サーバーアプリケーションの移行および再配備

Sun ONE Application Server

Version 7

817-0603-10
2002 年 10 月

目 次

本書について

この『サーバーアプリケーションの移行および再配備』では、以前のバージョンの Sun ONE Application Server (従来の名称は「iPlanet Application Server」) から Sun ONE Application Server 7 への J2EE アプリケーション移行方法を説明します。

さらに、NetDynamics と Netscape Application Server (NAS) のアプリケーションを Sun ONE Application Server 7 へ移行する方法も説明します。

このマニュアルは、移行の詳細を知る必要があるシステム管理者、ネットワーク管理者、アプリケーションサーバー管理者、および Web 開発者を対象にしています。

前提事項

このマニュアルでは、次の項目について熟知していることを前提とします。

- HTML
- アプリケーションサーバー
- クライアント / サーバープログラミングモデル
- インターネットおよび WWW (World Wide Web)
- Windows 2000 または Solaris [TM] オペレーティングシステム、あるいはその両方
- Java プログラミング
- EJB および JSP (Java Server Pages) の仕様に定義されている Java API
- Java Database Connectivity (JDBC)
- SQL などの構造化データベースクエリ言語
- リレーショナルデータベースの概念
- デバッグ、ソースコード制御を含むソフトウェア開発プロセス

マニュアルの構成

このマニュアルの構成は次のとおりです。

- Sun ONE Application Server 7 について - Sun ONE Application Server 7 のアーキテクチャ、J2EE 標準間の変更点、およびこのバージョンと以前のバージョンの Sun ONE Application Server の導入アプリケーションの違いについての説明
- 移行と再配備 - 移行の必要があるアプリケーションコンポーネントとその理由、および移行したアプリケーションの再配備プロセスの説明
- 移行に関する注意事項および手法 - 競合するプラットフォーム、および以前のバージョンの Sun ONE Application Server からアプリケーションを移行する際の注意事項と、その方法についての説明。また、移行プロセス全体の説明に使用するサンプルアプリケーションを掲載
- 移行の自動化 - 競合するプラットフォーム、および以前のバージョンの Sun ONE Application Server からアプリケーションを移行する際に使用できる自動化ツールについての説明
- 移行したアプリケーションの再配備 - 移行したアプリケーションがどのように Sun ONE Application Server に再配備されるかについての説明

マニュアルの表記規則

ファイルとディレクトリのパスは、Windows の形式で表記されます (ディレクトリ名を円記号で区切って表記)。UNIX バージョンでは、ディレクトリパスについては Windows と同じですが、ディレクトリの区切りには円記号ではなくスラッシュが使われます。

このマニュアルでは次のような URL 形式を使います。
`http://server.domain/path/file.html`

- server はアプリケーションを実行するサーバー名
- domain はインターネットのドメイン名
- path は、サーバー上のディレクトリ構造
- file は個々のファイル名

次の表は、Sun ONE のマニュアルで採用しているフォントの規約です。

表 1 フォントの規約

書体	意味	例
モノスペース	ファイル名、ディレクトリ、サンプルコード、コードの一覧表示、および HTML タグ	`Hello.html` ファイルを開きます。 `<HEAD1>` は、最上位の見出しを作成します。
イタリック	変数、コードのプレースホルダ、およびリテラルに使われる語句	名前のフィールドに「*Login*」と入力します。
太字	テキストに初めて登場した用語	**テンプレート**は、ページのアウトラインです。

第 1 章

Sun ONE Application Server 7 について

この章では、Sun ONE Application Server 7 のアーキテクチャ、およびサーバー環境の重要な J2EE コンポーネントについて説明します。さらに、Sun ONE Application Server 7 の環境とそれ以前の Sun ONE Application Server 環境の違いについても説明します。

次の項目があります。

- Sun ONE Application Server 7 のアーキテクチャ
- J2EE コンポーネント標準
- 開発環境
- 管理ツール
- データベース接続
- J2EE アプリケーションコンポーネントと移行
- 移行と再配備

Sun ONE Application Server 7 のアーキテクチャ

アプリケーションサーバーは、クライアントがバックエンドソースに接続するための基盤を提供し、アプリケーションロジックを実行し、実行結果をクライアントに戻します。アプリケーションサーバーは 3 層コンピューティングモデルの中間層を使用します。

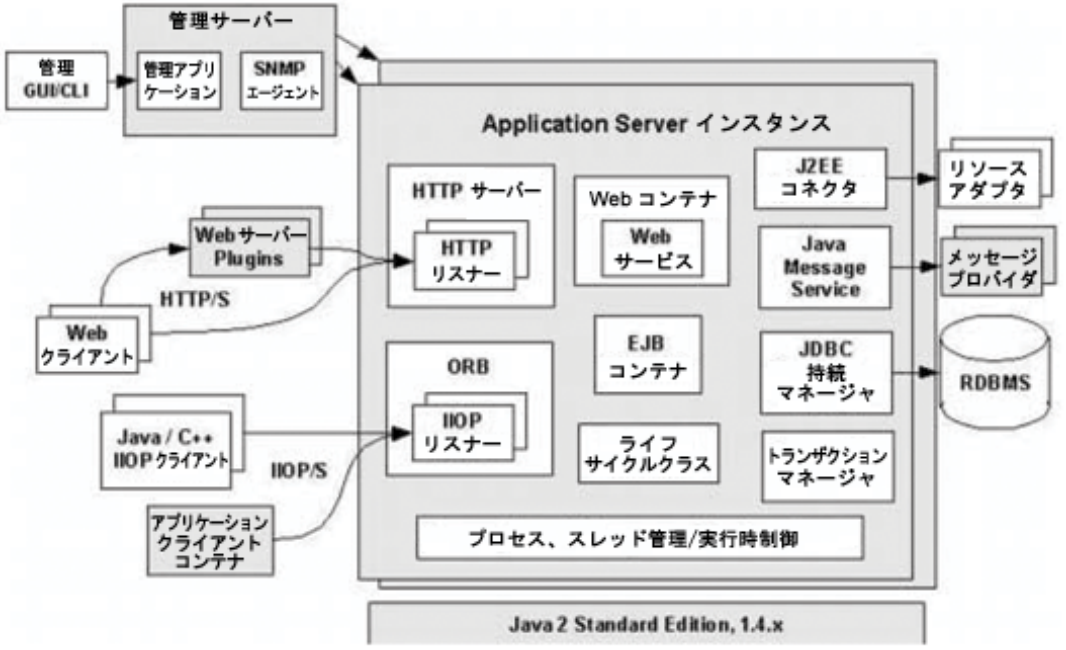
Sun ONE Application Server 7 は Java アプリケーションサーバーであり、Java 2 Enterprise Edition (J2EE™) 仕様に完全に準拠しています。J2EE では、完全でしかも安全な基盤を提供し、セキュリティ、開発、配備、コードの再使用、および移植についての標準を豊富に用意しています。これによって移植が容易な、ベンダーに依存しないアプリケーション開発ができます。

Sun ONE Application Server 7 は、多種多様なサーバー、クライアント、およびデバイスを対象とした電子商取引アプリケーションサービスを開発、配備、管理するための堅牢な J2EE プラットフォームです。

Sun ONE Application Server 7 は J2EE 1.3 に準拠したアプリケーションサーバーです。

このアーキテクチャの主な目的は、あらゆる方向の拡張性、および高い可用性、信頼性、パフォーマンス、および標準への準拠です。Sun ONE Application Server 7 はまた、第一世代の Sun ONE アプリケーションサーバー製品から大きく進歩したアーキテクチャを持っています。既存の強力な Sun ONE 製品およびテクノロジと J2EE 1.3 標準を組み合わせた Sun ONE Application Server 7 のアーキテクチャは、様々なテクノロジの基盤として最適なものです。

Sun ONE Application Server 7 のアーキテクチャ



Sun ONE Application Server のアーキテクチャを図で示したものが Sun ONE Application Server 7 のアーキテクチャです。Sun ONE Application Server のコンポーネント、サブシステム、アクセスパス、および外部エンティティとコアサーバーのインタフェースを図解しています。

Sun ONE Application Server 7 のアーキテクチャは高度にコンポーネント化されており、非常に管理しやすいものになっています。J2EE 仕様で要求されるすべてのサービスは、明確に定義された標準インタフェースとして提供され、アプリケーションから呼び出して使用できます。

Sun ONE Application Server 7 で新たに提供されている Web ユーザーインタフェースで、簡単にリモート管理を行うことができます。実際は 1 つの管理サーバーで複数のサーバーを管理することができる設計になっています。このバージョンでは、新しい管理されるサーバーインスタンスの作成作業が大幅に単純化されます。

Sun ONE Application Server の以前のバージョンは Type 2 JDBC ドライバをサポートしていましたが、このバージョンではサポートしていません。この結果、このプラットフォームの JDBC リソース管理に対するアプローチがより標準的なものになっています。

Sun ONE Application Server のオペレーションで JDK 1.4 を使用することにより、このバージョンで拡張された機能を利用できるようになっています。

一般的な J2EE アプリケーションは *n* 層のシステムで構成され、クライアントは処理された情報を Web サーバーまたはアプリケーションサーバーから取得します。サーバーは RDBMS や ERP などの企業規模のシステムが持つ情報にアクセスし、ビジネスロジックで処理し、処理後の情報を適切な形式でクライアントに配信します。これらのレイヤはそれぞれクライアントレイヤ (Web ブラウザまたはリッチ Java クライアント)、中間レイヤ (Web サーバーおよびアプリケーションサーバー)、およびバックエンドレイヤまたはデータレイヤ (データベースなどの企業規模のシステム) に対応します。

Sun ONE Application Server の J2EE アプリケーションモデルを使用すれば、J2EE コンポーネントが低レベルの詳細に関するすべての処理を受け持つため、開発者はビジネスロジックの作成に集中することができます。したがって、アプリケーションおよびサービスを容易に拡張して短期間で配備し、競合相手の変化に素早く対応することができます。J2EE プラットフォームでオープンスタンダードアーキテクチャを提供している Sun ONE Application Server は、拡張性と高い可用性を持ち、安全で信頼性の高い多層サービスの開発における複雑さと費用の問題を解決します。

J2EE コンポーネント標準

Sun ONE Application Server 7 は J2EE v1.3 に準拠したサーバーです。Java コミュニティによって開発された、サーブレット、Java Server Pages、および Enterprise JavaBeans (EJB) 用のコンポーネント標準をベースにしています。

Sun ONE Application Server 7 以前の Sun ONE Application Server 6.0/6.5 は、J2EE v1.2 に準拠したサーバーです。2 つの J2EE バージョンの間には、J2EE アプリケーションコンポーネント API に関して大きな違いがあります。

J2EE v1.3 準拠の Sun ONE Application Server 7 と、J2EE v1.2 準拠の Sun ONE Application Server 6.0/6.5 で使用されるコンポーネント API 間の相違点を次の表に示します。

J2EE コンポーネント用 API の Application Server 間でのバージョン比較

コンポーネント API	Sun ONE Application Server 6.0/6.5	Sun ONE Application Server 7
JDK	1.2.2	1.4
サーブレット	2.2	2.3
JSP	1.1	1.2
JDBC	2.0	2.0
EJB	1.1	2.0
JNDI	1.2	1.2
JMS	1.0	2.0
JTA	1.0	1.01

さらに、この 2 つの製品では、XML 標準および Web サービスと関連のあるテクノロジを多数サポートしています。これらは J2EE 仕様の一部ではないものの、企業アプリケーションでますます広く使用されるようになっています。これらの標準を次の表に示します。

Application Server がサポートするその他のテクノロジ

テクノロジ	Sun ONE Application Server 6.0/6.5	Sun ONE Application Server 7
XML ドキュメント処理 (API および XML パーサ)	JAXP 1.0、Apache Xerces	JAXP 1.1
Web サービス用 SOAP/Java サポート	SOAP 1.1 (IBM SOAP4J フレームワーク)	Apache SOAP 2.2、JAX-RPC 1.0、JAXM 1.1、JAXR 1.0

開発環境

この節では Sun ONE Application Server 6.0/6.5 と Sun ONE Application Server 7 の開発環境の違いを説明します。次の項目があります。

- Sun ONE Application Server 6.0/6.5 の開発環境
- Sun ONE Application Server 7 の開発環境

Sun ONE Application Server 6.0/6.5 の開発環境

Sun ONE Application Server 6.0/6.5 では Sun ONE Studio for Java の評価バージョンを提供しています。これは特にこのバージョンの Sun ONE Application Server のアプリケーション開発向けに調整されています。

これは NetBeans プラットフォームをベースにした完全な Java 開発環境です。この IDE は、Java アプリケーションおよび EJB コンポーネントの設計および開発のための、非常に広範囲の機能を提供します。また、プラグインによって Sun ONE Application Server と統合し、アプリケーションの様々な J2EE コンポーネントのアセンブリ、配備およびデバックに使用できます。Windows および Solaris のどちらの環境でも利用できます。

市場に出回っているサードパーティベンダー提供のソリューションの中では、最近リリースされた Borland JBuilder 6 Enterprise が非常に完成度が高く、利用範囲の広い製品です。Windows、Solaris、Linux、および MacOS X など複数のプラットフォームで動作するという利点もあります。JBuilder は、サーブレット、JSP ページ、EJB コンポーネント、グラフィックアプリケーションなどの Java の開発機能だけではなく、UML 設計、単体テスト、共同開発、および XML 開発の機能も提供しています。さらに、JBuilder を Sun ONE Application Server を含むメインストリームのアプリケーションサーバーと完全に統合し、Web アプリケーションおよび EJB コンポーネントのアセンブリ、配備およびデバッグに使用できます。

Sun ONE Application Server 7 の開発環境

Sun ONE Application Server 7 をうまく使いこなせるかどうかは、完全に統合された開発環境を利用できるかどうかで決まります。Sun ONE Studio for Java Enterprise Edition 4 は Sun ONE アプリケーション開発のための重要なツールです。

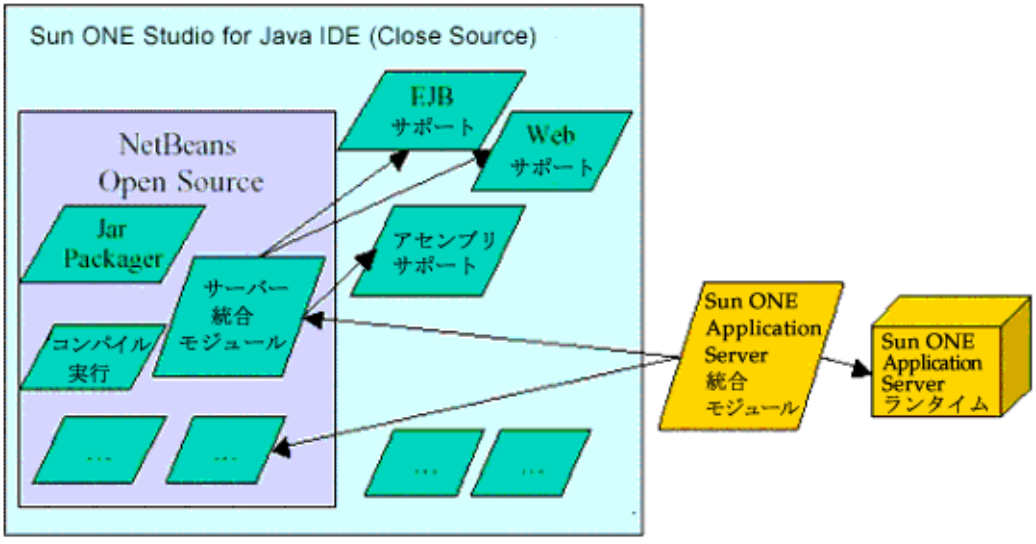
Sun ONE Studio for Java 4 は Sun ONE Application Server と共に提供されます。

Sun ONE Studio for Java Enterprise Edition 4 の中心的な機能を次に示します。

- EJB を短期間で容易に構築する機能
- EJB および配備用パッケージアプリケーションからアプリケーションをアセンブルする機能
- 配備のためのアプリケーションサーバーの統合
- Web サービスの開発およびパブリッシュ機能
- Java enterprise service presentation toolkit 用の Sun ONE studio
- Sun ONE Application Server 7 との統合機能

Sun ONE Studio Enterprise Edition と Sun ONE Application Server 7 の統合で図解されているように、Sun ONE Application Server 7 統合モジュールは、Sun ONE Studio Close Source で実装される NetBeans Open Source モジュールを使用しています。

Sun ONE Studio Enterprise Edition と Sun ONE Application Server 7 の統合



管理ツール

この節では Sun ONE Application Server 6.0/6.5 と Sun ONE Application Server 7 の管理ツールの違いについて説明します。次の項目があります。

- Sun ONE Application Server 6.0 の管理ツール
- Sun ONE Application Server 6.5 の管理ツール
- Sun ONE Application Server 7 の管理ツール

Sun ONE Application Server 6.0 の管理ツール

Sun ONE Application Server 6.0 は、サーバー管理のあらゆる機能をカバーする、完全なグラフィカル管理ツールを提供しています。

- Sun ONE Console - 管理ツールのメインコントロールパネル。Sun ONE console で Administration Server Console、Directory Server、および Administration Tool へのすばやいアクセスが可能
- Administration Server Console - イベントログオプションの設定と SSL セキュリティ証明書の作成に使用
- Sun ONE Directory Server Console - Sun ONE Directory Server の管理に使用。Directory Server は、ユーザーおよび組織単位の管理に関する情報を持つユーザーディレクトリ、およびサーバー設定情報を持つ設定ディレクトリの、2 つの主要な情報ディレクトリツリーを管理
- Sun ONE Administration Tool - Sun ONE Application Server 6.0 のインスタンスを、配備済みのアプリケーションと共に管理するために使用。JDBC ドライバとデータソースの設定も行う
- Sun ONE Registry Editor (*kregedit*) - Windows のレジストリエディタ (*regedit*) と似たグラフィカルなツール。特定のレジストリに格納された、Sun ONE Application Server 固有のパラメータ調整に使用

Sun ONE Application Server 6.5 の管理ツール

Sun ONE Application Server 6.5 は、統合されている Administration Tool、Sun ONE registry editor およびコマンド行ツールで管理できます。以下はその説明です。

- Sun ONE Application Server Administration Tool - グラフィカルユーザーインタフェースを持つ、スタンドアロンの Java アプリケーション。管理アプリケーションコンポーネントと共に使用し、Sun ONE Application Server のインスタンス管理が可能
- **コマンド行ツール** - Windows のコマンドプロンプト、および Solaris のシェルプロンプトから実行するツール。基本設定からアプリケーションの配備まで、様々なタスクを実行できる。コマンドプロンプトで `[command] -help` と入力することで詳しい説明を表示する。ほとんどのツールは Sun ONE Application Server Administration Tool および Sun ONE Application Server Deployment Tool と統合されており、簡単に使用することができる
- Sun ONE Registry Editor (*kregedit*) - Windows のレジストリエディタ (regedit) と似たスタンドアロンの GUI ツール。Sun ONE Application Server のレジストリ情報の表示および編集が可能

Sun ONE Application Server 7 の管理ツール

Sun ONE Application Server 7 の管理サーバーはサーバーの特別なインスタンスです。管理インタフェースとしての機能を持ち、すべてのサーバーインスタンスに共通のグローバル設定を管理します。Web ベースのサーバーであり、Sun ONE Application Server 設定用のフォームを持ちます。

アプリケーションサーバーを管理するためのグラフィカルツールです。エラーやアクセスログの表示、サーバーの稼動状況の監視、仮想サーバーの作成と編集、構成内容の変更、サーバーインスタンスの起動および停止などを行います。

Sun ONE Application Server をインストールする際に、管理サーバーのポート番号を選択していない場合は、デフォルトのポート番号 4848 を使用しています。管理インタフェースにアクセスするには、Web ブラウザで次の URL を指定します。

http://「ホスト名」*:*「ポート番号」*/admin*

設定済みのユーザー名とパスワードの入力が要求されます。入力して「了解」ボタンをクリックすると、図「管理インタフェースのホームページ」のような管理インタフェースのホームページが表示されます。

左側のペインには、Sun ONE Application Server で構成できるすべての項目がツリー表示されます。管理インタフェースを使用するには、この左側のペインのどれか 1 つの項目をクリックします。右側のペインに、クリックした項目に対応するページが表示されます。

管理インタフェースのどのページのヘルプも、管理インタフェースの最上部のバナーにある「ヘルプ」ボタンをクリックして表示できます。オンラインヘルプには、ユーザーがアクセスしているページの使用方法、およびそのページに表示されているフィールドへの入力内容などの説明があります。

管理インタフェースのホームページ

Sun ONE Application Server 7 はコマンド行インタフェースを装備しています。ユーティリティおよびコマンドを使用して、管理インタフェースと同じタスクを実行できます。コマンドは、シェルのコマンドプロンプトまたは他のスクリプトやプログラムから呼び出すことができます。これらのコマンドを使用して、繰り返し行う管理タスクを自動化できます。

Sun ONE Application Server 7 のコマンド行ユーティリティは *asadmin* と呼ばれ、Windows のコマンドプロンプト、および Solaris のシェルプロンプトで実行できます。*asadmin* ユーティリティは、管理タスクを実行するためのコマンドセットを備えています。これらのコマンドを使用して、基本設定からアプリケーション配備までのすべてのタスクを、管理インタフェースと同じように実行できます。コマンドの説明が必要な場合は、*help* を *asadmin* ユーティリティコマンドの後に入力します。

asadmin はシングルモードまたはマルチモードで実行できます。シングルモードでは、コマンドプロンプトからコマンドを 1 つずつ実行します。マルチモードでは、環境レベルの情報を繰り返し入力することなく、複数のコマンドを実行できます。

データベース接続

この節では、Sun ONE Application Server 6.0/6.5、および Sun ONE Application Server 7 に含まれるドライバのタイプ、および各タイプのドライバでサポートされるデータベースについて説明します。

次の項目があります。

- 「Sun ONE Application Server 6.0 がサポートするデータベース」
- 「Sun ONE Application Server 6.5 がサポートするデータベース」
- 「Sun ONE Application Server 7 がサポートするデータベース」

Sun ONE Application Server 6.0 がサポートするデータベース

Sun ONE Application Server 6.0 では、Type 2 JDBC ドライバを提供しています。次に示す主要なデータベースバックエンドに接続できますが、各 DBMS にアクセスするためにはネイティブクライアントライブラリをインストールする必要があります。

- DB2 6.1、7.1
- Informix 7.3、9.1.4、9.2
- Oracle 8.0.5、8i、9i
- Sybase 11.9.2、12
- Microsoft SQL Server 7
- Pointbase 3.5

サードパーティが提供する Type 4 JDBC ドライバは、Sun ONE Application Server 管理ツールまたは独自に提供されるユーティリティで宣言すれば使用できるようになります。Solaris では db_setup.sh、Windows では jdbcsetup をこの目的で使用します。

JDBC データソースおよび接続プールのプロパティを、Sun ONE Application Server 管理インタフェース 、または *iasdeploy* コマンド行ユーティリティを使用して追加し、設定できます。iasdeploy には、定義するデータソースのプロパティを定義している XML ファイルを渡します。

Sun ONE Application Server 6.5 がサポートするデータベース

Sun ONE Application Server 6.5 では、JDBC Type 2 ドライバを提供しており、次に示すデータベースをサポートしています。

- DB2 5.1 および 6.1。クライアントのバージョンは 7.1
- Informix 7.3、9.1.4、9.2。クライアントのバージョンは SDK 2.40
- Oracle 8i、9i
- Sybase 12
- Microsoft SQL Server 7

Solaris でのネイティブ JDBC ドライバの設定は、専用のユーティリティ *db_setup.sh* を使用して行います。Windows では、データベースクライアントライブラリがマシンにインストールされていれば、ネイティブドライバはインストール時に自動設定されます。Sun ONE Application Server のインストール後にデータベースクライアントライブラリをインストールする場合は、Sun ONE Application Server を再起動することでネイティブドライバが自動設定されます。

サードパーティ提供の Type 4 JDBC ドライバは、Sun ONE Application Server 管理ツールで宣言すれば使用できるようになります。Solaris および Windows のどちらでも手順は同じです。

Sun ONE Application Server では、Sun ONE Application Server 管理インタフェースで接続パラメータを設定し、データベース接続を調整することができます。接続パラメータは次のカテゴリに分類されます。

- 接続
- スレッド
- データベースキャッシュ

Sun ONE Application Server 7 がサポートするデータベース

Sun ONE Application Server 7 では、Type 2 および Type 4 XA に対応した JDBC 2.0 スタイルのドライバを提供しており、次に示す主要なデータベースバックエンドに接続できます。

- DB2 v7
- Oracle 8.1.7

- Sybase v11
- PointBase バージョン 4.2RE

Sun ONE Application Server は外部の JDBC に準拠するドライバをすべてサポートしています。

JDBC データソースおよび接続プールのプロパティを、Sun ONE Application Server 管理インタフェース、または *asadmin* コマンド行ユーティリティを使用して追加し、設定できます。

JDBC データソースおよび接続プール設定についての詳細は、「JDBC 2.0 データソースの使用方法」の節を参照してください。

J2EE アプリケーションコンポーネントと移行

J2EE は、標準化されたモジュールコンポーネントをベースとして、これらのコンポーネントへの一連のサービスを提供し、アプリケーションの詳細な動作の多くを自動処理することにより、企業アプリケーションの開発を単純化します。複雑なプログラミングは必要ありません。J2EE v1.3 アーキテクチャは、複数のコンポーネント API を含んでいます。主な J2EE コンポーネントを次に示します。

- サーブレット
- Java Server Pages (JSP)
- EJB。メッセージ駆動型 Beans (MDB) を含む
- Java Database Connectivity (JDBC)
- Java Transaction Service (JTS)
- Java Naming and Directory Interface (JNDI)
- Java Message Service (JMS)

J2EE コンポーネントは個別にパッケージ化され、配備用 J2EE アプリケーションにバンドルされます。各コンポーネント、GIF および HTML ファイルやサーバーサイドユーティリティクラスなどの関連ファイル、および配備記述子は、1 つのモジュールとしてアセンブルされ、J2EE アプリケーションに追加されます。J2EE アプリケーションは、1 つまたは複数の Enterprise JavaBeans、Web、またはアプリケーションクライアントコンポーネントモジュールで構成されます。最終的な企業用ソリューションは、設計要件に応じて 1 つの J2EE アプリケーションを使用することも、2 つ以上の J2EE アプリケーションで構成することもできます。

J2EE アプリケーションおよびその各モジュールは、独自の配備記述子を備えています。配備記述子は、拡張子 .xml を持つ XML ドキュメントであり、コンポーネントの配備設定を記述します。たとえば、Enterprise JavaBean モジュールの配備記述子は、Enterprise JavaBean 用のトランザクション属性およびセキュリティ認証を宣言します。配備記述子情報が宣言されるため、Bean ソースコードを修正しなくても変更できます。J2EE サーバーは、実行時に配備記述子を読み込み、そのコンポーネント上で適切に動作します。

J2EE アプリケーションとそのすべてのモジュールは、Enterprise Archive (EAR) ファイルで配布されます。EAR ファイルは、拡張子 .ear を持つ標準 Java Archive (JAR) ファイルです。EAR ファイルは、EJB JAR ファイル、およびアプリケーションクライアント JAR ファイルまたは Web Archive (WAR) ファイル、またはその両方を含みます。これらのファイルの特徴を次に示します。

- 各 EJB JAR ファイルは、配備記述子、Enterprise JavaBeans ファイル、および関連ファイルを含む
- 各アプリケーションクライアントの JAR ファイルは、配備記述子、アプリケーションクライアントのクラスファイル、および関連ファイルを含む
- 各 WAR ファイルは、配備記述子、Web コンポーネントファイル、および関連リソースを含む

モジュールおよび EAR ファイルを使用すると、同一コンポーネントのいくつかを使用して、種類の異なる多数の J2EE アプリケーションをアセンブルできます。他のコーディングは必要ありません。さまざまな J2EE モジュールを単純に J2EE EAR ファイルにアセンブルするだけです。

移行プロセスでは、主に J2EE アプリケーションコンポーネント、モジュール、およびファイルの移動を行います。

種類の異なる J2EE コンポーネントの移行に関しては、第 2 章の Sun ONE Application Server 6.x から 7 への移行に関する問題の節を参照してください。

J2EE の基本情報の詳細については、以下を参照してください。

- J2EE チュートリアル - http://java.sun.com/j2ee/tutorial/
- J2EE の概要 - http://java.sun.com/j2ee/overview.html
- J2EE トピック - http://java.sun.com/j2ee

移行と再配備

この節では、J2EE アプリケーション、および特定のファイルを移行する理由について説明します。正常に移行された J2EE アプリケーションを Sun ONE Application Server に再配備できます。再配備についてもこの節で説明します。

次の項目があります。

- 移行が必要な理由
- 移行を必要とする要素
- 再配備とは

移行が必要な理由

J2EE 仕様は、アプリケーションに対する要求を幅広くカバーしていますが、その一方で標準として進化を続けています。これは、アプリケーションのある面をカバーしていない、または実装の詳細をアプリケーションプロバイダに任せているということではありません。

これらの製品の実装に依存する面は、アプリケーションサーバーの構成方法の違い、およびアプリケーションサーバー上への J2EE コンポーネントの配備方法の違いとして明らかになります。使用可能な構成の配列、および特定のアプリケーションサーバー製品で使用する配備ツールも、製品実装の差異の原因となります。

仕様が進化すること自体が、アプリケーションプロバイダにとっては課題になります。各コンポーネント API は別々に進化します。このため、製品ごとに統一性の度合いが異なります。特に、Sun ONE Application Server などの新しく開発された製品は、他の確立されたアプリケーションサーバープラットフォームに配備された J2EE アプリケーションコンポーネント、モジュール、およびファイルの違いに対処しなければなりません。このような違いに対処するには、ファイル命名規則、メッセージ構文など、J2EE 標準以前の実装詳細間でのマッピングが必要になります。

さらに、製品プロバイダは通常、製品に追加機能とサービスをバンドルしています。これらの機能は、カスタムの JSP タグや、専用の Java API ライブラリとして利用できます。

ただし、このような専用機能を使用すると、アプリケーションを移植できなくなります。

移行を必要とする要素

移行のために、J2EE アプリケーションは次のファイルカテゴリで構成されています。

- 配備記述子 (XML ファイル)
- 専用の API を含む JSP ソースファイル
- 専用の API を含む Java ソースファイル

配備記述子 (XML ファイル)

配備は、EJB (ejb-jar)、フロントエンド Web コンポーネント (war)、およびエンタープライズアプリケーション (ear) のそれぞれに対する配備記述子 (DD) を指定して行います。配備記述子は、J2EE コンポーネントおよびアプリケーションのすべての外部依存関係の解決に使用されます。DD の J2EE 仕様はすべてのアプリケーションサーバー製品で共通です。ただし、製品の実装に依存するアプリケーション関連コンポーネントの配備に関する面で、仕様が規定されていない部分があります。

JSP ソースファイル

J2EE では、カスタムタグ追加による JSP 拡張方法を指定しています。一部の製品には開発者のタスクを単純化する JSP 拡張が含まれています。ただし、これらの専用カスタムタグを使用すると、JSP ファイルを移植できなくなります。また、JSP では別の Java ソースファイルで定義されているメソッドも同様に呼び出せます。JSP で専用の API を使用している場合は、移行前にその部分を変更する必要があります。

Java ソースファイル

Java ソースファイルはサーブレット、EJB、または他のヘルパークラスのいずれかです。サーブレットおよび EJB は標準 J2EE サービスを直接呼び出せます。また、ヘルパークラスで定義されたメソッドも呼び出せます。Java ソースファイルは、EJB などのビジネスレイヤのアプリケーションのエンコードに使用します。ベンダーはそれぞれの製品にいくつかのサービスおよび専用 Java API をバンドルしています。このような専用 Java API を使用すると、アプリケーションが移植できなくなります。J2EE は標準として進化し続けているため、製品が変わればサポートしている J2EE コンポーネント API のバージョンが変わります。このような点についても移行で対応します。

上記のファイルカテゴリ内のファイルは、Sun ONE Application Server に移行する必要があります。ここに示された各ファイルカテゴリを移行する方法の詳細については、「Sun ONE Application Server 6.x から 7 への移行に関する問題」を参照してください。

再配備とは

再配備とは、以前のバージョンの Sun ONE Application Server や、これまで配備されていた移行済みのアプリケーション、また競合するアプリケーションサーバープラットフォームから、以前に配備されていたアプリケーションを再び配備することです。

アプリケーションの再配備は、『Sun ONE Application Server 管理者ガイド』に示されている標準的な配備概要に従って行います。ただし、移行を行う際に、Sun ONE Migration Tool for Application Servers (J2EE アプリケーション用)、または Sun ONE Migration Toolbox (Netdynamics および Netscape Application Servers 用) などの自動化ツールを使用する場合は、移行されたアプリケーションを配備する前に、移行後または配備前の作業が必要になることがあります。

使用可能な移行ツールについての詳細は、「移行の自動化」を参照してください。

第 2 章

移行に関する注意事項および手法

この章では、Sun ONE Application Server 6.0 または 6.5 から Sun ONE Application Server 7 に、J2EE アプリケーションを移行する場合の注意事項と手法について説明します。

この節では、コンポーネントレベルでの特定の移行作業についても説明します。

次の項目について説明します。

- Sun ONE Application Server 6.0/6.5 について
- Sun ONE Application Server 6.x から 7 への移行に関する問題
- 移行例 : iBank

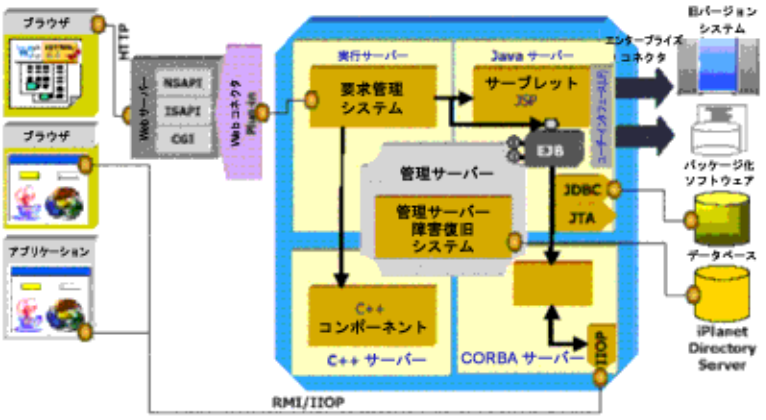
Sun ONE Application Server 6.0/6.5 について

Sun ONE Application Server 6.0 は、J2EE 1.2 仕様を全面的にベースにしたマルチプラットフォームのアプリケーションサーバーです。Windows NT/2000、Solaris、AIX、HP-UX などのプラットフォームがサポートされています。

また、Sun ONE Application Server 6.0 は、付属の専用 Web コネクタプラグインによって、多数の Web サーバーを統合化します。これらのコネクタによって、Sun ONE Web Server、Microsoft IIS、Apache などの Web サーバーと連携して動作することが可能になります。

次の図は、Sun ONE Application Server 6.0/6.5 のアーキテクチャを示しています。

Sun ONE Application Server 6.0/6.5 のアーキテクチャ



「Sun ONE Application Server 6.0/6.5 のアーキテクチャ」の図に示されるように、一般的にエンジンまたはプロセスと呼ばれる 4 つの内部サーバーが存在します。これらのプロセスが、Sun ONE Application Server でのすべての処理を実行します。Sun ONE Application Server 6.0/6.5 の 4 つの内部サーバーには、次のようなものがあります。

実行サーバー - ほとんどのシステムサービスを提供する (一部のサービスは、「管理サーバー」によって管理される)

管理サーバー - Sun ONE Application Server の管理および障害復旧用のシステムサービスを提供する

Java サーバー - Java アプリケーションのサービスを提供する

C++ サーバー - C++ で作成されたコンポーネントは、「C++ サーバー」で動作する

Web サーバーが Sun ONE Application Server 6.0/6.5 に要求を転送すると、要求はまず「実行サーバー」プロセス (KXS) によって受信されます。KXS プロセスは、その要求を「Java サーバー」プロセス (KJS) または「C++ サーバー」プロセス (KCS) のどちらかに転送します。KJS プロセスが Java プログラミングロジックを実行するのに対し、KCS プロセスは C++ プログラミングロジックを実行します。KJS プロセスと KCS プロセスはそれぞれ、特定数のスレッドを管理し、それらのスレッドでプログラミングロジックを最後まで実行します。結果は Web サーバーに返されてから、クライアントブラウザに送信されます。

Sun ONE Application Server 6.x から 7 への移行に関する問題

この節では、Sun ONE Application 6.0 または 6.5 から Sun ONE Application Server 7 に、標準的な J2EE アプリケーションの主要コンポーネントを移行する場合に生じる可能性のある問題について説明します。

この節で説明する移行時の問題点は、オンラインバンキングサービスをシミュレートする *iBank* という名前の J2EE アプリケーションを Sun ONE Application Server 6.0 または 6.5 から Sun ONE Application Server 7 に、実際に移行したときのプロセスに基づいています。このアプリケーションは、通常の J2EE アプリケーションを構成するすべての要素を反映しています。

iBank アプリケーションに適用される J2EE 仕様の重要なポイントとして、次のようなものが挙げられます。

- サーブレット、特に JSP ページへの切り替えを行うサーブレット (model-view-controller アーキテクチャ)
- JSP ページ、特に静的および動的なページの取り込みを含む JSP ページ
- JSP カスタムタグライブラリ
- HTTP セッションの作成および管理
- JDBC API 経由のデータベースアクセス
- Enterprise JavaBeans (ステートフル、またはステートレスなセッション Beans、CMP、および BMP エンティティ Beans)
- J2EE アプリケーションの標準パッケージ化メソッドを使用した、ラインへのアセンブリおよび配備

iBank アプリケーションの詳細については、付録 A の「iBank アプリケーションの仕様」を参照してください。

次の移行プロセスについては、以下に説明します。

- JDBC コードの移行
- サーブレットの移行
- Java Server Pages および JSP カスタムタグライブラリの移行
- JNDI コンテキストからのデータソースの取得
- EJB の移行
- Sun ONE Application Server 7 に対応した EJB の変更
- Web アプリケーションの移行

- エンタープライズ EJB モジュールの移行
- エンタープライズアプリケーションの移行

JDBC コードの移行

JDBC API を使用したデータベースアクセスには、2 つの方法があります。

- DriverManager インタフェース経由の接続の確立

  (JDBC 1.0 API)。特定のドライバを読み込み、接続 URL を提供します。この方法は、IBM の WebSphere 4.0 のような他のアプリケーションサーバーで使用されます。

- JDBC 2.0 データソースの使用方法

  データソースインタフェース (JDBC 2.0 API) は、設定可能な接続プールを介して使用できます。J2EE 1.2 に従って、データソースは JNDI ネーミングサービス経由でアクセスされます。

DriverManager インタフェース経由の接続の確立

この方法でデータベースにアクセスするのは、古い方法であまり効率的でないためお勧めしませんが、まだこの方法を使用しているアプリケーションもいくつか存在します。

この場合、アクセスコードは、以下のようになります。

```
public static final String driver =
"oracle.jdbc.driver.OracleDriver";

public static final String url =
"jdbc:oracle:thin:tmb_user/tmb_user@iben:1521:tmbank";

Class.forName(driver).newInstance();

Properties props = new Properties();

props.setProperty("user", "tmb_user");

props.setProperty("password", "tmb_user");

Connection conn = DriverManager.getConnection(url, props);
```

このコードは、Sun ONE Application Server 6.0/6.5 から Sun ONE Application Server 7 に完全に移植できます。ただし、Sun ONE Application Server が適切な JDBC ドライバを読み込むのに必要なクラスを検出できることが前提になります。Sun ONE Application Server 7 に配備されたアプリケーションにアクセスできる、必要なクラスを作成するには、次のいずれかの方法を実行します。

- Sun ONE Application Server 7 のインストールディレクトリ内の */lib* ディレクトリに、ドライバ実装用のアーカイブ (JAR または ZIP) を置きます。
- 管理サーバーの GUI を介して、ドライバのパスを設定し、CLASSPATH を修正します。サーバーインスタンス「server1」をクリックした後、右側のペインの「JVM 設定」タブをクリックします。次に、「パス設定」オプションをクリックし、テキスト入力ボックスの「クラスパスのサフィックス」にパスを追加します。設定変更を行ったら、「保存」をクリックして、新しい設定を適用します。サーバーを再起動すると、設定ファイル `server.xml` が変更されます。

「JVM 設定を使用したクラスパスのサフィックスの設定」の図では、GUI を介して、「クラスパスのサフィックス」にドライバのパスを追加しています。

JVM 設定を使用したクラスパスのサフィックスの設定

JDBC 2.0 データソースの使用方法

データベースへのアクセスに JDBC 2.0 データソースを使用すると、透過的な接続プーリングなど、パフォーマンス上の利点があります。コードや実装を単純化することで、生産性を向上させ、コードの移植性を高めることができます。

アプリケーション内のデータソースを使用するには、初期設定のフェーズを実行する必要があります。その後で、アプリケーションサーバーの JNDI ネーミングコンテキスト内でデータソースを登録します。データソースを登録すると、アプリケーションでは、JNDI コンテキストから対応する DataSource オブジェクトを取り出すことによって、データベースへの接続を簡単に取得できます。これらの作業については、次の項目で説明します。

- 「データソースの設定」
- 「JNDI 経由でのデータソースの検索および接続の取得」

データソースの設定

Sun ONE Application Server 6.0 では、データソースとそれに対応する JDBC ドライバを、サーバーのグラフィック管理コンソールから設定します。接続プールは、アプリケーションサーバーによって自動的に管理され、その管理ツールはプロパティの設定に使用できます。組み込まれている Type 2 の JDBC ドライバを使用すると、接続プールのプロパティは、ドライバごとに定義され、そのドライバを使用するすべてのデータソースで共有されます。

一方、サードパーティの JDBC ドライバを使用した場合、接続プールのプロパティは、データソースごとに定義されます。サードパーティの JDBC ドライバは、管理ツールから、または別のユーティリティ (Sun Solaris では `db_setup.sh`、Windows NT/2000 では `jdbcsetup`) から設定できます。さらに、コマンド行ユーティリティ `iasdeploy` を使用すると、プロパティを記述している XML ファイルからデータソースを設定できます。これらのユーティリティはすべて、Sun ONE Application Server のインストールルートディレクトリ内のサブディレクトリ `/bin/` に格納されています。

Sun ONE Application Server 7 では、サーバーのグラフィック管理コンソールまたはコマンド行ユーティリティ *asadmin* を使用して、データソースを設定できます。コマンド行ユーティリティ *asadmin* を起動するには、Windows の場合は asadmin.bat ファイルを、また Solaris の場合は asadmin ファイルをそれぞれ実行します。これらのファイルは、Sun ONE Application Server 7 のインストールディレクトリ内の bin ディレクトリに格納されています。次に、asadmin プロンプトから次のコマンドを入力すると、接続プールと JNDI リソースが作成されます。

asadmin ユーティリティを起動して、接続プールを作成するための構文は、次のとおりです。

```
asadmin>create-jdbc-connection-pool -u username -w password -H
hostname -p adminport [-s] [--instance instancename]
--datasourceclassname classname [--steadypoolsize=8]
[--maxpoolsize=32] [--maxwait=60000] [--poolresize=2]
[--idletimeout=300] [--isconnectvalidatereq=false]
[--validationmethod=auto-commit] [--validationtable tablename]
[--failconnection=false] [--description text] [--property
(name=value)[:name=value]*] connectionpoolid
```

次に例を示します。

```
asadmin>create-jdbc-connection-pool -u admin -w password -H cl1
-p 4848 –instance server1 --datasourceclassname
oracle.jdbc.pool.OracleConnectionPoolDataSource --property
(user-name=ibank_user):(password=ibank_user) oraclepool
```

この場合、Oracle データベース用の JDBC 接続プール「oraclepool」は、「ibank_user」というユーザー名と「ibank_user」というパスワードを持つデータベーススキーマを使用して作成されます。

JDBC リソースを作成する構文は、次のとおりです。

```
asadmin>create-jdbc-resource -u username -w password -H hostname
-p adminport [-s] [--instance instancename] --connectionpoolid id
[--enabled=true] [--description text] [--property
(name=value)[:name=value]*] jndiname
```

次に例を示します。

```
asadmin>create-jdbc-resource -u admin -w password -H cl1 -p 4848
--instance server1 --connectionpoolid oraclepool jdbc/IBANK
```

この場合、上記の「jdbc/IBANK」という JNDI 名で作成された接続プールに対して、JDBC リソースが作成されます。

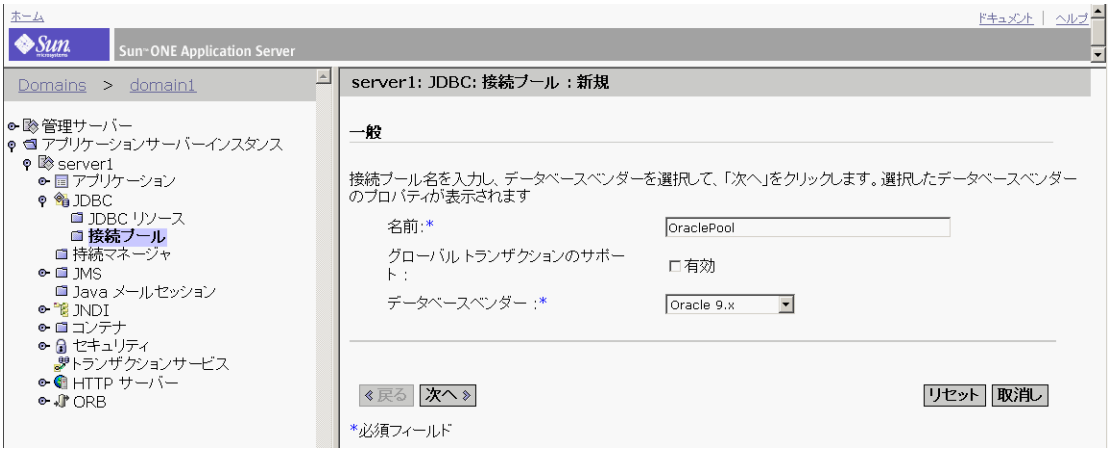
グラフィカルインタフェースを使用して、Sun ONE Application Server 7 にデータソースを登録する場合の実行手順は以下のとおりです。

1. データソースのクラス名を登録します。
   a. Sun ONE Application Server 7 のインストールディレクトリ内の */lib* ディレクトリに、データソースのクラス実装用のアーカイブ (JAR または ZIP) を置きます。
   b. 管理サーバーの GUI を介して、ドライバのパスを設定し、CLASSPATH を修正します。サーバーインスタンス「server1」をクリックした後、「JVM 設定」タブをクリックします。次に、「パス設定」をクリックして、「クラスパスのサフィックス」にパスを追加します。設定変更を行ったら、変更内容を保存して、新しい設定を適用します。サーバーを再起動すると、設定ファイル server.xml が変更されます。
2. データソースを登録します。

Sun ONE Application Server 7 では、データソースとそれに対応する JDBC ドライバを、サーバーのグラフィック管理インタフェースから設定します。

左側のペインには、Sun ONE Application Server で設定できるすべての項目がツリー表示されます。左側のペインにある「接続プール」をクリックすると、関連するエントリを指定できるページが右側のペインに表示されます。

GUI を使用した接続プールの設定



同様に、「データソース」をクリックすると、データソースのセットアップに必要なエントリが右側のペインに表示されます。

Sun ONE Application Server 7 固有の配備記述子 *sun-web.xml* は、それに応じて変更する必要があります。

たとえば、iBank アプリケーションに新しいデータソースを設定する場合、sun-web.xml は次のようなエントリを保持します。

```
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN' 'Http://localhost:8000/sun-web-app_2_3.dtd'>

<sun-web-app>

   <resource-ref>

      <res-ref-name>jdbc/iBank</res-ref-name>

      <jndi-name>jdbc/iBank</jndi-name>

      <default-resource-principal>

          <name>ibank_user</name>

          <password>ibank_user</password>

      </default-resource-principal>

   </resource-ref>

</sun-web-app>
```

JNDI 経由でのデータソースの検索および接続の取得

データソースから接続を取得するためのプロセスは、以下のようになります。

- 初期 JNDI コンテキストを取得する
- JNDI 検索を使用して、データソースへの参照を取得する
- この参照を使用して、接続を取得する

1. 初期 JNDI コンテキストを取得する

   異なる環境間での移植性を保証するため、InitialContext オブジェクト (サーブレット、JSP ページ、または EJB 内にある) を取得するのに使用するコードは、次のような単純なものにする必要があります。

   ```
   InitialContext ctx = new InitialContext();
   ```

2. データソースへの参照を取得する

   JNDI コンテキストにバインドされたデータソースへの参照を取得するには、初期のコンテキストオブジェクトから、データソースの JNDI 名を検索します。次に、この方法で検索されたオブジェクトを、DataSource タイプオブジェクトとしてキャストします。

   ```
   ds = (DataSource)ctx.lookup(JndiDataSourceName);
   ```

3. 接続を取得する

   この操作は非常に単純なもので、次のようなコード行を指定する必要があります。

   ```
   conn = ds.getConnection();
   ```

Sun ONE Application Server 6.0/6.5 および 7 はどちらも上記の技術に従って、データソースから接続を取得します。つまり、移行を行うときにコードを変更する必要はありません。

Java Server Pages および JSP カスタムタグライブラリの移行

Sun ONE Application Server 6.0/6.5 が JSP 1.1 仕様に準拠しているのに対し、Sun ONE Application Server 7 は JSP 1.2 仕様に準拠しています。

JSP 1.2 仕様には、多数の新機能に加えて、JSP 1.1 仕様ではあまり適切ではなかった部分の修正および説明が含まれています。

最も重要な変更点を以下に示します。

- JSP 1.2 は、サーブレット 2.3 および Java 2 に基づいている。JSP 1.2 アプリケーションは、JDK 1.1 だけをサポートするプラットフォームでは動作しない。JSP 1.2 は、JSP 1.1 との下位互換性があるため、JSP 1.1 アプリケーションは JSP 1.2 に準拠するコンテナを調整しなくても動作する
- JSP ページ用の XML 構文の定義が確定している。このため、JSP 1.2 に準拠するコンテナは、JSP 1.1 形式と JSP Document と呼ばれる新しい XML 形式の両方のファイルを受け入れる必要がある
- タグライブラリでは、サーブレット 2.3 のイベントリスナーを利用できる
- タグライブラリに、JSP ページを検証する新しいタイプの検証が追加されている
- タグライブラリの配布および配備用の新しいオプションが追加されている

基本的に、これらの変更は機能の拡張なので、JSP ページを JSP API 1.1 から 1.2 に移行する場合には必要ありません。

Sun ONE Application Server 6.0 または 6.5 の JSP カスタムタグライブラリの実装は、J2EE 仕様に準拠します。このため、JSP カスタムタグライブラリを Sun ONE Application Server 7 に移行する場合に、特別な問題が生じたり、修正が必要になることはありません。

サーブレットの移行

Sun ONE Application Server 6.0 および 6.5 で、サーブレット 2.2 API がサポートされているのに対し、Sun ONE Application Server 7 では、サーブレット 2.3 API がサポートされます。

サーブレット API 2.3 では、サーブレットのコアは比較的変更を受けずに残されており、ほとんどの変更は、コアの外側に追加された新しい機能に関係しています。

最も重要な機能を以下に示します。

- サーブレットには JDK 1.2 以降が必要
- フィルタメカニズムを新たに作成

- アプリケーションライフサイクルイベントを追加
- 新しい国際化サポートを追加
- 新しいエラーおよびセキュリティ属性を追加
- HttpUtils クラスを廃止
- 複数の DTD 動作を拡張、明確化

基本的に、これらの変更は機能の拡張なので、サーブレットをサーブレット API 2.2 から 2.3 に移行する場合には必要ありません。

ただし、アプリケーション内のサーブレットが JNDI を使用して J2EE アプリケーションのリソース (データソース、EJB など) にアクセスする場合、ソースファイルまたは配備記述子の変更が必要な場合があります。

これらの変更の詳細については、次の節で説明します。

- 「JNDI コンテキストからのデータソースの取得」
- 「JNDI コンテキスト内での EJB の宣言」

最後のケースとして、JSP ページが既存の Java クラスと同じ名前を持つ場合に、Sun ONE Application Server で命名の競合が発生するため、サーブレットのコード修正が必要になることがあります。この場合、競合は問題の生じている JSP ページの名前を修正することで解決します。また、さらに、この JSP ページを呼び出すサーブレットのコードを編集する必要があります。Sun ONE Application Server 6.0/6.5 と比べて、Sun ONE Application Server 7 では新しいクラスローダ階層が採用されているため、この問題は解決しています。この新しい方式では、特定のアプリケーションについて、1 つのクラスローダがすべての EJB モジュールを読み込み、別のクラスローダが Web モジュールを読み込みます。これらの 2 つのローダ同士は通信しないため、命名の競合が発生することはありません。

JNDI コンテキストからのデータソースの取得

JNDI コンテキストにバインドされたデータソースへの参照を取得するには、初期のコンテキストオブジェクトから、データソースの JNDI 名を検索します。次に、この方法で検索されたオブジェクトを、DataSource タイプオブジェクトとしてキャストします。

```
ds = (DataSource)ctx.lookup(JndiDataSourceName);
```

詳細については、前述の「JDBC コードの移行」を参照してください。

JNDI コンテキスト内での EJB の宣言

付録 C の「JNDI コンテキスト内での EJB の宣言」を参照してください。

EJB の移行

「Sun ONE Application Server 7 について」で説明したように、Sun ONE Application Server 6.0 および 6.5 で EJB 1.1 仕様がサポートされているのに対し、Sun ONE Application Server 7 では EJB 2.0 仕様もサポートされます。EJB 2.0 仕様は、次のような新機能をアーキテクチャに導入しています。

- メッセージ駆動型 Beans (MDB)
- コンテナ管理による持続性 (CMP) の向上
- CMP によるエンティティ Beans のコンテナ管理関係
- ローカルインタフェース
- EJB クエリ言語 (EJB QL)

Sun ONE Application Server 7 でも、EJB 1.1 仕様は引き続きサポートされますが、その拡張機能を活用するためには、EJB 2.0 アーキテクチャの使用をお勧めします。

EJB 1.1 から EJB 2.0 への移行については、付録 C を参照してください。

Sun ONE Application Server 7 に対応した EJB の変更

Sun ONE Application Server 6.0/6.5 から Sun ONE Application Server 7 に EJB を移行する場合、EJB のコード自体を変更する必要はありません。ただし、次のような DTD の変更が必要となります。

セッション Beans :

- `ejb-jar.xml` のような J2EE 標準 DD に備えて、最新の DTD URL を示すように、<!DOCTYPE 定義を変更します。
- `ias-ejb-jar.xml` ファイルを、DD に従って手動で作成した `sun-ejb-jar.xml` という名前の修正バージョンのファイルに置き換えます。詳細については、下記の URL を参照してください。

  ```
  <!DOCTYPE sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Sun
  ONE Application Server 7 EJB 2.0//EN'
  'http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2
  _0-0.dtd'>
  ```

- `sun-ejb-jar.xml` の場合、すべての EJB の JNDI 名は `ejb/` で開始する必要があります。これは Sun ONE Application Server 6.5 の場合に必要です。EJB の JNDI 名には、`ejb/<ejb-name>` だけしか指定できません。`<ejb-name>` には、`ejb-jar.xml` 内で宣言した EJB の名前が入ります。ただし、Sun ONE

Application Server 7 では、EJB の JNDI 名を宣言できる新しいタグが sun-ejb-jar.xml 内に導入されたため、この点が変わりました。このような柔軟性が Sun ONE Application Server 7 で提供されたので、アプリケーション全体で JNDI 名を変更するのを避けるために、EJB の JNDI 名は `<jndi-name>` タグ内で、`ejb/<ejb-name>` と宣言することをお勧めします。

エンティティ Beans :

- `ejb-jar.xml` のような J2EE 標準 DD に備えて、最新の DTD URL を示すように、`<!DOCTYPE` 定義を変更します。
- `ejb-jar.xml` のすべての CMP に対して、1.1 の値を指定した <cmp-version> タグを挿入します。
- すべての `<ejb-name>-ias-cmp.xml` ファイルを、手動で作成した 1 つの `sun-cmp-mappings.xml` ファイルに置き換えます。詳細については、下記の URL を参照してください。

  ```
  <!DOCTYPE sun-cmp-mappings PUBLIC '-//Sun Microsystems, Inc.//DTD
  Sun ONE Application Server 7 OR Mapping //EN'
  'http://www.sun.com/software/sunone/appserver/dtds/sun-cmp_mappi
  ng_1_0.dtd'>
  ```

- Sun ONE Application Server 7 のインストールディレクトリ内の bin ディレクトリに格納されている *capture-schema* ユーティリティを使用して dbschema を生成し、エンティティ Beans 用の `META-INF` フォルダより上位に置きます。
- `ias-ejb-jar.xml` を、Sun ONE Application Server 7 で `sun-ejb.jar.xml` という名前に変更された新しいバージョンのファイルに置き換えます。
- Sun ONE Application Server 6.5 では、ファインダ sql が `<ejb-name>-ias-cmp.xml` 内に直接組み込まれていましたが、Sun ONE Application Server 7 では、数学式を使用して、さまざまな検索メソッド用の `<query-filter>` を宣言するように変更されています。

Web アプリケーションの移行

Sun ONE Application Server 6.0 および 6.5 で、サーブレット (サーブレット API 2.2) と JSP (JSP 1.1) がサポートされているのに対し、Sun ONE Application Server 7 では、サーブレット (サーブレット API 2.3) と JSP (JSP 1.2) がサポートされます。

これらの環境内では、異なるアプリケーションコンポーネント (サーブレット、JSP、HTML ページ、およびその他のリソース) を 1 つのアーカイブファイル (J2EE 標準 Web アプリケーションモジュール) にまとめてから、そのアーカイブファイルをアプリケーションサーバー上に配備する必要があります。

J2EE 1.3 仕様によると、Web アプリケーションは、次のような構造を持ったアーカイブファイル (.WAR ファイル) です。

- HTML ページ、JSP ページ、画像、およびその他の「静的な」アプリケーションのリソースを含むルートディレクトリ
- 使用している SDK のバージョン情報、およびアーカイブに含まれるファイルのリスト (オプション) などが入ったアーカイブマニフェストファイル (MANIFEST.MF) を含む *META-INF/* ディレクトリ
- アプリケーション配備記述子 (web.xml ファイル)、およびアプリケーションで使用されるすべての Java クラスとライブラリを含む *WEB-INF/* ディレクトリ。次のように構成される
    - サブディレクトリ *classes/* には、アプリケーションのコンパイル済みのクラス (サーブレット、補助クラスなど) がツリー構造で格納され、パッケージに編成される
    - *lib/* ディレクトリには、アプリケーションによって使用される任意の Java ライブラリ (.jar ファイル) が含まれる

Web アプリケーションモジュールの移行

Sun ONE Application Server 6.0/6.5 から Sun ONE Application Server 7 にアプリケーションを移行する場合、Java/JSP のコードを変更する必要はありませんが、次のような変更が必要です。

- web.xml

  Sun ONE Application Server 7 は J2EE 1.3 標準に準拠しているため、WAR 内の `web.xml` ファイルは、改訂された DTD に従う必要があります。この改訂版 DTD は、`http://java.sun.com/dtd/web-app_2_3.dtd` から入手できます。幸い、この DTD は以前のバージョンの DTD のスーパーセットなので、移行する `web.xml` 内の `<! DOCTYPE` 定義だけを変更する必要があります。修正した `<! DOCTYPE` 宣言は、次のようになります。

  ```
  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
  Application 2.3//EN"  "http://java.sun.com/dtd/web-app_2_3.dtd">
  ```

- ias-web.xml

  Sun ONE Application Server 7 では、このファイルの名前は sun-web.xml に変更されています。

  この XML ファイルは、Web アプリケーションで必要な Sun ONE Application Server 7 固有のプロパティまたはリソースを宣言するときに使用します。

  注 : このファイルの内容については、次の節を参照してください。

  Sun ONE Application Server 6.5 アプリケーションの ias-web.xml が存在し、Sun ONE Application Server 6.5 固有のプロパティが宣言されている場合、Sun ONE Application Server 7 標準に移行するときに、このファイルが必要となります。ファイル名は sun-web.xml に変更する必要があります。その他の詳細については、下記を参照してください。

  ```
  <!DOCTYPE sun-web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Sun
  ONE Application Server 7 Servlet 2.3//EN'
  'http://www.sun.com/software/sunone/appserver/dtds/sun-web-app_2
  _3-0.dtd'>
  ```

前述した方法で web.xml と ias-web.xml を移行すると、管理サーバーの Sun ONE Application Server 7 の GUI インタフェース、またはコマンド行ユーティリティ *asadmin* を使用して、Web アプリケーション (.WAR アーカイブ) を配備することができます。配備コマンドでは、アプリケーションのタイプを *web* として指定する必要があります。

コマンド行ユーティリティ asadmin を起動するには、Sun ONE Application Server 7 のインストールディレクトリ内の bin ディレクトリに格納されている asadmin.bat ファイルを実行します。

asadmin プロンプトから入力するコマンドは、次のようになります。

```
asadmin> deploy -u username -w password -H hostname -p adminport
--type web [--contextroot contextroot] [--force=true] [--name
component-name] [--upload=true] [--instance instancename]
filepath
```

「Sun ONE Application Server 7 でのアプリケーションの配備」に記載されるように、配備は Sun ONE Studio 開発環境からでも行うことができます。

サーブレットおよび JSP の移行時の特定の障害

Sun ONE Application Server 6.0/6.5 から Sun ONE Application Server 7 に、サーブレット /JSP アプリケーションのコンポーネントを実際に移行する場合、コンポーネントのコードを修正する必要はありません。

Web アプリケーションがデータソースなどのサーバーリソースを使用している場合、Sun ONE Application Server 7 では、このリソースを web.xml 内および sun-web.xml 内で同様に宣言する必要があります。jdbc/iBank という名前のデータソースを宣言する場合、web.xml 内で宣言される <resource-ref> タグは、次のようになります。

```
<resource-ref>
  <res-ref-name>jdbc/iBank</res-ref-name>
  <res-type>javax.sql.XADataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

これに対応する sun-web.xml 内の宣言は、次のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<! DOCTYPE FIX ME: need confirmation on the DTD to be used for
this file
<sun-web-app>
   <resource-ref>
      <res-ref-name>jdbc/iBank</res-ref-name>
      <jndi-name>jdbc/iBank</jndi-name>
   </resource-ref> </sun-web-app>
```

エンタープライズ EJB モジュールの移行

Sun ONE Application Server 6.0 および 6.5 で、EJB 1.1 API がサポートされているのに対し、Sun ONE Application Server 7 では、EJB 2.0 API がサポートされます。その結果、どちらも以下のものをサポートできます。

- ステートフル、またはステートレスなセッション Beans
- Bean 管理による持続性 (BMP)、またはコンテナ管理による持続性 (CMP) を持つエンティティ Beans

ただし、EJB 2.0 API では、セッション Beans およびエンティティ Beans に加えて、メッセージ駆動型 Beans と呼ばれる新しいタイプの Enterprise Java Beans が導入されています。

J2EE 1.3 仕様によると、EJB のさまざまなコンポーネントは、次のような構造を持った JAR ファイルにまとめる必要があります。

- *ejb-jar.xml* という名前の XML 配備記述子を持つ `META-INF/` ディレクトリ
- パッケージ内のホームインタフェースおよびリモートインタフェース、Bean の実装クラスおよび補助クラスに対応する `.class` ファイル

Sun ONE Application Server では、このアーカイブ構造を維持します。ただし、EJB 1.1 仕様では、各 EJB コンテナベンダーに、適切であると思われる次のような機能を実装する余地を残しています。

- CMP EJB のデータベースの持続性 (特に、データベース表内の Bean の CMP フィールドとカラム間でのマッピング設定)
- CMP Beans 用カスタムファインダメソッドのロジックの実装

予期されていたように、Sun ONE Application Server 6.0 または 6.5 と Sun ONE Application Server 7 では、異なる点がいくつか存在するため、アプリケーションを移行する場合に、特別な留意が必要な部分があります。以下のように、いくつかの XML ファイルを修正する必要があります。

- `ejb-jar.xml` のような J2EE 標準 DD に備えて、最新の DTD URL を示すように、`<!DOCTYPE` 定義を変更します。
- `ias-ejb-jar.xml` ファイルを、DTD に従って手動で作成した `sun-ejb-jar.xml` という名前の修正バージョンのファイルに置き換えます。下記の URL を参照してください。

  ```
  <!DOCTYPE sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Sun
  ONE Application Server 7 EJB 2.0//EN'
  'http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2
  _0-0.dtd'>
  ```

- すべての `<ejb-name>-ias-cmp.xml` ファイルを、手動で作成した 1 つの `sun-cmp-mappings.xml` ファイルに置き換えます。下記の URL を参照してください。

  ```
  <!DOCTYPE sun-cmp-mappings PUBLIC '-//Sun Microsystems, Inc.//DTD
  Sun ONE Application Server 7 OR Mapping //EN'
  'http://www.sun.com/software/sunone/appserver/dtds/sun-cmp_mappi
  ng_1_0.dtd'>
  ```

- CMP エンティティ Beans についてのみ、Sun ONE Application Server 7 のインストールディレクトリ内の bin ディレクトリに格納されている *capture-schema* ユーティリティを使用して dbschema を生成し、エンティティ Beans 用の `META-INF` フォルダより上のディレクトリに置きます。

エンタープライズアプリケーションの移行

J2EE 仕様によると、エンタープライズアプリケーションは、次のような構造を持った EAR ファイルです。

- *application.xml* という名前の J2EE アプリケーションの XML 配備記述子を持つ `META-INF/` ディレクトリ
- エンタープライズアプリケーションの EJB モジュール用の .JAR アーカイブファイルと Web モジュール用の .WAR アーカイブファイル

アプリケーション配備記述子では、エンタープライズアプリケーションと Web アプリケーションのコンテキストルートを構成するモジュールを定義します。

Sun ONE Application Server 6.0/6.5 と 7 では、主に J2EE モデルがサポートされます。このモデルでは、アプリケーションがエンタープライズアーカイブ (EAR) ファイル (拡張子は .ear) の形式でパッケージ化されます。アプリケーションはさらに J2EE モジュールの集まりに分割され、EJB 用の Java アーカイブ (拡張子が .jar の JAR ファイル) とサーブレットおよび JSP 用の Web アーカイブ (拡張子が .war の WAR ファイル) にパッケージ化されます。

このため、エンタープライズアプリケーションを配備する前に、以下の手順を実行する必要があります。

- EJB を 1 つ以上の EJB モジュールにパッケージ化します。
- Web アプリケーションのコンポーネントを Web モジュールにパッケージ化します。
- EJB モジュールと Web モジュールをエンタープライズアプリケーションモジュールにアセンブルします。
- エンタープライズアプリケーションのルートコンテキスト名を定義します。これによって、アプリケーションにアクセスするための URL を特定します。

注 : Sun ONE Application Server 7 では、Sun ONE Application Server 6.0/6.5 にはなかった新しいクラスローダ階層を使用しています。この新しい方式では、特定のアプリケーションについて、1 つのクラスローダがすべての EJB モジュールを読み込み、別のクラスローダが Web モジュールを読み込みます。これら 2 つには、親子階層の関係があり、JAR モジュールクラスローダが WAR モジュールクラスローダの親モジュールになります。このため、JAR クラスローダによって読み込まれたクラスはすべて、WAR モジュールでも使用可能またはアクセス可能ですが、反対に WAR クラスローダによって読み込まれたクラスを JAR モジュールからアクセスすることはできません。したがって、JAR ばかりでなく WAR が必要とするクラスが存在する場合、そのクラスは JAR モジュール内だけでパッケージ化します。この指針に従わない場合、ClassCastException のクラス競合が発生する可能性があります。

アプリケーションルートコンテキストとアクセス URL

アプリケーションのアクセス URL (アプリケーションの Web モジュールのルートコンテキスト) に関して、Sun ONE Application Server 6.0/6.5 と Sun ONE Application Server 7 では、特別な違いが 1 つあります。

hostname という名前のサーバー上に配備されたアプリケーションのルートコンテキスト名が AppName である場合、このアプリケーションのアクセス URL は、使用されるアプリケーションサーバーによって異なります。

- Sun ONE Application Server 6.0 または 6.5 の場合、常に Web フロントエンドと連携して使用されるので、アプリケーションのアクセス URL は、次のような形式になる (Web サーバーは、標準の HTTP ポート番号 80 で設定されているものとする)

  ```
  http://hostname/NASApp/AppName/
  ```

- Sun ONE Application Server 7 の場合、URL は次のような形式になる

  ```
  http://hostname:port/AppName/
  ```

Sun ONE Application Server 7 がデフォルトで使用する TCP ポートは、ポート番号 80 です。

Sun ONE Application Server 6.0/6.5 と Sun ONE Application Server 7 でのアクセス URL についての違いはわずかなものに見えますが、絶対 URL 参照を使用するアプリケーションを移行する場合に、問題が生じる可能性があります。このような問題が発生する場合、Sun ONE Application Server 6.0/6.5 用の Web サーバープラグインで使用される固有のマーカーが付加されないように、コードを編集して、絶対 URL 参照を更新する必要があります。

固有の拡張子の移行

Sun ONE Application Server 6.0/6.5 環境独自の多数のクラスが、アプリケーションで使用されている可能性があります。Sun ONE Application Server 6.x で使用される独自の Sun ONE パッケージの一部を以下に示します。

- com.iplanet.server.servlet.extension
- com.kivasoft.dlm
- com.iplanetiplanet.server.jdbc
- com.kivasoft.util
- com.netscape.server.servlet.extension
- com.kivasoft
- com.netscape.server

これらの API は、Sun ONE Application Server 7 ではサポートされていません。上記のパッケージに属するクラスを使用するアプリケーションの場合、アプリケーションが標準 J2EE API を使用するように作成し直す必要があります。また、カスタム JSP タグと UIF フレームワークを使用するアプリケーションについても、標準 J2EE API を使用するように作成し直す必要があります。

移行例 : iBank

この節では、Sun ONE Application 6.0 または 6.5 から Sun ONE Application Server 7 に、標準的な J2EE アプリケーションの主要コンポーネントを移行するためのプロセスについて説明します。状況ごとに、移行によって生じる問題に焦点をあてて、それらの問題に対する実用的な解決策を提言します。

この移行プロセスで示す J2EE アプリケーションは、「iBank」という名前で、これは、Sun ONE Application Server 6.0 または 6.5 から Sun ONE Application Server 7 への iBank アプリケーションの実際の移行に基づいています。iBank はオンラインバンキングサービスをシミュレートし、通常の J2EE アプリケーションに付随するあらゆる特徴を持っています。

iBank アプリケーションに適用される J2EE 仕様の重要なポイントを以下に示します。

- サーブレット、特に JSP ページへの切り替えを行うサーブレット (model-view-controller アーキテクチャ)
- JSP ページ、特に静的および動的なページの取り込みを含む JSP ページ
- JSP カスタムタグライブラリ
- HTTP セッションの作成および管理
- JDBC API 経由のデータベースアクセス
- Enterprise JavaBeans (ステートフル、またはステートレスなセッション Beans、CMP、および BMP エンティティ Beans)
- J2EE アプリケーションの標準パッケージ化メソッドに対応したアセンブリおよび配備

iBank アプリケーションの詳細については、付録 A の「iBank アプリケーションの仕様」を参照してください。

iBank アプリケーションを Sun ONE Application Server 7 に移行するには、配備記述子を手動で変更するか、または Sun ONE Studio や Sun ONE Migration Tool を使用します。上記の 3 つのプロセスの中で、お勧めするのは Sun ONE Migration Tool です。CMP を 2.0 に変換しないで移行を行う必要がある場合は、次の「iBank アプリケーションの手動移行」を行うか、または Sun ONE Migration Tool を使用します。

このガイドでは、手動の移行プロセスと Sun ONE Studio を使用した移行について説明します。Sun ONE Migration Tool を使用して、iBank を自動的に移行する手順については、Migration Tool に付属するマニュアルを参照してください。

iBank アプリケーションの手動移行

Sun ONE Application Server 7 で CMP 1.1 がサポートされているため、手動の移行でソースコードを大幅に修正する必要はありません。ただし、アプリケーションを手動で移行する場合、以下の点について、いくつか変更が必要になります。

Web アプリケーションの変更

Sun ONE Application Server 6.0/6.5 から Sun ONE Application Server 7 に iBank を移行する場合、iBank アプリケーションの Web アプリケーション部分を変更する必要はありません。ソースディレクトリから `ias-web.xml` ファイルを削除します。これは Sun ONE Application Server 7 の配備記述子 `sun-web.xml` ファイル内の対応するデータにアクセスできる情報がこのファイルに入っていないからです。`web.xml` は変更する必要ありません。

ただし、一般的には、サーバー固有のリソースにマップする必要がある web.xml 内に情報が含まれていることがあるため、このような場合、`sun-web.xml` での宣言が必要となります。たとえば、web.xml ファイルで javax.sql.Datasource タイプのリソース参照が宣言されている場合、sun-web.xml 内で、サーバー上の実際のデータソースの JNDI 名に、その参照をマップする必要があります。

移行を行う場合は、新しい sun-web.xml を作成する必要があります。作成プロセスは、以下のとおりです。

1. 最上部に、次のような DOCTYPE 定義を記述した新しい XML ファイルを作成します。

   ```
   <!DOCTYPE sun-web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Sun
   ONE Application Server 7 Servlet 2.3//EN'
   'http://www.sun.com/software/sunone/appserver/dtds/sun-web-app_2
   _3-0.dtd'>
   ```

   `sun-web.xml` という名前で、このファイルを保存します。

2. DOCTYPE 定義から見て分かるように、この XML ファイルのルートタグは `sun-web` です。DTD では、この要素は次のように定義します。

```
<!ELEMENT sun-web-app (security-role-mapping*, servlet*,
session-config?, resource-env-ref*, resource-ref*, ejb-ref*,
cache?, class-loader?, jsp-config?, locale-charset-info?,
property*)>
```

上記の宣言から、タグはすべて任意指定であることが明らかなので、デフォルトの sun-web.xml は、次のようになります。

```
<!DOCTYPE sun-web-app SYSTEM
"http://www.sun.com/software/sunone/appserver/dtds/sun-web-app_2
_3-0.dtd">

</sun-web-app>
```

3. リソース参照を宣言する場合、要素の宣言は、次のようになります。

```
<!ELEMENT resource-ref (res-ref-name, jndi-name,
default-resource-principal?)>
```

サブ要素は、次のようになります。

```
<!ELEMENT res-ref-name (#PCDATA)>

<!ELEMENT default-resource-principal (name, password)>

<!ELEMENT jndi-name (#PCDATA)>
```

iBank のリソース参照の詳細の場合、sun-web.xml は、次のようになります。

```
<sun-web-app>
  <resource-ref>
    <res-ref-name>jdbc/IBank</res-ref-name>
    <jndi-name>jdbc/IBank</jndi-name>
    <default-resource-principal>
      <name>ibank_user</name>
      <password>ibank_user</password>
    </default-resource-principal>
  </resource-ref>
</sun-web-app>
```

EJB の変更

Sun ONE Application Server 6.5 から Sun ONE Application Server 7 に iBank を移行する場合、EJB のコードを変更する必要はありません。

セッション Beans :

ejb-jar.xml : ejb-jar.xml に備えて、最新の DTD URL を示すように、<!DOCTYPE 定義を変更します。この新しい定義は、次のようになります。

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
```

ias-ejb-jar.xml : Sun ONE Application server 6.5 の ias-ejb-jar.xml は、Sun ONE Application server 7 の sun-ejb-jar.xml に置き換えられています。これらの 2 つの XML ファイルの DTD は、根本的に異なっているため、移行を行う場合は、ejb-jar.xml と ias-ejb-jar.xml から必要な情報を抽出して、新しい sun-ejb-jar.xml を作成する必要があります。作成プロセスは、以下のとおりです。

1. 最上部に、次のような DOCTYPE 定義を記述した新しい XML ファイルを作成します。

   ```
   <!DOCTYPE sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Sun
   ONE Application Server 7 EJB 2.0//EN'
   'http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2
   _0-0.dtd'>
   ```

   このファイルに sun-ejb-jar.xml という名前を付けて、修正した ejb-jar.xml と一緒に保存します。

2. DOCTYPE 定義から見て分かるように、この XML ファイルのルートタグは sun-ejb-jar です。DTD では、この要素は次のように定義します。

   ```
   <!ELEMENT sun-ejb-jar (security-role-mapping*,enterprise-beans)>
   ```

   security-role-mapping タグは、ejb-jar.xml で宣言されているセキュリティロールをマッピングするためのものです。iBank アプリケーションの場合、ejb-jar.xml ファイルで宣言されているセキュリティがないので、security-role-mapping タグ (任意指定) は飛ばして、enterprise-beans タグに重点を置きます。現在、sun-ejb-jar.xml ファイルは、次のようになっています。

   ```
   <sun-ejb-jar>

       <enterprise-beans>

       </enterprise-beans>

   </sun-ejb-jar>
   ```

   注 : ここでは、ドキュメントのヘッダー部分、すなわち、XML 宣言と DOCTYPE 定義は省略しています。

3. *enterprise-beans* 要素は、次のように DTD で定義します。

   ```
   <!ELEMENT enterprise-beans (name?, unique-id?, ejb*,
   pm-descriptors?, cmp-resource?)>
   ```

   任意指定の *name* 要素には、*enterprise-beans* の正準名が入ります。この要素には、任意の名前を付けることができます。ここでは、このタグは省略します。

unique-id 要素は、Sun ONE Application Server で使用されるもので、アプリケーションの配備時に、Application Server によって自動的に挿入されます。ここでは、このタグは省略します。

ここで着目すべきタグは、*EJB* 要素です。このタグは、単一の EJB の実行時バインドを指定する要素です。これは、次のように DTD で定義します。

```
<!ELEMENT ejb (ejb-name, jndi-name?, ejb-ref*, resource-ref*,
resource-env-ref*, pass-by-reference?, cmp?, principal?,
mdb-connection-factory?, jms-durable-subscription-name?,
jms-max-messages-load?, ior-security-config?,
is-read-only-bean?, refresh-period-in-seconds?, commit-option?,
gen-classes?, bean-pool?, bean-cache?)>
```

この場合、*ejb* 要素に *ejb-name* 要素が含まれます。*ejb-name* 要素には、EJB の正準名が入ります。この名前は、その EJB 用の `ejb-jar.xml` の *ejb-name* 要素内で宣言された名前と同じになります。また、この要素には、EJB の *jndi-name* も含まれます。Sun ONE Application Server 6.5 から 7 で改善された点の 1 つは、Bean の開発者が EJB の *ejb-name* と *jndi-name* を自由に変更できるようになったという柔軟性です。Sun ONE Application Server 6.5 では、EJB の JNDI 名はデフォルトで `ejb/<ejb-name>` に設定されていました。

移行を円滑に行うためには、EJB の jndi-name をそのままの状態にしておき、その他のすべてのリソースについても Sun ONE Application Server 6.5 のときと同じものにしておく必要があります。このため、ここではすべての EJB の *ejb-name* を `ejb/<ejb-name>` と宣言します。

上記のロジックを使用すると、`sun-ejb-jar.xml` は、次のようになります。

```
<sun-ejb-jar>
   <enterprise-beans>
      <ejb>
         <ejb-name>BankTeller</ejb-name>
         <jndi-name>ejb/BankTeller</jndi-name>
      </ejb>
      <ejb>
         <ejb-name>InterestCalculator</ejb-name>
         <jndi-name>ejb/InterestCalculator</jndi-name>
      </ejb>
   </enterprise-beans>
</sun-ejb-jar>
```

4. ejb-jar.xml 内の各 <ejb-ref> 要素について、sun-ejb-jar.xml 内に対応する <ejb-ref> 要素が存在します。ejb-jar.xml 内の <ejb-ref> 要素は、その EJB の Bean クラス内から、参照されるすべての EJB を宣言するのに使用します。Bean クラスコードは、<ejb-ref-name> を使用して、EJB を参照するときに、この <ejb-ref-name> をアプリケーションサーバー上の Bean の実際の <jndi-name> にマップする必要があります。このため、これは EJB 実装で参照される名前と Bean の実際の JNDI 名の間に、抽象化層を追加するためのメカニズムとして機能します。

   上記で説明したロジックを使用して、BankTellerEJB について見てみます。ejb-jar.xml では、2 つの <ejb-ref> 宣言がこの EJB 内に存在します。1 つめの宣言は、Cutomer EJB (エンティティ Bean モジュールのエンティティ Bean) 用です。前述の 3 で説明したように、すべての EJB の JNDI 名は、ejb/<ejb-name> のままにしておき、この宣言を sun-ejb-jar.xml 内に追加します。

```
<sun-ejb-jar>
   <enterprise-beans>
      <ejb>
         <ejb-name>BankTeller</ejb-name>
         <jndi-name>ejb/BankTeller</jndi-name>
         <ejb-ref>
            <ejb-ref-name>Customer</ejb-ref-name>
            <jndi-name>ejb/Customer</jndi-name>
         </ejb-ref>
      </ejb>
      <ejb>
         <ejb-name>InterestCalculator</ejb-name>
         <jndi-name>ejb/InterestCalculator</jndi-name>
      </ejb>
   </enterprise-beans>
</sun-ejb-jar>
```

   同様に、Account EJB 用の <ejb-ref> タグを追加します。InterestCalculator Bean の場合、<ejb-ref> タグは、ejb-jar.xml 内に存在しないため、sun-ejb-jar.xml 内でも必要ありません。現在、sun-ejb-jar.xml ファイルは、次のようになっています。

```
<sun-ejb-jar>
   <enterprise-beans>
```

```
        <ejb>
           <ejb-name>BankTeller</ejb-name>
           <jndi-name>ejb/BankTeller</jndi-name>
           <ejb-ref>
              <ejb-ref-name>Customer</ejb-ref-name>
              <jndi-name>ejb/Customer</jndi-name>
           </ejb-ref>
           <ejb-ref>
              <ejb-ref-name>Account</ejb-ref-name>
              <jndi-name>ejb/Account</jndi-name>
           </ejb-ref>
        </ejb>
        <ejb>
           <ejb-name>InterestCalculator</ejb-name>
           <jndi-name>ejb/InterestCalculator</jndi-name>
        </ejb>
     </enterprise-beans>
  </sun-ejb-jar>
```

5. ejb 要素には、pass-by-reference 要素 `<!ELEMENT pass-by-reference (#PCData)` が含まれます。

   pass-by-reference 要素は、参照渡しセマンティクスの使用を制御します。EJB 仕様では、デフォルトの動作モードとなる参照渡しを必要としています。これは、対応できない操作やより高速なパフォーマンスに向けて、true を設定することができます。これは、閉ざされた環境にある EJB モジュールすべてに適用できます。設定できる値は、true と false です。デフォルト値は false です。

6. ejb 要素には、bean-cache 要素も含まれます。

```
<!ELEMENT bean-cache (max-cache-size?,
is-cache-overflow-allowed?, cache-idle-timeout-in-seconds?,
removal-timeout-in-seconds?, victim-selection-policy?)>
```

この要素は、ステートフルセッション Beans とエンティティ Beans だけに使用されます。iBank の場合、*BankTeller* セッション Bean だけが、このエントリを含みます。

このタグでは、*max-cache-size* によって、キャッシュ内の Beans の最大数を定義します。*cache-idle-timeout-in-seconds* では、ステートフルセッション Bean やエンティティ Bean がアイドル状態でキャッシュ内に存在できる最大時間を指定します。この時間の経過後、Bean はバックアップストアに非活性化されます。この時間は、サーバーが参考にします。*cache-idle-timeout-in-seconds* のデフォルト値は 10 分です。

Bean が非活性化される (バックアップストアでアイドル状態になっている) 時間は、*removal-timeout-in-seconds* パラメータによって制御されます。Bean は removal-timeout-in-seconds の値を超えた時間アクセスされなかった場合、バックアップストアから削除されてしまうので、クライアントからアクセスできなくなります。removal-timeout-in-seconds のデフォルト値は 60 分です。

上記のエントリが追加されて、`sun-ejb-jar.xml` ファイルは、次のようになります。

```
<sun-ejb-jar>
   <enterprise-beans>
      <ejb>
         <ejb-name>BankTeller</ejb-name>
         <jndi-name>ejb/BankTeller</jndi-name>
         <ejb-ref>
            <ejb-ref-name>Customer</ejb-ref-name>
            <jndi-name>ejb/Customer</jndi-name>
         </ejb-ref>
         <ejb-ref>
            <ejb-ref-name>Account</ejb-ref-name>
            <jndi-name>ejb/Account</jndi-name>
         </ejb-ref>
         <pass-by-reference>false</pass-by-reference>
         <bean-cache>
            <cache-idle-timeout-in-seconds>
               0
```

```
            </cache-idle-timeout-in-seconds>
            <removal-timeout-in-seconds>
               0
            </removal-timeout-in-seconds>
         </bean-cache>
      </ejb>
      <ejb>
         <ejb-name>InterestCalculator</ejb-name>
         <jndi-name>ejb/InterestCalculator</jndi-name>
         <pass-by-reference>false</pass-by-reference>
      </ejb>
   </enterprise-beans>
</sun-ejb-jar>
```

7. ステートレスセッション Bean とメッセージ駆動型 Bean のプールだけに使用される要素は、bean-pool です。

   ```
   <!ELEMENT bean-pool (steady-pool-size?, resize-quantity?,
   max-pool-size?, pool-idle-timeout-in-seconds?,
   max-wait-time-in-millis?)>
   ```

   Beans の初期数と最小数を指定する *steady-pool-size* は、プールで保持する必要があります。

   resize-quantity では、プールが「プールマネージャ」によってサービスが実行されている場合に、作成または削除する Bean の数を指定します。

   max-pool-size では、プールの最大サイズを指定します。指定できる値は、0 から MAX_INTEGER の範囲です。

   pool-idle-timeout-in-seconds では、ステートレスセッション Bean やメッセージ駆動型 Bean がアイドル状態でプール内に存在できる最大時間を指定します。

   最終的に、`sun-ejb-jar.xml` は、次のような構成になります。

   ```
   <sun-ejb-jar>
      <enterprise-beans>
         <ejb>
            <ejb-name>BankTeller</ejb-name>
            <jndi-name>ejb/BankTeller</jndi-name>
   ```

```
            <ejb-ref>
               <ejb-ref-name>Customer</ejb-ref-name>
               <jndi-name>ejb/Customer</jndi-name>
            </ejb-ref>
            <ejb-ref>
               <ejb-ref-name>Account</ejb-ref-name>
               <jndi-name>ejb/Account</jndi-name>
            </ejb-ref>
            <pass-by-reference>false</pass-by-reference>
            <bean-cache>
               <cache-idle-timeout-in-seconds>
                  0
               </cache-idle-timeout-in-seconds>
               <removal-timeout-in-seconds>
                  0
               </removal-timeout-in-seconds>
            </bean-cache>
         </ejb>
         <ejb>
            <ejb-name>InterestCalculator</ejb-name>
            <jndi-name>ejb/InterestCalculator</jndi-name>
            <pass-by-reference>false</pass-by-reference>
            <bean-pool>
               <pool-idle-timeout-in-seconds>
                  0
               </pool-idle-timeout-in-seconds>
            </bean-pool>
         </ejb>
      </enterprise-beans>
   </sun-ejb-jar>
```

エンティティ Beans :

ejb-jar.xml：ejb-jar.xml に備えて、最新の DTD URL を示すように、<!DOCTYPE 定義を変更します。この新しい定義は、次のようになります。

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN'
'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
```

ejb-jar.xml のすべての CMP に対して、1.1 の値を指定した <cmp-version> タグを挿入します。

エンティティ Bean のエントリは、次のようになります。

```
<entity>
      <description>Account CMP entity bean</description>
      <ejb-name>Account</ejb-name>
      <home>com.sun.bank.ejb.entity.AccountHome</home>
      <remote>com.sun.bank.ejb.entity.Account</remote>
      <ejb-class>com.sun.bank.ejb.entity.AccountEJB</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>
      com.sun.bank.ejb.entity.AccountPK
      </prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>1.x</cmp-version>
 <cmp-field>
        <field-name>branchCode</field-name></cmp-field>
 <cmp-field>
       <field-name>accTypeId</field-name></cmp-field>
 <cmp-field>
          <field-name>accBalance</field-name></cmp-field>
 <cmp-field>
          <field-name>custNo</field-name></cmp-field>
 <cmp-field>
          <field-name>accNo</field-name></cmp-field>
</entity>
```

同様に、すべての CMP Beans がこのエントリを保持します。

セッション Beans と同様に、Sun ONE Application server 6.5 の ias-ejb-jar.xml は、Sun ONE Application server 7 の sun-ejb-jar.xml に置き換えられています。これらの 2 つの XML ファイルの DTD は、根本的に異なっているため、移行を行う場合は、ejb-jar.xml と ias-ejb-jar.xml から必要な情報を抽出して、新しい sun-ejb-jar.xml を作成する必要があります。作成プロセスは、以下のとおりです。

1. 最上部に、次のような DOCTYPE 定義を記述した新しい XML ファイルを作成します。

   ```
   <!DOCTYPE sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Sun
   ONE Application Server 7 EJB 2.0//EN'
   'http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2
   _0-0.dtd'>
   ```

   このファイルに sun-ejb-jar.xml という名前を付けて、修正した ejb-jar.xml と一緒に保存します。

2. DOCTYPE 定義から見て分かるように、この XML ファイルのルートタグは sun-ejb-jar です。DTD では、この要素は次のように定義します。

   ```
   <!ELEMENT sun-ejb-jar (security-role-mapping*, enterprise-beans) >
   ```

   security-role-mapping タグは、ejb-jar.xml で宣言されているセキュリティロールをマッピングするためのものです。iBank アプリケーションの場合、ejb-jar.xml ファイルで宣言されているセキュリティがないので、security-role-mapping タグ (任意指定) は飛ばして、enterprise-beans タグに重点を置きます。現在、sun-ejb-jar.xml ファイルは、次のようになっています。

   ```
   <sun-ejb-jar>
      <enterprise-beans>
      </enterprise-beans>
   </sun-ejb-jar>
   ```

   注 : ここでは、ドキュメントのヘッダー部分、すなわち、XML 宣言と DOCTYPE 定義は省略しています。

3. enterprise-beans 要素は、次のように DTD で定義します。

   ```
   <!ELEMENT enterprise-beans (name?, unique-id?, ejb*,
   pm-descriptors?, cmp-resource?)>
   ```

   任意指定の name 要素には、enterprise-beans の正準名が入ります。この要素には、任意の名前を付けることができます。ここでは、このタグは省略します。

unique-id 要素は、Sun ONE Application Server で使用されるもので、アプリケーションの配備時に、Application Server によって自動的に挿入されます。ここでは、このタグは省略します。

ここで着目すべきタグは、*ejb* 要素です。このタグは、単一の EJB の実行時バインドを指定する要素です。これは、次のように DTD で定義します。

```
<!ELEMENT ejb (ejb-name, jndi-name?, ejb-ref*, resource-ref*,
resource-env-ref*, pass-by-reference?, cmp?, principal?,
mdb-connection-factory?, jms-durable-subscription-name?,
jms-max-messages-load?, ior-security-config?,
is-read-only-bean?, refresh-period-in-seconds?, commit-option?,
gen-classes?, bean-pool?, bean-cache?)>
```

この場合、*ejb* 要素に *ejb-name* 要素が含まれます。*ejb-name* 要素には、EJB の正準名が入ります。この名前は、その EJB 用の `ejb-jar.xml` の *ejb-name* 要素内で宣言された名前と同じになります。また、この要素には、EJB の *jndi-name* も含まれます。Sun ONE Application Server 6.5 から 7 で改善された点の 1 つは、Bean の開発者が EJB の *ejb-name* と *jndi-name* を自由に変更できるようになったという柔軟性です。Sun ONE Application Server 6.5 では、EJB の JNDI 名はデフォルトで `ejb/<ejb-name>` に設定されていました。

移行を円滑に行うためには、EJB の jndi-name をそのままの状態にしておき、その他のすべてのリソースについても Sun ONE Application Server 6.5 のときと同じものにしておく必要があります。このため、ここではすべての ejb の *ejb-name* を `ejb/<ejb-name>` と宣言します。

上記のロジックを使用すると、`sun-ejb-jar.xml` は、次のようになります。

```
<sun-ejb-jar>
   <enterprise-beans>
      <ejb>
         <ejb-name>Account</ejb-name>
         <jndi-name>ejb/Account</jndi-name>
      </ejb>
      <ejb> --- </ejb>
      <ejb> --- </ejb>
                        other ejb's
      <ejb> --- </ejb>
      <ejb> --- </ejb>
   </enterprise-beans>
</sun-ejb-jar>
```

4. ejb 要素には、pass-by-reference 要素 `<!ELEMENT pass-by-reference (#PCData)` が含まれます。

   pass-by-reference 要素は、参照渡しセマンティクスの使用を制御します。EJB 仕様では、デフォルトの動作モードとなる参照渡しを必要としています。これは、対応できない操作やより高速なパフォーマンスに向けて、true を設定することができます。これは、閉ざされた環境にある EJB モジュールすべてに適用できます。設定できる値は、true と false です。デフォルト値は false です。

5. CMP エンティティ Beans の場合、cmp 要素を宣言し、EJB1.1 および EJB2.0 の Beans の CMP EntityBean オブジェクトに関する実行時情報を記述します。

   ```
   <!ELEMENT cmp (mapping-properties?, is-one-one-cmp?,
   one-one-finders?)>
   ```

   この *mapping-properties* には、持続性ベンダー固有の O/R マッピングファイルの場所が入ります。*is-one-one-cmp* フィールドは、CMP 1.1 を古い記述子として扱う場合に使用します。CMP 1.1 を古い記述子とする場合、このフィールドに true を指定します。*one-one-finders* には、CMP 1.1 のファインダが入ります。

   このルート要素 *finder* には、メソッド名とクエリパラメータを持つ CMP 1.1 のファインダが入ります。

   ```
   <!ELEMENT finder (method-name, query-params?, query-filter?,
   query-variables?)>
   ```

   method-name 要素には、クエリフィールドのメソッド名が入ります。*query-params* 要素には、CMP 1.1 ファインダのクエリパラメータが入ります。

   query-filter は、CMP 1.1 ファインダのクエリフィルタが入る任意指定の要素です。

   上記の iBank のエントリを作成すると、sun-ejb-jar は、次のようになります。

```
<sun-ejb-jar>
   <enterprise-beans>
      <ejb>
         <ejb-name>Account</ejb-name>
         <jndi-name>ejb/Account</jndi-name>
         <pass-by-reference>false</pass-by-reference>
         <cmp>
           <mapping-properties>
            META-INF/sun-cmp-mappings.xml
           </mapping-properties>
           <is-one-one-cmp>true</is-one-one-cmp>
           <one-one-finders>
```

```
                <finder>
                <method-name>
                   findOrderedAccountsForCustomer
                </method-name>
                <query-params>int custNo</query-params>
                <query-filter>
                   custNo == custNo
                </query-filter>
                </finder>
              </one-one-finders>
            </cmp>
        </ejb>
        <ejb> --- </ejb>
        <ejb> --- </ejb>
                             other ejb's
        <ejb> --- </ejb>
        <ejb> --- </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

Account は、主キー以外のファインダを持つエンティティ Bean だけです。このため、上記のファインダのエントリは、*Account* Bean の場合だけになります。

6. `<!ELEMENT commit-option (#PCDATA)>` では、コミットのオプションを指定します。
7. ejb 要素には、bean-cache 要素も含まれます。

   ```
   <!ELEMENT bean-cache (max-cache-size?,
   is-cache-overflow-allowed?, cache-idle-timeout-in-seconds?,
   removal-timeout-in-seconds?, victim-selection-policy?)>
   ```

   この要素は、ステートフルセッション Bean とエンティティ Bean だけで使用されます。このタグでは、*max-cache-size* によって、キャッシュ内の Beans の最大数を定義します。*cache-idle-timeout-in-seconds* では、ステートフルセッション Bean やエンティティ Bean がアイドル状態でキャッシュ内に存在できる最大時間を指定します。この時間の経過後、Bean はバックアップストアに非活性化されます。この時間は、サーバーが参考にします。*cache-idle-timeout-in-seconds* のデフォルト値は 10 分です。

Bean が非活性化される (バックアップストアでアイドル状態になっている) 時間は、*removal-timeout-in-seconds* パラメータによって制御されます。Bean は *removal-timeout-in-seconds* の値を超えた時間アクセスされなかった場合、バックアップストアから削除されてしまうので、クライアントからアクセスできなくなります。*removal-timeout-in-seconds* のデフォルト値は 60 分です。

上記のエントリが追加されて、`sun-ejb-jar.xml` ファイルは、次のようになります。

```
<sun-ejb-jar>
   <enterprise-beans>
      <ejb>
         <ejb-name>Account</ejb-name>
         <jndi-name>ejb/Account</jndi-name>
         <pass-by-reference>false</pass-by-reference>
         <cmp>
           <mapping-properties>
            META-INF/sun-cmp-mappings.xml
           </mapping-properties>
           <is-one-one-cmp>true</is-one-one-cmp>
           <one-one-finders>
             <finder>
            <method-name>
               findOrderedAccountsForCustomer
            </method-name>
            <query-params>int custNo</query-params>
            <query-filter>
               custNo == custNo
            </query-filter>
             </finder>
          </one-one-finders>
        </cmp>
        <commit-option>C</commit-option>
        <bean-cache>
                <max-cache-size>60</max-cache-size>
```

```
                <cache-idle-timeout-in-seconds>
                   0
                </cache-idle-timeout-in-seconds>
            </bean-cache>
        </ejb>
        <ejb> --- </ejb>
        <ejb> --- </ejb>
                              other ejb's
        <ejb> --- </ejb>
        <ejb> --- </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

8. `<!ELEMENT enterprise-beans (name?, unique-id?, ejb*, pm-descriptors?, cmp-resource?)>`

   pm-descriptors 要素は、`<!ELEMENT pm-descriptors (pm-descriptor+, pm-inuse)>` になります。持続マネージャの記述子には、1 つ以上の pm 記述子が含まれますが、特定の時間にその 1 つが使用されている必要があります。

   pm-descriptor は、エンティティ Bean に関連付けられた持続マネージャのプロパティを示します。*pm-identifier* 要素は、PM 実装を提供したベンダーを示します。さらに、*pm-version* では、使用する PM ベンダー製品のバージョンを指定します。*pm-config* では、使用するベンダー固有の設定ファイルを指定します。*pm-class-generator* では、ベンダー固有の具象クラスジェネレータを指定します。このクラスの名前はベンダーによって異なります。*pm-mapping-factory* では、ベンダー固有のマッピングファクトリを指定します。このクラスの名前はベンダーによって異なります。*pm-insue* では、この特定の PM を使用する必要があるかどうかを指定します。

   cmp-resource 要素には、ejb-jar の CMP Bean を格納するのに使用するデータベースが入ります。`<!ELEMENT cmp-resource (jndi-name, default-resource-principal?)>`

   jndi-name 要素では、JNDI 名の文字列を指定します。*default-resource-principal* 要素には、リソースへのアクセス時に何も指定しない場合に使用する要素名とパスワードがあります。

   `<!ELEMENT default-resource-principal (name, password)>`

最終的に、sun-ejb-jar.xml は、次のような構成になります。

```
<sun-ejb-jar>
   <enterprise-beans>
      <ejb>
         <ejb-name>Account</ejb-name>
         <jndi-name>ejb/Account</jndi-name>
         <pass-by-reference>false</pass-by-reference>
         <cmp>
           <mapping-properties>
            META-INF/sun-cmp-mappings.xml
           </mapping-properties>
           <is-one-one-cmp>true</is-one-one-cmp>
           <one-one-finders>
             <finder>
            <method-name>
               findOrderedAccountsForCustomer
            </method-name>
            <query-params>int custNo</query-params>
            <query-filter>
               custNo == custNo
            </query-filter>
             </finder>
           </one-one-finders>
         </cmp>
         <commit-option>C</commit-option>
         <bean-cache>
                 <max-cache-size>60</max-cache-size>
            <cache-idle-timeout-in-seconds>
               0
```

```
            </cache-idle-timeout-in-seconds>
         </bean-cache>
      </ejb>

      <ejb> --- </ejb>
      <ejb> --- </ejb>
                          other ejb's
      <ejb> --- </ejb>
      <ejb> --- </ejb>
      <pm-descriptors>
         <pm-descriptor>
         <pm-identifier>IPLANET</pm-identifier>
         <pm-version>1.0</pm-version>
         <pm-class-generator>
            com.iplanet.ias.persistence.
            internal.ejb.ejbc.JDOCodeGenerator
         </pm-class-generator>
         <pm-mapping-factory>
         com.iplanet.ias.cmp.NullFactory
         </pm-mapping-factory>
         </pm-descriptor>
         <pm-inuse>
         <pm-identifier>IPLANET</pm-identifier>
         <pm-version>1.0</pm-version></pm-inuse>
      </pm-descriptors>
      <cmp-resource>
      <jndi-name>jdo/pmf</jndi-name>
      </cmp-resource>
   </enterprise-beans>
</sun-ejb-jar>
```

Sun ONE Application Server 7 のインストールディレクトリ内の bin ディレクトリに格納されている *capture-schema* ユーティリティを使用して、dbschema を生成します。bin ディレクトリに格納されている capture-schema.bat ファイルを実行して、データベース URL、ユーザー名、パスワードについて有効な値を指定した後、スキーマを生成する必要がある表を指定します。デフォルトでは、アプリケーションで使用されるすべての表について、スキーマを生成する必要があります。iBank の場合、スキーマを生成する必要がある表は 6 つです。このスキーマファイルには、`myschema.dbschema` という名前を付けます。iBank で使用される表を以下に示します。

```
ACCOUNT

ACCOUNT_TYPE

BRANCH

CUSTOMER

TRANSACTION_HISTORY

TRANSACTION_TYPE
```

この myschema.dbschema ファイルをエンティティ Beans の META-INF フォルダ上に置きます。

`<ejb-name>-ias-cmp.xml` : Sun ONE Application Server 6.0/6.5 のすべての `<ejb-name>-ias-cmp.xml` ファイルを 1 つの `sun-cmp-mappings.xml` ファイルに置き換えます。このファイルは、1 つ以上の Beans のセットを特定の db スキーマの表およびカラムにマップします。これらの 2 つの XML ファイルの DTD は、根本的に異なっているため、移行を行う場合は、以下に示す手順に従って、新しいファイルを実際に作成する必要があります。

1. 最上部に、次のような DOCTYPE 定義を記述した新しい XML ファイルを作成します。

   ```
   <!DOCTYPE sun-cmp-mappings PUBLIC '-//Sun Microsystems, Inc.//DTD
   Sun ONE Application Server 7 OR Mapping //EN'
   'http://www.sun.com/software/sunone/appserver/dtds/sun-cmp_mappi
   ng_1_0.dtd'>
   ```

   `sun-cmp-mappings.xml` という名前で、このファイルを保存します。

2. DOCTYPE 定義から見て分かるように、この XML ファイルのルートタグは sun-cmp-mappings です。DTD では、この要素は次のように定義します。

   ```
   <!ELEMENT sun-cmp-mappings ( sun-cmp-mapping+ ) >
   ```

   sun-cmp-mapping 要素は、次のようになります。

   ```
   <!ELEMENT sun-cmp-mapping ( schema, entity-mapping+) >
   ```

   この場合、schema 要素は、スキーマファイルへのパス名になります。

cmp Bean は、名前、主表、1 つ以上のフィールド、0 個以上の関係、0 個以上の二次表、および整合性チェック用フラグを持ちます。entity-mapping 要素は、次のような要素を保持します。

```
<!ELEMENT entity-mapping (ejb-name, table-name,
cmp-field-mapping+, cmr-field-mapping*, secondary-table*,
consistency?)>
```

ejb-name 要素は、標準 EJB-jar DTD の EJB 名です。*table-name* 要素は、データベース表の名前です。*cmp-field-mapping* は、フィールド、すなわち、*cmr-field mapping* にマップする 1 つ以上のカラムを持ちます。cmr フィールドは、関係を定義する名前および 1 組以上のカラムペアを持ちます。*secondary-table* 要素は、使用する二次表です。iBank では、二次表は使用しません。

現在、Account エンティティ Bean のエントリを持つ `sun-cmp-mappings.xml` ファイルは、次のようになっています。

```
<sun-cmp-mapping>
  <schema>mySchema</schema>
  <entity-mapping>
    <ejb-name>Account</ejb-name>
    <table-name>ACCOUNT</table-name>
    <cmp-field-mapping>
       <field-name>custNo</field-name>
       <column-name>CUST_NO</column-name>
    </cmp-field-mapping>
    <cmp-field-mapping>
       <field-name>branchCode</field-name>
       <column-name>BRANCH_CODE</column-name>
    </cmp-field-mapping>
    <cmp-field-mapping>
       <field-name>accTypeId</field-name>
       <column-name>ACCTYPE_ID</column-name>
    </cmp-field-mapping>
    <cmp-field-mapping>
```

```
            <field-name>accNo</field-name>
            <column-name>ACC_NO</column-name>
        </cmp-field-mapping>
        <cmp-field-mapping>
            <field-name>accBalance</field-name>
            <column-name>ACC_BALANCE</column-name>
        </cmp-field-mapping>
    </entity-mapping>
</sun-cmp-mapping>
```

注 : ここでは、ドキュメントのヘッダー部分、すなわち、XML 宣言と DOCTYPE 定義は省略しています。

すべての CMP エンティティ Beans について、エントリを作成する必要があります。

上記の変更内容については、このガイドに付属する `iBankWithCMP1.1.zip` ファイルで参照することができます。

配備用アプリケーションのアセンブル

Sun ONE Application Server 7 では、主に J2EE モデルがサポートされます。このモデルでは、アプリケーションがエンタープライズアーカイブ (EAR) ファイル (拡張子は .ear) の形式でパッケージ化されます。アプリケーションはさらに J2EE モジュールの集まりに分割され、EJB 用の Java アーカイブ (拡張子が .jar の JAR ファイル) とサーブレットおよび JSP 用の Web アーカイブ (拡張子が .war の WAR ファイル) にパッケージ化されます。

このため、すべての JSP とサーブレットが WAR ファイルに、またすべての EJB が JAR ファイルに、それぞれパッケージ化され、最終的には WAR ファイルと JAR ファイルが配備記述子と一緒に EAR ファイルにパッケージ化されます。この EAR ファイルは、配備が可能なコンポーネントです。

asadmin ユーティリティを使用した Sun ONE Application Server 7 での iBank アプリケーションの配備

最後の作業は、Sun ONE Application Server 7 のインスタンスでのアプリケーションの配備です。アプリケーションを配備するプロセスは、以下に示すとおりです。

Sun ONE Application Server 7 の *asadmin* には、「ヘルプ」メニューからアクセスできる配備に関するヘルプ項目が含まれています。

コマンド行ユーティリティ *asadmin* を起動するには、Windows の場合は `asadmin.bat` ファイルを、また Solaris の場合は `asadmin` ファイルをそれぞれ実行します。これらのファイルは、Sun ONE Application Server 7 のインストールディレクトリ内の bin ディレクトリ (`<Install_dir>/AppServer7/appserv/bin` など) に格納されています。

asadmin プロンプトから、次のような配備用のコマンドを入力します。

asadmin> deploy -u username -w password -H hostname -p adminport [--type application | ejb | web | client | connector] [--contextroot contextroot] [--force=true] [--name component-name] [--upload=true] [--instance instancename] filepath

サーバーインスタンスを再起動してから、ブラウザに *http://<machine_name>:<port_number>/IBank* という URL を入力して、アプリケーションをテストします。有効なユーザー名とパスワード、たとえば「jatkins」というユーザー名と「Monday」というパスワードを使用してテストを行います。これによって、iBank アプリケーションのメインメニューページが表示されます。

Sun ONE Studio for Java 4.0 を使用した iBank の移行

「iBank」という名前で定義を行ったサンプルアプリケーションは、基本的なオンラインバンキングサービスをシミュレートし、次のような機能を実行できます。

- オンラインバンキングサービスへのログオン
- 個人データおよび支店データの表示および編集
- 清算勘定を示す預金口座の概要表示
- 個々のトランザクション履歴を表示するために、口座ごとにドリルダウンする機能
- 口座から口座へ資金をオンラインで移動できる振替サービス
- 一定の元本および年利回り率で複数年に及ぶ複利配当額の見積り

iBank アプリケーションを移行する場合に実行する主な手順を以下に示します。

- この移行で、最初に行うべき最も重要な要件は、Sun ONE Application Server 7 と Sun ONE Studio のインストールです。
- ローカルディレクトリ内に zip 形式で格納されているアプリケーションを抽出します。

  iBank アプリケーションのソース (`iBank65.zip`) は、移行サイト http://www.sun.com/migration/sunonetools.html から入手できます。`iBank65.zip` ファイルを解凍すると、次のようなディレクトリ構造が作成されます。

  このディレクトリ構造内には、`Docroot` 、`SessionContent` 、`EntityContent` 、および `Scripts` の 4 つのサブディレクトリが作成されます。

  - `Docroot` には、HTML ファイル、JSP ファイル、および画像ファイルがそのルートに含まれます。また、サーブレットや EJB などのソースファイルも、パッケージ構造 com.sun.bank.* に従って、サブフォルダ `WEB-INF¥` クラスに含まれます。WAR ファイルは、このディレクトリの内容から生成されます。
  - `SessionContent` には、パッケージ構造 com.sun.bank.ejb.session に従って、セッション Beans のソースコードが含まれます。このディレクトリは、セッション Beans の EJB モジュールを構成します。
    1. `EntityContent` には、パッケージ構造 com.sun.bank.ejb.entity に従って、エンティティ Beans が含まれます。このディレクトリは、エンティティ Beans の EJB モジュールを構成します。
  - `Scripts` には、データベースセットアップ用の SQL スクリプトが含まれます。

- Scripts フォルダに入っている SQL スクリプトを実行して、iBank アプリケーションのスキーマをセットアップします。これらのスクリプトは、Oracle データベース用です。これらのスクリプトによって、ユーザーの作成、表の作成、および表へのデータの挿入が行われます。次の順序で、スクリプトを実行します。
  - 01_iBank_CreateUser.sql
  - 02_iBank_CreateTables.sql
  - 03_iBank_InsertData.sql

  手動移行の場合は、次の手順が必要です。

  a. サーブレット、JSP、および JSP カスタムタグライブラリの移行

  b. セッション Beans の移行

  c. エンティティ Beans の移行

  d. JDBC コードの移行

  これらの手順は手動で行う必要があり、必要に応じて、以降の節で手順を説明します。移行ツールをオプションとして使用する場合、この時点で実行する必要があります。手動による方法を実行する場合は、以降の節で説明するように、変更を行う必要があります。
- サンプルアプリケーションの「iBank」のアセンブルと配備を行うために、Sun ONE Studio を用意します。

  Sun ONE Studio を起動するには、<Sun ONE App Server ROOT>/<AppServ>/<SUN ONE STUDIO FOR JAVA_ROOT>/bin ディレクトリに格納されている runide.exe ファイル (Solaris の場合は、runide.sh ファイル) を実行します。

  (注 : 次の手順を開始する前に、Sun ONE Application Server 7 を起動しておく必要があります)
  - 「エクスプローラ」ウィンドウを開きます。
  - 「実行時」タブをクリックします。
  - 「サーバーレジストリ」をクリックします。
  - 「インストールされたサーバー」をクリックします。
  - 「Sun ONE Application Server」を選択します。
  - 「Sun ONE Application Server」を右クリックし、「管理サーバーを追加」を選択して、管理サーバーをセットアップします。
  - ホストの詳細 (ローカルマシン名)、ポート番号 (デフォルトは 4848)、ユーザー名、およびパスワードを入力します。

- 管理サーバーのセットアップ後、管理サーバーをクリックして、サーバーインスタンスをインストールします。
- このサーバーインスタンスをデフォルトサーバーとして設定するには、サーバーインスタンスを右クリックし、そのインスタンスをデフォルトとして設定するためのオプションを選択します。

- 「Sun ONE Studio for Java での Web アプリケーションモジュールの作成」に記載される指示に従って、Web モジュールを作成します。
- アプリケーションの移行で、移行ツールを使用しない場合は、EJB を手動で移行します。手動移行については、「EJB の移行」を参照してください。この手順を実行するには、Sun ONE Studio で EJB のソースコードを開いて、修正します。
- アプリケーションの移行で、移行ツールを使用しない場合は、「JDBC コードの移行」に従って、JDBC コードを移行します。
- iBank アプリケーションは、CMP 1.1 のエンティティ Bean を持つため、移行ツールを使用してアプリケーションを移行しない場合は、「CMP エンティティ EJB の移行」に記載される手動移行の手順に従って、これらのエンティティ Beans を CMP 2.0 に変換させる必要があります。

  ツールを使用してアプリケーションを移行する場合は、Account エンティティ Bean のように、コードで列挙 (Enumeration) を使用しているエンティティ Bean を除いたすべてのエンティティ Beans が移行されます。このコードは、「CMP エンティティ EJB の移行」に記載される指示に従って、手動で変更する必要があります。CMP を 1.1 から 2.0 に変換する変更作業の例については、「CMP エンティティ EJB の 1.1 から 2.0 への変換」を参照してください。
- 「Sun ONE Studio for Java での EJB モジュールの作成」に記載される指示に従って、エンティティ Beans とセッション Beans について、別々の EJB モジュールを作成します。
- 「Sun ONE Studio for Java でのエンタープライズアプリケーションの作成」に記載される指示に従って、エンタープライズアプリケーションを作成します。これには、Web モジュールと EJB モジュールも含まれます。この手順を実行すると、最終的に配備が可能な .ear ファイルが作成されます。
- 「Sun ONE Application Server 7 でのアプリケーションの配備」に記載される指示に従って、Sun ONE Application server 7 で .ear ファイルを配備します。

Sun ONE Studio for Java での Web アプリケーションモジュールの作成

Sun ONE Studio for Java で Web アプリケーションモジュールを作成するには、以下の手順を実行します。

1. 「ファイルシステム」を右クリックし、マウントのオプションを選択して、Sun ONE Studio for Java の「エクスプローラのファイルシステム」ウィンドウ内で、ソースファイル、すなわち「Docroot」が格納されているディレクトリをマウントします。
2. ソースファイルが格納されているルートディレクトリ構造で、Web モジュール用に、「WarContent」などの空のディレクトリを作成します。
3. 「ファイルシステム」を右クリックし、マウントのオプションを選択して、Sun ONE Studio for Java の「エクスプローラのファイルシステム」ウィンドウ内に、新しく作成したディレクトリ「WarContent」をマウントします。
4. ソースファイルディレクトリ構造に、「EntityContent」や「SessionContent」など、EJB を含むその他のディレクトリをマウントします。
5. フォルダ名を右クリックし、Web モジュールへの変換オプションがあるツールを選択して、ファイルシステム (WarContent) を Web モジュールに変換します。
6. ソースの JSP、HTML、および画像ファイルを Web アプリケーションルートに、すなわち、「Docroot」ディレクトリから「WarContent」ディレクトリにコピーします。
7. サーブレットと補助クラスソースを `WEB-INF/classes` ディレクトリにコピーします。すなわち、「Docroot」ディレクトリ内のサブフォルダ「com」を「WarContent」ディレクトリの `WEB-INF/classes` ディレクトリにコピーします。
8. 「Docroot」ディレクトリの `WEB-INF` にあるタグライブラリを、「WarContent」ディレクトリの `WEB-INF` にコピーします。
9. ソースコードを Sun ONE Application Server 7 に移行する必要がある場合 (移行ツールを使用して変更しない場合) は、以下の手順に従ってソースコードを編集します。
   - 変更する必要がある JSP を見つけます。
   - カスタム JSP タグがアプリケーションで使用されているかどうかを調べます。
   - ファイルを右クリックし、「開く」オプションを選択して、Sun ONE Studio で指定した JSP コードを開きます。
   - 「Java Server Pages および JSP カスタムタグライブラリの移行」に記載される手順に従って、ソースを修正します。

- 同様に、「サーブレットの移行」に記載される手順に従って、サーブレットを移行します。

10. アプリケーションをアセンブルし、(`WEB-INF/` ディレクトリ内の) 配備記述子 `web.xml` に必要な情報を入力します。`web.xml` ファイルをクリックし、このファイルのプロパティを編集します。すなわち、このアセンブリフェーズでは、各サーブレット、JSP ページ、および JSP タグライブラリの他に、Web アプリケーションで使用される EJB またはデータソース参照も設定します。

次の画面ショットは、Sun ONE Studio for Java を使用して、このアセンブリフェーズを実行しているところです。

サーブレットの設定

Web モジュールで、Web をクリックし、「プロパティ」ウィンドウを表示します。

`web.xml` の「プロパティ」ウィンドウ内の「配備」タブをクリックします。次に、サーブレットを設定するために、「サーブレット」をクリックします。

プロパティエディタが表示されるので、「追加」ボタンをクリックして、新しいサーブレットを追加します。

Web アプリケーションの各サーブレットについて、「ブラウズ」ボタンをクリックして、サーブレット名、サーブレットの実装クラスの完全名を指定します。また、「マッピング」をクリックして、サーブレットのマッピング要素を、さらに、初期パラメータを指定します。

サーブレットの設定

iBank アプリケーションのサーブレットとそれらのマッピングについてのリストを以下に示します。

サーブレットとマッピング

サーブレット名	表示名	マッピング
LoginServlet	LoginServlet	/CheckLogin
CheckTransferServlet	CheckTransferServlet	/CheckTransfer
CustomerProfileServlet	CustomerProfileServlet	/CustomerProfile
DataSourceTestServlet	DataSourceTestServlet	/DataSourceTest
HelloWorldServlet	HelloWorldServlet	/HelloWorld
LookUpDataSourceTestServlet	LookUpDataSourceTestServlet	/LookUpDataSourceTest
ProjectEarningsServlet	ProjectEarningsServlet	/ProjectEarnings

サーブレットとマッピング (続き)

サーブレット名	表示名	マッピング
ShowAccountSummaryServlet	ShowAccountSummaryServlet	/ShowAccountSummary
TestContextServlet	TestContextServlet	/TestContext
TransferFundsServlet	TransferFundsServlet	/TransferFunds
UpdateCustomerDetailsServlet	UpdateCustomerDetailsServlet	/UpdateCustomerDetails

上記のサーブレットはすべて、`web.xml` にすべてのサーブレットのエントリが保持されるように設定する必要があります。

最終的に、「配備」タブに 11 個のサーブレットのマッピングとサーブレットが表示されます。

JSP タグライブラリの設定

`web.xml` の「Properties」ウィンドウ内の「Deployment」タブをクリックします。「Tag Libraries」をクリックして、タグライブラリを設定します。

Web アプリケーションの配備記述子で、JSP タグライブラリを定義するには、ライブラリの URI (JSP ページがライブラリにアクセスするための識別子) とライブラリの配備記述子 (.tld ファイル) へのパスを指定します。

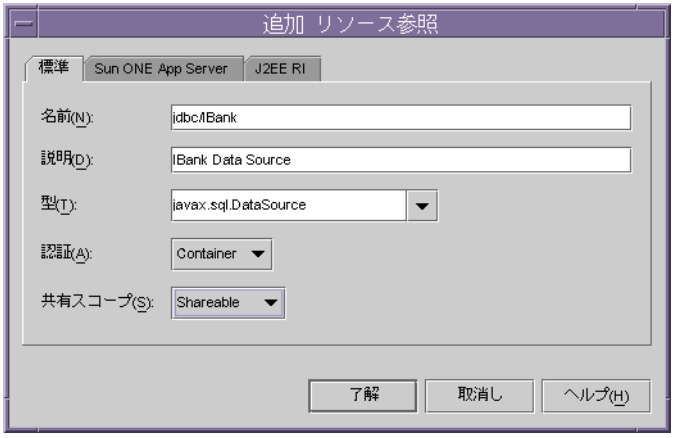
iBank には、1 つの JSP タグライブラリ `TMBHisto.tld` があります。この配備記述子は、`WEB-INF` に格納されます。次のようなエントリを作成する必要があります。

タグライブラリの設定

リソース参照の追加

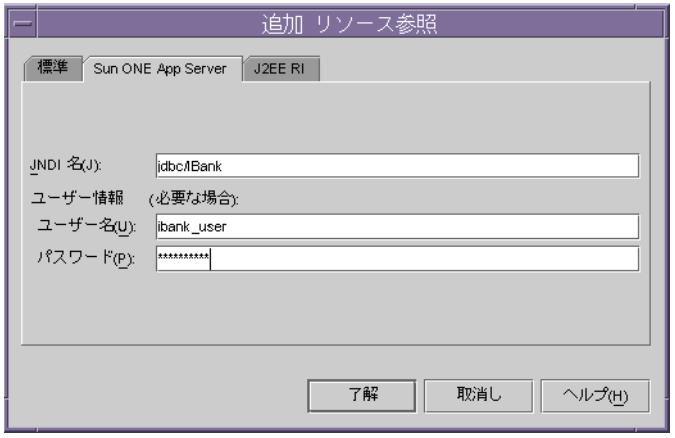
`web.xml` の「Properties」ウィンドウ内の「References」タブをクリックします。「Resource Reference」をクリックして、新しいリソースを追加します。次の画面ショットは、iBank に新しいデータソースのリソース `jdbc/iBank` を追加しているところです。

リソース参照の追加



「Sun ONE App Server」タブをクリックして、「JNDI 名」に「`jdbc/IBank`」を設定し、さらに、使用するデータベーススキーマに応じたユーザー名とパスワードも設定します。

Sun ONE Application Server 用のリソース参照エントリの追加



コンテキストパラメータの追加

JNDI 名のコンテキストパラメータのエントリを追加し、iBank データソースを検索します。

次の画面ショットは、コンテキストパラメータのエントリを示しています。これは、`web.xml` の「配備」タブで、「プロパティ」ウィンドウ内の「コンテキストパラメータ」をクリックすることで表示されます。

コンテキストパラメータの追加

Welcome ファイルの指定

「開始ファイル」をクリックして、「プロパティ」ウィンドウ内の Welcome ファイルを指定します。

iBank の場合、`index.jsp` が開始ファイルです。これについて説明します。

CMP エンティティ EJB の 1.1 から 2.0 への変換

「CMP エンティティ EJB の移行」に記載される手動プロセスを参照して、CMP 1.1 から CMP 2.0 に変換した Account エンティティ Bean の例を以下に示します。

Account Bean の関連ファイルには、次のようなものがあります。

`Account.java`

`AccountEJB.java`

`AccountHome.java`

`AccountPK.java`

実行する変更の詳細について、以下に示します。

- Account.java :

  主キーのセッターをコメントアウトする以外は、コードを大幅に変更する必要はありません。その他のセッターは、そのままの状態にしておきます。

  修正前のコードを以下に示します。

```
public String getBranchCode()
```

```
   throws RemoteException;
public void setBranchCode(String branchCode)
   throws RemoteException;
public String getAccNo()
   throws RemoteException;
public void setAccNo(String accNo)
   throws RemoteException;
      ------
      ------
-----other getters and setters----
```

branchCode や accNo など、主キーのセッターをコメントアウトした後のコードは、以下のようになります。

```
public String getBranchCode()
   throws RemoteException;

/* public void setBranchCode(String branchCode)
   throws RemoteException; */

public String getAccNo()
   throws RemoteException;

/* public void setAccNo(String accNo)
   throws RemoteException; */
      ------
      ------
-----other getters and setters----
```

- AccountEJB.java :

  Bean クラスに取り入れられる変更は、以下のとおりです。

- Bean クラス宣言の先頭に、キーワード「abstract」を付けます。

  修正前 :

```
public class AccountEJB implements EntityBean
{
--
--
}
```

  修正後 :

```
public abstract class AccountEJB implements EntityBean
{
--
--
}
```

- すべての cmp フィールドをコメントアウトし、accessor メソッドの前にキーワード「abstract」を付けます。メソッドのコード行がコメントになるので、メソッドの後ろにセミコロンを付けます。つまり、下記の「修正前」のコードを「修正後」のコードに置き換えます。

  修正前 :

```
   public String branchCode;
   public String accNo;
   public int custNo;
   public String accTypeId;
   public double accBalance;
   public String accTypeDesc;
   public double accTypeInterestRate;
   private EntityContext context;

   public String getBranchCode() {
```

```
        return(branchCode);
}

public void setBranchCode(String branchCode) {
this.branchCode = branchCode;
}

public String getAccNo() {
        return(accNo);
}

public void setAccNo(String accNo) {
this.accNo = accNo;
}

public int getCustNo() {
        return(custNo);
}

public void setCustNo(int custNo) {
        this.custNo = custNo;
}

public String getAccTypeId() {
        return(accTypeId);
}

public void setAccTypeId(String accTypeId) {
        this.accTypeId = accTypeId;
}

public BigDecimal getAccBalance() {
```

```
        return new BigDecimal(accBalance);
    }

    public void setAccBalance(BigDecimal accBalance) {
        this.accBalance = accBalance.doubleValue();
    }
```

修正後 :

```
    private EntityContext context;
    public abstract void setBranchCode(String branchCode);
    public abstract String getBranchCode();
    public abstract void setAccNo(String accNo);
    public abstract String getAccNo();
    public abstract void setCustNo(int custNo);
    public abstract int getCustNo();
    public abstract void setAccTypeId(String accTypeId);
    public abstract String getAccTypeId();
    public abstract void setAccBalance(BigDecimal accBalance);
    public abstract BigDecimal getAccBalance();
```

- すべての ejbCreate() メソッド本体 (複数の ejbCreate が存在する場合もある) を読み込みます。「<cmp-field>= 値またはローカル変数」のパターンを探して、それを「abstract ミュテータメソッド名 (値またはローカル変数)」の形式に置き換えます。したがって、コードの変更は、以下のようになります。

修正前 :

```
    public void setEntityContext(EntityContext ec) {
        context = ec;
    }
```

```
    public void unsetEntityContext() {
            this.context = null;
    }

    public void ejbActivate() {

            this.branchCode =
((com.sun.bank.ejb.entity.AccountPK)

context.getPrimaryKey()).branchCode;

            this.accNo = ((com.sun.bank.ejb.entity.AccountPK)
                    context.getPrimaryKey()).accNo;
    }

    public void ejbPassivate() {
    }

    public void ejbLoad() {
    }

    public void ejbStore() {
    }

    public AccountPK ejbCreate(String branchCode,
            String accNo, int custNo, String accTypeId,
            BigDecimal accBalance) {
            this.branchCode = branchCode;
            this.accNo      = accNo;
            this.custNo     = custNo;
            this.accTypeId = accTypeId;
            this.accBalance = accBalance.doubleValue();
            return null;
```

```
    }

    public void ejbPostCreate(String branchCode,
           String accNo, int custNo, String accTypeId,
           BigDecimal accBalance) {
    }

    public void ejbRemove() {
    }
```

修正後 :

```
    public void setEntityContext(EntityContext ec) {
           context = ec;
    }

    public void unsetEntityContext() {
           this.context = null;
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbLoad() {
    }

    public void ejbStore() {
    }
```

```
public AccountPK ejbCreate(String branchCode,
       String accNo, int custNo, String accTypeId,
       BigDecimal accBalance) {
       setBranchCode(branchCode);
       setAccNo(accNo);
       setCustNo(custNo);
       setAccTypeId(accTypeId);
       setAccBalance(accBalance);
       return null;
}

public void ejbPostCreate(String branchCode,
       String accNo, int custNo, String accTypeId,
       BigDecimal accBalance) {
}

public void ejbRemove() {
}
```

- AccountPK.java

  このファイルは、変更する必要がありません。

- AccountHome.java

  Bean のホームインタフェースでは、検索メソッドの戻り値の型が java.util.Enumeration のときにだけ、変更が必要となります。Account Bean の場合、ホームインタフェースは戻り値の型が列挙 (Enumeration) であるファインダ findOrderedAccountsForCustomer を保持します。このような場合、戻り値の型を Collection に変更する必要があります。また、この変更によって影響を受けるコード、すなわち、この検索メソッドを使用するセッション Bean のコードも、Collection でこの検索メソッドの結果を受け付けられるように、変更する必要があります。

  ホームインタフェースで行う修正を以下に示します。

  修正前 :

  ```
  public interface AccountHome extends EJBHome
  {
  ```

```
   public Account findByPrimaryKey(AccountPK key)
          throws FinderException, RemoteException;


   public Enumeration findOrderedAccountsForCustomer(int
custNo)
          throws FinderException, RemoteException;
}
```

修正後 :

```
public interface AccountHome extends EJBHome
{
   public Account findByPrimaryKey(AccountPK key)
          throws FinderException, RemoteException;


   public Collection findOrderedAccountsForCustomer(int
custNo)
          throws FinderException, RemoteException;
}
```

上記の変更の結果、この検索メソッドにアクセスするセッション Bean の BankTeller についても、Collection で検索メソッドの結果を受け付けられるように変更する必要があります。

次の抜粋コードは、`BankTellerEJB.java` への変更内容を示しています。

検索メソッド *findOrderedAccountsForCustomer* を使用するメソッド getAccountSummary について見てみます。

修正前 :

```
public AccountSummary getAccountSummary()
throws EJBException
{
    int custNo            = 0;
    Enumeration   accEnum = null;
    AccountSummary accSum = new AccountSummary();
    -----
```

```
    ----
    try
    {
      AccountHome home=(AccountHome) PortableRemoteObject.
          narrow(accHomeHandle.getEJBHome(),
AccountHome.class);
      AccountTypeHome accTypeHome = (AccountTypeHome)
PortableRemoteObject.narrow(accTypeHomeHandle.getEJBHome(),
AccountTypeHome.class);
      accEnum  = (Enumeration) home.
            findOrderedAccountsForCustomer(this.custNo);
      AccountTypePK accTypePK  = new AccountTypePK();
      Account      accRef      = null;
      AccountType  accTypeRef  = null;
      String       accTypeDesc = null;
      int i = 0;
      while(accEnum.hasMoreElements())
      {
        i++;
        accRef = (Account) accEnum.nextElement();
        accTypePK.accTypeId = accRef.getAccTypeId();
        accTypeRef  = (AccountType) PortableRemoteObject.

narrow(accTypeHome.findByPrimaryKey(accTypePK),
                AccountType.class);
        accTypeDesc =  accTypeRef.getAccTypeDesc();
        accSum.addElement(
                          accRef.getBranchCode(),
                          accRef.getAccNo(),
                          accRef.getAccBalance(),
                          accTypeDesc
                         );
      }
```

```
        }
       -----
       ----
    }
```

修正後 :

```
    public AccountSummary getAccountSummary()
    throws EJBException
    {
        int custNo             = 0;
        //Enumeration   accEnum = null;
        Collection   accEnum = null;
        AccountSummary accSum = new AccountSummary();
       ---
       ---

        try
        {
          AccountHome home  = (AccountHome) PortableRemoteObject.
         narrow(accHomeHandle.getEJBHome(), AccountHome.class);
          AccountTypeHome accTypeHome = (AccountTypeHome)
              PortableRemoteObject.narrow(accTypeHomeHandle.
              GetEJBHome(), AccountTypeHome.class);
        //  accEnum  = (Enumeration) home.
        //  findOrderedAccountsForCustomer(this.custNo);
          accEnum  = (Collection) home.
                findOrderedAccountsForCustomer(this.custNo);
          AccountTypePK accTypePK  = new AccountTypePK();
          Account      accRef      = null;
          AccountType  accTypeRef  = null;
          String       accTypeDesc = null;
          int i = 0;
```

```
    Iterator iterator = accEnum.iterator();
     //  while(accEnum.hasMoreElements())
       while(iterator.hasNext())
       {
         i++;
        // accRef = (Account) accEnum.nextElement();
    accRef = (Account) PortableRemoteObject.
                narrow(iterator.next(), Account.class);
         accTypePK.accTypeId = accRef.getAccTypeId();
         accTypeRef  = (AccountType) PortableRemoteObject.

narrow(accTypeHome.findByPrimaryKey(accTypePK),
                AccountType.class);
         accTypeDesc =  accTypeRef.getAccTypeDesc();
         accSum.addElement(
                           accRef.getBranchCode(),
                           accRef.getAccNo(),
                           accRef.getAccBalance(),
                           accTypeDesc
                          );
       }
     }
    ----
    ----
  }
```

Sun ONE Studio for Java での EJB モジュールの作成

ここでは、既存のソースファイルを使用して、Sun ONE Studio for Java で EJB モジュールを作成する方法について説明します。

セッション Beans 用のモジュールの作成

1. セッション Beans 用の SessionContent ディレクトリには、次のようなものが含まれます。

   次のセッション Beans の Bean クラスとインタフェースがあります。

   BankTeller

   InterestCalculater

   この他に、例外クラスも含まれます。

2. 既存のソースファイルから、新しい EJB を作成します。

   Sun ONE Studio for Java では、既存のソースファイルから EJB を作成することができます。

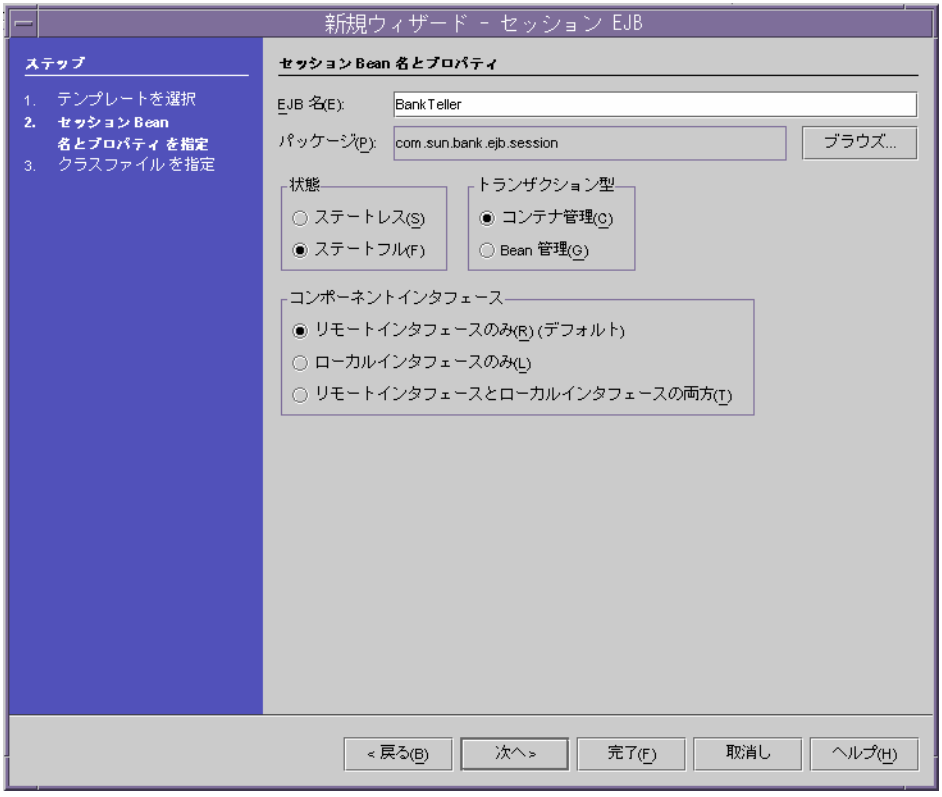
   マウントされたディレクトリ SessionContent を選択し、パッケージ session が表示されるまで、そのサブフォルダを展開します。session を右クリックし、新しい J2EE のオプションを選択してから、「セッション EJB」をクリックすると、新しい EJB ウィザードが表示されます。

   EJB の主な特性 (セッション、ステートフル、またはステートレスなど) を指定し、EJB の名前とパッケージを定義した後で、既存のソースファイルと EJB の別のコンポーネント (実装クラス、ホームインタフェース、およびリモートインタフェース) を一致させます。既存のソースファイルと一致させるには、ダイアログボックスの「Modify」ボタンを使用し、「Select an existing source file」を選択します。

   セッション Beans はすべて同じ方法で作成する必要があります。

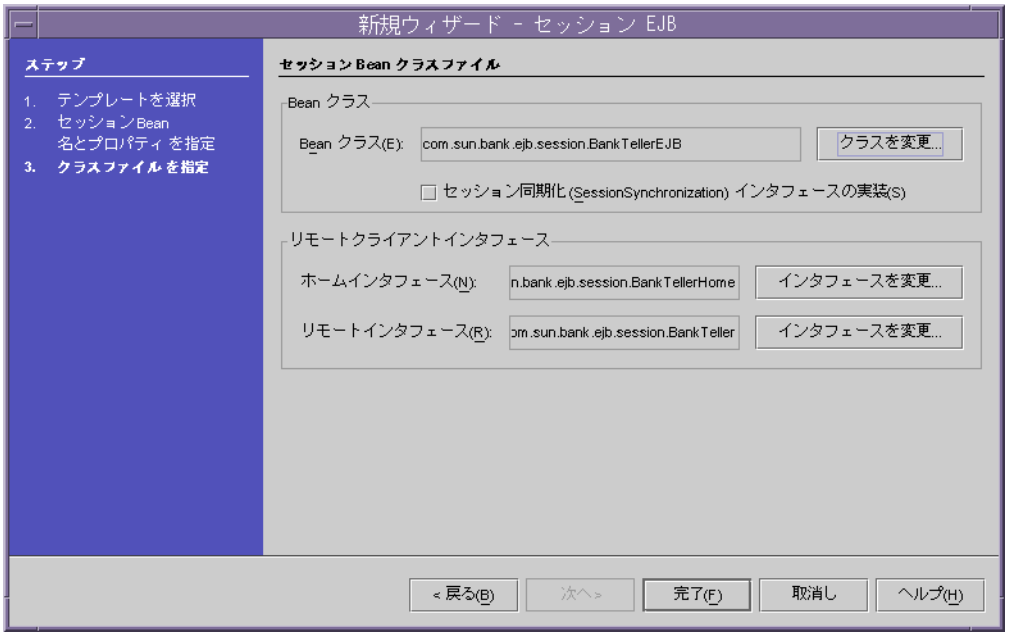
   次の画面ショットは、ステートフルセッション Bean となるセッション Bean *BankTeller* を作成しているところです。このように、指定している状態がステートフルであるのに対し、*InterestCalculator* セッション Bean はステートレスになるため、*InterestCalculator* Bean を作成する間は、指定する状態をステートレスにする必要があります。「Browse」ボタンをクリックして、パッケージを指定します。

新しいセッション Bean の作成



次の画面ショット (「次へ」をクリックした後の画面) は、Bean クラス、ホームインタフェース、およびリモートインタフェースを指定しているところです。「インタフェースを変更」ボタンをクリックし、既存のクラスを使用するオプションを選択すると、選択が可能な既存ファイルが表示されます。

Bean クラス、ホームインタフェース、およびリモートインタフェースの指定

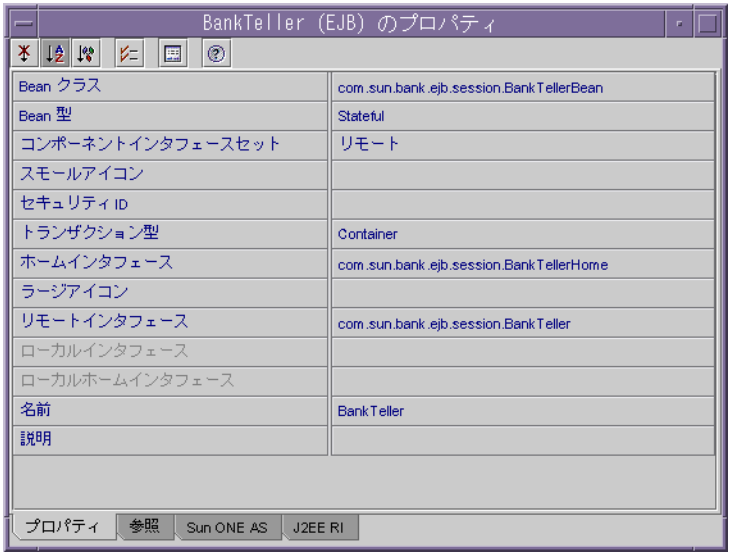


同じ方法で、*InterestCalculator* セッション Bean を作成します。

3. EJB のプロパティを編集します。

   EJB のプロパティを編集すると、EJB リソース参照を宣言し、EJB の環境エントリを指定することができます。

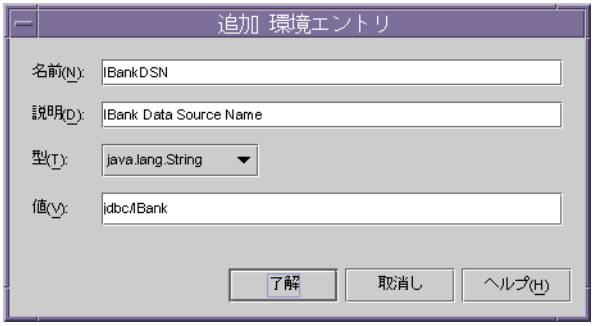
BankTeller セッション Bean の「プロパティ」ウィンドウ



次の画面ショットは、BankTeller セッション Bean の環境エントリを宣言しているところです。InterestCalculator Bean では、このエントリは必要ありません。

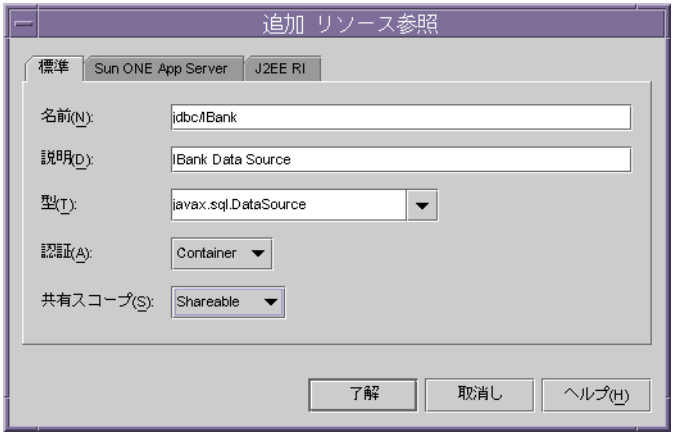
「参照」タブの「環境エントリ」をクリックしてから、「追加」をクリックして、DSN の新しいエントリを追加します。

BankTeller セッション Bean への環境エントリの追加



BankTeller セッション Bean の「プロパティ」ウィンドウ内の「参照」のタブで、「リソース参照」をクリックして、新しいリソースを追加します。次の画面ショットは、iBank に新しいデータソースのリソース `jdbc/iBank` を追加しているところです。

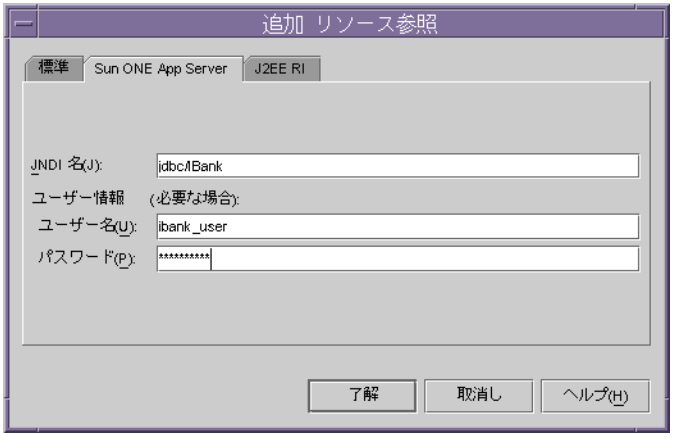
リソース参照の追加



「Sun ONE App Server」タブをクリックして、「JNDI 名」に `jdbc/iBank` を設定し、さらに、使用するデータベーススキーマに応じたユーザー名とパスワードも設定します。

InterestCalculator Bean では、このエントリは必要ありません。

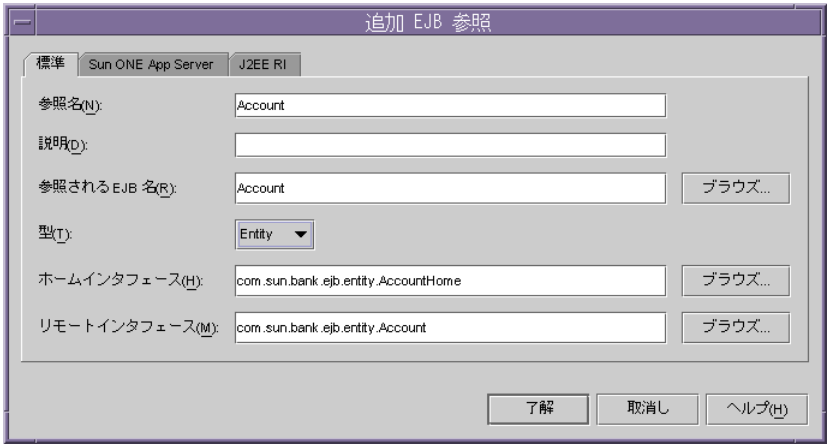
Sun ONE Application Server 用のリソース参照の追加



「Properties」ウィンドウ内の「Reference」タブで、「EJB Reference」をクリックして、EJB 参照を追加します。次の画面ショットは、BankTeller セッション Bean の EJB 参照を追加しているところです。BankTeller セッション Bean には、エンティティ Bean の「Account」と「Customer」の参照があります。このため、両方のエンティティ Bean について、エントリを作成する必要があります。

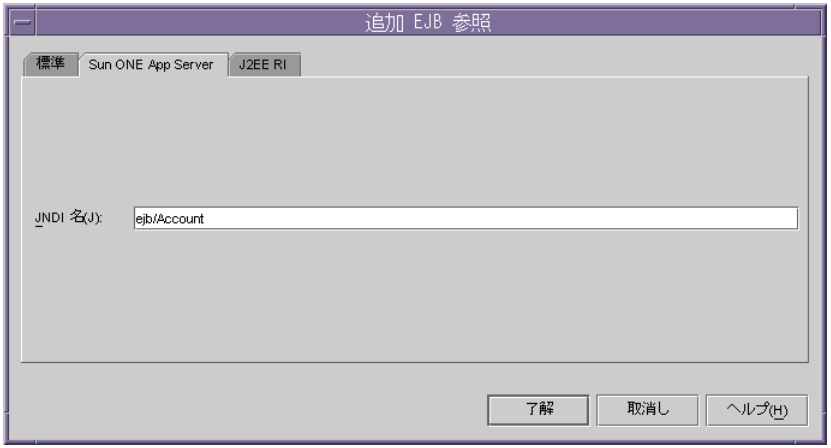
ホームインタフェースとリモートインタフェースについては、「Modify」ボタンをクリックし、既存の Beans のソースを選択して指定する必要があります。

EJB 参照の追加



次に、「EJB 参照」の「Sun ONE App Server」タブをクリックして、JNDI 名を指定します。次の画面ショットは、Account エンティティ Bean 用に作成した JNDI エントリ `ejb/Account` を示しています。同様に、`Customer` Bean の EJB 参照が追加されると、「Sun ONE App Server」タブで指定した JNDI 名が `jndi/Customer` になります。

Sun ONE Application Server 用の EJB 参照エントリの追加

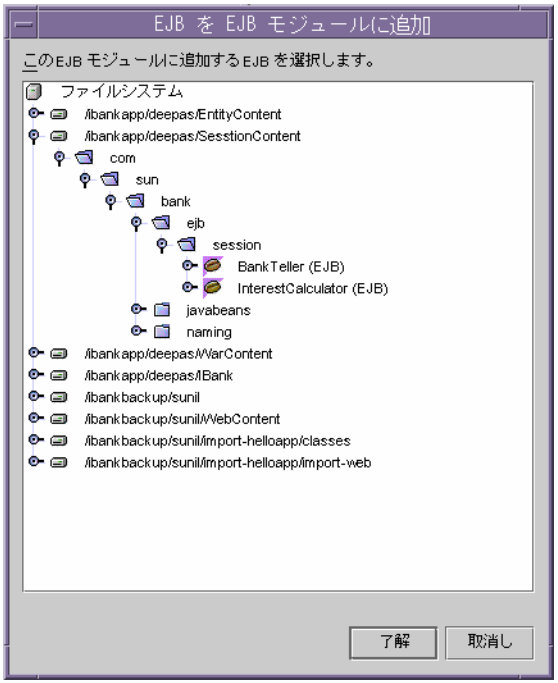


4. ソースファイルをコンパイルします。
5. EJB モジュールを作成し、モジュール内で EJB をアセンブルします。

   J2EE 1.2 仕様に従い、Sun ONE Application Server 7 で、複数の EJB を EJB モジュールにまとめる必要があります。ルートディレクトリ `SessionContent` で、新しい EJB モジュール SessionModule を作成します。これを行うには、フォルダを右クリックし、「新規」を選択した後、J2EE を選択してから、最後に「新規 EJB モジュール」を選択します。新しい EJB モジュールが作成されたら、そこにセッション EJB を追加します。

   次の画面ショットは、BankTeller EJB と InterestCalculator EJB を EJB モジュール SessionModule に追加しているところです。

EJB モジュールへのセッション Beans の追加



エンティティ Beans 用のモジュールの作成

1. エンティティ Beans のディレクトリには、次のようなものが含まれます。

   次のエンティティ Beans の Bean クラス、リモートインタフェース、およびホームインタフェース

   a. Account

   b. AccountType

   c. Branch

   d. Customer

   e. Transaction

   f. TransactionType

   Customer エンティティ Bean は Bean 管理で、その他のエンティティ Bean はコンテナ管理になります。

2. JDBC ドライバを設定します。

「エクスプローラ」の「実行時」ビュー内の「*Databases / Drivers / Add Driver:*」で、ドライバ名、実装クラス、および関連 URL の接頭辞を指定します。対応するドライバ用の JAR または ZIP は、Sun ONE Studio for Java にアクセスできるようにする必要があるため、`<SUN ONE STUDIO FOR JAVA_ROOT>/lib/ext` ディレクトリにコピーする必要があります。

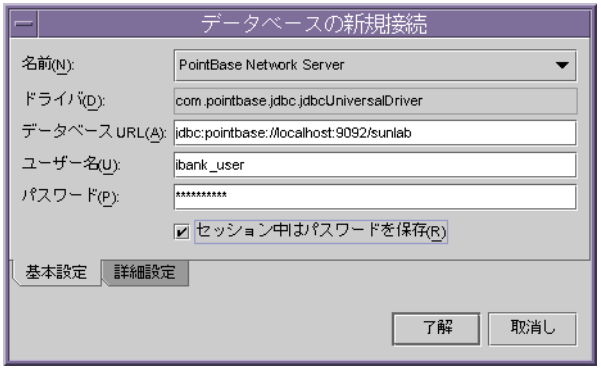
Solaris で、適切な Sun ONE Studio for Java ディレクトリにドライバクラスを格納するには、シェル (sh または ksh) から次のコマンド行を実行します。

```
cp $ORACLE_HOME/jdbc/lib/classes12.zip  <SUN ONE STUDIO FOR
JAVA_ROOT>/lib/ext
```

3. データベース接続プロパティを定義します。

「エクスプローラ」の「実行時」ビュー内の「Databases」の「接続を追加」では、使用するドライバ、完全な接続 URL、ユーザー名とパスワード、および適切なデータベーススキーマを指定します。

Sun ONE Studio for Java でのデータベース接続 (Oracle) の設定



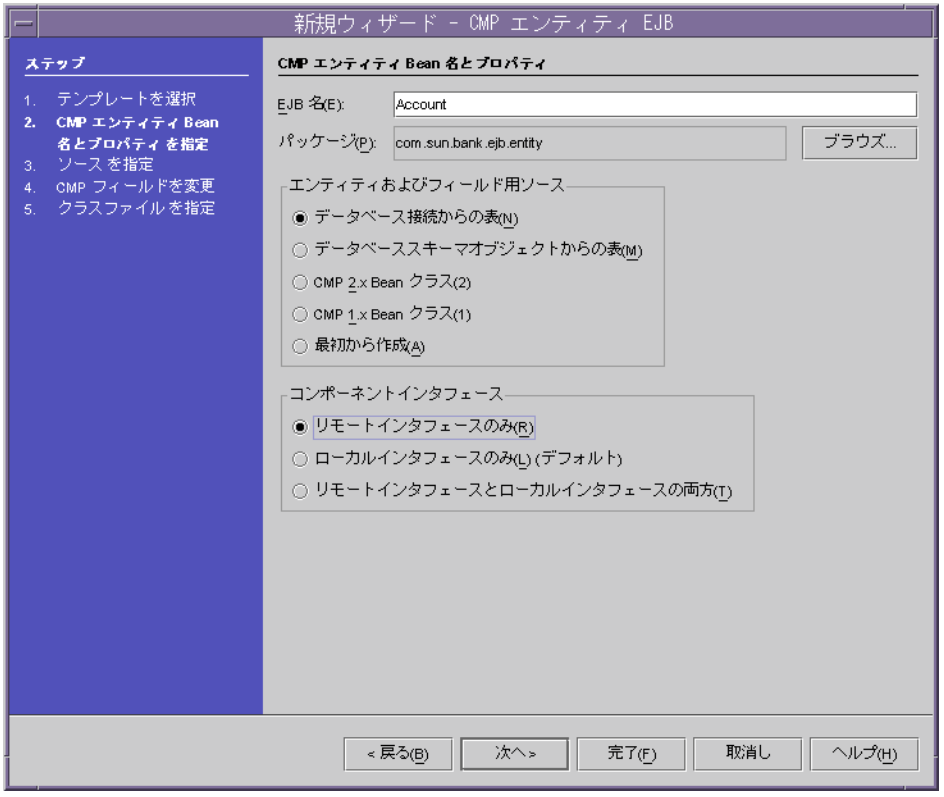
4. 既存のソースファイルから、新しい EJB を作成します。

Sun ONE Studio for Java では、既存のソースファイルから EJB を作成することができます。マウントされたディレクトリ EntityContent を選択し、entity サブフォルダが表示されるまで、ディレクトリを展開します。entity サブフォルダを右クリックして、新しい J2EE のオプションを選択してから、「エンティティ EJB(CMP/BMP)」をクリックすると、新しい EJB ウィザードが表示されます。

EJB の主な特性 (エンティティ、BMP、または CMP) を指定し、EJB の名前とパッケージを定義した後で、既存のソースファイルと EJB の別のコンポーネント (実装クラス、ホームインタフェース、およびリモートインタフェース) を一致させます。既存のソースファイルと一致させるには、ダイアログボックスの「インタフェースを変更」ボタンを使用し、「既存ユーザ実装クラスを選択」を選択します。エンティティ Beans の場合は、さらに cmp フィールドと表のマッピングを指

定する必要があります。「エクスプローラのファイルシステム」ビューで、「新規」「CMP エンティティ Bean」オプションを選択した後、EJB フィールドの持続性に使用するデータベース表を指定できるようにするために、「データベース接続からの表」オプションを選択します。

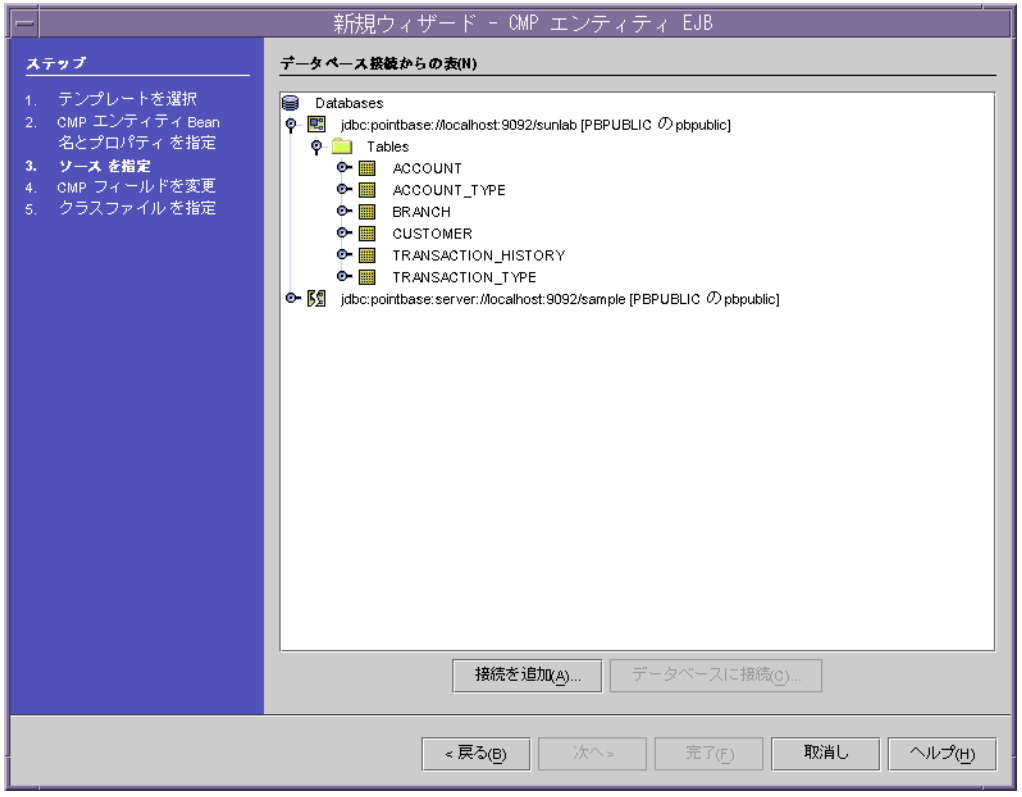
コンテナ管理による持続性を保持するエンティティ Bean の作成



次の画面ショットでは、定義されているデータベース接続のリストから、適切な接続を選択することができます。

接続を選択すると、この接続からアクセスできる表のリストが表示されるので、適切な表を選択します。

CMP Bean フィールドのマッピング用の表の選択

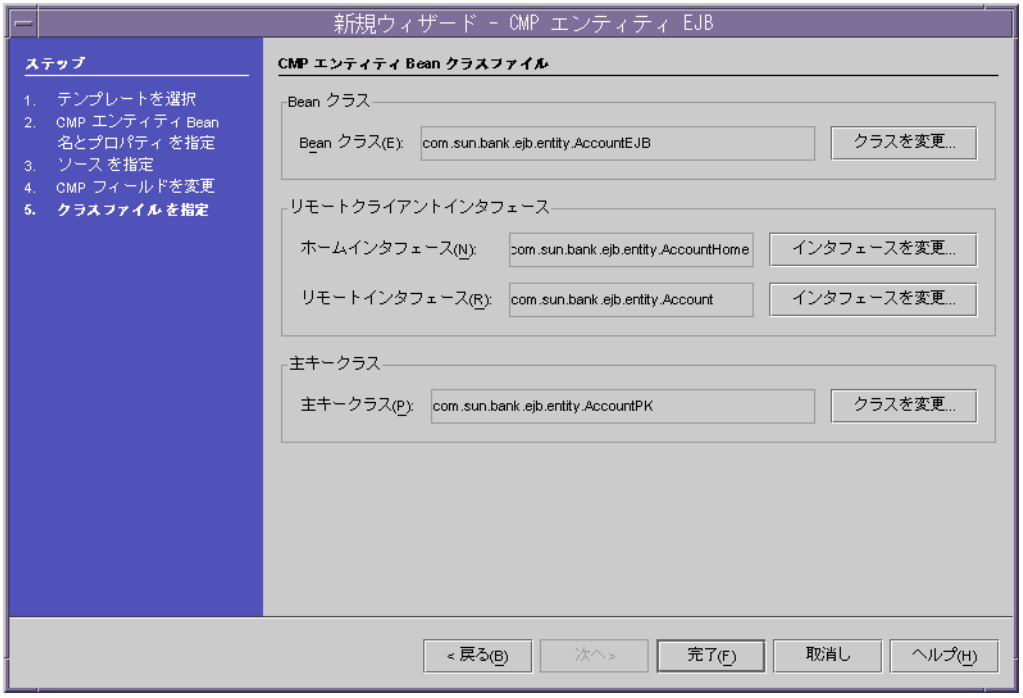


次の画面では、選択した表のカラムと Bean の CMP フィールド間でのマッピングを設定します。ここでは、Bean のフィールド名と対応する Java タイプを正しく指定するように、注意する必要があります。

表のカラムと Bean の CMP フィールド間でのマッピング

次の画面ショットは、エンティティ Bean のソースファイルを指定しているところです。

エンティティ Bean のソースファイルの指定

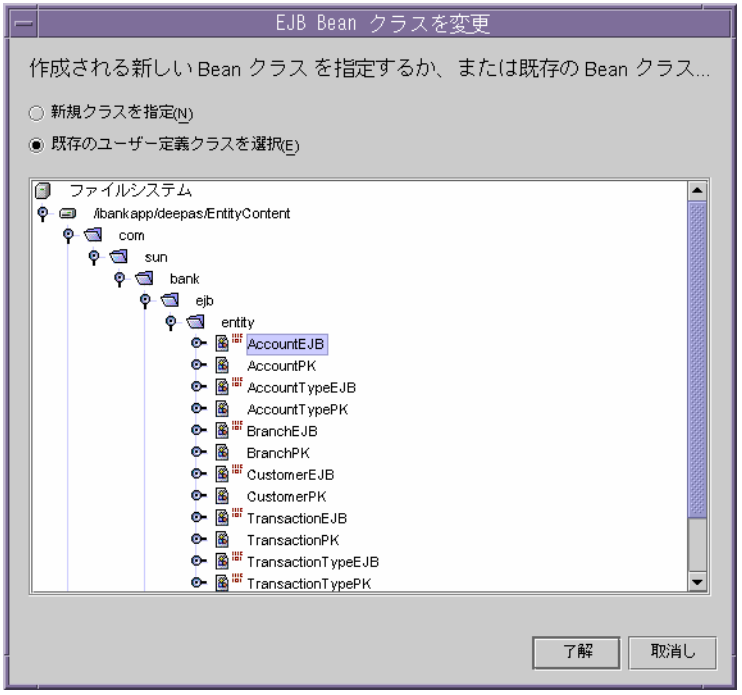


次に、「クラスを変更」ボタンをクリックして、既存のソースファイルから EJB を作成することを Sun ONE Studio for Java に指定します。

既存のソースファイルを選択したときに、エラーが発生した場合、前の手順をきちんと行っていないか、あるいはソースが正しく移行されていない可能性があります。このようなエラーが発生した場合、修正を行って対処する必要があります。

次の画面ショットは、EJB Bean クラス用に、既存のソースファイルを選択しているところです。

既存のソースファイルのオプションを選択した EJB Bean クラスの指定



次の作業では、必要に応じて、新しい EJB のプロパティを編集します。

エンティティ Beans はすべて同じ方法で作成する必要があります。

(注 : ここでは、既存のクラスを選択するオプションを表示するか、または別のクラスを使用するエラーが発生する場合があるので、「このクラスを使用」をクリックします。このような状況では、Sun ONE Studio で予期せぬ結果が生じる可能性があるため、Sun ONE Studio をいったん終了し、再起動してください。)

5. EJB のプロパティを編集します。

   プロパティがプロパティインスペクタに表示されるように、「エクスプローラ」ウィンドウで、新しい EJB を選択します。

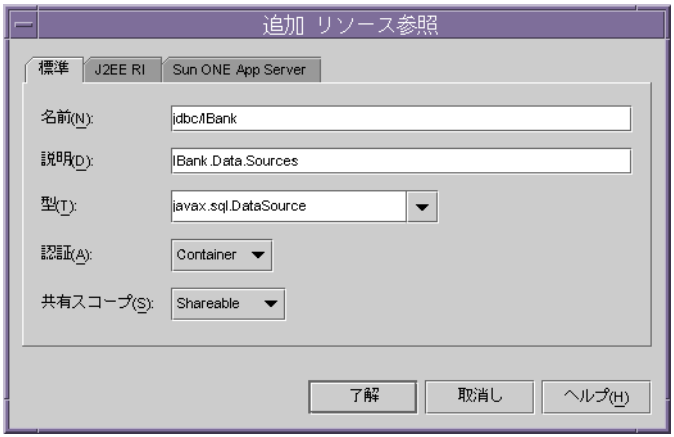
   「プロパティ」ウィンドウの「参照」タブを選択し、「リソース参照」ラベルの右側のテキストゾーンをクリックしてから、このテキストゾーンの右端にある省略符号 ("...") を示すボタンをクリックします。

   次のようなプロパティは、エンティティ Bean Customer だけで設定する必要があります。

次の画面ショットは、エンティティ Bean Customer のリソース参照を追加しているところです。

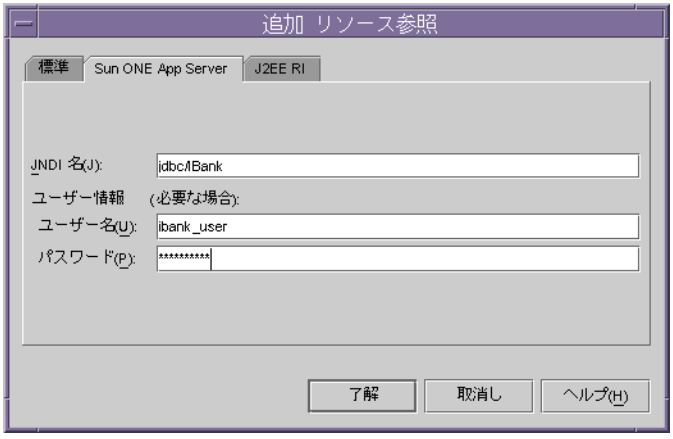
「標準」タブで、データソースの完全名 (`jdbc/`*DataSourceName*) とリソースタイプ (`javax.sql.DataSource`) を指定し、このリソースへのアクセスを管理するオプション (「認証」) のドロップダウンリストから、「コンテナ」を選択します。

リソース参照の追加



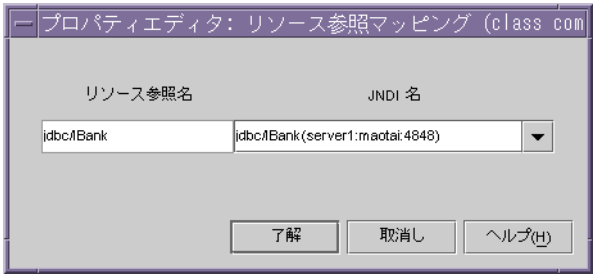
宣言を作成したら、「Sun ONE App Server」タブを選択し、前に定義したリソース参照に対応するエントリの JNDI 名のカラムに、データソースの JNDI 名 `jdbc/iBank` を指定します。ユーザー名とパスワードも指定します。

リソース参照の編集



「プロパティ」ウィンドウの「Sun ONE AS」タブを選択し、「リソース参照マッピング」をクリックしてから、使用する必要があるサーバーインスタンスで、データソース `jdbc/IBank` を選択します。次の画面ショットは、同じ内容を表しています。

Sun ONE Application Server 用のリソース参照のマッピング

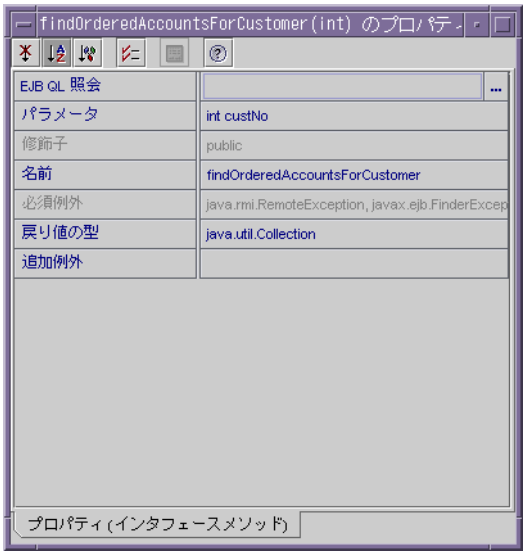


6. *findByPrimaryKey* メソッド以外の検索の EJB QL を設定します。

   EJB QL は、検索に対して指定する必要があります。CMP 2.0 仕様に従って、検索は EJB QL を使用します。

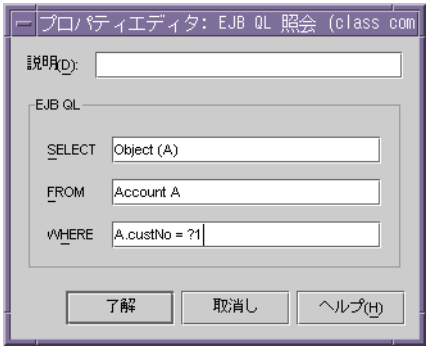
   iBank アプリケーションの場合、このタイプの編集が必要なエンティティ Bean は Account Bean です。Sun ONE Studio の「エクスプローラ」ウィンドウで、*AccountEJB* ノードを選択し、そのノードのファインダメソッドを展開します。*findByPrimaryKey* 以外の検索メソッドをクリックして、その「プロパティ」ウィンドウを開きます。

ファインダメソッドのプロパティ



「EJBQL 照会」をクリックして、クエリを入力します。次の画面ショットは、クエリを入力したところです。

検索メソッドの EJB QL の編集



7. EJB モジュールを作成し、モジュール内で EJB をアセンブルします。

   EntityModule という名前で、新しい EJB モジュールを作成し、このモジュールにすべてのエンティティ Beans を追加します。これを行うには、EJB モジュールを右クリックし、EJB を追加するオプションを選択します。J2EE 1.2 仕様に従って、EJB は 1 つの EJB モジュールにまとめる必要があります。

8. 新しいデータベーススキーマを作成します。

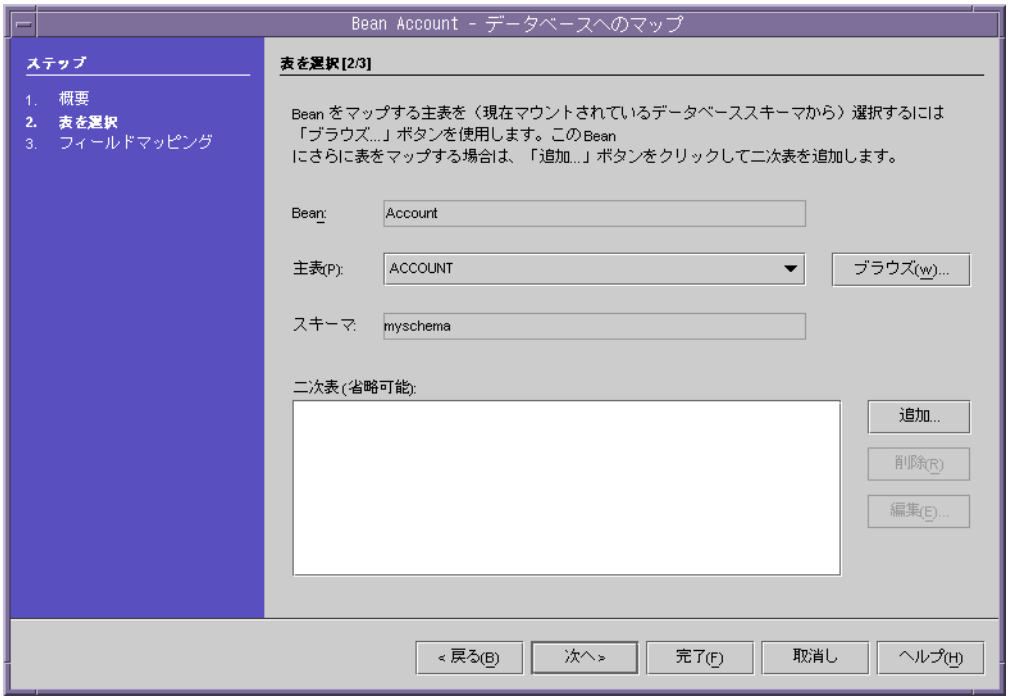
   「ファイル」メニューから「新規」をクリックし、新しいデータベーススキーマを選択します。スキーマを取り込む必要があるデータベースの接続情報を定義します。

9. Sun ONE Application Server 7 のデータベースエントリをマップします。

   EJB モジュールの EJB ノードを選択し、このノードを右クリックして、「プロパティ」ウィンドウを選択し、さらに「Sun ONE AS」タブを選択します。この特定のエンティティ Bean について、データベーススキーマと主表名を指定します。EJB モジュール内の他のエンティティ Bean についても、同様のプロセスを繰り返します。

   次の画面ショットは、エンティティ Bean「Account」の主表を選択しているところです。

データベースのマッピング

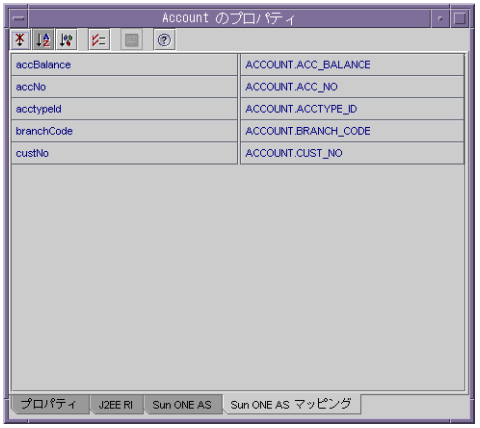


「次へ」をクリックして、Bean の cmp フィールドと表フィールドのマッピングを指定します。

次に、「プロパティ」ウィンドウから「Sun ONE マッピング」タブを選択し、同じマッピングを入力します。

次の画面ショットは、Account EJB のマッピングを示しています。

エンティティ Bean「Account」のプロパティ



同様に、すべてのエンティティ Beans のマッピングを設定する必要があります。

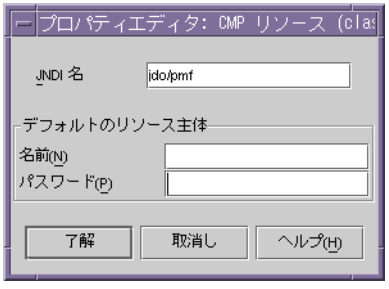
データベース表フィールドに対応する特定のエンティティ Bean のマッピングについては、「付録 A」を参照してください。

10. CMP リソースを追加します。

EntityModule を選択し、そのプロパティを表示して、「Sun ONE AS」タブをクリックします。続いて、「CMP リソース」をクリックして、持続マネージャのファクトリを設定します。

次の画面ショットは、設定を行っているところです。

CMP リソースの追加



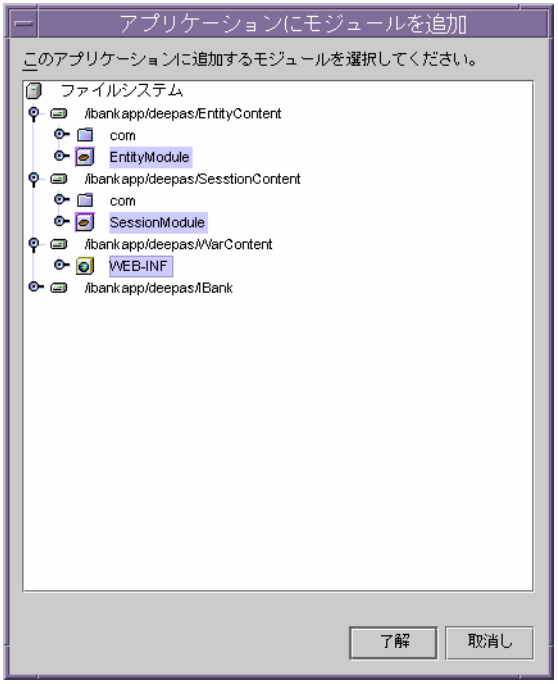
Sun ONE Studio for Java でのエンタープライズアプリケーションの作成

Web アプリケーションと EJB ファイルを作成したら、今度はすべてのモジュールを 1 つにまとめるエンタープライズアプリケーションを作成します。エンタープライズアプリケーションの作成プロセスは、以下のとおりです。

1. ソースに利用できる同じパッケージの下にある新しいディレクトリ IBank に、新しいエンタープライズアプリケーションモジュールを作成します。
2. Web モジュールと EJB モジュールをエンタープライズアプリケーションモジュールに追加します。

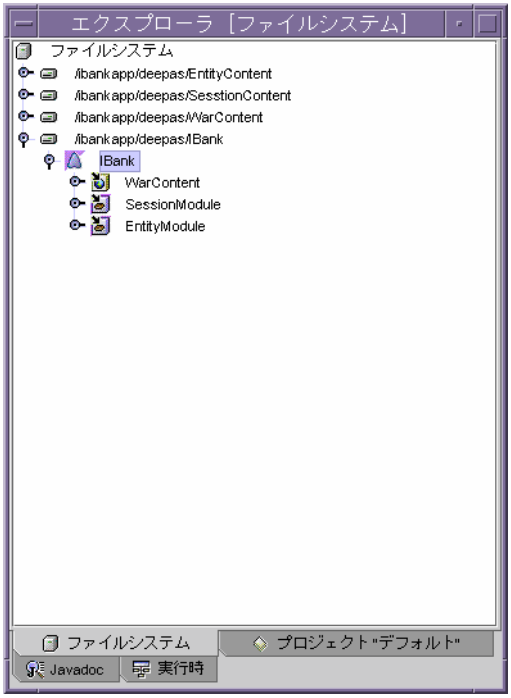
   次の画面ショットは、WarContent という名前の Web モジュールと SessionModule と EntityModule という名前の EJB モジュールを含んだ iBank という名前のエンタープライズアプリケーションを示しています。

アプリケーションへのモジュールの追加



次の画面ショットは、3 つのモジュールを含んだ iBank アプリケーションを示しています。

異なるモジュールを含む iBank アプリケーションを示すファイルシステム



3. エンタープライズアプリケーションのプロパティを編集します。

   プロパティエディタを使用すると、エンタープライズアプリケーションモジュールに、異なるプロパティを設定することができます。特に、このエディタでは、エンタープライズアプリケーションの Web モジュールについて、ルートコンテキスト名を定義します。

Web コンテキストの指定

4. EAR ファイルをエクスポートします。

   エンタープライズアプリケーションを右クリックし、EAR ファイルをエクスポートするオプションを選択して、EAR ファイルをエクスポートします。この EAR ファイルには、JAR ファイル、WAR ファイル、および XML ファイルが含まれます。この EAR ファイルには、Sun ONE Application Server 7 での配備に必要な Sun ONE 固有の XML ファイルが含まれます。この EAR ファイルは、これで配備することができます。

Sun ONE Application Server 7 でのアプリケーションの配備

最後の作業は、Sun ONE Application Server 7 のインスタンスでのアプリケーションの配備です。アプリケーションを配備するプロセスは、以下に示すとおりです。

1. Sun ONE Studio for Java から Sun ONE Server 7 のインスタンスへのアプリケーションの配備

   EAR ファイルを右クリックし、「配備」オプションを選択します。これによって、デフォルトのサーバーインスタンスに、アプリケーションが配備されます。サーバーインスタンスを再起動し、アプリケーションをテストします。

2. Sun ONE Application Server 7 の *asadmin* ユーティリティを使用した、Sun ONE Application Server 7 のインスタンスへのアプリケーションの配備

Sun ONE Studio for Java を使用しないで、Sun ONE サーバーインスタンスに、エンタープライズアプリケーションを配備する場合、Sun ONE Studio for Java からアプリケーションの EAR アーカイブを作成およびエクスポートした後で、Sun ONE Application Server 7 の *asadmin* ユーティリティを使用します。

asadmin 配備ユーティリティを使用した iBank アプリケーションの配備手順については、「iBank アプリケーションの手動移行」内の「asadmin ユーティリティを使用した Sun ONE Application Server 7 での iBank アプリケーションの配備」を参照してください。

BEA WebLogic Server v6.1 および IBM WebSphere v4.0 からの移行

BEA WebLogic v6.1 および IBM WebSphere v4.0 についての J2EE アプリケーション移行プロセスとサンプルアプリケーションの移行の詳細は、移行サイトから該当するマニュアルを参照してください。

第 3 章

KIVA/NAS 4.1 から Sun ONE AS 7 への移行

Kiva/NAS 4.1 Java AppLogic のアプリケーションは、iPlanet Migration Toolkit (iMT 1.2.3) で J2EE Web モジュールに移行できます。この結果生成される Web モジュールは、JATO と thin KFC (Kiva Foundation Classes) adaption layer を利用し、任意の J2EE web コンテナでの AppLogic コードの実行をサポートします。

はじめに

移行プロセスの開始前に、リリースノートを読んで、それぞれの環境に関連する最新情報と問題点を把握してください。`%MIGTBX_HOME%/bin/readme.txt` ファイルも参照してください。この readme ファイルには Migration Toolbox の正しいインストール方法と設定方法、および動作環境についての情報も記載されています。後の節に記載されている移行プロセスを開始する前に、これらのインストールおよび設定を完了し、環境を整備しておく必要があります。

`%MIGTBX_HOME%` は Sun ONE Migration Toolbox (S1MT) をインストールまたは解凍したディレクトリを表します。

移行準備

移行プロセスの概要

AppLogics で作成されたプロジェクトを同等の J2EE プロジェクトに完全に移行するための主要なフェーズは 2 種類あります。1 つは自動移行フェーズ、もう 1 つは手動移行フェーズです。自動移行には抽出および変換の 2 つの手順があります。

自動移行フェーズ

このフェーズでは、AppLogic アプリケーションソースの移行準備、および S1MT による自動抽出および変換を行います。AppLogic ファイル、GXR、クエリファイル、テンプレート、静的コンテンツ、通常の Java ソースおよびプロパティなど、元のアプリケーションソースファイルを含むアーカイブ (JAR/ZIP) を入力として使用します。このファイルはユーザーが準備します。このファイルはアプリケーション抽出アーカイブと呼ばれます。標準 Java アーカイブ (JAR/ZIP) を使用して既存のアプリケーションをパッケージ化すれば、NAS/iAS の実行環境は不要であり、移行環境がよりフレキシブルになります。単体テストのための手動移行で必要になるのは、データベース、Web サーバー、アプリケーションサーバーなどの実行時のインフラストラクチャだけになるため、顧客のサイトでも移行処理ができるようになります。重要なことは、このアーカイブは単にアプリケーションサーバーの ./nas/APPS ディレクトリの移行先となるコンテンツであり、Web サーバーのドキュメントルートであるということです。KIVA AppLogic の抽出ツールはこのアーカイブを読み込み、アプリケーション記述子を作成します。iMT v1.2.3 は、現在、アプリケーション抽出アーカイブの自動生成をサポートしています。Kiva Migration Toolbox Builder の「Addin」メニューにある「Addin」を参照してください。

アプリケーション記述子の XML ファイルは、アーカイブの各ファイルの後処理を行う Translation Tool のガイドとしての役割を持ちます。移行ではアプリケーション記述子の調整が必要な場合があります。アプリケーション記述子の編集に関してはテクニカルノートを参照してください。Translation Tool を実行すると、JATO および KIVA 移行ライブラリをベースにした J2EE に準拠するコンポーネントだけで構成された部分移行済みのアプリケーションが、配備記述子、サーブレット、JSP、コマンド、クエリファイルなどを含む Web アプリケーションアーカイブとして作成されます。移行が完了する場合もあります。

変換プロセスが終了すると、HTML テンプレートはすべて JSP に変換され、GX タグは KIVA 移行ライブラリで使用される新しい JSP タグに変換されます。AppLogic ソースファイルは KIVA 移行ライブラリを使用するように調整されます。ただし文をインポートするための最小限の変更だけが行われます。変換プロセスでは、JATO アプリケーションのすべてのコンポーネントとコマンド直接起動モジュールを含む、Web アプリケーションのインフラストラクチャも作成されます。ただし、変換フェーズでは、KFC API 対応で書かれたコードでも、KIVA 移行ライブラリの「対象外」のものについては自動移行を行いません。この移行が手動移行フェーズの主要なタスクになります。iMT v1.2.3 は、現在、静的ドキュメントの自動移行をサポートしています。これは URL の修正に有効です。Kiva Migration Toolbox Builder の「Addin」メニューから「Addin」を選択してください。また新しい Kiva Document Translation Tool もサポートしています。

手動移行フェーズ

一般に手動移行フェーズでは、自動移行されたアプリケーションの出力内容の確認、および対象外の KFC API コードの J2EE 固有のコードへの移行を行います。このプロセスでは通常、アプリケーションまたはアーキテクチャの再設計は不要です。通常は、手動移行の必要があるコードを、KIVA 移行ライブラリ `kivaMIGRATION.jar` の MIGRATION バージョンで「deprecation」でコンパイルすれば、手動処理対象の箇所が明示されます。

作業環境の準備

先に進む前に次の事項を行ってください。

1. iPlanet Migration Toolbox がインストール済みかどうかを確認します。
    - バージョンアーカイブをターゲットディレクトリへ解凍します。方法は `readme.txt` ファイルを参照してください。
    - Toolbox アプリケーションを起動してインストールのテストを行います。`%MIGTBX_HOME%/bin` ディレクトリで `toolbox.bat` を実行してください。空の Toolbox が表示されます。何も表示されない場合は、Migration Toolbox が正常にインストールされているかどうか、`%MIGTBX_HOME%/bin/setenv.bat` の環境設定がすべて適切かどうかをチェックしてください。
2. クラスのバージョン問題を回避するため、Toolbox アプリケーションを実行する時にはすべての JAR ファイルを JDK の拡張ディレクトリ `%JAVA_HOME%/jre/lib/ext` から必ず削除することをお勧めします。Toolbox の実行に必要なクラスはすべてバージョンに含まれています。拡張ディレクトリの JAR ファイルの名称を変更するだけでは不十分であり、別の場所に移動する必要があります。
3. 移行対象の AppLogic ベースアプリケーションを決定します。
4. アプリケーション抽出アーカイブを生成します。`./nas/APPS` の下のアプリケーションに関連するすべてのファイルおよびディレクトリを含む ZIP または JAR ファイルを生成するのが一番簡単な方法です。AppLogics 用の iMT では、実際はアプリケーションの Java クラスまたはライブラリをロードまたは実行しません。ソースレベルの抽出および変換が終了した時点で、アーカイブにすべての従属クラスまたはライブラリが含まれていなくても問題はありません。これらが必要になるのは自動移行後にコンパイルを行う場合です。
5. ここで、Sun ONE Application Server 7 (S1AS)、Forte for Java 4.0 EE、または他の J2EE に準拠するサーブレットまたは JSP コンテナをインストールできます。
    - サーバーまたはコンテナのインストール手順に従ってインストールを行います。

- サーバーまたはコンテナを起動し、デフォルトのホームまたはインデックスページを読み込んでインストール状況をテストします。エラーがある場合は、インストールプロセスの問題を解決してから先に進みます。

自動移行プロジェクトの準備

AppLogic および KFC による開発は自由度が大きく、規定は実質的にはないとみなすことができます。したがって、すべてのプロジェクトの置き換えを iMT では把握できません。このため、移行プロジェクトの準備時には Sun Professional Services に援助を依頼することを強くお勧めしています。

iMT Kiva BETA では、手動移行時に JDK 1.3.1 のコードを J2EE 環境でコンパイルすると障害が発生することが知られています。このような考慮点と移行を行う前に必要な処理を次に示します。

iMT 使用前に J2EE 環境に合わせたアプリケーションコードの準備が必要です。コードは JDK 1.3 (または少なくとも JDK 1.2.2) および J2EE API に対応するようにコンパイルする必要があります。たとえば、iMT には KFC 用の `kfcjdk11.jar` ライブラリがあります。これは既存のアプリケーションを、Forte for Java (FFJ) などのより高度な J2EE 対応 IDE でコンパイルするために用意されています。標準 AppLogic アプリケーションは、`kfcjdk11.jar` をファイルシステムのクラスパスに追加するだけで FFJ でコンパイルできるようになります。コンパイル前に非推奨フラグを TRUE に設定し、非推奨コードを明確にします。

JDBC をすでに使用しており、JDBC、Oracle、および Sybase などのベンダーが新しい JDK への対応を推奨している場合は、データベースサービスを最新のサードパーティドライバに対応させておくことを強くお勧めします。

移行タスクを正確に決定し、正しいデータを提供するためには、説明した特別な注意事項をすべて最初に検討する必要があります。簡単に言うと、コードの「例外的な箇所」をすべて特定する必要があります。開発者と締結している J2EE コンテナ規約の同時サーバーパターンとの不整合を発生させる、コードパターンまたは Java サービスの使用がこれにあたります。たとえばコードが java.lang.Thread を直接、または共有リソースとして使用する場合は、このコードの J2EE 適合性を検査する必要があります。

コード自体が J2EE に対応していても、使用しているサードパーティの Java サービスが対応していない場合もあります。たとえば Visibroker for Java や Iona など、旧バージョンの CORBA がこれにあたるため、アップグレードの必要があります。

J2EE では、論理アプリケーションは個別の Web アプリケーション、つまり WAR として配備する必要があります。移行を進める前に論理アプリケーションと共通ライブラリを分離することが、より簡単な方法です。

外部 URL の変更に対して準備する必要があります。J2EE の移行にどの方法を使用しても URL は変更されるため、ブックマークとこれまでにパブリッシュされた URL の移行が必要になります。iMT v1.2.3 は静的ドキュメントの URL の自動移行を部分的にサポートしています。しかし、それでも既存のシステムを調査し、手動で行う必要がある変更を明確にしておく必要があります。

GXR ファイルの準備

アプリケーション記述子の生成時の抽出フェーズを正確に実行するために、NameTrans で使用される AppLogic ファイルと AppLogic 名、および URL を識別する GXR ファイルが必要になります。ほとんどのアプリケーションで GXR ファイルを 1 つ以上使用します。アプリケーションの各パッケージで 1 つ以上使用する場合もあります。抽出フェーズでは、アプリケーション抽出アーカイブごとに GXR ファイルが 1 つだけ必要になります。GXR ファイルが複数ある場合は結合する必要があります。GXR ファイルがない場合は正しい GXR 構文に基づいて作成する必要があります。KIVA レジストリ、つまり `./nas/bin/kreg -save temp.out SOFTWARE` のダンプを取ることでソースデータが得られます。要約すると、抽出ツールでは GXR ファイルを使用して、アプリケーション抽出アーカイブのどのファイルを AppLogic のソースファイルとするかを決定し、GUID の AppLogic や AppLogic クラス名へのマッピングを行います。

Extraction Tool 実行前の注意点

AppLogic アプリケーションが完全にアプリケーションサーバーサイド Java をベースにしており、クエリファイル、HTML テンプレート、AppLogic ソース、サポート Java ソースなどの要素も完全にアプリケーションサーバーサイドのものである場合、通常は `./nas/APPS` ディレクトリの関連するコンテンツのアーカイブを生成するだけでアプリケーション抽出アーカイブを作成できます。ただし、アプリケーションが静的なコンテンツも含む場合は追加作業が必要になります。一般的には静的コンテンツは Netscape Enterprise Web サーバーに置き、動的コンテンツを AppLogics やテンプレートとしてアプリケーションサーバーに置くのがより効率的です。ただし、このようにコンテンツを分散させるのが良いのか、それとも同じサーバーに置くのが良いのかは J2EE サーバーのベンダーによっても変わります。静的コンテンツは自動移行時、またはその終了後に WAR に追加できます。通常これが一番簡単な方法です。

元の AppLogics アプリケーションを J2EE JATO に iMT を使用して移行する場合には、考慮すべき重要な点が 1 つあります。AppLogics (POST/GET) を起動する URL が `http://host/cgi-bin/gx.cgi/AppLogic+HelloWorld` のような絶対パスの場合でも、移行後の URL は ServletContext で定義されたコンテキストに対する相対パスになるため、必ず相対パスで記述するようにします。URL の変換は静的コンテンツや HTML テンプレートなどの動的コンテンツの変換とは別に行う必要があります。iMT

はすべての AppLogics を、コマンド直接起動モジュールと呼ばれる特別な JATO モジュールで JATO コマンドの実装にマップします。変換されるすべての AppLogics が ServletContext 内の同じパスから呼び出されるため、結果として生成される HTML マークアップでは、イントラ AppLogic 起動 (URL) が最も予想しやすいものになります。したがって、すべての AppLogic 起動 URL はイントラ AppLogic 向けに変換されます。アプリケーション抽出アーカイブの HTML テンプレートに静的コンテンツが含まれる場合、この静的コンテンツのコンテキストはコマンド直接起動モジュール (ModuleServlet) の ServletContext で指定されているパスとは適合しないため、AppLogic URL を調整する必要があります。OnlineBankSample の移行を行うとこの調整の必要性を確認でき、Kiva Document Translation Tool を使用して静的ドキュメントの自動変換を実際に行うことができます。

OnlineBankSample の移行

この節では、onlineBankSample を J2EE に自動または手動で移行する手順について説明します。

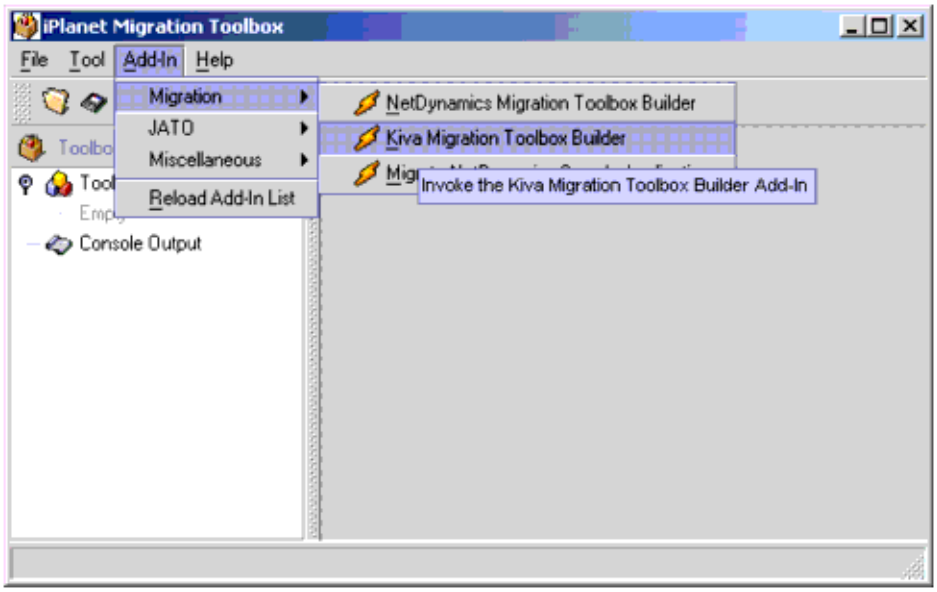
Migration Toolbox の実行

iMT 1.2.3 をまだインストールしていない場合はインストールします。iMT のインストールおよび起動を説明している「移行準備」節を参考にしてください。

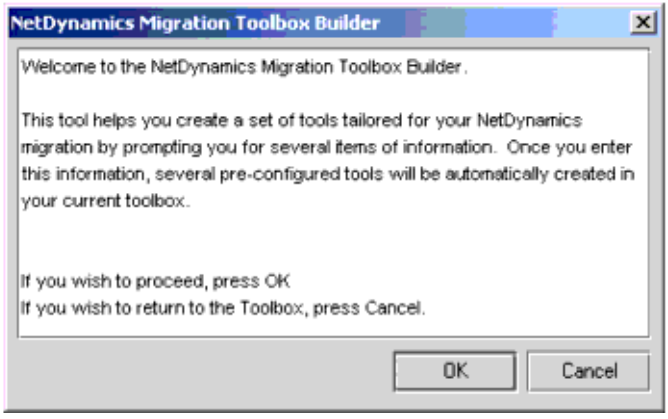
`%MIGTBX_HOME%¥bin¥setenv.bat` を編集し、iMT のインストール場所と JDK ホームディレクトリを適切に設定します。

Toolbox の作成

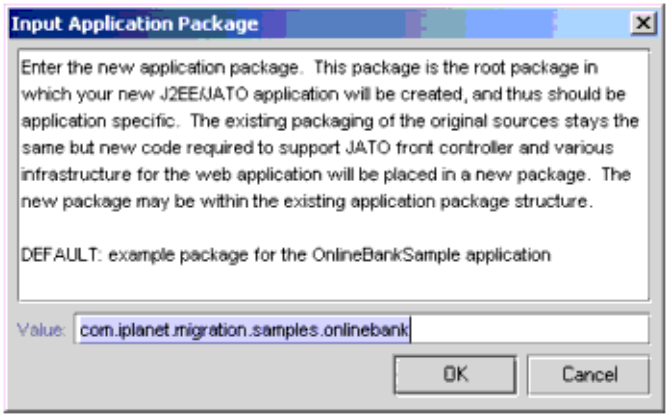
1. 「Addin」>「Migration」メニューから「Kiva Migration Toolbox Builder」を選択します。



モーダルダイアログウィザードが表示されます。

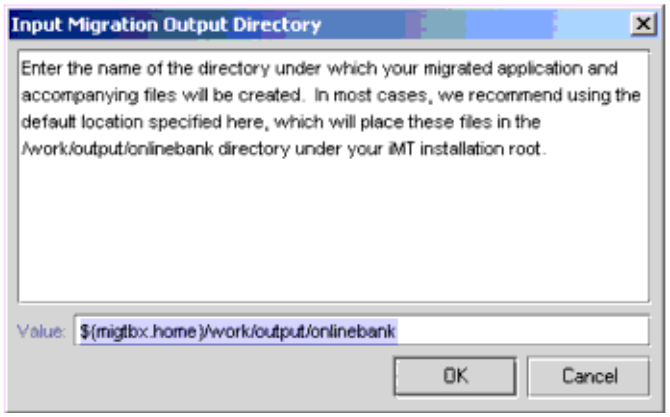


「OK」をクリックしてウィザードの最初の手順に進みます。

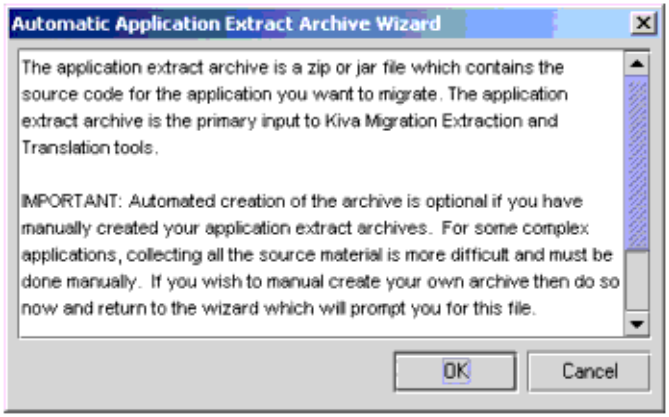


2. 自動 iMT 移行では新しい Java JATO ファイルを含む J2EE インフラストラクチャをいくつか生成します。これらの新しいファイルにパッケージを割り当てる必要があります。元のアプリケーションの既存の Java ソースがパッケージ化されて残る場合でも、これらの新しいファイルにパッケージを割り当てる必要があります。パッケージ名に関する制限はありません。OnlineBankSample アプリケーションにはデフォルト値が提示されます。

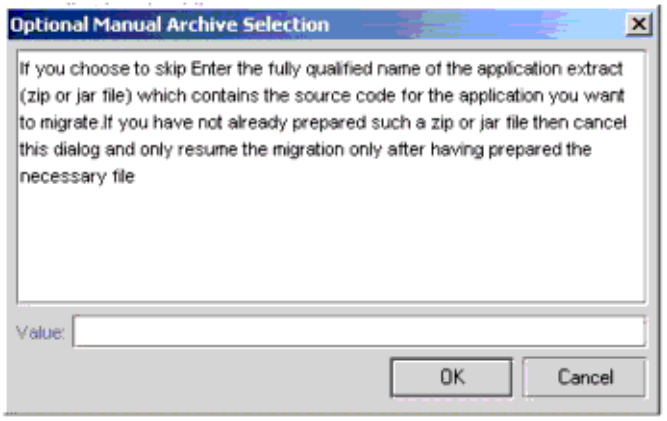
   パッケージを入力し、「OK」をクリックしてウィザードの次の手順に進みます。



3. iMT で生成されるすべてのデータを格納するディレクトリを入力します。通常はデフォルトの値を変更する必要はありません。この例でもデフォルトの値を使用しています。「OK」をクリックしてウィザードの次の手順に進みます。

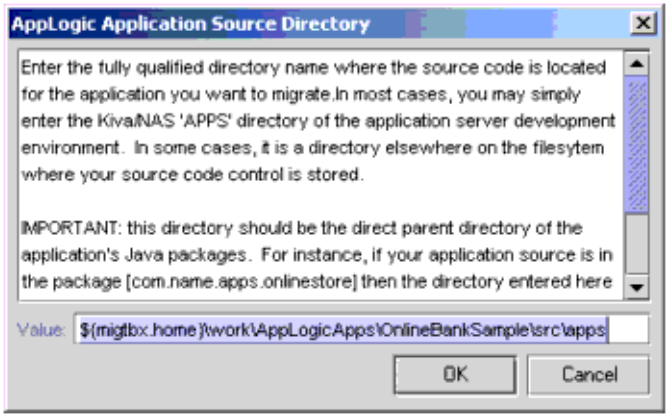


4. Automatic Application Extract Archive Wizard を使用して、アーカイブを自動構築するツールを作成できます。「OK」を選択すると手順 (5) に進みます。「Cancel」を選択すると以下の抽出アーカイブ選択ダイアログが表示されます。

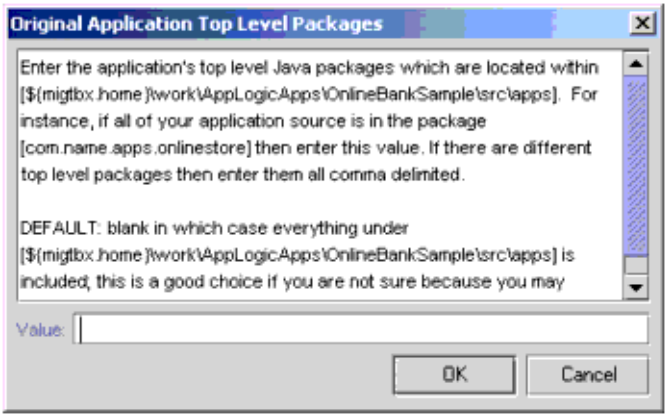


このダイアログボックスで手動で作成されたアーカイブを指定できます。抽出アーカイブをすでに作成しており、新しいツールボックスだけを構築する場合はこの手順が便利です。

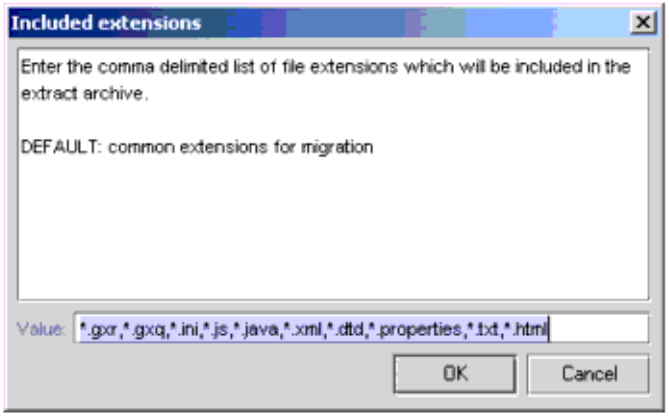
Automatic Application Extract Archive Wizard で「OK」を選択すると次のダイアログが表示されます。



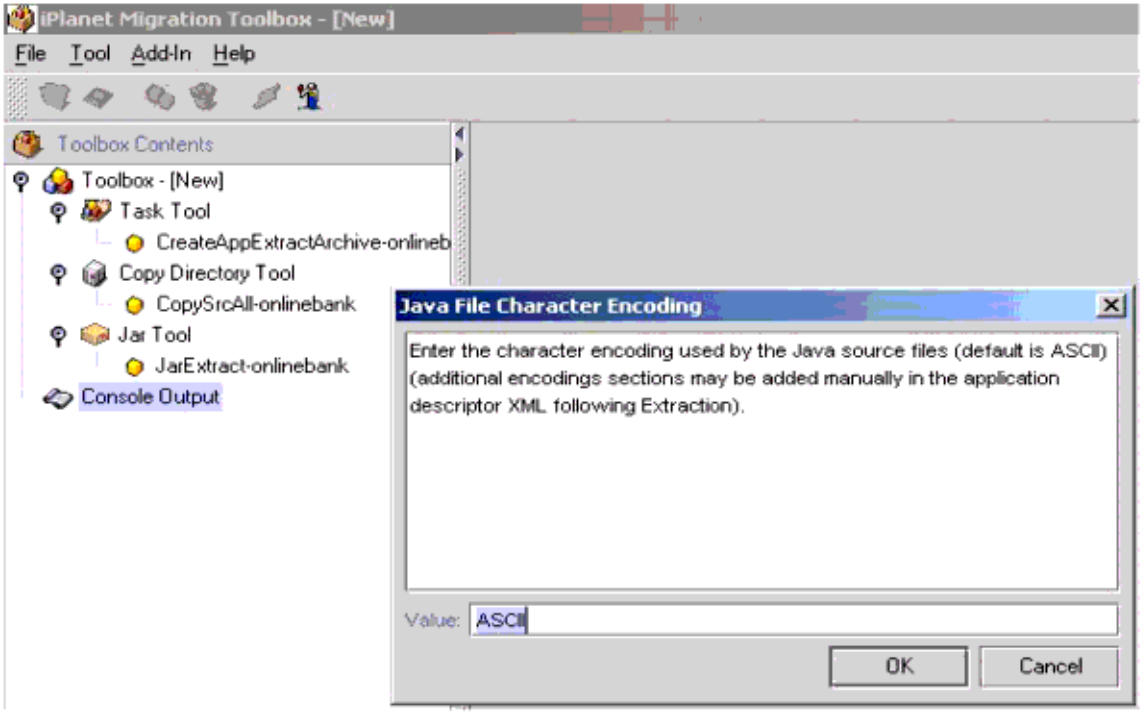
5. デフォルト値を変更せずに「OK」を選択して、次の手順に進みます。



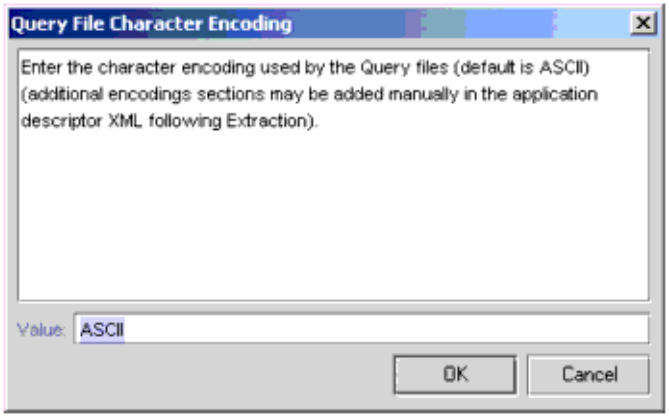
6. デフォルトの空白のリストを変更せずに「OK」を選択して、ウィザードの次の手順に進みます。



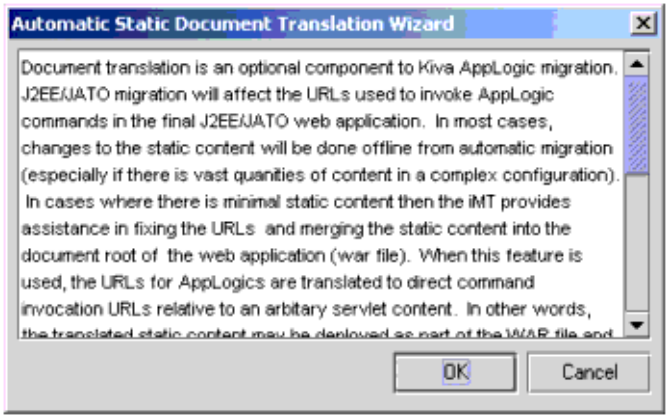
7. デフォルトを変更せずに「OK」を選択します。ツールボックスに 3 つのツールが新しく生成され、ウィザードの次の手順に進みます。



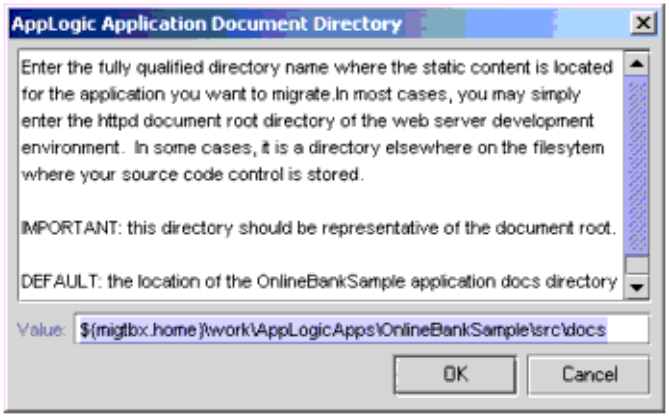
8. OnlineBankSample の Java ソースはすべて ASCII エンコードを使用しています。したがってデフォルトをそのまま使用します。ユーザー独自のアプリケーションを移行する時に、日本語 Shift_JIS など他の文字エンコードを使用している Java ソースがある場合は、その文字エンコードを指定します。「OK」をクリックしてウィザードの次の手順に進みます。



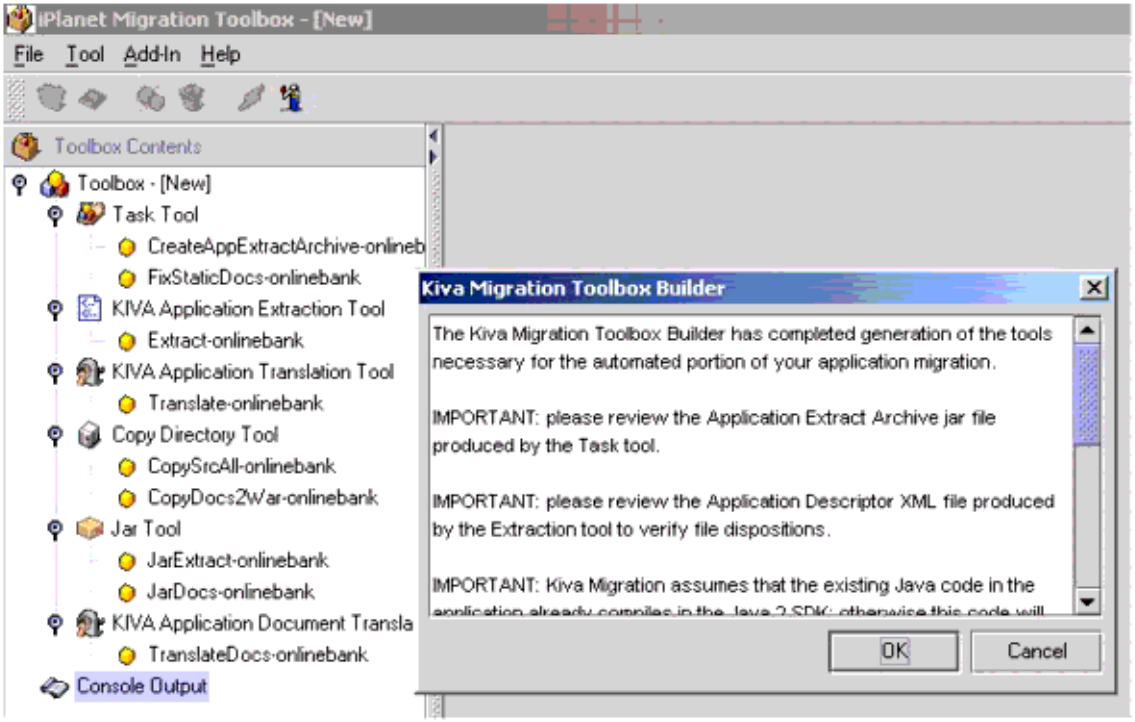
9. OnlineBankSample のクエリファイルはすべて ASCII エンコードを使用しています。したがってデフォルトをそのまま使用します。ユーザー独自のアプリケーションを移行する時に、日本語 Shift_JIS など他の文字エンコードを使用しているクエリファイルがある場合は、その文字エンコードを指定します。「OK」を選択するとツールボックスに Kiva Extraction and Translation tools が作成され、ウィザードの次の手順に進みます。



10. iMT v1.2.3 には静的 HTML ドキュメントを自動変換し、WAR ファイルと自動結合する機能が用意されています。この機能をスキップするとウィザードは終了し、ツールボックス生成は終了します。OnlineBankSample に対して「OK」を選択して自動機能を使用します。



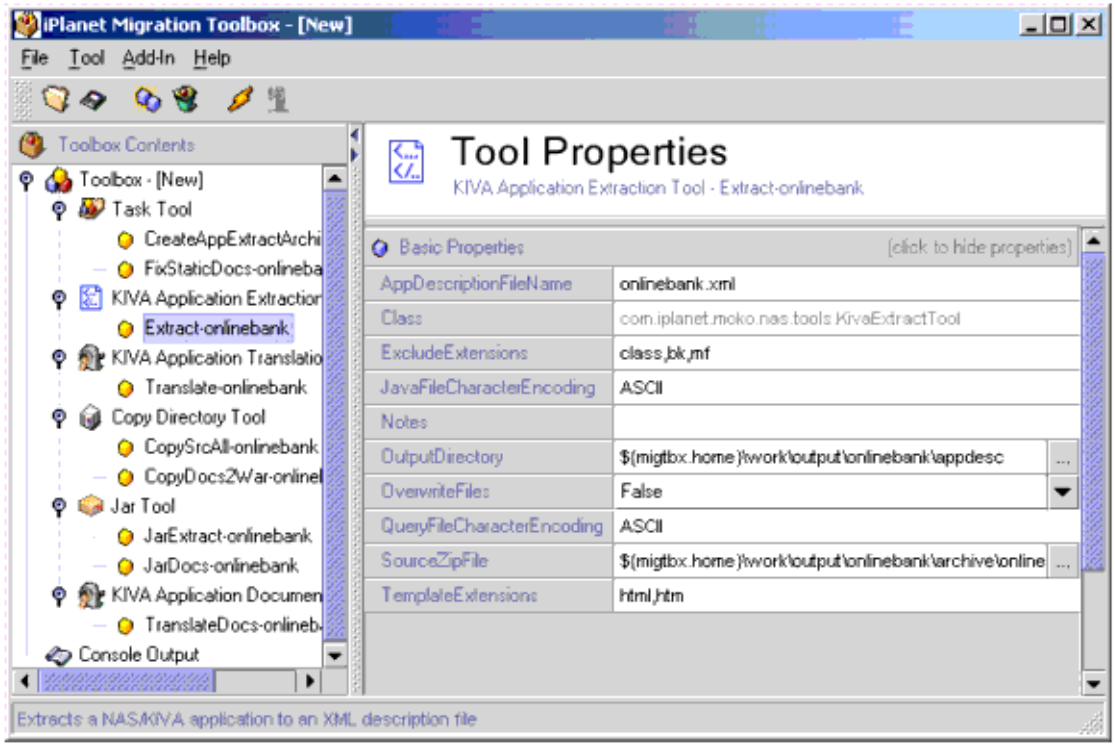
11. 「OK」を選択して OnlineBankSample のデフォルトのドキュメントディレクトリを指定します。4 つの新しいツールがツールボックスに生成され、ウィザードが終了します。



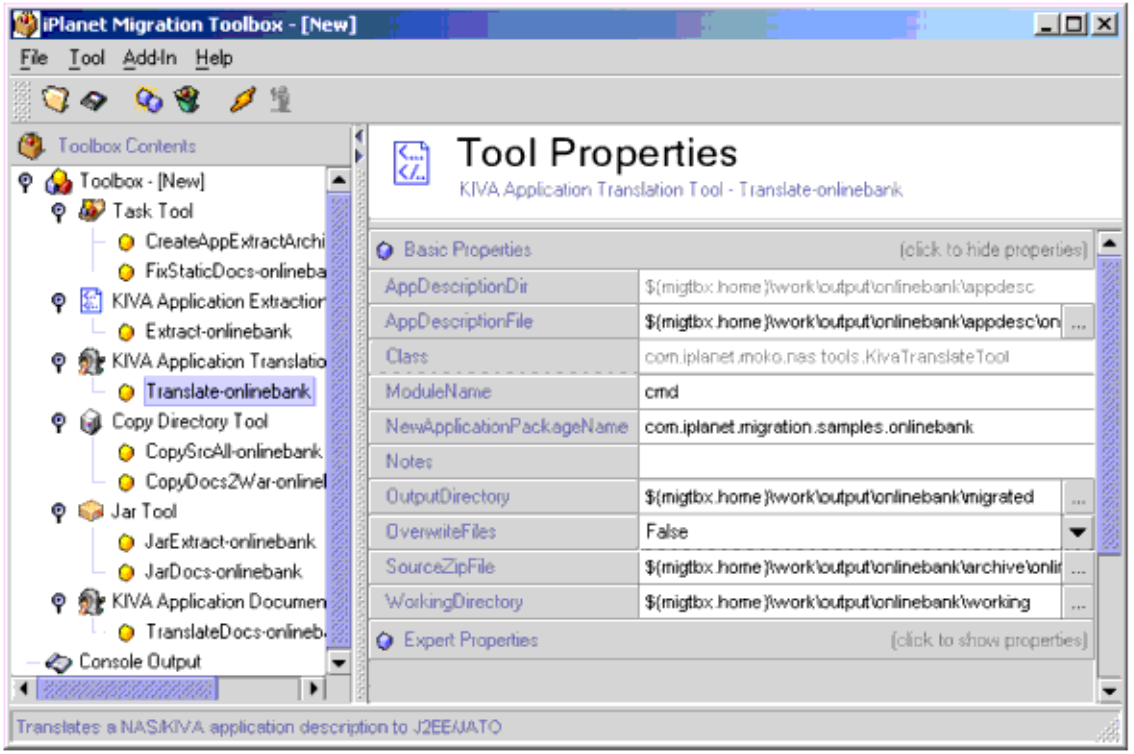
「OK」をクリックして必要なツールの生成を完了します。アドインの結果、抽出ツール、変換ツール、およびアプリケーション抽出アーカイブ自動生成とドキュメント自動変換で使用されるオプションツールが作成され、ツールボックスに必要なツールがすべて揃います。各ツールの分岐を選択すると、詳細なヘルプが右側のフレームに表示されます。ヘルプにはツールの各プロパティの説明があります。ツールの各インスタンスをクリックすると、Bean プロパティパネルが右側のフレームに表示されます。基本プロパティおよび高度なプロパティを編集できます。

Task Tool はリストに記載された他のツールを単純に順番どおり実行します。ツールを個別に実行すれば、コンソールへの出力を注意深く確認することができ、より詳細な情報を得ることができます。

抽出ツールのプロパティを以下に示します。


iPlanet Migration Toolbox - [New]
File Tool Add-In Help
Toolbox Contents
Toolbox - [New]
Task Tool
CreateAppExtractArchi
FixStaticDocs-onlineba
KIVA Application Extractior
Extract-onlinebank
KIVA Application Translatio
Translate-onlinebank
Copy Directory Tool
CopySrcAll-onlinebank
CopyDocs2War-onlinel
Jar Tool
JarExtract-onlinebank
JarDocs-onlinebank
KIVA Application Documen
TranslateDocs-onlineb
Console Output
Tool Properties
KIVA Application Extraction Tool - Extract-onlinebank
Basic Properties
(click to hide properties)
AppDescriptionFileName
onlinebank.xml
Class
com.iplanet.moko.nas.tools.KivaExtractTool
ExcludeExtensions
class,bk,mf
JavaFileCharacterEncoding
ASCII
Notes
OutputDirectory
$(migtbx.home)\work\output\onlinebank\appdesc
OverwriteFiles
False
QueryFileCharacterEncoding
ASCII
SourceZipFile
$(migtbx.home)\work\output\onlinebank\archive\online
TemplateExtensions
html,htm
Extracts a NAS/KIVA application to an XML description file

変換ツールのプロパティを以下に示します。



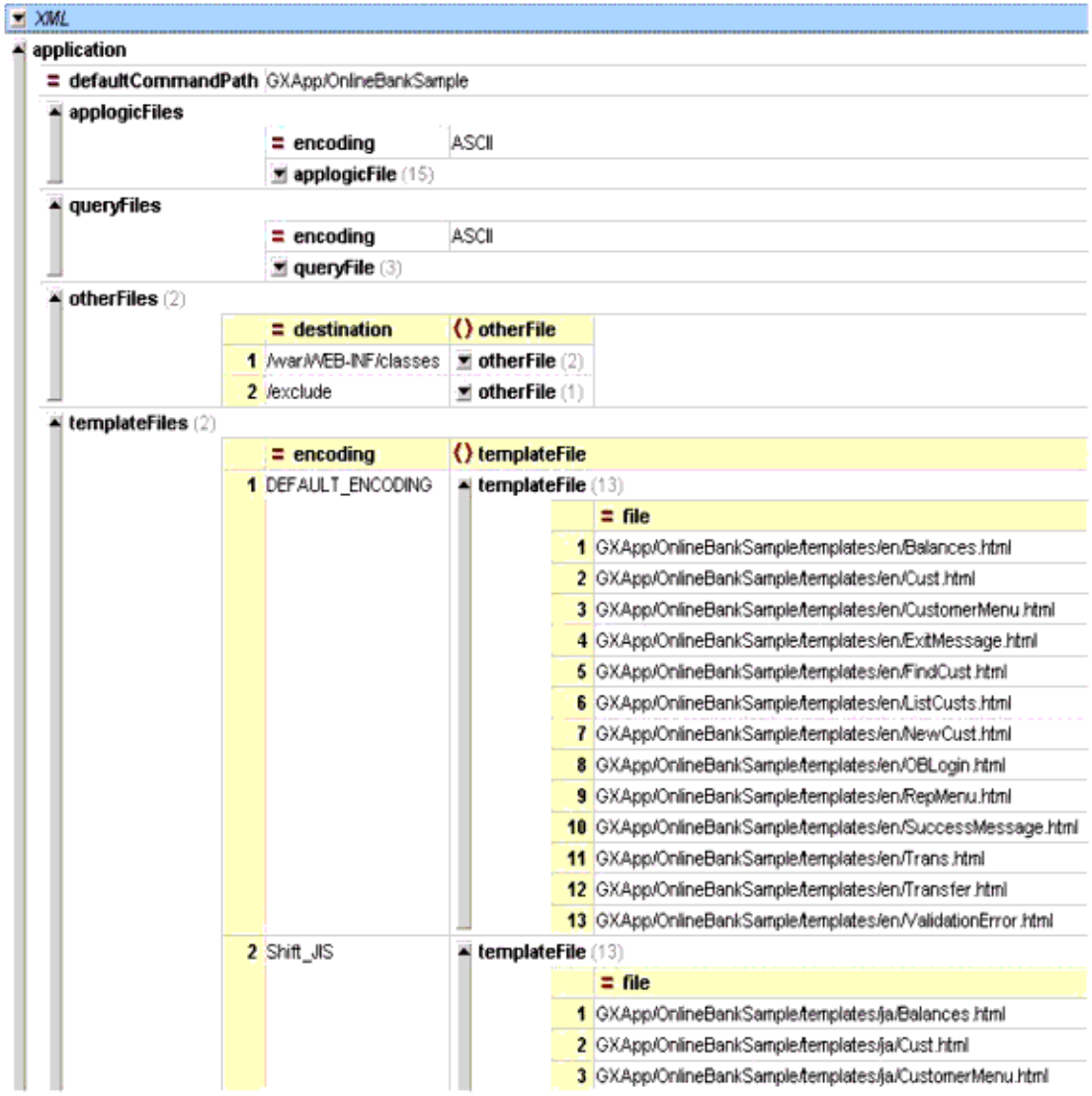
12. CreateAppExtractArchive-onlinebank Task ツールを起動します。このツールは CopySrcAll-onlinebank と JarExtract-onlinebank の各ツールを順番に実行し、アプリケーション抽出アーカイブを生成します。

    ```
    %MIGTBX_HOME%¥work¥onlinebank¥archive¥onlinebankApps.jar
    ```

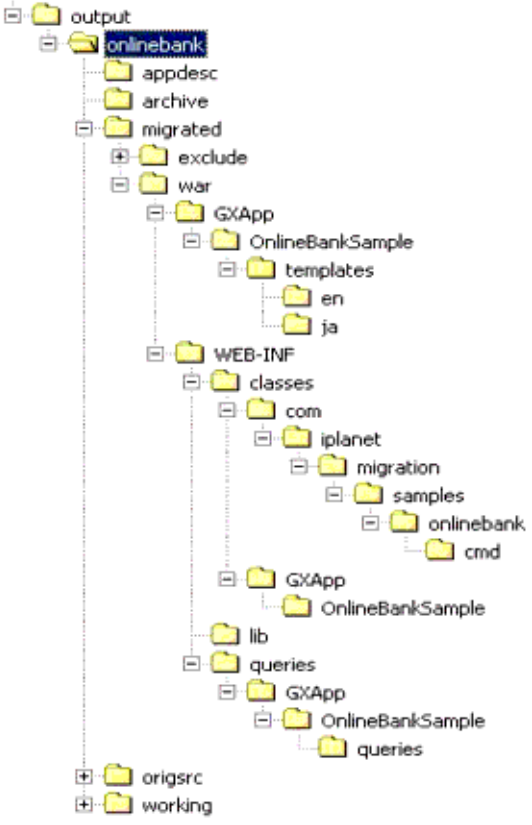
13. Extract-onlinebank を起動します。このツールは非常に高速に実行されます。ツール実行のトレースがコンソールフレームに表示されます。アプリケーション抽出ファイルを内観し、GXR ファイルを収集して、アプリケーションを記述する XML ファイルを生成します。このアプリケーション記述子の内容を確認し、必要に応じてファイルの構成が正しくなるように編集します。編集することで、正しいエンコードを含むアーカイブの各ファイルの後処理を Translation tool が明確に認識できるようになります。iMT 1.2.3 の Extraction tool は HTML テンプレートのエンコードを自動認識します。アプリケーション記述子の内容を確認し、各テンプレートに対して正しいエンコードが選択されていることを確認します。アプリケーション記述子は次の場所に置かれます。

    ```
    %MIGTBX_HOME%¥work¥onlinebank¥appdesc¥onlinebank.xml
    ```

XML エディタでこのファイルを慎重に確認し、編集することをお勧めします。次の図は XML Spy でこのファイルの一部を表示したものです。



14. Translate-onlinebank を起動します。時間は抽出よりも少し長くなり、またアーカイブの AppLogic ソースファイル、Java ファイルおよび Html テンプレートの数によって変わります。変換時には常にコンソールの出力に注意し、エラーが発生していないかどうかを確認します。Translation tool はエラーが発生すると処理をスキップし、残りのアプリケーションの変換を進めます。トレースの出力が大きくなれば警告やエラーを見逃しやすくなります。高度なプロパティを変更してデバッグと詳細なトレースを有効にし、変換の詳細を確認することができます。正規表現マッピング規則の内部呼び出しやエレメント処理の使用などが確認対象となります。変換結果は出力ディレクトリの下の migrated ディレクトリに置かれます。



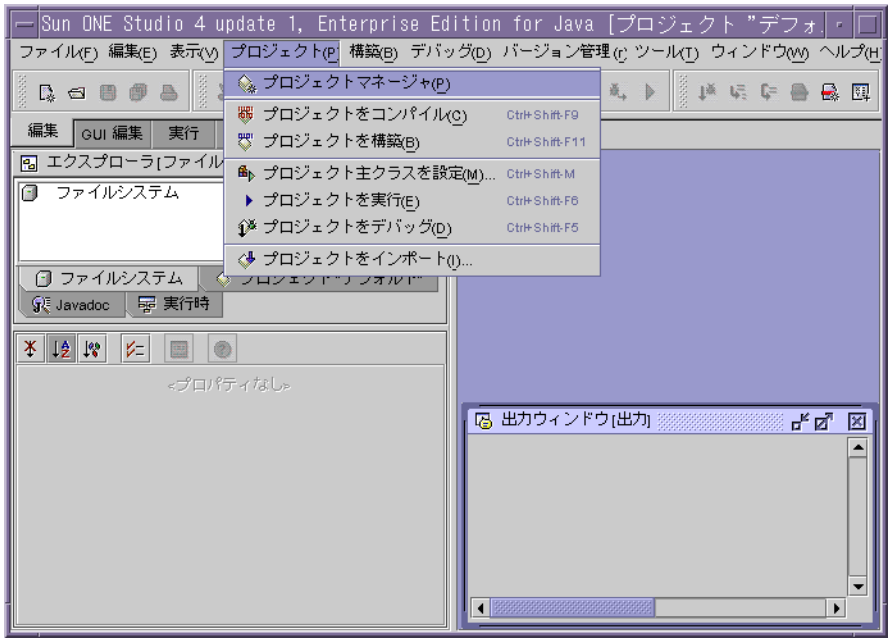
完全な J2EE JATO Web アプリケーションが migrated/war に作成されます。

15. FixStaticDocs-onlinebank Task ツールを起動します。このタスクは、JarDocs-onlinebank、TranslateDocs-onlinebank、および CopyDocs2War-onlinebank を順番に呼び出し、AppLogics の静的コンテンツ URL を修正し、コンテンツを WAR のドキュメントルートにコピーします。

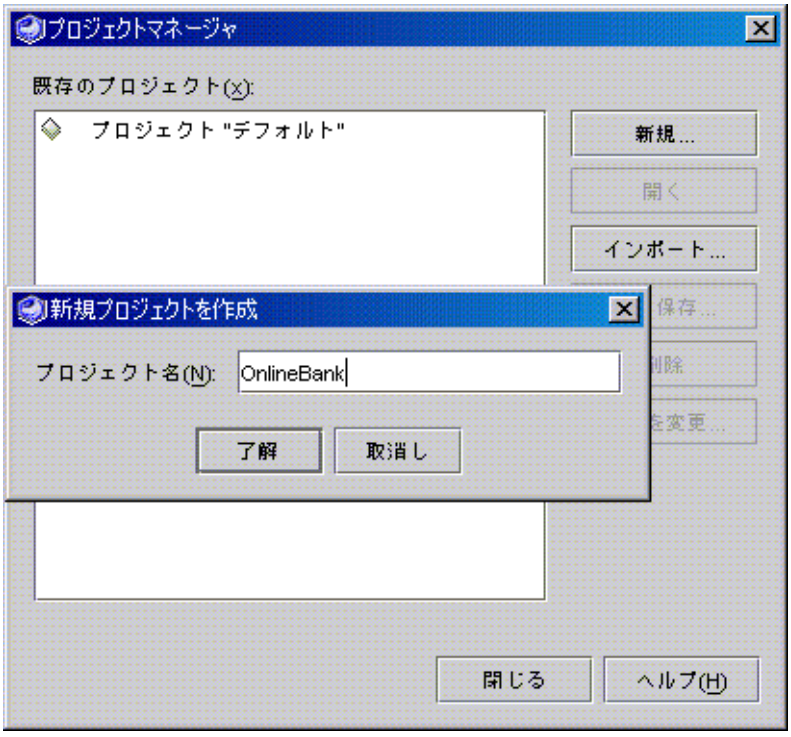
    自動移行はこの時点で完了します。次に手動移行を開始します。

    手動移行を進めていく最も簡単な方法は、Web アプリケーションを J2EE IDE にロードすることです。この例では Forte for Java EE (FFJ) を使用しています。

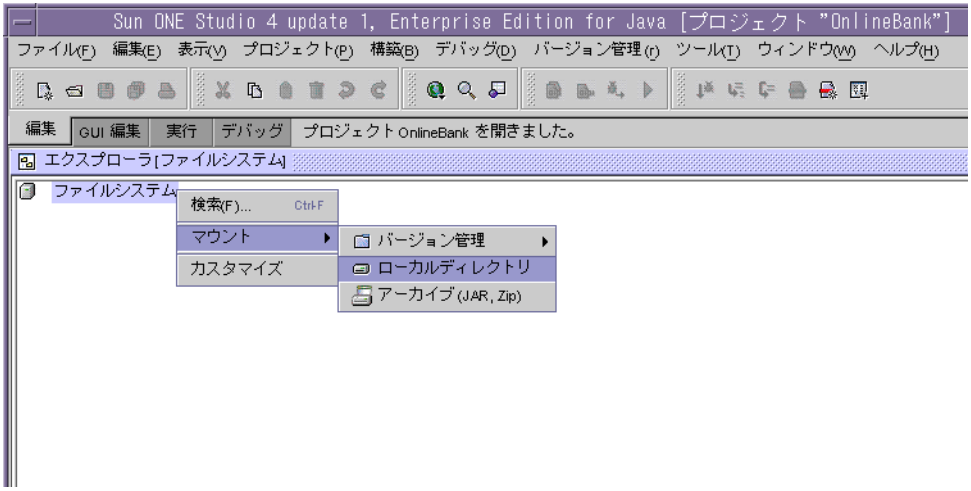
16. FFJ 4.0 を起動し、OnlineBank という名前の新しいプロジェクトを作成します。新しいプロジェクトには既存のファイルシステムを置かないようにします。「プロジェクト」をメニューから選択し、「プロジェクトマネージャ」をクリックします。



17. 「プロジェクトマネージャ」ウィンドウで「新規」をクリックし、プロジェクト名を入力します。ここでは OnlineBank を入力しています。

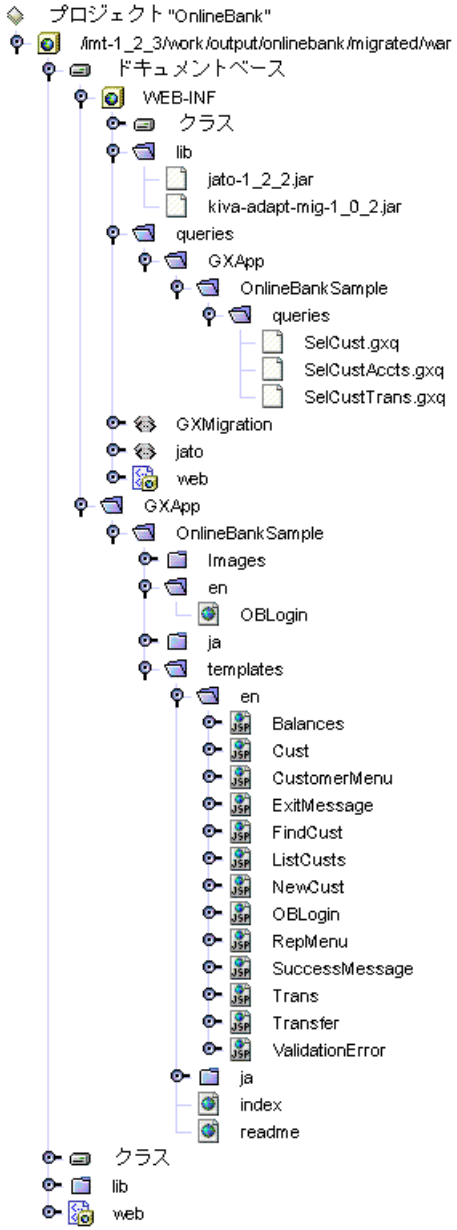


18. 「ファイルシステム」アイコンを右クリックし、エクスプローラパネルで「マウント」>「ローカルディレクトリ」を選択します。
${migtbox_home}¥work¥output¥onlinebank¥migrated¥war を選択し、「OK」をクリックします。Forte ではこのディレクトリを標準の WAR ディレクトリとして認識し、WAR ビューをエクスプローラに表示します。

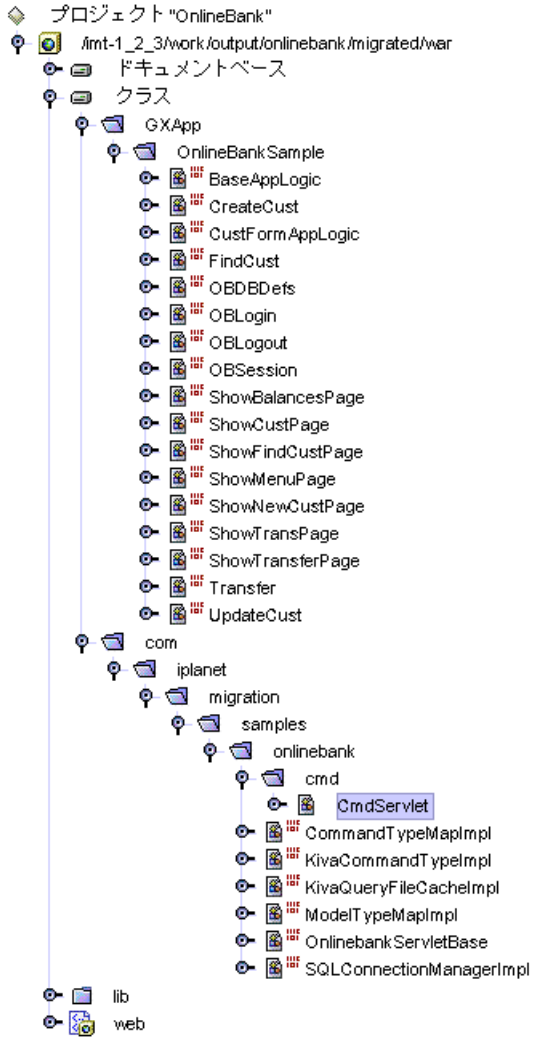


FFJ では、ファイルシステムという用語を、プロジェクトのクラスパスのエントリを指す用語として使用します。WAR ディレクトリのマウント時に自動的にクラスパスの一部となるのは、`./war/WEB-INF/classes` に置かれている war ファイルだけではありません。`./war/WEB-INF/lib` に置かれている ZIP および JAR の各ライブラリも追加されます。下記の OnlineBank プロジェクトのファイルシステムを参照してください。

新しい Web アプリケーションのドキュメントルートは以下の位置になります。JSP に変換された静的コンテンツおよび HTML テンプレートを確認してください。

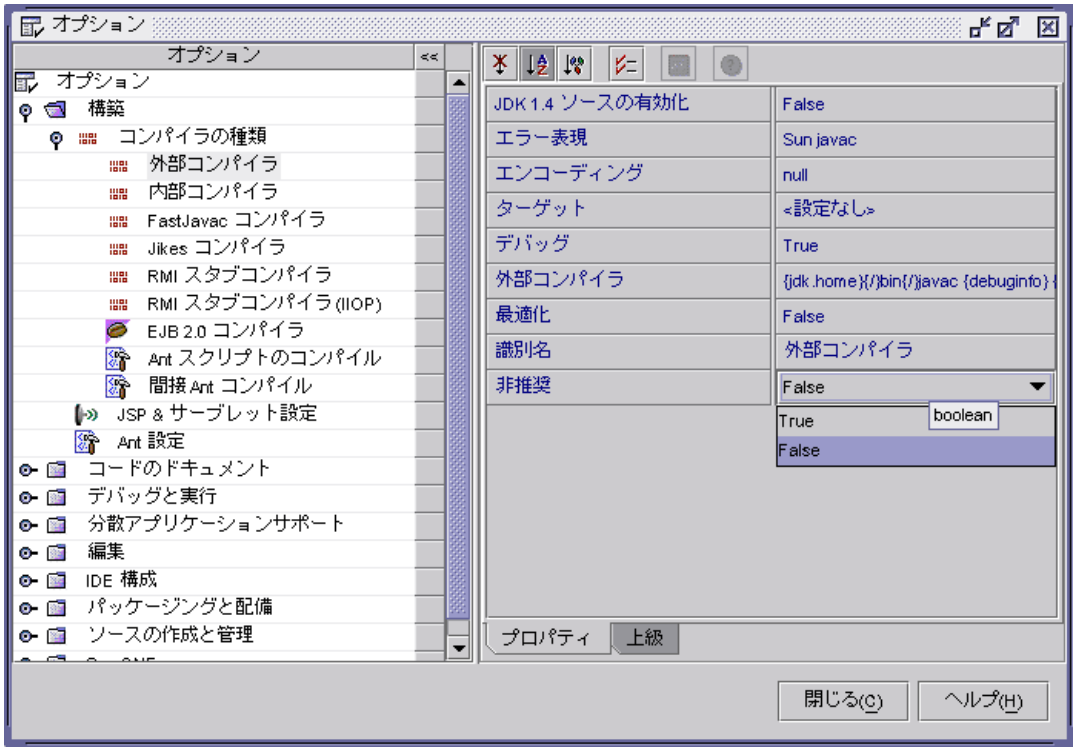

プロジェクト "OnlineBank"
/imt-1_2_3/work/output/onlinebank/migrated/war
ドキュメントベース
WEB-INF
クラス
lib
jato-1_2_2.jar
kiva-adapt-mig-1_0_2.jar
queries
GXApp
OnlineBankSample
queries
SelCust.gxq
SelCustAccts.gxq
SelCustTrans.gxq
GXMigration
jato
web
GXApp
OnlineBankSample
Images
en
OBLogin
ja
templates
en
Balances
Cust
CustomerMenu
ExitMessage
FindCust
ListCusts
NewCust
OBLogin
RepMenu
SuccessMessage
Trans
Transfer
ValidationError
ja
index
readme
クラス
lib
web

Java クラスは Web アプリケーションの以下の位置に新たに置かれます。元の Java ソースが元のパッケージ形態を維持していることを確認してください。AppLogics は JATO コマンドに変換されます。影響を受けるコードはごくわずかです。新しい JATO ソースファイルは Translation tool のプロパティで指定された新しいパッケージに置かれます。



Java ソースはコンパイルする必要があります。ここで重要な点は、コンパイラの非推奨 (deprecation) フラグを設定することです。Translation tool は KFC adaption library のデバッグまたは「移行」バージョンを WAR に自動的に配備します。このライブラリを使用し、非推奨 (deprecation) フラグを指定して変換されたアプリケーションをコンパイルすると、「対象外の」API を使用しているコードの各行に対してレポートを生成します。この目的は、できるだけ早くコンパイルを完了し、手動移行に必要なタスクのレポートを生成することです。アプリケーションが「対象外の」API を使用していても、コンパイルすれば実行はできます。ただし、null や EXE.FAILURE を返すなどの対象外 API が動作しないため、正常には動作しません。アプリケーション全体の移行は負荷が高い作業であるため、このような段階的移行およびテストは有効な手段です。つまり、移行される AppLogic JATO コマンドは 1 つずつテストできます。また、非推奨 (deprecation) レポートで必要な作業を特定することができるという利点もあります。

19. 外部コンパイラをコンパイラとして指定してプロジェクトのプロパティを編集し、deprecation を true に設定します。「ツール」をメニューから選択し、「オプション」をクリックします。最初に「構築」、次にその下の「コンパイラの種類」ノードを展開し、「オプション」ウィンドウの「外部コンパイラ」の「非推奨 (deprecation) 」を true に設定します。



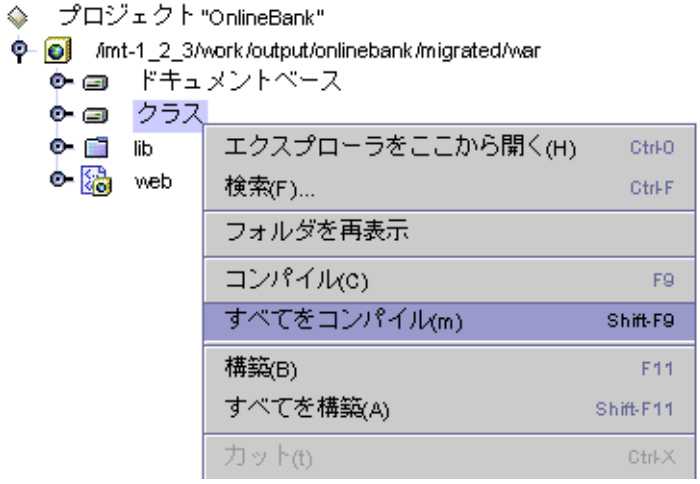
20. エクスプローラ の「プロジェクト」で「クラス」の分岐を選択し、右クリックしてメニューを開き、「すべてをコンパイル」を選択します。AppLogics などの移行されたコードは次のディレクトリに置かれているものが処理対象になります。

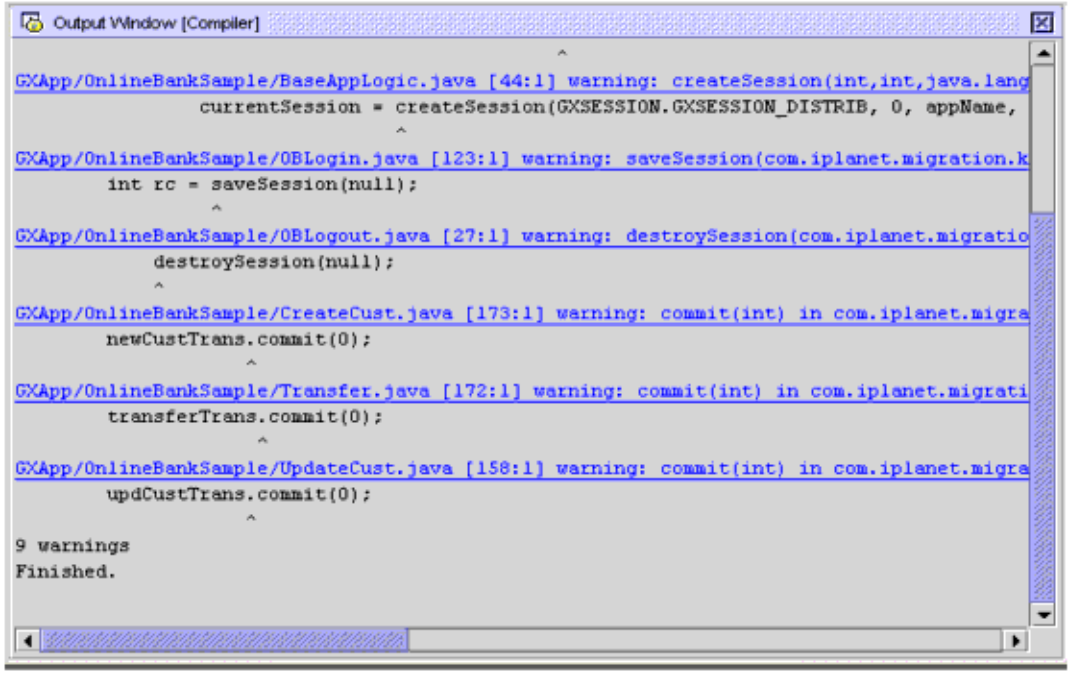
    ```
    ${migtbox_home}¥work¥output¥onlinebank¥migrated¥war¥WEB-INF¥classes¥
    ```

    新しく生成された JATO インフラストラクチャは次のディレクトリに置かれているものが処理対象になります。

    ```
    ${migtbox_home}¥work¥output¥onlinebank¥migrated¥war¥WEB-INF¥classes¥com
    ```

    コンパイルはただちに実行されます。





OnlineBankSample では**対象外のセッション KFC API が 6 箇所で**、iMT のバージョンでは **ITrans の対象外「コミット」メソッドが 3 箇所で**指定されています。

対象外 API の中で最もよく使用されるのがセッション API です。KIVA Application Server および KFC では、任意のセッション API に対して ISessionIDGen 参照を設定できます。このインタフェースでセッション ID および関連する動作を制御できます。J2EE にはこのような機能はありません。アプリケーションが ISessionIDGen を使用している場合、その部分については手動で再設計する必要があります。ほとんどの開発者は API に null オブジェクト参照を設定し、この機能を使用しません。それでもやはり、すべての KFC「セッション」API にこのパラメータが必要であり、ISessionIDGen タイプは対象外であるため、すべての KFC「セッション」API は対象外になります。ほとんどの対象外メソッドに対して、ISessionIDGen パラメータを必要としない API が提供されています。このような対象外のセッション API を使用している場合は、移行時に変更を行い、代わりの API を使用するようにします。通常、これらのセッション API はアプリケーション内の 2、3 箇所にまとまって存在するので、手動変更作業の負荷はそれほど高くありません。セッション API では、特別に考慮する点が 2 つあります。IAppLogic.saveSession(ISessionIDGen) の代わりになるメソッドはありません。J2EE には HttpSession の「保存またはフラッシュ」の概念がないためです。この API は削除されます。IAppLogic.createSession(int, int, String, String,

ISessionIDGen) API の代わりになる API にはパラメータがありません。J2EE のサーブレット API は、KFC API のような制御機能を開発者に提供していません。ただしコンテナのベンダーは、便利な設定機能や配備記述子およびアプリケーションサーバー設定による HttpSession を提供している場合があります。

ITrans.commit メソッドの引数は 1 つですが、KIVA では使用されません。この API を引数のない API に置き換えてあります。CreateCust.java、Transfer.java、および UpdateCust.java の 3 つのメソッドから、値「0」を削除する必要があります。

21. OnlineBankSample アプリケーションでは、BaseAppLogic.java、OBLogin.java、および OBLogout.java でセッション API を使用しています。以下の変更が必要になります。

BaseAppLogic.java、38 行目

```
ISession2 currentSession = getSession(); // getSession(0,
appName, null);
```

BaseAppLogic.java、44 行目

```
currentSession = createSession();
//createSession(GXSESSION.GXSESSION_DISTRIB, 0, appName,
null, null);
```

OBSession.java、52 行目

```
// result = m_logic.saveSession(null);
```

OBLogin.java、123 行目

```
int rc = GXE.SUCCESS; // saveSession(null);
```

OBLogout.java、27 行目

```
destroySession(); // destroySession(null);
```

通常は HTML ソースの手動変更が必要になります。HTML テンプレートのソースまたは JSP の変更が必要な場合もあります。変更はアプリケーションごとに異なります。ほとんどの手動処理は iMT で系統的に行うことができます。繰り返されるパターンを検出し、正規表現マッピングツールを使用して、作業を自動化できます。マークアップで変更が必要になるのはほとんどが URL のパスです。動的コンテンツから静的コンテンツへのリンクで絶対パスを使用している場合は、Web アプリケーションコンテキストが付加されることでパスが無効になります。

22. ExitMessage.jsp の両方のバージョンを編集します。静的コンテンツを WAR ファイルに移動しているため、動的コンテンツから静的コンテンツへの参照は無効になります。コンテンツが WAR ファイル外に配備されればこれらの参照は正しくなります。絶対参照の前には「..」が追加されます。ExitMessage.jsp は、実際はサーブレットコンテキスト内の /cmd サーブレットマッピングのコンテキストに存在するため、パスを 1 セグメント上に移動するだけでサーブレットコンテキストのドキュメントルートに戻ることができます。

    ```
    /GXApp/OnlineBankSample/templates/en/ExitMessage.jsp
    ```

    ```
    /GXApp/OnlineBankSample/templates/ja/ExitMessage.jsp
    ```

    15 行目 (html -> jsp のリンクおよびパス) (以下参照)

    ```
    href="../GXApp/OnlineBankSample/en/OBLogin.html"> Back to Login
    Page </a><br>
    ```

23. (省略可能) `/WEB-INF/web.xml` を編集し、ルートコンテキスト要求時の自動起動を許可します (以下参照)。開始ファイル要素をサーブレットマッピングと taglib 要素の間に追加する必要があります。

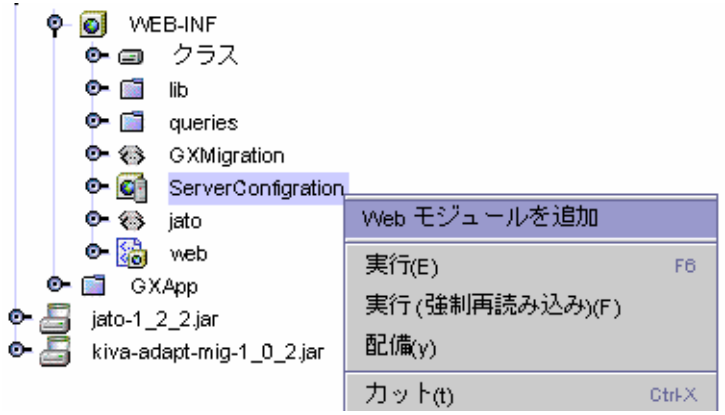
    ```
    </servlet-mapping>
    <welcome-file-list>
    <welcome-file>
    GXApp/OnlineBankSample/index.html
    </welcome-file>
    </welcome-file-list>
    <taglib>
    ```

    手動移行の主な作業はアプリケーション内の URL の確認です。静的なコンテンツと動的なコンテンツのリンクは、J2EE 配備の移植を可能にするため相対パスに変更する必要があります。JavaScript の変更も必要になります。

    手動移行処理が終了し、最終的に生成される Web アプリケーションは任意の J2EE Web コンテナに配備できます。FFJ では WAR ファイルをエクスポートし、iAS 6.5 に配備できます。また、Web アプリケーションは FFJ に組み込まれた TomCat サーバーで直接実行できます。

24. サーバーモジュールグループを FFJ に追加します。エクスプローラの「WEB-INF」分岐を右クリックし、「新規」->「JSP& サーブレット」->「Web モジュールグループ」を選択し、サーバーグループを追加します。ウィザード画面のデフォルト値をそのまま使用し、「完了」をクリックします。エクスプローラの「WEB-INF」に、「ServerConfiguration」という名前の新しい要素が表示されま

す。エクスプローラの「ServerConfiguration」分岐を右クリックし、「Web モジュールを追加」を選択して現在の Web アプリケーションを追加します。サーブレットコンテキスト名が「Web モジュールを追加」ウィンドウに表示されます。たとえば「Demo」などが表示されます。



25. エクスプローラで「ServerConfiguration」分岐を右クリックし、「実行」を選択して FORTE で実行します。

第 4 章

NetDynamics から Sun ONE AS 7 への移行

NetDynamics のアプリケーションは、iPlanet Migration Toolkit (iMT 1.2.3) で J2EE Web モジュールに移行できます。結果として生成される Web モジュールは任意の J2EE Web コンテナに配備し、実行できます。

はじめに

先に進む前に、`%MIGTBX_HOME%/bin/readme.txt` を読んで、それぞれの環境に関連する最新情報と問題点を把握してください。この readme ファイルには Migration Toolbox の正しいインストール方法と設定方法、および動作環境についての情報も記載されています。このマニュアルに記載されている移行プロセスを開始する前に、これらのインストールおよび設定を完了し、環境を整備しておく必要があります。

`%MIGTBX_HOME%` は iPlanet Migration Toolbox (iMT) をインストールまたは解凍したディレクトリを表します。

このマニュアルでは、NetDynamics アプリケーションを J2EE に移行するための、最低限のプロセスだけを説明します。移行プロセスの完全なリファレンスとしては作成されていません。この理由は主に、移行プロセスがそれぞれの環境によって大きく異なるためです。代わりに、このマニュアルでは iPlanet Migration Toolbox (iMT) による基本的な移行プロセスを理解するために必要な情報を提供します。

移行準備

移行プロセスの概要

NetDynamics で作成されたプロジェクトを同等の J2EE プロジェクトに完全に移行するための主要なフェーズは 2 種類あります。1 つは自動移行フェーズ、もう 1 つは手動移行フェーズです。自動移行には抽出および変換の 2 つの手順があります。

自動移行フェーズ

このフェーズでは、手動による NetDynamics プロジェクトの移行準備、および iMT による自動抽出および変換を行います。このフェーズでは、NetDynamics で作成されたプロジェクトを部分的に、場合によっては完全に移行し、全く形態の異なるサーブレットや JSP などの J2EE に準拠するコンポーネントで構成されるアプリケーションを生成します。

変換プロセスでは、元の NetDynamics プロジェクトのコンポーネント構造を完全に複製します。このプロセスではまた、プロジェクトの INTRP ファイルにある宣言型プロパティ情報を使用し、移行アプリケーションに同等の機能を生成します。ただし、現在の変換フェーズでは、NetDynamics Spider API で書かれたコードの J2EE への自動移行は行いません。この移行が手動移行フェーズの主要な作業になります。変換フェーズでは、元のソースコードを移行先の適切な場所にそのままコピーします。たとえば NetDynamics `onBeforeDisplay` イベントハンドラは、移行アプリケーションのイベントハンドラメソッドが配備される場所にコピーされます。

手動移行フェーズ

自動フェーズでアプリケーションがどの程度移行されるかは、元のアプリケーションの宣言型の機能と API 機能の割合によって決定されます。まれに、機能がすべてプロジェクト内で宣言されている場合があります。このようなプロジェクトは完全に自動移行できる場合が多く、手動の操作を全く行わなくてもすぐに J2EE コンテナに配備し、実行することができます。これに対して、内部で宣言されている機能が少ないプロジェクトは、J2EE アプリケーションとして動作させるためには多くの手作業が必要になります。

一般に手動移行フェーズでは、自動移行されたアプリケーションの出力内容の確認、および Spider API 固有のコードの J2EE 固有のコードへの移行を行います。通常、このプロセスではアプリケーションまたはそのアーキテクチャの再設計は不要です。主な作業は API コールを 1 対 1 でマッピングすることです。これが可能になるのは、自動変換プロセスをターゲットにした J2EE に準拠する強力な Web アプリケーションをベースとしている JATO を使用しているためです。

作業環境の準備

先に進む前に次の事項を確認します。

1. iPlanet Migration Toolbox がインストール済みかどうかの確認
    - バージョンアーカイブのターゲットディレクトリへの解凍。方法は `readme.txt` ファイルを参照してください。

- Toolbox アプリケーション起動によるインストールのテスト。`%MIGTBX_HOME%/bin` ディレクトリで `toolbox.bat` を実行してください。空の Toolbox が表示されます。何も表示されない場合は、Migration Toolbox が正常にインストールされているかどうか、`%MIGTBX_HOME%/bin/setenv.bat` の環境設定がすべて適切かどうかを確認してください。

2. クラスのバージョン問題を回避するため、Toolbox アプリケーションを実行する時にはすべての JAR ファイルを JDK の拡張ディレクトリ、`%JAVA_HOME%/jre/lib/ext` から必ず削除することをお勧めします。Toolbox 実行に必要なクラスはすべてバージョンに含まれています。拡張ディレクトリの JAR ファイルの名称を変更するだけでは不十分であり、別の場所に移動する必要があります。

3. 移行する NetDynamics プロジェクトを `%MIGTBX_HOME%/work/NDProjects` ディレクトリ、または他の適切なディレクトリにコピーします。このディレクトリは NetDynamics プロジェクトディレクトリとして以下で参照されます。このディレクトリは、同じマシンの NetDynamics インストールで使用される実際のプロジェクトディレクトリをそのまま指定できますが、別のディレクトリを指定しても構いません。代わりに指定できるのは移行対象の NetDynamics プロジェクトを配置するディレクトリです。NetDynamics を Migration Toolbox が稼働するマシンにインストールする必要はありません。NetDynamics を iMT が稼働するマシンにインストールする場合は、iMT の動作が NetDynamics のインストールによって影響を受けないことを確認してください。インストールされる NetDynamics のクラスパスがシステム環境変数 `CLASSPATH` で参照されていると、影響を受ける場合があります。iMT が起動される時には、iMT が必要とするクラスパスがシステムクラスパスの最後に追加されます。インストールされている NetDynamics のクラスパスがシステムクラスパスの一部である場合は、iMT は正常に動作しません。

4. ここで、Sun ONE Application Server 7 または他の J2EE に準拠するサーブレットまたは JSP コンテナをインストールできます。

    - サーバーまたはコンテナのインストール手順に従ってインストールを行います。
    - サーバーまたはコンテナを起動し、デフォルトのホームまたはインデックスページを読み込んでインストール状況をテストします。エラーがある場合は、インストールプロセスの問題を解決してから先に進みます。

自動移行プロジェクトの準備

NetDynamics による開発は自由度が大きくなっています。これには利点と欠点があり、欠点としては、すべてのプロジェクトの置き換えを iMT で把握できないということが挙げられます。特に標準外の NetDynamics Spider API を使用している場合、またはマニュアルに記載されていない標準外の方法でこの API を使用している場合がこ

れに当てはまります。したがって、アプリケーションによっては iMT によって移行される前に手動の準備が必要になります。特別な問題の発生しやすい機能がプロジェクト、または一連のプロジェクト全体で使用されている場合は、この準備が重要になります。

自動移行の 2 つのフェーズの中では、プロジェクトの抽出で最初に問題が発生しやすくなっています。これは単に前述の問題の結果であり、珍しいことではありません。抽出時に問題が発生しないプロジェクトのほうが多く、プロジェクトからのアプリケーション記述の抽出が終了すれば、変換ではほとんど問題は発生しません。

プロジェクト抽出ランタイムと NetDynamics 実行時環境の違い

iMT は組み込み型の NetDynamics Connection Processor (CP) を使用し、プロジェクトからの情報のインスタンス化および抽出を行います。プロジェクト側から見た場合、通常の NetDynamics 5.x サーバー環境内でインスタンス化が行われます。ただし、抽出の実行時環境は NetDynamics サーバーの環境とは大きく異なります。特に JDBC サービス、PE サービス、および PAC は iMT の組み込み型ランタイムでインスタンス化された環境では利用できません。ただしそれでも問題なく必要な情報を抽出できます。

プロジェクトオブジェクトによっては、コンストラクタ、静的イニシャライザ、初期化イベント、または Spider 以外のスレッドでこれらのランタイムの機能に依存したタスクを実行する場合があります。iMT では、NetDynamics 4/5.x の `onBeforeInit` および `onAfterInit` イベントを開始しないようにし、これらのイベントのカスタマコードを初期化時に実行しないようにしています。ただし、静的イニシャライザ、オーバーライドされる `init()` メソッド、NetDynamics 3.x の `onBeforeInit` および `onAfterInit` イベントなど、初期化時の他のメソッドは実行されます。これらのメソッドのコードが、iMT ランタイム内で正常終了できない動作を実行しようとする場合は、その箇所のコードをコメントアウトする必要があります。コードは元の場所にそのまま置いておくことができます。このコードは変換時に正しい場所に自動的に移動されます。このような問題の発生するケースを最も簡単に特定する方法は、エラーメッセージまたは *Extraction Tool* で生成される例外を参照することです。

NetDynamics Extraction Tool 実行前の注意点

上記の理由から、通常 *Extraction Tool* をプロジェクトに対して実行する場合は最小限の準備だけを行うことをお勧めしています。この場合、抽出処理でエラーが発生し、失敗する可能性は高くなりますが、移行プロセス全体の時間は通常短縮されます。これはツールのエラー情報を使用して問題を特定し、修正するほうが、潜在的な問題をあらかじめ検出して修正するよりも簡単であり、また短期間で修正を行えるためです。ただし潜在的な問題がよく知られている場合はあらかじめ対応する方法が有効です。

ただし、抽出時に発生する問題は他にもいくつかあるため、それを避けるために次の処理を NetDynamics Extraction Tool 実行前に行うことをお勧めします。

- NetDynamics プロジェクトのインスタンスの中には、Studio で開いた場合、または NetDynamics サーバーで実行した場合に、一見正常に見えても、詳しく見ると壊れている参照やプロジェクトオブジェクトを含んでいるものがあることがわかっています。また、壊れたクラスファイルの中には、組み込み型 NetDynamics ランタイムによる対応するオブジェクトの読み込みを妨げ、一見関連がないように見える例外をスローする原因があることがわかっています。したがって、トラブルを防ぐために次の手順を移行前に実行することを強くお勧めします。

  - プロジェクトがクライアントや同僚などの別のソースにある場合は、プロジェクトの `links` ディレクトリが存在し、複数の `.sid` ファイルを含みます。これらのファイルのいくつかをテキストエディタで開き、その中でプロジェクトオブジェクトの名前を指定します。また、必要な外部クラスをプロジェクトに格納します。

  - Studio の自動変換機能でプロジェクトを NetDynamics 5 に変換しておく必要があります。この処理を行うと NetDynamics 5 Studio でプロジェクトが開き、アップグレード確認のメッセージが表示されます。変換時には、Studio はオブジェクトのプロパティをアップグレードし、DataObjects を NetDynamics 5.x 互換のバージョンに変換します。
    重要 : プロジェクトを実際に NetDynamics 5.x で実行する必要はありません。Studio でプロジェクトを変換するだけで十分です。

  - 移行するプロジェクトを NetDynamics 5.x Studio で開き、不足はないか、また移行して問題はないかどうかを確認します。プロジェクトディレクトリ自体もチェックします。たとえば、1 つの NetDynamics ページには、1 つの `<project>.spj` または `<project>Project.spj` および `<project>.class` ファイル、1 つの `<page>.spg`、`<page>.class`、および `<page>.html` ファイルが必要です。また、1 つの `DataObject` には、1 つの `<dataobject>.sdo` および `<dataobject>.class` ファイルが必要です。

  - すべての `.class` および `.ser` ファイルをプロジェクトディレクトリから削除し、プロジェクト全体を再コンパイルします。プロジェクトは NetDynamics 5.x バイナリに対応した形式でコンパイルする必要があります。この場合、最も簡単な方法は Studio の「すべてをコンパイル」コマンドを使用することです。Migration Toolbox アプリケーションの Java Compilation Tool で、NetDynamics 5 バイナリを使用してコンパイルすることもできます。ただしこれはさらに多くの設定処理が必要になるためお勧めしません。

- 可能な場合は、プロジェクトを NetDynamics 5.x でテスト実行します。プロジェクトをサーバーで正常に実行できれば、問題なく移行できる確率はより高くなります。稼働している NetDynamics のコピーがある場合は、CP を設定して移行するプロジェクトをあらかじめ読み込みます。現在の設定から JDBC サービス、PE サービス、およびすべての PAC を削除するためには Command Center を使用し

ます。CP を再起動します。CP が正常に起動した後、NetDynamics および Service Manager (SM) のログをチェックし、例外がスローされていないかどうかを確認します。この時点で例外をスローしているプロジェクトは、抽出時にも例外をスローする可能性があります。

ToolBox サンプルアプリケーションの移行

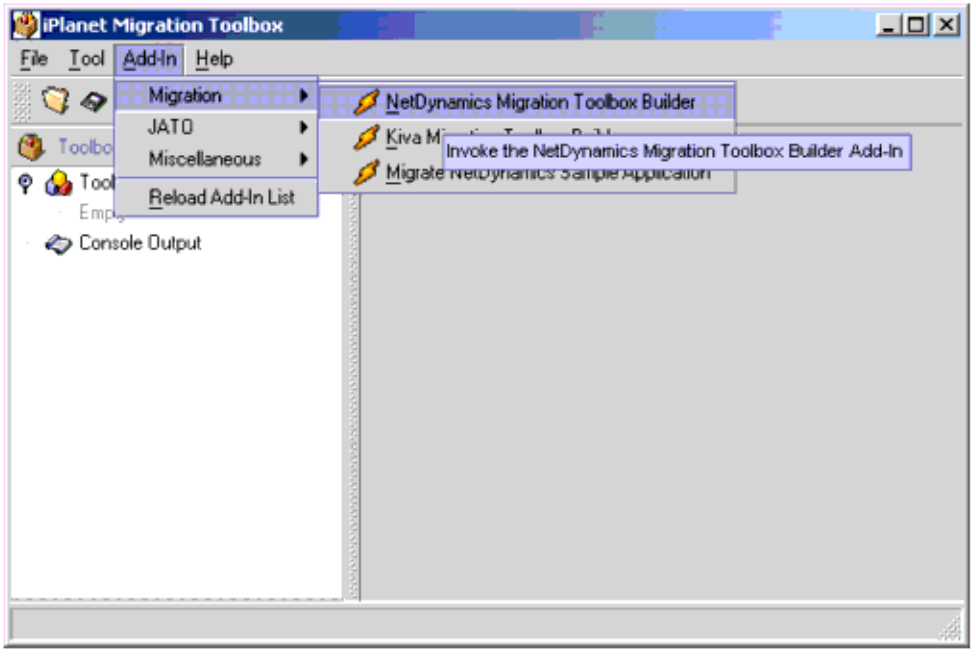
この節では、ToolBox のサンプルアプリケーションの自動および手動移行処理について説明します。

Migration Toolbox の実行

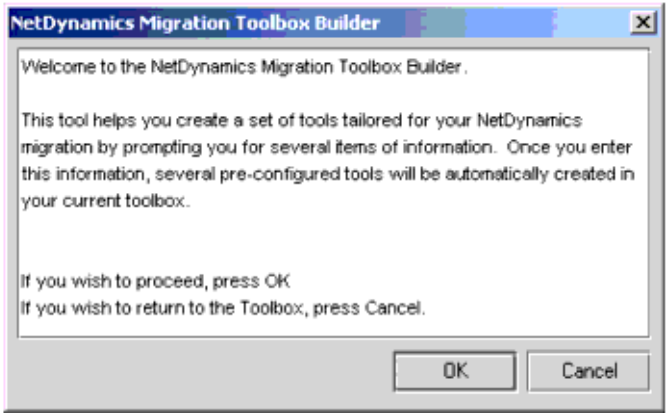
Toolbox アプリケーションが稼働していない場合は、「作業環境の準備」の説明に従ってツールボックスの設定を行います。

Toolbox Builder の作成

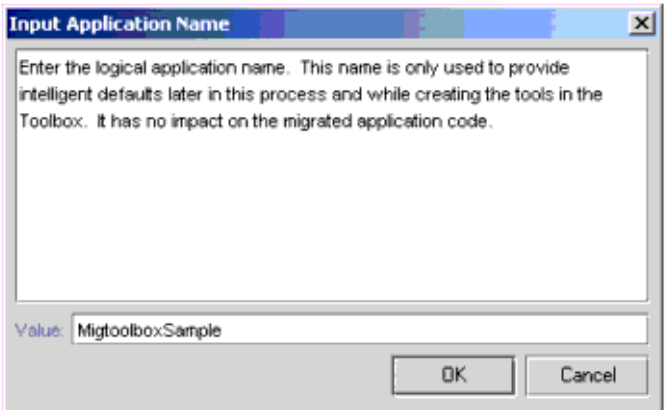
1. ツールボックスを起動し、最初のダイアログで「Migrate an application」オプションを選択し、「OK」をクリックします。Toolbox を起動すると空の (New) ツールボックスが表示されます。メニューオプション「Add-In」->「Migration」->「NetDynamics Migration Toolbox Builder」を選択します。



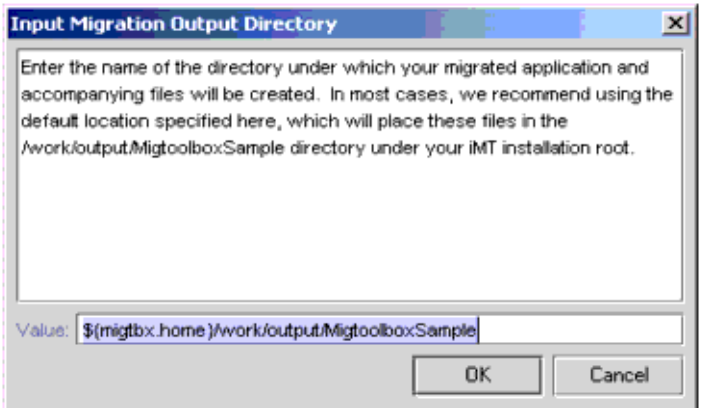
モーダルダイアログウィザードが表示されます。



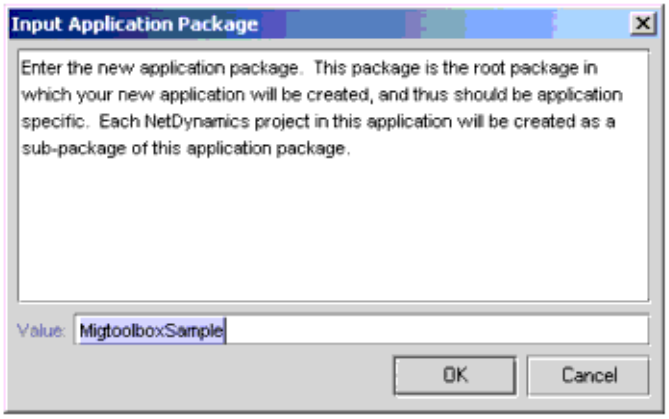
「OK」をクリックしてウィザードの最初の手順に進みます。



2. 移行するアプリケーション名を「Input Application Name」ダイアログボックスに入力します。例では「MigtoolboxSample」と入力しています。「OK」をクリックして次の手順に進みます。

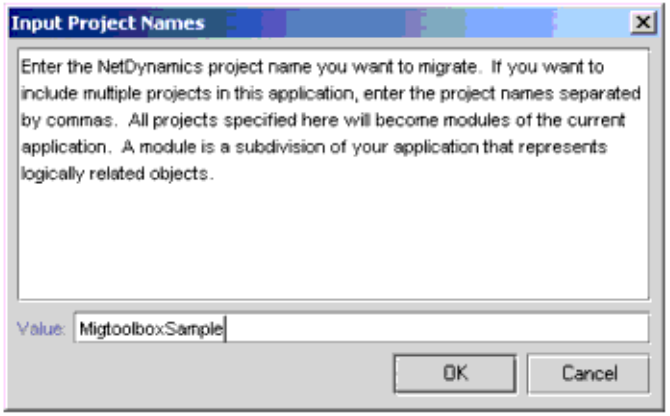


3. iMT で生成されるすべてのデータを格納するディレクトリを入力します。通常はデフォルトの値を変更する必要はありません。この例でもデフォルトの値を使用しています。「OK」をクリックしてウィザードの次の手順に進みます。

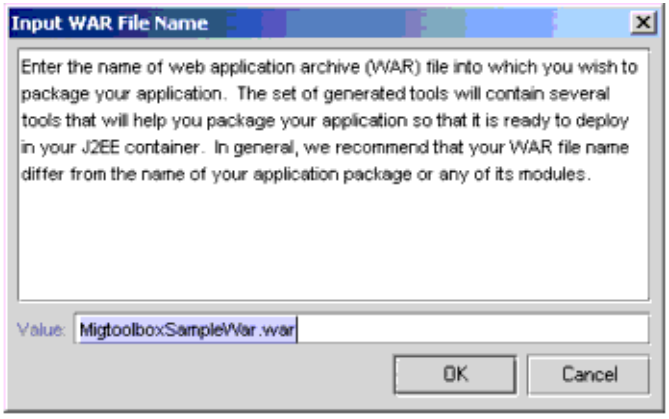


4. 自動 iMT 移行では新しい Java JATO ファイルを含む J2EE インフラストラクチャをいくつか生成します。これらの新しいファイルにパッケージを割り当てる必要があります。元のアプリケーションの既存の Java ソースがパッケージ化されて残る場合でも、これらの新しいファイルにパッケージを割り当てる必要があります。パッケージ名に関する制限はありません。MigtoolboxSample アプリケーションにはデフォルト値が提示されます。

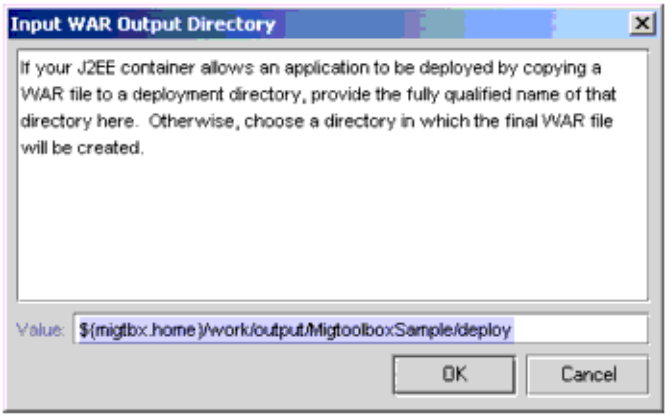
   パッケージを入力し、「OK」をクリックしてウィザードの次の手順に進みます。



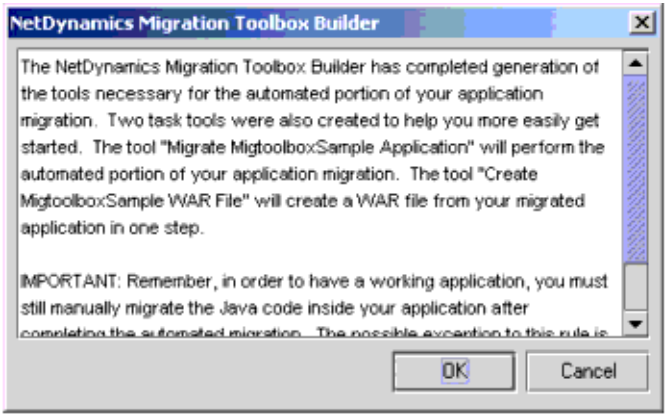
5. 移行するプロジェクト名を入力します。このプロジェクトは {MIGTBX_HOME}¥work¥NDProjects¥ フォルダに配置する必要があります。



6. アプリケーションをパッケージ化する Web アプリケーションアーカイブ (WAR) ファイル名を入力します。このアプリケーションのデフォルト値が提示されます。「OK」をクリックして次の手順に進みます。



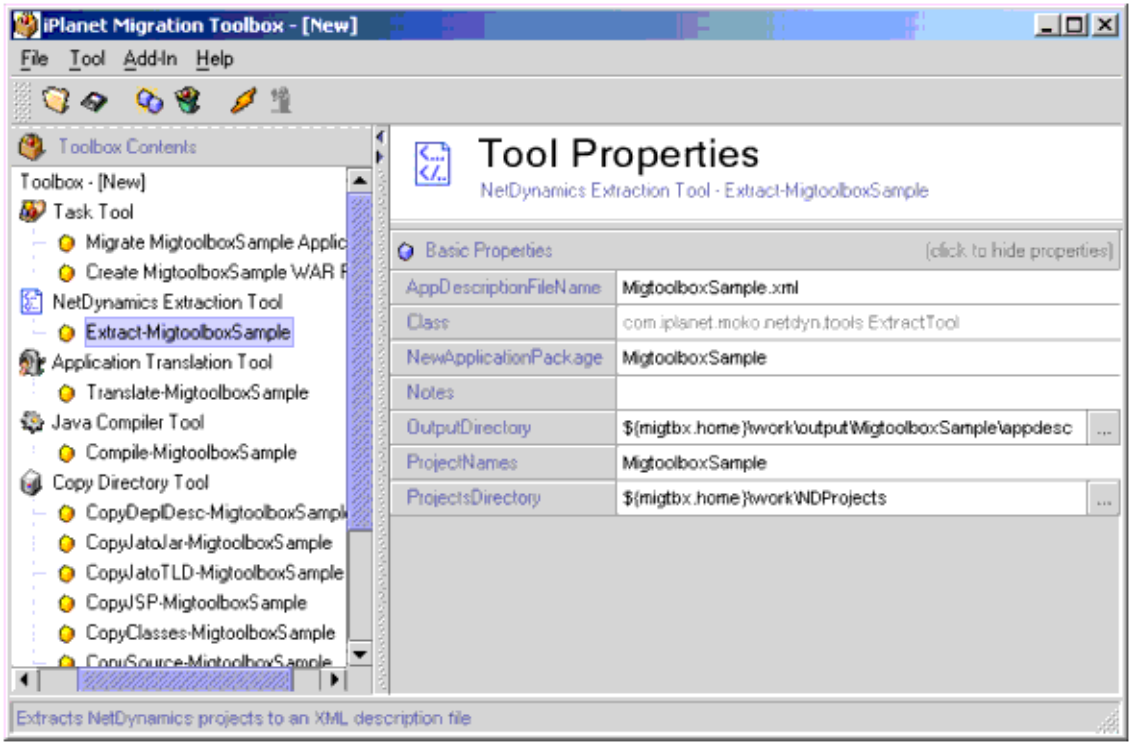
7. iMT による WAR ファイルの生成先となる出力ディレクトリ名を入力します。「OK」をダイアログボックスで選択すると、Toolbox builder がアプリケーションの自動移行に必要な一連のツールを生成します。



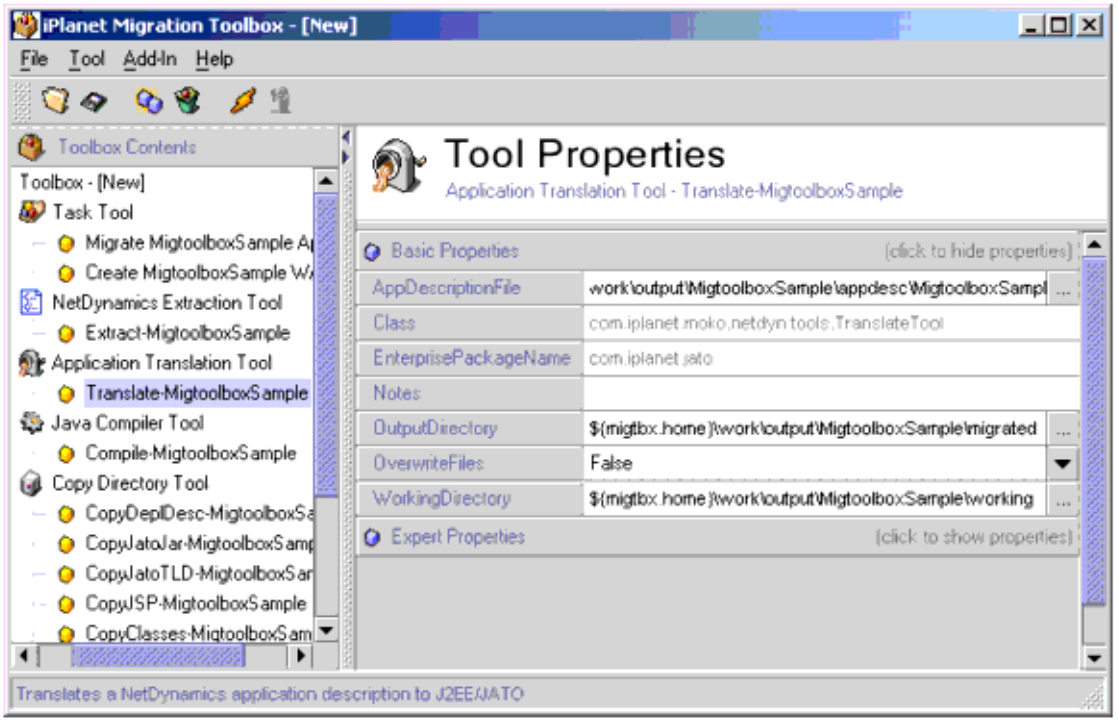
8. 「OK」を選択して NetDynamics Migration Toolbox Builder ウィザードを終了します。アドインの結果、抽出ツール、変換ツール、およびアプリケーション抽出アーカイブ自動生成とドキュメント自動変換で使用されるオプションツールが生成され、ツールボックスに必要なツールがすべて揃います。左側のフレームにあ

る各ツールの分岐を選択すると、詳細なヘルプを右側のフレームに表示します。ヘルプにはツールの各プロパティの説明があります。ツールの各インスタンスをクリックすると、Bean プロパティパネルが右側のフレームに表示されます。基本プロパティおよび高度なプロパティを編集できます。

Task Tool はリストに記載された他のツールを単純に順番どおりに実行します。ツールを個別に実行すれば、コンソールへの出力を注意深く確認することができ、より詳細な情報を得ることができます。抽出ツールのプロパティを以下に示します。



Translation Tool のプロパティを以下に示します。



9. Migrate MigtoolboxSample Application Task ツールを起動します。このツールは Extract-MigtoolboxSample、Translate-MigtoolboxSample、および MapSpider2JATO-MigtoolboxSample の各ツールを順番に呼び出し、移行コードおよびアプリケーション記述ファイル `MigtoolboxSample.xml` を生成します。

10. Create MigtoolboxSample War File Task ツールを起動します。このツールは次に挙げるツールを起動して Web アプリケーションアーカイブ (WAR) ファイルを生成し、アプリケーションを J2EE コンテナに自動配備できるようにします。アプリケーションを J2EE コンテナに配備する場合は、この WAR ファイルだけが必要になります。

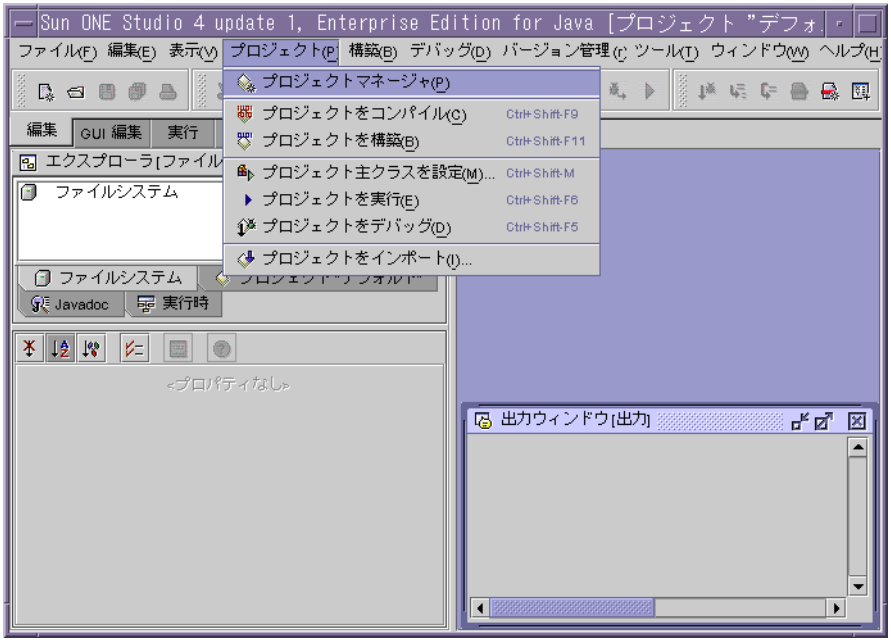
    ```
    CopyDeplDesc-MigtoolboxSample, CopyJatoJar-MigtoolboxSample,
    CopyJatoTLD-MigtoolboxSample, CopyJSP-MigtoolboxSample,
    CopyClasses-MigtoolboxSample, CopySource-MigtoolboxSample,
    JarWarFile-MigtoolboxSample
    ```

11. Compile-MigtoolboxSample ツールを起動し、JATO Foundation classes と新しい J2EE アプリケーションコンポーネントをコンパイルします。実際には JDK で提供される `javac` コマンド行ツールの呼び出しだけを行います。

    自動移行はこの時点で完了します。必要に応じて手動移行を開始します。

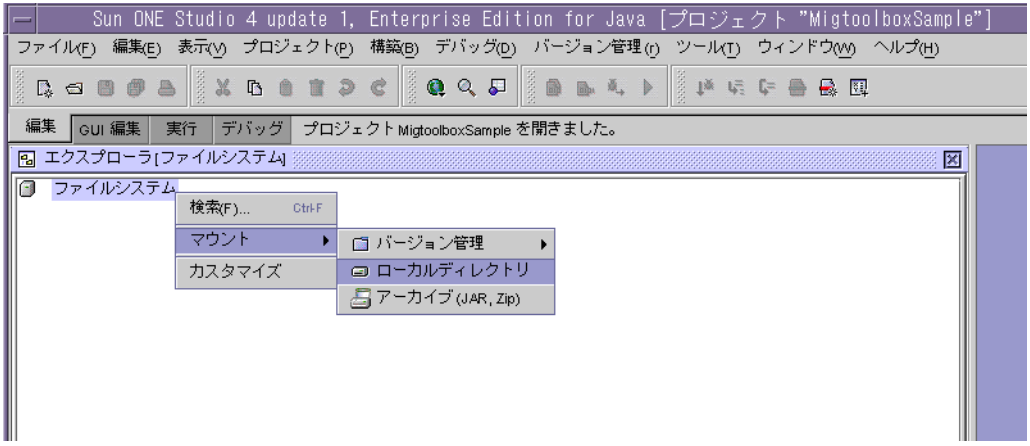
手動移行を進めていく最も簡単な方法は、Web アプリケーションを J2EE IDE に実装することです。この例では Forte for Java EE (FFJ) を使用しています。

12. FFJ 4.0 を起動し、OnlineBank という名前の新しいプロジェクトを生成します。新しいプロジェクトには既存のファイルシステムを置かないようにします。「プロジェクト」をメニューから選択し、「プロジェクトマネージャ」をクリックします。



13. 「プロジェクトマネージャ」ウィンドウで「新規」をクリックし、プロジェクト名を入力します。ここでは MigtoolboxSample を入力しています。

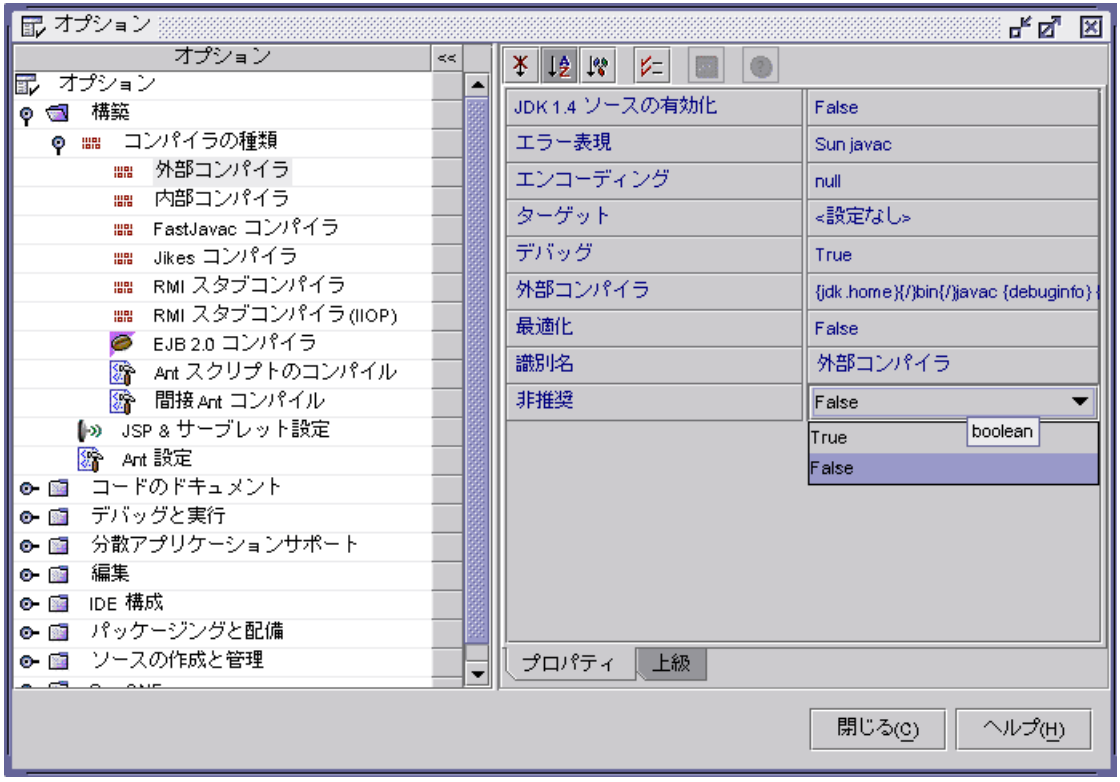
14. 「ファイルシステム」アイコンを右クリックし、エクスプローラパネルで「マウント」「ローカルディレクトリ」を選択します。
`${migtbox_home}¥work¥output¥MigtoolboxSample¥war` を選択し、「OK」をクリックします。Forte ではこのディレクトリを標準の WAR ディレクトリとして認識し、WAR ビューを画像の処理に表示します。



FFJ では、FILESYSTEM という用語を、プロジェクトの CLASSPATH のエントリを指す用語として使用します。WAR ディレクトリのマウント時に自動的に CLASSPATH の一部となるのは、./war/WEB-INF/classes に置かれている「war」ファイルだけではありません。./war/WEB-INF/lib に置かれている ZIP および JAR の各ライブラリも追加されます。

Java ソースをコンパイルする必要はありません。コンパイラの「非推奨 (deprecation)」フラグは必ず指定します。このフラグを指定して変換されたアプリケーションのコンパイルを実行すると、「自動移行対象外の」API を使用しているコードの各行についてレポートをコンパイル時に生成します。この目的は、できるだけ早くコンパイルを完了し、手動移行に必要なタスクのレポートを生成することです。

15. 外部コンパイラをコンパイラとして指定してプロジェクトのプロパティを編集し、「非推奨 (deprecation)」を true に設定します。「ツール」をメニューから選択し、「オプション」をクリックします。最初に「構築」、次に「コンパイラの種類」ノードを展開し、「オプション」ウィンドウの「外部コンパイラ」の「非推奨」を true に設定します。以下にその図を示します。



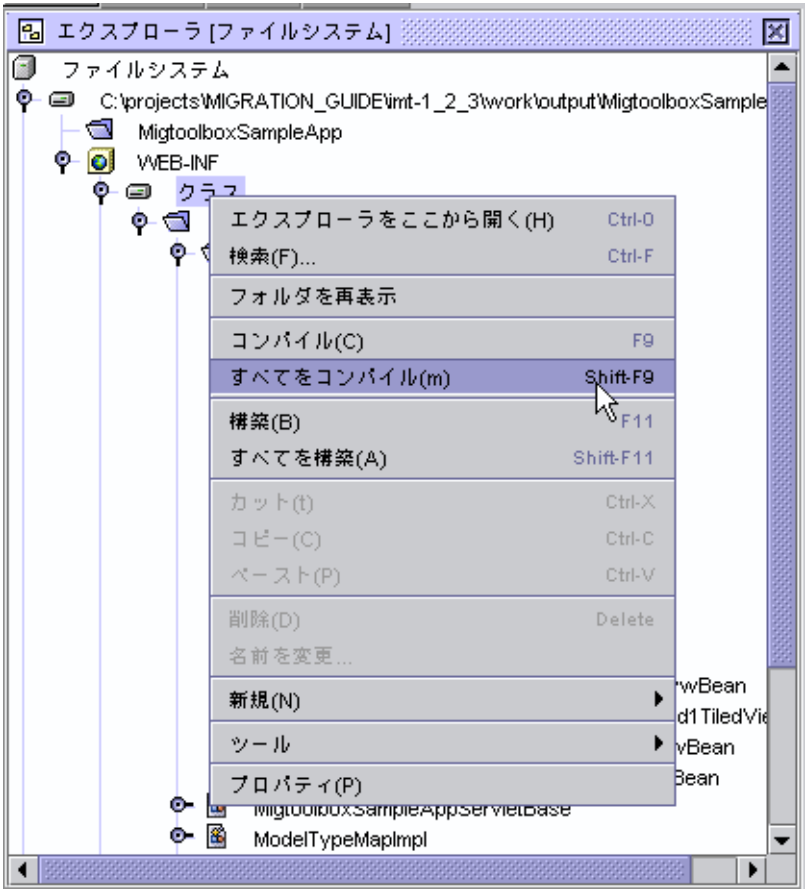
16. エクスプローラの「プロジェクト」でクラス分岐を選択し、右クリックしてメニューを開き、「すべてをコンパイル」を選択します。移行されたコードは次のディレクトリに置かれているものが処理対象になります。

    ```
    ${migtbox_home}¥work¥output¥MigtoolboxSample¥war¥WEB-INF¥classes¥
    ```

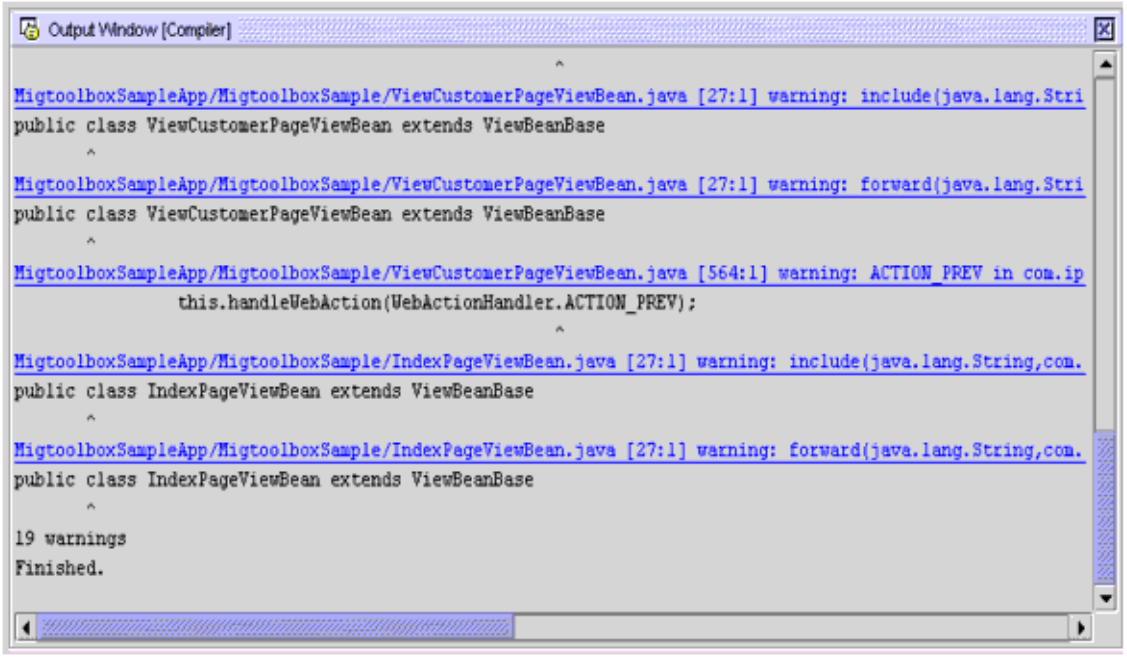
    新しく生成された JATO インフラストラクチャは次のディレクトリに置かれているものが処理対象になります。

    ```
    ${migtbox_home}¥work¥output¥MigtoolboxSample¥war¥WEB-INF¥classes¥
    ```

    コンパイルはただちに実行されます。

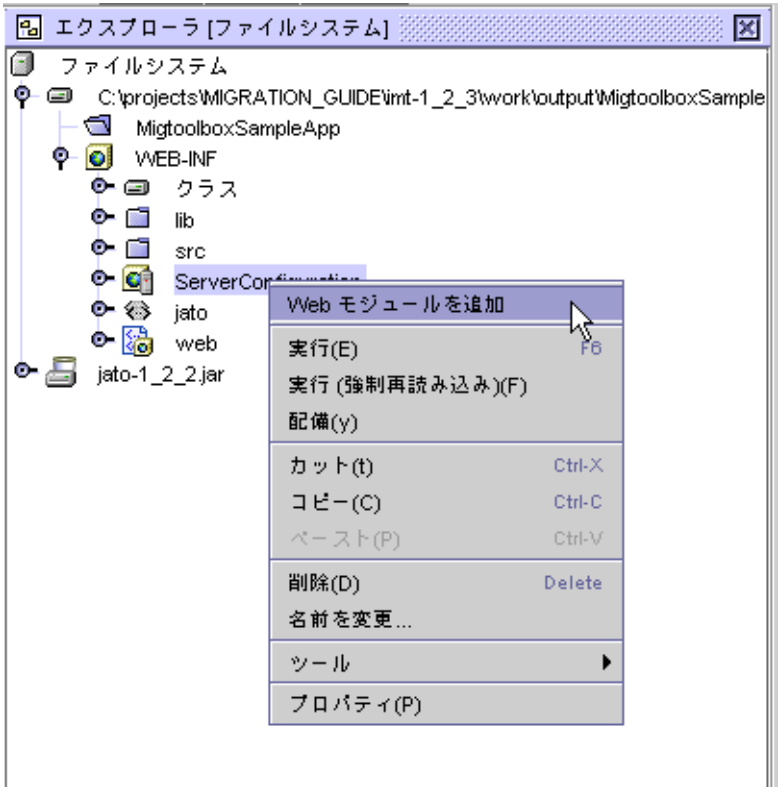


コンパイラが出す警告の例を以下に示します。



17. アプリケーションの手動移行を完了するためには、これらの警告内容を修正する必要があります。最終的に生成される Web アプリケーションは任意の J2EE Web コンテナに配備できます。FFJ では WAR ファイルをエクスポートし、Sun ONE Application Server 7 に配備できます。Web アプリケーションは FFJ に組み込まれた TomCat サーバーで直接実行することもできます。

18. サーバーモジュールグループを FFJ に追加します。エクスプローラの「WEB-INF」分岐を右クリックし、「新規」->「JSP& サーブレット」->「Web モジュールグループ」を選択し、サーバーグループを追加します。ウィザード画面のデフォルト値をそのまま使用し、「完了」をクリックします。エクスプローラの「WEB-INF」に、ServerConfiguration という名前の新しい要素が表示されます。エクスプローラの「ServerConfiguration」分岐を右クリックし、「Web モジュールを追加」を選択して現在の Web アプリケーションを追加します。「Web モジュールを追加」ウィンドウでサーブレットコンテキスト名を指定します。たとえば「Demo」という名前にします。



19. Explorer で「ServerConfiguration」分岐を右クリックし、「実行」を選択して FORTE で実行します。

第 5 章

移行の自動化

この章では、以前のバージョンの Sun ONE Application Server および他のアプリケーションサーバープロバイダからの移行プロセスの自動化に利用できる移行ツールの使用方法について説明します。

次の移行ツールが利用できます。

- Sun ONE Migration Tool for Application Servers
- Sun ONE Migration Toolbox (従来の iPlanet Migration Toolbox)

Sun ONE Migration Tool for Application Servers

Sun ONE Migration Tool for Application Servers は、J2EE[tm] アプリケーションを他のサーバープラットフォームから Sun ONE Application Server (バージョン 6.5 / 7) に移行します。

Sun ONE Application Server 6.5 の動作環境として以下のソースプラットフォームがサポートされています。

- WebSphere Application Server (WAS) 4.0
- WebLogic Application Server (WLS) 5.1

Sun ONE Application Server 7 の動作環境として以下のソースプラットフォームがサポートされています。

- WebLogic Application Server (WLS) 5.1、6.0、6.1
- WebSphere Application Server (WAS) 4.0
- J2EE Reference Implementation Application Server (RI) 1.3
- Sun ONE Application Server 6.x
- Sun ONE Web Server 6.0

移行ツールの仕様と移行プロセスは変更される場合があるため、ツールを使用する移行の例はこのガイドには記載しません。サンプルアプリケーションの移行プロセスはこのツールのドキュメントに記載します。Sun ONE Migration Tool for Application Servers の最新バージョンは Sun Download center からダウンロードできます。Sun ONE Migration Tool の最新情報は http://www.sun.com/migration/sunonetools.html にあります。

Sun ONE Migration Toolbox (従来の iPlanet Migration Toolbox)

Sun ONE Migration Toolbox の情報は「付録 B」にあります。

移行したアプリケーションの再配備

利用可能な移行ツールで自動的に移行されたアプリケーションでは、ほとんどの場合、『Sun ONE Application Server 管理者ガイド』で説明する標準配備タスクを使用します。

場合によっては、自動移行では、ソースアプリケーションから特定のメソッドやシンタックスを移行できないことがあります。Sun ONE Migration Tool for Application Server でこの問題に直面した場合、移行を完了するのに必要な手順があります。この場合には、移行後の手動による手順をいったん終了した後、『Sun ONE Application Server 管理者ガイド』で説明する標準方式でアプリケーションを配備できます。

付録 A

iBank アプリケーションの仕様

サンプルアプリケーションとして使用するのは「iBank」です。「iBank」は次の機能を持ち、基本的なオンラインバンキングサービスをシミュレートします。

- オンラインバンキングサービスへのログオン
- 詳細な個人情報および支店情報の表示と編集
- 口座およびその決済後残高の一覧表示
- 口座ごとの取引明細の表示
- 口座間で資金のオンライン振替を可能にする振替サービス
- 指定された元金と年利に基づいた複数年の複利予測

アプリケーションは MVC (Model-View-Controller) モデルに基づいて設計されています。

- EJB はアプリケーションのビジネスおよびデータモデルコンポーネント定義に使用
- Java Server Pages はプレゼンテーションロジックを処理し、ビューを表す
- サーブレットはコントローラとして動作し、アプリケーションロジックの処理、ビジネスロジックコンポーネントの呼び出し、EJB 経由のビジネスデータへのアクセス (モデル)、Java Server Pages 表示のための処理データのディスパッチ (ビュー) を行う

アプリケーションコンポーネントのパッケージングと配備には、標準 J2EE メソッド、配備記述子のインクルード定義、およびアーカイブファイル内のアプリケーションコンポーネントのパッケージングが使用されます。

- HTML ページ、画像、サーブレット、JSP、カスタムタグライブラリ、および必要に応じてサーバーサイド Java クラスを含み、Web アプリケーションで使用する WAR アーカイブファイル
- 配備記述子、Bean クラスおよびインタフェース、スタブおよびスケルトンクラス、および必要に応じてその他のヘルパークラスを含み、EJB パッケージングで使用する EJB-JAR アーカイブファイル

- Web アプリケーションモジュールおよびアプリケーションで使用される EJB モジュールを含み、エンタープライズアプリケーションモジュールのパッケージングで使用する EAR アーカイブファイル

標準の J2EE パッケージングメソッドを使用すると、Sun ONE Application Server 6.0/6.5 と Sun ONE Application Server 7 の差異、そしてそれが原因で発生する問題が指摘されるという利点があります。

アプリケーション開発用ツール

Sun ONE Studio Enterprise Edition for Java、リリース 4.0

Sun ONE Application Server 7 は EJB 1.0 および EJB 1.1 標準をサポートしています。したがって iBank アプリケーションの他の EJB (2 セッション EJB および BMP エンティティ Bean) は Sun ONE Studio for Java で開発され、提供されている Application Assembly Tool を使用して Sun ONE Application Server 7 にパッケージされ、配備されます。このような方法で開発するため、1.1 EJB 開発用のサードパーティ IDE を Sun ONE Application Server 7 でテストできます。また、Sun ONE Application Server 6.5 用に開発された 1.1 EJB も Sun ONE Application Server 7 に移行し、テストできます。

Java 開発環境用の Sun ONE Studio は、iBank アプリケーションの EJB コンポーネントを Sun ONE Application Server に移行する場合にも使用されます。EJB 1.0 標準コードの EJB 1.1 標準向けの調整、CMP エンティティ Beans 用の O/R マッピング、アプリケーションの複数のモジュールの配備プロパティおよびパッケージングの設定などを行います。

Oracle 8i 8.1.6

データベースは Oracle 8i バージョン 8.1.6 で開発されており、アクセスに使用される JDBC ドライバは thin Oracle ドライバ (type 4) です。

データベーススキーマ

- iBank データベーススキーマは次のビジネスルールに基づいて決定されます。
- iBank は国内の大都市に支店を持つ
- 支店は各地域のすべての顧客を管理する
- 顧客は自分の居住地域の支店に 1 つ以上の口座を持つ
- 顧客の口座は支店番号と口座番号で一意に識別され、これとは別に顧客はそれぞれ顧客番号を持つ。引落可能な現在の決済残高が口座に記録される
- 口座の種類は当座預金や普通預金などの口座タイプで区別される
- 各口座タイプには、支店や顧客に関わらず、このタイプのすべての口座に適用される金利や当座借越限度額などの細則が保存される

- 顧客の口座に入金があった場合、または口座からの引き落としがあった場合、その取引は取引履歴と呼ばれるすべての取引のログに記録される
- 取引履歴には支店コード、口座番号、記帳 (記録) 日付、取引タイプ識別コード、特定の取引に対する補足説明、および取引金額など、各取引についての詳細な情報が保存される
- 現金預金、クレジットカード支払、口座間の送金など、様々な取引が取引タイプで識別される

これらのビジネスルールを図解したものが次のエンティティ関連図です。

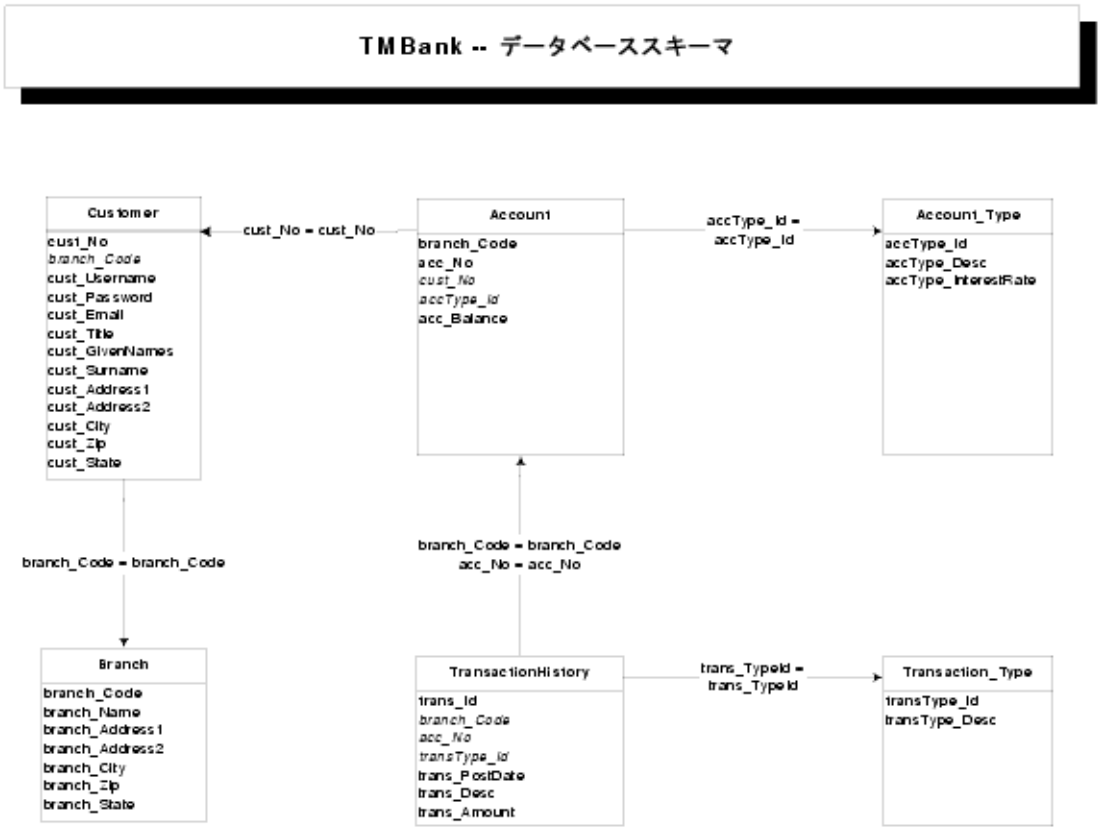


図 1　　データベーススキーマ

次に示すのはデータベースモデルから作成された表定義です。主キー列は太字、外部キー列は斜体で示されます。

BRANCH			
BRANCH_CODE	CHAR(4)	NOT NULL	支店を識別する 4 桁のコード
BRANCH_NAME	VARCHAR(40)	NOT NULL	支店名
BRANCH_ADDRESS1	VARCHAR(60)	NOT NULL	支店の住所、1 行目
BRANCH_ADDRESS2	VARCHAR(60)		支店の住所、2 行目
BRANCH_CITY	VARCHAR(30)	NOT NULL	支店の所在都市
BRANCH_ZIP	VARCHAR(10)	NOT NULL	支店の郵便番号
BRANCH_STATE	CHAR(2)	NOT NULL	支店の住所、都市名略称

CUSTOMER			
CUST_NO	INT	NOT NULL	iBank 顧客番号 (グローバル)
BRANCH_CODE	CHAR(4)	NOT NULL	この顧客が口座を持つ支店
CUST_USERNAME	VARCHAR(16)	NOT NULL	顧客のログインユーザー名
CUST_PASSWORD	VARCHAR(10)	NOT NULL	顧客のログインパスワード
CUST_EMAIL	VARCHAR(40)		顧客の電子メールアドレス
CUST_TITLE	VARCHAR(3)	NOT NULL	顧客の称号
CUST_GIVENNAMES	VARCHAR(40)	NOT NULL	顧客の名
CUST_SURNAME	VARCHAR(40)	NOT NULL	顧客の姓
CUST_ADDRESS1	VARCHAR(60)	NOT NULL	顧客の住所、1 行目
CUST_ADDRESS2	VARCHAR(60)		顧客の住所、2 行目
CUST_CITY	VARCHAR(30)	NOT NULL	顧客の居住都市
CUST_ZIP	VARCHAR(10)	NOT NULL	顧客の郵便番号
CUST_STATE	CHAR(2)	NOT NULL	顧客の住所、都市名略称

ACCOUNT_TYPE			
ACCTYPE_ID	CHAR(3)	NOT NULL	3 桁の口座タイプコード
ACCTYPE_DESC	VARCHAR(30)	NOT NULL	口座タイプの説明
ACCTYPE_INTERESTRATE	DECIMAL(4,2)	DEFAULT 0.0	年利

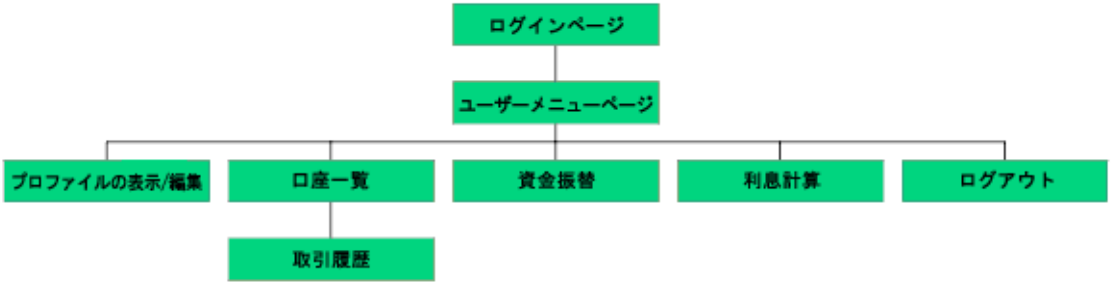
ACCOUNT			
BRANCH_CODE	CHAR(4)	NOT NULL	支店コード (主キー構成要素 1)
ACC_NO	CHAR(8)	NOT NULL	口座番号 (主キー構成要素 2)
CUST_NO	INT	NOT NULL	口座名義人
ACCTYPE_ID	CHAR(3)	NOT NULL	口座タイプ。ACCOUNT_TYPE を参照
ACC_BALANCE	DECIMAL(10.2)	DEFAULT 0.0	決済後の利用可能残高

TRANSACTION_TYPE			
TRANSTYPE_ID	CHAR(4)	NOT NULL	4 桁の取引タイプコード
TRANSTYPE_DESC	VARCHAR(40)	NOT NULL	取引コードの説明

TRANSACTION_HISTORY			
TRANS_ID	LONGINT	NOT NULL	全取引のシリアル番号
BRANCH_CODE	CHAR(4)	NOT NULL	ACCOUNT 参照キー構成要素 1
ACC_NO	CHAR(8)	NOT NULL	ACCOUNT 参照キー構成要素 2
TRANSTYPE_ID	CHAR(4)	NOT NULL	TRANSACTION_TYPE を参照する項目
TRANS_POSTDATE	TIMESTAMP	NOT NULL	取引の記帳日付および時刻
TRANS_DESC	VARCHAR(40)		取引に関する追記事項
TRANS_AMOUNT	DECIMAL(10.2)	NOT NULL	取引額

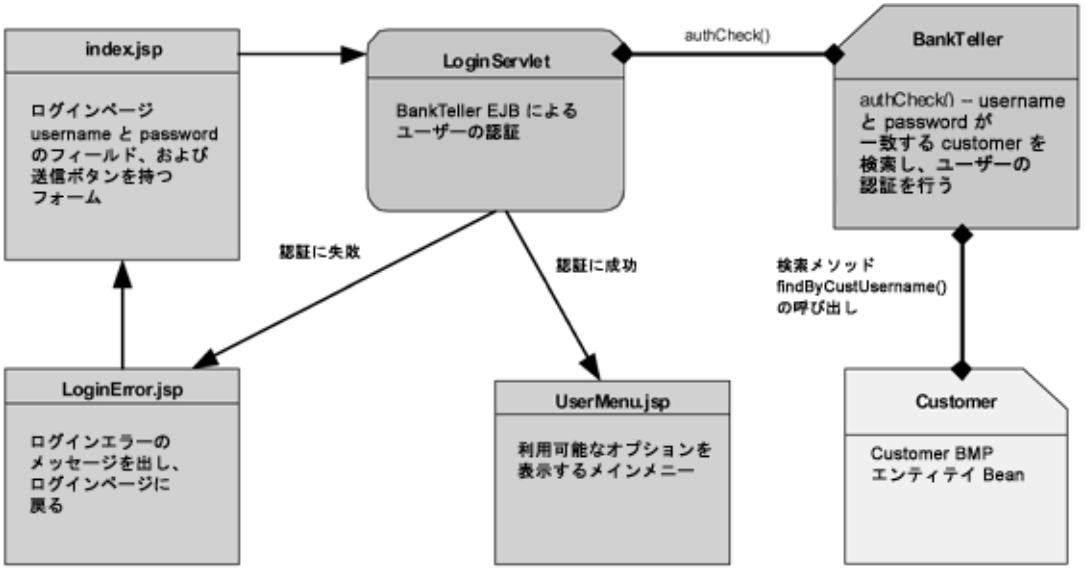
アプリケーション間の移動とロジック

アプリケーション間の移動についての上位レベルビュー

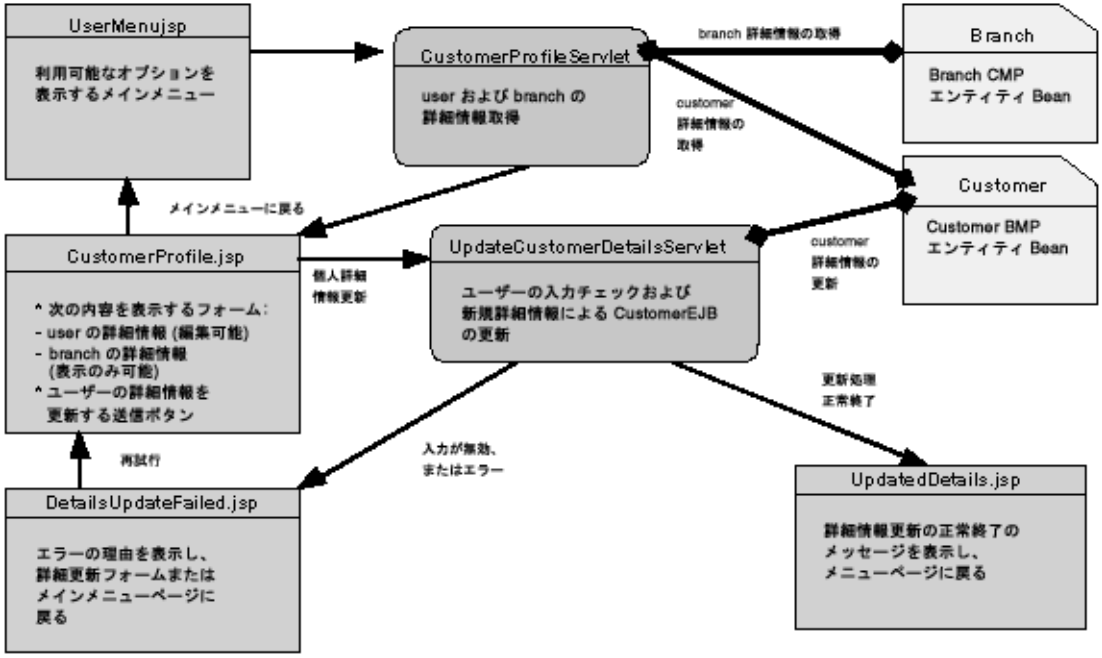


詳細なアプリケーションロジック

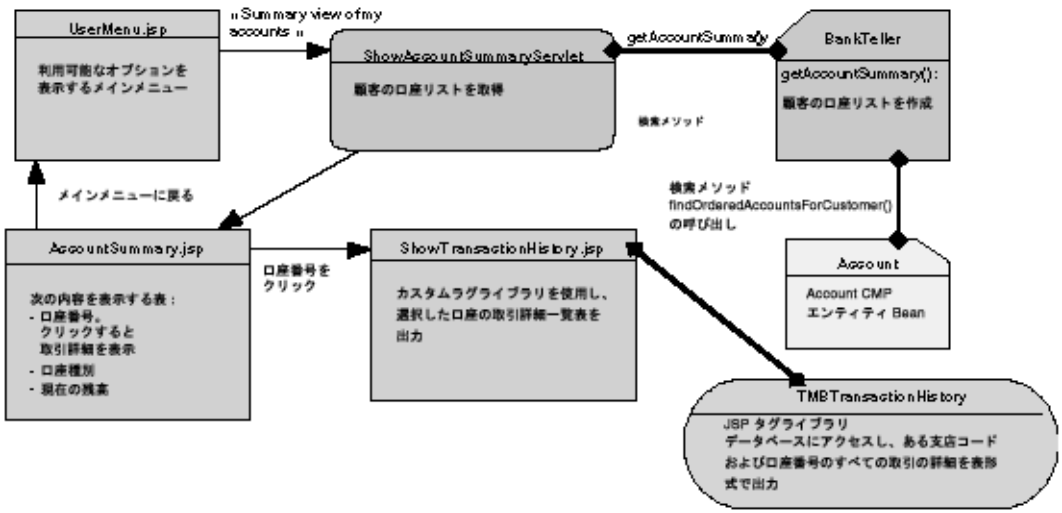
- ログインの手順



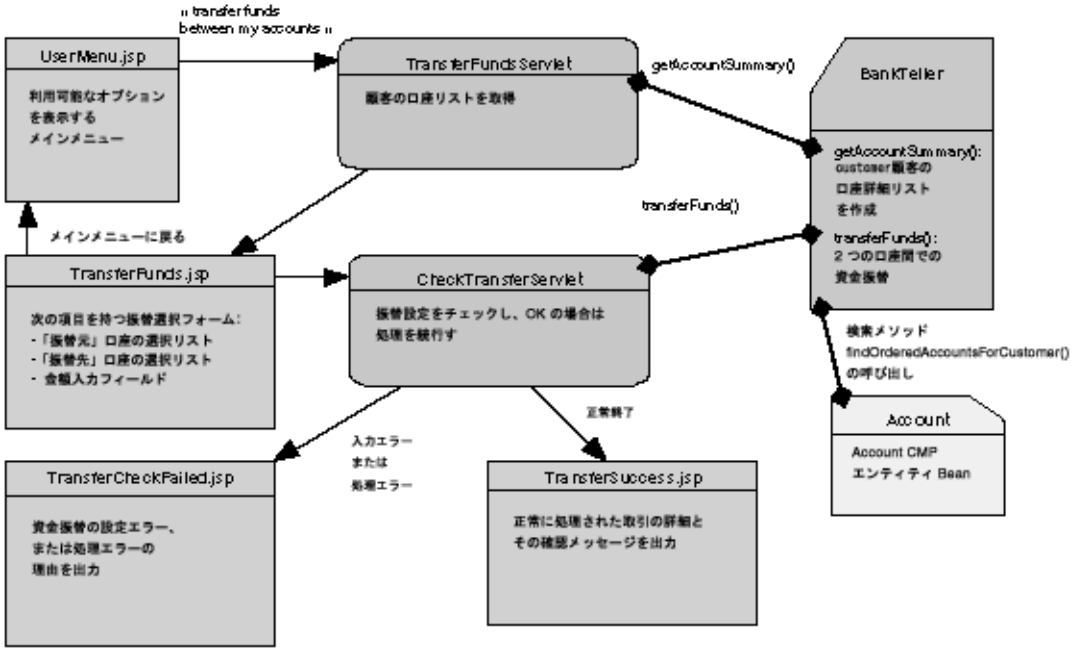
- 詳細情報の表示 / 編集


UserMenu.jsp
利用可能なオプションを
表示するメインメニュー
CustomerProfileServlet
user および branch の
詳細情報取得
branch 詳細情報の取得
Branch
Branch CMP
エンティティ Bean
customer
詳細情報の
取得
Customer
Customer BMP
エンティティ Bean
メインメニューに戻る
CustomerProfile.jsp
^ 次の内容を表示するフォーム:
- user の詳細情報 (編集可能)
- branch の詳細情報
(表示のみ可能)
^ ユーザーの詳細情報を
更新する送信ボタン
個人詳細
情報更新
UpdateCustomerDetailsServlet
ユーザーの入力チェックおよび
新規詳細情報による CustomerEJB
の更新
customer
詳細情報の
更新
更新処理
正常終了
再試行
入力が無効、
またはエラー
DetailsUpdateFailed.jsp
エラーの理由を表示し、
詳細更新フォームまたは
メインメニューページに
戻る
UpdatedDetails.jsp
詳細情報更新の正常終了の
メッセージを表示し、
メニューページに戻る

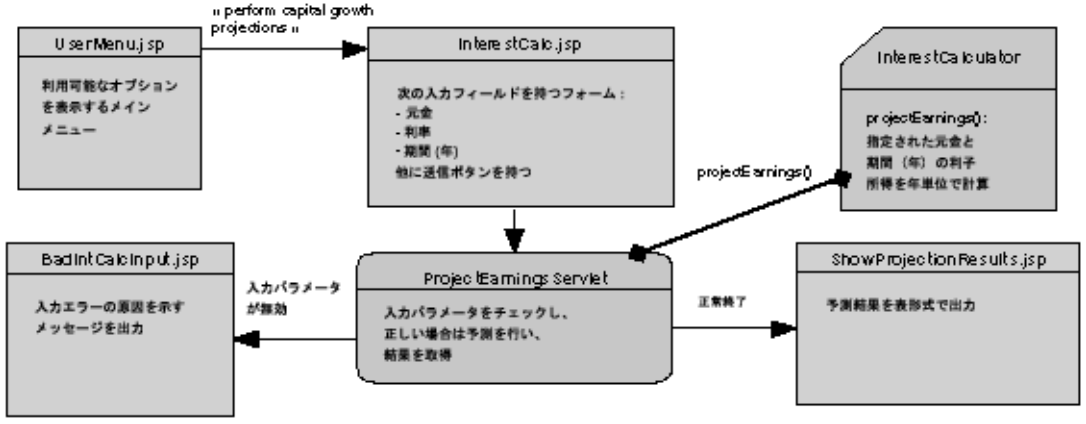
- 口座一覧と取引履歴


UserMenu.jsp
利用可能なオプションを
表示するメインメニュー
« Summary view of my
accounts »
ShowAccountSummaryServlet
顧客の口座リストを取得
getAccountSummary()
BankTeller
getAccountSummary():
顧客の口座リストを作成
検索メソッド
メインメニューに戻る
検索メソッド
findOrderedAccountsForCustomer()
の呼び出し
AccountSummary.jsp
次の内容を表示する表:
- 口座番号。
クリックすると
取引詳細を表示
- 口座種別
- 現在の残高
口座番号を
クリック
ShowTransactionHistory.jsp
カスタムタグライブラリを使用し、
選択した口座の取引詳細一覧表を
出力
Account
Account CMP
エンティティ Bean
TMBTransactionHistory
JSP タグライブラリ
データベースにアクセスし、ある支店コード
および口座番号のすべての取引の詳細を表形
式で出力

- 資金振替


« transfer funds between my accounts »
UserMenu.jsp
利用可能なオプションを表示するメインメニュー
TransferFundsServlet
顧客の口座リストを取得
getAccountSummary()
BankTeller
getAccountSummary(): customer顧客の口座詳細リストを作成
transferFunds(): 2つの口座間での資金振替
transferFunds()
メインメニューに戻る
TransferFunds.jsp
次の項目を持つ振替選択フォーム:
- 「振替元」口座の選択リスト
- 「振替先」口座の選択リスト
- 金額入力フィールド
CheckTransferServlet
振替設定をチェックし、OKの場合は処理を続行す
検索メソッド findOrderedAccountsForCustomer() の呼び出し
Account
Account CMP エンティティ Bean
正常終了
入力エラーまたは処理エラー
TransferCheckFailed.jsp
資金振替の設定エラー、または処理エラーの理由を出力
TransferSuccess.jsp
正常に処理された取引の詳細とその確認メッセージを出力

- 利息計算


« perform capital growth projections »
UserMenu.jsp
利用可能なオプションを表示するメインメニュー
InterestCalc.jsp
次の入力フィールドを持つフォーム:
- 元金
- 利率
- 期間(年)
他に送信ボタンを持つ
InterestCalculator
projectEarnings(): 指定された元金と期間(年)の利子所得を年単位で計算
projectEarnings()
BadIntCalcInput.jsp
入力エラーの原因を示すメッセージを出力
入力パラメータが無効
ProjectEarningsServlet
入力パラメータをチェックし、正しい場合は予測を行い、結果を取得
正常終了
ShowProjectionResults.jsp
予測結果を表形式で出力

アプリケーションコンポーネント

- データコンポーネント

データベーススキーマの各表はエンティティ Bean としてカプセル化されます。

エンティティ Bean	データベーステーブル
Account	ACCOUNT 表
AccountType	ACCOUNT_TYPE 表
Branch	BRANCH 表
Customer	CUSTOMER 表
Transaction	TRANSACTION_HISTORY 表
TransactionType	TRANSACTION_TYPE 表

ほとんどすべてのエンティティ Beans がコンテナ管理による持続性 (CMP) を使用します。ただし *Customer* は例外で、Bean 管理による持続性 (BMP) を使用します。

現在アプリケーション側では *Account*、*AccountType*、*Branch*、および *Customer beans* だけを使用しています。

- ビジネスコンポーネント

  アプリケーションのビジネスコンポーネントはセッション Beans でカプセル化されます。

  BankTeller Bean はステートフルセッション Bean であり、顧客とシステムとの対話をすべてカプセル化します。*BankTeller* は特に次の目的で使用されます。

  - `authCheck()` メソッドによる顧客の認証
  - `getAccountSummary()` メソッドによる顧客の口座リスト取得
  - `transferFunds()` メソッドによる 1 人の顧客の口座間の資金振替

InterestCalculator Bean はステートレスセッション Bean であり、資金計算をカプセル化します。`projectEarnings()` メソッドによる複利計算で使用されます。

- アプリケーションロジックコンポーネント (サーブレット)

コンポーネント名	目的
LoginServlet	BankTeller セッション Bean (authCheck() メソッド) によるユーザーの認証、HTTP セッションの生成、およびセッションのユーザー情報保存を行う。正常に認証された場合は要求をメインメニューページ (UserMenu.jsp) に転送する
CustomerProfileServlet	顧客および支店に関する詳細情報をそれぞれのエンティティ Beans から取得し、要求を「view/edit details」ページ (CustomerProfile.jsp) に転送する
UpdateCustomerDetails Servlet	CustomerProfile.jsp での顧客の詳細情報の変更が妥当かどうかをチェックし、妥当な場合は顧客のエンティティ Bean を更新して反映する。処理が正常に実行された場合は UpdatedDetails.jsp にリダイレクトし、入力が正しくない場合は DetailsUpdateFailed.jsp にリダイレクトする
ShowAccountSummary Servlet	顧客の口座リストを BankTeller セッション Bean から getAccountSummary() メソッドで取得し、要求を AccountSummary.jsp に転送して表示する
TransferFundsServlet	顧客の口座リストを BankTeller セッション Bean から getAccountSummary() メソッドで取得後、要求を TransferFunds.jsp に転送し、ユーザーによる転送設定を許可する
CheckTransferServlet	ユーザーが振替元および振替先として指定した口座をチェックし、入力した金額の振替ができるかどうかを確認する。BankTeller セッション Bean の transferFunds() メソッドを呼び出し、振替を行う。入力または処理エラーの場合は CheckTransferFailed.jsp にリダイレクトし、処理が正常に実行された場合は TransferSuccess.jsp にリダイレクトする
ProjectEarningsServlet	InterestCalc.jsp でユーザーが定義した利息計算パラメータを取得し、InterestCalculator ステートレスセッション Bean の projectEarnings() メソッドを呼び出して計算を行う。結果を ShowProjectionResults.jsp ページに転送して表示する。入力が無効な場合は BadIntCalcInput.jsp にリダイレクトする

- プレゼンテーションロジックコンポーネント (JSP ページ)

コンポーネント名	目的
index.jsp	ログインページとしても使用されるアプリケーションの索引ページ
LoginError.jsp	入力されたユーザー証明書が無効な場合に表示されるログインエラーのページ。ログインできなかった理由を示すメッセージを表示する
Header.jsp	アプリケーションのすべての HTML ページに動的に追加されるページヘッダ
CheckSession.jsp	このページはアプリケーションのすべてのページに静的に追加され、ユーザーがログインしているかどうか、つまり有効な HTTP セッションを保有しているかどうかを確認する。有効なセッションを保有していない場合は、NotLoggedIn.jsp ページにリダイレクトされる
NotLoggedIn.jsp	最初にログインの手順を実行せずに、アプリケーションのページにアクセスしようとした場合にリダイレクトされるページ
UserMenu.jsp	正常にログインするとリダイレクトされる、メインのアプリケーションメニューページ。実行できるすべての操作に対するリンクを持つ
CustomerProfile.jsp	編集可能な顧客の詳細情報や支店の詳細情報が表示されるページ。このページで住所変更が可能
UpdatedDetails.jsp	詳細情報の正常更新後にリダイレクトされるページ
DetailsUpdateFailed.jsp	入力エラーのため詳細情報を更新できない場合にリダイレクトされるページ
AccountSummaryPage.jsp	ある顧客の持つ複数の口座を一覧表として表示するページ。口座番号、口座のタイプ、現在の残高を表示する。表の口座番号をクリックすると、その口座の詳細な取引履歴が表示される
ShowTransactionHistory.jsp	ある口座の詳細な取引履歴を表示するページ。カスタムタグライブラリを使用して表示を行う
TransferFunds.jsp	ある口座から別の口座への指定された金額の振替を設定するページ

TransferCheckFailed.jsp	選択された資金振替の設定が正しくない場合にリダイレクトされるページ
TransferSuccess.jsp	ユーザーが設定した送金が正常に実行された場合に表示されるページ。確認メッセージが表示される
InterestCalc.jsp	複利計算パラメータ入力用のページ
BadIntCalcInput.jsp	複利計算パラメータが正しくない場合にリダイレクトされるページ
ShowProjectionResults.jsp	利息計算が正常に実行された場合にリダイレクトされるページ。予測結果を表形式で表示する
Logout.jsp	アプリケーションの終了ページ。ユーザーに関連付けられたステートフルセッション Bean を削除し、HTTP セッションを無効化する
Error.jsp	予期しないエラーが発生するとリダイレクトされるページ。発生したエラーの詳細情報を表示する

移行時に発生する問題を考慮した最適な設計の選択

アプリケーションの設計については、特に「実際の」コンテキストで議論すべきものが多いのは確かですが、一般的な J2EE アプリケーションの移行で発生する可能性がある問題を、サンプルアプリケーションでできるだけ網羅することも考慮する必要があります。

このセクションでは J2EE アプリケーションの移行時に発生する可能性がある問題を検討し、移行時にこれらの問題をチェックするために iBank に用意されているコンポーネントについて説明します。

選択された移行を処理するための次のテクノロジーに焦点を当てて説明します。

サーブレット

- iBank にはサーブレットがいくつか用意されており、以下についての潜在的な問題を特定できます。
- サーブレット API の一般的な機能の使用
- HTTP セッションおよび HTTP 要求の属性の保存または取得
- サーブレットコンテキスト初期化パラメータの取得
- ページのリダイレクト

Java Server Pages

JSP の仕様に関しては次の面を重視しています。

- JSP 宣言、スクリプトレット、式、およびコメントの使用
- 静的なインクルード (`<%@ include file="…" %>`)。特に `CheckSession.jsp` ファイルの各ページへの追加でテスト
- 動的なインクルード (`<jsp:include page=… />`)。`Header.jsp` の各ページへの動的な追加でテスト
- カスタムタグライブラリの使用。カスタムタグライブラリを `ShowTransactionHistory.jsp` で使用
- JSP 例外処理のエラーページ。アプリケーションエラーリダイレクト用の `Error.jsp` ページ

JDBC

iBank アプリケーションは接続プールおよびデータソース経由でデータベースにアクセスします。BMP エンティティ Bean、BankTeller セッション Bean、カスタムタグライブラリでアクセスロジックをプログラミングすることもでき、CMP エンティティ Beans でアクセス方法を宣言することもできます。

Enterprise Java Beans (EJB)

iBank では様々な Enterprise Java Beans を使用しています。

エンティティ Beans :

Bean 管理による持続性 (*Customer* Bean) では、次のテストが可能です。

- 初期コンテキストの JNDI 検索
- JDBC 経由のプールデータソースアクセス
- BMP カスタム検索の定義 ("`findByCustUsername()`")

コンテナ管理による持続性 ("*Account*" および "*Branch*" Beans) では、次のテストが可能です。

- 開発ツールを使用する、配備記述子内のオブジェクトとリレーショナル間の (O/R) マッピング
- 複合主キーの使用 ("*Account*")

- *Account* Bean とその `findOrderedAccountsForCustomer()` メソッドを使用するカスタム CMP 検索の定義。これによって配備記述子のクエリロジック宣言の差異を確認し、オブジェクト集計を返す場合の複雑な例を見ることができる

セッション Beans :

ステートレスセッション Beans。*InterestCalculator* では次のテストが可能です。

- ステートレスセッション Bean の使用と配備
- 計算用ビジネスメソッドの呼び出し

ステートフルセッション Beans。*BankTeller* では次のテストが可能です。

- JNDI と初期コンテキストを使用する様々なインタフェースの検索
- JDBC によるデータベースクエリ
- Bean methods の様々なトランザクション属性の使用
- コンテナ境界設定トランザクションの使用
- 呼び出し間の会話型ステートの維持
- "`getAccountSummary()`" メソッドなど、エンティティ Beans のフロントエンドとして動作するビジネスメソッド

アプリケーションのパッケージ化

iBank は、次の J2EE 標準プロシージャをそれぞれ指定のファイルを使用してパッケージ化します。

- Web アプリケーションモジュールには Web アプリケーションアーカイブファイルを使用し、EJB には EJB-JAR アーカイブを使用
- エンタープライズアプリケーションアーカイブファイル (EAR ファイル) は、Web アプリケーションおよび EJB モジュールの最終パッケージ化で使用

付録 B

Sun ONE Migration Toolbox

Sun ONE Migration Toolbox (S1MT) は、主に NetDynamics または Kiva/NAS プラットフォームで構築されたアプリケーションを Sun ONE Application Server または J2EE 互換コンテナに移行するために使用されます。Sun ONE Migration Toolbox のメインインタフェースは、Toolbox アプリケーション または Toolbox GUI と呼ばれます。このアプリケーションは `%MIGTBX_HOME%/bin/toolbox.bat` スクリプトを実行して呼び出します。ただし `setenv.bat` ファイルが適切にカスタマイズされていなければなりません。詳細については `README.txt` を参照してください。

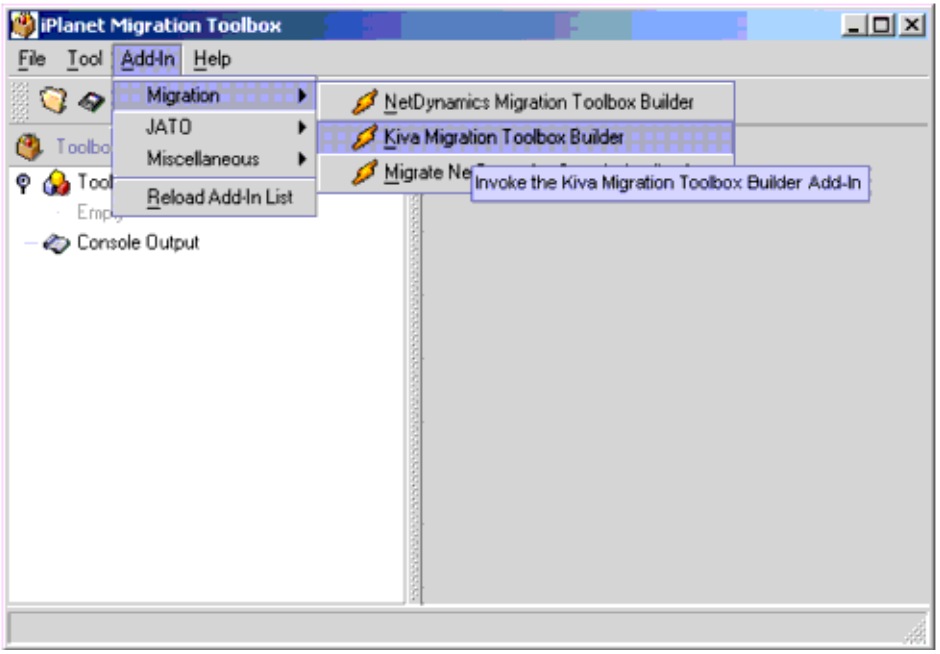
サポートされるプラットフォーム

Microsoft Windows NT 4.0 および Windows 2000 は現時点で S1MT をサポートしています。Windows 95/98/Me など、他の Win32 プラットフォームでも動作すると思われますが、これらについてはテストしていないため、S1MT インストールのマニュアルに記載された以外の追加設定が必要となることがあります。

Toolbox を正常に実行するためには少なくとも JavaSoft JDK 1.2.2 が必要になります。JDK 1.3.1 ではテストされています。

移行

Toolbox は移行時の様々な処理を行う一連のツールです。S1MT 1.2.3 は NetDynamics および Kiva/NAS プラットフォームからの移行をサポートしています。各プラットフォーム用に提供されている Toolbox Builder を実行すると、一連のアプリケーション移行ツールが生成されます。Kiva Migration Toolbox Builder は Kiva/NAS アプリケーションの移行ツールを生成し、NetDynamics Migration Toolbox Builder は NetDynamics アプリケーションの移行ツールを生成します。Toolbox builder の呼び出し方法を次の図に示します。



Toolbox Builder

各移行で基本的には同じツールセットを使用しますが、特定の移行のためにカスタマイズが必要になる場合があります。このようなツールの作成は無駄な作業であり、命名規則およびディレクトリ構成のレイアウトで不整合が発生しやすくなります。これを防ぐためにツールボックスアドインが用意されています。これは設定済ですぐに実行できるツールであり、ツールの作成とプロパティの設定を適切に簡素化します。ツールのプロパティは似通っており、全く同じ場合も少なくありません。整合性を保つことは移行を成功させる重要な条件であるため、このツールは有効です。

Kiva Migration Toolbox Builder

次の手順によって、Kiva Migration Toolbox Builder アドインを使用する新規ツールボックスを生成します。

1. Toolbox アプリケーションが稼働していない場合は、Toolbox を起動してメニューオプション「`Add-In|Migration|Kiva Migration Toolbox Builder`」を選択します。
2. 続けて、必要な情報の入力を求めるダイアログボックスがいくつか表示されます。Toolbox Builder はこの情報を使用し、生成するツールのプロパティ値を設定します。入力を要求されないプロパティにはデフォルト値が指定されています。これらは Toolbox Builder によるツールの生成後に必要に応じて変更します。

3. 最初に新しい JATO アプリケーションが配置されるパッケージの名前を入力します。この名前がどのように使用されるかは、OnlineBankSample の移行を実行し、./war/WEB-INF/ クラスの下に JATO のデータを含む新しいパッケージがどのように配置されるかを確認することにより理解できます。既存のすべての Java コードが同じパッケージに残っていますが、JATO アプリケーションインフラストラクチャ用の新しい Java コードをいくつか生成する必要があります。新しいパッケージはこのコード向けに生成されています。元のアプリケーションの Java ソースもすべて同じパッケージに残しておくことができます。JATO リソースで新しいパッケージを定義する必要があるのは新しい Java ソースだけです。どのパッケージを選択した場合でも (たとえば com.iplanet.migration.samples.onlinebank など)、**パッケージ名の最後の部分**が移行結果のデフォルトディレクトリ名として使用されます。このディレクトリの場所は次のパネルで変更できますが、デフォルト値をそのまま使用することをお勧めします。

4. 次に Automatic Application Extract Archive Wizard を使用するかどうかを選択します。このウィザードを使用すると、アプリケーション抽出アーカイブ作成用ツールの生成が容易になります。「Cancel」を選択すると、application extract archive (ZIP/JAR) のパス名の入力だけが必要になります。これはアプリケーションのすべてのソースを含む zip または JAR ファイルの名前です。この場合はアーカイブをあらかじめ手動で作成しておく必要があります。ウィザードはエンコード仕様に進みます。

5. Automatic Application Extract Archive Wizard に対して「OK」を選択すると、アプリケーションソースのルートディレクトリの入力が要求されます。通常は ./nas/APPS ディレクトリです。

6. 次にこの移行に関連するアプリケーションソースディレクトリにトップレベルパッケージを指定します。ディレクトリにあるすべてのソースを指定する場合は値の指定を省略できます。

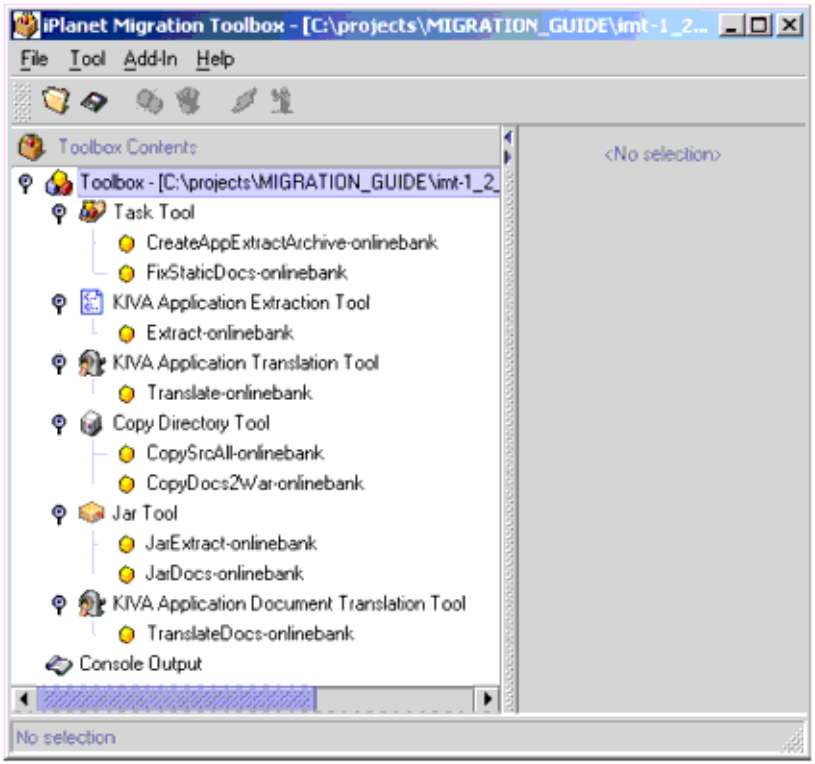
7. 次にアプリケーション抽出アーカイブの対象とするファイルの拡張子リストを指定します。

8. Automatic Application Extract Archive Wizard に対して「OK」を選択すると、Task tool、Copy Directory tool、および Java tool がツールボックスに追加されます。

9. Java ソースおよびクエリファイルの文字エンコードを入力するパネルは 2 つあります。多くの顧客が Java ソースに ASCII 以外の文字エンコードを指定しています。たとえばアジアの開発者や顧客は 2 バイト文字を含むソースファイルを使用しています。S1MT BETA から変更する場合は、各ファイルタイプに 1 つのエンコードだけを指定できます。1 つのアプリケーションでは共通のエンコード標準が使用されていることを想定しています。エンコードが複数使用されている場合は、アプリケーションを記述している XML ファイルを抽出後に変更する必要があることがあります。S1MT 1.2.3 では、ソースファイルの <meta> タグを調べ、HTML テンプレートの文字エンコーディングを自動的に検出しようとします。ただし、アプリケーションを記述する XML ファイルのエンコード状態を確認し、変換が正しく行われるかどうかを確認する必要があります。

10. この時点で Kiva Extraction and Translation Tools がツールボックスに追加されます。
11. 最後に Automatic Static Document Translation Wizard を使用するかどうかを選択します。このウィザードを使用すると、静的ドキュメントコンテンツのアセンブル、AppLogic 呼び出し用 URL の変更ドキュメントの変換、および結果 WAR ディレクトリ構造へのドキュメントコピー用のツール生成が容易になります。
12. 「Cancel」を選択するとウィザードは終了します。「OK」を選択すると、アプリケーションのドキュメントルートの入力が要求されます。入力すると別の Task tool、JAR tool、Document Translation tool、および Copy Directory tool がツールボックスに追加されます。
13. メニューオプション「`File|Save`」を選択し、名前を指定して、ツールボックスをディスクに保存します。

    Kiva Toolbox Builder で生成されるツールを次に示します。



ツールの起動

生成されたツールを使用してアプリケーションを移行する準備ができました。最初に抽出、次に変換を行います。ツールを呼び出す前にプロパティを調べ、必要に応じて変更します。Toolbox Builder に設定されている初期値をそのまま使用する場合は、変更の必要はありません。(注 : Toolbox Builder で各ツールボックスに Task Tool を 1 つ生成すれば、すべてのツールを呼び出すことができます。ただし、アプリケーションをいくつか移行し、各ツールの出力内容に慣れるまでは個別に呼び出しを行うことをお勧めします。)

Kiva Migration Toolbox Builder で生成されるツール

1. KIVA Application Extraction Tool

   Kiva アプリケーションファイルを含むアプリケーション抽出ファイル、つまり zip または JAR ファイルを読み込み、アプリケーション記述ファイル、つまり XML ドキュメントを生成します。アプリケーション記述ファイルには、アプリケーション抽出アーカイブの各ファイルの状態を含む、アプリケーション記述のための上位レベルの情報が含まれます。

   このツールは、テンプレート、アプリケーションロジック java ファイル、およびその他のアプリケーション固有のリソースなど、元の NAS/KIVA アプリケーションソースをすべて含む zip ファイルが、入力内容にあらかじめ存在することを想定しています。この zip ファイルに実際に元のクラスファイルが含まれている必要はありません。これは移行を行うことでソースファイルが変更されてしまうためです。

   移行プロセスを自動化するためには、最初にアプリケーション記述ファイルを生成します。このファイルは他の目的でも使用されますが、Kiva Application Translation Tool (com.iplanet.moko.nas.tools.KivaTranslateTool) の入力ファイルとしての使用が主な用途です。

2. KIVA Application Translation Tool

   Kiva アプリケーションファイルを含むアプリケーション抽出ファイル、つまり zip または JAR ファイル、およびアプリケーション記述ファイル、つまり XML ドキュメントを読み込みます。アプリケーション記述ファイルは、そのアプリケーションに対応する一連の J2EE コンポーネントおよびファイルの生成に使用されます。アプリケーション記述ファイル (XML ドキュメント) は、Kiva Extraction Tool (com.iplanet.moko.nas.tools.KivaExtractTool) による、一連のソース Kiva プロジェクトからの情報抽出処理で生成されます。変換ツールの使用は Kiva アプリケーション自動移行の 2 番目の手順です。

3. Copy Directory Tool

   ソースディレクトリの内容をターゲットディレクトリにコピーします。

4. JAR Tool

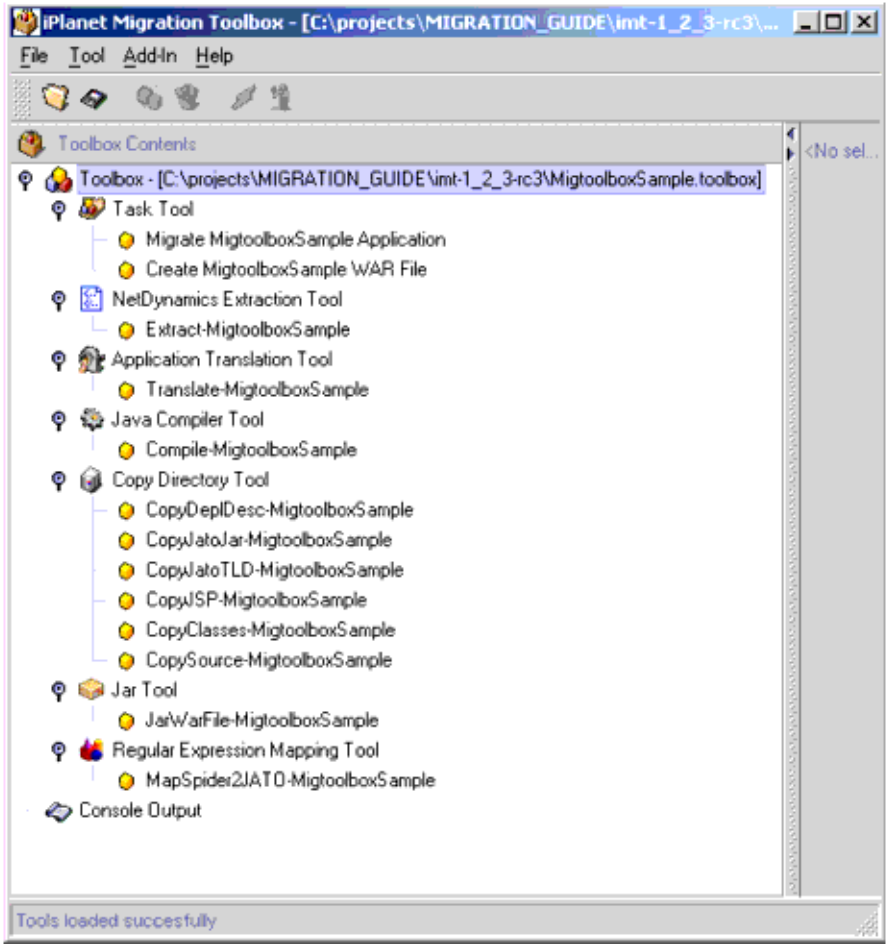
   ソースディレクトリとそのすべてのサブディレクトリを含む JAR ファイルを生成します。

NetDynamics Migration Toolbox Builder

次の手順に従って、NetDynamics Migration Toolbox Builder アドインを使用する新規ツールボックスを生成します。

1. Toolbox アプリケーションが現在稼働していない場合は、Toolbox を起動し、最初のダイアログで「Migrate an application」オプションを選択し、「OK」をクリックします。Toolbox を起動すると空の (New) ツールボックスが表示されます。メニューオプション「`Add-In|Migration|NetDynamics Migration Toolbox Builder`」を選択します。
2. 続けてダイアログボックスがいくつか表示され、必要な情報の入力が要求されます。Toolbox Builder はこの情報を使用し、生成するツールのプロパティ値を設定します。入力を要求されないプロパティにはデフォルト値が指定されています。これらは Toolbox Builder によるツールの生成後に必要に応じて変更します。
3. 最初に次のメッセージが表示されます。`Enter the logical application name` (論理アプリケーション名を入力してください。) これはアプリケーション全体を表す名前です。アプリケーションには NetDynamics プロジェクトが複数含まれる場合があります。アプリケーションが 1 つのプロジェクトで構成される場合でも、プロジェクト名とアプリケーション名は別にすることをお勧めします。たとえば、プロジェクト名が `foo` の場合は、アプリケーション名を `foo` ではなく `fooapp` にします。これによって他の同種のプロパティとの混同を防ぎ、配備が容易になります。
4. アプリケーション名を入力すると、Toolbox Builder からさらに情報の入力が要求されます。デフォルト値が設定されている場合は提示されます。すべてデフォルト値を使用することをお勧めします。
5. すべての情報を入力すると、Toolbox Builder によってツールボックスにいくつかのツールが生成されます。メニューオプション「`File|Save`」を選択し、名前を指定して、ツールボックスをディスクに保存します。アプリケーション名 (この例では最初のプロンプトで指定した `fooapp`) をツールボックス名とすることをお勧めします。

   NetDynamics Toolbox Builder で生成されるツールを次に示します。



ツールの起動

ツールが生成され、NetDynamics アプリケーションを移行する準備ができました。ツールを呼び出す前にプロパティを調べ、必要に応じて変更します。Toolbox Builder に設定されている初期値をそのまま使用する場合は、変更の必要はありません。(**注** : Toolbox Builder で各ツールボックスに Task Tool を 1 つ生成すれば、すべてのツールを呼び出すことができます。ただし、アプリケーションをいくつか移行し、各ツールの出力内容に慣れるまでは個別に呼び出しを行うことをお勧めします。)

Kiva Migration Toolbox Builder で生成されるツール

1. NetDynamics Extraction Tool

   このツールはソース NetDynamics プロジェクトからできるだけ多くの情報を集め、アプリケーション記述ファイルと呼ばれる XML ファイルに書き込みます。このアプリケーション記述ファイルを Application Translation Tool の入力ファイルとして使用します。

   このツールを呼び出す前に、次のプロパティが正しいかどうかを確認してください。

   > **`ProjectsDirectory`** は抽出時に使用される NetDynamics プロジェクトディレクトリのパスです。デフォルト値は `%MIGTBX_HOME%/work/NDProjects` です。移行するすべての NetDynamics プロジェクトをこのディレクトリに配置することをお勧めします。

   Toolbox Builder アドインでの入力時に間違えていなければ、他のプロパティの現在の値を変更する必要はありません。他のプロパティについてはこのマニュアルの後の部分で詳しく説明します。

   変更した場合はツールボックスを保存し、ツールを起動します。アプリケーション記述の XML ファイルは、**`OutputDirectory`** プロパティで指定された位置に書き出されます。

   アプリケーション記述ファイルを開いてその内容を見ればプロジェクト抽出処理の詳細を理解できます。XML Spy のような XML ブラウザの使用をお勧めします。このファイルの内容は変更しないでください。変更すると変換フェーズに大きな影響を与え、完全にエラーになるか、不正な結果が生成されます。

2. Application Translation Tool

   NetDynamics Extraction Tool で生成されたアプリケーション記述ファイルから、元の NetDynamics アプリケーションの構造および動作を正確に再現する一連の J2EE に準拠するコンポーネントを生成します。

   Toolbox Builder アドインでの入力時に間違えていなければ、他のプロパティの現在の値を変更する必要はありません。他のプロパティについてはこのマニュアルの後の部分で詳しく説明します。

   変更した場合はツールボックスを保存し、ツールを呼び出します。新しい J2EE コンポーネントは、**`OutputDirectory`** プロパティで指定された位置に書き出されます。

   さらにこのツールは移行ログファイル `MigrationLog.csv` を変換出力ディレクトリに書き出します。このファイルには、移行に特別な注意が必要であると変換時に判断された項目が記録されます。変換で自動処理されなかった項目があることを移行の開発者に警告し、必要な手動処理を記録するためにこのファイルを生成します。このファイルは移行に最低限必要な手動処理のリストとして使用されます。他に変換には直接関係しない処理が必要な場合もあります。

3. Regular Expression Mapping Tool

   このツールは Regexp Tool とも呼ばれ、ファイルの変更を反映するため XML の置換機能を使用します。Perl 5 の正規表現構文を使用します。Toolbox Builder が生成する Regexp Tool は、移行された Java ソースファイルの一般的な Spider API Java コンストラクトを同等の JATO コンストラクトに置き換えるようにあらかじめ設定されています。

   このツールを呼び出す前に、次のプロパティが正しいかどうかを確認してください。

   `SourceDirectory` は移行アプリケーションが置かれる場所です。JATO ソースファイルを処理すると予期しないエラーが発生するため、このディレクトリには置かないようにしてください。

   Toolbox Builder アドインでの入力時に間違えていなければ、他のプロパティの現在の値を変更する必要はありません。

   変更した場合はツールボックスを保存し、ツールを起動します。移行されたソースファイルが処理され、変更はコンソールに記録されます。ファイル変更前には元のファイルがバックアップされます。ファイルには拡張子 `.orig` が付けられ、元の位置に保存されます。

   重要：自動移行のフェーズはこれで終了です。次に移行アプリケーションの Java コードを移植し、J2EE/JATO API を NetDynamics Spider API の代わりに使用します。これ以降で説明するツールは、手動移行が一度完了した後でパッケージングおよび配備を行う場合に有効です。ただし 1 つ例外があります。移行されたアプリケーションはこの時点で正常にコンパイルされ、配備時にはページ呼び出しができるなど、最低限動作しなければなりませんが、NetDynamics Spider API を使用している場合は本来の機能を果たしません。したがって、正常かどうかのチェック、または Spider に依存する機能の移行チェックのみを行う場合を除いては、移行アプリケーションの最低限の部分だけを移植し、作業を続行することをお勧めします。

4. Java Compiler Tool

   このツールを使用すると、JATO Foundation Classes および新しい J2EE アプリケーションコンポーネントをマウスのクリック 1 つでコンパイルできます。実際には JDK で提供される `javac` コマンド行ツールの呼び出しだけを行います。

   前に実行したツールで出力ディレクトリのプロパティを変更していなければ、このツールで調整の必要なプロパティはありません。プロパティについてはすべてこのマニュアルの後の部分で説明します。

   変更した場合はツールボックスを保存し、ツールを呼び出します。`SourceDirectory` で指定したディレクトリにある、拡張子 .java の java クラスソースファイルがすべてコンパイルされます。

5. Copy Directory Tools (WAR ファイルディレクトリ構成の生成)

   このツールはディレクトリまたはファイルをある場所から別の場所にコピーします。またファイルのフィルタ機能を持ちます。このタイプの生成ツールの目的は「WAR ファイル作成のための」ディレクトリ構造を生成することです。最初に 4 回 Copy Directory Tools を実行し、配備記述子、タグライブラリ定義、JSP、および Java クラスを適切なディレクトリにコピーします。これで J2EE コンテナに配備する WAR ファイル生成のために Jar Tool を使用できます。

   `CopySource` が指定されている Copy Directory Tool インスタンスはオプションです。WAR ファイル生成にはソースファイルは不要ですが、配備されるアプリケーションと共に保存しておけば、正しいバージョン管理および配備先での急な修正に役立ちます。これらのソースファイルはアプリケーションクライアントから参照することは**できません**。したがって配備先のサーバーにあっても安全です。

   これ以前に実行したツールで出力ディレクトリのプロパティを変更していなければ、プロパティの値を変更する必要はありません。プロパティについてはすべてこのマニュアルの後の部分で説明します。

   変更を行った場合はツールボックスを保存し、`CopyDeplDesc`、`CopyTLD`、`CopyJSP`、および `CopyClasses` の 4 つのディレクトリコピーツールを呼び出します。環境によって必要な場合は `CopySource` を呼び出します。これらのツールを呼び出すと、選択されたファイルが各ツールの **`OutputDirectory`** プロパティで指定されたディレクトリに書き出されます。これで「WAR ファイル作成」の準備ができました。

6. Jar Tool

   このツールは JDK の JAR コマンド行ツールを使用し、この前のコピーディレクトリツールで生成されたディレクトリ構造の WAR ファイルを生成します。アプリケーションを J2EE コンテナに配備する場合はこの WAR ファイルだけが必要になります。iAS の配備の手順については『JATO 配備ガイド』で説明します。通常、コンテナはそれぞれ独自の配備処理を行うため、各コンテナの説明に従ってください。

   これ以前に実行したツールで出力ディレクトリのプロパティを変更していなければ、プロパティの値を変更する必要はありません。プロパティについてはこのマニュアルの後の部分で説明します。

   変更した場合はツールボックスを保存し、ツールを呼び出します。WAR ファイルが生成され、**`OutputDirectory`** プロパティで指定された位置に書き出されます。これでアプリケーション配備の準備ができました。

Tool と Toolbox

ツールボックスは拡張子が .toolbox のツールボックスファイルとしてディスク上に置かれます。ツールボックスの各ツールはツールボックスファイルに直列化オブジェクトとして格納されています。ツールは拡張子が .tools のファイルとしてそれぞれ独立して存在することもできます。フォーマットはほとんど同じです。ツールボックスの生成、コピー、削除、2 つのツールボックスのマージ、およびツールの個別インポートまたはグループ単位のインポートを行うメニューコマンドなどがあります。

新しいツールの生成

ツールの新しいインスタンスを生成する場合は、「Tool|New」メニューオプションを使用し、抽出、変換、またはコンパイルなど、生成するツールのタイプを選択します。現在開いているツールボックスのツールボックスツリーに新しいツールが追加されます。ツールはタイプで分類されます。名前の形式はデフォルトでは CopyDirectoryTool7 のように、<ToolType><##> になります。ツール名をトリプルクリックするか、「F2」を押して名前を変更できます。ツール名を空白にしておくこともできます。

Cloning Tools

ツールのコピーを生成する場合は、「Tool|Clone」メニューオプションを使用します。元のものと同じプロパティを持つ、同じタイプのツールが新しく生成されます。必要に応じて名前を変更し、プロパティを調整します。

Deleting Tools

ツールを削除する時は、「Tool|Delete」メニューオプションを使用します。ツールボックスからツールが削除されます。削除前に確認のメッセージが表示されますが、削除を取り消す方法はありません。Ctrl キーまたは Shift キーを押しながら選択すると、複数のツールを同時に選択し、削除できます。

インポートツールとエクスポートツール

様々な NetDynamics アプリケーションの移行に焦点を当てた複数のツールボックス、つまり拡張子が .toolbox のファイルを持つことができます。インポートコマンドでツールを .tools ファイルにエクスポートし、インポートコマンドで別のツールボックス、つまり .toolbox ファイルにインポートできます。

ツールをエクスポートする場合は、そのツールが含まれるツールボックスを開き、ツールボックスのツリーでツールを選択し、「File|Export」メニューオプションを使用して、ツールをエクスポートする .tools ファイルの名称を入力します。現在のツールボックスからツールは削除されません。

ツールを別のツールボックスにインポートする場合は、インポート先のツールボックスを開き、「File|Import」メニューオプションを使用し、インポートする .tools ファイルを指定し、ツールボックスを保存します。

ツールボックスのマージ

2 つのツールボックスを 1 つにマージする場合は、「Open Toolbox」メニューオプションの「Open Toolbox」機能を使用します。2 つのツールボックスをマージする場合は、まず一方のツールボックスを開き、開いた状態でもう一方のツールボックスを開きます。先に開いたツールボックスの内容を置き換えるか、2 つのツールボックスをマージするか、処理を中止するかを尋ねるメッセージが表示されます。

トラブルシューティング

重要 : 先に進む前に、最新の S1MT パッチを Sun ONE Migration Website からダウンロードし、適用しておきます。問題が発見され、原因が特定されるとパッチがリリースされます。これらのパッチのほとんどは S1MT のユーザーによって発見された問題を解決するためのものです。問題を発見したら S1MT チームに連絡してください。原因を特定し、必要に応じてパッチをリリースします。

ツールボックスのインストールおよび設定

Toolbox アプリケーションで問題があった場合は次の点を確認してください。

- すべての %MIGTBX_HOME%/bin/setenv.bat スクリプトが環境に合わせてカスタマイズされているかどうか。JDK が制限されているため、ディレクトリ名に空白を含む場合はそのパスに S1MT をインストールできません。たとえばアーカイブを C:¥Program Files ディレクトリには解凍しないでください。c:¥iPlanet または c:¥ にアーカイブを解凍することをお勧めします。
- WinZip の古いバージョンで JDK の zip/jar ツールで作成されたアーカイブを解凍すると問題が発生することが報告されています。解凍時にファイルが切り捨てられ、長さが 0 バイトになってしまいます。したがって、S1MT アーカイブの解凍時には http://www.winzip.com を参照し、最新版の WinZip を使用しているかどうかを確認してください。
- クラスのバージョン問題を回避するため、Toolbox アプリケーションを実行する時にはすべての JAR ファイルを JDK の拡張ディレクトリ、%JAVA_HOME%/jre/lib/ext から必ず削除することをお勧めします。Toolbox の実行に必要なクラスはすべてバージョンに含まれています。拡張ディレクトリの JAR ファイルの名称を変更するだけでは不十分であり、別の場所に移動する必要があります。

- 複数のコピーまたはバージョンの JDK がインストールされている開発マシンが多いため、どの JDK を使用しているかを確認してください。JAVA_HOME 環境変数を %MIGTBX_HOME%/bin/setenv.bat ファイルに設定し、Toolbox アプリケーションに対して適切なコピーを使用するようにしてください。

抽出

これまで説明してきたように、アプリケーション記述の抽出はエラーや問題がもっとも発生しやすい部分です。また、前述したとおり、このような失敗は**移行プロセスでは頻繁に発生すること**であり、もしエラーが発生したとしても落胆する必要はありません。これまでに説明されていないような問題が発生した場合は、次のヒントを参考にしてください。

一般的な問題

- 抽出時にはすべての外部クラス、つまり NetDynamics プロジェクトクラス以外のクラスが Toolbox のクラスパスに置かれます。これらのクラスを利用可能にする最も簡単な方法は、JAR ファイルまたはパッケージされていないクラスを %MIGTBX_HOME%/lib/ext ディレクトリに配置することです。このディレクトリのクラスおよび JAR ファイルは、起動時に Toolbox クラスパスに自動的に追加されます。これで不十分な場合は、クラスをクラスパスに追加するか、または %MIGTBX_HOME%/bin/setclasspath.bat ファイルを編集することができます。
- 抽出ツールおよび変換ツールの出力の最後のコメントを確認し、自動プロセスで処理できなかったプロジェクトオブジェクトがあるかどうかを判断します。
- 埋め込み型 NetDynamics ランタイムの使用には固有の制限があるため、抽出で例外がスローされてもレポートされたツールのステータスには影響しません。また実際は抽出でエラーが発生しても正常終了としてレポートされる場合もあります。したがって、抽出時にスローされた例外はすべて記録して調査することを強くお勧めします。影響がないと思われた例外の副作用によって、抽出されたプロジェクト情報が不正確なものになってしまったという例もあります。たとえば、ある抽出時に、一見あまり重要でないクラスを検索していた NetDynamics ランタイムが ClassNotFoundException をスローしたことがありました。この例外が原因となって特定の DataObject プロパティを抽出できなくなり、移行アプリケーションが正常に動作しなくなりました。したがって、移行をできるだけ完全なものとするためには、抽出フェーズの例外の原因をすべてなくしてから先に進むようにします。
- 埋め込み型 NetDynamics CP は、プロジェクトの抽出前と抽出後で別々のプロジェクトのコピーをインスタンス化するようになっています。通常これによる悪影響はありませんが、インスタンス化の時にプロジェクトが例外をスローした場合は、Toolbox のコンソールログに 2 セットのスタックトレースが表示されることになります。

抽出時の致命的でないエラー

自動移行で一部エラーが発生した場合は次のようにしてください。

- 先のセクションで説明したヒントを参考にしてエラーの原因を検出および修正し、抽出または変換を必要に応じて再実行します。
- NetDynamics による移行で問題が発生した場合は、NetDynamics Studio で新しいプロジェクトを作成し、問題の原因になったオブジェクトをインポートします。オブジェクトを減らしていき、プロジェクトを適切に実行できるツールを探します。これらのファイルを元の移行済のオブジェクトにコピーします。
- エラーとなったオブジェクトの移行は手作業で行います。これはそれほど大変な作業ではありません。JATO フレームワークもアプリケーションを手作業でオーサリングするために設計されています。`application` パッケージのテンプレートにある、同じタイプの移行オブジェクトのサンプルを使用して処理を行います。新しい JATO オブジェクトの手作業による生成方法を記載したマニュアルが作成されています。JATO eGroups フォーラムの「ファイル」をチェックしてください。
- 問題はできるだけ完全に原因を突き止めてから、ディスカッションフォーラムまたは S1MT チームに相談してください。

抽出時の致命的なエラー

次の項目がエラーの原因になっていないかどうかを確認してください。同様の他の項目が原因になっている場合もあります。

1. 環境設定が正しくない。`%MIGTBX_HOME%/bin/setenv.bat` ファイルの設定が環境に合っているかどうかを確認してください。
2. 外部クラスがない
3. ツールのプロパティ設定が正しくない。*Extraction Tool* のプロパティ設定が正しいかどうか確認します。
4. 重要な箇所で存在しないランタイム機能を使用している。バックグラウンドタスク実行のために Spider スレッド以外のクラス初期化を使用している場合など
5. `links` ディレクトリが存在しないか、クラスファイルが壊れている
6. 使用している JDK バージョンまたはプラットフォームが正しくない
7. ブートクラスパスのクラスファイルバージョンの矛盾。JDK の拡張ディレクトリにクラスファイルが存在する場合など

問題の原因が上記のどれにもあてはまらない場合は、S1MT のバグを発見した可能性があります。NetDynamics によるプロジェクト開発は自由度が大きいため、Sun ONE ではすべての状況を予測することができず、すべての顧客に対して移行時にエラーが発生しないと断言することはできません。しかし、S1MT ではできるだけ移行を容易に行えるよう努力しています。問題が発生した場合は S1MT チームまたはディスカッションフォーラムに報告してください。すぐに対処し、必要に応じてパッチをリリースします。

変換

アプリケーション変換でエラーが発生した場合は、最初に次の処理を行います。

- アプリケーション記述ファイルの内容が完全であるかどうか、また XML として有効かどうかを確認します。XMLSpy や Internet Explorer などのツールを使用してドキュメントを開き、内容を確認します。
- Translation Tool の設定が正しいかどうかを確認します。
- `%MIGTBX_HOME%/bin/setenv.bat` ファイルの設定が環境に合っているかどうかをチェックします。
- Toolbox が完全にインストールされているかどうかを確認します。

問題の原因が上記のどれにもあてはまらない場合は、S1MT のバグを発見した可能性があります。NetDynamics によるプロジェクト開発は自由度が大きいため、Sun ONE ではすべての状況を予測することができず、すべての顧客に対して移行時にエラーが発生しないと断言することはできません。しかし、S1MT ではできるだけ移行を容易に行えるよう努力しています。問題が発生した場合は S1MT チームまたはディスカッションフォーラムに報告してください。すぐに対処し、必要に応じてパッチをリリースします。

移行後の問題

移行後またはテスト時に問題が発生する場合があります。一般にこのような問題はディスカッションフォーラムに投稿するか、S1MT チームと解決策を話し合う必要があります。ただし連絡する前に次の項目をチェックします。

- 各サーブレットのモジュール URL および各ビュー bean の表示 URL が、プロジェクト変換時に特定のデフォルトに設定されている。これらのデフォルトは配備環境では正しいものであっても、場合によっては正しくない場合があります。『JATO Deployment Guide』を参照するか、またはディスカッションフォーラムで相談し、これらの URL を配備と異なる設定にする方法を確認します。
- JDBC ドライバの特定の列タイプ処理に不整合がある。JATO には、特定のデータベースを使用してアプリケーションを動かす場合に変更しなければならないオプションが いくつかあります。ターゲットのデータベースで移行したアプリケーションを動作させるのが困難な場合は、Sun ONE Migration Web サイトを参照するか、またはディスカッションフォーラムで相談し、特定のデータベース関連の微調整についての情報を確認します。

付録 C

EJB 1.1 から EJB 2.0 への移行

Sun ONE Application Server 7 でも EJB 1.1 仕様は引き続きサポートされますが、その拡張機能を活用するため、EJB 2.0 アーキテクチャの使用をお勧めします。

EJB 1.1 を EJB 2.0 へ移行するには、コンポーネントのソースコード内部を含め、多くの修正が必要です。

基本的に、必要とされる修正は、EJB 1.1 と EJB 2.0 の間の相違に関連しています。これらの相違のすべてについて、次の各項目で説明します。

- 「EJB クエリ言語」
- 「ローカルインタフェース」
- 「EJB 2.0 コンテナ管理による持続性 (CMP)」
- 「持続性フィールドの定義」
- 「エンティティ Bean の関係の定義」
- 「メッセージ駆動型 Beans」

EJB クエリ言語

EJB 1.1 仕様には、個別のアプリケーションサーバーに対する検索メソッドのクエリを形成し、記述する方法および言語が残されています。多くのアプリケーションサーバーのベンダーは、開発者が SQL を使用してクエリを作成するよう求めますが、一方で、特定のアプリケーションサーバー製品に固有の、独自の専用言語を使用する場合もあります。このようにクエリの実装が混在しているため、アプリケーションサーバー間の不整合が生じています。

EJB 2.0 仕様では、これらの不整合および不備を修正した *EJB Query Language*、または *EJB QL* と呼ばれるクエリ言語が導入されています。EJB QL は、SQL92 に基づいています。EJB QL は、検索と選択メソッドの両形式で、特にコンテナ管理による持続性を使用したエンティティ Beans 用にクエリメソッドを定義します。SQL に対する EJB QL の主な長所は、EJB コンテナを通じた移植性と、エンティティ Beans の関係をナビゲートする機能にあります。

ローカルインタフェース

EJB 1.1 アーキテクチャでは、セッション Beans およびエンティティ Beans にリモートインタフェースという 1 つのインタフェースタイプがあり、そのインタフェースを通じて、クライアントや他のアプリケーションコンポーネントからのアクセスが可能になります。リモートインタフェースは、Bean インスタンスにリモート機能があるものとして設計されています。Bean は RMI からそれを継承し、ネットワークを通じて分散クライアントとやり取りを行うことができます。

EJB 2.0 では、セッション Beans およびエンティティ Beans はクライアントにメソッドを公開するインタフェースを 2 種類利用できます。1 つはリモートインタフェース、もう 1 つはローカルインタフェースです。2.0 リモートインタフェースは、1.1 アーキテクチャで使用されているリモートインタフェースと同じものです。Bean は RMI からリモートインタフェースを継承し、ネットワーク層を通じてそのメソッドを公開します。また、分散クライアントとやり取りするための同一の機能を備えています。

ただし、セッションおよびエンティティ Beans 用のローカルインタフェースは、ローカルクライアント、つまり同一の EJB コンテナに共存しているクライアントである EJB からの軽量アクセスをサポートしています。EJB 2.0 仕様では、さらに、ローカルインタフェースを使用した同一アプリケーション内にある EJB を必要とします。つまり、ローカルインタフェースを使用したアプリケーションの EJB 用の配備記述子が、1 つの `ejb-jar` ファイルに含まれている必要があります。

ローカルインタフェースは、標準 Java インタフェースです。これは、RMI からは継承されません。Enterprise JavaBeans は、ローカルインタフェースを使用して、同一コンテナ内にある他の Beans にメソッドを公開します。ローカルインタフェースを使用することにより、Bean はクライアントと緊密に連携でき、リモートメソッド呼び出しのオーバーヘッドなしに直接アクセスが可能になります。

さらに、ローカルインタフェースは、参照セマンティクスにより Beans 間で渡される値を許可します。この方法では、オブジェクト自体ではなくオブジェクトへの参照を渡すため、大量のデータを持つオブジェクトを渡す際に発生するオーバーヘッドが軽減し、パフォーマンス上の利点があります。

リモートインタフェースではなくローカルインタフェースを使用すると、セッションまたはエンティティ Bean の設定は簡単になります。クライアントに公開されている Bean のメソッド経由のローカルインタフェースは、`EJBObject` ではなく `EJBLocalObject` を拡張します。同様に、Bean のホームインタフェースは、`EJBHome` ではなく `EJBLocalHome` を拡張します。実装クラスは、同じ EntityBean または SessionBean インタフェースを拡張します。

注　EJB 2.0 でリモートに予定されている Bean は、リモートインタフェースで `EJBObject` を、ホームインタフェースで EJBHome を、EJB 1.1 の場合と同様に拡張します。

EJB 2.0 コンテナ管理による持続性 (CMP)

EJB 2.0 仕様では、CMP を拡張し、CMP 間で関係のある複数のエンティティ Beans を許可します。これは、コンテナ管理による関係 (CMR) と呼ばれます。コンテナでは、関係と、関係の参照整合性を管理します。

EJB 1.1 仕様で提示されていた CMP モデルは、より制限の多いものでした。1.1 アーキテクチャでの CMP は、データベースやリソースマネージャタイプから独立したデータアクセスに制限されていました。CMP は、リモートインタフェースを経由したエンティティ Bean のインスタンスの状態表示だけを行い、Bean の関係は表示しませんでした。CMP の 1.1 バージョンでは、エンティティ Bean クラスのインスタンス変数を、データベースまたはリソースマネージャで状態を表すデータ項目にマッピングする必要があります。CMP インスタンスフィールドは配備記述子で指定されており、Bean を配備する際に、記述子はツールを使用してインスタンスフィールドのデータ項目へのマッピングを実装するコードを生成します。

Bean の実装クラスのコーディング方法も変更する必要があります。2.0 仕様に従って CMP を使用するエンティティ Bean 用の実装クラスは、抽象クラスとして定義されます。

持続性フィールドの定義

EJB 2.0 仕様では、エンティティ Bean のインスタンス変数を CMP フィールドまたは CMR フィールドとして指定できます。これらのフィールドは、配備記述子で定義します。CMP フィールドは、`cmp-field` 要素でマークされ、コンテナ管理による関係のフィールドは、`cmr-field` 要素でマークされます。

実装クラスでは、CMP および CMR フィールドを public 変数として宣言しないことに注意してください。代わりに、エンティティ Bean で `get` および `set` メソッドを定義して、これらの CMP および CMR フィールドの値を取得し、設定することができます。このような意味で、2.0 CMP を使用した Beans は、JavaBeans モデルに準拠しています。クライアントは、インスタンス変数に直接アクセスする代わりに、エンティティ Bean の `get` および `set` メソッドを使用してこれらのインスタンス変数を取得し、設定します。`get` および `set` メソッドは、CMP または CMR フィールドとして指定された変数にのみ関連しています。

エンティティ Bean の関係の定義

前述したとおり、EJB 1.1 アーキテクチャでは、エンティティ Beans 間の CMR をサポートしていません。EJB 2.0 アーキテクチャは、1 対 1、および 1 対多の CMR を両方ともサポートしています。関係は CMR フィールドを使用して表現されており、これらのフィールドは、配備記述子に関係としてマークされています。アプリケーションサーバーで適切な配備ツールを使用して、配備記述子に CMR フィールドを設定します。

CMP フィールドと同様、Bean はインスタンス変数として CMR フィールドを宣言しません。代わりに、Bean は、これらのフィールド用に `get` および `set` メソッドを備えています。

メッセージ駆動型 Beans

メッセージ駆動型 Beans は、EJB 2.0 アーキテクチャで導入されたもう一つの新しい機能です。メッセージ駆動型 Beans は、Java Message Service (JMS) 経由で配布された非同期メッセージを処理するトランザクション認識型のコンポーネントです。JMS API は、J2EE 1.3 プラットフォームの重要な部分です。

非同期メッセージングでは、アプリケーションはメッセージの交換によって通信することができるため、送信者と受信者を独立させることができます。送信者は、メッセージを送信した後に、受信者がそのメッセージを受信または処理するのを待つ必要はありません。これは、同期通信のプロセスとは異なります。同期通信では、別のコンポーネント上のメソッドを呼び出し、処理が完了して呼び出しコンポーネントに制御が戻るまで待機またはブロックするコンポーネントが必要です。

EJB クライアントアプリケーションの移行

この節には次の項目があります。

- 「JNDI コンテキスト内での EJB の宣言」
- 「EJB JNDI 参照の使用方法の要約」

JNDI コンテキスト内での EJB の宣言

Sun ONE Application Server 7 では、EJB は JNDI のサブコンテキスト *ejb/* に計画的にマッピングされています。JNDI 名 *Account* を EJB に所属させると、Sun ONE Application Server 7 は参照 *ejb/Account* をグローバル JNDI コンテキストに自動生成します。したがって、この EJB のクライアントは、対応するホームインタフェースを取得するために *ejb/Account* を検索する必要があります。

Sun ONE Application Server 6.0/6.5 に配備されているサーブレットメソッドのコードを見てみましょう。

ここに示すサーブレットは、JNDI コンテキストのルートにマッピングされているステートフルセッション Bean の BankTeller を呼び出します。ここに挙げるコードでは EJB のホームインタフェースを取得し、BankTeller オブジェクトのインスタンス化とこのオブジェクトのリモートインタフェース取得を可能にし、ビジネスメソッドからのこのコンポーネントの呼び出しを可能にします。

```
/**
   * Look up the BankTellerHome interface using JNDI.
```

```
   */
private BankTellerHome lookupBankTellerHome(Context ctx)
    throws NamingException
{
    try
    {
      Object home = (BankTellerHome) ctx.lookup("ejb/BankTeller");
      return (BankTellerHome) PortableRemoteObject.narrow(home,
BankTellerHome.class);
    }
    catch (NamingException ne)
    {
     log("lookupBankTellerHome: unable to lookup BankTellerHome" +
        "with JNDI name 'BankTeller': " + ne.getMessage() );
     throw ne;
    }
}
```

このコードでは ejb/BankTeller を検索の引数としてすでに使用していますが、Sun ONE Application Server 7 に配備されるコードを変更する必要はありません。

EJB JNDI 参照の使用方法の要約

この節では、EJB JNDI 参照を使用した場合の注意事項をまとめて説明します。ここで説明する注意事項の詳細については、特定のソースアプリケーションサーバーのプラットフォームに固有のものであることに注意してください。

JNDI コンテキスト内の EJB 参照の配置

既存の WebLogic アプリケーションで JNDI コンテキストのルートに EJB がマップされている場合は、上述の JNDI コンテキスト内の EJB 参照の名前だけを変更する必要があります。変更によってこれらの参照が JNDI コンテキストルートからサブコンテキスト *ejb/* に移動します。

これらの EJB が、既存のアプリケーションの JNDI サブコンテキスト `ejb/` にすでにマップされている場合、変更の必要はありません。

ただし、重要なのは、Forte for Java の IDE 内の配備記述子に EJB の JNDI 名を設定する場合に、EJB の JNDI 名の中に接頭辞 ejb/ を含めないことです。これらの EJB 参照は、Sun ONE Application Server 7 では自動的に JNDI の ejb/ サブコンテキスト内に配備されることに留意してください。このため、EJB に、その配備記述子で *BankTeller* という JNDI 名が与えられた場合、この EJB への参照は Sun ONE Application Server によって ejb/BankTeller に「変換」されます。この EJB のクライアントコンポーネントは検索時にこの JNDI 名を使用することになります。

グローバル JNDI コンテキストとローカル JNDI コンテキスト

EJB 参照を取得するためにグローバル JNDI コンテキストを使用することは、Sun ONE Application Server 7 では完全に有効かつ最適なアプローチです。しかしながら、J2EE 仕様にできるだけ近い形で、EJB クライアントアプリケーションのローカル JNDI コンテキストを経由して EJB 参照を取得することをお勧めします。ローカル JNDI コンテキストを使用する場合、まずクライアント部分の配備記述子で EJB リソース参照を宣言する必要があります。これは Web アプリケーションでは web.xml、EJB コンポーネントでは ejb-jar.xml になります。

CMP エンティティ EJB の移行

この節では、アプリケーションコンポーネントを EJB 1.1 アーキテクチャから EJB 2.0 アーキテクチャに移行する手順について説明します。

CMP 1.1 の Bean を CMP 2.0 に移行するためには、まず特定の Bean が移行可能であることを確認する必要があります。確認の手順は次のとおりです。

1. ejb-jar.xml ファイルから <cmp-field> 名へ移動し、オプションのタグ <prim-key-field> が ejb-jar.xml 内にあり、指示された値を保持していることを確認します。問題がなければ、次の手順に進みます。

   ejb-jar.xml 内でフィールド名 <prim-key-class> を探し、クラス名を取得して、クラス内で宣言された public instance variables を取得します。これらの変数のシグネチャ (名前と大文字小文字の区別) が上記の <cmp-field> 名と一致するかどうかを確認します。見つかったフィールドは分割します。これらの分割されたフィールドで、大文字で始まっているものがあるかどうかを調べます。もしある場合は、移行は実行できません。

2. Bean クラスのソースコードを調べて、すべての <cmp-field> 変数の java 型を取得します。

3. すべての <cmp-field> 名を小文字に変更し、それらの名前からアクセサを構築します。たとえば、元のフィールド名が Name で、その java 型が String である場合、アクセサメソッドシグネチャは次のようになります。

   ```
   Public void setName(String name)

      Public String getName()
   ```

4. これらのアクセサメソッドシグネチャを、Bean クラスのメソッドシグネチャと比較します。正確に一致しない場合は、移行は実行されません。
5. カスタム検索のメソッドシグネチャと、対応する SQL を取得します。SQL 内に Join 、Outer join 、OrderBy があるかどうかを確認します。ある場合は、移行は実行できません。これは、EJB QL では joins 、Outer join 、および OrderBy をサポートしていないためです。
6. CMP 1.1 検索が java.util.Enumeration を使用している場合は、代わりに java.util.Collection を使用する必要があります。これを反映するようにコードを修正してください。CMP2.0 検索では、java.util.Enumeration を返すことができません。

次の項目、「Bean クラスの移行」では移行プロセスを実行します。

Bean クラスの移行

この節では、Bean クラスの Sun ONE Application Server への移行に必要な手順について説明します。

1. Bean クラス宣言の先頭にキーワード *abstract* を付けます。たとえば、Bean クラス宣言は次のようになります。

   ```
   Public class CabinBean implements EntityBean // before
   modification

   abstract Public class CabinBean implements EntityBean // after
   modification
   ```

2. アクセサの接頭辞にキーワード abstract を付けます。
3. 変更後、Bean クラスのソースファイル、つまり拡張子 .java のファイルにすべてのアクセサをクラスレベルで挿入します。
4. Bean クラスのソースファイル内で、すべての cmp フィールドをコメントにします。
5. 小文字の cmp-field 名で protected インスタンス変数の宣言を構築し、クラスレベルでそれを挿入します。

6. すべての ejbCreate() メソッド本体 (複数の ejbCreate が存在する場合もある) を読み込みます。パターン「<cmp-field>= 値またはローカル変数」を検索し、「abstract ミューテータメソッド名 (値またはローカル変数)」に置き換えます。たとえば、移行前の ejbCreate の本体が次のとおりであるとします。

```
public MyPK ejbCreate(int id, String name)
       {
    this.id = 10*id;
    Name = name;//1
    return null;
    }
```

移行後、メソッド本体は変更され、次のようになります。

```
public MyPK ejbCreate(int id, String name)
       {
    setId(10*id);
    setName(name);//1
    return null;
       }
```

注 //1 の abstract アクセサのメソッドシグネチャは、EJB 2.0 仕様で定められた Camel Case 規約に従っています。また、キーワード「*this*」は、元のソース内に存在する場合としない場合がありますが、変更後のソースファイルからは削除する必要があります。

7. 手順 5 で ejbPostCreate() メソッドに宣言したすべての protected 変数を初期化する必要があります。protected 変数の数は、ejbCreate() メソッドの数と同じになります。初期化は、次のように初期化コードを挿入することにより行われます。

```
protected String name;//from step 5

protected int id;//from step 5

public void ejbPostCreate(int id, String name)

    {

name /*protected variable*/ = getName();/*abstract accessor*/
//inserted in this step

id /*protected variable*/ = getId();/*abstract accessor*/
//inserted in this step

    }
```

8. ejbLoad メソッドの内部で、Beans のデータベースの状態に protected 変数を設定する必要があります。次のコードを挿入します。

```
public void ejbLoad()

    {

    name = getName();//inserted in this step

    id = getId(); //inserted in this step

           ……..       //already present code

    }
```

9. 同様に、データベースの状態が更新されるように、ejbStore() 内部の Beans の状態を更新する必要があります。ただし、ejbCreate() の外部の主キーに対応するセッターは更新できないため、このメソッドの内部にはセッターを含めないでください。次のコードを挿入します。

```
public void ejbStore()

    {

    setName(name);//inserted in this step

//  setId(id);//Do not insert this if it is a part of the
primary key

        ……..//already present code

    }
```

10. Bean クラスソース、つまり拡張子 .java のファイルに対する最後の変更は、コード全体を調べ、すべての <cmp-field> 変数名を、手順 5 で宣言した対応する protected 変数名と置き換えることです。

    Bean を移行しない場合、少なくとも <cmp-version>1.*x*</cmp-version> タグを ejb-jar.xml 内部の適切な場所に挿入する必要があります。これで、移行しない Bean も Sun ONE Application Server 上で動作し続けます。

ejb-jar.xml の移行

ejb-jar.xml ファイルを Sun ONE Application Server に移行するには、次の手順を実行します。

1. ejb-jar.xml で、すべての <cmp-field> を小文字に変更します。
2. ejb-jar.xml ファイルで、<reentrant> タグの後に <abstract-schema-name> タグを挿入します。このスキーマ名は、<ejb-name> タグと同様、接頭辞 ias_ が付いた Bean 名になります。
3. <primkey-field> タグの後に、次のタグを挿入します。

   ```
   <security-identity><use-caller-identity/></security-identity>
   ```

4. SQL から EJB QL を構築するために、上記で取得した SQL を使用します。
5. <query> タグ、およびネストされた子タグすべてを、すべての必要な情報とともに ejb-jar.xml に挿入します。挿入位置は <security-identity> タグのすぐ後です。

カスタム検索メソッド

カスタム検索メソッドは、エンティティ Bean のホームインタフェースで定義される findBy... メソッドです。デフォルトの findByPrimaryKey メソッドとは異なります。EJB 1.1 仕様は、これらの検索メソッドのロジックを定義する標準を定めていないため、EJB サーバーのベンダーは、実装を自由に選択できます。その結果、メソッドの定義に使用される手順は、ベンダーが選択する実装によって大幅に異なります。

Sun ONE Application Server 6.0 および 6.5 では、ファインダーロジックの指定に標準 SQL を使用します。

EJB の持続性記述子 Account-ias-cmp.xml に格納されるこの検索メソッドの定義に関する情報は、次のとおりです。

```
<bean-property>
  <property>
    <name>findOrderedAccountsForCustomerSQL</name>
    <type>java.lang.String</type>
    <value>
```

```
        SELECT BRANCH_CODE,ACC_NO FROM ACCOUNT where CUST_NO = ?
    </value>
    <delimiter>,</delimiter>
  </property>
</bean-property>
<bean-property>
  <property>
    <name>findOrderedAccountsForCustomerParms</name>
    <type>java.lang.Vector</type>
    <value>CustNo</value>
    <delimiter>,</delimiter>
  </property>
</bean-property>
```

このように、各 `findXXX` 検索メソッドには、配備記述子 (クエリ用 SQL コード、および関連パラメータ) 内に対応する 2 つのエントリがあります。

Sun ONE Application Server では、カスタム検索メソッドロジックも宣言型ですが、EJB のクエリ言語、EJB QL に基づいています。

EJB-QL 言語自体は使用できません。`ejb-jar.xml` ファイル内部の、`<ejb-ql>` タグで指定する必要があります。このタグは、`<query>` タグの内部にあり、EJB 内にクエリ (検索または選択メソッド) を定義します。EJB コンテナでは、各クエリを検索または選択メソッドの実装に変換できます。ここでは、`<ejb-ql>` タグの例を示します。

```
<ejb-jar>
  <enterprise-beans>
    <entity>
    <ejb-name>hotelEJB</ejb-name>
      ...
      <abstract-schema-name>TMBankSchemaName</abstract-schema-name>
      <cmp-field>...
      ...
      <query>
        <query-method>
          <method-name>findByCity</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
```

```
        <ejb-ql>
          <![CDATA[SELECT OBJECT(t) FROM TMBankSchemaName AS t
WHERE t.city = ?1]]>
       </ejb-ql>
      </query>
    </entity>
  ...
  </enterprise-beans>
...
</ejb-jar>
```

索引

O

P

R

S

T

U

せ

つ

て

は

ほ

り