# J2EE™ アプリケーションのプログラミング

Sun ONE™ Studio プログラミングシリーズ

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 817-0839-10
2002 年 9 月 Revision A

**原典：** *Building J2EE Applications With Sun ONE Studio 4*
Part No: 816-7863-10
Revision A

# 目次

# 図目次

# はじめに

Sun™ Open Net Environment (Sun ONE) Studio 4 統合開発環境 (IDE) については、Sun ONE Studio 4 プログラミングシリーズという一連のマニュアルで解説されています。この『J2EE アプリケーションのプログラミング』では、IDE を使用して、Java™ 2 Platform, Enterprise Edition の構造を満たすアプリケーション (J2EE™ アプリケーション) をアセンブル、配備、実行する手順を学習します。

このマニュアルで説明しているプログラム例は、実際に作成することができます。作業環境については、以下の Web サイトにあるリリースノートを参照してください。

`http://sun.co.jp/forte/ffj/documentation/index.html`

使用するプラットフォームによっては、このマニュアルに掲載している画面イメージと異なることがあります。その場合でも表示上の違いはわずかであるため、内容を理解するのには問題ありません。ほとんどの手順で Sun ONE Studio 4 ソフトウェアのユーザーインタフェースを使用しますが、場合によっては、コマンド行にコマンドを入力する必要があります。その場合は、Microsoft Windows の「コマンドプロンプトウィンドウ」で次の構文を入力します。

```
c:\>cd MyWorkSpace\MyPackage
```

UNIX® や Linux 環境では、次のようなプロンプトとなり、¥ マーク (またはバックスラッシュ) ではなくスラッシュを使用します。

```
% cd MyWorkSpace/MyPackage
```

# お読みになる前に

このマニュアルは、Sun ONE Studio 4 IDE を使用してアプリケーションをアセンブル、配備、または実行するすべてのユーザーを対象としています。最初の章は、J2EE プラットフォームのアセンブルと配備の概念をまとめています。この章は、アセンブルと配備に関する一般的な理解を求めるすべてのユーザーに役立ちます。

このマニュアルを理解するには、次の知識が必要です。

- Java プログラミング言語
- J2EE の概念
- Web サーバーソフトウェアおよびアプリケーションサーバーソフトウェア

このマニュアルを理解するには、次の資料で説明されている J2EE の概念を知っている必要があります。

- Java 2 Platform, Enterprise Edition Blueprints, Version 1.3
  `http://java.sun.com/j2ee/blueprints`
- *Java 2 Platform, Enterprise Edition Specification*, v 1.4
  `http://java.sun.com/j2ee/download.html#platformspec`
- The J2EE Tutorial (J2EE SDK バージョン 1.3向け)
  `http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html`
- Java Servlet Specification Version 2.3
  `http://java.sun.com/products/servlet/download.html#specs`
- JavaServer Pages Specification Version 1.2
  `http://java.sun.com/products/jsp/download.html#specs`

---

**注 -** Sun では、本マニュアルに掲載した第三者の Web サイトのご利用に関しましては責任はなく、保証するものでもありません。また、これらのサイトあるいはリソースに関する、あるいはこれらのサイト、リソースから利用可能であるコンテンツ、広告、製品、あるいは資料に関して一切の責任を負いません。Sun は、これらのサイトあるいはリソースに関する、あるいはこれらのサイトから利用可能であるコンテンツ、製品、サービスのご利用あるいは信頼によって、あるいはそれに関連して発生するいかなる損害、損失、申し立てに対する一切の責任を負いません。

---

# 内容の紹介

J2EE は、企業アプリケーション開発へのコンポーネント指向のアプローチです。ビジネスロジックは、エンタープライズ JavaBean™ (EJB™) コンポーネントおよび Web コンポーネントにカプセル化されています。コンポーネントの集合体がモジュールです。モジュールは、認識可能なビジネスタスクを実行するロジックの単位となります。モジュールの集合体がア J2EE プリケーションです。J2EE アプリケーションは、ビジネス処理全体を実行します。

J2EE プラットフォームは、Java RMI や Java Messaging Serviceなど、アプリケーション同士が通信するための多くの方法を提供します。このマニュアルは、Sun ONE Studio 4 開発環境を使用し、コンポーネントをアセンブルしてモジュールを作成する方法、さらにモジュールをアセンブルしてアプリケーションを作成する方法を説明します。 これらの情報は、「シナリオ」によって示されています。

第 1 章では、J2EE のアセンブルと配備の概念をまとめています。モジュールとアプリケーションの J2EE ユニットを示し、モジュールおよびアプリケーション配備記述子について説明します。また、モジュールとアプリケーションを IDE でアセンブルする方法を説明します。特に、モジュールとアプリケーションのプロパティシートを使用して、モジュールとアプリケーションの配備記述子を設定する方法を取り上げます。

第 2 章は、Web モジュールのアセンブル方法を示すシナリオです。Web モジュールを J2EE アプリケーションのフロントエンドとして利用する短いプログラム例を紹介した後、J2EE アプリケーション内で利用可能な Web モジュールのプログラミング方法を説明します。

第 3 章は、EJB モジュールのアセンブル方法を示すシナリオです。J2EE アプリケーションで EJB モジュールを利用する短いプログラム例を紹介した後、いくつかのエンタープライズ bean を含むモジュールのプログラミング方法を説明します。

第 4 章は、Web モジュールと EJB モジュールを組み合わせて J2EE アプリケーションをアセンブルする方法を示すシナリオです。同じアプリケーションの中で2種類のモジュールを一緒に使う短いプログラム例を紹介した後、J2EE アプリケーションのプログラミング方法を説明します。特に、Java RMI を使用した2 つのモジュール間での同期対話を取り上げます。

第 5 章は、メッセージ駆動型エンタープライズ beans (MDB) を使用したモジュール間での非同期通信の設定方法を示すシナリオです。ビジネスアプリケーションの中で非同期通信を使う短いプログラム例を紹介した後、非同期通信の送信側と受信側の両

方のプログラミング方法を説明します。ここでは EJB モジュールと通信する Web モジュールを取り上げますが、プログラム例はほかのモジュールの組み合わせにも応用できます。

第 6 章では、J2EE アプリケーションクライアントとサーバー側 J2EE アプリケーションとの間の通信を設定する方法を示します。ビジネスアプリケーションでアプリケーションクライアントを使う短いプログラム例を紹介した後、アプリケーションクライアントのプログラミング方法を説明します。このシナリオでは、アプリケーションクライアントが Java RMI を使ってエンタープライズ bean ビジネスメソッドを同期しながら呼び出していますが、プログラム例は非同期通信を使うアプリケーションクライアントにも応用できます。

第 7 章では、IDE を使ってコンテナ管理トランザクションをプログラミングする方法を説明します。

第 8 章では、IDE を使用し J2EE アプリケーション内のリソースのセキュリティを保護する方法を説明します。モジュールレベルでセキュリティロールをセットアップする方法、セキュリティロールを使用して Web リソースやエンタープライズ bean モジュールへのアクセスを制限する方法を示します。また、モジュールを組み合わせてアプリケーションを作成するときのセキュリティロールのマップ方法を示します。

第 9 章では、アセンブルされたアプリケーションを配備して実行する方法を説明します。特に、モジュールとアプリケーションのプロパティシートを使用して、モジュールとアプリケーションの配備記述子を設定する方法について説明します。

付録 A では、Sun ONE Studio 4 IDEが Web サーバーおよびアプリケーションサーバーとの対話に使用するメカニズムを説明します。ここには、配備プロセスの詳しい説明を示します。

# 書体と記号について

| 書体または記号 | 意味 | 例 |
|---|---|---|
| `AaBbCc123` | コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コーディング例。 | `.cvspass` ファイルを編集します。<br>`DIR` を使用してすべてのファイルを表示します。<br>`Search is complete.` |
| **`AaBbCc123`** | ユーザーが入力する文字を、画面上のコンピュータ出力と区別して表わします。 | `> login`<br>`Password:` |
| *AaBbCc123* または ゴシック | コマンド行の可変部分。実際の名前または実際の値と置き換えてください。 | 削除するには **`DEL`** *filename* と入力します。<br>`rm` ファイル名 と入します。 |
| 『 』 | 参照する書名を示します。 | 『Solaris ユーザーマニュアル』 |
| 「 」 | 参照する章、節、または、強調する語を示します。 | 第 6 章「データの管理」を参照してください。<br>これらは、「クラス」オプションと呼ばれます。 |
| \ | 枠で囲まれたコード例で、テキストがページ行幅を超える場合、バックスラッシュは、継続を示します。 | `machinename% grep 'ˆ#define \`<br>`XV_VERSION_STRING'` |
| ➤ | 階層メニューのサブメニューを選択することを示します。 | 作成:「返信」➤「送信者へ」 |

## シェルプロンプトについて

| シェル | プロンプト |
|---|---|
| UNIX の C シェル | `machine_name%` |
| UNIX の Bourne シェルと Korn シェル | `machine_name$` |
| スーパーユーザー (シェルの種類を問わない) | # |

## 関連マニュアル

Sun ONE Studio 4 のマニュアルは、Acrobat Reader (PDF) ファイル、リリースノート、オンラインヘルプ、サンプルアプリケーションの `readme` ファイル、Javadoc™ 文書の形式で提供しています。

### オンラインで入手可能なマニュアル

以下に紹介するマニュアルは、Sun ONE Studio 4 のドキュメントサイト (http://sun.co.jp/forte/ffj/documentation/index.html) および `docs.sun.com`™ (http://docs.sun.com) から入手できます。

`docs.sun.com` ウェブサイトでは、サンのマニュアルをインターネットを通じて閲覧、印刷、購入することができます。サイト内でマニュアルを見つけられない場合には、製品と一緒にローカルシステムまたはローカルネットワークにインストールされているマニュアルインデックスを参照してください。

- リリースノート (HTML 形式)

  Sun ONE Studio 4 の Edition ごとに用意されています。このリリースでの変更情報と技術上の注意事項を説明しています。

- インストールガイド (PDF 形式)

  対応プラットフォームへの Sun ONE Studio 4 統合開発環境 (IDE) のインストール手順を説明しています。さらに、システム要件、アップグレード方法、Web サーバーやアプリケーションサーバーのインストール、コマンド行での操作、インストールされるサブディレクトリ、Javadoc の設定、データベースの統合、アップデートセンターの使用方法などが含まれます。

  - 『Sun ONE Studio 4, Community Edition インストールガイド』 - Part No. 817-0845-10
  - 『Sun ONE Studio 4, Enterprise Edition for Java インストールガイド』 - Part No. 817-0844-10
  - 『Sun ONE Studio 4, Mobile Edition インストールガイド』 - Part No. 817-0846-10

- Sun ONE Studio 4 プログラミングシリーズ (PDF 形式)

  Sun ONE Studio 4 の各機能を使用して、優れた J2EE アプリケーションを開発するための方法を詳細に説明しています。

  - 『Web コンポーネントのプログラミング』 - Part No. 817-0837-10

    JSP ページ、サーブレット、タグライブラリを使用し、クラスやファイルをサポートする Web アプリケーションを J2EE Web モジュールとして構築する方法を説明しています。

  - 『J2EE アプリケーションのプログラミング』 - Part No. 817-0839-10

    EJB モジュールや Web モジュールを J2EE にアセンブルする方法を説明しています。また、J2EE アプリケーションの配備や実行についても説明しています。

  - 『Enterprise JavaBeans コンポーネントのプログラミング』 - Part No. 817-0838-10

    Sun ONE Studio 4 の EJB ビルダーウィザードや、他の IDE コンポーネントを使用し、EJB コンポーネント (コンテナ管理や bean 管理の持続性の機能を持つセッション bean やエンティティ bean、メッセージ駆動型 bean) を作成する方法を説明しています。

- 『Web サービスのプログラミング』- Part No. 817-0816-10

  Sun ONE Studio 4 IDE を使用して Web サービスを構築したり、UDDI レジストリを経由して第三者に Web サービスを利用させたり、また、ローカル Web サービスや UDDI レジストリから Web サービスクライアントを生成する方法などを説明しています。

- 『Java DataBase Connectivity の使用』 - Part No. 817-0840-10

  Sun ONE Studio 4 IDE の JDBC 生産性向上ツールを使用し、JDBC アプリケーションを作成する方法について説明しています。

- Sun ONE Studio 4 チュートリアル (PDF 形式)

  Sun ONE Studio 4 の Edition ごとに用意されており、主な機能の活用方法を紹介しています。

  - 『Sun ONE Studio 4, Community Edition チュートリアル』
    - Part No. 817-0842-10

    簡単な J2EE Web アプリケーションの構築方法を順を追って解説します。

  - 『Sun ONE Studio 4, Enterprise Edition for Java チュートリアル』
    - Part No. 817-0841-10

    EJB コンポーネントと Web サービス技術を使用したアプリケーションの構築方法を順を追って解説します。

  - 『Sun ONE Studio 4, Mobile Edition チュートリアル』
    - Part No. 817-0843-10

    携帯やPDA 端末などの無線機器を対象とした簡単なアプリケーションの構築方法を順を追って解説します。このアプリケーションは Java 2 Platform, Micro Edition (J2ME™ プラットフォーム) に準拠し、Mobile Information Device Profile (MIDP) と Connected, Limited Device Configuration (CLDC) を満たすものです。

  チュートリアルアプリケーションは、以下のサイトからもアクセスできます。
  `http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

## オンラインヘルプ

オンラインヘルプは、Sun ONE Studio 4 IDE から参照できます。ヘルプを起動するには、ヘルプキー (Windows および Linux 環境では F1 キー、Solaris オペレーティング環境では Help キー) を押すか、「ヘルプ」 -> 「内容」を選択します。ヘルプの項目と検索機能が表示されます。

## プログラム例

Sun ONE Studio 4 の機能を紹介したプログラム例とチュートリアルアプリケーションを、以下の Sun ONE Studio Developer Resource のポータルサイトからダウンロードすることができます。

```
http://forte.sun.com/ffj/documentation/tutorialsandexamples.html
```

このチュートリアルで使用するアプリケーションも上記サイトに収録されています。

## Javadoc

Javadoc 形式のマニュアルは、Sun ONE Studio 4 の多くのモジュールに用意されており、IDE の中で参照できます。このマニュアルの使用方法については、リリースノートを参照してください。IDE を起動すると、エクスプローラの Javadoc タブで Javadoc マニュアルを参照できます。

# ご意見の送付先

Sun のマニュアルについてのご意見やご要望をお寄せください。今後のマニュアル作成の参考にさせていただきます。次のアドレスまで電子メールをお送りください。

```
docfeedback@sun.com
```

電子メールのタイトルに、このマニュアルの Part No. (817-0839-10) を明記してください。

# 第1章

# アセンブル、配備、実行の基本

J2EE プラットフォームはモジュール式です。つまり、小さな単位を組み合わせて大きな単位にする方式です。コンポーネントを組み合わせてモジュールを作成し、複数のモジュールを組み合わせてアプリケーションを作成します。このように小さな単位を組み合わせて大きな単位を作成するのは、アセンブルと呼ばれています。

J2EE のもう 1 つの重要な特徴は、アプリケーションが J2EE プラットフォームの提供する実行時サービスを使用することです。このような実行時サービスには、コンテナ管理の持続性、コンテナ管理のトランザクション、コンテナ管理のセキュリティ妥当性検査があります。コンポーネントからモジュールを作成し、モジュールからアプリケーションを作成するときは、どの実行時サービスが必要かを判断し、必要なサービスを J2EE 配備記述子に明確に指定しなければなりません。このようにモジュールやアプリケーションに必要な実行時サービスを要求することもアセンブル処理の一部分です。

この章では、アセンブルを行うときに考えなければならない J2EE モジュールと J2EE アプリケーションの基本的な特徴を確認します。また、Sun ONE Studio 4 IDE を使用したアセンブルの基本も紹介します。

## アセンブルの基本

アセンブルは、多くの独立した開発作業を含む J2EE の概念です。アセンブル作業を正しく実行すれば、J2EE アプリケーションサーバーに配備され、サーバーの環境で実行できるモジュールとアプリケーションが得られます。

アセンブルを成功させる上で最も大きな障害は、その工程が多種多様であることです。アセンブルするモジュールまたはアプリケーションによって、必要な実行時サービスの組み合わせが異なります。モジュールやアプリケーションのアセンブルには標準的な手順がないため、アセンブル処理を行うためには、正しくアセンブルされたモ

ジュールまたはアプリケーションがどのようなものか、ある程度理解している必要があります。ここでは、アセンブリ処理の理解に役立つ、J2EE モジュールや J2EE アプリケーションに関するいくつかの背景情報を提供します。

## J2EE アプリケーションはモジュール構造である

J2EE アプリケーションは、コンポーネントの集まりです。コンポーネントを組み合わせてモジュールが作成され、モジュールを組み合わせてアプリケーションが作成されます。コンポーネントを組み合わせてモジュールを作成し、モジュールを組み合わせてアプリケーションを作成するための J2EE の仕組みが配備記述子です。配備記述子は、基本的にリストです。モジュールの配備記述子はそのモジュールを構成するコンポーネントのリストで、アプリケーションの配備記述子はそのアプリケーション内のモジュールのリストです。

J2EE プラットフォームで配備記述子が使用される理由を理解するために、アプリケーションのソースコードがどのように使用されるのかを考えてみましょう。編集時、コンポーネントは開発環境内に多くのソースファイルとして存在します。これらのコンポーネントは、配備するか、J2EE 実行環境にインストールするまで実行できません。

アプリケーションを配備することにより、配備記述子によって識別されるソースファイルがコンパイルされ、アプリケーションサーバーによって管理されるディレクトリにコンパイル後のファイルがインストールされます。配備が終わったアプリケーションは、アプリケーションサーバーの環境で実行できます。言い換えれば、アプリケーションは、それを配備したときに作成されます。

つまり、配備記述子は、1つのモジュールまたはアプリケーションとしてまとめて配備されるファイル群を指定するための編集時の仕組みです。モジュールまたはアプリケーションを編集時にアセンブルするとき、実際にソースファイルが変更されることはありません。アセンブル段階では、モジュールまたはアプリケーションの内容を示す配備記述子を準備しているにすぎません。この配備記述子が配備処理の入力として使用されます。

配備記述子は XML ファイルです。このファイルでは、アプリケーションとそのアプリケーションを構成するモジュール群 (または、モジュールとそのモジュールを構成するコンポーネント群) を識別するために独自の XML タグが使用されます。コード例 1-1 は、アプリケーションの配備記述子の例です。

コード例 1–1　　J2EE アプリケーション配備記述子

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
      1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">
<application>
 <?xml version="1.0" encoding="UTF-8"?>
 <display-name>hello</display-name>
 <description>J2EE Application CatalogApp</description>
 <module>
  <ejb>CatalogData.jar</ejb>
  <alt-dd>CatalogData.xml</alt-dd>
 </module>
 <module>
  <web>
   <web-uri>CatalogWebModule.war</web-uri>
   <context-root>catalog</context-root>
  </web>
  <alt-dd>CatalogWebModule.xml</alt-dd>
 </module>
</application>
```

この例は、J2EE アプリケーション、CatalogApp の配備記述子です。このプログラムは、CatalogData と CatalogWebModule という 2 つのモジュールを含んでいます。各モジュールは、<module> タグで識別されます。

アプリケーションを構成する各モジュールは、そのモジュールを構成するコンポーネントのリストであるモジュールレベルの配備記述子を持っています (モジュールレベルの配備記述子の例については、コード例 1–2 とコード例 1–3 を参照してください)。アプリケーションを配備すると、アプリケーションサーバーがこの配備記述子を読み取り、2 つのモジュールレベルの配備記述子を読み取ります。モジュールレベルの配備記述子から個々の J2EE コンポーネントのソースファイルが識別されます。

## J2EE アプリケーションは J2EE 実行環境によってサポートされる

実行時、J2EE アプリケーションは配備先の J2EE アプリケーションサーバーから提供されるサービスをフル活用します。数多くのサービスの中でも特に重要なのは、コンテナ管理の持続性、コンテナ管理のトランザクション、コンテナ管理のセキュリティ検査です。

これらのサービスを利用するために、アプリケーション、またはアプリケーションを構成するモジュール群は、必要としているサービスをアプリケーションサーバーに伝えなければなりません。そのための仕組みが配備記述子です。たとえば、J2EE コンテ

ナ管理のトランザクションを考えてみます。このサービスを使用するためには、エンタープライズ JavaBean コンポーネントをプログラミングするときに、それぞれのエンタープライズ Bean のトランザクション属性プロパティを設定してアプリケーショントランザクション境界を定義します。この情報を実行環境に伝達するために、この値はエンタープライズ Bean が入っている EJB モジュールの配備記述子に含まれています。コード例 1-2 は、EJB モジュール、CatalogData の配備記述子です (コード例 1-1 を参照してください)。トランザクション属性値を持つタグは、配備記述子の最後にあります。

コード例 1–2　　EJB モジュール配備記述子

```
<?xml version="1.0" encoding="UTF-8"?>
<!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>
 <display-name>CatalogData</display-name>
 <enterprise-beans>
  <session>
   <display-name>CatalogManagerBean</display-name>
   <ejb-name>CatalogManagerBean</ejb-name>
   <home>CatalogBeans.CatalogManagerBeanHome</home>
   <remote>CatalogBeans.CatalogManagerBean</remote>
   <ejb-class>CatalogBeans.CatalogManagerBeanEJB</ejb-class>
   <session-type>Stateful</session-type>
   <transaction-type>Container</transaction-type>
   <ejb-local-ref>
    <ejb-ref-name>ejb/ItemBean</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>CatalogBeans.ItemBeanLocalHome</local-home>
    <local>CatalogBeans.ItemBeanLocal</local>
    <ejb-link>ItemBean</ejb-link>
   </ejb-local-ref>
   <ejb-local-ref>
    <ejb-ref-name>ejb/ItemDetailBean</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>CatalogBeans.ItemDetailBeanLocalHome</local-home>
    <local>CatalogBeans.ItemDetailBeanLocal</local>
    <ejb-link>ItemDetailBean</ejb-link>
   </ejb-local-ref>
  </session>
  <entity>
   <display-name>ItemBean</display-name>
   <ejb-name>ItemBean</ejb-name>
   <local-home>CatalogBeans.ItemBeanLocalHome</local-home>
   <local>CatalogBeans.ItemBeanLocal</local>
   <ejb-class>CatalogBeans.ItemBeanEJB</ejb-class>
```

コード例 1–2　　EJB モジュール配備記述子 ( 続き )

```
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <abstract-schema-name>ItemBean</abstract-schema-name>
    <cmp-field>
     <field-name>itemsku</field-name>
    </cmp-field>
    <cmp-field>
     <field-name>itemname</field-name>
    </cmp-field>
    <primkey-field>itemsku</primkey-field>
    <query>
     <query-method>
      <method-name>findAll</method-name>
      <method-params/>
     </query-method>
     <ejb-ql>SELECT Object (I) FROM ItemBean AS I</ejb-ql>
    </query>
   </entity>
   <entity>
    <display-name>ItemDetailBean</display-name>
    <ejb-name>ItemDetailBean</ejb-name>
    <local-home>CatalogBeans.ItemDetailBeanLocalHome</local-home>
    <local>CatalogBeans.ItemDetailBeanLocal</local>
    <ejb-class>CatalogBeans.ItemDetailBeanEJB</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <abstract-schema-name>ItemDetailBean</abstract-schema-name>
     <cmp-field>
      <field-name>itemsku</field-name>
     </cmp-field>
     <cmp-field>
      <field-name>description</field-name>
     </cmp-field>
     <primkey-field>itemsku</primkey-field>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
   <container-transaction>
    <description>This value was set as a default by Sun ONE Studio.</description>
    <method>
     <ejb-name>CatalogManagerBean</ejb-name>
     <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
    </container-transaction>
```

コード例 1–2　　EJB モジュール配備記述子 ( 続き )

```
    <container-transaction>
    <description>This value was set as a default by Sun ONE Studio.</description>
     <method>
      <ejb-name>ItemBean</ejb-name>
      <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
    <description>This value was set as a default by Sun ONE Studio.</description>
     <method>
     <ejb-name>ItemDetailBean</ejb-name>
     <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
   </container-transaction>
  </assembly-descriptor>
 </ejb-jar>
```

この EJB モジュールを含むアプリケーションが配備されて、実行されるとき、J2EE アプリケーションサーバーは配備記述子に指定されているトランザクション境界を認識し、適切なポイントでトランザクションのオープンとコミット (またはロールバック) を実行します。

## J2EE アプリケーションは分散型である

J2EE アプリケーションは、モジュール構造であり、実行時サービスを利用することに加え、分散型です。J2EE アプリケーションの各モジュールを別々のマシンに配備し、それぞれ独自のプロセスで実行することによって、分散型アプリケーションを作成できます。図 1-1 に、典型的な多層アプリケーションアーキテクチャを実装する、2 つのモジュールから構成される J2EE アプリケーションを示します。

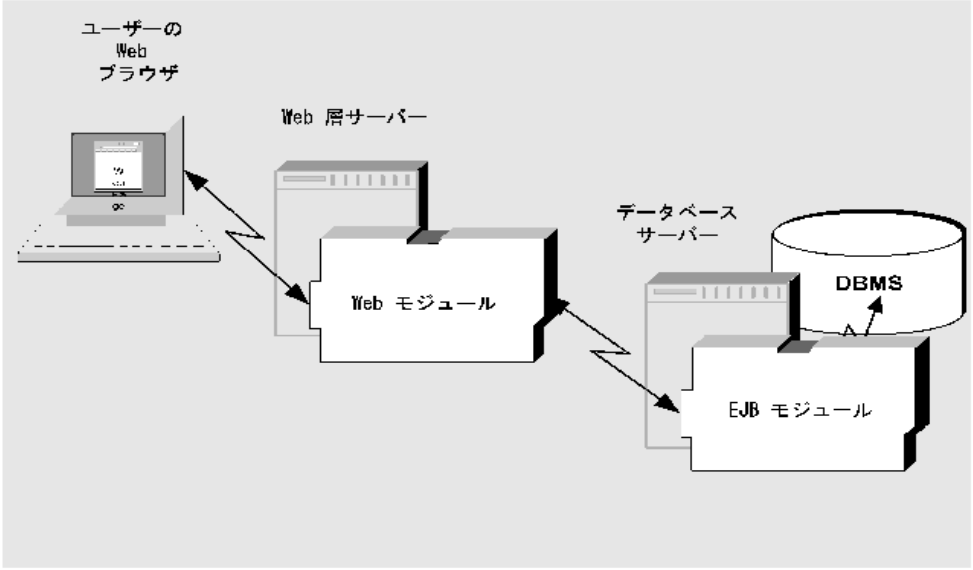


図 1-1　J2EE コンポーネントとモジュールを使用した多層アプリケーション

Web モジュールは、ユーザーとの HTTP 対話専用のマシンに配備されます。アプリケーションサーバーは HTTP 接続を使用してWeb モジュールで定義されている Web ページがユーザーのデスクトップマシンのブラウザに表示します。EJB モジュールはデータベース操作専用の別のマシンに配備されます。Web ブラウザと Web モジュールとの間の HTTP 接続、および 2 つのモジュール間の分散型対話は、J2EE アプリケーションサーバーによってサポートされます。

J2EE プラットフォームは、モジュール間の様々なモードの通信をサポートする多くのテクノロジを提供します。たとえば、次の通信モードがサポートされます。

- HTTP 接続上での Web ベースの通信 (エンドユーザーとアプリケーションとの間で多く使用される)
- Java RMI-IIOP を使用した同期メソッド呼び出し
- JMS を使用した非同期メッセージング (メッセージの宛先としてキューまたはトピックを指定できる)

このように、J2EE は多種のコンポーネントとのさまざまな対話をサポートします。たとえば、メッセージ駆動型エンタープライズ JavaBean は、モジュール間の非同期メッセージングをサポートします。

同じアプリケーション内のモジュール間で対話するときのモードや、その対話を実装するコンポーネントの種類は、アプリケーションの設計段階で選択します。しかし、アプリケーションをアセンブルする段階でも、設計時にどのような種類の対話が選択され、その対話がどのように実装されてたかを知る必要があります。つまり、アセンブル時に EJB 参照の設定 (Java RMI による対話を実装する場合) またはキューの設定 (JMS メッセージングによる対話を実装する場合) などの作業を行います。

また、J2EE プラットフォームは、アプリケーションが使用する外部リソース (データソースなど)と J2EE モジュールとの間の対話もサポートします。これらの対話をサポートするテクノロジには、次のものが含まれます。

- JDBC/JTS
- コンテナ管理の持続性

アプリケーションをアセンブルするときは、配備対象のアプリケーションが使用する外部リソースが正しく識別されていることを確認する必要もあります。配備記述子は、このような外部リソースを識別するための編集時の仕組みとしても使用されます。

コード例 1–3 に、CatalogWebModule という Web モジュールの配備記述子を示します。このモジュールは、EJB モジュール (コード例 1–2 の配備記述子で識別される EJB モジュール) とともにアセンブルされ、J2EE アプリケーションが作成されます。2 つのモジュール間での対話に使用されるテクノロジは、Java RMI です。Java RMI は、この配備記述子の最後の `<ejb-ref>` タグによって宣言されているリモート EJB 参照に依存します。

コード例 1–3 Web モジュール配備記述子

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
 <servlet>
  <servlet-name>AllItemsServlet</servlet-name>
  <servlet-class>AllItemsServlet</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>DetailServlet</servlet-name>
  <servlet-class>DetailServlet</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>AllItemsServlet</servlet-name>
```

コード例 1–3　　Web モジュール配備記述子 ( 続き )

```
   <url-pattern>/servlet/AllItemsServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
   <servlet-name>DetailServlet</servlet-name>
   <url-pattern>/servlet/DetailServlet</url-pattern>
  </servlet-mapping>
  <session-config>
   <session-timeout>
    30
   </session-timeout>
  </session-config>
  <welcome-file-list>
   <welcome-file>
    index.jsp
    </welcome-file>
    <welcome-file>
     index.html
    </welcome-file>
    <welcome-file>
     index.htm
    </welcome-file>
   </welcome-file-list>
   <ejb-ref>
    <ejb-ref-name>ejb/CatalogManagerBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>CatalogBeans.CatalogManagerBeanHome</home>
    <remote>CatalogBeans.CatalogManagerBean</remote>
    <ejb-link>CatalogManagerBean</ejb-link>
   </ejb-ref>
 </web-app>
```

## 「エクスプローラ」ウィンドウでのモジュールとアプリケーションの表示

多くの場合、J2EE アセンブル手順を説明するときには、先のコード例のような配備記述子ファイルの内容を参照しながら配備記述子用 XML のコーディング方法を説明します。Sun ONE Studio 4 開発環境では、コンポーネント、モジュール、アプリケーションのイメージ (ノード) が表示されます。つまり、開発者は配備記述子ファイルを直接操作するのではなく、コンポーネント、モジュール、アプリケーションを表すエクスプローラウィンドウノードを操作します。

アプリケーションノードには、そのモジュールを表すサブノードがあります。また、モジュールノードには、そのコンポーネントを表すサブノードがあります。アセンブル時には、作成中のモジュールまたはアプリケーションのノードが「エクスプローラ」ウィンドウに表示されます。「エクスプローラ」ウィンドウに表示されるイメージを操作すると、IDE によって対応する配備記述子が作成されます。

各ノードには、そのノードが表すコンポーネント、モジュール、またはアプリケーションを設定するためのプロパティシートがあります。プロパティの多くは、配備記述子タグに対応します (プロパティの数は、配備記述子タグの数より多くなっています) 。コンポーネント、モジュール、またはアプリケーションのプロパティを設定すると、IDE によってその配備記述子にタグが追加されます。このタグによって、モジュール間の分散型対話を可能にするサービスなど、アプリケーションサーバーに要求するサービスが識別されます。

この後、IDE によるモジュールおよびアプリケーションの表示方法を紹介します。

## Web モジュール

Web モジュールのノードとサブノードは、モジュール内の個々のファイルを表します。Web モジュールには標準的なディレクトリ構造 (詳細については、『Web コンポーネントのプログラミング』 を参照してください) があり、「エクスプローラ」ウィンドウにこの構造が示されます。図 1-2 は、エクスプローラに表示された Web モジュールを示しています。

Web モジュールの最上位ノードは、Web モジュールの最上位ディレクトリを表します。IDE がディレクトリを Web モジュールとして認識するには、そのディレクトリを「エクスプローラ」ウィンドウのファイルシステムとしてマウントする必要があります。Web モジュールディレクトリが別のファイルシステムのサブディレクトリとして表示された場合、IDE はこれを Web モジュールとして認識しません。この詳細は、『Web コンポーネントのプログラミング』でも説明されています。

最上位ノードには、`WEB-INF` ディレクトリを示すサブノードがあります。`WEB-INF` ディレクトリには、JAR ファイル形式の Web コンポーネントに使用される `lib` サブディレクトリを示すサブノードと、.java ファイル形式のすべての Web コンポーネントに使用されるクラスサブディレクトリを示すサブノードがあります。`WEB-INF` ノードには、モジュールの配備記述子ファイルを表す `web.xml`サブノードもあります。これは、Web モジュールの標準的なディレクトリ構造です。

この例の Web モジュールには、JSP ページ myNewJSP、HTML ページ index.html、およびタグライブラリ myTagLib を示すノードもあります。これらのノードは、Web コンポーネントプロバイダによって追加されたコンポーネントとリソースを表します。これらのノードに加え、クラスディレクトリには、サーブレットクラス myNewServlet を示すノードが含まれます。これは、コンポーネントプロバイダによって追加されたもう 1 つのリソースです。


図 1-2　Web モジュールノードとそのサブノード

この Web モジュールの表示は、特定のディレクトリとその内容に対応します。配備記述子 (`web.xml` ファイル) は、ソースコードに含まれます。

## EJB モジュール

EJB モジュールの表示は、Web モジュールとは異なります。EJB モジュールの最上位ノードは、特定のディレクトリとその内容を表しません。代わりに、EJB モジュールノードはモジュールの配備記述子を表します。これは、1 つのディレクトリ、または異なる複数のファイルシステム内にある多数のディレクトリに存在するエンタープライズ Bean のリストとして機能します。配備記述子は、コンポーネントのソースコードのある場所を指定します。

EJB モジュールを 「論理」ノードによって表すと、異なるディレクトリに存在する複数のエンタープライズ Bean を組み合わせて 1 つの EJB モジュールを作成できます。その場合は、配備記述子の構成情報がソースコードとは別に保持されます。EJB モジュールを配備すると、配備記述子ファイルが生成されます。その配備記述子に指定されているコンポーネントのソースファイルは、ファイルシステム内のどこにあってもコンパイルされ、EJB JARファイルが生成されます。

図 1-3 は、「エクスプローラ」ウィンドウでの EJB モジュールの表示例です。このモジュールには、モジュールに含まれている 3 つのエンタープライズ Bean を示すサブノードがあります。これらのエンタープライズ Bean はそれぞれ異なるディレクトリに存在する可能性があります。さらに、1 つのエンタープライズ Bean が、ファイルシステム内の異なる場所に複数存在している可能性もあります。たとえば、エンタープライズ Bean のインタフェースのソースコードと、そのインタフェースを実装するクラスが別のディレクトリに存在する場合もあります。

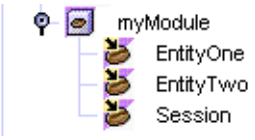


図 1-3　EJB モジュールノードとそのサブノード

## J2EE アプリケーション

J2EE アプリケーションも論理ノードによって表されます。EJB モジュールノードと同様、J2EE アプリケーションの最上位ノードも単一のディレクトリまたはファイルシステムを表しません。このノードはアプリケーションレベルの配備記述子を表し、アプリケーションを構成するモジュールのリストとして機能します。これらのモジュールのソースコードは、複数のディレクトリまたはファイルシステムに存在する可能性があります。

IDE は、アプリケーションレベルの配備記述子をソースコードとは別に保持し、同じソースコードを複数の J2EE アプリケーションで使用できるようにします。アプリケーションを配備する (またはアプリケーションの EAR ファイルを生成する) 場合にだけ、IDE はすべてのソースファイルをコピーして EAR ファイル内の配備記述子と関連付けます。

図 1-4 は、「エクスプローラ」ウィンドウでの J2EE アプリケーションの表示例です。図 1-2 と 図 1-3 のモジュールがアプリケーションに追加され、アプリケーションノードのサブノードによって表されています。


図 1-4　J2EE アプリケーションノードとそのサブノード

## プロパティシート

J2EE モジュールと J2EE アプリケーションを表すすべてのノードに、プロパティシートがあります。プロパティシートのプロパティを使用し、モジュールに必要なアプリケーションサーバーのサービスを記述できます。これらのプロパティはモジュールまたはアプリケーションの配備記述子内に現れるタグに対応します。そのため、プロパティの値を設定することは、配備記述子内で使用する情報を提供することになります。つまり、プロパティシートを使用すればプロパティが設定できるので、テキストエディタを使用して XML 配備記述子を編集したり書式を設定したりする必要がありません (一部のプロパティは配備記述子のタグに対応しません) 。

- Web モジュールの場合、配備記述子はファイルとして存在し、このファイル (`web.xml`) は「エクスプローラ」ウィンドウに表示されます。その表示例を 図 1-2 に示します。つまり、配備記述子で指定する値は、ソースファイルと関連付けられることになります。
- EJB モジュールや J2EE アプリケーションの場合、モジュールを配備する (EAR ファイルが必要) か 、EAR ファイルを生成するまで配備記述子ファイルは生成されません。これらの処理によって配備記述子ファイルが生成され、必要なソースファイルのコピーとともに保存されます。

多くのプロパティには、正しい値を選択するのに役立つプロパティエディタがあります。プロパティシートが開いているとき、プロパティをクリックして、省略符号ボタン (...) をクリックすると、そのプロパティのエディタが開きます。プロパティエディタの使用手順はプロパティごとに異なります。IDEノードを閲覧するものもあれば、別のレベルのダイアログを開くものもあります。より複雑なプロパティエディタには、オンラインヘルプがあります。

各プロパティシートにはいくつかのタブがあります。「プロパティ」という名前のタブには、J2EE アプリケーションによって定義されている標準プロパティの一覧が表示されます。その他のタブには、アプリケーションサーバー製品の名前が付いています。これらのタブは、J2EE アプリケーションでは定義されていない、個々のアプリケーションサーバー製品に必要な追加情報を収集するためのものです。これらをプロパティシートのサーバー固有タブと呼びます。モジュールまたはアプリケーションのアセンブル時には、標準プロパティと使用しているサーバー製品のサーバー固有プロパティの両方を設定する必要があります。

## 配備の基本

前述のように、配備とは、1 つの J2EE アプリケーションを構成するソースファイルをコンパイルし、コンパイル後のファイルを J2EE アプリケーションサーバーによって管理されるディレクトリにインストールすることです。この処理は、アプリケーションサーバーや、そのサーバーとともに配布されている配備ツールや管理ツールなどのソフトウェアによって実行されます。

ソースファイルのコーディング、実行、テスト、変更、および再実行という反復開発をサポートするため、Sun ONE Studio 4 IDE では IDE 内からの配備と実行が可能になっています。配備はアプリケーションサーバーソフトウェアによって行われ、実行はアプリケーションサーバーの環境内で行われます。配備と実行は IDE 内から管理できます。J2EE ソースコードを作成し、J2EE アプリケーションをアセンブルして、IDE コマンドを使い、アプリケーションの配備と実行を行うことができます。アプリケーションの実行とテストの後、ソースコードを変更し、アプリケーションを再配備して、新しいバージョンを実行できます。この一連の処理を必要なだけ繰り返すことができます。

アセンブルした後のアプリケーションを配備する操作は非常に簡単です。アプリケーションノードを右クリックし、配備コマンドを選択するだけです。しかし、そのためには、IDE が特定のアプリケーションサーバーと連携するように設定されている必要があります。その手順を要約すると、次のようになります (詳細な手順については、『Sun ONE Studio 4, Enterprise Edition インストールガイド』を参照してください) 。

1. アプリケーションサーバーをインストールします。

2. IDE と アプリケーションサーバーとの間の通信を可能にする モジュールをインストールします。これらのモジュールは「サーバープラグイン」と呼ばれ、各プラグインモジュールが 1 つのアプリケーションサーバー製品をサポートします。広く使われているアプリケーションサーバー製品には多くのプラグインがあります (プラグインの機能については、付録 Aを参照してください) 。
3. プラグインを使用し、IDE とアプリケーションサーバー間の通信を開きます。これにより、サーバーインスタンスを表す「エクスプローラ」ウィンドウが作成されます。
4. サーバーインスタンスノードの 1 つを参照することにより、アプリケーションの配備先アプリケーションサーバーを指定します。
5. アプリケーションノードを右クリックし、配備コマンドを実行します。この操作により、配備ソフトウェアによってアプリケーションノードによって表されている配備記述子が読み取られ、配備記述子に指定されているソースファイルが処理されます。

## 実行の基本

アプリケーションの配備が終われば、そのアプリケーションを実行できます。IDE は、配備されたアプリケーションとアプリケーションノードを関連付けます。関連付けが終われば、実行コマンドを出すことができます。実行コマンドを出すと、IDE がアプリケーションサーバーに対して配備済みアプリケーションを実行するよう命令します。

J2EE アプリケーションノードを含め、多くの IDE ノードに「実行」コマンドがあります。ノードを右クリックし、「実行」を選択すると、IDE によって必要に応じてアプリケーションが配備され、実行されます。「実行」コマンドの結果は、アプリケーションの内容によって異なります。たとえば、アプリケーションに Web モジュールが含まれている場合、「実行」コマンドを実行すると Web ブラウザが起動してアプリケーションの URL が開きます。

配備済みのアプリケーションはアプリケーションサーバーの環境で (IDE を使用せずに) 実行することもできます。たとえば、Web モジュールを含むアプリケーションを実行するには、Web ブラウザを起動し、アプリケーションの URL を開きます。

# このマニュアルの使用法

Java Community Process は、Sun Microsystems, Inc. のサポートを得て J2EE コンポーネントを使用した分散型エンタープライズアプリケーション設計のための標準を進化させました。ix ページの「お読みになる前に」 に示されている J2EE のマニュアルは、アプリケーションの設計やアーキテクチャに関するこれらの標準を取り上げています。

このマニュアルは、Sun ONE Studio 4 IDE を使用したこれらのアーキテクチャの実装方法を説明するものです。IDE を使用し、コンポーネントの組み合わせによって J2EE モジュールを作成する方法を説明します。このとき、すべてのコンポーネントがアプリケーションの設計どおりに連携するようにします。また、J2EE モジュールの組み合わせによって J2EE アプリケーションを作成する方法を説明します。このとき、モジュール間の分散型対話がアプリケーションの設計どおりに機能するようにします。

このマニュアルでは、いくつかの例やシナリオを提示することによってこのテーマに取り組みます。それぞれのシナリオでは、コンポーネントまたはモジュールの現実的な組み合わせによってモジュールやアプリケーションを作成する方法を示します。シナリオで扱っている業務上の問題は現実的なものですが、このマニュアルは J2EE アプリケーションの設計ガイドではありません。J2EE テクノロジの適切な使用法についてのヒントは示しますが、詳しくは説明しません。

シナリオの本来の目的は、コンポーネントとモジュール間における特定の対話のプログラミング方法を示すことです。アプリケーションの設計が決まった後、アプリケーション内のコンポーネントとモジュール間に必要な接続をプログラミングするときに、このマニュアルのシナリオを役立てることができます。

1 つのシナリオで必要なすべてのものを見つけることはできないでしょう。各シナリオで扱っているのは J2EE の 1、2 種類の対話にすぎません。J2EE アプリケーションには何十種類または何百種類ものコンポーネントや関係が含まれる可能性があります。一方、接続に関しては、このマニュアルですべての種類について 1 つずつ例を示しています。

Web モジュールと EJB モジュールを組み合わせた一般的な J2EE アプリケーションをプログラミングする場合は、Web モジュールのアセンブルを扱っている 第 2 章 、EJB モジュールのアセンブルを扱っている第 3 章、Web モジュールと EJB モジュールのア

センブルによる J2EE アプリケーションの作成を扱っている第 4 章が参考になります。トランザクションの設定方法については第 7 章、セキュリティについては第 8 章を参照します。

このマニュアルは、Sun ONE Studio 4 IDE を使用して分散型エンタープライズ環境を開発するためのガイドです。J2EE モジュールの開発方法や、さまざまな対話を実現するモジュールのプログラミング方法を示します。さらに、セキュリティ検査やトランザクション管理などのエンタープライズサービスを J2EE プラットフォームに要求する方法を示します。

# 第2章

# シナリオ： Web モジュール

図 2-1 は、Web モジュールを示しています。Web コンポーネントは多くの組み合わせが可能ですが、このモジュールは 2 つのサーブレットと 1 つの静的な HTML ページを含んでいます。このモジュールは、J2EE アプリケーションの 2 つのモジュール間に特有の対話を行います。Web モジュールは J2EE アプリケーションのフロントエンドを提供するため、HTTP 接続を介してエンドユーザーと対話したり (図の矢印 1) 、EJB モジュールによって提供される Web サービスと対話したり (図の矢印 3) する必要があります。また、モジュール内のコンポーネント間での対話 (図の矢印 2) もあります。

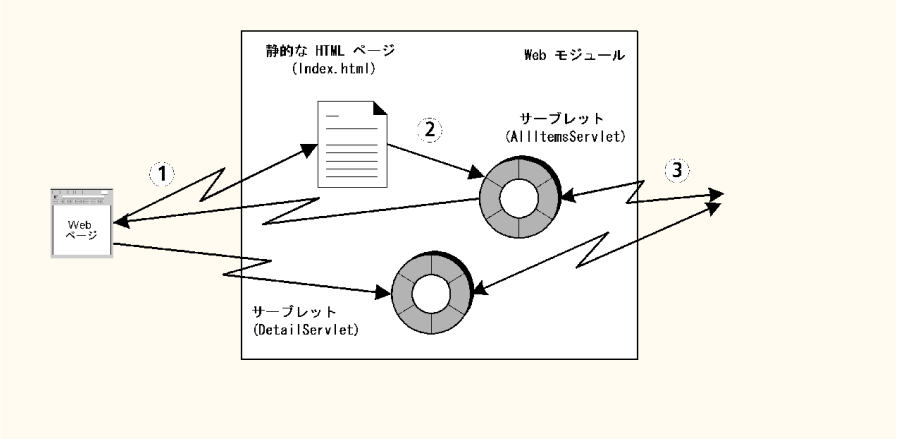


図 2-1　J2EE アプリケーションの Web モジュール

## モジュール内での対話

このシナリオでは、図 2-1 に示す 3 種類の対話に関与する Web モジュールを取り上げます。このモジュールは、ショッピング Web サイトをサポートする J2EE アプリケーションの一部で、Web サイトアプリケーション内で J2EE アプリケーションのフロントエンド機能を提供するために使用されます。これは、J2EE アプリケーションにおける Web モジュールの典型的な役割です。

利用者の側から見れば、アプリケーションは一連の Web ページです。オンラインショッピングの利用者は、Web ブラウザを使ってアプリケーションのコンテキストルートにマップされる既知の URL への HTTP 接続を開いたり、Web ページ上に表示されるテキストフィールド、ボタンなど、アプリケーションに指令を送るコントロールを使用して入力を行います。開発者の側から見れば、アプリケーションは要求を受け取って処理するコンポーネント群です。

このシナリオは、2 種類の要求を処理する単純な Web モジュールを使用し、ユーザー、Web モジュール、EJB モジュールの間で必要な対話を Web コンポーネントがどのようにして提供するかを示します。具体的に、オンラインショッピングカタログの表示には、次の対話を使用します。

1. オンラインショッピングの利用者が Web ブラウザを起動し、アプリケーションのルート URL を開くことによって、アプリケーションへの接続を開きます。この操作により、アプリケーションの開始ページが開きます。現実のショッピングサイトの開始ページには、商品をカテゴリ別で表示するための要求や、キーワード検索の要求、現在の顧客サービスに関する情報の要求など、多くのオプションが表示されます。単純な例では、カタログ全体を表示するためのリンクが 1 つあるだけです。

2. 利用者は、カタログ表示へのリンクをクリックします。この要求は、要求の中で 指定された AllItemsServlet という名前がサーブレットによって処理されます。AllItemsServlet は、EJB モジュールのビジネスメソッド、getAllItems を呼び出すことによって要求を処理します。このメソッドが要求を返します。

3. AllItemsServlet は、getAllItems が返すデータを書式化して HTML 出力ストリームに追加します。エンドユーザーの Web ブラウザが HTML 出力を書式化してカテゴリ表示します。この出力には、各商品の詳細情報へのリンクが含まれます。

4. エンドユーザー (利用者) は表示されたカタログを閲覧し、詳細情報へのリンクの 1 つをクリックします。要求はもう 1 つのサーブレット、DetailServlet によって処理されます。DetailServlet は、EJB モジュールのもう 1 つのビジネスメソッド、getOneItemDetail を呼び出し、商品の詳細情報の要求を処理します。

5. DetailServlet は、EJB モジュールが返すデータを HTML 出力ストリームに追加します。この出力ストリームが別の Web ページとしてエンドユーザーに表示されます。

このシナリオには 2 つの要求が可能で、それぞれが別の Web コンポーネントによって処理されます。この例にはサーブレットを使用したため、AllItemsServlet と ItemDetailServlet の 2 つのサーブレットがあります。

この 2 つのサーブレットによって書き出される HTML 出力は、テキストだけを含む単純なものです。これらの例では、Web コンポーネントでリモートメソッド呼び出しを使用して、EJB モジュールからデータを取得し、取得したデータを HTML 出力ストリームに書き込めることが分かります。経験のある Web デザイナや Web プログラマは、同じ種類の操作を利用してさらに複雑な出力を作成できます。

Web モジュールが Java RMI を使用して EJB モジュールと行う対話の種類は、EJB モジュールの設計によって決定します。詳細については、42 ページの「モジュール内での対話」 を参照してください。

Web コンポーネントの作成、Web コンポーネントでのエンタープライズビジネスロジックの記述などの作業手順については、『Web コンポーネントのプログラミング』を参照してください。

# モジュールのプログラミング

表 2-1 は、図 2-1 の EJB モジュールを作成するために必要なプログラミング作業をまとめたものです。.

表 2-1　このシナリオに必要なプログラミング

| アプリケーション要素 | 必要なプログラミング作業 |
|---|---|
| アプリケーションサーバー | なし |
| Web モジュール | Web モジュールの作成<br>開始ページ、`index.html` の作成します。このページには、AllItemsServlet を実行する HTML リンクを含みます<br>AllItemsServlet と ItemDetailServlet の 2 つのサーブレットを作成します。<br>HTTP 要求を処理し、HTTP 応答を生成する processRequest メソッドをコーディングします。この processRequest メソッドは次の作業を行います<br>1. EJB モジュールのビジネスメソッドを呼び出すために JNDI ルックアップを実行します<br>2. モジュールから取得したデータをサーブレットの応答に書き込みます<br>AllItemsServlet が出力する HTML ページには、ItemDetailServlet への HTML リンクが含まれます<br>各サーブレットのサーブレット名の設定します |
| J2EE アプリケーション | J2EE アプリケーションへの Web モジュールの追加方法については、第 4 章を参照してください |

この後、これらのプログラミングの作業手順を説明していきます。メソッドシグニチャを使用し、各対話の入力と出力を示します。手順は、これらの入力や出力をほかのコンポーネントやほかのモジュールに接続する方法を示すものです。Web モジュールの作成手順、Web コンポーネントをモジュールに追加する手順は説明しません。それらの作業手順については、オンラインヘルプまたは 『Web コンポーネントのプログラミング』 を参照してください。

## 開始ページの作成

カタログ表示 Web サイトは、ユーザーが開始ページでサイトとの対話を始めるよう設計されています。Web サイト特有の機能の 1 つである開始ページは、ユーザーがサイトの内容を確認し、利用可能なオプションを知る入り口の役目を果たします。開始ページを見たユーザーは、自分が使用したい機能への扉 (リンク) をクリックします。

ユーザーの最初の要求は、アプリケーションのルート URL に入ることです。開始ページが用意されている場合、サーバーはそれを表示します。

### HTML ページの作成

エクスプローラで、開始ページの HTML ファイルは Web モジュールの`WEB_INF` ノードと同じレベルになければなりません。HTML ファイルを正しいレベルに作成するには、次のようにします。

1. 緑色の `WEB-INF` ノードを含むファイルシステムのノードを右クリックし、コンテキストメニューの「新規」>「JSP & サーブレット 」>「HTMLファイル」の順に選択します。「index」という名前を入力し、「完了」をクリックします。

   エクスプローラに HTML ファイルノードが作成され、ソースエディタに新しいファイルが開きます。

2. 開始ページの HTML コードを入力します。

   コード例 2-1 に、このシナリオで使用する簡単な開始ページの HTML コードを示します。

コード例 2-1　カタログ表示モジュールの開始ページ

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Online Catalog</title>
  </head>
  <body>
<h2>
Inventory List Application
</h2>

<p>
<a href="allItems">Display the Catalog
</a>

</body>
</html>
```

この開始ページは、カタログ全体を表示する 1 つのオプションだけをユーザーに提示します。このオプションは、「Display the Catalog」というテキストリンクとして表示されます。このリンクは、モジュール内のサーブレットの 1 つ、AllItemsServlet を URL で指定します。ユーザーがこのリンクをクリックすると、ブラウザが別の要求を Web モジュールに送信します。この要求は、AllItemsServlet の doGet メソッドを実行することによって処理され、ユーザーに表示される次のページは、AllItemsServlet によって出力されるページです (AllItemsServlet によって出力されるページを見るには、コード例 2-2 を参照してください。

現実の Web サイトの開始ページにはさまざまな機能へのリンクが含まれますが、どのような機能もこの例に示す原則に従います。各要求は Web モジュール内のコンポーネントによって処理されます。そのコンポーネントは、ユーザーに表示する次のページを書き出すことによって応答します。

この場合、リンクはサーブレット名を指定します。デフォルトの動作は、サーブレットの doGet メソッドの実行です。doPost など、サーブレットのほかのメソッドをリンクに指定すれば、そのメソッドを実行できます。サーブレットのメソッドの詳細については、『Web コンポーネントのプログラミング』 を参照してください。

## 開始ページとして使用するページの指定

IDE で Web モジュールを作成するとき、「開始ファイル」プロパティには開始ファイルのいくつかのデフォルト名がリスト表示されます。図 2-2 に、デフォルトの開始ファイル名が表示された 「開始ファイル」プロパティエディタを示します。ユーザー

がアプリケーションのルート URL にアクセスするとき、アプリケーションサーバーがモジュールディレクトリから「開始ファイル」プロパティに指定されている名前のファイルを検索します。最初に見つかったファイルが開始ページとして表示されます。

モジュールの開始ファイルを作成する最も簡単な方法は、これらのデフォルト名を持つファイルを作成し、それをモジュールに追加するというものです。たとえば、このシナリオでは `index.html`. という名前のファイルを作成しています。

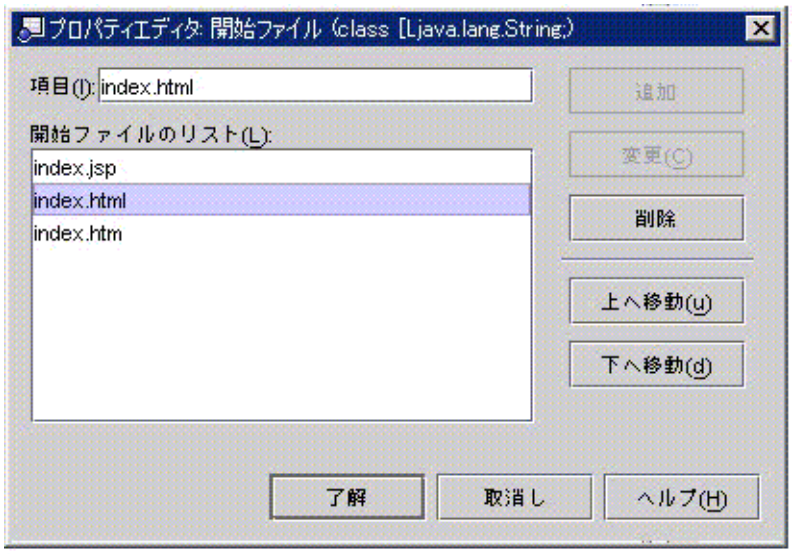


図 2-2　「開始ファイル」プロパティエディタ

モジュールの開始ページに別の名前を持つファイルを使用する場合は、「開始ファイル」プロパティエディタを開き、使用する名前を追加します。このプロパティエディタには、ほかにも、ファイルの順序を変更したり、リストからデフォルト名を削除したり、追加した名前の下にデフォルト名を移動したりするボタンが用意されています。

## サーブレットの `processRequest()` メソッドのプログラミング

ここでは 2 つの作業を行います。1つはメソッド本体のコーディング、もう 1 つはメソッド本体で使用される EJB 参照の参照宣言の設定です。最初に、メソッド本体のコードについて説明します。

この例のサーブレットは、IDE のサーブレットテンプレートを使用して作成されていますこのテンプレートによって作成されたサーブレットは HttpServlets で、processRequest という名前のメソッドを含んでいます。doGet メソッドと doPost メソッドはどちらも processRequest を呼び出すため、要求を実際に処理するコードが processRequest メソッドに追加されています。

## メソッド本体

コード例 2-2 は、AllItemsServlet の processRequest メソッドの実装です。このメソッドは、開始ページに表示される「Display the Catalog」リンクをユーザーがクリックすると実行されます (ブラウザによって生成される要求には、メソッドではなく、AllItemsServlet が URL で指定されています。そのため、アプリケーションサーバーはデフォルトの動作を実行します。つまり、サーブレットの doGet メソッドを実行し、このメソッドが processRequest を呼び出します。

このメソッドの実装は、サーブレットがカタログデータを取得してユーザーに表示する方法を示します。この一連の動作は、次の 3 段階に分けることができます。

1. サーブレットが JNDI ルックアップを使用し、アプリケーションの別のモジュールにあるセッションエンタープライズ Bean へのリモート参照を取得する。
2. サーブレットがセッション Bean のビジネスメソッド、getAllItems を呼び出す (CatalogData モジュールの詳細については、第 3 章を参照してください)。
3. サーブレットは、リモートメソッド呼び出しによって返されるデータを HTML 出力ストリームに書き込みます。このストリームがユーザーのブラウザウィンドウに返されます。

コード例 2–2　AllItemsServlet の processRequest メソッド

```
protected void processRequest(HttpServletRequest request,
 HttpServletResponse response)
throws ServletException, java.io.IOException {
        response.setContentType("text/html");
        java.io.PrintWriter out = response.getWriter();
        /* output your page here */
        out.println("<html>");
        out.println("<head>");
        out.println("<title>AllItemsServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h2>The Inventory List</h2>");
```

コード例 2–2　AllItemsServlet の processRequest メソッド ( 続き )

```
        out.println("<table>");
        out.println("<tr>");
        out.println("<td>Item");
        out.println("<td>Item SKU");
        out.println("<td>Detail");

        CatalogBeans.CatalogManagerBeanHome catHome;
        CatalogBeans.CatalogManagerBean catRemote;

        try {
            InitialContext ic = new InitialContext();
           Object objref  = ic.lookup("java:comp/env/ejb/CatalogManagerBean");
            catHome = (CatalogBeans.CatalogManagerBeanHome) objref;
            catRemote = catHome.create();

            java.util.Vector allItems = catRemote.getAllItems();

            Iterator i = allItems.iterator();
            while (i.hasNext()) {

              CatalogBeans.iDetail itemDetail = (CatalogBeans.iDetail)i.next();
                out.println("<tr>" +
                "<td>" +
                itemDetail.getItemname() +

                "<td>" +
                itemDetail.getItemsku() +

                "<td>" +
                "<a href=\"" + response.encodeURL("DetailServlet?sku=" +
itemDetail.getItemsku()) +
                "\"> " +
                "Get Item Detail" +
                "</a>");
            }
        }
        catch (javax.naming.NamingException nE) {
            System.out.println("Naming Exception on Lookup" + nE.toString());
        }
        catch (javax.ejb.CreateException cE) {
            System.out.println("CreateException" + cE.toString());
        }
        catch (java.rmi.RemoteException rE) {
            System.out.println("RemoteException" + rE.toString());
        }
        catch (Exception e) {
            System.out.println(e.toString());
```

コード例 2–2　AllItemsServlet の processRequest メソッド ( 続き )

```
        }

        out.println("</table>");

        out.println("</body>");
        out.println("</html>");
        out.close();
    }
```

lookup 文は「CatalogManagerBean」を指定していますがこの文字列は実際には参照先のエンタープライズ Bean の名前ではなく、参照の名前を表します。このように、エンタープライズ Bean の名前を参照名として使用して、どの Bean を意図しているか分かりやすくすることがよくあります。エンタープライズ Bean は、次のステップで具体的に指定します。

## EJB 参照の参照宣言

AllItemsServlet のように、Web コンポーネントから EJB モジュール内のエンタープライズ Bean のメソッドを呼び出すときは EJB 参照を利用します。EJB 参照には、次の 2 つの部分があります。

- JNDI ルックアップコード
  JNDI 命名機能を使用し、名前付きエンタープライズ Bean へのリモート参照を取得します。
- 参照の宣言
  実行時環境でルックアップコードの参照先 Bean を識別するのに使用されます。

ルックアップコードは、processRequest メソッドの中にあります (コード例 2–2 参照)。ルックアップコードを使用するには、参照宣言を設定する必要があります。参照宣言は、lookup 文で使用されている参照名を、Web モジュールに配備したい実際のエンタープライズ Bean にマップします。Web モジュールの EJB 参照宣言を設定するには、次のようにします。

1. `web.xml` ノードを右クリックし、コンテキストメニューの「プロパティ」>「参照」タブ >「EJB 参照」> 省略符号ボタン (...) の順にクリックします。

   「EJB 参照」プロパティエディタが開きます。

2. 「追加」ボタンをクリックします。

   参照宣言の設定に使用するダイアログが表示されます。

3. 参照宣言を使用するためには、lookup 文で使用する参照名、メソッド呼び出しで使用するホームインターフェイスとリモートインターフェイスを指定する必要があります。

   図 2-3 に、値がフィールドに指定された追加ダイアログを示します。参照が実行時に機能し、JNDI ルックアップがエンタープライズ Bean へのリモート参照を返すためには、必ず参照を同じアプリケーションの特定のエンタープライズ Bean にリンクさせます。この作業は、アプリケーションの配備と実行の前に完了しておく必要がありますが、今の段階では完了させる必要はありません。

   開発のこの時点では参照をリンクさせないでおき、Web モジュールをアセンブルして J2EE アプリケーションを作成した後にリンクさせることもできます。状況によっては、開発のこの段階で参照を解決する場合もあります。

   a. 使用中の開発環境に参照先のエンタープライズ Bean がない場合は、参照をリンクできません。インタフェースの名前を入力し、「了解」をクリックします。JNDI ルックアップコードをコンパイルするためには、使用中の開発環境にインタフェースをコピーしておく必要があります。

      図 2-3 に、リンクしていない参照を設定しているプロパティエディタを示します。「ホームインターフェース」と「リモートインターフェース」は設定されていますが、「参照される EJB 名」フィールドに何も表示されていません。参照は、後で、アプリケーションプロパティシート上でリンクされます。

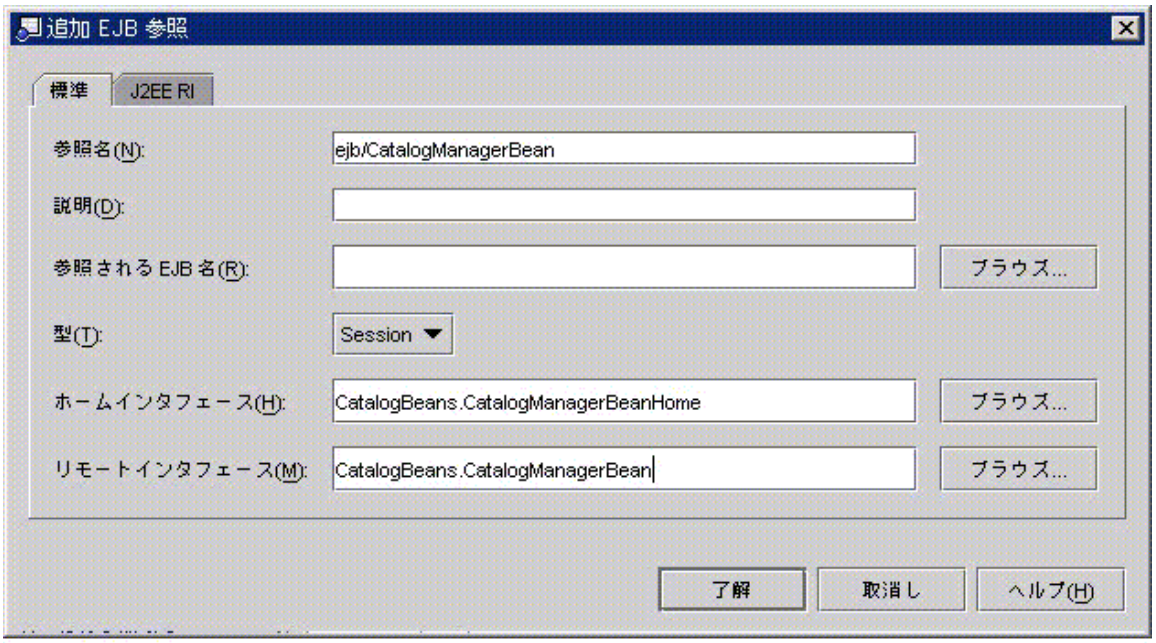


図 2-3　参照がリンクされていない。「EJB 参照」プロパティエディタ

b. 使用中の開発環境に参照先のエンタープライズ Bean がある場合は、この時点で参照をリンクできます。「参照される EJB 名」フィールドの隣の「ブラウズ」ボタンをクリックします。表示されるダイアログを使用して、エンタープライズ Bean を選択し、「了解」をクリックします。図 2-3 の ejb/CatalogManagerBean という名前の参照は、CatalogManagerBean にリンクされています。

参照先のエンタープライズ Bean がある場合でも、この時点で参照をリンクしないでおくことがあります。リンクしない理由は複数挙げられます。たとえば、作成中の Web コンポーネントが複数のアプリケーションで使用される可能性がある場合は、モジュールのプロパティシートで参照をリンクしないでおくといいでしょう。後に同じモジュールを使用する開発者が、同じインタフェースを実装したほかのエンタープライズ Bean にリンクし直すことも考えられます。

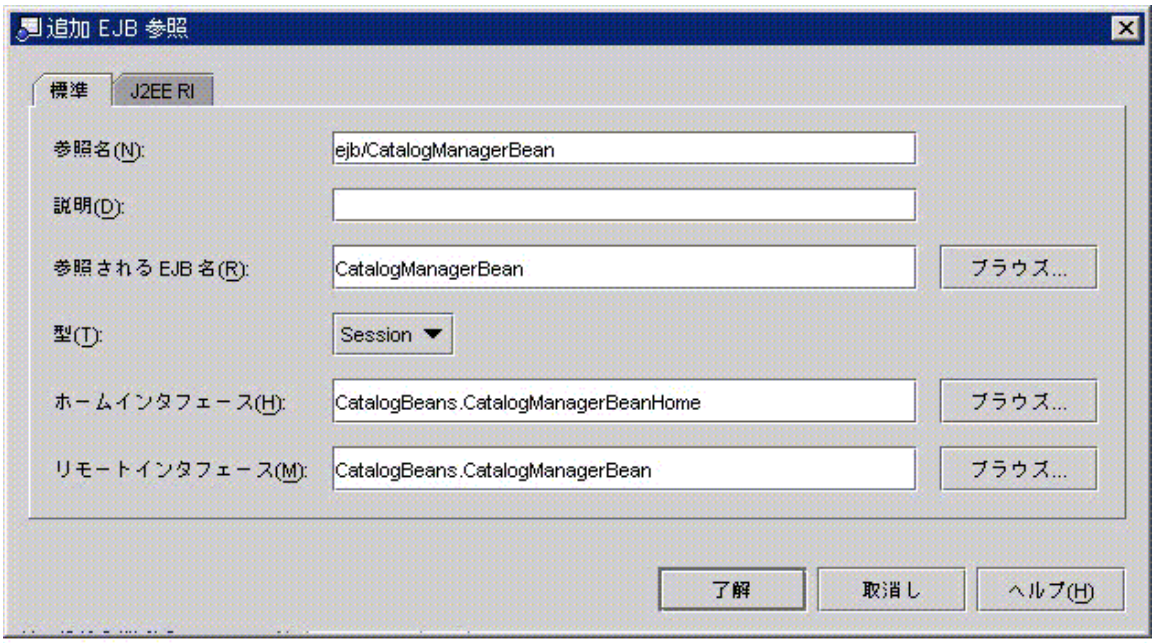


図 2-4　参照がリンクされている「EJB 参照」プロパティエディタ

このシナリオでは、図 2-3 のように、この時点で参照をリンクせず、J2EE アプリケーションの作成後にアプリケーションノードのプロパティシートでリンクします。70 ページの「EJB 参照のリンク」を参照してください。

## URL からサーブレットへのマッピング

開始ページで設定したリンクは、(利用者がリンクをクリックした時に) 実行されるビジネスロジックを次の URL で指定しました。

```
<a href="allItems">
```

このリンクが適切に機能するためには、指定した URL が、実行するサーブレットにマップされていなければなりません。その仕組みを理解するには、URL が Web リソースにどのようにマップされるのかを知る必要があります。

## サーブレットのマッピングについて

サーバーに配備されたアプリケーションでは、Web モジュールリソースには URL パスの後ろにそのリソースの名前を付加した URL が割り当てられます。J2EE リファレンス実装の場合、URL パスは次の汎用形式を持っています。

```
http://hostnmame:port/web_context/URL_pattern
```

このパスの要素は、次のように決定されます。

- `hostnmame` はアプリケーションサーバーが実行されているコンピュータの名前、`port` はそのサーバーインスタンスの HTTP 要求のために指定されたポートです。通常、ポート番号はアプリケーションサーバーのインストール時に割り当てられます。Sun ONE Studio 4 によってインストールされる J2EE RIの場合、HTTP ポート番号は 8000 です。
- *web_context* は、(インクルードされた Web モジュールのプロパティシート上で) モジュールを J2EE アプリケーションに追加した後で、モジュールのプロパティとして指定する文字列です。この文字列は、モジュール内のすべての Web リソースを修飾します。
- *URL_pattern* (URL パターン) は、特定のサーブレットまたは JSP ページを識別する文字列です。この文字列は、Web モジュールのプロパティシートで割り当てます。

つまり、Web モジュールで割り当てる URL パターンは、後で J2EE アプリケーションにモジュールを追加するときに割り当てる Web コンテキストからの相対的な位置になります。このシナリオの HTML は Web コンテキストからの相対的なリンクを使用するため、アプリケーションのアセンブル時に提供する Web コンテキストに関係なく、アプリケーションの実行時にリンクは適切に機能します。Web コンテキストの設定の詳細については、67 ページの「Web モジュールの Web コンテキストの設定」を参照してください。

## サーブレットマッピングの編集

サーブレットの作成時、Servelet Wizard のデフォルトの動作では、ウィザードの先頭のページで指定したクラス名がサーブレット名として使用され、サーブレット名を含む URL パターンにそのサーブレット名がマップされます。図 2-5 は、デフォルトの設定になっている AllItemsServlet の「サーブレットマッピング」プロパティエディタです。

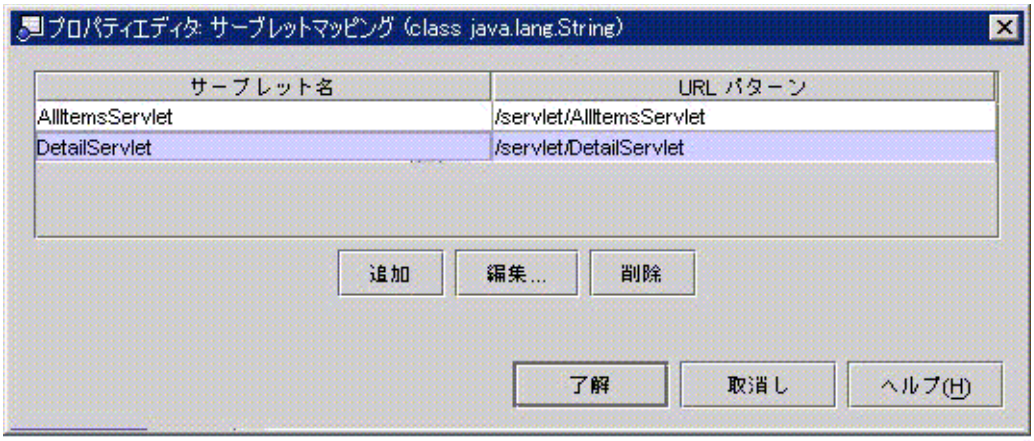


図 2-5　「サーブレットマッピング」プロパティエディタ

このサーブレットマッピングで Web モジュールを配備する場合は、AllItemsServlet が次の URL にマップされます。

```
http://hostname:port/web_context/servlet/AllItemsServlet
```

サーブレットに別の URL をマップしたい場合は、「サーブレットマッピング」プロパティエディタでマッピングを編集します。このシナリオでは、URL を処理しやすい形式に 変更します。

サーブレットマッピングを編集するには、次のようにします。

1. 「サーブレットマッピング」プロパティエディタを開きます。Web モジュールの `web.xml` ノードを右クリックし、コンテキストメニューの「プロパティ」>「サーブレットマッピング」> 省略符号ボタン (...) の順にクリックします。

   「サーブレットマッピング」プロパティエディタには、モジュール内のサーブレットとそれらのために設定されている URL が一覧表示されます。

2. 現在のマッピングを選択し、「編集...」ボタンをクリックします。

   サーブレットマッピングを編集するためのダイアログが開きます。「URL パターン」フィールドに **`/allItems`** と入力し、「了解」をクリックします。図 2-6 のサーブレットマッピング編集ダイアログには、AllItemsServlet と Detail Servlet の新しいマッピングが表示されています。

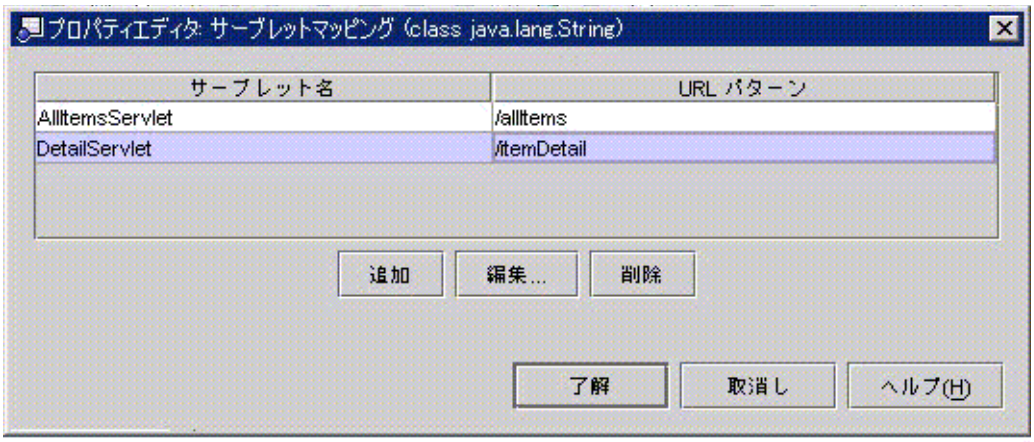


図 2-6　「サーブレットマッピング」プロパティエディタ

サーブレットの編集が終わると、次の URL で `AllItemsServlet` を実行できます。

```
http://hostnmame:port/web_context/allItems
```

これは、開始ページでリンクとして使用した URL です (コード例 2-1 参照) 。このリンクをクリックすると AllItemsServlet が実行されます。

## その他のアセンブル作業

この章のシナリオで使用しなかったアセンブル作業のいくつかを取り上げます。Web モジュールを実際に作成するときに役立ててください。

### エラーページの設定

モジュールのエラーページを指定する場合は、モジュールの配備記述子にそれらのページを指定する必要があります。それには、「エラーページ」プロパティエディタを使用します (このプロパティエディタを開くには、`web.xml` アイコンを右クリックし、コンテキストメニューの「プロパティ」>「配備」タブ >「エラーページ」> 省略符号ボタン (...) の順にクリックします。

エラーは、HTTP エラーコードと Java 例外クラスのどちらのカテゴリによっても識別できます。エディタには、エラーカテゴリごとに 2 つの「追加」ボタンがあります。どちらのカテゴリを使用する場合も、エラーを指定してページにマップします。図 2-7 は、エラーページが HTTP エラーコード 404 に割り当てられた後のプロパティエディタを示しています。

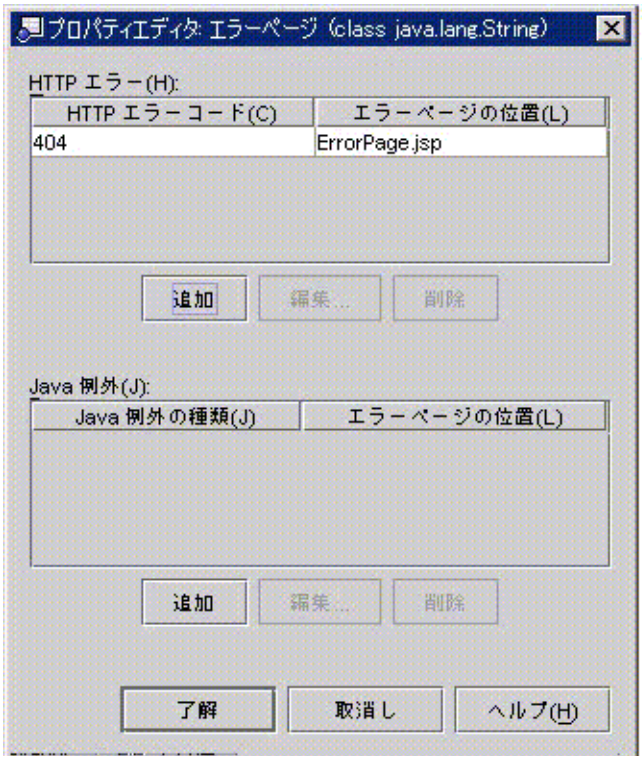


図 2-7　「エラーページ」プロパティエディタ

## JSP ページの設定

アセンブルしようとする Web モジュールに JSP ページコンポーネントが含まれている場合、これらのコンポーネントにはいくつかの実行方法があります。たとえば、`myJsp` という名前の新しい JSP ページを作成する場合は、次のいずれかの方法で実行できます。

### HTML リンクによる JSP ページの実行

JSP ページを HTML リンクから実行したい場合は、リンクを次の例のように設定します。

```
<a href="myJsp.jsp">Execute myJsp</a>
```

### プログラムによる JSP ページの実行

JSP ページを IDE で作成する場合、JSP ページの配備記述子エントリは作成されません。これは、ビジネスロジックがプログラムを使用して JSP ページにアクセスするときは問題になりません。たとえば、次のコードは、myJsp を実行するサーブレットの一部です。このコードは、JSP ページの実際のファイル名 (myJsp.jsp) を指定しています。

```
...
response.setContentType("text/html");
        RequestDispatcher dispatcher;
        dispatcher = getServletContext().getRequestDispatcher ("/myJsp.jsp");
        dispatcher.include(request, response);
...
```

### URL から JSP へのマッピング方法

先の例は、実行対象の JSP ページをその実際の名前 myJsp.jsp で識別しています。一方、URL パターンを JSP ページにマップし、その URL パターンを参照することによって JSP ページを実行する方法もあります。

JSP ページの URL マッピングを設定するには、次のようにします。

1. JSP ページのサーブレットを設定します。

   JSP ページのサーブレットを設定するには、「JSP ファイル」プロパティエディタを使用します

   a. web.xml ファイルを右クリックし、コンテキストメニューの「プロパティ」>「配備」区画タブ >「JSP ファイル」> 省略符号ボタン (...) の順にクリックします。

      プロパティエディタが開きます。

b. 「追加」ボタンをクリックし、ダイアログを開きます。このダイアログで JSP ページファイルを選択し、サーブレット名を割り当てます。

図 2-8 は、サーブレット名 `itemDetailPage` が `myJsp.jsp` にマップされた後の「JSP ファイル」プロパティエディタです。

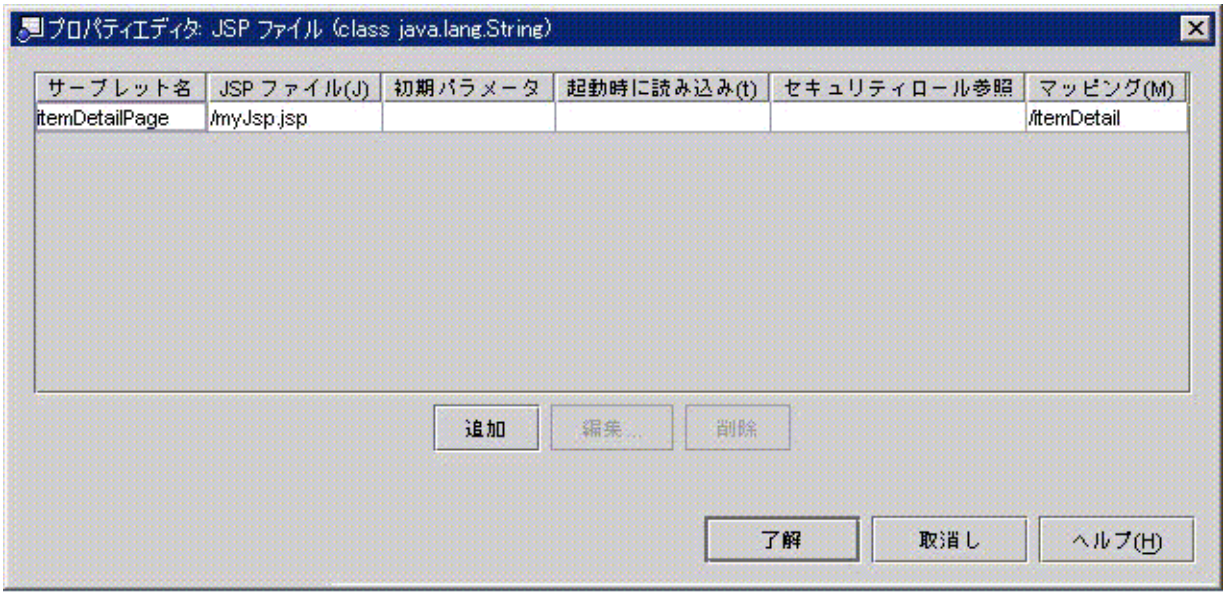


図 2-8　「JSPファイル」プロパティエディタ

2. URL パターンをサーブレット名にマップします。

a. 「サーブレットマッピング」プロパティエディタを開きます。「サーブレットマッピング」プロパティをクリックし、省略符号ボタン (...) をクリックします。

b. 「サーブレットマッピング」プロパティエディタの「追加...」ボタンをクリックして、新しいサーブレットマッピングを設定するためのダイアログを開きます。

JSP ページに割り当てたサーブレット名を入力します。次に、JSP ページにマップする URL パターンを入力します。図 2-9 の「サーブレットマッピング」プロパティエディタでは、URL パターン ItemDetail をサーブレット名 ItemDetailPage にマップしています。


図 2-9　「サーブレットマッピング」プロパティエディタ

このマッピングの結果、JSP ファイル　`myNewJSP.jsp` によって定義された JSP ページは、次の URL を使用して実行できるようになります。

```
http://hostnmame:port/web_context/ItemDetail
```

## 環境エントリ参照の設定

環境エントリには次の 2 つの部分があります。

1. JNDI ルックアップ
   環境エントリを使用する Web コンポーネントは、JNDI 命名機能を使用してエントリの値をルックアップします。
2. 環境エントリを示すエントリと、Web モジュールの配備記述子にある環境エントリ値
   このエントリは、J2EE 実行環境への参照を宣言します。

### 環境エントリ参照の JNDI ルックアップ

環境エントリ値を使用する Web コンポーネントには、次の例に示すようなコードが必要です。

```
try {
    // Obtain Initial Context--Opens Communication With JNDI Naming:
    Context ic = new InitialContext();
    // Request Lookup of Environment Entry Named &dlq;Cache Size&drq;:
    Integer cacheSize = (Integer)
     ic.lookup("java:comp/env/NumberOfRecordsCached");
}
catch(Exception e) {
    System.out.println(e.toString());
    e.printStackTrace();
    return;
}
```

コメントは、各行の動作を説明しています。

### 環境エントリの参照宣言

ほかの J2EE 参照と同様、環境エントリもプロパティエディタで設定します。この場合は、「環境エントリ」プロパティエディタを使用します (このプロパティエディタを開くには、`web.xml` ノードを右クリックし、コンテキストメニューの「プロパティ」>「参照」タブ >「環境エントリ」>「...」)ボタンの順にクリックします) 。

「追加」ボタンをクリックすると、新しい参照を追加するためのダイアログが開きます。参照を追加するには、Web コンポーネントコードの JNDI 名に一致する参照名、データの種類および初期値を指定する必要があります。「説明」には、アプリケーションのアセンブル担当者および配備担当者が各環境の正しい値を指定するのに役立つ情報を指定できます。図 2-10 は、これらのフィールドに値を入力した「環境エントリ」プロパティエディタを示しています。

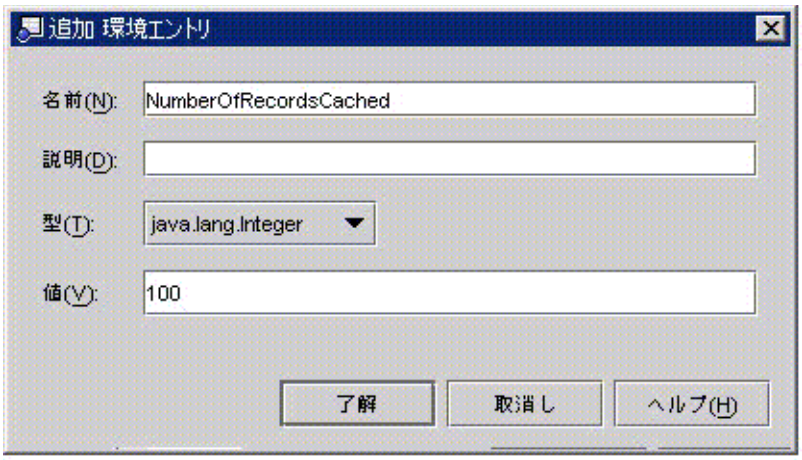


図 2-10 「環境エントリ」プロパティエディタ

# 第3章

# シナリオ： EJB モジュール

図 3-1 に、典型的な形式 (1 つのセッション Bean といくつかのエンティティ Bean を組み合わせたもの) で、典型的な対話を行う EJB モジュールを示します。EJB モジュールは、J2EE アプリケーション内でサーバー側のサービスを提供するため、アプリケーション内のほかのモジュールと対話できなければなりません (図の矢印 1)。ほとんどの EJB モジュールに複数の EJB コンポーネントが含まれ、エンタープライズ Bean が互いに対話します (図の矢印 2)。また、ほとんどの EJB モジュールは、リレーショナルデータベース管理システムなどの外部リソースと対話します (図の矢印 3)。

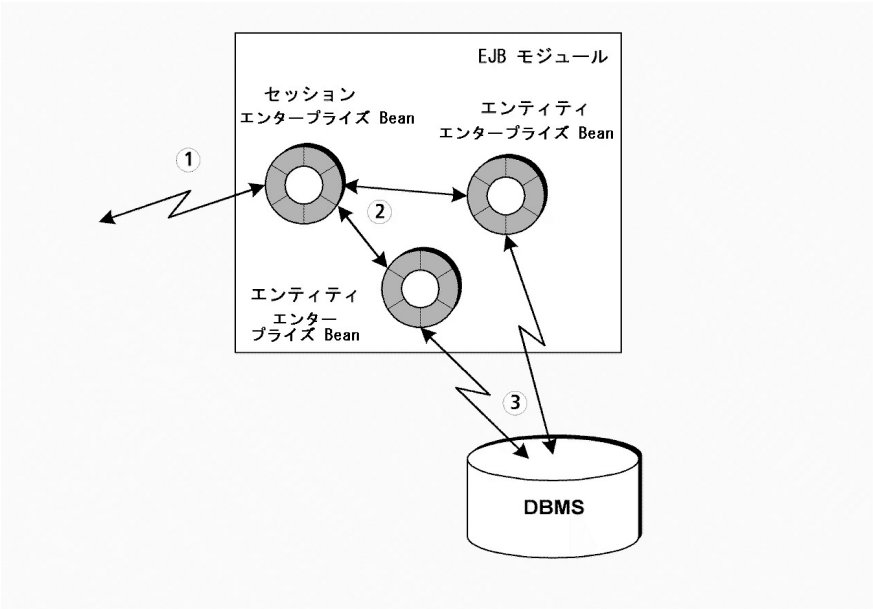


図 3-1　カタログ EJB モジュール

## モジュール内での対話

このシナリオでは、図 3-1 に示す 3 種類の対話に関与する EJB モジュールを取り上げます。このモジュールは、ショッピング Web サイトをサポートする J2EE アプリケーションの一部です。Web サイトアプリケーション内で、この EJB　モジュールはオンラインカタログのデータを提供します。これは、J2EE アプリケーションにおける EJB モジュールの代表的な役割です。

アプリケーション内のほかのモジュールのどれかがこのカタログ EJB モジュールのクライアントとして機能します。このクライアントモジュールは、通常は Web モジュールで、データを要求し、取得したデータを書式化してオンラインショッピングの利用者に表示します。カタログデータ EJB モジュールの仕事は、要求されたデータをデータベースから取得してクライアントモジュールに渡すことです。カタログデータ EJB モジュールは、データの要求を処理して正しいデータを返すことができなければなりません。具体的に、カタログデータを提供するには、次の対話が必要になります。

1. クライアントモジュールがカタログデータ EJB モジュールにカタログデータを要求します。この単純な例では、クライアントがカタログ内の全商品のリストまたは各商品の詳細情報のどちらかを要求します。
2. カタログデータ EJB モジュールが要求されたデータを取得するデータベース照会を生成することによって要求を処理します。
3. カタログデータ EJB モジュールが照会を実行し、クライアントモジュールにデータを返します。クライアントは受け取ったデータを書式化してユーザーに表示します。

クライアントモジュールとカタログ EJB モジュール間のこれらの対話には、選択する J2EE テクノロジとカタログデータモジュールの内部構造の双方とを決定するいくつかの特徴があります。まず、クライアントモジュールとカタログモジュール間の対話は同期対話です。つまり、オンラインショッピングの利用者は、カタログを要求しアプリケーションがそれを表示するのを待ちます。同期対話を提供するには、この対話を Java RMI で実装します。つまり、クライアントモジュールはカタログデータを要求するときに、カタログデータモジュールのメソッドを呼び出します。

2番目は、クライアントモジュールとカタログ EJB モジュール間の対話がセッション指向であることです。オンラインショッピングの利用者は、カタログ全体を見た後で特定の商品の詳細を要求できます。複数のエンドユーザーがカタログを同時に見ている場合、EJB モジュールは要求をユーザーセッションにマップし、各セッションに正

しいデータを返さなければなりません。このことから、クライアントがカタログ内のステートフルセッション Bean やエンタープライズ Bean と対話する必要があることが分かります。セッション Bean は、クライアントモジュールが送信する各要求処理を管理します (セッション Bean を使用したセッションのモデル化の詳細については、『Enterprise JavaBeans コンポーネントのプログラミング』を参照してください)。

データベース操作の場合、J2EE はエンティティエンタープライズ Bean と呼ばれるエンタープライズ Bean の一種を提供します。エンティティ Bean はデータベーステーブルを原型とし、照会を実行したり、原型としたテーブルに保管されているデータを操作したりするメソッドを持っています。カタログデータは 2 つのテーブルに保管されているため、カタログデータモジュールには、それぞれ 1 つのテーブルを原型とする 2 つのエンティティエンタープライズ Bean があります。データベース操作の接続、照会言語などの部分は、これらのエンティティ Bean によって処理されます。

これらの要件を考慮して、図 3-1 に示すようなモジュールを設計します。クライアントモジュールは、セッションエンタープライズ Bean とだけ対話します。セッション Bean は着信要求を管理し、エンティティエンタープライズ Bean のメソッドを呼び出して要求されたデータを取得します。

# モジュールのプログラミング

表 3-1 に、図 3-1 の EJB モジュールを作成するために必要なプログラミングをまとめ

ます。

表 3-1　このシナリオに必要なプログラミング

| アプリケーション要素 | 必要なプログラミング作業 |
|---|---|
| アプリケーションサーバー | なし |
| EJB モジュール | リモートインタフェース付きのセッション Bean (CatalogManagerBean) を作成します (リモートインターフェースは、ほかのモジュールからのリモートメソッド呼び出しの処理に適しています)<br>カタログデータを呼び出し側に返すビジネスメソッドを追加します。これらのビジネスメソッドの 1 つは、カタログ内のすべての商品を返します。ほかのメソッドは、クライアントによって選択された商品の詳細を返します<br>カタログデータを含んだ 2 つのデータベーステーブルを表現する 2 つのエンティティエンタープライズ Bean (itemBean と itemDetailBean) を作成します。エンティティエンタープライズ Bean のローカルインタフェースを作成します (これは、モジュール内のメソッド呼び出しに適しています)<br>カタログ内のすべての商品を返すメソッドを itemBean に追加します。特定の商品に関する詳細を要求するときは、itemDetailBean の findByPrimayKey メソッドが使用されます。セッション Bean はこれらのメソッドを呼び出してカタログデータを取得します<br>詳細クラス (Item テーブルと ItemDetail テーブルにそれぞれ 1 つずつ) を作成します。カタログデータモジュールは、これらのクラスのインスタンスを呼び出し側に返します<br>カタログデータ EJB モジュールを作成します。これにより、IDE のエクスプローラに EJB モジュールノードが作成されます<br>モジュールに 3 つのエンタープライズ Bean を追加します<br>EJB モジュールの作成後、モジュールのプロパティの 1 つとしてカタログデータのデータソースを指定します |
| J2EE アプリケーション | J2EE アプリケーションに EJB モジュールを追加する方法については、第 4 章 を参照してください |

この後、これらのプログラミングの作業手順を説明していきます。インタフェースとメソッドシグニチャを使用し、カタログデータモジュールへの入力とモジュールからの出力を示します。これらの入力と出力をほかのコンポーネント、ほかのモジュール、およびカタログデータベースに接続する手順を示します。エンタープライズ Bean の作成手順、ビジネスメソッドの追加手順、Java コードを使用したビジネスメソッドの実装手順は説明していません。これらの作業について学習する必要がある場合は、オンラインヘルプまたは 『Enterprise JavaBeans コンポーネントのプログラミング』を参照してください。

## セッションエンタープライズ Bean のリモートインタフェースの作成

カタログデータモジュールの設計は、クライアントモジュールが Java RMI を使用し、ステートフルセッション Bean のメソッドを呼び出すことによってカタログデータを取得するというものです。この対話を可能にするには、セッションエンタープライズ Bean がリモートインタフェースを持っていなければなりません。リモートインタフェースは、「セッション Bean」ウィザードでセッション Bean を作成するときに生成します (「セッション Bean」ウィザードの詳細についてはオンラインヘルプを参照してください)。コード例 3–1 に、ステートフルセッション Bean の完成したホームインタフェースとリモートインタフェースを示します。

コード例 3–1　　セッション Bean のホームインタフェースとリモートインタフェース

```
public interface CatalogManagerBeanHome extends javax.ejb.EJBHome {
  public CatalogBeans.CatalogManagerBean create()
    throws javax.ejb.CreateException, java.rmi.RemoteException;
}


public interface CatalogManagerBean extends javax.ejb.EJBObject {
  public java.util.Vector getAllItems() throws java.rmi.RemoteException;
  public CatalogBeans.idDetail getOneItemDetail(java.lang.String sku)
    throws java.rmi.RemoteException;
}
```

クライアントモジュールは、このインタフェースを使用して 2 つのことができます。クライアントがカタログ全体を見たいときは、getAllItems() メソッドを呼び出します。詳細を見たいときは、getOneItemDetail() メソッドを呼び出します (現実のショッピングアプリケーションのほとんどは、これより多くの機能を備えています)。機能を追加する場合は、これらのインタフェースにメソッドを追加します)。

カタログモジュールによってカプセル化されているこれらのメソッドの実装 (コード例 3-3）は、エンティティ Bean のメソッドを呼び出します。エンティティ Bean はデータベースに接続し、要求されたデータをエンティティ Bean インスタンスとして返します。

カタログデータモジュールのクライアントはデータベースを更新しないため、EJB モジュールはエンティティ Bean インスタンスの形でデータを返す必要がありません (エンティティ Bean インスタンスを返すと、クライアントがインスタンスのフィールドを更新できます。コンテナがエンティティ Bean インスタンスへの変更を追跡し、変更があった場合はデータベースの更新データを自動的に生成します)。代わりに、リモート参照をエンティティ Bean インスタンスに渡すときに消費されるネットワーク帯域幅を減らすため、カタログデータモジュールは通常の Java クラスのインスタンスを返します。これらのクラスは詳細クラスと呼ばれ、エンティティ Bean と同じフィールドを持っています。エンティティ Bean インスタンスからのデータが詳細クラスインスタンスにコピーされ、詳細クラスのインスタンスがクライアントに返されます。エンティティエンタープライズ Bean での詳細クラスの詳しい使用法については、『Enterprise JavaBeans コンポーネントのプログラミング』 を参照してください。

クライアントがこれらのリモートインタフェースを使用してカタログデータモジュールと対話する仕組みについては、第 2 章 を参照してください。

## エンティティエンタープライズ Bean のローカルインタフェースの作成

クライアントが要求するデータを取得するため、CatalogManagerBean はエンティティ Bean のメソッドを呼び出そうとします。これらの呼び出しはモジュール内で行われるため、リソースを消費するリモートインタフェースをエンティティ Bean が持つ必要はありません。代わりにローカルインタフェースを使用できます。ローカルインタフェースは、リモートインタフェースより高速で効率的であるため、リモート対話が必要ないときはローカルインタフェースを使用してください。

「エンティティ Bean」ウィザードを使用してエンティティ Bean を作成するとき、ローカルインタフェースを作成します (「セッション Bean」ウィザードの詳細についてはオンラインヘルプを参照してください)。コード例 3–2 に、エンティティ Bean の完成したローカルインタフェースを示します。 ほかのエンティティのインタフェースもほぼ同じになります。

コード例 3–2　　エンティティ Bean のローカルホームインタフェースとローカルインタフェース

```
public interface ItemBeanLocalHome extends javax.ejb.EJBLocalHome {
  public CatalogBeans.ItemBeanLocal findByPrimaryKey(java.lang.String aKey)
    throws javax.ejb.FinderException;
  public java.util.Collection findAll() throws javax.ejb.FinderException;
}


public interface ItemBeanLocal extends javax.ejb.EJBLocalObject {
  public CatalogBeans.iDetail getIDetail();
}
```

セッション Bean の CatalogManagerBean は、findAll() を呼び出してカタログ内の全商品のリストを取得できます。

---

**注 -** Sun ONE Studio 4 のテストアプリケーション機能を使用して個々のエンタープライズ Bean をテストする予定があれば、リモートインタフェースとローカルインタフェースの両方を生成しておく必要があります。テストアプリケーション機能は、1 つのエンタープライズ Bean のメソッド群を実行する Web モジュールクライアントを生成するため、この Web モジュールクライアントにリモートインタフェースが必要になるからです。

---

## セッションエンタープライズ Bean でのローカルインタフェースの使用法

クライアントモジュールは、CatalogManagerBean のビジネスメソッドの 1 つを呼び出すことによってデータを要求します (CatalogManagerBean インタフェースでメソッドが宣言されています。コード例 3–1 を参照してください)。これらのビジネスメソッドの実装は、カタログデータ要求の処理を管理します。管理する処理には、エンティティ Bean のメソッドの呼び出しが含まれます。エンティティ Bean メソッドを呼

び出すため、セッション Bean には、エンティティ Bean 向けに作成したローカルインタフェースへのローカル EJB 参照が必要です。ほとんどの J2EE インタフェースと同様、参照では次の 2 つの部分をプログラミングする必要があります。

- JNDI ルックアップのコード
  セッション Bean のビジネスメソッドのそれぞれに、JNDI 命名機能を使用してエンティティ Bean のローカルネームへの参照を取得するコードが含まれます。
- 参照の宣言
  実行時環境がルックアップコードによって参照される特定の Bean を識別するときに使用します。

## ローカル EJB 参照の JNDI ルックアップコード

コード例 3-3 は、セッション Bean の getAllItems() メソッドの実装です。オンラインカタログ内の全項目を一覧表示するデータを取得したいとき、クライアントモジュールはこのメソッドを呼び出します。このメソッドの実装から、CatalogManagerBean がどのようにして要求を管理するかが分かります。メソッドの実装は次の 3 つのステップに分けることができます。

1. セッション Bean が JNDI ルックアップを使用してエンティティ Bean へのローカル参照を取得します。
2. セッション Bean がエンティティ Bean の findAll() メソッドを呼び出します。
3. セッション Bean が、エンティティ Bean インスタンスから詳細クラスインスタンスにカタログデータをコピーした後で詳細クラスインスタンスをベクタにコピーし、ベクタをクライアントに返します。

コード内のコメントは、これらのステップを示しています。

コード例 3–3　　CatalogManagerBean の getAllItems メソッド

```
public java.util.Vector getAllItems() {
  java.util.Vector itemsVector = new java.util.Vector();
  try {
    if (this.itemHome == null) {
    try {
      // JNDI ルックアップを使用してエンティティ Bean のローカルへの参照を取得する
      InitialContext iC = new InitialContext();
      Object objref = iC.lookup("java:comp/env/ejb/ItemBean");
      itemHome = (ItemBeanLocalHome) objref;
    }
    catch(Exception e) {
      System.out.println("lookup problem" + e);
    }

    // ローカル参照を使用して findAll() を呼び出す
    java.util.Collection itemsColl = itemHome.findAll();
    if (itemsColl == null) {
      itemsVector = null;
    }
    else {
       // 詳細クラスインタフェースにデータをコピーする
      java.util.Iterator iter = itemsColl.iterator();
      while ( iter.hasNext() ) {
      iDetail detail;
      detail = ((CatalogBeans.ItemBeanLocal) iter.next()).getIDetail();
      itemsVector.addElement(detail);
    }
  catch(Exception e) {
    System.out.println(e);
  }
  return itemsVector;
}
```

lookup 文は「ItemBean」を指定していますが、この文字列は実際には参照先のエンタープライズ Bean の名前ではなく、参照の名前を表します。このように、エンタープライズ Bean の名前を参照名として使用して、どのエンタープライズ Bean を意図しているか分かりやすくすることがよくあります。エンタープライズ Bean は、次のステップで指定します。

## ローカル EJB 参照の参照宣言

JNDI ルックアップコードを完成させるには、CatalogManagerBean プロパティとして参照宣言を設定する必要があります。参照宣言は、参照名 (lookup 文で使用されている名前) を実際のエンタープライズ Bean にマップします。参照宣言を設定するには、次のようにします。

1. 呼び出し側 Bean のノード (この場合は、CatalogManagerBean) を右クリックし、コンテキストメニューの「プロパティ」をクリックし、「参照」タブ、「EJB ローカル参照」プロパティ、省略符号ボタン (...) 、「追加」ボタンの順にクリックします。

   「追加」ボタンをクリックすると、新しい参照宣言を追加するためのダイアログが開きます。

2. 参照宣言を追加するには、lookup 文で使用されている名前と一致する参照名を指定し、それをエンタープライズ Bean の名前にマップする必要があります。

   通常、そのためには、参照名として JNDI lookup 文で使用した文字列と正確に一致する文字列を入力した後、「ブラウズ」ボタンをクリックします。そうすると、一致するエンタープライズ Bean を選択するダイアログが開きます。図 3-2 に、CatalogManagerBean の「追加 EJB 参照」ダイアログを示します。ここでは、各フィールドに ItemBean へのローカル参照の値が入力されています。

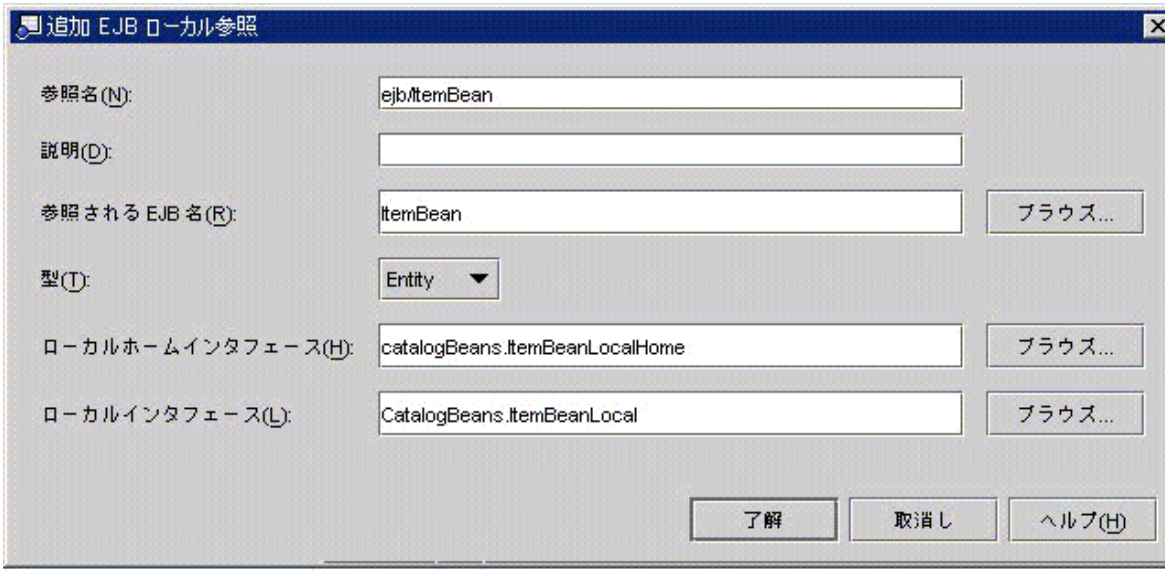


図 3-2　「追加 EJB 参照」ダイアログ

## EJB モジュールのアセンブル

カタログデータモジュールを形成するエンタープライズ Bean 群の作成が終わったら、EJB モジュールの作成と構成を行う必要があります。Sun ONE Studio 4 IDE では、モジュールのプロパティシート上でプロパティを設定することによってモジュールを構成し、モジュールの実行時の動作を記述したり、J2EE 環境に特定の実行時サービスを要求したりします。

モジュールのプロパティには、さまざまな J2EE 実行環境 (さまざまなアプリケーションサーバー製品) のためのサーバー固有プロパティがあります。

## EJB モジュールの作成

IDE で EJB モジュールを作成するには、2 つの方法があります。どちらの方法で作成しても、指定した場所に EJB モジュールノードが作成されます。EJB モジュールノードは、モジュールの内容の記述を表現します。モジュールに追加するすべてのコンポーネントのソースファイルがこのノードによって識別されます。ただし、EJB モジュールノードを置いたディレクトリにソースファイルはコピーされません。EJB モジュールの配置場所を決めるときは、このことを忘れないでください。

モジュールのソースコードすべてを単一のファイルシステムに保持する場合は、そのファイルシステムの最上位にモジュールノードを配置できます。一方、モジュールのソースコードが複数のファイルシステムに存在し、それらのファイルシステムを複数の開発者が所有している場合は、ソースコードを含むファイルシステムとは別に、モジュールと J2EE アプリケーションだけを含むディレクトリ群を作成する方法もあります。

エンタープライズ Bean ノードから EJB モジュールを作成するには、次のようにします。

1. エンタープライズ Bean ノードを右クリックし、コンテキストメニューの「新規 EJB モジュール」を選択します。

   モジュールの名前とファイルシステム内での位置を指定するためのダイアログが開きます。

2. このダイアログで、ファイルシステム、ディレクトリ、またはパッケージを選択します。

   選択したファイルシステム、ディレクトリ、またはノードの下に新しいモジュールを表すノードが作成され、この作業の最初に右クリックしたエンタープライズ Bean が新しいモジュールに自動的に含まれます。

   モジュールに複数のエンタープライズ Bean を追加できます。53 ページの「エンタープライズ Bean などのリソースのモジュールへの追加」 を参照してください。

   ファイルシステム、パッケージ、またはディレクトリノードから EJB モジュールを作成するには、次のようにします。

1. ディレクトリノードを右クリックし、コンテキストメニューの 「新規」>「J2EE」>「EJBモジュール」を順にクリックします。

   モジュールの名前を指定するためのダイアログが開きます。

2. 「了解」をクリックします。

   新しいモジュールを表すノードが、選択したファイルシステム、パッケージ、またはディレクトリの下に作成されます。

   どちらの手順でも、新しいモジュールは選択したディレクトリ内のノードによって表現されます。モジュールを記述する情報およびモジュールの配備記述子を生成するために最終的に使用される情報がこのディレクトリに保存されます。ただし、モジュールのコンポーネントのソースコードはこのディレクトリにコピーされません。

## エンタープライズ Bean などのリソースのモジュールへの追加

モジュールの作成が終われば、そのモジュールにエンタープライズ Bean を追加できます。EJB モジュールにエンタープライズ Bean を追加するには、次のようにします。

1. モジュールノードを右クリックし、コンテキストメニューの 「EJB を追加...」をクリックします。

   ダイアログが開き、エンタープライズ Bean のためにマウントされているすべてのファイルシステムを参照できます。

2. エンタープライズ Bean を選択し、「了解」をクリックします。

   モジュールノードの下に、エンタープライズ Bean を表すノードが追加されます。

   モジュールに属するすべてのエンタープライズ Bean を追加するまで、このコマンドを使用し続けることができます。

   エンタープライズ Bean をモジュールに追加すると、IDE によって、そのエンタープライズ Bean とほかの種類のリソース (Java クラス、ファイル、その他) との依存関係が管理されます。その結果、追加したエンタープライズ Bean と依存関係を持つすべてのリソースがモジュールのコンパイル時または配備時に自動的に組み込まれます。

   これにはいくつかの例外があります。詳細については、59 ページの「追加のファイルの指定」 を参照してください。

   なお、モジュールに追加するエンタープライズ Bean のソースコードは、モジュールノードを保持するディレクトリにコピーされません。

## エンティティエンタープライズ Bean のデータソースの指定

このシナリオでは、CatalogData モジュールがデータベースにアクセスしてカタログデータを取得しなければなりません。モジュールを構成すると、つまりアセンブル時に、データソースを指定する必要があります。エンティティ Bean に対し、データソースを指する 2 通りの方法があります。まず、開発時に、データベースへの接続を指定し、テーブルを選択し、「エンティティ Bean」ウィザードを使用して選択したテーブルをモデルとするエンティティ Bean を作成します (直接接続の代わりにデータベーススキーマを使用することもできます)。この手順は、『Enterprise JavaBeans コンポーネントのプログラミング』 に記載されています。

2 番目は、エンティティ Bean を配備する前に、実行時に使用するデータベースを指定する必要があります。ここでは、この 2 番目のケースを取り上げます。実行時に使用するデータソースを指定するには、EJB モジュールのプロパティシートを使用してデータソースのリソース参照を設定します。

リソース参照の設定方法は、使用しているエンティティ Bean の種類によって異なります。

- コンテナ管理によるエンティティ Bean を使用している場合は、モジュールのサーバー固有プロパティタブの 1 つを使用してデータソースを指定します。データソースの識別方法は、サーバー製品によって決まります。J2EE リファレンス実装を含め、ほとんどの場合は データソースを JNDI 名によって識別します。
- Bean 管理によるエンティティ Bean を使用している場合は、JNDI ルックアップコードとルックアップをデータソースの JNDI 名にマップする参照宣言とによってリソース参照を設定する必要があります。

### コンテナ管理によるエンティティ Bean の場合のサーバー固有タブの使用法

コンテナ管理によるエンティティ Bean の場合は、モジュールのプロパティシートで JNDI 名によってデータソースを指定するだけです。そうすれば、IDE によってリソース参照が自動的に作成されます。

データソースの JNDI 名は、特定のアプリケーションサーバーに対してデータソースを定義するときに割り当てられます。使用中の J2EE 実行環境 (アプリケーションサーバー製品) のサーバー固有タブで、その環境に有効な JNDI 名を指定する必要がありま

す。図 3-3 に、CatalogData モジュールの「J2EE RI」タブを示します。「Data Source JNDI name」プロパティには、デフォルトの Pointbase データベースの JNDI 名が入っています。

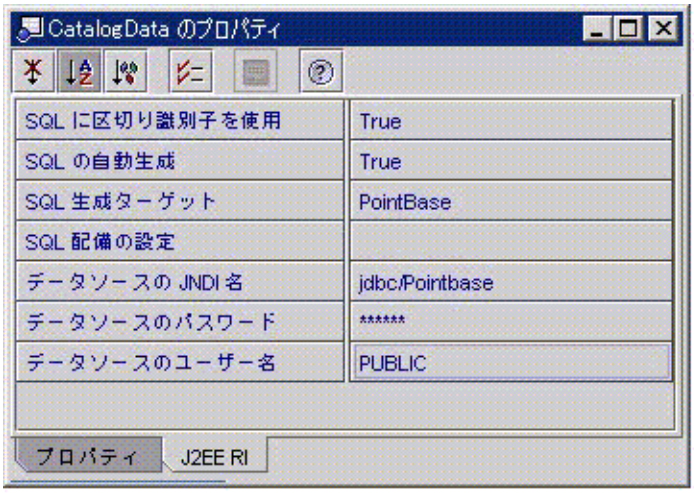


図 3-3　EJB モジュールのサーバー固有プロパティ

J2EE RI サーバーと PointBase データベースサーバーはどちらも、Sun ONE Studio 4 とともにインストールされています。さらに、Pointbase データベースは J2EE RI サーバーに対して 「jdbc/Pointbase」として定義されているため、「データソースの JNDI 名」プロパティでこの名前を指定するときは特定のデータベースインスタンスを指します。

J2EE RI サーバーとともに別のデータベース製品を使用している場合は、J2EE RI 管理ツールを使用して J2EE RI サーバーに対してデータソースを定義する必要があります。その作業が終われば、認識可能な JNDI 名がデータソースに割り当てられます。

管理されたテスト環境または本番環境に配備しようとしている場合、アプリケーションサーバーに対するデータソースの定義はおそらくシステム管理担当者の仕事です。その場合は、管理担当者からデータソースの JNDI 名を入手するだけです。

J2EE RI 以外のアプリケーションサーバーを使用している場合は、そのアプリケーションサーバーのデータベース定義を行わなければなりません。その手順は、使用するアプリケーションサーバー製品によって異なります。おそらくこれも、管理されたテスト環境または本番環境ではシステム管理担当者の仕事です。

### Bean 管理によるエンティティ Bean の場合のリソースファクトリ参照の明示的な作成

EJB モジュールに、コンテナ管理による持続性機能ではなく、Bean 管理による持続性機能を使用するエンティティ Bean が含まれている場合は、データベースからデータを抽出する JDBC と JTA はすでに作成済みです。この手順は、『Enterprise JavaBeans コンポーネントのプログラミング』 に記載されています。さらに、特定のデータソースに接続するときに使用するリソース参照を明示的にコーディングする必要があります。CatalogManagerBean でコーディングしたローカル EJB 参照と同様、この参照には次の 2 つの部分があります。

- JNDI ルックアップのコード
  セッション Bean のビジネスメソッドのそれぞれに、JNDI 命名機能を使用してエンティティ Bean のローカルホームへの参照を取得するコードが含まれます。
- 参照の宣言
  実行時環境がルックアップコードによって参照される特定の Bean を識別するときに使用します。

明示的なルックアップで指定するデータソースは、コンテナ管理による Bean の場合のリソース参照が自動的に生成されるのとまったく同じように、名前付きのリソースでなければなりません。

### リソースファクトリ参照の JNDI ルックアップのコード

コード例 3–4 に、BMP エンティティエンタープライズ Bean の中でデータソースのルックアップに使用するコードを示します。

コード例 3–4　データベースの JNDI ルックアップ

```
try {
  // Obtain Initial Context--Opens Communication With JNDI Naming:
  Context ic = new InitialContext();
  // Request Lookup of Resource--In This Example a JDBC Datasource:
  javax.sql.DataSource hrDB = (javax.sql.DataSource)
                                  ic.lookup("java:comp/env/jdbc/Local_HR_DB");
}
catch(Exception e) {
  System.out.println(e.toString());
  e.printStackTrace();
  return;
}
```

### リソース参照の参照宣言

参照宣言を設定するには、エンティティ Bean のプロパティシートを使用します。

1. エンタープライズ Bean の論理ノードを右クリックし、コンテキストメニューの「プロパティ」>「参照」タブ >「リソース参照」> 省略符号ボタン (...) の順にクリックします。

   「リソース参照」プロパティエディタが開きます。

2. 「追加」ボタンをクリックし、新しいリソース参照を追加するためのダイアログを開きます。

   参照には、参照名 (JNDI lookup 文で使用している名前)、リソースの種類、および承認の種類が含まれていなければなりません。図 3-4 に、これらのフィールドに コード例 3-4 と一致する値が入力された「リソース参照プロパティの追加」ダイアログを示します。

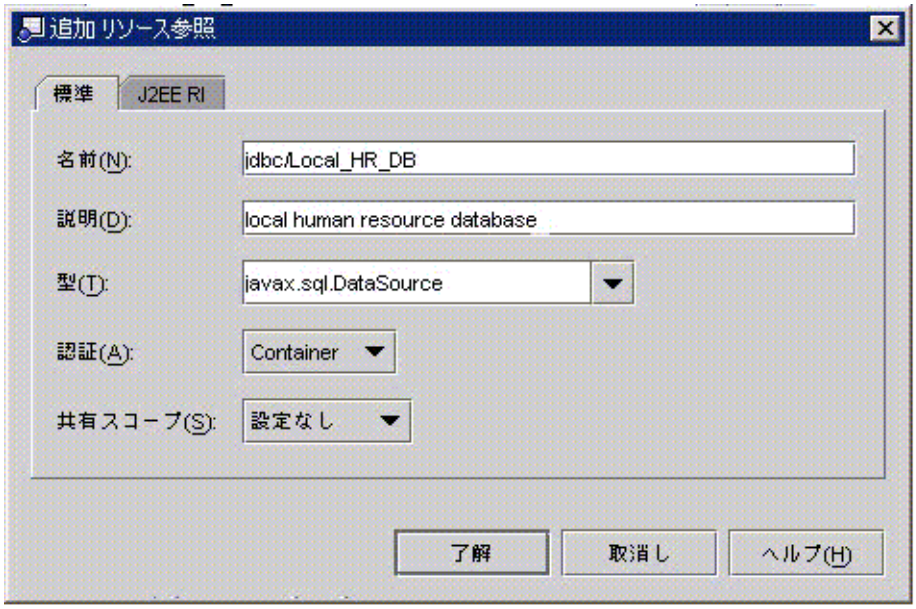


図 3-4　「追加リソース参照」ダイアログボックス

EJB 参照と異なり、リソース参照はアセンブル時または配備時にほかの J2EE コンポーネントにリンクされません。その代わり、アプリケーションサーバーの環境で名前付きリソースが別に定義されます。エンタープライズ Bean は JNDI ルックアップを使用し、実行時にこれらのリソースへの参照を取得します。この参照がデータベースインスタンスに接続することになります。

3. 「追加」ダイアログのサーバー固有タブをクリックし、リソースを確認します。

   図 3-5 は、「追加」ダイアログの「J2EE RI」タブです。「JNDI 名」フォールドには、jdbc/Pointbase という値が入力されています。この値は、jdbc/Local_HR_DB という名前のリソース参照を該当するデータベースにリンクします。

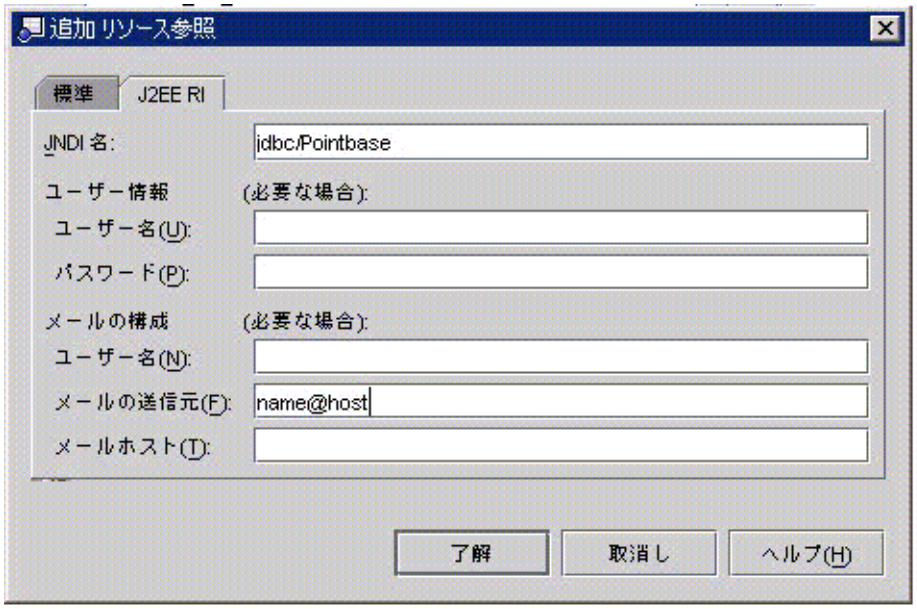


図 3-5　「追加リソース参照」ダイアログボックスのサーバー固有タブ

## 追加のファイルの指定

ほとんどの場合、IDE は EJB モジュールに含まれるエンタープライズ Bean の依存関係を認識し、配備時に作成する EJB JAR ファイルに必要なすべてのファイルを組み込みます。ただし、次のような依存関係は IDE に認識されません。

- モジュール内の Enterpise Bean が直接呼び出しでなくヘルプファイルを使用する場合。
- エンタープライズ Bean がクラスに動的にアクセスし、クラス名がクラス宣言ではなく文字列としてコードに記述されている場合。

この 2 つのケースや、それと似たようなケースでは、IDE が依存関係を認識せず、作成するアーカイブファイルにヘルプファイルやクラスファイルを含めません。これらのファイルが実行時に利用可能になるように追加のファイルとして指定する必要があります。

追加のファイルを指定するには、次のようにします。

1. モジュールのノードを右クリックします。コンテキストメニューの「プロパティ」>「追加のファイル」> 省略符号ボタン (...) の順にクリックします。

   「追加のファイル」プロパティエディタが開きます。

2. モジュールの一部として配備またはアーカイブする必要のある追加のファイルを選択します。

   「追加のファイル」プロパティエディタでは、マウントされているすべてのファイルシステムを参照できます。追加のファイルをクリックし、「追加」ボタンをクリックしてファイルをリストに追加します。

## 重複した JAR ファイルの除外

ファイルの依存関係の中にも、IDEに処理してほしくないものもあります。特に、モジュール内のコンポーネントが一般に使われる JAR ファイルとの依存関係を持ち、その JAR ファイルが実行環境にすでに存在していることが分かっている場合があります。その場合は、JAR ファイルの不必要なコピーをモジュールのアーカイブファイルに追加しないようにすることができます。

重複した JAR ファイルを除外するには、次のようにします。

1. モジュールのノードを右クリックします。コンテキストメニューの「プロパティ」>「除外されるライブラリ Jar ファイル」> 省略符号ボタン (...) の順にクリックします。

   「除外されるライブラリ Jar ファイル」プロパティエディタが開きます。マウントされている JAR　ファイルのリストが表示されます。

2. 除外する JAR ファイルを選択し、「追加」ボタンをクリックして除外対象 JAR ファイルのリストに追加します。

## モジュールのアセンブル時に必要なその他の作業

CatalogData モジュールの、実行時データソースの指定は、EJB モジュールのアセンブルと構成の段階で行う作業の 1 つです。モジュールによっては、アセンブル時にほかの作業が必要になります。どのような作業が必要かをしっかりと見極めなければなりません。たとえば、次の点を確認してください。

- モジュール内で使用される参照がすべてリンクされているか。ほかのモジュール内に存在するコンポーネントへの参照は、モジュールをアセンブルして J2EE アプリケーションを作成するまでリンクできません。
- 汎用セキュリティロールがモジュールに設定されているか
  モジュールのエンタープライズ Bean 内のセキュリティロール参照がこれらの汎用セキュリティロールにリンクされているか。メソッドアクセス権がこれらの汎用セキュリティロールにマップされているか
  詳細については、第 8 章 を参照してください。
- コンテナ管理によるトランザクションが定義されているか
  詳細については、第 7 章 を参照してください。

# 第4章

# シナリオ： Web モジュールと EJB モジュール

図 4-1 に、Web モジュールと EJB モジュールをアセンブルして作成した J2EE アプリケーションを示します。 2 つのモジュール間の対話は Java RMI です。つまり、Web モジュール内のサーブレットが EJB モジュールへのリモートメソッド呼び出しを行います。図に示されているほかの対話 (ユーザーのブラウザとアプリケーションとの間の HTTP 要求と HTTP 応答、アプリケーションが実行するデータベース照会) は、モジュール内にプログラムされています。

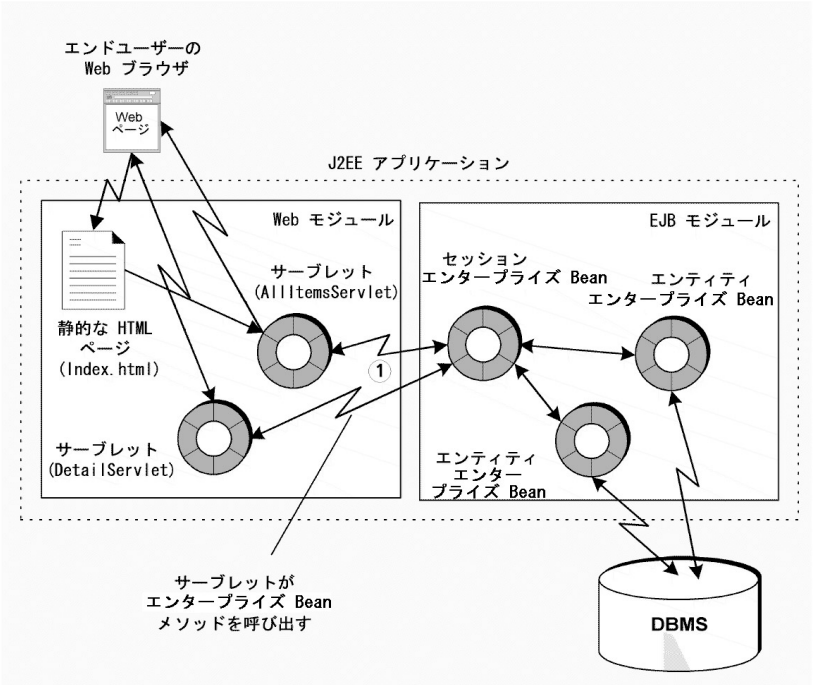


図 4-1　Web モジュールと EJB モジュールをアセンブルして作成した J2EE アプリケーション

## アプリケーション内での対話

このシナリオでは、図 4-1 に示すすべての対話に関与する J2EE アプリケーションを取り上げます。ユーザーとの HTTP 対話に必要なプログラミングは Web モジュール内部で行います。詳細については、第 2 章を参照してください。データベースとの対話に必要なプログラミングは EJB モジュール内で行います。詳細については、第 3 章を参照してください。このシナリオでは、2 つのモジュールをアセンブルして J2EE アプリケーションを作成する手順を示します。

両方のモジュールに依存する 1 つの対話は、モジュール間のリモートメソッド呼び出しです。この対話に必要な論理はすでにモジュール内に存在します。Web モジュールは JNDI ルックアップコードと EJB 参照宣言を含み (28 ページの「EJB 参照の参照宣言」 参照してください)、EJB モジュールはリモートメソッド呼び出しをサポートするリモートインタフェースを含んでいます (46 ページの「セッションエンタープライズ Bean のリモートインタフェースの作成」)。

この章で説明する作業を実施することで、2 つのモジュールがアセンブルされ、J2EE アプリケーションが作成されます。その後、アプリケーションを 1 つの単位として配備し、実行することができます。リモートメソッド呼び出しにつながるエンドユーザーの操作については、20 ページの「モジュール内での対話」 を参照してください。

## アプリケーションのプログラミング

表 4-1 に、このシナリオで説明する J2EE アプリケーションを作成するために必要なプログラミングをまとめます。

**表 4-1**　このシナリオに必要なプログラミング

| アプリケーション要素 | 必要なプログラミング作業 |
|---|---|
| アプリケーションサーバー | なし |

表 4-1　このシナリオに必要なプログラミング (続き)

| アプリケーション要素 | 必要なプログラミング作業 |
|---|---|
| Web モジュール | 第 2 章を参照してください。 |
| EJB モジュール | 第 3 章を参照してください。 |
| J2EE アプリケーション | Catalog アプリケーションを作成します。これにより、Sun ONE Studio 4 エクスプローラに J2EE アプリケーションノードが作成されます。アプリケーションに 2 つのモジュールを追加します。<br>Web モジュールの Web コンテキストを指定します。<br>Web モジュールの EJB 参照が EJB モジュールのエンタープライズ Bean に正しくリンクされていることを確認します |

この後、単純な例を使用してこれらのプログラミング作業の大部分を説明していきます。

## J2EE アプリケーションのアセンブル

Catalog アプリケーションを構成するモジュールを作成した後、両方のモジュールを含む J2EE アプリケーションの作成と設定を行う必要があります。Forte for Java 4 IDE では、アプリケーションノードを作成した後、作成したノードのプロパティシートを使用してアプリケーションを設定します。アプリケーションのプロパティシートでは、アプリケーションの実行時の動作を記述したり、J2EE 環境の独自の実行時サービスを要求したりします。

アプリケーションのプロパティには、J2EE アプリケーションによって定義されている標準プロパティと個々のアプリケーションサーバー製品専用のサーバー固有プロパティの両方が含まれます。

### J2EE アプリケーションの作成

IDE で J2EE アプリケーションを作成するには、2 つの方法があります。どちらの方法で作成しても、指定した場所にアプリケーションノードが作成されます。J2EE アプリケーションノードは、アプリケーションの内容の記述を表現します。モジュールに追加するすべてのコンポーネントのソースファイルがこのノードによって識別されます。ただし、アプリケーションノードを置いたディレクトリにソースファイルはコピーされません。J2EE アプリケーションノードの配置場所を決めるときは、このことを忘れないでください。

モジュールのソースコードすべてを単一のファイルシステムに保持する場合は、そのファイルシステムの最上位にモジュールノードを配置できます。一方、モジュールのソースコードが複数のファイルシステムに存在し、それらのファイルシステムを複数の開発者が所有している場合は、ソースコードを含むファイルシステムとは別に、モジュールと J2EE アプリケーションだけを含むディレクトリ群を作成する方法もあります。このようなファイルシステムは、ソースコードのディレクトリ構造とは異なるアプリケーションの構造を確認するのに役立ちます。

モジュールノードから J2EE アプリケーションを作成するには、次のようにします。

1. EJB モジュールノード (または Web モジュールの `WEB-INF` ノード) を右クリックし、コンテキストメニューの「新規 J2EE アプリケーション」を選択します。

   アプリケーションの名前とファイルシステム内での位置を指定するためのダイアログが開きます。

   このダイアログで、ファイルシステム、ディレクトリ、またはパッケージノードを選択し、「完了」をクリックします。

2. 選択したファイルシステム、ディレクトリ、またはノードの下に新しいモジュールを表すノードが作成され、この作業の最初に右クリックしたモジュールが新しいアプリケーションに自動的に含まれます。

   アプリケーションにさらにモジュールを追加できます。67 ページの「J2EE アプリケーションへのモジュールの追加」 を参照してください。

ファイルシステム、パッケージ、またはディレクトリのノードから EJB モジュールを作成するには、次のようにします。

1. エクスプローラウィンドウのノードを右クリックし、コンテキストメニューの 「新規」>「J2EE」>「アプリケーション」を選択します。

   アプリケーションの名前を指定するためのダイアログが開きます。

2. 「完了」をクリックします。

   新しいアプリケーションを表すノードが、選択したファイルシステム、パッケージ、またはディレクトリの下に作成されます。

どちらの手順でも、新しいモジュールは選択したディレクトリ内のノードによって表現されます。モジュールを記述する情報、およびモジュールの配備記述子を生成するために最終的に使用される情報がこのディレクトリに保存されます。ただし、モジュールのコンポーネントのソースコードはこのディレクトリにコピーされません。

### J2EE アプリケーションへのモジュールの追加

アプリケーションが作成できれば、そのアプリケーションにモジュールを追加できます。アプリケーションにモジュールを追加するには、次のようにします。

1. アプリケーションノードを右クリックし、コンテキストメニューの「モジュールを追加...」を選択します。

   ダイアログが開き、モジュールのためにマウントされているすべてのファイルシステムを参照できます。

2. モジュールを選択し、「了解」をクリックします。

   アプリケーションノードの下にモジュールを表すノードが追加されます。

   - Web モジュールを追加するには、モジュールの `WEB-INF` ノードを選択します。
   - EJB モジュールを追加するには、モジュールノードを選択します。

   この操作を繰り返し、アプリケーションに属するモジュールすべてを追加します。

   J2EE アプリケーションにモジュールを追加するとき、IDE によってエンタープライズ Bean とほかの種類のリソース (Java クラス、ファイル、その他) との依存関係が認識され、必要なファイルが自動的にモジュールに組み込まれます。

   なお、アプリケーションに追加するモジュールのソースコードは、アプリケーションノードを保持するディレクトリにコピーされません。

## Web モジュールの Web コンテキストの設定

J2EE アプリケーションを J2EE アプリケーションサーバーに配備するとき、Web モジュールリソースには URL パスの後ろにそのリソースの名前を付加した URL が割り当てられます。J2EE RI の場合、URL パスは一般に次の形式になります。

```
http://hostnmame:port/web_context/URL_pattern
```

このパスの要素は、次のように決定されます。

- *hostnmame* はアプリケーションサーバーが実行されているコンピュータの名前、*port* はそのサーバーインスタンスの HTTP 要求のために指定されたポートです。通常、ポート番号はアプリケーションサーバーのインストール時に割り当てられます。Sun ONE Studio 4 によってインストールされる J2EE RIの場合、HTTP ポート番号は 8000 です。

- *web_context* は、J2EE アプリケーションにモジュールを追加した後でモジュールのプロパティとして指定する文字列です。この文字列は、モジュール内のすべての Web リソースを修飾します。
- *URL_pattern* (URL パターン) は、Web モジュールのプロパティシートで特定のサーブレットにマップされている文字列です。31 ページの「URL からサーブレットへのマッピング」 を参照してください。

つまり、Web モジュールのプロパティシートで割り当てられている URL パターンは、この手順で割り当てるコンテキストからの相対位置になります。Web コンテキストを設定するには、次のようにします。

1. 組み込まれている Web モジュールノード (J2EE アプリケーションノードの下の Web モジュールノード) を右クリックし、コンテキストメニューの「プロパティ」を選択します。
2. 「Web コンテキスト」プロパティを選択し、使用したい文字列を入力します。

   図 4-2 に、カタログ Web モジュールのプロパティシートを示します。ここでは、Web コンテキストが `catalog` に設定されています。

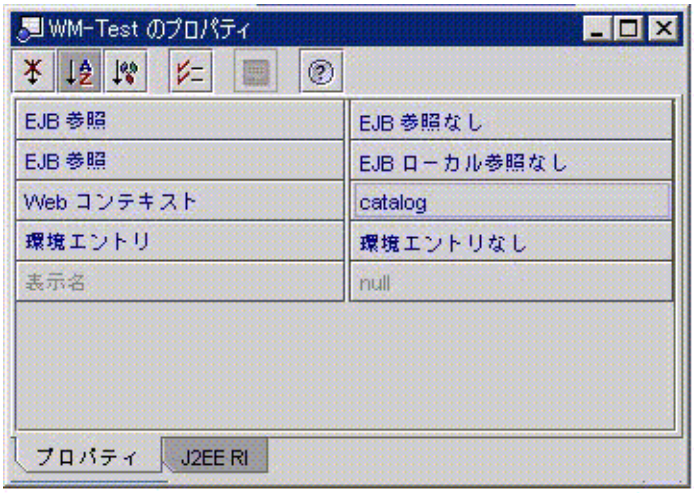


図 4-2　カタログ Web モジュールのプロパティシート

この値を使用した場合、アプリケーションの Web リソースの URL は、一般に次の形式をとるようになります。

```
http://hostnmame:port/catalog/URL_pattern
```

Web コンテキストを指定しない場合、コンテキストはデフォルトでブランクに設定され、アプリケーションの Web リソースの URL は、一般に次の形式をとるようになります。

```
http://hostnmame:port/URL_pattern
```

## EJB 参照のリンク

モジュール間の Java RMI による対話には、リンクされている EJB 参照が必要です。Web モジュールには、EJB 参照の両方の部分 (JNDI ルックアップのコードと参照宣言) が含まれています。リモートメソッド呼び出しを正常に行うには、参照がアプリケーションのエンタープライズ Bean にリンクされている必要があります。

参照のプログラミングについては、第 2 章のシナリオで取り上げました。そこでは、参照のリンクに関し、状況によって 2 つの選択肢があることを説明しました。1つはアプリケーションを作成する前に Web モジュールのプロパティシート上で参照をリンクする方法、もう 1 つは Web モジュールのプロパティシートでは参照をリンクしないで、Web モジュールをアセンブルしてアプリケーションを作成するときにリンクする方法です。

ここでは、アプリケーションノードのエンタープライズ Bean の参照プロパティを操作する方法を説明します。アプリケーションで宣言されている参照の状態を確認する方法と、必要に応じて参照をリンクする方法を説明します。

アプリケーションの参照を操作するには、次のようにします。

1. アプリケーションノードを右クリックし、コンテキストメニューの「プロパティ」>「EJB参照」> 省略符号ボタン (...) を選択します。

   アプリケーションの「EJB 参照」エディタが開きます。

2. EJB 参照の状態を確認します。

   このエディタには、アプリケーションを構成するモジュールで宣言されているすべての参照が表示されます。参照は、モジュールと参照名 (JNDI lookup 文で使用されている名前) によって識別されます。

   図 4-3 に、このシナリオのアプリケーションのプロパティエディタを示します。カタログ Web モジュールには、1 つの EJB 参照があります。この参照は `ejb/CatalogManagerBean` という名前で、EJB モジュールのエンタープライズ Bean (名前は `CatalogBeans.CatalogManagerBean`) によって解決されます。

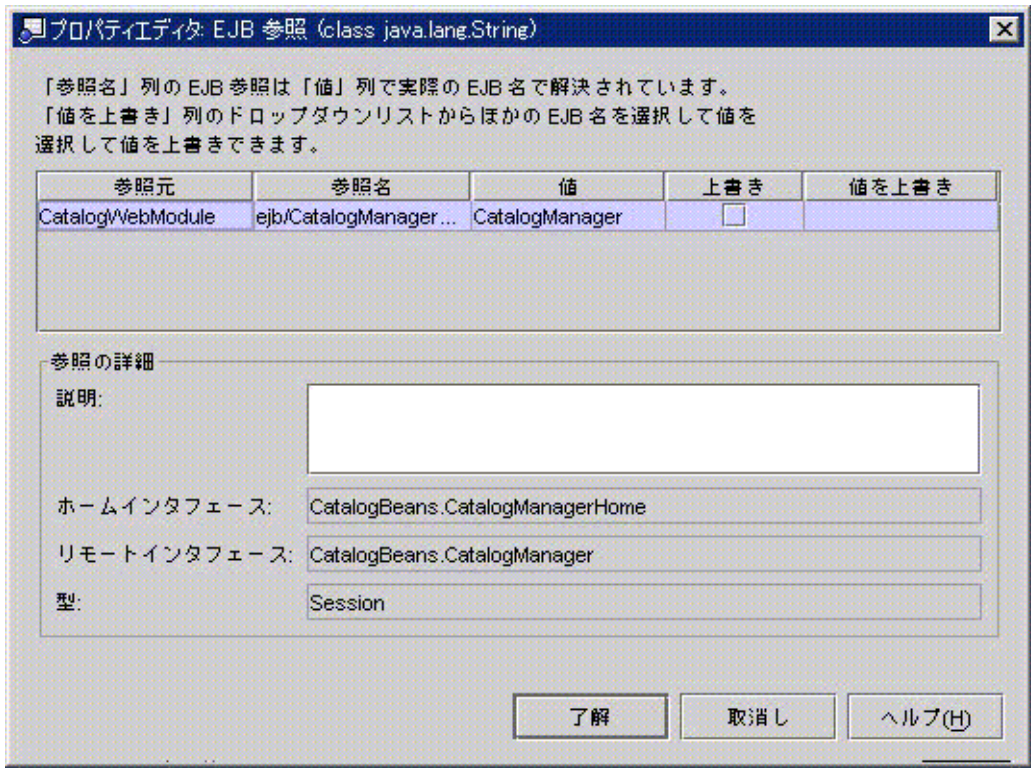


図 4-3　アプリケーションノードの「EJB参照プロパティ」エディタ

参照がリンクされていない場合は、その「値」フィールドが空白で、「エラー状態」フィールドに参照の問題点が示されます。この参照をWeb モジュールのプロパティシートでリンクしなかった場合は、図 4-4 のように表示されます。

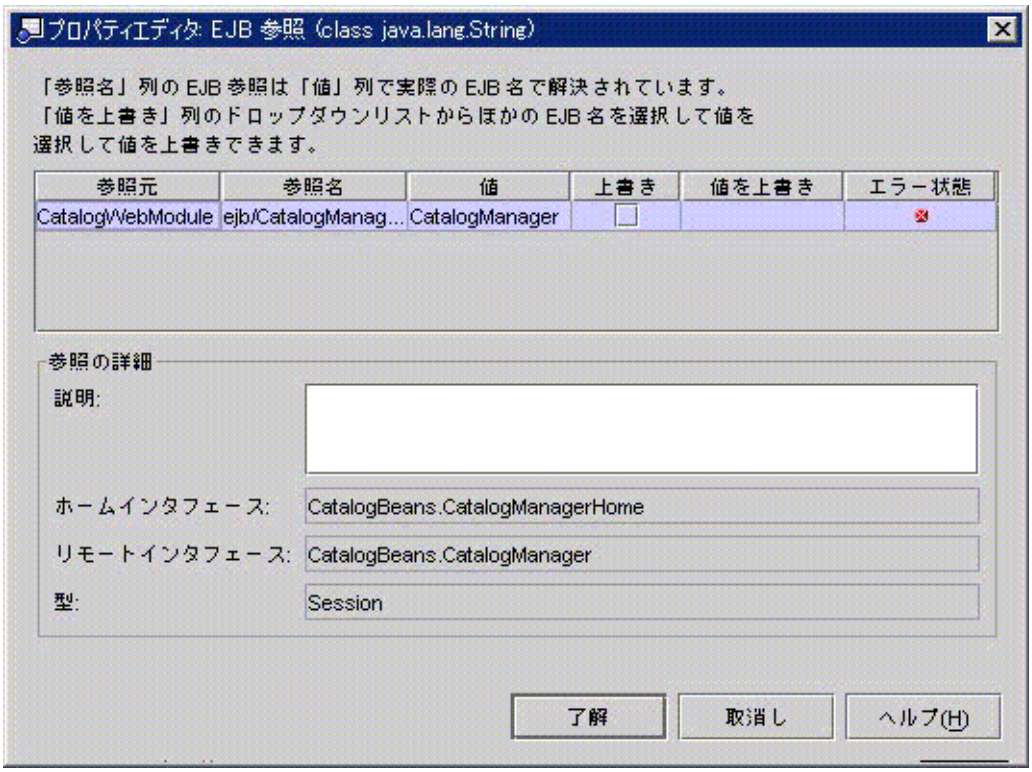


図 4-4　リンクされていない EJB 参照

3. 解決されていない参照をがあればリンクします。

   それには、「値を上書き」フィールドをクリックします。参照で指定されているインタフェースを実行するアプリケーション内のエンタープライズ Bean のリストが表示されます。これらのエンタープライズ Bean の 1 つを選択します。ここで選択したエンタープライズ Bean は、アプリケーションの実行時、Web モジュールでコーディングされているメソッド呼び出しによって呼び出されます。

## その他のアセンブル作業

この章のシナリオで使用しなかったアセンブル作業のいくつかを取り上げます。J2EE アプリケーションを実際に作成するときに役立ててください。

### 環境エントリのオーバーライド

アプリケーションに環境エントリが含まれる場合は、それらに設定された値をモジュールのプロパティシートで変更できます。この作業は、アプリケーションの「環境エントリプロパティ」エディタで行うことができます (このエディタを開くには、アプリケーションノードを右クリックし、コンテキストメニューの「プロパティ」>「環境エントリ」> 省略符号ボタン (...) を選択します)。

このエディタには、アプリケーションを構成するモジュールで宣言されているすべての環境エントリが示されます。環境エントリは、名前 (JNDI lookup 文で使用されている名前) とモジュールによって識別されます。

図 4-5 に、このシナリオのモジュールの「EJB 参照プロパティ」エディタを示します。1 つの環境エントリがあります。このエントリは、Web モジュールのプロパティシートで作成されたものです (38 ページの「環境エントリ参照の設定」 を参照してください)。この環境エントリは NumberOfRecordsCached という名前で、Web モジュールのプロパティシートでデフォルト値の 100 に設定されています。

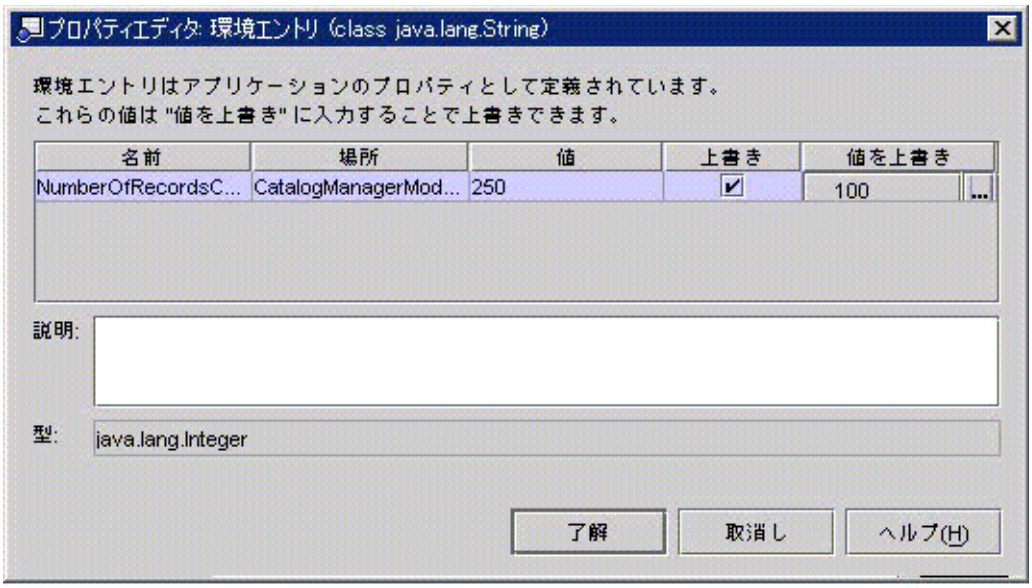


図 4-5　アプリケーションノードの「環境エントリプロパティ」エディタ

開発環境で Web モジュールのソースファイルを扱っている場合は、この値をモジュールのプロパティシートで変更できます。ただし、Web モジュールを複数のアプリケーションで使用する可能性がある場合は、この値をアプリケーションのプロパティシートでカスタマイズした方がいいでしょう。そうすれば、同じ Web コンポーネントを次に使用する開発者がモジュールプロパティシートで値を変更した場合、アプリケーションを再配備するときに新しい値が反映されるからです。

モジュールのプロパティシートで設定された値をオーバーライドするには、「上書き」列のチェックボックスをクリックし、「値を上書き」フィールドにオーバーライド値を指定します。

## 配備記述子の表示と編集

通常は、モジュールおよびアプリケーションのプロパティシートを使用して、配備記述子の内容を制御します。プロパティを設定して、配備記述子の内容を設定します。IDE を使用すると、モジュールおよびアプリケーションの実際の XML 配備記述子を表示できます。

### 配備記述子の表示

J2EE アプリケーション、取り込まれた Web モジュール、および取り込まれた EJB モジュールの配備記述子を表示できます。配備記述子を表示するには、ノードを右クリックし、コンテキストメニューの「配備記述子の表示」を選択します。配備記述子はソースエディタに読み取り専用モードで表示されます。

### EJB モジュール配備記述子の編集

EJB モジュール配備記述子は編集できます。EJB モジュールを右クリックし、コンテキストメニューの「配備記述子を編集」を選択します。

# 第5章

# シナリオ： Web モジュールとキューモードのメッセージ駆動型 Bean

図 5-1 は、Web モジュールフロントエンドとビジネスロジック付き EJB モジュールとから構成される J2EE アプリケーションです。このアプリケーション機能の特徴は、モジュール間の非同期通信です。この通信は Web モジュールによって開始され、EJB モジュールによって処理されます。

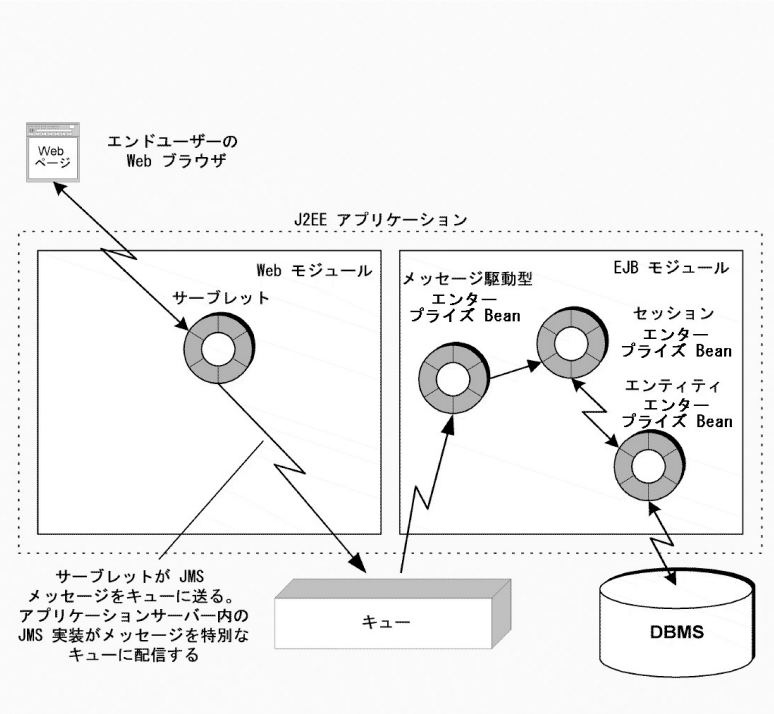


図 5-1　キューモードのメッセージ駆動型 Bean を含む J2EE アプリケーション

## アプリケーション内での対話

この非同期通信を実現するためのプログラミングを考える前に、アプリケーションが非同期通信をビジネスロジックにどう利用するかについて考えてみましょう。このアプリケーションで行われる主な対話は、次のとおりです。

1. ネットワークユーザーがショッピングサイトを訪れ、Web モジュールによって提供されているフロントエンドと対話します。ユーザーが商品リストを検索し、Web モジュールによって管理されるオンラインショッピングカート (買い物かご) に商品を追加します。この操作により、モジュール内の多くの静的な HTML ページ、サーブレット、JSP ページが開きます。これらのコンポーネントの中には、特に持続データの取得と記録を行うために EJB モジュールのビジネスメソッドを呼び出すものもあります。そのすべての操作が同期して行われます。つまり、ユーザーがある情報を要求した後、アプリケーションが情報を返すまで次の作業を開始しません。このようなアプリケーション論理は第 2 章の Web モジュールで紹介しています。

2. 最終的に、ユーザーは、開発者が非同期に処理したい操作を行います。その代表的な例として、チェックアウト手続きの完了があります。この時点で、ユーザーはショッピングカードの内容の確認、配送方法の選択、クレジットカード番号の入力をすでに済ませています。アプリケーションは注文処理を完了した後、電子メールによって、商品の在庫があり、発送されることをユーザーに通知します。

3. Web モジュールは、顧客と注文を識別するメッセージをメッセージキューに送信することで、この非同期処理を開始します (実際の顧客と注文の詳細は注文データベースにあります)。メッセージキューはアプリケーションの外部にあり、アプリケーションサーバーによって保持されます。

4. メッセージ駆動型エンタープライズ Bean がこのメッセージをキューから読み取ります。コンテナがキューからメッセージを取り出し、メッセージ駆動型 Bean の onMessage メソッドを呼び出すことによってメッセージ駆動型 Bean にメッセージを中継します。コンテナはメッセージを onMessage メソッドのパラメータの 1 つとして渡します。

5. メッセージ駆動型 Bean は、注文を処理するビジネスロジックのすべてを含んでいるわけではありません。メッセージを調べ、必要な処理を開始するだけです。このシナリオでは、メッセージ駆動型 Bean がモジュール内のほかのエンタープライズ Bean のビジネスメソッドを呼び出すことで処理を開始します。これは、通常使用

される手法です。EJB モジュールの開発者なら、メッセージ駆動型 Bean のほかに、実際の処理を行うセッションエンタープライズ Bean とエンティティエンタープライズ Bean を開発する (ビジネスメソッドを書く) 方法もあるでしょう。

このアプリケーションを作成するには、そのほかにも、Web モジュールリソースへの URL のマッピング、セッションエンタープライズ Bean とエンタープライズ Bean 間の対話のプログラミングなどが必要になります。それらについては、それぞれの話題を扱っている章を参照してください。

## メッセージ駆動型通信のプログラミング

ここまでは、キューベースのメッセージ駆動型通信をビジネスアプリケーションでどのように使用できるかを見てきました。今度は、その通信を機能させるためにどのようなプログラミングが必要かを見ていきましょう。

表 5-1　このアプリケーションに必要なプログラミング

| アプリケーション要素 | 必要な設定 |
|---|---|
| アプリケーションサーバー | キューとキュー接続ファクトリを設定します。この作業は、IDE の外部で、アプリケーションサーバーの管理ツールを使用して行います |
| Web モジュール | メッセージを送信する Web コンポーネント上で、キューとキュー接続ファクトリへの参照を宣言します。コンポーネントのソースで、JNDI ルックアップを使用してキューへの参照を取得するコードを書きます。また、メッセージを書式化して送信するコードも書きます |
| メッセージ駆動型エンタープライズ Bean が組み込まれた EJB モジュール | メッセージ駆動型 Bean で、その Bean をキューおよびキュー接続ファクトリの宛先にする参照を設定します。Bean の onMessage メソッドのコードを書きます |

この後、これらの項目についてシナリオを使用して詳細に説明していきます。

## アプリケーションサーバーの設定

メッセージ駆動型アプリケーションのフロントエンドがメッセージキューにメッセージを送信します。ほとんどのビジネスロジックを持つバックエンドが、キューからメッセージを読み取ります。キュー自体は、アプリケーションによって作成され、保持されます。本番環境では、システム管理者がキューの定義、設定、管理を行うのが一般的です。

### キューの設定

アプリケーションサーバーのデフォルトキューを利用できますが、メッセージの回線争奪が発生しないように、アプリケーション専用のキューを作成するといいでしょう。

開発環境またはテスト環境では、キューを作成して、管理できます。ここでは、必要な設定の例として、J2EE リファレンス実装の管理ツールを使用してリファレンス実装サーバーにキューを追加する手順を示します。

1. コマンドラインから J2EE RI 管理ツールを使用します。このときの作業ディレクトリは `<j2sdkee1.3_home>/bin` です。
2. 次のコマンドを実行してキューを追加します。

   ```
   j2eeadmin -addJMSDestination jms/MyQueue queue
   ```

3. 次のコマンドを実行してキューが追加されたことを確認します。

   ```
   j2eeadmin -listJmsDestination
   ```

4. 次のコマンドで RI サーバーを起動します。

   ```
   j2ee -verbose
   ```

   起動メッセージにより、`jms/MyQueue` の存在が示されます。

   アプリケーションを本番環境または管理テスト環境に配備するときは、このキューとキューのファクトリ参照が、システム管理者によって指定されているキューにリンクするように設定できます。

### キュー接続ファクトリの設定

キューを使用するためには、アプリケーションでキュー接続を開く必要があります。それには、キュー接続ファクトリのメソッドを呼び出します (キュー接続ファクトリは、メッセージングシステムのドライバです。アプリケーションが接続ファクトリの JMS API メソッドを呼び出すと、インストールされている JMS API の特定の実装に合わせて接続ファクトリがメソッドを解釈します。各アプリケーションサーバーは JMS API のためのキュー接続ファクトリを独自に持っていることがよくあります)。

アプリケーションサーバーは、開発またはテストに適したデフォルトのキュー接続ファクトリを持っている場合があります。たとえば、J2EE 参照実装には、jms/QueueConnectionFactory というデフォルトキュー接続ファクトリが付属しています。これは開発とテストに適しているなので、このシナリオのコードでも J2EE リファレンス実装に付属する接続ファクトリを使用しています。

本番環境では、実際の環境とセキュリティの要件に合わせてシステム管理者が設定し、管理しているキュー接続ファクトリが使用されることになります。アプリケーションを本番環境または管理テスト環境に配備するとき、システム管理者が指定するキュー接続ファクトリを使用するように設定できます。

## Web モジュールのプログラミング

このシナリオでは、Web モジュール内のサーブレットによってメッセージが送信されます。サーブレットはメッセージを送信すると、アプリケーション用に指定されているキューとキュー接続ファクトリを使用する必要があります。そのため、JNDI ルックアップを使用してアプリケーションサーバー環境からキューとキュー接続サーバーへの参照を取得します。

ほとんどの J2EE 参照と同様、これらのキューとキュー接続ファクトリへの参照は、Web モジュールの配備記述子の宣言済み参照とサーブレット内の JNDI ルックアップのコードから構成されます。ここではまず、宣言済み参照について取り上げ、参照を使用するコードの作成方法を説明します。

### キューの参照宣言

Sun ONE Studio 4 IDE では、参照宣言はサーブレットのプロパティとして設定されます。キュー参照は、Web モジュールのリソース環境プロパティエディタで設定されるリソース環境参照です。図 5-1 に、このシナリオで使用されている値を示します。


図 5-2　キューのリソース環境参照

ここには、間接参照があります。このプロパティエディタの「標準」タブで設定されているのは参照の名前であり、実際の JNDI 名ではありません。JNDI 名は、プロパティエディタのサーバー固有タブで設定されます。図 5-3 では、参照名「QueueName」が JNDI 名 「MyQueue」にマップされています。80 ページの「アプリケーションサーバーの設定」 をもう一度読むと、これが J2EE RI で作成され、このアプリケーションのために指定されたキューであることが分かるでしょう。サーブレットの JNDI コードが「QueueName」のルックアップを実行するとき、この名前が自動的に JNDI 名 「MyQueue」にマップされ、そのキューへの参照が「追加リソース環境参照」によってを返されます。

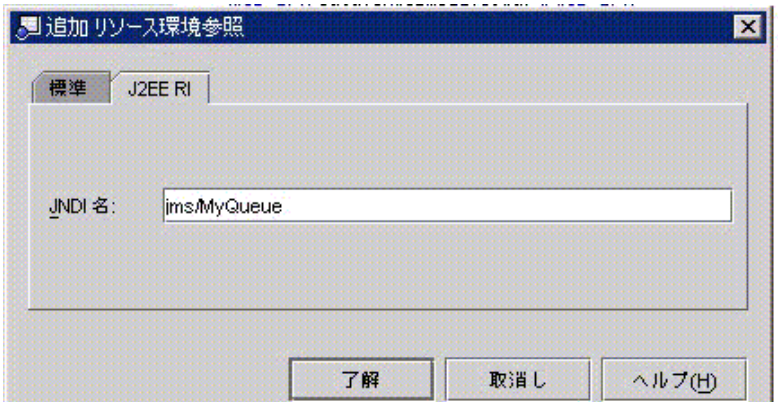


図 5-3　キュー参照の JNDI 名

## キュー接続ファクトリの参照宣言

キュー接続ファクトリの参照もキュー参照と同じような方法で宣言します。これはリソース参照であるため、サーブレットのリソース参照プロパティエディタで設定します。図 5-4 にはこのシナリオで使用されている値が示されています。

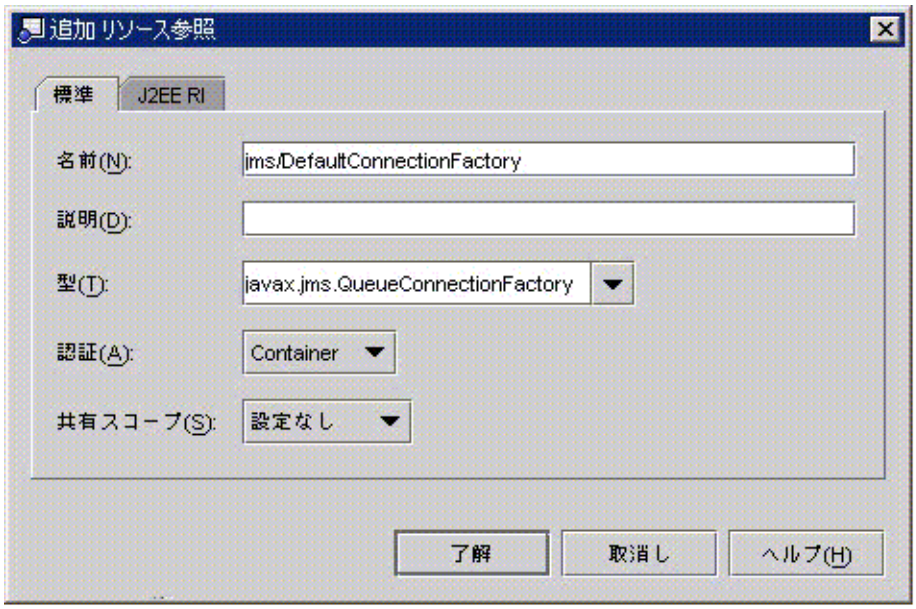


図 5-4　キュー接続ファクトリのリソース参照

ほかの承認の種類については『Enterprise JavaBeans コンポーネントのプログラミング』のメッセージ駆動型 Bean に関する記述を参照してください。

この参照は、キュー参照と同じ間接参照を使用します。図 5-5 では、「defaultconnectionfactory」参照が、 J2EE RI のデフォルトキュー接続ファクトリの JNDI 名にマップされています。

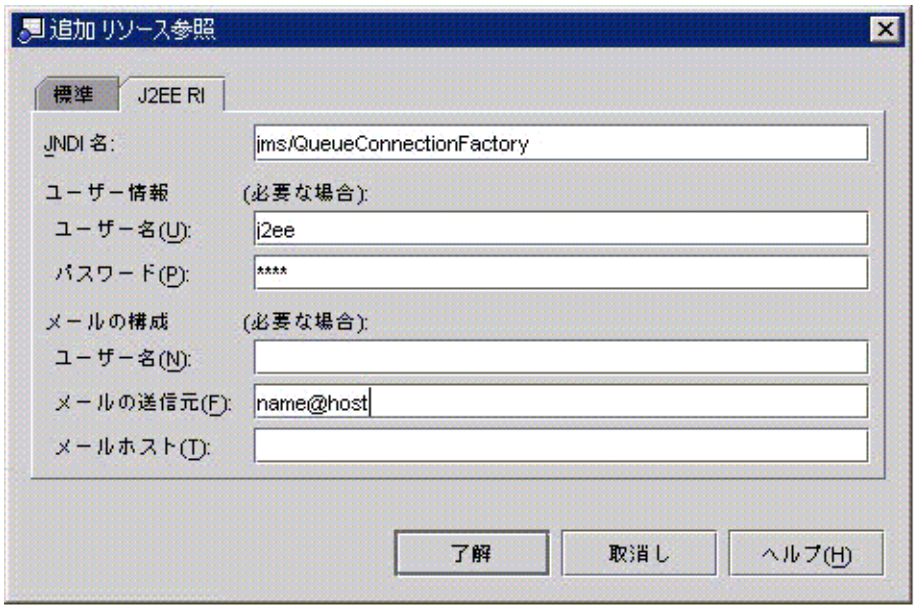


図 5-5　キュー接続ファクトリ参照の JNDI 名

## JNDI ルックアップのコード

ほかの種類の J2EE 参照と同様、JMS キューの宣言済み参照と JMS 接続ファクトリもアプリケーションのコードの中で使用されます。アプリケーションは JNDI ルックアップを実行し、宣言済み参照に名前があるオブジェクトへの参照を取得し、その参照を使用してサービスを要求します。このシナリオでは、Web モジュール内のサーブレットが JNDI ルックアップを実行し、キューとキュー接続ファクトリへの参照を使用してキューへの接続を開いて、キューにメッセージを送信します。

コード例 5–1 に、JNDI ルックアップを実行し、メッセージを作成して送信するコードを示します。これはサーブレットの processRequest メソッドです。このコードには、実行される各操作を示すコメントが挿入されています。

どのようなクライアントも、同じ操作を実行するときは同じようなコードを使用します。同様のコードは、アプリケーションクライアントや、メッセージプロバイダとして機能するエンタープライズ Bean でも使用します。

コード例 5–1　サーブレットの processRequest() メソッド

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;

...

protected void processRequest(HttpServletRequest request, HttpServletResponse
response) throws ServletException, java.io.IOException {

    // デフォルトのメソッド本体を削除し、次の行を挿入する

    response.setContentType("text/html");
    java.io.PrintWriter out = response.getWriter();
    Context jndiContext = null;
    javax.jms.TextMessage msg = null;
    QueueConnectionFactory queueConnectionFactory = null;
    QueueConnection queueConnection = null;
    QueueSession queueSession = null;
    Queue queue = null;
    QueueSender queueSender = null;
    TextMessage message = null;

    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet</title>");
    out.println("</head>");
    out.println("<body>");

    try {
        // アプリケーションによって管理されているデフォルトのネーミングサービスに接続する
        jndiContext = new InitialContext();
    }
&slq;catch (NamingException e) {
        out.println("Could not create JNDI " + "context: " + e.toString());
    }

    try {
        // デフォルトのキュー接続ファクトリとこのシナリオで作成したキューについて
        // JNDI ルックアップを実行する
```

コード例 5–1　　サーブレットの processRequest() メソッド ( 続き )

```
import java.io.*;
        // ここでは、汎用の参照名が使用されている
        queueConnectionFactory = (QueueConnectionFactory) jndiContext.lookup
("java:comp/env/jms/DefaultConnectionFactory");
        queue = (Queue) jndiContext.lookup("java:comp/env/jms/QueueName");
    }
    catch (NamingException e) {
        out.println("JNDI lookup failed: " + e.toString());
    }

    try {
        // 参照を使用してキューに接続し、メッセージを送信する
        queueConnection = queueConnectionFactory.createQueueConnection();
        queueSession = queueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
        queueSender = queueSession.createSender(queue);
        message = queueSession.createTextMessage();
        message.setText("Hello World!");
        queueSender.send(message);
        }
    }
    catch (JMSException e) {
        out.println("Exception occurred: " + e.toString());
    }
    finally {
        if (queueConnection != null) {
            try {
                queueConnection.close();
            }
            catch (JMSException e) {}
        }
    } // finally の終わり
    // 挿入するコードの終わり


} // end of processRequest()
```

メッセージの作成と送信の詳細については、『Enterprise JavaBeans コンポーネントのプログラミング』 を参照してください。

## EJB モジュールの作成

このシナリオでは、利用者のチェックアウト要求を処理するためのビジネスロジックが EJB モジュールにあります。このロジックは、Web モジュールのフロントエンドからのメッセージによって開始されます。このロジックが機能するためには、Web モジュールがキューに送信するメッセージを、EJB モジュールが受信する必要があります。

それには、メッセージをキューから読み取るメッセージ駆動型エンタープライズ Bean を作成します。ここでは、指定されたキューから読み取りを行うメッセージ駆動型 Bean のプログラミング方法を示します。アプリケーションが配備されるとき、Bean に指定したキュー接続ファクトリがコンテナによって使用され、指定したキューへの接続が開かれます。キューとキュー接続ファクトリは、参照によって指定します。

### 「Message Driven Destination」プロパティ

最初に、キューからメッセージを読み取るようにメッセージ駆動型 Bean を設定する必要があります (トピックを宛先とするメッセージを受信する方法あります)。Sun ONE Studio 4 IDE では、この作業を Bean のプロパティシートで行います。図 5-6 に、このシナリオで使用されているメッセージ駆動型 Bean のプロパティシートを示します。「Message Driven Destination」プロパティでは、この Bean をキューのコンシューマとして使用しています。


図 5-6　メッセージ駆動型 Bean のプロパティシート

## キュー参照とキュー接続ファクトリ参照

キュー接続を開くために使用するキューとキュー接続ファクトリを識別する必要もあります。Sun ONE Studio 4 IDE では、使用するアプリケーションサーバーのサーバー固有タブで設定します。図 5-7 は、同じメッセージ駆動型 Bean のJ2EE RI プロパティタブに切り替えた表示です。「接続ファクトリ名」プロパティはデフォルトの J2EE 接続ファクトリの JNDI 名 (jms/QueueConnectionFactory) に設定され、「送信先 JNDI 名」プロパティはこのアプリケーションのために作成した J2EE RIキューの JNDI 名 (jms/MyQueue) に設定されています。

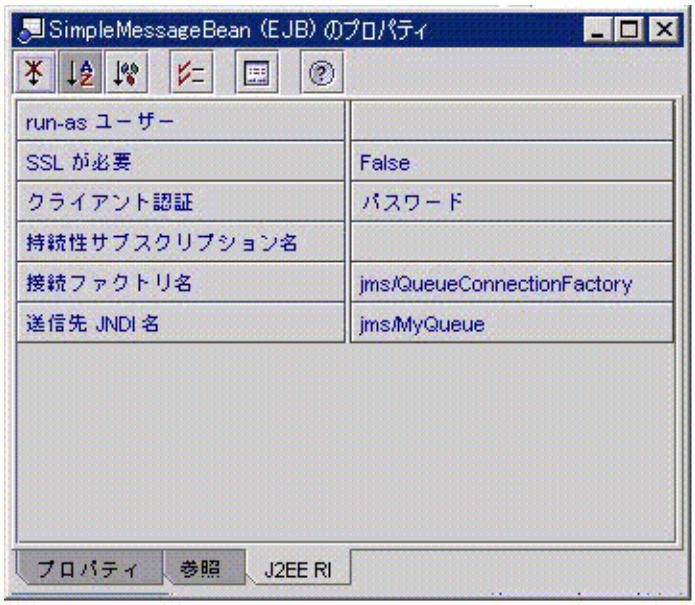


図 5-7　メッセージ駆動型 Bean の「J2EE RI Property」タブ

このメッセージ駆動型 Bean が配備されるとき、指定したキューへの接続が、指定したキュー接続ファクトリを使用して IR コンテナによって開かれます。

ほかのアプリケーションサーバーのタブにも、キューおよびキュー接続ファクトリを指定するための同じようなプロパティがあります。

## onMessage メソッド

メッセージ駆動型 Bean にビジネスロジックを追加するには、その onMessage メソッドを完成させます。このメソッドは、コンテナがメッセージ駆動型 Bean にメッセージを配信するとき自動的に呼び出されます。この例では、メッセージ駆動型 Bean がモジュール内の別のエンタープライズ Bean の適切なビジネスメソッドを直ちに呼び出します。これは、onMessage の通常の動作です。onMessage メソッドの詳しい作成方法については、『Enterprise JavaBeans コンポーネントのプログラミング』 を参照してください。

## J2EE アプリケーションのアセンブル

図 5-1 には、J2EE アプリケーションのサーブレット (Web モジュール内) とメッセージ駆動型 Bean (EJB モジュール内) の両方が示されています。これらのモジュールを組み合わせて 1 つのアプリケーションにするには、アプリケーションを作成し、両方のモジュールを追加するだけです。2 つのモジュールには、メッセージ駆動型通信の配備と実行に必要なすべての情報が含まれています。J2EE アプリケーションのプロパティシートを開いたり、ほかのアセンブル作業を行なったりする必要はありません。

アプリケーションの作成とモジュールの追加の詳細については、65 ページの「J2EE アプリケーションのアセンブル」を参照してください。

# 第6章

# シナリオ：J2EE アプリケーションクライアントと J2EE アプリケーション

図 6-1 に、もう 1 つの J2EE アプリケーションを示します。これまでのシナリオで紹介したアプリケーションと同様に、このアプリケーションは EJB モジュールに重要なビジネスロジックを持っています。ただし、これまでの例では Web モジュールのフロントエンドを使用しましたが、この例は J2EE クライアントアプリケーションによって提供されるフロントエンドを使用します。

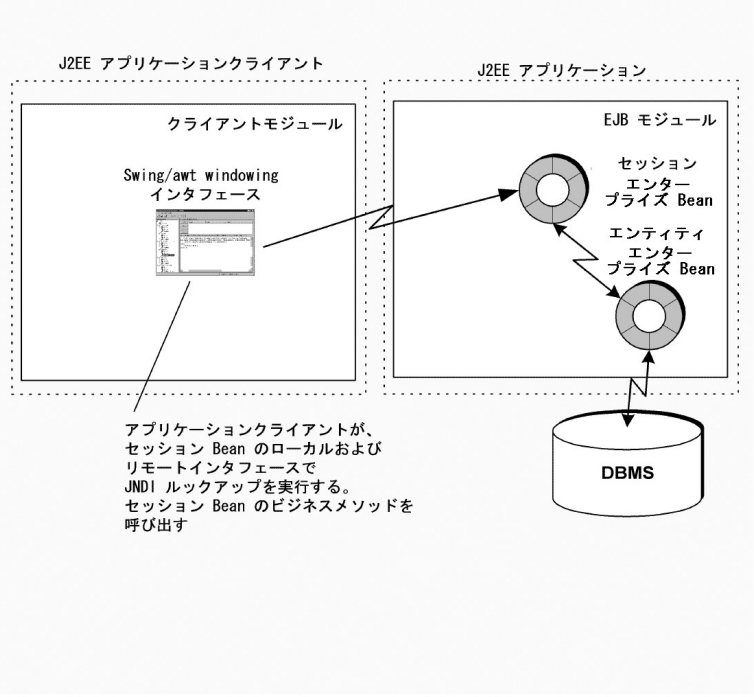


図 6-1　J2EE アプリケーションとアプリケーションクライアント

これまでのシナリオでは (第 4 章と第 5 章を参照)、Web モジュールのフロントエンドと具体的なビジネス論理を備えた EJBモジュールを組み合わせて J2EE アプリケーションを作成し、アプリケーションを 1 つの単位として配備しました。エンドユーザーは Web ブラウザを使用し、Web モジュールに定義された Web ページにアクセスしました。

J2EE アプリケーションも同様にユーザーインターフェースを提供しますが、Web モジュール以上にプログラミングされて、配備され、実行されます。アプリケーションクライアントは、独自の main メソッドを持つ独立した Java プログラムです。さらに、アプリケーションクライアントは、サーバー側の J2EE アプリケーションとは別に配備されます。ほとんどの場合、クライアントはエンドユーザーのマシンに配備されます。クライアントは独自の main メソッドを持つ独立したプログラムであるため、エンドユーザーはサーバー側 J2EE アプリケーションとは別にアプリケーションクライアントを起動したり、停止したりできます。

アプリケーションクライアントは J2EE クライアントコンテナの中で実行されます。つまり、サーバー側 J2EE アプリケーションとは別に配備され、実行されますが、Java RMI を使用してサーバー側アプリケーションのビジネスメソッドを呼び出すことができます。サーバー側のビジネスメソッドを呼び出すときは、J2EE のコンテナ管理によるトランザクションとコンテナ管理によるセキュリティに参加できます。

## アプリケーション内での対話

このシナリオでは、図 6-1 の対話を含む J2EE アプリケーションについて考えます。このアプリケーションは、アプリケーションクライアントがどのように使用され、J2EE のトランザクションとセキュリティにどのように参加するかを示します。また、アプリケーションクライアントによって解決できるアプリケーション設計上の問題の一例でもあります。

これまでのほとんどのシナリオでは、アプリケーションはオンラインショッピングカタログであり、エンドユーザーはオンラインショッピングの利用者 (買い物客) で、Web モジュールのフロントエンドを介してオンラインカタログにアクセスしました。

しかし、オンラインカタログアプリケーションのエンドユーザーは、買い物客だけではありません。オンラインカタログの商品は頻繁に変更されます。商品が追加されたり、削除されたり、セール対象になったりします。これらの作業は、買い物客ではなく、オンライン販売会社の従業員であるアプリケーション管理者が行います。

これらの管理者によって行われる作業は、買い物客の作業よりも複雑です。たとえば、カタログに新しい商品を追加するときは、SKU をキーにした製品レコードの設定、オンラインカタログに表示する画像とテキストの設定、買い物客が製品の検索に使用するキーワードの設定などの作業が必要です。

通常、買い物客はアプリケーションとすばやく対話し、カタログの新しいページを要求したり、特定の商品の詳細情報を要求したりします。一方、アプリケーション管理者はそれとは異なる方法でアプリケーションと対話します。通常は長い時間をかけてカタログエントリを操作し、エントリを保存します。アプリケーションクライアントがサーバー側アプリケーションのサービスを必要とするのは、カタログエントリを保存するときだけです。そのため、アプリケーション管理者は、サーバー側ロジックから独立して実行されるビジネスロジックを利用できます。

アプリケーション管理者によって行われるこうした作業を考えれば、オンラインカタログの管理には、独自のビジネス論理を備えたアプリケーションクライアントの方が WEb モジュールインターフェスより適していると判断できます。このアプリケーションの実行時には、主に、次の対話が行われます。

1. データベースの実際の挿入、更新、削除を行うのは、サーバー側アプリケーションです。こちらは、セッションエンタープライズ Bean とエンティティエンタープライズ Bean を持つ J2EE アプリケーションです。サーバー側アプリケーションは、オンラインショッピングの利用者と同じデータベースにアクセスします。このアプリケーションは、通常、システム管理者によって管理され、アプリケーション管理者がアプリケーションクライアントを起動する前に起動され、稼働しています。

2. アプリケーション管理者は、何らかのカタログ管理作業を実施する場合はアプリケーションクライアントを起動します。アプリケーションクライアントは、管理者のマシン上で、独自のプロセスで実行されます。J2EE アプリケーションクライアントとして配備されているため、J2EE クライアントコンテナの中で実行されます。アプリケーション管理者はログインし、セキュリティアイデンティティを確立します。

3. アプリケーション管理者は、テキストと画像を使用して新しいカテゴリエントリを作成します。管理者が新しいカテゴリエントリを完成させて「保存」ボタンをクリックします。すると、アプリケーションクライアントが JNDI ルックアップを使用してサーバー側アプリケーションのセッション Bean へのリモート参照を取得し、Java RMI を使用して、新しいエントリをオンラインカタログデータベースに挿入するビジネスメソッドを呼び出します。この操作により、複数のテーブルに挿入が行われます。

ビジネスメソッドが呼び出されると、アプリケーション管理者のセキュリティアイデンティティがサーバー側 J2EE アプリケーションに渡されます。管理者が承認されたユーザーであることをサーバー側 EJB コンテナが確認すると、ビジネスメソッドが実行され、新しいカタログエントリが挿入されます。このエンタープライズ Bean のビジネスメソッドはトランザクション型であるため、EJB コンテナは新しいトランザクションを開始し、すべてのデータベース挿入操作が正常に完了すると、トランザクションをコミットします。

このアプリケーションには、セッションエンタープライズ Bean とエンティティエンタープライズ Bean 間の対話のプログラミングなども必要になります。それらについては、それぞれの話題を扱っているシナリオを参照してください。

# このアプリケーションのプログラミング

表 6-1 は、図 6-1 で示した J2EE アプリケーションとアプリケーションクライアントの開発に必要なプログラミング作業をまとめています。

表 6-1　このシナリオに必要なプログラミング

| アプリケーション要素 | 必要なプログラミング作業 |
|---|---|
| アプリケーションサーバー | なし |
| J2EE アプリケーションクライアント | まず、必要な機能を提供するスタンドアロン Java クライアントプログラム (main メソッドを持つもの) を書きます。プログラムフローのサーバー側ビジネスメソッドを呼び出すポイントで、ホームオブジェクトとリモートオブジェクトのリモート参照を取得する JNDI ルックアップを書いた後、リモート呼び出しを書きます。<br>Java クライアントプログラムを書いた後、それを Java アプリケーションクライアントに変更します。JDNI ルックアップで指定されているエンタープライズ Bean の宣言済み参照を設定します。JNDI ルックアップに必要なスタブクラスを識別します。 |
| J2EE アプリケーション (サーバー側) | 特別なプログラミングは必要ありません。サーバー側アプリケーションは、クライアントが完成する前に配備する必要があります。配備によってクライアントに必要なスタブクラスが生成されるからです |

この後、単純な例を使用してこれらのプログラミング作業のそれぞれを説明していきます。

## J2EE クライアントアプリケーションのプログラミング

このシナリオでは、アプリケーションクライアントが Java RMI を使用してサーバー側アプリケーションのエンタープライズ Bean と通信します。この後、このシナリオに合った非常に単純なプログラムの作成方法を説明します。

説明は、大きく 2 つの部分に分かれています 前半は、クライアントプログラムの Java コードの書き方を説明します。後半は、プログラムを J2EE アプリケーションサーバーに配備可能なアプリケーションクライアントに変更する方法を説明します。

## クライアントコードの作成

アプリケーションクライアントがほかのスタンドアロン Java プログラムと異なるのは、それが EJB 参照などの J2EE サービスを使用してほかのコンテナ内で実行中の J2EE アプリケーションと対話することです。コード例 6–1 に、いくつかの Swing ウィンドウクラスとエンタープライズ Bean 上で JNDI ルックアップを実行するコードとを組み合わせた非常に単純な Java プログラムを示します。

J2EE のほかの種類の参照と同じように、このアプリケーションクライアントの EJB 参照はルックアップのコードと対応する宣言済み参照とから構成されます。この後の、プログラムを J2EE アプリケーションプログラムに変更するステップでは、参照宣言の設定方法を説明します。ここで重要なのは、ルックアップのコードです。

このルックアップ用コードは、Web モジュールがエンタープライズ Bean のビジネスメソッドを呼び出すときに使用するコードに似ています。J2EE 対話に必要な部分はコメントを付加して強調してあります。

コード例 6–1　A `JFrame` Class With JNDI Lookup Code

```
/*
 * helloClient.java
 *
 * Created on January 29, 2002, 4:56 PM
 */

package HelloClient;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;
// エンタープライズ Bean のホームインタフェースと
// リモートインタフェースの import 文を追加する
import HelloBean.*;

/**
 *
 * @author J2EE Client Developer
 */
public class helloClient extends javax.swing.JFrame {

    // JNDI ルックアップとリモートメソッド呼び出しで使用される変数を
```

コード例 6–1　A JFrame Class With JNDI Lookup Code ( 続き )

```
/*
    // 宣言する。
    HelloHome hHome = null;
    Hello hRemote = null;
    String returnedText = null;

    /** Creates new form helloClient */
    public helloClient() {
        initComponents();
        try {
        // JNDI ルックアップを実行し、エンタープライズ Bean の
        // ホームオブジェクトへのリモート参照を取得する
            Context ic = new InitialContext();
            Object objref = iC.lookup("java:comp/env/ejb/ItemBean");
            System.out.println("get the home");
           hHome = (HelloHome) PortableRemoteObject.narrow(o, HelloHome.class);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    private void initComponents() {
        invokeBeanButton = new javax.swing.JButton();
        returnTextField = new javax.swing.JTextField();

        getContentPane().setLayout(new
org.netbeans.lib.awtextra.AbsoluteLayout());

        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });

        invokeBeanButton.setText("Click Me");
      invokeBeanButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                invokeBeanButtonActionPerformed(evt);
            }
        });
```

コード例 6–1　　A JFrame Class With JNDI Lookup Code ( 続き )

```
/*
        getContentPane().add(invokeBeanButton, new
org.netbeans.lib.awtextra.AbsoluteConstraints(150, 130, -1, -1));

        returnTextField.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                returnTextFieldActionPerformed(evt);
            }
        });

        getContentPane().add(returnTextField, new
org.netbeans.lib.awtextra.AbsoluteConstraints(110, 190, 160, -1));

        pack();
    }

   private void returnTextFieldActionPerformed(java.awt.event.ActionEvent evt)
{
        // ここに処理コードを追加する
    }

    private void invokeBeanButtonActionPerformed(java.awt.event.ActionEvent
evt) {
        // ここに処理コードを追加する
        // リモートメソッド呼び出しを行い、返された値を
        // テキストウィンドウに表示する
        try {
            System.out.println("inside invokeBeanButtonActionPerformed");
            hRemote = hHome.create();
            System.out.println("performed create");
            returnedText = hRemote.sayHello();
            System.out.println("invoked sayHello");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        returnTextField.setText(returnedText);

    }

    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt) {
        System.exit(0);
    }

    /**
```

コード例 6–1　　A JFrame Class With JNDI Lookup Code ( 続き )

```
 /*
      * @param args the command line arguments
      */
     public static void main(String args[]) {
         new helloClient().show();
     }


     // 変数宣言 - 編集不可
     private javax.swing.JTextField returnTextField;
     private javax.swing.JButton invokeBeanButton;
     // 変数宣言の終わり
  }
```

現実のアプリケーションクライアントはこれより複雑になります。クライアントコードの作成が終わったら、J2EE アプリケーションサーバーに配備する前にテストを行う必要があるでしょう。プログラムは、IDE とコマンドラインのどちらからでも、その main メソッドを実行することによって実行できます。ただし、まだ J2EE クライアントコンテナの中で実行されないため、JNDI ルックアップの呼び出しとリモートメソッド呼び出しは異常終了します。テストを行うには、JNDI の呼び出しをコメントアウトし、リモート呼び出しを、プログラムのロジックをテストできるようなダミー値を返すコードに置き換える必要があります。それでも、「GUI コード」とビジネスロジックはテストできます。

この単純な例では、Jframe のコンストラクタに JNDI ルックアップを追加しました。また、エンタープライズ Bean のインスタンスを返す create メソッドと sayHello のリモート呼び出しを、ボタンの actionPerformed メソッドに追加しました。これは、小さなプログラムに J2EE コードを追加する簡単な方法です。J2EE コードをクライアントコードに追加するほかの方法は、後ほど紹介します。このシナリオでは、次に Swign プログラムを J2EE アプリケーションクライアントに変更する方法を説明します。

## J2EE クライアントアプリケーションへの変更

これで、アプリケーションクライアントで必要な機能を提供する Java プログラムが記述されました。この後、Sun ONE Studio 4 IDE を使用して Java プログラムを J2EE クライアントアプリケーションに変更し、設定する手順を説明します。

### クライアントアプリケーションノードの作成とクライアントコードの追加

クライアントアプリケーションノードを作成し、クライアントプログラムに関連付けるには、次のようにします。

1. アプリケーションクライアントを作成するフォルダまたはパッケージを右クリックします。コンテキストメニューの「新規」>「J2EE」>「アプリケーションクライアント」を選択します。

   エクスプローラにアプリケーションクライアントノードが作成されます。

2. アプリケーションクライアントノードを右クリックします。コンテキストメニューの「主クラスを設定」を選択します。ブラウザを使用してクライアントクラスを選択します。「了解」をクリックします。

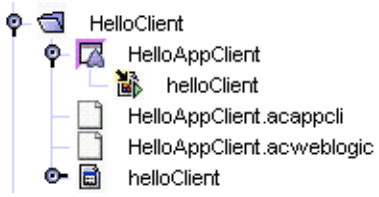


図 6-2　Java クライアントノードのサブノードを持つアプリケーションクライアントノード

IDE には、Java クライアントプログラムを表すアプリケーションクライアントノードの新しいサブノードが表示されます。図 6-2 には、HelloAppClient という名前のアプリケーションノードが示されています。helloClient プログラム (コード例 6–1）がこのアプリケーションクライアントノードの主クラスとして指定され、IDE によって helloClient という名前のサブノードが作成されました。アプリケーションクライアントノードは、配備可能な形式の Java クライアントプログラムを表します。そのプロパティを使用し、クライアントコンテナから特定の J2EE サービスを要求します。この場合、HelloAppClient は配備可能な形式の helloClient プログラムを表します。

### EJB 参照の宣言

アプリケーションクライアントノードを作成し、そのノードにプログラムを関連付けた後、アプリケーションクライアントを設定する必要があります。この場合、エンタープライズ Bean のビジネスメソッドを呼び出す Java クライアントプログラムを書いたため、プログラムコードの JNDI ルックアップと一致する宣言済み参照をアプリケーションクライアントノード上で設定する必要があります。

アプリケーションクライアントの EJB 参照を設定するには、次のようにします。

1. アプリケーションクライアントノードを右クリックします。コンテキストメニューの「プロパティ」を選択します。「参照」タブをクリックして前面に出します。「EJB 参照プロパティ」をクリックし、表示される省略符号ボタン (...) をクリックします。エンタープライズ Bean 参照プロパティエディタが表示されます。「追加」をクリックし、「追加 EJB 参照」ダイアログを開きます。

   図 6-3 に、このダイアログを示します。各フィールドには、コード例 6–1 のプログラムの正しい値が示されています。

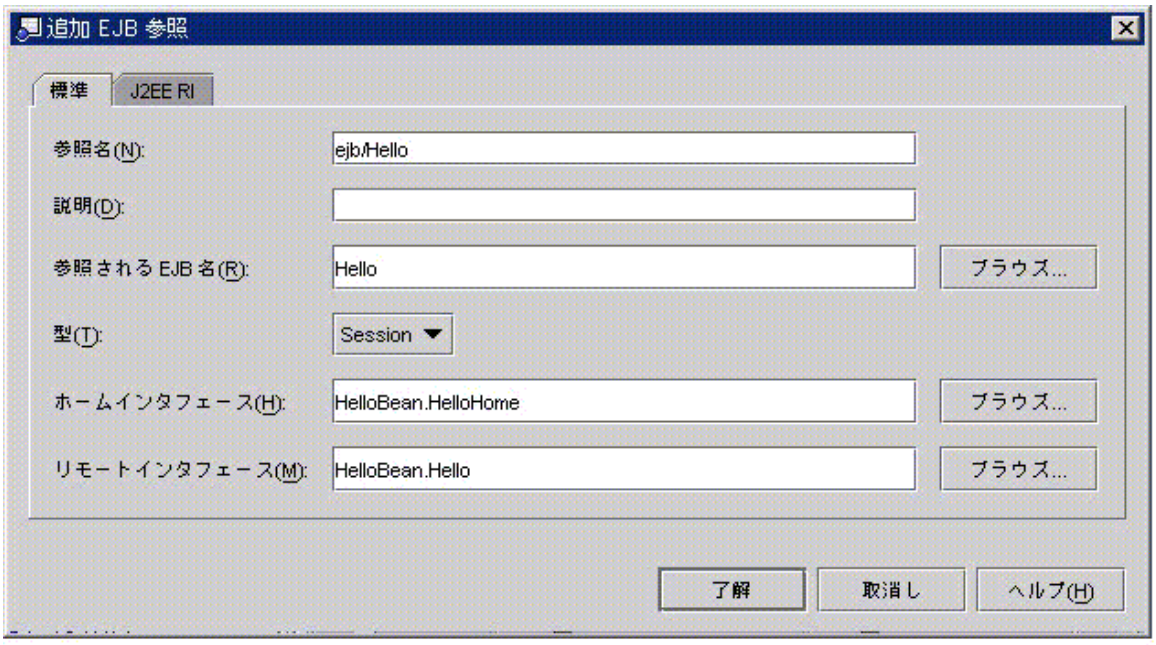


図 6-3　「追加 EJB 参照」ダイアログ

2. 使用中の J2EE アプリケーションサーバーのタブをクリックして前面に出します。

   このタブに表示されるフィールドは、アプリケーションサーバーによって異なります。図 6-4 は、J2EE RI (Reference Implementation) のタブです。このタブには、JNDI 名のフィールドがあります。このフィールドを使用し、アプリケーションクライアントが対話するエンタープライズ Bean を識別します。エンタープライズ Bean は、そのエンタープライズ Bean が配備されたときに指定された JNDI 名によって識別されなければなりません。この例では、サーバー側エンタープライズ Bean が Hello という名前で配備されました。

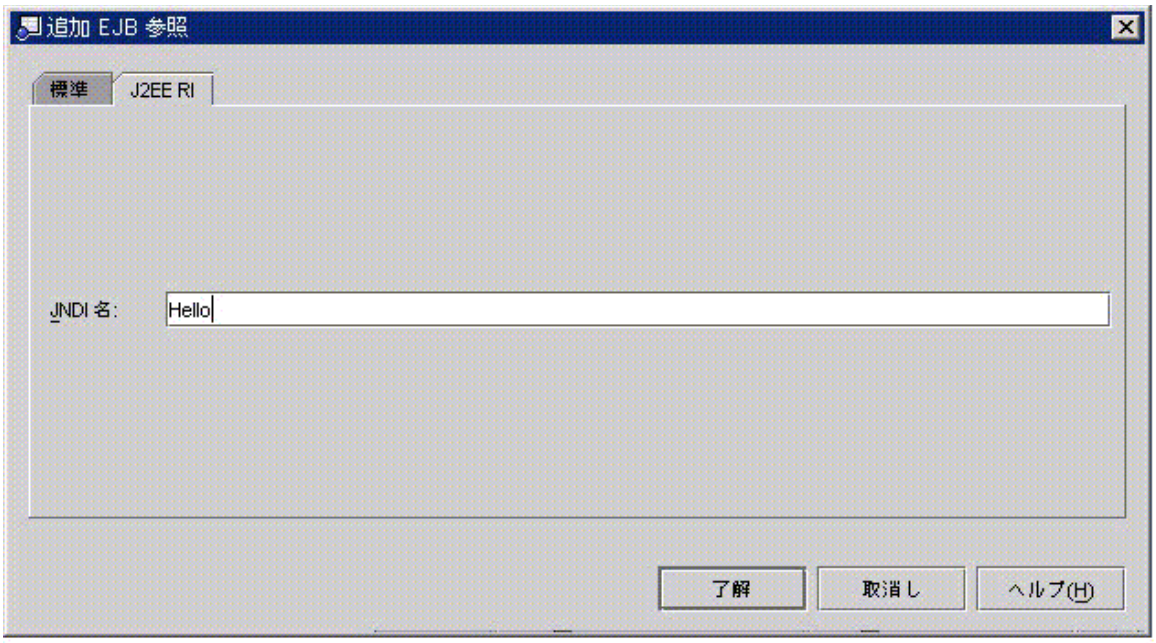


図 6-4　「編集 EJB 参照」ダイアログの J2EE RI 固有タブ

ほかの J2EE アプリケーションサーバーを使用している場合は、「ヘルプ」ボタンをクリックすると、タブ上のフィールドに関するオンラインヘルプが表示されます。

3. 「了解」をクリックして「編集 EJB 参照」ダイアログを閉じ、「了解」をもう一度クリックしてプロパティエディタを閉じます。

### ターゲットアプリケーション (または Client Jar Path) の指定

エンタープライズ Bean の参照のほかに、参照で識別したエンタープライズ Bean 上でのリモートメソッド呼び出しをサポートするスタブファイルのコピーが必要です。これらのファイルはサーバー固有のもので、サーバー側アプリケーションを配備するときに、アプリケーションサーバーの配備ツールによって生成されます。これらは、通常、サーバー側アプリケーションの配備時に生成される JAR ファイルにあります。

WebLogic など、アプリケーションサーバーの中には、アプリケーションクライアントの実行時、アプリケーションクライアントにスタブファイルをダウンロードできるものもあります。そのようなサーバーを使用している場合は、スタブファイルに関して何もする必要はありません。エンタープライズ Bean 参照の設定後、そのままアプリケーションを実行できます。

一方、スタブファイルをダウンロードしないアプリケーションサーバーを使用している場合は、アプリケーションクライアントを実行する前にスタブ JAR ファイルのコピーを提供する必要があります。J2EE RI もスタブファイルをダウンロードしないため、ここでは J2EE RI にスタブ JAR ファイルの存在場所を指定する手順を示します。RI にアプリケーションクライアントを配備する場合は、スタブ JAR ファイルも一緒に配備されます。

サーバー側アプリケーションとアプリケーションクライアントの両方を開発している場合は、おそらく、エクスプローラのアプリケーションクライアントの隣にサーバー側アプリケーションのコピーがマウントされています。その場合は、サーバー側アプリケーションをアプリケーションクライアントのターゲットアプリケーションとして識別すれば、IDE がクライアント JAR ファイルを検索します。図 6-5 の HelloAppClient ノードは、HelloApplication ノードによって表されるサーバー側アプリケーションと同じファイルシステムにあります。

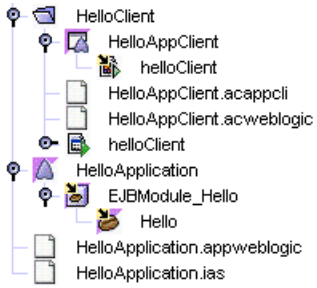


図 6-5　IDE におけるアプリケーションクライアントとサーバー側アプリケーション

「ターゲットアプリケーション」プロパティを設定するには、次のようにします。

1. アプリケーションクライアントを右クリックします。コンテキストメニューの「プロパティ」を選択します。「ターゲットアプリケーション」をクリックし、省略符号ボタン (...) をクリックします。
2. ブラウザを使用し、IDE 内でサーバー側アプリケーションノードを見つけます。ノードを選択し、「了解」をクリックします。

   図 6-6 に、HelloAppClient のプロパティシートを示します。HelloApplication をターゲットアプリケーションとして使用するように設定されています。

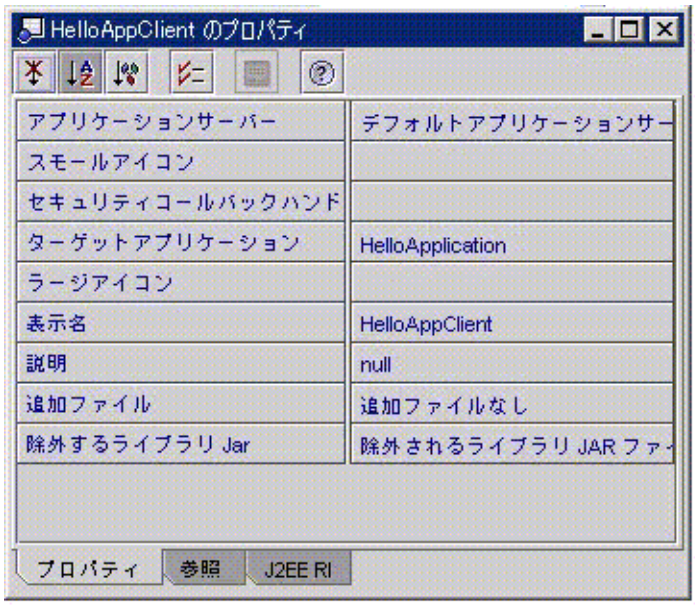


図 6-6　アプリケーションクライアントのプロパティシート

利用するサーバー側アプリケーションがすでに別のマシン上に配備され、開発用マシン上にない場合は、別のプロパティを使用してアプリケーションクライアントに必要なクライアント JAR へのパスを指定します。場合によっては、サーバー側アプリケーションの開発者、またはサーバー側アプリケーションを管理するシステム管理者からクライアント JAR ファイルのコピーを入手することが必要です。

サーバー側アプリケーションが Sun ONE Studio 4 で開発されたものである場合は IDE を使用して必要なスタブ JAR ファイルを作成できます (この作業は、サーバー側アプリケーションの開発者またはシステム管理者が行う場合もあります)。

IDE にマウントされているサーバー側アプリケーションのクライアント JAR ファイルを作成するには、次のようにします。

- J2EE アプリケーションノードを右クリックします。コンテキストメニューの「クライアントサポートをエクスポート」を選択します。これにより、「クライアント .jar の場所を指定」ダイアログが表示されます。このダイアログを使用し、クライアント JAR を置くディレクトリを指定します。

  サーバー側アプリケーションを Sun ONE Studio 4 IDE にマウントできないときは、この方法でクライアント JAR を作成できません。その場合は、サーバー側の開発者 (またはシステム管理者) が、アプリケーションサーバーの配備ツールによって生成されたクライアント JAR のコピーを取得しなければなりません。

  クライアント JAR のコピーを入手したら、それをファイルシステムにマウントした後、クライアントの 「.jar」プロパティを使用し、それをアプリケーションクライアントに対して指定します。

  アプリケーションクライアントのクライアント JAR を指定するには、次のようにします。

- アプリケーションクライアントノードを右クリックします。コンテキストメニューの「プロパティ」を選択します。使用中のアプリケーションサーバーのタブをクリックして前面に出します。クライアント JAR パスを設定するプロパティをクリックし、省略符号ボタン (...) をクリックします。

  図 6-7 は、HelloAppClient ノードの「J2EE RI」タブです。このタブには、「Stab Jar File」というプロパティがあります。このプロパティを使用してスタブ JAR ファイルを指定します。


図 6-7　アプリケーションクライアントの J2DD RI 固有タブ

クライアント JAR ファイルを指定する必要のあるほかのアプリケーションサーバーでも、作業手順はほぼ同じです。まず、クライアント JAR ファイルのコピーを入手します。次に、サーバー固有プロパティタブで、スタブ JAR ファイルへのパスを設定するためのプロパティを見つけます。WebLogic など、一部のアプリケーションサーバーはアプリケーションの実行時にクライアント JAR ファイルをアプリケーションに自動的にダウンロードします。

## アプリケーションクライアントの実行

アプリケーションクライアントを実行する手順は以下のとおりです。

1. アプリケーションサーバーが選択されていない場合は選択します。クライアントアプリケーションノードを右クリックし、コンテキストメニューの「プロパティ」を選択します。プロパティシートで「アプリケーションサーバー」プロパティを見つけます。

   このプロパティの初期値は、「デフォルトアプリケーションサーバー」です。この値のままでアプリケーションクライアントを実行すると、デフォルトサーバーインスタンスとして選択されたアプリケーションサーバーインスタンスによってアプリケー

ションクライアントが実行されます (デフォルトサーバーを調べるには、エクスプローラウィンドウの「実行時」タブに切り替え、「サーバーレジストリ」ノードとその「デフォルトサーバー」サブノードを調べます) 。

また、このプロパティでサーバーインスタンスを指定することもできます。「アプリケーションサーバー」プロパティをクリックし、次に省略符号ボタン (...) をクリックします。プロパティエディタが開きます。

2. アプリケーションクライアントノードを右クリックし、コンテキストメニューの「プロパティ」を選択します。

   この操作により、手順 1 で指定したアプリケーションサーバーによって提供されるクライアントコンテナ内でアプリケーションクライアントが実行されます。

3. 一部のアプリケーションサーバー (J2EE RI を含む) では、この時点でログイン画面が開きます。有効なユーザー名とパスワードを入力します。

   図 6-8 は、J2EE RI のログイン画面です。ここでは、デフォルトの J2EE RI ユーザー名とパスワードを入力しています。

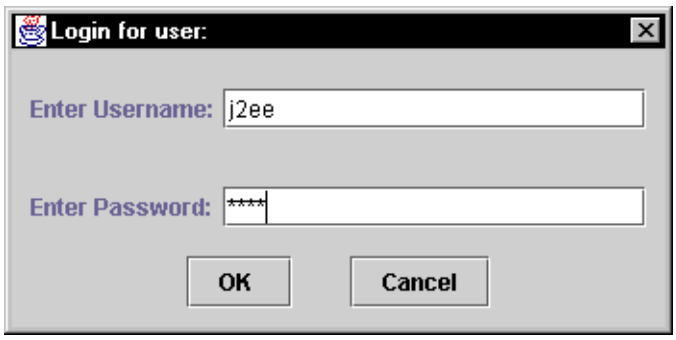


図 6-8 J2EE RI (リファレンス実装) のログイン画面

4. ログインが正常に行われると、アプリケーションサーバーがクライアントプログラムの main メソッドを実行します。

### アプリケーションクライアントの配備

アプリケーションクライアントを配備してコマンドラインから実行したい場合は、クライアント JAR ファイルを作成できます。クライアント JAR ファイルの実行には、クライアントを実行するためのアプリケーションサーバーのツールが必要になります。

クライアント JAR を作成するには、次のようにします。

1. アプリケーションクライアントノードを右クリックし、コンテキストメニューの「Jar をエクスポート」を選択します。表示されるダイアログを使用し、アプリケーションサーバーとクライアント jar ファイルの場所を指定します。
2. アプリケーションサーバーが別のマシンにインストールされている場合は、そのマシンにクライアント JAR ファイルを移動できます。
3. アプリケーションサーバーによって提供されているツールを使用してクライアントを実行します (J2EE RI の場合は、runclient というツールを使用します)。

## サーバー側 J2EE アプリケーションの操作

このシナリオでは、アプリケーションクライアントがサーバー側 J2EE アプリケーションと通信します。図 6-1 のサーバー側アプリケーションは 1 つの EJB モジュールから構成されていますが、ほかの構成も可能です。サーバー側アプリケーションを複数の EJB モジュールから構成することや、アプリケーションクライアントとは異なるエンドユーザー機能を提供する Web モジュールを組み込むこともできます (Web モジュールと EJB モジュールを組み合わせて 1 つの J2EE アプリケーションを作成する手順については、ほかのシナリオで扱っています)。

アプリケーションクライアントがサーバー側アプリケーションにアクセス可能にするために、特別な作業は必要ありません。しかし、アプリケーションクライアントをプログラミングするときは、サーバー側アプリケーションに関する情報が必要です。

- まず、プログラムの通信相手となるすべての J2EE サービスの JNDI 名が必要です。この例では、サーバー側アプリケーションのエンタープライズ Bean の JNDI 名が必要でした。現実のアプリケーションクライアントでは、複数のエンタープライズ Bean の JNDI 名が必要なことがあります。また、JMS キューとキュー接続ファクトリのほか、様々なJ2EE リソースの JNDI 名が必要になることがあります。
- アプリケーションクライアントで Java RMI を使用してサーバー側のエンタープライズ Bean と通信しようとしている場合は、次の 2 つが必要になります。

- エンタープライズ Bean のホームインタフェースとリモートインタフェースのコピー
- 使用中のアプリケーションサーバーによっては、サーバー側アプリケーションの配備時に生成されるスタブファイルのコピーが必要になることがあります。スタブファイルはサーバー固有のものであるため、IDE ではなくアプリケーションサーバーによって、配備動作の一部として生成されなければなりません (サーバー側アプリケーションが Sun ONE Studio 4 IDE にマウントされている場合は、「クライアントサポートをエクスポート」コマンドを使用して`stub.jar`ファイルを生成できます)。スタブファイルは、アプリケーションクライアントの配備先のアプリケーションサーバーによって生成されなければなりません。

本番環境または管理テスト環境では、この情報をシステム管理者から入手することが必要な場合があります。

## J2EE アプリケーションクライアントとのトランザクション

一般に、トランザクションはサーバー側 J2EE アプリケーションの一部であり、トランザクション境界の定義は、サーバー側アプリケーションのプログラミングの中で行います。サーバー側コンテナは、アプリケーションクライアントがトランザクションビジネスメソッドを呼び出すと起動し、トランザクション境界内のすべてのビジネスロジックが正常に完了するとトランザクションをコミットします。

## このシナリオの変更

このシナリオは、アプリケーションクライアントとサーバー側 J2EE アプリケーションとの間で行われる特定の対話のプログラミング方法を説明します。Java RMI を使用してサーバー側 J2EE アプリケーションにサービスを要求するアプリケーションクライアントを示します。また、main() メソッドを持つ swing クラス内の J2EE コードを示します。

### ヘルパークラスの使用法

プログラム例 (コード例 6–1）では、アプリケーションのインタフェースを提供する swing クラス内の J2EE コードを示しました。コードのすべてをこのクラスに入れると、クライアントアプリケーションに必要なすべてのものが見やすくなりますが、現実のアプリケーションクラアントは別の方法でプログラミングしたい場合もあります。

たとえば、ヘルパークラスを利用し、J2EE コードをユーザーインタフェースやビジネスロジックから分離する方法があります。図 6-9 に、ヘルパークラスを使用したクライアントアプリケーションを示します。

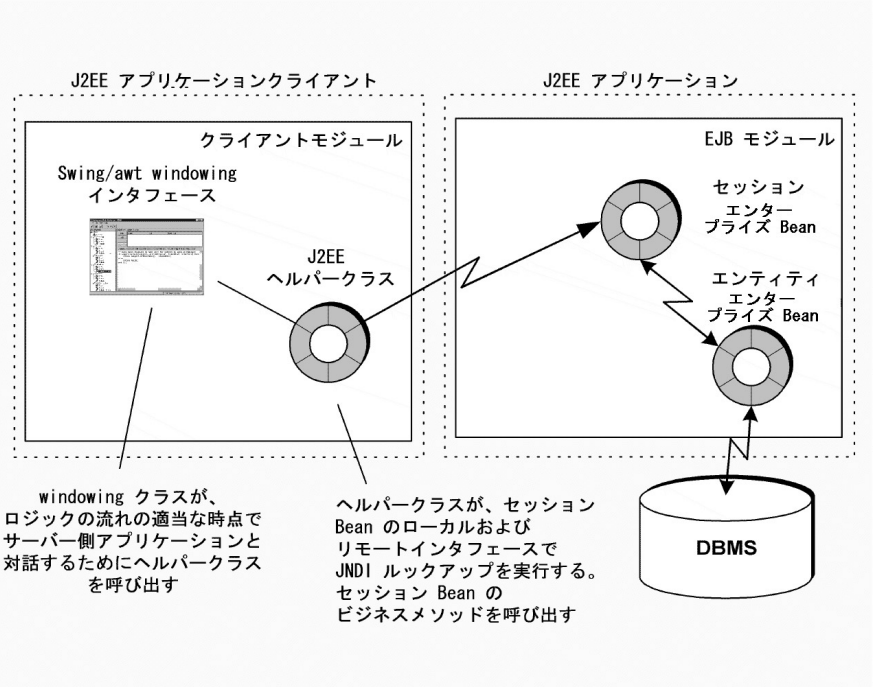


図 6-9　ヘルパークラスを使用したクライアントアプリケーション

このアプリケーションクライアントは、シナリオで説明した、アプリケーション管理者がオンラインカタログの新しいエントリを設定するための機能 (92 ページの「アプリケーション内での対話」) を提供するものとします。ただし、今度のバージョンは、エンドユーザーが保存ボタンをクリックしても、GUI クラス自体は　JNDI ルックアップやリモート呼び出しを実行しません。その代わり、新しいカタログエントリをヘルパークラスに渡します。保存ボタンの actionPerformed メソッドには、次のようなコードが含まれます。

コード例 6-2　ヘルパークラスの使用法

```
...
myHelperClass helper = new myHelperClass();
helper.saveNewEntry(newCatalogEntry);
....
```

ヘルパークラスの saveNewEntry メソッドは、JNDI ルックアップを実行し、必要なサーバー側メソッドを呼び出します。エンタープライズ Bean の参照は、アプリケーションクライアントノードで、同じ方法で設定します。

この方法では、Java クライアントプログラムのテストが簡単になります。ヘルパークラスを、ダミー値を返す別の Java クラスに置き換えるだけです。

### 複数のリモートメソッド呼び出しの組み込み

アプリケーションクライアントの能力は、サーバー側アプリケーションの特定のエンタープライズ Bean との Java RMI を使用した対話に限定されません。複数のエンタープライズ Bean のビジネスメソッドを呼び出すことができます。その場合は、クライアントですべてのエンタープライズ Bean のホームインタフェースとリモートインタフェースをインポートし、すべてのエンタープライズ Bean の JNDI ルックアップ用コードを追加する必要があります。また、アプリケーションクライアントノードにすべてのエンタープライズ Bean の参照を追加する必要があります。

### Java メッセージングの使用法

アプリケーションクライアントがサーバー側アプリケーションと通信する際、Java メッセージングを利用することもできます。その場合は、サーバー側アプリケーションの配備時にキューおよびキュー接続ファクトリに割り当てられた JNDI 名が必要になります。エンタープライズ Bean 参照の代わりに、キューのリソース環境参照 (81 ページの「キューの参照宣言」) とキュー接続ファクトリのリモート参照 83 ページの「キュー接続ファクトリの参照宣言」 参照) が必要です。また、これらの参照の JNDI ルックアップ用のコードを追加する必要があります (84 ページの「JNDI ルックアップのコード」 参照)。

# 第7章

# トランザクション

この章では、EJB モジュールのプロパティシートを使用したコンテナ管理によるトランザクションのプログラミングを取り上げます。Bean 管理によるトランザクションについては、『Enterprise JavaBeans コンポーネントのプログラミング』を参照してください。

## デフォルトトランザクション境界

トランザクション境界は、トランザクションに関与する個々のエンタープライズ Bean の「トランザクション属」プロパティによって決定します。IDE の EJB ウィザードでエンタープライズ Bean を作成し、コンテナ管理によるトランザクションを指定すると、デフォルトのトランザクション属性を持つ Bean が作成されます。

ここでは、トランザクション属性のデフォルトの設定を表示する方法と個々の表示の意味を説明します。

「トランザクション設定」プロパティエディタを開き、デフォルトの設定を確認するには、次のようにします。

- EJB モジュールノードを右クリックし、コンテキストメニューの「プロパティ」>「トランザクション設定プロパティ」> 省略符号ボタン (...) を選択します。

  図 7-1 に、EJB モジュールの「トランザクション設定」プロパティエディタを示します。これらは、トランザクション属性のデフォルトの設定です。

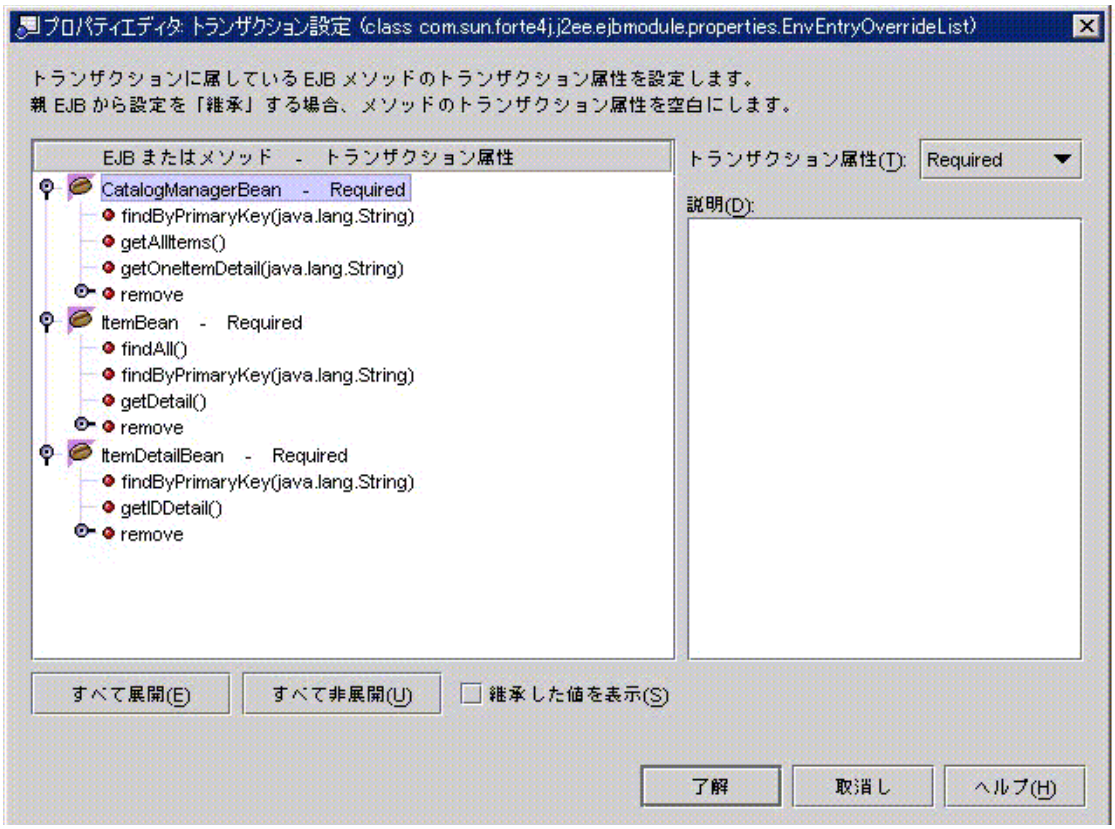


図 7-1　デフォルトのトランザクション属性

このエディタでのトランザクション属性設定について、次の点に注目してください。

- モジュール内のすべてのエンタープライズ Bean が一覧表示されています。
- 各エンタープライズ Bean が独自のトランザクション属性を持っており、それが Bean の名前の後ろに表示されています。図 7-1 では、すべてのエンタープライズ Bean がデフォルトの設定 (必須) になっています。
- エンタープライズ Bean ノードを展開し、エンタープライズ Bean 個々のメソッドを見ることができます。メソッドのトランザクション属性が独自の値に設定されていない場合は、エンタープライズ Bean の値が継承されます。図 7-1 の場合、独自のトランザクション属性値を持つメソッドはありません。これはデフォルトの設定です。

トランザクション属性が「必須」に設定されているメソッドはトランザクション形式で実行されなければなりません。アクティブなトランザクションがないときにメソッドが呼び出されると、コンテナは新しいトランザクションを開始します。アクティブなトランザクションの進行中にメソッドが呼び出されると、コンテナはアクティブなトランザクションのメソッドを取り込みます。これは、エンタープライズ Bean のデフォルトの動作です。

## トランザクション境界の再定義

EJB モジュールのビジネストランザクションが複数のエンタープライズ Bean にまたがることがよくあります。EJB の一般的なアーキテクチャは、1 つのセッション Bean といくつかのエンティティ Bean から構成されます。セッション Bean はすべてのメソッド呼び出しを受け取って必要な処理を実行し、モジュール内のエンティティ Bean のメソッドを呼び出します。図 7-2 に、このタイプのトランザクションを示します。

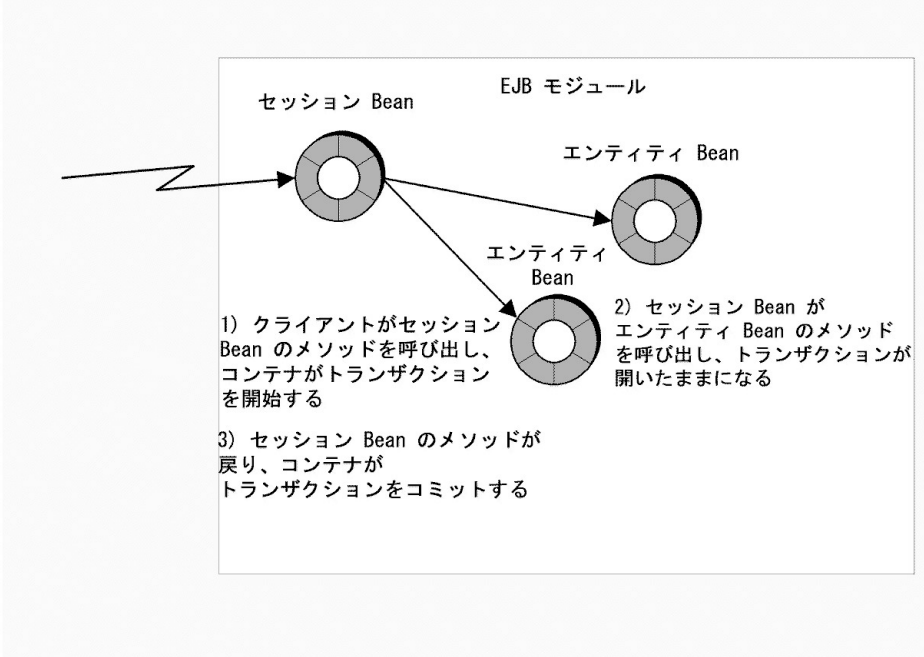


図 7-2　複雑なトランザクション

コンテナにこれらのビジネストランザクション境界を認識させます。クライアントがセッション Bean を呼び出し、そのセッション Bean が単一のデータベーストランザクションとして処理する複数のエンタープライズ Bean を呼び出したときに、すべての処理を行うようにして、一括してコミットまたはロールバックします。クライアントによってセッション Bean が呼び出されたときにコンテナがトランザクションを開くようにし、セッション Bean が 1 つ以上のエンティティ Bean を呼び出す間は、トランザクションを開いたままにする必要があります。最後に、エンティティ Bean の最後の呼び出しが戻り、クライアントが呼び出したセッション Bean のメソッドが完了すると、コンテナがトランザクションをコミットします。これにより、クライアントによるセッション Bean の呼び出しから開始されるすべての作業が 1 つのデータベーストランザクションに含まれます。

トランザクション境界をプログラムするには、「トランザクション設定プロパティ」を開き、トランザクションに関与するエンタープライズ Bean のトランザクション属性を変更します。トランザクション属性を変更するには、次のようにします。

1. EJB モジュールノードを右クリックし、コンテキストメニューの「プロパティ」>「トランザクション設定プロパティ」> 省略符号ボタン (...) を選択します。

   「トランザクション設定」プロパティエディタが開きます。

2. エンティティ Bean のノードを展開します。

3. セッション Bean のビジネスメソッドをクリックして選択します。

   この例では、セッション Bean のビジネスメソッドが呼び出されるたびに新しいトランザクションが起動するようにしたいため、個々のメソッドを選択し、そのトランザクション属性値を選択する必要があります。

4. 「トランザクション属性」フィールドをクリックし、値を変更します。

   選択したメソッドのトランザクション属性が変更されます。メソッド名の後ろに新しいトランザクション設定が表示されます。この例の場合は RequiresNew です。この設定により、コンテナはこのメソッドが呼び出されるたびに新しいトランザクションを開きます。

5. この処理を繰り返してトランザクション境界を再定義します。

   この例では、エンティティ Bean のビジネスメソッドのトランザクション属性が Mandatory に設定されています。これらのメソッドは、セッション Bean のメソッドによって開かれたトランザクションの境界内で実行する必要があります。 Mandatory に設定することにより、これらのメソッドはトランザクションがすでに進行中の場合のみ呼び出すようにコンテナに指示します。

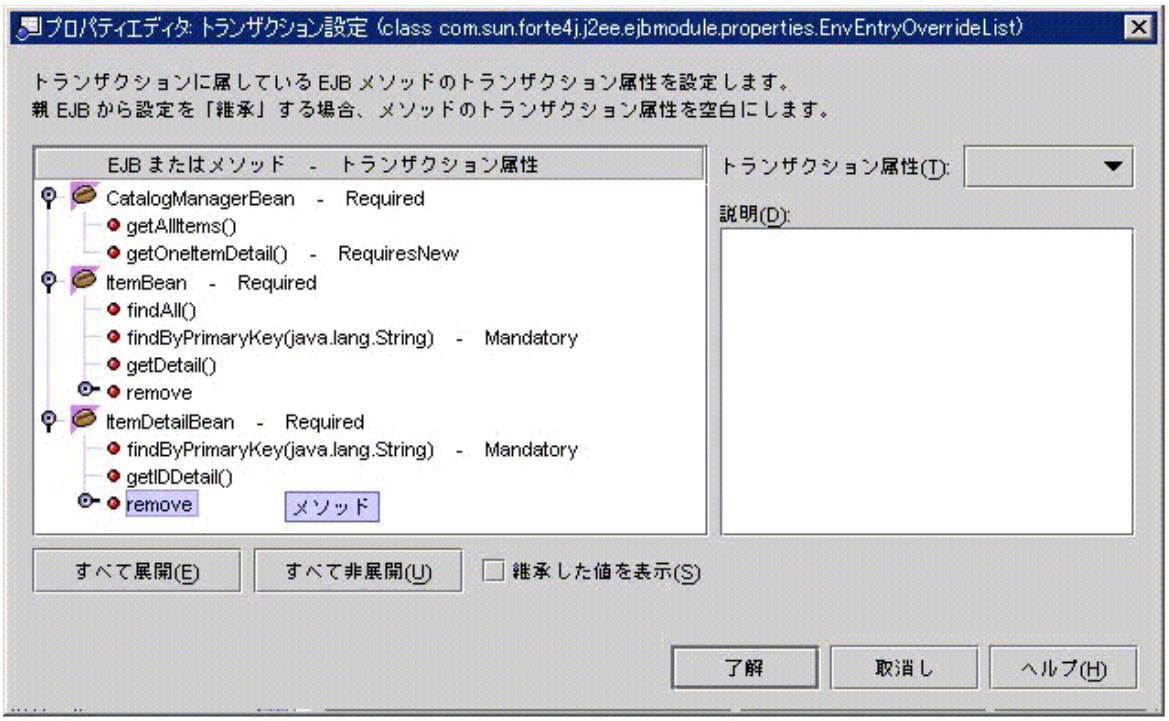


図 7-3　変更後のトランザクション設定

図 7-3 に、トランザクション属性変更後の EJB モジュールの「トランザクション設定」プロパティエディタを示します。トランザクション境界がビジネストランザクションと一致しています。次の変更が有効です。

- セッション Bean のメソッドのトランザクション属性が `RequiresNew` に設定されています。これは、クライアントがこれらのメソッドのどれかを呼び出すたびに、コンテナが新しいトランザクションを開くようにするためです。
- エンティティ Bean のビジネスメソッドの属性が `Mandatory` に設定されます。つまり、これらのメソッドはトランザクションの進行中に呼び出されなければなりません。これらのメソッドは、セッション Bean がトランザクションを開いた後に呼び出す必要があります。また、これらのメソッドは、セッション Bean のトランザクションの境界内で完全に実行する必要があります。

トランザクション属性をメソッドレベルで変更したため、メソッドノードのメソッド名の後ろに新しいトランザクション属性値が表示されています。

処理する EJB モジュールごとにビジネスロジックを分析し、そのロジックによって暗黙指定される様々なトランザクションモデルを判断する必要があります。その後、それらのトランザクションを実装するために、「トランザクション属性」プロパティエディタを使用し、それらのトランザクションに関与するエンタープライズ Bean (またはそのメソッド) のトランザクション属性を設定します。

# 第8章

# セキュリティ

一般に J2EE モデルは、セキュリティがモジュールレベルで設定されているものと想定しています。また、コンポーネントプロバイダがビジネスロジックを設計し、開発するときに、セキュリティを考慮すると想定しています。コンポーネントプロバイダの多くは、ビジネスロジックを利用するユーザーのロールと、どのロールにどの機能の使用を許可すべきかについて一般的な知識を持っていると想定しています。

この後、IDE を使用して Web モジュールと EJB モジュールのセキュリティを設定し、この 2 つのモジュールをアセンブルして J2EE アプリケーションを作成するときに両方のセキュリティを統合する方法を説明します。

## Web モジュールのセキュリティ

ここでは、自分が人材データを処理する Web モジュールのコンポーネントプロバイダであると仮定してください。社員の個人情報を保守するためにすべての社員が利用できなければならない Web リソースと、人材関連事務、監督、監査などの各ロールだけが利用できるようにしなければならない Web リソースがなにかは分かっています。

コンポーネントプロバイダであり、初期モジュールアセンブル担当者として、これらのロールを表す汎用セキュリティロールを設定し、作成する Web リソースにロールをマップできます。これらの汎用ロールは、後でアプリケーションサーバー環境の実際のユーザー名とグループ名にマップできます。このマッピングは、アプリケーションのアセンブル時か、またはその後、アプリケーションが本番環境に配備されるときにアプリケーションレベルで実行できます。

また、プロジェクトによっては、コンポーネントプロバイダで初期モジュールアセンブルを担当者したとしても、セキュリティをビジネスロジックにどのように適用すればよいかが分からない場合があります。このような場合は、セキュリティロールの設定を開発プロセスのもっと後の段階にまで延期できます。たとえば、アプリケーションのアセンブル担当者は、モジュールをアセンブルしてアプリケーションを作成する前に、モジュールレベルの配備記述子を操作できます。

この後、モジュールレベルでセキュリティロールを設定し、それらを Web リソースにマップする方法を説明します。

Web モジュールの宣言型セキュリティは、Web リソースへのセキュリティロールのマッピングから構成されます。この作業を行うには、次のことが必要です。

1. セキュリティロールを宣言します。
2. 保護対象の「Web リソース」を定義します。Web リソースはモジュール内の URI です。
3. セキュリティロールを Web リソースにマップします。これにより、マップされたロールに、指定 Web リソースへのアクセス権が与えられます。

## セキュリティロールの宣言

セキュリティロールは、「セキュリティロール」プロパティエディタで宣言します (このプロパティエディタを開くには、`web.xml` ノードを右クリックし、コンテキストメニューの「プロパティ」>「セキュリティ」タブ >「セキュリティロール」> 省略符号 (...) ) ボタンを選択します) 。

プロパティエディタの「追加」ボタンをクリックすると、新しいセキュリティロールを宣言するためのダイアログが開きます。「編集」ボタンおよび「削除」ボタンをクリックすると、すでに宣言されているロールを処理できます。

図 8-1 は、`Me` と `EveryoneElse` の 2 つのロールが宣言された後の「セキュリティロール」プロパティエディタを示しています。


図 8-1　Web モジュールの「セキュリティロール」プロパティエディタ

## Web リソースの定義とセキュリティロールのマッピング

「セキュリティ制限」プロパティエディタでは、Web リソースを定義し、そのリソースへのアクセスを許可するロールを指定できます (このプロパティエディタを開くには、`web.xml` ノードを右クリックし、コンテキストメニューの「プロパティ」>「セキュリティ」タブ >「セキュリティ制限」> 省略符号ボタン (...) ) ボタンを選択します) 。

プロパティエディタの「追加」ボタンをクリックすると、Web リソースを定義するためのダイアログが開きます。

図 8-2 に、URL パターンの `/allItems` を Web リソースとして定義する方法示します (この URL パターンが Web コンポーネントにマップされる仕組みについては、31 ページの「URL からサーブレットへのマッピング」 と35 ページの「JSP ページの設定」 を参照してください)。


図 8-2　Web モジュールの「Web リソースコレクション」ダイアログボックス

Web リソースは、URL パターンまたはそれらのサブセットに関連するすべての HTTP メソッドとして定義できます。

自分で定義した Web リソースをセキュリティロールにマップすることもできます。このダイアログを完了すると、情報の要約が「セキュリティ制限」プロパティエディタに表示されます。図 8-3 は、Web リソースが `AllItems` という名前で設定された後の「セキュリティ制限」プロパティエディタを示しています。


図 8-3　Web モジュールの「セキュリティ制限」プロパティエディタ

「編集」ボタンおよび「削除」ボタンを使用すると、プロパティエディタに表示されるセキュリティ制約をどれでも変更できます。

## プログラム可能なセキュリティ

モジュール内の Web コンポーネントがプログラム可能なセキュリティを使用している場合は、セキュリティチェックコードで使用されるセキュリティロール参照を、モジュールレベルで宣言されたセキュリティロールにマップする必要があります。

プログラム可能なセキュリティ機能を使用する Web コンポーネントには、ユーザーの資格に直接アクセスするコードが含まれており、コンテナの宣言型セキュリティ機構が実行する以上の検証を実行します。次に例を示します。

```
...
context.isCallerInRole(roleRefMe);
...
```

ロールはモジュールレベルで定義されるため、このコンポーネントレベルのコードを書く段階ではロールについておそらく分かっていません。したがって、このコードは、後で実際のセキュリティロールにマップされるセキュリティロール参照 (`roleRefMe`) を使用します。このマッピングは、「編集サーブレット」ダイアログで行います。このダイアログを開くには「サーブレット」プロパティエディタでロール参照を使用するサーブレットを選択し、「編集」をクリックします。図 8-4 では、参照 `roleRefMe` がロール `Me` にマップされています。

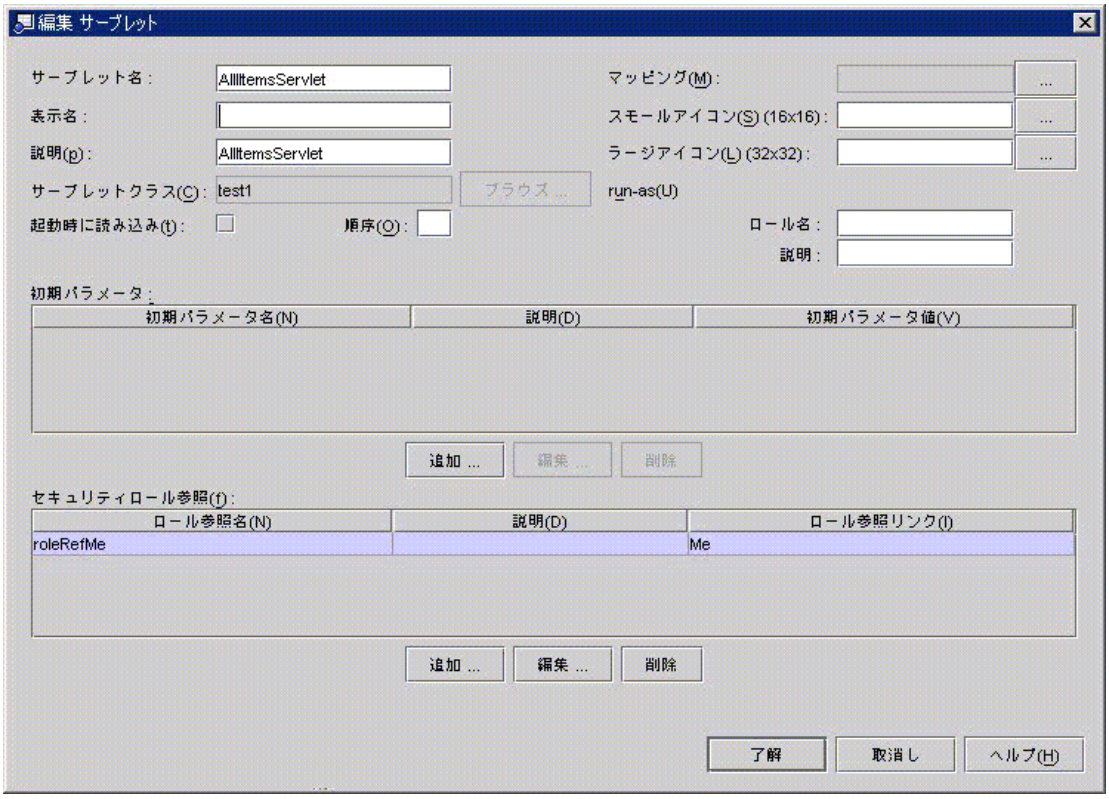


図 8-4　Web モジュールの「編集サーブレット」ダイアログボックス

このようなマッピングを行う前に、モジュールレベルでロールが宣言されていなければなりません。

## EJB モジュールのセキュリティ

ここでは、自分がコンポーネントプロバイダであると仮定してください。人材データにアクセスするいくつかのエンタープライズ Bean をすでに開発し、それらをアセンブルして EJB モジュールを作成しようとしています。社員の個人情報を保守するためにすべての社員が利用できなければならないデータと、人材関連事務、監督、監査などの各ロールだけが利用できなければならないデータがなにかは分かっています。

コンポーネントプロバイダとして、また初期モジュールアセンブル担当者として、各ロールを表す汎用セキュリティロールを設定し、これらのデータにアクセスするメソッドにロールをマップできます。これらの汎用ロールは、配備環境の実際のユーザー名とグループ名に後でマップできます (この 2 番目のマッピングは、アプリケーションのアセンブル時、またはその後、アプリケーションが本番環境に配備されるときに、アプリケーションレベルで実行できます)。

また、プロジェクトによっては、コンポーネントプロバイダで、初期モジュールアセンブルを担当したとしても、セキュリティをビジネスロジックにどのように適用すればよいかが分からない場合があります。そのような場合は、セキュリティの設定を開発プロセスのもっと後の段階にまで延期できます。たとえば、アプリケーションのアセンブル担当者は、モジュールをアセンブルしてアプリケーションを作成する前に、モジュールレベルのセキュリティを操作できます。

EJB モジュールのセキュリティの設定は、エンタープライズ Bean のメソッドにセキュリティロールをマップすることによって行います。この作業を行うには、次のことが必要です。

1. モジュールのサービスを使用できるユーザーのカテゴリを表す汎用セキュリティロールを宣言します。
2. これらのセキュリティロールを、モジュールのエンタープライズ Bean のメソッドにマップします。これにより、どのロールがどのメソッドにアクセスできるかが決まります。
3. プログラム可能なセキュリティ機能を使用するエンタープライズ Bean がモジュールに含まれる場合は、これらのエンタープライズ Bean で使用されるセキュリティロール参照をセキュリティロールにマップします。

この後、これらのアセンブル作業を実行する方法を説明します。

## セキュリティロールの宣言

セキュリティロールは、「セキュリティモジュールの編集」ダイアログボックスで宣言します (このダイアログボックスを開くには、モジュールノードを右クリックし、コンテキストメニューの「プロパティ」>「セキュリティロール」> 省略符号ボタン (...) を選択します。「セキュリティロール」プロパティエディタが開いたら、「モジュールロールの編集」ボタンをクリックします)。

図 8-5 は、`Me` と `EveryoneElse` の 2 つのロールが宣言された後の「モジュールロールの編集」ダイアログを示しています。

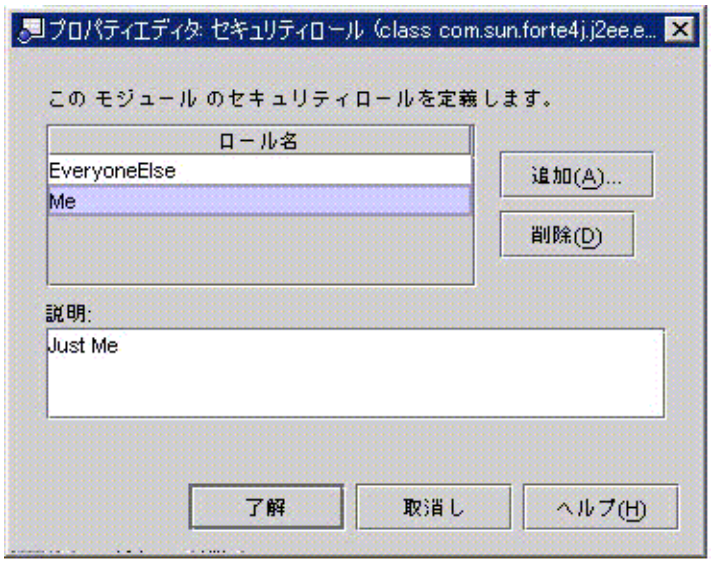


図 8-5　EJB モジュールの「セキュリティロール」プロパティエディタ

## メソッドアクセス権へのセキュリティロールのマッピング

モジュールのセキュリティロールを宣言したら、モジュール内に含まれるエンタープライズ Bean の特定のサブセットを実行する権限を各ロールに与えることができます。この作業は「メソッドアクセス権」プロパティエディタで行います (このプロパティエディタを開くには、含まれている EJB ノードを操作します。EJB モジュールのサブノードがあります)。

モジュールに含まれているエンタープライズ Bean を表すノードを右クリックし、コンテキストメニューの「プロパティ」>「メソッドのアクセス権」> 省略符号ボタン (...) を選択します。プロパティエディタは表であり、行がエンタープライズ Bean の各メソッドを、列がモジュールに宣言された各セキュリティロールを示します。図 8-6 に「メソッドアクセス権」プロパティエディタを示します。ここでは、先の 2 つのセキュリティロールがすでに宣言されています。

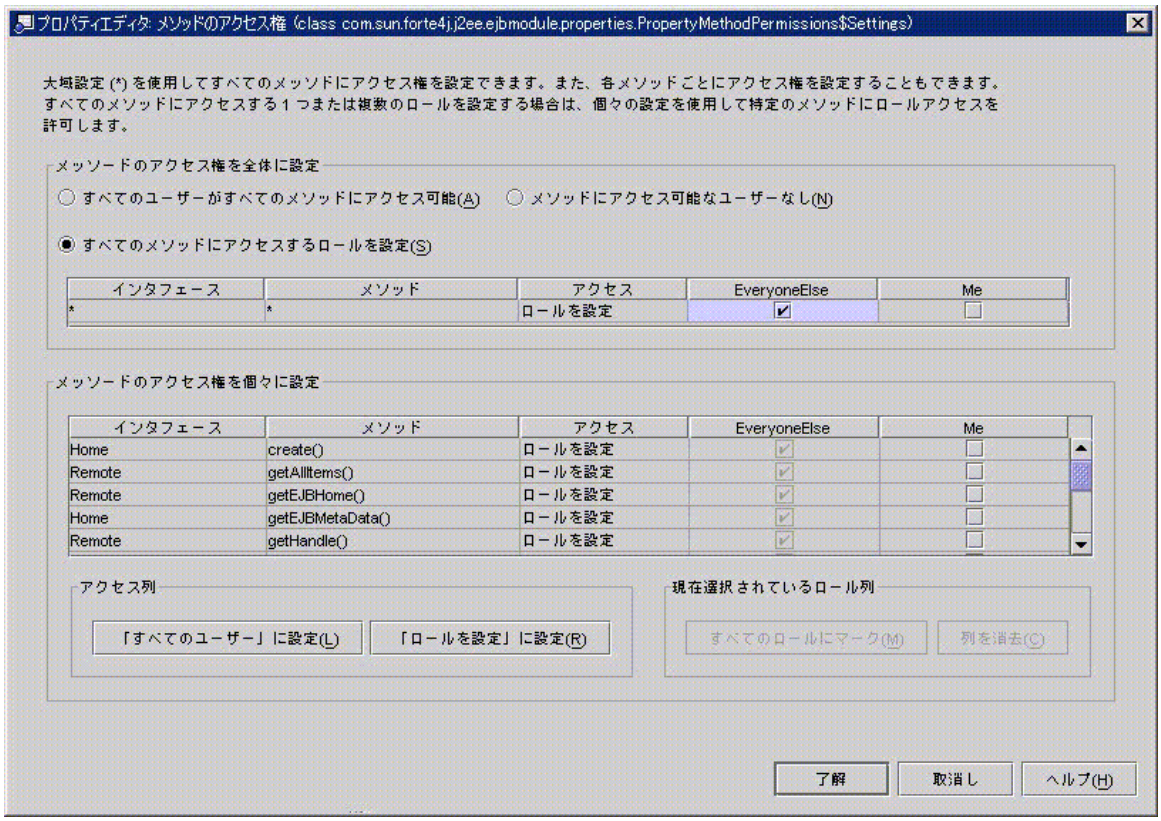


図 8-6　EJB モジュールの「メソッドのアクセス権」プロパティエディタ

このダイアログは、さまざまな設定を行うことができます。

- 上部パネルのボタンは、アクセス権をグローバルに適用する場合に使用します。すべてのユーザーにすべてのメソッドの呼び出しを許可することも、すべてのアクセスを禁止することもできます。
- 「すべてのメソッドにアクセスするロールを設定」を選択すると、アクセス権をより細かく制御できます。ボタンの下の小さな表を使用し、モジュールで宣言されたロールから選択を行います。列にチェックマークを付けると、エンタープライズ Bean のすべてのメソッドを実行するためのアクセス権がロールに与えられます。図 8-6では、EveryoneElse 列にチェックマークが付いています。結果として、このロールを持つユーザーは、エンタープライズ Bean のどのメソッドも実行できます。「Me」列にはチェックマークがありません。このロールを持つユーザーは、エンタープライズ Bean のどのメソッドも実行できません。

- 下の表では、アクセス権を最も細かく制御できます。行をクリックすると、その行のメソッドのアクセス権だけを設定できます。この設定は、ほかのメソッドの設定から完全に独立しています。たとえば、2 行目をクリックし、getAllItems の「アクセス」フィールドを「すべてのユーザー」に設定します。そうすると、どのロールもこのメソッドを実行できるようになります。その後、別の行に移動し、「アクセス」フィールドを「ロールを設定」に設定し、そのメソッドの実行を許可するロールを個別に選択することができます。

表の下のボタンは、表の中で編集をするためのいくつかのショートカットとして利用できます。

## プログラム可能なセキュリティ

モジュール内にプログラム可能なセキュリティを使用しているエンタープライズ Bean がある場合は、セキュリティチェックコードで使用されるセキュリティ参照を、モジュールレベルで宣言されたセキュリティロールにマップする必要があります。

プログラム可能なセキュリティ機能を使用するエンタープライズ Bean には、ユーザーの資格に直接アクセスするコードが含まれており、コンテナの宣言型セキュリティ機構が実行する以上の検証を実行します。次に例を示します。

```
...
context.isCallerInRole(everyOne);
...
```

セキュリティロールはモジュールレベルで定義されるため、このエンタープライズ Bean のコードが書かれた段階ではロールについておそらく分かっていません。したがって、このコードは、モジュールにエンタープライズ Bean を追加するときに実際のセキュリティロールにマップされるセキュリティロール参照 (`everyOne`) を使用します。ロールはエンタープライズ Bean のプロパティとして宣言されなければなりません。図 8-7 に、エンタープライズ Bean の「セキュリティロール参照」プロパティエディタ上で宣言されたロールを示します。このロールはリンクされていません。


図 8-7　エンタープライズ Beanの「セキュリティロール参照」プロパティエディタ

この場合、このロール参照はモジュールレベルでロールにマップされています。このマッピングは、モジュールの「セキュリティロール」プロパティエディタで行います。このエディタを開くと、モジュール内のすべてのセキュリティロール参照と各参照がリンクされているかどうかが示されます。図 8-8 の場合、everyOne 参照がまだリンクされていません。

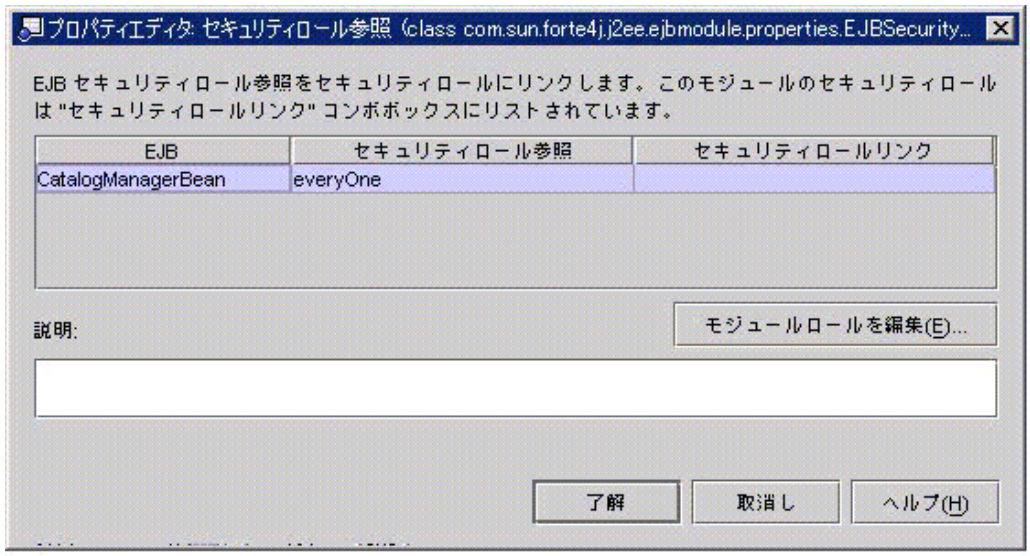


図 8-8　EJB モジュールの「セキュリティロール参照」プロパティエディタ

参照をマップするには、「セキュリティロールリンク」フィールドをクリックしてロールを選択します。図 8-9 は、everyOne リンクがマップされた後のプロパティエディタを示しています。

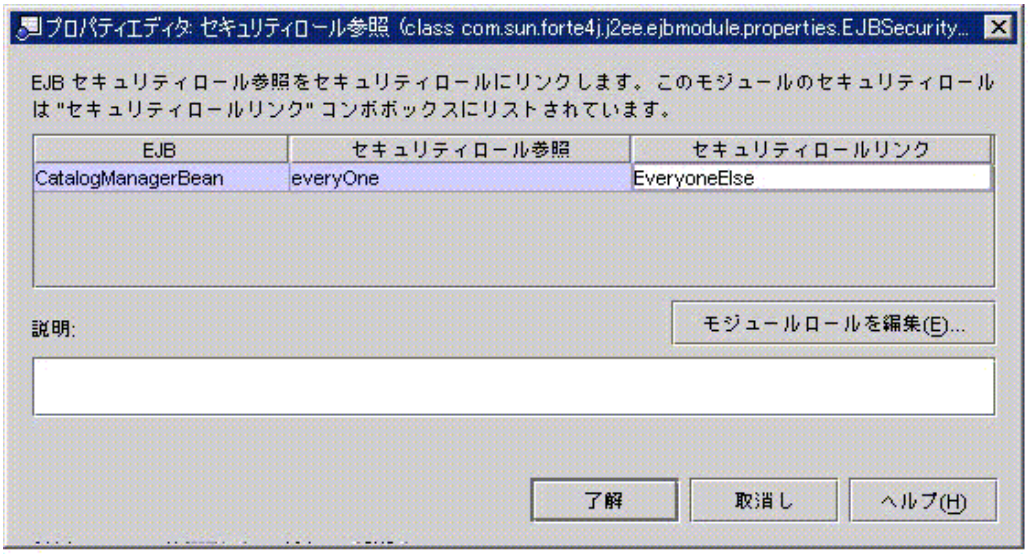


図 8-9　EJB モジュールの「セキュリティロール参照」プロパティエディタ

## J2EE アプリケーションのセキュリティ

アプリケーションアセンブル担当者に対して、新しくアセンブルされたアプリケーションは次のいずれかの状態で示されます。

- アプリケーション内の 1 つまたは複数のモジュールにセキュリティロールが定義されていない場合は、モジュールレベルでロールを定義する必要があります。Web モジュールと EJB モジュールのセキュリティについては、前の節を参照してください。
- セキュリティが汎用ロールによってモジュールレベルで設定されている場合は、同様のロールにさまざまなモジュールのさまざまな名前が与えられている場合があります。その場合は、同じアプリケーションレベルのロールに等価のロールのすべてをマップする必要があります。

- モジュールレベルでセキュリティを設定したコンポーネントプロバイダが配備環境を分かっていて、モジュールレベルのロールが相互にマップされていれば、追加のマッピングが不要な場合があります。

追加のマッピングが必要な場合は、アプリケーションの「セキュリティロール」プロパティエディタで行います (このエディタを開くには、アプリケーションノードを右クリックし、コンテキストメニューの「プロパティ」>「セキュリティロール」> 省略符号ボタン (...) を選択します)。図 8-10 の場合、セキュリティロールがモジュールレベルで定義されています。

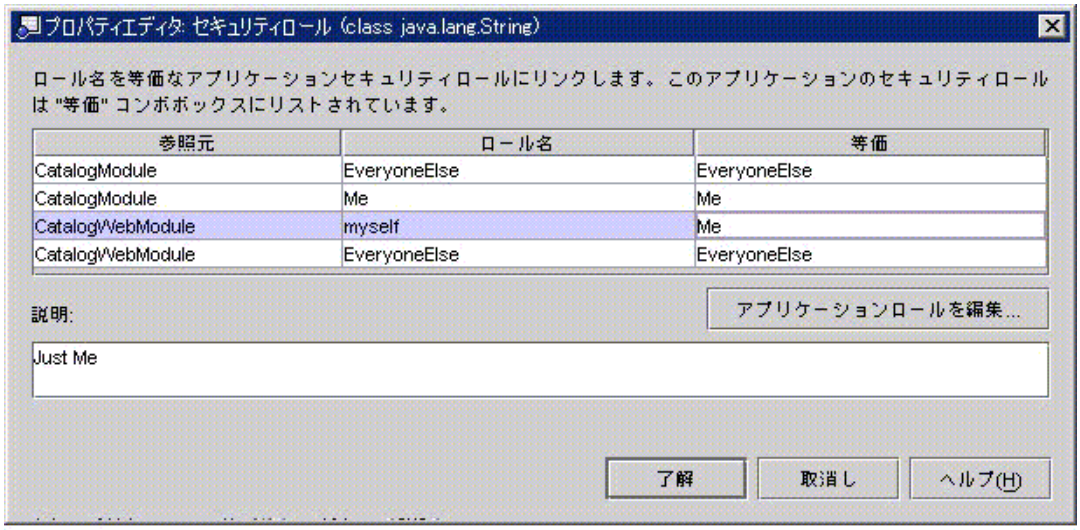


図 8-10　J2EE アプリケーションの「セキュリティロール」プロパティエディタ

モジュールレベルで宣言されたセキュリティロールは、ダイアログの最初の 2 つの列に表示されます。各ロールは、そのモジュールと名前によって識別されます。モジュールレベルのロールごとに、IDE はデフォルトのアプリケーションレベルのロールを作成しますが、この名前はモジュールレベルのロールと同じです。アプリケーションレベルのロールは、「等価」列に表示されます。

このエディタでロールの違いを修正するには、2 つの方法があります。図 8-10 の例は、2 つのモジュールを持つアプリケーションを示しています。各モジュールには 2 つのセキュリティロールがあります。異なる点は、Web モジュールのロール名は `myself` 、EJB モジュールのロール名は `Me` であるということです。これらのロールは等価であるため、アプリケーションレベルのロールを 1 つだけにします。図では、

ロール myself をロール Me にマップし直すことによってこの違いを修正しています。実際には、モジュールレベルのロールの Me および myself の両方がアプリケーションレベルのロール Me にマップされています。

まったく新しいロールをアプリケーションレベルで作成し、モジュールレベルの複数のロールをその新しいロールにマップすることもできます。たとえば、アプリケーションのモジュールの 1 つに sa という名前のロールがあり、もう一方のモジュールに sadmin という名前のロールがあるとします。この違いをなくすために、sysadmin という名前でアプリケーションレベルの新しいロールを作成することにします。それには、「アプリケーションロールの編集」ボタンをクリックしてアプリケーションレベルのロールを宣言できるダイアログを開きます。

sysadmin というロールを宣言した後、「セキュリティロール」プロパティエディタに戻ります。モジュールレベルの各ロールの「等価」列をクリックします。これにより、アプリケーションレベルのロールが表示されます。モジュールレベルの各ロールの「等価」列を sysadminに設定します。

# 第9章

# J2EE モジュールと J2EE アプリケーションの配備と実行

IDE の配備機能と実行機能を使用すると、エンタープライズアプリケーションを対話形式で開発できます。適切な Web サーバーアプリケーションサーバーがインストールされていれば、個人またはチームとして、Web アプリケーションや J2EE アプリケーションの開発とアセンブル、配備、テスト目的での実行、ソースコードやコンポーネントプロパティの修正、再配備と再テストなどを行うことができます。テストでアセンブルの問題が生じなかった場合は、アセンブルし直す必要はありません。

実際の配備では、実行機能はサーバーに用意されている配備ツールの代替手段にはなりません。アプリケーションのテストを終了したら、WAR ファイルまたは EAR ファイルを生成し、サーバーの配備ツールを使用してそのファイルを配備できます。

この章では、アセンブルされた Web アプリケーションおよび J2EE アプリケーションを IDE 内から配備し、Web ブラウザを介してテスト目的で実行する方法について説明します。IDE には、コンポーネントレベルでテストを実行するための機能もあります。これらの機能については、『Web コンポーネントのプログラミング』、『Enterprise JavaBeans コンポーネントのプログラミング』、『Web サービスのプログラミング』を参照してください。

## 「エクスプローラ」ウィンドウでのサーバーの表示

Web アプリケーションまたは J2EE アプリケーションをサーバーに配備するには、サーバーと対話する必要があります。このプロセスを簡略化するために、Sun ONE Studio 4 は、Web サーバーおよびアプリケーションサーバーをノードとして「エクスプローラ」ウィンドウに表示します。

エクスプローラウィンドウのほかのノードと同様、これらのサーバーノードにもプロパティシートとコンテキストメニューコマンドがあり、IDE 内からサーバーとの対話を管理するのに役立ちます。ここでは、サーバーノードについて説明します (これらのノードの表示を可能にする仕組みについては、付録 A を参照してください)。

## サーバーレジストリノード

最上位にはサーバーレジストリノードがあります。このノードは、ほかのサーバー関連ノードをグループ化するものです。このノードには、独自のコマンドやプロパティがありません。図 9-1 は、デフォルトのサブノードを持つサーバーレジストリを示しています。

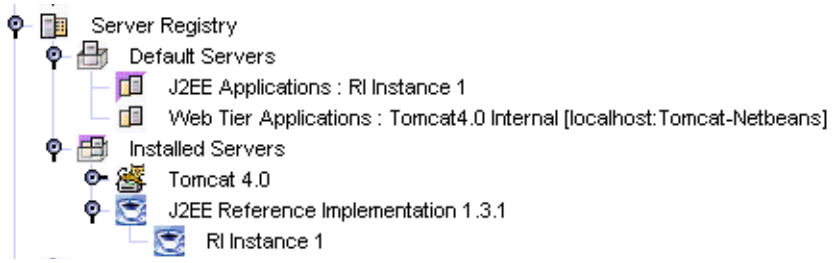


図 9-1　サーバーレジストリとデフォルトのサブノード

## 「インストールされているサーバー」ノード

このノードは、ほかのノードをグループ化するために存在します。このノードには、独自のコマンドやプロパティがありません。図 9-1 は、デフォルトのサブノードを持つ「インストールされているサーバー」ノードを示しています。

## サーバー製品ノード

これらのノードは、「インストールされているサーバー」ノードのサブノードです。これらのノードはそれぞれ IDE によって認識される Web サーバー製品またはアプリケーションサーバー製品を表します (実際には、特定のサーバー製品と対話できる IDE プラグインモジュールの存在を表します。サーバープラグインの詳細については、付録 A を参照してください)。図 9-1 は、デフォルトインストール後の IDE です。「インストールされているサーバー」ノードには、Tomcat Web サーバーと J2EE アプリケーションサーバーが示されています。

サーバー製品がインストールされ、対応するサーバー製品ノードがエクスプローラウィンドウに表示されると、IDE はサーバーのインスタンスを認識し、それにアプリケーションを配備できます。これらの各ノードにはコンテキストメニューとプロパティシートがありますが、各ノードの機能は、サーバー製品とプラグインモジュールによって異なります。

手順はサーバー製品によって異なりますが、通常、サーバーを使用するには、これらのノードの 1 つを設定し、サーバー製品の特定のインストールを認識します。次に例を示します。

- Tomcat サーバーは製品とともにインストールされるため、サーバーの場所はすでにわかっています。Tomcat 4.0 を起動してそこに配備するために、追加の設定をする必要はありません。
- J2EE リファレンス実装 1.3.1 は製品とともにインストールされるため、サーバーの場所はすでに分かっています。サーバーのインスタンスが作成されます。「RI インスタンス 1」ノードを右クリックし、コンテキストメニューのコマンドを使用してサーバーの起動と停止を行うことができます。

IDE でサーバー製品を設定する方法については、『Sun ONE Studio 4, Enterprise Edition for Java インストールガイド』 を参照してください。

## サーバーインスタンスノード

サーバー製品ノードの下には、サーバー製品のインスタンスを表すノードがあります。配備時には特定のサーバーインスタンスに配備するため、配備を行うには使用しているサーバー製品のサーバーインスタンスノードがなければなりません。

- J2EE RI の場合、サーバーインスタンスとそれを表すノードは、IDE のインストール時に作成されます。このノードは、図 9-1 に RI インスタンス 1 ノードとして表示されています。
- ほかのサーバー製品を使用する場合は、配備または実行の前に、サーバーインスタンスを作成し、それをサーバーインスタンスノードで表現する必要があります。サーバーインスタンスの作成手順については、『インストールガイド』を参照してください。

サーバーインスタンスノードを使用してそのノードが表すサーバーインスタンスの停止と起動を行うことができます。この手順は、使用中のサーバー製品によって異なります。

### デフォルトサーバーノード

これらのノードは、現在デフォルトサーバーインスタンスとして指定されているサーバーインスタンスを表します。サーバーインスタンスがデフォルトサーバーインスタンスのとき、アプリケーションはほかのインスタンスを指定しない限りこのインスタンスに配備されます (アプリケーションノードには、「アプリケーションサーバー」プロパティがあります。このプロパティのデフォルト値は、デフォルトサーバーです。この値を任意のサーバーインスタンスノードの名前に変更できます)。

サーバーインスタンスをデフォルトのサーバーにするには、そのサーバーインスタンスノードを右クリックし、コンテキストメニューの「デフォルトとして設定」を選択します。図 9-1 の場合、J2EE アプリケーションのデフォルトサーバーは RI インスタンス 1 です。

## サーバー固有のプロパティ

処理対象のモジュールとアプリケーションにはプロパティシートがあり、そのプロパティシートを使用して、モジュールまたはアプリケーションがアプリケーションサーバーから必要とするサービスを記述します。プロパティシートには、J2EE アプリケーションによって定義されているプロパティを示す「プロパティ」タブのほかに、サーバー製品に必要な情報を指定するためのプロパティを含むサーバー固有タブがあります。

たとえば、図 9-2 は、EJB モジュールのプロパティシートの「J2EE RI」タブです。このモジュールにはいくつかの CMP エンティティ Bean が含まれ、このタブのプロパティのほとんどが、コンテナ管理による持続性の J2EE RI 実装に必要な情報を提供するために使用されます。


図 9-2　EJB モジュールの J2EE RI 固有のプロパティ

# サーバーインスタンスノードを使用した配備と実行

ここでは、IDE 内から J2EE アプリケーションを配備し、実行するためのガイドラインを示します。

アプリケーションを配備し、実行するには、次のようにします。

1. アセンブルされた J2EE アプリケーションから始めます。アプリケーションのアセンブルが完全に行われていることを確認します。
2. アプリケーションサーバーのインスタンスを選択します。

   アプリケーションノードには、「アプリケーションサーバー」プロパティがあります。このプロパティの初期値は、「デフォルトアプリケーションサーバー」です。この設定のまま処理を進めると、アプリケーションは、サーバーレジストリで現在 J2EE アプリケーションのデフォルトサーバーとして指定されているサーバーインスタンスに配備されます。

   このプロパティのプロパティエディタを開き、配備先のサーバーインスタンスの名前を選択することもできます。プロパティエディタはブラウザであるため、サーバーレジストリのすべてのサーバーインスタンスを確認し、その中の 1 つを選択できます。
3. アプリケーションの配備と実行を続けて行う場合は、アプリケーションノードを右クリックし、コンテキストメニューの「実行」を選択します。

   配備処理が開始されるので、出力ウィンドウで処理を監視します。配備が完了すると、アプリケーションサーバーの環境でアプリケーションが実行されます。何が起こるかは、アプリケーションによって異なります。たとえば、アプリケーションに Web モジュールが含まれている場合は、Web ブラウザが起動し、アプリケーションの開始ページが開きます。
4. 配備と実行を個別に実行することもできます。その場合は、アプリケーションを右クリックし、コンテキストメニューの「配備」を選択します。配備が完了したら、アプリケーション自体を実行してください。

   たとえば、アプリケーションに Web モジュールが含まれている場合は、Web ブラウザを起動し、アプリケーションの開始ページを開くことができます。

# 付録 A

# IDE による J2EE モジュールおよび J2EE アプリケーションの配備

第 9 章では、J2EE モジュールと J2EE アプリケーションの配備について簡単に説明しました。この付録では、Sun ONE Studio 4 IDE 内から J2EE モジュールまたは　J2EE アプリケーションを配備して実行することを可能にしている仕組みについて詳しく説明します。

## 反復開発のサポート

IDE の開発機能は、エンタープライズアプリケーションの反復開発をサポートしています。適切な Web サーバーまたはアプリケーションサーバーがインストールされていれば、個人またはチームとして、Web モジュールや J2EE アプリケーションの開発とアセンブル、配備、テスト目的での実行、ソースコードやコンポーネントプロパティの修正、再配備と再テストなどを行うことができます。テストでアセンブルの問題が生じなかった場合、アセンブルし直す必要ありません。

実際の配備では、Sun ONE Studio 4 配備機能は、サーバーに用意されている配備ツールの代替手段にはなりません。開発のこの段階に達したら、WAR ファイルまたは EAR ファイルとしてアプリケーションをエクスポートし、サーバーが提供するツールを使用して配備します。

この付録では IDE と Web サーバーまたはアプリケーションサーバーとの対話について詳しく説明します。ここでは、IDE の配備コマンドを使用するときに何が起こるかについて説明します。配備機能の動作を理解すると、その機能を効果的に使用できます。配備機能の実際の使用手順については、第 9 章 を参照してください。

# サーバープラグインの概念

配備とは、配備可能な形式のモジュールまたはアプリケーションを J2EE 実行環境に配信することをいいます。実行環境は、Web サーバーまたはアプリケーションサーバーの形式を使用します。特定のサーバーに配備するために、IDE は有効なコマンドをサーバーの配備ツールに対して実行する必要があります。これに加えて、ほとんどのサーバーには、J2EE 標準配備記述子だけでなくサーバー固有のプロパティが必要であり、IDE はこれらのプロパティを提供できる必要があります。

IDE が各種の Web サーバーおよびアプリケーションサーバーに配備できるようにするために、サーバープラグインという概念が開発されました。プラグインとは、IDE と特定のサーバー製品間の対話を管理する IDE モジュールをいいます。開発者がアプリケーションを配備するとき、その配備先のサーバーを選択すると、IDE が適切なプラグインを使用して配備コマンドを処理します。IDE はサーバーの配備ツールに適したコマンドを生成して、サーバーに渡すファイルに適切な非標準プロパティファイルを取り込むことができます。図 A-1 に、この手順を示します。

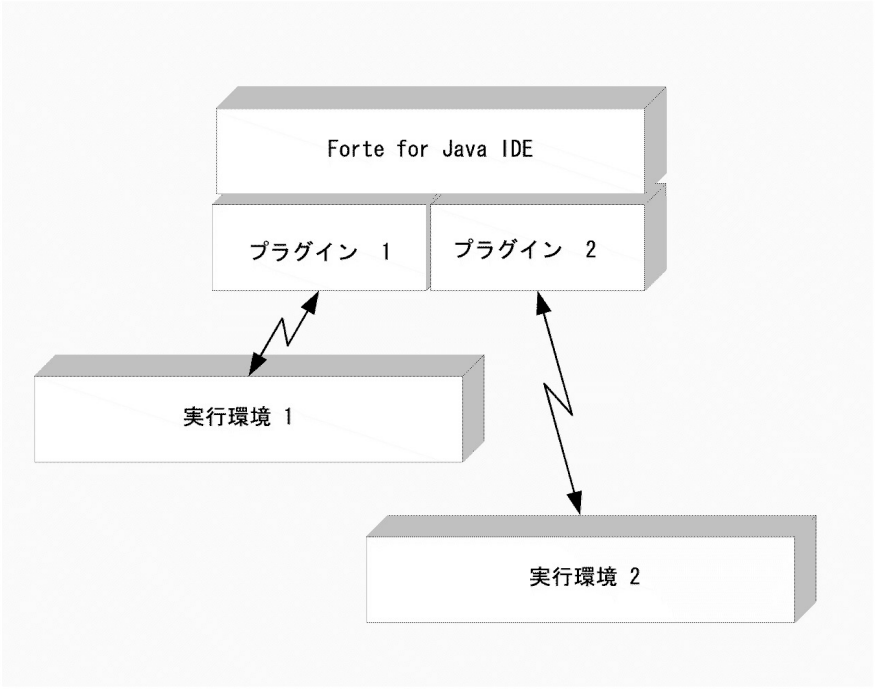


図 A-1　IDE が J2EE 実行環境と通信することを可能にするサーバープラグイン

アプリケーションを配備するアプリケーション開発者に対して、プラグインは次の機能を提供します。

- IDE の「エクスプローラ」ウィンドウにおけるプラグインの表示
  各プラグインは、サーバー製品ノードで表されます。開発者は、サーバー製品のインストールディレクトリによって、サーバー製品ノードを設定します。サーバー製品ノードの表示と使用法については、134 ページの「サーバー製品ノード」を参照してください。
- サーバー製品ノードのサブノードとしての、実行サーバーインスタンスの表示
  開発者は、配備ターゲットとして、「エクスプローラ」ウィンドウに表示された任意のサーバーインスタンスを選択できます。サーバーインスタンスノードの表示と使用法については、135 ページの「サーバーインスタンスノード」を参照してください。

- コンポーネント、モジュール、およびプロパティシートのサーバー固有のタブ
  これらのタブには、サーバー製品に必要な非標準プロパティが表示され、サーバー製品に必要な値の入力が開発者に要求されます。
- 選択したサーバーに合わせて配備コマンドを処理する機構
  この処理の詳細については、次の節で説明します。

## プラグインを使用する配備プロセス

1. IDE のインストール時、使用する Web サーバーまたはアプリケーションサーバーと適切なプラグインをインストールします (いくつかのサーバーといくつかのプラグインがデフォルトでインストールされます) 。詳細については、『Sun ONE Studio 4, Enterprise Edition for Java インストールガイド』を参照してください) 。
2. アプリケーションのビジネスロジックに合わせて J2EE コンポーネントを開発します。
3. コンポーネントをアセンブルしてモジュールを作成し、モジュールをアセンブルしてアプリケーションを作成します。プロパティシートを使用し、J2EE 標準配備記述子要素とサーバーに必要な非標準要素を指定します。
4. アプリケーションがアセンブルされたら、ターゲットサーバーインスタンスを特定します。
5. IDE の配備コマンドを使用し、配備処理を開始します。

第 9 章 に、配備に向けてアプリケーションを準備し、配備コマンドを発行する手順を示します。この章では、主にコマンド発行後に何が起こるかについて説明します。

6. IDE は、アプリケーションの WAR ファイルまたは EAR ファイルを作成するために必要なすべてのファイルを特定します。

これには、配備記述子で特定された J2EE コンポーネントと、それらのファイルで使用される Java クラスまたは静的リソースが含まれます。IDE は、コンポーネントのファイル依存関係をすべて特定します。

7. IDE は、アプリケーションの配備先であるサーバー製品を特定します。
8. プラグインは、WAR ファイルまたは EAR ファイルの妥当性検査を行います。

9. IDE は、アプリケーションの WAR ファイルまたは EAR ファイルを生成します。これには、J2EE 配備記述子、サーバー固有のタブを持つ個別ファイル、リモートメソッドの呼び出しに必要なスタブクラスまたはスケルトンクラスが含まれます。

10. プラグインは WAR ファイルまたは EAR ファイルをサーバーに渡します。

サーバー製品によって、プラグインは同じアプリケーションの初期の配備を自動的にクリーンアップするか、サーバーインスタンスにすでに配備されているアプリケーションとの競合をなくそうとします。

11. サーバーが引き継ぎ、配備記述子とサーバー固有の配備ファイルを読み取り、独自の標準に従って WAR ファイルまたは EAR ファイルを配備します。

この後、開発者は Web ブラウザを起動して、サーバーで実行中の配備済みアプリケーションへの HTTP 接続を開くことができます。開発者が Web アプリケーションを実行する場合、IDE は Web ブラウザを自動的に起動して、アプリケーションの開始ページを開きます。

## Web モジュールおよび J2EE アプリケーション以外のコンポーネントの配備

Web モジュールと J2EE アプリケーションだけが、サーバーに実際に配備して実行できる項目です。ただし、開発中のビジネスロジックの小さなユニットのテストが必要な場合があります。Sun ONE Studio 4 IDE では、ビジネスロジックのより小さいユニットを含むモジュールとアプリケーションを作成することによって、これらのコンポーネントを配備して実行することができます。また、いくつかのタイプのコンポーネントに対して、テストクライアントを生成することもできます。これらの機能については、『Web コンポーネントのプログラミング』 と 『Enterprise JavaBeans コンポーネントのプログラミング』を参照してください。

# 索引