取れる場合はtrue。
canWrite	boolean	ファイルを変更できる場合はtrue。
canExecute	boolean	ファイルを実行できる場合はtrue。
isDirectory	boolean	ファイルが存在し、それがディレクトリである場合はtrue。
isFile	boolean	ファイルが存在し、それが標準のファイルである場合はtrue。

フィールド名	型	説明
isHidden	boolean	ファイルが非表示の場合はtrue。
lastModified	date	ファイルが最後に変更された時間。
size	long	バイト単位のファイル・サイズの場合はtrue。
result	boolean	操作が成功した場合はtrue (「**Redirect error output**」が有効な場合はfalseになることがあります)。
errorMessage	string	操作が失敗した場合、このフィールドにエラー・メッセージが含まれます(「**Redirect error output**」が有効な場合に使用されます)。
stackTrace	string	操作が失敗した場合、このフィールドにエラーのスタック・トレースが含まれます(「**Redirect error output**」が有効な場合に使用されます)。

エラー・マッピング: エディタを使用して、エラーおよび入力データをエラー・ポートにマップできます。

フィールド名	型	説明
result	boolean	常にfalseに設定されます。
errorMessage	string	エラー・メッセージ。
stackTrace	string	エラーのスタック・トレース。

MoveFiles

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第51章「ファイル操作の共通プロパティ」(p.334)

目的に適した**ファイル操作**コンポーネントを見つけるには、ファイル操作の比較(p.334)を参照してください。

コンポーネントは「**Palette**」→「**File Operations**」にあります。

要約

MoveFilesは、ファイルおよびディレクトリを移動するために使用できます。

注意

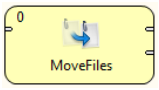
このコンポーネントを使用するには、個別のジョブフロー・ライセンスが必要です。

コンポーネント	入力	出力
MoveFiles	0-1	0-2

概要

MoveFilesは、複数のソースを1つの宛先ディレクトリにコピーしたり、1つの標準ソース・ファイルを1つのターゲット・ファイルに移動できます。オプションで、既存のファイルをその変更日に基づいてスキップまたは更新できます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	コンポーネント属性にマップする入力データ・レコード。	任意
出力	0	いいえ	結果	任意
	1	いいえ	エラー	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

MoveFilesの属性

属性	必須	説明	可能な値
Basic			
Source file URL	はい1)	ソース・ファイルまたはディレクトリへのパス(ファイル操作にサポートされているURL形式(p.335)を参照してください)。	
Target file URL	はい1)	宛先ファイルまたはディレクトリへのパス(ファイル操作にサポートされているURL形式(p.335)を参照してください)。ディレクトリを指している場合、ソースはそのディレクトリに移動されます。 これは単一のファイルまたはディレクトリへのパスである必要があります。	
Overwrite	いいえ	既存ファイルを上書きするかどうかを指定します。更新モードでは、ソース・ファイルが宛先ファイルより新しい場合にのみターゲットが上書きされます。	always (デフォルト) | update | never
Create parent directories	いいえ	存在しない親ディレクトリの作成を試みます。 「**Create parent directories**」オプションが有効で、「**Target file URL**」がスラッシュ(/)で終了している場合は、親ディレクトリとして扱われます。つまり、ソース・ディレクトリまたはファイルは、ターゲット・ディレクトリが存在しない場合でも、ターゲット・ディレクトリに移動されます。	false (デフォルト) | true
Input mapping	2)	コンポーネント属性への入力レコードのマッピングを定義します。	
Output mapping	2)	標準出力ポートへの結果のマッピングを定義します。	
Error mapping	2)	エラー出力ポートへのエラーのマッピングを定義します。	
Redirect error output	いいえ	有効な場合、エラーはエラー・ポートではなく出力ポートに送信されます。	false (デフォルト) | true
Verbose output	いいえ	有効な場合、1つの入力レコードが原因で複数のレコードが出力に送信されることがあります(たとえば、ワイルドカード拡張の結果として)。そうでない場合、入力レコードごとに累積出力レコードが1つのみ生成されます。	false (デフォルト) | true
Advanced			
Stop processing on fail	いいえ	デフォルトでは、失敗するとコンポーネントは以降の操作をすべてスキップし、スキップした操作に関する情報をエラー出力ポートに送信します。この属性で、この動作をオフにできます。	true (デフォルト) | false

説明

1) **入力マッピング**で指定されていないかぎり、この属性は必須です。

2) 対応するエッジが接続されている場合に必須です。

詳細説明

入力、**出力**または**エラー・マッピング**のいずれかを編集すると、変換エディタ(p.286)が開きます。

入力マッピング: エディタを使用して、コンポーネントの選択した属性を入力フィールドの値でオーバーライドできます。

フィールド名	属性	型	可能な値
sourceURL	Source file URL	string	
targetURL	Target file URL	string	
overwrite	Overwrite	string	"always" | "update" | "never"
makeParentDirs	Create parent directories	boolean	true | false

出力マッピング: エディタを使用して、結果および入力データを出力ポートにマップできます。

フィールド名	型	説明
sourceURL	string	ソース・ファイルのURL。
targetURL	string	宛先のURL。
resultURL	string	正常に移動されたファイルの新しいURL。「**Verbose output**」モードでのみ設定されます。
result	boolean	操作が成功した場合はtrue (「**Redirect error output**」が有効な場合はfalseになることがあります)。
errorMessage	string	操作が失敗した場合、このフィールドにエラー・メッセージが含まれます (「**Redirect error output**」が有効な場合に使用されます)。
stackTrace	string	操作が失敗した場合、このフィールドにエラーのスタック・トレースが含まれます (「**Redirect error output**」が有効な場合に使用されます)。

エラー・マッピング: エディタを使用して、エラーおよび入力データをエラー・ポートにマップできます。

フィールド名	型	説明
result	boolean	常にfalseに設定されます。
errorMessage	string	エラー・メッセージ。
stackTrace	string	エラーのスタック・トレース。
sourceURL	string	ソース・ファイルのURL。
targetURL	string	宛先のURL。

第59章 クラスタ・コンポーネント

コンポーネントとは何かを理解していることを想定しています。概要は、第19章「コンポーネント」(p.97)を参照してください。

このカテゴリのコンポーネントは、主に**CloverETL Cluster**環境でのデータ・フロー管理用であり、データ変換処理の大規模なパラレル化を可能にします。クラスタ環境で実行される変換グラフでの各コンポーネントは複数のインスタンスで実行でき、これはコンポーネントの割当てと呼ばれます。コンポーネントの割当てでは、実行されるインスタンス数、およびどこで実行されるか(どのクラスタ・ノード上であるか)を指定します。詳細は、**CloverETL Cluster**のドキュメントを参照してください。

通常、クラスタ・コンポーネントは2つのサブカテゴリ(**パーティショナ**および**収集**)に分けられます。

クラスタ・パーティショナは、単一ワーカーからのデータ・レコードを様々なクラスタ・ワーカーに分配します。クラスタ・パーティショナは、実際には単一ワーカー割当てを複数ワーカー割当てに変更するために使用されます。

- ClusterPartition(p.751)は、データ・レコードを様々なワーカーに分配します。このコンポーネントのアルゴリズムは**Partition**コンポーネントに基づいています。
- ClusterLoadBalancingPartition(p.753)は、データ・レコードを様々なワーカーに分配します。このコンポーネントのアルゴリズムは**LoadBalancingPartition**コンポーネントに基づいています。
- ClusterSimpleCopy(p.755)は、データ・レコードを様々なワーカーにコピーします。このコンポーネントのアルゴリズムは**SimpleCopy**コンポーネントに基づいています。このため、受信データは複製され、すべての出力ワーカーに送信されます。

一方、**クラスタ収集**は、様々なクラスタ・ワーカーからのデータ・レコードを単一ワーカーに収集します。クラスタ収集は、実際には複数ワーカー割当てを単一ワーカー割当てに変更するために使用されます。

- ClusterSimpleGather(p.757)は、データ・レコードを様々なクラスタ・ワーカーから収集します。このコンポーネントのアルゴリズムは**SimpleGather**コンポーネントに基づいています。
- ClusterMerge(p.759)は、データ・レコードを様々なクラスタ・ワーカーから収集します。このコンポーネントのアルゴリズムは**Merge**コンポーネントに基づいています。

両方の基本クラスタ・コンポーネント・グループからClusterRepartitionコンポーネントが派生します。

- ClusterRepartition(p.761)は、すでにパーティション化されたデータのパーティションを変更し、データは再パーティション化されます。たとえば、キーに応じてデータがClusterPartitionコンポーネントですでにパーティション化されている場合、このキーを変更するか、またはパーティション数を変更する必要があるとき、このコンポーネントはこの処理を1つの手順で行い(この際、単一ワーカーへのすべてのパーティション化済データの収集をクラスタ収集で行う必要がありません(ボトルネックの回避))、新規ルールに応じてデータを再びクラスタ・パーティショナでパーティション化します。

各コンポーネントには様々なプロパティを設定できます。ただし、一部のものが共通する場合があります。すべてのコンポーネントに共通のプロパティもあれば、ほとんどのコンポーネントに共通のプロパティや、一部のコンポーネントのみに共通のプロパティもあります。次のことを学習している必要があります。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第47章「クラスタ・コンポーネントの共通プロパティ」(p.330)

ClusterPartition

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第47章「クラスタ・コンポーネントの共通プロパティ」(p.330)

要約

ClusterPartitionは、受信データ・レコードを様々な**CloverETL Cluster**ワーカーに分配します。このコンポーネントのアルゴリズムは、標準のPartition(p.619)コンポーネントから導出されます。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
ClusterPartition	はい	いいえ	1	1[1)]	はい/いいえ[2)]	はい/いいえ[2)]

説明

1) 1つの出力ポートが複数の仮想出力ポートを表します。

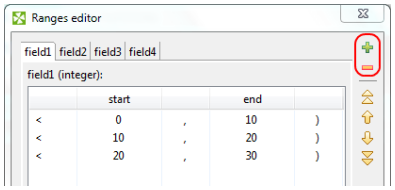
2) **ClusterPartition**では、変換またはその他の2つの属性(「**Ranges**」および「**Partition key**」)のいずれかを使用できます。これらの属性が1つ以上指定されていない場合は、変換を定義する必要があります。

概要

ClusterPartitionは、受信データ・レコードを様々な**CloverETL Cluster**ワーカーに分配します。

このコンポーネントのアルゴリズムは、標準のPartition(p.619)コンポーネントから導出されます。属性およびコンポーネント固有のその他の動作の詳細は、Partition(p.619)コンポーネントのドキュメントを参照してください。

「**Ranges**」属性がパーティションに使用される場合、定義済範囲の数が、次のコンポーネントの割当て(p.273)に一致する必要があります。次に示すツールバーの**追加**ボタンと**削除**ボタンを使用して、定義済範囲の数を調整します。

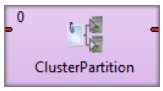


このコンポーネントは、単一ワーカー割当てから複数ワーカー割当てへの変更を可能にするクラスタ・コンポーネントのグループに属します。このため、**ClusterPartition**コンポーネントの前のコンポーネントの割当てでは、単一ワーカーのみを指定する必要があります。**ClusterPartition**コンポーネントの後のコンポーネントの割当てでは、複数のワーカーを指定できます。

注意

このコンポーネントの使用方法の詳細は、**CloverETL Cluster**のドキュメントを参照してください。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	出力データ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。

ClusterLoadBalancingPartition

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第47章「クラスタ・コンポーネントの共通プロパティ」(p.330)

要約

ClusterLoadBalancingPartitionは、受信データ・レコードを様々な**CloverETL Cluster**ワーカーに分配します。このコンポーネントのアルゴリズムは、標準のLoadBalancingPartition(p.626)コンポーネントから導出されます。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
ClusterLoadBalancingPartition	はい	いいえ	1	1[1)]	いいえ	いいえ

説明

1) 1つの出力ポートが複数の仮想出力ポートを表します。

概要

ClusterLoadBalancingPartitionは、受信データ・レコードを様々な**CloverETL Cluster**ワーカーに分配します。

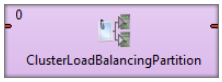
このコンポーネントのアルゴリズムは、標準のLoadBalancingPartition(p.626)コンポーネントから導出されます。属性およびコンポーネント固有のその他の動作の詳細は、LoadBalancingPartitionコンポーネントのドキュメントを参照してください。

このコンポーネントは、単一ワーカー割当てから複数ワーカー割当てへの変更を可能にするクラスタ・コンポーネントのグループに属します。このため、**ClusterLoadBalancingPartition**コンポーネントの前のコンポーネントの割当てでは、単一ワーカーのみを指定する必要があります。**ClusterLoadBalancingPartition**コンポーネントの後のコンポーネントの割当てでは、複数のワーカーを指定できます。

注意

このコンポーネントの使用方法の詳細は、**CloverETL Cluster**のドキュメントを参照してください。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	出力データ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。

ClusterSimpleCopy

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第47章「クラスタ・コンポーネントの共通プロパティ」(p.330)

要約

ClusterSimpleCopyは、受信データ・レコードをすべての出力**CloverETL Cluster**ワーカーにコピーします。このコンポーネントのアルゴリズムは、標準のSimpleCopy(p.647)コンポーネントから導出されます。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
ClusterSimpleCopy	はい	いいえ	1	1[1)]	いいえ	いいえ

説明

1) 1つの出力ポートが複数の仮想出力ポートを表します。

概要

ClusterSimpleCopyは、受信データ・レコードをすべての出力**CloverETL Cluster**ワーカーにコピーします。各受信レコードは複製され、すべての出力パーティションに送信されます。

このコンポーネントは、データをすべてのワーカーで使用できるようにする必要がある場合に役立ちます。たとえば、大量のビジネス・トランザクションを並行して処理することを決定します。**ClusterPartition**は、データを複数のワーカーに分割する場合に適したコンポーネントです。ここで、たとえばトランザクションを(トランザクションが実行される)国コードと結合する必要があることに気付きます。すべての国コードのリストをすべてのワーカーで使用可能にする必要があります。各ワーカーは、国コードをそれぞれ取得できますが、データ読取りが高負荷である場合(低速のWebサービスからの読取りの場合など)は、一度読み取ってからclover機能を使用してそれらをすべてのワーカーにコピーすることをお薦めします。これにより、単一ワーカーで低速なデータ・ソースから国コードを一度だけ読み取り、**ClusterSimpleCopy**を使用してすべてのワーカーにそれらをコピーし、そこでトランザクションとの結合に使用できます。

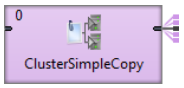
このコンポーネントのアルゴリズムは、標準のSimpleCopy(p.647)コンポーネントから導出されます。詳細は、SimpleCopy(p.647)コンポーネントのドキュメントを参照してください。

このコンポーネントは、単一ワーカー割当てから複数ワーカー割当てへの変更を可能にするクラスタ・コンポーネントのグループに属します。このため、**ClusterSimpleCopy**コンポーネントの前のコンポーネントの割当てでは、単一ワーカーのみを指定する必要があります。**ClusterSimpleCopy**コンポーネントの後のコンポーネントの割当てでは、複数のワーカーを指定できます。

注意

このコンポーネントの使用方法の詳細は、**CloverETL Cluster**のドキュメントを参照してください。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	出力データ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。

ClusterSimpleGather

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第47章「クラスタ・コンポーネントの共通プロパティ」(p.330)

要約

ClusterSimpleGatherは、複数の**CloverETL Cluster**ワーカーからデータ・レコードを収集します。このコンポーネントのアルゴリズムは、標準のSimpleGather(p.648)コンポーネントから導出されます。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
ClusterSimpleGather	はい	いいえ	1[1)]	1	-	-

説明

1) 1つの入力ポートが複数の仮想入力ポートを表します。

概要

ClusterSimpleGatherは、複数の**CloverETL Cluster**ワーカーからデータ・レコードを収集します。

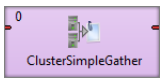
このコンポーネントのアルゴリズムは、標準のSimpleGatherコンポーネントから導出されます。属性およびコンポーネント固有のその他の動作の詳細は、SimpleGather(p.648)コンポーネントのドキュメントを参照してください。

このコンポーネントは、複数ワーカー割当てから単一ワーカー割当てへの変更を可能にするクラスタ・コンポーネントのグループに属します。このため、**ClusterSimpleGather**コンポーネントの前のコンポーネントの割当てでは、複数のワーカーを指定できます。**ClusterSimpleGather**コンポーネントの後のコンポーネントの割当てでは、単一ワーカーのみを指定する必要があります。

注意

このコンポーネントの使用方法の詳細は、**CloverETL Cluster**のドキュメントを参照してください。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	収集されたデータ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。

ClusterMerge

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第47章「クラスタ・コンポーネントの共通プロパティ」(p.330)

要約

ClusterMergeは、複数の**CloverETL Cluster**ワーカーからデータ・レコードを収集します。このコンポーネントのアルゴリズムは、標準のMerge(p.607)コンポーネントから導出されます。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
ClusterMerge	はい	はい	1[1)]	1	-	-

説明

1) 1つの入力ポートが複数の仮想入力ポートを表します。

概要

ClusterMergeは、複数の**CloverETL Cluster**ワーカーからデータ・レコードを収集します。

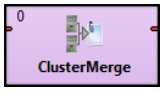
このコンポーネントのアルゴリズムは、標準のClusterMerge(p.759)コンポーネントから導出されます。属性およびコンポーネント固有のその他の動作の詳細は、Mergeコンポーネントのドキュメントを参照してください。

このコンポーネントは、複数ワーカー割当てから単一ワーカー割当てへの変更を可能にするクラスタ・コンポーネントのグループに属します。このため、**ClusterMerge**コンポーネントの前のコンポーネントの割当てでは、複数のワーカーを指定できます。**ClusterMerge**コンポーネントの後のコンポーネントの割当てでは、単一ワーカーを指定する必要があります。

注意

このコンポーネントの使用方法の詳細は、**CloverETL Cluster**のドキュメントを参照してください。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	収集されたデータ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。

ClusterRepartition

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)
- 第47章「クラスタ・コンポーネントの共通プロパティ」(p.330)

要約

ClusterRepartitionコンポーネントは、すでにパーティション化されたデータを新規ルールに応じて様々な一連の**CloverETL Cluster**ワーカーに再分配します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
ClusterRepartition	はい	いいえ	1[1)]	1[2)]	はい/いいえ[3)]	はい/いいえ[3)]

説明

1) 1つの入力ポートが複数の仮想入力ポートを表します。

2) 1つの出力ポートが複数の仮想出力ポートを表します。

3) **ClusterRepartition**は、変換またはその他の2つの属性(「**Ranges**」および「**Partition key**」)のいずれかを使用できます。これらの属性が1つ以上指定されていない場合は、変換を定義する必要があります。

概要

ClusterRepartitionコンポーネントは、すでにパーティション化されたデータを新規ルールに応じて様々な一連の**CloverETL Cluster**ワーカーに再分配します。

このコンポーネントは機能的にClusterPartition(p.751)コンポーネントに似ており、受信データ・レコードを様々な**CloverETL Cluster**ワーカーに分配します。ClusterPartitionとは異なり、受信データがすでにパーティション化されていてもかまいません。

このコンポーネントの背景の詳細は、再パーティショナの次の使用例で確認します。

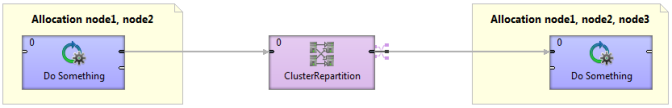


図59.1. ClusterRepartitionコンポーネントの使用例

ClusterRepartitionコンポーネントは、互換性のない2つの割当て間の境界を定義します。ClusterRepartitionの前のデータは、node1およびnode2上で、たとえばキーAに応じてすでにパーティション化されています。ClusterRepartitionコンポーネントを使用すると、割当て(カーディナリティも含む)を変更でき、ここでは、再パー

ティショナの後の割当ては、新しいキーBに応じてnode1、node2およびnode3になります。すべてが1つの手順で実行されます。再パーティショナが機能する仕組みを示す次のイメージを確認してください。

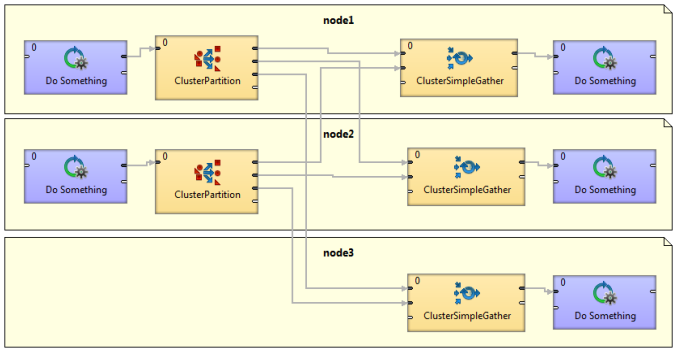


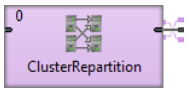
図59.2. 実行時のClusterRepartitionコンポーネントの実際の動作例

3つのグラフが、node1、node2およびnode3という3つのノードそれぞれで実行されます。ClusterRepartitionコンポーネントは、ソース・パーティションごとに1つのPartitionコンポーネント、およびターゲット・パーティションごとに1つのSimpleGatherコンポーネントで置き換えられます。このように、実際には全部で5つのコンポーネントがClusterRepartitionのかわりに処理を行います。各Partitionは、単一の入力パーティションからのデータをすべての出力パーティションに分割し、ここで、SimpleGatherコンポーネントによりデータが収集されます。

注意

このコンポーネントの使用方法の詳細は、**CloverETL Cluster**のドキュメントを参照してください。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	出力データ・レコード用	入力0[1)]

説明:

1): メタデータはこのコンポーネントを介して伝播できます。

第60章 データ品質

コンポーネントとは何かを理解していることを想定しています。概要は、第19章「コンポーネント」(p.97)を参照してください。

一部のコンポーネントは、データ品質の決定と保証に焦点を当てています。このコンポーネント・グループを**データ品質**と呼びます。

データ品質は、複数の異種タスクを実行するために使用します。

コンポーネントには様々なプロパティを設定できます。ただし、一部のものが共通する場合があります。すべてのコンポーネントに共通のプロパティもあれば、ほとんどのコンポーネントに共通のプロパティもあります。次のことを学習している必要があります。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

データ品質は、異種コンポーネント・グループであるため、共通のプロパティはありません。

データ品質グループの各コンポーネントは、実行するタスクに応じて識別できます。

- Address Doctor 5(p.764)は、住所書式を検証または修正します。
- EmailFilter(p.768)は、電子メール・アドレスを検証し、有効なアドレスを送信します。無効な電子メール・アドレスが含まれたデータ・レコードは、オプションの2番目の出力ポートを介して送信できます。
- ProfilerProbe(p.773)はコンポーネントを通過するデータの統計解析を実行します。

Address Doctor 5

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**データ品質**コンポーネントを見つけるには、データ品質の比較(p.332)を参照してください。

注意

このコンポーネントはトランスフォーマですが、**「Palette」→「Data Quality」**にあります。

要約

Address Doctor 5は、入力レコードの住所書式を検証、修正または完成させます。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
Address Doctor 5	-	いいえ	1	1-2	-	-

概要

Address Doctor 5は、その単一の入力ポートでレコードを受信します。次に、ユーザー定義の、住所書式への変換を実行します(住所書式の修正またはそのフィールドの完成など)。最後に、編集済の住所を最初の出力ポートに送信します。2番目の出力ポートは、変換に適合しなかったレコードにオプションで使用できます(標準のエラー・ポート)。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意1
出力	0	はい	変換されたデータ・レコード用	任意2
出力	1	いいえ	変換できなかったレコード用(エラー・ポート)	任意2

Address Doctor 5の属性

属性	必須	説明	可能な値
Basic			
Config file	1)	構成を定義する外部ファイル。	
Parameter file	1)	パラメータを定義する外部ファイル。	
Configuration	2)	住所データベースとその場所を指定します。	
Parameters	2)	変換の実行方法を制御します。	
Input mapping	はい	処理内容を決定します。	
Output mapping	はい	出力へのマッピング内容を制御します。	
Element item delimiter		住所全体が単一行に格納されている場合、この属性は住所フィールドを区切る特殊文字を指定します。	デリミタを使用しない(デフォルト) |; : # | \n \r\n clover_item_delimiterのいずれか

説明:

1): これら2つが設定されている場合、「**Configuration**」および「**Parameters**」を定義する必要はありません。

2): これら2つを設定するか、または「**Config file**」および「**Parameter file**」をかわりに定義します。

詳細説明

Address Doctor 5は、入力データおよび構成をサード・パーティのAddress Doctorライブラリに渡すことによって動作します。その後、コンポーネントは、ライブラリからの出力をCloverETLにマップします。

そのため、コンポーネントの操作がわからない場合には、公式のAddress Doctor 5ドキュメントを参照することをお薦めします。ここには、**Address Doctor 5**コンポーネントの詳細な構成に必要な情報が含まれています。CloverETLは、実際にはパラメータ設定用のGUIとして機能します。

コンポーネントの操作は、次のタスクの実行であるとみなすことができます。

- グラフへのAddress Doctorライブラリ(`*.jar`およびネイティブ・ライブラリ)の場所の通知。これらはJava引数として渡したり、Program Filesにコピーすることができます。ライブラリは各自で取得する必要があります。CloverETLには埋め込まれていません。
- 住所データベースの取得。
- コンポーネント属性の設定。Address Doctor 5の構成(p.765)を参照してください。

Address Doctor 5の構成

コンポーネントは、次の属性を設定する4つの基本手順で構成されます。

1. **Configuration**: 住所データベースの格納場所を指定します。データベースはいずれかのモード(BATCH_INTERACTIVEなど)で提供されるため、一致する**タイプ**を設定する必要があります(「**Parameters**」の**エンリッチメント・データベース・セット**にも適用)。データベースを操作するには、適切な**ロック解除コード**(汎用または固有)を取得する必要があります。

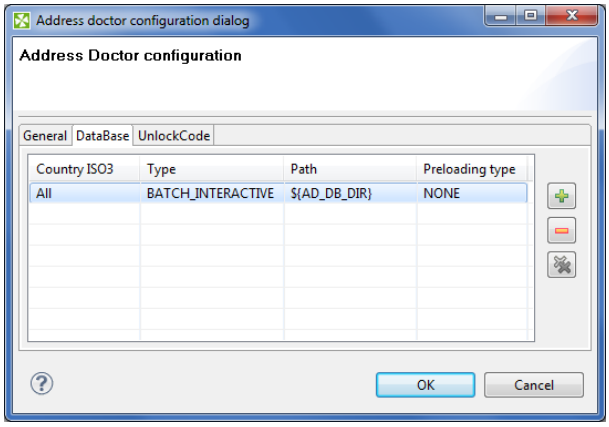


図60.1. データベース構成

2. **Parameters**: 実行される変換の内容を制御します。特定の設定は非常に固有なものであり、公式のAddress Doctor 5ドキュメントを参照する必要があります。

   たとえば、ダイアログの「**Process**」タブでは、様々な**エンリッチメント**を構成できます。これらを使用すると、住所書式の証明書を追加できます。証明書は、特定の住所書式が国の郵政機関の公式書式と一致していることを保証します。エンリッチメントの追加により、一般的にデータ処理が遅くなるため、オプションで追加データベースが必要となる場合があります。

3. **Input mapping**: 処理内容を決定します。3つの基本手順で設定できるウィザードを使用します。

   - 入力で許可されるすべてのAddress Doctor内部フィールド(メタデータ)から住所プロパティを選択します。フィールド名とともにカッコで囲まれた数が表示され、この数で、プロパティを形成できるフィールド(出力メタデータ)の数が示されます。たとえば、Street name (6)は、番地名が入力ファイルの最大6行に書き込まれる可能性があることを示します。

   - Address Doctorの内部マッピングを指定します。前の手順で選択した入力フィールドを「**Input mapping**」の使用可能フィールドにドラッグします。

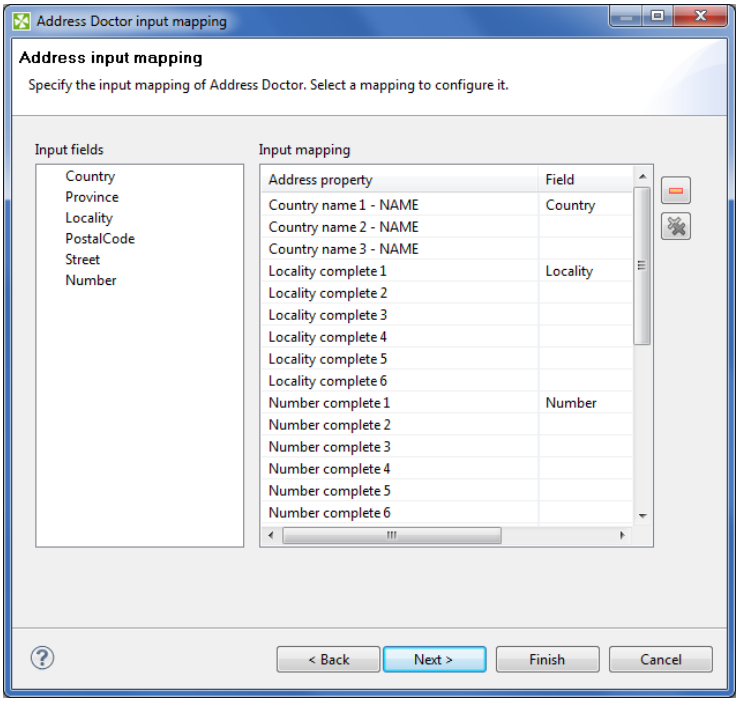


図60.2. Input mapping

   - 入力マッピングのサマリーを確認します。

4. **Output mapping**: ここでは、出力、つまり最初の出力ポートにマップする内容を決定します。オプションで、2番目のエラー・ポートにデータをマップできます(このようなマッピングを行わなかった場合、エラー・コードおよびエラー・メッセージが生成されます)。

   「**Input mapping**」と同様に、これらの手順を構成するわかりやすいウィザードを使用して、構成を実行します。

   - マッピングに使用する住所プロパティを選択します。
   - 特定の出力マッピングを指定します。これには、前に選択した内部フィールドを出力フィールドに割り当てることが必要です。「**Error port**」タブでは、コンポーネントで住所変換を実行できなかった場合に2番目の出力ポートに送信されるエラー出力(そのフィールド)の構造を設計します。

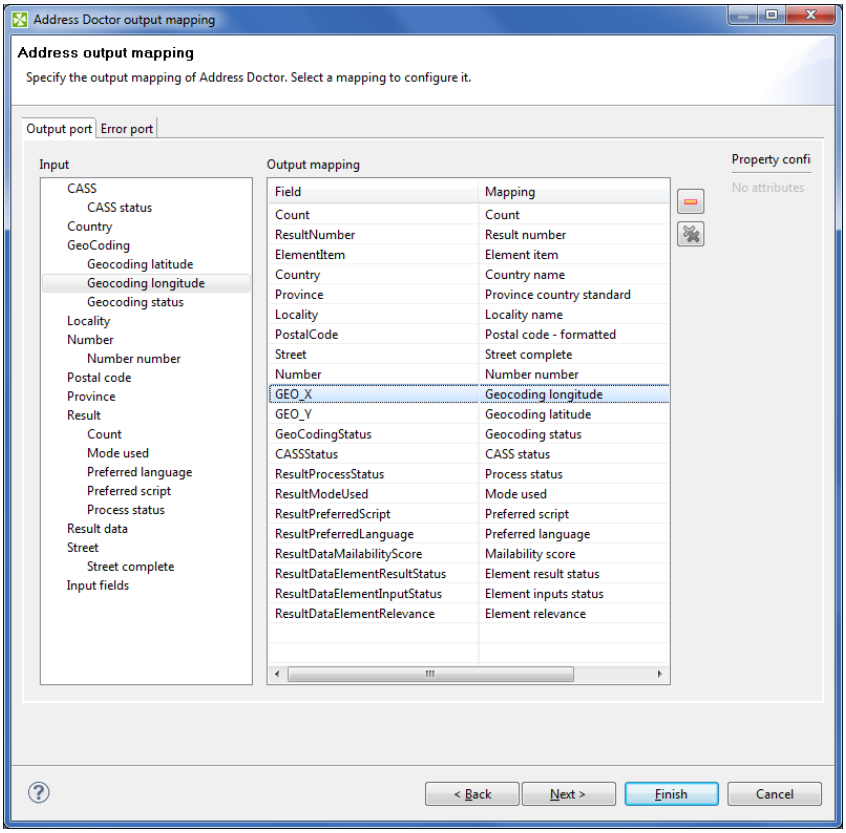


図60.3. Output mapping

   - 出力マッピングのサマリーを確認します。

コンポーネント操作の派生操作が、文字変換です。これは、たとえば、キリル文字で住所を入力してローマ字に変換できることを意味します。このタスクに追加のデータベースは必要ありません。

EmailFilter

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**データ品質**コンポーネントを見つけるには、データ品質の比較(p.332)を参照してください。

注意

このコンポーネントはトランスフォーマですが、「**Palette**」→「**Data Quality**」にあります。

要約

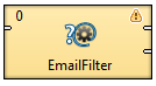
EmailFilterは、指定された条件に従って、入力レコードをフィルタリングします。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	Java	CTL
EmailFilter	-	いいえ	1	0-2	-	-

概要

EmailFilterは、入力ポートを介して受信レコードを受け取り、特定のフィールドについて有効な電子メール・アドレスであるかを検証します。有効として受け入れられたデータ・レコードは、オプションの最初の出力ポートが接続されている場合には、このポートを介して送信されます。拒否された入力の特定のフィールドは、オプションの2番目の出力ポートが他のコンポーネントに接続されている場合は、このポートを介して送信できます。オプションの2番目の出力ポートのメタデータには、エラー情報を含む最大2つの追加フィールドを含めることもできます。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	いいえ	有効なデータ・レコード用	入力0[1)]
	1	いいえ	拒否されたデータ・レコード用	任意[2)]

説明:

1): このコンポーネントを介してメタデータを伝播することはできません。

2): 出力ポート0のメタデータには、いずれかの入力データ・フィールドと最大2つの追加フィールドが含まれます。名前が入力メタデータ内のフィールド名と同じであるフィールドは、これらのフィールドの入力値を使用して値が入力されます。

表60.1. EmailFilterのエラー・フィールド

フィールド番号	フィールド名	データ型	説明
FieldA	**エラー・フィールド**の属性値	string	エラー・フィールド
FieldB	**ステータス・フィールド**の属性値	integer[1)]	ステータス・フィールド

説明:

1): 最も一般的なエラー・コードは次のとおりです。

- **0** `No error`: 電子メール・アドレスは受け入れられました。
- **1** `Syntax error`: 電子メール・アドレスの書式指定に準拠していない文字列は、すべてこのエラー・コードで拒否されます。
- **2** `Domain error`: アドレスに対するドメインの検証に失敗しました。ドメインが存在しないか、DNSシステムでメール交換サーバーを特定できません。
- **3** `SMTP handshake error`: SMTPレベルで、このエラー・コードは、指定されたドメインのメール交換サーバーが到達不可能であるか、または他の理由(サーバーがビジー状態など)で接続に失敗していることを示します。
- **4** `SMTP verify error`: SMTPレベルで、このエラー・コードは、VRFYコマンドにより無効であるとして、サーバーがアドレスを拒否したことを意味します。アドレスは正式に無効です。
- **5** `SMTP recipient error`: SMTPレベルで、このエラー・コードは、サーバーがアドレスの配信を拒否したことを意味します。
- **6** `SMTP mail error`: MAILレベルで、このエラーは、サーバーはテスト・メッセージの配信を受け入れたが、送信中にエラーが発生したことを示します。

EmailFilterの属性

属性	必須	説明	可能な値
Basic			
Field list	はい	含まれる値が有効な電子メール・アドレスであるか無効な電子メール・アドレスであるかを検証する必要がある、選択済入力フィールドの名前リスト。コロン、セミコロンまたはパイプで区切られたフィールド名のシーケンスによって表現します。	
Level of inspection		電子メール・アドレスの検証に使用する様々な方法を指定できます。各レベルは、その左側にある先行レベルを包含かつ拡張しています。詳細は、検査レベル(p.771)を参照してください。	SYNTAX | DOMAIN (デフォルト) | SMTP | MAIL
Accept empty		デフォルトでは、空のフィールドも有効なアドレスとして受け入れられます。`false`に設定すると、これをオフに切り替えることができます。詳細は、受入れ条件(p.771)を参照してください。	true (デフォルト) | false

属性	必須	説明	可能な値
Error field		エラー・メッセージを書き込むことのできる出力フィールドの名前(拒否されたレコードのみ)。	
Status field		エラー・コードを書き込むことのできる出力フィールドの名前(拒否されたレコードのみ)。	
Multi delimiter		個々のフィールド値を複数の電子メール・アドレスに分割する正規表現。空の場合、各フィールドは単一の電子メール・アドレスとして処理されます。	[,;] (デフォルト) | その他
Accept condition		デフォルトでは、たとえ1つでもフィールド値が有効な電子メール・アドレスとして検証されている場合、レコードは受け入れられます。`STRICT`に設定した場合、**フィールド・リスト**のすべてのフィールドのすべてのフィールド値が有効である場合にのみ、レコードは受け入れられます。詳細は、受入れ条件(p.771)を参照してください。	LENIENT (デフォルト) | STRICT
Advanced			
E-mail buffer size		バルク処理されるまでにメモリーに読み取られるレコードの最大数。詳細は、バッファおよびキャッシュ・サイズ(p.771)を参照してください。	2000 (デフォルト) | 1-N
E-mail cache size		キャッシュされる電子メール・アドレス検証結果の最大数。詳細は、バッファおよびキャッシュ・サイズ(p.771)を参照してください。	2000 (デフォルト) | 0 (キャッシュはオフ) | 1-N
Domain cache size		キャッシュされるDNS問合せ結果の最大数。`SYNTAX`レベルでは無視されます。	3000 (デフォルト) | 0 (キャッシュはオフ) | 1-N
Domain retry timeout (ms)		各DNS問合せ試行のミリ秒単位のタイムアウト。解決にかかるミリ秒単位の最大時間は、**Domain retry timeout**と**Domain retry count**の積に等しくなります。	800 (デフォルト) | 1-N
Domain retry count		失敗したDNS問合せの再試行回数。	2 (デフォルト) | 1-N
Domain query A records		デフォルトでは、SMTP標準に準拠して、MXレコードが見つからなかった場合はAレコードが検索されます。`false`に設定した場合、DNS問合せの速度は2倍になりますが、このSMTP標準は無視されます。	true (デフォルト) | false
SMTP connect attempts (ms,...)		接続およびHELOの試行。カンマで区切られた数のシーケンスで表現します。この数は個々の接続試行間の遅延を示します。	1000,2000 (デフォルト)
SMTP anti-greylisting attempts (s,...)		アンチグレーリスト機能。カンマ区切りの数のシーケンスによって表現された、試行回数および個々の試行間の遅延。空の場合、アンチグレーリストはオフになります。詳細は、SMTPグレーリスト試行(p.772)を参照してください。	30,120,240 (デフォルト)
SMTP retry timeout (s)		SMTPリクエストが失敗するまでの秒単位のTCPタイムアウト。	300 (デフォルト) | 1-N

属性	必須	説明	可能な値
SMTP concurrent limit		アンチグレーリストがオンの場合のパラレル・タスクの最大数。	10 (デフォルト) | 1-N
Mail From		MAILレベルで送信されたダミー・メッセージのFromフィールド。	CloverETL <clover@cloveretl.org> (デフォルト) | その他
Mail Subject		MAILレベルで送信されたダミー・メッセージのSubjectフィールド。	Hello, this is a test message (デフォルト) | その他
Mail Body		MAILレベルで送信されたダミー・メッセージのBody。	Hello,\nThis is CloverETL text message.\n\nPlease ignore and don't respond.Thank you, have a nice day!(デフォルト) | その他

詳細説明

バッファおよびキャッシュ・サイズ

「**E-mail buffer size**」を増やすと、単一の問合せでより多くのレコードが処理されるため、DNSシステムやSMTPサーバーに対する問合せが不要に繰り返されなくなります。一方、「**E-mail cache size**」を増やすと、キャッシュに格納されているアドレスを瞬時に検証できるため、パフォーマンスが向上します。ただし、どちらのパラメータも追加のメモリーを必要とするため、システムで割り当てられる最大値に設定してください。

受入れ条件

デフォルトでは、**フィールド・リスト**に指定した入力データ・レコードの空のフィールドも、有効な電子メール・アドレスとみなされます。「**Accept empty**」属性は、デフォルトではtrueに設定されています。より厳密にする場合は、この属性をfalseに切り替えることができます。

つまり、レコードは、1つ以上の有効な電子メール・アドレスがある場合に、受入れ対象とみなされます。

一方、「**Accept condition**」がSTRICTに設定されている場合は、**フィールド・リスト**にあるすべての電子メール・アドレス(「**Accept empty**」属性に応じて、空の値を含むまたは除外する)が有効である必要があります。

そのため、「**Accept empty**」属性と「**Accept condition**」属性を設定する場合には注意が必要です。フィールド・リストに指定されたフィールドに空のフィールドがあり、他の空以外の値がすべて無効なアドレスとして検証されている場合、「**Accept condition**」がLENIENTに設定され、かつ「**Accept empty**」がtrueに設定されていると、このようなレコードが受け入れられます。ただし、実際には、このようなレコードに有用かつ有効な電子メール・アドレスは含まれていないため、このようなレコードが受け入れられることを保証する空の文字列のみが含まれることになります。

検査レベル

1. **SYNTAX**

   検証の第1レベル(SYNTAX)では、電子メール表現の構文がチェックされ、厳密でない条件と国際文字(TLDを除く)の両方が許可されます。

2. **DOMAIN**

   検証の第2レベル(DOMAIN) (デフォルト)では、ドメインの妥当性を確認し、メール変換サーバーの情報を取得するために、DNSシステムに問い合せます。偽陰性レスポンスに対するパフォーマンス比率を最適化するために、「**Domain cache size**」、「**Domain retry timeout**」、「**Domain retry count**」および「**Domain query**

A records」の4つの属性を設定できます。DNSサーバーに送信される問合せの数は、「**Domain retry count**」属性で指定されます。このデフォルト値は2です。送信される個々の問合せ間の時間間隔は、「**Domain retry timeout**」でミリ秒単位で定義されます。デフォルトは800ミリ秒です。したがって、問合せが解決される際の全体的な時間は、「**Domain retry count**」と「**Domain retry timeout**」の積に等しくなります。問合せの結果はキャッシュできます。キャッシュされる結果の数は、「**Domain cache size**」で定義されます。デフォルトでは、3000個の結果がキャッシュされます。この属性を0に設定した場合、キャッシュはオフになります。また、MXレコードが見つからなかった場合にAレコードを検索するかどうも決定できます(「**Domain query A records**」)。デフォルトでは、`true`に設定されます。つまり、MXレコードが見つからなかった場合、Aレコードが検索されます。ただし、この属性を`false`に設定すると、オフに切り替えることができます。これで、検索速度を2倍にできますが、SMTP標準は無視されます。

3. **SMTP**

   検証の第3レベル(`SMTP`)では、SMTPサーバー接続が試行されます。試行回数および個々の試行間の時間間隔を指定する必要があります。これは、「**SMTP connect attempts**」属性を使用して定義します。この属性は、カンマで区切られた整数のシーケンスです。それぞれの数は、サーバー接続の2つの試行間の時間(秒)です。つまり、最初の数は最初の試行と2番目の試行の間隔を示し、2番目の数は2番目の試行と3番目の試行の間隔を示す、というようになります。デフォルト値は、試行が3回で、最初の試行と2番目の試行の時間間隔が1000ミリ秒、2番目の試行と3番目の試行の時間間隔が2000ミリ秒です。

   また、`SMTP`および`MAIL`レベルの**EmailFilter**コンポーネントでは、精度を向上させ、サーバー組込みのグレーリストによって発生する偽陰性を排除できます。グレーリストは、不明なホストに対する配信の拒否に基づいた非常に一般的なスパム対策技術の1つです。特定の期間(通常、1から5分)が経過した後、配信が再試行されたときに、ホストは既知になりグレーリストに載せられます(つまり、許可されません)。ほとんどのスパマーは、単純にパフォーマンスを高めるために、最初の失敗の後に配信を再試行しません。**EmailFilter**にはアンチグレーリスト機能があり、指定された回数および遅延で、失敗した各`SMTP/MAIL`テストを再試行します。最後の再試行が失敗した後でのみ、アドレスは無効とみなされます。

4. **MAIL**

   第4レベル(`MAIL`)では、すべてが正常に実行された場合に、指定した電子メール・アドレスにダミー・メッセージを送信できます。メッセージには、「**Mail From**」、「**Mail Subject**」および「**Mail Body**」のプロパティがあります。デフォルトでは、メッセージは`CloverETL <clover@cloveretl.org>`から送信され、その件名は`Hello, this is a test message`です。さらに、そのデフォルトの本文は、`Hello,\nThis is CloverETL test message.\n\nPlease ignore and don't respond.Thank you and have a nice day!`です。

SMTPグレーリスト試行

アンチグレーリスト機能を有効にするには、「**SMTP grey-listing attempts**」属性を指定します。デフォルト値は30,120,240です。これらの数は、4回試行でき、各試行の時間間隔が30秒(最初の試行と2番目の試行の間)、120秒(2番目の試行と3番目の試行の間)および240秒(3番目の試行と4番目の試行の間)であることを意味します。デフォルト値は、他のカンマ区切りの整数のシーケンスで変更できます。アンチグレーリストがオンになっている場合に実行されるパラレル・タスクの最大数は、「**SMTP concurrent limit**」属性で指定されます。このデフォルト値は10です。

ProfilerProbe

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**データ品質**コンポーネントを見つけるには、データ品質の比較(p.332)を参照してください。

コンポーネントは「**Palette**」→「**Data Quality**」にあります。

要約

ProfilerProbeは、入力データを分析(プロファイリング)します。コンポーネントはData Profilerアプリケーションの軽量バージョンで、CloverETL環境内に完全に統合されています。コンポーネントの大きな利点は、CloverETLソリューションの能力がデータ・プロファイリング機能と統合されていることです。このため、データ統合、データ・クレンジング、その他のETLタスクなど、非常に複雑なワークフローでもプロファイリングを利用できます。

ProfilerProbeは分離されたデータソースのプロファイリングのみに制限されず、様々なソースからのデータ(一般的なDB、フラット・ファイル、スプレッドシートなど)のプロファイリングに使用できます。ProfilerProbeは、CloverETLのリーダー(p.339)でサポートされているすべてのデータソースを処理できます。このため、このコンポーネントはData Profilerでのスタンドアロンの Cloverプロファイリング・ジョブ(`cpj`)と比較して大きな一歩であると考えられます。

注意

このコンポーネントを使用するには、個別のデータ・プロファイリング・ライセンスが必要です。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信	Java	CTL
ProfilerProbe	-	いいえ	1	1-n	はい	いいえ	いいえ

概要

ProfilerProbeは、その最初の入力ポートを経由するデータのメトリックを計算します。入力メタデータの各フィールドにどのメトリックを適用するかを選択できます。エッジに対するプローブとしてこのコンポーネントを使用すると、グラフに入力されるデータをより詳細に(統計的に)把握できます。

コンポーネントは、入力データの正確なコピーを出力ポート0に送信します(**SimpleCopy**として動作します)。つまり、グラフで**ProfilerProbe**を使用して、グラフのビジネス・ロジック自体に影響を及ぼすことなく、グラフに入力されるデータを調べることができます。

残りの出力ポートには、プロファイリングの結果、つまり個々のフィールドのメトリック値が含まれています。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	メトリックによって分析される入力データ・レコード	任意
出力	0	いいえ	入力データ・レコードのコピー	入力ポート0
	1-n	いいえ	個々のフィールド当たりのデータ・プロファイリングの結果	任意

ProfilerProbeの属性

属性	必須	説明	可能な値
Basic			
Metrics	1)	メタデータ・フィールドに対して計算する統計。スタンドアロンのProfilerジョブと同様、すべてのメトリックを適用できます。メトリックの詳細はここで確認してください。	すべてのメトリックのリスト
Output mapping	2)	ポート番号1から始めてプロファイリング結果を出力ポートにマップします。詳細説明(p.775)を参照してください。	
Advanced			
Metrics URL	1)	**メトリック**設定が含まれているProfilerジョブ・ファイル。	`*.cpj`
Output mapping URL	2)	**出力のマッピング**定義が含まれる外部XMLファイル。	
Processing mode		`Always active` (デフォルト): **ProfilerProbe**コンポーネントをローカルおよびリモート(サーバーで実行した場合)から実行するデフォルトのモード。 `Debug mode only`: デバッグ目的で実行データを取得するには、このモードを選択します。コンポーネント・エッジに対するデバッグ・モードに似ています。**ProfilerProbe**に対してこのモードを選択してグラフを実行するときは、次の点に注意してください。 サーバー`debug_mode = true`のときに想定どおりに動作します(サーバー・グラフ構成プロパティ。Clover Serverのドキュメントを参照してください)。 サーバー`debug_mode= false`の場合、入力データは最初の出力ポートを介して続行しますが、データのプロファイリングが後続の出力ポートに送信されません。	Always active (デフォルト) | Debug mode only
Persist results		サーバー環境では、プロファイリング結果はプロファイリング結果データベースにも格納されます。この属性をfalseに設定して、これをオフに切り替えることができます。	true (デフォルト) | false

説明

1) これらの1つの属性のみを指定します。(両方が設定されている場合は、**Metrics URL**が優先されます。)

2) これらの1つの属性のみを指定します。(両方が設定されている場合は、**Output mapping URL**が優先されます。)

詳細説明

- **Output mapping**: この属性を編集すると変換エディタ(p.286)が開き、ここでどのメトリックを出力ポートに送信するかを決定できます。

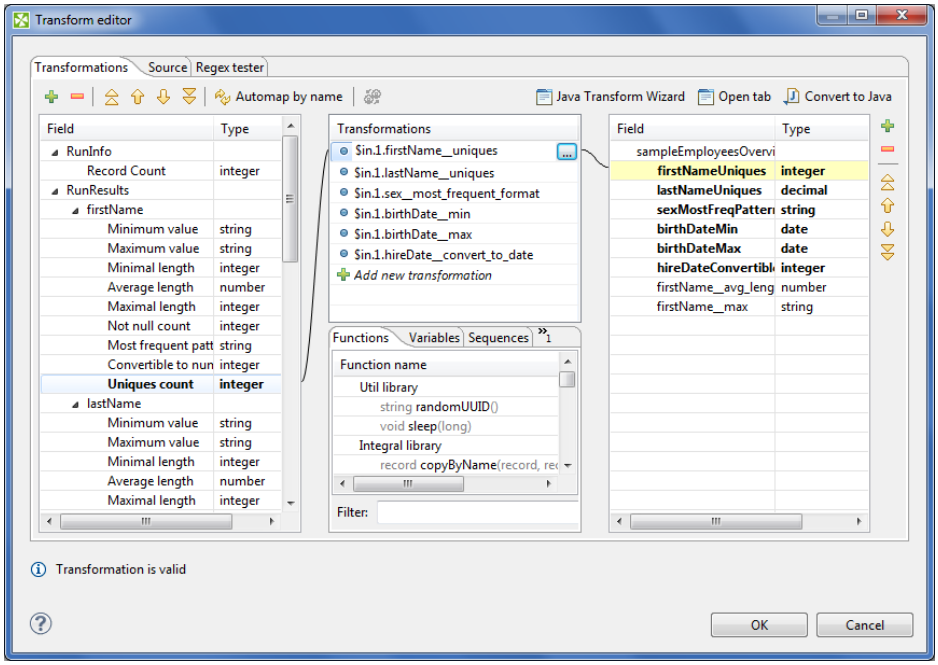


図60.4. ProfilerProbeの変換エディタ

ダイアログには、変換エディタおよびCTL(p.889)で既知のすべての機能が表示されます。また、左側のメタデータには特別な書式があります。「**Metrics**」属性を介して割り当てた入力フィールドおよびメトリックのツリーです。フィールドおよびメトリックは、`RunResults`レコードの下にグループ化されます。`RunResults`レコードの各フィールドには、`fieldName__metric_name`という特別な名前があります(アンダースコアはセパレータとして2倍の長さです)。たとえば、`firstName__avg_length`です。また、1つのフィールドが含まれているもう1つの特別なレコード(`inputRecordCount`)があります。グラフの実行後、フィールドにはコンポーネントによってプロファイリングされたレコードの合計数が格納されます。フィールド/メトリックを右クリックし、**Expand All**または**Collapse All** metricsを選択できます。

注意

ProfilerComponentは次のようにエラーを報告できます。

```
CTL code compilation finished with 1 errors
Error: Line 5 column 23 - Line 5 column 39: Field 'field1  avg length' does not exist in record
'RunResults'!
```

つまり、出力マッピングで無効なメトリックにアクセスしています。この例では、**平均長**はフィールド`field1`に対して有効ではありません。

マッピングを実行するための基本手順は次のとおりです。

1. すでにいくつかの出力メタデータがある場合は、左側のペインでメトリックを左クリックして出力フィールドにドラッグします。これでその特定のメトリックのプロファイリング結果が出力に送信されます。
2. 出力メタデータがない場合は、次の手順を実行します。
   a. 左側のペインから右側のペイン(空の領域)に**フィールド**をドラッグしてドロップします。

b. これにより、出力メタデータに新しいフィールドが生成されます。その書式は `fieldName__metric_name`です(アンダースコアはセパレータとして2倍の長さです)。たとえば、`firstName__avg_length`です。

c. ポート0を除く任意の出力ポートのフィールドにメトリックをマップできます。このポートは(影響を受けずにコンポーネントを通過する)入力データ用に予約されています。

注意

出力マッピングではCTLを使用します(「**Source**」タブに切り替えることができます)。様々な種類の関数を使用して、データの詳細を確認できます。例:

```
double uniques = $out.0.firstName__uniques; // conversion from integer
double uniqInAll = (uniques / $in.0.recordCount) * 100;
```

すべてのレコードで一意の最初の名前のパーセントを計算します。

- **メトリックのインポートおよび外部化**: **メトリック**のダイアログでは、フィールドおよびそのメトリックの設定をProfilerジョブ(*.cpj)ファイルに外部化したり、Profilerジョブ(*.cpj)ファイルからこの属性にインポートできます。この目的でダイアログの最下部に「**Import from .cpj**」と「**Externalize to .cpj**」の2つのボタンがあります。外部化された.cpjファイルは「**Metrics URL**」属性に使用できます。「**Externalize to .cpj**」アクションは、この属性に自動的に値を入力します。

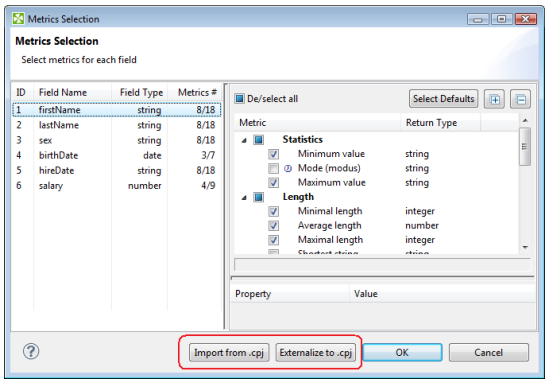


図60.5. メトリックのインポート/外部化ボタン

ProfilerProbeの注意事項と制限

この短い項では、**ProfilerProbe**コンポーネントの使用と`*.cpj`ジョブによるデータのプロファイリングとの主な違いについて説明します。

- これは、その入力エッジを経由するデータに対して分析を実行します。プロファイリング結果は出力ポートに送信されます。結果データベースはいっさい必要ありません。サーバー環境では、コンポーネントは結果をプロファイリング結果データベースにも送信します。このような結果は、CloverETL Data Profiler Reporting Consoleを使用してさらに詳しく表示できます。

- 「**Metrics URL**」属性を介してデータ・プロファイリング・ジョブ(`*.cpj`)を使用できます。

- 入力データのサンプリングを使用する場合は、**DataSampler** (または他のフィルタ)コンポーネントをグラフに接続します。**ProfilerProbe**に組込みのサンプリングはありません。

- クラスタ環境では、コンポーネントは実行中の各ノードからのデータをプロファイリングします。このため、結果は特定のノードで処理されるデータの一部にのみ適用可能です。すべてのノードからのデータのメトリックを計算する必要がある場合は、まずこのコンポーネントが実行される単一ノードにデータを収集します(たと

えば、ClusterSimpleGather(p.757)を使用します)。注意: コンポーネントが複数のノードで実行中である場合、プロファイリング結果データベースに複数の実行結果も生成されます。それぞれの結果は、各単一ノードで処理されるデータの一部にのみ適用可能です。このためクラスタ環境の場合は通常、**永続結果**機能をオフにすることをお薦めします。

第61章 その他

コンポーネントとは何かを理解していることを想定しています。概要は、第19章「コンポーネント」(p.97)を参照してください。

一部のコンポーネントは、前述のすべてのコンポーネントと少し異なります。コンポーネントのこのグループを**「その他」**と呼びます。

「その他」は、複数および異機種間のタスクの実行に使用できます。

コンポーネントには様々なプロパティを設定できます。ただし、一部のものが共通する場合があります。すべてのコンポーネントに共通のプロパティもあれば、ほとんどのコンポーネントに共通のプロパティもあります。次のことを学習している必要があります。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

「その他」はコンポーネントの異機種間グループであるため、共通プロパティがありません。

「その他」グループの各コンポーネントは、それぞれが実行するタスクに従って区別できます。

- SystemExecute(p.803)はシステム・コマンドを実行します。
- JavaExecute(p.791)はJavaコマンドを実行します。
- DBExecute(p.782)はデータベースに対してSQL/DML/DDL文を実行します。
- RunGraph(p.795)は指定の**CloverETL**グラフを実行します。
- HTTPConnector(p.786)はHTTPリクエストを送信し、Webサーバーからレスポンスを受信します。
- WebServiceClient(p.806)はWebサービスを呼び出し、レスポンスを出力ポートにマップします。
- CheckForeignKey(p.779)は外部キー値をチェックし、無効な値をデフォルト値で置き換えます。
- SequenceChecker(p.799)は入力データ・レコードがソートされているかどうかをチェックします。
- LookupTableReaderWriter(p.793)は参照表からデータを読み取り、参照表にデータを書き込みます。
- SpeedLimiter(p.801)はデータがコンポーネントを通過する速度を低下させます。

CheckForeignKey

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

CheckForeignKeyは、外部キー値の妥当性をチェックし、無効な値を有効な値で置き換えます。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1]	Java	CTL
CheckForeignKey	-	いいえ	2	1-2	-	いいえ	いいえ

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

概要

CheckForeignKeyは、2つの入力ポートを介してデータ・レコードを受信します。最初の入力ポート上のデータ・レコードが、2番目の入力ポート上のデータ・レコードと比較されます。指定した外部キー(入力ポート0)の一部の値が主キー(入力ポート1)の値内に見つからない場合は、その無効な値ではなくデフォルト値が外部キーに指定されます。その後、すべての外部レコードが新しい(修正済)外部キー値とともに最初の出力ポートに送信されます。また、外部キー値が無効な元の外部レコードは、オプションの2番目の出力ポートに送信できます(接続済の場合)。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	外部キーを持つデータ用	任意1
	1	はい	主キーを持つデータ用	任意2
出力	0	はい	修正済キーを持つデータ用	入力0[1]
	1	いいえ	無効なキーを持つデータ用	入力0[1]

説明:

1): このコンポーネントを介してメタデータを伝播することはできません。

CheckForeignKeyの属性

属性	必須	説明	可能な値
Basic			
Foreign key	はい	両方の受信データ・フローを比較し、データ・レコードを異なる出力ポートに分配するために比較されるキー。詳細は、Foreign Key(p.780)を参照してください。	
Default foreign key	はい	「**Foreign key**」のデータ型に対応する、セミコロンで区切られた値のシーケンス。無効な外部キー値の置換えに使用できます。詳細は、Foreign Key(p.780)を参照してください。	
Equal NULL		デフォルトでは、フィールドの値がnullのレコードは異なるとみなされます。`true`に設定した場合、nullは等しいとみなされます。	false (デフォルト) | true
Advanced			
Hash table size		キー値を格納するための表。キー値が一意であるレコードの数よりも大きくしてください。	512 (デフォルト) | properties
Deprecated			
Primary key		セミコロンで区切られた、2番目の入力ポートからのフィールド名のシーケンス。詳細は、Deprecated: 「Primary Key」(p.781)を参照してください。	

詳細説明

- **Foreign Key**

  「**Foreign key**」は、セミコロンで区切られた個々の割当てのシーケンスです。これらの各割当ては`$foreignField=$primaryKey`のようになります。

  「**Foreign key**」を定義するには、「**Foreign key definition**」ウィザードの「**Foreign key**」タブで目的のフィールドを選択する必要があります。左側の「**Fields**」ペインからフィールドを選択して、右側の「**Foreign key**」ペインに移動します。

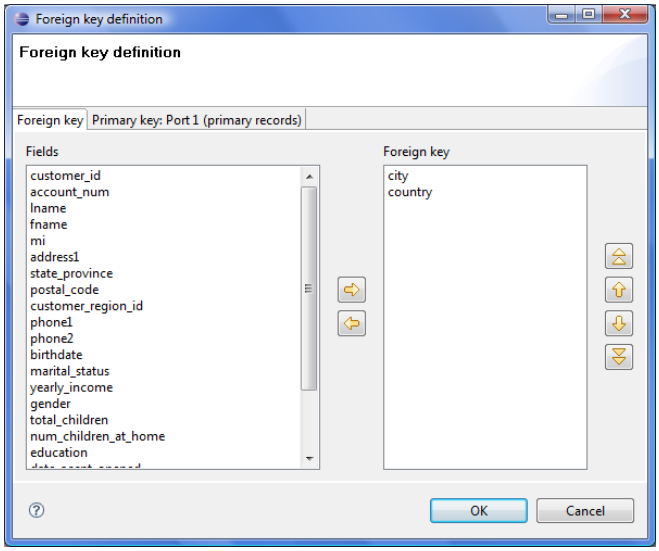


図61.1. 「Foreign key definition」ウィザード(「Foreign key」タブ)

「**Primary key**」タブに切り替えると、選択した外部フィールドが「**Foreign key definition**」ペインの「**Foreign key**」列に表示されます。

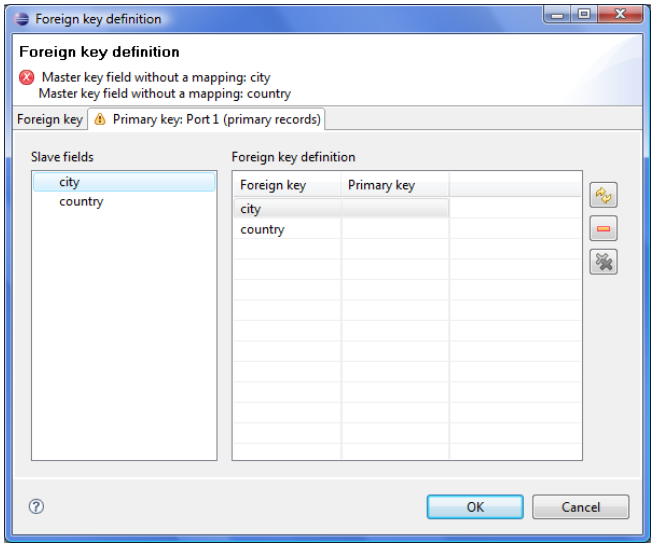


図61.2.「Foreign key definition」ウィザード(「Primary key」タブ)

左側のペインからいくつかの主フィールドを選択して、右側の「**Foreign key definition**」ペインの「**Primary key**」列に移動することのみが必要です。

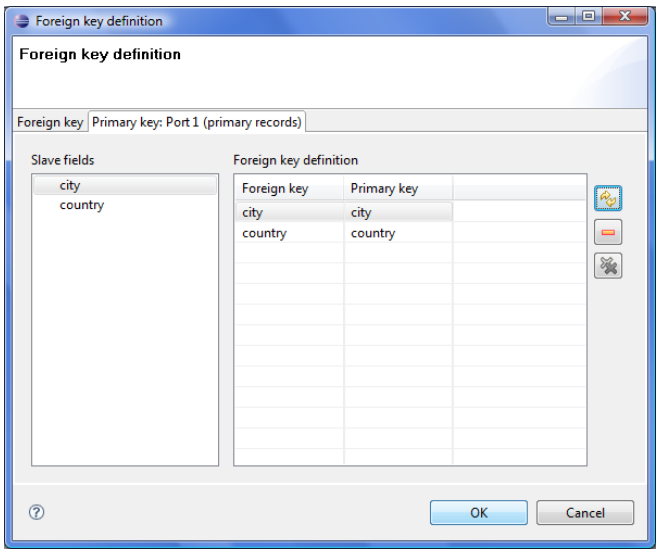


図61.3.「Foreign key definition」ウィザード(外部キーと主キーの割当て)

また、デフォルトの外部キー値(**Default foreign key**)も定義する必要があります。このキーも、セミコロンで区切られた、対応するデータ型の値のシーケンスです。数値およびデータ型は、外部キーのメタデータに対応している必要があります。

デフォルトの外部キー値を定義する場合は、「**Default foreign key**」属性行をクリックし、すべてのフィールドのデフォルト値を入力する必要があります。

- **Deprecated:「Primary Key」**

  Cloverの旧バージョンでは、「**Primary key**」属性と「**Foreign key**」属性を使用して主キーと外部キーの両方を指定する必要がありました。いずれも、書式はセミコロンで区切られたフィールド名のシーケンスでした。ただし、現在「**Primary key**」の使用は推奨されていません。

DBExecute

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

DBExecuteは、データベースに対してSQL/DML/DDL文を実行します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1]	Java	CTL
DBExecute	-	✖	0-1	0-2	-	✖	✖

[1]コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

概要

DBExecuteは、JDBCドライバを使用して接続されたデータベースに対して指定のSQL/DML/DDL文を実行します。問合せやトランザクションを実行し、ストアド・プロシージャまたはファンクションを呼び出すことができます。単一の入力ポートを介して入力パラメータを受信でき、出力パラメータまたは結果セットは最初の出力ポートに送信されます。エラー情報は、2番目の出力ポートに送信できます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	[1]	ストアド・プロシージャまたはSQLコマンド全体用の入力レコード	任意
出力	0	[2]	ストアド・プロシージャの出力パラメータまたは問合せの結果セット	任意
	1	✖	エラー情報用	入力メタデータに基づく[3]

[1]「**Query input parameters**」属性が指定されている場合または入力ポートを介してSQL問合せ全体が受信される場合は、入力ポートが接続されている必要があります。

[2]「**Query output parameters**」属性または「**Return set output fields**」属性が必須である場合は、出力ポートが接続されている必要があります。

[3]出力ポート1上のメタデータには、エラー情報に関する最大2つの追加フィールドとともに、入力からの任意の数のフィールド(同じ名前および型)が含まれていることがあります。入力メタデータは、その名前と型に基づいて自動的にマップされます。2つのエラー・フィールドには任意の名前を付けることができ、次の自動入力関数(p.132)の`ErrCode`および`ErrText`に設定する必要があります。

DBExecuteの属性

属性	必須	説明	可能な値
Basic			
DB connection	はい	使用されるDB接続のID。	
Query URL	1)	この2つのいずれかのオプション:「**SQL query**」属性で説明するのと同じ特性でSQL問合せを定義する外部ファイルの名前(パスを含む)、またはポートの読取りに使用される「**File URL**」属性文字列。詳細は、入力ポートから受信したSQL問合せ(p.784)を参照してください。	
SQL query	1)	グラフで定義されているSQL問合せ。データベースに対して実行されるSQL/DML/DDL文が含まれています。パラメータのあるストアド・プロシージャまたはファンクションを呼び出す場合、または出力データセットが生成される場合、文は*{[? =]call procedureName([?[,?,[...]])}*という形式にする必要があります。(文は必ず中カッコで囲みます。)また、入力パラメータや出力パラメータが必須である場合、その対応する属性を(それぞれ「**Query input parameters**」、「**Query output parameters**」、「**Result set output fields**」に)定義します。さらに、問合せが複数の文で構成されている場合は、指定の**SQL文デリミタ**で区切る必要があります。文は1つずつ実行されます。	
SQL statement delimiter		「**SQL query**」属性または「**Query URL**」属性内の個々のSQL文間のデリミタ。デフォルトのデリミタはセミコロンです。	";" (デフォルト) | その他の文字
Print statements		デフォルトでは、SQLコマンドは出力されません。`true`に設定した場合はstdoutに送信されます。	false (デフォルト) | true
Transaction set		トランザクションで文を実行するかどうかを指定します。詳細は、Transaction Set(p.784)を参照してください。データベースでトランザクションがサポートされている場合にのみ適用されます。	SET (デフォルト) | ONE | ALL | NEVER_COMMIT
Advanced			
Query source charset		「**Query URL**」属性に指定されている外部ファイルのエンコーディング。	ISO-8859-1 (デフォルト) | <other encodings>
Call as stored procedure		この属性を`true`に切り替えないかぎり、デフォルトではSQLコマンドはストアド・プロシージャ・コールとして実行されません。ストアド・プロシージャとして呼び出された場合は、`JDBC CallableStatement`が使用されます。	false (デフォルト) | true
Query input parameters		入力パラメータのあるストアド・プロシージャ/ファンクションを呼び出すときに使用します。これは、`1:=$inputField1;...;n:=$inputFieldN`というタイプのシーケンスです。各指定の入力フィールドの値が、対応するパラメータにマップされます(**SQL問合せ**でのパラメータの位置は指定された番号と等しくなります)。入力ポートを介してSQLコマンドが受信される場合、この属性は指定できません。	

属性	必須	説明	可能な値
Query output parameters		出力パラメータまたは戻り値のあるストアド・プロシージャ/ファンクションを呼び出すときに使用します。これは、`1:=$outputField1;...;n:=$outputFieldN`というタイプのシーケンスです。各出力パラメータ(**SQL問合せ**内でのその位置は指定された番号と等しくなります)の値が、指定されたフィールドにマップされます。ファンクションが値を返す場合、この値は最初のパラメータによって表されます。	
Result set output fields		ストアド・プロシージャまたはファンクションが一連のデータを返す場合、その出力は指定された出力フィールドにマップされます。属性は、セミコロンで区切られた出力フィールド名のシーケンスで表されます。	
Error actions		指定の問合せがSQL例外をスローしたときに実行するアクションの定義。変換の戻り値(p.283)を参照してください。	
Error log		指定した「**Error actions**」に対するエラー・メッセージを書き込むファイルのURL。設定しない場合は、**コンソール**に書き出されます。	

説明:

1): これらのいずれかを設定する必要があります。両方が指定された場合は、**Query URL**が優先されます。

詳細説明

入力ポートから受信したSQL問合せ

SQL問合せは入力ポートから受信することもできます。

この場合、「**Query URL**」属性の2つの値が許可されます。

- SQLコマンドは、入力エッジを介して送信されます。

  属性値は`port:$0.fieldName:discrete`です。

  このエッジのメタデータにはデフォルトのデリミタもレコード・デリミタもありませんが、**EOF as delimiter**を`true`に設定する必要があります。

- SQLコマンドが含まれているファイルの名前(パスを含む)は、入力エッジを介して送信されます。

  属性値は`port:$0.fieldName:source`です。

入力ポートからのデータの読取りの詳細は、入力ポートの読取り(p.303)を参照してください。

Transaction Set

オプションは次のとおりです。

- **1つの文**

  コミットは、各問合せの実行後に実行されます。

- **文の1つのセット**

  各入力レコードに対してすべての文が実行されます。コミットは、文のセット後に実行されます。

  このため、任意のレコードに対する任意の文の実行中にエラーが発生した場合、そのようなレコードに対してすべての文がロールバックされます。

- **すべての文**

  コミットは、すべての文の後にのみ実行されます。

  このため、エラーが発生した場合は、すべての操作がロールバックされます。

- **コミットなし**

  コミットは呼び出されません。

  異なるフェーズで他のコンポーネントから呼び出される可能性があります。

ヒントおよびポイント

- 通常、INSERT文およびSELECT文には**DBExecute**コンポーネントを使用しないでください。データベースへのデータのアップロードには、**DBOutputTable**コンポーネントを使用してください。ダウンロードの場合も、同様に**DBInputTable**コンポーネントを使用してください。

特定のケース

- データベース内でのデータの転送: 同じデータベースのある表から別の表へのデータのロードは、データベース内で行うことがベスト・プラクティスです。次のような問合せで**DBExecute**コンポーネントを使用できます。

  ```
  insert into my_table select * from another_table
  ```

  読取り中にデータを解析し、書込み時に書式を設定する必要があるため、データベースからデータを取得して格納する方が時間がかかります。

HTTPConnector

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

HTTPConnectorは、HTTPリクエストをWebサーバーに送信してレスポンスを受信します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1)]	Java	CTL
HTTPConnector	-	いいえ	0-1	0-2	-	いいえ	いいえ

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

概要

HTTPConnectorは、HTTPリクエストをWebサーバーに送信してレスポンスを受信します。リクエストはファイルまたはグラフ自体に書き込まれるか、単一入力ポートを介して受信されます。レスポンスは出力ポートへの送信、指定されたファイルへの保存または一時ファイルへの保存が可能です。ファイルへのパスは、指定した出力ポートに送信できます。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	コンポーネントの様々な属性の設定用	任意
出力	0	いいえ	レスポンス・コンテンツ、レスポンス・ファイル・パス、ステータス・コード、コンポーネント属性用など	任意
	1	いいえ	エラーの詳細用	任意

このコンポーネントにはメタデータ・テンプレート(p.275)があります。

HTTPConnectorの属性

属性	必須	説明	可能な値
Basic			
URL	[1)]	コンポーネントの接続先HTTPサーバーのURL。1つ以上のプレースホルダを`*{<field name>}`という形式で含めることができます。URL形式は、リモート・ファイルの読取り(p.299)を参照してください。`HTTP`、`HTTPS`、`FTP`および`SFTP`プロトコルがサポートされています。`http:(proxy://proxyHost:proxyPort)//www.domain.com`のようにして、プロキシ・サーバー経由で接続することもできます。	
Request method		リクエストのメソッド。	GET (デフォルト) | POST
Add input fields as parameters		入力エッジからの追加パラメータをURLに追加するかどうかを指定します。注意: パラメータが入力エッジから読み取られて問合せ文字列に追加される場合、これらのパラメータには特殊文字(?、@、:など)を含めることも可能です。このような文字を%表記に置き換えないでください。**HTTPConnector**はこれらを自動的にURLにエンコードします。この機能はClover 3.3-M3に導入され、下位非互換性を生じています。	false (デフォルト) | true
Add input fields as parameters to		入力フィールドを問合せ文字列に追加するかメソッド本体に追加するかを指定します。パラメータは、「**Request method**」がPOSTに設定されている場合のみ、メソッド本体に追加できます。	QUERY (デフォルト) | BODY
Ignored fields		パラメータとして追加されない入力フィールドを指定します。セミコロンで区切られた入力フィールドのリストが予期されます。	
Additional HTTP header properties		サーバーに送信されるリクエストの追加プロパティ。ダイアログを使用してこれを作成し、最終的な形式は`key=value`ペアがカンマで区切られたシーケンスとなり、シーケンス全体が中カッコで囲まれます。値は、`${fieldName}`表記法を使用してフィールドを参照できます。	
Multipart entities		マルチパート・エンティティとして`POST`リクエストに追加されるフィールドを指定します。フィールド名がエンティティ名として使用されます。セミコロンで区切られた入力フィールドのリストが予期されます。	
Request/response charset		入力/出力ファイルの文字エンコーディング。	ISO-8859-1 (デフォルト)| その他のエンコーディング
Request content		グラフに直接定義されるリクエスト・コンテンツ。	
Input file URL		単一HTTPリクエストの読取り元ファイルのURL。URLファイル・ダイアログ(p.69)を参照してください。	
Output file URL		HTTPレスポンスの書込み先ファイルのURL。URLファイル・ダイアログ(p.69)を参照してください。出力ファイルは自動的に削除されず、手動で削除するか、変換の一部として削除する必要があります。	
Append output		デフォルトでは、新しいレスポンスで古いレスポンスが上書きされます。この属性を`true`に切り替えると、新しいレスポンスは古いレスポンスに追加されます。出力ファイルにのみ適用されます。	false (デフォルト) | true
Input Mapping		コンポーネントの様々なプロパティ値を入力レコードからマッピングすることにより、これらのプロパティの設定を可能にします。	

属性	必須	説明	可能な値
Output Mapping		レスポンス・データ(コンテンツ、ステータス・コードなど)から出力レコードへのマッピングを可能にします。入力フィールドおよびエラーの詳細からの値のマッピング(**「Redirect error output」**が`true`に設定されている場合)も可能です。	
Error Mapping		エラー・メッセージの出力レコードへのマッピングを可能にします。入力フィールドおよび属性からの値のマッピングも可能です。	
Redirect error output		エラーの詳細の標準出力ポートへのリダイレクトを可能にします。	false (デフォルト) | true
Advanced			
Authentication method		使用する認証方式を指定します。	HTTP BASIC (デフォルト) | HTTP DIGEST | ANY
Username		サーバーに接続するために必要なユーザー名。	
Password		サーバーに接続するために必要なパスワード。	
OAuth Consumer key		サービスに関連付けられたコンシューマ・キー。**「OAuth Consumer secret」**とともに署名リクエストのアクセス・トークン(2-legged OAuth)を定義します。	
OAuth Consumer secret		サービスに関連付けられたコンシューマ・シークレット。**「OAuth Consumer key」**とともに署名リクエストのアクセス・トークン(2-legged OAuth)を定義します。	
Store HTTP response to file	[2]	この属性が`true`に切り替えられると、レスポンスは**「Prefix for response names」**属性で指定された接頭辞を持つ一時ファイルに書き込まれます。これらの一時ファイルへのパスは、**「Output Mapping」**を使用して取得できます。レスポンスの一時ファイルへの保存は、レスポンス本体が大きすぎて単一の文字列データ・フィールドに保存できない場合に必要です。一時ファイルは、グラフの終了後に自動的に削除されます(デバッグ・モードで実行していない場合)。	false (デフォルト) | true
Prefix for response files		HTTPレスポンスの各出力ファイルの名前に使用される接頭辞。この接頭辞に、識別番号が追加されます。	"http-response-" (デフォルト) | その他の接頭辞
Redirect error output		`true`である場合、エラーの詳細が**出力ポート1**ではなく**出力ポート0**に送信されます。	false (デフォルト) | true
Deprecated			
URL from input field	[1]	取得するターゲットURLを指定している文字列フィールドの名前。フィールド値には、プレースホルダを`*{<field name>}`という形式で含めることができます。URL形式は、「リモート・ファイルの読取り」の項を参照してください。`HTTP`、`HTTPS`、`FTP`および`SFTP`プロトコルがサポートされています。	
Input field	[2]	リクエスト・コンテンツの取得元となる、入力メタデータのフィールドの名前。文字列データ型である必要があります。複数HTTPリクエストに使用できます。	
Output field		レスポンスの送信先となる、出力メタデータのフィールドの名前。文字列データ型である必要があります。複数HTTPレスポンスに使用できます。	

[1] URLは、**「URL」**属性または**「URL from field」**属性のいずれかを設定するか、**「Input mapping」**内でマッピングすることによって指定する必要があります。

[2] レスポンスは、**「Output file URL」**で指定された出力ファイルまたは一時ファイル(**「Store reponse file URL to output field」**が`true`に設定されている場合)に保存できます。両方のオプションを使用することはできません。

詳細説明

- **Input mapping**: この属性を編集すると変換エディタ(p.286)が開き、ここで入力レコードを使用してどのコンポーネント属性を設定するかを決定できます。

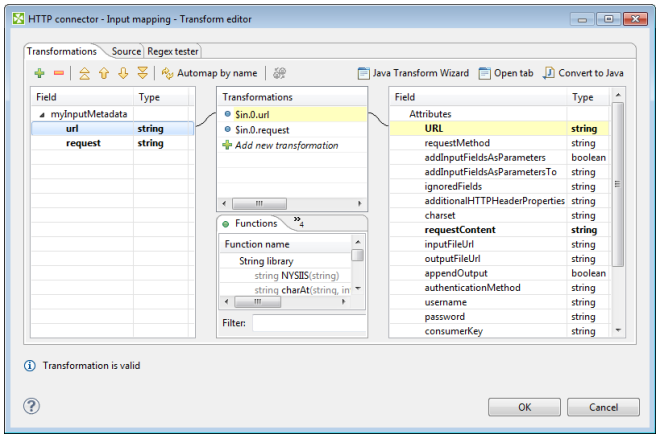


図61.4. HTTPConnectorの変換エディタ

ダイアログには、変換エディタおよびCTL(p.889)で既知のすべての機能が表示されます。

注意

様々な種類のCTL関数を使用して、使用する入力フィールド値を変更できます。

- **Output mapping**: この属性を編集すると変換エディタ(p.286)が開き、ここで何を出力ポートに送信するかを決定できます。

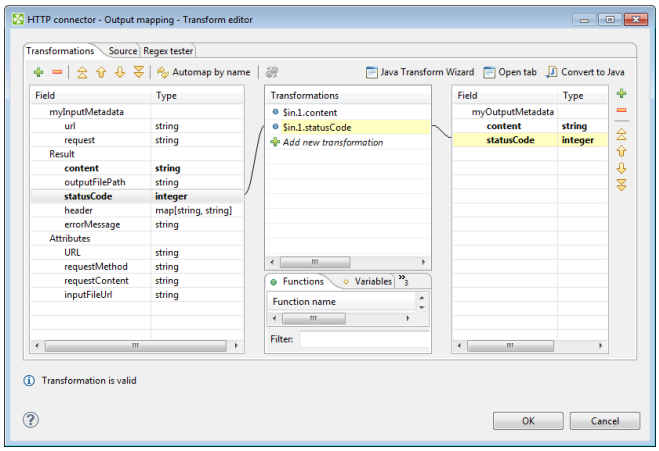


図61.5. HTTPConnectorの変換エディタ

ダイアログには、変換エディタおよびCTL(p.889)で既知のすべての機能が表示されます。

マッピングを実行するための基本手順は次のとおりです。

1. 出力メタデータがすでに存在する場合、左側のペインで項目を左クリックして出力フィールドにドラッグします。これで、結果データが出力に送信されます。
2. 出力メタデータがない場合は、次の手順を実行します。
   a. 左側のペインから右側のペイン(空の領域)に**フィールド**をドラッグしてドロップします。
   b. これにより、出力メタデータに新しいフィールドが生成されます。

様々なデータを出力ポートにマッピングできます。

- 入力メタデータからのフィールドの値: 入力フィールドからの値を出力ポートに送信できます。このことは、主に、HTTPリクエストになんらかの種類のセッション識別子を使用する場合に役立ちます。
- Result: 結果データを提供します。これらには、次が含まれます。
  - **content**: HTTPレスポンスのコンテンツ。レスポンスがファイルに書き込まれている場合は、このフィールドは`null`になります。
  - **outputFilePath**: レスポンスが書き込まれたファイルへのパス。レスポンスがファイルに書き込まれていない場合は、このフィールドは`null`になります。
  - **statusCode**: レスポンスのHTTPステータス・コード。
  - **header**: レスポンスからのHTTPヘッダー・プロパティを表すマップ。
  - **errorMessage**: エラー出力が標準出力ポートにリダイレクトされている場合のエラー・メッセージ。
- Attributes: コンポーネント属性の値を提供します。
  - **URL**: リクエストが送信された先のURL。
  - **requestMethod**: リクエストに使用されたメソッド。
  - **requestContent**: 送信されたリクエストのコンテンツ。
  - **inputFileUrl**: リクエスト・コンテンツが含まれているファイルのURL。

注意

出力マッピングではCTLを使用します(「**Source**」タブに切り替えることができます)。様々な種類の関数を使用して、出力フィールドに保存する値を変更できます。

```
$out.0.prices = find($in.1.content, "price: .*?USD")
```

`price: [任意のテキスト] USD`形式のすべての出現をレスポンス・コンテンツ内で見つけます。

- **Error mapping**: この属性を編集すると変換エディタ(p.286)が開き、ここでエラーの詳細を出力ポートにマッピングできます。この動作は、Output mappingとよく似ています。

注意

- v3.3.0-M3以降では、問合せパラメータとして使用されるフィールド値を**HTTPConnector**に渡す前のこれらのエンコードは不要になりました。これらは自動的にエンコードされます。ただし、これによって下位互換性が維持されなくなっているため、このことを理解しておいてください。
- v3.3.0-M3以降では、レスポンスがファイルに保存されている場合は(一時ファイルまたはユーザー指定のファイルのいずれに保存されていても)、**出力マッピング**を使用して出力ファイルへのパスを取得できます。ファイル・パスは出力ポートに自動的に送信されなくなりました(一時ファイルの場合)。

JavaExecute

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

JavaExecuteは、Javaコマンドを実行します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1)]	Java	CTL
JavaExecute	-	-	0	0	-	はい	いいえ

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

概要

JavaExecuteは、Javaコマンドを実行します。実行可能な変換はこのコンポーネントで必須であり、`JavaRunnable`インタフェースを実装し、その他の共通メソッドを`Transform`インタフェースから継承しています。共通Javaインタフェース(p.295)を参照してください。

`JavaRunnable`インタフェース・メソッドのリストを次に示します。詳細は、JavaExecute用Javaインタフェース(p.792)を参照してください。

アイコン

ポート

JavaExecuteには、入力ポートも出力ポートもありません。

JavaExecuteの属性

属性	必須	説明	可能な値
Basic			
Runnable	1)	グラフに定義されている、Javaで実行可能な変換	
Runnable URL	1)	実行可能な変換をJavaで定義する外部ファイル	

属性	必須	説明	可能な値
Runnable class	1)	外部の実行可能な変換クラス	
Runnable source charset		変換を定義する外部ファイルのエンコーディング	ISO-8859-1 (デフォルト)
Advanced			
Properties		Javaコマンドの実行時に使用されるプロパティ	

説明:

1) これらのいずれかを設定する必要があります。これらの変換属性は指定する必要があります。これらの変換属性は、すべてJavaRunnableインタフェースを実装します。

詳細は、JavaExecute用Javaインタフェース(p.792)を参照してください。

変換の詳細は、変換の定義(p.279)も参照してください。

JavaExecute用Javaインタフェース

実行可能な変換はこのコンポーネントで必須であり、JavaRunnableインタフェースを実装し、その他の共通メソッドをTransformインタフェースから継承しています。共通Javaインタフェース(p.295)を参照してください。

JavaRunnableインタフェースのメソッドを次に示します。

- `boolean init(Properties parameters)`

  javaクラス/関数を初期化します。このメソッドは、変換プロセスの開始時に1回のみ呼び出されます。すべてのオブジェクト割当ておよび初期化はここで実行する必要があります。

- `void free()`

  これは、このグラフ要素の初期化解除メソッドです。init()メソッドに割り当てられたすべてのリソースは、ここで解放されます。このメソッドは、要素の存在の最後に1回のみ起動されます。

- `void run()`

  JavaExecuteコンポーネントにより実行されるJavaコードの実装を保持する、コア・メソッドです。

- `void setGraph(TransformationGraph graph)`

  変換が実行される変換グラフ・インスタンスに渡されるメソッドです。TransformationGraphシングルトン・パターンは削除されたため、TransformationGraph.getInstance()を介してグラフのパラメータやその他の要素(メタデータ定義など)にアクセスできなくなりました。

LookupTableReaderWriter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

LookupTableReaderWriterは、参照表からのデータの読取り、または参照表へのデータの書込み、あるいはその両方を行います。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1]	Java	CTL
LookupTableReaderWriter	-	いいえ	0-1	0-n	はい	いいえ	いいえ

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

概要

LookupTableReaderWriterは、次のいずれかの方法で動作します。

- 接続済の単一入力ポートを介してデータを受信し、それを指定された参照表に書き込みます。
- 指定された参照表からデータを読み取り、すべての接続済出力ポートを介してそれを送信します。
- 接続済の単一入力ポートを介してデータを受信し、指定された参照表を更新し、更新された参照表を読み取ってすべての接続済出力ポートを介してデータを送信します。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	1)	参照表に書き込まれるデータ・レコード用	任意
出力	0-n	1)	参照表から読み取られるデータ・レコード用	入力0[1]

説明:

1): 少なくともこれらの1つが接続されている必要があります。入力ポートが接続されている場合、コンポーネントはこれを介してデータを受信し、そのデータを参照表に書き込みます。出力ポートが接続されている場合、コンポーネントは参照表からデータを読み取り、このポートを介してそのデータを送信します。

入力ポートが接続され、指定された**参照表**(LookupTableReaderWriterの属性(p.794)の項を参照)にコンポーネントが書き込むことができない場合、エラーが表示されます。

重要

データベース参照表への**書込み**はサポートされていません。かわりに、DBOutputTable(p.473)を使用する必要があります。

LookupTableReaderWriterの属性

属性	必須	説明	可能な値
Basic			
Lookup table	はい	次の用途で使用される参照表のID • レコードのソース(コンポーネントがリーダーとして使用されている場合) • 書込み先(コンポーネントがライターとして使用されている場合) • 両方(コンポーネントが読取りと書込みの両方に使用されている場合)	
Advanced			
Free lookup table after finishing		デフォルトでは、参照表のコンテンツはグラフの終了後に削除されません。`true`に設定すると、処理の終了後に参照表は空になります。	false (デフォルト) | true

RunGraph

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

RunGraphは、**CloverETL**グラフを実行します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1]	Java	CTL
RunGraph	-	いいえ	0-1	1-2	-	いいえ	いいえ

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

概要

RunGraphは、コンポーネント属性で名前が指定されたか、または入力ポートを介して名前を受信した**CloverETL**グラフを実行します。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	グラフ名およびCloverコマンドライン引数用	RunGraphの入力メタデータ(入力-出力モード)(p.796)
出力	0	はい	グラフ実行の情報用[1]	RunGraphの出力メタデータ(p.796)
	1	2)	失敗したグラフ実行用	RunGraphの出力メタデータ(p.796)

説明:

1): グラフがコンポーネント自体で指定されている場合は、指定されたグラフの成功した実行に関する情報が最初の出力ポートに送信され、グラフ(複数可)が入力ポート上で指定された場合は、成功および失敗の両方の情報がこのポートに送信されます。

2): 実行される単一のグラフの名前がコンポーネント自体で指定されている場合、2番目の出力ポートを接続する必要があります。指定されたグラフが失敗した場合にのみ、ここにデータ・レコードが送信されます。実行される1つ以上のグラフの名前を入力ポートを介して受信した場合、2番目の出力ポートを接続する必要はありません。成功および失敗の両方の情報は、最初の出力ポートにのみ送信されます。

表61.1. RunGraphの入力メタデータ(入力-出力モード)

フィールド番号	フィールド名	データ型	説明
0	<anyname1>	string	パスを含む、実行するグラフの名前
1	<anyname2>	string	Cloverコマンドライン引数。**警告:「The same JVM」**属性が`true`である場合、このフィールドに送信される引数は無視されます(RunGraphの属性(p.796)を参照してください)。

表61.2. RunGraphの出力メタデータ

フィールド番号	フィールド名	データ型	説明
0	graph	string	パスを含む、実行するグラフの名前
1	result	string	グラフ実行の結果(`Finished OK`、`Aborted`または`Error`)
2	description	string	グラフの失敗の詳細な説明
3	message	string	`org.jetel.graph.Result`の値
4	duration	integer、longまたはdecimal	グラフの実行に要した時間(ミリ秒)
5	runID	decimal	**CloverETL Server**上で実行するグラフ実行の識別情報

RunGraphの属性

属性	必須	説明	可能な値
Basic			
Graph URL	1)	パスを含む、このコンポーネントで実行されるグラフの名前。この場合、両方の出力ポートを接続する必要があり、成功または失敗の情報はそれぞれ最初の出力ポートまたは2番目の出力ポートに送信されます。(パイプライン・モード)	
The same JVM		デフォルトでは、指定されたグラフの実行に同じJVMインスタンスが使用されます。`false`に切り替えられると、グラフ(複数可)は外部プロセスとして実行します。サーバー環境で動作する場合は、この属性は常に`true`である必要があります(したがって、グラフ引数をポート0のフィールド1を介して渡すことはできません。ポート(p.795)を参照してください)。	true (デフォルト) | false
Graph parameters to pass		実行されるグラフに使用されるパラメータ。セミコロンで区切られたシーケンスで表します。**「The same JVM」**属性が`false`に切り替えられた場合は、この属性は無視されます。詳細は、詳細説明(p.797)を参照してください。	

属性	必須	説明	可能な値
Alternative JVM command	2)	外部プロセスを実行するためのコマンドライン。この**RunGraph**コンポーネントにより実行される各グラフに、より多くのメモリーを割り当てる場合は、指定されたいずれかのグラフで必要な最大メモリーに応じて、ここに`java -Xmx1g -cp`または同等のコマンドを入力します。	java -cp (デフォルト) | その他のjavaコマンド
Advanced			
Log file URL		パスを含む、外部プロセスのログが含まれるファイルの名前。ロギングは、「**The same JVM**」属性の値とは関係なく、指定されたファイルに行われます。「**The same JVM**」が`true`に設定されている場合(デフォルト設定)、ロギングはコンソールに対しても実行されます。これが`false`に切り替えられた場合、コンソールへのロギングは実行されず、ログ情報は指定されたファイルに書き込まれます。URLファイル・ダイアログ (p.69)を参照してください。	
Append to log file	2)	デフォルトでは、指定されたログ・ファイル内のデータは、各グラフが実行されるたびに上書きされます。	false (デフォルト) | true
Graph execution class	2)	グラフ(複数可)を実行するための完全なクラス名。	org.jetel.main.runGraph (デフォルト) | その他の実行クラス
Command line arguments	2)	グラフを実行するときに実行されるJavaコマンドの引数。	
Ignore graph fail		デフォルトでは、指定されたいずれかのグラフ(**RunGraph**が入力ポートを介して名前を受信)が失敗した場合、**RunGraph** (該当グラフを実行)に指定されたグラフも失敗します。この属性を`true`に設定すると、実行された各グラフの失敗は無視されます。**RunGraph** (他の1つのグラフを実行)のグラフがコンポーネント自体に指定されている場合も、成功情報が最初の出力ポートに送信され、失敗情報は2番目の出力ポートに送信されるため、これは無視されます。	false (デフォルト) | true

説明:

1): 入力ポートが接続されていない場合は指定する必要があります。

2): これらの属性は、「**The same JVM**」属性が`false`に設定されている場合のみ適用されます。

詳細説明

- パイプライン・モード

  コンポーネントがパイプライン・モードで動作している場合(入力エッジが存在せず「**Graph URL**」属性が指定されている)、「**Command line arguments**」属性には少なくとも`-plugins <plugins of CloverETL>`の形式で**CloverETL**プラグインが指定されている必要があります。

- 入力-出力モード

  コンポーネントが入力-出力モード(入力ポートが接続され「**Graph URL**」属性は空白)で動作している場合、プラグインを「**Command line arguments**」属性で指定する必要はありません。

- **コマンドライン引数の処理**

  RunGraphコンポーネントに渡されるすべてのコマンドライン引数(入力レコードの2番目のフィールドとして、またはcloverCmdLineArgsコンポーネント・プロパティとして)は、引用符で囲むことができる引数のスペース区切りリストとみなされます。また、引用符文字自体は、バックスラッシュによってエスケープできます。

  例61.1. 引用符で囲まれたコマンドライン引数の使用

  次の引数リストがあるとします。

  ```
  firstArgument "second argument with spaces" "third argument with spaces and \" a quote"
  ```

  子JVMに渡される結果のコマンドライン引数は次のようになります。

  1) `firstArgument`

  2) `second argument with spaces`

  3) `third argument with spaces and " a quote`

  2)では、実際には引数が引用符で囲まれていません。これにより、OS非依存のアプローチと、すべてのプラットフォーム上での円滑な実行が可能になります。

SequenceChecker

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

SequenceCheckerは、入力データ・レコードのソート順をチェックします。

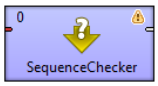
コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1)]	Java	CTL
SequenceChecker	-	いいえ	1	1-n	はい	いいえ	いいえ

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

概要

SequenceCheckerは、単一入力ポートを介してデータ・レコードを受信し、そのソート順をチェックします。これが指定された**ソート・キー**に対応しない場合、グラフは失敗します。ソート順が指定に対応する場合、データ・レコードはすべての接続済出力ポートに送信されます(オプション)。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0-n	いいえ	チェック済およびコピー済のデータ・レコード用	入力0[1)]

説明:

1): データ・レコードが適切にソートされている場合、これらは接続済出力ポートに送信されます。すべてのメタデータが同じである必要があります。メタデータはこのコンポーネントを介して伝播できます。

SequenceCheckerの属性

属性	必須	説明	可能な値
Basic			
Sort key	はい	レコードのソート基準となるキー。レコードが別の方法でソートされている場合、グラフは失敗します。詳細は、ソート・キー (p.277)を参照してください。	
Unique keys		デフォルトでは、「**Sort key**」の値は一意である必要があります。`false`に設定すると、「**Sort key**」の値は重複できます。	true (デフォルト) | false
Equal NULL		デフォルトでは、フィールドの値がnullのレコードは等しいとみなされます。`false`に設定すると、nullは異なるとみなされます。	true (デフォルト) | false
Deprecated			
Sort order		ソート順序(`Ascending`または`Descending`)。最初の文字(`A`または`D`)のみで指定できます。すべてのキー・フィールドに対して同様です。デフォルトのソート順は昇順です。レコードがこの方法でソートされていない場合、グラフは失敗します。	Ascending (デフォルト) | Descending
Locale		国際化が`true`に設定されている場合に使用されるロケール。デフォルトでは、`defaultProperties`ファイルに指定されている「**Locale**」の値がコメント解除され、必要な**ロケール**に設定されていないかぎり、システム値が使用されます。`defaultProperties`で**ロケール**を変更する方法の詳細は、デフォルトのCloverETL設定の変更(p.88)を参照してください。	システム値または指定されたデフォルト値(デフォルト) | その他のロケール
Use internationalization		デフォルトでは、国際化は使用されません。`true`に設定すると、各国のプロパティに応じたソートが実行されます。	false (デフォルト) | true

SpeedLimiter

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

SpeedLimiterは、これを通過するデータ・レコードを低速化します。

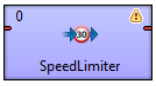
コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1]	Java	CTL
SpeedLimiter	-	いいえ	1	1-n	はい	いいえ	いいえ

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

概要

SpeedLimiterは、その単一入力ポートを介してデータを受信し、指定されたミリ秒数で各入力レコードを遅延させ、各入力レコードをすべての接続済出力ポートにコピーします。合計遅延は、出力ポート数に依存しません。これは、入力レコード数のみに依存します。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	はい	入力データ・レコード用	任意
出力	0	はい	コピーされたデータ・レコード用	入力0[1]
	1-n	いいえ	コピーされたデータ・レコード用	入力0[1]

説明:

1): SimpleCopy(p.647)とは異なり、出力のメタデータは入力のメタデータと同じである必要があります。すべてのメタデータが同じである必要があります。メタデータはこのコンポーネントを介して伝播できます。

SpeedLimiterの属性

属性	必須	説明	可能な値
Basic			
Delay	はい	各入力レコード処理の遅延。デフォルトではミリ秒単位ですが、その他の時間単位(p.275)も使用できます。解析の合計遅延は、この値と入力レコードとの乗算結果と等しくなります。	0-N

ヒントおよびポイント

Speedlimiterを使用する場合は、そのバッファがいっぱいになった後にのみ、レコードがコンポーネントから送信されます(デフォルト)。次のことを実行することが必要となる場合があります。

1. **Speedlimiter**にレコードを送信します。
2. 指定された秒数、遅延します。
3. そのレコードを出力ポートにすぐに送信します。

このような場合、**Speedlimiter**の出力エッジを「**Direct fast propagate**」に変更する必要があります。詳細は、エッジのタイプ(p.100)を参照してください。

SystemExecute

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

SystemExecuteは、システム・コマンドを実行します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1)]	Java	CTL
SystemExecute	-	いいえ	0-1	0-1	-	いいえ	いいえ

1) コンポーネントは、各データ・レコードをすべての接続済出力ポートに送信します。

概要

SystemExecuteは、コンポーネント自体で指定されたコマンドと引数を、独立したプロセスとして実行します。コマンドは、入力ポートを介して標準入力を受信し、出力ポートに標準出力を送信します(コマンドが出力を作成した場合)。

アイコン

ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	指定されたシステム・コマンドの標準入力用(プロセスの入力)	任意1
出力	0	1)	指定されたシステム・コマンドの標準出力用(プロセスの出力)	任意2

説明:

1): 標準出力は、出力ポートまたは出力ファイルに書き込まれる必要があります。両方の出力ポートが接続され、出力ファイルが指定されている場合、出力は出力ポートにのみ送信されます。

SystemExecuteの属性

属性	必須	説明	可能な値
Basic			
System command	はい	システムにより実行されるコマンド。コマンドは、常にスクリプトとして一時ファイルに保存されます。インタプリタが指定されている場合、これがそのスクリプトを実行します。コマンドに入力が必要な場合、オプションの入力ポートを介してコマンドに送信する必要があります。詳細は、動作の仕組み(p.805)を参照してください。	
Process input/output charset		システム・プロセス入出力のデータの書式設定や解析に使用されるエンコーディング。	ISO-8859-1 | <その他>
Output file URL	1)	出力エッジが接続されておらず、**システム・コマンド**が標準出力を作成する場合に、プロセスの出力(およびエラー)が書き込まれるファイルのパスを含む名前。詳細は、URLファイル・ダイアログ(p.69)を参照してください。	
Append		デフォルトでは、出力ファイルのコンテンツは常に削除され、新しいデータで上書きされます。`true`に設定すると、新しい出力データは出力ファイルに追加され、ファイルのコンテンツは削除されません。	false (デフォルト) | true
Command interpreter		コマンドを実行するインタプリタ。指定されている場合、**システム・コマンド**はスクリプトとして一時ファイルに保存され、このインタプリタにより実行されます。`<interpreter name> [parameters] ${} [parameters]`という形式にする必要があります。インタプリタが定義されると、**システム・コマンド**は一時ファイルに保存されてスクリプトとして実行されます。このような場合、コンポーネントはこの`${}`式をこの一時スクリプト・ファイルの名前で置き換えます。	
Working directory		コンポーネントの作業ディレクトリ。	現在のディレクトリ(デフォルト) | その他のディレクトリ
Advanced			
Number of error lines		コマンドがエラーで終了した場合に表示される行数。	2 (デフォルト) | 1-N
Delete tmp batch file		**重要:** この属性は、一時領域管理の競合により、Clover v. **3.3.0.M3**から削除されました。 デフォルトでは、作成された一時バッチ・ファイルは、コマンドの実行後に削除されます。`false`に設定すると、削除されません。	true (デフォルト) | false
Environment		変数から値へのシステム依存マッピング。マッピングは、コロン、セミコロンまたはパイプにより区切られます。デフォルトでは、新しい値が現在のプロセスの環境に追加されます。Both `PATH=/home/user/mydir`および`PATH=/home/user/mydir!true`は、`/home/user/mydir`が既存の`PATH`に追加されることを意味します。一方、`PATH=/home/user/mydir!false`では、古い`PATH`値が新しい値(`/home/user/mydir`)で置き換えられます。	例: PATH=/home/user/mydir[!true] (デフォルト) | PATH=/home/user/mydir!false

属性	必須	説明	可能な値
Timeout for producer/consumer workers (ms)		タイムアウト。デフォルトはミリ秒単位ですが、他の時間単位(p.275)も使用できます。詳細は、タイムアウト(p.805)を参照してください。	0 (制限なし) | 1-n
Ignore exit value		実行されたシステム・コマンドが0以外の値を返す場合、コンポーネントは失敗します。このオプションを使用すると、この動作を変更でき、終了値を無視できます。	true | false (デフォルト)

説明:

1): 出力ポートが接続されていない場合、標準出力は指定された出力ファイルに書き込むことのみができます。出力ポートが接続されている場合、出力ファイルは作成されず、標準出力は出力ポートに送信されます。

詳細説明

動作の仕組み

SystemExecuteは、「**System command**」で指定されたコマンドを実行し、2つのスレッドを作成します。

- 最初のスレッド(プロデューサ)ではレコードを入力エッジから読み取ってシリアライズし、コマンドの`stdin`に送信します。
- 2番目のスレッド(コンシューマ)は、コマンドの`stdout`を読み取り、それを解析して出力エッジに送信します。

タイムアウト

- コマンドが終了しても、コンポーネントはプロデューサおよびコンシューマの動作の終了を待機します。その時間は、「**Timeout**」属性で定義されます。
- デフォルトでは、タイムアウトは無制限です。予期しないデッドロックが発生する場合は、タイムアウトを任意のミリ秒数に設定できます。

WebServiceClient

商用コンポーネント

次で説明されている内容をすでに学習済であることを想定しています。

- 第41章「すべてのコンポーネントの共通プロパティ」(p.266)
- 第42章「多くのコンポーネントの共通プロパティ」(p.275)

目的に適した**その他**のコンポーネントを見つけるには、「その他」の比較(p.331)を参照してください。

要約

WebServiceClientは、Webサービスを呼び出します。

コンポーネント	同じ入力メタデータ	ソート済入力	入力	出力	全出力に送信[1)]	Java	CTL
WebServiceClient	-	いいえ	0-1	0-n	いいえ	いいえ	いいえ

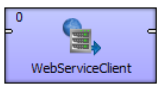
1) コンポーネントは、定義されたマッピングに従って処理済のデータ・レコードを接続済の出力ポートに送信します。

概要

WebServiceClientは、受信したデータ・レコードをWebサービスに送信し、レスポンスを出力ポート(接続済の場合)に渡します。**WebServiceClient**では、ドキュメント/リテラルスタイルのみがサポートされています。

WebServiceClientでは、ドキュメント・スタイル・バインディングとリテラルの使用(ドキュメント/リテラル・バインディング)によるSOAP (バージョン1.1および1.2)メッセージング・プロトコルのみがサポートされています。

アイコン



ポート

ポート・タイプ	番号	必須	説明	メタデータ
入力	0	いいえ	リクエスト用	任意1(In0)
出力	0-N	いいえ[1)]	これらのポートにマッピングされたレスポンス用	任意2(Out#)

説明:

1): 出力ポートが接続されていない場合、レスポンスを出力ポートに送信する必要はありません。

WebServiceClientの属性

属性	必須	説明	可能な値
Basic			
WSDL URL	はい	コンポーネントの接続先WSDサーバーのURL。`http:(proxy://proxyHost:proxyPort)//www.domain.com`のようにして、プロキシ・サーバー経由で接続することもできます。	
Operation name	はい	実行する操作の名前。詳細説明(p.808)を参照してください。	
Request Body structure	はい	入力ポートから受信されるリクエスト、またはグラフに直接記述されるリクエストの構造。リクエスト生成の詳細は、詳細説明(p.808)を参照してください。	
Request Header structure		「**Request Body structure**」のオプション属性。指定されない場合は、自動生成が無効化されます。リクエスト生成の詳細は、詳細説明(p.808)を参照してください。	
Response mapping		成功レスポンスの出力ポートへのマッピング。**XMLExtract**の場合と同じマッピングです。詳細は、XMLExtractのマッピング定義(p.429)を参照してください。	
Fault mapping		失敗レスポンスの出力ポートへのマッピング。**XMLExtract**の場合と同じマッピングです。詳細は、XMLExtractのマッピング定義(p.429)を参照してください。	
Namespace bindings		カスタム名前空間を定義する、一連の名前と値の割当て。	例: `weather = http://ws.cdyne.com/WeatherWS/`
Use nested nodes		trueの場合、ツリー内の深さに関係なく、同じ名前のすべての要素がマッピングされます。詳細説明(p.808)の例を参照してください。	true (デフォルト) | false
Advanced			
Username	1)	サーバーへの接続時に使用するユーザー名。	
Password	1)	サーバーへの接続時に使用するパスワード。	
Auth Domain	1)	認証ドメイン。設定されていない場合、NTLM認証スキームが無効化されます。DigestおよびBasic認証方式には影響しません。	
Auth Realm	1)	指定された資格証明の適用先認証レルム。空白のままにすると、資格証明はすべてのレルムに使用されます。NTLM認証スキームには影響しません。	
Timeout (ms)		リクエストのタイムアウト。デフォルトはミリ秒単位ですが、他の時間単位(p.275)も使用できます。	
Override Server URL		WSDL定義で指定されたURLのかわりにリクエストに使用するURLを指定します。	
Override Server URL from field		WSDL定義で指定されたURLのかわりにリクエストに使用するURLが含まれるフィールドを指定します。	

属性	必須	説明	可能な値
Disable SSL Certificate Validation		trueの場合、コンポーネントはSSL接続の証明書検証の問題を無視します。	true | false (デフォルト)

[1] 詳細は、認証(p.809)を参照してください。

詳細説明

- サーバーにリクエストを送信した後、**WebServiceClient**は最大10分間レスポンスを待機します。ない場合、コンポーネントはエラーで失敗します。
- ログ・レベル(p.85)をDEBUGに切り替えた場合、すべてのSOAPリクエストおよびレスポンスをログ内で調査できます。このことは、開発および問題調査の目的で役立ちます。
- 「**Operation name**」を指定すると、次に示すダイアログが開き、WS操作をダブルクリックするのみで選択できます。入力メッセージのドキュメント・スタイルをサポートしない操作には、赤色のエラーアイコンが表示されます。

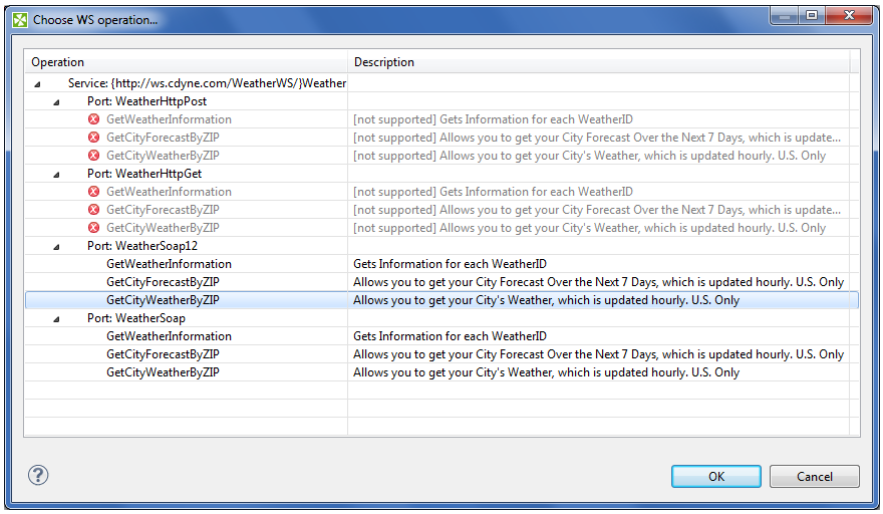


図61.6. WebServiceClientでのWS操作の選択

- 「**Request Body structure**」および「**Request Header structure**」では、リクエスト構造を示すダイアログが開きます。「**Generate**」ボタンにより、選択した操作用に定義されたスキーマに基づいたリクエスト・サンプルが生成されます。このボタンのドロップダウン・メニューで「**Customized generation...**」オプションを選択すると、生成されたリクエスト・サンプルのカスタマイズを支援するダイアログが開き、適切な要素のみの選択や要素のサブタイプの選択が可能になります。

例61.2.「Use nested nodes」の例

マッピング

```
<?xml version="1.0" encoding="UTF-8"?>
<Mappings>
 <Mapping element="request">
  <Mapping element="message" outPort="0" />
 </Mapping>
</Mappings>
```

適用先

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
 <message>msg1</message>
 <operation>
  <message>msg2</message>
 </operation>
</request>
```

次が生成されます。

- msg1およびmsg2 (**「Use nested nodes」**がオンの場合(デフォルト動作))
- msg1 (**「Use nested nodes」**がオフの場合)。msg2を抽出するには、明示的に<Mapping>タグを作成する必要があります(ネストされた各要素に対して1つずつ)。

認証

Webサービス・サーバーで認証が必要な場合、**「Username」**、**「Password」**、さらにNTLM認証の場合は**「Auth Domain」**の各コンポーネント・プロパティを設定する必要があります。

現在、NTLM、DigestおよびBasicの3つの認証スキームがサポートされています。NTLMが最も安全であり、Basicはこれらの中で最も安全度が低い方式です。サーバーはサポートしている認証方式を通知し、**WebServiceClient**は自動的に最も安全な方式を選択します。

BasicまたはDigest認証スキームが選択された場合は、**「Auth Realm」**を使用して、指定された資格証明が必要なレルムにのみ適用されるように制限できます。

注意

NTLM認証では、**「Auth Domain」**が必要です。これが設定されていない場合、DigestおよびBasic認証スキームのみが有効化されます。

サーバーでNTLM認証が必要であるにもかかわらず**「Auth Domain」**が空白のままである場合、グラフ実行ログに次のようなエラーが示されます。

- ERROR [Axis2 Task] - Credentials cannot be used for NTLM authentication: org.apache.commons.httpclient.UsernamePasswordCredentials
- ERROR [WatchDog] - Node WEB_SERVICE_CLIENT0 finished with status: ERROR caused by: org.apache.axis2.AxisFault: Transport error: 401 Error: Unauthorized

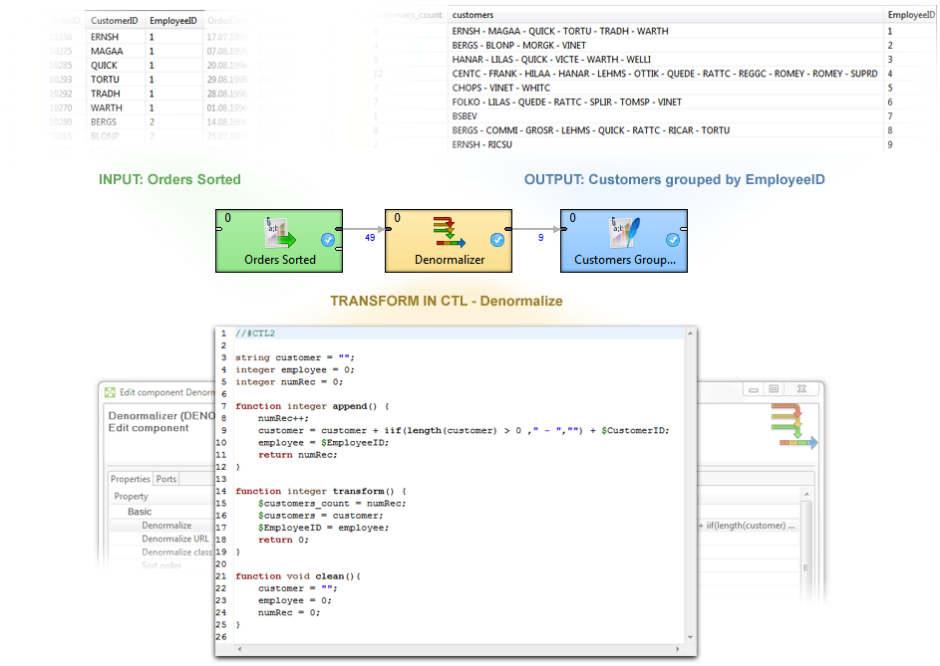
また、ドメインは、一部の慣例のように、**ユーザー名**の一部としてdomain\usernameの形式で指定することはできません。ドメイン名は、**「Auth Domain」**コンポーネント・プロパティで個別に指定する必要があります。

第IX部 CTL: CloverETL Transformation Language

第62章 概要

CTLは、**CloverETL**の変換コンポーネントにおけるデータ処理のための、専用のスクリプト言語です。

データの処理方法のシンプルで理解しやすい表記法を実現すると同時に、データを操作する十分な手段を提供するように設計されています。



言語の構文はJavaと同様であり、一部の構成はスクリプト言語で一般的なものです。CTLはスクリプト言語自体ではありますが、関数のコード編成は、コード・エントリ・ポイントを指定する、明確に定義されたメソッドを持つJavaクラスの構造と同様です。

CTLは、処理データの抽象化およびその実行環境の両方において高レベルの言語です。この言語は、プログラマに対して全体的なデータ変換の複雑性を隠蔽し、プログラマは、処理されるすべてのデータ・レコードに適用可能な一連の操作として、単一の変換ステップを開発することに集中できます。

また、プログラマにとっては、この言語が**CloverETL**環境と緊密に統合されていることにより、実行するコンポーネントの外にあるデータ変換の要素に対する一律のアクセス、レコード・フィールドで使用可能な型の値の演算、および検証や操作の一連の豊富な機能などのメリットもあります。

変換の実行中に、CTLコードを実行している各コンポーネントは個別のインタプリタを使用しているため、大量のパラレル・マルチスレッドが存在する実行環境で発生する可能性がある衝突を回避できます。

CTLの基本機能

1. **簡単なスクリプト言語**

   Clover Transformation Language (CTL)は非常にシンプルなスクリプト言語であり、数多くの**CloverETL**コンポーネントで変換の記述に使用できます。

   これらのすべてのコンポーネントではJavaを使用できますが、CTLを使用すると、作業が非常に簡単になります。

2. **数多くのCloverETLコンポーネントでの使用**

   CTLは、変換の概要(p.282)で概要が示されているすべてのコンポーネントで使用できます(**JMSReader**、**JMSWriter**、**JavaExecute**および**MultiLevelReader**を除きます)。

3. **Javaの知識が必要でない**

   Javaの知識がないユーザーでも、CTLでコードを記述できます。

4. **Javaと同等の速度**

   CTLで記述された変換は、Javaで記述された変換とほとんど同じ速度です。

   CTL2のソース・コードは、Javaクラスにコンパイルすることもできます。

CTLの2つのバージョン

バージョン3.0以降の**CloverETL**では、ユーザーは2つのうちいずれかのバージョンのCTLで変換コードを記述できます。

以降の章および項では、両バージョンのCTLの詳細な説明と、それぞれの組込み関数のリストを示します。

CTL1リファレンスおよび組込み関数

- 言語リファレンス(p.829)
- 関数リファレンス(p.859)

CTL2リファレンスおよび組込み関数

- 言語リファレンス(p.890)
- 関数リファレンス(p.919)

注意

CTL2バージョンのClover Transformation Languageの使用をお薦めします。

第63章 CTL1とCTL2の比較

CTL2は、**CloverETL** Transformation Languageの新しいバージョンです。CTLの概念に多くの改良が加えられています。

表63.1. CTLバージョンの比較

機能	CTL1	CTL2
強い型定義	✖	✔
インタプリタ・モード	✔	✔
コンパイル・モード	✖	✔
速度	低速	高速

型定義された言語

CTLは、CTL2で強い型定義として再設計されました。変数宣言にすでに型情報が含まれていることによって、型チェックの導入は、一般的に、コンテナ・タイプおよび関数に影響を及ぼします。CTL2では、コンテナ・タイプ(listsおよびmaps)では要素タイプを宣言する必要があり、ユーザー定義関数では戻り型と引数の型を宣言する必要があります。

関数の厳密な型定義では、値を返さない関数には`void`型を導入する必要があり、型定義によって、ローカル・コードと組込みライブラリにおける関数のオーバーロードも導入されます。

任意のコード順序

CTL2では、コードの任意の場所で変数および関数を宣言できます。唯一、満たす必要がある条件は、各変数および関数を使用する前に宣言する必要があることです。

また、CTL2では、変換の任意の場所でマッピングを定義し、その後に他のコードを続けることができます。

CTL2コードの各部分は、ほとんど任意で自由に配置できます。

コンパイル・モード

CTL2コードは、変換の処理速度が大幅に向上するPure Javaに変換できます。これはコンパイル・モードと呼ばれ、CTL2コードを使用してグラフを実行するたびに、**CloverETL**で透過的に実行できます。コンパイルされた形式への変換は内部で実行されるため、これを使用するためにJavaでプログラムを記述する必要はありません。

CTL2変換でコンポーネントを使用し、コンパイル・モードで動作するように明示的に設定すると、常に、**CloverETL**によってCTLからメモリー内Javaコードが生成され、ネイティブにJavaが実行されるため、処理速度が大幅に向上します。

グラフ要素へのアクセス(参照、シーケンス、...)

厳密な型チェックは、参照表およびシーケンス・アクセスの検証にまでさらに拡張されています。参照表では、詳細な型チェックで表によって返されるレコードを使用して、参照操作の実際の引数が参照表キーに対して検証されます。

シーケンスは、ユーザーによって明示的に設定された3つの使用可能な戻り型(integer、longおよびstring)をサポートしています。CTL1レコードでは、参照表およびシーケンスはそのIDで識別されていましたが、CTL2では名前で定義されます。そのため、これらのグラフ要素の名前は常にグラフ内で固有にしておく必要があります。

メタデータ

CTL2では、メタデータはすべてデータ型とみなされます。コードで直接メタデータ名を使用して変数を宣言できるため、レコードをCTL変換コードで宣言する方法が変わります。

```
Employee tmpEmployee;
recordName1 myRecord;
```

次の表に、両方のCTLバージョン間の相違点の概要を示します。

表63.2. CTLバージョンの相違点

CTL1	CTL2
ヘッダー(インタプリタ・モード)	
//#TL	//#CTL2
//#CTL1	
ヘッダー(コンパイル・モード)	
使用不可	//#CTL2:COMPILED
プリミティブ変数の宣言	
int	integer
bytearray	byte
コンテナ変数の宣言	
list myList;	<element type>[] myList; 例: integer[] myList;
map myMap;	map[<type of key>, <type of value>] myMap; 例: map[string,boolean] myMap;
レコードの宣言	
record (<metadataID>) myRecord;	<metadataName> myRecord;
関数の宣言	
function fName(arg1,arg2) { <functionBody> }	function <data type> fName(<type1> arg1,<type2> arg2) { <functionBody> }
マッピング演算子	
$0.field1 := $0.field1;(':='と'='に注意)	$0.field1 = $0.field1;(':='と'='に注意)
入力レコードへのアクセス	
@<port No>	使用不可。次のもので置換可能: $<port No>.*
@<metadata name>	使用不可。次のもので置換可能: $<metadata name>.*

CTL1	CTL2
フィールド値へのアクセス	
`@<port No>[<field No>]`	使用不可。次のもので置換可能: `$<port No>.<field name>`
`@<metadata name>[<field No>]`	使用不可。次のもので置換可能: `$<metadata name>.<field name>`
`<record variable name>["<field name>"]`	`<record variable name>.<field name>`
条件付き失敗式(インタプリタ・モードのみ)	
`$0.field1 := expr1 : expr2 : ...: exprN;`	`$0.field1 = expr1 : expr2 : ...: exprN;`
使用不可	`myVar = expr1 : expr2 : ...: exprN;`
使用不可	`myFunction(expr1 : expr2 : ...: exprN)`
ディクショナリ宣言	
定義は不要	常に定義が必要
ディクショナリ・エントリ型	
string、readable.channel、writable.channel	boolean、byte、date、decimal、integer、long、number、string、readable.channel、writable.channel、object
ディクショナリへの書込み	
署名: `void write_dict(string name, string value)`	構文: `dictionary.<entry name> = value;`
例1: `write_dict("customer", "John Smith");`	例: `dictionary.customer = "John Smith";`

例2:
`string customer;`
`write_dict(customer, "John Smith");`	

署名: `boolean dict_put_str(string name, string value);`	
例3: `dict_put_str("customer", "John Smith");`	

例4:
`string customer;`
`dict_put_str(customer, "John Smith");`	

ディクショナリからの読取り	
署名: `string read_dict(string name)`	構文: `value = dictionary.<entry name>;`

例1:
`string myString;`
`myString = read_dict("customer");`
	例:
	`string myString;`
	`myString = dictionary.customer;`

例2:
`string myString;`
`string customer;`
`myString = read_dict(customer);`	

CTL1	CTL2
署名: `string dict_get_str(string name)`	

例3:
`string myString;`
`myString = dict_get_str("customer");`	

例4:
`string myString;`
`string customer;`
`dict_get_str(customer);`	

参照表関数	
`lookup_admin(<lookup ID>,init)`[1)]	使用不可
`lookup(<lookup ID>,keyValue)`	`lookup(<lookup name>).get(keyValue)`
`lookup_next(<lookup ID>)`	`lookup(<lookup name>).next()`
`lookup_found(<lookup ID>)`	`lookup(<lookup name>).count(keyValue)`
`lookup_admin(<lookup ID>,free)`[1)]	使用不可
シーケンス関数	
`sequence(<sequence ID>).current`	`sequence(<sequence name>).current()`
`sequence(<sequence ID>).next`	`sequence(<sequence name>).next()`
`sequence(<sequence ID>).reset`	`sequence(<sequence name>).reset()`
switch文	

`switch (Expr) {`
`  case (Expr1) : { StatementA StatementB }`
`  case (Expr2) : { StatementC StatementD }`
`  [default : { StatementE StatementF }]`
`}`
	`switch (Expr) {`
	`  case Const1 : StatementA StatementB break;`
	`  case Const2 : StatementC StatementD break;`
	`  [default : StatementE StatementF]`
	`}`

forループ	

`int myInt;`
`for(Initialization;Condition,Iteration)`
(初期化、条件および繰返しが必要)
	`for(integer myInt;Condition;Iteration)`
	(初期化、条件および繰返しはオプション)

foreachループ	

`int myInt;`
`list myList;`
`foreach(myInt : myList) Statement`
	`integer[] myList;`
	`foreach(integer myInt : myList) Statement`

エラー処理	
`string MyException;` `try Statement1 catch(MyException) [Statement2]`	使用不可
次のオプション関数のセットはCTL1とCTL2の両方で使用可能:	
`<required template function>OnError()` (transformOnError()など)	
ジャンプ文	
`break`	`break;`

CTL1	CTL2
`continue`	`continue;`
`return Expression`	`return Expression;`
包含演算子	

`myVar .in. myContainer`	`in(myVar,myContainer)`
	または
	`myVar.in(myContainer)`

eval関数	
`eval()`	使用不可
`eval_exp()`	使用不可
三項演算子	

使用不可
ただし
`iif(Condition,ExprIfTrue,ExprIfFalse)`
がかわりに使用可能
	`Condition ?ExprIfTrue : ExprIfFalse`
	ただし
	`iif(Condition,ExprIfTrue,ExprIfFalse)`
	も可能

説明:

1) これらの関数は**CloverETL**のバージョン3.0以上では機能しないため、コードから削除できます。

第64章 CTL1からCTL2への移行

CTL1で記述された変換コードをCTL2に移行する場合、次の手順を実行する必要があります。

手順1: ヘッダーの置換

変換コードの最初にあるヘッダーを置換します。

解釈モードの場合、CTL1では//#TLまたは//#CTL1のいずれかを使用しますが、CTL2では//#CTL2を使用します。

また、CTL1では使用できないコンパイル・モードも選択できます。この場合、CTL2のヘッダーは//#CTL2:COMPILEDになります。

CTL1	CTL2
インタプリタ・モード	
//#TL	//#CTL2
//#CTL1	
コンパイル・モード	
使用不可	//#CTL2:COMPILED

手順2:プリミティブ変数の宣言の変更(integer、byte、decimalデータ型)

どちらのCTLバージョンも同じ変数を使用しますが、一部についてはキーワードが異なります。

- integerデータ型は、CTL1ではintという語を使用して宣言しますが、CTL2ではintegerとして宣言します。
- byteデータ型は、CTL1ではbytearrayという語を使用して宣言しますが、CTL2ではbyteとして宣言します。
- decimalデータ型については、CTL1での宣言にLengthおよびScaleが含まれる場合があります。

  たとえば、decimal(15,6) myDecimal;はCTL1で有効な小数の宣言で、Lengthが15に等しく、Scaleが6に等しくなります。

  CTL2では、LengthまたはScaleをdecimalの宣言で使用できません。

  デフォルトでは、小数の変数は最大で32桁に加えて、その値に含める小数点を使用できます。

  このような小数がエッジに送信され、そのメタデータでLengthおよびScaleが定義されている場合にのみ(デフォルトは8と2)、精度または長さを変更できます。

  したがって、これと同等の(CTL2での)宣言は次のようになります。

  ```
  decimal myDecimal;
  ```

  小数フィールドは、メタデータでこれらの**Length**および**Scale**を定義します。または、デフォルトの8と2をそれぞれ使用します。

CTL1	CTL2
integerデータ型	
int myInt;	integer myInt;
byteデータ型	
bytearray myByte;	byte myByte;

CTL1	CTL2
decimalデータ型	
`decimal(15,6) myDecimal;`	`decimal myDecimal;`
	このような変数を`decimal`フィールドに割り当てる場合、フィールドで**Length**および**Scale**をそれぞれ15と6に定義しておく必要があります。

手順3: 構造化変数の宣言の変更(list、map、recordデータ型)

- それぞれの`list`は、同じデータ型の要素で構成される一様構造です。

  `list`は、CTL1では次のように宣言されます。

  ```
  list myListOfStrings;
  ```

  CTL2では、これと同等の`list`宣言は次のようになります。

  ```
  string[] myListOfStrings;
  ```

  CTL2での`list`の宣言は、次の構文を使用します。

  ```
  <data type of element>[] identifier
  ```

  そのため、CTL1の`list`の宣言をCTL2で有効な別の宣言で置換します。

- それぞれの`map`は、キーと値のペアで構成される一様構造です。キーは常に`string`データ型で、値はプリミティブ・データ型です(CTL1の場合)。

  マップの宣言は次のようになります。

  ```
  map myMap;
  ```

  CTL1とは異なり、CTL2では、キーを`string`に加えて他のプリミティブ・データ型にもできます。

  そのため、CTL2では、次のようにキーの型と値の型の両方を指定する必要があります。

  ```
  map[<key type>, <value type>] myMap;
  ```

  マップの宣言をCTL1の構文からCTL2の構文に記述しなおすには、CTL1の古い宣言を置き換えます。

  ```
  map myMap;
  ```

  これをCTL2の新しい宣言で置換します。

  ```
  map[string, <value type>] myMap;
  ```

  たとえば、`map[string, integer] myMap;`などです。

- それぞれの`record`は、指定された数のフィールドで構成される異種構造です。個々のフィールドをそれぞれ異なるプリミティブ・データ型にできます。フィールドごとに名前と番号を持ちます。

  CTL1では、各レコードを3つの異なる方法で宣言できます。

  これらのうちの2つはメタデータIDを使用し、3つ目はポート番号を使用します。

  CTL1 (メタデータをIDで識別)とは異なり、CTL2ではメタデータを一意の名前で識別します。

CTL1およびCTL2でのレコードの宣言方法は、次の表を参照してください。

CTL1	CTL2
リストの宣言	
`list myList;`	`<element type>[] myList;`
	例: `string[] myList;`
マップの宣言	
`map myMap;`	`map[<type of key>,<type of value>] myMap;`
	例: `map[string,integer] myMap;`
レコードの宣言	
`record (<metadata ID>) myRecord;`	`<metadata name> myRecord;`
`record (@<port number>) myRecord;`	
`record (@<metadata name>) myRecord;`	

手順4: 関数の宣言の変更

1. 戻りデータ型を関数の宣言に追加します。(CTL2では`void`戻り型も使用できます。)
2. 引数のデータ型を追加します。
3. `void`以外のデータ型を返す関数は、それぞれ`return`文で終了する必要があります。必要に応じて、対応する`return`文を追加します。

次の表を参照してください。

CTL1	CTL2

`function transform(idx) {`
`  <other function body>`
`  $0.customer := $0.customer;`
`}`
	`function integer transform(integer idx) {`
	`  <other function body>`
	`  $0.customer = $0.customer;`
	`  return 0;`
	`}`

手順5: レコードの構文の変更

CTL1では、入力レコードおよびフィールドの割当てに@記号を使用しました。

CTL2では、別の構文を使用する必要があります。次の表を参照してください。

CTL1	CTL2
レコード全体	
`<record variable name> = @<port number>;`	`<record variable name>.* = $<port number>.*;`
`<record variable name> = @<metadata name>`	`<record variable name>.* = $<metadata name>.*;`
個々のフィールド	
`@<port number>[<field number>]`	`$<port number>.<corresponding field name>`
`@<metadata name>[<field number>]`	`$<metadata name>.<corresponding field name>`
`<record variable name>["<field name>"]`	`<record variable name>.<field name>`

注意

Rollupで使用するgroupAccumulatorの構文も変更する必要があります。

CTL1	CTL2
groupAccumulator["<field name>"]	groupAccumulator.<field name>

手順6: ディクショナリ構文でのディクショナリ関数の置換

CTL1では、ディクショナリ関数のセットを使用できます。

CTL2では、ディクショナリ構文を定義し、使用する必要があります。

CTL1	CTL2
ディクショナリへの書込み	
write_dict(string <entry name>, string <entry value>);	dictionary.<entry name> = <entry value>;
dict_put_str(string <entry name>, string <entry value>);	
ディクショナリからの読取り	
string myVariable; myVariable = read_dict(<entry name>);	string myVariable; myVariable = dictionary.<entry name>;
string myVariable; myVariable = dict_get_str(<entry name>);	

例64.1. ディクショナリの使用例

CTL1	CTL2
ディクショナリへの書込み	
write_dict("Mount_Everest", "highest");	dictionary.Mount_Everest = "highest";
dict_put_str("Mount_Everest", "highest");	
ディクショナリからの読取り	
string myVariable; myVariable = read_dict("Mount_Everest");	string myVariable; myVariable = dictionary.Mount_Everest;
string myVariable; myVariable = dict_get_str("Mount_Everest");	

手順7: 必要な箇所へのセミコロンの追加

CT1では、jump、continue、breakまたはreturnの各文の終わりにセミコロンを使用できない場合があります。

CTL2ではセミコロンが必要です。

そのため、必要に応じて、jump、continue、breakまたはreturnの各文の終わりにセミコロンを追加します。

手順8: 組込みCTL関数の確認、置換または名前変更

一部のCTL1関数は、CTL2では使用できません。関数リファレンス(p.919)でCTL2関数のリストを確認してください。

例:

CTL1関数	CTL2関数
`uppercase()`	`upperCase()`
`bit_invert()`	`bitNegate()`

手順9: switch文の変更

`switch`文の`case`部分にある式を定数で置換します。

CTL1では、*式*を`switch`文の`case`部分で使用でき、各`case`部分の後には中カッコが必要です。1つ以上の式の値が互いに等しい場合もあり、このような場合はすべての文が実行されます。

CTL2では、`switch`文の`case`部分で定数を使用する必要があり、各`case`部分の後に中カッコは使用できず、また、各`case`部分の最後に`break`文が必要です。`break`文を使用しないと、その下のすべての文が実行されます。個々の`case`部分に指定する定数は、それぞれ異なっている必要があります。

CTL バージョン	switch文の構文

CTL1	`//#CTL1`
	
	`int myInt;`
	`int myCase;`
	
	`myCase = 1;`
	
	`// Transforms input record into output record.`
	`function transform() {`
	
	`  myInt = random_int(0,1);`
	
	`  switch(myInt) {`
	`  case (myCase-1) : { print_err("two"); print_err("first case1"); }`
	`  case (myCase) : { print_err("three"); print_err("second case1"); }`
	`  }`
	
	`  $0.field1 := $0.field1;`
	
	`  return 0`
	`}`

CTL2	`//#CTL2`
	
	`integer myInt;`
	
	`// Transforms input record into output record.`
	`function integer transform() {`
	
	`  myInt = randomInteger(0,1);`
	
	`  switch(myInt) {`
	`  case 0 : printErr("zero"); printErr("first case"); break;`
	`  case 1 : printErr("one"); printErr("second case");`
	`  }`
	
	`  $0.field1 = $0.field1;`
	
	`  return 0;`
	`}`

手順10: シーケンスおよび参照表構文の変更

CTL1では、メタデータ、参照表およびシーケンスをそれらのIDで識別していました。

CTL2では、それらの名前で識別されます。

そのため、すべてのメタデータ、参照表およびシーケンスで名前が必ず一意になるようにします。そうでない場合は、名前を変更します。

次の2つの表は、参照表またはシーケンス構文を含むコードの変更方法を示しています。これらがIDで識別されているか(CTL1)、名前で識別されているかに注目してください。

CTL バージョン	シーケンス構文

CTL1	`//#CTL1`
	
	`// Transforms input record into output record.`
	`function transform() {`
	`  $0.field1 := $0.field1;`
	`  $0.field2 := $0.field2;`
	`  $0.field3 := sequence(Sequence0).current;`
	`  $0.field4 := sequence(Sequence0).next;`
	`  $0.field5 := sequence(Sequence0, string).current;`
	`  $0.field6 := sequence(Sequence0, string).next;`
	`  $0.field7 := sequence(Sequence0, long).current;`
	`  $0.field8 := sequence(Sequence0, long).next;`
	
	`  return 0`
	`}`

CTL2	`//#CTL2`
	
	`// Transforms input record into output record.`
	`function integer transform() {`
	`  $0.field1 = $0.field1;`
	`  $0.field2 = $0.field2;`
	`  $0.field3 = sequence(seqCTL2).current();`
	`  $0.field4 = sequence(seqCTL2).next();`
	`  $0.field5 = sequence(seqCTL2, string).current();`
	`  $0.field6 = sequence(seqCTL2, string).next();`
	`  $0.field7 = sequence(seqCTL2, long).current();`
	`  $0.field8 = sequence(seqCTL2, long).next();`
	
	`  return 0;`
	`}`

CTL バージョン	参照表の使用
CTL1	

```
//#CTL1

// variable for storing number of duplicates
int count;
int startCount;

// values of fields of the first records
string Field1FirstRecord;
string Field2FirstRecord;

// values of fields of next records
string Field1NextRecord;
string Field2NextRecord;

// values of fields of the last records
string Field1Value;
string Field2Value;

// record with the same metadata as those of lookup table
record (Metadata0) myRecord;

// Transforms input record into output record.
function transform() {

   // getting the first record whose key value equals to $0.Field2
   // must be specified the value of both Field1 and Field2
   Field1FirstRecord = lookup(LookupTable0,$0.Field2).Field1;
   Field2FirstRecord = lookup(LookupTable0,$0.Field2).Field2;

   // if lookup table contains duplicate records with the value specified above
   // their number is returned by the following expression
   // and assigned to the count variable
   count = lookup_found(LookupTable0);

   // it is copied to another variable
   startCount = count;

   // loop for searching the last record in lookup table
   while ((count - 1) > 0) {

      // searching the next record with the key specified above
      Field1NextRecord = lookup_next(LookupTable0).Field1;
      Field2NextRecord = lookup_next(LookupTable0).Field2;

      // decrementing counter
      count--;
   }

   // if record had duplicates, otherwise the first record
   Field1Value = nvl(Field1NextRecord,Field1FirstRecord);
   Field2Value = nvl(Field2NextRecord,Field2FirstRecord);

   // mapping to the output
   // last record from lookup table
   $0.Field1 := Field1Value;
   $0.Field2 := Field2Value;

   // corresponding record from the edge
   $0.Field3 := $0.Field1;
   $0.Field4 := $0.Field2;

   // count of duplicate records
   $0.Field5 := startCount;

   return 0
}
```

CTL バージョン	参照表の使用

CTL2	//#CTL2
	
	// record with the same metadata as those of lookup table
	recordName1 myRecord;
	
	// variable for storing number of duplicates
	integer count;
	integer startCount;
	
	// Transforms input record into output record.
	function integer transform() {
	
	  // if lookup table contains duplicate records,
	  // their number is returned by the following expression
	  // and assigned to the count variable
	  count = lookup(simpleLookup0).count($0.Field2);
	
	  // This is copied to startCount
	  startCount = count;
	
	  // getting the first record whose key value equals to $0.Field2
	  myRecord = lookup(simpleLookup0).get($0.Field2);
	
	  // loop for searching the last record in lookup table
	  while ((count-1) > 0) {
	
	  // searching the next record with the key specified above
	  myRecord = lookup(simpleLookup0).next();
	
	  // decrementing counter
	  count--;
	  }
	
	  // mapping to the output
	  // last record from lookup table
	  $0.Field1 = myRecord.Field1;
	  $0.Field2 = myRecord.Field2;
	
	  // corresponding record from the edge
	  $0.Field3 = $0.Field1;
	  $0.Field4 = $0.Field2;
	
	  // count of duplicate records
	  $0.Field5 = startCount;
	
	  return 0;
	}

警告

参照表については、別の構文を使用することをお薦めします。

その理由は、次のCTL2の式にあります。

```
lookup(Lookup0).count($0.Field2);
```

これは、多数のレコードが含まれている可能性がある参照表全体でレコードを検索します。

この構文のかわりに、次のループを使用できます。

```
myRecord = lookup(<name of lookup table>).get(<key value>);
while(myRecord != null) {
  process(myRecord);
  myRecord = lookup(<name of lookup table>).next();
}
```

特にDB参照表では、指定したキー値を持つレコードの実際の数ではなく、-1を返す可能性があります(**最大キャッシュ・サイズ**をゼロ以外の値に設定しなかった場合)。

CTL1の`lookup_found(<lookup table ID>)`関数も一般的にはお薦めしません。

手順11: 関数のマッピングの変更

CTL2構文に従って、マッピングを記述しなおします。前述のとおり、マッピング演算子を変更し、`@`を使用している式を削除します。

CTL バージョン	マッピングを含む変換

CTL1	`//#TL`
	
	`int retInt;`
	`function transform() {`
	
	`  if ($0.field3 < 5) retInt = 0; else retInt = 1;`
	
	`  // the following part is the mapping:`
	`  $0.* := $0.*;`
	`  $0.field1 := uppercase($0.field1);`
	`  $1.* := $0.*;`
	`  $1.field1 := uppercase($0.field1);`
	
	`  return retInt`
	`}`

CTL2	`//#CTL2`
	
	`integer retInt;`
	`function integer transform() {`
	
	`  // the following part is the mapping:`
	`  $0.* = $0.*;`
	`  $0.field1 = upperCase($0.field1);`
	`  $1.* = $0.*;`
	`  $1.field1 = upperCase($0.field1);`
	
	`  if ($0.field3 < 5) return = 0; else return 1;`
	`}`

第65章 CTL1

この章では、CTL1の構文および使用について説明します。言語リファレンスまたは組込み関数の詳細は、次の項を参照してください。

- 言語リファレンス(p.829)
- 関数リファレンス(p.859)

例65.1. CTL1構文の例(Rollup)

```
//#TL
list customers;
int Length;

function initGroup(groupAccumulator) {
}

function updateGroup(groupAccumulator) {
    customers = split($0.customers," - ");
    Length = length(customers);

    return true
}

function finishGroup(groupAccumulator) {
}

function updateTransform(counter, groupAccumulator) {
    if (counter >= Length) {
        remove_all(customers);

        return SKIP;
    }

    $0.customers := customers[counter];
    $0.EmployeeID := $0.EmployeeID;

    return ALL
}

function transform(counter, groupAccumulator) {
}
```

言語リファレンス

Clover Transformation Language (CTL)は、多数の変換コンポーネントで変換を定義するために使用されます。(すべての**ジョイナ**、**DataGenerator**、**Partition**、**DataIntersection**、**Reformat**、**Denormalizer**、**Normalizer**および**Rollup**で)

注意

CloverETLのバージョン2.8.0以上では、パラメータでCTL式を使用することもできます。このようなCTL式ではCTL言語のすべての機能を使用できます。ただし、これらのCTL式はバックォートで囲む必要があります。

たとえば、パラメータ`TODAY="`today()`"`を定義してCTLコード内で使用する場合、このような`${TODAY}`式はその日の日付に解決されます。

バックォートをそのまま表示する場合は、そのバックォートの前に\`のようにバックスラッシュを付けて使用する必要があります。

重要

CTL1バージョンはこのような式で使用されます。

この項では、次の領域について説明します。

プログラム構造

CTLで記述する各プログラムは、次の構造にする必要があります。

```
ImportStatements
VariableDeclarations
FunctionDeclarations
Statements
Mappings
```

ImportStatementsがプログラムの最初にあり、Mappingsが最後にある必要があります。ImportStatementsおよびMappingsはどちらも複数の文またはマッピングで構成でき、文やマッピングはそれぞれセミコロンで終了する必要があります。プログラムの中間部分は任意の場所に配置できます。変数および関数の個々の宣言と各文の順序をこのとおりにする必要はありません。ただし、常に宣言された変数および関数のみを使用する必要があります。したがって、まず変数または関数(あるいはその両方)を宣言する必要があり、その後、文または別の変数や関数の宣言でそれを使用できます。

コメント

プログラム全体でコメントを使用できます。これらのコメントは処理されず、プログラム内の処理の説明としてのみ使用します。

コメントには2種類あります。1行コメントまたは複数行コメントです。次の2つのオプションを参照してください。

```
// This is an one-line comment.
```

```
/* This is a multiline comment. */
```

インポート

まず、CTLでのプログラムの始めに、CTLの既存プログラムの一部をインポートできます。方法は次のとおりです。

```
import 'fileURL';
```

```
import "fileURL";
```

一重引用符または二重引用符のいずれを使用するかを決定する必要があります。一重引用符では、エスケープ・シーケンスをエスケープしません。詳細は、リテラル(p.833)を参照してください。このfileURLには、既存のソース・コード・ファイルのURLを入力する必要があります。

ただし、これらのファイルは、その他すべての宣言または文(あるいはその両方)より前に、始めにインポートする必要があります。

CTLのデータ型

メタデータで使用されるデータ型の基本情報は、データ型およびレコード・タイプ(p.111)を参照してください。

どのプログラムでも変数を使用できます。CTLのデータ型は次のとおりです。

boolean

次のように宣言します。`boolean identifier;`

bytearray

このデータ型は、最大長が`Integer.MAX_VALUE`のバイト配列です。これはlistデータ型と同様に動作します(次を参照してください)。

次のように宣言します。`bytearray [(size)] identifier;`

date

次のように宣言します。`date identifier;`

decimal

次のように宣言します。`decimal[(length,scale)] identifier;`

デフォルトの`length`および`scale`は、それぞれ12および2です。

DECIMAL_LENGTHおよびDECIMAL_SCALEのデフォルト値は、`org.jetel.data.defaultProperties`ファイルに格納されており、他の値への変更が可能です。

浮動小数点数の末尾にdの文字を付加すると、浮動小数点数をdecimalデータ型にキャストできます。

int

次のように宣言します。`int identifier;`

整数の末尾にlの文字を付加すると、整数をlongデータ型にキャストできます。

long

次のように宣言します。`long identifier;`

整数の末尾にlの文字を付加すると、任意の整数をこのデータ型にキャストできます。

number (double)

次のように宣言します。`number identifier;`

string

次のように宣言します。`string identifier;`

list

各`list`は、`boolean`、`byte`、`date`、`decimal`、`integer`、`long`、`number`、`string`の各プリミティブ・データ型のコンテナです。

listデータ型は、0で始まる整数でインデックスが付けられます。

次のように宣言します。`list identifier;`

デフォルト・リストは空のリストです。

例:

```
list list2; examplelist2[5]=123;
```

割当て:

- `list1=list2;`

  これは、両方のリストが同じ要素を参照することを意味します。

- `list1[ ]=list2;`

  これは、`list2`のすべての要素を`list1`の末尾に追加します。

- `list1[ ]="abc";`

  これは、`"abc"`文字列を`list1`の最後の要素として新しく追加します。

- `list1[ ]=NULL;`

  これは、`list1`の最後の要素を削除します。

map

このデータ型は、任意のデータ型のコンテナです。

マップは、文字列でインデックスが付けられます。

次のように宣言します。`map identifier;`

デフォルト・マップは空のマップです。

例: `map map1; map1["abc"]=true;`

割当ては、リストに有効なものと同様です。

record

このデータ型は、一連のデータ・フィールドです。

レコードの構造はメタデータに基づいています。

その宣言は、次のオプションのいずれかとなります。

1. `record (<metadata ID>) identifier;`
2. `record (@<port number>) identifier;`
3. `record (@<metadata name>) identifier;`

可能な式およびレコードの使用方法の詳細は、データ・レコードおよびフィールドへのアクセス(p.849)を参照してください。

この変数にデフォルト値はありません。

整数および文字列の両方でインデックスを付けることができます。インデックスが数字である場合、フィールドのインデックスは0から始まります。

リテラル

リテラルは、任意のデータ型の値を記述するために使用します。

表65.1. リテラル

リテラル	説明	宣言構文	例
整数	整数を表す数字。	[0-9]+	95623
長整数	絶対値が2^{31}より大きく2^{63}より小さい整数を表す数字。	[0-9]+L?	257Lまたは9562307813123123
16進数整数	16進数形式で整数を表す数字と文字。	0x[0-9A-F]+	0xA7B0
8進数整数	8進数形式で整数を表す数字。	0[0-7]*	0644
数値(double)	倍精度形式の64ビットで表される浮動小数点数。	[0-9]+.[0-9]+	456.123
小数	小数を表す数字。	[0-9]+.[0-9]+D	123.456D
二重引用符付き文字列	二重引用符で囲まれた文字列値/リテラル。エスケープ文字[\n,\r,\t, \\, \", \b]は対応する制御文字に変換される。	"...["]以外の任意の文字列..."	"hello\tworld\n\r"
一重引用符付き文字列	一重引用符で囲まれた文字列値/リテラル。エスケープ文字は[\']のみが対応する文字[']に変換される。	'...[']以外の任意の文字列...'	'hello\tworld\n\r'
リテラルのリスト	リテラルのリスト。各リテラルはその他のリスト、マップまたはレコードにもできる。	[<any literal> (, <any literal>)*]	[10, 'hello', "world", 0x1A, 2008-01-01], [[1 , 2]] , [3 , 4]]
日付	日付値。	予期されるマスク: yyyy-MM-dd	2008-01-01
日時	日時値。	予期されるマスク: yyyy-MM-dd HH:mm:ss	2008-01-01 18:55:00

重要

`bytearray`データ型にはどのリテラルも使用できません。`bytearray`値を書き込むには、`bytearray`を返す変換関数のいずれかを使用し、引数値に適用する必要があります。

これらの変換関数の詳細は、変換関数(p.860)を参照してください。

重要

小数値を小数フィールドに割り当てる必要がある場合は、小数リテラルを使用する必要があります。そうしない場合、数値は小数ではなく倍精度浮動小数点数になります。

例:

1. **小数フィールドへの小数値の割当て(正しく正確である)**

```
// correct - assign decimal value to decimal field
myRecord.decimalField = 123.56d;
```

2. **小数フィールドへの倍精度浮動小数点値の割当て(不正確な可能性がある)**

```
// possibly inaccurate - assign double value to decimal field
myRecord.decimalField = 123.56;
```

後者は不正確な結果を生成する可能性があります。

変数

変数を定義するには、変数のデータ型、空白、変数の名前およびセミコロンを入力する必要があります。

これらの変数は後から初期化できますが、その宣言自体で初期化することもできます。式の値は変数と同じデータ型である必要があります。

変数の宣言および初期化の両方の例は、次のとおりです。

```
dataType variable;

...

variable=expression;

dataType variable=expression;
```

演算子

演算子は、プログラム内でより複雑な式を作成するためのものです。算術、関係および論理演算子を使用できます。関係および論理演算子は、結果としてブール値を生成する式を作成します。算術演算子は、論理式のみでなく、すべての式で使用できます。

すべての演算子は3つのカテゴリに分類できます。

- 算術演算子(p.836)
- 関係演算子(p.838)
- 論理演算子(p.840)

算術演算子

次の演算子は、異なる式の値をまとめるために使用します(ブール値の式は除く)。これらの記号は1つの式で複数回使用できます。そのような場合、カッコを使用して演算子の優先度を表現できます。結果は式の順序によって異なります。

- 加算

+

この演算子は、2つの式の値を加算するために使用します。

ただし、2つのブール値または2つの日付データ型の加算はできません。2つのブール値から新しい値を作成するには、かわりに論理演算子を使用する必要があります。

文字列に任意のデータ型を追加する場合は、2番目のデータ型が自動的に文字列に変換され、最初の(文字列の)被加数と連結されます。ただし、文字列を先頭に置く必要があります。2つの文字列も同じ方法で加算できます。また、`concat()`関数の方が高速であるため、文字列への追加よりも、この関数を使用することをお薦めします。

任意の数値データ型を日付に追加することもできます。結果は、数値の整数部分だけ日数が増加した日付になります。この場合も、日付を先頭に置く必要があります。

2つの数値データ型の合計は、それらのデータ型の順序によって異なります。結果のデータ型は最初の被加数のデータ型と同じになります。2番目の被加数は最初のデータ型に自動的に変換されます。

- 減算およびユニタリのマイナス

−

この演算子は、ある数値データ型を他の数値データ型から減算するために使用します。この場合も、結果のデータ型は被減数のデータ型と同じになります。減数は、被減数のデータ型に自動的に変換されます。

また、日付データ型から数値データ型を減算することもできます。結果は、減数の整数部分だけ日数が減少した日付になります。

- 乗算

*

この演算子は、2つの数値データ型を乗算するためにのみ使用します。

乗算では、最初の被乗数によって演算結果のデータ型が決まります。最初の被乗数が整数で、2番目の被乗数が小数の場合、結果は整数になります。一方、最初の被乗数が小数で、2番目の被乗数が整数の場合、結果は小数データ型になります。つまり、被乗数の順序が重要になります

- 除算

/

この演算子は、2つの数値データ型を除算するためにのみ使用します。ゼロでは除算できません。ゼロで除算すると、TransformLangExecutorRuntimeExceptionがスローされるか、Infinityが返されます(数値データ型の場合)。

除算では、分子によって演算結果のデータ型が決まります。分子が整数で、分母が小数の場合、結果は整数になります。一方、分子が小数で、分母が整数の場合、結果は小数データ型になります。つまり、分子と分母のデータ型が重要になります。

- 法

%

この演算子は、浮動小数点数データ型および整数データ型の両方に使用できます。これは除算の余りを返します。

- 増分

++

この演算子は、数値データ型を1ずつ増分するために使用します。この演算子は、浮動小数点数データ型および整数データ型の両方に使用できます。

これを接頭辞として使用すると、数値は最初に増分されてから式で使用されます。

これを接尾辞として使用すると、数値は最初に式で使用されてから増分されます。

- 減分

--

この演算子は、数値データ型を1ずつ減分するために使用します。この演算子は、浮動小数点数データ型および整数データ型の両方に使用できます。

これを接頭辞として使用すると、数値は最初に減分されてから式で使用されます。

これを接尾辞として使用すると、数値は最初に式で使用されてから減分されます。

関係演算子

次の演算子は、副次式を比較してブール値の結果を取得する場合に使用します。次に示す各記号を使用できます。.operator.記号を選択する場合は、空白で囲む必要があります。これらの記号は1つの式で複数回使用できます。そのような場合、カッコを使用して比較の優先度を表現できます。

- より大きい

  次の2つの記号は、数値、日付および文字列データ型で構成される式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

>

.gt.

- 以上

  次の3つの記号は、数値、日付および文字列データ型で構成される式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

>=

=>

.ge.

- より小さい

  次の2つの記号は、数値、日付および文字列データ型で構成される式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

<

.lt.

- 以下

  次の3つの記号は、数値、日付および文字列データ型で構成される式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

<=

=<

.le.

- 等しい

  次の2つの記号は、任意のデータ型の式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

==

.eq.

- 等しくない

  次の3つの記号は、任意のデータ型の式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

`!=`

`<>`

`.ne.`

- 正規表現と一致

  この演算子は、文字列および正規表現(p.963)を比較するために使用します。文字列全体が正規表現と一致する場合は`true`を返し、そうでない場合は`false`を返します。

`~=`

`.regex.`

- 含まれる

  この演算子は、ある値が他の値のリストまたはマップに含まれているかどうかを指定するために使用します。

`.in.`

論理演算子

値がブール・データ型である必要がある式が複雑な場合、論理接続詞(AND、OR、NOT、EQUAL TO、NOT EQUAL TO)で接続された副次式(前述)で式を構成できます。そのような式で優先度を示すには、カッコを使用します。次に示す接続詞のいずれかの形式を選択できます(たとえば、`&&`または`and`)。

`.operator.`という形式の各記号は、空白で囲む必要があります。

- 論理AND

`&&`

`and`

- 論理OR

`||`

`or`

- 論理NOT

`!`

`not`

- 論理EQUAL TO

`==`

`.eq.`

- 論理NOT EQUAL TO

`!=`

`<>`

`.ne.`

単純文および文のブロック

すべての文は次の2つのグループに分けることができます。

- **単純文**はセミコロンで終了する式です。

  例:

```
int MyVariable;
```

- **文のブロック**は一連の単純文です(各文はセミコロンで終了しています)。ブロック内の文は、すべてを1行に記述することも、複数行に記述することもできます。これらは中カッコで囲みます。閉じ中カッコの後ろにセミコロンは使用しません。

  例:

```
while (MyInteger<100) {
    Sum = Sum + MyInteger;
    MyInteger++;
}
```

制御文

一部の文はプログラムのプロセスを制御するために使用します。

すべての制御文は次のカテゴリに分類できます。

- 条件文(p.841)
- 繰返し文(p.842)
- ジャンプ文(p.843)

条件文

これらの文はプログラムのプロセスを分岐するために使用します。

If 文

この文は、Conditionの値に基づいてStatementを実行するかどうかを決定します。Conditionがtrueの場合、Statementが実行されます。falseの場合、Statementは無視され、プロセスはif文の後に進みます。Statementは単純文または文のブロックのいずれかです。

```
if (Condition) Statement
```

if文の前のバージョン(StatementはConditionがtrueの場合にのみ実行される)とは異なり、Conditionの値がfalseの場合でも実行する必要がある他のStatementをif文に追加できます。したがって、Conditionがtrueの場合はStatement1が実行され、falseの場合はStatement2が実行されます。次を参照してください。

```
if (Condition) Statement1 else Statement2
```

Statement2は別のif文にすることも、それにelse分岐を含めることもできます。

```
if (Condition1) Statement1
    else if (Condition2) Statement3
        else Statement4
```

switch 文

より分岐の多いif文を作成した場合、文が非常に複雑になることがあります。そのような場合は、switch文を使用する方が適しています。

このとき、Conditionが評価され、Expressionの値に従ってプロセスを分岐できます。Expressionの値がExpression1の値に等しい場合は、Statement1が実行されます。他のExpression : Statementのペアについても同様です。ただし、Expressionの値がExpression1,...,ExpressionNのいずれにも等しくない場合、処理は行われず、プロセスがswitch文を省略します。また、Expressionの値が複数のExpressionKの値に等しい場合は、複数のStatementK (Kが異なる)が実行されます。

```
switch (Expression) {
    case Expression1 : Statement1
    case Expression2 : Statement2
    ...
    case ExpressionN : StatementN
}
```

次の場合では、Expressionの値がExpression1,...,ExpressionNの値に等しくない場合、StatementN+1が実行されます。

```
switch (Expression) {
    case Expression1 : Statement1
    case Expression2 : Statement2
    ...
    case ExpressionN : StatementN
    default:StatementN+1
}
```

switch関数のヘッダーにあるExpressionの値がそのボディにある複数のExpressions#の値に等しい場合、各Expression#値はExpression値と連続して比較され、ExpressionおよびExpression#の値が互いに等しい対応するStatementが1つずつ実行されます。ただし、あるExpression#値に対して1つのStatement#のみが実行されるようにするには、すべての、または少なくとも一部のcase Expression#式に続く文のブロックの最後に、break文を置く必要があります。

その結果、次のようになります。

```
switch (Expression) {
    case Expression1 : {Statement1; break;}
    case Expression2 : {Statement2; break;}
    ...
    case ExpressionN : {StatementN; break;}
    default:StatementN+1
}
```

繰返し文

これらの繰返し文は、実行サイクルを制限するConditionがfalseになるか同じデータ型のすべての値について実行されるまで、内側のStatementを循環的に実行することによって、なんらかのプロセスを繰り返します。

for ループ

まず、初期化が設定され、その後Conditionが評価されます。その値がtrueの場合、Statementが実行され、最終的にIterationが行われます。

ループの次のサイクルではConditionが再度評価され、trueの場合はStatementが実行され、Iterationが行われます。このようにして、Conditionがfalseになるまで、プロセスが繰り返されます。その後、ループが終了し、プロセスはプログラムの別の部分に進みます。

最初の時点でConditionがfalseだった場合、プロセスはStatementを省略し、ループを終了します。

```
for (Initialization;Condition;Iteration) {
    Statement
}
```

do-while ループ

最初にStatementが実行されます。次のプロセスはConditionの値によって異なります。値がtrueの場合、Statementが再度実行され、続けてConditionが再度評価されます。サブプロセスは続行する(再度trueの場合)か、停止して次またはより上位レベルのサブプロセスにジャンプします(falseの場合)。Conditionはループの最後にあるため、サブプロセスの始めにこれがfalseだった場合でも、Statementが1回以上実行されます。

```
do {
    Statement
} while (Condition)
```

while ループ

このプロセスはConditionの値によって異なります。値がtrueの場合、Statementが実行され、続けてConditionが再度評価されます。サブプロセスは続行する(再度trueの場合)か、停止して次またはより上位レベルのサブプロセスにジャンプします(falseの場合)。Conditionはループの先頭にあるため、サブプロセスの始めにこれがfalseだった場合、Statementは実行されず、ループが省略されます。

```
while (Condition) {
    Statement
}
```

for-each ループ

foreach文は、コンテナ内の同じデータ型を持つすべてのフィールドで実行されます。構文は次のとおりです。

```
foreach (variable : iterableVariable) {
    Statement
}
```

同じデータ型のすべての要素(データ型は変換コードの最初でvariableに対して宣言されます)が、iterableVariableコンテナ内で検索されます。iterableVariableには、リスト、マップまたはレコードを指定できます。同じデータ型の各変数について、指定したStatementが実行されます。これには、単純文または文のブロックのいずれかを指定できます。

そのため、たとえば、同じStatementを1つのレコードのすべてのstringフィールドで実行できます。

ジャンプ文

場合によっては、Conditionの値に基づいて決定するのとは別の方法でプロセスを制御する必要があります。このことを行うには、次のオプションがあります。

break 文

サブプロセスを停止する場合は、プログラム内で次の語を使用できます。

```
break
```

サブプロセスが中断し、プロセスはより上位レベルまたは次のStatementにジャンプします。

continue 文

繰返しサブプロセスを停止する場合は、プログラム内で次の語を使用できます。

continue

サブプロセスが中断し、プロセスは次の繰返しステップにジャンプします。

return 文

関数で、returnという語を単独でまたはexpressionとともに使用できます。(次の2つのオプションを参照してください。)return文は関数の最後に配置する必要があります。この文が最後に配置されなかった場合、それより後にあるvariableDeclarations、StatementsおよびMappingsはすべて無視され、スキップされます。return語があってもreturn語がなくても関数全体はnullを返しますが、return expressionのある関数はexpressionの値を返します。

return

return expression

エラー処理

Clover Transformation Languageは、発生が予想されるエラーを捕捉して処理するための簡単なメカニズムも提供しています。

最初の手順として、文字列変数を宣言する必要があります。

```
string MyException;
```

その後、変換のコード内で次のtry-catch文を使用します。

```
try Statement1 catch(MyException) [Statement2]
```

Statement1 (単純文または文のブロック)の実行に成功した場合、グラフは失敗せずに処理が続行されます。

一方、Statement1が失敗した場合は、例外がスローされてMyException変数に割り当てられます。

MyException変数は、他の方法で出力または管理できます。

例外がスローされると、グラフは失敗して別のStatement2 (失敗したStatement1を修正する)が実行されます。

さらに、**CloverETL**のバージョン3.0以上では、CTL1は必須変換関数ごとに存在するオプションのOnError()関数セットを使用します。

たとえば、必須関数(append()、transform()など)に、次のようなオプションの関数があります。

appendOnError()、transformOnError()など

これらの必須関数にはそれぞれ、元の(必須の)関数の名前にOnError接尾辞が追加された名前を持つ、(オプションの)対応する関数がある場合があります。

さらに、すべての<required ctl template function>OnError()関数は元の必須関数と同じ値を返します。

このようにして、元の必須関数によりスローされる例外によって、対応する<required ctl template function>OnError()関数が呼び出されます(たとえば、transform()が失敗するとtransformOnError()が呼び出されます)。

このtransformOnError()では、正しくないコードを修正する、エラー・メッセージをコンソールに出力するなどの操作を実行できます。

重要

これらのOnError()関数は、元の必須関数が**エラー・コード**(-1未満の値)を返した場合は呼び出されません。

OnError()関数を呼び出す必要がある場合は、raiseError(string arg)関数を使用する必要があります。または(前述したように)、元の必須関数により例外がスローされると、対応するOnError()関数が呼び出されます。

関数

独自の関数を次の方法で定義できます。

```
function functionName (arg1,arg2,...){
    variableDeclarations
    Statements
    Mappings
    [return [expression]]
}
```

return文は最後に置く必要があります。return文の詳細は、return文(p.844)を参照してください。その直前にはMappingsがある場合があります。variableDeclarationsおよびStatementsは最初に置く必要があり、variableDeclarationsとStatementsは任意の場所に配置できますが、宣言および初期化されていない変数は使用できません。そのため、最初に変数を宣言してからのみ、Statementsを指定することをお薦めします。

メッセージ関数

CloverETLバージョン2.8.0から、ユーザー独自のエラー・メッセージ用の関数も定義できるようになりました。

```
function getMessage() {
        return message;
}
```

このmessage変数は、getMessage()関数がある場所で使用されるように、グローバル文字列変数として宣言し、コード内の任意の場所で定義する必要があります。messageはコンソールに書き出されます。

Eval

CTL1は、文字列がCTLコードであるかのようにその文字列をインラインで評価できる2つの関数を提供しています。この方法では、たとえば、データベースのどこかにコードとして格納されているテキストを解釈できます。

- `eval(string):string`をCTLコードとして実行します。
- `eval_exp(string):string`をCTL式(`eval(string)`と同様に)として評価し、結果の値を返します。この値は変数に保存できます。

この関数を使用する場合は、有効なCTLコードのみを関数に渡す必要があります。このため、適切な識別子を使用し、式をセミコロンで終了する必要があります。評価される式のスコープは限られ、式の外部で宣言された変数は認識できません。

例65.2. Eval()関数の例

例1:

```
int value = eval_exp("2+5"); // the result of the expression is stored to 'value'
```

例2:

```
int out; // the variable has to be declared as global

function transform() {

   eval("out = 3 + 5;");
   print_err(out);
   $0.Date := $0.DATE;
   return ALL
}
```

条件付き失敗式

CTL1では条件付き失敗式を使用できます。

ただし、これは出力フィールドに対するマッピングを定義する場合にのみ使用できます。

重要

(CTL2のインタプリタ・モードでは)この式は複数の方法(変数への値の割当てや出力フィールドへの値のマッピングのために、または関数の引数として)で使用できます。

条件付き失敗式は、次のようになります(1つの出力フィールドの場合)。

```
expression1 : expression2 : expression3 : ...: expressionN;
```

式は、最初の式から順に、左から右に1つずつ評価されます。

1. これらの式のいずれかが出力フィールドに正しくマッピングされると、その式がただちに使用され、その他の式は評価されません。
2. これらの式がどれも出力フィールドにマッピングされなかった場合、グラフは失敗します。

データ・レコードおよびフィールドへのアクセス

この項では、レコード・フィールドの操作方法について説明します。各コンポーネントにはポートがある場合があります。入力ポートおよび出力ポートのいずれも、0から始まる番号が付けられます。

接続済エッジのメタデータは、名前またはIDによって識別できます。

メタデータはフィールドで構成されます。

レコードおよび変数の操作

ここでは、最初のポート(ポート0)に接続したエッジのメタデータのIDを(入力または出力かに関係なく)Customers、その名前をcustomers、および3番目のフィールド(フィールド2)をfirstnameと想定します。

次の式は、指定したメタデータの3番目のフィールド(フィールド2)の値を表しています。

- `$<port number>.<field number>`

  例: `$0.2`

  `$0.*`は、最初のポート(ポート0)のすべてのフィールドを意味します。

- `$<port number>.<field name>`

  例: `$0.firstname`

- `$<metadata name>.<field number>`

  例: `$customers.2`

  `$customers.*`は、最初のポート(ポート0)のすべてのフィールドを意味します。

- `$<metadata name>.<field name>`

  例: `$customers.firstname`

入力データの場合は、次の構文を使用することもできます。

- `@<port number>[<field number>]`

  例: `@0[2]`

  `@0`は、最初のポート(ポート0)を使用して受信する入力レコード全体を意味します。

- `@<metadata name>[<field number>]`

  例: `@customers[2]`

  `@customers`は、メタデータ名がcustomersの入力レコード全体を意味します。

整数変数は、入力レコードのフィールド番号を識別するためにも使用できます。整数変数を宣言すると(`int index;`)、次のことが可能になります。

- `@<port number>[index]`

  例: `@0[index]`

- `@<metadata name>[index]`

  例: `@customers[index]`

  複数のエッジでメタデータ名が同じになる場合があります。

このようにしてループを作成することもできます。たとえば、最初の入力ポートにフィールド値を出力するには、次のように入力します。

```
int i;
for (i=0; i<length(@0); i++){
     print_err(@0[i]);
}
```

CTLコードでレコードを定義することもできます。この定義は次のようになります。

- `record (metadataID) MyCTLRecord;`

  例: `record (Customers) MyCustomers;`

- `record (@<port number>) MyCTLRecord;`

  例: `record (@0) MyCustomers;`

  これは入力ポートの場合にのみ可能です。

- `record (@<metadata name>) MyCTLRecord;`

  例: `record (@customers) MyCustomers;`

入力エッジからのレコードは、CTLで宣言されたレコードに次のように割り当てることができます。

- `MyCTLRecord = @<port number>;`

  例: `MyCTLRecord = @0;`

変数へのレコードのマッピングは次のようになります。

- `myVariable = $<port number>.<field number>;`

  例: `FirstName = $0.2;`

- `myVariable = $<port number>.<field name>;`

  例: `FirstName = $0.firstname;`

- `myVariable = $<metadata name>.<field number>;`

  例: `FirstName = $customers.2;`

  メタデータ名は一意である必要があります。そうでない場合は、かわりにポート番号を使用してください。

- `myVariable = $<metadata name>.<field name>;`

  例: `FirstName = $customers.firstname;`

  メタデータ名は一意である必要があります。そうでない場合は、かわりにポート番号を使用してください。

- `myVariable = @<port number>[<field number>];`

  例: `FirstName = @0[2];`

- `myVariable = @<metadata name>[<field number>];`

  例: `FirstName = @customers[2];`

  メタデータ名は一意である必要があります。そうでない場合は、かわりにポート番号を使用してください。

レコードへの変数のマッピングは次のようになります。

- `$<port number>.<field number> := myVariable;`

  例: `$0.2 := FirstName;`

- `$<port number>.<field name> := myVariable;`

  例: `$0.firstname := FirstName;`

- `$<metadata name>.<field number> := myVariable;`

  例: `$customers.2 := FirstName;`

- `$<metadata name>.<field name> := myVariable;`

  例: `$customers.firstname := FirstName;`

マッピング

マッピングは、一部の**CloverETL**コンポーネントで定義される各変換の一部です。

計算または生成された値または入力フィールドの値が出力フィールドに割り当てられます(マップされます)。

1. マッピングにより出力フィールドに値が割り当てられます。
2. マッピング演算子は次のとおりです。

   `:=`

3. マッピングは常に関数の内部で定義する必要があります。
4. マッピングは関数の最後で定義する必要があり、その後に続くのは1つのreturn文のみです。
5. ユーザー定義関数にマッピングをラップし、後で、別の関数の任意の場所で使用することもできます。
6. 様々な入力メタデータを様々な出力メタデータにフィールド名によってマップすることもできます。

異なるメタデータの(名前による)マッピング

次のように入力を出力にマップする場合、

```
$0.* := $0.*;
```

入力メタデータが出力のメタデータとも異なる場合があります。

前述の式では、入力フィールドが入力と同じ名前および型の出力フィールドにマップされます。各メタデータでの格納順序およびいずれのメタデータの全フィールドの数も重要ではありません。

例65.3. メタデータの名前によるマッピング

最初の2つのフィールドが`firstname`および`lastname`である入力メタデータがある場合、これらの2つのフィールドはそれぞれ出力の対応するフィールドにマップされます。そのような出力の`firstname`フィールドが5番目で、`lastname`フィールドが3番目である場合でも、これら2つの入力フィールドはこれら2つの出力フィールドにマップされます。

入力メタデータに複数のフィールドがあり、出力メタデータにも複数のフィールドがある場合でも、入力フィールドのいずれかと同じ名前のフィールドが存在しない場合、そのようなフィールドは互いにマップされません(対応するメタデータ内のフィールドの相互位置に関係なく)。

重要

メタデータ・フィールドは、その順序および全フィールドの数とは関係なく、名前およびデータ型によって入力から出力にマップされます。

ユースケース1: 1つの文字列フィールドを大文字に変換する場合

マッピングがどのように行われるかを示すために、マッピングの例をいくつか示します。

Reformatコンポーネントを持つグラフがあります。その入力および出力のメタデータは同一です。最初の2つのフィールド(`field1`および`field2`)は文字列データ型で、3番目のフィールド(`field3`)は整数データ型です。

1. `field1`の値の文字は大文字に変更し、他の2つのフィールドは変更しないで出力に渡します。
2. また、`field3`の値に基づいてレコードを分配します。`field3`の値が5未満のレコードは出力ポート0に送信され、それ以外は出力ポート1に送信されるようにします。

マッピングの例

最初の方法として、CTLテンプレートの`transform()`関数内で、両方のポートおよびすべてのフィールドのマッピングを定義します。

例65.4. 個々のフィールドを使用したマッピングの例

マッピングは、関数の最後に定義する必要があり(この場合は`transform()`関数)、その後に続くのは1つのreturn文のみです。

各レコードの出力ポートの数を`return`文で返すことが必要であるため、マッピングを定義する*前*に、対応する値にこの数を割り当てる必要があります。

マッピングはすべてのレコードについて実行されます。つまり、レコードが出力ポート1に送信される場合にも、出力ポート0のマッピングも実行され、その逆も同様です。

また、マッピングは個々のフィールドで構成され、レコードに多数のフィールドが含まれる場合は複雑になる可能性があります。次の例では、より適切な方法で解決できる方法を示します。

```
//#TL

// declare variable for returned value (number of output port)
int retInt;

function transform() {
    // create returned value whose meaning is the number of output port.
    if ($0.field3 < 5) retInt = 0; else retInt = 1;

    // define mapping for all ports and for each field
    // (each field is mapped separately)
    $0.field1 := uppercase($0.field1);
    $0.field2 := $0.field2;
    $0.field3 := $0.field3;
    $1.field1 := uppercase($0.field1);
    $1.field2 := $0.field2;
    $1.field3 := $0.field3;

    // return the number of output port
    return retInt
}
```

2番目の方法でも、CTLテンプレートの`transform()`関数内で、両方のポートおよびすべてのフィールドのマッピングを定義します。ただし、今回は、マッピングでワイルドカードが使用されています。レコードは変更しないで出力に渡され、このワイルドカード・マッピングの後で、変更する必要があるフィールドが指定されます。

例65.5. ワイルドカードを使用したマッピングの例

マッピングは、関数の最後に定義する必要もあり(この場合はtransform()関数)、その後に続くのは1つのreturn文のみです。

各レコードの出力ポートの数をreturn文で返すことが必要であるため、マッピングを定義する前に、対応する値にこの数を割り当てる必要があります。

マッピングはすべてのレコードについて実行されます。つまり、レコードが出力ポート1に送信される場合にも、出力ポート0のマッピングも実行され、その逆も同様です。

ただし、今回は、マッピングは最初にワイルドカードを使用して、レコードを変更しないで出力に渡し、ワイルドカードを使用したマッピングの後で最初のフィールドが変更されます。

これは、変更しないフィールドが多数あり、変更するフィールドが少数ある場合に便利です。

```
//#TL

// declare variable for returned value (number of output port)
int retInt;

function transform() {
    // create returned value whose meaning is the number of output port.
    if ($0.field3 < 5) retInt = 0; else retInt = 1;

    // define mapping for all ports and for each field
    // (using wild cards and overwriting one selected field)
    $0.* := $0.*;
    $0.field1 := uppercase($0.field1);
    $1.* := $0.*;
    $1.field1 := uppercase($0.field1);

    // return the number of output port
    return retInt
}
```

3番目の方法として、CTLテンプレートのtransform()関数の外で、両方のポートおよびすべてのフィールドのマッピングを定義します。各出力ポートに独自のマッピングがあります。

ここでも、ワイルドカードが使用されます。

出力ポートごとに個別の関数でマッピングを定義すると、次のようなことが改善されます。

1. transform()関数のコード内部でマッピングを使用できるようになります。関数の最後のみではありません。
2. マッピングは対象の出力ポートについてのみ実行されます。つまり、ポート0に送信されるレコードをポート1にマップする必要がなくなり、その逆も同様です。
3. また、出力ポートの数のための変数は必要ありません。出力ポートの数は、対応するマッピング関数の直後にある定数によって定義されます。

例65.6. 別個のユーザー定義関数でのワイルドカードを使用したマッピングの例

マッピングは、関数の最後に定義する必要があります(2つの個別関数、出力ポートごとに1つずつ)。

さらに、マッピングは最初にワイルドカードを使用して、レコードを変更しないで出力に渡し、ワイルドカードを使用したマッピングの後で最初のフィールドが変更されます。これは、変更しないフィールドが多数あり、変更するフィールドが少数ある場合に便利です。

```
//#TL

// define mapping for each port and for each field
// (using wild cards and overwriting one selected field)
// inside separate functions
function mapToPort0 () {
    $0.* := $0.*;
    $0.field1 := uppercase($0.field1);
}

function mapToPort1 () {
    $1.* := $0.*;
    $1.field1 := uppercase($0.field1);
}

// use mapping functions for all ports in the if statement
function transform() {
    if ($0.field3 < 5) {
        mapToPort0();
        return 0
    }
    else {
        mapToPort1();
        return 1
    }
}
```

ユースケース2: 2つの文字列フィールドを大文字に変換する場合

Reformatコンポーネントを持つグラフがあります。その入力および出力のメタデータは同一です。最初の2つのフィールド(`field1`および`field2`)は文字列データ型で、3番目のフィールド(`field3`)は整数データ型です。

1. `field1`および`field2`の値の文字は大文字に変更し、最後のフィールド(`field3`)は変更しないで出力に渡します。

2. また、`field3`の値に基づいてレコードを分配します。`field3`の値が5未満のレコードは出力ポート0に送信され、それ以外は出力ポート1に送信されるようにします。

例65.7. 個別のユーザー定義関数での連続マッピングの例

2つの個別ユーザー定義関数にマッピングを定義します。最初の関数では、最初の入力フィールドを両方の出力ポートにマップします。2番目では、その他のフィールドを両方の出力フィールドにマップします。

これらの関数は、文字列データ型の1つの入力パラメータ(valueString)を受け入れます。

変換では、CTLレコード変数を宣言します。入力レコードはこの変数にマップされ、レコードはforeachループで使用されます。

出力ポートの数は、マッピング関数のコードに続くif内に定義されています。

```
//#TL

string myString;
string newString;
string valueString;

// declare the count variable for counting string fields
int count;

// declare CTL record for the foreach loop
record (@0) CTLRecord;

// declare mapping for field 1
function mappingOfField1 (valueString) {
    $0.field1 := valueString;
    $1.field1 := valueString;
}

// declare mapping for field 2 and 3
function mappingOfField2and3 (valueString) {
    $0.field2 := valueString;
    $1.field2 := valueString;
    $0.field3 := $0.field3;
    $1.field3 := $0.field3;
}

function transform() {

    // count is initialized for each record
    count = 0;

    // input record is assigned to CTL record
    CTLRecord = @0;

    // value of each string field is changed to upper case letters
    foreach (myString : CTLRecord) {
        newString = uppercase(myString);
        // count variable counts the string fields in the record
        count++;

        // mapping is used for fields 1 and the other fields - 2 and 3
        switch (count) {
            case 1 : mappingOfField1(newString);
            case 2 : mappingOfField2and3(newString);
        }
    }

    // output port is selected based on the return value
    if($0.field3 < 5) return 0 else return 1
}
```

パラメータ

パラメータは、Clover Transformation Languageで`${nameOfTheParameter}`のように使用できます。このパラメータが文字列データ型とみなされるようにするには、`'${nameOfTheParameter}'`または`"${nameOfTheParameter}"`のように、パラメータを一重引用符または二重引用符で囲む必要があります。

重要

1. エスケープ・シーケンスは常に、パラメータに割り当てられるとすぐに解決されます。このため、これらが解決されないようにする場合は、これらの文字列で、単一ではなく二重のバックスラッシュを入力してください。
2. また、パラメータを使用して、環境変数の値を取得することもできます。方法は、環境変数 (p.223)を参照してください。

関数リファレンス

Clover Transformation Languageには、ユーザーが使用できる一連の関数があります。ここではこれらについて説明します。

すべての関数は次のカテゴリに分類できます。

重要

`string`データ・フィールドのメタデータで「**Null value**」プロパティを空ではない文字列に設定した場合、`string`データ・フィールドを引数として受け入れ、`null`で適用された場合にNPEをスローする関数(たとえば、`length()`)は、そのような文字列に適用された場合にNPEをスローします。

たとえば、`field1`で「**Null value**」プロパティを"<null>"に設定した場合、`length($0.field1)`は、レコードの`field1`の値が"<null>"である場合は失敗し、フィールドが空の場合は0になります。

詳細は、Null value(p.164)を参照してください。

変換関数

値のデータ型を変換する必要がある場合があります。

データ型を変換する関数では、日付や数値の書式パターンを定義する必要がある場合があります。また、ロケールもそれらの書式設定に影響する場合があります。

- 日付の書式設定または解析(あるいはその両方)の詳細は、日付と時刻の書式(p.113)を参照してください。
- 数値データ型の書式設定または解析(あるいはその両方)の詳細は、数値の書式(p.120)を参照してください。
- ロケールの詳細は、ロケール(p.126)を参照してください。

注意

数値および日付の書式は、その他の**ロケール**が明示的に指定されていないかぎり、システム値の**ロケール**または`defaultProperties`ファイルで指定された**ロケール**を使用して表示されます。

`defaultProperties`で**ロケール**を変更する方法の詳細は、デフォルトのCloverETL設定の変更(p.88)を参照してください。

これらの関数のリストを次に示します。

- `bytearray` **`base64byte`**`(string` *`arg`*`);`

  `base64byte(string)`関数は、`base64`表現の文字列引数を1つ取得して、それをバイト配列に変換します。対応する関数は`byte2base64(bytearray)`関数です。

- `string` **`bits2str`**`(bytearray` *`arg`*`);`

  `bits2str(bytearray)`関数は、バイト配列を取得して、それを2つの文字(`0`または`1`)で構成される文字列に変換します。各バイトは8文字(`0`または`1`)で表現されます。各バイトでは、最もビット数が小さいものがこれらの8文字の最初に置かれます。対応する関数は`str2bits(string)`関数です。

- `int` **`bool2num`**`(boolean` *`arg`*`);`

  `bool2num(boolean)`関数は、ブール引数を1つ取得し、それを整数の1 (引数がtrueの場合)または整数の0 (引数がfalseの場合)に変換します。対応する関数は`num2bool(<numeric type>)`関数です。

- `<numeric type>` **`bool2num`**`(boolean` *`arg`*`, typename` *`<numeric type>`*`);`

  `bool2num(boolean, typename)`関数は、2つの引数を受け入れます。最初はブールで、もう1つは任意の数値データ型の名前です。これらを取得し、最初の引数を2番目の引数で指定された`numeric`表現の対応する`1`または`0`に変換します。関数の戻り型は2番目の引数と同じです。対応する関数は`num2bool(<numeric type>)`関数です。

- `string` **`byte2base64`**`(bytearray` *`arg`*`);`

  `byte2base64(bytearray)`関数は、バイト配列を取得して、それを`base64`表現の文字列に変換します。対応する関数は`base64byte(string)`関数です。

- `string` **`byte2hex`**`(bytearray` *`arg`*`);`

  `byte2hex(bytearray)`関数は、バイト配列を取得して、それを`hexadecimal`表現の文字列に変換します。対応する関数は`hex2byte(string)`関数です。

- long **date2long**(date *arg*);

  date2long(date)関数は、日付引数を1つ取得し、それをlong型に変換します。その値は、January 1, 1970, 00:00:00 GMTから引数として指定した日付までに経過したミリ秒と等しくなります。対応する関数はlong2date(long)関数です。

- int **date2num**(date *arg*, unit *timeunit*);

  date2num(date, unit)関数は2つの引数を受け入れます。1番目は日付で、もう1つは任意の時間単位です。単位は、year、month、week、day、hour、minute、second、millisecのいずれかです。単位は定数として指定する必要があります。エッジを介して取得することも、変数として設定することもできません。この関数はこれらの2つの引数を取得し、それらを整数に変換します。日付に時間単位が含まれている場合、これは整数値として返されます。含まれていない場合、関数は0を返します。月には0から始まる番号が付けられます。したがって、date2num(2008-06-12, month)は5を返します。また、date2num(2008-06-12, hour)は0を返します。

- string **date2str**(date *arg*, string *pattern*);

  date2str(date, string)関数は、日付と文字列の2つの引数を受け入れます。関数はこれらを取得し、2番目の引数として指定されているpatternに従って日付を変換します。したがって、date2str(2008-06-12,"dd.MM.yyyy")は文字列"12.6.2008"を返します。対応する関数はstr2date(string, string)関数です。

- string **get_field_name**(record *argRecord*, integer *index*);

  get_field_name(record, integer)関数は、2つの引数(recordおよびinteger)を受け入れます。関数はこれらを取得し、指定されたインデックスの付いたフィールドの名前を返します。フィールドには0から始まる番号が付けられます。

- string **get_field_type**(record *argRecord*, integer *index*);

  get_field_type(record, integer)関数は、2つの引数(recordおよびinteger)を受け入れます。この関数はこれらを取得し、指定されたインデックスの付いたフィールドの型を返します。フィールドには0から始まる番号が付けられます。

- bytearray **hex2byte**(string *arg*);

  hex2byte(string)関数は、hexadecimal表現の文字列引数を1つ取得して、それをバイト配列に変換します。対応する関数はbyte2hex(bytearray)関数です。

- date **long2date**(long *arg*);

  long2date(long)関数は、long引数を1つ取得して、それを日付に変換します。これはミリ秒数の引数をJanuary 1, 1970, 00:00:00 GMTに追加し、結果を日付として返します。対応する関数はdate2long(date)関数です。

- bytearray **long2pacdecimal**(long *arg*);

  long2pacdecimal(long)関数は、longデータ型の引数を1つ取得して、パック10進数表現の値を返します。これはpacdecimal2long(bytearray)関数に対応する関数です。

- bytearray **md5**(bytearray *arg*);

  md5(bytearray)関数は、バイト配列で構成される1つの引数を受け入れます。これはこの引数を取得して、MD5ハッシュ値を計算します。

- bytearray **md5**(string *arg*);

  md5(string)関数は、文字列データ型の1つの引数を受け入れます。これはこの引数を取得して、MD5ハッシュ値を計算します。

- boolean **num2bool**(<numeric type> *arg*);

  num2bool(<numeric type>)関数は、1または0を表す任意の数値データ型の引数を1つ取得して、booleanのtrueまたはfalseをそれぞれ返します。

- <numeric type> **num2num**(<numeric type> *arg*, typename *<numeric type>*);

  num2num(<numeric type>, typename)関数は、2つの引数を受け入れます。最初は任意の数値データ型、2番目は任意の数値データ型の名前です。これらを取得し、最初の引数値を、2番目の引数として指定されているnumeric型の値に変換します。関数の戻り型は2番目の引数と同じです。変換は、情報を失わずに実行可能な場合にのみ成功し、そうでない場合は関数から例外がスローされます。したがって、num2num(25.4, int)は例外をスローしますが、num2num(25.0, int)は25を返します。

- string **num2str**(<numeric type> *arg*);

  num2str(<numeric type>)関数は、任意の数値データ型の引数を1つ取得し、それを文字列表現に変換します。したがって、num2str(20.52)は"20.52"を返します。

- string **num2str**(<numeric type> *arg*, int *radix*);

  num2str(<numeric type>, int)関数は、2つの引数を受け入れます。最初は任意の数値データ型、2番目は整数です。これは、これらの2つの引数を取得して、最初の引数をradixベースの数値システムの文字列表現に変換します。したがって、num2str(31, 16)は"1F"を返します。

- string **num2str**(<numeric type> *arg*, string *format*);

  num2str(<numeric type>, string)関数は、2つの引数を受け入れます。最初は任意の数値データ型、2番目は文字列です。これらの2つの引数を取得し、2番目の引数として指定された書式を使用して、最初の引数をその文字列表現に変換します。

- long **pacdecimal2long**(bytearray *arg*);

  pacdecimal2long(bytearray)関数は、long数値のパック10進数表現を表すバイト配列の引数を1つ取得します。これは値をlongデータ型として返します。これはlong2pacdecimal(long)関数に対応する関数です。

- bytearray **sha**(bytearray *arg*);

  sha(bytearray)関数は、バイト配列で構成される1つの引数を受け入れます。これはこの引数を取得して、SHAハッシュ値を計算します。

- bytearray **sha**(string *arg*);

  sha(string)関数は、文字列データ型の1つの引数を受け入れます。これはこの引数を取得して、SHAハッシュ値を計算します。

- bytearray **str2bits**(string *arg*);

  str2bits(string)関数は、文字列引数を1つ取得して、それをバイト配列に変換します。対応する関数はbits2str(bytearray)関数です。文字列は、次の文字で構成されます。各文字は1またはその他の任意の文字です。文字列で、文字1はそれぞれ1ビットに変換され、その他のすべての文字(0のみでなく、a、z、/なども)は0ビットに変換されます。文字列の文字数が8の整数倍でない場合、文字列が右から0の文字で埋められます。その後、文字列は、その文字数が8の整数倍であるかのようにバイトの配列に変換されます。

- boolean **str2bool**(string *arg*);

  str2bool(string)関数は、文字列引数を1つ取得し、それを対応するブール値に変換します。文字列は、"TRUE"、"true"、"T"、"t"、"YES"、"yes"、"Y"、"y"、"1"、"FALSE"、"false"、"F"、"f"、"NO"、"no"、"N"、"n"、"0"のいずれかです。文字列はブール値のtrueまたはブール値のfalseに変換されます。

- `date` **`str2date`**`(string` *`arg`*`, string` *`pattern`*`);`

  `str2date(string, string)`関数は、2つの文字列引数を受け入れます。これらを取得し、最初の文字列を、2番目の引数として指定されているpatternに従って日付に変換します。patternは、最初の引数の構造と対応している必要があります。したがって、`str2date("12.6.2008", "dd.MM.yyyy")`は`2008-06-12`という日付を返します。

- `date` **`str2date`**`(string` *`arg`*`, string` *`pattern`*`, string` *`locale`*`, boolean` *`lenient`*`);`

  `str2date(string, string, string, boolean)`関数は、3つの文字列引数および1つのブールを受け入れます。これはその引数を取得し、最初の文字列を、2番目の引数として指定されているpatternに従って日付に変換します。patternは、最初の引数の構造と対応している必要があります。したがって、`str2date("12.6.2008", "dd.MM.yyyy")`は`2008-06-12`という日付を返します。3番目の引数は、日付のロケールを定義します。4番目の引数は、日付の解釈を緩くするか(true)厳しくするか(false)どうかを指定します。これがtrueの場合、関数は日付の解釈がロケールまたはパターン(あるいはその両方)に一致しない場合でも日付の解釈を試みます。この関数の引数が3つのみの場合、3番目はlocale (文字列の場合)またはlenient (booleanの場合)として解釈されます。

- `<numeric type>` **`str2num`**`(string` *`arg`*`);`

  `str2num(string)`関数は、文字列引数を1つ取得し、それを対応する数値に変換します。したがって、関数がdoubleの戻り型で宣言されている場合、`str2num("0.25")`は`0.25`を返しますが、integerの戻り型で宣言されている場合は、同じものから例外がスローされます。この関数の戻り型には任意の数値型が可能です。

- `<numeric type>` **`str2num`**`(string` *`arg`*`, typename` *`<numeric type>`*`);`

  `str2num(string, typename)`関数は、2つの引数を受け入れます。最初は文字列、2番目は任意の数値データ型の名前です。これは最初の引数を取得し、2番目の引数で指定されたnumericデータ型の対応する値を返します。関数の戻り型は2番目の引数と同じです。

- `<numeric type>` **`str2num`**`(string` *`arg`*`, typename` *`<numeric type>`*`, int` *`radix`*`);`

  `str2num(string, typename, int)`関数は、文字列、任意の数値データ型の名前および整数の3つの引数を受け入れます。これはradixベースの数値システムで表現されたかのように最初の引数を取得し、2番目の引数で指定されたnumericデータ型の対応する値を返します。戻り型は2番目の引数と同じです。3番目の引数は、2番目の引数がnumberデータ型の場合は`10`または`16` (ただし、文字列が数値の10進数文字列表現か16進数文字列表現かは、その文字列形式のみで判定できるため、radixを指定する必要はありません)、2番目の引数がdecimal型の場合は`10`、および2番目の引数がintおよびlong型の場合は、`Character.MIN_RADIX`から`Character.MAX_RADIX`までの任意の整数が可能です。

- `<numeric type>` **`str2num`**`(string` *`arg`*`, typename` *`<numeric type>`*`, string` *`format`*`);`

  `str2num(string, typename, string)`関数は、3つの引数を受け入れます。最初は数値に変換する文字列、2番目は戻り数値データ型の名前、3番目は最初の引数で使用される数値の文字列表現の書式です。2番目の引数として指定される型名は、エッジとして取得することも変数として定義することもできません。関数で直接指定する必要があります。この関数は最初の引数を取得し、それをシステム値localeを使用してformatと比較して、2番目の引数で指定されたデータ型の数値を返します。

- `<numeric type>` **`str2num`**`(string` *`arg`*`, typename` *`<numeric type>`*`, string` *`format`*`, string` *`locale`*`);`

  `str2num(string, typename, string, string)`関数は、4つの引数を受け入れます。最初は数値に変換する文字列、2番目は戻り数値データ型の名前、3番目は最初の引数で使用される数値の文字列表現の書式、4番目は形式を適用するときに使用するロケールです。2番目の引数として指定される型名は、エッジとして取得することも変数として定義することもできません。関数で直接指定する必要があります。この

関数は最初の引数を取得し、同時にそれをlocaleを使用してformatと比較して、2番目の引数として指定されたデータ型の数値を返します。

- `string` **`to_string`**`(<any type> arg);`

  `to_string``(<any type>)` 関数は、任意のデータ型の引数を1つ取得し、それを文字列表現に変換します。

- `returndatatype` **`try_convert`**`(<any type> from, typename returndatatype);`

  `try_convert(<any type>, typename)` 関数は、2つの引数を受け入れます。最初は任意のデータ型、2番目はその他の任意のデータ型の名前です。2番目の引数の名前は、エッジとして取得することも変数として定義することもできません。関数で直接指定する必要があります。この関数はこれらの引数を取得し、最初の引数の、指定されたデータ型への変換を試みます。変換できる場合、関数は、最初の引数を、2番目の引数として指定されたデータ型に変換します。変換できない場合、関数はnullを返します。

- `date` **`try_convert`**`(string from, datetypename date, string format);`

  `try_convert(string, nameofdatedatatype, string)` 関数は3つの引数を受け入れます。1番目は文字列データ型、2番目は日付データ型の名前、3番目は最初の引数の書式です。2番目の引数として指定されるdate語は、エッジとして取得することも変数として定義することもできません。関数で直接指定する必要があります。この関数はこれらの引数を取得し、最初の引数の日付への変換を試みます。最初の引数として指定された文字列が3番目の引数の書式に対応する場合は変換を行うことができ、日付が返されます。変換できない場合、関数はnullを返します。

- `string` **`try_convert`**`(date from, stringtypename string, string format);`

  `try_convert(date, nameofstringdatatype, string)` 関数は、3つの引数を受け入れます。最初は日付データ型、2番目は文字列データ型の名前および3番目は日付の文字列表現の書式です。2番目の引数として指定されるstring語は、エッジとして取得することも変数として定義することもできません。関数で直接指定する必要があります。この関数はこれらの引数を取得し、最初の引数を、3番目の引数で指定された書式の文字列に変換します。

- `boolean` **`try_convert`**`(<any type> from, <any type> to, string pattern);`

  `try_convert(<any type>, <any type>, string)` 関数は、3つの引数を受け入れます。2つは任意のデータ型、3番目は文字列です。この関数はこれらの引数を取得し、最初の引数の、2番目への変換を試みます。変換に成功すると、2番目の引数が最初の引数から値を受け取ります。関数はbooleanのtrueを返します。変換に失敗した場合、関数はbooleanのfalseを返し、最初と2番目の引数には元の値が保持されます。3番目の引数はオプションで、最初の2つの引数のどれかが文字列の場合にのみ使用します。たとえば、`try_convert("27.5.1942", dateA, "dd.MM.yyyy")` はtrueを返し、dateAには27 May 1942という値が入ります。

日付関数

日付を使用する場合は、日付を処理する関数を使用できます。

これらの関数では、日付または数値の書式パターンを定義する必要がある場合があります。また、ロケールもそれらの書式設定に影響する場合があります。

- 日付の書式設定または解析(あるいはその両方)の詳細は、日付と時刻の書式(p.113)を参照してください。
- ロケールの詳細は、ロケール(p.126)を参照してください。

注意

数値および日付の書式は、その他の**ロケール**が明示的に指定されていないかぎり、システム値の**ロケール**または`defaultProperties`ファイルで指定された**ロケール**を使用して表示されます。

`defaultProperties`で**ロケール**を変更する方法の詳細は、デフォルトのCloverETL設定の変更(p.88)を参照してください。

これらの関数のリストを次に示します。

- `date` **`dateadd`**`(date` *`arg`*`, <numeric type>` *`amount`*`, unit` *`timeunit`*`);`

  `dateadd(date, <numeric type>, unit)`関数は、3つの引数を受け入れます。最初は日付、2番目は任意の数値データ型および最後は任意の時間単位です。単位は、`year`、`month`、`week`、`day`、`hour`、`minute`、`second`、`millisec`のいずれかです。単位は定数として指定する必要があります。エッジを介して取得することも、変数として設定することもできません。関数は最初の引数を取得して、時間単位の数量をその引数に追加し、結果を日付として返します。`amount`および時間の`unit`は、それぞれ2番目および3番目の引数として指定します。

- `int` **`datediff`**`(date` *`later`*`, date` *`earlier`*`, unit` *`timeunit`*`);`

  `datediff(date, date, unit)`関数は3つの引数を受け入れます。2つは日付で、もう1つは時間単位です。これらの引数を取得して、最初の引数から2番目の引数を減算します。単位は、`year`、`month`、`week`、`day`、`hour`、`minute`、`second`、`millisec`のいずれかです。単位は定数として指定する必要があります。エッジを介して取得することも、変数として設定することもできません。この関数は、結果の時間の差異を3番目の引数として指定される時間単位で返します。このため、2つの日付の差異は定義された時間単位で表されます。結果は整数として表されます。したがって、`date(2008-06-18, 2001-02-03, year)`は7を返します。ただし、`date(2001-02-03, 2008-06-18, year)`は-7を返します。

- `date` **`random_date`**`(date` *`startDate`*`, date` *`endDate`*`);`

  `random_date(date, date)`関数は、2つの日付引数を受け入れ、`startDate`と`endDate`の間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値は`startDate`または`endDate`である場合もあります。ただし、`startDate`より前の日付および`endDate`より後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、`endDate`が返されるようにするには、その翌日を`endDate`として入力してください。`locale`にはシステム値が使用されます。デフォルトの`format`は`defaultProperties`ファイルで指定されます。

- `date` **`random_date`**`(date` *`startDate`*`, date` *`endDate`*`, long` *`randomSeed`*`);`

  `random_date(date, date, long)`関数は2つの日付引数と1つのlong引数を受け入れ、`startDate`と`endDate`の間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値は`startDate`または`endDate`である場合もあります。ただし、`startDate`より前の日付および`endDate`より後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、`endDate`が返される

ようにするには、その翌日をendDateとして入力してください。localeにはシステム値が使用されます。デフォルトのformatはdefaultPropertiesファイルで指定されます。3番目の引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、randomSeed値を変更することによってのみ変更できます。

- date **random_date**(date *startDate*, date *endDate*, string *format*);

  random_date(date, date, string)関数は2つの日付引数と1つの文字列引数を受け入れ、3番目の引数で指定されたformatに対応する、startDateとendDateの間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値はstartDateまたはendDateである場合もあります。ただし、startDateより前の日付およびendDateより後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、endDateが返されるようにするには、その翌日をendDateとして入力してください。localeにはシステム値が使用されます。

- date **random_date**(date *startDate*, date *endDate*, string *format*, long *randomSeed*);

  random_date(date, date, string, long)関数は2つの日付引数、1つの文字列引数および1つのlong引数を受け入れ、3番目の引数で指定されたformatに対応する、startDateとendDateの間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値はstartDateまたはendDateである場合もあります。ただし、startDateより前の日付およびendDateより後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、endDateが返されるようにするには、その翌日をendDateとして入力してください。localeにはシステム値が使用されます。4番目の引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、randomSeed値を変更することによってのみ変更できます。

- date **random_date**(date *startDate*, date *endDate*, string *format*, string *locale*);

  random_date(date, date, string, string)関数は2つの日付引数と2つの文字列引数を受け入れ、startDateとendDateの間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値は、それぞれ3番目および4番目の引数で指定されるformatおよびlocaleに対応するstartDateまたはendDateになる場合もあります。ただし、startDateより前の日付およびendDateより後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、endDateが返されるようにするには、その翌日をendDateとして入力してください。

- date **random_date**(date *startDate*, date *endDate*, string *format*, string *locale*, long *randomSeed*);

  random_date(date, date, string, string, long)関数は2つの日付引数、2つの文字列引数および1つのlong引数を受け入れ、startDateとendDateの間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値は、それぞれ3番目および4番目の引数で指定されるformatおよびlocaleに対応するstartDateまたはendDateになる場合もあります。ただし、startDateより前の日付およびendDateより後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、endDateが返されるようにするには、その翌日をendDateとして入力してください。5番目の引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、randomSeed値を変更することによってのみ変更できます。

- date **today**();

  today()関数は引数を受け入れず、現在の日付と時刻を返します。

- date **trunc**(date *arg*);

  trunc(date)関数は、日付引数を1つ取得し、同じ年、月および日を持つ日付を返しますが、時、分、秒およびミリ秒は0に設定されます。

- `long` **`trunc`**`(<numeric type>` *`arg`*`);`

  `trunc(<numeric type>)`関数は、任意の数値データ型の引数を1つ取得し、それをlong値に切り捨てた値を返します。

- `null` **`trunc`**`(list` *`arg`*`);`

  `trunc(list)`関数は、list引数を1つ取得し、その値を空にしてnullを返します。

- `null` **`trunc`**`(map` *`arg`*`);`

  `trunc(map)`関数は、マップ引数を1つ取得し、その値を空にしてnullを返します。

- `date` **`trunc_date`**`(date` *`arg`*`);`

  `trunc_date(date)`関数は、日付引数を1つ取得し、同じ時、分、秒およびミリ秒を持つ日付を返しますが、年、月および日は0に設定されます。0の日付は1970-01-01です。

算術関数

算術関数も使用できます。

- `<numeric type> abs(<numeric type> arg);`

  `abs(<numeric type>)`関数は、任意の数値データ型の引数を1つ取得し、その絶対値を返します。

- `long bit_and(<numeric type> arg1, <numeric type> arg2);`

  `bit_and(<numeric type>, <numeric type>)`関数は、任意の数値データ型の2つの引数を受け入れます。これは両方の引数の整数部分を取得し、ビット単位andに対応する数値を返します。(たとえば、`bit_and(11,7)`は3を返します。)10進数の11はビット単位の1011と表すことができ、10進数の7はビット単位の111と表すことができるため、結果は10進数の3に対応する11です。戻りデータ型は`long`ですが、その他の数値データ型に送信される場合は、その数値表現で示されます。

- `long bit_invert(<numeric type> arg);`

  `bit_invert(<numeric type>)`関数は、任意の数値データ型の1つの引数を受け入れます。これはその整数部分を取得し、ビット単位の`inverted number`に対応する数値を返します。(たとえば、`bit_invert(11)`は-12を返します。)この関数は引数に含まれるすべてのビットを反転させます。戻りデータ型は`long`ですが、その他の数値データ型に送信される場合は、その数値表現で示されます。

- `boolean bit_is_set(<numeric type> arg, <numeric type> Index);`

  `bit_is_set(<numeric type>, <numeric type>)`関数は、任意の数値データ型の2つの引数を受け入れます。これは両方の引数の整数部分を取得し、最初の引数の`Index`位置にあるビット値を特定して、`true`または`false`を返します(そのビットがそれぞれ1または0の場合)。(たとえば、`bit_is_set(11,3)`は`true`を返します。)10進数の11はビット単位の1011と表すことができ、インデックスが3であるビット(右から4番目)は1であるため、結果は`true`です。また、`bit_is_set(11,2)`は`false`を返します。

- `long bit_lshift(<numeric type> arg, <numeric type> Shift);`

  `bit_lshift(<numeric type>, <numeric type>)`関数は、任意の数値データ型の2つの引数を受け入れます。これは両方の引数の整数部分を取得し、元の数値にビットをいくつか追加したものに対応する数値を返します(左側に`Shift`の数のビットが追加され、0に設定されます。)(たとえば、`bit_lshift(11,2)`は44を返します。)10進数の11はビット単位の1011と表すことができるため、右側に2ビット(10)が追加されると、結果は10進数の44に対応する101100になります。戻りデータ型は`long`ですが、その他の数値データ型に送信される場合は、その数値データ型で示されます。

- `long bit_or(<numeric type> arg1, <numeric type> arg2);`

  `bit_or(<numeric type>, <numeric type>)`関数は、任意の数値データ型の2つの引数を受け入れます。これは両方の引数の整数部分を取得し、ビット単位orに対応する数値を返します。(たとえば、`bit_or(11,7)`は15を返します。)10進数の11はビット単位の1011と表すことができ、10進数の7は111と表すことができるため、結果は10進数の15に対応する1111です。戻りデータ型は`long`ですが、その他の数値データ型に送信される場合は、その数値データ型で示されます。

- `long bit_rshift(<numeric type> arg, <numeric type> Shift);`

  `bit_rshift(<numeric type>, <numeric type>)`関数は、任意の数値データ型の2つの引数を受け入れます。これは両方の引数の整数部分を取得し、元の数値からビット数をいくつか削除したものに対応する数値を返します(右側の`Shift`ビットが削除されます。)(たとえば、`bit_rshift(11,2)`は2を返します。)10進数の11はビット単位の1011と表すことができるため、右側の2ビットが削除されると、結果は10進数の2に対応する10になります。戻りデータ型は`long`ですが、その他の数値データ型に送信される場合は、その数値データ型で示されます。

- long **bit_set**(<numeric type> *arg1*, <numeric type> *Index*, boolean *SetBitTo1*);

  bit_set(<numeric type>, <numeric type>, boolean)関数は、3つの引数を受け入れます。最初の2つは任意の数値データ型、3番目はブールです。これは最初の2つの引数の整数部分を取得し、最初の引数のビット値のうち、2番目の引数で指定されたIndex位置のビット値を1または0に設定し(それぞれ3番目の引数がtrueまたはfalse)、結果をlong値で返します。(たとえば、bit_set(11,3,false)は3を返します。)10進数の11はビット単位の1011と表すことができ、インデックスが3であるビット(右から4番目)が0に設定されるため、結果は10進数の3に対応する11になります。また、bit_set(11,2,true)は10進数の15に対応する1111を返します。戻りデータ型はlongですが、その他の数値データ型に送信される場合は、その数値データ型で示されます。

- long **bit_xor**(<numeric type> *arg*, <numeric type> *arg*);

  bit_xor(<numeric type>, <numeric type>)関数は、任意の数値データ型の2つの引数を受け入れます。これは両方の引数の整数部分を取得し、ビット単位exclusive orに対応する数値を返します。(たとえば、bit_or(11,7)は12を返します。)10進数の11はビット単位の1011と表すことができ、10進数の7は111と表すことができるため、結果は10進数の15に対応する1100です。戻りデータ型はlongですが、その他の数値データ型に送信される場合は、その数値データ型で示されます。

- number **e**();

  e()関数は引数を受け入れず、オイラー数を返します。

- number **exp**(<numeric type> *arg*);

  exp(<numeric type>)関数は、任意の数値データ型の1つの引数を取得し、この引数の指数関数の結果を返します。

- number **log**(<numeric type> *arg*);

  log(<numeric type>)関数は、任意の数値データ型の1つの引数を取得し、この引数の自然対数の結果を返します。

- number **log10**(<numeric type> *arg*);

  log10(<numeric type>)関数は、任意の数値データ型の1つの引数を取得し、この引数の10を底とする対数の結果を返します。

- number **pi**();

  pi()関数は引数を受け入れず、パイ数を返します。

- number **pow**(<numeric type> *base*, <numeric type> *exp*);

  pow(<numeric type>, <numeric type>)関数は、任意の数値データ型の2つの引数(データ型が同じである必要はありません)を取得し、最初の引数を指数、2番目を底とする指数関数を返します。

- number **random**();

  random()関数は引数を受け入れず、0.0以上1.0未満のランダムな正の倍精度浮動小数点数を返します。

- number **random**(long *randomSeed*);

  random(long)関数は、longデータ型の1つの引数を受け入れ、0.0以上1.0未満のランダムな正の倍精度浮動小数点数を返します。引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、randomSeed値を変更することによってのみ変更できます。

- `boolean` **`random_boolean`**`();`

  `random_boolean()`関数は引数を受け入れず、ブール値`true`または`false`をランダムに生成します。これらの値が数値データ型フィールドに送信されると、値は自動的に数値表現に変換されます(それぞれ`1`または`0`)。

- `boolean` **`random_boolean`**`(long` *`randomSeed`*`);`

  `random_boolean(long)`関数は、longデータ型の1つの引数を受け入れ、ブール値`true`または`false`をランダムに生成します。これらの値が数値データ型フィールドに送信されると、値は自動的に数値表現に変換されます(それぞれ`1`または`0`)。引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、`randomSeed`値を変更することによってのみ変更できます。

- `<numeric type>` **`random_gaussian`**`();`

  `random_gaussian()`関数は引数を受け入れず、ガウス分布の戻り数値データ型の正と負の値をランダムに生成します。

- `<numeric type>` **`random_gaussian`**`(long` *`randomSeed`*`);`

  `random_gaussian(long)`関数は、longデータ型の1つの引数を受け入れ、ガウス分布の戻り数値データ型の正と負の値をランダムに生成します。引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、`randomSeed`値を変更することによってのみ変更できます。

- `int` **`random_int`**`();`

  `random_int()`関数は引数を受け入れず、正と負の整数値をランダムに生成します。

- `int` **`random_int`**`(long` *`randomSeed`*`);`

  `random_int(long)`関数は、longデータ型の1つの引数を受け入れ、正と負の整数値をランダムに生成します。引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、`randomSeed`値を変更することによってのみ変更できます。

- `int` **`random_int`**`(int` *`Minimum`*`, int` *`Maximum`*`);`

  `random_int(int, int)`関数は、整数データ型の2つの引数を受け入れ、`Minimum`以上`Maximum`以下のランダムな整数値を返します。

- `int` **`random_int`**`(int` *`Minimum`*`, int` *`Maximum`*`, long` *`randomSeed`*`);`

  `random_int(int, int, long)`関数は、3つの引数を受け入れます。最初の2つは整数データ型、3番目はlongです。この関数はこれらを取得し、`Minimum`以上`Maximum`以下のランダムな整数値を返します。3番目の引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、`randomSeed`値を変更することによってのみ変更できます。

- `long` **`random_long`**`();`

  `random_long()`関数は引数を受け入れず、正と負のlong値をランダムに生成します。

- `long` **`random_long`**`(long` *`randomSeed`*`);`

  `random_long(long)`関数は、longデータ型の1つの引数を受け入れ、正と負のlong値をランダムに生成します。引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、`randomSeed`値を変更することによってのみ変更できます。

- `long` **`random_long`**`(long` *`Minimum`*`, long` *`Maximum`*`);`

  `random_long(long, long)`関数は、longデータ型の2つの引数を受け入れ、`Minimum`以上`Maximum`以下のランダムなlong値を返します。

- `long` **`random_long`**`(long` *`Minimum`*`, long` *`Maximum`*`, long` *`randomSeed`*`);`

  `random_long(long, long, long)`関数は、longデータ型の3つの引数を受け入れ、`Minimum`以上`Maximum`以下のランダムなlong値を返します。引数は、グラフのすべての実行で生成される値が同じになるようにします。生成される値は、`randomSeed`値を変更することによってのみ変更できます。

- `long` **`round`**`(<numeric type>` *`arg`*`);`

  `round(<numeric type>)`関数は、任意の数値データ型の1つの引数を取得し、この引数に最も近いlongを返します。

- `number` **`sqrt`**`(<numeric type>` *`arg`*`);`

  `sqrt(<numeric type>)`関数は、任意の数値データ型の1つの引数を取得し、この引数の平方根を返します。

文字列関数

文字列を操作する関数もあります。

文字列を操作する関数では、日付または数値の書式パターンを定義する必要がある場合があります。

- 日付の書式設定または解析(あるいはその両方)の詳細は、日付と時刻の書式(p.113)を参照してください。
- 数値データ型の書式設定または解析(あるいはその両方)の詳細は、数値の書式(p.120)を参照してください。
- ロケールの詳細は、ロケール(p.126)を参照してください。

注意

数値および日付の書式は、その他の**ロケール**が明示的に指定されていないかぎり、システム値の**ロケール**または`defaultProperties`ファイルで指定された**ロケール**を使用して表示されます。

`defaultProperties`で**ロケール**を変更する方法の詳細は、デフォルトのCloverETL設定の変更(p.88)を参照してください。

これらの関数のリストを次に示します。

- `string` **`char_at`**`(string` *`arg`*`, <numeric type>` *`index`*`);`

  `char_at(string, <numeric type>)`関数は、2つの引数を受け入れます。最初は文字列、2番目は任意の数値データ型です。これは文字列を取得し、`index`で指定された位置にある文字を返します。

- `string` **`chop`**`(string` *`arg`*`);`

  `chop(string)`関数は1つの文字列引数を受け入れます。関数はこの引数を取得し、引数で指定された文字列の末尾から改行文字および復帰文字を削除して、これらの文字が含まれない新しい文字列を返します。

- `string` **`chop`**`(string` *`arg1`*`, string` *`arg2`*`);`

  `chop(string, string)`関数は2つの文字列引数を受け入れます。最初の引数を取得し、2番目の引数で指定された文字列を最初の引数の末尾から削除して、2番目の引数で指定された文字列が含まれない最初の文字列引数を返します。

- `string` **`concat`**`(<any type>` *`arg1`*`, ... ..., <any type>` *`argN`*`);`

  `concat(<any type>, ..., <any type>)`関数は、任意のデータ型の無制限の数の引数を受け入れます。ただし、これらは同じである必要はありません。これは、取得した引数を連結して返します。一部の引数が文字列ではない場合、それらは連結が行われる前に文字列表現に変換されます。プラス記号を使用して引数を連結することもできますが、引数が3つ以上ある場合は、この関数を使用する方が高速です。

- `int` **`count_char`**`(string` *`arg`*`, string` *`character`*`);`

  `count_char(string, string)`関数は、2つの引数を受け入れます。最初は文字列、2番目は1つの文字です。これらを取得し、最初の引数で指定された文字列に、2番目の引数で指定された文字が出現する回数を返します。

- `list` **`cut`**`(string` *`arg`*`, list` *`list`*`);`

  `cut(string, list)`関数は、2つの引数を受け入れます。最初は文字列、2番目は数値のリストです。この関数は文字列のリストを返します。2番目の引数で指定するリストの要素の数は偶数である必要があります。list引数の隣接する数値の各ペアの整数部分は、位置(奇数位置の各数値)および長さ(偶数位置の各数値)として機能します。指定した長さの部分文字列が、最初の引数で指定された文字列の指定位置から取得されます(指定位置の文字は除く)。取得された部分文字列が文字列のリストとして返されます。たとえば、`cut("somestringasanexample",[2,3,1,5])`は`["mes","omest"]`を返します。

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`);`

  `edit_distance(string, string)` 関数は、2つの文字列引数を受け入れます。これらの文字列が互いに比較されます。比較の強度はデフォルトでは4、比較に使用するロケールのデフォルト値はシステム値、最大の差異はデフォルトでは3です。

  (詳細は、`edit_distance()` 関数の別のバージョンである次の`edit_distance(string, string, int, string, int)` 関数を参照してください。)

  この関数は、2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`, string` *`locale`*`);`

  `edit_distance(string, string, string)` 関数は、3つの引数を受け入れます。最初の2つは互いに比較される文字列で、3つ目(文字列)は比較に使用されるロケールです。デフォルトの比較の強度は4です。デフォルトの最大の差異は3です。

  (詳細は、`edit_distance()` 関数の別のバージョンである次の`edit_distance(string, string, int, string, int)` 関数を参照してください。)

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

  ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`, int` *`strength`*`);`

  `edit_distance(string, string, int)` 関数は、3つの引数を受け入れます。最初の2つは互いに比較される文字列で、3つ目(整数)は比較の強度です。比較に使用されるデフォルト・ロケールはシステム値です。最大の差異のデフォルトは3です。

  (詳細は、`edit_distance()` 関数の別のバージョンである次の`edit_distance(string, string, int, string, int)` 関数を参照してください。)

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`, int` *`strength`*`, string` *`locale`*`);`

  `edit_distance(string, string, int, string)` 関数は、4つの引数を受け入れます。最初の2つは互いに比較される文字列、3つ目(整数)は比較の強度、4つ目(文字列)は比較に使用されるロケールです。最大の差異のデフォルトは3です。

  (詳細は、`edit_distance()` 関数の別のバージョンである次の`edit_distance(string, string, int, string, int)` 関数を参照してください。)

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

  ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`, string` *`locale`*`, int` *`maxDifference`*`);`

  `edit_distance(string, string, string, int)` 関数は、4つの引数を受け入れます。最初の2つは互いに比較される文字列、3つ目(文字列)は比較に使用されるロケール、4つ目(整数)は最大の差異です。比較の強度のデフォルトは4です。

  (詳細は、`edit_distance()` 関数の別のバージョンである次の`edit_distance(string, string, int, string, int)` 関数を参照してください。)

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

  ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`, int` *`strength`*`, int` *`maxDifference`*`);`

  `edit_distance(string, string, int, int)` 関数は、4つの引数を受け入れます。最初の2つは互いに比較される文字列で、残りの2つはいずれも整数です。これらは比較の強度(3番目の引数)および最大の差異(4番目の引数)です。ロケールはデフォルトのシステム値です。

  (詳細は、`edit_distance()` 関数の別のバージョンである次の`edit_distance(string, string, int, string, int)` 関数を参照してください。)

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`, int` *`strength`*`, string` *`locale`*`, int` *`maxDifference`*`);`

  `edit_distance(string, string, int, string, int)` 関数は、5つの引数を受け入れます。最初の2つは文字列、残りの3つはそれぞれ整数、文字列および整数です。この関数は最初の2つの引数を取得し、他の3つの引数を使用してそれらを比較します。

  3番目の引数(整数)は比較の強度を指定します。これには1から4の任意の値を指定できます。

  4 (同一比較)の場合は、同一の文字のみが一致とみなされることを意味します。3の場合は(3次比較)、大文字と小文字が一致とみなされることを意味します。2の場合は(2次比較)、発音区別文字の付いた文字が一致とみなされることを意味します。また、比較の強度が1 (1次比較)の場合は、特定の記号が付いた文字が一致とみなされることを意味します。この強度の比較を指定しない`edit_distance()` 関数のその他のバージョンでは、デフォルト強度として数値4が使用されます(前述)。

  4番目の引数は文字列データ型です。これは比較に使用されるロケールです。`edit_distance()` 関数の他のバージョンでロケールが指定されなかった場合、そのデフォルト値はシステム値となります(前述)。

  5番目の引数(整数)は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を意味します。`edit_distance()` 関数の他のバージョンでこの最大の差異を指定しなかった場合は、デフォルトの最大の差異として数値3が受け入れられます(前述)。

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

実際に、この関数はCA、CZ、ES、DA、DE、ET、FI、FR、HR、HU、IS、IT、LT、LV、NL、NO、PL、PT、RO、SK、SL、SQ、SV、TRの各ロケールに実装されています。

ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- `list` **`find`**`(string` *`arg`*`, string` *`regex`*`);`

  `find(string, string)`関数は、2つの文字列引数を受け入れます。2番目の引数は正規表現(p.963)です。関数はこれらを取得し、最初の引数で指定された文字列に含まれる、正規表現パターンに対応する部分文字列のリストを返します。

- `string` **`get_alphanumeric_chars`**`(string` *`arg`*`);`

  `get_alphanumeric_chars(string)`関数は、1つの文字列引数を取得し、文字列引数に含まれる文字および数字のみを、それらが文字列に出現する順序で返します。その他の文字は削除されます。

- `string` **`get_alphanumeric_chars`**`(string` *`arg`*`, boolean` *`takeAlpha`*`, boolean` *`takeNumeric`*`);`

  `get_alphanumeric_chars(string, boolean, boolean)`関数は、3つの引数を受け入れます。1つは文字列、2つはブールです。これらを取得し、2番目の引数または3番目の引数(あるいはその両方)がtrueに設定されている場合、それぞれ文字または数字(あるいはその両方)を返します。

- `int` **`index_of`**`(string` *`arg`*`, string` *`substring`*`);`

  `index_of(string, string)`関数は、2つの文字列を受け入れます。これらを取得し、最初の引数で指定された文字列に初めて出現した`substring`のインデックスを返します。

- `int` **`index_of`**`(string` *`arg`*`, string` *`substring`*`, int` *`fromIndex`*`);`

  `index_of(string, string, int)`関数は、3つの引数を受け入れます。2つの文字列および1つの整数です。これらを取得し、3番目の引数で指定された位置にある文字から数えて初めて出現した`substring`のインデックスを返します。

- `boolean` **`is_ascii`**`(string` *`arg`*`);`

  `is_ascii(string)`関数は、1つの文字列引数を取得し、その文字列がASCII文字列としてエンコードできるか(true)そうでないか(false)に応じてブール値を返します。

- `boolean` **`is_blank`**`(string` *`arg`*`);`

  `is_blank(string)`関数は、1つの文字列引数を取得し、その文字列に空白文字が含まれているか(true)そうでないか(false)に応じてブール値を返します。

- `boolean` **`is_date`**`(string` *`arg`*`, string` *`pattern`*`);`

  `is_date(string, string)`関数は、2つの文字列引数を受け入れます。これらを取得し、2番目の引数をパターンとして最初の引数と比較します。指定した`pattern`に従って、システム値の`locale`内で有効な日付に最初の文字列を変換できる場合、関数はtrueを返します。できない場合はfalseを返します。

  (詳細は、`is_date()`関数の別のバージョンである次の`is_date(string, string, string, boolean)`関数を参照してください。)

  この関数は、3番目の引数のデフォルト値がシステム値に設定され、4番目の引数がデフォルトでfalseに設定されている、前述の`is_date(string, string, string, boolean)`関数のバリアントです。

- `boolean` **`is_date`**`(string` *`arg`*`, string` *`pattern`*`, string` *`locale`*`);`

  `is_date(string, string, string)` 関数は、3つの文字列引数を受け入れます。これらを取得し、3番目の引数(`locale`)を使用し、2番目の引数をパターンとして最初の引数と比較します。指定した`pattern`に従って、指定した`locale`内で有効な日付に最初の引数を変換できる場合、関数はtrueを返します。できない場合はfalseを返します。

  (詳細は、`is_date()` 関数の別のバージョンである次の`is_date(string, string, string, boolean)` 関数を参照してください。)

  この関数は、4番目の引数(`lenient`)のデフォルト値がデフォルトでfalseに設定されている、前述の`is_date(string, string, string, boolean)` 関数のバリアントです。

  ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- `boolean` **`is_date`**`(string` *`arg`*`, string` *`pattern`*`, boolean` *`lenient`*`);`

  `is_date(string, string, boolean)` 関数は、2つの文字列引数および1つのブールを受け入れます。

注意

CloverETLのバージョン2.8.1以上では、`lenient`引数は無視され、暗黙的に`false`に設定されます。

  この関数はこれらの引数を取得し、2番目の引数をパターンとして最初の引数と比較します。指定した`pattern`に従って、システム値の`locale`内で有効な日付に最初の文字列を変換できる場合、関数はtrueを返します。できない場合はfalseを返します。

  (詳細は、`is_date()` 関数の別のバージョンである次の`is_date(string, string, string, boolean)` 関数を参照してください。)

  この関数は、3番目の引数(`locale`)のデフォルト値がシステム値に設定されている、前述の`is_date(string, string, string, boolean)` 関数のバリアントです。

- `boolean` **`is_date`**`(string` *`arg`*`, string` *`pattern`*`, string` *`locale`*`, boolean` *`lenient`*`);`

  `is_date(string, string, string, boolean)` 関数は、3つの文字列引数および1つのブールを受け入れます。

注意

CloverETLのバージョン2.8.1以上では、`lenient`引数は無視され、暗黙的に`false`に設定されます。

  この関数はこれらの引数を取得し、3番目の引数(`locale`)を使用し、2番目の引数をパターンとして最初の引数と比較します。指定した`pattern`に従って、指定した`locale`内で有効な日付に最初の文字列を変換できる場合、関数はtrueを返します。できない場合はfalseを返します。

  ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- `boolean` **`is_integer`**`(string` *`arg`*`);`

  `is_integer(string)` 関数は、1つの文字列引数を取得し、その文字列が整数に変換できるか(true)そうでないか(false)に応じてブール値を返します。

- `boolean` **`is_long`**`(string` *`arg`*`);`

  `is_long(string)` 関数は、1つの文字列引数を取得し、その文字列がlong数値に変換できるか(true)そうでないか(false)に応じてブール値を返します。

- `boolean` **`is_number`**`(string` *`arg`*`);`

  `is_number(string)`関数は、1つの文字列引数を取得し、その文字列が倍精度浮動小数点数に変換できるか(true)そうでないか(false)に応じてブール値を返します。

- `string` **`join`**`(string` *`delimiter`*`, <any type>` *`arg1`*`, ... ..., <any type>` *`argN`*`);`

  `join(string, <any type>, ..., <any type>)`関数は、無制限の数の引数を受け入れます。最初は文字列、その他は任意のデータ型です。すべてのデータ型が同じである必要はありません。文字列ではない引数はその文字列表現に変換され、最初の引数をデリミタに使用して結合されます。

注意

この関数は、それぞれ2.7.0または2.2.0よりも前のリリースである**CloverETL Engine**および**CloverETL Designer**にすでに含まれていました。ただし、この`join()`関数の古いバージョンは、要素の各ペア間にのみデリミタが挿入される`join()`の新しいバージョンとは異なり、シーケンスの最後にも終端デリミタが追加されます。

古いリリースの`join(";",argA,argB)`は、`"argA;argB;"`という文字列を返します。**CloverETL Engine** 2.7.0および**CloverETL Designer** 2.2.0以上では、同じ式の結果は`"argA;argB"`となります(最後の`"argA;argB"`がない)。

このため、**CloverETL Engine** 2.7.0および**CloverETL Designer** 2.2.0以上でこの関数を使用する、**CloverETL Engine**および**CloverETL Designer**の古いバージョン用に作成された古いグラフを実行する場合は、グラフ内の古い`join(delimiter,...)`式を`join(delimiter,...) + delimiter`で置き換えてください。前述の場合では、古い`join(";",argA,argB)`式を新しい`join(";",argA,argB) + ";"`式で置き換える必要があります。

- `string` **`join`**`(string` *`delimiter`*`, arraytype` *`arg`*`);`

  `join(string, arraytype)`関数は、2つの引数を受け入れます。最初は文字列、その次は配列です。配列引数の要素はその文字列表現に変換され、最初の引数を要素の各ペア間のデリミタに使用して結合されます。

注意

この場合でも、**CloverETL Engine**および**CloverETL Designer**の古いバージョンでは、生成されるシーケンスの最後にデリミタが追加されますが、新しいバージョン(**CloverETL Engine** 2.7.0および**CloverETL Designer** 2.2.0以上)では最後にデリミタが追加されません。

- `string` **`left`**`(string` *`arg`*`, <numeric type>` *`length`*`);`

  `left(string, <numeric type>)`関数は、2つの引数を受け入れます。最初は文字列、2番目は任意の数値データ型です。これらを取得し、最初の引数で指定された文字列の先頭から数えて、2番目の引数で指定された長さの部分文字列を返します。

- `int` **`length`**`(structuredtype` *`arg`*`);`

  `length(structuredtype)`関数は、構造化されたデータ型の1つの引数を受け入れます。これは、`string`、`bytearray`、`list`,`map`または`record`です。これはこの引数を取得し、その引数を構成する要素の数を返します。

- `string` **`lowercase`**`(string` *`arg`*`);`

  `lowercase(string)`関数は、1つの文字列引数を取得し、それを小文字のみに変換した別の文字列を返します。

- `string` **`metaphone`**`(string` *`arg`*`, int` *`maxLength`*`);`

  `metaphone(string, int)`関数は、1つの文字列引数および最大長を表す1つの整数を受け入れます。この関数はこれらの引数を取得し、最初の引数のmetaphoneコードを指定された最大長で返します。デフォルトの最大長は4です。詳細は、次のサイトを参照してください。www.lanw.com/java/phonetic/default.htm

- `string` **`NYSIIS`**`(string` *`arg`*`);`

  `NYSIIS(string)`関数は、1つの文字列引数を取得し、その引数のNew York State Identification and Intelligence System表音コードを返します。詳細は、次のサイトを参照してください。
  http://en.wikipedia.org/wiki/New_York_State_Identification_and_Intelligence_System

- `string` **`random_string`**`(int` *`minLength`*`, int` *`maxLength`*`);`

  `random_string(int, int)`関数は2つの整数引数を取得し、長さが`minLength`と`maxLength`の間である、小文字で構成される文字列を返します。これらの結果の文字列は、レコードおよびフィールドについてランダムに生成されます。各レコードおよび各フィールドの両方について異なる場合もあります。長さが`minLength`または`maxLength`に等しくなることもありますが、`minLength`より短くなることも`maxLength`より長くなることもありません。

- `string` **`random_string`**`(int` *`minLength`*`, int` *`maxLength`*`, long` *`randomSeed`*`);`

  `random_string(int, int, long)`関数は2つの整数引数と1つのlong引数を取得し、長さが`minLength`と`maxLength`の間である、小文字で構成される文字列を返します。これらの結果の文字列は、レコードおよびフィールドについてランダムに生成されます。各レコードおよび各フィールドの両方について異なる場合もあります。長さが`minLength`または`maxLength`に等しくなることもありますが、`minLength`より短くなることも`maxLength`より長くなることもありません。引数は、グラフのすべての実行で生成される値が同じになるようにします。

- `string` **`remove_blank_space`**`(string` *`arg`*`);`

  `remove_blank_space(string)`関数は、1つの文字列引数を取得し、空白を削除した別の文字列を返します。

- `string` **`remove_diacritic`**`(string` *`arg`*`);`

  `remove_diacritic(string)`関数は、1つの文字列引数を取得し、発音区別文字を削除した別の文字列を返します。

- `string` **`remove_nonascii`**`(string` *`arg`*`);`

  `remove_non ascii(string)`関数は、1つの文字列引数を取得し、非ASCII文字を削除した別の文字列を返します。

- `string` **`remove_nonprintable`**`(string` *`arg`*`);`

  `remove_nonprintable(string)`関数は、1つの文字列引数を取得し、印刷不能文字を削除した別の文字列を返します。

- `string` **`replace`**`(string` *`arg`*`, string` *`regex`*`, string` *`replacement`*`);`

  `replace(string, string, string)`関数は、文字列、正規表現(p.963)および置換の3つの文字列引数を取得し、文字列内にあるすべての正規表現の一致を指定した置換文字列で置換します。文字列の正規表現に一致するすべての部分が置換されます。置換文字列で後方参照を使用して、一致したテキストを参照することもできます。一致全体に対する後方参照は$0と指定します。キャプチャするカッコを使用している場合は、特定のグループを$1、$2、$3などのように参照できます。

  `replace("Hello","[Ll]","t")`は"Hetto"を返します。

  `replace("Hello","[Ll]",$0)`は"HeHelloHelloo"を返します。

- `string` **`right`**`(string` *`arg`*`, <numeric type>` *`length`*`);`

  `right(string, <numeric type>)`関数は、2つの引数を受け入れます。最初は文字列、2番目は任意の数値データ型です。これらを取得し、最初の引数で指定された文字列の最後から数えて、2番目の引数で指定された長さの部分文字列を返します。

- `string` **`soundex`**`(string` *`arg`*`);`

  `soundex(string)`関数は、1つの文字列引数を取得し、その文字列を別の文字列に変換します。生成される文字列は、引数として指定された文字列の最初の文字と3つの数字で構成されます。3つの数字は文字列に含まれる子音に基づき、数字は発音が似ている子音に対応しています。したがって、`soundex("word")`は`"w600"`を返します。

- `list` **`split`**`(string` *`arg`*`, string` *`regex`*`);`

  `split(string, string)`関数は、2つの文字列引数を受け入れます。2番目は正規表現(p.963)です。これが最初の文字列引数で検索され、検出されると、文字列がその正規表現の文字または部分文字列の間にある部分に分割されます。この文字列から生成された部分がリストとして返されます。したがって、`split("abcdefg", "[ce]")`は`["ab", "d", "fg"]`を返します。

- `string` **`substring`**`(string` *`arg`*`, <numeric type>` *`fromIndex`*`, <numeric type>` *`length`*`);`

  `substring(string, <numeric type>, <numeric type>)`関数は、3つの引数を受け入れます。最初は文字列、その他の2つは任意の数値データ型です。2つの数値型が同じである必要はありません。この関数は引数を取得し、2番目の引数で定義される位置から`length`文字を取得して、元の文字列から定義された長さの部分文字列を取得して返します。2番目および3番目の引数が整数ではない場合は、それらの整数部分のみが関数で使用されます。したがって、`substring("text", 1.3, 2.6)`は`"ex"`を返します。

- `string` **`translate`**`(string` *`arg`*`, string` *`searchingSet`*`, string` *`replaceSet`*`);`

  `translate(string, string, string)`関数は、3つの文字列引数を受け入れます。2番目および3番目の引数両方で文字数が同じである必要があります。2番目の引数として指定された文字列に含まれる文字が最初の引数として指定された文字列内で検出されると、3番目の引数として指定された文字列から取得される文字でその文字が置換されます。3番目の文字列の文字は、2番目の文字列の文字と同じ位置にある必要があります。したがって、`translate("hello", "leo", "pii")`は`"hippi"`を返します。

- `string` **`trim`**`(string` *`arg`*`);`

  `trim(string)`関数は、1つの文字列引数を取得し、先頭および末尾のスペース文字を削除した別の文字列を返します。

- `string` **`uppercase`**`(string` *`arg`*`);`

  `uppercase(string)`関数は、1つの文字列引数を取得し、それを大文字のみに変換した別の文字列を返します。

コンテナ関数

コンテナ(list、map、record)を操作する場合は、次の関数を使用します。

- `list` **`copy`**`(list arg, list` *`arg`*`);`

  `copy(list, list)` 関数は、2つの引数を受け入れます。引数のそれぞれはリストです。すべてのリストの要素は同じデータ型とする必要があります。この関数は2番目の引数を取得し、最初のリストの末尾にそれを追加して、生成される新しいリストを返します。したがって、生成されるリストは、引数で指定された両文字列の結合です。また、最初の引数として指定されるリストもこの新しい値に変更されます。

- `boolean` **`insert`**`(list` *`arg`*`, <numeric type>` *`position`*`, <element type>` *`newelement1`*`, ... ..., <element type>` *`newelementN`*`);`

  `insert(list, <numeric type>, <element type>1, ..., <element type>N)` 関数は、最初にリスト、2番目に任意の数値データ型、それ以外に任意のデータ型(このデータ型はすべての引数で同じ)の引数を受け入れます。同時に、このデータ型は、リストの要素のデータ型と等しくなります。この関数は、3番目の引数から始まる、関数に含まれる要素を取得し(3番目の引数を含む)、2番目の引数の整数部分で定義される位置から始まるリストに要素を1つずつ挿入します。最初の引数として指定されるリストはこの新しい値に変更されます。この関数は、成功するとtrueを返します。失敗した場合はfalseを返します。リスト要素には0から始まるインデックスが付きます。

- `<element type>` **`poll`**`(list` *`arg`*`);`

  `poll(list)` 関数は、listデータ型の1つの引数を受け入れます。この引数を取得し、リストから最初の要素を削除して、その要素を返します。引数として指定されるリストがこの新しい値に変更されます(削除された最初の要素は含まれません)。

- `<element type>` **`pop`**`(list` *`arg`*`);`

  `pop(list)` 関数は、listデータ型の1つの引数を受け入れます。この引数を取得し、リストから最後の要素を削除して、その要素を返します。引数として指定されるリストがこの新しい値に変更されます(削除された最後の要素は含まれません)。

- `boolean` **`push`**`(list` *`arg`*`, <element type>` *`list_element`*`);`

  `push(list,<element type>)` 関数は、2つの引数を受け入れます。最初はリスト、2番目は任意のデータ型です。ただし、2番目の引数は、リストの各要素と同じデータ型とする必要があります。この関数は2番目の引数を取得し、最初の引数の末尾にそれを追加します。最初の引数として指定されるリストはこの新しい値に変更されます。この関数は、成功するとtrueを返します。失敗した場合はfalseを返します。

- `list` **`remove`**`(list` *`arg`*`, <numeric type>` *`position`*`);`

  `remove(list, <numeric type>)` 関数は、2つの引数を受け入れます。最初はリスト、2番目は任意の数値データ型です。この関数は2番目の引数の整数部分を取得し、指定された位置のリストの要素を削除します。最初の引数として指定されるリストがこの新しい値に変更されます(削除された要素は含まれません)。関数はこの新しいリストを返します。リスト要素には0から始まるインデックスが付きます。

- `boolean` **`remove_all`**`(list` *`arg`*`);`

  `remove_all(list)` 関数は、1つのlist引数を受け入れます。この関数はこの引数を取得し、リストを空にします。これはブール値を返します。引数として指定されるリストは空のリストに変更されます。

- `list` **`reverse`**`(list` *`arg`*`);`

  `reverse(list)`関数は、listデータ型の1つの引数を受け入れます。これはこの引数を取得し、リストの要素の順序を逆にして、その新しいリストを返します。引数として指定されるリストはこの新しい値に変更されます。

- `list` **`sort`**`(list` *`arg`*`);`

  `sort(list)`関数は、listデータ型の1つの引数を受け入れます。これはこの引数を取得し、リストの値に従ってその要素を昇順でソートして、その新しいリストを返します。引数として指定されるリストはこの新しい値に変更されます。

その他の関数

残りはその他の関数と呼ぶことができます。これらを次に示します。

- `void **breakpoint**();`

  `breakpoint()`関数は引数を受け入れず、すべてのグローバル変数およびローカル変数を出力します。

- `<any type> **iif**(boolean *con*, <any type> *iftruevalue*, <any type> *iffalsevalue*);`

  `iif(boolean, <any type>, <any type>)`関数は3つの引数を受け入れます。1つはブールで、2つは任意のデータ型です。2つの引数のデータ型および戻り型は同じです。

  この関数は最初の引数を取得し、最初の引数がtrueの場合は2番目の引数を返し、falseの場合は3番目の引数を返します。

- `boolean **isnull**(<any type> *arg*);`

  `isnull(<any type>)`関数は1つの引数を取得し、その引数がnull (true)であるかそうでないか(false)に応じてブール値を返します。引数のデータ型は任意です。

重要

`string`データ・フィールドのメタデータの「**Null value**」プロパティを空でない文字列に設定すると、`isnull()`関数はそのような文字列に適用された場合に`true`を返します。また、空のフィールドで適用された場合は`false`を返します。

たとえば、`field1`で「**Null value**」プロパティを`"<null>"`に設定した場合、`isnull($0.field1)`は、`field1`の値が`"<null>"`であるレコードでは`true`を返し、その他のレコードでは、レコードが空である場合でも`false`を返します。

詳細は、Null value(p.164)を参照してください。

- `<any type> **nvl**(<any type> *arg*, <any type> *default*);`

  `nvl(<any type>, <any type>)`関数は、任意のデータ型の2つの引数を受け入れます。両方の引数が同じ型である必要があります。最初の引数がnullでない場合、関数は引数の値を返します。nullである場合、関数は2番目の引数で指定されたデフォルト値を返します。

- `<any type> **nvl2**(<any type> *arg*, <any type> *arg_for_non_null*, <any type> *arg_for_null*);`

  `nvl2(<any type>, <any type>, <any type>)`関数は、任意のデータ型の3つの引数を受け入れます。このデータ型は、すべての引数および戻り値で同じである必要があります。最初の引数がnullでない場合、関数は2番目の引数の値を返します。最初の引数がnullの場合、関数は3番目の引数の値を返します。

- `void **print_err**(<any type> *message*);`

  `print_err(<any type>)`関数は、任意のデータ型の1つの引数を受け入れます。この引数を取得し、エラー出力に`message`を出力します。

注意

この関数を**CloverETL Server**上で実行されているグラフで使用する場合、`message`は**サーバー**のログ(たとえば、**Tomcat**のログ)に保存されます。かわりに`print_log()`関数を使用してください。これは、グラフが**CloverETL Server**上で実行されている場合でも、エラー・メッセージをコンソールに記録します。

- `void` **`print_err`**`(<any type>` *`message`*`, boolean` *`printLocation`*`);`

  `print_err(type, boolean)`関数は、2つの引数を受け入れます。最初は任意のデータ型、2番目はブールです。これらを取得し、`message`およびエラーの場所(2番目の引数がtrueの場合)を出力します。

注意

この関数を**CloverETL Server**上で実行されているグラフで使用する場合、`message`は**サーバー**のログ(たとえば、**Tomcat**のログ)に保存されます。かわりに`print_log()`関数を使用してください。これは、グラフが**CloverETL Server**上で実行されている場合でも、エラー・メッセージをコンソールに記録します。

- `void` **`print_log`**`(level` *`loglevel`*`, <any type>` *`message`*`);`

  `print_log(level, <any type>)`関数は、2つの引数を受け入れます。最初は2番目の引数で指定されたmessageのログ・レベル、2番目は任意のデータ型です。最初の引数は、`debug`、`info`、`warn`、`error`、`fatal`のいずれかです。ログ・レベルは定数として指定する必要があります。エッジを介して取得することも、変数として設定することもできません。関数は引数を取得し、`message`をログ出力に送信します。

注意

この関数は、特に**CloverETL Server**上で実行されるグラフでは、**サーバー**のログ(たとえば、**Tomcat**のログ)にエラー・メッセージを記録する`print_err()`関数のかわりに使用する必要があります。`print_err()`とは異なり、`print_log()`は、グラフが**CloverETL Server**上で実行されている場合でも、エラー・メッセージをコンソールに記録します。

- `void` **`print_stack`**`();`

  `print_stack()`関数は引数を受け入れず、スタックからのすべての変数を出力します。

- `void` **`raise_error`**`(string` *`message`*`);`

  `raise_error(string)`関数は、1つの文字列引数を取得し、引数として指定された`message`を含むエラーをスローします。

ディクショナリ関数

CTL1は、stringデータ型のディクショナリ・エントリを操作できる関数を提供します。

注意

これらの関数を使用すると、グラフに定義されていないエントリも操作できます。

最初にディクショナリ値をエントリに書き込み、ディクショナリの読取り用関数を使用して、この値にアクセスします。

- `string` **`read_dict`**`(string` *`name`*`);`

  この関数はディクショナリを取得し、nameで指定されるエントリを選択して、文字列データ型のその値を返します。

- `string` **`dict_get_str`**`(string` *`name`*`);`

  この関数はディクショナリを取得し、nameで指定されるエントリを選択して、文字列データ型のその値を返します。

- `void` **`write_dict`**`(string` *`name`*`, string` *`value`*`);`

  この関数は、ディクショナリを取得し、関数の最初の引数の指定に従い、文字列データ型の新規エントリを書き込むか既存エントリを更新し、そのエントリに、2番目の引数で指定されている、同じく文字列データ型の値を割り当てます。

- `boolean` **`dict_put_str`**`(string` *`name`*`, string` *`value`*`);`

  この関数は、ディクショナリを取得し、関数の最初の引数の指定に従い、文字列データ型の新規エントリを書き込むか既存エントリを更新し、そのエントリに、2番目の引数で指定されている、同じく文字列データ型の値を割り当てます。

- `void` **`delete_dict`**`(string` *`name`*`);`

  この関数はディクショナリを取得し、指定されたnameを持つプロパティを削除します。

現在は、文字列ディクショナリ型のみを操作できます。このため、heightMinプロパティの値にアクセスするには、次のCTLコードを使用する必要があります。

```
value = read_dict("heightMin");
```

参照表関数

グラフでは参照表も使用します。これらをCTLで使用するには、参照表のIDを指定し、それを`lookup()`、`lookup_next()`、`lookup_found()`または`lookup_admin()`関数の引数として入力します。

注意

`lookup_admin()`関数は**CloverETL**のバージョン3.0以上では機能しないため、コードから削除できます。

警告

CTLテンプレートの`init()`、`preExecute()`または`postExecute()`関数では、次に示す関数は使用できません。

参照表に対して行う操作に応じた、5つのオプションがあります。参照表の作成、指定したキーに関連する参照表からの指定したフィールド名の値の取得、参照表からの指定したフィールド名の次の値の取得、(レコードが重複している場合は)同じフィールド名の値を持つレコード数のカウント、または参照表の削除を行うことができます。

ここで、後述の関数でのキーは、(セミコロンではなく)カンマで区切られたフィールド名の値のシーケンスとなります。したがって、`keyValue`は`keyValuePart1,keyValuePart2,...,keyValuePartN`のような形式となります。

前述の5つのオプションは次のとおりです。

- `lookup_admin(<lookup ID>,init)`[1)]

  この関数は、指定された参照表を初期化します。

- `lookup(<lookup ID>,keyValue).<field name>`

  この関数は、この関数の2番目の引数として指定された値に等しいキー値を持つ最初のレコードを検索し、`<field name>`の値を返します。ここで、`<field name>`は参照表メタデータのフィールドです。

- `lookup_next(<lookup ID>).<field name>`

  `lookup()`関数の呼出し後、`lookup_next()`関数は、`lookup()`関数の2番目の引数として指定された値に等しいキー値を持つ次のレコードを検索し、`<field name>`値を返します。ここで、`<field name>`は参照表のフィールドです。

- `lookup_found(<lookup ID>)`

  `lookup()`関数の呼出し後、`lookup_found()`関数は、`lookup()`関数の2番目の引数として指定された値に等しいキー値を持つレコードの数を返します。

- `lookup_admin(<lookup ID>,free)`[1)]

  この関数は、指定された参照表を削除します。

説明:

1) これらの関数は**CloverETL**のバージョン3.0以上では機能しないため、コードから削除できます。

警告

CTL1の`lookup_found(<lookup ID>)`関数の使用は一般的にはお薦めしません。

これは、この式が多数のレコードが含まれている可能性がある参照表全体でレコードを検索するためです。

ループ内では、次の2つの関数をペアで使用することをお薦めします。

```
lookup(<lookup ID>,keyValue).<field name>
```

```
lookup_next(<lookup ID>).<field name>
```

特にDB参照表では、指定したキー値を持つレコードの実際の数ではなく、-1を返す可能性があります(**最大キャッシュ・サイズ**をゼロ以外の値に設定しなかった場合)。

シーケンス関数

グラフではシーケンスも使用します。これらをCTLで使用するには、シーケンスのIDを指定し、それを`sequence()`関数の引数として入力します。

警告

CTLテンプレートの`init()`、`preExecute()`または`postExecute()`関数では、次に示す関数は使用できません。

シーケンスに対して行う操作に応じた、3つのオプションがあります。現在のシーケンス番号を取得、次のシーケンス番号を取得またはシーケンス番号を初期番号値にリセットできます。

前述の3つのオプションは次のとおりです。

```
sequence(<sequence ID>).current

sequence(<sequence ID>).next

sequence(<sequence ID>).reset
```

これらの式は整数値を返しますが、long値または文字列値を取得する必要がある場合もあります。この場合、次のいずれかの方法を使用します。

```
sequence(<sequence ID>,long).current

sequence(<sequence ID>,long).next

sequence(<sequence ID>,string).current

sequence(<sequence ID>,string).next
```

カスタムCTL関数

準備済のCTL関数の他に、独自のCTL関数を作成できます。このことを行うには、カスタムCTL関数を定義する独自のコードを記述し、プラグインを指定する必要があります。

各カスタムCTL関数ライブラリは、次のものから派生/継承する必要があります。

`org.jetel.interpreter.extensions.TLFunctionLibrary`クラス。

各カスタムCTL関数は、次のものから派生/継承する必要があります。

`org.jetel.interpreter.extensions.TLFunctionPrototype`クラス。

これらのクラスには、定義済の標準操作およびカスタム関数を使用するために定義する必要があるいくつかの抽象メソッドがあります。カスタム関数コード内では、既存のコンテキストを使用するかカスタム・コンテキストを定義する必要があります。コンテキストは、繰り返し、つまり複数のレコードに対して関数を実行する場合にオブジェクトを保存するために使用します。

カスタム関数コードとともに、カスタム関数プラグインも定義する必要があります。ライブラリおよびプラグインの両方が**CloverETL**で使用されます。詳細は、次のwikiページを参照してください。
wiki.cloveretl.org/doku.php?id=function_building

第66章 CTL2

この章では、CTL2の構文および使用について説明します。言語リファレンスまたは組込み関数の詳細は、次の項を参照してください。

- 言語リファレンス(p.890)
- 関数リファレンス(p.919)

例66.1. CTL2構文の例(Rollup)

```
//#CTL2

string[] customers;
integer Length;


function void initGroup(VoidMetadata groupAccumulator) {
}

function boolean updateGroup(VoidMetadata groupAccumulator) {
    customers = split($0.customers," - ");
    Length = length(customers);

    return true;
}

function boolean finishGroup(VoidMetadata groupAccumulator) {
    return true;
}

function integer updateTransform(integer counter, VoidMetadata
groupAccumulator) {
    if (counter >= Length) {
        clear(customers);

        return SKIP;
    }

    $0.customers = customers[counter];
    $0.EmployeeID = $0.EmployeeID;

    return ALL;
}

function integer transform(integer counter, VoidMetadata groupAccumulator) {
    return ALL;
}
```

言語リファレンス

Clover Transformation Language (CTL)は、多数のコンポーネントで変換を定義するために使用されます。(すべての**ジョイナ**、**DataGenerator**、**Partition**、**DataIntersection**、**Reformat**、**Denormalizer**、**Normalizer**および**Rollup**が対象)

この項では、次の領域について説明します。

プログラム構造

CTLで作成する各プログラムには、次の部分が含まれている必要があります。

```
ImportStatements
VariableDeclarations
FunctionDeclarations
Statements
Mappings
```

これらはすべて任意の場所に配置できますが、次のようないくつかの基本事項が適用されます。

- インポート文を定義する場合、この文はコードの先頭に配置する必要があります。
- 変数および関数を使用するには、その前にそれらを宣言する必要があります。
- 変数および関数の宣言、文およびマッピングも互いに自由な位置に配置できます。

重要

CTL2では、変数および関数の宣言は変換コードの任意の場所に配置でき、他のコードの後に配置できます。ただし、各変数および各関数を使用するには、常にその前に宣言する必要があります。

これは、**CloverETL** Transformation Languageの2つのバージョンの違いの1つです。

(CTL1では、コードの各部分の順序が固定されており、変更できませんでした。)

コメント

プログラム全体でコメントを使用できます。これらのコメントは処理されず、プログラム内の処理の説明としてのみ使用します。

コメントには2種類あります。1行コメントまたは複数行コメントです。次の2つのオプションを参照してください。

```
// This is an one-line comment.
```

```
/* This is a multiline comment. */
```

インポート

まず、CTLでのプログラムの始めに、CTLの既存プログラムの一部をインポートできます。方法は次のとおりです。

```
import 'fileURL';
```

```
import "fileURL";
```

一重引用符または二重引用符のいずれを使用するかを決定する必要があります。一重引用符では、エスケープ・シーケンスをエスケープしません。詳細は、リテラル(p.895)を参照してください。このfileURLには、既存のソース・コード・ファイルのURLを入力する必要があります。

ただし、これらのファイルは、その他すべての宣言または文(あるいはその両方)より前に、始めにインポートする必要があります。

CTL2のデータ型

メタデータで使用されるデータ型の基本情報は、データ型およびレコード・タイプ(p.111)を参照してください。

どのプログラムでも変数を使用できます。CTLのデータ型は次のとおりです。

boolean

次のように宣言します。`boolean identifier;`

byte

このデータ型は、最大長が`Integer.MAX_VALUE`のバイト配列です。これはlistデータ型と同様に動作します(次を参照してください)。

次のように宣言します。`byte identifier;`

cbyte

このデータ型は、最大長が`Integer.MAX_VALUE`の圧縮バイト配列です。これはlistデータ型と同様に動作します(次を参照してください)。

次のように宣言します。`cbyte identifier;`

date

次のように宣言します。`date identifier;`

decimal

次のように宣言します。`decimal identifier;`

デフォルトでは、小数の有効桁数は最大32桁です。異なる**長さ**または**スケール**にするには、メタデータでこれらの`decimal`フィールドのプロパティを設定する必要があります。

例66.2. CTL2でのdecimalデータ型の使用方法の例

小数変数に100.0 /3を割り当てた場合、その値は`33.333333333333335701809119200333`のようになります。これを小数フィールド(デフォルトの**長さ**および**スケール**はそれぞれ12および2)に割り当てると、`33.33D`に変換されます。

浮動小数点数の末尾に`d`の文字を付加すると、浮動小数点数をdecimalデータ型にキャストできます。

integer

次のように宣言します。`integer identifier;`

整数の末尾に`l`の文字を付加すると、整数をlongデータ型にキャストできます。

long

次のように宣言します。`long identifier;`

整数の末尾に`l`の文字を付加すると、整数をこのデータ型にキャストできます。

number (double)

次のように宣言します。`number identifier;`

string

次のように宣言します。`string identifier;`

list

各listは、boolean、byte、date、decimal、integer、long、number、string、recordのいずれかのデータ型のコンテナです。

listデータ型は、0で始まる整数でインデックスが付けられます。

次のように宣言します。`string[] identifier;`

リストをlistまたはmapのリストとして作成することはできません。

デフォルト・リストは空のリストです。

例:

```
integer[] myIntegerList; myIntegerList[5] = 123;

Customer JohnSmith;

Customer PeterBrown;

Customer[] CompanyCustomers;

CompanyCustomers[0] = JohnSmith;

CompanyCustomers[1] = PeterBrown
```

割当て:

- `myStringList[3] = "abc";`

  これは、指定された文字列が文字列リストの4番目に配置されることを意味します。その他の値は、次のようにnullで埋められます。

  myStringListは[null,null,null,"abc"]

- `myList1 = myList2;`

  これは、両方のリストが同じ要素を参照することを意味します。

- `myList1 = myList1 + myList2;`

  これはmyList2のすべての要素をmyList1の末尾に追加します。

  いずれのリストも同じプリミティブ・データ型に基づいている必要があります。

- `myList1 = myList1 + "abc";`

  これは、"abc"という文字列をmyList1の新しい最後の要素として追加します。

  myList1はstringデータ型に基づく必要があります。

- `myList1 = null;`

  これはmyList1を破棄します。

リスト操作(追加など)を実行する場合は注意してください。警告(p.894)を参照してください。

map

このデータ型はキーと値のペアのコンテナです。

次のように宣言します。`map[<type of key>, <type of value>] identifier;`

KeyおよびValueのいずれも、boolean、byte、date、decimal、integer、long、number、stringのプリミティブ・データ型にすることができます。Valueはrecord型にすることもできます。

マップをlistまたは他のmapのマップとして作成することはできません。

デフォルト・マップは空のマップです。

例:

```
map[string, boolean] map1; map1["abc"]=true;

Customer JohnSmith;

Customer PeterBrown;

map[integer, Customer] CompanyCustomersMap;

CompanyCustomersMap[JohnSmith.ID] = JohnSmith;

CompanyCustomersMap[PeterBrown.ID] = PeterBrown
```

割当ては、リストに有効なものと同様です。

record

このデータ型は、一連のデータ・フィールドです。

レコードの構造はメタデータに基づいています。メタデータ項目はいずれもデータ型を表します。

次のように宣言します。`<metadata name> identifier;`

メタデータ名はグラフ内で一意である必要があります。メタデータの名前はそれぞれ異なる必要があります。

可能な式およびレコードの使用方法の詳細は、データ・レコードおよびフィールドへのアクセス(p.912)を参照してください。

レコードにはデフォルト値はありません。

これには、整数および文字列(フィールド名)両方のインデックスを付けることができます。インデックスが数字である場合、フィールドのインデックスは0から始まります。

警告

レコードがtransform()などの関数内でレコードのリストにプッシュ、追加、挿入(push()、append()、insert())される場合は注意してください。レコードがグローバル変数として宣言されている場合、リスト内の最後の項目は常に同じレコードを参照します。これを回避するには、レコードを(transform()内で)ローカル変数として宣言してください。transform()を呼び出すと、新しい参照が作成され、正しい値がリストに入れられます。

リテラル

リテラルは、任意のデータ型の値を記述するために使用します。

表66.1. リテラル

リテラル	説明	宣言構文	例
整数	整数を表す数字。	[0-9]+	95623
長整数	2^{31}より大きく2^{63}より小さい絶対値を持つ整数を表す数字。	[0-9]+L?	257Lまたは 9562307813123123
16進数整数	16進数形式で整数を表す数字と文字。	0x[0-9A-F]+	0xA7B0
8進数整数	8進数形式で整数を表す数字。	0[0-7]*	0644
数値(double)	倍精度形式の64ビットで表される浮動小数点数。	[0-9]+.[0-9]+	456.123
小数	小数を表す数字。	[0-9]+.[0-9]+D	123.456D
二重引用符付き文字列	二重引用符で囲まれた文字列値/リテラル。エスケープ文字[\n,\r,\t, \\, \", \b]は対応する制御文字に変換される。	"...["]以外の任意の文字列..."	"hello\tworld\n\r"
一重引用符付き文字列	一重引用符で囲まれた文字列値/リテラル。エスケープ文字は[\']のみが対応する文字[']に変換される。	'...[']以外の任意の文字列...'	'hello\tworld\n\r'
リテラルのリスト	リテラルのリスト。各リテラルはその他のリスト、マップまたはレコードにもできる。	[<any literal> (, <any literal>)*]	[10, 'hello', "world", 0x1A, 2008-01-01], [[1 , 2]] , [3 , 4]]
日付	日付値。	予期されるマスク: yyyy-MM-dd	2008-01-01
日時	日時値。	予期されるマスク: yyyy-MM-dd HH:mm:ss	2008-01-01 18:55:00

重要

byteデータ型にはリテラルを使用できません。byte値を書き込むには、byteを返す変換関数を使用して、それを引数値に適用する必要があります。

これらの変換関数の詳細は、変換関数(p.921)を参照してください。

重要

小数値を小数フィールドに割り当てる必要がある場合は、小数リテラルを使用する必要があります。そうしない場合、数値は小数ではなく倍精度浮動小数点数になります。

例:

1. **小数フィールドへの小数値の割当て(正しく正確)**

```
// correct - assign decimal value to decimal field

myRecord.decimalField = 123.56d;
```

2. **小数フィールドへの倍精度浮動小数点値の割当て(不正確な可能性がある)**

```
// possibly inaccurate - assign double value to decimal field
myRecord.decimalField = 123.56;
```

後者は不正確な結果を生成する可能性があります。

変数

変数を定義するには、変数のデータ型、空白、変数の名前およびセミコロンを入力する必要があります。

これらの変数は後から初期化できますが、その宣言自体で初期化することもできます。式の値は変数と同じデータ型である必要があります。

変数の宣言および初期化の両方の例は、次のとおりです。

```
dataType variable;

...

variable = expression;

dataType variable = expression;
```

CTL2のディクショナリ

グラフでディクショナリを使用し、CTL2からエントリにアクセスできるようにするには、第31章「ディクショナリ」(p.228)に示すようにグラフでディクショナリを定義する必要があります。

CTL2からエントリにアクセスするには、次のようなドット構文を使用します。

```
dictionary.<dictionary entry>
```

この式は、次の目的に使用できます。

- エントリの値を定義します。

  ```
  dictionary.customer = "John Smith";
  ```

- エントリの値を取得します。

  ```
  myCustomer = dictionary.customer;
  ```

- エントリの値を出力フィールドにマップします。

  ```
  $0.myCustomerField = dictionary.customer;
  ```

- 関数の引数として使用します。

  ```
  myCustomerID = isInteger(dictionary.customer);
  ```

演算子

演算子は、プログラム内でより複雑な式を作成するためのものです。算術、関係および論理演算子を使用できます。関係および論理演算子は、結果としてブール値を生成する式を作成します。算術演算子は、論理式のみでなく、すべての式で使用できます。

すべての演算子は次の4つのカテゴリに分類できます。

算術演算子

次の演算子は、異なる式の値をまとめるために使用します(ブール値の式は除く)。これらの記号は1つの式で複数回使用できます。そのような場合、カッコを使用して演算子の優先度を表現できます。結果は式の順序によって異なります。

- 加算

+

この演算子は、2つの式の値を加算するために使用します。

ただし、2つのブール値または2つの日付データ型の加算はできません。2つのブール値から新しい値を作成するには、かわりに論理演算子を使用する必要があります。

文字列に任意のデータ型を追加する場合は、2番目のデータ型が自動的に文字列に変換され、最初の(文字列の)被加数と連結されます。ただし、文字列を先頭に置く必要があります。2つの文字列も同じ方法で加算できます。また、`concat()`関数の方が高速であるため、文字列への追加よりも、この関数を使用することをお薦めします。

任意の数値データ型を日付に追加することもできます。結果は、数値の整数部分だけ日数が増加した日付になります。この場合も、日付を先頭に置く必要があります。

2つの数値データ型の合計は、それらのデータ型の順序によって異なります。結果のデータ型は最初の被加数のデータ型と同じになります。2番目の被加数は最初のデータ型に自動的に変換されます。

- 減算およびユニタリのマイナス

−

この演算子は、ある数値データ型を他の数値データ型から減算するために使用します。この場合も、結果のデータ型は被減数のデータ型と同じになります。減数は、被減数のデータ型に自動的に変換されます。

また、日付データ型から数値データ型を減算することもできます。結果は、減数の整数部分だけ日数が減少した日付になります。

- 乗算

*

この演算子は、2つの数値データ型を乗算するためにのみ使用します。

乗算では、最初の被乗数によって演算結果のデータ型が決まります。最初の被乗数が整数で、2番目の被乗数が小数の場合、結果は整数になります。一方、最初の被乗数が小数で、2番目の被乗数が整数の場合、結果は小数データ型になります。つまり、被乗数の順序が重要になります

- 除算

/

この演算子は、2つの数値データ型を除算するためにのみ使用します。ゼロでは除算できません。ゼロで除算すると、TransformLangExecutorRuntimeExceptionがスローされるか、Infinityが返されます(数値データ型の場合)。

除算では、分子によって演算結果のデータ型が決まります。分子が整数で、分母が小数の場合、結果は整数になります。一方、分子が小数で、分母が整数の場合、結果は小数データ型になります。つまり、分子と分母のデータ型が重要になります。

- 法

%

この演算子は、浮動小数点数データ型および整数データ型の両方に使用できます。これは除算の余りを返します。

- 増分

++

この演算子は、数値データ型を1ずつ増分するために使用します。この演算子は、浮動小数点数データ型および整数データ型の両方に使用できます。

これを接頭辞として使用すると、数値は最初に増分されてから式で使用されます。

これを接尾辞として使用すると、数値は最初に式で使用されてから増分されます。

重要

増分演算子は、リテラル、レコード・フィールド、マップおよび整数データ型のリスト値には適用できません。

これは整数変数でのみ使用できます。

- 減分

--

この演算子は、数値データ型を1ずつ減分するために使用します。この演算子は、浮動小数点数データ型および整数データ型の両方に使用できます。

これを接頭辞として使用すると、数値は最初に減分されてから式で使用されます。

これを接尾辞として使用すると、数値は最初に式で使用されてから減分されます。

重要

減算演算子は、リテラル、レコード・フィールド、マップおよび整数データ型のリスト値には適用できません。

これは整数変数でのみ使用できます。

関係演算子

次の演算子は、副次式を比較してブール値の結果を取得する場合に使用します。次に示す各記号を使用できます。これらの記号は1つの式で複数回使用できます。そのような場合、カッコを使用して比較の優先度を表現できます。

重要

`.operator.`構文を選択した場合、演算子を空白で囲む必要があります。eq演算子の構文例は次のとおりです。

コード	機能するかどうか
5 .eq. 3	✔
`5 == 3`	✔
5 eq 3	✖
5.eq(3)	✖

- より大きい

  次の2つの記号は、数値、日付および文字列データ型で構成される式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

`>`

`.gt.`

- 以上

  次の3つの記号は、数値、日付および文字列データ型で構成される式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

`>=`

`=>`

`.ge.`

- より小さい

  次の2つの記号は、数値、日付および文字列データ型で構成される式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

`<`

`.lt.`

- 以下

  次の3つの記号は、数値、日付および文字列データ型で構成される式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

`<=`

`=<`

`.le.`

- 等しい

  次の2つの記号は、任意のデータ型の式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

==

.eq.

- 等しくない

  次の3つの記号は、任意のデータ型の式を比較するために使用できます。式の両方のデータ型が比較可能である必要があります。2つの式のデータ型が異なる場合、結果は式の順序によって異なる場合があります。

!=

<>

.ne.

- 正規表現と一致

  この演算子は、文字列および一部の正規表現(p.963)を比較するために使用します。文字列全体が正規表現と一致する場合はtrueを返し、そうでない場合はfalseを返します。

~=

.regex.

- 正規表現を含む

  この演算子は、文字列および一部の正規表現(p.963)を比較するために使用します。正規表現と一致する部分文字列が文字列に含まれる場合はtrueを返し、そうでない場合はfalseを返します。

?=

論理演算子

値がブール・データ型である必要がある式が複雑な場合、論理接続詞(AND、OR、NOT、EQUAL TO、NOT EQUAL TO)で接続された副次式(前述)で式を構成できます。そのような式で優先度を示すには、カッコを使用します。次に示す接続詞のいずれかの形式を選択できます(たとえば、`&&`または`and`)。

`.operator.`という形式の各記号は、空白で囲む必要があります。

- 論理AND

`&&`

`and`

- 論理OR

`||`

`or`

- 論理NOT

`!`

`not`

- 論理EQUAL TO

`==`

`.eq.`

- 論理NOT EQUAL TO

`!=`

`<>`

`.ne.`

割当て演算子

Clover 3.3から、=演算子は単にオブジェクト参照を渡すだけでなく、値の**ディープ・コピー**も実行するようになりました。このため、パフォーマンスの負荷が高まります。ディープ・コピーは、リスト、マップ、レコードおよび日付などの可変データ型に対してのみ実行されます。CTL2では既存オブジェクトの状態を(そのオブジェクトがJavaで可変であっても)変更できないため、その他の型は不変とみなされます。したがって、値をコピーするのではなく、参照を渡す方が安全です。この前提はカスタムCTL2関数ライブラリには当てはまらない可能性があります。

例66.3. コピーされたリスト、マップおよびレコードの変更

```
    integer[] list1 = [1, 2, 3];
    integer[] list2;
    list2 = list1;

    list1.clear(); //  only list1 is cleared (older implementation: list2 was
cleared, too)

    map[string, integer] map1;
    map1["1"] = 1;
    map1["2"] = 2;
    map[string, integer] map2;
    map2 = map1;

    map1.clear(); //  only map1 is cleared (older implementation: map2 was cleared,
too)

    myMetadata record1;
    record1.field1 = "original value";
    myMetadata record2;
    record2 = record1;

    record1.field1 = "updated value"; // only record1 will be updated (older
implementation: record2 was updated, too)
```

単純文と文のブロック

すべての文は次の2つのグループに分けることができます。

- **単純文**はセミコロンで終了する式です。

  例:

```
integer MyVariable;
```

- **文のブロック**は一連の単純文です(各文はセミコロンで終了しています)。ブロック内の文は、すべてを1行に記述することも、複数行に記述することもできます。これらは中カッコで囲みます。閉じ中カッコの後ろにセミコロンは使用しません。

  例:

```
while (MyInteger<100) {
    Sum = Sum + MyInteger;
    MyInteger++;
}
```

制御文

一部の文はプログラムのプロセスを制御するために使用します。

すべての制御文は次のカテゴリに分類できます。

- 条件文(p.905)
- 繰返し文(p.906)
- ジャンプ文(p.907)

条件文

これらの文はプログラムのプロセスを分岐するために使用します。

if 文

この文は、Conditionの値に基づいてStatementを実行するかどうかを決定します。Conditionがtrueの場合、Statementが実行されます。falseの場合、Statementは無視され、プロセスはif文の後に進みます。Statementは単純文または文のブロックのいずれかです。

```
if (Condition) Statement
```

if文の前のバージョン(StatementはConditionがtrueの場合にのみ実行される)とは異なり、Conditionの値がfalseの場合でも実行する必要がある他のStatementをif文に追加できます。したがって、Conditionがtrueの場合はStatement1が実行され、falseの場合はStatement2が実行されます。次を参照してください。

```
if (Condition) Statement1 else Statement2
```

Statement2は別のif文にすることも、それにelse分岐を含めることもできます。

```
if (Condition1) Statement1
    else if (Condition2) Statement3
        else Statement4
```

switch 文

より分岐の多いif文を作成した場合、文が非常に複雑になることがあります。そのような場合は、switch文を使用する方が適しています。

この場合、2つの値(trueまたはfalse)のみを取るif文のConditionのかわりに、Expressionが評価され、その値がswitch文で指定されるConstantと比較されます。

Expressionの値に等しいConstantのみが、どのStatementが実行されるかを決定します。

たとえば、Expressionの値がConstant1の場合はStatement1が実行されます。

重要

リテラルはswitch文内で一意である必要があります。

```
switch (Expression) {
    case Constant1 : Statement1 StatementA [break;]
    case Constant2 : Statement2 StatementB [break;]
    ...
    case ConstantN : StatementN StatementW [break;]
}
```

オプションのbreak;文によって、定数に対応する文のみが実行されるようになります。そうでない場合、後続の文もすべて実行されます。

次の例では、Expressionの値がConstant1,...,ConstantNの値に等しくない場合でも、デフォルト文(StatementN+1)が実行されます。

```
switch (Expression) {
    case Constant1 : Statement1 StatementA [break;]
    case Constant2 : Statement2 StatementB [break;]
    ...
    case ConstantN : StatementN StatementW [break;]
    default : StatementN+1 StatementZ
}
```

繰返し文

これらの繰返し文は、実行サイクルを制限するConditionがfalseになるか同じデータ型のすべての値について実行されるまで、内側のStatementを循環的に実行することによって、なんらかのプロセスを繰り返します。

For ループ

まず、初期化が設定され、その後Conditionが評価されます。その値がtrueの場合、Statementが実行され、最終的にIterationが行われます。

ループの次のサイクルではConditionが再度評価され、trueの場合はStatementが実行され、Iterationが行われます。このようにして、Conditionがfalseになるまで、プロセスが繰り返されます。その後、ループが終了し、プロセスはプログラムの別の部分に進みます。

最初の時点でConditionがfalseだった場合、プロセスはStatementを省略し、ループを終了します。

```
for (Initialization;Condition;Iteration)
    Statement
```

重要

ForループのInitializationの部分にも、ループで使用される変数の宣言が含まれる場合があります。

Initialization、ConditionおよびIterationはオプションです。

do-while ループ

最初にStatementが実行されます。次のプロセスはConditionの値によって異なります。値がtrueの場合、Statementが再度実行され、続けてConditionが再度評価されます。サブプロセスは続行する(再度trueの場合)か、停止して次またはより上位レベルのサブプロセスにジャンプします(falseの場合)。Conditionはループの最後にあるため、サブプロセスの始めにこれがfalseだった場合でも、Statementが1回以上実行されます。

```
do Statement while (Condition)
```

while ループ

このプロセスはConditionの値によって異なります。値がtrueの場合、Statementが実行され、続けてConditionが再度評価されます。サブプロセスは続行する(再度trueの場合)か、停止して次またはより上位レベルのサブプロセスにジャンプします(falseの場合)。Conditionはループの先頭にあるため、サブプロセスの始めにこれがfalseだった場合、Statementは実行されず、ループが省略されます。

```
while (Condition) Statement
```

for-each ループ

foreach文は、コンテナ内の同じデータ型を持つすべてのフィールドで実行されます。構文は次のとおりです。

```
foreach (<data type> myVariable : iterableVariable) Statement
```

iterableVariableコンテナで、同じデータ型のすべての要素が検索されます(データ型はこの文で宣言されます)。iterableVariableにはリストまたはレコードを指定できます。同じデータ型の各変数について、指定したStatementが実行されます。これには、単純文または文のブロックのいずれかを指定できます。

そのため、たとえば、同じStatementを1つのレコードのすべてのstringフィールドで実行できます。

注意

(<entries>全体ではなく)マップの*value*を繰り返すことができます。ループ変数の型はマップの値の型と同じである必要があります。

```
map[string, integer] myMap;

myMap["first"] = 1;
myMap["second"] = 2;

foreach(integer value: myMap) {
  printErr(value); // prints 1 and 2
}
```

マップのキーをlist[]として取得するには、getKeys()(p.944)関数を使用します。

ジャンプ文

場合によっては、Conditionの値に基づいて決定するのとは別の方法でプロセスを制御する必要があります。このことを行うには、次のオプションがあります。

break 文

サブプロセスを停止するには、プログラムで次の文を使用します。

```
break;
```

サブプロセスが中断し、プロセスはより上位レベルまたは次のStatementにジャンプします。

continue 文

繰返しサブプロセスを停止するには、プログラムで次の文を使用します。

```
continue;
```

サブプロセスが中断し、プロセスは次の繰返しステップにジャンプします。

return 文

関数で、returnという語を単独でまたはexpressionとともに使用できます。(次の2つのオプションを参照してください。)return文は、関数内の任意の場所に置くことができます。複数のreturn文があり、条件などに応じてそのうちの特定の文が実行される場合もあります。

```
return;
```

```
return expression;
```

エラー処理

CTL2には、発生する可能性のあるエラーを検出して処理する単純なメカニズムも備わっています。

ただし、CTL2のエラー処理はCTL1のものとは異なります。CTL2ではtry-catch文は使用されません。

必須の変換関数それぞれに存在するオプションのOnError()関数のみを使用します。

たとえば、必須関数(append()、transform()など)に、次のようなオプションの関数があります。

appendOnError()、transformOnError()など

これらの必須関数にはそれぞれ、元の(必須の)関数の名前にOnError接尾辞が追加された名前を持つ、(オプションの)対応する関数がある場合があります。

さらに、すべての<required ctl template function>OnError()関数は元の必須関数と同じ値を返します。

このようにして、元の必須関数によりスローされる例外によって、対応する<required ctl template function>OnError()関数が呼び出されます(たとえば、transform()が失敗するとtransformOnError()が呼び出されます)。

このtransformOnError()では、正しくないコードを修正する、エラー・メッセージをコンソールに出力するなどの操作を実行できます。

重要

これらのOnError()関数は、元の必須関数が**エラー・コード**(-1未満の値)を返した場合は呼び出されません。

OnError()関数を呼び出す必要がある場合は、raiseError(string arg)関数を使用する必要があります。または(前述したように)、元の必須関数により例外がスローされると、対応するOnError()関数が呼び出されます。

関数

独自の関数を次の方法で定義できます。

```
function returnType functionName (type1 arg1, type2 arg2,..., typeN argN) {
    variableDeclarations
    otherFunctionDeclarations
    Statements
    Mappings
    return [expression];
}
```

return文は最後に置く必要があります。return文の詳細は、return文(p.908)を参照してください。一部の関数では、その内部にMappingsを置くことができます。これらは関数内部の任意の場所に置くことができます。

前述のデータ型に加えて、関数はvoidを返すこともできます。

メッセージ関数

CloverETLバージョン2.8.0から、ユーザー独自のエラー・メッセージ用の関数も定義できるようになりました。

```
function string getMessage() {
        return message;
}
```

このmessage変数は、getMessage()関数がある場所で使用されるように、グローバル文字列変数として宣言し、コード内の任意の場所で定義する必要があります。messageはコンソールに書き出されます。

条件付き失敗式

条件付き失敗式を使用することもできます。

これは次のようなものです。

```
expression1 : expression2 : expression3 : ...: expressionN;
```

この条件付き失敗式は、マッピングおよび変数への割当てに使用でき、関数の引数としても使用できます。

式は、最初の式から順に、左から右に1つずつ評価されます。

1. これらの式のいずれかが正常に変数に割り当てられるか、出力フィールドにマップされるか、関数の引数として使用されるとすぐにその式が使用され、その他の式は評価されません。
2. これらの式のいずれも使用されない(変数に割り当てられず、出力フィールドにマップされず、引数としても使用されない)場合、グラフが失敗します。

重要

CTL2では、この式は複数の方法(変数への割当て、出力フィールドへのマッピング、関数の引数としての使用)で使用できます。

(CTL1では、出力フィールドへのマッピングにのみ使用されていました。)

また、この式はCTL2のインタプリタ・モードでのみ使用できます。

データ・レコードおよびフィールドへのアクセス

この項では、レコード・フィールドの操作方法について説明します。各コンポーネントにはポートがある場合があります。入力ポートおよび出力ポートのいずれも、0から始まる番号が付けられます。

接続されているエッジのメタデータは、その名前で識別される必要があります。メタデータの名前はそれぞれ異なる必要があります。

レコードおよび変数の操作

重要

バージョン3.2より、構文は次のように変更されました。

`$in.portID.fieldID`および`$out.portID.fieldID`

例: `$in.0.* = $out.0.*;`

このようにして、入力および出力メタデータを明確に区別できます。

以前に作成した変換は古い構文と互換性があります。

ここで、`Customers`がメタデータのID、その名前が`customers`、その3番目のフィールド(フィールド2)が`firstname`であると仮定します。

次の式は、指定したメタデータの3番目のフィールド(フィールド2)の値を表しています。

- `$<port number>.<field number>`

  例: `$0.2`

  `$0.*`は、最初のポート(ポート0)のすべてのフィールドを意味します。

- `$<port number>.<field name>`

  例: `$0.firstname`

- `$<metadata name>.<field number>`

  例: `$customers.2`

  `$customers.*`は、最初のポート(ポート0)のすべてのフィールドを意味します。

- `$<metadata name>.<field name>`

  例: `$customers.firstname`

CTLコードでレコードを定義することもできます。この定義は次のようになります。

- `<metadata name> MyCTLRecord;`

  例: `customers myCustomers;`

- この後、次の式を使用できます。

  `<record variable name>.<field name>`

  例: `myCustomers.firstname;`

変数へのレコードのマッピングは次のようになります。

- `myVariable = $<port number>.<field number>;`

  例: `FirstName = $0.2;`

- `myVariable = $<port number>.<field name>;`

  例: `FirstName = $0.firstname;`

- `myVariable = $<metadata name>.<field number>;`

  例: `FirstName = $customers.2;`

- `myVariable = $<metadata name>.<field name>;`

  例: `FirstName = $customers.firstname;`

- `myVariable = <record variable name>.<field name>;`

  例: `FirstName = myCustomers.firstname;`

レコードへの変数のマッピングは次のようになります。

- `$<port number>.<field number> = myVariable;`

  例: `$0.2 = FirstName;`

- `$<port number>.<field name> = myVariable;`

  例: `$0.firstname = FirstName;`

- `$<metadata name>.<field number> = myVariable;`

  例: `$customers.2 = FirstName;`

- `$<metadata name>.<field name> = myVariable;`

  例: `$customers.firstname = FirstName;`

- `<record variable name>.<field name> = myVariable;`

  例: `myCustomers.firstname = FirstName;`

重要

コンポーネントに1つの入力ポートまたは1つの出力ポートがある場合、次のような構文を使用できます。

```
$firstname
```

通常、構文は次のようになります。

```
$<field name>
```

重要

次の構文を使用して、内部CTLレコードに入力を割り当てることができます。

```
MyCTLRecord.* = $0.*;
```

また、次の構文を使用して、出力に内部レコードの値をマップすることもできます。

```
$0.* = MyCTLRecord.*;
```

マッピング

マッピングは、一部の**CloverETL**コンポーネントで定義される各変換の一部です。

計算または生成された値または入力フィールドの値が出力フィールドに割り当てられます(マップされます)。

1. マッピングにより出力フィールドに値が割り当てられます。
2. マッピング演算子は次のとおりです。

   =

3. マッピングは常に関数の内部で定義する必要があります。
4. マッピングは関数内部の任意の場所で定義できます。

重要

CTL2マッピングは変換コードの任意の場所に置くことができ、マッピングの後で任意のコードを使用できます。これは、**CloverETL** Transformation Languageの2つのバージョンの違いの1つです。

(CTL1では、マッピングは関数の最後に置く必要があり、マッピングの後で使用できるのは1つの`return`文のみでした。)

CTL2ではマッピング演算子は単なる等号です。

5. ユーザー定義関数にマッピングをラップし、後で、別の関数内でその関数を使用することもできます。
6. フィールド名またはフィールド位置により、異なる入力メタデータを異なる出力メタデータにマップすることもできます。次の例を参照してください。

異なるメタデータの(名前による)マッピング

次のように入力を出力にマップする場合、

```
$0.* = $0.*;
```

入力メタデータが出力のメタデータとも異なる場合があります。

前述の式では、入力フィールドが入力と同じ名前および型の出力フィールドにマップされます。各メタデータでの格納順序およびいずれのメタデータの全フィールドの数も重要ではありません。

最初の2つのフィールドが`firstname`および`lastname`である入力メタデータがある場合、これらの2つのフィールドはそれぞれ出力の対応するフィールドにマップされます。そのような出力の`firstname`フィールドが5番目で、`lastname`フィールドが3番目である場合でも、これら2つの入力フィールドはこれら2つの出力フィールドにマップされます。

入力メタデータおよび出力メタデータにさらにフィールドがあっても、入力のいずれかと同じ名前の出力フィールドが(対応するメタデータでのフィールドの互いの位置に関係なく)存在しない場合、それらのフィールドは互いにマップされません。

前述の単純なマッピング(`$0.* = $0.*;`)に加えて、次の関数も使用できます。

```
void copyByName(record to, record from);
```

例66.4. 名前によるメタデータのマッピング(copyByName()関数を使用)

```
recordName2 myOutputRecord;
copyByName(myOutputRecord.*,$0.*);
$0.* = myOutputRecord.*;
```

重要

メタデータ・フィールドは、その順序および全フィールドの数とは関係なく、名前およびデータ型によって入力から出力にマップされます。

次の構文も使用できます。`myOutputRecord.copyByName($0.*);`

異なるメタデータの(位置による)マッピング

入力を出力にマップする必要があるが、入力フィールドの名前が出力フィールドの名前と異なる場合があります。そのような場合、位置で入力を出力にマップできます。

このことを行うには、次の関数を使用する必要があります。

```
void copyByPosition(record to, record from);
```

例66.5. 位置によるメタデータのマッピング

```
recordName2 myOutputRecord;
copyByPosition(myOutputRecord,$0.*);
$0.* = myOutputRecord.*;
```

重要

(前述の例に示すように)メタデータ・フィールドは位置により入力から出力にマップできます。

次の構文も使用できます。`myOutputRecord.copyByPosition($0.*);`

ユースケース1: 1つの文字列フィールドを大文字に変換する場合

マッピングがどのように行われるかをより詳しく示すために、マッピングの例をいくつか示します。

Reformatコンポーネントを持つグラフがあります。その入力および出力のメタデータは同一です。最初の2つのフィールド(`field1`および`field2`)は文字列データ型で、3番目のフィールド(`field3`)は整数データ型です。

1. `field1`の値の文字は大文字に変更し、他の2つのフィールドは変更せずに出力に渡します。
2. また、`field3`の値に基づいてレコードを分配します。`field3`の値が5未満のレコードは出力ポート0に送信され、それ以外は出力ポート1に送信されるようにします。

マッピングの例

最初の方法として、CTLテンプレートの`transform()`関数内で、両方のポートおよびすべてのフィールドのマッピングを定義します。

例66.6. 個々のフィールドを使用したマッピングの例

マッピングはすべてのレコードについて実行されます。つまり、レコードが出力ポート1に送信される場合にも、出力ポート0のマッピングも実行され、その逆も同様です。

また、マッピングは個々のフィールドで構成され、レコードに多数のフィールドが含まれる場合は複雑になる可能性があります。次の例では、より適切な方法で解決できる方法を示します。

```
function integer transform() {

    // mapping input port records to output port records
    // each field is mapped separately
    $0.field1 = upperCase($0.field1);
    $0.field2 = $0.field2;
    $0.field3 = $0.field3;
    $1.field1 = upperCase($0.field1);
    $1.field2 = $0.field2;
    $1.field3 = $0.field3;

    // output port number returned
    if ($0.field3 < 5) return 0; else return 1;
}
```

注意

CTL2ではマッピングの*後*でコードを使用できるため、ここではマッピングの後にif文を2つのreturn文とともに使用しています。

CTL2マッピングは変換コードの任意の場所に置くことができ、マッピングの後で任意のコードを使用できます。

2番目の方法でも、CTLテンプレートのtransform()関数内で、両方のポートおよびすべてのフィールドのマッピングを定義します。ただし、今回は、マッピングでワイルドカードが使用されています。レコードは変更せずに出力に渡され、このワイルドカード・マッピングの後で、変更する必要があるフィールドが指定されます。

例66.7. ワイルドカードを使用したマッピングの例

マッピングはすべてのレコードについて実行されます。つまり、レコードが出力ポート1に送信される場合にも、出力ポート0のマッピングも実行され、その逆も同様です。

ただし、今回は、マッピングは最初にワイルドカードを使用して、レコードを変更せずに出力に渡し、ワイルドカードを使用したマッピングの後で最初のフィールドが変更されます。

これは、変更しないフィールドが多数あり、変更するフィールドが少数ある場合に便利です。

```
function integer transform() {

    // mapping input port records to output port records
    // wild cards for mapping unchanged records
    // transformed records mapped aditionally
    $0.* = $0.*;
    $0.field1 = upperCase($0.field1);
    $1.* = $0.*;
    $1.field1 = upperCase($0.field1);

    // return the number of output port
    if ($0.field3 < 5) return 0; else return 1;
}
```

注意

CTL2ではマッピングの*後*でコードを使用できるため、ここではマッピングの後にif文を2つのreturn文とともに使用しています。

CTL2マッピングは変換コードの任意の場所に置くことができ、マッピングの後で任意のコードを使用できます。

3番目の方法として、CTLテンプレートのtransform()関数の外で、両方のポートおよびすべてのフィールドのマッピングを定義します。各出力ポートに独自のマッピングがあります。

ここでも、ワイルドカードが使用されます。

出力ポートごとに個別の関数でマッピングを定義すると、次のようなことが改善されます。

- マッピングは対象の出力ポートについてのみ実行されます。つまり、ポート0に送信されるレコードをポート1にマップする必要がなくなり、その逆も同様です。

例66.8. 別個のユーザー定義関数でのワイルドカードを使用したマッピングの例

さらに、マッピングは最初にワイルドカードを使用して、レコードを変更せずに出力に渡し、ワイルドカードを使用したマッピングの後で最初のフィールドが変更されます。これは、変更しないフィールドが多数あり、変更するフィールドが少数ある場合に便利です。

```
// mapping input port records to output port records
// inside separate functions
// wild cards for mapping unchanged records
// transformed records mapped aditionally
function void mapToPort0 () {
    $0.* = $0.*;
    $0.field1 = upperCase($0.field1);
}

function void mapToPort1 () {
    $1.* = $0.*;
    $1.field1 = upperCase($0.field1);
}

// use mapping functions for all ports in the if statement
function integer transform() {
    if ($0.field3 < 5) {
        mapToPort0();
        return 0;
    }
    else {
        mapToPort1();
        return 1;
    }
}
```

パラメータ

パラメータは、Clover Transformation Languageで`${nameOfTheParameter}`のように使用できます。このパラメータが文字列データ型とみなされるようにするには、`'${nameOfTheParameter}'`または`"${nameOfTheParameter}"`のように、パラメータを一重引用符または二重引用符で囲む必要があります。

重要

1. エスケープ・シーケンスは常に、パラメータに割り当てられるとすぐに解決されます。このため、これらが解決されないようにする場合は、これらの文字列で、単一ではなく二重のバックスラッシュを入力してください。
2. また、パラメータを使用して、環境変数の値を取得することもできます。方法は、環境変数 (p.223)を参照してください。

関数リファレンス

Clover Transformation Languageには、ユーザーが使用できる一連の関数があります。ここではこれらについて説明します。

すべての関数は次のカテゴリに分類できます。

- 変換関数(p.921)
- コンテナ関数(p.943)
- レコード関数(動的フィールド・アクセス)(p.947)
- 日付関数(p.928)
- 算術関数(p.930)
- 文字列関数(p.934)
- その他の関数(p.951)
- 参照表関数(p.955)
- シーケンス関数(p.958)
- カスタムCTL関数(p.959)

重要

CTL2では、**CloverETL**組込み関数およびユーザー独自の関数を、次に示すいずれかの方法で使用できます。

組込み関数

- `substring(upperCase(getAplhanumericChars($0.field1))1,3)`
- `$0.field1.getAlphanumericChars().upperCase().substring(1,3)`

これらの2つの式は同等です。関数本体の前に最初の引数がある2つ目のオプションは、**オブジェクト表記**と呼ばれることがあります。必ず$port.field.function()構文を使用してください。したがって、`arg.substring(1,3)`は`substring(arg,1,3)`と等しくなります。

また、次のように任意のデータ型の引数を持つユーザー独自の関数を宣言することもできます。

```
function integer myFunction(integer arg1, string arg2, boolean arg3) {
<function body>
}
```

ユーザー定義関数

- `myFunction($0.integerField,$0.stringField,$0.booleanField)`
- `$0.integerField.myFunction($0.stringField,$0.booleanField)`

警告

オブジェクト表記法(<first argument>.function(<other arguments>)は、**その他**の関数では使用できません。その他の関数(p.951)を参照してください。

重要

stringデータ・フィールドのメタデータで「**Null value**」プロパティを空ではない文字列に設定した場合、stringデータ・フィールドを引数として受け入れ、nullで適用された場合にNPEをスローする関数(たとえば、length()は、そのような文字列に適用された場合にNPEをスローします。

たとえば、field1で「**Null value**」プロパティを"<null>"に設定した場合、length($0.field1)は、レコードのfield1の値が"<null>"である場合は失敗し、フィールドが空の場合は0になります。

詳細は、Null value(p.164)を参照してください。

変換関数

値のデータ型を変換する必要がある場合があります。

データ型を変換する関数では、日付や数値の書式パターンを定義する必要がある場合があります。また、ロケールもそれらの書式設定に影響する場合があります。

- 日付の書式設定または解析(あるいはその両方)の詳細は、日付と時刻の書式(p.113)を参照してください。
- 数値データ型の書式設定または解析(あるいはその両方)の詳細は、数値の書式(p.120)を参照してください。
- ロケールの詳細は、ロケール(p.126)を参照してください。

注意

数値および日付の書式は、その他の**ロケール**が明示的に指定されていないかぎり、システム値の**ロケール**または`defaultProperties`ファイルで指定された**ロケール**を使用して表示されます。

`defaultProperties`で**ロケール**を変更する方法の詳細は、デフォルトのCloverETL設定の変更(p.88)を参照してください。

これらの関数のリストを次に示します。

- `byte` **`base64byte`**`(string` *`arg`*`);`

  `base64byte(string)`関数は、`base64`表現の文字列引数を1つ取得して、それをバイト配列に変換します。対応する関数は`byte2base64(byte)`関数です。

- `string` **`bits2str`**`(byte` *`arg`*`);`

  `bits2str(byte)`関数はバイト配列を取得し、それを0または1の2文字で構成される文字列に変換します。各バイトは8文字(0または1)で表現されます。各バイトでは、最もビット数が小さいものがこれらの8文字の最初に置かれます。対応する関数は`str2bits(string)`関数です。

- `integer` **`bool2num`**`(boolean` *`arg`*`);`

  `bool2num(boolean)`関数は、ブール引数を1つ取得し、それを整数の`1` (引数がtrueの場合)または整数の`0` (引数がfalseの場合)に変換します。対応する関数は`num2bool(<numeric type>)`関数です。

- `string` **`byte2base64`**`(byte` *`arg`*`);`

  `byte2base64(byte)`関数はバイト配列を取得し、それを`base64`表現の文字列に変換します。対応する関数は`base64byte(string)`関数です。

- `string` **`byte2hex`**`(byte` *`arg`*`);`

  `byte2hex(byte)`関数はバイト配列を取得し、それを`hexadecimal`表現の文字列に変換します。対応する関数は`hex2byte(string)`関数です。

- `string` **`byte2str`**`(byte` *`payload`*`, string` *`charset`*`);`

  指定されたキャラクタ・セットを使用して`string`に変換されたバイトを返します。

- `long` **`date2long`**`(date` *`arg`*`);`

  `date2long(date)`関数は、日付引数を1つ取得し、それをlong型に変換します。その値は、`January 1, 1970, 00:00:00 GMT`から引数で指定された日付までに経過したミリ秒数に等しくなります。対応する関数は`long2date(long)`関数です。

- integer **date2num**(date *arg*, unit *timeunit*);

  date2num(date, unit)関数は2つの引数を受け入れます。1番目は日付で、もう1つは任意の時間単位です。単位は、year、month、week、day、hour、minute、second、millisecのいずれかです。単位は定数として指定する必要があります。エッジを介して取得することも、変数として設定することもできません。関数は、これら2つの引数を取得し、システム・ロケールを使用してそれらを整数に変換します。日付に時間単位が含まれている場合、これは整数値として返されます。含まれていない場合、関数は0を返します。CTL1の場合とは異なり、月には1から始まる番号が付けられます。したがって、date2num(2008-06-12, month)は6を返します。また、date2num(2008-06-12, hour)は0を返します。

- integer **date2num**(date *arg*, unit *timeunit*, string *locale*);

  date2num(date, unit, string)関数は、3つの引数を受け入れます。1つ目は日付、2つ目は任意の時間単位、3つ目はロケールです。単位は、year、month、week、day、hour、minute、second、millisecのいずれかです。単位は定数として指定する必要があります。エッジを介して取得することも、変数として設定することもできません。関数は、これら2つの引数を取得し、指定されたロケールを使用してそれらを整数に変換します。日付に時間単位が含まれている場合、これは整数値として返されます。含まれていない場合、関数は0を返します。CTL1の場合とは異なり、月には1から始まる番号が付けられます。したがって、date2num(2008-06-12, month)は6を返します。また、date2num(2008-06-12, hour)は0を返します。

- string **date2str**(date *arg*, string *pattern*);

  date2str(date, string)関数は、日付と文字列の2つの引数を受け入れます。関数はこれらを取得し、2番目の引数として指定されているpatternに従って日付を変換します。したがって、date2str(2008-06-12, "dd.MM.yyyy")は文字列"12.6.2008"を返します。対応する関数はstr2date(string, string)関数です。

- string **date2str**(date *arg*, string *pattern*, string *locale*);

  dateフィールド型を、pattern (日付と時刻の書式を示す)およびlocale (どの日付形式記号を使用するかを定義する)に従ってstringデータ型の日付に変換します。したがって、date2str(2009/01/04,"yyyy-MMM-d","fr.CA")は2009-janv.-4を返します。ロケール設定の詳細は、ロケール(p.126)を参照してください。

- number **decimal2double**(decimal *arg*);

  decimal2double(decimal)関数は小数データ型の引数を1つ取得し、それを倍精度浮動小数点値に変換します。

  変換は縮小的です。また、decimal値をdoubleに変換できない場合(doubleデータ型の範囲にすべてのdecimal値が含まれていないため)、関数は例外をスローします。したがって、decimal2double(92378352147483647.23)は9.2378352147483648E16を返します。

  一方、doubleはすべてdecimalに変換できます。小数の**長さ**および**スケール**の両方がそれに合せて調整される場合があります。

- integer **decimal2integer**(decimal *arg*);

  decimal2integer(decimal)関数は小数データ型の引数を1つ取得し、それを整数に変換します。

  変換は縮小的です。また、decimal値をintegerに変換できない場合(integerデータ型の範囲にdecimal値の範囲がすべて含まれていないため)、関数は例外をスローします。したがって、decimal2integer(352147483647.23)は例外をスローしますが、decimal2integer(25.95)は25を返します。

  一方、integerはすべて精度を損なわずにdecimalに変換できます。小数の**長さ**がそれに合せて調整される場合があります。

- long **decimal2long**(decimal *arg*);

  decimal2long(decimal)関数は小数データ型の引数を1つ取得し、それをlong値に変換します。

  変換は縮小的です。また、decimal値をlongに変換できない場合(longデータ型の範囲にすべてのdecimal値が含まれていないため)、関数は例外をスローします。したがって、decimal2long(9759223372036854775807.25)は例外をスローしますが、decimal2long(72036854775807.79)は72036854775807を返します。

  一方、longはすべて精度を損なわずにdecimalに変換できます。小数の**長さ**がそれに合せて調整される場合があります。

- integer **double2integer**(number *arg*);

  double2integer(number)関数は倍精度浮動小数点データ型の引数を1つ取得し、それを整数に変換します。

  変換は縮小的です。また、double値をintegerに変換できない場合(doubleデータ型の範囲にすべてのinteger値が含まれていないため)、関数は例外をスローします。したがって、double2integer(352147483647.1)は例外をスローしますが、double2integer(25.757197)は25を返します。

  一方、integerはすべて精度を損なわずにdoubleに変換できます。

- long **double2long**(number *arg*);

  double2long(number)関数は倍精度浮動小数点データ型の引数を1つ取得し、それをlongに変換します。

  変換は縮小的です。また、double値をlongに変換できない場合(doubleデータ型の範囲にすべてのlong値が含まれていないため)、関数は例外をスローします。したがって、double2long(1.3759739E23)は例外をスローしますが、double2long(25.8579)は25を返します。

  一方、longは常にdoubleに変換できます。ただし、精度が失われる可能性があることを考慮する必要があります。

- byte **hex2byte**(string *arg*);

  hex2byte(string)関数は、hexadecimal表現の文字列引数を1つ取得して、それをバイト配列に変換します。対応する関数はbyte2hex(byte)関数です。

- string **json2xml**(string *arg*);

  json2xml(string)関数は、JSON形式の文字列引数を1つ取得し、それをXML形式の文字列に変換します。対応する関数はxml2json(string)関数です。

- date **long2date**(long *arg*);

  long2date(long)関数は、long引数を1つ取得して、それを日付に変換します。これはミリ秒数の引数をJanuary 1, 1970, 00:00:00 GMTに追加し、結果を日付として返します。対応する関数はdate2long(date)関数です。

- integer **long2integer**(long *arg*);

  long2integer(decimal)関数はlongデータ型の引数を1つ取得し、それを整数値に変換します。変換は、情報を失わずに実行可能な場合にのみ成功し、そうでない場合は関数から例外がスローされます。したがって、long2integer(352147483647)は例外をスローしますが、long2integer(25)は25を返します。

  一方、integer値はすべて精度を損なわずにlong数値に変換できます。

- byte **long2packDecimal**(long *arg*);

  long2packDecimal(long)関数はlongデータ型の引数を1つ取得し、その値のパック10進数表現を返します。これはpackDecimal2long(byte)関数の対応関数です。

- byte **md5**(byte *arg*);

  md5(byte)関数はバイト配列で構成される引数を1つ受け入れます。これはこの引数を取得して、MD5ハッシュ値を計算します。

- byte **md5**(string *arg*);

  この関数はstringデータ型の引数を1つ受け入れます。これはこの引数を取得して、MD5ハッシュ値を計算します。

- boolean **num2bool**(<numeric type> *arg*);

  num2bool(<numeric type>)関数は、任意の数値データ型(integer、long、numberまたはdecimal)の引数を1つ取得し、それが0の場合はブールのfalseを返し、その他の値の場合はtrueを返します。

- string **num2str**(<numeric type> *arg*);

  <numeric type> arg;

  num2str(<numeric type>)関数は任意の数値データ型(integer、long、numberまたはdecimal)の引数を1つ取得し、それを10進表現の文字列に変換します。ロケールはシステム値です。したがって、num2str(20.52)は"20.52"を返します。

- string **num2str**(<numeric type> *arg*, integer *radix*);

  num2str(<numeric type>,integer)関数は2つの引数を受け入れます。1つ目は3つの数値データ型(integer、long、number)のいずれかで、2つ目は整数です。これは、これらの2つの引数を取得して、最初の引数をradixベースの数値システムの文字列表現に変換します。したがって、num2str(31, 16)は"1F"を返します。ロケールはシステム値です。

  integerおよびlongの両方のデータ型について、任意の整数を基数として使用できます。double(number)の場合は、基数として使用できるのは10または16のみです。

- string **num2str**(<numeric type> *arg*, string *format*);

  num2str(<numeric type>, string)関数は2つの引数を受け入れます。1つ目は任意の数値データ型(integer、long、numberまたはdecimal)で、2つ目は文字列です。これら2つの引数を取得し、2番目の引数として指定された書式を使用して、最初の引数を10進表現の文字列に変換します。ロケールはシステム値を取ります。

- string **num2str**(<numeric type> *arg*, string *format*, string *locale*);

  num2str(<numeric type>, string, string)関数は3つの引数を受け入れます。1つ目は任意の数値データ型(integer、long、numberまたはdecimal)で、その他2つは文字列です。これらの引数を取得し、2番目の引数として指定された書式および3番目の引数として指定されたロケールを使用して、最初の引数を文字列表現に変換します。

- long **packDecimal2long**(byte *arg*);

  packDecimal2long(byte)関数は、long数値のパック10進表現を意味するバイト配列の引数を1つ取得します。これは値をlongデータ型として返します。これはlong2packDecimal(long)関数の対応関数です。

- byte **sha**(byte *arg*);

  sha(byte)関数はバイト配列で構成される引数を1つ受け入れます。これはこの引数を取得して、SHA-1ハッシュ値を計算します。

- byte **sha**(string *arg*);

  sha(string)関数は、文字列データ型の1つの引数を受け入れます。これはこの引数を取得して、SHA-1ハッシュ値を計算します。

- byte **sha256**(byte *arg*);

  sha256(byte)関数はバイト配列で構成される引数を1つ受け入れます。これはこの引数を取得して、SHA-256ハッシュ値を計算します。

- byte **sha256**(string *arg*);

  sha256(string)関数は、文字列データ型の引数を1つ受け入れます。これはこの引数を取得して、SHA-256ハッシュ値を計算します。

- byte **str2bits**(string *arg*);

  str2bits(string)関数は、文字列引数を1つ取得して、それをバイト配列に変換します。対応する関数はbits2str(byte)関数です。文字列は、次の文字で構成されます。各文字は1またはその他の任意の文字です。文字列で、文字1はそれぞれ1ビットに変換され、その他のすべての文字(0のみでなく、a、z、/などもは0ビットに変換されます。文字列の文字数が8の整数倍でない場合、文字列が右から0の文字で埋められます。その後、文字列は、その文字数が8の整数倍であるかのようにバイトの配列に変換されます。

  最初の文字が最小のビットを表します。

- boolean **str2bool**(string *arg*);

  str2bool(string)関数は、文字列引数を1つ取得し、それを対応するブール値に変換します。文字列は、"TRUE"、"true"、"T"、"t"、"YES"、"yes"、"Y"、"y"、"1"、"FALSE"、"false"、"F"、"f"、"NO"、"no"、"N"、"n"、"0"のいずれかです。文字列はブール値のtrueまたはブール値のfalseに変換されます。

- string **str2byte**(string *payload*, string *charset*);

  指定されたキャラクタ・セット・エンコーダを使用して入力バイトから変換された文字列を返します。

- date **str2date**(string *arg*, string *pattern*);

  str2date(string, string)関数は、2つの文字列引数を受け入れます。これらを取得し、最初の文字列を、2番目の引数として指定されているpatternに従って日付に変換します。patternは、最初の引数の構造と対応している必要があります。したがって、str2date("12.6.2008", "dd.MM.yyyy")は2008-06-12という日付を返します。

- date **str2date**(string *arg*, string *pattern*, string *locale*);

  str2date(string, string, string)関数は、3つの文字列引数と1つのブールを受け入れます。これらの引数を取得し、2番目および3番目の引数でそれぞれ指定されているpatternおよびlocaleに従って、最初の文字列を日付に変換します。patternは、最初の引数の構造と対応している必要があります。したがって、str2date("12.6.2008", "dd.MM.yyyy",cs.CZ)は2008-06-12という日付を返します。3番目の引数は、日付のロケールを定義します。

- decimal **str2decimal**(string *arg*);

  str2decimal(string)関数は、文字列引数を1つ取得し、それを対応する小数値に変換します。

- decimal **str2decimal**(string *arg*, string *format*);

  str2decimal(string, string)関数は最初の文字列引数を取得し、それを2番目の引数として指定されている書式に従って対応する小数値に変換します。ロケールはシステム値を取ります。

- decimal **str2decimal**(string *arg*, string *format*, string *locale*);

  str2decimal(string, string, string)関数は、最初の文字列引数を取得し、2番目の引数として指定されている書式および3番目の引数として指定されているロケールに従って、それを対応する小数値に変換します。

- number **str2double**(string *arg*);

  str2double(string)関数は、文字列引数を1つ取得し、それを対応する倍精度浮動小数点値に変換します。

- number **str2double**(string *arg*, string *format*);

  str2double(string, string)関数は最初の文字列引数を取得し、2番目の引数として指定されている書式に従って、それを対応する倍精度浮動小数点値に変換します。ロケールはシステム値を取ります。

- number **str2double**(string *arg*, string *format*, string *locale*);

  str2decimal(string, string, string)関数は、最初の文字列引数を取得し、2番目の引数として指定されている書式および3番目の引数として指定されているロケールに従って、それを対応する倍精度浮動小数点値に変換します。

- integer **str2integer**(string *arg*);

  str2integer(string)関数は、文字列引数を1つ取得し、それを対応する整数値に変換します。

- integer **str2integer**(string *arg*, integer *radix*);

  str2integer(string, integer)関数は、文字列と整数の2つの引数を受け入れます。これは最初の引数をradixベースの数値システム表現で示されているかのように取得し、対応する整数の10進数値を返します。

- integer **str2integer**(string *arg*, string *format*);

  str2integer(string, string)関数は、2番目の引数として指定されている書式およびシステム・ロケールに対応する整数の10進文字列表現として最初の文字列引数を取得し、それを対応する整数値に変換します。

- integer **str2integer**(string *arg*, string *format*, string *locale*);

  str2integer(string, string, string)関数は、2番目の引数として指定されている書式および3番目の引数として指定されているロケールに対応する整数の10進文字列表現として最初の文字列引数を取得し、それを対応する整数値に変換します。

- long **str2long**(string *arg*, integer *radix*);

  str2long(string, integer)関数は、文字列と整数の2つの引数を受け入れます。これは最初の引数をradixベースの数値システム表現で示されているかのように取得し、対応するlong型の10進値を返します。

- long **str2long**(string *arg*, string *format*);

  str2long(string, string)関数は、2番目の引数として指定されている書式およびシステム・ロケールに対応するlong数の10進文字列表現として最初の文字列引数を取得し、それを対応するlong値に変換します。

- long **str2long**(string *arg*, string *format*, string *locale*);

  str2long(string, string, string)関数は、2番目の引数として指定されている書式および3番目の引数として指定されているロケールに対応するlong数の10進文字列表現として最初の文字列引数を取得し、それを対応するlong値に変換します。

- `string` **`toString`**`(<numeric|list|map type>` *`arg`*`);`

  `toString(<numeric|list|map type>)`関数は1つの引数を取得し、それを文字列表現に変換します。これは、任意の数値データ型、任意のデータ型のリストおよび任意のデータ型のマップを受け入れます。

- `string` **`xml2json`**`(string` *`arg`*`);`

  `xml2josn(string)`関数は、XML形式の文字列引数を1つ取得し、それをJSON形式の文字列に変換します。対応する関数は`json2xml(string)`関数です。

日付関数

日付を使用する場合は、日付を処理する関数を使用できます。

これらの関数では、日付または数値の書式パターンを定義する必要がある場合があります。また、ロケールもそれらの書式設定に影響する場合があります。

- 日付の書式設定または解析(あるいはその両方)の詳細は、日付と時刻の書式(p.113)を参照してください。
- ロケールの詳細は、ロケール(p.126)を参照してください。

注意

数値および日付の書式は、その他の**ロケール**が明示的に指定されていないかぎり、システム値の**ロケール**または`defaultProperties`ファイルで指定された**ロケール**を使用して表示されます。

`defaultProperties`で**ロケール**を変更する方法の詳細は、デフォルトのCloverETL設定の変更(p.88)を参照してください。

これらの関数のリストを次に示します。

- `date` **`dateAdd`**`(date` *`arg`*`, long` *`amount`*`, unit` *`timeunit`*`);`

  `dateAdd(date, long, unit)`関数は3つの引数を受け入れます。1つ目は日付、2つ目はlongデータ型、最後は任意の時間単位です。単位は、`year`、`month`、`week`、`day`、`hour`、`minute`、`second`、`millisec`のいずれかです。単位は定数として指定する必要があります。エッジを介して取得することも、変数として設定することもできません。関数は最初の引数を取得して、時間単位の数量をその引数に追加し、結果を日付として返します。`amount`および時間の`unit`は、それぞれ2番目および3番目の引数として指定します。

- `long` **`dateDiff`**`(date` *`later`*`, date` *`earlier`*`, unit` *`timeunit`*`);`

  `dateDiff(date, date, unit)`関数は3つの引数を受け入れます。2つは日付で、もう1つは時間単位です。これらの引数を取得して、最初の引数から2番目の引数を減算します。単位は、`year`、`month`、`week`、`day`、`hour`、`minute`、`second`、`millisec`のいずれかです。単位は定数として指定する必要があります。エッジを介して取得することも、変数として設定することもできません。この関数は、結果の時間の差異を3番目の引数として指定される時間単位で返します。このため、2つの日付の差異は定義された時間単位で表されます。結果は整数として表されます。したがって、`dateDiff(2008-06-18, 2001-02-03, year)`は7を返します。一方、`dateDiff(2001-02-03, 2008-06-18, year)`は-7を返します。

- `date` **`extractDate`**`(date` *`arg`*`);`

  `extractDate(date)`関数は1つの日付引数を取得し、年、月および日の値を含む情報のみを返します。関数の引数が戻り値によって変更されることはありません。

- `date` **`extractTime`**`(date` *`arg`*`);`

  `extractTime(date)`関数は1つの日付引数を取得し、時間、分、秒、ミリ秒を含む情報のみを返します。関数の引数が戻り値によって変更されることはありません。

- `date` **`randomDate`**`(date` *`startDate`*`, date` *`endDate`*`);`

  `randomDate(date, date)`関数は2つの日付引数を受け入れ、`startDate`と`endDate`の間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値は`startDate`または`endDate`である場合もあります。ただし、`startDate`より前の日付および`endDate`より後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、`endDate`が返されるようにするには、その翌日を`endDate`として入力してください。`locale`にはシステム値が使用されます。デフォルトの`format`は`defaultProperties`ファイルで指定されます。

- `date` **`randomDate`**`(long` *`startDate`*`, long` *`endDate`*`);`

  `randomDate(long, long)`関数は、それぞれが日付を表すlongデータ型の引数を2つ受け入れ、`startDate`と`endDate`の間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値は`startDate`または`endDate`である場合もあります。ただし、`startDate`より前の日付および`endDate`より後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、`endDate`が返されるようにするには、その翌日を`endDate`として入力してください。`locale`にはシステム値が使用されます。デフォルトの`format`は`defaultProperties`ファイルで指定されます。

- `date` **`randomDate`**`(string` *`startDate`*`, string` *`endDate`*`, string` *`format`*`);`

  `randomDate(string, string, string)`関数は3つの文字列引数を受け入れます。最初の2つは日付を表し、3つ目は書式を表します。この関数は、3番目の引数で指定されている`format`に従って、`startDate`と`endDate`の間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値は`startDate`または`endDate`である場合もあります。ただし、`startDate`より前の日付および`endDate`より後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、`endDate`が返されるようにするには、その翌日を`endDate`として入力してください。`locale`にはシステム値が使用されます。

- `date` **`randomDate`**`(string` *`startDate`*`, string` *`endDate`*`, string` *`format`*`, string` *`locale`*`);`

  `randomDate(string, string, string, string)`関数は4つの文字列引数を受け入れます。最初と2番目の引数は日付を表します。3番目は書式で、4番目はロケールです。この関数は`startDate`と`endDate`の間のランダムな日付を返します。これらの結果の日付は、異なるレコードおよび異なるフィールドについてランダムに生成されます。レコードとフィールドの両方で異なる場合もあります。戻り値は、それぞれ3番目および4番目の引数で指定される`format`および`locale`に対応する`startDate`または`endDate`になる場合もあります。ただし、`startDate`より前の日付および`endDate`より後の日付にはできません。日付は指定された日の0時間0分0秒0ミリ秒を表すため、`endDate`が返されるようにするには、その翌日を`endDate`として入力してください。

- `date` **`today`**`();`

  `today()`関数は引数を受け入れず、現在の日付と時刻を返します。

- `date` **`zeroDate`**`();`

  `zeroDate()`関数は引数を受け入れず、1.1.1970を返します。

注意

次の2つの関数は**非推奨**です。これらの戻り値は同時に引数を変更します。

- `date` **`trunc`**`(date` *`arg`*`);`

  `trunc(date)`関数は1つの日付引数を取得し、年、月および日は同じで、時間、分、秒、ミリ秒は0の値に設定された日付を返します。

- `date` **`truncDate`**`(date` *`arg`*`);`

  `truncDate(date)`関数は1つの日付引数を取得し、時間、分、秒、ミリ秒は同じで、年、月および日は0の値に設定された日付を返します。0日付は0001-01-01です。

算術関数

算術関数も使用できます。

- <numeric type> **abs**(<numeric type> *arg*);

  abs(<numeric type>)関数は任意の数値データ型(integer、long、numberまたはdecimal)の引数を1つ取得し、その絶対値を同じデータ型で返します。

- integer **bitAnd**(integer *arg1*, integer *arg2*);

  bitAnd(integer, integer)関数は整数データ型の引数を2つ受け入れます。これらを取得し、ビット単位andに対応する数値を返します。(たとえば、bitAnd(11,7)は3を返します。)10進数の11はビット単位の1011と表すことができ、10進数の7はビット単位の111と表すことができるため、結果は10進数の3に対応する11です。

- long **bitAnd**(long *arg1*, long *arg2*);

  bitAnd(long, long)関数はlongデータ型の引数を2つ受け入れます。これらを取得し、ビット単位andに対応する数値を返します。(たとえば、bitAnd(11,7)は3を返します。)10進数の11はビット単位の1011と表すことができ、10進数の7はビット単位の111と表すことができるため、結果は10進数の3に対応する11です。

- boolean **bitIsSet**(integer *arg*, integer *Index*);

  bitIsSet(integer, integer)関数は整数データ型の引数を2つ受け入れます。これを取得し、Indexの位置にある最初の引数のビット値を判別して、ビットが1の場合はtrueを、ビットが0の場合はfalseを返します。(たとえば、bitIsSet(11,3)はtrueを返します。)10進数の11はビット単位の1011と表すことができ、インデックスが3であるビット(右から4番目)は1であるため、結果はtrueです。また、bitIsSet(11,2)はfalseを返します。

- boolean **bitIsSet**(long *arg*, integer *Index*);

  bitIsSet(long,integer)関数はlongデータ型の1つの引数と1つの整数を受け入れます。これらの引数を取得し、Indexの位置にある最初の引数のビット値を判別して、ビットが1の場合はtrueを、ビットが0の場合はfalseを返します。(たとえば、bitIsSet(11,3)はtrueを返します。)10進数の11はビット単位の1011と表すことができ、インデックスが3であるビット(右から4番目)は1であるため、結果はtrueです。また、bitIsSet(11,2)はfalseを返します。

- integer **bitLShift**(integer *arg*, integer *Shift*);

  bitLShift(integer, integer)関数は整数データ型の引数を2つ受け入れます。これらを取得し、元の数値にビットをいくつか追加したものに対応する数値を返します(左側にShiftの数のビットが追加され、0に設定されます。)(たとえば、bitLShift(11,2)は44を返します。)10進数の11はビット単位の1011と表すことができるため、右側に2ビット(10)が追加されると、結果は10進数の44に対応する101100になります。

- long **bitLShift**(long *arg*, long *Shift*);

  bitLShift(long,long)関数はlongデータ型の引数を2つ受け入れます。これらを取得し、元の数値にビットをいくつか追加したものに対応する数値を返します(左側にShiftの数のビットが追加され、0に設定されます。)(たとえば、bitLShift(11,2)は44を返します。)10進数の11はビット単位の1011と表すことができるため、右側に2ビット(10)が追加されると、結果は10進数の44に対応する101100になります。

- integer **bitNegate**(integer *arg*);

  bitNegate(integer)関数は、整数データ型の引数を1つ受け入れます。これは、ビット単位の反転数に対応する数値を返します。(たとえば、bitNegate(11)は-12を返します。)この関数は引数に含まれるすべてのビットを反転させます。

- long **bitNegate**(long *arg*);

  bitNegate(long)関数は、longデータ型の引数を1つ受け入れます。これは、ビット単位の反転数に対応する数値を返します。(たとえば、bitNegate(11)は-12を返します。)この関数は引数に含まれるすべてのビットを反転させます。

- integer **bitOr**(integer *arg1*, integer *arg2*);

  bitOr(integer, integer)関数は整数データ型の引数を2つ受け入れます。これらを取得し、ビット単位orに対応する数値を返します。(たとえば、bitOr(11,7)は15を返します。)10進数の11はビット単位の1011と表すことができ、10進数の7は111と表すことができるため、結果は10進数の15に対応する1111です。

- long **bitOr**(long *arg1*, long *arg2*);

  bitOr(long, long)関数はlongデータ型の引数を2つ受け入れます。これらを取得し、ビット単位orに対応する数値を返します。(たとえば、bitOr(11,7)は15を返します。)10進数の11はビット単位の1011と表すことができ、10進数の7は111と表すことができるため、結果は10進数の15に対応する1111です。

- integer **bitRShift**(integer *arg*, integer *Shift*);

  bitRShift(integer, integer)関数は整数データ型の引数を2つ受け入れます。これらを取得し、元の数値からビット数をいくつか削除したものに対応する数値を返します(右側のShiftビットが削除されます。)(たとえば、bitRShift(11,2)は2を返します。)10進数の11はビット単位の1011と表すことができるため、右側の2ビットが削除されると、結果は10進数の2に対応する10になります。

- long **bitRShift**(long *arg*, long *Shift*);

  bitRShift(long, long)関数はlongデータ型の引数を2つ受け入れます。これらを取得し、元の数値からビット数をいくつか削除したものに対応する数値を返します(右側のShiftビットが削除されます。)(たとえば、bitRShift(11,2)は2を返します。)10進数の11はビット単位の1011と表すことができるため、右側の2ビットが削除されると、結果は10進数の2に対応する10になります。

- integer **bitSet**(integer *arg1*, integer *Index*, boolean *SetBitTo1*);

  bitSet(integer, integer, boolean)関数は3つの引数を受け入れます。最初の2つは整数データ型で、3つ目はブールです。これらを取得し、2番目の引数として指定されるIndexにある最初の引数のビットの値を1または0に設定し(それぞれ3番目の引数がtrueまたはfalseの場合)、結果を整数で返します。(たとえば、bitSet(11,3,false)は3を返します。)10進数の11はビット単位の1011と表すことができ、インデックスが3であるビット(右から4番目)が0に設定されるため、結果は10進数の3に対応する11になります。また、bitSet(11,2,true)は10進数の15に対応する1111を返します。

- long **bitSet**(long *arg1*, integer *Index*, boolean *SetBitTo1*);

  bitSet(long, integer, boolean)関数は3つの引数を受け入れます。1つ目はlong、2つ目は整数、3つ目はブールです。これらを取得し、2番目の引数として指定されるIndexにある最初の引数のビットの値を1または0に設定し(それぞれ3番目の引数がtrueまたはfalseの場合)、結果を整数で返します。(たとえば、bitSet(11,3,false)は3を返します。)10進数の11はビット単位の1011と表すことができ、インデックスが3であるビット(右から4番目)が0に設定されるため、結果は10進数の3に対応する11になります。また、bitSet(11,2,true)は10進数の15に対応する1111を返します。

- integer **bitXor**(integer *arg*, integer *arg*);

  bitXor(integer, integer)関数は整数データ型の引数を2つ受け入れます。これらを取得し、ビット単位exclusive orに対応する数値を返します。(たとえば、bitXor(11,7)は12を返します。)10進数の11はビット単位の1011と表すことができ、10進数の7は111と表すことができるため、結果は10進数の15に対応する1100です。

- long **bitXor**(long *arg*, long *arg*);

  bitXor(long, long)関数はlongデータ型の引数を2つ受け入れます。これらを取得し、ビット単位exclusive orに対応する数値を返します。(たとえば、bitXor(11,7)は12を返します。)10進数の11はビット単位の1011と表すことができ、10進数の7は111と表すことができるため、結果は10進数の15に対応する1100です。

- number **ceil**(decimal *arg*);

  ceil(decimal)関数は小数データ型の引数を1つ取得し、その引数以上で算術整数に等しい最小の(負の無限大に最も近い)倍精度浮動小数点値を返します。

- number **ceil**(number *arg*);

  ceil(number)関数は倍精度浮動小数点データ型の引数を1つ取得し、その引数以上で算術整数に等しい最小の(負の無限大に最も近い)倍精度浮動小数点値を返します。

- number **e**();

  e()関数は引数を受け入れず、オイラー数を返します。

- number **exp**(<numeric type> *arg*);

  exp(<numeric type>)関数は任意の数値データ型(integer、long、numberまたはdecimal)の引数を1つ取得し、この引数の指数関数の結果を返します。

- number **floor**(decimal *arg*);

  floor(decimal)関数は小数データ型の引数を1つ取得し、その引数以下で算術整数に等しい最大の(正の無限大に最も近い)倍精度浮動小数点値を返します。

- number **floor**(number *arg*);

  floor(number)関数は倍精度浮動小数点データ型の引数を1つ取得し、その引数以下で算術整数に等しい最大の(正の無限大に最も近い)倍精度浮動小数点値を返します。

- void **setRandomSeed**(long *arg*);

  setRandomSeed(long)は1つのlong引数を取得し、値をランダムに生成するすべての関数のシードを生成します。

  この関数はpreExecute()関数またはメソッドで使用する必要があります。

  そのような場合、ランダムに生成されるすべての値はグラフの各実行で変わらず、グラフがリセットされた後でも同じ値のままです。

- number **log**(<numeric type> *arg*);

  log(<numeric type>)は任意の数値データ型(integer、long、numberまたはdecimal)の引数を1つ取得し、この引数の自然対数の結果を返します。

- number **log10**(<numeric type> *arg*);

  log10(<numeric type>)関数は任意の数値データ型(integer、long、numberまたはdecimal)の引数を1つ取得し、この引数の10を底とする対数の結果を返します。

- number **pi**();

  pi()関数は引数を受け入れず、パイ数を返します。

- number **pow**(<numeric type> *base*, <numeric type> *exp*);

  pow(<numeric type>, <numeric type>)関数は任意の数値データ型の2つの引数(2つが同じデータ型である必要はなく、integer、long、numberまたはdecimalのいずれか)を取得し、最初の引数を指数、2番目を指数の底とする指数関数を返します。

- number **random**();

  random()関数は引数を受け入れず、0.0以上1.0未満のランダムな正の倍精度浮動小数点数を返します。

- boolean **randomBoolean**();

  randomBoolean()関数は引数を受け入れず、ブール値trueまたはfalseをランダムに生成します。これらの値が数値データ型フィールドに送信されると、値は自動的に数値表現に変換されます(それぞれ1または0)。

- number **randomGaussian**();

  randomGaussian()関数は引数を受け入れず、ガウス分布に従う数値データ型の正および負両方の値をランダムに生成します。

- integer **randomInteger**();

  randomInteger()関数は引数を受け入れず、正および負両方の整数値をランダムに生成します。

- integer **randomInteger**(integer *Minimum*, integer *Maximum*);

  randomInteger(integer, integer)関数は整数データ型の引数を2つ受け入れ、Minimum以上Maximum以下のランダムな整数値を返します。

- long **randomLong**();

  randomLong()関数は引数を受け入れず、正および負両方のlong値をランダムに生成します。

- long **randomLong**(long *Minimum*, long *Maximum*);

  randomLong(long, long)関数はlongデータ型の引数を2つ受け入れ、Minimum以上Maximum以下のランダムなlong値を返します。

- long **round**(decimal *arg*);

  round(decimal)関数は小数データ型の引数を1つ取得し、この引数に最も近いlongを返します。小数は操作前に数値に変換されます。

- long **round**(number *arg*);

  round(number)関数は数値データ型の引数を1つ取得し、この引数に最も近いlongを返します。

- number **sqrt**(<numeric type> *arg*);

  sqrt(mumerictype)関数は任意の数値データ型(integer、long、numberまたはdecimal)の引数を1つ取得し、この引数の平方根を返します。

文字列関数

文字列を操作する関数もあります。

文字列を操作する関数では、日付または数値の書式パターンを定義する必要がある場合があります。

- 日付の書式設定または解析(あるいはその両方)の詳細は、日付と時刻の書式(p.113)を参照してください。
- 数値データ型の書式設定または解析(あるいはその両方)の詳細は、数値の書式(p.120)を参照してください。
- ロケールの詳細は、ロケール(p.126)を参照してください。

注意

数値および日付の書式は、その他の**ロケール**が明示的に指定されていないかぎり、システム値の**ロケール**または`defaultProperties`ファイルで指定された**ロケール**を使用して表示されます。

`defaultProperties`で**ロケール**を変更する方法の詳細は、デフォルトのCloverETL設定の変更(p.88)を参照してください。

これらの関数のリストを次に示します。

- `string` **`charAt`**`(string` *`arg`*`, integer` *`index`*`);`

  `charAt(string, integer)`関数は2つの引数を受け入れます。1つ目は文字列で、2つ目は整数です。これは文字列を取得し、`index`で指定された位置にある文字を返します。

- `string` **`chop`**`(string` *`arg`*`);`

  `chop(string)`関数は1つの文字列引数を受け入れます。関数はこの引数を取得し、引数で指定された文字列の末尾から改行文字および復帰文字を削除して、これらの文字が含まれない新しい文字列を返します。

- `string` **`chop`**`(string` *`arg1`*`, string` *`arg2`*`);`

  `chop(string, string)`関数は2つの文字列引数を受け入れます。最初の引数を取得し、2番目の引数で指定された文字列を最初の引数の末尾から削除して、2番目の引数で指定された文字列が含まれない最初の文字列引数を返します。

- `string` **`concat`**`(string` *`arg1`*`, string ..., string` *`argN`*`);`

  `concat(string, ..., string)`関数は、文字列データ型の引数をいくつでも受け入れます。これは、取得した引数を連結して返します。プラス記号を使用して引数を連結することもできますが、引数が3つ以上ある場合は、この関数を使用する方が高速です。

- `integer` **`countChar`**`(string` *`arg`*`, string` *`character`*`);`

  `countChar(string, string)`関数は2つの引数を受け入れます。1つ目は文字列で、2つ目は文字です。これらを取得し、最初の引数として指定されている文字列内での2番目の引数として指定されている文字の出現数を返します。

- `string[]` **`cut`**`(string` *`arg`*`, integer[]` *`indeces`*`);`

  `cut(string, integer[])`関数は2つの引数を受け入れます。1つ目は文字列で、2つ目は整数のリストです。この関数は文字列のリストを返します。2番目の引数で指定するリストの要素の数は偶数である必要があります。リストの整数は、位置(奇数の位置にある各数字)および長さ(偶数の位置にある各数字)を示します。指定した長さの部分文字列が、最初の引数で指定された文字列の指定位置から取得されます(指定位置の文字は除く)。取得された部分文字列が文字列のリストとして返されます。たとえば、`cut("somestringasanexample",[2,3,1,5])`は`["mes","omest"]`を返します。

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`);`

  `editDistance(string, string)` 関数は、2つの文字列引数を受け入れます。これらの文字列が互いに比較されます。比較の強度はデフォルトでは4、比較に使用するロケールのデフォルト値はシステム値、最大の差異はデフォルトでは3です。

  この関数は、2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

  詳細は、`editDistance()` 関数の別のバージョンである次のeditDistance (string, string, integer, string, integer) (p.936)関数を参照してください。

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, string` *`locale`*`);`

  `editDistance(string, string, string)` 関数は3つの引数を受け入れます。最初の2つは互いに比較される文字列で、3つ目(文字列)は比較に使用されるロケールです。デフォルトの比較の強度は4です。デフォルトの最大の差異は3です。

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

  詳細は、`editDistance()` 関数の別のバージョンである次のeditDistance (string, string, integer, string, integer) (p.936)関数を参照してください。

  ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, integer` *`strength`*`);`

  `editDistance(string, string, integer)` 関数は3つの引数を受け入れます。最初の2つは互いに比較される文字列で、3つ目(整数)は比較の強度です。比較に使用されるデフォルト・ロケールはシステム値です。最大の差異のデフォルトは3です。

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

  詳細は、`editDistance()` 関数の別のバージョンである次のeditDistance (string, string, integer, string, integer) (p.936)関数を参照してください。

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, integer` *`strength`*`, string` *`locale`*`);`

  `editDistance(string, string, integer, string)` 関数は4つの引数を受け入れます。最初の2つは互いに比較される文字列、3つ目(整数)は比較の強度、4つ目(文字列)は比較に使用されるロケールです。最大の差異のデフォルトは3です。

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値として`maxDifference + 1`を返します。

  詳細は、`editDistance()` 関数の別のバージョンである次のeditDistance (string, string, integer, string, integer) (p.936)関数を参照してください。

  ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- integer **editDistance**(string *arg1*, string *arg2*, string *locale*, integer *maxDifference*);

  editDistance(string, string, string, integer)関数は4つの引数を受け入れます。最初の2つは互いに比較される文字列、3つ目(文字列)は比較に使用されるロケール、4つ目(整数)は最大の差異です。比較の強度のデフォルトは4です。

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値としてmaxDifference + 1を返します。

  詳細は、editDistance()関数の別のバージョンである次のeditDistance (string, string, integer, string, integer)関数を参照してください。

  ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- integer **editDistance**(string *arg1*, string *arg2*, integer *strength*, integer *maxDifference*);

  editDistance(string, string, integer, integer)関数は4つの引数を受け入れます。最初の2つは互いに比較される文字列で、残りの2つはいずれも整数です。これらは比較の強度(3番目の引数)および最大の差異(4番目の引数)です。ロケールはデフォルトのシステム値です。

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値としてmaxDifference + 1を返します。

  詳細は、editDistance()関数の別のバージョンである次のeditDistance (string, string, integer, string, integer)関数を参照してください。

- integer **editDistance**(string *arg1*, string *arg2*, integer *strength*, string *locale*, integer *maxDifference*);

  editDistance(string, string, integer, string, integer)関数は5つの引数を受け入れます。最初の2つは文字列、残りの3つはそれぞれ整数、文字列および整数です。この関数は最初の2つの引数を取得し、他の3つの引数を使用してそれらを比較します。

  3番目の引数(整数)は比較の強度を指定します。これには1から4の任意の値を指定できます。

  4 (同一比較)の場合は、同一の文字のみが一致とみなされることを意味します。3の場合は(3次比較)、大文字と小文字が一致とみなされることを意味します。2の場合は(2次比較)、発音区別文字の付いた文字が一致とみなされることを意味します。最後に、比較の強度が1 (1次比較)の場合は、特定の記号が付いた文字も一致とみなされることを意味します。この比較の強度が指定されないeditDistance()関数の別のバージョンでは、4の数値がデフォルトの強度として使用されます(前述の説明を参照してください)。

  4番目の引数は文字列データ型です。これは比較に使用されるロケールです。editDistance()関数の別のバージョンでロケールが指定されない場合、デフォルト値はシステム値です(前述の説明を参照してください)。

  5番目の引数(整数)は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を意味します。editDistance()関数の別のバージョンでこの最大の差異が指定されない場合、デフォルトの最大の差異は数値3です(前述の説明を参照してください)。

  この関数は、最初の2つの引数の一方をもう1つの引数に変換するために変更する必要がある文字数を返します。ただし、関数が実行され、変更する必要がある文字数が最大の差異として指定された数以上であることが検出された場合、実行は終了し、関数は戻り値としてmaxDifference + 1を返します。

実際に、この関数はCA、CZ、ES、DA、DE、ET、FI、FR、HR、HU、IS、IT、LT、LV、NL、NO、PL、PT、RO、SK、SL、SQ、SV、TRの各ロケールに実装されています。これらのロケールには、すべて言語固有の文字が含まれるという1つの共通点があります。これらの文字の完全なリストは、CTL2付録: 各国語固有キャラクタのリスト(p.960)を参照してください。

ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- `string` **`escapeUrl`**`(string` *`arg`*`);`

  この関数は、指定されたURLのコンポーネントに含まれる無効な文字をエスケープします(URLコンポーネントの説明は、isUrl() CTL2関数(p.939)を参照してください)。無効な文字は、パーセント文字`%`記号の後に、その文字のISO-Latinコード・ポイントの2桁の16進表現(大文字小文字の区別なし)を付けたものでエスケープする必要があります。たとえば、`%20`はUS-ASCIIスペース文字のエスケープ・エンコーディングです。

- `string[]` **`find`**`(string` *`arg`*`, string` *`regex`*`);`

  `find(string, string)`関数は、2つの文字列引数を受け入れます。2番目の引数は正規表現(p.963)です。関数はこれらを取得し、最初の引数で指定された文字列に含まれる、正規表現パターンに対応する部分文字列のリストを返します。

- `string` **`getAlphanumericChars`**`(string` *`arg`*`);`

  `getAlphanumericChars(string)`関数は文字列引数を1つ取得し、その文字列に含まれる文字および数字のみを、文字列での出現順で返します。その他の文字は削除されます。

- `string` **`getAlphanumericChars`**`(string` *`arg`*`, boolean` *`takeAlpha`*`, boolean` *`takeNumeric`*`);`

  `getAlphanumericChars(string, boolean, boolean)`関数は3つの引数を受け入れます。1つの文字列と2つのブールです。これらを取得し、2番目の引数または3番目の引数(あるいはその両方)がtrueに設定されている場合、それぞれ文字または数字(あるいはその両方)を返します。

- `string` **`getFieldLabel`**`(reference` *`record`*`, string` *`arg`*`);`

  この関数は、`arg`で指定された名前を持つフィールドのラベルを返します。フィールドは`record`から取得されます。

- `string` **`getFieldLabel`**`(reference` *`record`*`, integer` *`arg`*`);`

  この関数は、`arg`で指定されたインデックスを持つフィールドのラベルを返します。フィールドは`record`から取得されます。

- `string` **`getUrlHost`**`(string` *`arg`*`);`

  この関数は、指定されたURLを解析してホスト名を取得します(スキームは、isUrl() CTL2関数(p.939)を参照してください)。URL引数にホスト名の部分が存在しない場合、空の文字列が返されます。URLが無効な場合は`null`が返されます。

- `string` **`getUrlPath`**`(string` *`arg`*`);`

  この関数は、指定されたURLを解析してパスを取得します(スキームは、isUrl() CTL2関数(p.939)を参照してください)。URL引数にパスの部分が存在しない場合、空の文字列が返されます。URLが無効な場合は`null`が返されます。

- `integer` **`getUrlPort`**`(string` *`arg`*`);`

  この関数は、指定されたURLを解析してポート番号を取得します(スキームは、isUrl() CTL2関数(p.939)を参照してください)。URL引数にポートの部分が存在しない場合、-1が返されます。URLの構文が無効な場合は、-2が返されます。

- `string` **`getUrlProtocol`**`(string` *`arg`*`);`

  この関数は、指定されたURLを解析してプロトコル名を取得します(スキームは、isUrl() CTL2関数(p.939)を参照してください)。URL引数にプロトコルの部分が存在しない場合、空の文字列が返されます。URLが無効な場合は`null`が返されます。

- `string` **`getUrlQuery`**`(string` *`arg`*`);`

  この関数は、指定されたURLを解析して問合せ(パラメータ)を取得します(スキームは、isUrl() CTL2関数(p.939)を参照してください)。URL引数に問合せの部分が存在しない場合、空の文字列が返されます。URL構文が無効な場合は`null`が返されます。

- `string` **`getUrlUserInfo`**`(string` *`arg`*`);`

  この関数は、指定されたURLを解析してユーザー名とパスワードを取得します(スキームは、isUrl() CTL2関数(p.939)を参照してください)。URL引数にユーザー情報の部分が存在しない場合、空の文字列が返されます。URL構文が無効な場合は`null`が返されます。

- `string` **`getUrlRef`**`(string` *`arg`*`);`

  この関数は、指定されたURLを解析して、#文字(ref、参照またはアンカーとも呼ばれる)より後の部分を取得します(スキームは、isUrl() CTL2関数(p.939)を参照してください)。URL引数にこの部分が存在しない場合、空の文字列が返されます。URL構文が無効な場合は`null`が返されます。

- `integer` **`indexOf`**`(string` *`arg`*`, string` *`substring`*`);`

  `indexOf(string, substring)`関数は、`string`内で最初に出現した`substring`のインデックス(ゼロから始まる)を返します。出現が検出されない場合は`-1`を返します。

- `integer` **`indexOf`**`(string` *`arg`*`, string` *`substring`*`, integer` *`fromIndex`*`);`

  `indexOf(string, substring, fromIndex)`関数は、`string`内の、`fromIndex`以降で最初に出現した`substring`のインデックス(ゼロから始まる)を返します。出現が検出されない場合は`-1`を返します。

- `boolean` **`isAscii`**`(string` *`arg`*`);`

  `isAscii(string)`関数は1つの文字列引数を取得し、その文字列をASCII文字列としてエンコードできるかどうかに応じてブール値を返します。エンコードできる場合はtrue、できない場合はfalseです。

- `boolean` **`isBlank`**`(string` *`arg`*`);`

  `isBlank(string)`関数は1つの文字列引数を取得し、その文字列に空白文字のみが含まれているかどうかに応じてブール値を返します。空白文字のみが含まれている場合はtrue、そうでない場合はfalseです。

- `boolean` **`isDate`**`(string` *`arg`*`, string` *`pattern`*`);`

  `isDate(string, string)`関数は2つの文字列引数を受け入れます。これらを取得し、2番目の引数をパターンとして最初の引数と比較します。指定した`pattern`に従って、システム値の`locale`内で有効な日付に最初の文字列を変換できる場合、関数はtrueを返します。できない場合はfalseを返します。

  (詳細は、`isDate()`関数の別のバージョンである次の`isDate(string, string, string, boolean)`関数を参照してください。)

  この関数は前述の`isDate(string, string, string)`関数のバリアントで、3番目の引数のデフォルト値がシステム値に設定されたものです。

- `boolean` **`isDate`**`(string` *`arg`*`, string` *`pattern`*`, string` *`locale`*`);`

  `isDate(string, string, string)`関数は3つの文字列引数を受け入れます。これらを取得し、3番目の引数(`locale`)を使用し、2番目の引数をパターンとして最初の引数と比較します。指定した`pattern`に従って、指定した`locale`内で有効な日付に最初の引数を変換できる場合、関数はtrueを返します。できない場合はfalseを返します。

  (詳細は、`isDate()`関数の別のバージョンである次の`isDate(string, string, string, boolean)`関数を参照してください。)

  ロケールの詳細は、http://docs.oracle.com/javase/6/docs/api/java/util/Locale.htmlを参照してください。

- `boolean` **`isInteger`**`(string` *`arg`*`);`

  `isInteger(string)`関数は1つの文字列引数を取得し、その文字列を整数に変換できるかどうかに応じてブール値を返します。変換できる場合はtrue、できない場合はfalseです。

- `boolean` **`isLong`**`(string` *`arg`*`);`

  `isLong(string)`関数は1つの文字列引数を取得し、その文字列をlong数に変換できるかどうかに応じてブール値を返します。変換できる場合はtrue、できない場合はfalseです。

- `boolean` **`isNumber`**`(string` *`arg`*`);`

  `isNumber(string)`関数は1つの文字列引数を取得し、その文字列を倍精度浮動小数点に変換できるかどうかに応じてブール値を返します。変換できる場合はtrue、できない場合はfalseです。

- `boolean` **`isUrl`**`(string` *`arg`*`);`

  この関数は、指定された文字列が次の構文を持つ有効なURLであるかどうかを確認します。

```
foo://username:passw@host.com:8042/there/index.dtb?type=animal;name=cat#nose
\_/   \_____________/ \______/ \__/\_________________/ \____________________/ \__/
 |          |            |      |         |                    |             |
protocol  userinfo      host   port      path                query          ref
```

  URI標準の詳細は、http://www.ietf.org/rfc/rfc2396.txtを参照してください。

- `string` **`join`**`(string` *`delimiter`*`, <element type>[]` *`arg`*`);`

  `join(string, <element type>[])`関数は2つの引数を受け入れます。1つ目は文字列、2つ目は任意のデータ型の要素のリストです。文字列以外の要素はその文字列表現に変換され、デリミタとして最初の引数と結合されます。

- `string` **`join`**`(string` *`delimiter`*`, map[<type of key>,<type of value>]` *`arg`*`);`

  `join(string, map[<type of key>,<type of value>])`関数は2つの引数を受け入れます。1つ目は文字列、2つ目は任意のデータ型のマップです。マップ要素は`key=value`文字列として示されます。これらはデリミタとして最初の引数と結合されます。

- `string` **`left`**`(string` *`arg`*`, integer` *`length`*`);`

  `left(string, integer)`関数は2つの引数を受け入れます。1つ目は文字列で、2つ目は整数です。これらを取得し、最初の引数で指定された文字列の先頭から数えて、2番目の引数で指定された長さの部分文字列を返します。入力文字列が*`length`*パラメータより短い場合、例外がスローされ、グラフは失敗します。このような失敗を回避するには、次に説明する`left(string, integer, boolean)`関数を使用します。

- `string` **`left`**`(string` *`arg`*`, integer` *`length`*`, boolean` *`spacePad`*`);`

  この関数は、指定された長さの接頭辞を返します。入力文字列の長さが*`length`*パラメータ以上である場合、この関数は`left(string, integer)`関数と同様に動作します。入力文字列が指定した長さより短い場合は動作が異なります。3番目の引数が`true`の場合、結果が左揃えで指定された長さになるよう、結果の右側がスペースで埋められます。一方`false`の場合は、スペースは追加されず、入力文字列が結果として返されます。

- `integer` **`length`**`(structuredtype` *`arg`*`);`

  `length(structuredtype)`関数は、構造化されたデータ型を引数として受け入れます。これは、`string`、`<element type>[]`、`map[<type of key>,<type of value>]`または`record`です。これは引数を取得し、構造化されたデータ型を形成する要素を返します。

- `string` **`lowerCase`**`(string` *`arg`*`);`

  `lowerCase(string)`関数は、1つの文字列引数を取得し、それを小文字のみに変換した別の文字列を返します。

- `boolean` **`matches`**`(string` *`arg`*`, string` *`regex`*`);`

  `matches(string, string)`関数は2つの文字列引数を取得します。2番目の引数は正規表現(p.963)です。最初の引数をその正規表現で表せる場合、関数は`true`を返し、そうでない場合は`false`を返します。

- `string[]` **`matchGroups`**`(string` *`text`*`, string` *`regex`*`);`

  `text`が正規表現(p.963)`regex`に一致する場合、`matchGroups(text, regex)`関数はグループ一致(`regex`のキャプチャ・グループに一致する部分文字列)のリストを返します。リストはゼロから始まり、インデックスが0の要素は式全体の一致です。後続の要素(1, ...)は、左から右に1から始まるインデックスが付けられたキャプチャ・グループに対応しています。返されるリストは変更できません。`text`が`regex`に一致しない場合は、`null`が返されます。

- `string` **`metaphone`**`(string` *`arg`*`, integer` *`maxLength`*`);`

  `metaphone(string, integer)`関数は1つの文字列引数および最大長を意味する1つの整数を受け入れます。この関数はこれらの引数を取得し、最初の引数のmetaphoneコードを指定された最大長で返します。デフォルトの最大長は4です。詳細は、次のサイトを参照してください。
  www.lanw.com/java/phonetic/default.htm

- `string` **`NYSIIS`**`(string` *`arg`*`);`

  `NYSIIS(string)`関数は、1つの文字列引数を取得し、その引数のNew York State Identification and Intelligence System表音コードを返します。詳細は、次のサイトを参照してください。
  http://en.wikipedia.org/wiki/New_York_State_Identification_and_Intelligence_System

- `string` **`randomString`**`(integer` *`minLength`*`, integer` *`maxLength`*`);`

  この関数は小文字で構成される文字列を返します。長さは`<minLength; maxLength>`の間です。生成される文字列の文字は常に['a'-'z']に属します(特殊記号は含まれません)。

  ランダムな文字列の例: `qjfxq`

- `string` **`randomUUID`**`();`

  ランダムに一意の文字列識別子を生成します。生成される文字列の書式は次のようになります。

  `hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh`

  ここでhは`[0-9a-f]`に属します。つまり、ランダムな128ビット数の16進コードが生成されます。

  生成される文字列の例: cee188a3-aa67-4a68-bcd2-52f3ec0329e6

  使用されるアルゴリズムの詳細は、Javaのドキュメントを参照してください。

- `string` **`removeBlankSpace`**`(string` *`arg`*`);`

  `removeBlankSpace(string)`関数は1つの文字列引数を取得し、空白を削除した別の文字列を返します。

- `string` **`removeDiacritic`**`(string` *`arg`*`);`

  `removeDiacritic(string)`関数は1つの文字列引数を取得し、発音区別文字を削除した別の文字列を返します。

- `string` **`removeNonAscii`**`(string` *`arg`*`);`

  `removeNonAscii(string)`関数は1つの文字列引数を取得し、非ASCII文字を削除した別の文字列を返します。

- `string` **`removeNonPrintable`**`(string` *`arg`*`);`

  `removeNonPrintable(string)`関数は1つの文字列引数を取得し、印刷不可能な文字を削除した別の文字列を返します。

- `string` **`replace`**`(string` *`arg`*`, string` *`regex`*`, string` *`replacement`*`);`

  `replace(string, string, string)`関数は、文字列、正規表現(p.963)および置換の3つの文字列引数を取得し、文字列内にあるすべての正規表現の一致を指定した置換文字列で置換します。文字列の正規表現に一致するすべての部分が置換されます。置換文字列で後方参照を使用して、一致したテキストを参照することもできます。一致全体に対する後方参照は$0と指定します。キャプチャするカッコを使用している場合は、特定のグループを$1、$2、$3などのように参照できます。

  `replace("Hello","[Ll]","t")`は"Hetto"を返します。

  `replace("Hello", "e(l+)", "a$1")`は"Hallo"を返します。

  重要: $0や$1などの類似の構文には注意してください。置換文字列内で使用された場合、これは一致する正規表現のカッコを(出現順に)示します。文字列の外側で使用された場合は、入力フィールドへの参照を意味します。次の例を参照してください。

  `replace("Hello", "e(l+)", $0.name)`は、ポート0の入力フィールドnameにJohnという名前が含まれている場合、HJohnoを返します。

  正規表現の先頭で修飾子を使用することもできます。大文字小文字を区別しない検索の場合は`(?i)`、複数行モードの場合は`(?m)`、ドット(.)が改行文字にも一致するdotallモードの場合は`(?s)`です。

  `replace("Hello", "(?i)L", "t")`はHettoを生成し、一方`replace("Hello", "L", "t")`は単にHelloを生成します。

- `string` **`right`**`(string` *`arg`*`, integer` *`length`*`);`

  `right(string, integer)`関数は2つの引数を受け入れます。1つ目は文字列で、2つ目は整数です。これらを取得し、最初の引数で指定された文字列の最後から数えて、2番目の引数で指定された長さの部分文字列を返します。入力文字列が*`length`*パラメータより短い場合、例外がスローされ、グラフは失敗します。このような失敗を回避するには、次に説明する`right(string, integer, boolean)`関数を使用します。

- `string` **`right`**`(string` *`arg`*`, integer` *`length`*`, boolean` *`spacePad`*`);`

  この関数は、指定された長さの接尾辞を返します。入力文字列の長さが*`length`*パラメータ以上である場合、この関数は`right(string, integer)`関数と同様に動作します。入力文字列が指定した長さより短い場合は動作が異なります。3番目の引数が`true`の場合、結果が右揃えで指定された長さになるよう、結果の左側がスペースで埋められます。一方`false`の場合は、スペースは追加されず、入力文字列が結果として返されます。

- `string` **`soundex`**`(string` *`arg`*`);`

  `soundex(string)`関数は、1つの文字列引数を取得し、その文字列を別の文字列に変換します。生成される文字列は、引数として指定された文字列の最初の文字と3つの数字で構成されます。3つの数字は文字列に含まれる子音に基づき、数字は発音が似ている子音に対応しています。したがって、`soundex("word")`は`"w600"`を返します。

- `string[]` **`split`**`(string` *`arg`*`, string` *`regex`*`);`

  `split(string, string)`関数は、2つの文字列引数を受け入れます。2番目は正規表現(p.963)です。これが最初の文字列引数で検索され、検出されると、文字列がその正規表現の文字または部分文字列の間にある部分に分割されます。生成される文字列の部分は、文字列のリストとして返されます。したがって、`split("abcdefg", "[ce]")`は`["ab", "d", "fg"]`を返します。

- `string` **`substring`**`(string` *`arg`*`, integer` *`fromIndex`*`, integer` *`length`*`);`

  `substring(string, integer, integer)`関数は3つの引数を受け入れます。1つ目は文字列で、他の2つは整数です。この関数は引数を取得し、2番目の引数で定義される位置から`length`文字を取得して、元の文字列から定義された長さの部分文字列を取得して返します。したがって、`substring("text", 1, 2)`は`"ex"`を返します。

- `string` **`translate`**`(string` *`arg`*`, string` *`searchingSet`*`, string` *`replaceSet`*`);`

  `translate(string, string, string)`関数は、3つの文字列引数を受け入れます。2番目および3番目の引数両方で文字数が同じである必要があります。2番目の引数として指定された文字列に含まれる文字が最初の引数として指定された文字列内で検出されると、3番目の引数として指定された文字列から取得される文字でその文字が置換されます。3番目の文字列の文字は、2番目の文字列の文字と同じ位置にある必要があります。したがって、`translate("hello", "leo", "pii")`は`"hippi"`を返します。

- `string` **`trim`**`(string` *`arg`*`);`

  `trim(string)`関数は1つの文字列引数を取得し、先頭および末尾の空白を削除した別の文字列を返します。

- `string` **`unescapeUrl`**`(string` *`arg`*`);`

  この関数は、指定されたURLのコンポーネントに含まれる無効な文字のエスケープ・シーケンスをデコードします(URLコンポーネントの説明は、isUrl() CTL2関数(p.939)を参照してください)。エスケープ・シーケンスは、パーセント文字`%`記号の後に、その文字のISO-Latinコード・ポイントの2桁の16進表現(大文字小文字の区別なし)を付けたもので構成されます。たとえば、`%20`はUS-ASCIIスペース文字のエスケープ・エンコーディングです。

- `string` **`upperCase`**`(string` *`arg`*`);`

  `upperCase(string)`関数は、1つの文字列引数を取得し、それを大文字のみに変換した別の文字列を返します。

コンテナ関数

コンテナ(list、map、record)を使用する場合は通常、含まれている関数呼出しのinを使用します。この呼出しは、ある値が他の値のリストまたはマップに含まれているかどうかを識別します。次の2つの構文オプションがあります。

```
boolean myBool;
string myString;
string[] myList;
...


myBool = myString.in(myList)


myBool = in(myString,myList);
```

次の表に、inを使用できる場合とできない場合を示します。

コード	機能するかどうか
`"abc".in(["a", "b"])`	✔
`in(10, [10, 20])`	✔
`10.in([10, 20])`	✖

メタデータ内のリストおよびマップの場合は、次のようにinを使用します。

コード	コメント
`"abc" in ["a", "b"]`	標準的な演算子構文
`"abc" in $in.0.listField`	リスト・フィールドの演算子構文
`"abc" in $in.0.mapField`	マップ・フィールドの演算子

注意

演算子構文には短所があります。(線形である必要があるため)リスト全体の検索には常に時間がかかります。

コンテナで使用できる関数

- `<element type>[] append(<element type>[] arg, <element type> list_element);`

  `append(<element type>[], <element type>)` 関数は2つの引数を受け入れます。1つ目は任意の要素データ型のリストで、2つ目はその要素データ型です。この関数は2番目の引数を取得し、最初の引数の末尾にそれを追加します。関数は最初の引数として指定されたリストの新しい値を返します。

  この関数は`push(<element type>[], <element type>)` 関数の別名です。リストの観点からは、`append()` がより自然です。

- boolean **containsAll**(<element type>[] *list*, <element type>[] *subList*);

  containsAll(<element type>[], <element type>[])関数は、最初の引数として渡されるリストに2番目に渡されるリストのすべての要素が含まれる場合、つまり2番目のリストが最初のリストのサブリストである場合はtrueを返します。

- boolean **containsKey**(map[<type of key>,<type of value>] *map*, <type of key> *key*);

  containsKey(map[<type of key>,<type of value>], <type of key>)関数は、指定されたマップに指定されたキーのマッピングが含まれる場合にtrueを返します。

- boolean **containsValue**(map[<type of key>,<type of value>] *map*, <type of value> *value*);

  containsKey(map[<type of key>,<type of value>], <type of value>)関数は、指定されたマップが指定された値に1つ以上のキーをマップする場合にtrueを返します。

- void **clear**(<element type>[] *arg*);

  clear(<element type>[])関数は、任意の要素データ型のリスト引数を1つ受け入れます。この関数はこの引数を取得し、リストを空にします。これはvoidを返します。

- void **clear**(map[<type of key>,<type of value>] *arg*);

  clear(map[<type of key>,<type of value>])関数は1つのマップ引数を受け入れます。この関数はこの引数を取得し、マップを空にします。これはvoidを返します。

- <element type>[] **copy**(<element type>[] *arg*, <element type>[] *arg*);

  copy(<element type>[], <element type>[])関数は2つの引数を受け入れます。いずれもリストです。両方のリストの要素が同じデータ型である必要があります。関数は2番目の引数を取得し、それを最初のリストの末尾に追加して、最初の引数として指定されたリストの新しい値を返します。

- void **copyByName**(record *to*, record *from*);

  フィールド名に基づいて入力レコードから出力レコードにデータをコピーします。同じ名前のフィールドのマッピングのみを可能にします。

- void **copyByPosition**(record *to*, record *from*);

  フィールドの順序に基づいて入力レコードから出力レコードにデータをコピーします。出力メタデータのフィールド数により、どの入力フィールドが(最初のフィールドから順に)マップされるかが決まります。

- map[<type of key>, <type of value>] **copy**(map[<type of key>, <type of value>] *arg*, map[<type of key>, <type of value>] *arg*);

  copy(map[<type of key>, <type of value>], map[<type of key>, <type of value>])関数は2つの引数を受け入れます。いずれもマップです。両方のマップの要素が同じデータ型である必要があります。関数は2番目の引数を取得し、それを最初のマップの末尾に追加して既存のキー・マッピングを置換し、最初の引数として指定されたマップの新しい値を返します。

- list[] **getKeys**(map[<type of key>, <type of value>] *arg*);

  この関数はマップのキーのlistを返します。リストはマップのキーと同じ型である必要があります。次に例を示します。

```
map[string, integer] myMap;

// filling the map with values, e.g. myMap["first"] = 1;

string[] listOfKeys = getKeys(myMap);
```

- <element type>[] **insert**(<element type>[] *arg*, integer *position*, <element type> *newelement*);

  insert(<element type>[], integer, <element type>)関数は、最初が任意の要素データ型のリスト、2番目が整数、その他がその要素データ型の引数を受け入れます。関数は3番目の引数を取得し、リスト内の2番目の引数で定義された位置にそれを挿入します。最初の引数として指定されるリストがこの新しい値に変更され、関数によって返されます。リスト要素には0から始まるインデックスが付きます。

- boolean **isEmpty**(<element type>[] *arg*);

  isEmpty(<element type>[])関数は任意の要素データ型のリストの引数を1つ受け入れます。この引数を取得し、リストが空かどうかを確認して、trueまたはfalseを返します。

- boolean **isEmpty**(map[<type of key>,<type of value>] *arg*);

  isEmpty(map[<type of key>,<type of value>])関数は、任意の値データ型のマップ引数を1つ受け入れます。この引数を取得し、マップが空かどうかを確認して、trueまたはfalseを返します。

- integer **length**(structuredtype *arg*);

  length(structuredtype)関数は、構造化されたデータ型を引数として受け入れます。これは、string、<element type>[]、map[<type of key>,<type of value>]またはrecordです。これは引数を取得し、構造化されたデータ型を形成する要素を返します。

- <element type> **poll**(<element type>[] *arg*);

  poll(<element type>[])関数は任意の要素データ型のリストの引数を1つ受け入れます。この引数を取得し、リストから最初の要素を削除して、その要素を返します。引数として指定されるリストがこの新しい値に変更されます(削除された最初の要素は含まれません)。

- <element type> **pop**(<element type>[] *arg*);

  pop(<element type>[])関数は任意の要素データ型のリストの引数を1つ受け入れます。この引数を取得し、リストから最後の要素を削除して、その要素を返します。引数として指定されるリストがこの新しい値に変更されます(削除された最後の要素は含まれません)。

- <element type>[] **push**(<element type>[] *arg*, <element type> *list_element*);

  push(<element type>[], <element type>)関数は2つの引数を受け入れます。1つ目は任意のデータ型のリストで、2つ目はリスト要素のデータ型です。この関数は2番目の引数を取得し、最初の引数の末尾にそれを追加します。関数は最初の引数として指定されたリストの新しい値を返します。

  この関数はappend(<element type>[], <element type>)関数の別名です。スタックまたはキューの観点からは、push()がより自然です。

- <element type> **remove**(<element type>[] *arg*, integer *position*);

  remove(<element type>[], integer)関数は2つの引数を受け入れます。1つ目は任意の要素データ型のリストで、2つ目は整数です。関数は指定された位置にある要素を削除し、削除した要素を返します。最初の引数として指定されたリストの値が新しい値に変更されます。(リスト要素には0から始まるインデックスが付けられます。)

- `<element type>[] reverse(<element type>[] arg);`

  `reverse(<element type>[])`関数は任意の要素データ型のリストの引数を1つ受け入れます。この引数を取得し、リストの要素の順序を反転させて、最初の引数として指定されたリストを反転させた新しい値を返します。

- `<element type>[] sort(<element type>[] arg);`

  `sort(<element type>[])`関数は任意の要素データ型のリストの引数を1つ受け入れます。この引数を取得し、リストの要素をその値に基づいて昇順でソートし、最初の引数として指定されたリストをソートした新しい値を返します。

レコード関数(動的フィールド・アクセス)

これらの関数は、変換エディタ(p.286)内にある**動的フィールド・アクセス・ライブラリ**・セクションの「**Functions**」タブにあります。

- `integer` **`length`**`( );`

  関数が呼び出されたレコードのフィールド数を返します。

- `integer` **`compare`**`(reference` *`record1`*`, string` *`field1`*`, reference` *`record2`*`, string` *`field2`*`);`

  指定されたレコードの2つのフィールドを比較します。フィールドは名前で識別されます。この関数は、次のいずれかの整数値を返します。

  1. < 0 ... `field2`が`field1`より大きい
  2. > 0 ... `field2`が`field1`より小さい
  3. 0 ...フィールドが等しい

- `integer` **`compare`**`(reference` *`record1`*`, integer` *`field1`*`, reference` *`record2`*`, integer` *`field2`*`);`

  指定されたレコードの2つのフィールドを比較します。フィールドはインデックスで識別されます(0が最初のフィールドです)。この関数は、次のいずれかの整数値を返します。

  1. < 0 ... `field2`が`field1`より大きい
  2. > 0 ... `field2`が`field1`より小さい
  3. 0 ...フィールドが等しい

- `boolean` **`getBoolValue`**`(reference` *`record`*`, integer` *`field`*`);`

  ブール・フィールドの値を返します。フィールドはインデックスで識別されます。

- `boolean` **`getBoolValue`**`(reference` *`record`*`, string` *`field`*`);`

  ブール・フィールドの値を返します。フィールドは名前で識別されます。

- `byte` **`getByteValue`**`(reference` *`record`*`, integer` *`field`*`);`

  バイト・フィールドの値を返します。フィールドはインデックスで識別されます。

- `byte` **`getByteValue`**`(reference` *`record`*`, string` *`field`*`);`

  バイト・フィールドの値を返します。フィールドは名前で識別されます。

- `date` **`getDateValue`**`(reference` *`record`*`, integer` *`field`*`);`

  日付フィールドの値を返します。フィールドはインデックスで識別されます。

- `date` **`getDateValue`**`(reference` *`record`*`, string` *`field`*`);`

  日付フィールドの値を返します。フィールドは名前で識別されます。

- `decimal` **`getDecimalValue`**`(reference` *`record`*`, integer` *`field`*`);`

  小数フィールドの値を返します。フィールドはインデックスで識別されます。

- `decimal` **`getDecimalValue`**`(reference` *`record`*`, string` *`field`*`);`

  小数フィールドの値を返します。フィールドは名前で識別されます。

- `integer` **`getFieldIndex`**`(reference` *`record`*`, string` *`field`*`);`

  名前で識別されるフィールドの(ゼロから始まる)インデックスを返します。レコード内にフィールド名が見つからない場合、関数は-1を返します。

- `string` **`getFieldLabel`**`(reference` *`record`*`, integer` *`field`*`);`

  インデックスで識別されるフィールドのラベルを返します。ラベルはフィールドの名前ではありません。フィールド名とラベルと説明(p.161)を参照してください。

- `string` **`getFieldName`**`(record` *`argRecord`*`, integer` *`index`*`);`

  `getFieldName(record, integer)`関数はレコードおよび整数の2つの引数を受け入れます。関数はこれらを取得し、指定されたインデックスの付いたフィールドの名前を返します。フィールドには0から始まる番号が付けられます。

重要

`argRecord`は次のいずれかの書式を取ることができます。

- `$<port number>.*`

  例: `$0.*`

- `$<metadata name>.*`

  例: `$customers.*`

- `<record variable name>[.*]`

  例: `Customers`または`Customers.*` (いずれも、CTLで`Customers`がレコードとして宣言されている場合)

- `lookup(<lookup table name>).get(<key value>)[.*]`

  例: `lookup(Comp).get("JohnSmith")`または
  `lookup(Comp).get("JohnSmith").*`

- `lookup(<lookup table name>).next()[.*]`

  例: `lookup(Comp).next()`または`lookup(Comp).next().*`

- `string` **`getFieldType`**`(record` *`argRecord`*`, integer` *`index`*`);`

  ユーザーがインデックス(0から始まるフィールドの番号)で指定したフィールドの型を返します。返される文字列は型の名前(`string`、`integer`など)です。メタデータのデータ型(p.111)を参照してください。コードの例:

```
string dataType = getFieldType($in.0, 2);
```

  受信レコードごとに3番目のフィールドのデータ型を返します(`decimal`など)。

重要

引数としてのレコードは`getFieldName()`関数のレコードのように示されます。前述の説明を参照してください。

- integer **getIntegerValue**(reference *record*, integer *field*);

  整数フィールドの値を返します。フィールドはインデックスで識別されます。

- integer **getIntegerValue**(reference *record*, string *field*);

  整数フィールドの値を返します。フィールドは名前で識別されます。

- long **getLongValue**(reference *record*, integer *field*);

  longフィールドの値を返します。フィールドはインデックスで識別されます。

- long **getLongValue**(reference *record*, string *field*);

  longフィールドの値を返します。フィールドは名前で識別されます。

- number **getNumValue**(reference *record*, integer *field*);

  numberフィールドの値を返します。フィールドはインデックスで識別されます。

- number **getNumValue**(reference *record*, string *field*);

  numberフィールドの値を返します。フィールドは名前で識別されます。

- string **getStringValue**(reference *record*, integer *field*);

  stringフィールドの値を返します。フィールドはインデックスで識別されます。

- string **getStringValue**(reference *record*, string *field*);

  stringフィールドの値を返します。フィールドは名前で識別されます。

- string **getValueAsString**(reference *record*, string *field*);

  フィールドの値を(フィールドの型に関係なく)共通のstringとして返そうとします。フィールドは名前で識別されます。

- string **getValueAsString**(reference *record*, integer *field*);

  フィールドの値を(フィールドの型に関係なく)共通のstringとして返そうとします。フィールドはインデックスで識別されます。

- boolean **isNull**(reference *record*, string *field*);

  指定されたフィールドがnullかどうかを確認します。フィールドは名前で識別されます。

- boolean **isNull**(reference *record*, integer *field*);

  指定されたフィールドがnullかどうかを確認します。フィールドはインデックスで識別されます。

- void **setBoolValue**(reference *record*, integer *field*, boolean *value*);

  boolean値をフィールドに設定します。フィールドはインデックスで識別されます。

- void **setBoolValue**(reference *record*, string *field*, boolean *value*);

  boolean値をフィールドに設定します。フィールドは名前で識別されます。

- void **setByteValue**(reference *record*, integer *field*, byte *value*);

  byte値をフィールドに設定します。フィールドはインデックスで識別されます。

- void **setByteValue**(reference *record*, string *field*, byte *field*);

  byte値をフィールドに設定します。フィールドは名前で識別されます。

- void **setDateValue**(reference *record*, integer *field*, date *value*);

  date値をフィールドに設定します。フィールドはインデックスで識別されます。
- void **setDateValue**(reference *record*, string *field*, date *value*);

  date値をフィールドに設定します。フィールドは名前で識別されます。
- void **setDecimalValue**(reference *record*, integer *field*, decimal *value*);

  decimal値をフィールドに設定します。フィールドはインデックスで識別されます。
- void **setDecimalValue**(reference *record*, string *field*, decimal *value*);

  decimal値をフィールドに設定します。フィールドは名前で識別されます。
- void **setIntValue**(reference *record*, integer *field*, integer *value*);

  integer値をフィールドに設定します。フィールドはインデックスで識別されます。
- void **setIntValue**(reference *record*, string *field*, integer *value*);

  integer値をフィールドに設定します。フィールドは名前で識別されます。
- void **setLongValue**(reference *record*, integer *field*, long *value*);

  long値をフィールドに設定します。フィールドはインデックスで識別されます。
- void **setLongValue**(reference *record*, string *field*, long *value*);

  long値をフィールドに設定します。フィールドは名前で識別されます。
- void **setNumValue**(reference *record*, integer *field*, number *value*);

  number値をフィールドに設定します。フィールドはインデックスで識別されます。
- void **setNumValue**(reference *record*, string *field*, number *value*);

  number値をフィールドに設定します。フィールドは名前で識別されます。
- void **setStringValue**(reference *record*, integer *field*, string *value*);

  string値をフィールドに設定します。フィールドはインデックスで識別されます。
- void **setStringValue**(reference *record*, string *field*, string *value*);

  string値をフィールドに設定します。フィールドは名前で識別されます。

その他の関数

残りはその他の関数と呼ぶことができます。これらの関数を次に示します。

重要

このような**その他**の関数では、オブジェクト表記(たとえば、`arg.isnull()`)は使用できません。

オブジェクト表記の詳細は、関数リファレンス(p.919)を参照してください。

- `map[string,string] getEnvironmentVariables();`

  システム環境変数の変更不可能なマップを返します。環境変数は、システム依存で外部の名前付き値です。Java関数`System.getenv()`に似ています。キーでは大文字と小文字が区別されます。呼出しの例は次のとおりです。

  ```
  string envPath = getEnvironmentVariables()["PATH"];
  ```

- `map[string,string] getJavaProperties();`

  この関数はJava VMシステム・プロパティのマップを返します。Java関数`System.getProperties()`に似ています。呼出しの例は次のとおりです。

  ```
  string operatingSystem = getJavaProperties()["os.name"];
  ```

- `string getParamValue(string paramName);`

  指定されたグラフ・パラメータの値を返します。引数は`${ }`文字を含まないグラフ・パラメータの名前(たとえば、`PROJECT_DIR`)です。返される値は解決されています。つまり、他のグラフ・パラメータに対する参照は含まれません。

  関数は、存在しないパラメータに対しては`null`を返します。

  呼出しの例は次のとおりです。

  ```
  string datainDir = getParamValue("DATAIN_DIR"); // will contain "./data-in"
  ```

- `map[string,string] getParamValues();`

  グラフ・パラメータおよびその値を返します。キーは`${ }`文字を含まないパラメータの名前(たとえば、`PROJECT_DIR`)です。値は解決されています。つまり、他のグラフ・パラメータに対する参照は含まれません。マップは変更できません。呼出しの例は次のとおりです。

  ```
  string datainDir = getParamValues()["DATAIN_DIR"]; // will contain "./data-in"
  ```

- `<any type> iif(boolean con, <any type> iftruevalue, <any type> iffalsevalue);`

  `iif(boolean, <any type>, <any type>)`関数は3つの引数を受け入れます。1つはブールで、2つは任意のデータ型です。2つの引数のデータ型および戻り型は同じです。

  この関数は最初の引数を取得し、最初の引数がtrueの場合は2番目の引数を返し、falseの場合は3番目の引数を返します。

- `boolean isnull(<any type> arg);`

  `isnull(<any type>)`関数は1つの引数を取得し、その引数がnull (true)であるかそうでないか(false)に応じてブール値を返します。引数のデータ型は任意です。

重要

stringデータ・フィールドのメタデータの「**Null value**」プロパティを空でない文字列に設定すると、isnull()関数はそのような文字列に適用された場合にtrueを返します。また、空のフィールドで適用された場合はfalseを返します。

たとえば、field1で「**Null value**」プロパティを"<null>"に設定した場合、isnull($0.field1)は、field1の値が"<null>"であるレコードではtrueを返し、その他のレコードでは、レコードが空である場合でもfalseを返します。

詳細は、Null value(p.164)を参照してください。

- <any type> **nvl**(<any type> *arg*, <any type> *default*);

  nvl(<any type>, <any type>)関数は、任意のデータ型の2つの引数を受け入れます。両方の引数が同じ型である必要があります。最初の引数がnullでない場合、関数は引数の値を返します。nullである場合、関数は2番目の引数で指定されたデフォルト値を返します。

- <any type> **nvl2**(<any type> *arg*, <any type> *arg_for_non_null*, <any type> *arg_for_null*);

  nvl2(<any type>, <any type>, <any type>)関数は、任意のデータ型の3つの引数を受け入れます。このデータ型は、すべての引数および戻り値で同じである必要があります。最初の引数がnullでない場合、関数は2番目の引数の値を返します。最初の引数がnullの場合、関数は3番目の引数の値を返します。

- void **printErr**(<any type> *message*);

  printErr(<any type>)関数は任意のデータ型の1つの引数を受け入れます。この引数を取得し、エラー出力にmessageを出力します。

  この関数は、printLocationがfalseに設定されたvoid printErr(<any type> arg, boolean printLocation)として機能します。

注意

この関数を**CloverETL Server**上で実行されているグラフで使用する場合、messageは**サーバー**のログ(たとえば、**Tomcat**のログ)に保存されます。かわりにprintLog()関数を使用してください。これは、グラフが**CloverETL Server**上で実行されている場合でも、エラー・メッセージをコンソールに記録します。

- void **printErr**(<any type> *message*, boolean *printLocation*);

  printErr(type, boolean)関数は2つの引数を受け入れます。1つ目は任意のデータ型で、2つ目はブールです。これらを取得し、messageおよびエラーの場所(2番目の引数がtrueの場合)を出力します。

注意

この関数を**CloverETL Server**上で実行されているグラフで使用する場合、messageは**サーバー**のログ(たとえば、**Tomcat**のログ)に保存されます。かわりにprintLog()関数を使用してください。これは、グラフが**CloverETL Server**上で実行されている場合でも、エラー・メッセージをコンソールに記録します。

- void **printLog**(level *loglevel*, <any type> *message*);

  printLog(level, <any type>)関数は2つの引数を受け入れます。最初の引数は、任意のデータ型の、2番目の引数として指定されたmessageのログ・レベルです。最初の引数は、debug、info、warn、error、fatalのいずれかです。ログ・レベルは定数として指定する必要があります。エッジを介して取得することも、変数として設定することもできません。関数は引数を取得し、messageをログ出力に送信します。

注意

この関数は、特に**CloverETL Server**上で実行されるグラフで、**サーバー**のログ(たとえば、**Tomcat**のログ)にエラー・メッセージを記録するprintErr()関数のかわりに使用する必要があります。printErr()とは異なり、printLog()はグラフが**CloverETL Server**上で実行されている場合でも、エラー・メッセージをコンソールに記録します。

- `void` **`raiseError`**`(string` *`message`*`);`

  raiseError(string)関数は1つの文字列引数を取得し、引数で指定されたmessageとともにエラーをスローします。

- `string` **`resolveParams`**`(string` *`text`*`);`

  string resolveParams(string, false, true)として動作します。関連する関数(p.953)を参照してください。

- `string` **`resolveParams`**`(string` *`parameter`*`, boolean` *`resolveSpecialChars`*`, boolean` *`resolveCtl`*`);`

  この関数は文字列を取得し、その文字列内のすべてのグラフ・パラメータを対応する値で置換します。したがって、既存のグラフ・パラメータを参照するパターン${<*PARAMETER_NAME*>}のすべての出現がパラメータの値で置換されます。このことは、両方のブール引数の値に関係なく常に実行されます。この関数はシステム・プロパティ(たとえば、PATHやJAVA_HOME)も同様の方法で解決できます。

  さらに、その他に何が解決されるかを制御できます。

  - resolveSpecialChars: 特殊文字(たとえば、\n、\u)を解決します。例: コンテンツが\u002Aのasteriskという名前のメタデータ・フィールドがあるとします。次の変換により

    ```
    resolveParams(asterisk, true, false);
    ```

    *文字が生成されます。

  - resolveCtl: CTLコードを解決します。逆向きのカンマ内のCTLコードはCTL1(p.828)として解釈されます。例: コンテンツが1 plus 2 equals `to_string(1+2)`のcodeという名前のメタデータ・フィールドがあるとします。次の変換により

    ```
    resolveParams(code , false, true);
    ```

    1 plus 2 equals 3が生成されます。

注意

次のように、parameterをメタデータ・フィールドとしてではなく直接渡すとします。

```
string special = "\u002A"; // Unicode for asterisk - *
resolveParams(special, true, false); // this line is not needed
printErr(special);
```

これは自動的に解決されます。前述のコードは、2行目を省略した場合でも、アスタリスクを出力します。これは、パラメータを囲む引用符を処理する際に解決がトリガーされるためです。

- `void` **`sleep`**`(long` *`duration`*`);`

  この関数は指定されたミリ秒数だけ実行を停止します。

- `string` **`toAbsolutePath`**`(string` *`path`*`);`

  この関数は指定されたパスを、同じファイルに対するOS依存の絶対パスに変換します。入力はパスまたはURLのいずれかです。入力パスが相対パスである場合、実行中のグラフのコンテキストURLに対してパスが解決されます。

  サーバー上で実行されている場合、関数はサンドボックスURL(p.301)も処理できます。ただし、ファイルが現行のサーバー・ノードでローカルに使用可能である場合、サンドボックスURLは絶対パスにのみ変換できます。

  変換が失敗した場合は`null`を返します。

注意

返されるパスでは、Microsoft Windowsシステム上であっても、常にスラッシュをディレクトリ・セパレータとして使用します。

パスにバックスラッシュを含める必要がある場合は、次のように`translate()`関数を使用してください。

```
string absolutePath = toAbsolutePath(path).translate('/', '\\');
```

参照表関数

グラフでは参照表も使用します。これらをCTLで使用するには、参照表の名前を指定し、lookup()関数で引数として指定する必要があります。

警告

CTLテンプレートのinit()、preExecute()またはpostExecute()関数では、次に示す関数は使用できません。

ここで、後述の関数でのキーは、(セミコロンではなく)カンマで区切られたフィールド名の値のシーケンスとなります。したがって、キーの形式はkeyValuePart1,keyValuePart2,...,keyValuePartNのようになります。

次のオプションを参照してください。

- lookup(<lookup name>).get(keyValue)[.<field name>|.*]

  この関数は、キー値がget(keyValue)関数で指定された値に等しい最初のレコードを検索します。

  これは参照表のレコードを返します。これを(同じメタデータを持つ) CTL2の他のレコードにマップできます。フィールドの値を取得するには、式に.<field name>の部分を追加するか、.*を追加してすべてのフィールドの値を取得します。

- lookup(<lookup name>).count(keyValue)

  キー値がkeyValueに等しいレコードの数を取得するには、前述の構文を使用します。

- lookup(<lookup name>).next()[.<field name>|.*]

  lookup().count()関数を使用して参照表内の重複レコードの数を取得し、lookup().get()関数を使用して指定したキー値を持つ最初のレコードを取得したら、同じキー値を持つ参照表のすべてのレコードを(1つずつ)処理できます。

  ループ内でここに示す構文を使用し、参照表のすべてのレコードを処理する必要があります。各レコードが1つのループ・ステップ内で処理されます。

  前述の構文は参照表のレコードを返します。これを(同じメタデータを持つ) CTL2の他のレコードにマップできます。フィールドの値を取得するには、式に.<field name>の部分を追加するか、.*を追加してすべてのフィールドの値を取得します。

- lookup(<lookup name>).put(<record>)

  put()関数は、選択された参照表で引数として渡されるレコードを保存します。これは、操作が成功したかどうかを示すブール結果を返します。

  渡されるレコードのメタデータは参照表のメタデータと一致する必要があります。

  操作はすべてのタイプの参照表でサポートされているわけではありません(たとえば、**データベース参照表**ではサポートされません)。また、具体的なセマンティックは実装固有です(特に、保存されたレコードは同じフェーズですぐに読取り可能になるとはかぎりません)。

例66.9. 参照表関数の使用方法

```
//#CTL2

// record with the same metadata as those of lookup table
recordName1 myRecord;

// variable for storing number of duplicates
integer count;

// Transforms input record into output record.
function integer transform() {

    // if lookup table contains duplicate records,
    // their number is returned by the following expression
    // and assigned to the count variable
    count = lookup(simpleLookup0).count($0.Field2);

    // getting the first record whose key value equals to $0.Field2
    myRecord = lookup(simpleLookup0).get($0.Field2);

    // loop for searching the last record in lookup table
    while ((count-1) > 0) {

    // searching the next record with the key specified above
    myRecord = lookup(simpleLookup0).next();

    // incrementing counter
    count--;
    }

    // mapping to the output

    // last record from lookup table
    $0.Field1 = myRecord.Field1;
    $0.Field2 = myRecord.Field2;

    // corresponding record from the edge
    $0.Field3 = $0.Field1;
    $0.Field4 = $0.Field2;
    return 0;
}
```

警告

前述の例では、すべての参照表関数の使用方法を示しました。ただし、参照表では別の構文を使用することをお薦めします。

その理由は、次のCTL2の式にあります。

```
lookup(Lookup0).count($0.Field2);
```

これは、多数のレコードが含まれている可能性がある参照表全体でレコードを検索します。

この構文のかわりに、次のループを使用できます。

```
myRecord = lookup(<name of lookup table>).get(<key value>);
while(myRecord != null) {
  process(myRecord);
```

```
  myRecord = lookup(<name of lookup table>).next();
}
```

特にDB参照表では、指定したキー値を持つレコードの実際の数ではなく、-1を返す可能性があります(**最大キャッシュ・サイズ**をゼロ以外の値に設定しなかった場合)。

CTL1の`lookup_found(<lookup table ID>)`関数も一般的にはお薦めしません。

重要

DB参照表はコンパイル・モードでは使用できません。(次のヘッダーで始まるコード: `//#CTL2:COMPILE`)

CTL2からDB参照表にアクセスできるようにするには、(`//#CTL2`のヘッダーが付いた)インタプリタ・モードに切り替える必要があります。

シーケンス関数

グラフではシーケンスも使用します。これらをCTLで使用するには、シーケンスの名前を指定し、`sequence()`関数でそれを引数として指定します。

警告

CTLテンプレートの`init()`、`preExecute()`または`postExecute()`関数では、次に示す関数は使用できません。

シーケンスに対して行う操作に応じた、3つのオプションがあります。現在のシーケンス番号を取得、次のシーケンス番号を取得またはシーケンス番号を初期番号値にリセットできます。

次のオプションを参照してください。

```
sequence(<sequence name>).current()

sequence(<sequence name>).next()

sequence(<sequence name>).reset()
```

これらの式は整数値を返しますが、long値または文字列値を取得する必要がある場合もあります。この場合、次のいずれかの方法を使用します。

```
sequence(<sequence name>,long).current()

sequence(<sequence name>,long).next()

sequence(<sequence name>,string).current()

sequence(<sequence name>,string).next()
```

カスタムCTL関数

準備済のCTL関数の他に、独自のCTL関数を作成できます。このことを行うには、カスタムCTL関数を定義する独自のコードを記述し、プラグインを指定する必要があります。

各カスタムCTL関数ライブラリは、次のものから派生/継承する必要があります。

`org.jetel.interpreter.extensions.TLFunctionLibrary`クラス。

各カスタムCTL関数は、次のものから派生/継承する必要があります。

`org.jetel.interpreter.extensions.TLFunctionPrototype`クラス。

これらのクラスには、定義済の標準操作およびカスタム関数を使用するために定義する必要があるいくつかの抽象メソッドがあります。カスタム関数コード内では、既存のコンテキストを使用するかカスタム・コンテキストを定義する必要があります。コンテキストは、繰り返し、つまり複数のレコードに対して関数を実行する場合にオブジェクトを保存するために使用します。

カスタム関数コードとともに、カスタム関数プラグインも定義する必要があります。ライブラリおよびプラグインの両方が**CloverETL**で使用されます。詳細は、次のwikiページを参照してください。
wiki.cloveretl.org/doku.php?id=function_building

CTL2付録: 各国語固有キャラクタのリスト

editDistance (string, string, integer, string, integer)(p.936)などのいくつかの関数では、特別な各国語キャラクタを処理する必要があります。これらは、定義された比較の強度で項目をソートする場合に特に重要です。

次のリストでは、最初にロケールを示し、次に各文字の各国語固有の派生のリストを示します。定義する比較の強度に応じて、これらは同一または異なる文字として扱われます。

表66.2. 各国語キャラクタ

ロケール	各国語キャラクタ
CA -カタロニア語	"a=à=A=À", "e=è=é=E=È=É", "i=í=ï=I=Í=Ï", "o=ò=ó=O=Ò=Ó", "u=ú=ü=U=Ú=Ü", "c=ç=C=Ç"
CZ -チェコ語	"a=á=A=Á", "c=č=C=Č", "d=ď=D=Ď", "e=é=ě=E=É=Ě", "i=í=I=Í", "n=ň=N=Ň", "o=ó=O=Ó", "r=ř=R=Ř", "s=š=S=Š", "t=ť=T=Ť", "u=ů=ú=U=Ů=Ú", "y=ý=Y=Ý", "z=ž=Z=Ž"
DA -デンマーク語およびノルウェー語	"a=æ=å=A=Æ=Å", "o=ø=O=Ø"
DE -ドイツ語	"a=ä=A=Ä", "o=ö=O=Ö", "u=ü=U=Ü"
ES -スペイン語	"a=á=A=Á", "e=é=E=É", "i=í=I=Í", "o=ó=O=Ó", "u=ú=ü=U=Ú=Ü", "n=ñ=N=Ñ"
ET -エストニア語	"a=ä=A=Ä", "o=ö=õ=O=Ö=Õ", "u=ü=U=Ü", "s=š=S=Š", "z=ž=Z=Ž"
FI -フィンランド語	"a=ä=A=Ä", "o=ö=O=Ö"
FR -フランス語	"a=à=â=A=À=Â", "e=è=é=ê=ë=E=È=É=Ê=Ë", "i=ï=I=Ï", "o=ô=O=Ô", "u=ù=û=ü=U=Ù=Û=Ü", "c=ç=C=Ç"

ロケール	各国語キャラクタ
HR -クロアチア語	"c=ć=č=C=Ć=Č", "d=đ=D=Đ", "s=š=S=Š", "z=ž=Z=Ž"
HU -ハンガリー語	"a=á=A=Á", "e=é=E=É", "i=í=I=Í", "o=ó=ö=ő=O=Ó=Ö=Ő", "u=ú=ü=ű=U=Ú=Ü=Ű"
IS -アイスランド語	"a=á=æ=A=Á=Æ", "e=é=E=É", "i=í=I=Í", "o=ó=ö=O=Ó=Ö", "u=ú=U=Ú", "y=ý=Y=Ý", "d=ð=D=Đ"
IT -イタリア語	"a=à=A=À", "e=é=è=E=É=È", "i=ì=I=Ì", "o=ò=O=Ò", "u=ù=U=Ù"
LV -ラトビア語	"a=ā=A=Ā", "e=ē=E=Ē", "i=ī=I=Ī", "u=ū=U=Ū", "c=č=C=Č", "g=ģ=G=Ģ", "k=ķ=K=Ķ", "l=ļ=L=Ļ", "n=ņ=N=Ņ", "s=š=S=Š", "z=ž=Z=Ž"
PL -ポーランド語	"a=ą=A=Ą", "c=ć=C=Ć", "e=ę=E=Ę", "l=ł=L=Ł", "n=ń=N=Ń", "o=ó=O=Ó", "s=ś=S=Ś", "z=ż=ź=Z=Ż=Ź"
PT -ポルトガル語	"a=ã=á=à=â=A=Ã=Á=À=Â", "e=é=ê=E=É=Ê", "i=í=I=Í", "o=õ=ó=ô=O=Õ=Ó=Ô", "u=ú=U=Ú", "c=ç=C=Ç"
RO -ルーマニア語	"a=ă=â=A=Ă=Â", "i=î=I=Î", "s=ş=S=Ş", "t=ţ=T=Ţ"
RU -ロシア語	"Е=е=Ё=ё=Е́=е́", "И=и=Й=й=И'=и́", "О=о=О́=о́", "А=а=А́=а́", "Ы=ы=Ы'=ы'", "Я=я=Я'=я́", "Ю=ю=Ю'=ю'", "У=у=У'=у́", "Э=э=Э'=э́"

ロケール	各国語キャラクタ
SK -スロバキア語	"a=á=ä=A=Á=Ä", "c=č=C=Č", "d=ď=D=Ď", "e=é=E=É", "i=í=I=Í", "l=Í=I=L=Ĺ=L", "n=ň=N=Ň", "o=ó=ô=O=Ó=Ô", "r=ŕ=R=Ŕ", "s=š=S=Š", "t=ť=T=Ť", "u=ú=U=Ú", "y=ý=Y=Ý", "z=ž=Z=Ž"
SL -スロベニア語	"c=č=C=Č", "s=š=S=Š", "z=ž=Z=Ž"
SQ -アルバニア語	"e=ë=E=Ë", "c=ç=C=Ç"
SV -スウェーデン語	"a=ä=å=A=Ä=Å", "o=ö=O=Ö"

第67章 正規表現

正規表現は、単一の式で一連の文字列を指定するために使用される形式です。正規表現の実装はJava標準ライブラリ由来であるため、式の構文はJavaの場合と同じです。

ヘッダー1

例67.1. 正規表現の例

```
[p-s]{5}
```

- 文字列の長さが5文字である必要があり、p、q、rおよびsの文字のみを含めることができることを意味します。

```
[^a-d].*
```

- この正規表現の例は、次の理由により、a、b、c、d以外の文字で始まる任意の文字列に一致します。
  - ^記号は例外を意味します。
  - a-dは、aからdまでの文字を意味します。
  - これらの文字の後にはゼロ文字以上(*)の他の文字を置くことができます。
  - ドットは任意の文字を表します。

正規表現の使用方法の詳細は、Javaのドキュメントで`java.util.regex.Pattern`を参照してください。

正規表現の意味は埋込みフラグ表現を使用して変更できます。この表現には次のものが含まれます。

`(?i) - Pattern.CASE_INSENSITIVE`	大文字小文字を区別しない一致を有効にします。
`(?s) - Pattern.DOTALL`	dotallモードでは、ドット(.)が行の終了文字を含む任意の文字に一致します。
`(?m) - Pattern.MULTILINE`	複数行モードでは、^および$を使用して、それぞれ行の先頭および末尾を示すことができます(これには、式全体の先頭および末尾も含まれます)。

詳細およびその他のフラグの説明は、http://docs.oracle.com/javase/tutorial/essential/regex/pattern.htmlを参照してください。

図一覧

表一覧

例一覧