J2EE™ アプリケーションの プログラミング

Sun Java™ Studio Enterprise 7 2004Q4

Sun Microsystems, Inc.
www.sun.com

Part No. 819-1298-10
2004 年 12 月, Revision A

Copyright 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements. Use is subject to license terms.

この製品には第三者によって開発された成果物が含まれている場合があります。フォントテクノロジを含むサードパーティ製のソフトウェアの著作権およびライセンスは、Sun Microsystems, Inc. のサプライヤが保有しています。

Sun、Sun Microsystems、Sun のロゴ、Java、JavaHelp、docs.sun.com、および Solaris は、米国および他の各国における Sun Microsystems, Inc. の商標または登録商標です。すべての SPARC の商標はライセンス規定に従って使用されており、米国および他の各国における SPARC International, Inc. の商標または登録商標です。SPARC の商標を持つ製品は、Sun Microsystems, Inc. によって開発されたアーキテクチャに基づいています。

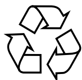
UNIX は、X/Open Company Limited が独占的にライセンスしている米国ならびに他の国における登録商標です。

本製品はライセンス規定に従って配布され、本製品の使用、コピー、配布、逆コンパイルには制限があります。本製品のいかなる部分も、その形態および方法を問わず、Sun Microsystems, Inc. およびそのライセンサーの事前の書面による許可なく複製することを禁じます。

本製品は、米国輸出管理法の対象となっています。また、他国においても輸出入管理法規の対象となっている場合があります。お客様は、それらのすべての法令および規制を厳守することに同意し、納品後に輸出、再輸出、または輸入の許可が必要となった場合には、お客様にそれらを取得する責任があるものとします。本製品を米国輸出規制法に指定されている各国または団体に提供することを禁じます。お客様は、本ソフトウェアが、核施設の設計、建設、運転または保守で使用するように設計、ライセンス、および意図されていないことを認識するものとします。Sun Microsystems, Inc. は、そのような目的の適合性に関して、明示的、黙示的を問わずいかなる保証も致しません。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含み、明示的であるか黙示的であるかを問わず、あらゆる説明および保証は、法的に無効である限り、拒否されるものとします。

原典*:* *Sun Java Studio Enterprise 7 2004Q4 Building J2EE Applications*
Part No: 819-0819-10
Revision A



目次

はじめに　xi

1. アセンブル、配備、実行の基本　1

アセンブルの基本　1

J2EE アプリケーションはモジュール構造である　2

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　3

J2EE アプリケーションは分散型である　6

モジュールとアプリケーションの表示　9

Web モジュール　10

EJB モジュール　11

J2EE アプリケーション　12

プロパティシート　12

配備の基本　13

実行の基本　16

このマニュアルの使用法　16

2. シナリオ: Web モジュール　19

モジュール内での対話　20

モジュールのプログラミング　22

開始ページの作成　23

サーブレットメソッドのプログラミング 26
URL からサーブレットへのマッピング 31
その他のアセンブル作業 34

3. シナリオ: EJB モジュール 41
モジュール内での対話 42
モジュールのプログラミング 44
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 45
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 46
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 47
EJB モジュールのアセンブル 50

4. シナリオ: Web モジュールと EJB モジュール 61
アプリケーション内での対話 62
アプリケーションのプログラミング 63
J2EE アプリケーションの作成 63
Web モジュールの Web コンテキストの設定 65
EJB 参照のリンク 67
その他のアセンブル作業 69

5. 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 73
アプリケーション内での対話 74
メッセージ駆動型通信のプログラミング 75
アプリケーションサーバーの設定 75
Web モジュールのプログラミング 78
EJB モジュールのプログラミング 84
J2EE アプリケーションのアセンブル 87

6. トランザクション 89
デフォルトトランザクション境界 89

トランザクション境界の再定義　91

7. セキュリティ　95

Web モジュールのセキュリティ　96

EJB モジュールのセキュリティ　104

J2EE アプリケーションのセキュリティ　110

8. J2EE モジュールと J2EE アプリケーションの配備と実行　113

「エクスプローラ」ウィンドウでのサーバーの表示　113

サーバーレジストリノード　114

「インストールされているサーバー」ノード　115

サーバー製品ノード　115

Sun Java System Application Server ノード　115

デフォルトサーバーノード　119

サーバー固有のプロパティ　119

サーバーインスタンスノードを使用した配備と実行　120

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　123

配備プロセス　123

サーバープラグインの概念　124

配備プロセス　125

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　126

索引　127

図目次

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　7

図 1-2　Web モジュールノードとサブノード　11

図 1-3　EJB モジュールノードとサブノード　12

図 1-4　J2EE アプリケーションノードとそのサブノード　12

図 1-5　「実行時」ウィンドウ　15

図 2-1　`CatalogWebModule` Web モジュール　19

図 2-2　「開始ファイル」プロパティエディタ　25

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　30

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　31

図 2-5　「サーブレットマッピング」プロパティエディタ　33

図 2-6　「サーブレットマッピング」プロパティエディタ　34

図 2-7　「エラーページ」プロパティエディタ　35

図 2-8　「JSP ファイル」プロパティエディタ　37

図 2-9　「サーブレットマッピング」プロパティエディタ　38

図 2-10　「追加 環境エントリ」ダイアログ　40

図 3-1　`CatalogData` EJB モジュール　41

図 3-2　「追加 EJB ローカル参照」ダイアログ　49

図 3-3　EJB モジュールの「CMP リソース」プロパティエディタ　54

図 3-4　「追加 リソース参照」ダイアログ　56

図 3-5　「追加 リソース参照」ダイアログのサーバー固有タブ　57

図 4-1 CatalogApp J2EE アプリケーション 61
図 4-2 CatalogWebModule のプロパティシート 66
図 4-3 リンクされていない「EJB 参照」 68
図 4-4 上書きによってリンクされた「EJB 参照」 69
図 4-5 J2EE アプリケーションの「環境エントリ」プロパティエディタ 70
図 4-6 環境エントリ値の上書き 71
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 73
図 5-2 CheckoutQueue のリソース環境参照の追加 82
図 5-3 キュー参照の JNDI 名の指定 82
図 5-4 キュー接続ファクトリのリソース参照 83
図 5-5 キュー接続ファクトリ参照の JNDI 名 84
図 5-6 「メッセージ駆動型送信先」プロパティシート 85
図 5-7 メッセージ駆動型 Bean の「MDB 接続ファクトリ」プロパティエディタ 86
図 6-1 デフォルトのトランザクション属性 90
図 6-2 複雑なトランザクション 92
図 6-3 変更後のトランザクション設定 94
図 7-1 Web モジュールに宣言されたセキュリティロール ME と EveryoneElse 97
図 7-2 allItems という名前の Web リソースの定義 99
図 7-3 「追加 セキュリティ制限」ダイアログの allItems リソース 100
図 7-4 allItems という名前の Web リソースに対する制限の指定 101
図 7-5 ロール ME にマップされたセキュリティロール参照 roleRefMe 103
図 7-6 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 105
図 7-7 EJB の「メソッドのアクセス権」プロパティエディタ 106
図 7-8 セキュリティロール参照 everyOne の宣言 108
図 7-9 EJB モジュールの everyOne セキュリティロール参照 109
図 7-10 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 110
図 7-11 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に表示されたセキュリティロール 111
図 7-12 ロール ME にマップされたロール myself 112
図 8-1 サーバーレジストリノード 114

図 8-2 EJB モジュールの Sun Java System AS セクション 120
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 124

はじめに

Java Community Process は、Sun Microsystems, Inc. のサポートを得て Java™ 2 Platform, Enterprise Edition (J2EE™ platform) 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。xii ページの「お読みになる前に」に示されている J2EE プラットフォームのマニュアルは、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。

このマニュアルでは、Sun™ Java Studio Enterprise 7 2004Q4 開発者ツールを使用したこれらのアーキテクチャの実装方法を説明しています。統合開発環境 (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。さらに、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も説明します。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。

画面イメージはプラットフォームによってやや異なります。ほぼすべての手順で Java Studio Enterprise ソフトウェアのインタフェースを使用しますが、コマンド行からコマンドを入力することもあります。このコマンドも、プラットフォームによってやや異なります。たとえば Microsoft Windows コマンドは以下のようになります。

```
c:\>cd MyWorkDir\MyPackage
```

UNIX® コマンドの場合は、以下のようになります。

```
% cd MyWorkDir/MyPackage
```

お読みになる前に

このマニュアルは、Java Studio Enterprise 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
- Enterprise JavaBeans™ (EJB™) テクノロジの概念
- J2EE アプリケーションのアセンブルと配備の概念

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
  `http://java.sun.com/reference/blueprints`
- Java 2 Platform, Enterprise Edition Specification
  `http://java.sun.com/j2ee/download.html#platformspec`
- J2EE Tutorial
  `http://java.sun.com/j2ee/learning/tutorial`
- Java Servlet Specification バージョン 2.3
  `http://java.sun.com/products/servlet/download.html#specs`
- JavaServer Pages Specification バージョン 1.2
  `http://java.sun.com/products`

XML ベースの RPC (JAX-RPC) の Java API の知識があると役立ちます。詳細については、以下の Web ページを参照してください。

`http://java.sun.com/xml/jaxrpc`

注 – Sun では、本マニュアルに掲載されている第三者の 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も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、サービスの利用あるいはそれらのものを信頼することによって、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。

マニュアルの構成

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を可能にします。アプリケーション開発者は、ビジネスロジックを 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します。コンポーネントを作成したら、これらをアセンブルして、コンポーネントの集合体であるモジュールを作成します。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。アセンブルしてモジュールを作成し、さらに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ます。J2EE アプリケーションはビジネスプロセス全体を実行します。

このマニュアルは、Java Studio Enterprise 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。このマニュアルでは、一連の「シナリオ」の中でこれを説明していきます。

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 ユニットを定義し、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 章では、Web モジュールのアセンブル方法を示すシナリオについて説明します。この章では、J2EE アプリケーションのフロントエンドとして使用される Web モジュールについて簡単に説明します。その後で Web モジュールのプログラミング方法を説明します。

第 3 章では、EJB モジュールのアセンブル方法を示すシナリオについて説明します。この章では、J2EE アプリケーションで使用される EJB モジュールについて簡単に説明します。その後で EJB モジュールのプログラミング方法を説明します。

第 4 章では、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について説明します。この章では、Web モジュールと EJB モジュールを組み合わせた J2EE アプリケーションについて簡単に説明します。その後でアプリケーションをアセンブルする方法を説明します。特に、Java RMI (Remote Method Invocation) 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 章では、メッセージ駆動型エンタープライズ Bean (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について説明します。この章では、ビジネスアプリケーションで使用される非同期通信について簡単に説明します。その後でアプリケーション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、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 7 章では、IDE を使って 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 モジュール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、モジュールをアセンブルし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 8 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、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機構を説明します。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、コード例。	`.login` ファイルを編集します。 `ls -a` を実行します。 `% You have mail.`
`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します。	`%` **`su`** `Password:`
AaBbCc123 またはゴシック	コマンド行の可変部分。実際の名前や値と置き換えてください。	`rm` *filename* と入力します。 `rm` **ファイル名** と入力します。
『 』	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。 この操作ができるのは「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅をこえる場合に、継続を示します。	`%` **`grep '^#define \`** **`XV_VERSION_STRING'`**

* 使用しているブラウザにより、これら設定と異なって表示される場合があります。

関連マニュアル

Java Studio Enterprise のマニュアルとしては、Acrobat Reader (PDF) 形式のマニュアル、チュートリアルと、HTML 形式のリリースノート、オンラインヘルプ、チュートリアルが提供されています。

オンラインで入手可能なマニュアル

ここで紹介しているマニュアルは、docs.sun.com[SM] Web サイトおよび Sun Java Studio Enterprise Developers Source ポータルサイト (http://developers.sun.com/jsenterprise) のドキュメントリンクから入手できます。

docs.sun.com Web サイト (http://docs.sun.com) では、インターネットで Sun のマニュアルを参照、印刷、購入することができます。

- 『Sun Java Studio Enterprise 7 2004Q4 リリースノート』- Part No. 819-1302-10

  最新のリリースの変更点や技術的な注意事項を説明しています。

- 『Sun Java Studio Enterprise 7 インストールガイド』(PDF 形式) - Part No. 819-1300-10

  サポートしている各プラットフォームへの Sun Java Studio Enterprise 7 統合開発環境 (IDE) のインストール方法を説明しています。システム要件やアップグレード方法、サーバー情報、コマンド行スイッチ、インストールされるサブディレクトリ、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の関連情報も記載されています。

- 『J2EE アプリケーションのプログラミング』- Part No. 819-1298-10

  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。

- Web アプリケーションフレームワークのマニュアル (PDF 形式)
  - 『Web アプリケーションフレームワーク コンポーネント作成ガイド』- Part No. 819-1284-10

    Web アプリケーションフレームワークのコンポーネントアーキテクチャと新しいコンポーネントの設計、作成、配布工程を説明しています。

  - 『Web アプリケーションフレームワーク コンポーネントリファレンスガイド』- Part No. 819-1286-10

    Web アプリケーションフレームワークライブラリに提供されているコンポーネントを説明しています。

  - 『Web アプリケーションフレームワーク 概要』- Part No. 819-1288-10

    Web アプリケーションフレームワークとその位置づけ、仕組み、他のアプリケーションフレームワークと異なる点を説明しています。

  - 『Web アプリケーションフレームワーク チュートリアル』- Part No. 819-1290-10

    Web アプリケーションフレームワークを使用して Web アプリケーションを構築する際の仕組みとその手法を紹介しています。

- 『Web アプリケーションフレームワーク 開発ガイド』- Part No. 819-1292-10

  Web アプリケーションフレームワークを使用し、開発するアプリケーションの構成要素として使用可能なアプリケーションコンポーネントの作成および使用の手順と、そのアプリケーションを大部分の J2EE コンテナに配備する方法を説明しています。

- 『Web アプリケーションフレームワーク IDE ガイド』- Part No. 819-1294-10

  Sun Java Studio Enterprise 7 2004Q4 IDE の各部の概要、および Web アプリケーションフレームワークアプリケーションを開発するためのビジュアルツールの使用方法を重点的に説明しています。

- 『Web アプリケーションフレームワーク タグライブラリリファレンス』- Part No. 819-1296-10

  Web アプリケーションフレームワークのタグライブラリを簡単に紹介し、タグライブラリに提供されているタグに対する包括的な参照を示しています。

- Sun Java System Web Server 6.1 のマニュアル

  「Getting Started」、「Installation and Migration」、「Administrator's Guides」を含む Sun Java System Web Server 6.1 のマニュアルは、`http://docs.sun.com/db/prod/s1websrv#hic` で入手できます。

チュートリアル

Sun Java Studio Enterprise 7 には、IDE の機能を理解する手助けとなるチュートリアルがいくつか用意されています。これらのチュートリアルにある技術、およびコード例は、そのまま、または編集を加えて、実際のアプリケーションの開発に利用することができます。すべてのチュートリアルで、Sun Java System Application Server への配備例が紹介されています。

チュートリアルは、すべて Developers Source ポータルのリンク「Tutorials & Code Camps」から利用可能です。IDE で「ヘルプ」>「コードサンプルとチュートリアル」>「概要」を選択すると、このサイトにアクセスできます。

- 「クイックスタートガイド」は、Sun Java Studio IDE の紹介をしています。チュートリアルは、Sun Java Studio を初めてご使用になる方や、特定の機能について早く知りたい場合は、このガイドから始めてください。これらのチュートリアルは、単純な Web アプリケーションや J2EE アプリケーションの開発方法、Web サービスの生成方法を説明しています。また、UML モデリング、リファクタリングの導入方法についても説明しています。ガイドを終えるための所要時間は数分です。
- 「チュートリアル」は、Sun Java Studio IDE の特定の 1 つの機能に焦点を当てています。ある機能の詳細に関心がある場合は、これらを実行してみてください。例で説明している機能によって、初めからアプリケーションを構築する場合と、提供されたソースファイルを使用して構築する場合があります。チュートリアルは 1 時間以内で完成できます。

- 「概要ビデオ」は、技術の説明がビデオで提供されています。IDE の視覚的な概要や、特定の機能の詳細説明を見ることができます。概要ビデオにかかる時間は数分です。概要ビデオは、任意の個所で開始、終了することもできます。

オンラインヘルプ

Sun Java Studio Enterprise 7 IDE には、オンラインヘルプが用意されています。ヘルプキー (Microsoft Windows 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」>「ヘルプ (すべて)」を選択して開くことができます。ヘルプの項目と検索機能が表示されます。

アクセシブルな製品マニュアル

マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセシブルなマニュアルは以下の表に示す場所から参照することができます。

マニュアルの種類	アクセシブルな形式と格納場所
マニュアルとチュートリアル	形式 : HTML 場所 : `http://docs.sun.com`
チュートリアル	形式 : HTML 場所 : Developers Source ポータル (`http://developers.sun.com/jsenterprise`) のリンク「Examples & Code Camps」
リリースノート	形式 : HTML 場所 : `http://docs.sun.com`

第 1 章

アセンブル、配備、実行の基本

Java 2 Platform, Enterprise Edition (J2EE プラットフォーム) の開発はモジュール方式です。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。小さな J2EE ソフトウェア単位を組み合わせて大きな単位を作成することを「アセンブル」と呼びます。

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。モジュールやアプリケーションをアセンブルする際は、どの実行時サービスが必要かを判断する必要があります。これらのサービスは J2EE 配備記述子で指定します。

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説明します。また、Java Studio Enterprise 開発者ツール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

J2EE アセンブルは、多くの独立した作業を含む工程です。正しくアセンブルしたアプリケーションやモジュールは、J2EE アプリケーションサーバーに配備して実行できます。

アセンブルを成功させる上で最も大きな障害は、アセンブル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。このため、必要なアセンブル作業も異なってきます。モジュールやアプリケーションをアセンブルするための標準的な手順はありません。アセンブル作業を開始する前に、正しくアセンブルされたモジュールやアプリケーションについて理解しておく必要があります。この節では、正しくアセンブルされたモジュールやアプリケーションの理解に役立つ、J2EE プラットフォームに関する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、一連のモジュールから構成されます。アプリケーション内のモジュールは、一連のコンポーネントから構成されます。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 プラットフォームの仕組みが配備記述子です。配備記述子は、モジュールやアプリケーションの「構成要素のリスト」です。

- アプリケーションの配備記述子は、アプリケーション内のモジュールを表します。
- モジュールの配備記述子は、モジュール内のコンポーネントを表します。

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配備され、実行されるのかを考えてみましょう。開発時には、コンポーネントは多数のソースファイルとして開発環境内に存在しています。J2EE アプリケーションサーバーに配備されるまで、これらのソースファイルを実行することはできません。コンポーネントは、アプリケーションサーバーが提供する実行時環境で実行される必要があります。

アプリケーションを配備することにより、アプリケーションの配備記述子にリストされてい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。ソースファイルは実際には、配備時に J2EE アプリケーションにコンパイルされます。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。

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リストするための開発時の仕組みです。開発時には、モジュールまたはアプリケーションを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。配備プロセス用のモジュールやアプリケーションを示す配備記述子を準備します。

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、CatalogApp という J2EE アプリケーションの配備記述子の例です。この配備記述子は、CatalogApp アプリケーション内のモジュールを表しています。

コード例 1-1　　CatlaogApp の配備記述子

```
<!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>CatalogApp</display-name>
 <description>J2EE Application CatalogApp</description>
 <module>
  <ejb>CatalogData.jar</ejb>
  <alt-dd>CatalogData.xml</alt-dd>
 </module>
```

コード例 1-1　　CatlaogApp の配備記述子 (続き)

```
  <module>
   <web>
    <web-uri>CatalogWebModule.war</web-uri>
    <context-root>catalog</context-root>
   </web>
   <alt-dd>CatalogWebModule.xml</alt-dd>
  </module>
</application>
```

CatalogApp アプリケーションには、CatalogData および CatalogWebModule という 2 つのモジュールが含まれています。配備記述子は、<module> タグによってモジュールを識別します。

CatalogApp 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はアプリケーションの配備記述子を読み取ります。CatalogApp 配備記述子にリストされている各モジュールは、それぞれモジュールレベルの配備記述子を持っています。アプリケーションサーバーは、この 2 つのモジュールの配備記述子を読み取ります。これらの配備記述子は、2 つのモジュールの J2EE コンポーネントのソースファイルを識別します。コード例 1-2 と コード例 1-3 は、モジュールレベルの配備記述子を示しています。

Java Studio Enterprise 統合開発環境 (IDE) で作業を行うと、IDE によって配備記述子が準備されます。開発者が自分で配備記述子タグを記述することはありませんが、IDE で作業をしているときに IDE によって配備記述子が準備されることを理解してください。

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

J2EE アプリケーションは、実行時に J2EE アプリケーションサーバーが提供するサービスを使用します。このような実行時サービスには、コンテナ管理による持続性、コンテナ管理によるトランザクション、およびコンテナ管理によるセキュリティ妥当性検査があります。

アプリケーションとアプリケーション内のモジュール群は、必要なサービスをアプリケーションサーバーに伝える必要があります。必要なサービスをアプリケーションサーバーに伝える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。コンテナ管理のトランザクションサービスを使用するには、必要なトランザクションサービスを J2EE アプリケーションサーバーに伝える必要があります。エンタープライズ Bean をアセンブルして Enterprise JavaBeans (EJB) モジュールを作成するときに、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トランザクション境界を定義します。IDE は、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の値を配備記述子に含めます。

コード例 1-2 は、CatalogData という EJB モジュールの配備記述子を示しています (CatalogData モジュールは コード例 1-1 にリストされている 2 つのモジュールの 1 つです)。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>
   <persistence-type>Container</persistence-type>
   <prim-key-class>java.lang.String</prim-key-class>
   <reentrant>False</reentrant>
   <abstract-schema-name>ItemBean</abstract-schema-name>
```

コード例 1-2　　EJB モジュール配備記述子 (続き)

```
    <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 Java Studio
Enterprise.</description>
    <method>
     <ejb-name>CatalogManagerBean</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 Java Studio
Enterprise.</description>
```

コード例 1-2　　EJB モジュール配備記述子 (続き)

```
     <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 Java Studio
Enterprise.</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を配備して実行すると、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、指定された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。

ほかの実行時サービスも、コンテナ管理によるトランザクションとほぼ同じように処理されます。各サービスは、必要なサービスを正確に示す配備記述子タグをそれぞれ持っています。これらのタグは IDE によって自動的に記述されるため、開発者が配備記述子で使用されるタグについて学習する必要はありません。

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、J2EE プラットフォームの実行時サービスを利用することに加え、分散型です。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、2 つのモジュールから構成されるアプリケーションを示します。このアプリケーションは、典型的な多層アプリケーションアーキテクチャを実装しています。

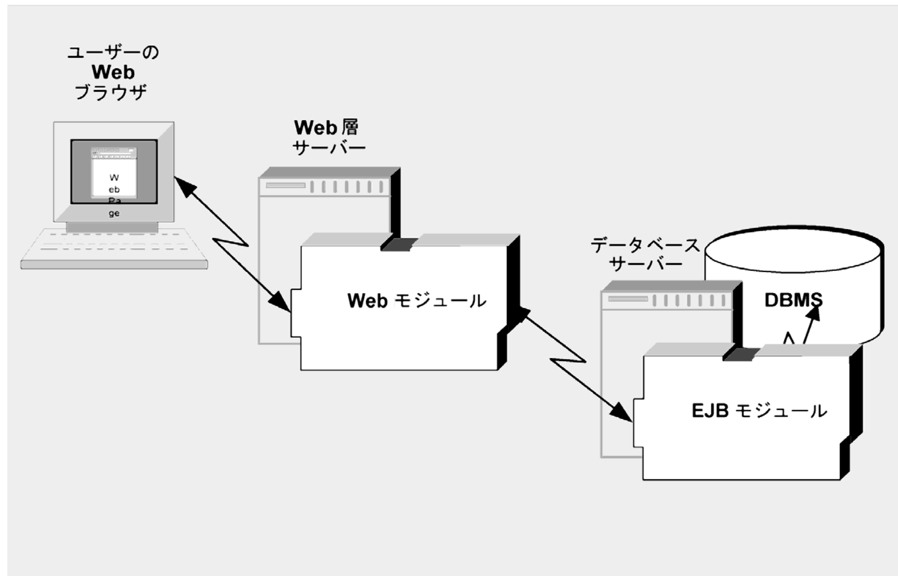


図 1-1　　J2EE コンポーネントとモジュールを使用した多層アプリケーション

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 接続を提供します。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 ページをユーザーのデスクトップマシンのブラウザに送信します。

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 モジュールは Java RMI (Remote Method Invocation) を使用して EJB モジュールと通信します。アプリケーションサーバーは、Java RMI 対話の実行時サポートを提供します。

J2EE プラットフォームは、モジュール間の分散型対話をサポートするいくつかのテクノロジを提供します。J2EE プラットフォームの分散型テクノロジには、以下があげられます。

- **HTTP 接続上での Web ベースの通信。**このテクノロジは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ます。
- **Java RMI-IIOP (Remote Method Invocation for Internet Inter-ORB Protocol) を使用した同期メソッド呼び出し。**このテクノロジはエンタープライズ Bean メソッドの呼び出しに使用されます。
- **JMS (Java Message Service) 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ます。

J2EE プラットフォームは、さまざまな対話をサポートする各種コンポーネントを提供します。たとえば、J2EE プラットフォームはメッセージ駆動型エンタープライズ Bean を使用して、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。

アプリケーションの分散型対話に使用するテクノロジの決定は、アプリケーション設計の 1 つの作業です。使用するコンポーネントの種類の決定も、設計作業の 1 つとなります。ただし、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たかを知る必要があります。対話の実装は、EJB 参照の設定 (Java RMI による対話を実装する場合) やキューの設定 (JMS メッセージングによる対話を実装する場合) などのアセンブル作業を実行することで行います。

また、J2EE プラットフォームは、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 J2EE テクノロジには、次のものが含まれます。

- JDBC (Java DataBase Connectivity) テクノロジ
- コンテナ管理による持続性

アプリケーションをアセンブルするときは、アプリケーションが使用する外部リソースが識別されていることを確認します。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開発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。このモジュールと CatalogData という EJB モジュールがアセンブルされて、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 の対話は、コード例 1-3 の最後の <ejb-ref> タグによって宣言されているリモート EJB 参照を必要とします。

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>
  <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>
```

コード例 1-3　　Web モジュール配備記述子 (続き)

```
    <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>
```

Java Studio Enterprise IDE で作業を行う場合、モジュールやアプリケーションの配備記述子をコーディングするのは開発者ではありません。開発環境では、IDE によって視覚的に表現されたコンポーネント、モジュール、アプリケーションを使って作業を行います。配備記述子は、IDE によって準備されます。

モジュールとアプリケーションの表示

多くの場合、J2EE アセンブル手順の説明は、コード例 1-1、コード例 1-2、およびコード例 1-3 のような配備記述子ファイルの内容に重点が置かれますが、これらの説明は XML のコーディング方法を示しています。Java Studio Enterprise IDE では、コンポーネント、モジュール、およびアプリケーションのイメージ (ノード) を提供しているため、開発者は配備記述子ファイルをコーディングするかわりに、コンポーネント、モジュール、アプリケーションを表す「ファイルシステム」ウィンドウノードを操作します。

IDE でのアセンブル時には、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「ファイルシステム」ウィンドウに表示されます。アプリケーションを表すノードには、そのモジュールを表すサブノードがあります。また、モジュールを表すノードに

は、そのコンポーネントを表すサブノードがあります。「ファイルシステム」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。プロパティのほとんどは、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配備のために設定すると、IDE によってその配備記述子にタグが追加されます。これらのタグによって、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、「ファイルシステム」ウィンドウにこの構造が示されます。図 1-2 は、「ファイルシステム」ウィンドウに表示された Web モジュールを示しています。Web モジュールのノードとサブノードは、モジュール内の個々のディレクトリとファイルを表します。

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「ファイルシステム」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。「ファイルシステム」ウィンドウで Web モジュールディレクトリを別のファイルシステムのサブディレクトリとしてマウントした場合、IDE はその Web モジュールディレクトリを Web モジュールとして認識しません。

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、Sun 以外のライブラリやタグライブラリなど、Java アーカイブ (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、サーブレットなど、`.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 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。

Web モジュールには、開発者が追加したコンポーネントやリソースを示すノードもあります。図 1-2 は、`index.html` という HTML ページのノードを示しています。クラスディレクトリには、`AllItemsServlet` および `ItemDetailServlet` という 2 つのサーブレットクラスのノードが含まれています。

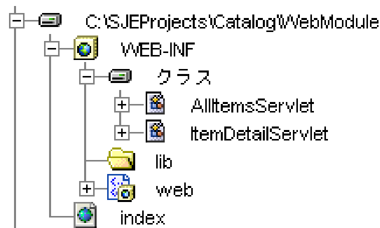


図 1-2 Web モジュールノードとサブノード

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。配備記述子は `web.xml` というファイルで、web ノードによって「ファイルシステム」ウィンドウに表示されます。`web.xml` ファイルは、Web モジュールのソースコードの一部です。

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。EJB モジュールノードは、エンタープライズ Bean のリストとして機能します。これらのエンタープライズ Bean 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ものです。配備記述子を表す最上位ノードは、コンポーネントのソースコードのある場所を追跡します。

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、配備記述子の XML 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コンパイルされて、EJB JAR ファイルが生成されます。

図 1-3 は、「ファイルシステム」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、異なるディレクトリ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。

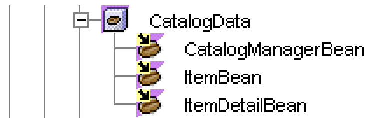


図 1-3　EJB モジュールノードとサブノード

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。このアプリケーションノードはアプリケーション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ます。同じソースコードを複数の J2EE アプリケーションで使用することができます。配備プロセスでは、すべてのソースファイルがコンパイルされ、コンパイル後のファイルがエンタープライズアーカイブ (EAR) ファイルの配備記述子に関連付けられます。

図 1-4 は、「ファイルシステム」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 に示されたモジュールがアプリケーションに追加されています。これらのモジュールは、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。

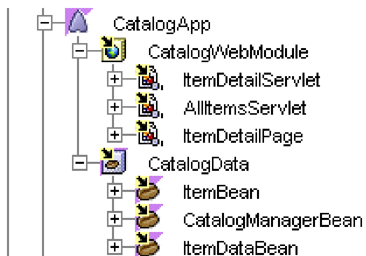


図 1-4　J2EE アプリケーションノードとそのサブノード

プロパティシート

J2EE モジュール、J2EE アプリケーション、および EJB 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て、モジュールやアプリケーションに必要なアプリケーションサーバーのサービスを指定できます。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化したりする必要がありません。

- 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、このファイルは「ファイルシステム」ウィンドウに表示されます (図 1-2)。Web モジュールの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、モジュールやアプリケーションを配備するまで配備記述子ファイルは生成されません。配備プロセスではソースファイルをコンパイルして、アーカイブファイル EJB JAR または EAR を生成します。配備記述子ファイルはアーカイブに含まれます。配備記述子は、ソースファイルの特定のアーカイブコピーに関連付けられます。

プロパティシートを表示する方法は、いくつかあります。

- 目的のノードを右クリックし、「プロパティ」を選択する。
- 「ウィンドウ」>「プロパティ」をクリックする。強調表示されているノードのプロパティシート情報がウィンドウに表示されます。

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、そのプロパティの省略符号ボタン (...) をクリックすると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、いくつかのセクションがあります。「プロパティ」という名前のセクションには、J2EE 仕様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。その他のセクションには、アプリケーションサーバー製品の名前が付いています。これらのセクションは、J2EE 仕様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セクションと呼びます。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を設定します。

配備の基本

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。配備は、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。

IDE はアプリケーションサーバーと通信します。IDE は、モジュールやアプリケーションをアプリケーションサーバーに配備して、そのモジュールやアプリケーションを実行するためのコマンドを提供します。配備や実行は、IDE で作業を行いながら実行できます。配備はプラグインや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。ただし、配備と実行は IDE で管理します。

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」コマンドを選択します。しかし、配備する前に、IDE が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。アプリケーションサーバーの設定手順を以下にまとめます。

アプリケーションサーバーをインストールします。

1. 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サーバープラグインをインストールします。Sun Java System Application Server Standard Edition 7 2004Q2 のプラグインは IDE とともにインストールされます。ほかにも、一般に普及しているアプリケーションサーバー製品用のプラグインが提供されています。プラグインをインストールすると、図 1-5 のように、「実行時」ウィンドウにアプリケーションサーバーノードが作成されます (プラグインの詳細は、付録 A を参照してください)。

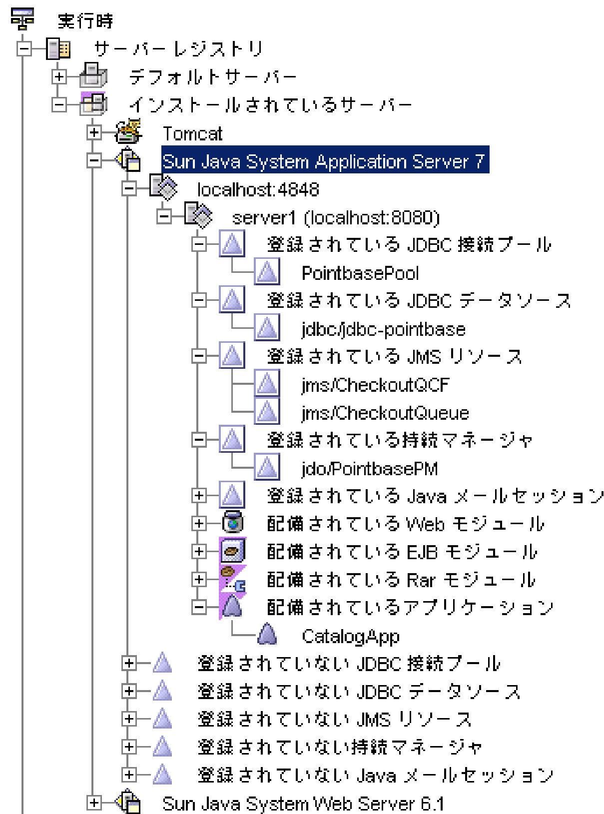


図 1-5　「実行時」ウィンドウ

2. アプリケーションサーバーノードのメニューコマンドを使用して、IDE とアプリケーションサーバーの間の通信を確立します。

3. アプリケーションのプロパティシートを使用して、アプリケーションを配備するアプリケーションサーバーを指定します。

4. アプリケーションノードを右クリックし、「配備」コマンドを選択します。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配備されます。

IDE のほとんどのインストールには Sun Java System Application Server Standard Edition 7 2004Q2 が含まれており、インストーラによってアプリケーションサーバーと連携するように IDE が自動的に構成されます。この詳細は、『Sun Java System Application Server 7 入門ガイド』を参照してください。

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、アプリケーションを表すファイルシステムノードと配備されたアプリケーションを関連付けます。ノードを右クリックし、「実行」コマンドを選択すると、配備済みのアプリケーションを実行するように IDE がアプリケーションサーバーに指示します。

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タイプ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 モジュールを含む配備済みの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 Web ページの 1 つを開きます。

このマニュアルの使用法

Java Community Process は、米国 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発展させました。xii ページの「お読みになる前に」に示されている J2EE プラットフォームのマニュアルは、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、Java Studio Enterprise 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も説明します。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してアセンブルについて説明しています。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にアセンブル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完全な設計ガイドではありません。

シナリオの目的は、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の対話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対話が含まれる可能性があります。対話に関しては、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 6 章、セキュリティについては第 7 章を参照してください。

このマニュアルは、Java Studio Enterprise 開発環境を使用して分散型エンタープライズアプリケーション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 モジュールとそれが関与する対話を示しています。このモジュールは Java 2 Platform, Enterprise Edition (J2EE プラットフォーム) アプリケーションの一部で、図に示された対話は J2EE アプリケーションの Web モジュールの典型的な対話です。Web モジュールは HTTP 接続を通してユーザーとの対話を行い (図の矢印「1」)、Enterprise JavaBeans (EJB) モジュールが提供する中間層サービスとの対話を行います (図の矢印「3」)。Web モジュール内では、Web コンポーネントが相互に対話を行います (図の矢印「2」)。この章ではこの Web モジュールについて説明し、図 2-1 に示された対話のプログラミング方法を解説します。

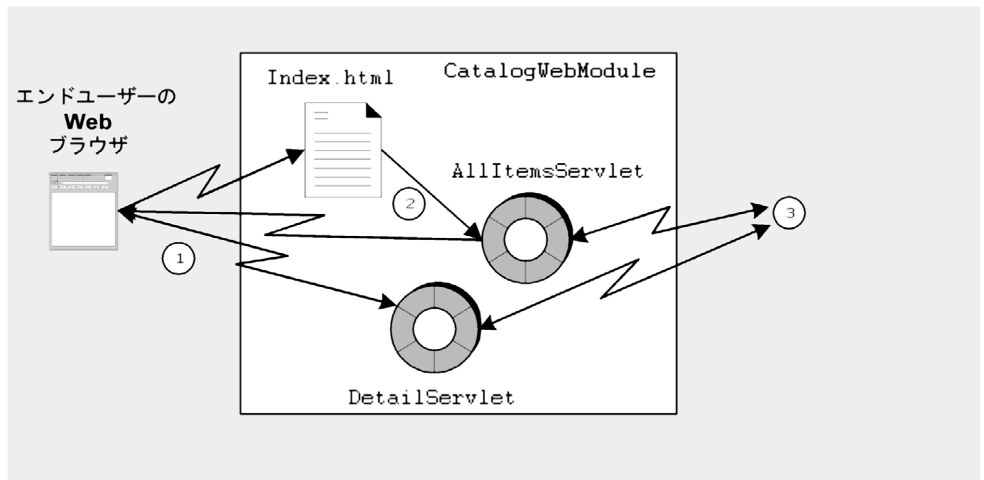


図 2-1 CatalogWebModule Web モジュール

モジュール内での対話

このシナリオでは、図 2-1 に示された Web モジュールと対話の使用例について説明します。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フロントエンドです。この Web モジュールには、サイトを訪れたオンラインショッピングの利用者 (買い物客) に対して表示される Web ページが含まれています。また、これらのページに対する利用者の入力をこのモジュールによって処理します。ページの表示や入力の処理は、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、このアプリケーションは一連の Web ページです。利用者は Web ブラウザを使ってアプリケーションのホームページを表示します。利用者は、テキストフィールドやボタンなど Web ページに表示されるコントロールを使って入力を行います。開発者の側から見れば、アプリケーションは HTTP 要求を受け取って HTTP 応答を返す Web コンポーネント群です。

このシナリオの Web モジュールには数個のコンポーネントしか含まれておらず、2 種類の要求だけを処理します。これらのコンポーネントや要求は単純なものではありますが、ユーザー、Web モジュール、および 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提供するかを示しています。このシナリオで扱っている特定の対話の概要を、以下に示します。

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。この操作によってアプリケーションのホームページが開きます。現実のショッピングサイトのホームページには、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。しかし、この例では、ホームページは単純に 1 つのオプション、つまりカタログ全体を表示する別のページへのリンクだけを表示します。

2. 利用者は、リンクをクリックします。この操作は要求を生成し、この要求はその中で指定された `AllItemsServlet` サーブレットで処理されます。`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。このメソッドがデータを返します。

3. `AllItemsServlet` は、利用者のブラウザに表示するために EJB モジュールが返すデータを準備します。サーブレットは、EJB モジュールから戻された個々のフィールド値を、フィールド値を書式化する HTML タグと組み合わせます。`AllItemsServlet` はフィールド値と HTML タグのこの組み合わせを出力ストリームに書き込み、これがユーザーの Web ブラウザに戻されます。利用者の Web ブラウザはこの HTML 出力を処理してカタログを表示します。カタログ表示画面では、それぞれの商品名がその商品の詳細情報への HTML リンクとして表示されます。

4. 利用者は表示されたカタログを閲覧し、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、利用者の Web ブラウザに表示するために EJB モジュールが返すデータを処理します。`AllItemsServlet` と同様に、`DetailServlet` はフィールド値と HTML タグを組み合わせた出力ストリームを準備します。利用者の Web ブラウザはこの出力ストリームを処理して、別の Web ページにこれを表示します。

`AllItemsServlet` と `DetailServlet` 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 コンポーネントが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んで HTTP 要求を処理す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、この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 モジュールと EJB モジュールの間の対話は、Java RMI (Remote Method Invocation) メソッド呼び出しによって実装されます。Java RMI は、EJB モジュールの設計で必要とされます。EJB モジュール設計の詳細は、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 に、前述の節と 図 2-1 に示された Web 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 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 開始ページ `index.html` の作成。このページには、サーブレット `AllItemsServlet` を実行する HTML リンクが含まれています。 `AllItemsServlet` と `ItemDetailServlet` の 2 つのサーブレットを作成します。 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 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 (Java Naming and Directory Interface) ルックアップを実行します。 2. EJB 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 `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 各サーブレットの URL パターンを設定します
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は、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

ショッピング Web サイトは、利用者がホームページを呼び出すとサイトとの対話が始まるように設計されています。Web サイト特有の機能の 1 つであるホームページは、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。利用者はホームページを表示して、使用したい機能を選択します。

利用者の最初の操作は、ブラウザを起動してショッピングサイトの URL を開くことです。Web モジュール内では、この URL はアプリケーションのコンテキストルートにマップされています。ユーザーがサイトの URL を開いたときは常にアプリケーションでホームページが表示されるようにしたい場合は、ページを開始ページに指定します。

HTML ページの作成

「ファイルシステム」ウィンドウのノード階層で、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。この例については、図 1-2 を参照してください。

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」>「すべてのテンプレート」を選択します。**

   「新規ウィザード」の「テンプレートを選択」ページが開きます。

2. **「HTML ファイル」テンプレートを選択します。**

   a. **「テンプレートを選択」フィールドで、「JSP & サーブレット」ノードを展開して「HTML ファイル」ノードを選択します。**

   b. **「次へ」をクリックします。**

   「新規オブジェクト名」ページが開きます。

3. **名前フィールドに「`index`」と入力します。「完了」をクリックします。**

   統合開発環境 (IDE) によって新しい HTML ファイルが作成され、`index` という名前の新しいノードが「ファイルシステム」ウィンドウに表示されます。ソースエディタが開き、新しいファイルにカーソルが置かれます。

4. 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。このリンクには、モジュール内のサーブレットの 1 つ、AllItemsServlet を URL パターンで指定します。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。要求は、AllItemsServlet を実行することで処理されます。ユーザーに次に表示されるページは、AllItemsServlet の出力ページです。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、どの機能もこの例に示す原則に従います。ユーザーに表示されるページには、HTTP 要求を生成するリンクや動作が含まれます。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、Web モジュール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。

この例では、リンクはサーブレットを指定していますが、サーブレットのメソッドは指定していません。要求がサーブレット名だけを指定した場合は、デフォルト動作であるサーブレットの doGet メソッドが実行されます。doPost など、サーブレットのほかのメソッドを指定するリンクを作成することもできます。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。

モジュールの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、モジュールの「開始ファイル」プロパティには開始ページファイル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、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これらの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。

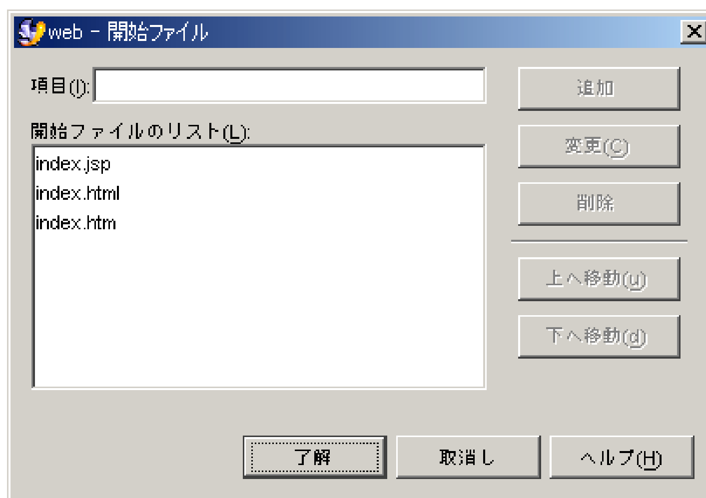


図 2-2　「開始ファイル」プロパティエディタ

別の名前のファイルをモジュールの開始ページとして使用するには、次のようにします。

1. **「ファイルシステム」ウィンドウで web ノードを右クリックし、「プロパティ」を選択します。**

   プロパティシートが開きます。

2. **「配備」セクションの「開始ファイル」の省略符号ボタン (...) をクリックします。**

   「開始ファイル」プロパティエディタが開きます。

3. **開始ページの名前を「項目」フィールドに入力して「追加」をクリックし、「了解」をクリックします。**

サーブレットメソッドのプログラミング

このシナリオのサーブレットは、エンタープライズ Bean メソッドを呼び出すことで EJB モジュールからデータを取得します。ここでは、これらのメソッド呼び出しのために 2 つのプログラミング作業を行います。1 つは呼び出し側サーブレット内の呼び出し元のメソッドのコーディング、もう 1 つは呼び出される側のエンタープライズ Bean リソース参照の設定です。最初に、メソッド本体のコーディングについて説明します。

注 – この例のサーブレットは、IDE サーブレットテンプレートを使って作成されています。このテンプレートを使って作成されたサーブレットは HttpServlets で、`processRequest` というメソッドが含まれています。`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。

開始ページの URL パターンはサーブレット `AllItemsServlet` を指定しますが、メソッドは指定しません。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て、サーブレットの `doGet` メソッドを実行します。`doGet` メソッドは `processRequest` メソッドを呼び出します。

コード例 2-2 は、`AllItemsServlet` が EJB モジュールから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. `AllItemsServlet` が JNDI ルックアップを使用し、`CatalogData` EJB 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します。`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。
2. `AllItemsServlet` が、セッション Bean のビジネスメソッドである `getAllItems` メソッドを呼び出します。

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。この出力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 サーブレット AllItemsServlet の processRequest メソッド

```
protected void processRequest(HttpServletRequest request,
                              HttpServletResponse response)
                              throws ServletException, IOException {
  response.setContentType("text/html");
  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>");

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


  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>" +

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

コード例 2-2　　サーブレット AllItemsServlet の processRequest メソッド (続き)

```
        "<td>" +
        "<a href=\"" + response.encodeURL("itemDetail?sku=" +
        itemDetail.getItemsku()) +
        "\"> " +
        "Get Item Detail" +
        "</a>" + "</td>" + "</tr>");
    }
  }
  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());
  }

  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。参照先のエンタープライズ Bean は実際にはリソース参照で指定されます。これについては次に説明します。

EJB リソース参照

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 ルックアップのコード。**これは、JNDI 命名機能を使用して、指定されたエンタープライズ Bean へのリモート参照を取得します。

- **リソース参照。**これは、配備記述子の一部で、特定の Bean のルックアップコードが参照するアプリケーションサーバーを指定します。

このシナリオでは、ルックアップコードは `AllItemsServlet` の `processRequest` メソッドにあります (コード例 2-2 を参照してください)。このコードはコンパイルできますが、リソース参照なしでは実行時に参照を返すことができません。リソース参照は、lookup 文で使用されている参照名をエンタープライズ Bean の実際の名前にマップします。

Web モジュールの EJB リソース参照を設定するには、次のようにします。

1. **Web モジュールの web ノードを右クリックし、「プロパティ」を選択します。「参照」セクションの「EJB 参照」の省略符号ボタン (...) をクリックします。**

   「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。**

   「追加 EJB 参照」ダイアログが開きます。このダイアログを使用して、リソース参照を設定します。

3. **参照を設定するためには、`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フェースとリモートインタフェースの名前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 EJB 参照」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。参照は、アプリケーションの配備と実行の前にリンク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。以下の点を調べてください。

   - 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。使用中の開発環境にインタフェースのコピーがマウントされていない場合は、JNDI ルックアップコードをコンパイルできません。

     図 2-3 に、リンクしていない参照を設定している「追加 EJB 参照」ダイアログを示します。「ホームインタフェース」と「リモートインタフェース」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します。

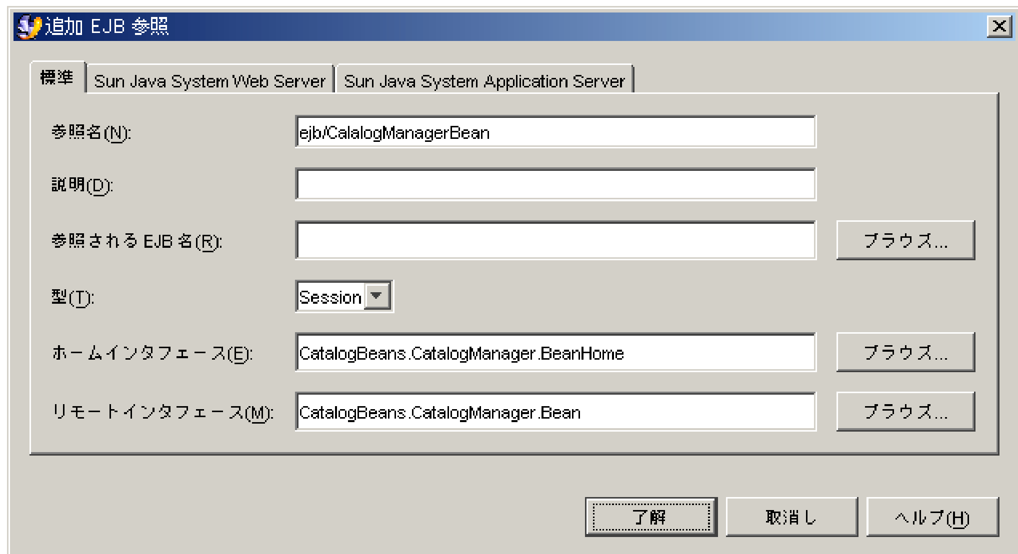


図 2-3 参照がリンクされていない「EJB 参照」プロパティエディタ

- 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、呼び出すエンタープライズ Bean を選択します。「了解」をクリックします。

  図 2-4 では、`ejb/CatalogManagerBean` という名前の参照が、`CatalogManagerBean` にリンクされています。

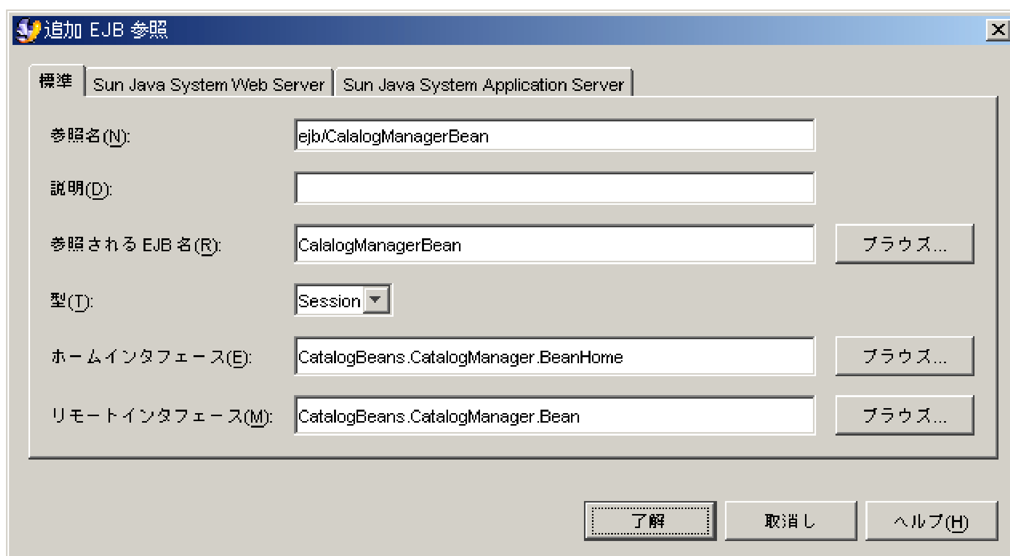


図 2-4　参照がリンクされている「EJB 参照」プロパティエディタ

呼び出すエンタープライズ 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いくつかあります。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、Web 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。その Web モジュールを再使用する開発者が、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 に参照をリンクし直すことも考えられます。参照をリンクし直すと `web.xml` 配備記述子ファイルの値が変更されて、そのソースコードの使用のすべてに影響が及びます。

このシナリオでは、図 2-3 に示されているとおり参照はリンクされないままにしています。参照は、J2EE アプリケーションの作成後、アプリケーションノードのプロパティシート上でリンクします。67 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、以下のように HTML タグで書式化されます。

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

利用者がこのリンクをクリックすると、アプリケーションサーバーは HTML タグの URL パターンを URL パスに追加して、この結果の URL を実行します。このリンクが適切に機能するためには、アプリケーションサーバーが生成した 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。次のシナリオでは、URL がサーブレットにマップされる方法を説明します。

注 – サーブレットの作成についての詳細は、『Web アプリケーションフレームワーク 開発ガイド』を参照してください。

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 Web モジュールでは、サーブレットなどの Web 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。Sun Java System Application Server Standard Edition 7 に配備されたモジュールの場合、URL パスは一般に次の形式になります。

`http://`*hostname*`:`*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。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。
- *web-context* (Web コンテキスト) は、J2EE アプリケーションに Web モジュールを追加した後で Web モジュールのプロパティとして指定する文字列です。Web コンテキストは、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 JavaServer Pages (JSP) ページを識別する文字列です。URL パターンは、Web モジュールのプロパティシートで設定します。これは、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する前に行うことができます。

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 URL は、Web コンテキストからの相対的な位置になる書式を使用しています。アプリケーションをアセンブルするときに Web コンテキストを指定すると、アプリケーションの実行時にこれらのリンクは正しく動作します。Web コンテキストの設定の詳細については、65 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。

デフォルトのサーブレットマッピングの調査

サーブレットの作成時、新規ウィザードのデフォルトでは、新規ウィザードの先頭のページで指定したオブジェクト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` を示す Web モジュールの「サーブレットマッピング」プロパティエディタです。

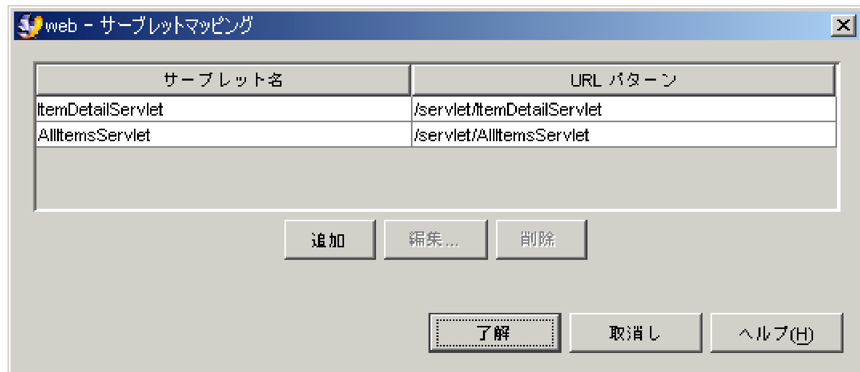


図 2-5　「サーブレットマッピング」プロパティエディタ

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`

サーブレットマッピングの編集

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 パターンのデフォルト値を、より意味のある値に変更します。

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. **Web モジュールの web ノードを右クリックし、「プロパティ」を選択します。「配備」セクションの「サーブレットマッピング」の省略符号ボタン (...) をクリックします。**

   「サーブレットマッピング」プロパティエディタが開きます。「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マッピングすべてが一覧表示されます。

2. **`allItemsServlet` の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。**

   「編集 サーブレットマッピング」ダイアログが開きます。

3. **「URL パターン」フィールドに「`/allItems`」と入力して「了解」をクリックします。**

   図 2-6 は、`AllItemsServlet` と `ItemDetailServlet` に新しいマッピングが指定された「サーブレットマッピング」プロパティエディタを示しています。

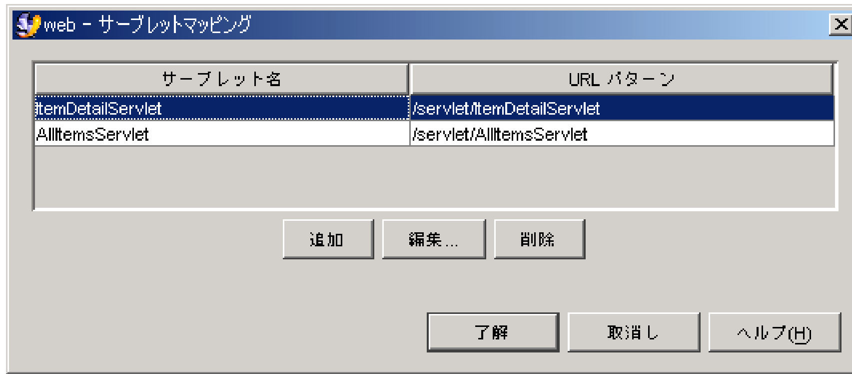


図 2-6　　「サーブレットマッピング」プロパティエディタ

サーブレットマッピングの編集が終わると、次の URL で `AllItemsServlet` を実行できます。

`http://`*hostname*`:`*port*`/`*web-context*`/allItems`

新しい URL パターンは、開始ページでリンクを作成する HTML タグで使用した文字列であることに注意してください (コード例 2-1 を参照)。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

これまでの項では、`CatalogWebModule` のアセンブルに必要な作業を取り上げました。この項では、`CatalogWebModule` のシナリオで必要とされない Web モジュールのアセンブル作業について説明します。

Web モジュールによっては、こうしたほかのアセンブル作業を実行する必要があります。ここでは、実行する可能性のある Web モジュールのアセンブル作業をいくつか取り上げます。

エラーページの設定

Web モジュールのエラーページを指定する場合は、「エラーページ」プロパティエディタで、モジュールの配備記述子にエラーページを指定する必要があります。

Web モジュールのエラーページを設定するには、次のようにします。

1. web ノードを右クリックし、「プロパティ」を選択します。「配備」セクションの「エラーページ」の省略符号ボタン (...) をクリックします。

   「エラーページ」プロパティエディタが開きます。

2. HTTP エラーコードまたは Java 例外クラスによってエラーを指定し、特定のエラーページにこれをマップします。「追加」をクリックします。

   エラーは、HTTP エラーコードと Java 例外クラスのいずれかによって識別できます。このプロパティエディタには、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、特定のエラーページに HTTP エラーコード 404 をマップした後のプロパティエディタを示しています。

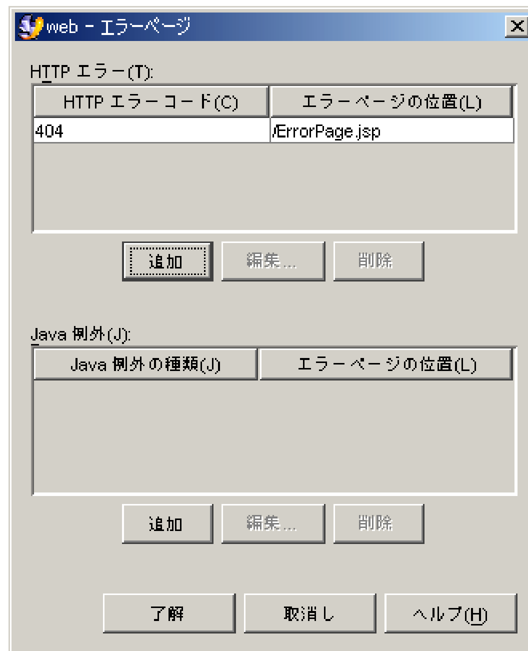


図 2-7　　「エラーページ」プロパティエディタ

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。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場合は、配備記述子エントリは必要ありません。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) を指定することで、実行される JSP ページを識別していることに注意してください。

コード例 2-3　プログラムを使用して JSP ページを実行するためのコード

```
...
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、JSP ページの実際のファイル名 myJsp.jsp を指定することで、JSP ページを実行しています。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。これは 2 つの手順を踏んで行います。まず JSP ファイルのサーブレット名を設定します。

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. **web ノードを右クリックし、「プロパティ」を選択します。「配備」セクションの「JSP ファイル」の省略符号ボタン (...) をクリックします。**

   「JSP ファイル」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。**

   「追加 JSP ファイル」ダイアログが開きます。

   a. **「JSP ファイル」フィールドに、設定する JSP ファイルの名前を入力します。**

   b. **「サーブレット名」フィールドに、JSP ファイルにマッピングするサーブレット名を入力します。**

c. 「了解」をクリックします。

「追加 JSP ファイル」ダイアログが閉じます。

図 2-8 は、サーブレット名 `itemDetailPage` が `myJsp.jsp` ファイル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。

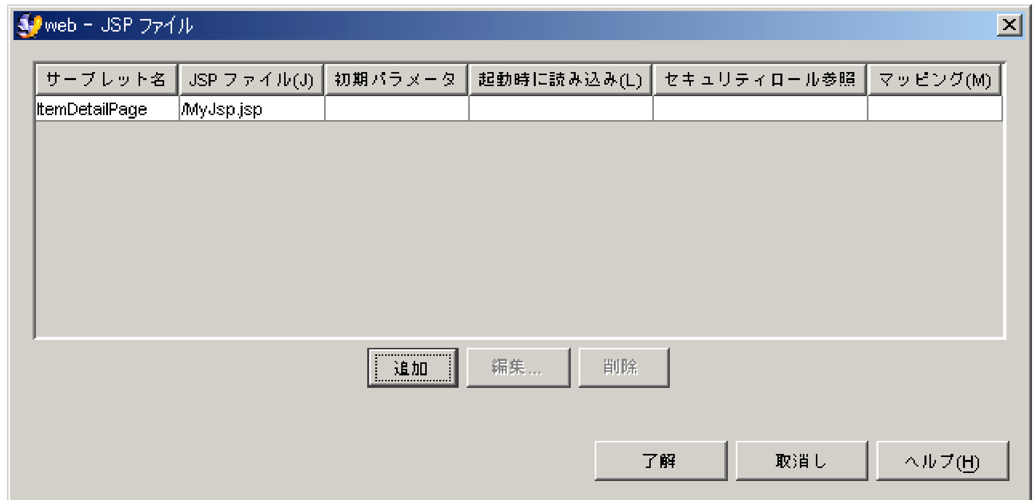


図 2-8　「JSP ファイル」プロパティエディタ

3. 「了解」をクリックして「JSP ファイル」プロパティエディタを閉じ、プロパティシートに戻ります。

4. プロパティシートで、「サーブレットマッピング」の省略符号ボタン (...) をクリックします。

   「サーブレットマッピング」プロパティエディタが開きます。

5. 「追加」ボタンをクリックします。

   「追加 サーブレットマッピング」ダイアログが開きます。

6. 「追加 サーブレットマッピング」ダイアログで、URL パターンを新しいサーブレット名にマップします。

   a. 「サーブレット名」フィールドに、JSP ファイルにマッピングしたサーブレット名を入力します。

   b. 「URL パターン」フィールドに、サーブレット名 (最終的には JSP ファイル) にマッピングする URL パターンを入力します。

c. 「了解」をクリックします。

「追加 サーブレットマッピング」ダイアログが閉じます。

図 2-9 は、URL パターン `ItemDetail` がサーブレット名 `ItemDetailPage` にマップされた「サーブレットマッピング」プロパティエディタを示しています。サーブレット名 `ItemDetailPage` はすでに JSP ファイル `myJsp.jsp` にマップされています。

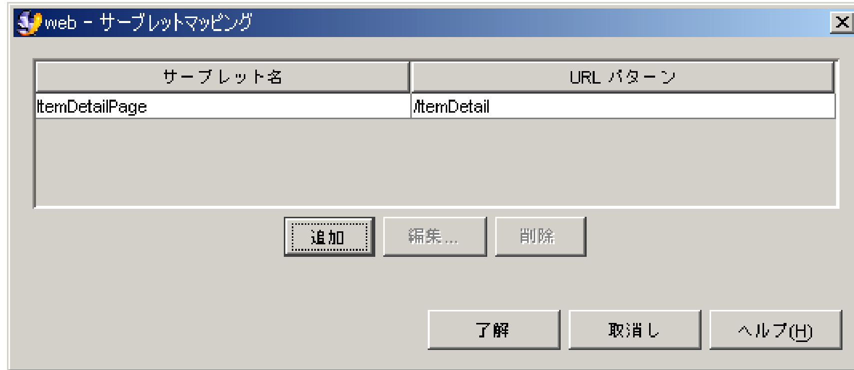


図 2-9 「サーブレットマッピング」プロパティエディタ

d. 「了解」をクリックして「サーブレットマッピング」プロパティエディタを閉じます。

このマッピングの結果、ファイル `myJSP.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://`*hostname*`:`*port*`/`*web-context*`/ItemDetail`

リソース環境エントリ参照の設定

環境エントリは、Web サービスの EJB モジュール配備記述子に格納するデータです。環境エントリは、ターゲットオブジェクトの作成または検索を行うメソッドのパラメータとして、Web サービスで利用できます。環境エントリは配備記述子に格納されるので、配備時に、実行時環境に適した値に設定できます。環境エントリについての詳細は、『Web サービスのプログラミング』を参照してください。

環境エントリには次の 2 つの部分があります。

- **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。
- **Web モジュールの配備記述子内のリソース環境参照。**リソース環境参照は、アプリケーションサーバーへの参照を宣言します。配備記述子には、環境エントリの値も含まれます。

リソース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、コード例 2-4 に示すようなルックアップコードが必要です。

コード例 2-4 環境エントリの lookup 文

```
try {
        // 最初のコンテキストの取得 -- JNDI 命名機能の使用開始:
        Context ic = new InitialContext();
        // 「Cache Size」という名前の環境エントリのルックアップ要求:
        Integer cacheSize = (Integer)
         ic.lookup("java:comp/env/NumberOfRecordsCached");
}
catch(Exception e) {
        System.out.println(e.toString());
        e.printStackTrace();
        return;
}
```

コード中のコメントは、各行の動作を説明しています。

環境エントリのリソース参照

環境エントリのリソース参照を設定するには、次のようにします。

1. **web ノードを右クリックし、「プロパティ」を選択します。「参照」セクションの「環境エントリ」の省略符号ボタン (...) をクリックします。**

   「環境エントリ」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。**

   「追加 環境エントリ」ダイアログが開きます。

3. **環境エントリの参照を宣言します。**

   a. **「名前」フィールドに、`lookup` 文で使用されている参照名を入力します。**

   b. **「型」フィールドで、環境エントリのデータの型を選択します。**

   c. **「値」フィールドに、環境エントリの初期値を入力します。**

   図 2-10 に、コード例 2-4 の `lookup` 文と一致し、値が指定された「追加 環境エントリ」ダイアログを示します。

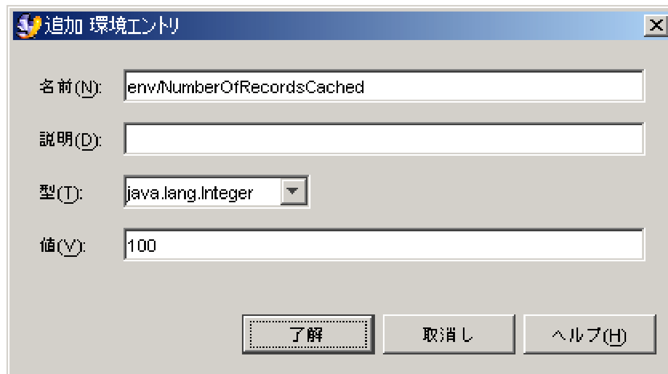


図 2-10　「追加 環境エントリ」ダイアログ

4. 「了解」をクリックしてダイアログを閉じ、環境エントリ定義を処理します。
5. 「了解」をクリックして、「環境エントリ」プロパティエディタを閉じます。

第3章

シナリオ: EJB モジュール

図 3-1 は、Enterprise JavaBeans (EJB) モジュールとそれが関与する対話を示しています。このモジュールは Java 2 Platform, Enterprise Edition (J2EE プラットフォーム) アプリケーションの一部で、図に示された対話は J2EE アプリケーションの EJB モジュールの典型的な対話です。EJB モジュールはクライアントモジュールとの対話を行います (この対話は図中の矢印 1 で示されています)。また、EJB モジュールは、外部リソース (この場合はリレーショナルデータベース管理システム) とも対話します (この対話は矢印 3 で示されています)。ほとんどの EJB モジュールと同様に、この EJB モジュールにも複数のエンタープライズ Bean が含まれており、エンタープライズ Bean が互いに対話します (この対話は矢印 2 で示されています)。

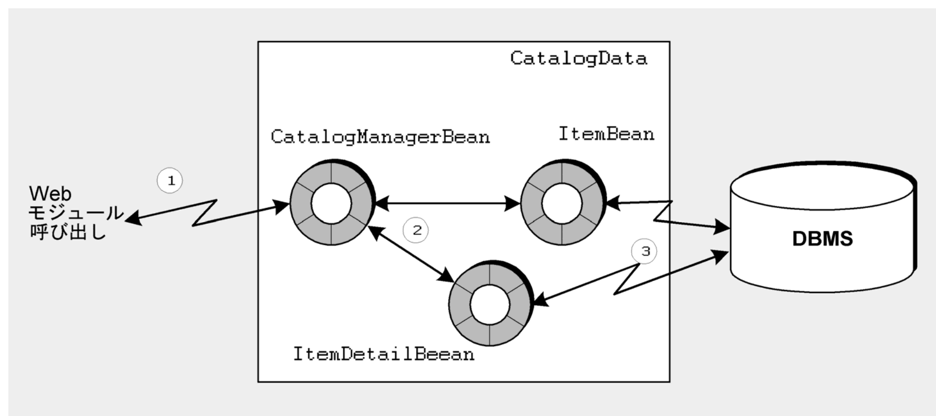


図 3-1 `CatalogData` EJB モジュール

モジュール内での対話

このシナリオでは、図 3-1 に示された EJB モジュールの使用法と対話について説明します。このシナリオでは、EJB 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バックエンドです。EJB モジュールには、カタログデータベースと対話する一連のエンタープライズ Bean が含まれています。アプリケーションのフロントエンドとして機能する Web モジュールは、この EJB モジュールのメソッドを呼び出して、ユーザーに表示するデータを取得します。メソッド呼び出しの処理とデータベースとの対話は、J2EE アプリケーションの EJB モジュールが行う典型的なタスクです。

アプリケーションの Web モジュールが、`CatalogData` EJB モジュールのクライアントとして機能します。利用者は Web モジュールによって生成された Web ページを使用して、表示したいデータを要求します。Web モジュールの仕事は、`CatalogData` モジュールを呼び出してデータを取得し、このデータを書式化して利用者に表示することです。

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。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。このシナリオの EJB モジュールには、3 つのエンタープライズ Bean が含まれており、2 つのビジネスメソッドを提供する単純なものです。ビジネスメソッドは 2 種類の要求のみを処理します。これらの単純なコンポーネントや対話によって、クライアントモジュールとデータベースの間に必要な対話を EJB モジュールがどのように提供するかが分かります。このシナリオで扱っている特定の対話の概要を、以下に示します。

1. クライアントモジュールが、`CatalogData` EJB モジュールのメソッドを呼び出して、データを要求します。このシナリオでは、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。クライアントモジュール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 `CatalogData` EJB モジュール間のこの対話によって、これらを実装するために選択する J2EE テクノロジと `CatalogData` EJB モジュールの内部構造が決定されます。次の点を考慮します。

- クライアントモジュールと `CatalogData` モジュール間の対話は同期対話です。つまり、カタログの表示を要求したオンラインショッピングの利用者は、アプリケーションがそれを表示するのを待ちます。

- EJB モジュールの内部設計は、クライアントモジュールと CatalogData EJB モジュール間のセッション指向の対話によって決定します。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。複数の利用者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と照合し、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。このニーズを満たすために、モジュールは単一のステートフルセッションエンタープライズ Bean を使用します。このセッション Bean は、クライアントモジュールが送信するすべての要求を管理します。セッション Bean は、データベース照会を生成するエンティティ Bean のメソッドを呼び出して、要求を処理します。

これは EJB モジュールによく見られるアーキテクチャです (セッション Bean を使ったセッションのモデリングの詳細については、『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)。

データベース照会を生成する場合、J2EE プラットフォーム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 はデータベース表を原型とし、照会を実行するメソッドを持っています。このシナリオでは、カタログデータは 2 つの表に保管されているため、カタログデータモジュールには 2 つのエンティティ Bean があります。それぞれのエンティティ Bean は、表の 1 つを基にしています。エンティティ Bean は、データベースと CatalogData EJB モジュールの対話の中で、接続や照会言語などの処理を担当します。

モジュールのプログラミング

表 3-1 に、前述の節と 図 3-1 に示された `CatalogData` 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エンタープライズ Bean (`CatalogManagerBean`) を作成します。リモートインタフェースは、ほかのモジュールによってリモートに呼び出されるメソッドに適しています。カタログデータを呼び出し側に返す 2 つのビジネスメソッドを、セッション Bean に追加します。1 つのビジネスメソッドは、カタログのすべての項目を返します。もう 1 つは、クライアントで指定された項目の詳細を返します。 カタログデータを含んだ 2 つのデータベース表を表現する 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。これらのエンティティエンタープライズ Bean に対しローカルインタフェースを作成します。ローカルインタフェースは、同じモジュールのほかのエンタープライズ Bean によって呼び出されるメソッドに適しています。カタログの全商品を返すメソッドを `ItemBean` に追加します。`ItemDetailBean` は、特定商品の詳細を返す `findByPrimayKey` メソッドを自動的に生成しています。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 2 つの転送クラス (Item データと ItemDetail データにそれぞれ 1 つずつ) を作成します。`CatalogData` EJB 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 `CatalogData` EJB モジュールを作成します。これは、「ファイルシステム」ウィンドウに EJB モジュールノードとして表示されます。モジュールに 3 つのエンタープライズ Bean を追加します。`CatalogData` EJB モジュールのプロパティシートを使用して、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。インタフェースのメソッドシグニチャは、各対話の入力と出力を示します。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。それらの作業手順については、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

CatalogData EJB モジュールの設計は、クライアントモジュールがステートフルセッション Bean のメソッドに対するリモート呼び出しを実行することでカタログデータを取得するというものです。このリモート呼び出しを可能にするには、セッション Bean がリモートインタフェースを持っている必要があります。リモートインタフェースは、セッション EJB の新規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。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;
}
```

これらのインタフェースは、CatalogData EJB モジュールとクライアントモジュール間の 2 つの対話を定義します。クライアントは、getAllItems メソッドを呼び出すことで、カタログ内の全商品のリストを取得できます。また、クライアントは、getOneItemDetail メソッドを呼び出すことで、指定した商品の詳細を取得できます。

現実のショッピングアプリケーションのほとんどは、この例より多くの機能を提供します。こうした現実のアプリケーションのインタフェースには、3 つ以上のビジネスメソッドが含まれています。

EJB モジュールがエンティティ Bean インスタンスへの参照を返すと、アプリケーションサーバーは、クライアントモジュールがエンティティ Bean インスタンスに対して行った変更をすべて追跡して、データベース更新を自動的に生成します。クライアントにリモート参照を渡すと、ネットワークリソースが消費されるため、これはクライアントがデータを更新できる場合のみ実行されるようにします。このシナリオでは、クライアントはデータの表示のみを行うため、この機能を使用する必要はありま

せん。CatalogData 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。これらのクラスは転送クラスと呼ばれ、エンティティ Bean と同じフィールドを持っています。CatalogData モジュールは、エンティティ Bean インスタンスから転送クラスインスタンスにデータをコピーし、転送クラスインスタンスをクライアントに返して表示します。エンティティエンタープライズ Bean での転送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。

クライアントモジュールでリモートインタフェースを使用して CatalogData モジュールのメソッドを呼び出す方法の詳細は、26 ページの「サーブレットメソッドのプログラミング」を参照してください。

getAllItems メソッドと getOneItemDetail メソッドは、エンティティ Bean のメソッドへの呼び出しを使って実装されています。エンティティ Bean は適切な照会を生成し、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。「getAllItems」メソッドの実装については、コード例 3-3 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に、CatalogManagerBean は 2 つのエンティティ Bean のメソッドを呼び出します。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、エンティティ Bean のローカルインタフェースを生成できます。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す。リモート対話が必要ない場合は、ローカルインタフェースを使用します。

ローカルインタフェースは、エンティティ EJB の新規ウィザードでエンティティ Bean を作成するときに生成します。コード例 3-2 に、Item 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。ItemDetail Bean のインタフェースもほぼ同じになります。

コード例 3-2 ItemBean エンタープライズ 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 LocalItemBeanHome extends javax.ejb.EJBLocalHome {
  public CatalogBeans.LocalItemBean findByPrimaryKey(java.lang.Integer aKey)
    throws javax.ejb.FinderException;
  public java.util.Collection findAll() throws javax.ejb.FinderException;
}


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

CatalogManagerBean は、findAll メソッドを呼び出してカタログ内の全商品のリストを取得します。

注 – 統合開発環境 (IDE) 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る場合は、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、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 ビジネスメソッドの 1 つを呼び出して、カタログデータを要求します。これらは CatalogManagerBean インタフェースで宣言されたメソッドで、コード例 3-1 に示されています。これらのビジネスメソッドの実装は、エンティティ Bean メソッドを呼び出します。このようにエンティティ Bean を呼び出すためには、CatalogManagerBean 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。これらのローカル EJB 参照について、以下の 2 つの部分をプログラミングする必要があります。

- **JNDI (Java Naming and Directory Interface) ルックアップコード。**セッション Bean のビジネスメソッドの両方に、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。

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、getAllItems メソッドセッション Bean の実装です。CatalogData EJB モジュールのクライアントはこのメソッドを呼び出して、オンラインカタログ内の全商品のリストを取得します。このメソッドの実装は、CatalogManagerBean がエンティティ Bean と対話することでカタログデータを取得する方法を示します。このコードは、次の 3 段階に分けることができます。

1. CatalogManagerBean が JNDI ルックアップを使用して、ItemBean のローカルホームインタフェースへのローカル参照を取得します。

2. CatalogManagerBean が itemBean.findAll メソッドを呼び出します。

3. CatalogManagerBean が、エンティティ Bean インスタンスから転送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、転送クラスインスタンスをベクタに追加し、ベクタをクライアントに返します。

コード例 3-3 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 のローカルホームへの参照を取得する
        InitialContext iC = new InitialContext();
        Object objref = iC.lookup("java:comp/env/ejb/ItemBean");
        itemHome = (LocalItemBeanHome) 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 {
      // 転送クラスインスタンスにデータをコピーする
      java.util.Iterator iter = itemsColl.iterator();
      while (iter.hasNext()) {
        iDetail detail;
        detail=((CatalogBeans.LocalItemBean)iter.next()).getIDetail();
        itemsVector.addElement(detail);
      }
    }
  }
  catch(Exception e) {
    System.out.println(e);
  }
  return itemsVector;
}
```

lookup 文の文字列は、参照先のエンタープライズ Bean の名前ではなく、参照名を指定しています。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、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 に対する参照名の実際のマッピングは、リソース参照で行います。これについては次の項で説明します。

ローカル EJB リソース参照

JNDI ルックアップコードを記述したら、ローカル EJB リソース参照を設定します。ローカル EJB リソース参照は、`lookup` 文で使用されている参照名を実際のエンタープライズ Bean 名にマップします。参照先のエンタープライズ Bean は、呼び出し側の Bean と同じモジュール内にある必要があります。

ローカル EJB リソース参照を設定するには、次のようにします。

1. **呼び出し元 Bean の論理ノードを右クリックし、「プロパティ」を選択します。「参照」セクションの「EJB ローカル参照」の省略符号ボタン (...) をクリックします。**

   「EJB ローカル参照」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。**

   「追加 EJB ローカル参照」ダイアログが開きます。

3. **ダイアログで参照を定義します。**

   a. **「参照名」フィールドに、`lookup` 文で使用されている参照名を入力します。**

      図 3-2 には、コード例 3-3 で使用されていた参照名が示されています。

   b. **参照名をエンタープライズ Bean にマップします。**

      「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。選択ダイアログが開いたら、対応するエンタープライズ Bean を選択します。図 3-2 は、`ItemBean` が選択されたフィールドを示しています。

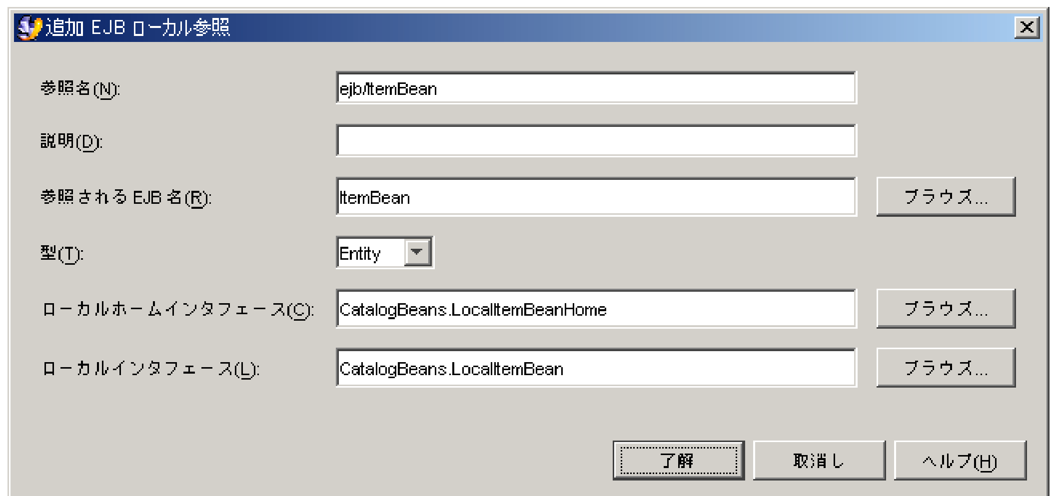


図 3-2　「追加 EJB ローカル参照」ダイアログ

EJB モジュールのアセンブル

エンタープライズ Bean 群の作成が終わったら、`CatalogData` EJB モジュールを作成して、エンタープライズ Bean を追加します。アプリケーションサーバーに特定の実行時サービスを要求するプロパティ値を選択することで、EJB モジュールを構成します。

アプリケーションサーバー製品の種類によって配備に必要な情報も異なり、EJB モジュールのプロパティにはサーバー固有のプロパティが含まれます。このシナリオでは、Sun Java System Application Server 固有のプロパティを使用します。

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、モジュールの配備記述子を表します。モジュールに追加するエンタープライズ Bean は、モジュールノードのサブノードとして表示されますが、IDE はエンタープライズ Bean のソースファイルをモジュールノードが格納されているディレクトリにコピーしません。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 EJB モジュールノードを配置します。
- 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、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を作成する方法もあります。これらのディレクトリは、ソースコードを含むディレクトリとは別に保持してください。

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 モジュールを作成」を選択します。

   「新規 EJB モジュール」ダイアログが開きます。

2. モジュールに名前を付けて、ファイルシステム内でのその配置場所を選択します。

   a. 「名前」フィールドに、新しいモジュールの名前を入力します。このシナリオの場合は、EJB モジュールの名前は `CatalogData` になります。

   b. 「新規 EJB モジュールのパッケージの位置を選択」フィールドで、新しいモジュールを作成したいファイルシステム、ディレクトリ、またはパッケージを選択します。

c. 「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、ディレクトリ、またはノードの下に作成されます。手順 1 で右クリックしたエンタープライズ Bean は、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。52 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. 「ファイルシステム」ウィンドウで任意のノード (ファイルシステム、パッケージ、またはディレクトリ) を右クリックし、コンテキストメニューの「新規」→「すべてのテンプレート」を選択します。

   「新規ウィザード」の「テンプレートを選択」ページが開きます。

2. EJB モジュールのテンプレートを選択します。

   a. 「テンプレートを選択」フィールドで、「J2EE」ノードを展開して「EJB モジュール」ノードを選択します。

   b. 「次へ」をクリックします。

   「新規オブジェクト名」ページが開きます。

3. 新しい EJB モジュールを定義します。

   a. 「名前」フィールドに、モジュールの名前を入力します。

   このシナリオの場合は、モジュールの名前は `CatalogData` になります。

   b. 「完了」をクリックします。

   「EJB を EJB モジュールに追加」ページが開きます。

   c. 「この EJB モジュールに追加する EJB を選択します」フィールドで、モジュールに追加するエンタープライズ Bean を選択し、「了解」をクリックします。「取消し」をクリックした場合は、「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」で説明する手順に従って、エンタープライズ Bean を任意の時点で追加できます。

   IDE によってモジュールが作成され、手順 1 で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の EJB モジュールとして「ファイルシステム」ウィンドウに表示されます。

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。IDE はモジュールの配備記述子情報をこのディレクトリに格納しますが、モジュール内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ないことに注意してください。

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ると、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、「EJB を追加」を選択します。**

   「EJB を EJB モジュールに追加」ダイアログが開きます。

2. **エンタープライズ Bean を 1 つ以上選択します。**

   a. **「この EJB モジュールに追加する EJB を選択します」フィールドで、モジュールに追加するエンタープライズ Bean を選択します。**

   b. **「了解」をクリックします。**

   選択したエンタープライズ Bean が、IDE によってモジュールに追加されます。「ファイルシステム」ウィンドウでは、選択した Bean を表すノードが IDE によってモジュールノードの下に追加されます。

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、イメージファイル、その他) との依存関係が管理されます。これらのリソースは IDE によってモジュール定義に自動的に追加されます。

これにはいくつかの例外があります。詳細については、58 ページの「追加のファイルの指定」を参照してください。

エンティティエンタープライズ Bean のデータソースの指定

このシナリオでは、`CatalogData` EJB 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ます。J2EE アプリケーションは、J2EE アプリケーションサーバーが提供する接続プールなどのサービスを使い、J2EE アプリケーションサーバーを通してデータベースにアクセスします。データベースは、JNDI 名を持つアプリケーションサーバーリソースとして設定する必要があります。アプリケーションサーバーリソースとしてデータベースを定義することで、データベースの JNDI 名を実際のデータベース URL にマップします。

データベースにアクセスする J2EE アプリケーションは、データベースの JNDI 名を含むリソース参照を使って構成する必要があります。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、アプリケーションサーバーによってデータベースの識別方法が決まります。Sun Java System Application Server を含むほとんどのアプリケーションサーバーでは、モジュールのサーバー固有プロパティの 1 つを使用して、JNDI 名によってデータソースを指定します。

- 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を記述し、ルックアップ名を JNDI 名にマップする、対応するリソース参照を設定する必要があります。

注 – 開発時には、IDE でデータベースへの接続を開き、エンティティ EJB の新規ウィザードでデータベース表を原型とするエンティティ Bean を作成できます。この種の接続は、『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。ここでは、実行時に使用されるデータベースの指定方法について説明しています。

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、EJB モジュールの「CMP リソース」プロパティエディタで JNDI 名によってデータソースを指定します。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。「CMP リソース」プロパティエディタには、デフォルトの Pointbase データベースの JNDI 名を入力します。

データベースを使用するようにコンテナ管理によるエンティティ Bean を構成するには、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、「プロパティ」を選択します。「Sun Java System AS」セクションの「CMP リソース」の省略符号ボタン (...) をクリックします。**

   「CMP リソース」プロパティエディタが開きます。

2. **EJB モジュールとそのエンティティ Bean のデータベースを識別する値を入力します。**

   図 3-3 は、`CatalogData` EJB モジュールに `sample` という名前の PointBase データベースへのアクセスを与える値が入力された「CMP リソース」プロパティエディタを示しています。

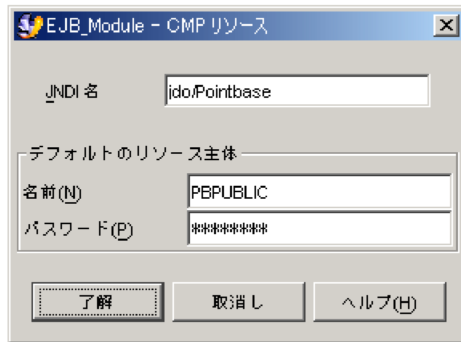


図 3-3 EJB モジュールの「CMP リソース」プロパティエディタ

注 – IDE のほとんどのインストールには、Sun Java System Application Server と PointBase データベースサーバーが含まれています。IDE をインストールすると、`sample` という名前の PointBase データベースが、`jdo/PointbasePM` という JNDI 名でアプリケーションサーバーリソースとして設定されます。このシナリオでは、`CatalogData` EJB モジュールが `sample` データベースを使用するように設定されます。

この手順は、Sun Java System Application Server およびインストールされた PointBase データベースとともに IDE を使用する場合のものです。別のデータベース製品や別のアプリケーションサーバー製品を使用する場合は、それに合わせて手順を変更してください。

- Sun Java System Application Server を使い、PointBase 以外のデータベース製品とともに IDE を使用する場合は、Application Server の管理ツールを使って JNDI 名付きの JDBC (Java DataBase Connectivity) データソースとして Application Server にデータベースを定義する必要があります。その後で、データソースの JNDI 名を使って EJB モジュールを設定する作業を実行できます。
- Sun Java System Application Server 以外のアプリケーションサーバーを使用する場合は、データベースをアプリケーションサーバーのデータソースリソースとして設定し、JNDI 名を付ける必要があります。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。

  JNDI 名を付けてデータベースを設定した後で、この JNDI 名を使って EJB モジュールを構成できます。これには、EJB モジュールのプロパティシートで、使用しているアプリケーションサーバー製品に固有のタブを使用します。プロパティ名は、「CMP リソース」と異なることがあります。

管理されたテスト環境または本番環境にアプリケーションを配備しようとしている場合、アプリケーションサーバーリソースとしてデータベースを設定し JNDI 名を割り当てる作業は、おそらくシステム管理担当者の仕事です。その場合は、システム管理担当者からデータソースの JNDI 名を入手します。

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、データベース照会を準備して実行する JDBC と JTA (Java Transcription API) コードがエンティティ Bean にすでに含まれています。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。このコードには、データソースへのリソースファクトリ参照を取得する JNDI ルックアップが含まれている必要があります。ルックアップはデータソースの JNDI 名を使用して参照を取得します。ほかの種類の J2EE 参照と同様に、この参照にも次の 2 つの部分があります。

- **JNDI ルックアップ。**各エンティティ Bean には、JNDI `lookup` 文を使って名前付きリソースへの参照を取得するためのコードが含まれています。
- **参照の宣言。**この宣言は、lookup 文で使用されている参照名をデータソースの JNDI 名にマップします。

明示的なルックアップで指定するデータソースは、割り当てられた JNDI 名を持つアプリケーションサーバーの名前付きのリソースでなければなりません。アプリケーションサーバーは、JNDI 名を実際のデータベース URL にマップする情報を持っています。

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 (Bean-Managed Persistence) 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 {
  // 最初のコンテキストの取得 -- JNDI 命名機能の使用開始:
  Context ic = new InitialContext();
  // リソースのルックアップ要求 -- この例では JDBC データソース:
  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;
}
```

リソース参照の宣言

lookup 文の記述に加え、データソースリソースのリソース参照を設定する必要があります。

データソース参照のリソース参照を設定するには、次のようにします。

1. エンタープライズ Bean の論理ノードを右クリックし、「プロパティ」を選択します。「参照」セクションの「リソース参照」の省略符号ボタン (...) をクリックします。

   「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ます。

   「追加 リソース参照」ダイアログが開きます。

3. データソースを識別するための情報を指定します。

   a. 「名前」フィールドに、`lookup` 文で使用されている参照名を入力します。

      図 3-4 には、コード例 3-4 で使用されていた参照名が示されています。

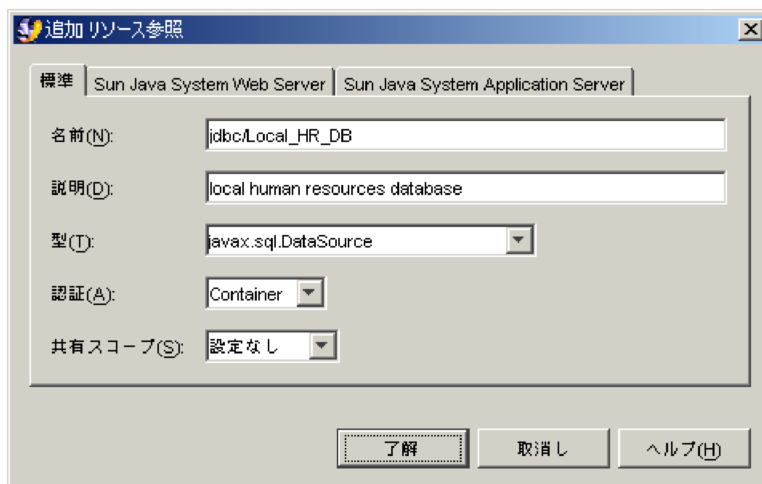


図 3-4　　「追加 リソース参照」ダイアログ

   b. 「Sun Java System Application Server」タブをクリックします。

   c. 参照名をデータソースにマップします。

      図 3-5 は、ほとんどの IDE インストールに含まれている PointBase データベースのデフォルト名 JNDI 名である値 `jdbc/jdbc-pointbase` が設定された「JNDI Name」フィールドです。

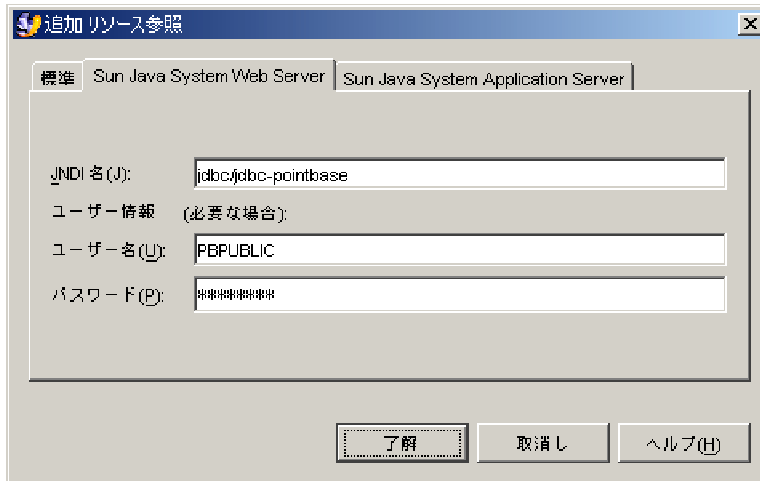


図 3-5　　「追加 リソース参照」ダイアログのサーバー固有タブ

この手順は、Sun Java System Application Server およびインストールされた PointBase データベースとともに IDE を使用する場合のものです。別のデータベース製品や別のアプリケーションサーバー製品を使用する場合は、それに合わせて次のように手順を変更してください。

- Sun Java System Application Server を使い、PointBase 以外のデータベース製品とともに IDE を使用する場合は、Application Server の管理ツールを使って JNDI 名付きの JDBC データソースとして Application Server にデータベースを定義する必要があります。その後で、データソースの JNDI 名を使って EJB モジュールを設定する作業を実行できます。
- Sun Java System Application Server 以外のアプリケーションサーバーを使用する場合は、使用するデータベースを JNDI 名付きのデータソースリソースとしてアプリケーションサーバーに設定する必要があります。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。JNDI 名を付けてデータベースを設定した後で、この JNDI 名を使って EJB モジュールを構成できます。「追加 リソース参照」ダイアログで、使用しているアプリケーションサーバー製品に固有のタブを使用してください。

モジュールのアセンブル時に必要なその他の作業

以上では、`CatalogData` EJB モジュールのアセンブルに必要な作業を取り上げました。この項では、`CatalogData` のシナリオでは取り上げられていない EJB モジュールのアセンブル作業について説明します。

EJB モジュールによっては、こうしたほかのアセンブル作業を実行する必要があります。ここでは、実行する可能性のある Web モジュールのアセンブル作業をいくつか取り上げます。

それぞれのモジュールについて、どのようなアセンブル作業が必要かをしっかりと見極めます。たとえば、次の点を確認してください。

- モジュール内の参照がすべてリンクされているか。モジュールには、ほかのモジュールのコンポーネントへの参照が含まれている場合があります。こうした参照は、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した後でのみリンクできます。
- 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、すべて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。これらの詳細は、第 7 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。この詳細は、第 6 章を参照してください。

以下では、`CatalogData` EJB モジュールのシナリオでは取り上げられていない EJB モジュール作業のいくつかを説明します。

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 Java アーカイブ (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、一部の依存関係は IDE に認識されません。こうした IDE に認識されない依存関係には、次のようなものがあげられます。

- モジュール内のエンタープライズ 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。
- モジュール内のエンタープライズ Bean がクラスに動的にアクセスし、クラス名をクラス宣言ではなく文字列として使用する場合。

この 2 つのケースや、類似のケースでは、IDE がエンタープライズ Bean の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が含まれません。これらのファイルがアーカイブに含まれて実行時に利用可能になるように、追加ファイルとして指定します。

追加ファイルを指定するには、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、「プロパティ」を選択します。「プロパティ」セクションの「追加のファイル」の省略符号ボタン (...) をクリックします。**

   「追加のファイル」プロパティエディタが開きます。

2. **この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。**

   a. **「ソース」フィールドで、モジュール内の追加ファイルを選択します。**

   b. **「追加」ボタンをクリックして、選択したファイルを「追加されるファイル」のリストに追加します。**

3. 「了解」をクリックしてプロパティエディタを閉じ、選択したファイルを処理します。

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 の処理から除外する必要がある場合もあります。たとえば、モジュール内のエンタープライズ Bean 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その 1 つです。その場合は、この JAR ファイルを重複した JAR ファイルとして指定して、IDE がこの JAR ファイルの不必要なコピーを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. EJB モジュールノードを右クリックし、「プロパティ」を選択します。「プロパティ」セクションの「除外されるライブラリ JAR ファイル」の省略符号ボタン (...) をクリックします。

   「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。プロパティエディタには、マウントされている JAR ファイルが表示されます。

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「除外されるライブラリ JAR ファイル」のリストに追加します。

3. 「了解」をクリックしてプロパティエディタを閉じ、選択したファイルを処理します。

第4章

シナリオ: Web モジュールと EJB モジュール

図 4-1 は、Java 2 Platform, Enterprise Edition (J2EE プラットフォーム) アプリケーションにアセンブルした Web モジュールと Enterprise JavaBeans (EJB) モジュールを示しています。図に示されている対話のほとんどは、第 2 章と第 3 章で説明されています。たとえば、ユーザーの Web ブラウザと Web モジュール間の HTTP 要求と応答については、第 2 章で取り上げています。図中の矢印 1 で示されたモジュール間の対話は、これらの章で説明されていない対話です。

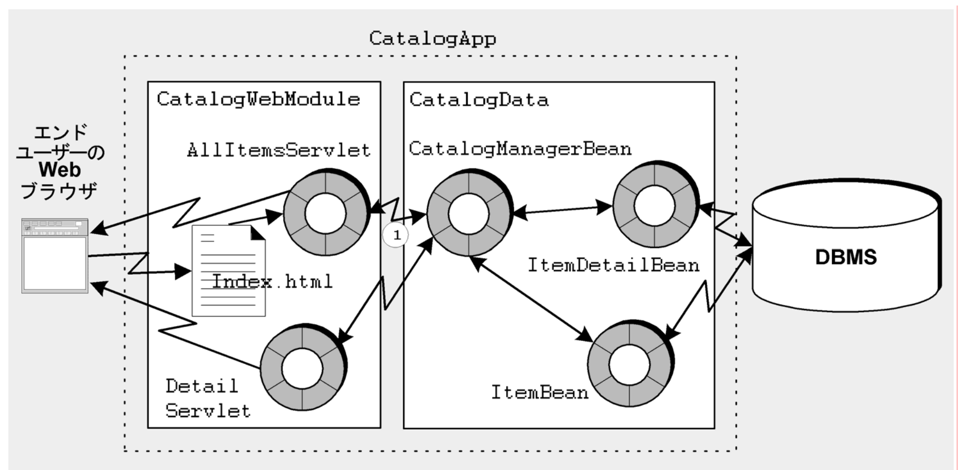


図 4-1 CatalogApp J2EE アプリケーション

アプリケーション内での対話

このシナリオでは、図 4-1 に示された J2EE アプリケーションの使用および対話について説明します。このシナリオでも、第 2 章 と 第 3 章 で説明されているオンラインショッピングアプリケーションを引き続き使用します。ユーザーとの HTTP 対話に必要なプログラミングは、Web モジュール内ですでに完了しています。データベースとの対話に必要なプログラミングは、EJB モジュール内ですでに完了しています。このシナリオでは、この 2 つのモジュールを組み合わせて、終端間の対話を実行する J2EE アプリケーションを作成する方法を説明します。

モジュール間には、アセンブル作業を必要とする対話が 1 つあります。この対話は、Web モジュールから EJB モジュールへのリモートメソッド呼び出しです。この対話に必要なコードのほとんどは、すでに Web モジュールか EJB モジュールに含まれています。すでにモジュールに含まれているコードを以下にまとめます。

- Web モジュールには、JNDI (Java Naming and Directory Interface) ルックアップコードと EJB リソース参照が含まれています。このコードの詳細は、28 ページの「EJB リソース参照」を参照してください。
- EJB モジュールには、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が含まれています。このコードの詳細は、45 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」を参照してください。

この章では、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方法と、モジュール間の対話を構成する方法を説明します。アプリケーションのアセンブルが終われば、そのアプリケーションを配備して実行できます。

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してアセンブル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
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	`CatalogApp` J2EE アプリケーションを作成します。これにより、Java Studio Enterprise の「ファイルシステム」ウィンドウに J2EE アプリケーションノードが作成されます。Web モジュールと EJB モジュールをアプリケーションに追加します。 Web モジュールの Web コンテキストを指定します。 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。

以下の節では、これらのプログラミング作業について説明していきます。

J2EE アプリケーションの作成

統合開発環境 (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配備記述子を表します。アプリケーションに追加するモジュールは、アプリケーションのサブノードとして表示されますが、IDE はモジュールのソースファイルをアプリケーションノードが格納されているディレクトリにコピーしません。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。

- アプリケーション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アプリケーションノードを配置します。
- 一方、アプリケーション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、J2EE アプリケーションとモジュールだけを含むディレクトリを作成する方法もあります。これらのディレクトリは、ソースコードを含むファイルシステムとは別に保持してください。

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 モジュールノードを右クリックし、「新規アプリケーション」を選択します。**

   「新規アプリケーション」ダイアログが開きます。

2. **J2EE アプリケーションに名前を付けて、ファイルシステム内でのその配置場所を選択します。**

   a. **「名前」フィールドに、新しいアプリケーションの名前を入力します。**

      このシナリオの場合は、新しい J2EE アプリケーションの名前は `CatalogApp` になります。

   b. **「新規アプリケーションのパッケージの位置を選択」フィールドで、新しいアプリケーションを作成したいファイルシステム、ディレクトリ、またはノードを選択します。**

   c. **「了解」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、ディレクトリ、またはノードの下に作成されます。手順 1 で右クリックしたモジュールは、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。この手順については、65 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. **「ファイルシステム」ウィンドウで任意のノード (ファイルシステム、パッケージ、またはディレクトリ) を右クリックし、コンテキストメニューの「新規」→「すべてのテンプレート」を選択します。**

   新規ウィザードの「テンプレートを選択」ページが開きます。

2. **J2EE アプリケーションのテンプレートを選択します。**

   a. **「テンプレートを選択」フィールドで、「J2EE」ノードを展開して、「アプリケーション」ノードを選択します。**

   b. **「次へ」をクリックします。**

      「新規オブジェクト名」ページが開きます。

3. **新しい J2EE アプリケーションを定義します。**

   a. **「名前」フィールドに、新しいアプリケーションの名前を入力します。**

      このシナリオの場合は、アプリケーションの名前は `CatalogApp` になります。

   b. **「完了」をクリックします。**

      IDE によってアプリケーションが作成され、手順 1 で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の J2EE アプリケーションノードとして「ファイルシステム」ウィンドウに表示されます。

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。IDE はアプリケーションの配備記述子情報をこのディレクトリに格納しますが、アプリケーションのモジュール内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ないことに注意してください。

J2EE アプリケーションへのモジュールの追加

J2EE アプリケーションを作成した後で、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. **アプリケーションを右クリックし、「モジュールを追加」を選択します。**

   「アプリケーションにモジュールを追加」ダイアログが開きます。

2. **モジュールを 1 つ以上選択します。**

   a. **「このアプリケーションに追加するモジュールを選択してください」フィールドで、アプリケーションに追加するモジュールを選択します。**

   - 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。

   b. **「了解」をクリックします。**

   IDE によって、選択したモジュールが J2EE アプリケーションに追加されます。「ファイルシステム」ウィンドウでは、選択したモジュールを表すノードが IDE によって J2EE アプリケーションノードの下に追加されます。

J2EE アプリケーションにモジュールを追加するとき、IDE によってモジュールとほかの種類のリソース (Java クラス、イメージファイル、その他) との依存関係が認識され、必要なリソースが自動的にアプリケーションに組み込まれます。

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、URL が Web モジュールの Web リソースにマップされます。URL は、URL パターンを URL パスに付加することでマップされます。Sun Java System Application Server の場合、URL パスは一般に次の形式になります。

`http://`*hostname*`:`*port*`/`*web-context*`/`*URL-pattern*

このパスの要素は、次のように決定されます。

- hostname 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。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。

- *web-context* (Web コンテキスト) は、この作業で指定する文字列です。Web コンテキストは、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 JavaServer Pages (JSP) ページを識別する文字列です。URL パターンは、Web モジュールのプロパティシートで設定します。これは、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する前に行うことができます。この作業の詳細は、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る URL パターンは、この作業で割り当てる Web 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、「プロパティ」を選択します。**
2. **「Web コンテキスト」の省略符号ボタン (...) をクリックし、使用する文字列を入力します。**

   図 4-2 に、このシナリオの `CatalogWebModule` のプロパティシートを示します。ここでは、Web コンテキストのプロパティが `catalog` に設定されています。

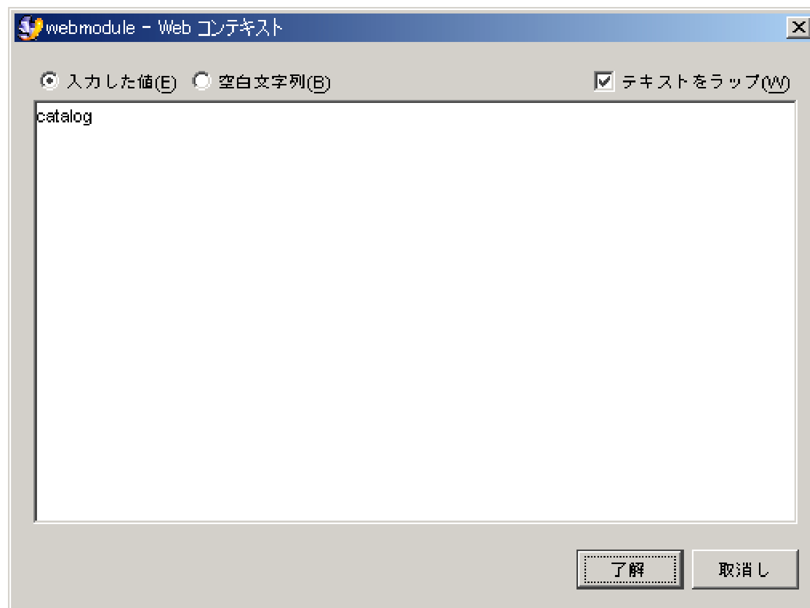


図 4-2 `CatalogWebModule` のプロパティシート

Web コンテキストのプロパティを `catalog` に設定した場合、`CatalogApp` J2EE 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://`*hostname*`:`*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空白に設定されます。CatalogApp J2EE アプリケーションの Web コンテキストをデフォルトのブランクにすると、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://*hostname*:*port*/*URL-pattern*

EJB 参照のリンク

CatalogWebModule には、JNDI ルックアップコードと EJB リソース参照が含まれています。CatalogWebModule コードと配備記述子は、CatalogManagerBeanHome と CatalogManagerBean の種類のインタフェースを指定します。ルックアップコードとリソース参照の設定方法については、28 ページの「EJB リソース参照」を参照してください。

CatalogData EJB モジュールには、CatalogManagerBeanHome および CatalogManagerBean という種類のリモートインタフェースと、インタフェースを実装する CatalogManagerBeanBean という名前の Bean クラスが含まれています。インタフェースの作成方法の詳細は、45 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」を参照してください。

CatalogApp J2EE アプリケーションを実行する前に、Web モジュールの EJB 参照を EJB モジュールのエンタープライズ Bean にリンクする必要があります。

場合によっては、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の参照をリンクします。CatalogWebModule の参照は、Web モジュールのプロパティシートでリンクされていません。このシナリオでは、参照は CatalogApp プロパティシートでリンクされます。

アプリケーションノードのプロパティシートの EJB 参照をリンクするには、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、「プロパティ」を選択します。「プロパティ」セクションの「EJB 参照」の省略符号ボタン (...) をクリックします。**

   「EJB 参照」プロパティエディタが開きます。

2. **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によって識別されます。

   図 4-3 は、CatalogApp J2EE アプリケーションの「EJB 参照」プロパティエディタを示しています。ここには、CatalogWebModule で宣言されている 1 つの EJB 参照があります。この参照には ejb/CatalogManagerBean という名前が付けられており、解決されていません。

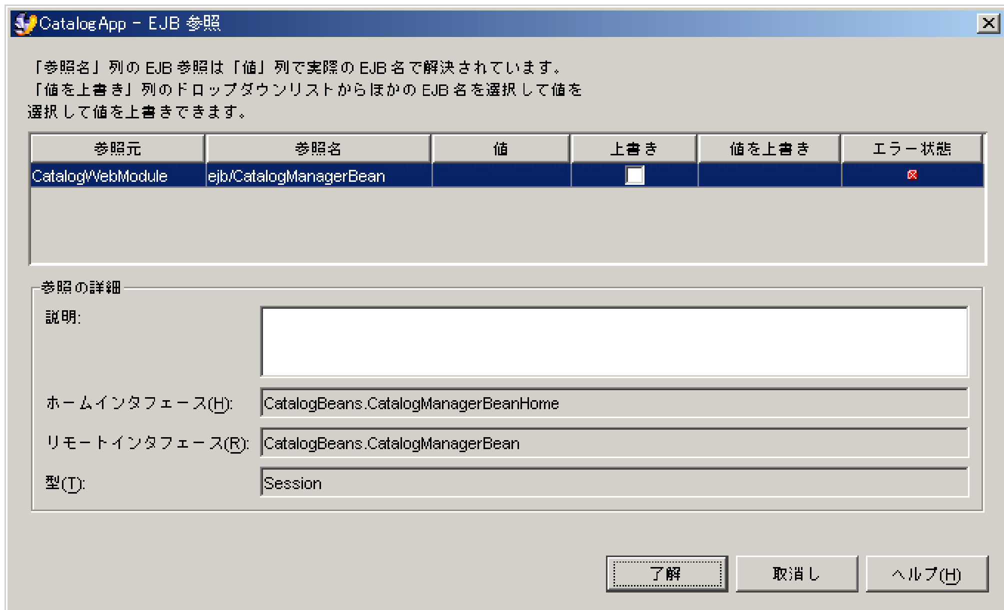


図 4-3　　リンクされていない「EJB 参照」

参照が解決されていない場合は、「値」フィールドは空になり、「エラー状態」フィールドにエラーを表すアイコンが表示されます。

3. **解決されていない参照があればリンクします。**

「値を上書き」フィールドをクリックします。フィールドには、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。

図 4-4 は、図 4-3 と同じ EJB 参照を示していますが、上書きによって参照がリンクされています。「値を上書き」フィールドで指定したエンタープライズ Bean は、アプリケーションの実行時、`CatalogWebModule` 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。図 4-4 の「値を上書き」フィールドには、`CatalogData` EJB モジュールの `CatalogBeans.CatalogManagerBean` というエンタープライズ Bean が指定されています。

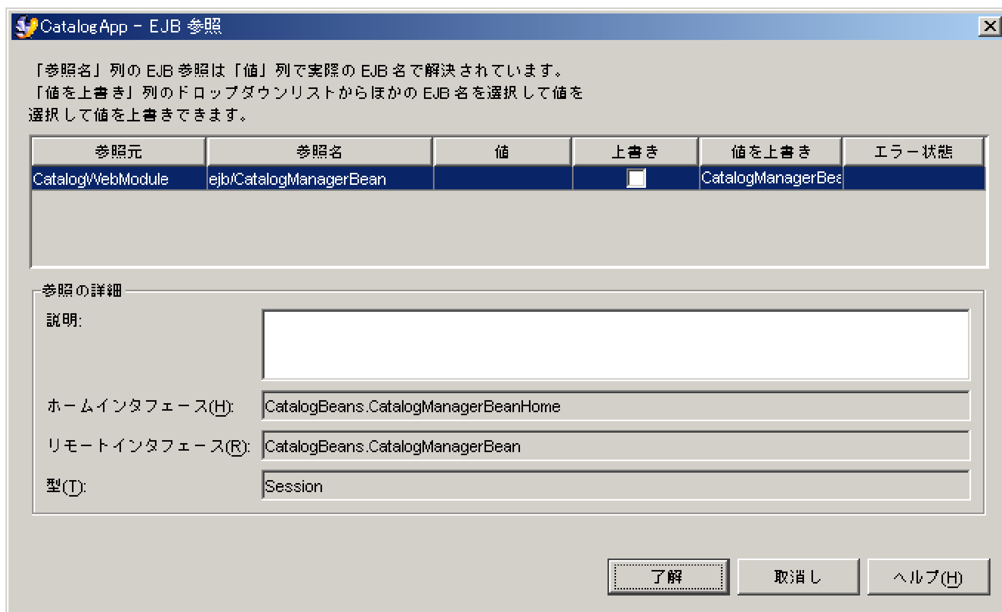


図 4-4　　上書きによってリンクされた「EJB 参照」

参照が解決すると、リンクされたエンタープライズ Bean の名前が「値を上書き」フィールドに表示され、「エラー状態」フィールドは空になります。

その他のアセンブル作業

これまでの項では、`CatalogApp` J2EE アプリケーションのアセンブルに必要な作業を取り上げました。この項では、`CatalogApp` のシナリオで必要とされない J2EE アプリケーションのアセンブル作業について説明します。

J2EE アプリケーションによっては、こうしたほかのアセンブル作業を実行する必要があります。ここでは、実行する可能性のあるアプリケーションのアセンブル作業を取り上げます。

環境エントリの上書き

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上書き (オーバーライド) することが必要な場合があります。これは、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います。

環境エントリ値を上書きするには、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、「プロパティ」を選択します。「プロパティ」セクションの「環境エントリ」の省略符号ボタン (...) をクリックします。**

   「環境エントリ」プロパティエディタが開きます。このエディタには、アプリケーションで宣言されているすべての環境エントリが表示されます。環境エントリは、参照名とモジュールによって識別されます。

2. **アプリケーションの環境エントリを調べます。**

   図 4-5 は、`CatalogApp` アプリケーションの「環境エントリ」プロパティエディタを示しています。プロパティエディタには、Web モジュールで宣言された環境エントリが表示されます (Web モジュールでの環境エントリの宣言方法については、38 ページの「リソース環境エントリ参照の設定」を参照してください)。

   「値」フィールドには `100` と表示されています。これは、Web モジュールのプロパティシートで設定された初期値です。

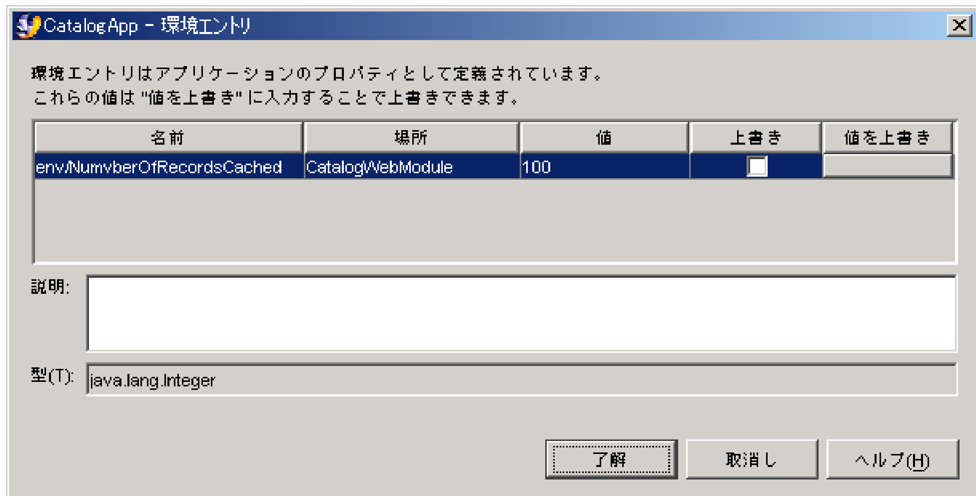


図 4-5　J2EE アプリケーションの「環境エントリ」プロパティエディタ

3. アセンブルして作成されたアプリケーションの初期値が適していない場合は、これらを上書きします。

a. 上書きする環境エントリの「値を上書き」フィールドをクリックし、次に省略符号ボタン (...) をクリックします。

「値を上書き」ダイアログが開きます。

b. 「値」フィールドに、上書きする値を入力します。「了解」をクリックしてダイアログを閉じ、プロパティエディタに戻ります。

図 4-6 は、上書きフィールドに値が入力された「環境エントリ」プロパティエディタを示しています。

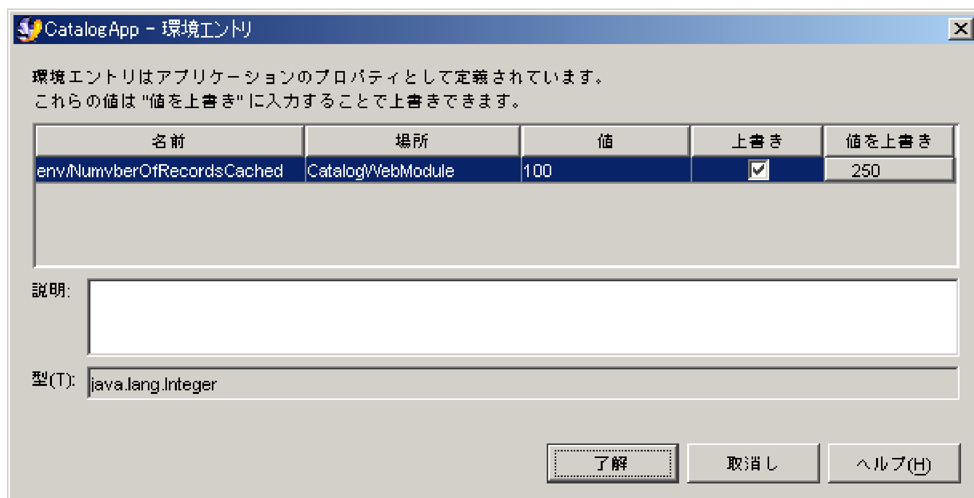


図 4-6　　環境エントリ値の上書き

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、環境エントリの値を Web 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上書きしたほうがいいでしょう。別の開発者が別のアプリケーションで Web モジュールを再利用して、Web モジュールのプロパティシートで環境エントリの値を変更すると、このアプリケーションの再配備時にアプリケーションの値が変更されてしまいます。

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でプロパティを設定し、配備記述子の内容を制御します。プロパティを設定することによって、配備記述子の内容を制御できます。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。

モジュールの配備記述子の表示

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、次のようにします。

- **J2EE アプリケーションノード、取り込まれた EJB モジュールノード (J2EE アプリケーションノードの下の EJB モジュールノード)、または取り込まれた Web モジュールを右クリックして、「配備記述子を表示」を選択します。**

配備記述子がソースエディタに読み取り専用モードで表示されます。

モジュール配備記述子の編集

EJB モジュール配備記述子を編集するには、次のようにします。

- **EJB モジュールノードを右クリックし、「配備記述子」>「最終的な編集」を選択します。**

警告ボックスが表示されます。「はい」をクリックすると、配備記述子がソースエディタに表示されます。

Web モジュール配備記述子を編集するには、次のようにします。

- **Web モジュールの web ノードを右クリックし、「編集」を選択します。**

配備記述子がソースエディタに表示されます。

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、Java 2 Platform, Enterprise Edition (J2EE プラットフォーム) アプリケーションにアセンブルした Web モジュールと Enterprise JavaBeans (EJB) モジュールを示しています。モジュール間の対話は非同期メッセージングです。Web モジュールはキューにメッセージを送信し、EJB モジュール内のメッセージ駆動型エンタープライズ Bean はキューからメッセージを読み取ります。キューへのメッセージ送信は、図中の矢印 1 で示されています。キューからのメッセージ読み取りは、矢印 2 で示されています。メッセージ駆動型 Bean はメッセージを読み取ってから、モジュール内のほかのエンタープライズ Bean のメソッドを呼び出して処理を開始します。

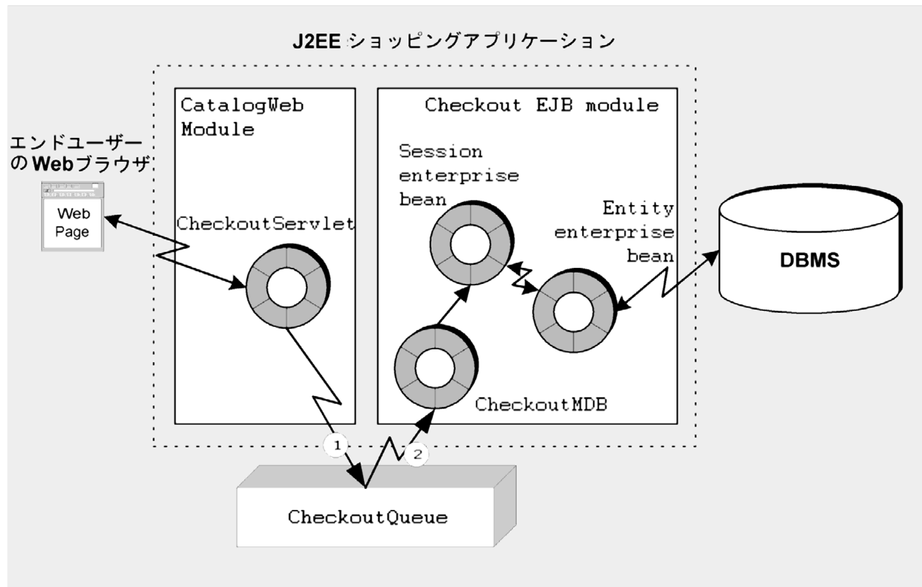


図 5-1 キューモードのメッセージ駆動型 Bean を含む J2EE アプリケーション

アプリケーション内での対話

このシナリオでは、図 5-1 に示された J2EE アプリケーションの使用法と対話について説明します。このシナリオでもショッピング Web サイトアプリケーションを使用しますが、異なる EJB モジュールに実装される別の種類の対話を示します。

このシナリオでは、利用者は商品を選択し、ショッピングカートに追加することによって、Web モジュールで定義された Web ページと対話します。利用者が買い物を終了するときに、EJB モジュールとの同期対話が実行されます。第 2 章、第 3 章、および第 4 章は、この種のロジックを J2EE アプリケーションに実装する方法を示しています。

最終的に、利用者はチェックアウト可能になります。利用者は、ショッピングカードの内容の確認、配送方法の選択、合計金額の承認、クレジットカード番号の入力を行います。利用者は注文を確認し、承認して、サイトを離れます。その後アプリケーションは注文を処理し、利用者に電子メールで通知します。このチェックアウトシナリオで扱っている特定の対話の概要は、次のとおりです。

1. Web モジュールが、注文した商品、配達先住所、配送方法、および支払い方法を示すページを表示します。利用者は、注文を承認します。

2. 利用者が注文を承認すると、Web モジュールはメッセージキューにメッセージを送信します。このメッセージは、処理される注文を識別します。

3. メッセージキューはアプリケーションの外部にあります。これはアプリケーションサーバーによって維持されています。

4. キューは、注文処理を実行する EJB モジュール内のメッセージ駆動型エンタープライズ Bean に通知します。キューは、`onMessage` イベントでメッセージ駆動型 Bean に通知し、メッセージをパラメータとして渡します。

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を含んでいるわけではありません。メッセージを調べ、注文完了処理を開始するだけです。

   メッセージ駆動型 Bean は、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注文完了処理を開始します。これは、メッセージ駆動型 Bean の典型的な使い方です。注文の詳細は、データベースに保存されます。

6. 注文が処理されると、アプリケーションは電子メールメッセージを送信して利用者に通知します。

この章では、メッセージキューとキュー接続ファクトリの設定方法、およびキューを使用するために送信側モジュールと受信側モジュールを構成する方法を示します。

メッセージ駆動型通信のプログラミング

表 5-1 に、このシナリオで説明し、図 5-1 に示されているメッセージ駆動型対話を実装するために必要なプログラミングをまとめます。

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	アプリケーションサーバーの管理ツールを使用して、キュー名 `CheckoutQueue` とキュー接続ファクトリ名 `CheckoutQCF` を統合開発環境 (IDE) に設定します。
Web モジュール	メッセージを送信するサーブレット `CheckoutServlet` を作成します。以下のコードを、`CheckoutServlet` の `processRequest` メソッドに追加します。 1. JNDI (Java Naming and Directory Interface) ルックアップを使用して、`CheckoutQueue` と `CheckoutQCF` への参照を取得します。 2. `CheckoutQueue` メソッドを呼び出して、メッセージを書式化して送信します。
EJB モジュール	メッセージ駆動型エンタープライズ Bean `CheckoutMDB` を作成します。 `CheckoutMDB` プロパティシートを使用して、`CheckoutMDB` を `CheckoutQueue` のメッセージの宛先に設定します。 `CheckoutMDB` の `onMessage` メソッドをコーディングします。

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。

アプリケーション全体のプログラミングには、ほかのプログラミング作業が必要です。こうした作業には、Web コンポーネントと Web モジュールの作成、セッションおよびエンティティエンタープライズ Bean の作成、および EJB モジュールの作成が含まれます。それらの作業については、それぞれの話題を扱っている章を参照してください。この章では、メッセージ駆動型の対話について説明します。

アプリケーションサーバーの設定

チェックリスト対話の設計では、Web モジュールがキューにメッセージを送信し、EJB モジュールがキューからメッセージを読み取ってメッセージで指定された注文を処理することが必要です。この対話では、キューとキュー接続ファクトリが必要とされます。

JMS (Java Message Service) キューとキュー接続ファクトリを表すアプリケーションサーバーのリソースは、アプリケーションサーバーに作成する必要があります。キューの作成は、Web モジュールとメッセージ駆動型 Bean のマッピングを確立するために必要です。キューとキュー接続ファクトリは、IDE 外部で作成される、アプリケーションサーバーのリソースです。

- スタンドアロンの開発環境で作業を行っている場合は、その開発のキューおよびキュー接続ファクトリを自ら管理する必要があります。
- 一方、管理されたテスト環境または本番環境で作業している場合は、キューとキュー接続ファクトリの定義、構成、および管理は、おそらくシステム管理担当者の仕事になります。この場合は、キューとキュー接続ファクトリの JNDI 名を取得します。それでもやはり、アプリケーションがこれらのリソースをどのように使用するかを理解するために、キューとキュー接続ファクトリの設定に関する以下の作業手順を読むことをお勧めします。

この節の作業を行うには、その前にアプリケーションサーバーと IDE アプリケーションサーバープラグインをインストールしておく必要があります。また、アプリケーションサーバーのインスタンスも必要です。アプリケーションサーバープラグインとサーバーインスタンスは、「実行時」ウィンドウに表示されるノードによって示されます。アプリケーションサーバープラグインとアプリケーションサーバーインスタンスノードの詳細は、115 ページの「サーバー製品ノード」を参照してください。

キューの設定

この節では、Sun Java System Application Server のメッセージキューを設定する方法を説明します。ほかのアプリケーションサーバーの場合も、作業手順はほとんど同じになります。

Sun Java System Application Server にキューを追加するには、次のようにします。

1. 「実行時」ウィンドウをクリックします。
2. 「Sun Java System Application Server 7」ノードを展開します。
3. 「登録されていない JMS リソース」ノードを右クリックし、「新規 JMS リソースを追加」を選択します。

   新規ウィザードの「JMS リソース」ページが開きます。
4. キューを定義します。

   a. 「JNDI 名」フィールドに「**`jms/CheckoutQueue`**」と入力します。

   b. 「リソースの種類」フィールドが `javax.jms.Queue` に設定されていることを確認します。

   c. 「次へ」をクリックします。

   新規ウィザードの「JMS プロパティ」ページが開きます。

5. imqDestinationName プロパティを定義します。

   a. 「追加」をクリックします。

      最初のプロパティ行が入力可能になります。

   b. 名前フィールドで「imqDestinationName」を選択します。

   c. 「値」フィールドに「**Checkout**」と入力します。

      Checkout は、作成する物理キューの名前です。J2EE アプリケーションは、指定した JNDI 名の CheckoutQueue を使って、Checkout という名前のキューにアクセスします。

   d. 「完了」をクリックします。

      「登録を継続しますか ?」ダイアログが開きます。

6. キューを登録します。

   a. 「登録」をクリックします。

      「JMS リソースの登録」ダイアログが開きます。

   b. 「サーバーインスタンス」フィールドで、キューを登録するアプリケーションサーバーインスタンスを選択します。

      J2EE アプリケーションを配備するアプリケーションサーバーを選択します。

   c. 「登録」をクリックします。

      「リソースが登録されました」というメッセージが表示されます。

   d. 「閉じる」をクリックします。

キュー接続ファクトリの設定

この節では、Sun Java System Application Server のキュー接続ファクトリを設定する方法を説明します。ほかのアプリケーションサーバーの場合も、作業手順はほとんど同じになります。

Sun Java System Application Server にキュー接続ファクトリを追加するには、次のようにします。

1. 「登録されていない JMS リソース」ノードを右クリックし、「新規 JMS リソースを追加」を選択します。

   新規ウィザードの「JMS リソース」ページが開きます。

2. キュー接続ファクトリを定義します。

   a. 「JNDI 名」フィールドに「**jms/CheckoutQCF**」と入力します。

   b. 「リソースの種類」フィールドが javax.jms.QueueConnectionFactory に設定されていることを確認します。

   c. 「完了」をクリックします。

      「登録を継続しますか ?」ダイアログが開きます。

3. キュー接続ファクトリを登録します。

   a. 「登録」をクリックします。

      「JMS リソースの登録」ダイアログが開きます。

   b. 「サーバーインスタンス」フィールドで、キュー接続ファクトリを登録するアプリケーションサーバーインスタンスを選択します。

      キューを作成したときに選択したアプリケーションサーバーインスタンスを選択します。

   c. 「登録」をクリックします。

      「リソースが登録されました」というメッセージが表示されます。

   d. 「閉じる」をクリックします。

Web モジュールのプログラミング

このシナリオでは、注文の最終処理を要求するメッセージを CheckoutServlet が送信します。このメッセージは、処理される注文を識別します。メッセージを送信するために、CheckoutServlet はキュー接続ファクトリとキューのメソッドを呼び出します。

キューメソッドとキュー接続ファクトリメソッドを呼び出すには、CheckoutServlet はキューとキュー接続ファクトリへの参照が必要です。CheckoutServlet は JNDI ルックアップを使って、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ファクトリの参照を取得します。

ほとんどの J2EE 参照ルックアップと同様に、キューおよびキュー接続ファクトリ参照ルックアップは、次の 2 つの部分に分けられます。

- **JNDI ルックアップコード**。サーブレットには、JNDI 命名機能を使用してキューまたはキュー接続ファクトリの参照を取得するコードが含まれています。
- **参照の宣言**。これによって、JNDI lookup 文で使用される名前がキューまたはキュー接続ファクトリの実際の JNDI 名にマップされます。

キューとキュー接続ファクトリは、アプリケーションサーバーの名前付きのリソースです。アプリケーションコンポーネントは JNDI 名を使って参照を取得します。キューとキュー接続ファクトリへの JNDI 名の指定方法は、75 ページの「アプリケーションサーバーの設定」を参照してください。

JNDI ルックアップのコード

コード例 5-1 は、CheckoutServlet の processRequest メソッドを示しています。processRequest メソッドは、JNDI ルックアップを実行します。キュー参照とキュー接続ファクトリ参照を取得すると、processRequest はキューとキュー接続ファクトリのメソッドを呼び出して、メッセージを作成し、送信します。コード例には、これらの操作を識別するコメントが含まれています。

コード例 5-1 はサーブレットのコード例ですが、どの種類の J2EE コンポーネントも同様のコードを使ってメッセージを送信できます。このコードは、メッセージの送信側として機能するアプリケーションクライアントやエンタープライズ Bean で再利用できます。

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。

コード例 5-1　CheckoutServlet の processRequest メソッド

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

// ...

protected void processRequest(HttpServletRequest request,
                              HttpServletResponse response)
                              throws ServletException, IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  // ページをここに出力

  // デフォルトのメソッド本体を削除し、次の行を挿入する
  Context jndiContext = null;
  javax.jms.TextMessage msg = null;
  QueueConnectionFactory queueConnectionFactory = null;
  QueueConnection queueConnection = null;
  QueueSession queueSession = null;
  Queue queue = null;
  QueueSender queueSender = null;
```

コード例 5-1　　CheckoutServlet の processRequest メソッド (続き)

```
    TextMessage message = null;

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

    try {
      // アプリケーションサーバー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();
    }
    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対して
      // JNDI ルックアップを実行する
      queueConnectionFactory = (QueueConnectionFactory)
                   jndiContext.lookup("java:comp/env/jms/CheckoutQCF");
      queue = (Queue)
                  jndiContext.lookup("java:comp/env/jms/CheckoutQueue");
    }
    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("Order #33454344");
      queueSender.send(message);
    }
    catch (JMSException e) {
      out.println("Exception occurred:" + e.toString());
    }
    finally {
      if (queueConnection != null) {
        try {
          queueConnection.close();
```

コード例 5-1　CheckoutServlet の processRequest メソッド (続き)

```
      }
      catch (JMSException e) {}
    } // if の終わり
  } // finally の終わり
  // 挿入するコードの終わり

  out.close();
}
```

キューのリソース環境参照

リソース環境参照は、モジュールの配備記述子に示されます。リソース環境参照は、lookup 文で使用される参照名を、アプリケーションサーバー環境の JNDI 名にマップします。

キューのリソース環境参照を設定するには、次のようにします。

1. **Web モジュールの web ノードを右クリックし、「プロパティ」を選択します。「参照」セクションの「リソース環境参照」の省略符号ボタン (...) をクリックします。**

   「リソース環境参照」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。**

   「追加 リソース環境参照」ダイアログが開きます。

3. **リソース環境参照を宣言します。**

   a. **「名前」フィールドに、`lookup` 文に示されている参照名を入力します。**

      図 5-1 では、「名前」フィールドに値 `jms/CheckoutQueue` が示されています。これは、コード例 5-1 で使用されていた参照名です。

   b. **「型」フィールドで、`javax.jms.Queue` を選択します。**

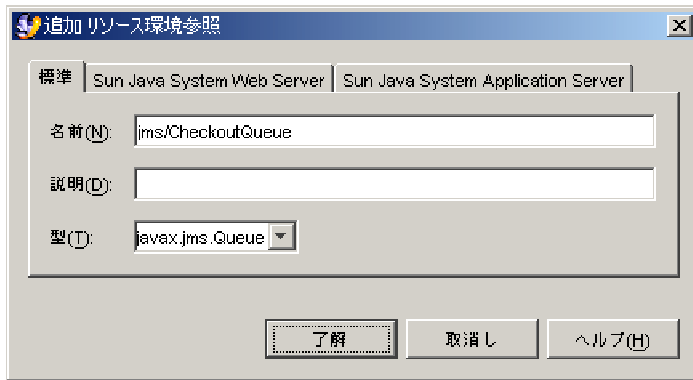


図 5-2　CheckoutQueue のリソース環境参照の追加

4. 参照名を JNDI 名にマップします。

a. 「追加 リソース参照」ダイアログの「Sun Java System Application Server」タブをクリックします。

b. 「JNDI 名」フィールドに、キューの JNDI 名を入力します。

図 5-3 では、「JNDI 名」フィールドに値 jms/CheckoutQueue が示されています。この値は、「標準」タブの参照名を、CheckoutQueue という名前のキューにマップします。キューの命名方法については、75 ページの「アプリケーションサーバーの設定」を参照してください。

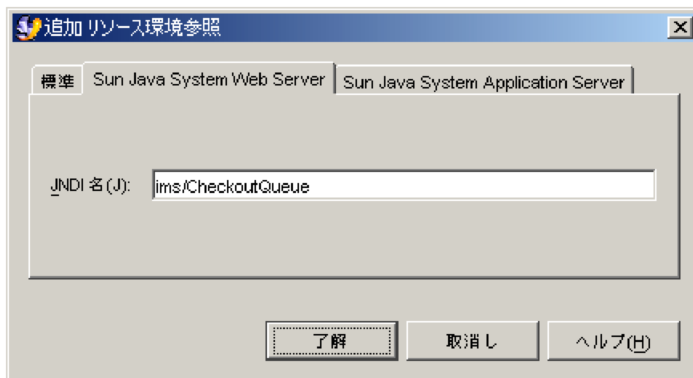


図 5-3　キュー参照の JNDI 名の指定

キュー接続ファクトリのリソース参照

キュー接続ファクトリのリソース参照を設定するには、次のようにします。

1. Web モジュールの web ノードを右クリックし、「プロパティ」を選択します。「参照」セクションの「リソース参照」の省略符号ボタン (...) をクリックします。

   「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ます。

   「追加 リソース参照」ダイアログが開きます。

3. リソース参照を宣言します。

   a. 「名前」フィールドに、lookup 文に示されている参照名を入力します。

   図 5-4 では、「名前」フィールドに値 jms/CheckoutQCF が示されています。これは、コード例 5-1 で使用されていた参照名です。

   b. 「型」フィールドで、javax.jms.QueueConnectionFactory を選択します。

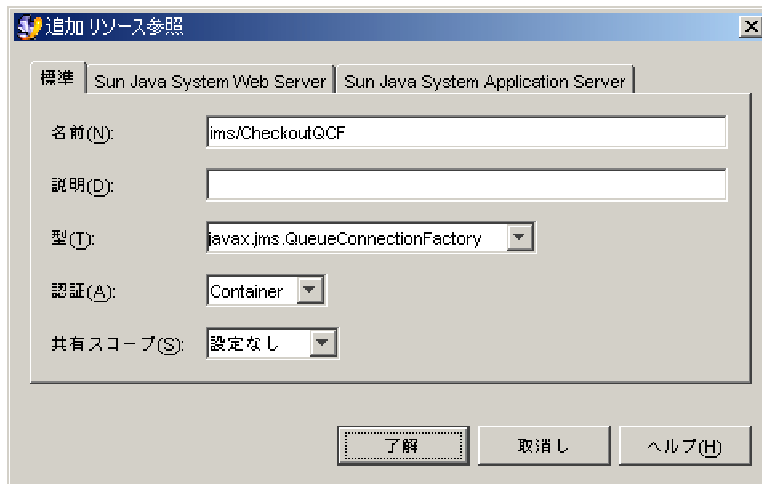


図 5-4　キュー接続ファクトリのリソース参照

4. 参照名を JNDI 名にマップします。

   a. 「追加 リソース参照」ダイアログの「Sun Java System Application Server」タブをクリックします。

b. 「JNDI 名」フィールドに、キュー接続ファクトリの JNDI 名を入力します。

図 5-5 では、「JNDI 名」フィールドに値 jms/CheckoutQCF が示されています。この値は、「標準」タブの参照名を、CheckoutQCF という名前のキュー接続ファクトリにマップします。キュー接続ファクトリの命名方法については、75 ページの「アプリケーションサーバーの設定」を参照してください。

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。

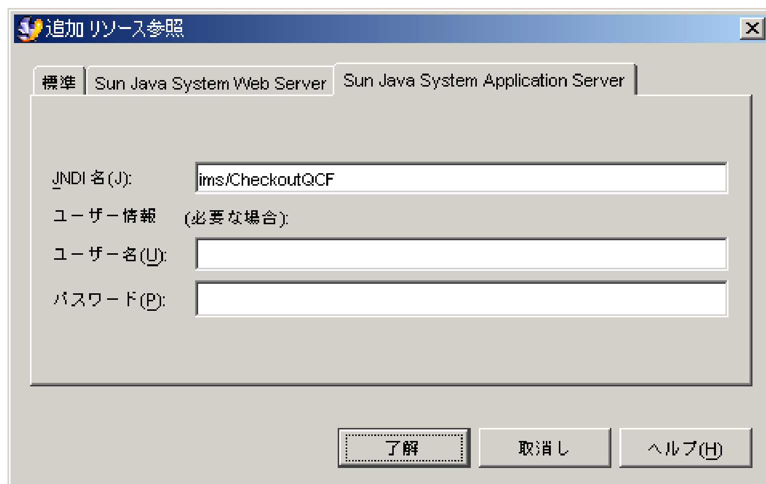


図 5-5　キュー接続ファクトリ参照の JNDI 名

EJB モジュールのプログラミング

このチェックアウトシナリオでは、利用者のチェックアウト要求を処理するビジネスロジックは、Checkout EJB モジュール内にあります。78 ページの「Web モジュールのプログラミング」には、Web モジュールがキュー参照とキュー接続ファクトリ参照をルックアップしてメッセージを送信する方法が示されています。この対話を完了するためには、EJB モジュール内のメッセージ駆動型 Bean がキューからメッセージを受信する必要があります。

メッセージ駆動型 Bean は、プログラムの参照は使用しません。メッセージ駆動型 Bean は、JNDI ルックアップコードは必要としません。メッセージ駆動型 Bean のプロパティシートを使用して、使用するキューとキュー接続ファクトリを指定します。これらのプロパティを設定すると、配備記述子のタグが設定されます。

メッセージ駆動型 Bean を構成するために設定するプロパティは、以下のとおりです。

- **「メッセージ駆動型送信先」プロパティ。**メッセージ駆動型 Bean が使用する送信先の種類を指定します。
- **「MDB 接続ファクトリ」プロパティ。**キュー接続ファクトリを指定します。
- **「JNDI 名」プロパティ。**キューを指定します。このプロパティは、Sun Java System Application Server に固有です。ほかのアプリケーションサーバーでは、別のプロパティを使用してキューを指定します。

アプリケーションが配備されるとき、アプリケーションサーバーは配備記述子に指定されたキュー接続ファクトリを自動的に使用して、メッセージ駆動型 Bean から配備記述子に指定されたキューへの接続を開きます。

「メッセージ駆動型送信先」プロパティの構成

キューとキュー接続ファクトリを指定する前に、メッセージ駆動型 Bean をキューのコンシューマとして構成する必要があります。

メッセージ駆動型 Bean をキューのコンシューマとして構成するには、次のようにします。

1. **メッセージ駆動型 Bean の論理ノードを右クリックし、「プロパティ」を選択します。「プロパティ」セクションの「メッセージ駆動型送信先」の省略符号ボタン (...) をクリックします。**

   「メッセージ駆動型送信先」プロパティエディタが開きます。

2. **メッセージ駆動型 Bean をキューのコンシューマとして指定します。**

   a. **「送信先のタイプ」フィールドで、「キュー」を選択します。**

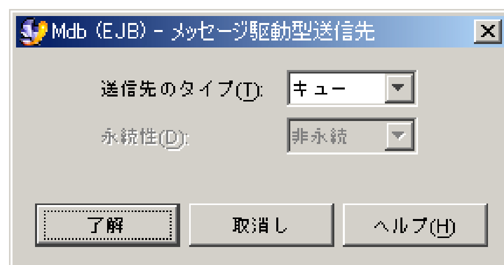


図 5-6　「メッセージ駆動型送信先」プロパティシート

   b. **「了解」をクリックします。**

接続ファクトリの指定

キュー接続ファクトリのメッセージ駆動型 Bean を構成するには、次のようにします。

1. メッセージ駆動型 Bean の論理ノードを右クリックし、「プロパティ」を選択します。「Sun Java System AS」セクションの「MDB 接続ファクトリ」の省略符号ボタン (...) をクリックします。

   「MDB 接続ファクトリ」プロパティエディタが開きます。

2. キュー接続ファクトリを指定します。

   a. 「JNDI 名」フィールドに、キュー接続ファクトリの JNDI 名を入力します。

   図 5-7 では、「JNDI 名」フィールドに値 jms/CheckoutQCF が示されています。jms/CheckoutQCF は、送信側の Web モジュールで指定されたキュー接続ファクトリです。

   b. ユーザー名とパスワードが必要な場合は、「名前」フィールドと「パスワード」フィールドにそれらを入力します。

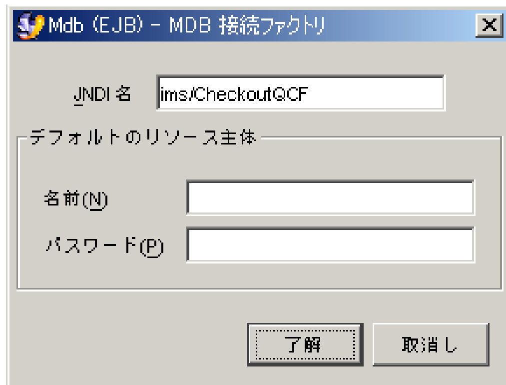


図 5-7　メッセージ駆動型 Bean の「MDB 接続ファクトリ」プロパティエディタ

   c. 「了解」をクリックします。

キューの指定

キューのメッセージ駆動型 Bean を構成するには、次のようにします。

1. メッセージ駆動型 Bean のローカルノードを右クリックし、「プロパティ」を選択します。「Sun Java System AS」セクションの「JNDI 名」の省略符号ボタン (...) をクリックします。

   「JNDI 名」プロパティエディタが開きます。

2. キューを指定します。

このシナリオでは、jms/CheckoutQueue を使用します。jms/CheckoutQueue は、送信側の Web モジュールで指定されたキューです。

onMessage メソッドのコーディング

アプリケーションサーバーは、メッセージ駆動型 Bean の onMessage メソッドを呼び出すことで、Bean にメッセージを送信します。メッセージは、onMessage メソッドのパラメータとして送信されます。コード例 5-2 は、onMessage メソッドを示しています。この例では、メッセージがパラメータとして渡されていて、メッセージ処理コードの追加位置が分かります。

コード例 5-2 onMessage メソッド

```
public void onMessage(javax.jms.Message aMessage) {
        // ここでメッセージを処理
    }
```

図 5-1 に示されているこのシナリオでは、メッセージ駆動型 Bean が同じ EJB モジュール内のセッション Bean のビジネスメソッドを直ちに呼び出します。セッション Bean は注文の処理を制御します。これは、onMessage の典型的な動作です。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。

メッセージ駆動型 Bean は、EJB ローカル参照を使ってセッション Bean を呼び出します。EJB ローカル参照を使ってメソッド呼び出しを実装する方法については、47 ページの「ローカル EJB 参照の JNDI ルックアップコード」と 49 ページの「ローカル EJB リソース参照」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、メッセージを送信する Web モジュールとメッセージを受信する EJB モジュールがアセンブルされて作成された J2EE アプリケーションが示されています。これらのモジュールは、この章で説明されているようにプログラミングします。そしてアプリケーションを作成し、2 つのモジュールをアプリケーションに追加します。どちらのモジュールも、CheckoutQueue および CheckoutQCF を使用するように構成されています。メッセージ駆動型の対話では、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、63 ページの「J2EE アプリケーションの作成」を参照してください。

第6章

トランザクション

この章では、Enterprise JavaBeans (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」プロパティによって決定します。

統合開発環境 (IDE) の EJB の新規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選択すると、デフォルトのトランザクション属性プロパティ値を持つエンタープライズ 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、「プロパティ」を選択します。「プロパティ」セクションの「トランザクション設定」の省略符号ボタン (...) をクリックします。**

  「トランザクション設定」プロパティエディタが開きます。

  図 6-1 に、第 3 章で取り上げられている `CatalogData` 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。図 6-1 に表示されている値は、トランザクション属性のデフォルトの設定です。

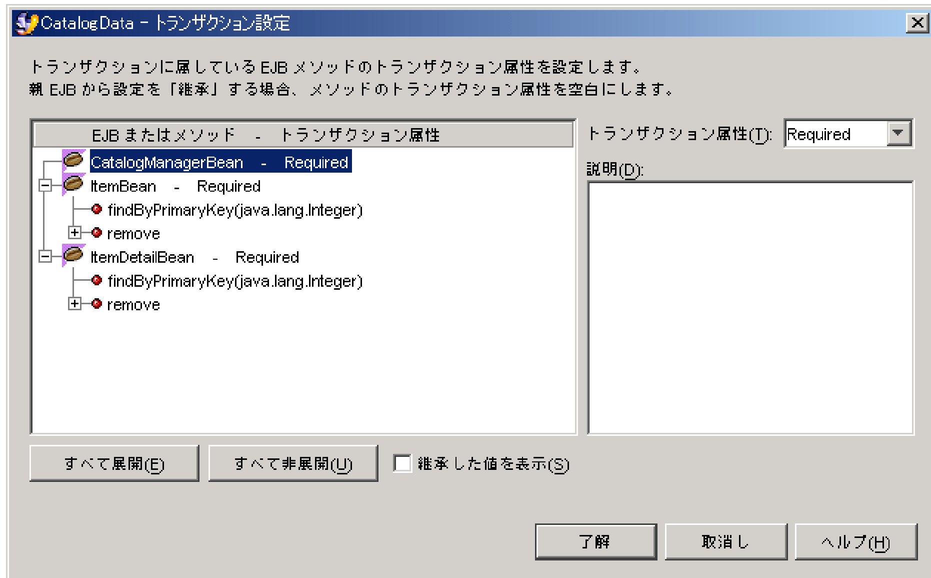


図 6-1　　デフォルトのトランザクション属性

トランザクション属性のデフォルト設定は、「EJB またはメソッド - トランザクション属性」フィールドに表示されます。この表示内容を理解するために、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。それぞれのエンタープライズ Bean はノードによって表されています。
- 各エンタープライズ Bean 名の後ろに、Bean のトランザクション属性が示されています。たとえば、図 6-1 の最初のノード、`CatalogManagerBean` の後ろには、「`Required`」と示されています。「`Required`」は、トランザクション属性プロパティのデフォルト設定です。図 6-1 では、すべてのエンタープライズ Bean がデフォルトの設定になっています。
- エンタープライズ Bean を表すノードを展開すると、各エンタープライズ Bean のメソッドを表すノードが表示されます。図 6-1 では、すべてのエンタープライズ Bean ノードが展開されています。
- メソッドは、独自のトランザクション属性値を持っています。トランザクション属性値はメソッド名の後に表示されます。トランザクション属性値が空白の場合、メソッドはエンタープライズ Bean からトランザクション属性値を継承します。
- 図 6-1 では、トランザクション属性値が表示されているメソッドノードはありません。すべてのメソッドが、デフォルト設定の値「`Required`」を継承しています。

「Required」は、Java 2 Platform, Enterprise Edition Specification で定義されているトランザクション値の 1 つです。トランザクション属性が「Required」に設定されているメソッドの規則を以下に示します。

- アクティブなトランザクションがないときに「Required」属性を持つメソッドが呼び出されると、アプリケーションサーバー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「Required」属性を持つメソッドが呼び出されると、アプリケーションサーバーはアクティブなトランザクション内でメソッドを実行します。

これは、エンタープライズ Bean メソッド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。エンタープライズ Bean ビジネスロジック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。クライアントはセッション Bean を呼び出し、セッション Bean はエンティティ Bean のメソッドを呼び出します。

たとえばクライアントに、関連する 2 つの新しいデータベースレコードのデータがあると仮定します。この場合セッション Bean は、2 つのエンティティ Bean の生成メソッドを呼び出すことで、データベース挿入を生成します。2 つのデータベース挿入は、同じトランザクションで実行される必要があります。この種のトランザクションを図 6-2 に示します。

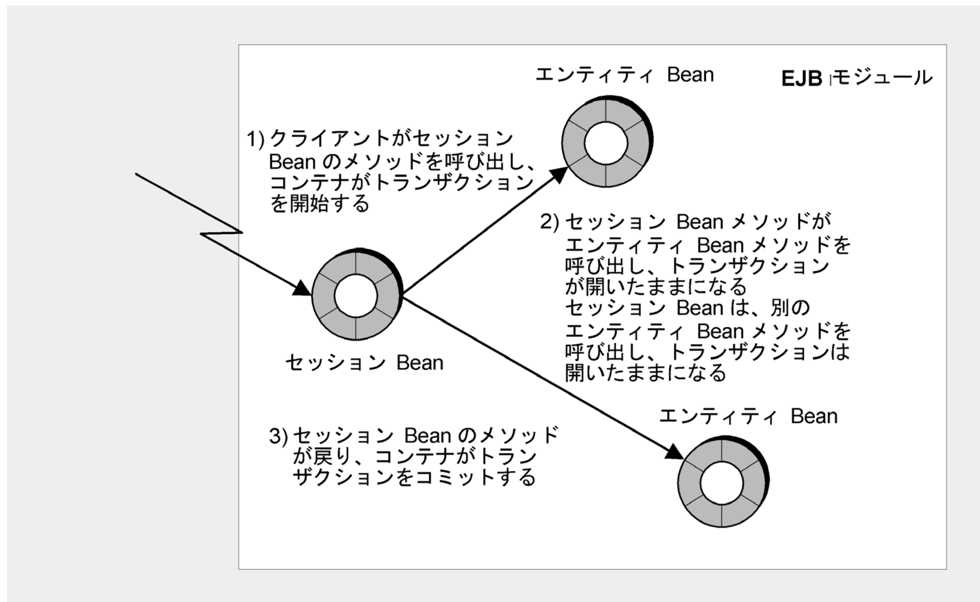


図 6-2　　複雑なトランザクション

これらのエンティティ Bean をアセンブルして EJB モジュールを作成するときは、アプリケーションサーバーにビジネストランザクション境界を認識させます。ここで、アプリケーションサーバーがクライアントの呼び出し後に EJB モジュールが実行するすべての作業を、単一のデータベーストランザクションで行えるようにすると仮定します。その場合、アプリケーションサーバーには以下のこと行わせます。

- クライアントがセッション Bean を呼び出したときに、トランザクションを開く。
- セッション Bean がエンティティ Bean を呼び出している間、トランザクションを開いたままにする。
- セッション Bean によるエンティティ 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 のメソッドが完了したらトランザクションを閉じる。

アプリケーションサーバーがこれらのトランザクション境界を認識すると、EJB モジュールが実行するすべての作業がコミットまたはロールバックされます。

トランザクション境界を設定するには、「トランザクション設定」プロパティエディタを開き、トランザクションに関係するエンタープライズ Bean の属性を変更します。

トランザクション属性の設定を編集する手順を以下に示します。以下の手順では、図 6-1 に示されているデフォルトのトランザクション設定を、図 6-2 に示されているトランザクション境界を生成するトランザクション属性設定に変更します。

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、「プロパティ」を選択します。「プロパティ」セクションの「トランザクション設定」の省略符号ボタン (...) をクリックします。

   「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. セッション Bean メソッドのトランザクション属性を RequiresNew に変更します。

   いずれかのビジネスメソッド (getAllItems と getOneItemDetail) を呼び出すと新しいトランザクションが開始するように、CatalogManagerBean メソッドの動作を変更するとします。そのためには、トランザクション属性を RequiresNew に変更します。

   a. セッション Bean のメソッドをクリックして選択します。

   b. 「トランザクション属性」の値を RequiresNew に変更します。

      新しいトランザクション属性値がメソッドノードの後ろに表示されます。

3. エンティティ Bean メソッドのトランザクション属性を Mandatory に変更します。

   セッション Bean メソッドが開いたトランザクション境界内でエンティティ Bean メソッドが実行されるように、エンティティ Bean メソッドの動作を変更するとします。そのためには、トランザクション属性を Mandatory に変更します。これによって、トランザクションがすでに進行中の場合のみこれらのメソッドを実行できることを、アプリケーションサーバーに伝えます。

   a. エンティティ Bean のメソッドをクリックして選択します。

   b. 「トランザクション属性」の値を Mandatory に変更します。

      新しいトランザクション属性値がメソッドノードの後ろに表示されます。

4. 「了解」をクリックしてエディタを閉じ、変更を保存します。

   図 6-3 に、新しいトランザクション属性が設定された「トランザクション設定」プロパティエディタを示します。

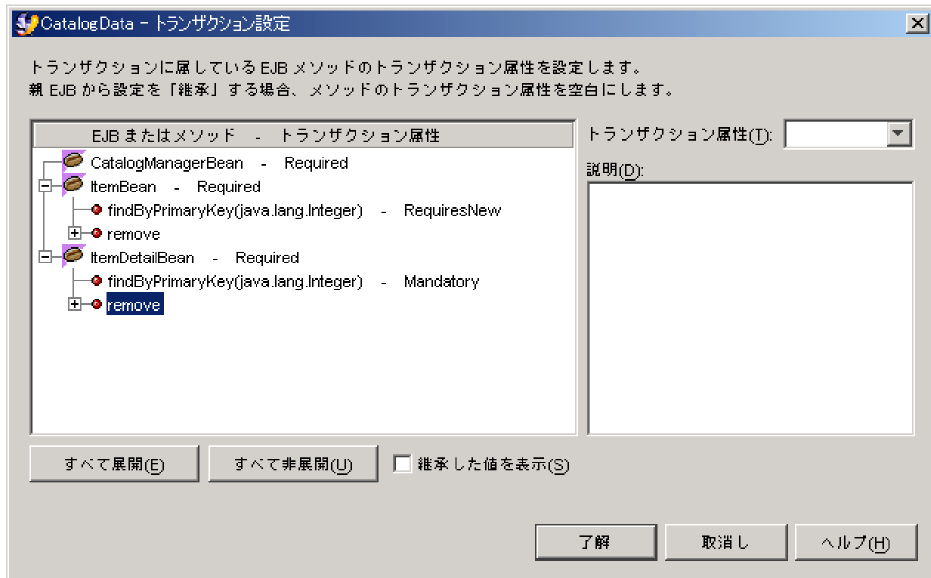


図 6-3　　変更後のトランザクション設定

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。プロパティエディタに表示されている変更点について以下に説明します。

- 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トランザクション属性が Mandatory に設定されています。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 モジュールごとにモジュール内の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トランザクション境界を判断する必要があります。その後、「トランザクション属性」プロパティエディタを使って、これらのトランザクション境界を実装します。こ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 や個々のメソッドのトランザクション属性を設定することで、トランザクション境界を指定します。

第7章

セキュリティ

アプリケーションのセキュリティ保護とは、その機能へのアクセスを制限することです。アプリケーション機能へのアクセスを制限することで、そのアプリケーションが管理するデータへのアクセスを制限します。

Java 2 Platform, Enterprise Edition (J2EE プラットフォーム) アプリケーションは、アプリケーションリソースおよびユーザーロールの処理によってセキュリティ保護されます。この 2 つの概念を以下に定義します。

- リソースは、表示や呼び出しが可能なアプリケーションの機能です。Enterprise JavaBeans (EJB) モジュールのリソースは、ホームインタフェースかリモートインタフェースで宣言された EJB の public メソッドです。Web モジュールのリソースは、JavaServer Pages (JSP) ページやサーブレットメソッドなどのコンポーネントにマップされた URL パターンです。
- ロールとは、ユーザーに関連付けられている名前です。

J2EE アプリケーションは、リソースをロールにマッピングすることでセキュリティ保護されます。呼び出し側はリソースを呼び出すときに、リソースへのアクセスが許可されたロール名を指定する必要があります。許可されたロールを呼び出し側が指定できないと、呼び出しは却下されます。J2EE アプリケーションでは、アプリケーションサーバーが呼び出し側のロールを確認してから、リソースの実行を呼び出し側に許可します。

許可されたロールとリソースの組み合わせは、配備記述子で宣言されます。アプリケーションサーバーは、配備記述子からこれらを読み取って、適用します。この処理は宣言型セキュリティと呼ばれます。

J2EE アプリケーションのセキュリティを保護するためには、以下の作業を実行します。

- ロールを宣言します。
- リソースへのアクセスを許可するロールを指定します。

たとえば、人材データを処理する Web モジュールを開発していると仮定しましょう。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 リソースは、モジュールの仕様で指定されています。この場合は、これらの各種ユーザーを表すセキュリティロールを宣言し、Web モジュール内のリソースをこれらのロールにマップします。

EJB モジュールのセキュリティ保護も同様に行います。モジュールの仕様で、各種ユーザーが指定されているほか、どの種類のユーザーにエンタープライズ Bean メソッドが返したデータへのアクセスを許可するかが指定されています。この場合は、これらのユーザーグループを表すセキュリティロールを宣言し、モジュール内の EJB メソッドを適切なロールにマップします。

通常、J2EE セキュリティはモジュールのプロパティシートで設定します。ここで、そのモジュールの一連のロールを宣言します。そしてモジュールのリソースを、モジュールで宣言した一連のロールにマップします。

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、これを配備するときに、モジュールで宣言したロールをアプリケーションサーバー環境の実際のユーザー名やグループ名にマップします。ユーザーやグループへのロールのマップは、J2EE アプリケーションのプロパティシートで行います。このマッピングは、ユーザーやグループが宣言されている、本稼動の環境にアプリケーションを配備するときに行います。

この後、統合開発環境 (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、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一連のセキュリティ宣言をマージする方法を説明します。

Web モジュールのセキュリティ

Web モジュールでは、いくつかのプロパティエディタやダイアログを使ってセキュリティロールを宣言し、セキュリティロールを Web リソースにマッピングします。この節では、これらのプロパティエディタやダイアログを使って、以下の 2 つの作業を行います。

- Web モジュールのセキュリティロールを宣言する。
- セキュリティ保護したい Web リソースを定義し、そのリソースをセキュリティロールにマッピングする。

それぞれの作業は別々の項で説明します。

Web モジュールのセキュリティロールの宣言

Web モジュールのセキュリティロールを宣言するには、次のようにします。

1. Web モジュールの web ノードを右クリックし、「プロパティ」を選択します。「セキュリティ」セクションの「セキュリティロール」の省略符号ボタン (...) をクリックします。

   「セキュリティロール」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。

   「追加 セキュリティロール」ダイアログが開きます。

3. 新しいセキュリティロールを定義します。

   a. 「ロール名」フィールドに、新しいセキュリティロールの名前を入力します。

   b. 「ロールの説明」フィールドに、そのロールの簡単な説明を入力します。

      アプリケーションにアセンブルされる複数モジュールのセキュリティロールをマージするとき、説明が役立ちます。また、アプリケーションサーバー環境のユーザーやグループにセキュリティロールをマップするときにも役立ちます。

   c. 「了解」をクリックして、ダイアログを閉じます。

      図 7-1 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。

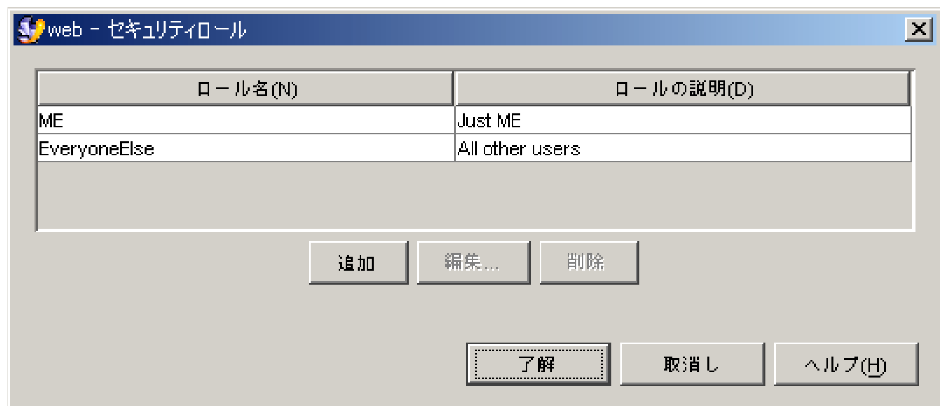


図 7-1　Web モジュールに宣言されたセキュリティロール ME と EveryoneElse

4. 「了解」をクリックして、プロパティエディタを閉じます。

Web リソースの定義とセキュリティロールへのリソースのマッピング

Web リソースを定義し、保護するには、次のようにします。

1. **モジュールの web ノードを右クリックし、「プロパティ」を選択します。「セキュリティ」セクションの「セキュリティ制限」の省略符号ボタン (...) をクリックします。**

   「セキュリティ制限」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。**

   「追加 セキュリティ制限」ダイアログが開きます。

3. **「追加 セキュリティ制限」ダイアログの「追加」ボタンをクリックします。**

   「追加 Web リソースコレクション」ダイアログが開きます。

4. **Web リソースを定義します。**

   a. **「リソース名」フィールドに、リソースの名前を入力します。**

   b. **「URL パターン」フィールドに URL パターンを入力します。**

      URL パターンは、すでにモジュールで定義されている必要があります。図 7-2 のフィールド値では、Web リソースとして URL パターン `/allItems` が定義されています。この URL パターンは、サーブレット `AllItemsServlet` にすでにマップされています。図 7-2 の値によって、`AllItemsServlet` を実行するための Web リソースが設定されます。この URL パターン `/allItems` のサーブレット `AllItemsServlet` へのマップ方法は、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。

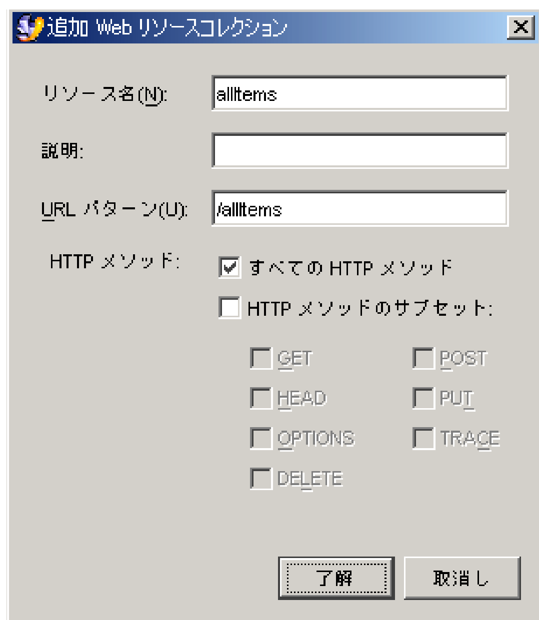


図 7-2　　allItems という名前の Web リソースの定義

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 メソッドに適用するように定義できます。

c. 「了解」をクリックしてこのダイアログを閉じ、「追加 セキュリティ制限」ウィンドウに戻ります。

図 7-3 は、allItems と itemDetail という名前の 2 つの Web リソースが設定された「追加 セキュリティ制限」ウィンドウを示しています。

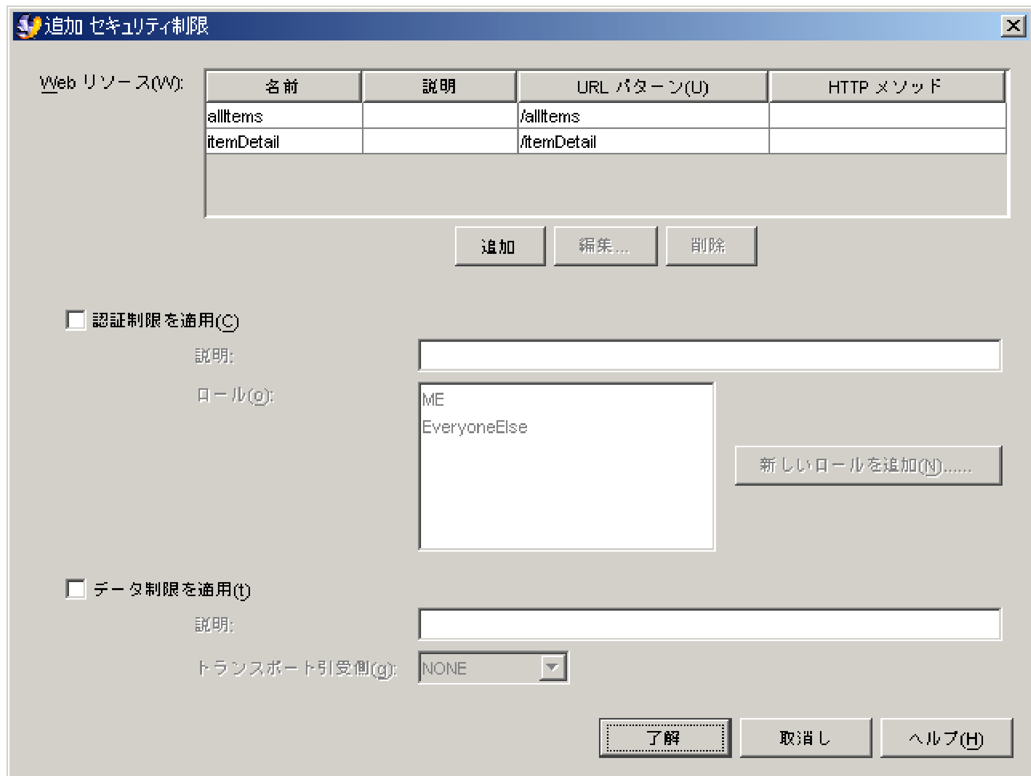


図 7-3 「追加 セキュリティ制限」ダイアログの allItems リソース

5. リソースへのアクセス制限を指定して、リソースのセキュリティを保護します。

   a. リソースをクリックして選択します。

   b. 「追加 セキュリティ制限」ダイアログのフィールドを使って、リソースへのアクセス制限を記述します。

   c. 制限としてセキュリティロールを使用するには、「認証制限を適用」チェックボックスをクリックしてから、「ロール」フィールドで 1 つ以上のロールを選択します。

   図 7-4 では、allItems リソースが選択されています。EveryoneElse ロールも選択されています。これらの選択は、allItems リソースは EveryoneElse ロールによって呼び出される必要があることを指定しています。ほかのロールの呼び出し側は拒否されます。

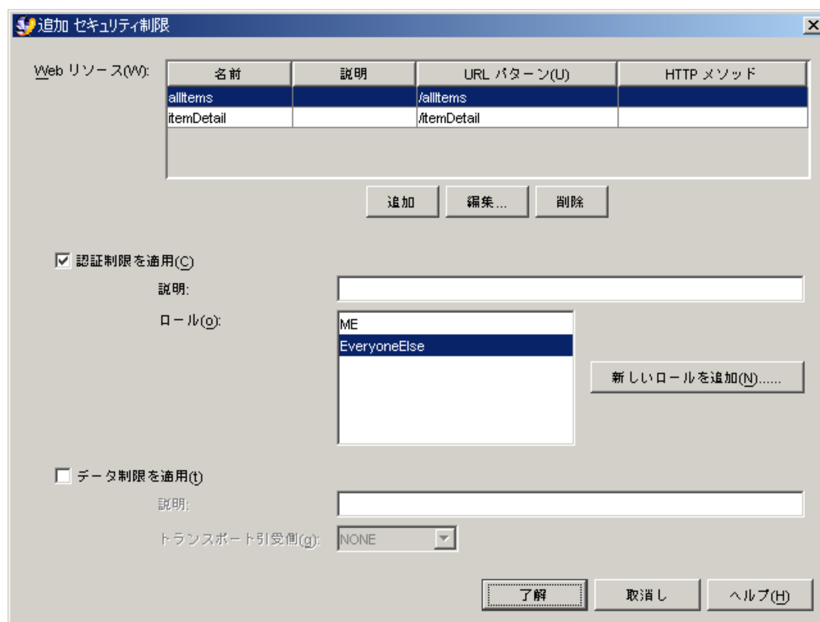


図 7-4　　`allItems` という名前の Web リソースに対する制限の指定

d. 「了解」をクリックして指定した制限を保存し、「セキュリティ制限」プロパティエディタに戻ります。

6. 「了解」をクリックして作業内容を保存し、プロパティシートに戻ります。

Web モジュールの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、メソッドコードで使用されるセキュリティロール参照を、モジュールのプロパティシート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、呼び出し側の資格に直接アクセスするコードが含まれており、アプリケーションサーバー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。コード例 7-1 は、メソッドコードの数行を示しています。このコードでは、セキュリティロール参照名 `roleRefMe` を使用しています。

コード例 7-1　　`roleRefMe` セキュリティロール参照を使用するメソッドコード

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

roleRefMe のようなセキュリティロール参照は、実際の参照名のプレースホルダです。メソッドコードはモジュールレベルでロールが宣言される前に作成されるため、実際のロール名は不明です。モジュールをアセンブルすると、セキュリティロール参照が宣言されて、宣言されたセキュリティロールにマップされます。

セキュリティロール参照を宣言して、セキュリティロールにマップするには、次のようにします。

1. **モジュールの web ノードを右クリックし、「プロパティ」を選択します。「配備」セクションの「サーブレット」の省略符号ボタン (...) をクリックします。**

   「サーブレット」プロパティエディタが開きます。

2. **セキュリティロール参照を含むサーブレットを選択し、「編集」ボタンをクリックします。**

   「編集 サーブレット」ダイアログが開きます。

3. **「セキュリティロール参照」フィールドの下にある「追加」ボタンをクリックします。**

   「追加 セキュリティロール参照」ダイアログが開きます。

4. **セキュリティロール参照を宣言して、セキュリティロール参照をロールにマップします。**

   a. **「ロール参照名」フィールドに、メソッドコードで使用されているロール参照名を入力します。**

   b. **「ロール参照リンク」フィールドに、参照名をリンクする既存のセキュリティロールの名前を入力します。**

   c. **「了解」をクリックしてダイアログを閉じ、「編集 サーブレット」ダイアログに戻ります。**

      図 7-5 は「編集 サーブレット」ダイアログを示しています。roleRefMe という名前のセキュリティロール参照が、ロール ME にマップされています。

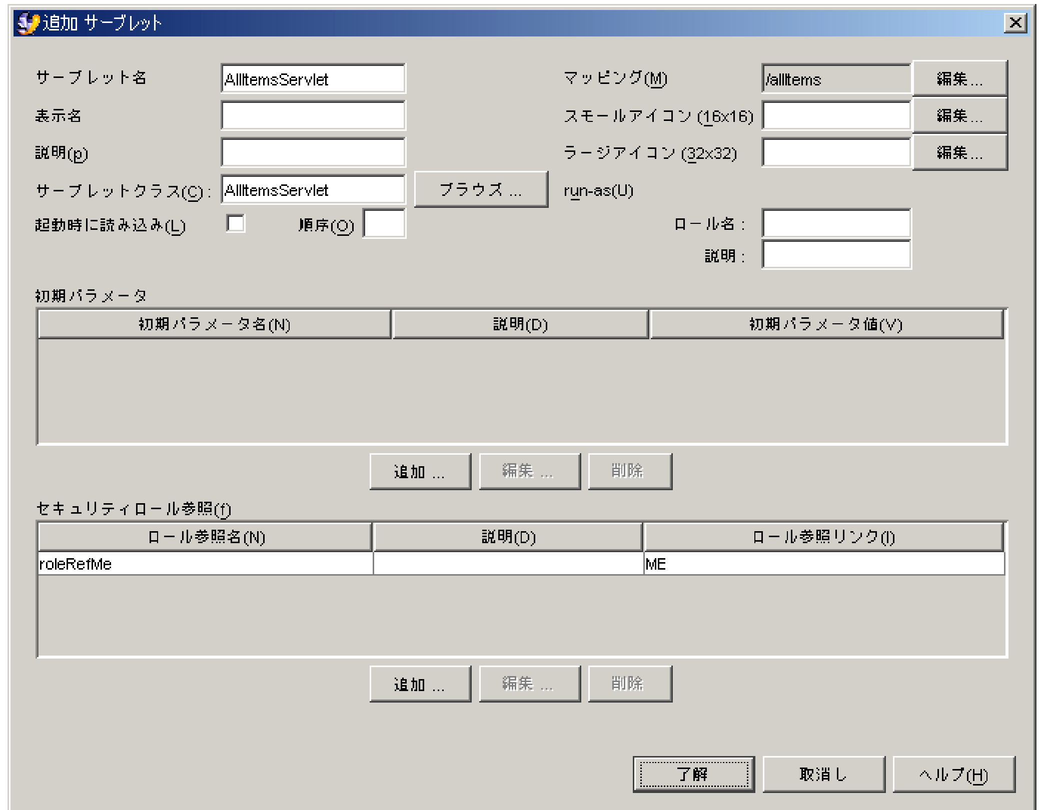


図 7-5　　ロール ME にマップされたセキュリティロール参照 roleRefMe

メソッドコードを実行すると、ロール参照がロール ME にマップされ、ME というロールについて呼び出し側の資格がテストされます。このようなマッピングを行う前に、モジュールのプロパティシートでセキュリティ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

EJB モジュールでは、いくつかのプロパティエディタやダイアログを使ってセキュリティロールを宣言し、セキュリティロールをエンタープライズ Bean メソッドにマッピングします。この節では、これらのプロパティエディタやダイアログを使って、以下の 2 つの作業を行います。

- EJB モジュールのセキュリティロールを宣言する。
- エンタープライズ Bean メソッドをセキュリティロールにマッピングする。

それぞれの作業は別々の項で説明します。

EJB モジュールのセキュリティロールの宣言

EJB モジュールのセキュリティロールを宣言するには、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、「プロパティ」を選択します。「プロパティ」セクションの「セキュリティロール」の省略符号ボタン (...) をクリックします。**

   「セキュリティロール」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。**

   「セキュリティロールを追加」ダイアログが開きます。

3. **ロールを定義する値を入力します。**

   a. **「名前」フィールドに、ロールの名前を入力します。**

   b. **「説明」フィールドに、そのロールの説明を入力します。**

   アプリケーションにアセンブルされる複数モジュールのセキュリティロールをマージするとき、説明が役立ちます。また、配備環境にあるユーザーやグループにセキュリティロールをマップするときにも役立ちます。

   c. **「了解」をクリックします。**

   「追加」ダイアログが閉じて、「セキュリティロール」プロパティエディタに戻ります。図 7-6 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。

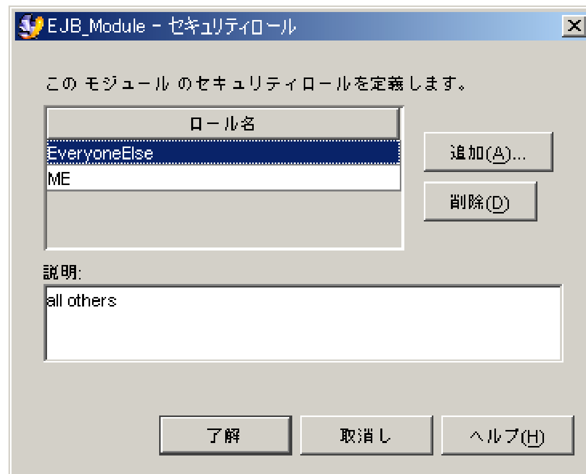


図 7-6　　EJB モジュールの「セキュリティロール」プロパティエディタ

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、ロールを使ってモジュールに含まれるエンタープライズ Bean メソッドに対するアクセスを制限します。

注 – セキュリティロールをマップするときは、「含まれている EJB ノード」を使用します。これらは、モジュール内のエンタープライズ Bean を表す EJB モジュールのサブノードです。

メソッドアクセス権にセキュリティロールをマップするには、次のようにします。

- **含まれている EJB ノードを右クリックし、「プロパティ」を選択します。「プロパティ」セクションの「メソッドのアクセス権」の省略符号ボタン (...) をクリックします。**

  「メソッドのアクセス権」プロパティエディタが開きます。

  「メソッドのアクセス権」プロパティエディタは表形式で、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ています。図 7-7 に `CatalogManagerBean` のプロパティエディタを示します。`CatalogData` EJB モジュールで宣言されている 2 つのセキュリティロール、`EveryoneElse` と `ME` が、列に示されています。

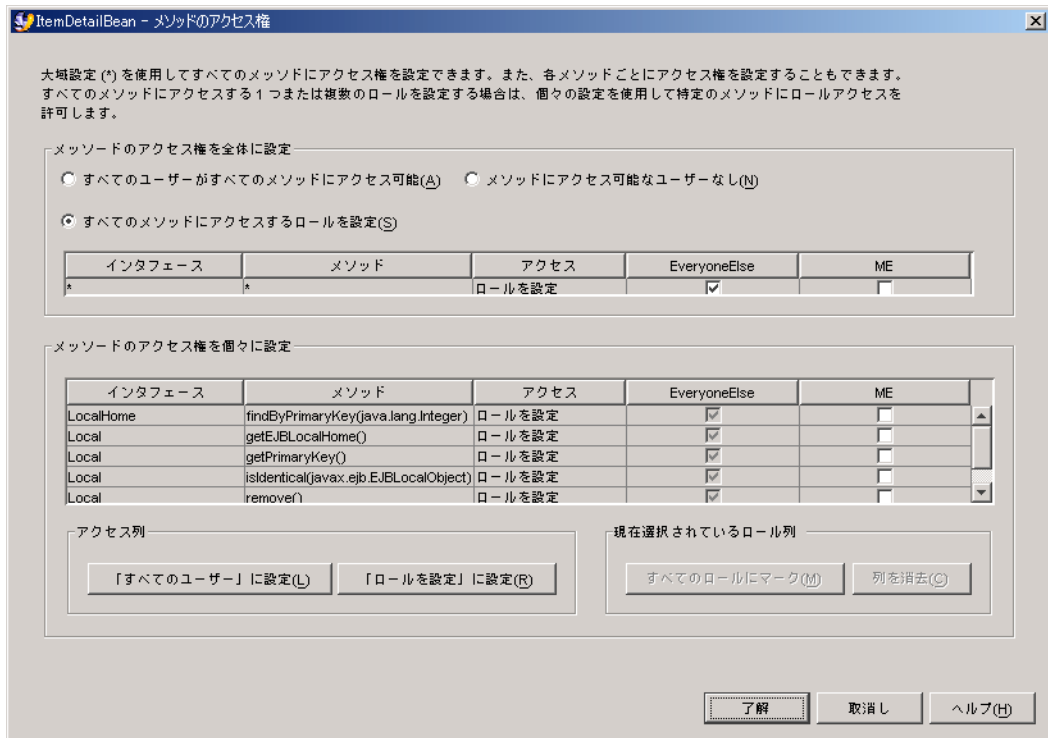


図 7-7　　EJB の「メソッドのアクセス権」プロパティエディタ

「メソッドのアクセス権」プロパティエディタでは、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、アクセス権を全体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。このオプショ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。列のチェックボックス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 7-7 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付いていません。この結果、`ME` 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定義できます。あるメソッドのアクセス権の設定は、エディタの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 メソッドの「アクセス」フィールドを「すべてのユーザー」に設定します。この設定によって、どのロールも getAllItems メソッドを実行できるようになります。その後、別のメソッドをクリックし、「アクセス」フィールドを「ロールを設定」に設定し、選択したメソッドのロールを個別に選択することができます。

EJB モジュールのプログラム可能なセキュリティ

モジュール内のエンタープライズ Bean 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、メソッドコードで使用されるセキュリティ参照を、EJB モジュールのプロパティシート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、呼び出し側の資格に直接アクセスするメソッドコードが含まれており、アプリケーションサーバー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。コード例 7-2 は、メソッドコードの数行を示しています。このコードは、セキュリティロール参照名 everyOne を使用しています。

コード例 7-2 everyOne セキュリティロール参照を使用するメソッドコード

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

セキュリティロール参照は、実際の参照名のプレースホルダです。メソッドコードはモジュールレベルでロールが宣言される前に作成されるため、実際のロール名は不明です。モジュールをアセンブルすると、セキュリティロール参照が宣言されて、宣言されたセキュリティロールにマップされます。

セキュリティロール参照を宣言して、セキュリティロールにマップするには、次のようにします。

1. **EJB 論理ノードを右クリックし、「プロパティ」を選択します。「参照」セクションの「セキュリティロール参照」の省略符号ボタン (...) をクリックします。**

   「セキュリティロール参照」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。**

   「追加 セキュリティロール参照」ダイアログが開きます。

3. セキュリティロール参照を宣言して、セキュリティロールにリンクします。

   a. 「名前」フィールドに、メソッドコードで使用されているセキュリティロール参照名を入力します。

   b. 「セキュリティロールリンク」フィールドに、セキュリティロールの名前を入力します。フィールドを空白のままにして、宣言した参照をリンクしないでおくこともできます。

   c. 「了解」をクリックします。

   「追加 セキュリティロール参照」ダイアログが閉じて、「セキュリティロール参照」プロパティエディタに戻ります。図 7-8 の「セキュリティロール参照」プロパティエディタには、コード例 7-2 で使用されている everyOne という名前のセキュリティロール参照が表示されています。このロールはリンクされていません。

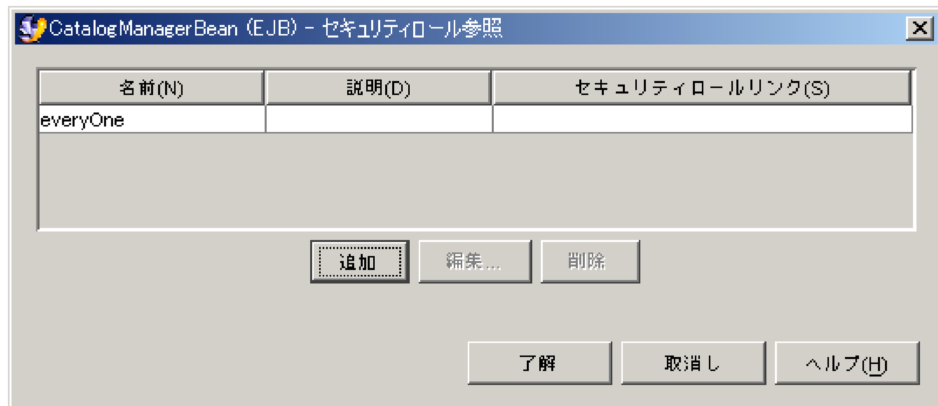


図 7-8　セキュリティロール参照 everyOne の宣言

このシナリオでは、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で everyOne セキュリティロール参照をリンクします。このマッピングは、エンタープライズ Bean をアセンブルして EJB モジュールを作成するときに実行します。

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でセキュリティロール参照をリンクするには、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、「プロパティ」を選択します。「プロパティ」セクションの「セキュリティロール参照」の省略符号ボタン (...) をクリックします。

   「セキュリティロール参照」プロパティエディタが開きます。

2. モジュール内の参照を検討します。

   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は、モジュール内のすべてのセキュリティロール参照を表示し、これらがリンクされているかどうかを示します。図 7-9 では、everyOne 参照はリンクされていません。

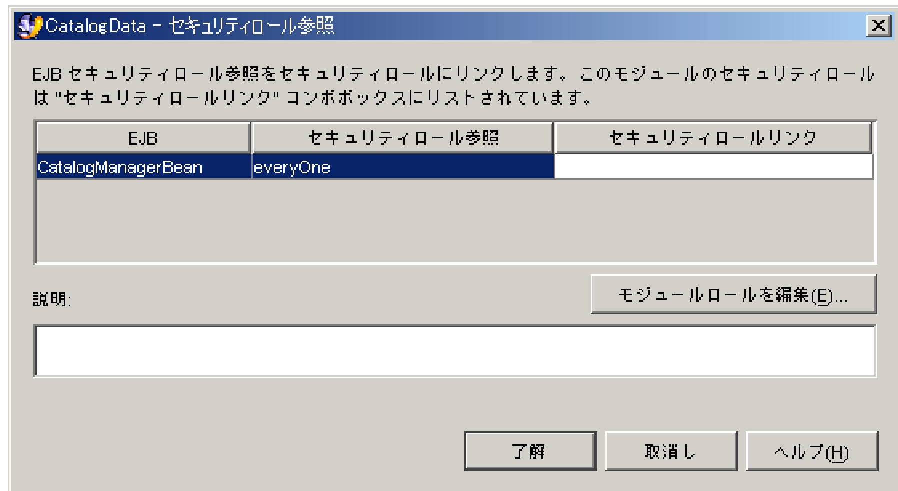


図 7-9　　EJB モジュールの everyOne セキュリティロール参照

3. リンクされていないロールをリンクします。

   a. 参照をクリックして選択します。

   b. 「セキュリティロールリンク」フィールドで、適切なセキュリティロールを選択します。

   c. 「了解」をクリックします。

図 7-10 は、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を示しています。ここでは、everyOne という名前のセキュリティロール参照のリンクが、EveryoneElse という名前のセキュリティロールにマップされています。

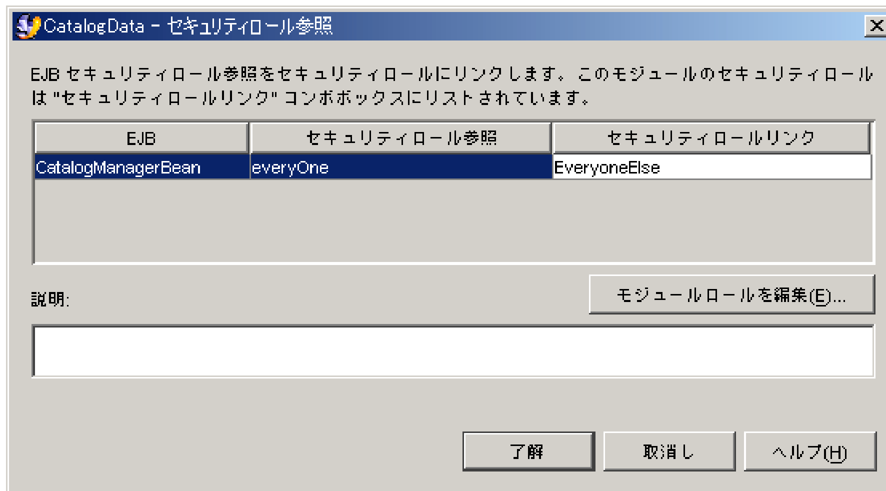


図 7-10　EJB モジュールの「セキュリティロール参照」プロパティエディタ

J2EE アプリケーションのセキュリティ

J2EE アプリケーションをアセンブルするときは、それがセキュリティ保護されているかどうかを判断します。このためには、以下の点を調べてください。

- 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セキュリティロールを定義する必要があります。96 ページの「Web モジュールのセキュリティ」と 104 ページの「EJB モジュールのセキュリティ」を参照してください。
- モジュールがセキュリティ保護されている場合は、それぞれのモジュールに名前の異なる類似のロールが含まれていることがあります。その場合は、J2EE アプリケーションのプロパティシートで、同じ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。

J2EE アプリケーションレベルでセキュリティロールをマップするには、次のようにします。

● **アプリケーションノードを右クリックし、「プロパティ」を選択します。「プロパティ」セクションの「セキュリティロール」の省略符号ボタン (...) をクリックします。**

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が開きます。図 7-11 に表示された「セキュリティロール」プロパティエディタには、モジュールで定義されたセキュリティロールが示されています。

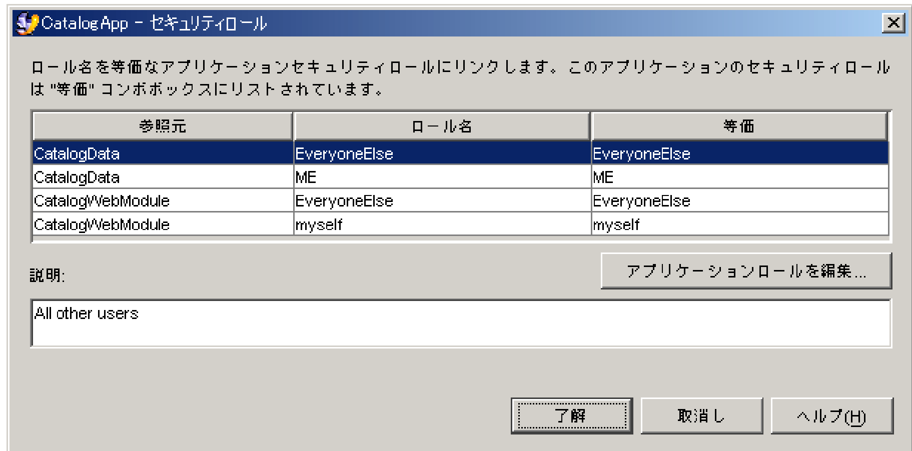


図 7-11 J2EE アプリケーションの「セキュリティロール」プロパティエディタに表示されたセキュリティロール

最初の 2 列には、モジュールで宣言されたセキュリティロールが表示されています。各ロールは、そのモジュールとロール名によって識別されます。モジュールレベルの各セキュリティロールは、デフォルトのアプリケーションレベルのロールにマップされます。アプリケーションレベルのロールは、モジュールレベルのロールと同じ名前を持ちます。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。

たとえば、プロパティエディタに表示されている 1 番目のセキュリティロールは、`CatalogData` モジュールの `EveryoneElse` です。このロールは、`EveryoneElse` という名前のアプリケーションレベルのロールにマップされています。

図 7-11 には、セキュリティロールマッピングの違いが示されています。`CatalogWebModule` には `myself` というロールがあり、`CatalogData` モジュールには `ME` というロールがあります。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 7-12 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。

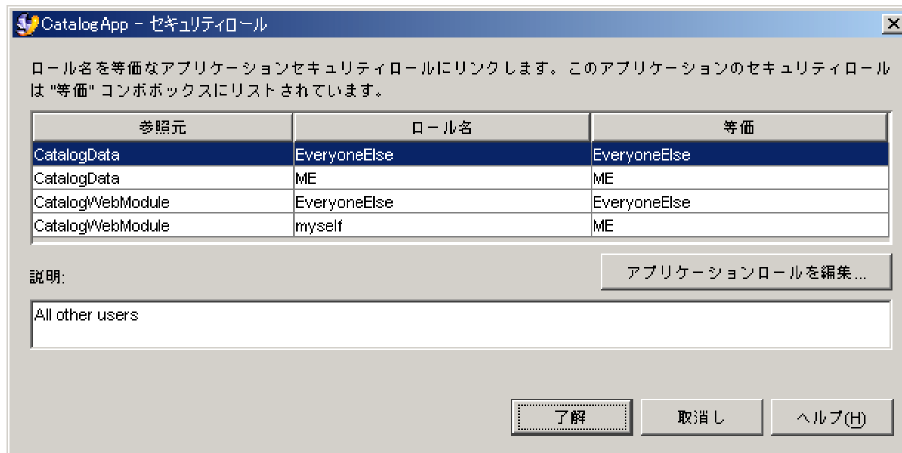


図 7-12　ロール ME にマップされたロール myself

モジュールレベルのロールの ME および myself の両方が同じ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。これを行うには、「アプリケーションロールを編集」をクリックして、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宣言します。

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。sysadmin ロールを選択します。

第8章

J2EE モジュールと J2EE アプリケーションの配備と実行

統合開発環境 (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反復開発できます。アプリケーションの開発とアセンブル、配備、および実行を、すべて IDE 内で行うことができます。

アプリケーションの実行後、ソースコードやプロパティを変更し、アプリケーションを再配備して、再度これを実行できます。テストでアセンブルの問題が検出されなかった場合は、再配備する前にアセンブルし直す必要はありません。

この章では、アセンブルされたアプリケーションを IDE 内で配備および実行する方法の基礎を説明します。

「エクスプローラ」ウィンドウでのサーバーの表示

アプリケーションをアプリケーションサーバーに配備するには、アプリケーションサーバーと対話する必要があります。アプリケーションサーバーとの対話を簡略化するために、IDE はアプリケーションサーバーをノードとして「実行時」ウィンドウに表示します。

ウィンドウのほかのノードと同様、これらのアプリケーションサーバーノードにもプロパティシートとメニューコマンドがあります。こうしたプロパティシートやメニューコマンドを使って、アプリケーションを IDE 内から配備し、実行します。使用するアプリケーションサーバー製品によっては、アプリケーションサーバーを管理することもできます。

この節では、サーバーノードの識別を行い、これらについて説明します。また、これらのノードで実行できる基本的な作業についても説明します。

サーバーレジストリノード

図 8-1 は、サーバーの構成、配備、および実行に使用するノードが表示された「実行時」ウィンドウを示しています。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。

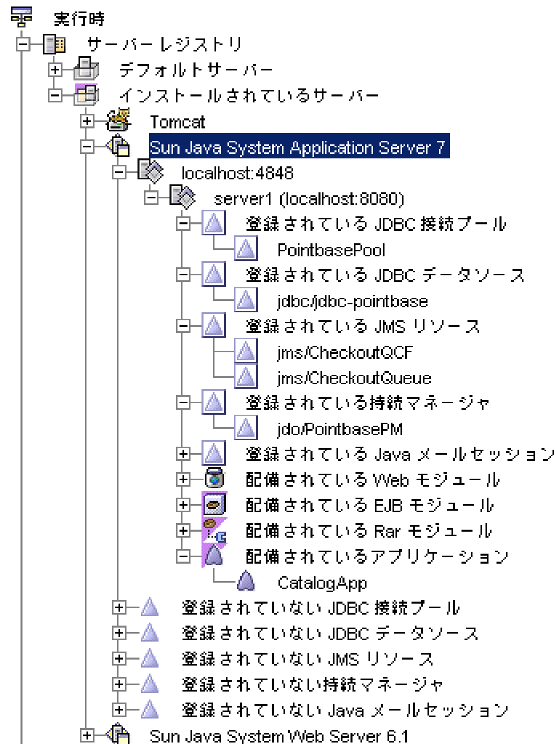


図 8-1 サーバーレジストリノード

図 8-1 のサーバーレジストリは、Microsoft Windows プラットフォームにインストールされた IDE のものです。また、このインストールは、管理者権限 (スーパーユーザー) を持つユーザーがスタンドアロンでインストールしたものです。システム管理者が管理者権限を持たないユーザーのために IDE をインストールした場合は、ノードによって表示されるホスト名とポート番号がこれとは異なります。また、マルチユーザーインストールの場合も、ホスト名とポート番号は異なります。

インストールオプションの詳細については、『Sun Java Studio Enterprise 7 インストールガイド』を参照してください。

「インストールされているサーバー」ノード

「インストールされているサーバー」ノードは、サーバー製品ノードをグループ化します。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 8-1 では、「インストールされているサーバー」ノードに、IDE のほとんどのインストールに含まれている 2 つのサーバー製品、Sun Java System Application Server と Tomcat Web サーバーのサブノードがあります。

サーバー製品ノード

「インストールされているサーバー」ノードの下には、特定の Web サーバー製品やアプリケーションサーバー製品のノードが表示されます。これらのサーバー製品ノードのそれぞれは、インストールされている IDE サーバープラグインを表します。

サーバー製品ノードには、コンテキストメニューとプロパティシートがあります。各サーバー製品ノードの機能は、サーバー製品とプラグインモジュールによって決まります。

手順はサーバー製品によって異なりますが、通常、アプリケーションサーバーを使用するには、サーバー製品の固有のインストール場所を認識するように使用するサーバー製品ノードを設定します。また、サーバー製品ノードを使ってサーバーインスタンスも作成できます。

図 8-1 では、「インストールされているサーバー」ノードに、IDE のほとんどのインストールに含まれている 2 つのプラグイン、Sun Java System Application Server Standard Edition 7 プラグインと Tomcat Web サーバープラグインのサブノードがあります。次の節では、Sun Java System Application Server Standard Edition 7 ノードとそのサブノードについて説明します。

IDE でほかのアプリケーションサーバー製品を設定する方法については、『Sun Java System Application Server 7 入門ガイド』を参照してください。

Sun Java System Application Server ノード

この節では、「実行時」ウィンドウの Sun Java System Application Server を表すノードの識別を行い、これらについて説明します。

Sun Java System Application Server 7 ノード

Sun Java System Application 7 サーバーノードは、アプリケーションサーバーを管理するためのノードです。

IDE をインストールすると、アプリケーションサーバードメインが作成されます。また、アプリケーションサーバードメインを管理する管理サーバーインスタンスも作成されます。ほとんどの場合は、インストール時に作成されたアプリケーションサーバードメインと管理サーバーですべての作業を行うことができます。

- IDE のインストール時に作成されるアプリケーションサーバードメインで、アプリケーションを配備し、実行できます。
- IDE のインストール時に作成される管理サーバーインスタンスを使って、このドメインを管理できます。

図 8-1 に示されているサーバーレジストリには、IDE のインストール時に作成されたアプリケーションサーバードメインと管理サーバーが含まれています。管理サーバーインスタンスは、Sun Java System Application Server 7 ノードのすぐ下の、「localhost:4848」というノードによって示されています。管理サーバーノードのプロパティシートを開くと、アプリケーションサーバードメイン名を表示できます。

スーパーユーザーまたは管理者権限を持っているユーザーは、必要であれば Sun Java System Application Server 7 ノードを使ってアプリケーションサーバードメインと管理サーバーを追加できます。

管理サーバーノード

「Sun Java System Application Server 7」ノードの下には、管理サーバーノードがあります。図 8-1 では、管理サーバーノードは「localhost:4848」と示されています。管理サーバーノードは、Sun Java System Application Server の管理サーバーのインスタンスを表します。それぞれの管理サーバーインスタンスは、アプリケーションサーバードメインを管理します。

アプリケーションサーバードメインと管理サーバーインスタンスは、Sun Java System Application Server が組み込まれた IDE をインストールする時に作成され、また、Sun Java System Application Server を管理する時にも作成されます。作業しているインストール環境によって、アプリケーションサーバードメインとサーバーインスタンスがどう作成されるかが決まります。以下にいくつかのケースを示します。

- スーパーユーザーまたは管理者権限を持つユーザーが IDE をインストールすると、初期サーバードメインと管理サーバーインスタンスが作成されます。図 8-1 には、インストール処理で作成されたサーバードメインと管理サーバーインスタンスが示されています。
- 一般ユーザーの権限を持つユーザーがインストールした場合は、システム管理者がサーバードメインと管理サーバーインスタンスを作成します。ユーザーは、管理サーバーノードを使って各自のドメインを管理できます。

管理サーバーノードに表示されているホスト名は、アプリケーションサーバーが動作しているマシンの名前です。図 8-1 は、スタンドアロンのシングルユーザーインストールを示しており、アプリケーションサーバードメインはローカルホスト上で動作しています。マルチユーザー環境の場合は、アプリケーションサーバーが別のマシンで稼働していてもかまいません。

管理サーバーに表示されるポート番号は、管理サーバーと通信するためのポート番号です。ポート番号は、アプリケーションサーバードメインと管理サーバーインスタンスを作成したときに設定されます。図 8-1 には、Microsoft Windows プラットフォーム上のシングルユーザーインストールの場合のデフォルトポート番号が示されています。

管理サーバーノードを使って、管理サーバーが制御するサーバードメインのサーバーインスタンスの起動や停止を実行します。

サーバーインスタンスノード

管理サーバーノードの下には、アプリケーションサーバーインスタンスノードがあります。図 8-1 では、アプリケーションサーバーインスタンスノードは「server1(localhost:8080)」と示されています。アプリケーションサーバーインスタンスノードは、サーバーインスタンスを表します。

モジュールやアプリケーションを配備するときは、特定のサーバーインスタンスに配備します。配備して実行するときは、その前にサーバーインスタンスノードを作成し、それが表すインスタンスを実行する必要があります。

図 8-1 は、IDE のインストールによって作成されたサーバーインスタンスを示しています。サーバーインスタンスは、アプリケーションサーバードメインを管理することによって作成することもできます。

管理サーバーとサーバーインスタンスの起動

管理サーバーとサーバーインスタンスを起動するには、次のようにします。

1. **必要に応じて、アプリケーションサーバーのホームディレクトリを設定します。アプリケーションサーバーを右クリックし、「プロパティ」を選択します。「Sun Java System Application Server Home」の省略符号ボタン (...) をクリックして、ホームディレクトリを選択します。**

2. **管理サーバーノードを右クリックし、「起動」を選択します。**

   進捗モニターウィンドウが開きます。管理サーバーが起動すると、進捗モニターは閉じます。サーバーインスタンスノードが管理サーバーノードの下に表示されます。

3. **サーバーインスタンスノードを右クリックし、「状態」を選択します。**

   「Sun Java System Application Server インスタンスの状態」ダイアログが開きます。「状態」フィールドにサーバーインスタンスのステータスが表示されます。これは「停止しました」になっています。

4. **「サーバーを起動」をクリックします。**

   「アプリケーションサーバーインスタンスを起動しています」というメッセージがダイアログに表示されます。サーバーインスタンスが起動すると、「ステータス」フィールドに「実行中」と表示されます。

5. **「閉じる」をクリックします。**

   サーバーインスタンスの使用準備が整います。

   コンテキストメニューには、管理サーバーノードで実行できるほかの作業が一覧表示されます。

登録リソースノード

サーバーインスタンスノードの下には、サーバーインスタンスで動作するアプリケーションが利用できる名前付きリソースを表すノード群があります。図 8-1 には、このマニュアルのシナリオで作成され使用されているリソースのノード群が表示されています。

- 「登録されている JDBC 接続プール」、「登録されている JDBC データソース」、および「登録されている持続マネージャ」というノードの下には、`sample` という名前のインストールされている PointBase データベースを表すノードがあります。これらのリソースは、IDE のインストール時に事前構成されています。あるシナリオでは、リソース名を入力することで PointBase `sample` データベースを使用するように `CatalogData` Enterprise JavaBeans (EJB) モジュールを構成しました。`CatalogData` EJB モジュールの構成方法については、52 ページの「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」を参照してください。
- 「登録されている JMS リソース」というノードの下には、このマニュアルのシナリオで作成されたキューおよびキュー接続ファクトリリソースのノードがあります。これらのリソースの作成・登録手順については、75 ページの「アプリケーションサーバーの設定」を参照してください。これらのリソースを使ってアプリケーションを構成する手順については、78 ページの「Web モジュールのプログラミング」と 84 ページの「EJB モジュールのプログラミング」を参照してください。

配備されているアプリケーションノード

登録リソースノードの下には、サーバーインスタンスに配備されたモジュールやアプリケーションを表すノード群があります。図 8-1 には、`CatalogApp` という名前のアプリケーションのノードが表示されています。`CatalogApp` のプログラミングと配備は、このマニュアルのシナリオで説明されています。`CatalogApp` の配備手順については、63 ページの「J2EE アプリケーションの作成」を参照してください。

登録されていないリソースノード

「配備されているアプリケーション」ノードの下には、登録されていないサーバーリソースを表すノード群があります。これらのノードは管理サーバーノードのサブノードで、サーバーインスタンスにまだ登録されていないリソースを表します。

これらのノードは、新しいリソースを作成して登録するためのメニューコマンドを持っています。このマニュアルのシナリオでは、「登録されていない JMS リソース」ノードを使ってキューとキュー接続ファクトリを作成しています。リソースの作成手順については、75 ページの「アプリケーションサーバーの設定」を参照してください。

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示します。アプリケーションを配備すると、アプリケーションのプロパティシートでほかの配備先を指定しない限り、アプリケーションはデフォルトサーバーインスタンスに配備されます。

図 8-1 では、デフォルトノードによって、Java 2 Platform, Enterprise Edition (J2EE プラットフォーム) アプリケーションのデフォルトサーバーが Sun Java System Application Server であることが示されています。

サーバーインスタンスをデフォルトサーバーにするには、次のいずれかの手順を実行します。

- **サーバーインスタンスノードを右クリックし、「デフォルトとして設定」を選択します。**
- **「デフォルトサーバー」の下の「J2EE アプリケーション」ノードを右クリックし、「デフォルトサーバーを設定」を選択します。「デフォルトアプリケーションサーバーを選択」ダイアログが開きます。デフォルトアプリケーションサーバーとして使用するサーバーインスタンスを選択します。**

サーバー固有のプロパティ

使用する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ます。このプロパティシートを使用して、モジュールやアプリケーションが必要とするアプリケーションサーバーのサービスを指定します。

多くのプロパティシートには、サーバー固有のセクションがあります。サーバー固有のセクションには、特定のサーバー製品に定義されたプロパティが一覧表示されます。

たとえば、図 8-2 には、CatalogData EJB モジュールのプロパティシートの Sun Java System AS セクションが示されています。このセクションには CMP リソースのプロパティが示されており、このプロパティを使って CatalogData モジュールの CMP エンティティ Bean のデータソースを識別します。図 8-2 に表示されているデータソース jdo/PointbasePM が、図 8-1 で Sun Java System Application Server インスタンスの「登録されている持続マネージャ」として表示されていることに注意してください。

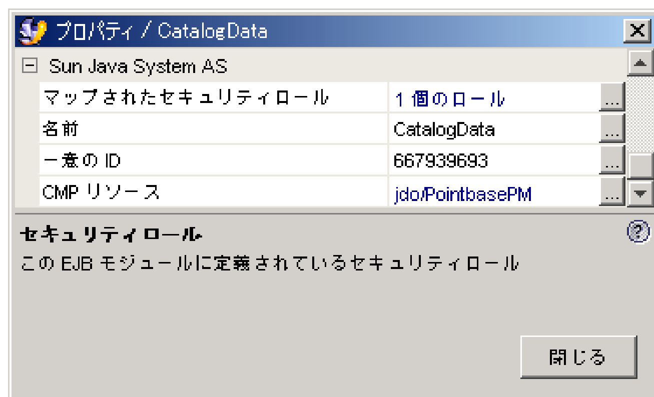


図 8-2　EJB モジュールの Sun Java System AS セクション

サーバーインスタンスノードを使用した配備と実行

この節では、IDE 内から J2EE アプリケーションを配備し、実行するための手順を概説します。

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。「アプリケーションサーバー」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、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、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表示して、その中の 1 つを選択できるブラウザダイアログです。

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。**

   これによって配備プロセスが開始します。出力ウィンドウで配備プロセスを監視してください。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。動作は、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、アプリケーションサーバーによって Web ブラウザが起動して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、IDE の外部で 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

この付録では、サーバープラグインについて簡単に説明します。サーバープラグインは、Java 2 Platform, Enterprise Edition (J2EE プラットフォーム) モジュールおよびアプリケーションの配備と実行を行う統合開発環境 (IDE) の機構です。サーバープラグインは以下の IDE 機能を提供します。

- 「実行時」ウィンドウにサーバー管理ノードを表示します。
- コンポーネント、モジュール、およびアプリケーションのサーバー固有の配備プロパティを表示します。
- J2EE アプリケーションまたはモジュールの配備、実行、デバッグを行います。

この付録では、「配備」および「実行」メニューコマンドについて説明します。ここでは、アプリケーションを配備するために必要な処理を説明し、この処理をプラグインがどのように実行するのかを解説します。

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。配備機能の使い方は、このマニュアルのいくつかのシナリオで説明されています。

配備プロセス

配備とは、実行可能な形式のモジュールまたはアプリケーションを J2EE アプリケーションサーバーに配信するプロセスです。アーカイブの形式でサーバーに配信される実行可能な形式には、モジュールあるいは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たものが含まれます。それらコンパイル済みファイルには、アーカイブの内容と構成を示す配備記述子が添付されます。アーカイブファイルは、アプリケーションサーバーが管理するディレクトリにインストールされます。

モジュールやアプリケーションを実行すると、アプリケーションサーバーは自身の制御プロセスの中で、インストールされているアプリケーションのコピーを実行します。このプロセスによって必要な実行時環境が提供されます。

問題なく配備するためには、その特定のアプリケーションサーバー製品と互換があるようにアプリケーションソースファイルをコンパイルする必要があります。配備記述子は、特定のアプリケーションサーバー製品で必要なすべての情報を含む必要があります。

このためには、製品のターゲットアプリケーションサーバーを特定し、特にそのターゲットアプリケーションサーバー用にソースファイルをコンパイルして配備記述子を生成します。

サーバープラグインの概念

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。アプリケーションを配備するときは、配備先のサーバーを選択します。IDE は適切なプラグインを使っ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サーバー固有の配備記述子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。

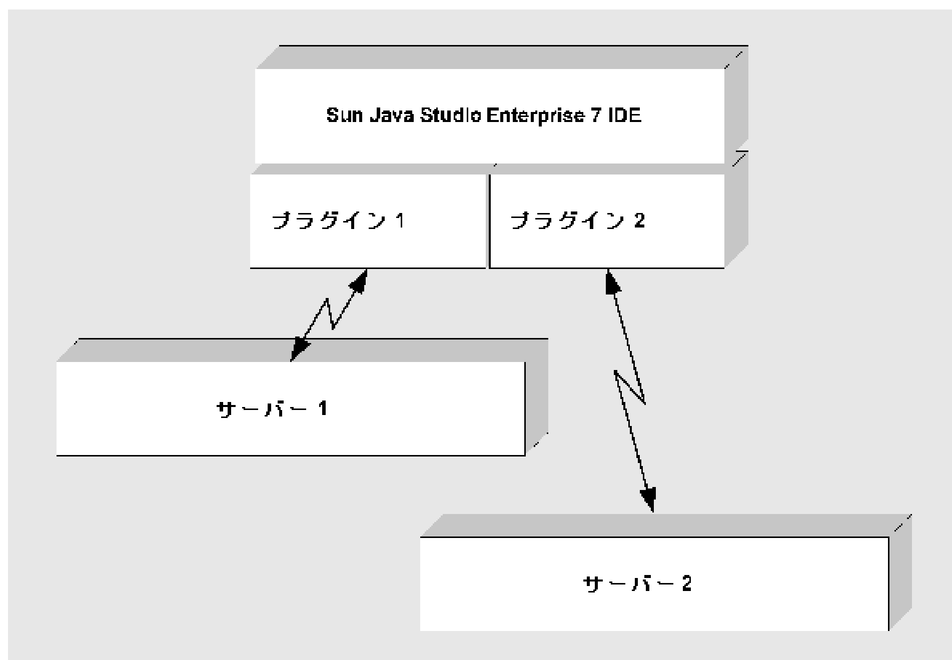


図 A-1　IDE が J2EE 実行環境と通信することを可能にするサーバープラグイン

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。

- サーバー製品を IDE で視覚的に表現します。サーバー製品ノードごとにプラグインがあります。サーバー製品ノードの表示と使用法については、115 ページの「サーバー製品ノード」を参照してください。
- 動作中のサーバーインスタンスをサーバー製品ノードのサブノードとして表示します。「実行時」ウィンドウに表示されている任意のサーバーインスタンスを配備ターゲットとして選択できます。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、117 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にサーバー固有のセクションを表示します。これらのセクションには、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。

配備プロセス

この節では、アプリケーションを配備・実行するときに行う処理をまとめています。

1. アプリケーションをアセンブルします。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。
2. アプリケーションのアセンブル後、ターゲットサーバーインスタンスを指定します。
3. IDE の配備コマンドを選択し、配備処理を開始します。
4. IDE は、アプリケーションの Web アーカイブ (WAR) ファイルまたはエンタープライズアーカイブ (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。このファイルには、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。
5. 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。
6. 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。
7. 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。このファイルには、J2EE 配備記述子、サーバー固有の配備タグ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。
8. 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。

9. 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。

このプロセスが完了すると、IDE によって Web ブラウザが自動的に起動して、アプリケーションの開始ページが開きます。配備と実行を別々に行うようにした場合は、Web ブラウザを起動してアプリケーションの Web ページの 1 つを開きます。

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。Java Studio Enterprise には、単一コンポーネント用のテストアプリケーションおよびテストクライアントを生成する機能があります。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』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

B

Bean 管理の持続性
- データベースの指定, 55
- 必要なコード, 55

E

EJB 参照
- EJB モジュール, 47
- Web コンポーネント, 28
- Web モジュール, 29
- アプリケーションレベルでのリンク, 67
- モジュールレベルでのリンク, 29
- モジュールレベルでリンクされていない, 29
- リモートインタフェース用, 26
- ローカル, 47
- ローカルインタフェース用, 47

EJB モジュール
- J2EE アプリケーションへの追加, 65
- エンタープライズ Bean の追加, 52
- 作成, 50
- 内部設計, 43
- ノード, 50
- 配備記述子, 13
- 「ファイルシステム」ウィンドウ, 11
- ファイルシステム内での配置, 50
- プロパティ, 50
- モジュールノードとソースコードの関係, 11

J

J2EE アプリケーション
- Web コンテキストの設定, 66
- アプリケーションサーバーの指定, 121
- 外部リソースの使用, 8
- 視覚表現 (イメージ), 9
- 実行, 16, 120
- ノード, 9
- ノードとソースコードの関係, 12
- 配備, 13, 113, 120
- 配備記述子, 13
- 分散型, 6
- 配備記述子により定義, 2
- 「ファイルシステム」ウィンドウ, 12
- モジュールから構成, 2
- アセンブル, 63
- 作成, 63
- ノード, 63
- 配備記述子, 63
- ファイルシステム内での配置, 63
- プロパティ, 63
- へのモジュールの追加, 65

JNDI 名
- キュー, 76
- キュー接続ファクトリ, 76
- データソース, 53
- データソースへの割り当て, 52, 54, 57

JNDI ルックアップ
- EJB ローカル参照, 47

EJB リモート参照, 26, 28
環境エントリの参照, 39
キュー, 79
キュー接続ファクトリ, 79
リソース参照, 55
例, 48
ローカルインタフェース用, 47
JSP ページ
URL, 36
Web モジュールでの表示, 11
実行, 36

S

Sun Java System Application Server
IDE とともにインストールされた, 54
サーバーインスタンスの作成, 116
サーバー製品ノード, 115
デフォルトインスタンス, 117
デフォルトの URL パス, 32

U

URL
HTML リンク内の, 24
HTML リンクへのカプセル化, 24
JSP ページの, 36
Web リソース, 32, 65
開始ページ, 23
サーブレットの URL の変更, 34
サーブレットのデフォルト URL, 32
データベース, 52
URL での実行, 31
URL パターン
HTML リンクへのカプセル化, 24
JSP ページの, 36
Web コンテキストの修飾子としての, 32
Web リソース定義での使用, 99
Web リソースの URL, 32, 65, 66
全 URL パス内の, 32
デフォルト値, 32
編集, 33

W

WEB-INF ノード, 10
web.xml ノード, 10
Web コンテキスト
J2EE アプリケーションのための設定, 65
URL パス内の位置, 32
Web モジュールの設定, 32, 66
Webリソースの URL, 32, 65, 66
Web サーバー
インスタンスの作成, 116
サーバー固有のプロパティ, 120
「実行時」ウィンドウ, 115
Web モジュール
配備記述子, 13
EJB 参照の使用, 26
ディレクトリ構造, 10
「ファイルシステム」ウィンドウ, 10
「ファイルシステム」ウィンドウでのマウント
, 10
Web モジュールでのタグライブラリ表示, 11
Web モジュール
HTML 出力を返す, 20
HTTP 要求の処理, 20
J2EE アプリケーションのフロントエンド, 20
J2EE アプリケーションへの追加, 65
Web コンテキストの設定, 66
エラーページの設定, 35
Web リソース
セキュリティロールへのマッピング, 100
定義, 98
Web サイトのホームページを開始ページにする
, 23

X

XML タグ
IDE による自動的記述, 6
コンテナ管理によるトランザクション用の, 4

あ

アプリケーションサーバー

IDE 内からの管理, 113
IDE のサーバーレジストリ内, 114
アプリケーションに対する指定, 121
インスタンスの作成, 116
環境エントリ, 38
サーバー固有のプロパティ, 120
サーバーリソースとしてのデータベースの設定, 52, 54, 57
「実行時」ウィンドウ, 115
提供される実行時サービス, 3
トランザクション, 89
ノードとして表示, 113
を使ったデータベースへのアクセス, 52

い

依存関係
EJB, 58
IDE による認識, 52, 58
モジュール, 65
「インストールされているサーバー」ノード, 115

え

エラーページ、Web モジュールの設定, 35
エンタープライズ Bean の参照
アプリケーションプロパティシートのリンク, 67
モジュールプロパティシートのリンク, 29
エンティティエンタープライズ Bean
データソースの指定, 55
データソースへのアクセスに使用, 43, 46

か

開始ファイルのデフォルト名, 25
環境エントリ
上書き, 70
参照, 38
モジュールプロパティシートの設定, 38

き

キュー
作成, 76
メソッド呼び出し, 79
メッセージの読み取り, 84
リソース環境参照, 78
アプリケーションサーバーのリソース, 76
キュー接続ファクトリ
アプリケーションサーバーのリソース, 76
カスタムの使用, 77
デフォルトの使用, 77
メソッド呼び出し, 79
リソース環境参照, 78

こ

コンテキストルートプロパティ, 32, 65
コンテナ管理によるトランザクション
実行時, 6
トランザクション属性での定義, 93, 4, 89

さ

サーバー固有のプロパティ, 13, 50, 63
サーバー製品ノード
サーバープラグインとの関係, 115
「実行時」ウィンドウ, 115
設定, 115
サーバープラグイン
IDE とサーバー間の対話の管理, 124
サーバー製品ノードで表現された, 115
サーブレット, 31
IDE のサーブレットテンプレートによる作成, 26
URL での実行, 24
URL マッピングの変更, 33
Web モジュールでの表示, 11
エンタープライズ Bean へのリモート呼び出しの作成, 26
デフォルト URL, 32
デフォルト URL の変更, 34
デフォルト URL マッピング, 33

参照
　環境エントリ, 38
　データベースのリソース参照, 52

し

実行
　IDE 内からの, 16
　手順, 16
「実行時」ウィンドウのサーバーレジストリ, 114

せ

セキュリティ
　Web モジュール内の Web リソース, 96
　エンタープライズ Bean のメソッド, 105
セキュリティロール
　EJB メソッドのアクセス権, 104, 107, 108
　EJB モジュール, 104, 107, 108
　Web モジュール, 97
　Web リソースへのマッピング, 98
　セキュリティロール参照へのマッピング, 102
セキュリティロールの参照
　ビジネスロジックでの使用, 101
　セキュリティロールへのマッピング, 102

つ

追加のファイル, 58

て

データソース
　指定, 55
　データソースリソースとして定義されたデータベース, 52
データベース
　Bean 管理の持続性, 55
　IDE とともにインストールされた PointBase, 53, 56
　アプリケーションサーバーを通したアクセス, 52
　エンティティエンタープライズ Bean の基になる, 43
　エンティティエンタープライズ Bean を使ったアクセス, 46
　指定, 52
　データソースリソースとしての設定, 54, 57
　リソース参照, 53
テストクライアントに必要なリモートインタフェース, 47
転送クラスの定義, 46

と

ローカルインタフェース
　生成されたテストクライアント, 47
トランザクション
　コンテナ管理, 3
トランザクション境界
　再定義, 91, 92
　デフォルト, 89
　トランザクション属性プロパティによる制御, 89
　トランザクション属性プロパティによる定義, 92
　配備記述子内, 3
トランザクション属性
　設定, 89, 93
　デフォルト値, 90
　配備記述子内, 4
トランザクション属性プロパティ, 3

の

ノード
　EJB モジュール, 11, 50
　EJB モジュール内のエンタープライズ Bean, 11, 50
　J2EE アプリケーション, 63
　J2EE アプリケーションのモジュール, 63
　Web コンポーネント, 10
　Web モジュール, 10
　アプリケーションサーバー, 113
　インストールされているアプリケーションサー

バー, 115
論理, 11

は

配備
Sun Java Studio Enterprise の機構, 124
手順, 14
配備記述子の使用, 13
配備記述子
EJB 参照, 29
EJB モジュール, 11, 13
IDE による生成, 3, 6, 9, 10, 13
J2EE アプリケーション, 12, 13, 63, 65
Web モジュール, 13, 11
XML ファイル, 2
アプリケーションサーバーサービスを要求するための, 3, 10
外部リソースを識別するための使用, 8
環境エントリの参照, 38
配備プロセス, 2, 13, 15
プロパティシートによる表現, 13
目的, 2
配備記述子の XML, 2
反復開発, 113

ふ

プロパティ
EJB モジュール, 50, 89
J2EE アプリケーション, 63
サーバー固有, 13, 50, 63, 120
配備記述子タグへのマップ, 10
標準, 13
プロパティエディタ, 13
プロパティシート
ノード, 10
配備記述子タグの表現, 13

ほ

ホームページ, 23

め

メソッドのアクセス権、セキュリティロールの使用, 105
メッセージ
作成, 79
送信, 79
メッセージ駆動型エンタープライズ Bean、キューコンシューマとしての構成, 84

も

モジュール
J2EE アプリケーション内, 10
組み合わせてアプリケーションを作成する, 1
コンポーネントから構成, 2
配備記述子, 3
配備記述子により定義, 2
モジュール間の対話, 6, 7
ノード, 9

り

リソース環境参照
リモート EJB 参照, 28
リソース参照
EJB モジュールのプロパティの設定, 56
JNDI ルックアップ, 55
データベースの指定, 52
ローカル EJB 参照, 49

ろ

ローカル EJB リソース参照, 49
ローカルインタフェース
JNDI ルックアップ, 47
作成, 46
リモートインタフェースとの比較, 46
ローカル参照、JNDI ルックアップ, 47