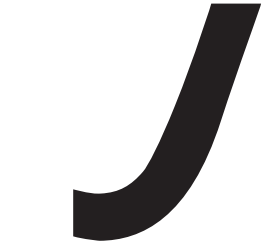


macromedia®
JRUN™4

JRun プログラマーガイド

パーツ番号 ZJR40M300J

マニュアル制作

プロジェクト管理：Randy Nielsen

執筆：Michael Peterson

編集：Linda Adler、Noreen Maher
日本語版制作管理：Sawako Gensure
日本語版制作・協力：Lionbridge Technologies, Inc.、Bart Vitti、Takashi Koto、Silvio Bichisecchi、Nathalie Delarbre、Akio Tanaka、Maasaki Suga、Yoko Kurihara、Hiroshi Okugawa、IT Frontier, Inc.

初版：2002 年 5 月

Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103, USA

マクロメディア株式会社
〒 107-0052
東京都港区赤坂 2-17-22
赤坂ツインタワー本館 13 F

目 次

このマニュアルの概要

JRun プログラマーガイド は、J2EE テクノロジーが組み込まれたアプリケーションを、JRun を使用して作成する開発者を対象としています。

ここでは、Web サイト、ドキュメント、テクニカルサポートなど、JRun および Macromedia に関連したリソースの入手方法について説明します。

目 次

JRun ドキュメントの概要

JRun ドキュメントは、JSP 開発者、サーブレット開発者、EJB クライアント開発者、EJB bean 開発者、システム管理者を含むすべての JRun ユーザーにサポートを提供することを目的としています。印刷物で提供されている場合でも、オンラインの場合でも、必要な情報を速やかに探し出せるように構成されています。JRun オンラインドキュメントには、HTML 形式と Adobe Acrobat ファイル形式があります。

印刷版ドキュメントとオンラインドキュメント

JRun のドキュメントセットには、次のドキュメントが含まれます。

マニュアル	説明
JRun インストールガイド	JRun のインストールおよび設定について説明します。
JRun 入門	J2EE の概要、概念、JSP のチュートリアル、サーブレット、EJB、および Web サービスについて説明します。
JRun 管理者ガイド	JRun サーバーを既存の環境に統合する方法について説明します。
JRun プログラマーガイド	JRun を使用して JSP、サーブレット、カスタムタグ、EJB、および Web サービスを開発する方法を説明します。
JRun アセンブルとデプロイガイド	J2EE アプリケーションコンポーネントのアセンブルおよびデプロイの方法を説明します。
JRun SDK ガイド	OEM/ISV のお客様、および JRun で API の埋め込み、カスタマイズ、使用を行う上級ユーザーを対象に情報を提供します。
JSP クイックリファレンス	JSP (JavaServer Pages) のディレクティブ、アクション、およびスクリプト要素の簡単な説明とシンタックスが記載されています。
オンラインヘルプ	JMC ユーザーに、使用上の注意、方法、および概念を提供します。

オンラインドキュメントへのアクセス

すべての JRun ドキュメントは、HTML 形式と Adobe Acrobat ファイル形式でオンラインで利用できます。ドキュメントにアクセスするには、JRun を実行しているサーバー上で <JRun のルートディレクトリ >/docs/dochome.htm ファイルを開きます。<JRun のルートディレクトリ > とは、JRun がインストールされているディレクトリのことです。

Macromedia 社では、JRun の全マニュアルのオンライン版を Adobe Acrobat Portable Document Format (PDF) ファイルで提供しています。PDF ファイルは JRun CD-ROM にも含まれており、オプションで JRun /docs ディレクトリにインストールされます。JRun 管理コンソールのトップページにある製品ドキュメントへのリンクをクリックすると、これらの PDF ファイルにアクセスできます。

その他のリソース

JRun のドキュメントで説明されているトピックの詳細については、次のリソースも参照してください。

書籍

サーブレット、JavaServer Pages、タグライブラリ	
Java Server Pages Application Development	Scott M. Stirling 他著 Sams 刊、2000 年 ISBN：067231939X
< 邦訳 > JSP アプリケーション開発ガイド - 実践的アプリケーションの構築	Ben Forta 監修 ピアソン・エデュケーション刊 ISBN：489471468X
More Servlets and JavaServer Pages	Marty Hall 著 Prentice Hall PTR 刊、2001 年 ISBN：0130676144
Core Servlets and JavaServer Pages	Marty Hall 著 Prentice Hall PTR 刊、2000 年 ISBN：0130893404
< 邦訳 > コア・サーブレット & JSP	Marty Hall 著 ソフトバンクパブリッシング 刊 ISBN：4797314311
Java Servlet Programming, Second Edition	Jason Hunter、William Crawford 著 O'Reilly & Associates 刊、2001 年 ISBN：0596000405
< 邦訳 > Java サーブレットプログラミング	Jason Hunter、William Crawford 著 オライリー・ジャパン 刊 ISBN：4873110718
Java Servlets Developer's Guide	Karl Moss 著 McGraw-Hill/Osborne Media 刊、2002 年 ISBN：0-07-222262-X
Inside Servlets:Server-Side Programming for the Java Platform, Second Edition	Dustin R. Callaway 著 Addison-Wesley 刊、2001 年 ISBN：0201709066

Web Development with JavaServer Pages	Duane K. Fields、Mark A. Kolb 著 Manning Publications Company 刊、2000 年 ISBN：1884777996
< 邦訳 > JSP による Web 開発 サーブレットアーキテクチャを利用した新しいコンテンツ開発技法	Duane K.Fields、Mark A.Kolb 著 翔泳社 刊 ISBN：4798100048
Enterprise Java Servlets	Jeff Genender 著 Addison-Wesley 刊、2001 年 ISBN：020170921X
Advanced JavaServer Pages	David Geary 著 Prentice Hall 刊、2001 年 ISBN：0130307041
JavaServer Pages (JSP)	Hans Bergsten 著 O'Reilly & Associates 刊、2000 年 ISBN：156592746X
JSP Tag Libraries	Gal Schachor、Adam Chace、Magnus Rydin 著 Manning Publications Company 刊、2001 年 ISBN：193011009X
Core JSP	Damon Hougland、Aaron Tavistock 共著 Prentice Hall 刊、2000 年 ISBN：0130882488
< 邦訳 > Core JSP	Damon Hougland、Aaron Tavistock 著 ピアソン・エデュケーション 刊 ISBN：4894714574
JSP:Javaserver Pages (Developer's Guide)	Barry Burd 著 Hungry Minds Inc. 刊、2001 年 ISBN：0764535358
Enterprise JavaBeans	
Mastering Enterprise JavaBeans, Second Edition	Ed Roman 著 John Wiley & Sons 刊、2002 年 ISBN：0471417114
Enterprise JavaBeans, Third Edition	Richard Monson-Haefel 著 O'Reilly & Associates 刊、2001 年 ISBN：0596002262
Professional EJB	Rahim Adatia 他著 Wrox Press 刊、2001 年 ISBN：1861005083

Special Edition Using Enterprise JavaBeans (EJB) 2.0	Chuck Cavaness、Brian Keeton 共著 Que 刊、2001 年 ISBN：0789725673
Applying Enterprise JavaBeans:Component-Based Development for the J2EE Platform < 邦訳 > Enterprise JavaBeans 開発ガイド	Vlada Matena、Beth Stearns 著 Addison-Wesley Pub Co 刊、2000 年 ISBN：0201702673 Vlada Matena、Beth Stearns 著 ピアソン・エデュケーション 刊 ISBN：4894714639
Enterprise Java プログラミング	
Professional Java Server Programming J2EE 1.3 Edition	Subrahmanyam Allamaraju 他著 Wrox Press 刊、2001 年 ISBN：1861005377
Server-Based Java Programming	Ted Neward 著 Manning Publications Company 刊、2000 年 ISBN：1884777716
Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition < 邦訳 > Java 2 Platform, Enterprise Edition アプリケーション設計ガイド	Nicholas Kassem 著 Addison-Wesley 刊、2000 年 ISBN：0201702770 (java.sun.com/j2ee/download.html#blueprints から無償でダウンロードできます。) ピアソン・エデュケーション 刊 ISBN：4894713233 (日本語版は、http://java.sun.com/blueprints/ja/index.html (から無償でダウンロードできます。)
Building Java Enterprise Systems with J2EE	Paul Perrone、Venkata S.R. "Krishna" .R. Chaganti 共著 Sams 刊、2000 年 ISBN：0672317958
J2EE:A Bird's Eye View (e-book)	Rick Grehan 著 Fawcette Technical Publications 刊、2001 年 ISBN：B00005BAZV
Java Message Service < 邦訳 > Java メッセージサービス	Richard Monson-Haefel、David Chappell 著 O'Reilly and Associates 刊、2001 年 ISBN：0596000685 Richard Monson - Haefel、David A. Chappell 著 オライリー・ジャパン 刊 ISBN：4873110580

J2EE Connector Architecture and Enterprise Application Integration	Rahul Sharma 他著 Addison-Wesley 刊、2001 年 ISBN：0201775808
Building Web Services with Java:Making Sense of XML, SOAP, WSDL and UDDI	Sim Simeonov、Glen Daniels、他著 Prentice Hall 刊、2002 年 ISBN：0672321815
Architecting Web Services	William L. Oellermann Jr. 著 Apress 刊、2001 年 ISBN：1893115585

オンラインリソース

Java Servlet API	http://java.sun.com/products/servlet
JavaServer Pages API	http://java.sun.com/products/jsp
Enterprise JavaBeans API	http://java.sun.com/products/ejb/
Java 2 Standard Edition API	http://java.sun.com/j2se/
Servlet Source	http://www.servletsource.com
JSP Resource Index	http://www.jspin.com
Server Side	http://www.theserverside.com
Dot Com Builder	http://dcb.sun.com
Servlet Forum	http://www.servletforum.com

Macromedia 社へのお問い合わせ

開発元： Macromedia, Inc.	600 Townsend Street San Francisco, CA 94103 U.S.A. Web：http:// www.macromedia.com
販売元： マクロメディア株式会社	〒 107-0052 東京都港区赤坂 2-17-22 赤坂ツインタワー本館 13F 電話：03-5563-1980 FAX：03-5563-1990 Web：http://www.macromedia.com/jp/
テクニカルサポート	オンライン Web サポートおよび電子メールでのテクニカルサポートを提供させていただいています。ユーザー登録はがき等に記載されている方法にてお問い合わせください。 テクニカルサポートサービスの詳細については、http://www.macromedia.com/jp/support/ をご覧ください。
セールス	製品のライセンス、価格、サポート、トレーニング、コンサルティングなど、OEM/ ホスティングライセンスなどについては、次の連絡先までお問い合わせください。 電話：(03)5563-1980 電子メール：service-j@macromedia.com

パート I
JRun によるプログラミングの導入

パート I では、JRun プログラミング環境を紹介し、主なトピックについて概説します。次の章で構成されています。

第 1 章
JRun プログラミングの概要

この章では、Macromedia JRun によるプログラミングの概要と、Java 2 Platform Enterprise Edition (J2EE) のプログラミングリソースを紹介します。

目 次

JRun プログラミングの導入

Macromedia JRun 4 は、Java 2 Platform Enterprise Edition (J2EE) をサポートするアプリケーションサーバーです。J2EE は、サーバーベースの Java アプリケーションの作成に使用する業界標準のプラットホームです。Java プラットホームと同様、J2EE プラットホームは、Sun Microsystems が一連のプラットホーム仕様を通じて管理します。各仕様は、特定のテクノロジーの機能や API (Application Programming Interface：アプリケーションプログラミングインターフェイス) を定義します。

J2EE プラットホーム仕様は、プラットホーム仕様を統一するだけでなく、さまざまなロールも定義します。各ロールは、エンタープライズアプリケーションの特定の機能を担当します。ただし、Java はプログラム言語であるため、中心的なロールはアプリケーション開発者のロールです。開発者は、HTML や JavaServer Pages (JSP)、Java などの言語を記述して、J2EE API の機能、クラス、インターフェイスなどを使用するサーバーベースのアプリケーションを作成します。

多くの開発者は、コーディングだけでなくその他の J2EE のロールも担当しています。たとえば、アプリケーションのアセンブリやデプロイなどがあります。

次の表で、このマニュアルに記述されている J2EE API について説明します。

J2EE API	説明	参照先
サーブレット	Java 開発者は、既存のビジネスシステムにアクセスするダイナミックな Web ページを作成できます。	パート II 「サーブレットプログラミング」
JSP	Web 開発者とデザイナーは、ダイナミックな Web ページとカスタムタグを作成できます。	パート III 「JSP プログラミング」
Enterprise JavaBeans (EJB)	Java 開発者は、トランザクション、セキュリティ、データベースコネクティビティなどのミドルウェアサービスを自動的にサポートするアーキテクチャのビジネスコンポーネントを作成、使用できます。	パート IV 「EJB プログラミング」
Java Message Service (JMS)	Java 開発者は、移植可能なメッセージベースのアプリケーションを作成できます。	パート V 「JMS プログラミング」
Web サービス	Java 開発者や Web 開発者は、XML や HTTP などの標準インターネットプロトコルを使用して、プラットフォームや位置に依存しないコンピューティングを行う分散型ソフトウェアコンポーネントをパブリッシュし、呼び出すことができます。Web サービスは、技術的には J2EE テクノロジーではありませんが、分散アプリケーションの開発に重要な機能を提供します。	パート VI 「Web サービスのプログラミング」

J2EE テクノロジー、標準、および準拠の詳細については、『JRun 入門』を参照してください。Java Authentication and Authorization Service (JAAS) および J2EE Connector Architecture (JCA) を用いた JRun の使用については、『JRun 管理者ガイド』を参照してください。

また、このマニュアルには、Macromedia Flash と JRun を使ったリッチインターネットアプリケーションの開発についての情報も含まれています。JRun は、Flash クライアントのインターフェイスと J2EE アプリケーションとのネイティブコネクティビティをサポートしています。詳細については、483 ページの第 24 章「Flash と JRun の併用」を参照してください。

プログラミングリソース

このマニュアルでは、JRun を使用した J2EE アプリケーションのプログラミング方法について説明します。ただし、JRun 関連のドキュメントは、入手可能な J2EE プログラミング情報のごく一部に過ぎません。ここでは、J2EE とそのプログラミングに関するさまざまな情報ソースについて説明します。

java.sun.com の Web サイト

J2EE テクノロジーは Sun Microsystems が管理しており、各種のリソースは http://java.sun.com の Web サイトより入手できます。J2EE プログラマーとして効率的なコーディング技術を習得するには、次のような Sun Microsystems が提供するオンラインリソースをよく理解する必要があります。

- **Java 2 Standard Edition SDK (http://java.sun.com/j2se/)** 複数のバージョンやプラットホームの Java 2 SDK、API ドキュメント、記事などにアクセスできます。
- **Java 2 Enterprise Edition (http://java.sun.com/j2ee/)** 複数のバージョンやプラットホームの J2EE SDK、API ドキュメント、仕様書、記事などにアクセスできます。J2EE SDK はリファレンス実装とも呼ばれます。
- **Java BluePrints (http://java.sun.com/blueprints/)** ガイドライン、パターン、サンプルコードなどを提供します。BluePrints の手始めには、Java Pet Store J2EE アプリケーション、SmarTicket J2ME アプリケーション、および『Designing Enterprise Applications with the J2EE Platform』のマニュアルが含まれています。
- **仕様書** 各仕様の機能、API、およびデプロイメントディスクリプタに関する詳細が解説されています。ご使用の J2EE API に関する最新の仕様書をダウンロードしてください。J2EE のホームページ (http://java.sun.com/j2ee) からも仕様書にアクセスできます。
- **開発者センター (http://www.sun.com/developer)** さまざまな技術リソースへのリンクを提供しています。

オンラインリソース

Sun Microsystems の Web サイトで利用できるリソースのほか、J2EE やそのテクノロジーを専門としている他の Web サイトでも有用な情報を見つけることができます。これらの Web サイトでは、記事や討論会、ニュースレター、ダウンロード、およびより有能な J2EE 開発者になるために利用できるリソースが提供されています。特に、次の Web サイトを参照してください。

- **Macromedia (http://www.macromedia.com/software/jrun)** Macromedia の JRun に関するあらゆるオンラインリソースに簡単にアクセスできます。
- **The Server Side (http://theserverside.com)** 討論会、レビュー、記事、マニュアルなど、J2EE に関するさまざまな情報を提供しています。
- **Servlet フォーラム (http://www.servletforum.com)** サーブレットのトピックに関するオンライン討論会を提供しています。

業界誌

過去数年間に渡り、繁栄する市場で J2EE テクノロジーに関する業界誌が刊行されてきました。これら業界誌の多くがお近くの書店で入手できるだけでなく、オンラインの情報ソースまたはオンライン書店でも入手可能です。これらの業界誌は、広範で、実践的かつ専門的な知識を持つ業界の専門家によって書かれています。また、詳細なコードが例示され、詳細なバックグラウンドによる議論が行われています。J2EE で効果的なプログラムを書くには、ご使用のテクノロジーに関するこれらの業界誌を最低 1 冊は購読されることをお勧めします。

これらの業界誌には、JRun による J2EE アプリケーションの開発方法が説明されていますが、詳細なプログラミングの問題に関しては、業界誌の他に Sun の仕様書を参照してください。外部リソースの一覧表については、xix ページの「その他のリソース」を参照してください。

デザインパターンとプログラミングテクニック

コンピュータプログラミングは、常に変化する専門分野です。構造化プログラミングから非手続き型 4GL (Fourth-Generation Languages：第 4 世代言語)、OOP (Object-Oriented Programming：オブジェクト指向プログラミング) まで、プログラマー、アナリスト、デザイナーたちは、パフォーマンス、生産性、保守性を向上するための最新技術を最適化する方法を模索しています。

J2EE 業界における最新のベストプラクティスは、デザインパターンの活用です。Sun BluePrints の Web サイトには、「デザインパターンは、繰り返し発生するデザイン問題に対して、問題を取り巻く背景や影響、あるいはソリューションの結果や影響を重視する、実績あるソリューションを提供するものです。」と、記載されています。

このマニュアルの一部の章では、特定のプログラミングテクノロジーを実践するコード例が紹介されています。このマニュアルのコンテキストでは、プログラミングテクニックは特定の用法やコーディング手順を実践するものであり、デザインパターンの効果的な使用方法を説明していないという点で、両者を区別しています。

JRun サンプル

samples JRun サーバーには、さまざまなプログラミングテクニックを紹介するアプリケーションが多数含まれています。次の表で、JRun に同梱されているサンプルアプリケーションについて説明します。

アプリケーション	説明
Compass Travel	簡単な旅行代理店のオンライン旅行予約システムです。このアプリケーションは、JSP、サーブレット、および EJB プログラミングについて説明するもので、オープンディレクトリ構造体のエンタープライズアプリケーションです。
TravelNet	Compass Travel で旅行を販売するオンラインの旅行代理店です。このアプリケーションは、JSP や Web サービスのプログラミングについて説明するもので、オープンディレクトリ構造体のエンタープライズアプリケーションです。
World Music	音楽販売の e- コマースアプリケーションです。このアプリケーションには、管理モジュールも含まれています。World Music アプリケーションでは、JSP、サーブレット、および JavaBean プログラミングによる一般的なデザインパターンが多数説明されています。
Web サービス	JRun の Web サービスプログラミングに使用されるプログラミングテクニックについて紹介しています。
プログラミングテクニック	このマニュアルで説明されているプログラミングテクニックを紹介しています。
Flash Gateway	Macromedia の Flash ムービーや Flash Gateway アダプタが含まれています。これは JRun と連動する Flash アプリケーションの作成方法を示します。
SmarTicket	Java BluePrints J2ME アプリケーション
PetStore	Java BluePrints J2EE アプリケーション

これらのサンプルは、JRun データソースを通してデータベースにアクセスします。データソース情報は、JRun 管理コンソール (JMC) から表示できます。これらのサンプルの詳細については、samples JRun サーバーを起動し、ホームページ (デフォルトは http://localhost:8200) を開いてください。

このマニュアルの内容

このマニュアルは、JRun 環境での J2EE プログラミングについて説明します。複数の部に分かれていますが、各部では、次の表で説明するように、特定のテクノロジーを取り扱っています。

部	内容
パート I 「JRun によるプログラミングの導入」	デザインパターン、XML、セキュリティなど、あらゆる J2EE テクノロジーで応用できます。
パート II 「サーブレットプログラミング」	サーブレットのプログラミングテクニックを解説します。
パート III 「JSP プログラミング」	JSP によるプログラミングテクニックや Java および JSP によるカスタムタグのコーディングに関する章が含まれています。
パート IV 「EJB プログラミング」	JRun EJB プログラミングおよび EJB プログラミングテクニックに関する章が含まれています。
パート V 「JMS プログラミング」	JRun JMS プログラミングおよび JMS プログラミングテクニックに関する章が含まれています。
パート VI 「Web サービスのプログラミング」	Web サービスの概念、JRun Web サービスのアーキテクチャ、および Web サービスプログラミングに関する章が含まれています。
パート VII 「プログラミングについてのその他のトピック」	Macromedia Flash MX を用いた JRun をバックエンドとするリッチメディアインターフェイス開発について解説します。

第 2 章
デザインパターン

この章では、プログラミング初心者が、適切なプログラミング方法を使用して作業を開始するのに役立つ情報を記載します。また、これらの問題に関する詳細情報が掲載されているさまざまな書籍や Web リソースについても紹介します。

目 次

デザインパターンについて

デザインパターン は、一般的であり、繰り返し起こる問題を解決する実証済みの解決策です。アブストラクションレベルでは、しばしば、問題に対する専門的な解決策をデザインパターンで記述します。

パターンによって、開発者は、問題を個々の要素に分けることができるようになります。これにより、コードの保守性や再利用性が高くなり、コードも理解しやすくなります。また、デザインパターンの要素もカスタマイズや埋め込みが可能で、互いに自在に結合します。パターンの部品は、アプリケーションを中断することなく追加または削除を行うことができます。このような透明性により、作業上の矛盾を最小限に抑えながら、さまざまなタイプの開発者がプロジェクトに取り組むことができます。

この章では、J2EE コンテキストでのデザインパターンについて説明していきますが、これらの多数の技術はどのようなプログラミング環境にも応用できます。

パターンテンプレート

通常、パターンはテンプレートフォームで作成します。1 つのパターンで複数のパターンを記述する場合もあります。この章では、デザインパターンを構成して記述するための簡略化されたテンプレートを使用します。次の表で、このテンプレートを説明します。

セクション	説明
名前	パターンの一般名。
問題	開発者が直面するデザイン上の課題を説明します。
解決策	問題を解決するための高レベルアプローチ。
戦略	低レベルにおけるパターンの実装方法を説明します。

Sun の Web サイトに掲載されているデザインパターンカタログには、より強力なテンプレートが紹介されています。Sun テンプレートには、目的、動機、応用性、構造、参加者、コラボレーション、結果、実装が含まれています。詳細については、http://java.sun.com/blueprints/patterns/j2ee_patterns を参照してください。

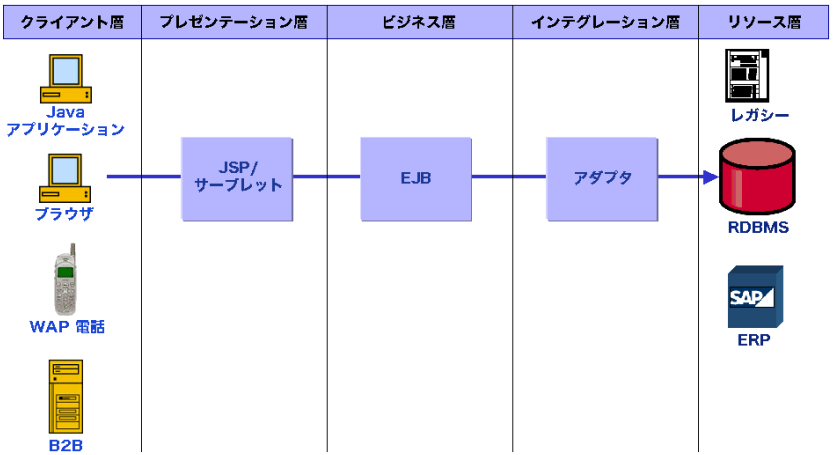
『Core J2EE Patterns』には、コンテキスト、問題、フォース、解決策、および結果を定義したテンプレートを若干変更したものを紹介しています。

階層手法

デザインパターンの実装には、Web アプリケーション構築のための階層手法を使用する必要があります。これらの階層は次の要素で構成されています。

- クライアント
- プレゼンテーション
- ビジネス
- インテグレーション

次の図は、各階層と互いの関係を示しています。



クライアント層

クライアント層は、アプリケーションにアクセスするすべてのデバイスおよびシステムで構成されます。これらのデバイスやシステムには次のものがあります。

- Web ブラウザ
- 端末装置
- PDA
- WAP デバイス
- Web サービスクライアント

各タイプのクライアントはさまざまな方法でアプリケーションと通信するので、Web アプリケーションからクライアント層を分離すると便利です。クライアントを識別し、適切な形式のデータを返す役割は、クライアント層の下の階層が担当します。

プレゼンテーション層

プレゼンテーション層には、ユーザーインターフェイス、認証、承認の構築やセッションステートの管理などのプレゼンテーションロジックがカプセル化されています。プレゼンテーション層は、ビジネス層にアクセスするためのさまざまな方法をクライアントに提供します。この階層はビューとも呼ばれます。

J2EE では、プレゼンテーション層は、サーブレットおよび JSP として実装されます。これらのコンポーネントは、クライアントビューを簡単に生成できるように特別にデザインされています。各クライアントには、さまざまな形式のデータが必要になります。プレゼンテーション層は、アプリケーションのロジックから出力 (またはレスポンス) の形式を分離するのに役立ちます。

ほとんどのパターンのデザインには、いくつかのフォームのプレゼンテーション層があります。次のパターンにはプレゼンテーション層が含まれています。

- View Helper
- Front Controller
- Intercepting Filter
- Service to Worker
- Dispathcer View
- Model-View-Controller

ビジネス層

ビジネス層は、アプリケーションのビジネスロジックを担当します。ビジネス層は、プレゼンテーション層からの変数データおよびユーザー入力に基づいて出力を生成し、プレゼンテーション層に渡します。ビジネス層は通常、EJB として実装されますが、サーブレットおよび JavaBeans が含まれている場合があります。ビジネス層は、しばしばヘルパークラスという形態をとります。

インテグレーション層

インテグレーション層は、データベース、CRM システム、レガシーアプリケーションなどの外部リソースとの通信の管理を担当します。

階層とロール

よくデザインされた Web アプリケーションの階層は、J2EE プラットフォームのロール (役割) の定義に相当します。J2EE プラットフォームでは、Web アプリケーションの開発ライフサイクルの中で、さまざまな人々が個々のロールを担当します。これらの人々はそれぞれ、次の表に示すように Web アプリケーションの 1 つのロールを担当します。

ロール	担当
アプリケーションコンポーネントプロバイダ	アプリケーションの構成要素を記述します。 さまざまな階層で作業を行うことができます。このプロバイダは、生成するコンポーネントのタイプによってさまざまなロールに分けられます。このようなコンポーネントのタイプには、プレゼンテーションコンポーネント (グラフィックデザイナーおよび HTML コーダー) やビジネスロジック (EJB として実装) などがあります。
アプリケーションアセンブル担当者	アプリケーションコンポーネントプロバイダによって提供されたコンポーネントを J2EE アプリケーションにまとめます。 アプリケーションを、基盤リソースへの直接リファレンスから分離します。 また、インテグレーション層におけるリソースも担当します。
デプロイ担当者デプロイ担当者	特定の環境におけるアプリケーションの設定を担当します。 通常はリソース層を担当します。
システム管理者	ランタイム環境においてアプリケーションの管理と監視を担当します。 通常は、Web アプリケーション構造のいずれの階層においてもロールを担当することはありません。

しばしば、1 人の人間が複数のロールと作業を担当します。デザインパターンは、疎結合性を利用することによってアプリケーションコンポーネントの独立性を維持するのに役立ちます。たとえば、データベースインテグレーションロジックはプレゼンテーションロジックから分離する必要があります。依存関係を排除することにより、アプリケーションはさまざまなシステムおよびプラットフォームに移植できるようになります。

クラスやコンポーネントが属している階層が明確に識別できない場合はデザインに問題があります。たとえば、データベースにアクセスする JSP があると、その JSP がプレゼンテーション層に属しているのか、インテグレーション層に属しているのかを識別できません。

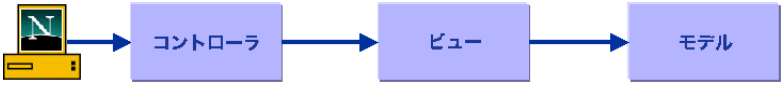
MVC (Model-View-Controller)

MVC (Model-View-Controller) デザインパターンはよく知られているデザインパターンです。MVC デザインパターンは、アプリケーションを 3 つの層に分けて定義します。

層	説明
モデル	アプリケーション内のビジネスロジック。前述のビジネス層およびインテグレーション層に相当します。 EJB およびサーブレットとして実装され、しばしばヘルパークラスで構成されます。
ビュー	アプリケーション内のプレゼンテーションロジック。前述のプレゼンテーション層に相当します。 しばしば JSP およびサーブレットとして実装されます。
コントローラ	アプリケーションの最初の接触ポイント。リクエストをディスパッチし、認証、承認、および他のアプリケーション動作およびシステムを実装します。 前のクライアント層およびプレゼンテーション層の間に位置します。 しばしばサーブレットまたは JSP として実装されます。

MVC アプリケーションには、コントローラを変更することによって、新規クライアントビューを追加したり、アプリケーションとのクライアント対話をすばやく更新したりできるという利点があります。

次の図にこの戦略を示します。



MVC の最も一般的な実装は Struts フレームワークです。詳細については、27 ページの「Struts について」を参照してください。MVC パターンは、Dispathcer View パターンおよびこの章で説明している他のいくつかのパターンに密接に関連しています。

パターンの実装を支援するコンポーネント

J2EE には、デザインパターンの概念構造をエミュレートするのに役立つ多数のコンポーネントやアブストラクションが用意されています。これらの一部のツールの機能は重複していますが、全体的な構造では独自の能力を発揮します。次の表で、これらのツールの一部について説明します。

コンポーネント	デザインパターンとの関係
JavaBeans	JSP およびサーブレットがビュー外部でビジネスロジックを実行する際に支援します。ステート情報を保管します。
EJB	スケーラブルなエンタープライズビジネスロジックを実行します。
JSP	プレゼンテーション層またはビュー層に使用します。基本的にビューロジックが含まれています。
サーブレット	コントローラ、フィルタ、ビジネスロジック、またはヘルパークラスとして動作する汎用性クラス。
ヘルパークラス	特定のクラスを実行します。 ビューからのビジネスロジックの分離に役立ちます。 ビューからモデルへのアクセスを提供します。
ファクトリクラス	オブジェクトの特殊な生成および廃棄を実装します。
データアクセスオブジェクト	特殊なデータベースロジックを他のコンポーネントから分離します。
カスタムタグライブラリ	JSP からのビジネスロジックの削除に役立ちます。 モデルからビューへのアクセスを提供します。
フィルタ	リクエストの前処理および後処理。 リクエストを転送できるため、Front Controller として適しています。
イベントリスナ	サーブレットおよび JSP からビジネスロジックを分離するのに役立ちます。 ロギング、初期化、オブジェクトの廃棄、ファクトリアクセスなどの管理ロジックを提供します。

View Helper パターン

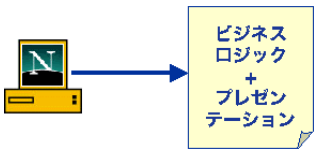
このセクションでは、12 ページの「パターンテンプレート」で定義されている形式の View Helper デザインパターンについて説明します。

問題

Web アプリケーション開発の一般的な問題は、ビューにおいて、ビジネスロジックとプレゼンテーションロジックが混在していることです。JSP やサーブレットの JSP スクリプトレットまたはメソッドにコードを詰め込みすぎると、次の点で問題が発生します。

- 再利用性
- 柔軟性
- 保守容易性
- ロールの独立性

次の図は、1 つのコンポーネントにビジネスロジックとプレゼンテーションロジックを組み合わせたところを示しています。



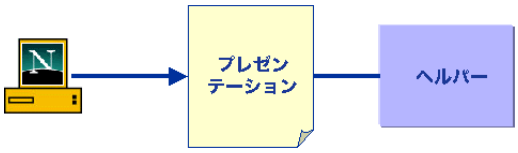
解決策

問題の解決策は、ビューからすべてのビジネスロジックを排除し、これを、ヘルパークラスとしてモデルロジック層またはビジネスロジック層に移動することです。ビューには、モデルの出力形式を設定するために使用するコードだけを含めてください。

ヘルパークラスの役目は次のとおりです。

- ビューによって要求されたデータの収集
- ビューが使用するデータモデルの最適化
- バックエンドシステムおよびデータベースへのアクセスの提供

次の図に、プレゼンテーションロジックとビジネスロジックを分離したところを示します。



戦略

View Helper パターンを実装する場合は、ビューを JSP として作成し、すべてのスクリプトレットを排除する方法をお勧めします。ヘルパークラスはカスタムタグまたは JavaBeans として実装できます。

Front Controller パターン

このセクションでは、12 ページの「パターンテンプレート」で定義されている形式の Front Controller デザインパターンについて説明します。

問題

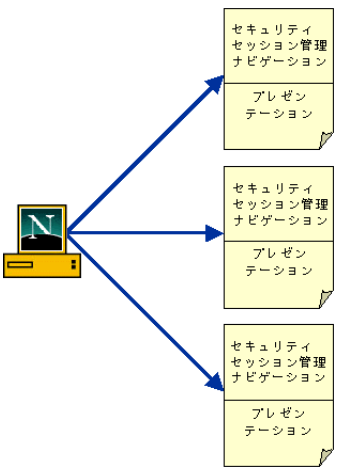
デザインが不適切な多くの Web アプリケーションでは、クライアントはビューに直接アクセスします。その結果、Web アプリケーションにおけるアクセス制御または他の管理機能を実装することが困難になります。該当するターゲットリソースにリクエストを転送するアプリケーションの場合、しばしばフロントエンドを実装する必要があります。

ビューには、次のようなシステムサービスを提供するロジックが含まれている場合があります。

- 認証
- 承認
- セッション管理
- ナビゲーション
- ローカリゼーション

したがって、アプリケーションのコードが分散または重複して、保守が困難になります。ビジネスロジックとプレゼンテーションロジックの分離が困難であるということも問題となります。これについては、17 ページの「View Helper パターン」で説明しています。

次の図に、別個のコンポーネントとして理想的に実装されるビジネスロジックが含まれている各コンポーネントを示します。



解決策

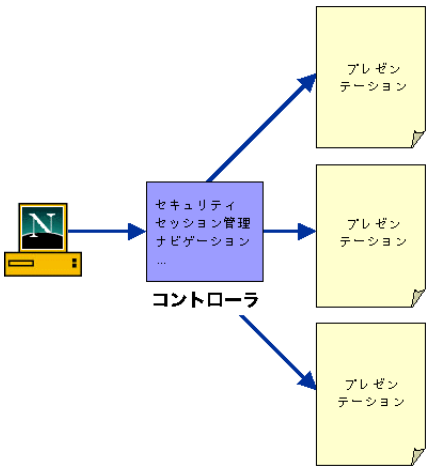
Front Controller パターンは、同じコンポーネントにおいて、プレゼンテーションロジックからビジネスロジックを強制的に分離します。このパターンでは、すべてのリクエストの最初のアクセスポイントとしてコントローラを使用します。コントローラは、リクエストにおける次の側面を処理します。

- 認証
- 承認
- セッション管理
- ナビゲーション
- デバッグ
- ローカリゼーション

システムサービスの仕事をコントローラに肩代わりさせることによって、Front Controller パターンは、プレゼンテーションおよびモデルの処理をターゲットリソースに渡します。

アプリケーションでは、サービスの個々のセットを呼び出すことによって複数のコントローラを使用できます。

次の図に、さまざまなプレゼンテーション層のコンポーネントにリクエストを転送するコントローラを示します。



ビジネスロジックの実装を支援する追加コードがコントローラに含まれている場合、パターンは、Service to Worker パターンと類似したものになります。詳細については、25 ページの「Service to Worker パターン」を参照してください。

戦略

Front Controller パターンを実装する場合は、JSP にリクエストを送信するフィルタまたはサーブレットのチェーンとして実装する方法をお勧めします。コントローラ内で論理リソースマッピングを使用するか、設定ファイル内でリソースマッピングを定義してください。

パターンを実行するには、JSP を /WEB-INF ディレクトリに配置してください。クライアントはそのディレクトリ内のリソースを直接リクエストできないため、サーブレットまたは他の Web コンポーネントがコントローラとして機能し、リクエストをこの JSP に送信する必要があります。

Front Controller サーブレットのサンプル

次は、コントローラ実装のサンプルコードです。サーブレットの doGet および doPost メソッドはいずれも processRequest メソッドを呼び出します。processRequest で、このコントローラは、リクエストパラメータを抽出し、その値を転送先として使用します。

```
public class Controller extends HttpServlet {
 protected void processRequest(HttpServletRequest req,
        HttpServletResponse res throws ServletException,
        java.io.IOException {
  page = req.getParameter("page");
  RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher(page);
  dispatcher.forward(req, res);
 }
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, java.io.IOException {
  processRequest(req, res);
 }
 protected void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, java.io.IOException {
  processRequest(req, res);
 }
}
```

Intercepting Filter パターン

このセクションでは、12 ページの「パターンテンプレート」で定義されている形式の Intercepting Filter デザインパターンについて説明します。

問題

状況により、リクエストとレスポンスの前処理および後処理が必要になる場合があります。次の疑問点を確認してください。

- クライアントは認証されたか。
- クライアントのセッションは有効か。
- クライアントの IP アドレスは信頼されているネットワークのものか。
- クライアントがデータを送信するときに使用するエンコードは何か。
- クライアントのブラウザはサポートされているか。

Web アプリケーションには、設定ファイルで定義されている、ネストされた if/else ステートメントまたは URL パターンマッチングよりも柔軟で再利用性が高い手法が必要になります。その理由は次のとおりです。

- アプリケーションが保守しやすくなります。
- 複数のアプリケーションにわたってサービスを再利用できます。
- サービスの追加や削除が可能です。

解決策

Intercepting Filter パターンは、次のような一般的なサービスを処理する埋め込み可能なフィルタのロールを定義します。

- セキュリティ
- ログ
- ステート管理
- デバッグ

フィルタは、受信リクエストと送信レスポンスを阻止します。フィルタの再設定は、Web アプリケーションの web.xml デプロイメントディスクリプタを使用して行うことができます。そのため、フィルタを追加および削除する際にアプリケーションを再コンパイルする必要はありません。また、フィルタの順番を変更したり、フィルタの動作をダイナミックに変更することができます。

フィルタを使用したサーブレットの詳細については、179 ページの第 7 章「フィルタ」を参照してください。

戦略

フィルタをサーブレットとして実装するには、`javax.servlet.Filter` インターフェイスを使用します。

フィルタを使用すると、チェーン内の他のフィルタにリクエストおよびレスポンスを転送したり、別のサーブレットとして実装されているフィルタコントローラにリクエストやレスポンスを渡したりすることができます。

フィルタを実装する方法としては、デコレータパターンまたは非デコレータパターンがあります。これらのマイクロパターンについては、次のセクションで説明します。

デコレータマイクロパターン

デコレータマイクロパターンは、新規機能を個々のオブジェクトにダイナミックかつ透過的に追加するクラスで構成されています。これらの機能は、他のフィルタの形態をとることがあり、あるフィルタから次のフィルタへの制御の流れをコントロールすることによって、フィルタをダイナミックに埋め込み可能にすることができます。

次の図にこの戦略を示します。

このフィルタは他のコンポーネントを使用してリクエストパラメータを設定します。

Decorator Filter のサンプル

次のサンプルは、基本的なフィルタに埋め込まれたフロー制御ロジックを示しています。承認されていないユーザーがターゲットリソースにアクセスしようとすると、フィルタは、2 つのリクエスト属性を設定した後にそのリクエストをエラーページに転送します。ユーザーにターゲットリソースへのアクセスが承認されている場合、フィルタは、チェーン内の次のフィルタにリクエストを渡します。実際には、チェーン内の各フィルタは、リクエストおよびレスポンスオブジェクトをデコレートするか、ラップします。

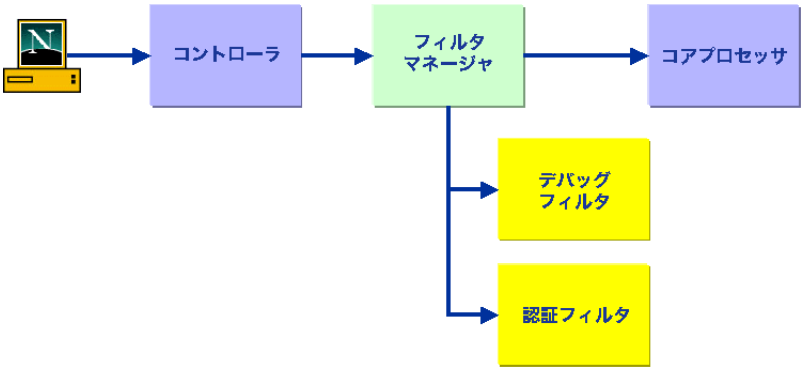
```
...
public class AuthFilter extends GenericFilter {

 public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
  System.out.println("AuthFilter の開始 ");
  HttpServletRequest req = (HttpServletRequest)request;
  String authorization = req.getHeader("Authorization");
  if (authorization != null) {
   System.out.println(" 承認されたユーザー :" + req.getRemoteUser() + " 承認
        されました ");
   chain.doFilter(request,response);
  } else {
   System.out.println(" 承認されなかったユーザー:" + req.getRemoteUser()  +
        " 拒否されました ");
   request.setAttribute("unauth_user", req.getRemoteUser());
   request.setAttribute("req_resource", req.getRequestURI());
   RequestDispatcher rd = request.getRequestDispatcher("/Error.jsp");
   rd.forward(request, response);
  }
  System.out.println("AuthFilter の終了 ");
 }
}
```

非デコレータマイクロパターン

フィルタオブジェクトは、非デコレータとして、または、その基本スコープ外には他に仕事を持たないオブジェクトとして実装できます。この場合、フィルタは、フロー制御ロジックを持たず、フィルタの機能だけを実装します。フィルタマネージャは、フィルタチェーンの集中アクセスポイントとして機能し、その独自のフロー制御ロジックセットに基づいてリクエストを転送します。

次の図にこの戦略を示します。



非 Decorator Filter のサンプル

次のコードは、ロギング機能を提供する非 Decorator Filter を示しています。アプリケーションを再コンパイルせずに、このフィルタをフィルタチェーンに追加したり、フィルタチェーンから削除したりすることができます。このフィルタは、標準 J2EE コンポーネントを使用するため、任意の Web アプリケーションに埋め込むことができます。

```
package jrunsamples.filters;

import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class HeaderFilter extends GenericFilter {
 public void doFilter(ServletRequest req, ServletResponse resp,
        FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {
  chain.doFilter(req, resp);
  HttpServletRequest request = (HttpServletRequest)req;
  System.out.println("****** リクエストヘッダー ******");
  System.out.println(" リクエスト :" + request.getRequestURI());
  Enumeration headers = request.getHeaderNames();
```

```
  if (headers == null) {
  } else {
   while (headers.hasMoreElements()) {
    String name = (String) headers.nextElement();
    String value = request.getHeader(name);
    System.out.println(name + "=" + value);
   }
  }
  HttpServletResponse response = (HttpServletResponse)resp;
  System.out.println("****** レスポンス情報 ******");
  String charencode = response.getCharacterEncoding();
  int bufsize = response.getBufferSize();
  System.out.println("文字エンコード：" + charencode);
  System.out.println("バッファーサイズ：" + bufsize);
 }
}
```

Service to Worker パターン

このセクションでは、12 ページの「パターンテンプレート」で定義されている形式の Service to Worker デザインパターンについて説明します。

問題

Service to Worker パターンは、Front Controller パターンと View Helper パターンの問題を解決します。Front Controller パターンと View Helper パターンは、ビューへのアクセスを直接扱ったり、ビジネスロジックとプレゼンテーションロジックが混在したりする場合に対応します。

Service to Worker パターンは、次のコンポーネントを組み合わせたマクロパターンです。

- ディスパッチャーコンポーネント
- Front Controller パターンコンポーネント
- View Helper パターンコンポーネント

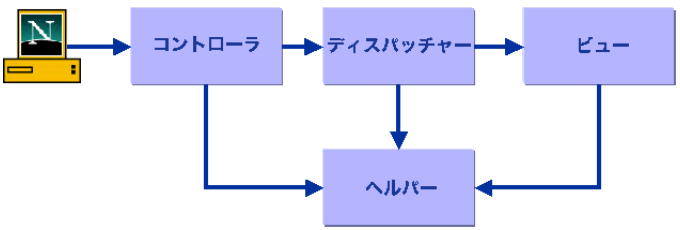
解決策

Service to Worker パターンの問題の解決策は、コントローラコンポーネントとビューコンポーネント間にディスパッチャーコンポーネントを追加することです。

コントローラは、コンテンツ取得をディスパッチャーに委ねるか、またはヘルパークラスに直接委ねます。コントローラは、ほとんどのリクエストのエントリポイントであり、セキュリティ、ロギング、ステート管理などのさまざまなサービスを管理します。これらのサービスの一部は、ヘルパークラスを介してビジネスロジックを呼び出すことが必要になります。

ディスパッチャーはリクエストをビューに送信します。このコンポーネントはコントローラコンポーネントから分離されているため、コードのモジュール化が行いやすくなっています。場合により、ディスパッチャーはヘルパークラス内のビジネスロジックを呼び出すことがあります。

次の図にこの戦略を示します。



ディスパッチャーコンポーネントはコントローラ内部にカプセル化できます。

戦略

コントローラは、単独のサーブレットか、またはサーブレットとサーブレットフィルタを組み合わせたものになります。ディスパッチャーは通常、サーブレットとして実装されます。

Dispathcer View パターン

このセクションでは、12 ページの「パターンテンプレート」で定義されている形式の Dispathcer View デザインパターンについて説明します。

問題

Dispathcer View パターンは、Front Controller パターンと View Helper パターンの問題を解決します。Front Controller パターンと View Helper パターンは、ビューへのアクセスを直接扱ったり、ビジネスロジックとプレゼンテーションロジックが混在したりする場合に対応します。

Dispathcer View パターンは、次のコンポーネントを組み合わせたマクロパターンです。

- ディスパッチャーコンポーネント
- Front Controller パターンコンポーネント
- View Helper パターンコンポーネント

Dispathcer View パターンを構成するコンポーネントは、Service to Worker パターンを反映します。ただし、これらのコンポーネントが担当するロールは互いに少しずつ異なり、フロー制御ロジックもさまざまです。

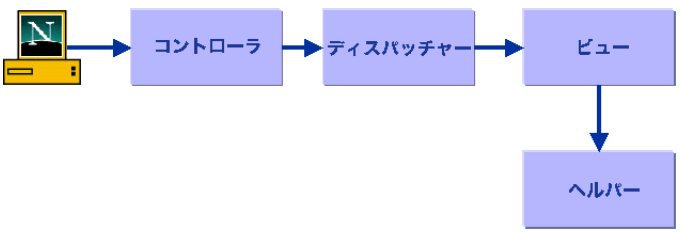
解決策

Dispathcer View パターンにおけるディスパッチャーコンポーネントのロールは、最低限のことしか行いません。通常は、コントローラからの入力に基づいたリクエストの転送を担当します。ヘルパークラスにアクセスしてビューを指定し、ビジネスサービスにリクエストを転送したり、ビジネスサービスを実行したりすることはありません。

Service to Worker パターンでは、ヘルパークラスへのアクセスはコントローラ、ディスパッチャー、および ビュー層で実行できます。また、ヘルパークラスへのアクセスは、ビューが Dispathcer View パターンで処理される場合にのみ実行できます。

このパターンは、基本的な Web アプリケーション、または複雑なビジネスサービスの必要条件 (セキュリティ管理) などを伴わない Web アプリケーションの一部分に適しています。

次の図にこの戦略を示します。



Struts について

Struts は、MVC デザインパターンを実装したり、MVC を実装するためのフレームワークを提供したりする、Apache Jakarta プロジェクトのオープンソースベースの取り組みです。Struts は、次のコンポーネントで構成され、これらのコンポーネントによって Web アプリケーション内部でフレームワークを使用できるようになります。

コンポーネント	説明
カスタムタグ ライブラリ	(ビュー) 国際化対応、フォーム検証、およびさまざまなアクションクラスの処理のためのカスタム JSP タグが含まれています。ビューは通常、HTML および JSP で構成されます。
コントローラ サーブレット	(コントローラ) `ActionServlet` クラスとして実装されます。 リクエストおよびレスポンスを転送する ActionMappings を使用してコントローラを設定します。Struts の中核です。
アクションクラス	(モデル) コントローラからのリクエストを受け取り、ビジネスロジックを実行するオブジェクトを呼び出します。 JavaBeans、EJB、および他のオブジェクトのステートを直接変更できます。ただし、MVC パターンの厳密な解釈では、このような用途は意図されていません。 アクションクラスの例としては `ActionForm` があります。

JRun への Struts のインストール

Struts フレームワークを使用する場合は、コンポーネントを使用する Web アプリケーションごとに Struts のコピーを個別にインストールする必要があります。Web アプリケーションをコンパイルする場合は、各 Web アプリケーションの WEB-INF/classes ディレクトリ内にローカル struts.jar を含める必要があります。

メモ：struts.jar を JRun サーバーのクラスパスに追加しないでください。追加すると、さまざまなサーバーの Web アプリケーションがその JAR ファイルを使用しようとするときにエラーが発生します。詳細については、30 ページの「その他の情報リソース」に記載されている Struts ドキュメントへのリンクにアクセスしてください。

JRun サーバーに Struts フレームワークをインストールするには

1 http://jakarta.apache.org/struts/index.htmlから最新の Struts バイナリをダウンロードします。

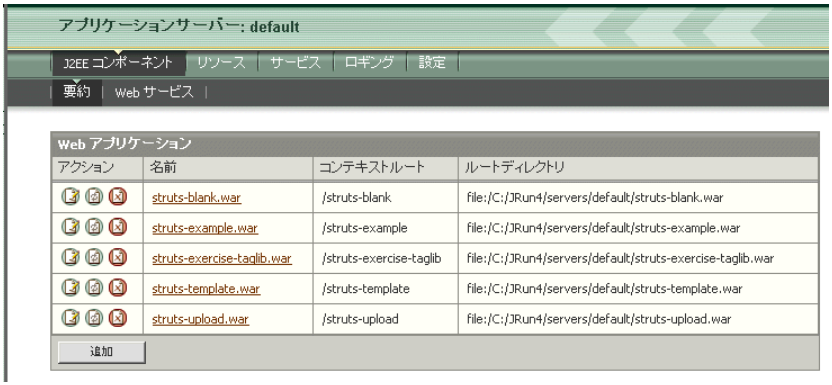
2 圧縮ファイルを解凍します。

3 */jakarta_struts_dir*/webapps ディレクトリから JRun サーバーのルートディレクトリに WAR ファイルをコピーします。

JRun によってこれらの Web アプリケーションが自動的にデプロイされます。

4 デプロイされた Struts アプリケーションを表示するには、JMC で JRun サーバーの [J2EE コンポーネント] パネルを開きます。

[J2EE コンポーネント] パネルが表示されます。



これらの各アプリケーションは独自のコンテキストルートにマッピングされます。たとえば、サンプル JRun サーバー上の Struts ドキュメントアプリケーションにアクセスするには、ブラウザでリクエストを次のように指定します。

```
http://localhost:8200/struts-documentation/
```

この例では、/struts-documentation がコンテキストルートです。ポート番号は使用しているサーバーと異なる場合があります。

5 *jakarta_struts_dir*/lib/struts.jar ファイルを、Web アプリケーションの WEB-INF/classes ディレクトリにコピーします。

メモ: このファイルは、Struts を使用する各 Web アプリケーションの WEB-INF/classes ディレクトリに含めます。

6 タグライブラリディスクリプタ (TLD) ファイルを *jakarta_struts_dir*/lib から該当ディレクトリにコピーします。JSP 仕様では、TLD ファイルを、META-INF/ ディレクトリ内の Web アプリケーションの WAR ファイルに保管するように推奨しています。

Web アプリケーションのコンパイル

Struts コンポーネントを使用する Web アプリケーションをコンパイルする場合は、必ず次の手順に従ってください。

- struts.jar ファイルをクラスパスに含めます。
- jdbc2_0-stdext.jar をクラスパスに含めます。一部の Struts コンポーネントは JDBC 2.0 オプションパッケージバイナリ (旧 Standard Extensions パッケージ) を使用します。
- WEB-INF/web.xml ファイル内で、コントローラサーブレットの `servlet` および `servlet-mapping` 要素を定義します。

  Struts サンプルアプリケーション内のサンプルのサーブレット設定を使用することができます。一部のリストを次に示します。

```
<!-- アクションサーブレットの設定 -->
 <servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet
     </servlet-class>
  ...
  <init-param>
   <param-name>detail</param-name>
   <param-value>2</param-value>
  </init-param>
 ...
</servlet>

<!-- アクションサーブレットのマッピング -->
<servlet-mapping>
 <servlet-name>action</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

- 使用する Struts タグライブラリごとに、WEB-INF/web.xml ファイル内にタグライブラリ定義を追加します。たとえば、次のように指定します。

```
<taglib>
 <taglib-uri>/WEB-INF/app.tld</taglib-uri>
 <taglib-location>/WEB-INF/app.tld</taglib-location>
</taglib>
```

- struts-config.xml ファイルを作成し、これを WEB-INF ディレクトリに保管します。struts-config.xml ファイル内で、アクションクラスに対するリクエストのマッピングを定義します。このファイルの作成方法の詳細については、Struts サンプルアプリケーションの struts-config.xml ファイルを参照してください。
- Struts カスタムタグを使用する各 JSP ページの上部に次のディレクティブを追加します。

  ```
  <%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="struts-bean" %>
  ```

  使用する TLD ファイルごとに、**taglib** ディレクティブを含める必要があります。接頭辞は任意の値に設定できます。

詳細については、30 ページの「その他の情報リソース」の Struts ドキュメントのリンクにアクセスしてください。

その他の情報リソース

この章では、基本的なデザインパターンの一部について紹介しました。デザインパターンの詳細については、次の情報リソースを参照してください。

リソース	格納場所
Core J2EE Patterns	Depak Alur、John Crupi、Dan Malks、Prentice Hall
Design Patterns	Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides (GoF の本としてよく知られています)、Addison-Wesley Publishing Company
Sun パターン概要	http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/
Sun パターンカタログ	http://java.sun.com/blueprints/patterns/j2ee_patterns/catalog.html
Struts プロジェクト	http://jakarta.apache.org/struts/index.html
Struts チュートリアル	http://www.computer-programmer.org/articles/struts/
Design Patterns Java Companion	http://www.patterndepot.com/put/8/JavaPatterns.htm
Thinking in Patterns in Java (プレリリース)	http://www.mindview.net/Books/TIPatterns/

第 3 章
国際化対応とローカリゼーション

この章では、Web アプリケーションの国際化対応とローカリゼーションの概念を紹介します。

目 次

国際化対応とローカリゼーションについて

国際化対応は、アプリケーションで複数の言語をサポートすることを意味する一般用語です。国際化対応という用語は、しばしば I18N と省略されますが、これは Internationalization (国際化対応) という単語が I の文字に続いて 18 文字あり、それに続く N で終わっていることに由来します。

ローカリゼーションは、アプリケーションで特定の言語や地域をサポートする処理です。ローカリゼーションという用語は、しばしば L10N と省略されますが、これは Localization (ローカリゼーション) という単語が L の文字に続いて 10 文字あり、それに続く N で終わっていることに由来します。

ロケールは、ある地域用にローカライズされた情報のセットです。たとえば、ロケールには米国があります。ロケールでは、使用するコンテンツの原語と文字セットを定義します。コンテンツと文字セットは両方ともプログラムで定義します。アプリケーションを正しくローカライズするには、Web アプリケーションの次のコンポーネントに存在する、地域的な違いに注意する必要があります。

- ラベル
- メッセージ
- オンラインヘルプのテキスト
- 日付と時刻の形式
- 数の形式
- 通貨
- 単位

特定のロケールを対象にする場合は、Web アプリケーションの次の側面も考慮する必要があります。

- ページレイアウト
- ナビゲーション要素
- カラー
- グラフィック

たとえば米国では、多くの場合、赤色は危険を暗示します。しかし、中国では赤は繁栄を意味します。米国をロケールとする Web アプリケーションでは、赤色で警告メッセージを表示しますが、中国の場合は、中国文化で警告を意味する色を選びます。

このセクションでは、Web アプリケーションをローカライズするテクニックを紹介します。ここでは、サーブレット API で利用できるロケール対応クラスの使用方法を示します。

Java によるローカリゼーション

Java は、I18N と L10N をサポートするようにデザインされており、アプリケーションのローカライズには次のクラスが役立ちます。

- `java.util.Locale`
- `java.text.DateFormat`
- `java.text.NumberFormat`
- `java.util.ResourceBundle`
- `java.io.InputStreamReader` と `java.io.OutputStreamWriter`

さらに、サーブレット API には、リクエストとレスポンスのオブジェクトをローカライズに対応させるメソッドがあります。JSP 標準タグライブラリ (JSTL) には、ローカライズされた JSP の開発を容易にするカスタムタグがあります。

このセクションでは、Local オブジェクトと、それが使用する言語コードと国コードの基本を説明します。J2EE でのローカリゼーションサポートの詳細については、45 ページの「リソース」を参照してください。

ロケールの理解

`java.util.Locale` クラスのインスタンスは直接使用されませんが、他のクラスによって使用され、地域情報と言語設定を提供します。次の例に示すように、ロケール情報を使用するクラスの 1 つには DateFormat クラスがあります。

```
DateFormat df  = DateFormat.getDateInstance(DateFormat.LONG,
        new Locale("ja","JP"));
String date = df.format(new Date());
```

この例では、日付を日本の日付形式で設定します。

Local オブジェクトをインスタンス化するときは、次の例のように、言語コードを表わす小文字 2 つと国コードを表わす大文字 2 つを組み合わせて指定します。

```
java.util.Locale locale = new Locale("ja","JP");
```

この例で、ja は言語コードで、JP は国コードです。言語コードと国コードを指定する理由は、一部の国々では複数の公用語が使用されているからです。たとえば、スイスには公用語が 4 つあります。

使用可能なロケールの表示

Java には、次のように、ロケール関連の情報を取得するメソッドが数多くあります。

- `Locale.getAvailableLocales`
- `Locale.getLanguage`
- `Locale.getCountry`
- `DateFormat.getDateTimeInstance`
- `NumberFormat.getNumberInstance`
- `NumberFormat.getCurrencyInstance`

次のコード例は、JVM を実行中のシステムで使用できるロケール情報をリストします。

```
<%@ page import="java.util.*, java.text.*" %>
<%@ page contentType="text/html; charset=UTF-8" %>

...
<% Locale[] availableLocales = Locale.getAvailableLocales(); %>
<TABLE>
<% for(int i=0; i < availableLocales.length; i++) {%>
<tr>
  <td><%= availableLocales[i].getLanguage() %></td>
  <td><%= availableLocales[i].getCountry() %></td>
  <td><% DateFormat df =
        DateFormat.getDateTimeInstance(DateFormat.FULL,
        DateFormat.FULL, availableLocales[i]); %>
        <%= df.format(new Date()) %>
        </td>
  <td><% NumberFormat nf =
        NumberFormat.getNumberInstance(availableLocales[i]); %>
        <%= nf.format(123456.78) %></td>
  <td><% nf = NumberFormat.getCurrencyInstance(availableLocales[i]);
        %><%= nf.format(1234567.89) %></td>
</tr>
<% } // for() ループの終わり %>
</TABLE>
...
```

言語コードの理解

言語コードは 2 つの小文字で言語を表し、ISO 639 標準で定義されています。

一般的な言語コードは次のとおりです。

- 英語：en
- スペイン語：es
- フランス語：fr
- 中国語：zh
- 日本語：ja
- 朝鮮語：ko

言語コードのリストについては、http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt をご覧ください。

国コードの理解

国コードは 2 つの大文字で国を表し、ISO 3166 標準で定義されています。

一般的な国コードは次のとおりです。

- 米国：US
- 英国：GB
- 中国：CN
- 日本：JP
- 韓国：KR

国コードのリストについては、http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html をご覧ください。

文字セットとエンコードの理解

文字セットは、バイトを文字にマッピングするテーブルです。Java では内部的に、すべてのコンテンツをエスケープされた ASCII Unicode 文字セットで処理します。Java クラスをコンパイルするとき、Java コンパイラは、クラスのコンテンツをシステムの文字セットからエスケープされた ASCII Unicode 文字セットに変換します。

各文字セットには、バイトシーケンスを定義する独自のエンコード / デコードのアルゴリズムがあります。エンコードは、リクエストの作成時にクライアントサイドの Web ブラウザで、また、レスポンスの作成時にサーバーサイドの Java で実行されます。開発者は、エンコードメカニズムやデコードメカニズムを提供する必要はなく、クライアントの Web ブラウザによって使用される文字セットを設定、取得できます。

ブラウザは、HTTP `Content-Type` ヘッダーで指定された文字セットを使用して、画面上でバイトを文字に変える方法を決定します。

一般的な文字セットは次のとおりです。

- ISO-8859-1 (英語：Latin-1)
- Big5 (繁体中国語)
- Shift_JIS (日本語)
- ECU-KR (韓国語)

Java では、文字セットに大文字と小文字の区別がないので、値 Shift_JIS と値 shift_jis は等価です。

1 つのページで複数の文字セットを使用するには、UTF-8 文字セットのエンコードを使用します。

一般的な文字セットのリストについては、http://www.eleves.ens.fr:8080/home/madore/computers/unicode/cstab.html をご覧ください。

Internet Explorer バージョン 5 以降でサポートされている文字セットのリストについては、http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/reference/charsets/charset4.asp をご覧ください。

Java でサポートされているエンコード文字セットの詳細については、http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html をご覧ください。

サーブレットでの文字セットの設定

Content-Type HTTP ヘッダーによって、レスポンスオブジェクトによってエンコードされる文字セットが定義されます。`Content-Type` ヘッダー内の定義の形式は、`charset=character_set` です。このヘッダーでは MIME タイプも定義されますが、それに関してはこのセクションでは説明しません。

クライアントのブラウザはヘッダーの `charset` を読み取って、クライアントの画面でページを正しくレンダリングできるように、レスポンスのエンコードを判断します。クライアントは、`charset` のついたレスポンスを受け取ったとき、文字セットをデコードできる必要があります。正しく出力するには、ブラウザがその文字セットをサポートしていること、および、システムが適切なフォントにアクセスできる必要があります。

サーブレット API には、`Content-Type` HTTP ヘッダーを設定する、次のような便利なメソッドが用意されています。

- `response.setContentType`
- `response.setLocale`

このセクションでは、これらの方法について説明します。

次の **setHeader** を使用して、HTTP ヘッダーを明示的に変更することもできます。

- response.setHeader("Content-Type", *contenttype*)
- response.setHeader("Content-Language", *languagecode*)

レスポンスオブジェクトで適切なエンコードを使用するには、コンテンツタイプを定義した後で、**getWriter** を呼び出す必要があります。PrintWriter では、デフォルトで ISO-8859-1 文字セットが使用されます。

1 つのページで複数の文字セットを使用するには、UTF-8 文字セットのエンコードを使用します。

setContentType の使用

HTTP **Content-Type** ヘッダーによって文字セットが設定されたのと同様に、**setContentType** メソッドによってレスポンスの MIME タイプが設定されます。**charset** によって、クライアントのブラウザでのページの表示に使用するデコードアルゴリズムの種類が、クライアントのブラウザに示されます。

次の例では、**Content-Type** ヘッダーを日本語文字セットに設定します。

```
response.setContentType("text/html; charset=Shift_JIS");
PrintWriter out = response.getWriter();
```

setLocale の使用

setLocale メソッドによって、レスポンスの文字セット、HTTP **Content-Language** ヘッダー、および HTTP **Content-Type** ヘッダーが設定されます。**Content-Language** ヘッダーは、しばしばブラウザに無視されます。**setLocale** を使用すると、**setContentType** を使用する場合よりも、より詳細に言語設定を管理できます。その場合は、Locale オブジェクトを用意する必要があります。

次の例では、レスポンスオブジェクトの文字セットを Shift_JIS に定義します。

```
response.setContentType("text/html");
response.setLocale(new Locale("ja","")); //ja のデフォルト文字セットを
        Shift_JIS に設定します
PrintWriter out = response.getWriter();
```

JSP での文字セットの設定

次の例に示すように、**page** ディレクティブの **contentType** 属性を使用して、JSP で文字セットを定義します。

```
<%@ page contentType="text/html; charset=Shift_JIS"
        pageEncoding="Shift_JIS" %>
```

1 つのページで複数の文字セットを使用するには、次の例に示すように、UTF-8 文字セットのエンコードを使用します。

```
<%@ page contentType="text/html; charset=utf-8" %>
```

非デフォルト文字セットでのコンパイル

デフォルトではない文字セットで JSP やサーブレットを作成する場合は、コンパイラでクラスファイルが正しいエンコードに変換されるように特に注意する必要があります。このセクションでは、サーブレットと JSP をコンパイルするときのエンコードタイプの変更方法を説明します。

サーブレットのコンパイル

Java クラスをコンパイルするときは、-encoding オプションとそれに続けてエンコードタイプを追加することにより、クラスのエンコードを設定します。-encoding オプションを指定しない場合、Java コンパイラはシステムの現在のエンコードをデフォルトとして使用します。次の行は、日本語エンコードでサーブレットをコンパイルする encoding オプションを示しています。

```
%> javac -encoding Shift_JIS -classpath c:/jrun4/lib/jrun.jar
        MyJapaneseServlet.java
```

デフォルトのエンコードタイプは、file.encoding システムプロパティで定義されます。次のコードで示すように、getProperties メソッドを使用して、システムプロパティをチェックできます。

```
...
Properties properties = System.getProperties();
Enumeration keys = properties.propertyNames();
while(keys.hasMoreElements()) {
  String key = (String) keys.nextElement();
  System.out.println(key + " = " + properties.getProperty(key));
}
...
```

JSP のコンパイル

JSP では、page ディレクティブの pageEncoding 属性を使用して、コンパイル時に使用するエンコードタイプが決定されます。

次の例は、ページを日本語文字セットでエンコードします。

```
<% page contentType="text/html; charset=UTF-8" pageEncoding="EUC-JP" %>
```

ページのエンコードは、ページがエディタで保存されたときのエンコードタイプに一致する必要があります。

ロケールの使用

クライアントのブラウザがサポートする言語のリストを取得することは、サーブレットや JSP をローカライズするときの重要な手順です。クライアントは、各リクエストで HTTP Accept-Language ヘッダーを送信します。ターゲットのサーブレットや JSP は、リクエスト内のヘッダーの値を読み込み、これを使用して、コンテンツタイプを設定し、カスタムレスポンスを生成できます。

サーブレット API には、クライアントのロケールを取得するための次の便利なメソッドがあります。

HttpServletRequest オブジェクトには、次のメソッドがあります。

- getLocale
- getLocales

getLocale メソッドは Locale オブジェクトを返します。このロケールは、クライアントのブラウザの設定で定義されます。この方法は、Accept-Language HTTP ヘッダーの値を取得するには便利です。getLocales メソッドは、重要度の降順でロケールのリストを返します。

HttpServletResponse オブジェクトには、レスポンスに割り当てられた local オブジェクトを返す getLocale メソッドもあります。

次の例は、getLocale を呼び出してリクエストのロケールを取得し、ダイナミックに文字セットを設定し、レスポンスのコンテンツを条件付きで返します。

```
...
<%
    String charset = "";
   Locale locale = request.getLocale();
    if(locale.getLanguage().equals("ko")) {
      charset = "EUR_KR";
    } else {
      charset = "ISO-8859-1";
    }
   response.setContentType("text/html; charset=" + charset);
   response.setLocale(locale);
    out.println("<p>charset :" + charset + "</p>");
%>
<%@ page import="java.util.*" %>
<% if(locale.getLanguage().equals("ko")) { %>
      // 韓国語のリクエストに特有のレスポンス
<% } else { %>
      // その他のすべてのレスポンス
<% } %>
...
```

JSP では、page ディレクティブはスタティックメソッドです。つまり、これを使用して文字セットをダイナミックに変更することはできません。その結果、前の例では、スクリプトレットで response.setContentType メソッドを使用して、文字セットを識別しています。

サーブレットや JSP での文字セットの使用については、36 ページの「文字セットとエンコードの理解」を参照してください。

外国語フォームの送信処理

Web サイトは、世界中のあらゆるブラウザからアクセスされます。クライアントはフォームを送信するとき、ブラウザで設定されたエンコードタイプで送信フォームをエンコードします。結果として、すべてのフォームが同じエンコードタイプで送信されるということは当てにできません。Internet　Explorer では、[表示] > [エンコード] を選択して、エンコードタイプを表示できます。

このセクションでは、リクエストデータをデコードする次のメソッドを説明します。

- エンコード対応の String コンストラクタの使用
- `setCharacterEncoding` の使用

リクエストのエンコードタイプの取得

ブラウザで ISO-8859-1 以外の文字セットが使用される場合、リクエストの `Content-Type` ヘッダーのエンコード文字セットが送信されることを 仮定できます。`Content-Type` ヘッダーから文字セットを取得するには、request オブジェクトの `getCharacterEncoding` メソッドを使用します。この値を使用してフォームデータをデコードし、正しい文字セットを使用してレスポンスで作業できます。

たとえば、クライアントが EUC-JP を使用したフォームを送信し、リクエストの `Content-Type` ヘッダーが `Shift_JIS` に設定されている場合、処理する側のサーブレットは、リクエストがエンコードされた方法やそれを正しくデコードする方法を判断できます。

エンコード対応の String コンストラクタの使用

サーバーは、クライアントのブラウザで使用されたエンコードタイプを無視して、デフォルト文字セット ISO-8859-1 (Latin-1 とも呼ばれます) を使用して、フォームデータをデコードします。EUC-JP 文字セットを使用してフォームが送信された場合、JRun はエンコード文字セットと異なるデフォルトのデコード文字を使用してリクエストデータをデコードすることになるので、リクエストデータは壊れてしまいます。

デコードが正しく行われるように、Java にはエンコードタイプを設定できる次の 2 つの String コンストラクタがあります。

- `public String (byte[] bytes, String encoding)`
- `public String (byte[] bytes, int offset, int length, String encoding)`

次の例は、文字エンコードタイプと元のフォーム入力の実際のバイト配列を取得して、正しいエンコードタイプで文字列を作成します。

```
...
String encoding = request.getCharacterEncoding();
String httpDefaultEncoding = "ISO-8859-1";
String corruptData = request.getParameter("name"); //データは、デフォルトで
        ISO-8859-1 を使用してデコードされます。
String correctData = new
        String(corruptData.getBytes(httpDefaultEncoding), encoding);
...
```

現在ほとんどのブラウザでは、リクエストで使用されているエンコードタイプにかかわらず、HTTP Content-Type リクエストヘッダーは設定されません。結果として、getCharacterEncoding メソッドを呼び出すと、通常は null 値が返されます。

このタイプの String コンストラクタは、エンコードタイプがサポートされない場合は、UnsupportedEncodingException が返されます。

setCharacterEncoding の使用

クライアントの文字エンコードを判断する断定的なソリューションはありませんが、サーブレット API には、リクエストオブジェクトのエンコードを設定する次に示す便利なメソッドがあるため、残りのリクエストデータは正しく処理されます。
request.setCharacterEncoding

このメソッドを使用してリクエストにエンコードタイプを割り当てることができるので、その後のこのリクエストオブジェクトへの呼び出しでは、リクエストのデータを正しくデコードできます。setCharacterEncoding を使用すると、リクエストデータがデフォルトのエンコードから他のエンコードに変換されるのを防ぐことができます。

リクエストのエンコードは、getParameter や getReader を呼び出す前に設定する必要があります。

次の例では、Shift_JIS でエンコードされたフォームの日本語パラメータを標準の getParameter メソッドで読めるように、エンコードタイプのサーブレットを設定します。

```
request.setCharacterEncoding("Shift_JIS");
String username = request.getParameter("username");
```

英字以外の文字の表示

文字セットを設定する代わりに特殊文字のシーケンスを使用することにより、クライアントのブラウザへのレスポンスに英字以外の文字を送信できます。このセクションでは、HTML エンティティ と Unicode シーケンスの使用方法を説明します。

HTML エンティティの使用

HTML 2.0 には、クライアントに特殊文字を送信できる HTML エンティティと呼ばれる文字シーケンスが導入されました。

HTML エンティティは、アンパサンド (&) で始まりセミコロン (;) で終わります。次の HTML エンティティでは、括弧 (< >) が表示されます。

```
&lt;HTML&gt;
```

この例は、ブラウザで次のように表示されます。

```
<HTML>
ブラウザは、特殊文字を HTML とは解釈しません。
```

英字以外の文字を HTML エンティティとして表現する場合、その文字を表わす言語のエンティティを使用する必要があります。次の例では、文字 A を 3 種類の言語で表示する HTML エンティティを使用します。

```
...
out.println("英語： &#65;");
out.println("ギリシャ語： &#937;");
out.println("キリル語： &#1105;")
...
```

HTML エンティティのリストについては、http://www.ramsch.org/martin/uni/fmi-hp/iso8859-1.html をご覧ください。

Unicode シーケンスの使用

HTML 4.0 では、Unicode シーケンスにマッピングされるように HTML エンティティが拡張されています。Unicode コンソーシアムは、ほとんどすべての言語を 1 つのテーブルにマッピングする Unicode 文字セットを定義しています。Unicode は、すべての文字が HTML エンティティの定義と同じ 2 バイトのシーケンス (1 バイトの ASCII 文字ではない) として定義されています。Unicode 2.1 標準では、約 40,000 種類の文字のエンコードが定義されています。

Unicode シーケンスは、エスケープ u (¥u) と、それに続いて文字を表す 2 バイトで構成されます。次の Unicode シーケンスでは括弧 (< >) が表示されます。

```
¥u003CHTML¥u003E
```

この例は、ブラウザで次のように表示されます。

```
<HTML>
```

ブラウザは、特殊文字を HTML とは解釈しません。

次のサーブレットコードは、Unicode を使用して日本語で Hello World をプリントします。

```
out.println("<b>¥u30cf¥u30ed¥u30fc¥u30ef¥u30fc¥u30eb¥u30c9</b>");
```

次の JSP コードは、英語、日本語、韓国語で Hello World をプリントします。

```
<h2><%= "¥u0048¥u0065¥u006C¥u006C¥u006F
        ¥u0057¥u006F¥u0072¥u006C¥u0064" %></h2>
<h2><%= "¥u30CF¥u30ED¥u30FC ¥u30EF¥u30FC¥u30EB¥u30C9" %></h2>
<h2><%= "¥uD5EC¥uB85C ¥uC6D4¥uB4DC" %></h2>
```

Unicode シーケンス文字のリストについては、http://www.unicode.org をご覧ください。

メモ：一部の文字セットを表示するには、追加言語サポートをコンピュータにインストールする必要があります。詳細については、ご使用のオペレーティングシステムのドキュメントを参照してください。

ResourceBundle の使用

ResourceBundles を使用することにより、Web アプリケーションでサポートしている各ロケール用のソースファイルを作成できます。これらのファイルには、各ロケール専用のテキストや画像などのオブジェクトが含まれます。

ResourceBundles は、**java.util** パッケージのアブストラクトクラスであり、次のサブクラスの 1 つとして実装する必要があります。

- **ListResourceBundle**　リソースバンドルにイメージなどのオブジェクトへのポインタを保管するために使用します。Java クラスを使用してリソース情報を保管します。
- **PropertyResourceBundle**　リソースバンドルにテキストを保管するために使用します。プレーンテキストのプロパティファイルを使用して生のデータを保管します。

Web アプリケーションで多くのローカライズされたテキスト情報が必要な場合は、クラスファイル (PropertyResourceBundle に必須) ではなくプロパティファイル (ListResourceBundles に必須) を使用してデータを保管する方が簡単です。

JRun は、次の規則を使用して、ファイル名に基づいて取り込むリソースバンドルを決定します。

ClassName_languagecode_COUNTRYCODE.class
ClassName_languagecode_COUNTRYCODE.properties

languagecode は、**en** などのデフォルトのシステムロケールの場合以外では必須です。COUNTRYCODE はオプションです。

ロケールを付けて **getBundle** メソッドを呼び出した場合、JRun では適切なリソースクラスまたはファイルが自動的に取り込まれます。たとえば、ロケールが日本語に設定されている場合、JRun では MyResources_ja.class または MyResources_ja.properties が取り込まれます。

Web アプリケーションでリソースバンドルを使用するには、クライアントロケールを付けて **ResourceBundle.getBundle** メソッドを呼び出します。次に、次の例に示すように、リソースバンドルの適切なメソッドを呼び出します。

```
...
ResourceBundle bundle =
        ResourceBundle.getBundle("Resources",request.getLocale());
<html>
<head><title><%= bundle.getString("title") %></title></head>
  <body>
  <%-- ResourceBundle に基づいてすべての内容を表示します --%>
  <h2><%= bundle.getString("hello_message") %></h2>
  </body>
</html>
...
```

リソースバンドルのプロパティファイルの内容は次のとおりです。

```
title=Morning Greeting
hello_message=Good Morning
```

例については、/samples/SERVER-INF/lib/jrun/samples/security ディレクトリ内の SampleJDBCLoginModule を参照してください。

リソース

次の表は、Web アプリケーションの国際化対応についての詳しい情報が得られるリソースをリストします。

リソース	格納場所
ISO (International Organization for Standardization：国際標準化機構)	http://iso.org
Sun internationalization tutorial	http://java.sun.com/docs/books/tutorial/i18n/
Java Internationalization and Localization Toolkit	http://java.sun.com/products/jilkit/
Web Globalization FAQ	http://www.sun.com/developers/gadc/faq/web.html
International Technical Communication by Nancy Hoft	John Wiley & Sons

第 4 章
JRun と XML

XML は J2EE と密接に結合されています。この章では、Java アプリケーションサーバーに XML を組み込む方法、JRun の内部で XML ツールがどのように使用されるか、さらに XML を生成して Web アプリケーションに変換する簡単な方法について説明します。

目 次

XML の概要

XML は J2EE に不可欠な要素です。XML は、J2EE アプリケーション開発の次の領域で利用されています。

- Web アプリケーションのデプロイメントディスクリプタ (web.xml)
- タグライブラリディスクリプタファイル (TLD)
- bean デプロイメントディスクリプタ (ejb-jar.xml)
- 変換を使用することにより、さまざまなタイプのクライアントに対応するデータの複数ビュー
- SOAP および Web サービスの開発
- B2B (企業間取引) において XML ベースでやり取りするための専用 DTD

このセクションでは、J2EE のどの部分に XML が組み込まれているか、また JRun で XML がどのように使用されるかについて説明します。また、XML に関するその他の情報リソースも記載します。

Java の XML 拡張機能

Java には、Web アプリケーションで XML を使用するための多数の拡張機能が用意されています。次の表で、Java 言語の XML 拡張機能を説明します。

拡張機能	説明
JAXP	XML を処理するための JAVA API。JAXP は、実際に使用されている JAXP の実装元にかかわらず、標準 SAX、DOM、および XSLT API を Java で作成して使用するための共通インターフェイスを提供します。
JAXM	XML メッセージ送信のための Java API。JAXM は、アプリケーション間で非同期の XML ベースメッセージをやり取りするためのメカニズムを定義します。
JAXB	XML バインディングのための Java アーキテクチャ。JAXB は、Java オブジェクトを XML で記述したり、そのような XML 構造から Java オブジェクトを作成したりするためのメカニズムを定義します。
JAX-RPC	XML ベースのリモートプロセス通信のための Java API。JAX-RPC は、アプリケーション間で同期 XML ベースメッセージをやり取りするためのメカニズムを定義します。
JAXR	XML レジストリのための Java API。JAXR は、外部レジストリで利用可能なサービスをパブリッシュしたり、レジストリを調べてそれらのサービスを見つけたりするためのメカニズムを提供します。

XML 処理ツール

Java には、Web アプリケーションの XML ファイルを処理するためのツールが多数含まれています。次の表で、XML ファイルを処理するためのいくつかのテクノロジーを説明します。

テクノロジー	説明
XSLT	XML を変換するための XML スタイルシート言語 XSLT は、ある XML 形式のドキュメントを別の XML 形式に変換します。
SAX	Simple API for XML。SAX は、要素ごとに処理を実行するイベント駆動型の順次アクセスメカニズムです。SAX は、データレポジトリまたは Web に対して XML の読み書きを行います。
DOM	Document Object Model。DOM は、テキストノード、要素ノード、命令処理ノード、CDATA ノード、エンティティリファレンス、およびその他の種類のノードが混在する非常に単純なデータ構造体です。
JDOM	Java Document Object Model。JDOM は、"Java に最適化された " ドキュメントオブジェクトモデル API で、http://www.jdom.org に公開されています。JDOM は、SAX (Simple API for XML) や DOM (Document Object Model) などの既存の規格と高い親和性がありますが、これらの API のアブストラクションレイヤーや拡張機能ではありません。 JSR-102 で定義します。
DOM4J	Document Object Model for Java。DOM4J は、DOM に代わるオープンソースのオブジェクト指向 API です。
XPath	XML Path Language。XPath は、XSLT および XPointer の両方で使用されるようにデザインされた、XML ドキュメントの特定の部分を示すための言語です。
XPointer	XPointer は、XPath (XML Path Language) をベースにしており、XML ドキュメント内の位置を示すことができます。XPointer を利用すると、階層ドキュメント構造を調べて、要素タイプ、属性値、文字コンテンツ、相対位置などのさまざまなプロパティに基づいて、ドキュメント内の特定の部分を選択できます。
XDoclet	XDoclet は、拡張 JavaDoc Doclet エンジンであり、テンプレートエンジンを使用し、ソースコードや他のファイルに基づいてカスタム Javadoc ファイルを作成します。また、XDoclet は、jrun-web.xml および ejb-jar.xml ファイルを生成します。

JRun と XML

XML は JRun に不可欠な要素です。JRun は 次の方法で XML を利用します。

- HTML/JSP シンタックスマークアップからコンパイル済みサーブレットコードに JSP ページを変換する。
- Ant/build ファイルを使用して JRun インストーラを作成する。
- SAX および DOM を使用して、標準 J2EE Web アプリケーションの設定ファイルの解析、読み取り、および更新を行う。
- XMLScript ユーティリティを使用して、設定ファイルで XPath コマンドのサブセットを実行する。

次の表で、JRun に含まれている XML 処理ツールを説明します。

ツール	説明	格納場所
Crimson	SAX/DOM パーサー。SAX 2 および DOM 2 をサポートします。バージョン 1.1。 org.apache.crimson パッケージにあります。	<JRun のルートディレクトリ>/lib/jrun.jar
Xalan	XSLT プロセッサ。JAXP バージョン 1.1 を使用します。	<JRun のルートディレクトリ>/lib/jrun.jar
JDOM	DOM パーサー。バージョン Beta 7。	<JRun のルートディレクトリ>/lib/jrun.jar
XMLScript	XPath ベースの XML スクリプトユーティリティ **`jrunx.xml`** パッケージにあります。	<JRun のルートディレクトリ>/lib/jrun.jar
XPath	Apache の XPath 実装。	<JRun のルートディレクトリ>/lib/jrun.jar
XDoclet	カスタム JavaDocs および XML デプロイメントディスクリプタ作成ユーティリティ。	<JRun のルートディレクトリ>/lib/jrun-xdoclet.jar

詳細情報

次の表に、XML やそれに関連するテクノロジーを初めて利用する場合に役立つ Sun の Web サイトで公開されている各種リソース (英語) のリンクを掲載します。

リソース	アドレス
XML テクノロジーの概要	http://developer.java.sun.com/developer/technicalArticles/xml/JavaTechandXML/
XML 入門チュートリアル	http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/overview/index.html
DOM チュートリアル	http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/dom/index.html
SAX チュートリアル	http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/sax/index.html
XSLT チュートリアル	http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/xslt/index.html

JRun で使用する XML API およびツールの詳細については、77 ページの「リソース」を参照してください。

JRun の XML ファイル

JRun は、サーバーおよび Web アプリケーションの設定での XML ファイルの使用を定義した J2EE 標準に準拠しています。また、JRun は XML ファイルを使用してコンテナ環境を定義します。次のセクションでは、JRun で使用する J2EE 標準の XML ファイルおよび JRun 特有の XML サーバー設定ファイルについて説明します。

標準 J2EE XML ファイル

次の表で、Web アプリケーションのデプロイ時に使用される J2EE 標準の XML ファイルを説明します。

XML ファイル	説明	格納場所
application.xml	エンタープライズアプリケーション (EAR ファイル) の設定をモジュールレベルで定義します。モジュール定義、コンテキストルート、およびセキュリティ設定が含まれます。	<JRun のルートディレクトリ >/servers/< サーバー名 >/< アプリケーション名 >/META-INF
web.xml	Web モジュール (WAR ファイル) の設定をコンポーネントレベル (サーブレット、フィルタ、イベントハンドラ) で定義します。サーブレット定義、Web アプリケーションの初期化パラメータ、およびセキュリティ設定値が含まれます。	<JRun のルートディレクトリ >/servers/< サーバー名 >/< アプリケーション名 >/WEB-INF
ejb-jar.xml	EJB モジュール (EJB JAR ファイル) の設定をコンポーネントレベル (EJB) で定義します。EJB 定義、環境エントリ、アクセス許可レベル、およびセキュリティ設定が含まれます。	<JRun のルートディレクトリ >/servers/< サーバー名 >/< アプリケーション名 >/META-INF

JRun 特有の XML ファイル

次の表で、JRun コンテナ環境を設定するが、J2EE 標準には準拠していない XML ファイルを説明します。

XML ファイル	説明	格納場所
servers.xml	現在インストールされている JRun のサーバーがリストされています。	<JRun のルートディレクトリ >/lib
jrun-web.xml	JRun アプリケーションサーバーに特有の Web アプリケーション要素が含まれています。	<JRun のルートディレクトリ >/servers/< サーバー名 >/< アプリケーション名 >/WEB-INF
default-web.xml	構造は標準の web.xml ファイルと同じですが、設定は JRun サーバーのすべての Web アプリケーションに適用されます。	<JRun のルートディレクトリ >/servers/< サーバー名 >/server-inf
jrun-dtd-mappings.xml	XML の DOCTYPE パブリック ID および物理的な DTD ファイル間のマッピングを指定します。通常、DTD ファイルはリモートサーバーとの接続を介して処理されます。このファイルでマッピングを指定すると、DTD ファイルをローカルで読み取ることができるようになります。	<JRun のルートディレクトリ >/servers/< サーバー名 >/server-inf
jrun.xml	JRun サーバーのコアサービス設定を定義します。	<JRun のルートディレクトリ >/servers/< サーバー名 >/server-inf
jrun-jms.xml	ビルトイン JMS プロバイダを定義します。	<JRun のルートディレクトリ >/servers/< サーバー名 >/server-inf
jrun-resources.xml	J2EE リソースファクトリ (JDBC データソース、JMS プロバイダ、JavaMail セッション、および URL) を定義します。オブジェクトのカスタムプールも定義します。	<JRun のルートディレクトリ >/servers/< サーバー名 >/server-inf
jrun-users.xml	JRun サーバーのユーザーを定義します。	<JRun のルートディレクトリ >/servers/< サーバー名 >/server-inf
xdoclet.xml	XDoclet ユーティリティを設定します。	<JRun のルートディレクトリ >/lib
server-config.wsdd	Web サービスおよび Web サービスエンジンを定義します。	<JRun のルートディレクトリ >/servers/< サーバー名 >/< アプリケーション名 >/WEB-INF

JRun 設定ファイルの使用方法の詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

Web コンポーネントからの XML の生成

Web コンポーネントには柔軟性があるため、JSP およびサーブレットを使用して XML データの生成、利用、または変換を行うことができます。このセクションでは、Web コンポーネントから XML を生成する方法について説明します。

JSP からの XML の生成

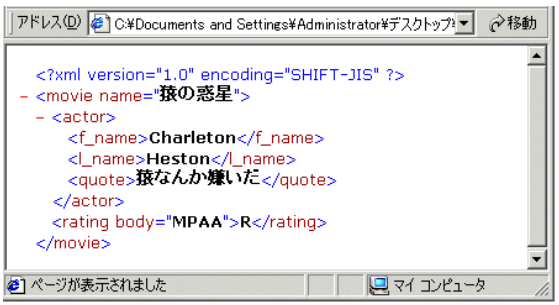
JSP から XML を生成するには、page ディレクティブの contentType 属性を text/xml に設定します。次のサンプルは、JSP で contentType を text/xml に設定する方法を示しています。

```
<%@ page contentType="text/xml" %>
<?xml version="1.0" encoding="UTF-8"?>
 <movie name="猿の惑星 ">
  <actor>
   <f_name>Charleton</f_name>
   <l_name>Heston</l_name>
   <quote> 猿なんか嫌いだ </quote>
  </actor>
  <rating body="MPAA">R</rating>
 </movie>
```

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

JSP から XML を生成すると、コンポーネントで XSLT を使用することによって、そのページコンテンツを変換したり、SAX/DOM を使用して XML をサーバーサイドオブジェクトに変換し、そのページのプロパティを抽出したりすることができます。

XML をサポートしているブラウザ (Microsoft の Internet Explorer など) で前のサンプルコードを呼び出すと、次のように表示されます。



メモ：XML シンタックスでは、<?xml version="1.0" encoding="UTF-8"?> の前に空白や改行を配置することは許されていませんが、Internet Explorer ではそれが許されています。その他のブラウザでは許されていない可能性があります。

データソースを繰り返し入力しながら適切な場所に HTML に似たタグを挿入することによって、JSP のデータから XML をダイナミックに生成することもできます。

次のサンプルコードは、Samples データソースの movies テーブルのデータを XML 形式に送信する方法を示しています。

```
<%@ page contentType="text/xml" %>
<%@ page import="java.sql.*,javax.naming.*,javax.sql.*" %>
<%
 String sqlstmt="SELECT movie FROM movies";
 String dsName="Samples";
 InitialContext ctx=new InitialContext();
 DataSource ds=(DataSource)ctx.lookup("java:comp/env/jdbc/" + dsName);
 Connection myConn=ds.getConnection();
 Statement myStmt=myConn.createStatement();
 ResultSet myRs=myStmt.executeQuery(sqlstmt);
%>
<?xml version="1.0" encoding="UTF-8"?>
<document>
<% while (myRs.next()) { %>
 <movie><%=myRs.getString("movie") %></movie>
 <rating><%=myRs.getString("rating") %></rating>
<% } %>
</document>
```

前のサンプルコードよって、次のような出力が生成されます。

```
<document>
  <movie> 猿の惑星 </movie>
  <rating>R</rating>
</document>
```

JSP は、前のサンプルコードで使用した従来の JSP シンタックスだけでなく、JSP XML シンタックスでも記述できます。詳細については、56 ページの「XML での JSP の記述」を参照してください。

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

サーブレットからの XML の生成

サーブレットでは、JSP と同様の方法で XML を生成します。大きな違いは、**setContentType** を **text/xml** に設定するために、**page** ディレクティブではなく、HttpServletResponse オブジェクトの **setContentType** メソッドを使用するという点です。また、XML 要素に使用する引用符文字もエスケープ処理する必要があります。

getWriter を呼び出す前に **setContentType** を呼び出す必要があります。この setContentType メソッド呼び出しでエンコードを設定できます。次のサンプルのように **println** ステートメントを使用すると、サーブレットから XML データを抽出できます。

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MovieServlet extends HttpServlet {
 public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException {
  response.setContentType("text/xml;charset=UTF-8");
  PrintWriter out = response.getWriter();
  out.println("<?xml version=¥"1.0¥" encoding=¥"UTF-8¥"?>");
  out.println("<movie name=¥"猿の惑星¥">");
  out.println("<actor><f_name>Charleton</f_name>");
  out.println("<l_name>Heston</l_name>");
  out.println("<quote>猿なんか嫌いだ</quote>");
  out.println("</actor><rating body=¥"MPAA¥">R</rating></movie>");
 }
}
```

Web ブラウザに出力される結果は、53 ページの「JSP からの XML の生成」で説明した JSP サンプルで生成されたものと同じです。コンテンツタイプが **text/xml** の HTML タグだけを含むページをエクスポートすることもできます。ただし、この場合は形式の整った HTML を記述する必要があります。つまり、**<P>** や **
** タグにもすべて終了タグを付けて XML バリデーションエラーの発生を避ける必要があります。

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

XML での JSP の記述

JSP は、基本的な JSP シンタックスまたは JSP XML シンタックスで記述できます。**JSP XML** シンタックスは、整形式の XML 文書に JSP を記述するための形式を定義します。従来の JSP シンタックスでは、HTML マークアップとともに、JavaScript および JSP ディレクティブ、スクリプトレット、アクション、および式を自由に使用することができました。JSP XML シンタックスの場合は、これらのすべての要素を使用することはできますが、ページ上の XML ルールに従う必要があります。

このセクションでは、JSP XML シンタックスで JSP のコードを記述する利点について説明し、これらのサンプルコードを記載します。

XML ビューについて

基本的な JSP シンタックスは、JSP 仕様によって定義されたスクリプトレット、アクション、ディレクティブ、式、および宣言で構成されています。Web ブラウザは、HTML のように JSP シンタックスを解釈することができません。ブラウザに JSP ページのコンテンツを表示するために、JRun はまず、JSP ソースコードをサーブレットとしてコンパイルする必要があります。JRun は、後でこのサーブレットを使用してレスポンスを生成します。このレスポンスは、HTML などの、リクエストされたマークアップ言語で生成されます。ただし、JSP からサーブレットへのコンパイルプロセスは直接行われません。まず、JSP を XML に変換する必要があります。

初めて JSP ページがリクエストされると、次の手順が実行されます。

1 最初は、クライアントが JSP をリクエストします。

2 JRun は、JSP を、XML ビューと呼ばれる一時的な XML 表現に変換します。

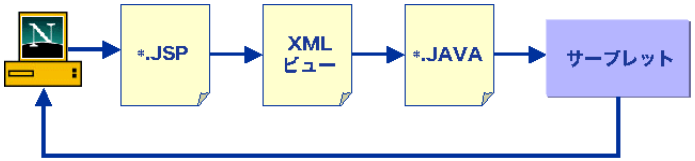
3 JRun は、JSP ページの XML ビューを検証します。

4 JRun は XML ビューを .java クラスファイルに変換します。
この .java クラスファイルは `jrun.jsp.runtime.HttpJSPServlet` を拡張し、`jrun.jsp.runtime.JRunJspPage` を実装します。サーブレットのソースコードは表示することができます。表示方法については、57 ページの「サーブレットのソースコードの表示」を参照してください。

5 JRun は、新規クラスをサーブレットにコンパイルします。

6 サーブレットは、クライアントリクエストに基づいて出力を生成します。

次の図は、クライアントが初めて JSP ページをリクエストしてから最終的にサーブレットにコンパイルされるまでの JSP ページのライフサイクルを示しています。その後、サーブレットはリクエストをクライアントに返します。



サーブレットのソースコードの表示

JSP ページを表すサーブレットのソースコードを表示することも可能です。JSP ページの XML 表現は、JRun がサーブレットを作成する間一時的にメモリ内に維持されるだけなので、表示できません。

デフォルトでは、JRun は、JSP のソースコードから生成されたサーブレットを保持しません。このセクションでは、Java ファイルを保持するように JRun を設定し、それらを表示する方法について説明します。

生成された JSP を保持するには

1 <JRun のルートディレクトリ>/servers/<JRun サーバー>/SERVER-INF/default-web.xml ファイルで、JSPServlet の **keepGenerated** 初期化パラメータを **true** に設定します。次にサンプルを示します。

```
<servlet>
  <servlet-name>JSPServlet</servlet-name>
  <servlet-class>jrun.jsp.JSPServlet</servlet-class>
  <init-param>
    <param-name>keepGenerated</param-name>
    <param-value>true</param-value>
  </init-param>
</servlet>
```

2 Web アプリケーションの /WEB-INF/jsp ディレクトリにある JSP クラスファイルを削除します。

このファイルを削除すると、それまでに生成されたサーブレットのクラスファイルが削除されます。JRun は、JSP を再コンパイルする前にクラスファイルの有無を確認します。クラスファイルが存在する場合、JSP は再コンパイルされません。また、JRun は JSP の日付 / 時刻の記録も確認します。したがって、ユーザーが JSP ソースコードを変更し、JSP ファイルを保存すると、JRun では Java ソースコードが再生成され、そのクラスが再コンパイルされます。

3 JRun サーバーを再起動します。

4 JSP をリクエストします。

5 Web アプリケーションの /WEB-INF/jsp ディレクトリに生成された、JSP の Java クラスファイル (*.java) を開きます。JRun は、ソースコードに "jrun__" で始まる名前を付け、この名前にバージョン情報を付加します。

JSP XML について

Java 1.2 JSP 仕様に、JSP シンタックスではなく XML で直接 JSP ページを記述できる機能が採用されました。JSP XML シンタックスは、従来の JSP シンタックスと似たマークアップを使用してページを記述しますが、XML 標準に準拠し、JSP 文書型記述 (DTD) に特有のタグがいくつか追加されています。

JSP XML シンタックスが JSP ページでサポートされていることによる利点は次のとおりです。

- JSP から XML ビューへの変換が不要なので、JRun は、JSP ページを高速でコンパイルできます。
- XML ツールを使用して JSP を処理できます。
- JRun は、Sun JSP 1.2 DTD に対して JSP を検証し、XML ファイルが well-formed (整形式) で、valid (妥当) かどうかを確認します。

- コンパイラによって、XML シンタックスに関して有用なデバッグメッセージが生成されます。

JSP XML シンタックススキーマの説明については、http://java.sun.com/dtd/jspxml.xsd をご覧ください。

JSP XML DTD の説明については、http://java.sun.com/dtd/jspxml.dtd をご覧ください。

簡単な JSP XML サンプル

次のサンプルは、前者が従来の JSP シンタックスで記述した JSP ページで、後者が JSP XML シンタックスで記述した JSP ページです。

従来の JSP シンタックス

```
<%-- 従来の JSP シンタックス -->
<%@ taglib uri=/WEB-INF/doclib.tld prefix="docs" %>
<HTML>
 <BODY>
  <docs:format><P> これはシンプルなページです。</docs>
  <I> とても </I> シンプルです。
 </BODY>
</HTML>
```

JSP XML シンタックス

```
<jsp:root xmlns:jsp="http://java.sun.com/jsp_1_2"
 xmlns:docs="/WEB-INF/doclib.tld"
 version="1.2">
 <jsp:text>
  <!-- XML JSP シンタックス -->
  <![CDATA[ <HTML><BODY> ]]>
 </jsp:text>
 <docs:format> これはシンプルなページです。
 <![CDATA[<I> とても </I>]]> シンプルです。</docs>
 <jsp:text>
  <![CDATA[ </HTML></BODY> ]]>
 </jsp:text>
</jsp:root>
```

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

JSP シンタックスと JSP XML シンタックスの違い

前のサンプルでもわかるように、JSP XML シンタックスは、従来の JSP シンタックスよりも詳細な指定が必要ですが、より構造化されたページを表現することができます。

前のサンプルの JSP シンタックスとの JSP XML シンタックスの違いは次のとおりです。

- ルート要素 `jsp:root` はページ全体をラップします。JRun はこのタグを JSP XML ページの始まりとして認識します。
- `jsp:root` 要素は、XML のネーム空間およびバージョン属性を定義します。ネーム空間属性は、ページで使用されるカスタムタグライブラリの接頭辞および URI を定義します。

- jsp:text 要素はテンプレートデータをラップします。テンプレートデータは、ページの本文、HTML タグ、JavaScript コードなどの JSP シンタックス要素が使用されていないすべてのデータを指します。
- CDATA 要素は、HTML タグなど、XML パーサーによる解釈から除外するテンプレートデータをラップします。XML パーサーは、このようなデータをうまく解釈できません。XML パーサーは、CDATA 要素内のすべてのデータを無視します。
- JSP XML タグに別の JSP XML タグを入れてネストすることはできません。

次の表に、従来の JSP シンタックスと JSP XML シンタックスの主な違いを示します。

タイプ	従来の JSP シンタックス	JSP XML シンタックス
page ディレクティブ	<% page ... %>	<jsp:directive.page ... />
include ディレクティブ	<%@ include ... %>	<jsp:directive.include ... />
taglib ディレクティブ	<%@ taglib ... %>	<jsp:root xmlns:prefix="*taglibURI*"> ... </jsp:root>
宣言	<%! *declaration* %>	<jsp:declaration> *declaration* </jsp:declaration>
式	<%= *expression* %>	<jsp:expression> *expression* </jsp:expression>
スクリプトレット	<% *scriptlet_code* %>	<jsp:scriptlet> *scriptlet_code* </jsp:scriptlet>
アクション	<jsp:include> ... </jsp:include> <jsp:forward> ... </jsp:forward> <jsp:plugin> ...</jsp:plugin> <jsp:useBean> ... </jsp:useBean> <jsp:setProperty> ... </jsp:setProperty> <jsp:getProperty> ... </jsp:getProperty>	旧バージョンの JSP 仕様では、標準アクションに XML シンタックスを使用するように規定されています。したがって、JSP XML でのアクションのシンタックスについても変更はありません。 jsp:include アクションで含めるファイルは JSP シンタックスまたは JSP XML シンタックスのいずれでもかまいませんが、同じファイルで両方のタイプのシンタックスを使用することはできません。
テンプレートデータ	template_data	<jsp:text> *template_data* </jsp:text>
コメント	<%-- コメント -->	<!-- コメント -->

新しい JSP XML タグ

次の表で、従来の JSP シンタックスの一部ではないが、JSP XML シンタックスで使用される新しい要素を説明します。

要素のシンタックス	説明
<jsp:root>	jsp:root 要素は、JSP XML シンタックスで最上位の要素です。この要素は、JSP のバージョン、ネーム空間、およびタグライブラリを定義します。 シンタックス <jsp:root xmlns:jsp="http://java.sun.com/jsp_1_2" xmlns:*tag_prefix*="*tag_library_URI*" version="1.2"> *Contents_of_JSP* </jsp:root> JSP XML ネーム空間は必須であり、http://java.sun.com/jsp_1_2 に設定する必要があります。 他の xmlns 属性はオプションです。これらの属性は、JSP ページのカスタムタグライブラリの接頭辞および URI を定義します。 version 属性は必須であり、1.2 (JSP 仕様のバージョン) に設定する必要があります。 例： <jsp:root xmlns:jsp="http://java.sun.com/jsp_1_2" xmlns:test="/WEB-INF/DocSamples.tld" version="1.2"> </jsp:root>
<jsp:text>	jsp:text 要素は、JSP のテンプレートデータをラップします。 XML パーサーは、jsp:text 要素内のテンプレートデータを検証します。したがって、テンプレートデータでは、引用符や括弧などの特殊文字に注意する必要があります。 詳細については、63 ページの「JSP XML での特殊文字の表現」を参照してください。
<![CDATA[*template_data*]]>	それ以外の JSP XML 要素では、CDATA 要素がテンプレートデータをラップします。XML パーサーは、CDATA 要素によってラップされているデータをすべて無視します。したがって、CDATA 要素は、特殊文字を多数含む HTML ブロック、JSP スクリプトレット、および宣言をラップするために広く使用されています。 シンタックス <*jsp_XML_element_tag*> <![CDATA[*template_data*]]> </*jsp_XML_element_tag*> 例： <jsp:text> **これは** <![CDATA[<I> **とても** </I>]]> **簡単な例です。** </jsp:text>

JSP XML シンタックスの詳細

次のセクションでは、JSP XML 要素のシンタックスについて詳しく説明します。

JSP XML での式と宣言の使用

括弧や引用符などの特殊文字が XML パーサーで正しく解釈されるようにするには、CDATA 要素内に宣言や式をラップする必要があります。次の例では、CDATA タグを使用して JSP XML 宣言全体をラップします。

```
<jsp:declaration><![CDATA[
 public int incrementCounter(int x) {
  x = x + 1;
  return x;
 } ]]>
</jsp:declaration>
```

jsp:text 要素には、jsp:expression および jsp:declaration 要素をラップしないでください。ただし、これらの要素には CDATA 要素を含めることができます。

式および宣言に含まれる特殊文字は、Unicode シーケンスに置き換えることもできます。次の例は、Unicode シーケンス ¥u0022 を使用して引用符を表した JSP XML 式を示しています。

```
<jsp:expression>request.getParameter(¥u0022backgroundcolor¥u0022)
        </jsp:expression>
```

JSP XML で Unicode シーケンスを使用する方法については、63 ページの「JSP XML での特殊文字の表現」を参照してください。

スクリプトレットの使用

JSP XML シンタックスのスクリプトレットと JSP シンタックスのスクリプトレットの動作の仕組みはほとんど同じです。ただし、特殊文字 (引用符や括弧など) を使用すると XML パーサーが誤って解釈することがあるので、これらは特別な方法でエンコードする必要があります。スクリプトレットに特殊文字を配置する場合は次を使用します。

- Unicode シーケンス
- HTML エンティティ

JSP XML で Unicode シーケンスと HTML エンティティを使用する方法については、63 ページの「JSP XML での特殊文字の表現」を参照してください。

jsp:text 要素には jsp:scriptlet 要素をラップしないでください。
また、jsp:scriptlet 要素には CDATA 要素を含めないでください。

アクションの使用

JSP アクションは、従来の JSP シンタックスおよび JSP XML シンタックスのいずれを使用した場合でも XML 標準に準拠します。したがって、コード内のアクションへのアクセス方法を変更する必要はありません。ただし、要求時属性値にアクセスするには、少し異なるシンタックスを使用する必要があります。つまり、次の例に示すように括弧を削除します。

従来の JSP シンタックス

```
<jsp:include page="<%= filename %>" />
```

JSP XML シンタックス

```
<jsp:include page="%= filename %" />
```

jsp:text 要素には JSP の action 要素をラップしないでください。また、JSP の action 要素には CDATA 要素を含めないでください。

JSP XML の例

次のサンプルコードは、JSP XML シンタックスで記述した JSP ページを示しています。

```
<jsp:root xmlns:jsp="http://java.sun.com/jsp_1_2">
 <jsp:scriptlet>
  String backgroundcolor=request.getParameter
        (¥u0022backgroundcolor¥u0022);
  String textcolor=request.getParameter(¥u0022textcolor¥u0022);
 </jsp:scriptlet>
 <jsp:text>
  <![CDATA[ <HTML> ]]>
  <![CDATA[ <BODY bgcolor="]]></jsp:text><jsp:expression>
        backgroundcolor</jsp:expression>
 <jsp:text><![CDATA[">]]>
  <![CDATA[ <FONT FACE="arial, helvetica" COLOR="]]>
        </jsp:text><jsp:expression>textcolor</jsp:expression>
 <jsp:text><![CDATA[">]]>
 </jsp:text>
 <jsp:text>
  <![CDATA[<P>]]>
  テキストの色は
 </jsp:text>
 <jsp:expression>textcolor</jsp:expression>
 <jsp:text>
  <![CDATA[<P>]]>
  背景の色は
 </jsp:text>
```

```
 <jsp:expression>backgroundcolor</jsp:expression>
 <jsp:text>
  <![CDATA[ </FONT></BODY></HTML>]]>
 </jsp:text>
</jsp:root>
```

このページを表示するには変数の設定を使用します。次の URL のように、リクエストパラメータを使用してフォントの色と背景色を設定してください。

http://localhost:8100/testxml.jsp?**textcolor=orange&backgroundcolor=black**

次のコードは、前のコードの HTML バージョンです。

```
<HTML>
<BODY bgcolor="black">
<FONT FACE="arial, helvetica" COLOR="orange">
テキストの色は
  オレンジ
<P>
背景の色は
  黒
</FONT></BODY></HTML>
```

サンプルの JSP を JSP XML シンタックスで表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

FONT タグなどの HTML タグを作成する場合は、もう一方のタグと同じ行に request-time 変数式を配置してください。そうしないと、ブラウザがタグを誤って解釈し、予期しない結果が出力される場合があります。

JSP ページに HTML タグを含める場合は必ず CDATA 要素を使用してください。そうしないと、XML パーサーは jsp:text 要素の内容を検証するときに特殊文字を誤って解釈してしまいます。

場合によっては、HTML エンティティまたは Unicode シーケンスを使用して特殊文字を表現できます。詳細については、63 ページの「JSP XML での特殊文字の表現」を参照してください。

JSP XML での特殊文字の表現

XML パーサーが誤って解釈する可能性のある特殊文字を表現するには、Unicode エスケープシーケンスまたは HTML エンティティを使用してください。すべての特殊文字を CDATA 要素内に配置するよりも、これらの文字コード体系を使用する方が簡単です。

JSP XML 要素内では、場合によっては引用符 (") の代わりにアポストロフィ (') を使用できます。JRun XML パーサーはアポストロフィを引用符として受け入れます。ただし、コンパイラによっては受け入れないものがあります。

次のセクションでは、JSP XML で HTML エンティティと Unicode シーケンスを使用する方法について説明します。

HTML エンティティリファレンスの使用

HTML エンティティリファレンスは記号名を使用して文字を表します。HTML エンティティリファレンスは次のシンタックスで指定します。

```
& + entity_name + ;
```

たとえば、右不等号 (>) の HTML エンティティリファレンスは次のとおりです。

```
&gt;
```

HTML エンティティのリストについては、http://www.ramsch.org/martin/uni/fmi-hp/iso8859-1.html をご覧ください。

XML パーサーは、JSP レスポンスを出力に書き込む際に、HTML エンティティを対応する文字表現に変換します。次の表で、JSP XML 要素内で使用できる HTML エンティティを説明します。

要素	例

jsp:text	次の例は、jsp:text 要素内で左不等号と右不等号の代わりに使用する HTML エンティティを示しています。
	`<jsp:text>`
	` <HR>`
	`</jsp:text>`

jsp:declaration	次の例は、jsp:declaration 要素内で引用符の代わりに使用する HTML エンティティを示しています。
	`<jsp:declaration>`
	` public String returnColor() {`
	`  String color=&quot;black&quot;;`
	`  return color;`
	` }`
	`</jsp:declaration>`

jsp:expression	次の例は、jsp:expression 要素内で引用符の代わりに使用する HTML エンティティを示しています。
	`<jsp:expression>`
	` request.getParameter(&quot;backgroundcolor&quot;)`
	`</jsp:expression>`

jsp:scriptlet	次の例は、jsp:scriptlet 要素内で引用符の代わりに使用する HTML エンティティを示しています。
	`<jsp:scriptlet>`
	` String power=request.getParameter(&quot;power&`
	` quot;);`
	`</jsp:scriptlet>`

CDATA	CDATA 要素内で HTML エンティティを使用することはできません。
JSP アクション	action 要素内で HTML エンティティを使用することはできません。

Unicode シーケンスの使用

Unicode は、すべての文字に対して固有の数値を提供しており、ほとんどのプラットフォームでサポートされています。Unicode シーケンスは 4 文字のシーケンスで構成され、これらの文字シーケンスが記号にマッピングされます。右不等号 (>) の Unicode シーケンスは次のとおりです。

```
003E
```

Java コードでは、次の例のように、¥u を付けてシーケンスをエスケープ処理する必要があります。

```
¥u003E
```

Unicode シーケンス文字のリストについては、http://www.unicode.org をご覧ください。

XML パーサーは、JSP レスポンスを出力に書き込む際に、Unicode シーケンスを対応する文字表現に変換します。次の表に、JSP XML 要素内で Unicode シーケンスを使用するためのガイドを示します。

要素	例
jsp:text	jsp:text 要素内では Unicode シーケンスを使用できません。

jsp:declaration	次の例は、jsp:declaration 要素内で引用符の代わりに使用する Unicode シーケンスを示しています。
	`<jsp:declaration>`
	` public String returnCheese() {`
	`  String cheese=¥u0022cheese¥u0022;`
	`  return cheese;`
	` }`
	`</jsp:declaration>`

jsp:expression	次の例は、jsp:expression 要素内で引用符の代わりに使用する Unicode シーケンスを示しています。 `<jsp:expression>request.getParameter(¥u0022power¥u0022)</jsp:expression>`

jsp:scriptlet	次の例は、jsp:scriptlet 要素内で引用符の代わりに使用する Unicode シーケンスを示しています。
	`<jsp:scriptlet>`
	` String power=request.getParameter(¥u0022power¥u0022);`
	` out.println("¥u003CH1¥u003EBehold the power of " + power);`
	`</jsp:scriptlet>`

CDATA	次の例は、CDATA 要素内で引用符の代わりに使用する Unicode シーケンスを示しています。
	`<jsp:declaration><![CDATA[`
	` public String returnCheese() {`
	`  String cheese=¥u0022cheese¥u0022;`
	`  return cheese;`
	` } ]]>`
	`</jsp:declaration>`

JSP アクション	action 要素内では Unicode シーケンスを使用できません。

Web アプリケーションでの XML の変換

JRun の Xalan XSLT プロセッサで JAXP を使用すると、XML ドキュメントを変換し、それをサーブレットまたは JSP として出力できます。XML を変換するには、XML 入力ソースと XSL スタイルシートが必要です。XML は、HTML またはその他のマークアップ言語 (WML や SGML など) に変換できます。

XSL スタイルシートの使用

XSL スタイルシートは、XML 要素のコンテンツの書式設定方法をクライアントに指示します。XSL スタイルシートを使用すると、一連の HTML タグを HTML ページとして生成できます。

HTML ページの BODY タグでは、次の XSL スタイルシートのサンプルは、HomePageLinks というラベルの付いたテキストをハイパーリンクに変換し、HTML ページを出力します。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
        Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
 <HEAD>
  <TITLE> ホームページ </TITLE>
 </HEAD>
 <body>
  <h2> 利用可能なリンク </h2><br/>
  <xsl:apply-templates select="HomePageLinks"/>
 </body>
</html>
</xsl:template>
<xsl:template match="A">
 <xsl:element name="A">
  <xsl:attribute name="HREF">
   <xsl:value-of select="@HREF"/>
  </xsl:attribute>
  <xsl:value-of select="."/>
 </xsl:element>
<br/>
</xsl:template>
</xsl:stylesheet>
```

XSL スタイルシートの使用方法の詳細については、77 ページの「リソース」を参照してください。

JAXP Transformer オブジェクトの使用

JAXP では Transformer オブジェクトが利用可能です。Transformer オブジェクトは、さまざまなソースからの XML 入力を処理し、その変換結果をすべての出力に書き込みます。Transformer オブジェクトを作成するには、TransformerFactory クラスの newTransformer メソッドを使用します。

次のサンプルサーブレットは、前の XSL ファイルのルールに従って XML ソースファイルを取得して変換します。

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.*;
import java.net.*;

public class XSLTHomePage extends HttpServlet {
 public void doGet (HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
  PrintWriter out = response.getWriter();
  try {
   //TransformerFactory クラスを作成します。
   TransformerFactory tFactory = TransformerFactory.newInstance();
   // XML 入力ドキュメントとスタイルシートを取得します。
   ServletContext sc = this.getServletContext();
   URL xslURL = sc.getResource("/homepagelinks.xsl");
   URL xmlURL = sc.getResource("/homepagelinks.xml");
   Source xslSource = new StreamSource(new
        java.net.URL(xslURL.toString()).openStream());
   Source xmlSource = new StreamSource(new
        java.net.URL(xmlURL.toString()).openStream());
   // Transformer オブジェクトを生成します。
   Transformer transformer = tFactory.newTransformer(xslSource);
   // 変換を実行し、レスポンスに出力を送ります。
   transformer.transform(xmlSource, new StreamResult(out));
  } catch (Exception e) {
   out.println(e.getMessage());
  }
 }
}
```

JAXP の使用方法の詳細については、次をご覧ください。

- JAXP specification
- JAXP JavaDoc

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

XMLScript

JRun には、XPath ベースの XML スクリプトユーティリティである XMLScript が含まれています。このユーティリティを使用すると、限られた一連の XPath ステートメントを実行することができます。

XMLScript を使用すると、XML ファイルの式を検索して、ノードに値を追加したり、ノードの値を書き換えたり、ノードを削除したりすることができます。また、単に、XPath 式によって生成されたノードを表示することもできます。その結果は、新規 XML ファイルに保存したり、元の XML ファイルを上書き保存したりすることができます。

XMLScript シンタックス

XMLScript シンタックスは次のとおりです。

```
> java -classpath classpath XMLScript [-i input-file] [-o output-file]
        [-f script-file] -[a|d|s|v statement ...]
```

XMLScript は <JRun のルートディレクトリ >/lib/jrun.jar ファイルにあります。XMLScript を使用するには、クラスパスに jrun.jar を含める必要があります。Windows プラットホームの場合、<JRun のルートディレクトリ >/bin ディレクトリの XMLScript.exe ファイルをダブルクリックして XMLScript を起動します。

`input-file` 引数を指定すると、標準入力がデフォルトになります。

`output-file` 引数を指定すると、`input-file` の値がデフォルトになります。ダッシュを指定すると、標準出力がデフォルトになります。

`script-file` 引数は、XPath ステートメントのリストを含んでいるファイルへのパスを指定します。スクリプトファイルを指定しないと、XMScript は、コマンド行に指定された XPath ステートメントを実行します。スクリプトファイルを使用して XMLScript を実行する方法については、69 ページの「スクリプトファイルを使用した XMLScript の実行」を参照してください。

XPath ステートメントは形式は次のとおりです。

```
[-a expression value]
[-d expression]
[-s expression value]
[-v expression]
```

- **-a**　XML ファイルに現在存在しているが値を持たないノード上の式に値を追加します。次に例を示します。

  ```
  <attribute name="targetHost" />
  ```

  このサンプルに相当するノードに値 "localhost" を追加すると、次の行が生成されます。

  ```
  <attribute name="targetHost">localhost</attribute>
  ```

- **-d**　式に一致する最初の XPath ノードを削除します。
- **-s**　XPath 式に一致するノードに値を代入します。
- **-v**　式に一致する XPath ノードを表示します。この引数をプログラムで起動すると、List オブジェクトが返されます。それ以外は、ノードのリストが標準出力に出力されます。

XPath ステートメントの記述方法の詳細については、http://www.w3.org/TR/xpath をご覧ください。

XMLScript の使用

XPath コマンドは、コマンド行またはスクリプトファイルを使用して XMLScript に渡すことができます。また、Java クラスファイルに XMLScript を含めると、XPath ステートメントをプログラムで実行できます。このセクションでは、XMLScript ユーティリティの使用例について説明します。

コマンド行を使用した XMLScript の実行

XMLScript をコマンド行で実行するには、次のサンプルのように、ステートメントの式および値を引数として渡します。

```
>java -classpath c:/jrun4/lib/jrun.jar jrunx.xml.XMLScript -i
        c:/jrun4/servers/default/server-inf/jrun.xml -s
        //service[@name='WebService']/attribute[@name='port']/
        text() 8842
```

このサンプルは、WebService という名前の service 要素を探し、そのサービスの port 属性を 8842 に置き換えます。次のような XML が生成されます。

```
<service class="jrun.servlet.http.WebService" name="WebService">
  <attribute name="port">8842</attribute>
</service>
```

メモ：XMLScript ステートメントをコマンド行で使用する際は、引数を引用符 (') で囲んでください。

コマンド行では、任意の数のステートメントを渡すことができます。次のコマンドを指定すると、次の 2 つのステートメントが実行されます。

```
>java -classpath c:/jrun4/lib/jrun.jar jrunx.xml.XMLScript
        -i c:/jrun4/servers/default/server-inf/jrun.xml
        -s //service[@name='ClusterManager']/
        attribute[@name='enabled']/text() true -s //
        service[@name='ClusterManager']/
        attribute[@name='clusterDomain']/text() newserver
```

スクリプトファイルを使用した XMLScript の実行

XMLScript を実行し、XPath コマンド、式、および値を含んでいるスクリプトファイルを渡すことができます。これにより、任意の数の XPath ステートメントを一度に実行できます。次のサンプルでは、scriptfile.txt ファイルから XPath ステートメントを抽出し、inputfile.xml ファイルに対してそれらのステートメントを実行します。

```
> java -classpath c:/jrun4/lib/jrun.jar jrunx.xml.XMLScript -i
        inputfile.xml -f scriptfile.txt
```

XPath ステートメントを含んでいる XMLScript スクリプトファイルのサンプルは次のとおりです。

```
-s //service[@name=¥"WebService¥"]/attribute[@name=¥"port¥"]/text() 8142
-s //service[@name='WebService']/attribute[@name='port']/text() 8842
-s //service[@name='ClusterManager']/attribute[@name='enabled']/text()
        true
-a //service[@name=¥"LauncherInfo¥"]/attribute[@name=¥"vmArgs¥"] -Xms128
```

スクリプトファイルを使用して XMLScript を実行する場合は、a または s コマンドの前のハイフン (-) を省略してもかまいません。

Java クラスでの XMLScript の使用

XMLScript は次のメソッドを公開します。

コマンド (Array)

この Array オブジェクトには、XPath ステートメントの各コマンド、式、および値が個々の要素に含まれています。XMLScript をプログラムで使用するには、**jrunx.xml.XMLScript** パッケージをインポートする必要があります。Java クラスをコンパイルするには、クラスパスに jrun.jar ファイルを含める必要があります。

次のコードは、Java クラスで XMLScript を使用する方法を示しています。

```
import java.util.*;
import java.io.*;
import jrunx.xml.XMLScript;

public class XMLScriptTest {
 public static void main() throws Exception {
  String xmlinfile = "c:/jrun4/servers/default/SERVER-INF/jrun.xml";
  String xmloutfile = "c:/jrun4/servers/newserver/SERVER-INF/jrun.xml";

  ArrayList args = new ArrayList();

  args.add("-i");
  args.add(xmlinfile);

  args.add("-o");
  args.add(xmloutfile);

  String port = "8142";
  args.add("-s");
  args.add("//service[@name=¥"WebService¥"]/
        attribute[@name=¥"port¥"]/text()");
  args.add(port);

  String vmarg = "-Xms128";
  args.add("-a");
  args.add("//service[@name=¥"LauncherInfo¥"]/
        attribute[@name=¥"vmArgs¥"]");
  args.add(vmarg);

  //引数を持つ XMLScript を実行します。
  String[] scriptArgs = new String[args.size()];
  args.toArray(scriptArgs);
  try {
   XMLScript xs = new XMLScript();
   xs.command(scriptArgs);
  } catch (Exception e) {
  }
 }
}
```

Web アプリケーションでの XDoclet の使用

JRun は XDoclet と統合できます。XDoclet は、EJB、Web アプリケーション、および JSP タグライブラリのコードおよびデプロイメントディスクリプタを生成するオープンソースのツールです。

XDoclet は、設定ファイルで定義する `WebDoclet` タスクの設定を使用して、web.xml ファイルと jrun-web.xml を生成します。サーブレットのコメントを使用して、サーブレットの定義、フィルタ、マッピングなどの設定を作成します。さらに、セッションのパーシスタンス情報や仮想マッピングを定義するデプロイメントディスクリプタにファイルをマージできます。

JRun は、web.xml ファイルに要素を生成する基本的な XDoclet タグ、および jrun-web.xml ファイルに要素を生成する JRun 特有の XDoclet タグをサポートします。

XDoclet は Apache の Ant を使用しているため、その設定ファイルは build.xml ファイルに似ています。XDoclet の設定ファイルは、<JRun のルートディレクトリ>/lib/xdoclet.xml です。

詳細については、XDoclet の Web サイト (http://sourceforge.net/projects/xdoclet/) をご覧ください。

XDoclet サービスの有効化

XDoclet は、jrun.xml ファイルの XDocletService セクションを使用して有効にします。

XDoclet を有効にするには

1 XDoclet コメントを含んでいるサーブレットクラスファイルを作成します。

2 JRUN サーバーの jrun.xml ファイルの `XDocletService` セクション内の `warSourceFiles` 属性がサポートしている命名規則を使用してそのサーブレットに名前を付けるか、または、サーブレットのファイル名に一致する `warSourceFiles` 属性を追加します。

デフォルトの命名規則は次のとおりです。

- `*Servlet.java`
- `*Tag.java`
- `*Filter.java`

これらのデフォルトの命名規則はすべてのサーブレットに適用され、名前は Servlet、Tag、または Filter で終わり、拡張子 .java が付きます。

3 JRun サーバーの jrun.xml ファイルの `XDocletService` セクションの `watchedWARDirectory` 属性を追加して、ディレクトリ内のすべてのファイルに対する XDoclet サポートを有効にします。

デフォルトのディレクトリは次のとおりです。

<JRun のルートディレクトリ>/default-ear/default-war ディレクトリ

このディレクトリは、デフォルトではコメントとして処理されます。XDoclet が `watchedWARDirectory` 属性を監視できるようにするには、この属性のコメントを解除する必要があります。補足ディレクトリを追加して、JRun が監視できるようにすることもできます。

4 XDoclet サポートが有効になっているディレクトリにサーブレットクラスを保存します。

JRun は、Web アプリケーションのデプロイメントディスクリプタにサーブレットのエントリを生成します。

XDoclet リソースの設定

XDoclet リソースは <JRun のルートディレクトリ >/lib/xdoclet.xml で設定します。また、XDoclet を JRun サブタスクのビルドの一部として割り当てることができます。JRun サブタスクは、xdoclet.xml ファイルに `<jrunwebxml/>` として表されます。次の表は、`<jrunwebxml/>` サブタスクの属性をリストしたものです。

属性	説明
`xmlencoding`	jrun-web.xml ファイルのエンコード。デフォルトは UTF-8 です。
`destdir`	JRun 特有のデプロイメントディスクリプタファイル出力の保存先ディレクトリを指定します。デフォルトは `webdoclet` タスクの `destdir` パラメータです。
`mergedir`	マージファイルのディレクトリを指定します。マージファイルは、session-config.xml および virtual-mapping.xml です。
`contextRoot`	Web アプリケーションのコンテキストルートを指定します。デフォルトでは、コンテキストルートは Web アプリケーションが含まれるディレクトリまたは JAR ファイルの名前です。
`reload`	サーブレット、サーブレットヘルパークラス、および JSP ヘルパークラスを自動的にリロードするかどうかを指定します。デフォルトは true です。
`compile`	サーブレット、サーブレットヘルパークラス、および JSP ヘルパークラスを自動的にコンパイルするかどうかを指定します。true に設定するとパフォーマンスが低下します。デフォルトは true です。
`loadSystemClassesFirst`	エンタープライズアプリケーションクラスおよび Web アプリケーションクラスの前にシステムクラスをロードするかどうかを指定します。通常、クラスローダー委任モデルでは、システムクラスを最初にロードすることになっていますが、サーブレット仕様では Web アプリケーションクラスを最初にロードするように推奨しています。デフォルトは true です。

`<jrunwebxml/>` タスクの例については、76 ページの「XDoclet を使用した EJB の例」を参照してください。

web.xml へのファイルのマージ

デプロイメントディスクリプタの構築時に XDoclet で使用するマージファイルを定義できます。これらの外部マージファイルで、セッションのパーシスタンス情報や仮想マッピングなどの設定を定義できます。これらのマージファイル名は session-config.xml および virtual-mapping.xml で、格納場所は XDoclet の設定ファイルで定義できます。

XDoclet タグの使用

サーブレットクラスファイル内で jrun.xml 属性と JavaDoc スタイルのコメントを組み合わせて使用すると、XDoclet が実行する処理を制御できます。このセクションでは、XDoclet で使用するソースコードの標準的な XDoclet タグと JRun 特有のタグについて説明します。

XDoclet のコメントは、@ マークで始まります。

Web アプリケーションの標準的な XDoclet タグ

XDoclet タグは、基本的に **`@jsp:`** または **`@web:`** という接頭辞で始まります。これらのタグは、基本的な XDoclet のアーキテクチャの一部です。これらのタグをソースファイルで使用する場合、XDoclet がこれらを使用して Web アプリケーションの web.xml ファイルにエントリを作成します。

次の表で、Web アプリケーションコンポーネントで使用する標準的な XDoclet タグを説明します。

タグ	説明
@jsp:attribute	指定のフィールドを JSP タグの属性として宣言します。このタグは getter メソッドに配置します。
@jsp:tag	指定のクラスを JSP タグの実装クラスとして宣言し、そのタグの様々なプロパティを指定します。
@jsp:validator-init-param	特定の parameter-name、value および description の Validator に対する初期化パラメータを宣言します。
@jsp:variable	JSP タグの変数および定義したスクリプト変数の情報を宣言します。
@web:ejb-local-ref	特定の name、type、home-interface-name、local-interface-name、link- name、および description のローカル EJB リファレンスを定義します。link-parameter の値は、同じ J2EE アプリケーションユニットにあるエンタープライズ bean の ejb-name である必要があります。
@web:ejb-ref	特定の name、type、home-interface-name、local-interface-name、link- name、および description のリモート EJB リファレンスを定義します。link-parameter の値は、同じ J2EE アプリケーションユニットにあるエンタープライズ bean の ejb-name である必要があります。
@web:env-entry	特定の name、type、および value の環境エントリを定義します。
@web:filter	特定の name、display-name、icon、および description のクラスを Filter クラスとして宣言します。Servlet 2.3 のみ適用可能です。
@web:filter-init-param	特定の parameter-name、value、および description の Filter に対する初期化パラメータを宣言します。Servlet 2.3 のみ適用可能です。
@web:filter-mapping	Filter のマッピングを定義します。url-pattern または servlet-name を指定する必要があります。Servlet 2.3 のみ適用可能です。

タグ	説明
@web:listener	特定のクラスを Listener クラスとして宣言します。
@web:resource-env-ref	特定の name、type、および description のリソース環境リファレンスを定義します。
@web:resource-ref	特定の name、type、description、authentication (auth)、および scope のリソースリファレンスを定義します。
@web:security-role	特定の role-name および description のセキュリティロールを定義します。
@web:security-role-ref	role-link という名前の security-role-link に対する特定の role-name のセキュリティロールリファレンスを定義します。
@web:servlet	特定の name、display-name、icon、および description のクラスを Servlet クラスとして宣言します。
@web:servlet-init-param	特定の parameter-name、value、および description の Servlet に対する初期化パラメータを宣言します。
@web:servlet-mapping	特定の url-pattern に対する Servlet のマッピングを定義します。

タグおよびそのパラメータの全リストを含む詳細については、XDoclet の Web サイト http://xdoclet.sourceforge.net/ を参照してください。

JRun 特有の XDoclet タグ

JRun 特有の XDoclet タグは、**`@jrun:`** という接頭辞で始まります。これらのタグをソースファイルで使用する場合、XDoclet がこれらを使用して jrun-web.xml ファイルにエントリを作成します。これらのタグは、すべてクラスレベルのタグです。次の表は、JRun 特有の XDoclet タグをリストします。

タグ	説明
@jrun:ejb-local-ref	**注意：**bean 開発者が提供した ejb-ref-name とその JNDI 名の間のマッピングを指定します。デプロイ担当者が実際の JNDI 名を指定します。
@jrun:ejb-ref	bean 開発者が提供した ejb-ref-name とその JNDI 名の間のマッピングを指定します。デプロイ担当者が実際の JNDI 名を指定します。使用場所は entity、message-driven、または session です。
@jrun:resource-env-ref	bean 開発者が提供した resource-env-name とその JNDI 名の間のマッピングを指定します。デプロイ担当者が実際の JNDI 名を指定します。
@jrun:resource-ref	bean 開発者が提供した resource-name とその JNDI 名の間のマッピングを指定します。デプロイ担当者が実際の JNDI 名を指定します。

JRun 特有の XDoclet タグの詳細については、ドキュメンテーションのホームページで利用できるオンラインディスクリプタドキュメントを参照してください。XDoclet を使用するオンラインの EJB の例については、samples JRun サーバーを参照してください。

XDoclet の例

このセクションでは、XDoclet タグを使用する Java ソースファイルと xdoclet.xml 設定ファイルの Java ソースファイルについて説明します。

サーブレットの例

次の例で示されている単純なサーブレットは、XDoclet を使用してサーブレットの url-pattern マッピングおよび初期化パラメータを定義します。

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

/**
 * A hello world.
 *
 * @web:servlet name="SimpleServlet" display-name="Simple Servlet"
        load-on-startup="1"
 * @web:servlet-init-param name="param1" value="value1"
 * @web:servlet-init-param name="param2" value="value2"
 * @web:servlet-mapping url-pattern="/hellaworld/*"
 *
 * @author     Macromedia
 * @created    Mar 14, 2002
 * @version    $1.0 $
 **/

public class SimpleServlet extends HttpServlet {
  public void doGet( HttpServletRequest request, HttpServletResponse
        response ) throws IOException, ServletException {
    PrintWriter out = response.getWriter()
    out.println("Hella World");
  }
}
```

フィルタサンプル

次の例で示されているフィルタサーブレットは、XDoclet を使用してフィルタの定義、すべての XML ファイルへのフィルタのマッピング、およびフィルタの初期化パラメータの定義を行います。

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
/**
```

```
 * xdoclet を使用して web.xml へのフィルタ追加をテストします。
 *
 * @web:filter name="XDocletFilter" display-name="XDoclet Test Filter"
 * @web:filter-init-param name="param1" value="value1"
 * @web:filter-init-param name="param2" value="value2"
 * @web:filter-mapping url-pattern="/*.xml"
 *
 * @author     Macromedia
 * @created    Mar 14, 2002
 * @version    $1.0 $
 **/
public class XDocletFilter extends GenericFilter {
  public void doFilter(ServletRequest req, ServletResponse resp,
        FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {
    chain.doFilter(req, resp);
  }
}
```

XDoclet を使用した EJB の例

次の例では、xdoclet.xml ファイルの WebDoclet タスクを定義します。

```
<webdoclet
  sourcepath="${xdoclet.working.dir}"
  destdir="${xdoclet.working.dir}"
  classpathref="project.class.path">
  <!-- Struts サポートのインクルード -->
  <fileset dir="${xdoclet.working.dir}">
    <include name="**/*Servlet.java" />
    <include name="**/*Filter.java" />
    <include name="**/*Tag.java" />
    <include name="**/*Action.java" />
  </fileset>
  <deploymentdescriptor servletspec="2.3" destdir="${war.meta.dir}" />
  <jsptaglib jspversion="1.2"
    destdir="${war.meta.dir}/tlds"
    shortname="j2ee"
  />
  <jrunwebxml
    destdir="${war.meta.dir}"
    contextRoot="webapp"
    reload="true"
    compile="true"
    loadSystemClassesFirst="false"
  />
</webdoclet>
```

リソース

XML の詳細については、次のリソースを参照してください。

リソース	リソースの位置
Java と XML	http://java.sun.com/xml/http://java.sun.com/xml/
Xalan	http://xml.apache.org/xalan-j/index.html
Crimson	http://xml.apache.org/crimson/
Inside XML	Steven Holzner、New Riders press 刊。
Developing XML Solutions with JavaServer Pages Technology	http://java.sun.com/products/jsp/html/JSPXML.html
JSP と XML の使用	http://java.sun.com/products/jsp/html/JSPXML.html
サーブレットと XML の使用	http://www.onjava.com/pub/a/onjava/2000/12/15/xslt_servlets.html
JSP と XML/XSLT の併用	http://developer.java.sun.com/developer/technicalArticles/xml/WebAppDev2/
JAXP タグライブラリ	http://www.jspinsider.com/jspkit/
XPath	http://www.w3.org/TR/xpath
XPointer	http://www.w3.org/TR/WD-xptr
JDOM	http://jdom.org/index.html
DOM4J	http://www.dom4j.org/
XDoclet	http://xdoclet.sourceforge.net/

第 5 章
Web アプリケーションのセキュリティ

この章では、Web アプリケーションデザインでセキュリティが果たすロールを紹介し、JRun の認証メカニズムを説明します。これには、サーブレット API を使用して認証制限を Web アプリケーションに適用する方法も含まれます。また、サーブレットと JSP でセキュアコードを書く最も一般的な方法についても説明します。

目 次

セキュリティの概要

セキュア Web アプリケーションデザインは、製品やプラットフォームに特有のものではなく、あらゆる Web アプリケーションを安全にデザインして実装する場合に役立ちます。このセクションでは、開発者とアプリケーションアーキテクトのためにセキュリティの概念を紹介します。これらの概念のほとんどは、Web アプリケーション以外のものを含めて、すべてのアプリケーション開発サイクルに当てはまるものです。

リスク評価

リスク評価 は、安全なシステムをデザインする作業の枠組み作りに役立つので、デザイン処理の最初のステップで行なう必要があります。リスク評価の基本的な手順は次のとおりです。

1 保護するリソースの特定

2 相対値の割り当て

3 可能性のあるアタッカーの特定

4 各タイプのアタッカーの相対的頻度の予測

5 アタックツリー分析の実行と可能性のあるアタックルートの特定

6 可能性のあるアタックルートすべての保護

保護するリソースには、データベース、ファイル、または組織外の事業者にとって価値があるリソースが含まれます。リスク評価処理は、顧客情報などの社会的、法的な問題の秘密度、処理、および取り扱いに関する社内ポリシーに依存します。

アタッカーのスキル、頻度、方法の予測はすべて、アタックツリー分析と呼ばれる関連処理に属します。この処理は、分析と評価の主観的処理の形式化に役立ち、プロジェクトのセキュリティゴールの優先順位の決定に役立ちます。

アタックツリー分析から、保護するアタックルートを特定します。分析を行うときは、デザイン内で実装するセキュリティのタイプに関する情報を整理します。この情報は、アプリケーションデザインに見合ったセキュリティポリシーやプライバシーポリシーの作成に役立ちます。

セキュリティポリシー

セキュリティポリシー は、組織のセキュリティニーズに関する分析を書き出したものです。特に、セキュリティを考慮したアプリケーションデザインを計画するときに役立ちます。ポリシーを正しく書き出しておけば、セキュリティポリシーの条件を厳守、保護するようにコードをより簡単に変更できます。また、製品の条件の変化に合わせて、セキュリティポリシーを最新の状態に保つ必要があります。

委任

委任は、アプリケーションアーキテクチャ内の専門モジュールやレイヤーにタスクを割り当てる行為です。たとえば、リソースファイルのグループを保護するタスクの権限を JAAS に委任したり、ユーザー管理やグループ管理のタスクの権限を LDAP サーバーに委任することがあります。これは、モジュラーアーキテクチャまたはレイヤー化とも呼ばれています。

モジュールの定義方法は人によって異なります。高レベルから、モジュールを必須とオプションのハードウェア、ドライバ、ソフトウェアのカテゴリーに分類することもあれば、入力、記憶、表示、処理、出力などの機能的なロールに分類することもあります。また、モジュールをサーブレット、JSP、JavaBean などのソフトウェアコンポーネントとして定義することもあります。

委任は、デザインと開発において非常に重要です。理論上は、セキュリティアーキテクチャが、オペレーティングシステムで使用される ACL (Access Control List：アクセスコントロールリスト) やファイルシステムによって実装されるアクセス許可などの、プラットフォームに存在する既存のレイヤーにアプリケーションの機能を委任します。

安全な実装では、他のプログラマーや開発者は、アプリケーションレイヤーに依存してデータの安全を確保できます。つまり、レイヤー化の観点からすると、データを外部から取得する場合、データを安全に処理することはあなたのタスクとなります。

検証

検証とは、受け取る情報の内容が正しいことを確認することです。アプリケーションで名前とアドレス情報を受け取る際、代わりに SQL コマンドを受け取ってしまうと、実行メソッドの呼び出し中にこの SQL コマンドが実行されてしまう可能性があります。一方、アプリケーションの検証メカニズムでは 、データを実行メソッドに渡す前に SQL の文字や文字列を確認およびフィルタして除外することができます。

検証とそのセキュリティ条件の間にある関係、正式な信頼は、人や物を信頼する行為を定義および分析する、公認された処理です。

セキュリティスペシャリストは、正式な処理を用いて信頼できるエンティティを識別します。正式な処理はセキュリティポリシーと非常に密接に関連しています。多くの場合、セキュリティポリシーでは、リソースやエンティティを信頼するために必要な条件が定義されます。つまり、会社の正式な信頼処理を定義する特定のポリシーは変わりやすいということです。

入手されたデータが信頼できるリソース (正式な信頼のための、ポリシーで定められたすべてのテストに合格したリソース) から来たものでないかぎり、そのデータを信頼することはできません。データを信頼できない場合は、検証が必要となります。

宣言セキュリティとプログラムセキュリティの比較

宣言セキュリティは、併用される他の Web コンポーネントとは別のレイヤーとして実装されます。ファイルアクセスの許可セットや、ユーザー、グループ、ロールなどのセキュリティシステムを設定してから、アプリケーションの認証メカニズムをそのレイヤーに組み込みます。

宣言セキュリティを使用すると、Web アプリケーションを作成するプログラマーはプログラムの作成環境を無視することができます。また、Web アプリケーションを更新する場合、一般的にセキュリティモデルのリファクタリングは必要ありません。

宣言セキュリティは、ファイルアクセスの許可セットや ACL として、また、Web アプリケーションへのすべてのリクエストを遮断するフィルタとして実装されます。

プログラムセキュリティでは、かなり詳細にセキュリティを設定することができます。Web アプリケーション内の各コンポーネントがセキュリティモデルを実装するので、コンポーネントのレベルでセキュリティを強化でき、また、必要な場合はページごとにセキュリティを強化することも可能です。これにより、柔軟性と精度が高まりますが、実装コストがかさみます。

プログラムセキュリティは、HTTP ヘッダーをチェックするすべてのサーブレットに追加する、一般的なメソッドのセットとして実装できます。

宣言セキュリティは、コードを再使用するデザインになっていて管理が容易であるため、ほとんどの J2EE アプリケーションでプログラムセキュリティよりも奨励されています。また、宣言セキュリティでは、セキュリティの責任はセキュリティの実装者にあるため、アプリケーションプログラマーはアプリケーション開発に専念でき、管理者はセキュリティポリシーの強化に専念できます。

Web アプリケーションセキュリティの概要

サーブレット API 仕様では、Web アプリケーションのセキュリティ問題が重視され、宣言セキュリティとプログラムセキュリティを実装するメソッドが定義されます。J2EE は、JAAS (Java Authentication and Authorization Service：Java 認証承認サービス)、Java セキュリティマネージャ、およびポリシーファイルを使用して、ユーザーのアクセスコントロールを強化して Web サーバーロールに関連付けます。

安全な Web アプリケーションを効果的に実装するには、次の概念を理解する必要があります。

- **認証**　ユーザー証明 (ユーザー名とパスワード) を収集してシステム内で検証する処理。この処理では、データベース、フラットファイル、LDAP 実装などのユーザーレポジトリを使用して証明を確認し、ユーザーが当人であることを認証する必要があります。
- **承認**　認証済みユーザーに、特定のリソースの表示またはアクセスを許可する処理。ユーザーにリソースを表示する権利が承認されていない場合はアクセスもできません。

J2EE セキュリティロール

システム内の認証と承認を処理するために、J2EE セキュリティは複数の領域を網羅しています。各領域は、組織内でそれぞれ異なる担当者によって処理されます。次の表で、各セキュリティタスクと、そのタスクを実装する担当者について説明します。

タスク	担当者	説明
ロールの解決と宣言セキュリティ	アプリケーションのアセンブル担当者	プログラマーが定義するロールを解決してシステムロールにリンクします。また、アプリケーションのアセンブル担当者は、Web デプロイメントディスクリプタと EJB デプロイメントディスクリプタで宣言セキュリティを指定します。
ロールの定義とプログラムセキュリティ	アプリケーション開発者	Web アプリケーションや EJB レベルで適用されるロールを定義します。アプリケーション開発者は、アプリケーションのリクエストに基づいて、プログラムセキュリティをオプションで実装できます。
クライアントのコーディング	アプリケーション開発者	Web アプリケーションと EJB クライアントが必要な証明 (通常はユーザー ID とパスワード) を適切な時間に渡すことができるようにします。Web アプリケーションクライアントによって、ユーザー ID とパスワードを要求するプロンプトが表示されます。EJB クライアントは、呼び出し側の Web アプリケーションコンポーネントからの証明を使用するか、ユーザー ID とパスワードをプロパティとして InitialContext コンストラクタに渡します。JMS クライアントは、jrun-resources.xml ファイル内でユーザー ID とパスワードを指定できます。
セキュリティアーキテクチャとユーザーストア管理	システム管理者	ユーザーストアを管理し、グローバルロールの定義を指定し、サイト特有のセキュリティ環境に対応するように JRun セキュリティをカスタマイズします。

管理者が行うタスクとセキュリティ管理に関するその他のトピックについては、『JRun 管理者ガイド』を参照してください。デプロイメントディスクリプタについては、『JRun アセンブルとデプロイガイド』を参照してください。

Web アプリケーション認証の理解

セキュリティはインターネット上でデプロイされるアプリケーションにとって重要です。インターネットアプリケーションに関するセキュリティ問題に対処するため、Java サーブレット API には、Web アプリケーション内部のリソースへのユーザーアクセスを制御する認証メカニズムが定義されています。JRun では、Java サーブレット API に基づいて、最新のセキュリティメカニズムをサポートしています。

認証メカニズムはロールベースです。つまり、Web アプリケーションにアクセスするすべてのユーザーに 1 つ以上のロールが割り当てられます。ロールの例は、manager、developer、および customer です。

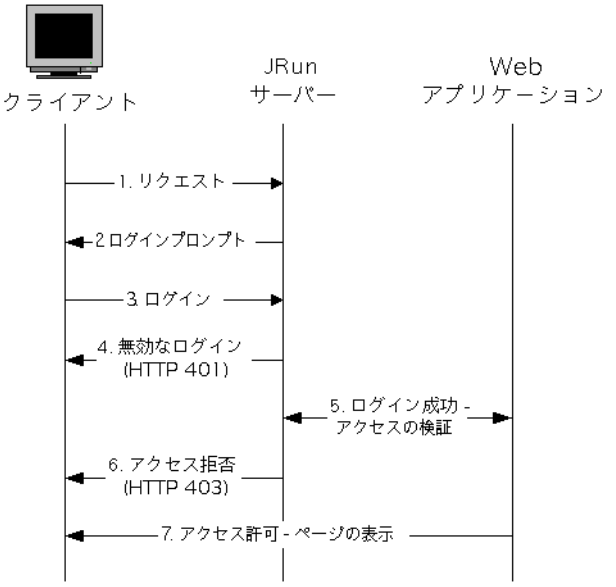
アプリケーション開発者は、Web アプリケーション、またはアプリケーションを構成する各リソースに用途別ロールを割り当てることができます。JRun は、ユーザーに Web アプリケーションリソースへのアクセス権を与える前に、ユーザーが認証されていること (つまりログインしていること)、そしてリソースへのアクセス権を持つロールに割り当てられていることを確認します。Web アプリケーションへのアクセスが認可されないと、HTTP 401 (未認可) ステータスコードが表示されます。

認証では、ユーザーに関する情報を Web サイトに保管する必要があります。情報には、各ユーザーに割り当てられているロールが含まれています。また、ユーザーアクセスを認証する Web サイトは通常、ログインメカニズムを実装し、パスワードによって各ユーザーの ID を検証します。Web サイトは、ユーザーを検証した後で、そのユーザーのロールを判断できます。

認証はリクエストがあるたびに行われます。JRun サーバーは Web アプリケーションに対するすべてのリクエストをチェックし、そして認証します。

認証の例

このセクションの例では、認証メカニズムがどのように機能するかを示します。この例では、developer のロールに割り当てられたユーザーだけが Web アプリケーションにアクセスできます。次の図で、基本的な認証メカニズムで起こる一連のリクエストとレスポンスを示します。



次に示す手順でこのメカニズムを説明します。

1 ユーザーが Web アプリケーションリソースをリクエストします。

Web アプリケーションでは、アプリケーションからページが返される前にリクエストが認証される必要があります。アプリケーション開発者は、Web アプリケーションレベルでの認証条件を設定します。JRun サーバーで実行される Web アプリケーションは、認証を個別に有効または無効に設定できます。

2 アプリケーションを実行するアプリケーションサーバーはリクエストをトラップし、ユーザーにログインをリクエストします。

ユーザーが既にログインしている場合、アプリケーションサーバーはこの手順をスキップします。

3 ログインするためには、ユーザー名とパスワードを入力して、アプリケーションサーバーに送り返します。

4 ログインが無効な場合、JRun からユーザーに HTTP 401 (未認可アクセス) ステータスコードが返されます。一定回数のログインが試行されて 401 コードが返されると、一部のブラウザではエラーページが表示されます。

5 ユーザー名とパスワードが有効であれば、アプリケーションサーバーは、アプリケーションへのアクセスに必要なロールにユーザーが割り当てられているかどうかを検証します。

6 ユーザーがリソースへのアクセス権を持っていない場合、JRun からユーザーに HTTP 403 (禁止) ステータスコードが返されます。

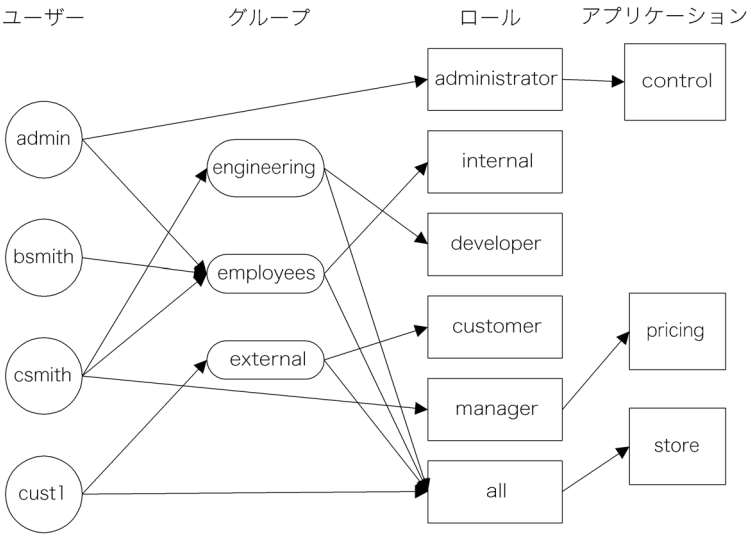
7 ユーザーがアプリケーションへのアクセス権を持っている場合、アプリケーションサーバーはリクエストされたページを返します。

この例からわかるように、Web アプリケーションと、アプリケーションを実行するアプリケーションサーバーが連動することによって認証は実行されます。アプリケーションは認証が必要かどうかを指定し、必要な場合は、アプリケーションへのアクセスに必要なユーザーのロールを指定します。JRun サーバーは、Web アプリケーションが認証を必要としていることを認識して、ユーザーアクセスを検証するメカニズムを実施します。

ユーザー、グループ、ロールの理解

認証はユーザーに割り当てられているロールに基づいて実行されます。ユーザーが Web アプリケーションにアクセスするには、アプリケーションへのアクセスが承認されているロールに割り当てられている必要があります。

ユーザー、グループ、ロールの 3 つのエンティティから構成される階層内にユーザーを配置します。次の図に、この階層を示します。



この図でわかるように、ユーザーはグループまたは直接ロールに割り当てることができます。グループを使用すると、ユーザーを 1 つにまとめて、そのユーザーグループ全体を特定のロールに割り当てることができます。ユーザーが、enigineering、employees、external の 3 つのグループに分類されています。ユーザーとグループは、administrator、internal、developer、customer、manager、all の特定のロールに割り当てられます。

アプリケーション認証とサーバー認証の比較

認証には 2 つの処理があります。Web サイトで Web アプリケーション認証を行うには、どちらも実装する必要があります。

認証の最初の部分はアプリケーションレベルで行われます。アプリケーション開発者は、アプリケーションへのアクセス権を持つロールを割り当てます。認証のこの部分は定義ステージと考えることができます。アプリケーション開発者は、アプリケーションへのアクセスに必要なアクセスロールを定義します。アプリケーションの認証権の設定の詳細については、88 ページの「Web アプリケーション認証の使用」を参照してください。

認証の 2 つめの処理は、アプリケーションを実行するサーバーによって行われます。アプリケーションサーバーは、ユーザーの証明を検証します。通常はログインメカニズムによってユーザーを確認して、Web アプリケーションへのユーザーのアクセス権を認証します。この処理は実施ステージと考えることができます。認証に関するアプリケーションサーバーの設定の詳細については、『JRun 管理者ガイド』を参照してください。

これらの 2 つの認証ステージは互いに独立しています。アプリケーション開発者は、Web アプリケーションを実行するアプリケーションサーバーが実際にどのように認証を行っているかを知る必要はありません。開発者に関係があるのは、アクセス権を指定する部分だけです。

JRun 認証メカニズムの設定

JRun サーバーの認証メカニズムを構成するセキュリティモジュールを選択するには、JMC を使用します。デフォルトの認証メカニズムはサーブレットの仕様で定義され、ユーザーストアとも呼ばれる、ロールとユーザーが保管されているファイルを含んでいます。

デフォルトのユーザーモジュールは JRun デフォルトユーザーモジュール、デフォルトのロールモジュールは JRun デフォルトロールモジュールです。ユーザーとロールは各 JRun サーバーの /<JRun のルートディレクトリ >/servers/< サーバー名 >/SERVER-INF/jrun-users.xml ファイルに保管されます。このファイルはデフォルト認証メカニズムのユーザーストアとして機能します。

JMC の [JRun ユーザーマネージャ] パネルと [JRun ロールマネージャ] パネルを使用して、選択した認証モジュールのユーザーやロールを追加または削除します。

1 台の JRun サーバーでは複数のアプリケーションを実行できるので、JRun サーバー内にあるアプリケーションへのアクセス権を持つユーザーは、同じサーバー内にある他のアプリケーションにも同じアクセス権でアクセスできます。

認証メカニズムは、デフォルトまたは独自のメカニズムのいずれも実装することができます。JRun には、デフォルトのメカニズムに加えて JMC で次の選択肢が用意されています。

- JDBC ログインモジュール
- LDAP ログインモジュール
- Windows ログインモジュール
- カスタム JAAS ログインモジュール

ロールマネージャやユーザーマネージャの使用の詳細については、『JRun 管理者ガイド』または JMC のオンラインヘルプを参照してください。

Web アプリケーション認証の使用

認証では、アプリケーション開発者が Web アプリケーションに対して定義したロールを、アプリケーションを実行するサーバーに適用する必要があります。このセクションでは、アプリケーションの開発時に、ロールとその他の認証情報をアプリケーションに設定する方法を説明します。

アプリケーションの開発とデプロイの一部として、次のアプリケーション認証を設定する必要があります。

1 アプリケーションへのアクセスロール

2 リソースの保護

3 アプリケーションサーバー検証メソッド

Web アプリケーションのデプロイメントディスクリプタである web.xml には、アプリケーション認証を管理する設定が含まれています。このファイルは、/<JRun のルートディレクトリ>/servers/<サーバー名>/<アプリケーション名>/WEB-INF ディレクトリ内にあります。

アクセスロールとリソースの設定

Web アプリケーションへのアクセスは、ユーザーがアプリケーションやアプリケーションリソースにアクセスするときに必要なロールを設定することで制御します。リソースにアクセスするには、ユーザーはそのリソースへのアクセス権を持つロールに割り当てられている必要があります。

web.xml ファイル内の `security-constraint` 要素や `web-resource-collection` 要素を使用して、アプリケーション全体や、その中のリソースにアクセス制限を割り当てることができます。

次の表で、これらの要素のシンタックスを説明します。

要素	シンタックス
security-constraint	<security-constraint> <(web-resource-collection*, auth-constraint†, user-data-constraint†)> </security-constraint>
web-resource-collection	<web-resource-collection> (web-resource-name, description†, url-pattern‡, http-method‡) </web-resource-collection>

* サブ要素を 1 つ以上定義できます。
† サブ要素をオプションで指定できます。
‡ サブ要素は指定しないことも、1 つ以上指定することもできます。

`web-resource-name` 要素は指定する必要があります。

認証ロールの例

次のアプリケーションの web.xml ファイルの抜粋では、Store アプリケーションにアクセスできるロールが定義されています。

```
<web-app>
  ...
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Store Application</web-resource-name>
      <url-pattern>/store/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
      <description> 販売情報リソース </description>
    </web-resource-collection>
    <auth-constraint>
      <role-name>manager</role-name>
      <description> 管理者専用 </description>
    </auth-constraint>
  </security-constraint>
  ...
</web-app>
```

この例では、security-constraint 要素を使用して次の情報を定義しています。

- アクセスを制限する Web アプリケーションリソースの URL パターン。
- /store を含んでいる URL を持つすべてのアプリケーションリソースのアクセスレベル。アプリケーション全体を認証するには、URL パターンを * に設定します。
- アプリケーションリソースの HTTP アクセスメソッド。この場合には、GET と POST の両方が設定されます。http-method 要素を省略すると、すべてのアクセスメソッドが認証されます。
- URL へのアクセスロールが manager だけであること。

アプリケーションにアクセスできるロールのリストを拡張できます。次の例では、アプリケーションにアクセスできるロールに developer のロールを追加します。

```
...
<auth-constraint>
  <role-name>manager</role-name>
  <role-name>developer</role-name>
  <description> 管理者と開発者 </description>
</auth-constraint>
...
```

認証ロールの選択的な割り当て

アプリケーション内のすべてのリソースに認証を適用する代わりに、一部のアプリケーションリソースを選択して、認証ロールを割り当てることができます。次の例では、アプリケーションの inventory ディレクトリにあるサーブレットと、アプリケーションの pricing ディレクトリにあるリソースにのみ認証を割り当てます。

```
<web-app>
  ...
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Store Application</web-resource-name>
      <url-pattern>/store/inventory/*</url-pattern>
      <url-pattern>/store/pricing/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
      <description> 販売情報リソース </description>
    </web-resource-collection>
    <auth-constraint>
      <role-name>manager</role-name>
      <description> 管理者専用 </description>
    </auth-constraint>
  </security-constraint>
  ...
</web-app>
```

検証メソッドの設定

アプリケーションサーバーでアプリケーションへのユーザーのアクセス権を認証するには、このサーバーでユーザーを識別できなければなりません。ユーザーを識別すると、サーバーは、ユーザーに割り当てられているロールやユーザーのアクセス権を識別できます。

保護された Web アプリケーションリソースにユーザーが最初にアクセスしようとすると、アプリケーションサーバーは、ユーザー名とパスワードを使用してログインするようにユーザーに要求します。このログイン情報から、サーバーはユーザーのロールを判別できます。

アプリケーションの web.xml ファイルには、次の 2 通りの検証メソッドを設定できます。これらのメソッドによって、アプリケーションサーバーからユーザーにログイン情報がリクエストされる方法が決まります。メソッドは次のとおりです。

- **BASIC**　HTTP リクエスト/レスポンスのメカニズムを使用してユーザーをログインさせます。
- **FORM**　アプリケーション開発者がデザインしたカスタムのログインフォームを使用します。

メモ：Java サーブレット API は、ユーザー検証を行う 4 つのメソッドを定義しています。今回の JRun リリースでは、`BASIC` と `FORM` の 2 つだけがサポートされます。

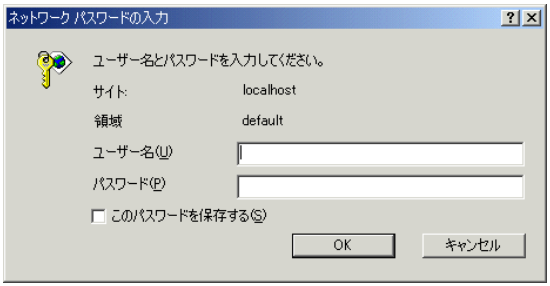
BASIC 検証の使用

BASIC 検証は、HTTP リクエスト / レスポンスのメカニズムを使用して現在のユーザーを認証します。このタイプの検証は、次の一連のイベントによって行われます。

1 ログインしていないユーザーが Web アプリケーションへのアクセスを試みます。

2 JRun がユーザーのブラウザに HTTP 401 (未認可アクセス) ステータスコードを返します。

3 ブラウザに、ユーザー名とパスワードの入力を要求するプロンプトが表示されます。ブラウザには、次のようなログインプロンプトが表示されます。



4 ユーザーはユーザー名とパスワードを入力します。

5 ブラウザから JRun サーバーにログイン情報が返され、認証が行われます。

ユーザー名とパスワードが有効な場合、JRun サーバーは、ユーザーがそのアプリケーションへのアクセス権を持っていることを認証し、リクエストされたページを返します。無効な場合、JRun は HTTP 403 (禁止) ステータスコードをユーザーに返します。

ユーザーが認証処理をキャンセルすると、サーバーから HTTP 401 (未認可) ステータスコードが返されます。

85 ページの「認証の例」の図でもこの手順を示しています。

アプリケーションが BASIC 認証を使用することを指定するには、Web アプリケーションの web.xml ファイル内の **login-config** 要素と **auth-method** サブ要素を使用します。次の例では、Sales 領域内の認証メソッドを BASIC に設定します。

```
<web-app>
  ...
  <login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>Sales</realm-name>
  </login-config>
  ...
  <security-constraint>
  ... //security contstraints
  </security-constraint>
  ...
</web-app>
```

各 Web アプリケーションに設定できる認証メソッドは 1 つだけです。

realm-name プロパティを省略すると、アプリケーションのホストであるサーバー名が領域名として使用されます。

ステータスコードの処理

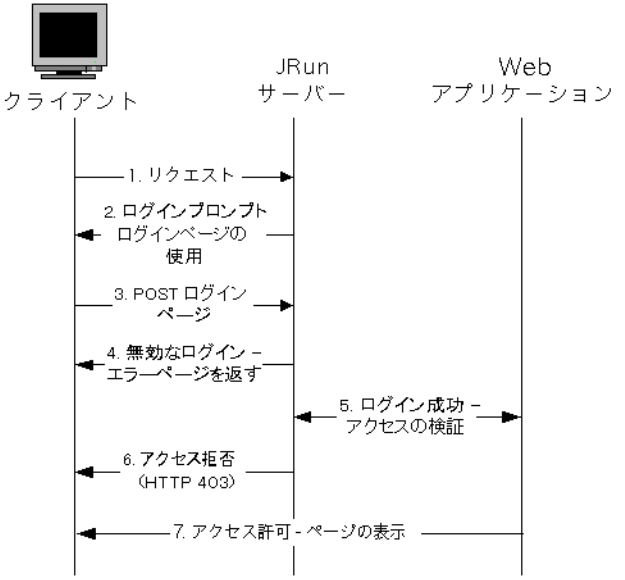
Web サーバーからクライアントに HTTP 401 (未認可) や 403 (禁止) のステータスコードが返されると、一部の Web サーバーはデフォルトでエラーページを送信します。Web サーバーによるステータスコードの処理方法は、Web アプリケーションの web.xml ファイルで無効にすることができます。これよって、ログインを処理するページをカスタマイズすることができます。

次の例は、無効にされたエラーコードと、ログインに失敗したユーザーに転送されるカスタムのエラーページを示しています。

```
<web-app>
...
<error-page>
  <error-code>401</error-code>
  <location>/errorPages/unauthorized.jsp</location>
</error-page>
<error-page>
  <error-code>403</error-code>
  <location>/errorPages/forbidden.jsp</location>
</error-page>
...
</web-app>
```

FORM 検証の使用

FORM 検証では、HTML フォームを使用してユーザー名とパスワードを取得できます。次の図に、FORM 検証の手順を示します。



次の例に示すように、web.xml ファイルには、ログインフォームとエラーフォームの両方を指定できます。

```
<web-app>
  ...
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/login.htm</form-login-page>
      <form-error-page>/loginerror.htm</form-error-page>
    </form-login-config>
  </login-config>
  ...
</web-app>
```

ユーザーが Web アプリケーションをリクエストすると、JRun サーバーは **form-login-page** 要素で指定されているページにユーザーを転送します。次の HTML ページ login.htm は、ログインページの例です。

```
<html><head><title> ログインページ </title></head>
<body>
<center><h2> 保護されているページがリクエストされました。ログインしてください。</h2>
<form method="POST" action="j_security_check">
<table>
<tr><td> ユーザー名 </td><td><input type=text name="j_username"></tr>
<tr><td>パスワード</td><td><input type=password name="j_password"></tr>
</table>
<br><br>
<input type=submit>
</form>
</center></body></html>
```

フォームには次の設定が含まれている必要があります。

- **アクション**　フォームは、**j_security_check** のアクションを使用して **POST** メソッドとして送信される必要があります。アプリケーションを実行するサーバーは、アクションを認識してフォームを処理します。
- **ユーザー名**　ユーザー名は **j_username** というフィールドに保管します。
- **パスワード**　パスワードは **j_password** というフィールドに保管します。

ユーザー名とパスワードが有効な場合、JRun サーバーは、ユーザーがそのアプリケーションへのアクセス権を持っていることを認証し、リクエストされたページを返します。無効な場合、JRun は 403（禁止）ステータスコードをユーザーに返します。ユーザー名やパスワードが無効な場合は、ユーザーは **form-error-page** 要素で指定されているページに転送されます。

プログラムセキュリティの実装

各リソースにプログラムセキュリティを追加することによって、それらのリソースへのアクセスを詳細に管理できます。サーブレット API に含まれているメソッドを使用すると、サーブレットと JSP に呼び出しを含めて、現在のユーザーと、そのユーザーが与えられたロールに属しているかどうかを判断できます。これによって、実行時に条件付きでコンテンツを作成できます。

メモ：サーブレットの指定では、サーブレット自体に認証制限を作成しません。認証制限はアプリケーションレベルで指定します。この指定では、サーブレットに適用される認証制限をそのサーブレットが認識できます。

このセクションでは、サーブレットや JSP のコンテンツを条件付きで変更するサーブレット API の使用例を示します。選択したサーブレットのグループに適用できるサーブレットフィルタでも、このセクションで説明するテクニックを使用できます。プログラムセキュリティと宣言セキュリティの組合せによって、コードの再利用が促進され、アプリケーションのメンテナンスが容易になります。フィルタの詳細については、179 ページの第 7 章「フィルタ」を参照してください。

リクエストメソッドの理解

サーブレットの HttpServletRequest オブジェクトのメソッドを使用するか、JSP ページの作成時に request オブジェクトのメソッドを使用して、サーブレットをリクエストしたユーザーの情報を取得します。次の表で、HttpServletRequest オブジェクトで利用できるセキュリティ関連のメソッドを説明します。

メソッド	説明
`getAuthType`	サーブレットを保護する認証メソッドを返します。可能性のある戻り値は、BASIC と FORM です。サーブレットが認証を必要としない場合は、null を返します。
`getRemoteUser`	現在のユーザー名を文字列で返します。認証されているユーザーがいない場合は、null を返します。
`isUserInRole (String role)`	現在のユーザーが、与えられた論理的なロール名の一部である場合は、Boolean タイプを true に設定して返します。現在のユーザーが認証されていない場合は、false を返します。
`getUserPrincipal`	現在のユーザーを表わす java.security.PrincipalString オブジェクトを返します。または、認証されているユーザーがいない場合は、null を返します。

認証されているユーザーがいない場合、`isUserInRole` メソッドは false を返します。プログラムセキュリティのためのサーブレット API のメソッドは、ユーザーの認証には使用できません。認証は、`resource-constraint` 要素を使用してリソースにアクセスを試みることによってユーザーが開始する必要があります。

ロール情報の使用

サーブレットの本文では、次の例に示すように、ユーザーのロールを判別してサーブレットを条件付きで実行できます。

```
...
public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
  ...
  if(req.isUserInRole("manager")) {
    // manager として処理します。
  }
  if(req.isUserInRole("all")) {
    // 他のすべてのユーザーとして処理します。
  }
...
```

JSP ページ内では、次の例を使用してページを条件付きで処理できます。

```
<%
  if(req.isUserInRole("manager")) {
    // manager として処理します。
  }
  if(req.isUserInRole("all")) {
    // 他のすべてのユーザーとして処理します。
  }
%>
```

移植可能ロールの作成

サーブレット API を使用することにより、アプリケーションでプログラムして使用するロール名から、ユーザーレポジトリに保管されたロール名へのクロスリファレンスを作成できます。このクロスリファレンスは、web.xml 内に作成します。

security-role-ref 要素を使用することにより、サーブレット内にハードコードされたロール名や、サーブレットコンテナによって role-link 要素として使用されるロール名を定義できます。これで、修正や再コンパイルを行わなくても、さまざまなロール名を使用するコンテナにサーブレットをデプロイできるようになります。

たとえば、マネージャのために MGR というロール名を使用するサーブレットコンテナと、manager のロールをチェックするサーブレットを考えます。web.xml 内のサーブレット定義には次のコードが含まれます。

```
...
<servlet>
  <servlet-name>myServlet</servlet-name>
  <servlet-class>myServlet</servlet-class>
  <security-role-ref>
    <role-name>manager</role-name>
    <role-link>MGR</role-link>
  </security-role-ref>
</servlet>
...
```

次のコードに示すように、isUserInRole が呼び出されると、サーブレットコンテナはこのクロスリファレンスを実行します。

```
if (request.isUserInRole("manager")) {
   … // manager 専用のロジックを追加します。
}
```

サーブレット開発者はロールを表す語句を使用でき、その語句をアプリケーション開発者が指定したロール名にリンクできます。アプリケーション開発者がアプリケーションに関連付けられているロールを修正した場合でも、JSP 開発者は web.xml ファイル内のロールのリンクを変更するだけで済みます。

ロールのリンクによって、サーブレットは、role-name タグで定義されているロールに基づいて常に条件付きで処理を行います。role-link タグを使用して、Web アプリケーションの一部としてこのようなサーブレットを使用することにより、サーブレットのロール定義をアプリケーションの定義に関連付けることができます。

セキュアな Web アプリケーションの作成

このセクションでは、Web アプリケーションのセキュリティを強化するコードを書く場合に利用できるテクニックを説明します。セキュリティ問題の詳細については、105 ページの「リソース」にリストされているリソースを参照してください。

クライアントによるなりすましの防止

IP アドレスと HTTP ヘッダーは、ホストベースの認証を実行するためによく使用されます。たとえば、次の例のように、Referer ヘッダーやクライアントの IP アドレスをチェックして、リクエストが信頼できるソースからのものであるということを確認できます。

```
...
String remoteaddr = request.getRemoteAddr();
if (remoteaddr.startsWith("10.")) {
  System.out.println(" リクエストを送信したホストは認証されました ");
  chain.doFilter(request, response);
} else {
  RequestDispatcher rd = request.getRequestDispatcher("/errorPages/
        forbidden.jsp");
  context.log(" リクエストを送信したホストは認証されませんでした ");
  rd.forward(request, response);
}
...
```

Referer などのリクエストヘッダーは簡単に偽造できます。クライアントは、ヘッダーを設定したり IP アドレスを偽造したりすることで、別のものになりすますことができます。

クライアントのなりすまし問題の解決策は、HTTP ヘッダーデータを認証メカニズムとして使用しないことです。

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

ステート情報の維持

ステート情報を提供する一般的な方法は、クエリ文字列パラメータ、非表示のフォームフィールド、または Cookie を使用して、クライアントサイドのステート情報を保管することです。たとえば、userID パラメータをクエリ文字列に追加し、その userID を Cookie に保管するか、または userID を非表示のフォームフィールドとしてページに含めます。クライアントが信頼できるものでなく、ステート情報に簡単にアクセスできるため、これらの方法はいずれも攻撃の対象になる可能性があります。

クラッカーは、多くの場合、クエリ文字列パラメータを使用して、Web アプリケーションの弱点をテストします。たとえば、クライアントが次の URL をリクエストしたとします。

```
http://www.hamsteak.com/accountBalance?userID=42
```

クライアントはリクエストを送信する前に、userID パラメータの値をブラウザで自由に変更できます。

HTML リクエストを生成するクライアントは、非表示のフォームフィールドを偽造できます。ブラウザに表示された Web アプリケーションの HTML ページのソースを表示するだけで、非表示のフォームフィールドは見つかってしまいます。クライアントは、どのようなリクエストも自由に作成できるので、リクエストを送信する前に、リクエスト文書の非表示のフォームフィールドを追加または削除できます。

Cookie は、通常クライアントに保管されたプレーンテキストファイルなので、悪質なデータを混入できます。さらに、クライアントは、リクエストの実行前に HTTP の `Cookie` ヘッダーを変更できるので、Cookie の存在は、信頼性を示す決定的証拠にはなりません。

次のテクニックを利用すれば、ステート情報を維持しながら、アプリケーションの信用を損なう危険性を軽減することができます。

- クエリ文字列パラメータ、非表示のフォームフィールド、Cookie データは、重要でないデータのみに使用します。
- ステートの維持には、HttpSession オブジェクトを使用します。JRun は長いランダムなセッションキーを生成し、セッションの実装に応じて、クライアントに Cookie として渡すか、URL に追加するか、あるいは非表示のフォームフィールドとして追加します。セッションデータを使用する場合は、この ID がクライアントとサーバーの間で渡される唯一のデータです。JRun はこの ID を使用して、サーバーが保管するセッションデータを参照します。
- セキュアな Cookie を使用します。jrun-web.xml ファイルで、HTTPS や SSL などのセキュアなプロトコルを使用した場合にのみ Cookie を送信できるよう指定できます。詳細については、155 ページの「jrun-web.xml ファイルでのセッション設定」を参照してください。

クライアントサイドでの検証

フォームを不正なデータ入力から保護するには、クライアントサイドの検証を信用しないでください。JavaScript や VBScript でフォームフィールド検証を実装するのは、一般的な方法です。たとえば、電話番号に数値しか入力できないようにしたり、フォームフィールドに不正な文字を入力できないようにすることができます。ただし、ユーザーがクライアントアプリケーションでこれらのスクリプト言語をオフにできるので、クライアントサイドの検証が行われることを前提にすることはできません。

ダイナミック Web ページでクライアントが提供したデータを使用する場合は、99 ページの「サーバーサイドでの検証」で説明するように、サーバーサイド検証や文字列操作を使用して、最初にデータを検証し、有害なコードを取り除く必要があります。

データベースの保護

データベースクエリでは、ユーザーが入力データをそのまま使用できないようにします。たとえば、次の例で示されているように、姓などのキーワードを入力するようにユーザーに要求し、そのキーワードを自分のクエリで使用するようにします。

```
String lname = request.getParameter("lastname");
sqlstmt = "SELECT firstname, lastname FROM users WHERE lastname =
        ¥"lname¥"";
```

検証がない場合、ユーザーはページを送信する前に、クエリ文字列やフォームフィールドの値を自由に変更できます。アタッカーが使用するテクニックの 1 つは、悪質なクエリをフォームフィールドに挿入することです。これを SQL インジェクションアタックと呼びます。たとえば、ユーザーが次のクエリをフォームフィールドに追加することがあります。

```
jones; select * from users;
```

データベースによっては、クエリが両方とも実行されることがあります。

ユーザーが入力したデータをそのままデータベースクエリに追加する必要がある実装では、次のテクニックを 1 つ以上試みて、不正なクエリの処理を防止します。

- ユーザー入力のサイズを制限します。詳細については、99 ページの「文字列の操作」を参照してください。
- 不正な文字を削除します。詳細については、100 ページの「HTML コードの除去」を参照してください。
- データベースのアクセス許可の設定を定義して実行します。詳細については、データベースのドキュメントを参照してください。

サーバーサイドでの検証

このセクションでは、クライアントによるダイナミックページへの有害コードの挿入を防止するテクニックを説明します。

文字列の操作

String クラスの標準メソッドを使用して、長すぎる入力を切り詰めたり、ユーザー入力から不正な文字を削除できます。ネガティブなフィルタリング (不正な文字をフィルタ) とポジティブなフィルタリング (正しい文字以外はすべてフィルタ) を使用できます。

次のネガティブなフィルタリングの例では、入力から & 文字と * 文字を削除して、下線に置き換えています。

```
String filteredQuery = req.getParameter("rogueQuery");

...
// 不正なクエリを切り詰めます。
if (filteredQuery.length()>20) {
  filteredQuery = filteredQuery.substring(0,20);
}
// 不正なクエリにある不正な文字を有効な文字で置き換えます。
char good = '_';
char bad[] = new char[2];
bad[0] = '&';
bad[1] = '*';
for (int i=0; i<bad.length; i++) {
  filteredQuery = filteredQuery.replace(bad[i],good);
}
...
```

Perl 言語で有名になった正規表現も、値の置換や変換に利用できる強力なツールです。次のリソースでは、Java の正規表現メソッドへのアクセスが提供されます。

- **Regexp カスタムタグライブラリ**　JSP で Perl の正規表現シンタックスをエミュレートするタグを提供します。詳細については、http://jakarta.apache.org/taglibs/doc/regexp-doc/intro.html をご覧ください。
- **java.util.regex パッケージ**　J2SE 1.4 には、正規表現を使用するためのコアクラスが含まれています。詳細については、http://java.sun.com/j2se/ をご覧ください。

HTML コードの除去

クライアントによるダイナミック Web ページへのコンテンツの追加を許可すると、サーバーはデータの処理方法を管理できなくなります。ユーザーによって送信されたコンテンツは、サーバーコードを攻撃したり、他のクライアントのコンピュータに転送されて表示される危険性があります。クライアントデータはすべて信頼できないと見なして、ユーザーが送信したデータは、サーブレットや JSP に取り込む前に無害なものに変換する必要があります。

たとえば、ユーザーが Web サイトに関するコメントを追加できるようにする場合は、HTML や JavaScript で使用される特殊文字を除去して HTML エンティティで置き換えるコードを使用する必要があります。HTML コードをクライアントブラウザに送信されるようにして、Web サーバーでは処理されないようにします。

次のコード例では、クライアントの入力から <、>、&、" の文字を探し、これらの文字を表わす HTML エンティティに置き換えています。

```
...
public static final char lt = '<';
public static final char gt = '>';
public static final char amp = '&';
public static final char quot = '"';
...
filteredHTML = htmlCodeFormat(rogueHTML);
...
public String htmlCodeFormat(String filteredHTML) {
  int thisHtmlLength = filteredHTML.length();
  StringBuffer thisText = new StringBuffer(Math.round(thisHtmlLength *
        1.5f));
  for(int count = 0; count <thisHtmlLength; count++) {
    char thisChar = filteredHTML.charAt(count);
    switch (thisChar) {
      case lt:
        thisText.append("&lt;");
        break;
      case gt:
        thisText.append("&gt;");
        break;
      case amp:
        thisText.append("&amp;");
        break;
      case quot:
        thisText.append("&quot;");
        break;
      default:
        thisText.append(thisChar);
        break;
    } //switch の終わり
  } //for の終わり
  return thisText.toString();
}
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

HTML エンティティのリストについては、http://www.ramsch.org/martin/uni/fmi-hp/iso8859-1.html をご覧ください。

CERT advisory CA-2000-02 には、信頼できないコンテンツによる Web サイトの悪用を防止するための処置が記載されています。詳細については、http://www.cert.org/tech_tips/malicious_code_mitigation.html や http://www.cert.org/advisories/CA-2000-02.html をご覧ください。

データベースのアクセス許可の設定

Web アプリケーションのデータを保護するには、クライアントから不正なクエリ出力が返されないようにするだけでなく、不正なクエリが実行されないようにしてください。クライアントの関心は必ずしも結果を表示することではなく、結果を表示せずにデータベースを破壊するクエリを送信することにある場合もあるからです。

たとえば、次の SQL ステートメントでは、テーブルからすべてのデータが削除され、続いてデータベースからテーブルが削除されます。

```
DROP FROM users;
```

ユーザーは、データベースを変更するこのコマンドの出力を見る必要はありません。このような攻撃を防止するには、データベースのアクセス許可を読み取り専用に変更し、特定の書き込み操作のみを許可します。詳細については、データベースのドキュメントを参照してください。

ユーザーパスの指定

アプリケーションの安全性を確保し、ユーザーによる重要なデータへのアクセスを防ぐ方法には、指定された順序でページにアクセスするようにして、クライアントが認証メカニズムを回避できないようにすることが含まれます。リクエスト属性に値を設定するクラスを使用し、JSP でその属性をチェックする場合は、必ず最初のページをリクエストしないと 2 番めのページをリクエストできないように設定できます。

次の例では、logcheck.jsp ページや logme.jsp ページが WEB-INF ディレクトリに保管されているので、ユーザーはそれらのページにアクセスできません。logme.jsp ページでは、**isLoggedIn** 属性が true に設定されています (より堅牢な実装では、なんらかの認証メカニズムを含みます)。ユーザーは直接このページにアクセスできないので、セッション値の設定は侵入者から保護されます。

次のコード例で、logme.jsp ファイルを示します。

```
<% session.setAttribute("isLoggedIn", "true"); %>
<HTML><BODY>
<A HREF="TargetResource.jsp"> 保護リソースに移動します </A>
... //html body
</BODY></HTML>
```

保護リソース (TargetResource.jsp) の一番上にあるコードには、**isLoggedIn** の値をチェックするファイルが含まれます。チェックに合格した場合は、ターゲットリソースが返されます。チェックに不合格した場合は、ページから例外が送信され、JRun はユーザーにエラーページを転送します。次のコードで、TargetResource.jsp ページを示します。

```
<%@ include file="/WEB-INF/logcheck.jsp" %>
<HTML><BODY>
おめでとうございます ... あなたはログインしています。
</BODY></HTML>
```

logcheck.jsp ページは TargetResource.jsp ページに含まれ、**isLoggedIn** 属性をチェックして、ユーザーがログインしたかどうかを判断します。次のコードは、logcheck.jsp ファイルを示します。

```
<%
  String isLoggedIn = (String) session.getAttribute("isLoggedIn");
  if (isLoggedIn.equals("true")) {
  } else {
   throw new Exception(" ページはアプリケーション内から呼び出されませんでした。");
  }
%>
```

ターゲットリソースには、**include** ディレクティブを持つ logcheck.jsp ファイルが含まれます。これには、**include** アクション (**jsp:include**) を使用しません。それは、このアクションによって、インクルードするファイルにリクエストが戻され、**isLoggedIn** チェックを実行した後で、保護されたページのコンテンツの表示を防ぐことができないからです。

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

リソースへの直接アクセスの防止

タグライブラリ、JSP ソースページ、クラス、およびその他の重要なデータを、Web アプリケーションの WEB-INF ディレクトリに保管します。クライアントは、このディレクトリのリソースに直接アクセスできません。

ServletContext オブジェクトの **getResource** メソッドや **getResourceAsStream** メソッドを使用すると、Web アプリケーションのコンポーネントからデータにアクセスできます。101 ページの「ユーザーパスの指定」の例では、logcheck.jsp ページを WEB-INF ディレクトリに保管することによって、このテクニックを使用します。

フィルタを使用して、承認や認証の宣言レイヤーを Web アプリケーションに追加することもできます。フィルタ使用の詳細については、179 ページの第 7 章「フィルタ」を参照してください。

エラーのキャッチ

コードから生成されるエラーはすべてキャッチできるようにしてください。エラー出力から、物理パス、プラットフォームの詳細、クラス名や変数名などの、Web アプリケーションが動作しているシステムに関する重要情報が明らかになることがあります。

コードからは、ランタイムエラーに加えてコンパイル時エラーも生成されることがあるので、JSP 作成時のエラーのキャッチは特に重要です。JSP をプリコンパイルするとコンパイル時エラーをキャッチできるので、これはよい習慣ですが、運用環境で使用する場合は、その必要はありません。

JSP の例外処理メカニズムの使用の詳細については、第 10 章を参照してください。

コメントの削除

コメントにはしばしば、重要な情報を開発者に説明する実装の詳細が書かれています。悪意のあるユーザーは、システムへの攻撃に使用する情報をコメントから収集することがあります。

適切なレベルのコメントを使用するか、JSP や HTML ページからコメントをすべて削除してから、アプリケーションを運用環境にデプロイします。

JSP で使用できるコメントには 2 種類あります。

- **コンテンツコメント**　JSP ページの出力に含まれます。ユーザーは、ページのソースを表示すると、ブラウザでこのコメントを読むことができます。コンテンツコメントのシンタックスは次のとおりです。

  ```
  <!-- クライアントが表示可能なコメント -->
  ```

- **JSP コメント**　JSP ページの出力に含まれません。これらのコメントが表示されるのは、JSP のソースコードや JRun が生成するサーブレットの中だけです。ユーザーは、ページのソースでは、このタイプのコメントを見ることはできません。JSP コメントのシンタックスは次のとおりです。

  ```
  <%-- ソースコードのみのコメント %>
  ```

また、次のようなスクリプトレットに挿入するコメントは、エンドユーザーには見えません。

```
<%
  /* 次のステートメントで、ユーザーテーブルから userID を取得します */
  String sqlstmt = "select userID from users where lastname =
        request.getParameter("lastname")";
%>
```

価値あるログエントリの作成

ServletContext では `log` メソッドを利用できるので、Web アプリケーションに有用なログエントリを生成できます。これは、クライアントが未認可のリソースにアクセスを試みているかどうかを判断するのに役立ちます。JRun は、`log` メソッドの出力を /<JRun のルートディレクトリ >/servers/< サーバー名 >/logs/< サーバー名 >-event.log ファイルに書き出します。

コンテキストオブジェクトの `log` メソッドを使用するには、次の例で示されているように、サーバーの /<JRun のルートディレクトリ>/servers/<サーバー名>/SERVER-INF/jrun.xml ファイルでデバッグレベルのロギングを有効にする必要があります。

```
<attribute name="debugEnabled">true</attribute>
```

使用可能なメソッドを使用して、リクエストしているクライアントの IP アドレス、`Referer` ヘッダー、クエリ文字列などのエントリをログに書き込みます。次のコード例で、価値あるログエントリの生成に有用なメソッドを示します。

```
...
ServletContext sc = this.getServletContext();
String[] info = new String[12];

info[0] = sc.getServletContextName();
info[1] = request.getMethod();
info[2] = getServletName();
info[3] = request.getContextPath();
info[4] = request.getRequestURI();
info[5] = request.getHeader("referer");
info[6] = request.getQueryString();
StringBuffer sb = request.getRequestURL();
info[7] = sb.toString();
info[8] = request.getHeader("Authorization");
info[9] = request.getRemoteAddr();
info[10] = request.getRemoteHost();
info[11] = request.getHeader("User-Agent");

sc.log("--------------------NEW REQUEST--------------------");

for (int i=0; i<info.length; i++) {
  sc.log(info[i]);
  out.println(info[i] + "<BR>");
}
...
```

このコード例では、次の出力に類似したログエントリが作成されます。

```
-----------------------NEW REQUEST---------------------------
02/28 10:43:51 debug JRun Default Web Application
02/28 10:43:51 debug GET
02/28 10:43:51 debug TestLog
02/28 10:43:51 debug
02/28 10:43:51 debug /servlet/TestLog
02/28 10:43:51 debug http://localhost:8100/JumpToTestLog.jsp
02/28 10:43:51 debug color=red&name=nick
02/28 10:43:51 debug http://127.0.0.1:8100/servlet/TestLog
02/28 10:43:51 debug
02/28 10:43:51 debug 127.0.0.1
02/28 10:43:51 debug 127.0.0.1
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

ロギングをフィルタとして実装することによって、すべての Web コンポーネントでログエントリを標準化できます。またこうすることで、ロギングメカニズムは、サーブレットと JSP に対して透過性を持つことができるようになります。フィルタ作成の詳細については、179 ページの第 7 章「フィルタ」を参照してください。

ロギングのレベルを上げるとコストがかさみます。`ServletContext.log()` への呼び出しでは、いずれも少量の処理リソースが使用されます。JRun のロギング実装方式では、ロギングの作業はログサービスに渡されますが、ここでもシステムリソースが少し消費されます。

ログファイルエントリの形式作成の詳細については、JMC のオンラインヘルプを参照してください。独自のロギングサービス設定の詳細については、『JRun 管理者ガイド』を参照してください。

リソース

このセクションでは、ご使用のアーキテクチャや設定に適用できるリソースを示します。

Java セキュリティリソース

次の表は、セキュアな Java コードの作成に役立つテクニカルリソースをリストします。

リソース	格納場所
Java Security (2nd Edition) by Scott Oaks	O'Reilly & Associates
Java Security homepage	http://java.sun.com/security/
Security Code Guidelines	http://java.sun.com/security/seccodeguide.html
Professional Java Security by Jess Garms, Daniel Somerfield	Wrox Press
Secure a Web Application Java-style by Michael Cymerman	http://www.javaworld.com/javaworld/ jw-04-2000/jw-0428-websecurity.html
Java Authentication and Authorization Service (JAAS)	http://java.sun.com/products/jaas/
Java Security Tutorial by David Wheeler	http://www.dwheeler.com/javasec/
jGuru Security FAQ	http://www.jguru.com/faq/Security
A Security Infrastructure for Distributed Java Applications	http://www.cs.princeton.edu/sip/pub/ placeless.pdf

Web セキュリティ

次の表は、安全な Web アプリケーションのデザインに役立つ一般的なリソースのリストします。

リソース	格納場所
Dos and Don'ts of Client Authentication on the Web	http://www.pdos.lcs.mit.edu/papers/ webauth:sec10.pdf
SANS Institute 20 Most Critical Internet Security Vulnerabilities	http://www.sans.org/top20.htm
Best Practices for Secure Web Development	http://www.cert.pl/PDF/secure_webdev-3.0.pdf
CERT Coordination Center Tech Tips	http://www.cert.org/tech_tips/
CERT Advisories	http://www.cert.org/advisories/
HTTP Authentication (RFC 2617)	http://www.rfc.net/rfc2617.html
HTTP File Upload (RFC 1867)	http://www.rfc.net/rfc1867.html

一般的なセキュリティリソース

次の表は、アプリケーションプログラミングのセキュリティのロールを理解するために役立つリソースをリストします。

リソース	格納場所
Fred Cohen & Associates	http://www.all.net/
Auditors Checklists	http://www.all.net/books/audit/
Microsoft Security homepage	http://www.microsoft.com/security/default.asp
The Basics of Security	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/iis/deploy/config/server03.asp
Security Policy Issues	http://rr.sans.org/policy/policy_list.php
How to Design Secure Web Applications	http://www.macromedia.com/v1/handlers/index.cfm?ID=20014
Formal Trust and Authentication	http://www.macromedia.com/v1/handlers/index.cfm?id=20457
Hack Proofing Your Network: Internet Tradecraft by Ryan Russell and Stace Cunningham	Syngress Media (Editor)
Building Secure Software: How to Avoid Security Problems the Right Way by John Viega and Gary McGrawGary McGraw	Addison Wesley Professional

パート II
サーブレットプログラミング

パート II では、JRun によるサーブレットプログラミングについて説明します。次の章で構成されています。

第 6 章
サーブレットのプログラミングテクニック

この章では、JRun コンテナで実行されるサーブレットのプログラミングテクニックを説明します。サーバーサイド Java プログラミングと J2EE スイートの基礎知識があることを前提としています。Java サーブレットの経験がまったくない場合は、『JRun 入門』をお読みください。

目 次

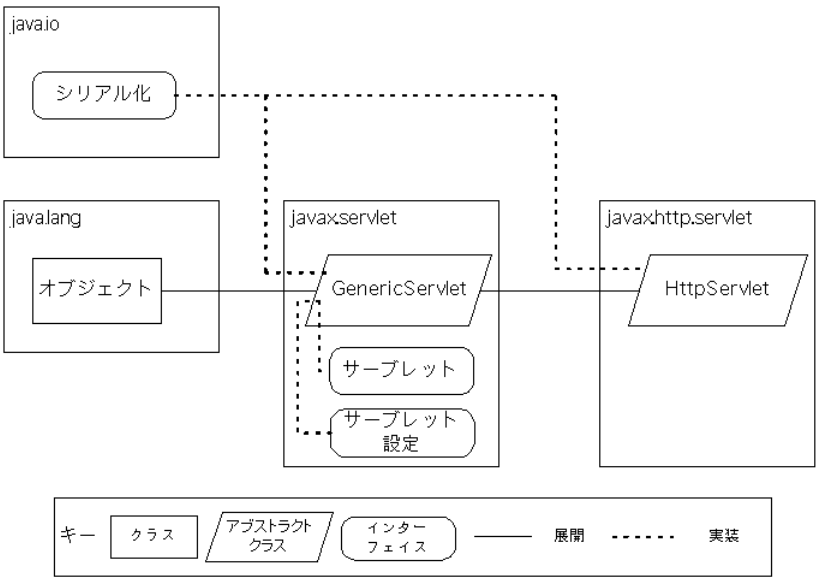
サーブレット API

サーブレットクラスは、javax.servlet.Servlet インターフェイスを実装する必要があります。サーブレット API には、このインターフェイスを実装し、Java サーブレットを作成するときに拡張できる次の 2 つのクラスが含まれています。

- GenericServlet　プロトコルに依存しない基本サーブレット機能を提供します。このクラスを拡張して、HTTP 以外のサービスをコーディングします。ただし、ほとんどの場合クラスは HttpServlet から拡張します。
- HttpServlet　GenericServlet を拡張して HTTP 特有の機能を追加します。ほとんどのクラスが HttpServlet を拡張します。また、この章では、クラスで HTTP 処理を実行することを想定しています。

基本サーブレットクラスおよびインターフェイス

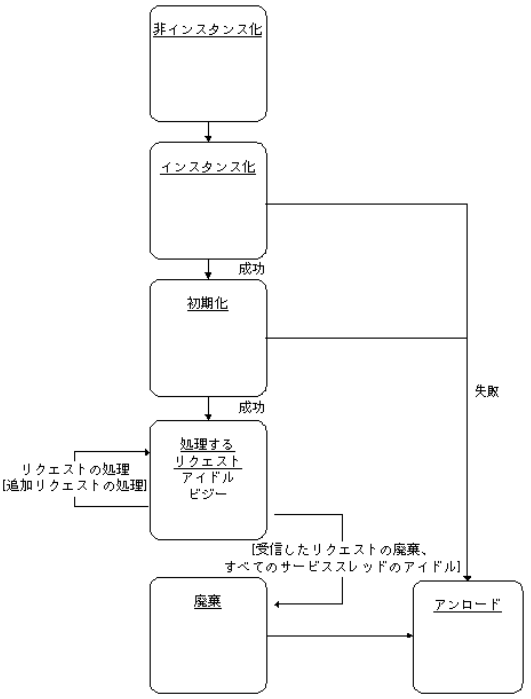
次の図で示すように、サーブレット API は、java.io、java.lang、javax.servlet、javax.http.servlet パッケージにあるクラスやインターフェイスを使用します。



サーブレット API のクラス、インターフェイス、および例外の詳細については、『Servlet API JavaDocs』を参照してください。

サーブレットのライフサイクル

次の図で示すように、サーブレットのライフサイクルにはいくつかのフェーズがあります。



初期化フェーズで、JRun はサーブレットの **init** メソッドを呼び出します。**init** メソッドは一度だけ呼び出されます。したがって、JRun がサーブレットを呼び出すたびに貴重なリソースを割り当てる必要はありません。**init** メソッドは、サーブレットがライフサイクル全体で必要とするリソースを割り当てるために使用します。JRun は、起動時 (JMC 内で有効にされている場合) や最初のクライアントリクエストを受け取ったときに **init** メソッドを呼び出します。

サーブレットクラス内で **init** メソッドをオーバーライドする必要はありません。初期化時に何らかのタイプの処理を実行する場合にのみ、このメソッドをオーバーライドします。

サーブレットが受信したすべてのリクエストを処理するとき、リクエスト処理フェーズに入ります。JRun は、GenericServlet をオーバーライドする場合は、**service** メソッドにリクエストを転送し、HttpServlet クラスをオーバーライドする場合は **do*Xxx*** メソッドにリクエストを転送します。

JRun サーバーをシャットダウンするときや、変更されたサーブレットを JRun サーバーがリロードするときは、JRun はサーブレットをメモリからアンロードします。アンロード時に、サーブレットは廃棄フェーズに入ります。廃棄リクエストの開始時に、JRun は **destroy** メソッドを呼び出します。**destroy** メソッド内で、サーブレットが使用していたリソースの割り当てを解放できます。JRun が **destroy** メソッドを呼び出すと、クラスはすぐにガーベッジコレクションを実行できます。

サーブレット API のパッケージ

サーブレット API は、Sun Microsystems の Java Software Division によって公開された仕様です。この仕様には、Java サーブレットと Java サーブレットをサポートするサーバーによってリクエストされるクラス、メソッド、動作の概要が記載されています。JRun Version 4.0 は、Java サーブレット仕様のバージョン 2.3 をサポートしています。

サーブレット API には、次のパッケージが含まれています。

- `javax.servlet`
- `javax.servlet.http`

JRun には詳細なサーブレット API オンラインドキュメントが添付されています。これは <JRun のルートディレクトリ>`/docs/api` にあります。また、最新の API ドキュメントは http://java.sun.com/products/servlet からアクセスできます。

javax.servlet

次の表に示すように、`javax.servlet` パッケージには、すべてのサーブレットに適用されるインターフェイス、クラス、例外が含まれています。

javax.servlet インターフェイス

次の表で、`javax.servlet` のインターフェイスを説明します。

インターフェイス	説明
`RequestDispatcher`	リクエストの処理を他のサーブレット、JSP、または HTML ファイルに転送するオブジェクトを定義します。 また、レスポンスに他のサーブレットの出力を含めることもできます。
`Servlet`	サーブレットインスタンスを初期化し、リクエストを処理し、サーブレットインスタンスを廃棄するメソッドを定義します。`GenericServlet` クラスはこのインターフェイスを実装します。
`ServletConfig`	サーブレットの `init` メソッドに情報を渡すオブジェクトを定義します。名前 / 値のペア、サーブレット名、Web アプリケーションの `ServletContext` オブジェクトへのリファレンスが含まれています。
`ServletContext`	サーブレットがサーブレットコンテナに関する情報にアクセスするために使用するメソッドを定義します。また、アプリケーションの init パラメータが含まれています。 1 台の仮想マシン上の Web アプリケーションごとに 1 つの `ServletContext` があります。
`ServletRequest`	クライアントリクエスト情報をカプセル化するオブジェクトを定義します。各インスタンスには、名前 / 値のペア、属性、入力ストリームが含まれています。
`ServletResponse`	クライアントに返す情報をカプセル化するオブジェクトを定義します。バイナリデータや文字データを送信できます。
`SingleThreadModel`	各サーブレットインスタンスに一度に 1 つのリクエストだけを実行させるオブジェクトを定義します。このインターフェイスの詳細については、『Servlet API JavaDoc』を参照してください。

javax.servlet クラス

次の表では javax.servlet のクラスを説明します。

クラス	説明
GenericServlet	プロトコルに依存しないサーブレット。通常はこのクラスではなく HttpServlet を使用します。
ServletInputStream	クライアントリクエストからバイナリデータを読み込むためのストリーム。通常、このクラスは使用しません。
ServletOutputStream	クライアントにバイナリデータを送信するためのストリーム。通常、このクラスは使用しません。

javax.servlet 例外

次の表では javax.servlet の例外を説明します。

例外	説明
ServletException	サーブレットの問題を示します。
UnavailableException	サーブレットが使用可能でないことを示します。この例外は、一時的に使用不能であるか、または永久的に使用不能であるかを示すために使用できます。

javax.servlet.http

javax.servlet.http パッケージには、HTTP の機能をリクエストするサーブレットに適用されるインターフェイスとクラスが含まれています。次の表で、これらを説明します。

javax.servlet.http インターフェイス

次の表では javax.servlet.http のインターフェイスを説明します。

インターフェイス	説明
HttpServletRequest	ServletRequest を HTTP サーブレット用に拡張します。このオブジェクトは、リクエストについて Cookie 、属性、およびその他の情報にアクセスするために使用します。
HttpServletResponse	ServletResponse を HTTP サーブレット用に拡張します。このオブジェクトは、ブラウザにレスポンスを送信するために使用します。
HttpSession	複数のページリクエストの間で保持されるユーザー情報が含まれています。セッションはセッション ID をキーとして識別され、セッション ID は Cookie の中や URL パラメーターによって維持されます。
HttpSessionBinding Listener	オブジェクトがセッションにバインドまたはバインド解除されたときに、オブジェクトにその旨を通知します。このインターフェイスは、セッション固有のリソースを初期化およびクリーンアップするために使用します。

javax.servlet.http classes

次の表では javax.servlet.http のクラスを説明します。

クラス	説明
Cookie	クライアント Cookie の作成、読み取り、変更をサーブレットに実行させます。Cookie を読み取るには、HttpServletRequest オブジェクトを使用します。Cookie は、HttpServletResponse オブジェクトを使用して設定します。
HttpServlet	HTTP サーブレットを作成するために拡張するアブストラクトクラス。このクラスはすべての Web 指向のサーブレットに使用します。
HttpSessionBindingEvent	2 つのメソッド getName と getSession が含まれています。 HttpSessionBindingListener インターフェイスを実装しているオブジェクトがセッションにバインドされたり、バインドを解除されたりしたときに、そのオブジェクトに渡されます。
HttpUtils	便利なユーティリティメソッドが含まれています。メソッドには、getRequestURL、parsePostData、parseQueryString があります。

HttpServlet の使用

HttpServlet クラスは、GenericServlet クラスを拡張します。開発するサーブレットクラスの大半で、HttpServlet クラスが拡張されます。HttpServlet を使用してプログラミングを行うには、service メソッドをオーバーライドするか、1 つ以上の次の HTTP 特有のリクエスト処理メソッドをオーバーライドする必要があります。

- doGet
- doPost
- doPut
- doDelete
- doHead
- doOptions
- doTrace

これらのメソッドはそれぞれ次のパラメータを取ります。

- HttpServletRequest。HTTP ヘッダーと、その他のクライアントリクエスト情報が含まれています。
- HttpServletResponse。ブラウザに HTML を返すことができます。

service メソッドのオーバーライド

JRun では、GenericServlet を拡張するサーブレットに対してサーブレットリクエストが行われるたびに、service メソッドが呼び出されます。service メソッドをオーバーライドする必要があります。また、オプションで init メソッドや destroy メソッドをオーバーライドすることもできます。

HttpServlet を拡張するサーブレットの場合は、service メソッドのデフォルトの実装で、リクエストが適切な doXxx メソッドに転送されます。ここで Xxx はリクエストのタイプ (GET または POST) です。たとえば、サーブレットで HTTP GET リクエストが受信されると、service メソッドによって doGet メソッドが呼び出されます。デフォルトの service メソッドをオーバーライドする場合、サーブレットは HTTP リクエストの全種類を処理するか、適切な doXxx メソッドにリクエストをディスパッチするロジックを含む必要があります。

doGet メソッドのオーバーライド

JRun は doGet メソッドを、HTTP GET リクエストのために呼び出します。ユーザーが URL を入力、リンクをクリック、または method=GET を指定するフォームを送信した場合、Web ブラウザにより HTTP GET リクエストが送信されます。GET メソッドは、ページをリクエストするための最も一般的なメソッドです。

次の例は、doGet メソッドをオーバーライドするサーブレットを示しています。

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

public class DisplayInfo extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html; charset=UTF-8");
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title> 情報の表示 ");
```

```
  out.println("</title></head><body>");
  out.println("リクエスト URI：" + req.getRequestURI() + "<br>");
  out.println("</body></html>");
}
}
```

doPost メソッドのオーバーライド

JRun は、HTTP POST リクエストのために doPost メソッドを呼び出します。ユーザーが method=POST を指定するフォームを送信した場合、Web ブラウザにより HTTP POST リクエストが送信されます。

次の例は、POST リクエストを使用してサーブレットを呼び出す HTML フォームを示しています。

```
<html><head><title>システムへのログイン</title></head>
  <body bgcolor="Silver">
    <h1>システムへのログイン</h1>
    <!-- ログインフォームの表示 -->
      <form action="/servlet/selectionForm" method="POST">
        <p>Name: 
        <input type="Text" name="myName" size="30">
        <p>
        <input type="Submit" value="Log In">
      </form>
  </body>
</html>
```

次の例は、doPost メソッドを使用して、渡された値にアクセスするサーブレットを示しています。

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

public class SelectionForm extends HttpServlet {

public void doPost(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html; charset=UTF-8");
  PrintWriter out = resp.getWriter();
  String thisName = "Unknown Name";
  // ログインフォームからユーザー名を取得します。
  String[] attrArray = req.getParameterValues("myName");
  // 呼び出し側フォームには、myName の値が 1 つしかないと想定します。
  if(attrArray != null) {
    thisName = attrArray[0];
  }
  out.println("<html><head><title>表示する情報を選択");
  out.println("</title></head><body>");
  out.println("<h1>Welcome " + thisName + "</h2>");
  out.println("<h2>表示する情報を選択</h2>");
  // ¥ を使用して、二重引用符をエスケープします。
```

```
  out.println("<form action=¥"/servlet/displayInfo¥"
        method=¥"post¥">");
  // HTTP リクエスト情報を表示するチェックボックス
  out.println("<p> リクエスト情報を表示しますか ? ");
  out.println("<input type=¥"Checkbox¥" name=¥"requestInfo¥" checked>");
  // Cookie 情報を表示するチェックボックス
  out.println("<p>Cookie を表示しますか ? ");
  out.println("<input type=¥"Checkbox¥" name=¥"showCookies¥"
        checked>");
  out.println("<br>");
  out.println("<input type=¥"Submit¥">");
  out.println("</form>");
  out.println("</body></html>");
}
}
```

doGet メソッドと doPost メソッド両方のオーバーライド

ほとんどのサーブレットは、doGet メソッドに応答します。ただし、開発者が doPost メソッドを処理することを忘れ、それからサーブレットにリクエストをポストするフォームを追加した場合は、サーブレットがエラーを返さないようにする必要があります。これを避けるための最も一般的な方法は、デフォルトで doPost メソッドから doGet メソッドに制御を渡すことです。

次の例は、両方のメソッドを実装したサーブレットを示しています。

```
import javax.servlet.*;
import javax.servlet.http.*;
public class GetServletConfigInfo extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException, ServletException {
    ... // ここでサーブレットを実装します。
  }
  public void doPost(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException {
    doGet(request, response);
  }
}
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

その他の HTTP メソッドのオーバーライド

次の表で説明するように、HttpServlet クラスには、追加の HTTP リクエストタイプをサポートするメソッドが用意されています。HTTP バージョン 1.1 ではすべてのリクエストタイプがサポートされていますが、HTTP バージョン 1.0 では GET、HEAD、POST だけがサポートされています。

リクエストタイプ	メソッド	コメント
DELETE	doDelete	ターゲットのリソースを削除するようにサーバーにリクエストします。 DELETE リクエストの詳細については、HTTP のドキュメントを参照してください。
PUT	doPut	リソースをリクエストします。 PUT リクエストの詳細については、HTTP のドキュメントを参照してください。
HEAD	doHead	doGet メソッドを実行しますが、返すのはヘッダーだけです。
OPTIONS	doOptions	サポートされているオプションのリストを返します。通常、このメソッドをオーバーライドする必要はありません。
TRACE	doTrace	すべてのヘッダーのリストを返します。通常、このメソッドをオーバーライドする必要はありません。

GenericServlet クラスにおけるメソッドのコーディング

GenericServlet クラスでは、Servlet インターフェイスを実装することにより、HTTP でないサーブレットに機能を提供します。HttpServlet クラスは GenericServlet をオーバーライドするため、これらのメソッドは HttpServlet を拡張するサーブレットでも使用できます。

サーブレットで GenericServlet クラスを拡張する場合、このサーブレットは service メソッドをオーバーライドします。必要に応じて、サーブレットで getServletInfo、init、destroy の各メソッドをオーバーライドすることもできます。さらに、GenericServlet クラスには、サーブレット、リクエスト、アプリケーション情報にアクセスするためのメソッドが含まれています。

GenericServlet クラスのメソッドの詳細については、サーブレット API のドキュメントを参照してください。

Web アプリケーションとサーブレットのマッピング

Web アプリケーションコンポーネントをリクエストの URL にマッピングする方法によって、ユーザーのエントリポイントが定義されます。マッピングをどのように設定するかによって、ユーザーのブラウザのアドレスバーの表示や JRun によるリクエストの処理方法が変わります。このセクションでは、暗黙の JRun マッピングを説明し、サーブレットや JSP などの Web アプリケーションや Web コンポーネントに独自のマッピングを確立する方法を説明します。

Web アプリケーションのマッピングを確立するには、そのアプリケーションのアセンブルとデプロイの方法を理解する必要があります。運用環境では、モジュールを適切なアーカイブファイルにパッケージしてデプロイすることをお勧めします。詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

マッピングのクイックスタート

すばやくスタートするには、次の表の情報を使用します。

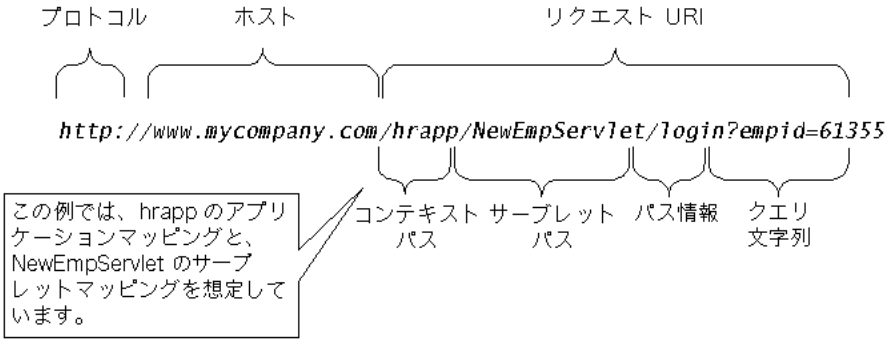
マッピングのトピック	説明
デフォルトのサーブレットマッピング	各 JRun サーバーの default-web.xml ファイルには、次のデフォルトのサーブレットマッピングが含まれています。 FileServlet / classes ディレクトリ内のサーブレット /servlet JSP ページ *.jsp JST *.jst Axis サーブレット *.jws Axis サーブレット /services

デフォルトのアプリケーションマッピングの設定	次の例に示すように、/META-INF/application.xml ファイルに、Web モジュール用に / に設定したコンテキストルートを追加します。
	`<application>`
	`  <display-name>MyApp</display-name>`
	`  <module>`
	`  <web>`
	`  <web-uri>default-war</web-uri>`
	`  <context-root>/</context-root>`
	`  </web>`
	`  </module>`
	`</application>`

デフォルトのサーブレットマッピングの設定	次の例に示すように、web.xml ファイルや default-web.xml ファイルで servlet を定義し、servlet-mapping の url-pattern を / に設定します。
	`<servlet>`
	`  <servlet-name>MyServlet</servlet-name>`
	`  <servlet-class>MyServlet</servlet-class>`
	`</servlet>`
	`...`
	`<servlet-mapping>`
	`  <servlet-name>MyServlet</servlet-name>`
	`  <url-pattern>/</url-pattern>`
	`</servlet-mapping>`

URL の理解

クライアントは、他の Web リソースと同様に、URL をリクエストすることによってサーブレットを呼び出します。次の図で示すように、URL はプロトコル、ホスト、ポート (オプション)、リクエスト URI で構成されています。



リクエスト URL を分解するサンプルサーブレットを表示するには、Samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

JRun 4.0 は、サーブレット API バージョン 2.3 の仕様書に記述されている Web アプリケーションアーキテクチャを完全に実装しています。この実装には、仕様に準拠したサーブレットへのリクエストのマッピングが含まれています。

アプリケーションマッピングとサーブレットマッピングは、リクエスト URI をいくつかのコンポーネントに分割し、その URI の異なる部分を使用して、呼び出すリソースを決定します。次の表でこれらのコンポーネントを説明します。

URI コンポーネント	説明
コンテキストパス	Web アプリケーションマッピングに関連付けられたパス接頭辞を指定します。Web サーバーの URL ネーム空間のルートディレクトリにあるデフォルトのアプリケーションの場合、コンテキストパスは空の文字列になります。デフォルト以外のアプリケーションの場合、コンテキストパスは、スラッシュ (/) で始まりますが、スラッシュで終了しません。たとえば、/techniques は、/techniques を含んでいるリクエストを techniques アプリケーションへマッピングします。 `request.getContextPath` メソッドは、コンテキストパスを示す文字列を返します。
サーブレットパス	サーブレットマッピングと一致する URL の部分を指定します。このパスはスラッシュ (/) で始まります。 `request.getServletPath` メソッドは、サーブレットパスを示す文字列を返します。
パス情報	クエリ文字列パラメータの前にあるリクエストパスの残りの部分です。 `request.getPathInfo` メソッドは、パスの残りの部分を示す文字列を返します。

マッピングのタイプ

JRun では、リクエストに応えてサービスするファイルを決定するときに、次のタイプのマッピングを使用します。

- **アプリケーションマッピング**　Web アプリケーションの URL をアプリケーションを含む物理ディレクトリに関連付けます。
- **サーブレットマッピング**　サーブレットを、`/servlet` などの接頭辞、または `*.jsp` などの接尾辞に関連付けます。
- **ウェルカムファイルマッピング**　リクエストが既存のアプリケーションやサーブレットのマッピングと一致しない場合の Web アプリケーションの動作を定義します。通常は、ファイルを index.html または他のデフォルトページに関連付けます。

JRun サーバーで実行される各 Web アプリケーションは、1 つのアプリケーションマッピングと複数のサーブレットマッピングを定義できます。Web アプリケーションを最適な方法で使用するには、HTML ファイル、JSP、およびサーブレットに対するリクエストを処理するために、JRun がどのようにアプリケーションマッピングとサーブレットマッピングを使用するかを理解する必要があります。

次の表は、JRun 設定ファイルが定義するマッピングをリストします。

設定ファイル	マッピング
default-web.xml	サーブレットマッピングとウェルカムファイルマッピング。 default-web.xml ファイルの設定は、JRun サーバーのすべての Web アプリケーションに適用されます。
web.xml	サーブレットマッピングとフィルタマッピング。
jrun-web.xml	Web アプリケーションのための仮想パスマッピングとオプションのコンテキストルート (WAR ファイル)。
application.xml	エンタープライズアプリケーションのためのコンテキストルート (EAR ファイル)。

アプリケーションマッピングは、コンテキストパスを Web アプリケーションの名前とディレクトリパスに関連付けます。これらのマッピングは、JRun 管理コンソール (JMC) を使用して管理します。JRun は、アプリケーションマッピングを META-INF/application.xml ファイルで維持します。

このマッピングは、Web サーバーのファイルシステム内の Web アプリケーションの物理的位置と一致する必要はありません。たとえば、`myapp` が Web アプリケーションのドキュメントルートディレクトリの場合に、Web アプリケーションをサーバーのディレクトリ `c:/apps/myapp` に配置しているものとします。Web アプリケーションのディレクトリ構造は、`myapp` の下です。

この場合、Web アプリケーションが http://www.mycomp.com/myapp の形式で URL に応答するように、`/myapp` に対してアプリケーションの URL マッピングを作成できます。このマッピングを設定すると、`/myapp` コンテキストパスを含むすべての URL が Web アプリケーションにマッピングされます。

アプリケーションマッピングの理解

アプリケーションアセンブル担当者は、すべての Web モジュールのコンテキストルートを定義します。アプリケーションアセンブル担当者は、このコンテキストルートがアプリケーション内の他のどのモジュールのコンテキストルートとも重複しないようにします。

リクエストが到着すると、JRun はそのリクエストをできるだけ最長のマッピングにマッピングしようと試みます。つまり、リクエスト URI に /foo/bar が含まれている場合は、JRun は最初に /foo/bar のマッピングを見つけようとします。これに失敗すると、JRun は /foo のマッピングを見つけようとします。一致するものを見つけられない場合、JRun は、このリクエストを / にマッピングします。サーブレットが / にマッピングされると、JRun はサーブレットパス情報として foo/bar を割り当てます。

ルートへの Web モジュールのマッピング

JRun サーバーでルートにマッピングできる Web モジュールは 1 つだけであり、すべての Web モジュールに固有のマッピングが必要です。JRun サーバーに到着したリクエストに明示的なマッピングがない場合、リクエストはルートにマッピングされた Web アプリケーションに送信されます。

Web アプリケーションに application.xml ファイルがない場合は、このアプリケーションを EAR ファイルとしてパッケージし、このファイルを定義する必要があります。詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

ルートへのサーブレットのマッピング

Web アプリケーションのルートへのすべてのリクエストを 1 つのサーブレットでキャッチする場合は、このサーブレットが / に一致するように URL パターンを設定する必要があります。これは、モデル - ビュー - コントローラなどのデザインパターンを実装するときの一般的なタスクであり、この場合は、1 つのサーブレットがコントローラまたはディスパッチャとして機能する必要があります。

たとえば、techniques-ear ファイルには Web モジュール techniques-war が含まれています。このモジュールのコンテキストルートは /techniques です。JRun が http://www.yourhost.com/techniques などのリクエストを受け取った場合、このリクエストは techniques-war モジュールにマッピングされます。次に JRun は、サーブレットマッピングについて techniques モジュールの web.xml ファイルをチェックします。

マッピングのないリクエスト処理

アプリケーションアセンブル担当者が、コンテキストルートにルート (/) マッピングを与えておらず、リクエストが他のマッピングと一致しない場合は、JRun は、リクエストしているユーザーに Web サーバーのルートディレクトリからインデックスファイルを返します。利用可能なインデックスファイルがない場合、JRun はディレクトリリストを返します。ディレクトリリストがオフになっている場合、JRun はエラーをユーザーに返します。

アプリケーションマッピングの定義

デプロイするアプリケーションのタイプによっても異なりますが、Web アプリケーションのコンテキストルートを定義する方法は複数あります。このセクションでは、EAR ファイルと WAR ファイルのためにコンテキストルートを定義する方法を説明します。

EAR ファイル

EAR ファイルをデプロイする場合は、次のシンタックスを使用して、/META-INF/application.xml ファイルで各 Web モジュール用にコンテキストルートを定義します。

```
<module><web>
  <web-uri>module-name</web-uri>
  <context-root>mapping</context-root>
</web></module>
```

次の例では、techniques-war Web モジュールを /techniques にマッピングします。

```
<module><web>
  <web-uri>techniques-war</web-uri>
  <context-root>/techniques</context-root>
</web></module>
```

application.xml ファイル内で各 Web モジュール用に context-root 要素を設定する必要があります。コンテキストルートを指定しないと、Web モジュールはエンタープライズアプリケーションからアクセスできません。

WAR ファイル

Web モジュールを WAR ファイルとしてホットデプロイする場合は、jrun-web.xml ファイルでコンテキストルートを定義するか、JRun に定義させることができます。

JRun は次の順序でコンテキストルートを判断します。

1. /WEB-INF/jrun-web.xml ファイルでコンテキストルートのマッピングをチェックします。
2. WAR ファイルが未圧縮の場合は、JRun は WAR ファイルのルートディレクトリ名を使用します。たとえば、WAR ファイルが <JRun のルートディレクトリ >/servers/samples/worldmusic-war/ ディレクトリにデプロイされている場合は、JRun は worldmusic-war をコンテキストルートとして使用します。
3. WAR ファイルが圧縮されている場合は、JRun は WAR ファイルの名前から ".war" を取り除いたものを使用します。たとえば Web モジュールが techniques.war に含まれている場合は、JRun は "techniques" をコンテキストルートとして使用します。

サーブレットマッピングの理解

次の表で、サーブレットをリクエスト URI にマッピングするためのいくつかのテクニックを説明します。

テクニック	説明

デフォルトサーブレットマッピング	JRun サーバー上のサーブレットにアクセスするには、サーブレットを /WEB-INF/classes ディレクトリに保管しているかぎり暗黙の /servlet マッピングを使用できます。たとえば、MyServlet.class ファイルを /WEB-INF/classes に保管している場合は、http://yourhost/servlet/MyServlet をリクエストすることにより、このサーブレットをリクエストできます。
	次の例に示すように、このマッピングは default-web.xml ファイルであらかじめ定義されています。
	`<servlet-mapping>`
	`  <servlet-name>ServletInvoker</servlet-name>`
	`  <url-pattern>/servlet/</url-pattern>`
	`</servlet-mapping>`

web.xml	url-pattern 要素を使用すると、web.xml ファイルでサーブレットのマッピングを定義できます。次の例では、AxisServlet を /services にマッピングします。
	`<servlet-mapping>`
	`  <servlet-name>AxisServlet</servlet-name>`
	`  <url-pattern>/services</url-pattern>`
	`</servlet-mapping>`
	
	http://www.yourhost.com/services などのリクエストは、AxisServlet にマッピングされます。

default-web.xml	web.xml ファイルの場合と同じシンタックスを使用して、default-web.xml ファイルでサーブレットや JSP のマッピングを定義できます。ただし、次の例のように、default-web.xml で定義されたサーブレットや JSP は、この JRun サーバー内のすべてのアプリケーションで利用可能です。
	`<servlet>`
	`  <servlet-name>JRunStatistics</servlet-name>`
	`  <jsp-file>/jrunx/instrument/Results.jsp</jsp-file>`
	`</servlet>`

サーブレットマッピングは、サーブレットを URL パターンに関連付けます。URL パターンには、`/MyServlet` などの接頭辞、または `*.jsp` などの接尾辞を使用できます。指定された URL パターンに一致するサーブレットパスがリクエスト URI に含まれる場合、JRun は関連付けられたサーブレットを呼び出します。サーブレットを明示的に定義しない場合は、/servlet 暗黙マッピングを使用して WEB-INF/classes ディレクトリに保管してあるサーブレットをリクエストできます。詳細については、126 ページの「暗黙のサーブレットマッピング」を参照してください。

default-web.xml ファイルの次の行で、*.jsp のパターンを JSPServlet にマッピングし、AxisServlet を /services にマッピングします。

```
<servlet-mapping>
  <servlet-name>AxisServlet</servlet-name>
  <url-pattern>/services</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>JSPServlet</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
```

Web サーバーがページやサーブレットに対するリクエストを受信すると、JRun は、まずリクエスト URI のコンテキストパスを、JRun に定義されているアプリケーション URL マッピングと比較して、Web アプリケーションを検索します。Web アプリケーションが見つかると、JRun はその Web アプリケーションのサーブレットマッピングを使用して、指定されたリソースを検索します。コンテキストパスに一致するアプリケーションマッピングが見つからないと、JRun はその Web サーバーのデフォルトのアプリケーションのサーブレットマッピングを使用してリソースを検索します。

明示的サーブレットマッピングは、web.xml ファイルで定義および管理できます。セキュリティを最大にするには、運用アプリケーションで、Web アプリケーションの各サーブレットごとに明示的サーブレットマッピングを定義する必要があります。

サーブレットマッピングの `url-pattern` には、先頭のスラッシュは必要ありません。

暗黙のサーブレットマッピング

web.xml ファイルでの明示的サーブレットマッピングだけでなく、JRun は各 JRun サーバーの default-web.xml ファイルでの一連の暗黙のサーブレットマッピングを維持します。これらのマッピングは JRun サーバー内のすべてのアプリケーションによって共有されます。次の表で、これらのマッピングについて説明します。

サーブレット	クラス	URL マッピング
FileServlet	jrun.servlet.file.FileServlet	/
ServletInvoker	jrun.servlet.ServletInvoker	/servlet/
JSPServlet	jrun.jsp.JSPServlet	*.jsp
JSTServlet	jrun.jsp.JSTServlet	*.jst
AxisServlet	org.apache.axis.transport.http.AxisServlet	*.jws /services

Web アプリケーションの web.xml ファイルで新しいマッピングを定義することにより、その Web アプリケーションの暗黙のサーブレットマッピングをオーバーライドできます。つまり、default-web.xml ファイルのすべてのアプリケーションのマッピングを変更できます。たとえば、/servlet を LoginServlet または 404Servlet に関連付けることで、ServletInvoker サーブレットのデフォルトサーブレットマッピングを変更できます。

ServletInvoker の使用

JRun には、/servlet を ServletInvoker サーブレットに関連付けるサーブレットマッピングが含まれています。このサーブレットマッピングにより、サーブレットパス /servlet を含むリクエスト URI は、ServletInvoker サーブレットによって処理されるようになります。ServletInvoker サーブレットには、web.xml ファイルや default-web.xml ファイルでは明示的に定義されていないサーブレットのための汎用呼び出しメカニズムが用意されています。

次の部分では、default-web.xml ファイルの ServletInvoker のマッピングを示しています。

```
<servlet-mapping>
  <servlet-name>ServletInvoker</servlet-name>
  <url-pattern>/servlet/</url-pattern>
</servlet-mapping>
```

このマッピングは、開発フェーズやテストフェーズで役立ちます。これにより、サーブレットマッピングを明示的に定義する必要がなくなります。ServletInvoker サーブレットが、クラス名を使用して仮のサーブレット登録を自動的に作成するからです。

セキュリティとパフォーマンス上の理由から、すべてのサーブレットに対して明示的マッピングを定義して、運用システムのデフォルトの ServletInvoker マッピングをオーバーライドする必要があります。運用アプリケーションでは、カスタマイズしたサーブレットが常にエラーを返す場合に、たとえば次のように、そのサーブレットに /servlet のマッピングを関連付ける方法も考えられます。

```
/servlet = 404servlet
```

JRun は次の方法で ServletInvoker サーブレットを使用します。

1 リクエスト URI にアクセスします (例：`/app1/servlet/SnoopServlet`)。

2 コンテキストパスを抽出します (例：`/app1`)。

3 サーブレットパスを抽出します (例：`/servlet`)。

4 パス情報からサーブレット名を抽出します (例：`/SnoopServlet`)。

5 `ServletContext.getNamedDispatcher(`*`servletname`*`)` を呼び出して、サーブレットを呼び出します。

JRun は最初に Web アプリケーションの web.xml ファイルと default-web.xml ファイルを調べて、このサーブレットが定義されているかどうかを判断します。一致するサーブレットが見つからない場合、JRun は アプリケーションのクラスパスにリストされたディレクトリ内でサーブレットを検索し、サーブレットのクラス名を使用してインスタンスを作成します。

ウェルカムファイルマッピングの理解

JRun では、サーブレット仕様に従って web.xml で指定されるウェルカムファイルをサポートしています。次の例に示すように、web.xml ファイルでウェルカムファイルを定義します。

```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>welcome.html</welcome-file>
</welcome-file-list>
```

welcome-file は、前後のスラッシュがない URI の一部分です。クライアントがディレクトリへのリクエストを行ったときに、JRun は welcome-file エントリをリクエストされたディレクトリに追加し、一致するものが見つかった場合は、そのリクエストを転送します。一致する welcome-file が見つからない場合やサービスできない場合は、リスト内の次の welcome-file を使用しようとします。また、welcome-file エントリがないと、JRun はクライアントにディレクトリリストを返します。ディレクトリリストがオフになっていると、JRun はエラーを返します。

ディレクトリ表示をオフにする方法については、『JRun 管理者ガイド』を参照してください。

URI の例

JRun は、アプリケーションマッピングとサーブレットマッピングに従って、URL の URI 部分をコンテキストパス、サーブレットパス、追加パス情報に分割します。

たとえば、/hrapp のアプリケーションマッピングが application.xml ファイルにあり、/NewEmpServlet のサーブレットマッピングが web.xml ファイルにあり、リクエスト URI が /hrapp/NewEmpServlet/login?empid=61355 の場合、JRun は次のように判断します。

URI コンポーネント	定義
/hrapp	コンテキストパス
/NewAppServlet	サーブレットパス
/login	パス情報
empid=61355	クエリ文字列パラメータ

アプリケーションマッピングがなく URI が同じ場合、結果は次のようになります。

URL コンポーネント	定義
(空)	コンテキストパス
/hrapp/NewEmpServlet	サーブレットパス
/login	パス情報
empid=6139	クエリ文字列パラメータ

メモ：リクエスト URI とパス部分の URL エンコードが異なる場合を除き、次のステートメントは常に true になります。
RequestURI = contextpath + servletpath + pathinfo

コンテキストとパス情報へのアクセス

サーブレット内からコンテキストとパス情報にアクセスできます。次の例は、リクエストから ServletContext 情報を取得するメソッドの使用方法を示しています。

```
...
ServletContext context = getServletContext();
out.println("コンテキストパス：" + req.getContextPath());
out.println("<BR>サーブレットパス：" + req.getServletPath());
out.println("<BR>パス情報：" + req.getPathInfo());
out.println("<BR>実際のパス：" +
        context.getRealPath(req.getServletPath()));
...
```

次の例では、Referer HTTP ヘッダーも使用して、「Back」リンクを構築します。

```
String referer = req.getHeader("Referer");
out.println("<BR><A HREF=¥"" + referer + "¥">Back</A>");
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

サーブレットの処理

HttpServlet クラスと GenericServlet クラスには、情報にアクセスするメソッドおよびメッセージを記録するメソッドがあります。次の表で、これらのメソッドについて説明します。

メソッド	説明
getInitParameter	サーブレットの初期化パラメータの値を返します。
getInitParameterNames	サーブレットの初期化パラメータ名を返します。
getServletConfig	ServletConfig オブジェクトを返し、初期化パラメータやコンテキスト情報にアクセスできます。サーブレットメソッドにはこの情報にアクセスするメソッドがあるので、通常 、**getServletConfig** は使用しません。
getServletContext	ServletContext オブジェクトを返すので JRun と対話するメソッドにアクセスできます。
getServletInfo	サーブレット情報を String として返します。情報を返すようにするには、このメソッドをオーバーライドする必要があります。
getServletName	サーブレットのインスタンス名を String として返します。
log	ログファイルにメッセージを書き込みます。

初期化パラメータの使用

ServletConfig オブジェクトを使用して各サーブレット定義から、またアプリケーション内のすべてのサーブレットのパラメータを提供する ServletContext から初期化パラメータを取得できます。

サーブレット特有の初期化パラメータ

サーブレット特有の初期化パラメータを取得するには、次の例に示すように、ServletConfig オブジェクトの **getInitParameterNames** メソッドと **getInitParameter** メソッドを使用します。

```
...
  ServletConfig config = getServletConfig();
  Enumeration eParmNames = config.getInitParameterNames();
  while (eParmNames.hasMoreElements()) {
    String parm = (String) eParmNames.nextElement();
    out.println(" " + parm + ":" + config.getInitParameter(parm) +
        "<br>");
  }
...
```

サーブレット特有の初期化パラメータは、次の例に示すように、Web アプリケーションのデプロイメントディスクリプタの servlet 定義で設定します。

```
<web-app>
...
<servlet>
  <servlet-name>ConfigParams</servlet-name>
  <servlet-class>ConfigParams</servlet-class>
  <init-param>
    <param-name>datasourcename</param-name>
    <param-value>lightgoldenrodyellow</param-value>
  </init-param>
</servlet>
...
</web-app>
```

詳細については、132 ページの「ServletConfig オブジェクトの使用」を参照してください。

アプリケーション全体の初期化パラメータ

アプリケーション内のすべてのサーブレットが共有する初期化パラメータを取得するには、次の例に示すように、ServletContext オブジェクトの `getInitParameter` メソッドと `getInitParameterNames` メソッドを使用します。

```
...
ServletContext context = this.getServletContext();
Enumeration parmEnum = context.getInitParameterNames();
if (parmEnum.hasMoreElements()) {
  out.println("<h2>ServletContext Init Parameters</h2>");
}
while (parmEnum.hasMoreElements()) {
  String name = (String)parmEnum.nextElement();
  out.println("<b>"+name+": </b>");
  out.println(context.getInitParameter(name) + "<br>");
}
...
```

次の例に示すように、アプリケーション全体の初期化パラメータは、Web アプリケーションのデプロイメントディスクリプタで、servlet 定義の外側で設定します。

```
<web-app>
...
<context-param>
  <param-name>datasourcename</param-name>
  <param-value>fred</param-value>
</context-param>
...
</web-app>
```

ServletContext オブジェクトの使用

ServletContext オブジェクトを使用すると、アプリケーション情報を保管したり、アプリケーションのさまざまなコンポーネント間で情報を共有したりすることができます。

たとえば、アプリケーションが複数のサーブレット (Java で書かれたものや JSP として書かれたもの)、HTML タグ、およびデータベースで構成されているとします。これらのアプリケーションコンポーネント間で情報をやり取りするには、アプリケーションコンテキストを使用して、その情報を保管したり、取り出したりすることができます。コンテキストオブジェクトを通じて使用できる情報には、次のものがあります。

- リクエストに渡される属性
- 初期化パラメータ
- MIME タイプ
- バージョン情報
- パス情報

また、アプリケーションコンテキストには、Web サーバーへのアプリケーションの実装に伴って、アプリケーション情報が保管されます。この情報には、アプリケーションコンポーネントのファイルの位置、サーブレットの初期化パラメータ、バージョン情報などの、アプリケーション特有の情報が含まれます。

次の方法で、サーブレットや JSP のアプリケーション情報にアクセスできます。

- Java サーブレットでは、`javax.servlet.ServletContext` オブジェクトを使用します。
- JSP では、暗黙の JSP application オブジェクトを使用します。

ServletContext オブジェクトを使用して、アプリケーションに関する情報を保存したり、次の環境情報にアクセスできます。

- 初期化パラメータ
- MIME タイプ
- バージョン情報
- パス情報

サーブレットは `getServletContext` メソッドを使用して ServletContext オブジェクトのリファレンスを取得します。

次の例では、いくつかのサーブレットコンテキスト情報を表示します。

```
...
  ServletContext scntxt = this.getServletContext();
  out.println("サーバー情報："+ scntxt.getServerInfo() + "<br>");
  int majorVersion = scntxt.getMajorVersion();
  int minorVersion = scntxt.getMinorVersion();
  out.println("メジャーバージョン：" + majorVersion + "<br>");
  out.println("マイナーバージョン：" + minorVersion + "<br>");
  java.util.Enumeration parmEnum = scntxt.getInitParameterNames();
  if (parmEnum.hasMoreElements()) {
    out.println("<h2>ServletContext Parameters</h2>");
  }
  while (parmEnum.hasMoreElements()) {
    String name = (String)parmEnum.nextElement();
    out.println("<b>"+name+": </b>");
    out.println(scntxt.getInitParameter(name) + "<br>");
  }
```

```
  // ServletContext 属性
  java.util.Enumeration attrEnum = scntxt.getAttributeNames();
  if (attrEnum.hasMoreElements()) {
    out.println("<h2>ServletContext Attributes</h2>");
  }
  while (attrEnum.hasMoreElements()) {
    // 常に属性を適切なクラスにキャストします。
    String attrName = (String)attrEnum.nextElement();
    out.println("<b>" + attrName + ": </b>");
    out.println(scntxt.getAttribute(attrName) + "<br>");
  }
  // サーブレットの実際のパスを取得します。
  String path = req.getServletPath();
  out.println("<b>サーブレットのフルパス：</b>");
  out.println(scntxt.getRealPath(path) + "<br>");
  // ServletContext のログを記録します。
  Date now = new Date();
  scntxt.log("Testing ServletContext:" + now);
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

ServletConfig オブジェクトの使用

JRun は、初期化時にサーブレットに設定情報を渡します。設定情報には、初期化パラメータを示す名前/値のペア、およびサーブレットが実行されるコンテキストを示す ServletContext オブジェクトが含まれます。

サーブレットは `getServletConfig` メソッドを使用して ServletConfig オブジェクトへのリファレンスを取得します。次に、ServletConfig オブジェクトを使用して初期化パラメータにアクセスできます。

次の方法で、サーブレットや JSP の設定情報にアクセスできます。

- Java サーブレットでは、`javax.servlet.ServletConfig` オブジェクトを使用します。
- JSP では、暗黙の JSP config オブジェクトを使用します。

次の例では、`getServletConfig` メソッドを使用して、サーブレットの `init` メソッド内の ServletConfig オブジェクトを取得します。

```
...
public void init(ServletConfig config) throws ServletException {
  super.init(config);
  try {
    bgcolor = config.getInitParameter("bgcolor");
    System.out.println("bgcolor:" + bgcolor);
  } catch (Exception e) {
    System.out.println("error:" + e.toString());
  }
}
```

```
...
public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException, ServletException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title>Servlet Config</title>");
  out.println("</head><body bgcolor=¥"" + bgcolor + "¥">");
  out.println("<h1>Servlet Config</h1>");
  out.println("</body></html>");
}
...
```

メモ：`init` メソッドが再び呼び出されるように JRun を再起動する必要があります。このメソッドはサーバーの起動時に 1 回だけ呼び出されます。

次の web.xml ファイルの抜粋では、初期化パラメータを設定します。

```
<servlet>
  <servlet-name>GetServletConfigInfo</servlet-name>
  <servlet-class>GetServletConfigInfo</servlet-class>
  <init-param>
    <param-name>bgcolor</param-name>
    <param-value>lightgoldenrodyellow</param-value>
  </init-param>
</servlet>
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

log メソッドの使用

`log` メソッドを使用して、プログラマが指定したメッセージを、サーブレットのホストになっている JRun サーバーのログファイルに書き込むことができます。

メモ：JRun では、記録されるログ情報の量は自動的に変わります。ログファイル設定の詳細については、『JRun 管理者ガイド』または JMC を参照してください。

次の例では、ユーザーのアクセス情報をログに記録します。

```
  HttpSession thisSession = req.getSession();
  String userName = (String)thisSession.getAttribute("name");
  if(userName != null) {
    out.println("<h2> ようこそ、" + userName + " さん </h2>");
    // ロギング用の情報を準備
    // この例では、ユーザー名と IP アドレスをログに記録します。
    String logMsg = userName + ", " + req.getRemoteAddr();
    log(logMsg);
  }
```

システムプロパティへのアクセス

他の Java アプリケーションと同様に、System オブジェクトを呼び出してプロパティを取得できます。これにより、ユーザーの言語、タイムゾーン、地域、ファイルセパレータなど Web アプリケーションをプラットフォーム独立にするために役立つ多くのプロパティにアクセスできます。これらのプロパティによって、JRun が使用するいくつかのサービスにもアクセスでき、JVM が使用するネーミングファクトリ、ディレクトリ構造、クラスパスを識別できます。

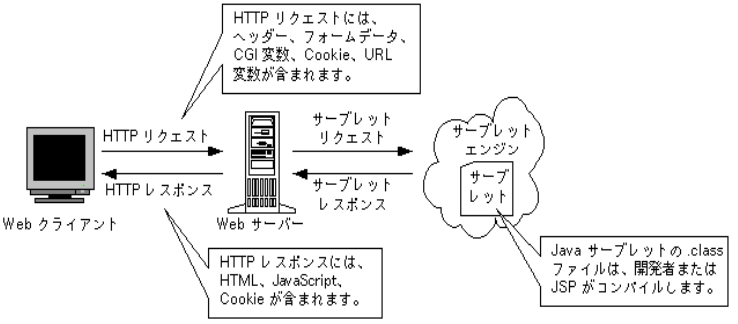
次の例では、すべての System プロパティを表にリストし、それから **user.region** プロパティにアクセスして JRun サーバーのロケーションに関する情報をクライアントに提供します。

```
...
Properties sysprops = System.getProperties();
...
out.println("<TABLE>");
Enumeration enum = sysprops.propertyNames() ;
while (enum.hasMoreElements()) {
  out.println("<TR><TD>");
  String key = (String) enum.nextElement();
  out.println(key + "</TD><TD>" + sysprops.getProperty(key) +
        "</TD></TR>");
}
out.println("</TABLE>");
...
String region = sysprops.getProperty("user.region");
out.println("<HR> このサーバーがホストされている地域：<B>" + region + "</B>");
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

HTTP リクエストとレスポンス

サーブレットは、HTTP リクエストがサーブレットを Java サーブレットとして直接的に参照した場合、または JSP ファイルとして間接的に参照した場合に起動します。サーブレットが実行する最も一般的なタスクは、HTTP リクエストに保管された情報にアクセスし、その情報を処理してから、結果を HTTP レスポンスの一部としてクライアントに返すことです。



HTTP リクエストには、クライアントからサーブレットに送信される情報が含まれています。たとえば、サーブレットがフォームによって呼び出された場合、サーブレットは処理を行う前に、リクエストに保管されているフォームデータにアクセスします。フォームデータには、たとえば、ユーザー認証で使用されるログイン情報、データベースに書き込まれる登録情報、ユーザーのショッピングカートに追加される商品情報などが含まれます。

次の方法で、サーブレットや JSP の HTTP リクエスト情報にアクセスできます。

- Java サーブレットでは、`javax.servlet.HttpServletRequest` オブジェクトを使用します。このオブジェクトは、リクエスト内に保管された情報のアクセスに使用できるメソッドを定義しています。
- JSP では、暗黙の JSP request オブジェクトを使用します。この request オブジェクトを使用すると、`javax.servlet.HttpServletRequest` オブジェクトの場合と同じメソッドを使用できます。

サーブレットは、HTTP レスポンスを構築し、このレスポンスをクライアントに返すことによって、リクエストに応じます。サーブレット内で、HTTP レスポンスにアクセスし、クライアントに返されるレスポンスに情報を書き込みます。

次の方法で、サーブレットや JSP の HTTP レスポンス情報にアクセスできます。

- Java サーブレットでは、`javax.servlet.HttpServletResponse` オブジェクトを使用します。このオブジェクトは、レスポンスに保管された情報にアクセスするメソッドを定義しています。
- JSP では、暗黙の JSP response オブジェクトを使用します。この response オブジェクトを使用すると、`javax.servlet.HttpServletResponse` オブジェクトの場合と同じメソッドを使用できます。

HTTP レスポンスには、クライアントへ結果を返送するのに使用する出力ストリームが含まれます。

response オブジェクトと request オブジェクトのメソッド

request オブジェクトと response オブジェクトには多くのメソッドがあり、クライアントからのリクエストを処理するために使用します。次の例では、サーブレットはこれらのオブジェクトを反映し、メソッドと戻り値のタイプを表示します。

```
...
resp.setContentType("text/html");
PrintWriter out = resp.getWriter();
Object object = null;
String targetClass = "javax.servlet.ServletConfig"; // デフォルト
String checkReq = req.getParameter("targetClass");
if (checkReq != null) {
  targetClass = checkReq;
}
if (targetClass.equals("javax.servlet.ServletContext")) {
  object = getServletContext();
}
if (targetClass.equals("javax.servlet.ServletConfig")) {
  object = getServletConfig();
}
...
try {
  Class c = Class.forName(targetClass);
  Method[] meths = c.getMethods();
  Object[] arguments = new Object[] {};
  for (int i=0; i<meths.length; i++) {
    String methodName = meths[i].getName();
    String returnType = meths[i].getReturnType().getName();
    String desc = meths[i].toString();
    Class[] parameterTypes = meths[i].getParameterTypes();
    int m = c.getModifiers();
    out.println("<TR><TD><I>" + methodName + "</I></TD><TD>" +
        returnType + "</TD><TD>" + desc + "</TD>");
  }
} catch (Exception e) {}
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

CGI 環境変数へのアクセス

サーブレットの request オブジェクトからは、一般的な CGI 環境変数から提供されるものと同じ環境に関する情報が提供されます。次の表に、request オブジェクトのメソッドとそれらに対応する CGI 環境変数をリストします。

HttpServletRequest メソッド	CGI 環境変数
getAuthType	AUTH_TYPE
getContentLength	CONTENT_LENGTH
getContentType	CONTENT_TYPE
getHeader("Accept")	HTTP_ACCEPT
getRealPath("/")	DOCUMENT_ROOT
getHeader("Referer")	HTTP_REFERER
getHeader("User-Agent")	HTTP_USER_AGENT
getPathInfo	PATH_INFO
getPathTranslated	PATH_TRANSLATED
getQueryString	QUERY_STRING
getRemoteAddr	REMOTE_ADDR
getRemoteHost	REMOTE_HOST
getRemoteUser	REMOTE_USER
getMethod	REQUEST_METHOD
getServletPath	SCRIPT_NAME
getServerName	SERVER_NAME
getServerPort	SERVER_PORT
getProtocol	SERVER_PROTOCOL

リクエストの処理

サーブレットのリクエストはさまざまな方法で到着します。サーブレットは、それらのリクエストの受け取り、クライアント入力の解析、そしてレスポンスの生成に責任があります。データをサーブレットに渡すメソッドには次のものがあります。

- クエリ文字列パラメータ
- フォーム入力
- リクエストヘッダー

使用できるリクエスト処理メソッドを判断するには、まず HTTP 仕様で定義されているさまざまなリクエストメソッドの違いを理解する必要があります。

GET と POST

クライアントが HTTP `GET` リクエストを JRun に送信すると、JRun はサーブレットの `doGet` メソッドを呼び出します (このメソッドをオーバーライドしていた場合)。ユーザーが URL を入力、リンクをクリック、または `method=GET` を指定するフォームを送信すると、Web ブラウザにより HTTP `GET` リクエストが送信されます。`GET` メソッドは、ページをリクエストするために呼び出す最も一般的なメソッドです。リクエストパラメータは、`GET` リクエストメッセージのリクエストメソッドヘッダーに保管されます。これにより、`GET` メソッドのサイズと構造が制限されます。

クライアントが JRun に HTTP `POST` リクエストを送信すると、JRun は `doPost` メソッドを呼び出します。ユーザーが `method=POST` を指定するフォームを送信すると、Web ブラウザにより HTTP `POST` リクエストが送信されます。`POST` リクエストメッセージの本文に、リクエストパラメータが保管されます。このために、`POST` メソッドには、`GET` メソッドより多くの情報が保管できます。

クエリ文字列パラメータの使用

HTTP リクエストでクライアントからサーバーに渡されるクエリ文字列。これは、color=red などの名前 / 値のペアです。これは URL の一部になるか、HTTP リクエストに付加されます。次の URL では、クエリ文字列パラメータ color が red に設定され、クエリ文字列パラメータ name が danger に設定されています。

```
http://localhost:8100/servlet/myservlet/?name=danger&color=red
```

最近のほとんどの Web サーバーでは、クエリ文字列の長さが 4K (4000 文字) に制限されています。しかし、この制限が原因でパフォーマンス上の問題が発生することがあります。クエリ文字列が長くなりすぎる場合は、`POST` メソッドを使用してクライアントからサーバーにデータを送信します。これは、`FORM` タグを使用すると可能です。

URL 書き換えでは、クエリ文字列パラメータを使用して、複数のリクエスト間でセッションデータを維持します。URL 書き換えの詳細については、159 ページの「URL 書き換えの使用」を参照してください。

クエリ文字列へのアクセス

次の例に示すように、request オブジェクトの getParameter メソッドを使用してクエリ文字列パラメータにアクセスします。

```
String name = request.getParameter("param-name");
```

次の行では、getParameter メソッドの署名が示されています。

```
public abstract String getParameter(String name)
```

たとえば、次の URL をリクエストする場合を考えます。

```
http://localhost:8100/servlet/myservlet/?name=danger&color=red
```

次のコードを使用すると、name と color の値を抽出できます。

```
String bgcolor = request.getParameter("color");
String first_name = request.getParameter("name");
```

次の例に示すように、getParameterNames メソッドを使用すると、すべてのクエリ文字列パラメータにアクセスできます。

```
...
Enumeration eParmNames = req.getParameterNames();
while (eParmNames.hasMoreElements()) {
  String parm = (String) eParmNames.nextElement();
  out.println(" " + parm + " = " + req.getParameter(parm) + "<br>");
}
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

配列を使用する作業

パラメータは複数の値を持つことができます。次の例に示すように、getParameterValues メソッドを使用すると、リクエストされたパラメータのすべての値の配列を取得できます。

```
...
String[] state = req.getParameterValues("state");
for (int i = 0; i<state.length; i++) {
  out.println("<BR> あなたの家がある地域：" + state[i]);
}
...
```

クエリ文字列のキャスト

クエリ文字列パラメータは String ですが、getParameter メソッドの結果は自由にキャストできます。たとえば、UserID を int で取得するには、次の行を使用します。

```
int id = (int)(Integer.parseInt(req.getParameter("id")));
```

クエリ文字列パラメータが付加された URL を構築することで、そうしたパラメータを response オブジェクトに設定します。

フォーム入力の使用

HTML の FORM タグを使用して、ユーザーが記入して送信できるフォームを作成します。FORM メソッドは通常 POST に設定されているために、サーブレットはこの種類のデータを通常 doPost メソッドで処理します。

次の例では、ユーザー情報の入力を促す HTML フォームを表示します。

```
...
out.println("<html><head><title> サンプルフォーム </title>");
out.println("</head><body>");
out.println("<h1> サンプルフォーム </h1>");
out.println("<FORM METHOD=¥"POST¥" ACTION=¥"SampleForm¥">");
out.println(" 姓： ");
out.println("<INPUT TYPE=¥"text¥" NAME=¥"lastname¥"
        SIZE=40>");out.println("<INPUT TYPE=¥"text¥"
        NAME=¥"lastname¥" SIZE=40>");
out.println("<BR> 名： ");
out.println("<INPUT TYPE=¥"text¥" NAME=¥"firstname¥" SIZE=40>");
out.println("<INPUT TYPE=¥"Submit¥" VALUE=¥"Submit¥"></FORM>");
out.println("</body></html>");
...
```

この後、次の例に示すように、request.getParameter メソッドを使用すると、POST メッセージからパラメータを抽出できます。

```
...
public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException, ServletException {
...
String fname = req.getParameter("firstname");
String lname = req.getParameter("lastname");
...
```

次の例に示すように、getParameter メソッドの結果を使用すると、出力を判断するロジックをサーブレットに組み込むことができます。

```
...
if (fname != null) {
  out.println(lname + fname " さん、こんにちは ");
} else {
  out.println(" 次のフォームに記入してください。");
}
...
```

このサンプルではフォームをそれ自体にポストして、リクエストパラメータの存在を調べます。パラメータが null でない場合はサーブレットはウェルカムメッセージをユーザーに表示し、パラメータが null の場合はサーブレットは姓名の入力を促がします。

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

フォーム送信でのさまざまな文字セットの処理については、40 ページの「外国語フォームの送信処理」を参照してください。

クライアントへの結果の返送

サーブレットで、リクエストクライアントにダイナミックコンテンツを返すことができます。出力は、サーブレットに渡された情報やサーブレットが計算する情報に基づいて生成されます。たとえば、サーブレットは渡されたフォーム属性 (`request` オブジェクトを使用してアクセスするもの) を使用して、フォーマットされたデータベースデータを返すことができます。一方、アプリケーションにユーザーの環境設定を維持するメソッドがある場合は、保管されている環境設定に基づいて、サーブレットでブラウザの表示色を設定できます。

HTTP レスポンスを使用して、次のように情報を返します。

- サーブレットは、`javax.servlet.HttpServletResponse` オブジェクトの `getWriter` メソッドや `getOutputStream` メソッドを使用して、レスポンスにデータを書き込む PrintWriter オブジェクトと ServletOutputStream オブジェクトを取得します。同じサーブレットでは、PrintWriter オブジェクトと ServletOutputStream オブジェクトは使用できません。
- JSP では、暗黙の JSP オブジェクト out が使用されます。out オブジェクトは PrintWriter です。

特殊文字の処理

PrintWriter のオブジェクトの print メソッドと println メソッドを使用する場合、引用符を使用するときに、特別な配慮をする必要があります。次の例に示すように、特殊文字は ¥ 記号でエスケープします。

```
out.println("<INPUT TYPE=¥"Submit¥" VALUE=¥"Submit¥"></FORM>");
```

引用符を出力に含めるには、HTML エンティティを使用できます。これらのエンティティは、ブラウザが解釈して表示しますが、サーブレットの処理では解釈されません。次の例は、引用符の HTML エンティティを示しています。

```
out.println("<B>&quot;ここは危ないよ。&quot; と彼が言った。</B>");
```

この例は、ブラウザで次のように表示されます。

" ここは危ないよ。" と彼が言った。

HTML エンティティのリストについては、http://www.ramsch.org/martin/uni/fmi-hp/iso8859-1.html をご覧ください。

ヘッダーの設定

HttpServletResponse オブジェクト API には、HTTP ヘッダーを response オブジェクトに設定するメソッドがいくつか含まれています。これらのメソッドは、現在のヘッダーに既存の値がある場合は、それを置き換えます。これらのメソッドの中のいくつかが ServletResponse オブジェクトから継承されています。

次の表で、これらのメソッドを説明します。

メソッド	説明
setHeader (String name, String value)	String を使用し、便利なメソッドを持たないヘッダーを設定します。この値が int の場合は、`setIntHeader` メソッドを使用します。 `Refresh` ヘッダーを設定するには、次のコードを使用します。 `response.setHeader("Refresh", "15");`
setDateHeader (String name, long date)	`Date` ヘッダーを設定します。
setIntHeader (String name, int value)	int 値を取得し、便利なメソッドを持たないヘッダーを設定します。値が String の場合は、`setHeader` メソッドを使用します。 `Expires` ヘッダーを設定するには、次のコードを使用します。 `response.setIntHeader("Expires", -1);`
setContentLength (int length)	バイト数を示す `Content-Length` ヘッダーを設定します。たとえば、次のようにコーディングします。 `response.setContentLength(data.length);` HttpServletResponse インターフェイスは ServletResponse からこのメソッドを継承しています。通常このヘッダーを変更することはありません。

メソッド	説明
setContentType()	Content-Type ヘッダーを設定します。Content-Type で、レスポンスドキュメントの MIME タイプと文字セットエンコードが識別されます。Content-Type ヘッダーを設定するには、次のコードを使用します。 `response.setContentType("text/html; charset=Shift_JIS");` デフォルト MIME タイプは text/plain です。一般的な MIME タイプの例は次のとおりです。 • HTML フォーマット (.htm または .html)：text/html • Adobe Portable Document (pdf)：application/pdf • Microsoft Word (.doc)：application/msword • Microsoft Excel (.xls)：application/msexcel • Microsoft Powerpoint (.ppt)：application/ms-powerpoint • Realaudio (.rm または .ram)：audio/x-pn-realaudio • テキストフォーマット (.txt)：text/txt • Zip 圧縮ファイル (.zip)：application/zip 登録された MIME タイプのリストをダウンロードするには、ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/ をご覧ください。 文字セット設定の詳細については、32 ページの第 3 章「国際化対応とローカリゼーションについて」を参照してください。
setContentLength(int len)	Content-Length ヘッダーを設定します。
setLocale(Locale locale)	Content-Language ヘッダーと Content-Type ヘッダーの charset 値を設定します。デフォルトはサーバーシステムのロケールです。 Web アプリケーションでの国際化対応テクニックの使用方法の詳細については、32 ページの第 3 章「国際化対応とローカリゼーションについて」を参照してください。
addCookie(Cookie cookie)	Set-Cookie ヘッダーを設定します。複数の Cookie をレスポンスに追加できます。Cookie の使用方法の詳細については、173 ページの「Cookie の処理」を参照してください。
sendRedirect (String location)	Location ヘッダーを設定しステータスコードを 302 (Found) に設定します。たとえば次のように、相対 URI や絶対 URI を指定することができます。 `response.sendRedirect("http://www.hamsteak.com/index.html");` 相対 URL を指定した場合は、JRun はリクエストを転送する前に絶対 URI を構築します。サーブレット API には、リクエストを転送するための便利なメソッドを提供する RequestDispatcher も含まれています。 詳細については、170 ページの「制御の受け渡し」を参照してください。

メソッド	説明
setStatus(int sc)	レスポンスドキュメントのためにステータスコードを設定します。ステータスコードには次のものがあります。 • 1xx: Informational　有効なリクエストが受信されました。 • 2xx: Success　リクエストは有効で、レスポンスは成功しました。 • 3xx: Redirection　リクエストを処理するには、追加情報が必要です。 具体的なステータスコードの詳細については、HTTP 仕様を参照してください。
sendError(int sc)	指定されたステータスコードを使用してエラーをブラウザに送信します。エラーステータスコードには次のものがあります。 • 4xx: Client Error　リクエストが無効です。 • 5xx: Server Error　サーバーは、リクエストを達成できませんでした。 具体的なステータスコードの詳細については、HTTP 仕様を参照してください。

response オブジェクトにも同等の addXxxxx メソッドがあり、既存のヘッダーを置き換えるのではなく、レスポンスに新しいヘッダーを追加します。詳細については、サーブレット API を参照してください。

次のコード例では、レスポンスヘッダーを設定し、リクエストヘッダーのリストを表示してます。

```
...
resp.setContentType("text/html");
int sc=200;
resp.setStatus(sc);
resp.setHeader("Cache-Control", "no-cache,must-revalidate");
resp.setHeader("Pragma", "no-cache");
resp.setHeader("Cache-Control", "no-store");
resp.setDateHeader("max-age", 0);
resp.setDateHeader("Expires", 0);
Enumeration enum = req.getHeaderNames();
while (enum.hasMoreElements()) {
  String headerName = (String) enum.nextElement();
  String headerVal = req.getHeader(headerName);
  out.println(headerName + " :" + headerVal + "<BR>");
}
...
```

リクエストとレスポンスに関して HTTP ヘッダーを表示するには、TCPMonitor ユーティリティを使用します。詳細については、236 ページの「TCPMonitor の使用」を参照してください。

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

PrintWriter の使用

PrintWriter は文字データを response オブジェクトに書き込みます。このオブジェクトには **print** メソッドと **println** メソッドがあります。PrintWriter を使用するには、**java.io.*** パッケージをインポートし、IOExceptions をキャッチする必要があります。

次の例は、サーブレットで PrintWriter の **out** メソッドを使用して、リクエストクライアントにレスポンスを返す方法を示しています。

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

public class PrintTester extends HttpServlet {

  public void doGet ( HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
  PrintWriter out = response.getWriter();
  out.println("<html><head><title>PrintTester</title></head>");
  out.println("<body>PrintTester よりこんにちは ");
  out.println( "</body></html>");
  }
}
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

ServletOutputStream の使用

ServletOutputStream オブジェクトは、バイナリデータをクライアントに返します。このオブジェクトには **print** メソッドと **println** メソッドがあります。これは **java.io.OutputStream** のサブクラスなので、**flush**、**close**、**write** メソッドも備えています。ServletOutputStream を使用するには、**java.io.*** パッケージをインポートし、IOExceptions をキャッチする必要があります。

次の例では、サーブレットで ServletOutputStream のメソッドを使用して、リクエストクライアントにレスポンスを返します。

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

public class SOSTester extends HttpServlet {

public void doGet ( HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
  ServletOutputStream sos = response.getOutputStream();
  String s = "SOSTester よりこんにちは ";
  sos.print(s);
  sos.close();
  }
}
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

ファイルへの書き込み

場合によっては、出力を HTTP リクエストに送信するのではなく、サーブレットからファイルにデータを書き込むことがあります。このファイルは、テンポラリデータを保管したり、パーシスタンスメカニズムとして使用できます。このセクションでは、ファイルを Web アプリケーションのディレクトリや、ホットデプロイされた Web アプリケーションのために JRun が生成したテンポラリディレクトリに書き込む方法を説明します。

Web アプリケーションのルートディレクトリへの書き込み

サーブレットコンテキストを使用すると、Web アプリケーションのルートディレクトリへのパスを取得できます。このパスは、/<JRun のルートディレクトリ >/servers/<Web アプリケーション名 > です。JRun サーバーにはデフォルトで書き込み許可があるため、このパスを取得すると、このディレクトリ内のファイルに書き込みを行うことができます。

次の例では、簡単な文字列をファイルに書き込み、それを Web アプリケーションのルートディレクトリに保管します。

```
...
Properties sysprops = System.getProperties();
String c = sysprops.getProperty("file.separator");
PrintWriter out = response.getWriter();
String notes = " これは文字列です ";
String filename = "file.txt";
String path = new String();
try {
  ServletContext sc = this.getServletContext();
  path = sc.getRealPath("/");
  FileOutputStream fos = new FileOutputStream(path + c + filename);
  byte[] bytes = notes.getBytes();
  fos.write(bytes);
  out.println("Created " + filename + " in " + path);
} catch (Exception e) {
  out.println(e.toString());
}
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

Web アプリケーションのテンポラリディレクトリへの書き込み

WAR ファイルや EAR ファイルを JRun サーバーのルートディレクトリに配置して JRun のホットデプロイ機能を使用すると、JRun は /<JRun のルートディレクトリ >/servers/< サーバー名 >/SERVER-INF/temp/< アプリケーション名 > temp にテンポラリディレクトリを作成して、テンポラリ情報を保管します。**javax.servlet.context.tempdir** 属性を取得すると、このディレクトリにアクセスできます。次の例を使用すると、このディレクトリにファイルを書き込むことができます。

```
...
ServletContext context = this.getServletContext();
Properties sysprops = System.getProperties();
String c = sysprops.getProperty("file.separator");
try {
  File baseDir =
        (File) context.getAttribute("javax.servlet.context.tempdir");
  String fullpath = baseDir.getAbsolutePath() + c + "settings.txt";
  File f = new File(fullpath);
  FileOutputStream fout = null;
  fout = new FileOutputStream(fullpath);
  sysprops.store(fout, null);
  out.println("次のファイルが作成されました：" + f.getAbsolutePath());
} catch (Exception e) {
  e.printStackTrace();
}
...
```

このコード例では、ServletContext を使用して Web アプリケーションのテンポラリディレクトリを取得し、System オブジェクトのプロパティを使用して、ファイルセパレータを取得しています。このコードでは、システムプロパティが settings.txt と名付けられたファイルに保管されます。

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

例外処理

例外とは、サーブレット内で検出されたエラーのことです。例外は、サーブレットがフォームデータを処理するときなどランタイムで発生することも、JSP をプリコンパイルするときなどコンパイル時に発生することもあります。Web アプリケーションを運用開始する前に、コンパイル時の例外をキャッチする必要があります。このセクションでは、サーブレットと JSP でランタイムの例外を処理する方法を説明します。

Java サーブレットでは、例外はクラス **javax.servlet.ServletException** のインスタンスで表されます。

WEB-INF/web.xml ファイルの **error-page** 要素を使用すると、Web アプリケーションがエラーを処理する方法を定義できます。**error-page** 要素を SERVER-INF/default-web.xml ファイルに追加することで、JRun サーバーのすべての Web アプリケーションに関してエラー処理を定義することもできます。

次のセクションで説明されているように、**error-page** 要素で、例外タイプ別やエラーコード別に例外を定義します。web.xml ファイル内のこれらの要素の順序によって、エラーハンドラが決まります。JRun は、**error-code** や **exception-type** と一致する最初の error-page 要素で指定された位置にエラー処理を転送します。

Java 例外処理

web.xml ファイルの **error-page** の exception-type サブ要素によって、サーブレットの処理中に投げられた Java 例外を JRun が処理する方法が決まります。

exception-type に対して完全修飾の Java エラーを定義し、そのエラーを location 要素内で送信先にマッピングします。次の例では、FileNotFound 例外を 404.jsp ページにマッピングします。

```
<error-page>
  <exception-type>java.io.FileNotFoundException</exception-type>
  <location>/error-pages/404.jsp</location>
</error-page>
```

HttpServletRequest オブジェクトと HttpServletResponse オブジェクトによってエラー情報にアクセスできるので、役に立つデバッグ情報や目的の例外ハンドラを生成できます。詳細については、149 ページの「エラー属性へのアクセス」を参照してください。

HTTP エラーコードの処理

web.xml ファイルの error-page の error-code サブ要素によって、サーブレットの処理中に生成された HTTP エラーコードを JRun が処理する方法が決まります。

error-code 要素のために HTTP ステータスコードを定義し、そのコードを location 要素で転送先にマッピングします。次の例では、HTTP 500 (内部サーバーエラー) ステータスコードを servererror.jsp ページにマッピングします。

```
<error-page>
  <error-code>500</error-code>
  <location>/error-pages/servererror.jsp</location>
</error-page>
```

次の表に、一般的なエラー関連 HTTP ステータスコードをリストします。

HTTP エラーコード	説明
400	不正なリクエスト
401	無許可
403	禁止
404	発見できず
408	リクエストタイムアウト
500	内部サーバーエラー

エラーコードの完全なリストについては、HTTP 1.1 RFC (http://rfc.net/rfc2616.html) をご覧ください。

HttpServletRequest オブジェクトと HttpServletResponse オブジェクトによってエラー情報にアクセスできるので、役に立つデバッグ情報や目的の例外ハンドラを生成できます。詳細については、149 ページの「エラー属性へのアクセス」を参照してください。

エラー属性へのアクセス

HttpServletRequest オブジェクトと HttpServletResponse オブジェクトによってエラー情報にアクセスできるので、役に立つデバッグ情報や目的の例外ハンドラを生成できます。

エラーが投げられると、JRun はいくつかの属性を request オブジェクトに設定します。次の表で、これらの属性を説明します。

属性	説明
javax.servlet.error.status_code	当てはまる場合は、HTTP エラーコードを int オブジェクトとして定義します。サーブレットが、HTTP と関連しない例外を投げた場合は、ステータスコードは通常 500 (内部サーバーエラー) に設定されます。
javax.servlet.error.message	例外やエラーメッセージを返します。
javax.servlet.error.exception_type	例外のタイプを定義します。
javax.servlet.error.exception	投げられる実際の例外を定義します。`printStackTrace` メソッドを使用すると、例外のスタックトレースを表示できます。
javax.servlet.error.request_uri	例外が投げられる前に、リクエスト URI を定義します。

次のコード例では、request オブジェクトからエラー属性を取得し、スタックトレースを含めてそれらをブラウザで表示します。

```
...
Object status_code =
        req.getAttribute("javax.servlet.error.status_code");
Object message = req.getAttribute("javax.servlet.error.message");
Object error_type =
        req.getAttribute("javax.servlet.error.exception_type");
Throwable throwable = (Throwable)
        req.getAttribute("javax.servlet.error.exception");
Object request_uri =
        req.getAttribute("javax.servlet.error.request_uri");
out.println("<HTML><BODY>");
out.println("<B>ステータスコード：</B> " + status_code.toString());
out.println("<BR><B> メッセージ</B>：" + message.toString());
out.println("<BR><B>エラータイプ</B>：" + error_type.toString());
out.println("<BR><B>リクエスト URI</B>：" + request_uri.toString());
out.println("<HR><PRE>");
if (throwable != null) {
  throwable.printStackTrace(out);
}
...
```

次に示すように、web.xml ファイルでは、エラーハンドラの場所とエラーハンドラが処理するエラーのタイプが定義されます。

```
<error-page>
  <exception-type>javax.servlet.ServletException</exception-type>
  <location>/servlet/ErrorHandler</location>
</error-page>
```

エラーを生成してこの例をテストするために、次のように数を 0 で除算する式を含む JSP を作成します。

```
<%= 42/0 %>
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

セッションの操作

HTTP はステートレスプロトコルです。Web サーバーは、リクエストを受け取ってレスポンスを返すと、クライアントとの接続を終了します。Web サーバーにはクライアントの情報は維持されません。そのため、同じクライアントから別のリクエストがきても、そのことを判断できません。Web サイトからクライアントやナビゲーションを自動追跡できないために、Web サイトで複雑なトランザクションを実行することが難しくなっています。

ただし、JRun では session オブジェクトがサポートされており、これを使用すれば、Web サーバーとの対話全体を通してユーザーを追跡できます。session オブジェクトを使用して、ショッピングするユーザーを追跡し、登録情報や環境設定情報を送ることができます。また、サイトに接続するたびに情報を再入力する手間からユーザーを解放することもできます。

session オブジェクトは、特定のユーザーセッションに関する情報を保管します。デフォルトでは、JRun は Cookie を使用してセッションを追跡しますが、JRun では URL 書き換え、Hidden フォームフィールドやカスタムメソッドを使用してもセッションを追跡できます。Cookie 自体に セッション ID と呼ばれる ID 番号だけを保管すれば、これによってリクエストをサーバーサイドの session オブジェクトと比較できます。URL 書き換えや Hidden フォームフィールドを使用するときは、セッション ID は Cookie の値としてでなく、追加パス情報やフォームフィールドとして渡されます。

ユーザーがアプリケーション内で別のページに移動しても、session オブジェクトに保管された変数は破棄されず、ユーザーセッションが継続している間は保持されます。クライアントがリクエストでセッション ID を送り続けているかぎり、JRun はそのクライアントのユーザー特有のセッションデータにアクセスできます。

まだセッションを開始していないユーザーからアプリケーションのページがリクエストされた場合、JRun は session オブジェクトを作成します。セッションが期限切れになった場合や放棄された場合は、Web サーバーによって session オブジェクトが廃棄されます。

session オブジェクトは、セッション ID をクライアントから、設定に従っていくつかの方法で取り出します。サーブレット API はクライアントのセッション ID を取得するサーバーの正確な詳細を抽出します。しかし、各メソッドで異なるコーディングテクニックが必要となるために、メソッド間の違いを理解する必要があります。

次の表で、session オブジェクトを実装できるさまざまな方法を説明します。

パーシスタンスメカニズム	説明
Cookie	デフォルトのステート管理メカニズム。Cookie がセッション ID を保管し、サーバーコンポーネントがセッション依存性メソッドを呼び出したときに、サーバーがこの ID を取り出します。この Cookie へのリファレンスは、すべてのリクエストでサーバーに渡され、すべてのレスポンスで HTTP ヘッダーとしてクライアントに返されます。 これは、特別なコーディングをする必要がないため、最も簡単に実装できる方法です。しかし Cookie を受け付けないブラウザもあります。 Cookie はデフォルトのセッションメカニズムです。
Hidden フォームフィールド	JRun は、ユーザーに隠されたフォームフィールドとしてセッション ID をすべてのリクエストに付加します。このメソッドを使用するには、すべてのページが POST リクエストを処理する必要があります。 セッションメカニズムとしての Hidden フォームフィールドの使用方法の詳細については、161 ページの「Hidden フォームフィールドの使用」を参照してください。
URL 書き換え	JRun は、追加なパス情報としてセッション ID をすべての URL に付加します。セッション ID を付加するには、すべての URL をエンコードする必要があります。 セッションメカニズムとして URL 書き換えを使用する方法の詳細については、159 ページの「URL 書き換えの使用」を参照してください。

アプリケーションは、単純な実装ではユーザー名を記憶し、セキュアな実装ではユーザー名とパスワードを記憶し、電子商取引システムではショッピングカートを維持します。サーブレット API を使用しないアプリケーションの場合、通常はアプリケーション特有のストレージメカニズムを使用してセッション情報を保管し、Hidden フォームフィールド、URL クエリ文字列、または Cookie を使用してキーの値をブラウザに返します。後続のリクエストでは、このキーの値を使用して前に保管されているセッション情報を取得します。

session オブジェクトは、ユーザーが Web サイトに接続している間情報を保管したり取り出したりする、唯一の場所を提供します。

次の方法で、サーブレットと JSP のセッション情報にアクセスできます。

- Java サーブレットでは、`javax.servlet.http.HttpSession` オブジェクトを使用します。
- JSP では、暗黙の JSP session オブジェクトを使用します。

Java サーブレット仕様では `HttpSession` インターフェイスが定義されています。このインターフェイスによって、実装の詳細を把握しなくてもセッション管理を行うことができます。`HttpSession` インターフェイスは `HttpServletRequest` クラスから利用できます。

セッションを使用するファイルを転送またはインクルードするときは、そのセッションを作成した元のリクエストによって、対応する response オブジェクトに Cookie が設定され、このオブジェクトが `include` メソッドや `forward` メソッドで使用されます。

セッションの確立

getSession メソッドを使用してセッションを確立し、getAttribute メソッドを使用してセッションの値にアクセスします。

セッションの確立

1 セッションに保存する値を確立します。

```
public class SelectionForm extends HttpServlet {
  public void doPost(HttpServletRequest req, HttpServletResponse
     resp) throws IOException, ServletException {
    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter();
    String thisName = "Unknown Name";
    // ログインフォームからユーザー名を取得します。
    String[] attrArray = req.getParameterValues("myName");
    // 呼び出し側フォームには、myName の値が 1 つしかないと想定します。
    if(attrArray != null && attrArray.length > 0) {
      thisName = attrArray[0];
    }
  }
...
```

2 HttpServletRequest オブジェクトの getSession メソッドを呼び出すことで新規セッションを作成します。

```
// セッションを作成します。
HttpSession thisSession = req.getSession(true);
```

3 session オブジェクトの setAttribute メソッドを使用して、属性をセッションに関連付けます。

```
// この例では、ユーザー名を保存します。
thisSession.setAttribute("name", thisName);
```

セッション情報へのアクセス

1 HttpServletRequest オブジェクトの getSession メソッドを呼び出すことで session オブジェクトへのリファレンスを取得します。

```
...
public class DisplayInfo extends HttpServlet {
public void doPost(HttpServletRequest req, HttpServletResponse
     resp) throws IOException, ServletException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title> 情報の表示 </title></head><body>");
  out.println("<h1>displayInfo Servlet</h1>");
  // session オブジェクトからログインユーザーを取得します。
  HttpSession thisSession = req.getSession(true);
...
```

2 HttpSession オブジェクトの getAttribute メソッドを使用して属性値にアクセスします。結果は必ず適切なタイプにキャストされます。

```
String userName = (String)thisSession.getAttribute("name");
```

3 コード内での属性値を使用します。

```
if(userName != null) {
  out.println("<h2> ようこそ " + userName + " さん </h2>");
}
```

セッションの設定

パーシスタンス設定と Cookie の設定は jrun-web.xml ファイルで行います。セッションタイムアウト間隔も、Web アプリケーションの web.xml デプロイメントディスクリプタで変更できます。

このセクションでは、jrun-web.xml ファイルと web.xml ファイルを編集することで、設定を変更する方法を説明します。MBean View ユーティリティを使用しても、ランタイムのセッション管理設定を変更できます。**jrun.servlet.session.SessionService** サービスではセッションパーシスタンスが定義されます。MBean View の使用方法の詳細については、『JRun 管理者ガイド』を参照してください。JRun には、セッションメカニズムのファイルと JDBC ストレージ実装が含まれています。

セッションタイムアウトの変更

session オブジェクトは少量のリソースを使用します。セッションがアクティブな間は、JRun は session オブジェクトを放置します。しかし、アクティブでない期間の後では、JRun は session オブジェクトを廃棄して、未使用のリソースを解放します。

web.xml ファイルでは、**session-config** という、1 つのセッション関連の要素が定義されています。この要素には、**session-timeout** という 1 つのサブ要素があります。**session-timeout** 要素では、JRun によってデータストアに保管されたり、廃棄される (パーシスタンスが無効な場合) 前に、session オブジェクトがアクティブでない状態で何分間いられるかが決まります。

デフォルトのセッションタイムアウトは 30 分です。最大値は 34560 分 (24 日) です。session-time-out に 0 や負の値を設定すると、JRun はタイムアウトを行わず、アクティブでない期間がどれだけ続いてもセッションを継続させます。しかし、非常に多くのセッションが作成され、そしてそれが廃棄されなかった場合、JRun は結局セッションをディスクに保存するなどのパーシスタンス方法で、そのステートを維持します。こうすることで、パフォーマンスが低下しリソースが枯渇することがあります。

次の例では、web.xml ファイルでセッションのタイムアウトを 42 分に設定します。

```
<web-app>
...
  <session-config>
    <session-timeout>42</session-timeout>
  </session-config>
...
</web-app>
```

Cookie を使用するときも、生存期間の最大値を設定できます。この値によって、その Cookie の有効期間がブラウザに指示されます。`cookie-max-age` の値と `session-timeout` の値は異なる設定を定義します。前に説明したように、セッションがそのセッションのタイムアウトに達すると、JRun はサーバーサイドの session オブジェクトを廃棄します。ブラウザサイドでは、`cookie-max-age` の有効期限が切れると、ブラウザはその Cookie ファイルを廃棄します。詳細については、155 ページの「jrun-web.xml ファイルでのセッション設定」を参照してください。

jrun-web.xml ファイルでのセッション設定

Web アプリケーションの jrun-web.xml ファイルを使用すると、各 Web アプリケーションのレベルで、ほとんどの設定が可能です。このセクションで説明した設定を jrun-web.xml ファイルに追加しなかった場合、JRun はデフォルト値を使用します。

セッション設定は、`jrun-web-app` ルート要素の下にある `session-config` サブ要素で定義されます。`session-config` の下で、次のサブ要素を設定できます。

要素	説明
persistence-config	Cookie、URL、Hidden フォームフィールドというすべてのセッション方式のためにパーシスタンス設定を定義します。まだアクティブなセッションがある中で JRun サーバーが停止するときに、パーシスタンス設定が利用されます。
cookie-config	セッション追跡のために使用される Cookie の設定を指定します。
replication-config	セッションのフェイルオーバーを可能にする、セッションのレプリケーションを設定します。クラスタリングが有効で、指定されたすべてのバディが指定されたクラスタ内にあるときにだけ、セッションレプリケーションは機能します。

次のセクションで、これらの各要素について説明します。jrun-web.xml ファイルでのセッション設定のサンプル設定については、158 ページの「セッション設定の例」を参照してください。

persistence-config

次の表で、persistence-config 要素で利用可能な要素を説明します。

要素	説明	デフォルト
active	セッションパーシスタンスが有効かどうかを示します。	true
persistence-type	使用するデフォルトのセッションストレージを指定します。有効な値は file と jdbc です。 詳細については、『JRun 管理者ガイド』を参照してください。	file
persistence-class	セッションのストレージプロバイダーとして使用するクラスを指定します。この要素は、persistence-type が指定されていないときに使用します。 ここで指定されるクラスは、jrun.servlet.session.SessionStorage を実装している必要があります。	該当なし
persistence-synchronized	セッションパーシスタンス書き込みが同期されるかどうかを指定します。	true
class-change-option	クラスローダーが変化したときにセッションで実行する作業を指定します。 有効な値は、reload、drop、ignore です。	reload
session-swapping	セッションをメモリーからストレージサービスにスワップするかどうかを指定します。	false
session-swap-interval	ストレージサービスにスワップされたセッションについてセッションプールをチェックする頻度を指定します。これは秒単位で指定します。	10
session-max-resident	ストレージサービスにスワップする前に、メモリーに保管可能なセッション数を指定します。 session-swap-interval での遅延と、直列化されないセッションのために、実際の最大数は異なる場合があります。0 に設定した場合は、すべてのセッションデータはリクエスト完了時にすぐに書き出されます。 このプロパティを有効にするには、session-swapping を true に設定する必要があります。	9999999
init-param	セッションのストレージプロバイダーに渡す初期化パラメータを指定します。サブ要素は、param-name と param-value です。 jdbc と file という persistance-type の初期化パラメータの詳細については、『JRun 管理者ガイド』を参照してください。	該当なし

cookie-config

次の表で、`cookie-config` 要素で設定できる要素を説明します。

要素	説明	デフォルト
active	Cookie が有効かどうかを指定します。	true
cookie-max-age	秒単位で Cookie の最大生存期間を設定します。この値はブラウザサイドの Cookie に保管されます。その Cookie の有効期間がブラウザに指示されます。 正の値を指定した場合、指定した時間サーバーとの対話がないと、ブラウザは Cookie ファイルを削除します。 負の値や 0 を指定した場合、現在のセッションが完了すると、ブラウザは Cookie を保管しません。 `cookie-max-age` の値と `session-timeout` の値は異なる設定を定義します。154 ページの「セッションタイムアウトの変更」で説明されているように、セッションがそのセッションタイムアウトに達すると、JRun はサーバーサイドの session オブジェクトを廃棄します。ブラウザサイドでは、`cookie-max-age` の有効期限が切れると、ブラウザはその Cookie ファイルを廃棄します。	-1
cookie-secure	HTTPS や SSL などのセキュアプロトコルだけを使用して、Cookie が送信されるかどうかをブラウザに示します。	false
cookie-domain	Cookie が送信されるドメインを設定します。 ドメイン名はドットで始まるため (.foo.com など)、特定の DNS (Domain Name System) 領域にあるサーバーでは Cookie を見ることができます (たとえば、www.foo.com。a.b.foo.com は違います)。ドメイン名の形式は、RFC2109 で指定されています。 デフォルトでは、Cookie はその送信元サーバーだけに返されます。	該当なし
cookie-comment	Cookie の目的を説明するコメントを指定します。Web ブラウザがユーザーに Cookie を表示する場合、このコメントは役に立ちます。このコメントは Netscape の Cookie バージョン 0 ではサポートされていません。	該当なし
cookie-path	クライアントが Cookie を返すパスを設定します。この Cookie は、指定するディレクトリ内のすべてのページや、そのディレクトリのサブディレクトリ内のすべてのページから可視です。Cookie のパスには、その Cookie を設定したサーブレットを含める必要があります。たとえば /catalog を指定すると、その Cookie は /catalog の下にあるそのサーバー上のすべてのディレクトリから可視になります。	/
cookie-name	セッションの Cookie 名を設定します。この名前に使用できるのは、コンマ、セミコロン、空白文字を除く ASCII 英数文字のみであり、$ 文字を先頭にすることはできません。 この名前は RFC2109 に準拠する必要があります。	JSESSIONID

replication-config

次の表で、replication-config 要素で設定できる要素を説明します。

要素	説明	デフォルト
active	セッションレプリケーションが有効かどうかを示します。	false
buddy-name	クラスタ内でローカルセッションを複製する特定のサーバーを指定します。* に設定すると、クラスタ内のすべてのサーバーに継続セッションを複製します。これは大規模クラスタではお勧めできません。	該当なし

セッション設定の例

次の例は、jrun-web.xml ファイルでのパーシスタンス設定を示しています。

```
<jrun-web-app>
...
  <session-config>
    <persistence-config>
      <active>true</active>
      <persistence-type>file</persistence-type>
      <persistence-synchronized>true</persistence-synchronized>
      <class-change-option>reload</class-change-option>
      <session-swapping>true</session-swapping>
      <session-swap-interval>5</session-swap-interval>
      <session-max-resident>500</session-max-resident>
      <init-param>
        <param-name>foo</param-name>
        <param-value>bar</param-value>
      </init-param>
    </persistence-config>
    <cookie-config>
      <active>true</active>
      <cookie-max-age>-1</cookie-max-age>
      <cookie-secure>true</cookie-secure>
      <cookie-domain>foo</cookie-domain>
      <cookie-comment>george</cookie-comment>
      <cookie-path>bar</cookie-path>
      <cookie-name>jsessionid</cookie-name>
    </cookie-config>
  </session-config>
...
</jrun-web-app>
```

Cookie の無効化

Cookie は無効にできますが、その場合でも JRun で URL 書き換えや Hidden フォームフィールドなどの他のセッションパーシスタンス手段を使用できます。JRun で Cookie を無効にするには、次の例に示すように、**cookie-config** の **active** 要素の値を false に変更します。

```
...
<session-config>
  <cookie-config>
    <active>false</active>
  </cookie-config>
</session-config>
...
```

URL 書き換えの使用

Cookie を無効にし、ユーザーに提供する URL で request オブジェクトの **encodeURL** メソッドを呼び出すことによって、URL 書き換えをセッションメカニズムとして使用できます。JRun がエンコードされた URL を構築すると、JRun は、**JSESSIONID** いう名前のクエリ文字列パラメータとしてセッション ID をそのエンコードされた URL に付加します。そして、その URL クエリ文字列からセッション ID を取得することによって、JRun はそのユーザーに関連付けられた session オブジェクトにアクセスできます。

URL 書き換えの有効化

URL 書き換えは明示的には有効にしません。クライアントのブラウザで Cookie が無効の場合や、管理者が Cookie を無効にした場合、JRun は、JRun の設定を管理者に確認せずに URL 書き換えを使用します。

しかし、セッションメカニズムとして URL 書き換えを使用するには、JRun に対して次の作業を行う必要があります。

- JRun で Cookie をオフにするか、クライアントで Cookie を無効にする必要があります。
- request オブジェクトの **encodeURL** メソッドを使用して、ユーザーに送信される URL をエンコードします。

  メモ：クライアントがブラウザで Cookie を無効にしていないか、または JRun で Cookie を無効にしていないかぎり、たとえ URL で **encodeURL** メソッドを使用している場合でも、JRun はセッション ID をこれらの URL に追加しません。

書き換えた URL の使用

サーブレットや JSP で URL 書き換えを実行するには、session オブジェクトを使用するページで、request オブジェクトの **encodeURL** メソッドを使用して、クライアントに返す各 URL をエンコードする必要があります。次の例では URL をエンコードします。

```
String originalURL = "/servlet/URLRewriter";
String encodedURL = response.encodeURL(originalURL);
out.println("<P><a href=¥"" + encodedURL + "¥">This link contains a
        JSESSIONID</a> (if cookies are disabled)");
```

URL をエンコードし、出力をクライアントに送信する前に session オブジェクトのインスタンスを作成する必要があります。

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

URL 書き換えのテスト

ローカルマシンで URL 書き換えをテストするには、いくつかの手順を行う必要があります。

URL 書き換えのテスト

1 次の手順のいずれかを行います。

- ブラウザで Cookie を無効にします。たとえば、Internet Explorer 6 では、[インターネットオプション] の [セキュリティ] タブで Cookie を無効にします。
- JRun で Cookie を無効にします。JRun で Cookie を無効にする方法の詳細については、159 ページの「Cookie の無効化」を参照してください。

2 session オブジェクトを作成し URI をエンコードするサーブレットを作成します。例は samples サーバーにあります。

3 サーブレットをコンパイルします。

4 ページを開いて、マウスをページのリンク上に合せます。これによって、各 URI に ;JSESSSIONID=xxxxxxxxxxxxxxxx が付加されます。ここで、連続した x はセッション ID を表します。

URL 書き換えのカスタマイズ

セミコロンに続く URI のパス情報の最後にセッション ID を付加すると、URL 書き換えを明示的に強制できます。これは多くの場合 URL の最後に現れますが、JSESSIONID はクエリ文字列パラメータではなく、パス情報の一部です。

次の例では、明示的に JSESSIONID パラメータを設定します。

```
...
HttpSession thisSession = request.getSession(true);
String originalURL = "/servlet/Logger";
String newURL = originalURL + ";JSESSIONID=" + thisSession.getId() +
        request.getQueryString();
...
```

Hidden フォームフィールドの使用

FORM タグで送信した隠しフィールドを使用して、クライアントからサーバーにセッション ID を渡すことができます。

セッションメカニズムとして Hidden フォームフィールドを使用することには、次の欠点があります。

- すべてのページに FORM タグが必要です。
- セッション関係のリクエストを行うには、ユーザーは FORM を送信する必要があります。

Hidden フォームフィールドを使用するには、セッションを作成した後そのセッションの ID を FORM 要素の **session** フィールドの値として使用します。次の例は、FORM が送信されたときに、クライアントからサーバーにセッション ID を渡すサーブレットの FORM を示しています。

```
...
HttpSession session = request.getSession(true);
out.println("<FORM METHOD=¥"POST¥" ACTION=¥"HiddenForm¥">");
out.println("<INPUT TYPE=¥"HIDDEN¥" NAME=¥"session¥" VALUE=¥"" +
        session.getId() + "¥">");
out.println("<INPUT TYPE=¥"Submit¥" VALUE=¥"Submit¥"></FORM>");
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

JSP ページとしてのサーブレットの作成

前のセクションでは、Java で作成したサーブレットの例を示しました。JRun は、Java コーディングにあまり依存しない、サーブレットの開発方法もサポートしています。それが JSP です。JSP によって、HTML とスクリプトコードの組み合わせを含むテキストファイルからサーブレットを作成できます。

JSP 内のスクリプトコードは、JSP シンタックスと、通常は JavaScript (ECMAScript のサブセット) または Java の組み合わせになります。JSP シンタックスの詳細とスクリプト言語の選択方法の詳細については、247 ページの第 10 章「JSP プログラミングテクニック」を参照してください。

JSP ファイルは、最初にリクエストされた時点で、JRun によって Java ソースファイルに変換され、続いて Java クラスファイルにコンパイルされます。したがって、Java コードを 1 行も書かずに、Java サーブレットを作成できます。JSP ファイルのランタイムイメージは Java クラスファイルになるため、Web サーバーは Java で作成されたファイルと、JSP として作成されたファイルの違いを認識できません。JSP ファイルからは、Java で書かれた他のサーブレットや JSP ファイルとして実装された他のサーブレットを呼び出すこともできます。

次の例は、ブラウザ画面に "`Hello World`" と 5 回表示する簡単な JSP ページです。

```
<html>
  <head>
  <title>Greetings</title>
  </head>
  <body>
    <% for(int i=0;i<5;i++) { %>
      <h1>Hello World!</h1>
    <% } %>
  </body>
</html>
```

JSP のファイル名の末尾には、拡張子 `.jsp` が付きます。JRun は JSP へのリクエストを認識し、その JSP を実行可能な Java サーブレットに トランスレートします。JSP 開発の詳細については、247 ページの第 10 章「JSP プログラミングテクニック」を参照してください。

同期化

通常、サーブレット開発者は、サーブレットインスタンスとの関連で同期化を認識していますが、この作業を実行するのはスレッドです。多くのスレッドが同時に実行されることもあります。

各インスタンスは複数のスレッドで実行される可能性があるため、オブジェクトスコープ変数などの共有リソースを使用する同期化で起こり得る問題を認識する必要があります。同期化とは、1 つのコードをシングルスレッドのように実行させることです。スレッド管理は、JMC を使用し、各 JVM に別々のスレッドパラメータを指定して制御します。

メモ：スレッド管理は高度な内容であるため、このマニュアルでは説明しません。詳細については、Java スレッド管理に関する市販の解説書を参照してください。

クラススコープのインスタンス変数への同時アクセスを防止するには多くの方法があります。次に例を示します。

- オブジェクトスコープ変数を更新する行を同期化します。
- `SingleThreadModel` インターフェイスを実装します。
- オブジェクトスコープ変数にアクセスするメソッドを同期化します。
- キーワード `synchronized` を `doXxx` メソッドの署名に含めます (この方法はお勧めしません)。

これらのテクニックについては、次のサブセクションで説明します。

メソッド署名での synchronized キーワードの使用

メソッド署名で `synchronized` キーワードを使用すると、メソッド全体へのアクセスを同期化できます。

```
...
public class TestCaller extends HttpServlet
{
  int visitorCounter = 0;

  // コンセプトの説明のみを目的とした例なので、実際には実行しないでください。
public synchronized void doGet(HttpServletRequest servReq,
  HttpServletResponse servRes) throws IOException, ServletException {
...
```

メモ：この例は、コンセプトについて説明するためのものです。この同期化テクニックはパフォーマンスを低下させるため、実際にはほとんど使用しません。

同期化されたコードの使用

`synchronized` ブロックを使用して、オブジェクトスコープ変数へのアクセスを同期化できます。

```
...
int thisCount;
synchronized(this) {
  // visitorCount はオブジェクトスコープ変数です。
  thisCount = visitorCounter++;
}
```

```
out.println("<p>You are visitor number " + thisCount);
...
```

この例では、synchronized ブロック内でカウンタを増分し、それによってオブジェクトスコープ変数へのシングルスレッドアクセスを保証します。

SingleThreadModel インターフェイスの使用

次の例に示すように、SingleThreadModel インターフェイスは JRun に一群のサーブレットインスタンスを作成して、各インスタンスについて同時スレッドが service メソッドを実行しないように指示します。

```
...
public class testSync extends HttpServlet
  implements SingleThreadModel {
...
```

JRun は SingleThreadModel を実装するサーブレットのインスタンスを複数作成するために、オブジェクトスコープのインスタンス変数がすべてのインスタンスについて同じである必要がある場合、このテクニックは使用できません。たとえば、このテクニックをヒットカウンタには使用できません。しかし、オブジェクトスコープのインスタンス変数を使用しないサーブレットの場合や、バッファー変数やデータベース接続など、オブジェクトスコープのインスタンス変数が異なってもよい場合には、SingleThreadModel の使用は効果的なテクニックです。

オブジェクトスコープ変数にアクセスするメソッドの同期化

次の例に示すように、オブジェクトスコープ変数へのすべてのアクセスが同期化されたメソッドで行われるアクセス方式を実装できます。

```
...
public class TestSync extends HttpServlet {
  int visitorCounter = 0;

public void doGet(HttpServletRequest req, HttpServletResponse res)
  throws IOException, ServletException {
   incrementCount();
...
public synchronized void incrementCount() {
  visitorCount++;
}
public synchronized int getCount() {
  return visitorCount;
}
...
```

データベースの使用

ダイナミックな Web サイトを生成する最も一般的な方法は、データベースを使用することです。Web アプリケーションはユーザーリクエストから入力を解釈し、データベースに保管されたデータを使用してダイナミックページで応答します。このセクションでは、JDBC (Java Database Connectivity：Java データベース接続) メカニズムを説明し、JRun データソースを使用して Web アプリケーションのデータベースにアクセスする方法を説明します。

JRun のインストールには、PointBase Embedded データベースが含まれています。JRun は、いくつかのサンプルアプリケーションでこのデータベースを使用しますが、Web アプリケーションでもこれを使用できます。データベースの使用方法の詳細については、『JRun 管理者ガイド』を参照するか、PointBase ヘルプファイル (<JRun のルートディレクトリ >/pointbase/docs/server_embedded/GettingStarted/serverindex.html) をご覧ください。

JDBC の理解

JRun でのデータベースアクセスは、JDBC API (Java database connectivity API：Java データベース接続 API) を使用して実行します。JDBC は、Sun が提供するドライバマネージャと JDBC ドライバを使用します。JDBC ドライバは、Sun によって JDBC-ODBC ブリッジが、サードパーティーベンダーからは他の JDBC ドライバが提供されています。

JDBC ドライバのタイプ

次の表で説明されているように、JDBC ドライバにはいくつかのタイプがあります。

ドライバ	説明
ネイティブ API ドライバ	このタイプの JDBC ドライバは Type 2 ドライバとも呼ばれています。これは Java コードをネイティブデータベースライブラリで ラップします。Type 2 ドライバを使用するには、JRun はデータベースクライアント API ライブラリにアクセスできる必要があります。
ネットプロトコルドライバ	このタイプの JDBC ドライバは Type 3 ドライバとも呼ばれています。これは汎用ネットワークプロトコルとミドルウェアコンポーネントを使用してデータベースと通信します。
ネイティブプロトコルドライバ	このタイプの JDBC ドライバは Type 4 ドライバとも呼ばれています。これはデータベース特有のネイティブプロトコルを使用してデータベースと通信します。

JRun は、Type 4 JDBC ドライバ付きで販売され、これを使用してさまざまなリレーショナルデータベースにアクセスできます。詳細については、JRun ドキュメントとともに配布されている『DataDirect Connect JDBC User's Guide and Reference』を参照してください。

データベース接続

Web アプリケーションにはデータベースに接続する方法が数多くあります。最も一般的な 2 つの方法は、JRun データソースを使用する方法と手動接続を作成する方法です。次の表で、これら 2 つの方法を説明します。

接続方法	説明
JRun データソース	JMC を使用して JRun データソースを設定した場合、データベースに接続するには、アプリケーションでデータソース名による JNDI 検索を実行する必要があるだけです。この処理については、168 ページの「JRun データソースの使用」で説明します。 JRun データソース接続方法を使用します。ビルトイン JRun コネクションプールメカニズムを透過的に使用します。 JRun には、ユーザーが使用できる Merant データベースドライバがあります。JMC では、ドライバクラス名、URL、ポートなどの多くの一般的な JDBC ドライバの設定に関して、ドライバ特有のデフォルト設定が用意されています。これらの設定は変更することもでき、必要な場合は、他の JDBC ドライバを使用することもできます。 詳細については、JMC オンラインヘルプを参照してください。JMC で JDBC データソースを設定する手順については、『JRun 入門』を参照してください。
手動データベース接続	Web アプリケーションで JDBC 接続を手動で作成するには、コードに次の情報を与える必要があります。 • **データベースドライバ**　`Class.forName` メソッドを使用してデータベースドライバのインスタンスを生成します。 • **データベース接続オブジェクト**　`DriverManager.getConnection` メソッドを使用して、Connection オブジェクトを確立します。 • **データベース URL**　データベースの URL には、プロトコル (jdbc)、odbc や sequelink などのドライバのサブプロトコル、データベースを識別するドライバ依存データが含まれています。データベース URL のフォーマットとコンテンツについては、ご使用のデータベースドライバのドキュメントを参照してください。 手動で作成したデータベース接続をプールするには、カスタムのコネクションプールメカニズムを実装する必要があります。 ドライバクラス名とデータベース URL の詳細については、ご使用の JDBC ドライバのドキュメントを参照してください。

JRun データソースの利点

JRun データソースによって、ビルトインコネクションプールだけでなく、サーブレットの移植性が与えられます。マクロメディアでは、JRun ではこの方法を使用してデータベースにアクセスすることをお勧めします。次のセクションでは、JRun データソース使用の利点を説明します。

設定の簡単さ

JRun には JMC に [JDBC データソース] パネルがあり、ここでデータソースの作成、編集、削除が簡単にできます。詳細については、JMC オンラインヘルプを参照してください。

移植性

データベースに接続するには、次の情報を与える必要があります。

- データベース URL
- データベースドライバ
- データベースサーバーの IP アドレスとポート番号
- ユーザー名とパスワード
- その他のベンダ引数

JRun データソースを使用すると、コードは、Web アプリケーションにハードコードされた JDBC ドライバ情報を使用するのではなく、データソース名だけを使用してデータソースを参照します。データソースは 1 度だけ設定します。これによって、サーブレット、EJB、JSP がいくつでもこのデータソースにその名前でアクセスできます。

データソースを使用して、JMC のデータベース定義を更新できます。データベースとデータベースサーバーの情報はコードの外に保管されるので、その情報はアプリケーションを再コンパイルせずにいつでも変更できます。

データソース使用によるもう 1 つの利点は、ユーザー名やパスワードなどの重要なデータベース接続情報を JRun 管理者以外のすべての人物から隠すことができることです。Web アプリケーション開発者は、この情報を指定せずにデータベースに接続できます。

コネクションプール

データベースとの接続にはコストがかかります。JRun データソースは、ビルトインプールメカニズムを使用します。JRun 定義のデータソース使用による利点は、JRun がデータベースコネクションプールを作成することです。データベースに接続するときは、このプールメカニズムによって、確立された接続をプールから取得できます。接続を作成する必要がないので、応答時間はずっと短くなります。

アプリケーションで接続を閉じると、JRun は実際に接続を閉じるのではなく、その接続をプールに返します。

コネクションプールをオフにして、独自のカスタムコネクションプールを実装できます。データソース設定の詳細については、JMC オンラインヘルプを参照してください。

サンプルデータソース

JRun には、samples JRun サーバーのためにあらかじめ定義された 3 つのデフォルトデータソースがあります。これらのデータソースは、`com.pointbase.jdbc.jdbcUniversalDriver` データベースドライバを使用します。次の表で、これらのデータソースを説明します。

データソース名	説明
compass	Compass サンプルアプリケーションで使用します。このアプリケーションはデフォルトで <JRun のルートディレクトリ >/servers/samples/compass-ear/ の samples JRun サーバーにインストールされます。
samples	Techniques サンプルアプリケーションで使用します。このアプリケーションはデフォルトで <JRun のルートディレクトリ >/servers/samples/techniques-ear/ の samples JRun サーバーにインストールされます。
smarticket	SmarTicket サンプルアプリケーションで使用します。このアプリケーションはデフォルトで <JRun のルートディレクトリ >/servers/samples/smarticket.ear の samples JRun サーバーにインストールされます。

JRun データソースの使用

JRun データソースを使用すると、InitialContext オブジェクトへの JNDI ルックアップを通じて、データソースにアクセスできます。サーブレットでデータソースを使用するには、JDBC API を使用する次のパッケージをインポートする必要があります。

- `javax.naming.InitialContext`
- `javax.sql.DataSource`
- `java.sql.Connection`
- `java.sql.ResultSet`

さらに、データベース接続を使用する方法に応じて、次の Java パッケージから 1 つをインポートする必要があります。

- `java.sql.Statement`
- `java.sql.PreparedStatement`

次のコード例では、JRun データソースサービスを使用して JDBC データソース情報にアクセスします。

```
...
import javax.naming.*;
import javax.sql.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
...
Connection dbConnection = null;
ResultSet dbResultSet = null;
ResultSetMetaData rsmd = null;
try {
  InitialContext ctx = new InitialContext();
  DataSource ds = (DataSource) ctx.lookup("compass");
  dbConnection = ds.getConnection();
  Statement stmt = dbConnection.createStatement();
  dbResultSet = stmt.executeQuery("SELECT * FROM user");
  rsmd = dbResultSet.getMetaData();
} catch (Exception e) {
}
...
// 結果セットと結果セットのメタデータをここで処理します。
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

データベースアクセスパフォーマンスの改善

Web アプリケーションでデータベースのパフォーマンスを改善するには、次のテクニックを使用します。

- データベースコネクションプールの使用
- PreparedStatement の使用
- connection、statement、resultset オブジェクトを閉じる
- フェッチの制限
- スタティックデータのキャッシュ

詳細とコード例については、231 ページの「JDBC の最適化」を参照してください。

制御の受け渡し

クライアントがリクエストするリソースは、時にはサーバーがサービスするリソースでないことがあります。次の例は、クライアントが明示的に送信していないリクエストにサーバーが応答するときの例です。

- **認証 (Authentication)**　リクエストに対して重要なデータで応答する場合は、その前にユーザー認証で確認する必要があります。ユーザー認証が成功しなかった場合は、ユーザーにエラーページを転送します。
- **転送 (Redirection)**　サイトが移動しました。新しい Web サイトに移動するためにユーザーがクリックする必要があるリンクを掲示するかわりに、そのサイトにユーザーを自動的に転送できます。

次の方法を使用して、あるサーブレットから別のサーブレットにリクエストを渡します。

- RequestDispatcher オブジェクト
- `response.sendRedirect` メソッド

forward メソッドを使用する場合、呼び出し側のサーブレットから出力ストリームに書き込むことはできません。必要な場合は、呼び出し側のサーブレットは、`setAttribute` メソッド (JSP の request オブジェクト) を使用して属性を設定することで、目的のサーブレットに情報を渡します。この場合、目的のプログラムは次の方法で属性にアクセスできます。

- サーブレットは、ServletRequest オブジェクトの `getAttribute` メソッドを使用して、これらの属性にアクセスできます。
- JSP は、暗黙の request オブジェクトの `getAttribute` メソッドを使用して、これらの属性にアクセスできます。

RequestDispatcher の使用

RequestDispatcher オブジェクトの `forward` メソッドを使用して、他のサーブレットや JSP に制御を渡すことができます。このセクションでは、この方法を説明します。

サーブレットに制御を渡す方法

RequestDispatcher オブジェクトへのリファレンスは、getRequestDispatcher メソッドで取得します。このメソッドは ServletContext オブジェクトと ServletRequest オブジェクトの両方にあります。唯一の違いは、`ServletRequest.getRequestDispatcher` で指定するパス名には先頭のスラッシュが不要なことです。したがって、相対 URL を使用できます。`ServletContext.getRequestDispatcher` の場合は、先頭のスラッシュが必要です。次の例では、`ServletContext.getRequestDispatcher` を使用します。

次の例では、他のサーブレットに制御を渡します。

```
...
ServletContext sc = this.getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher("/servlet/callMe");
if (rd !=null) {
  // 制御をサーブレットに渡します。
  try {
   rd.forward(req, resp);
  } catch (Exception e) {
    sc.log("Problem invoking servlet.", e);
  }
}
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

JSP に制御を渡す方法

RequestDispatcher オブジェクトを使用して、サーブレットから JSP に制御を渡すことができます。このテクニックを使用すると、HTML ページなどの Web リソースに制御を渡すこともできます。

次の例では、JSP に制御を渡します。

```
...
ServletContext sc = this.getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher("/test.jsp");
if (rd !=null) {
  // JSP に制御を渡します。
  try {
   rd.forward(req, resp);
  } catch (Exception e) {
    sc.log("Problem invoking JSP.", e);
  }
}
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

sendRedirect メソッドの使用

response オブジェクトには `sendRedirect` メソッドがあり、このメソッドでリクエストを新しいターゲットに転送できます。また、`sendRedirect` メソッドは `Location` HTTP ヘッダーをレスポンスに追加し、ステータスコードを 302 (Found) に変更します。リクエストが相対 URL や絶対 URL などの場所を参照するように設定できます。相対 URL を指定した場合、JRun はレスポンスを生成する前に現在の URI を使用して、その相対 URL を絶対 URL に変換します。

`sendRedirect` メソッドの署名は次のとおりです。

```
public void sendRedirect(String URL)
```

次のサーブレット例では、`target_location` リクエストパラメータから位置を取得して、そのリクエストをその位置に転送します。

```
...
String target_location = req.getParameter("target_location");
resp.sendRedirect(target_location);
...
```

`target_location` の値は、URL から他のサーブレット名までの String を指定できます。たとえば次のとおりです。

- **ReflectContext** (またはサーブレット名)　JRun はこれを相対 URL と解釈します。たとえば現在の URI が /servlet ディレクトリである場合は、JRun はこれを絶対 URI に変換して Location ヘッダーに設定します。変換後のヘッダーは、http://localhost:8100/servlet/ReflectContext となります。
- **http://www.google.com**　これは、絶対 URI として解釈され、JRun はこれに等しい `Location` ヘッダーを設定します。

`sendRedirect` メソッドを使用するときは、次の注意事項に配慮してください。

- 古いブラウザには、転送をサポートしていないものがあります。
- `GET` リクエストを処理するときは、`sendRedirect` だけが使用できます。
- `Referer` ヘッダーを使用すると、リクエストが転送された後で、そのリクエストの送信元が分かります。
- `sendRedirect` メソッドでセッションを使用するときは、メソッドを呼び出す前に必ず URL をエンコードします。

TCPMonitor ユーティリティを使用すると、JRun がリクエストメッセージとレスポンスメッセージに設定したヘッダーを表示できます。TCPMonitor の使用方法の詳細については、236 ページの「TCPMonitor の使用」を参照してください。

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

Cookie の処理

Cookie は、サーバーサイドアプリケーションによって使用される一般的なメカニズムで、個々のブラウザに情報を保管します。ブラウザに保管された Cookie はサーバーサイドアプリケーションによって取り出されます。Cookie を使用することで、Web アプリケーションは各ブラウザに特定の変数を作成できます。これらの変数には、ユーザー名や最終アクセスの日付を保管できます。

Cookie を使用したセッション追跡を有効にした場合、JRun は `JSESSIONID` と名付けたセッション追跡 Cookie を作成します。このセッション追跡 Cookie の名前は、jrun-web.xml ファイルの `cookie-config` セクションで変更できます。詳細については、155 ページの「jrun-web.xml ファイルでのセッション設定」を参照してください。

Cookie の有効期限定義

Cookie は、HTTP プロトコルのステートレスという性質を補完するセッションパーシスタンスメカニズムを提供します。Cookie にはテンポラリ Cookie とパーマネント Cookie があります。

- **テンポラリ**　ブラウザインスタンスを終了するまで有効です。テンポラリ Cookie は、セキュアシステムへのアクセスを認証する際に使用するユーザー名やパスワードを保持するのに適しています。デフォルトでは、サーブレット API はテンポラリ Cookie を作成します。
- **パーマネント**　その Cookie が期限切れになるか削除されるまで有効です。パーマネント Cookie は、ユーザー名や最終アクセスの日付などの情報を保持するのに適しています。パーマネント Cookie を作成するには、cookie オブジェクトの `setMaxAge` メソッドを使用します。

Cookie の寿命が、その Cookie のブラウザでの有効期間です。この値は Cookie に保管されます。jrun-web.xml ファイルの `cookie-max-age` でこの設定を定義します。ブラウザサイドでは、`cookie-max-age` の有効期限が切れると、ブラウザはその Cookie ファイルを削除します。詳細については、155 ページの「jrun-web.xml ファイルでのセッション設定」を参照してください。

Cookie のセッションタイムアウトは、セッションがアクティブでない間サーバーのメモリに JRun が Cookie を維持する期間を示します。154 ページの「セッションタイムアウトの変更」で説明されているように、セッションがそのセッションタイムアウトに達すると、JRun はサーバーサイドの session オブジェクトを廃棄します。

Cookie の作成

このセクションでは、Cookie を明示的に作成する方法を説明します。Cookie をセッションパーシスタンスのために使用している場合は、`getSession` メソッドが呼び出されると JRun は暗黙的に Cookie を作成します。

Cookie の作成

1 次のコードに示すように、Cookie に保存する値を確立します。

```
Date dt = new Date();
String todayString = dt.toString();
```

2 次のコードに示すように、新規 Cookie オブジェクトを作成します。

```
// Cookie インスタンスを作成します。
// この例では、現在の日付と時刻を保存します。
Cookie lastVisit = new Cookie("lastVisit", todayString);
```

3 次の行に示すように、秒単位で Cookie のタイムアウトを設定します。

```
lastVisit.setMaxAge(60*60*24*365); //1 年間この Cookie を廃棄しません。
```

4 次の行に示すように、Cookie オブジェクトをサーブレットの response オブジェクトに関連付けることで、Cookie をブラウザに返します。

```
resp.addCookie(lastVisit);
```

Cookies へのアクセス

このセクションでは、Cookie に保管された値に明示的にアクセスする方法を説明します。セッションパーシスタンスのために Cookie を使用している場合、JRun にはセッションの値にアクセスする便利な方法が用意されています。

Cookie へのアクセス

1 次の例に示すように、HttpServletRequest オブジェクトの `getCookies` メソッドで Cookie にアクセスします。

```
Cookie[] myCookies = req.getCookies();
```

2 Cookie オブジェクトの `getName` メソッドと `getValue` メソッドを使用して Cookie とその値にアクセスします。次の例は、Cookie の名前とそれに関連付けられた値を表示します。

```
for(int i=0; i<myCookies.length; i++) {
  out.println("Cookie の名前：" + myCookies[i].getName());
  out.println("値：" + myCookies[i].getValue() + "<br>");
}
```

Cookie に代わるもの

Cookie はステート管理で使用される極めて一般的なメカニズムであり、主要なすべての商用ブラウザでサポートされていますが、サポートの内容はブラウザによって異なります。また、クライアントはブラウザの設定で Cookie サポートを無効にできます。

セッションの詳細については、151 ページの「セッションの操作」を参照してください。

次の代替メカニズムを使用して、セッションのステートを維持することができます。

- URL 書き換え (159 ページの「URL 書き換えの使用」を参照)
- Hidden フォームフィールド (161 ページの「Hidden フォームフィールドの使用」を参照)

コンテンツのインクルード

次のメソッドを使用して、サーブレットにコンテンツをインクルードできます。

- RequestDispatcher オブジェクトの `include` メソッド
- ServletContext オブジェクトの `getResource` メソッド

RequestDispatcher オブジェクトの使用方法の詳細については、170 ページの「制御の受け渡し」を参照してください。

include メソッドの使用

RequestDispatcher オブジェクトの `include` メソッドを使用すると、サーブレットに複数のタイプのコンテンツをインクルードできます。

- **テキスト** RequestDispatcher オブジェクトがテキストファイルをラップしている場合、`include` メソッドはそのテキストを出力ストリームにコピーします。このテキストには HTML タグを含めることができます。
- **サーブレット** RequestDispatcher オブジェクトがサーブレットをラップしている場合、`include` メソッドはそのサーブレットを呼び出します。
- **JSP** `RequestDispatcher` オブジェクトが JSP をラップしている場合、`include` メソッドは JSP を呼び出します。

`include` メソッドを使用すると、呼び出し側サーブレットは `include` メソッドの呼び出し前後に `ServletOutputStream` オブジェクトや `PrintWriter` オブジェクトに書き込むことができます。必要な場合は、170 ページの「制御の受け渡し」で説明されているように、ServletRequest オブジェクトの `setAttribute` メソッドを使用して、ターゲットのサーブレットや JSP に情報を渡すことができます。また、151 ページの「セッションの操作」で説明されているように、session オブジェクトを使用しても情報を渡すことができます。

次の例では、サーブレットをインクルードします。

```
...
PrintWriter out = resp.getWriter();
ServletContext sc = this.getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher("/servlet/includeMe");
if (rd !=null) {
  try {
    // インクルードされたサーブレットは、それ自体のバッファーだけを制御します。
    rd.include(req, resp);
  }
  catch (Exception e) {
    sc.log("Problem invoking servlet.", e);
  }
}
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

getResource メソッドの使用

ServletContext オブジェクトの getResource メソッドを使用すると、サーブレットにコンテンツをインクルードできます。getResource メソッドは URL オブジェクトを返します。その後は、この URL オブジェクトを使用してこのコンテンツにアクセスできます。URL オブジェクトを使用する利点は、そのオブジェクトをブラウザに返す前にコンテンツを解析できることです。このテクニックを使用すると、/WEB-INF ディレクトリ内のファイルなど、他の方法ではユーザーが直接アクセスできないコンテンツもインクルードできます。

次の例では、getResource メソッドを使用してコンテンツをインクルードします。

```
...
resp.setContentType("text/html");
ServletOutputStream out = resp.getOutputStream();
ServletContext sc = this.getServletContext();
try {
  URL u = sc.getResource("/includedText.htm");
  if (u !=null) {
    // コンテンツにアクセスし、InputStream にキャストします。
    InputStream in = (InputStream)u.getContent();
    byte[] buf = new byte[255];
    int numRead = in.read(buf);
    while(numRead != -1){
      out.write(buf, 0, numRead);
      numRead = in.read(buf);
    }
  }else {
    out.println("<p>u was null");
  }
} catch (Exception e) {}
...
```

他の HTML ページからのコンテンツ取得

J2EE を使用すると、インターネットから Web ページを簡単に取得でき、それらのページを解析してそのコンテンツを独自のページにインクルードできます。このテクニックの一般的な応用には、野球などのボックススコアや株価表示を独自のページに追加してダイナミックデータを提供することがあります。

ターゲットの Web サイトから、既知の文字列を使用して解析できる規則正しいデータが出力される必要があります。規則正しいデータとは、各部分が一連の区切り文字列で区切られていることを意味します。たとえば、価格情報を取得するには、データ取得に必要な呼び出しがすべて成功するように、ターゲットのページでは製品価格が同じ HTML タグで囲まれ、同じように表示されている必要があります。

たとえば、Web サイト Funagain (www.funagain.com) では、ゲーム ID を使用してダイナミックページを生成します。Funagain は、各ゲームの価格、デザイナー、メーカーなどの多くの属性をリストするページを生成します。次の例に示すように、これらの属性は、Funagain Web ページの HTML では、Hidden フォームフィールドとして現れます。

```
<INPUT TYPE="HIDDEN" NAME="manufacturer" VALUE="Amigo">
```

この例では、Funagain でゲームの詳細を示す HTML ページは、メーカーを定義する Hidden フォームフィールドが含まれています。Funagain は、すべてのゲームについて同じ規則性でページを生成します。同様に、Yahoo! Finance (finance.yahoo.com) では、同じデータフォーマットを使用して株式相場のページを生成しています。

このセクションのテクニックを使用すると、メーカーの Hidden フォームフィールドの値を抽出できます。

Web アプリケーションの他のページからダイナミックコンテンツをインクルードする方法

1 URL を構築します。この例は、Funagain のベース URL から成る URL に、フォームフィールドに入力したゲーム ID を追加します。次の例は、ターゲットの URL を構築する方法を示しています。

```
...
String gameID = request.getParameter("gameID");
String urlString = "http://kumquat.com/cgi-kumquat/funagain/" +
     gameID;
...
funagainURL = new URL(urlString);
...
```

2 次の行に示すように、ターゲットの Web サイトへの URL 接続を開きます。

```
funagainConnection = funagainURL.openConnection();
```

3 次の例に示すように、新規の接続から InputStream を取得します。

```
webPageInputStream = funagainConnection.getInputStream();
```

4 次の例に示すように、ターゲットの InputStream をバッファーに読み込みます。

```
...
StringBuffer webPageDataBuffer = new StringBuffer(32000);
int totalBytesRead = 0;
boolean moreToRead = true;
byte[] readBuf = new byte[4096]; // Web ページを 4K のチャンクで読み込みます。
while (moreToRead) {
  int numBytesRead = 0;
  try {
    numBytesRead = webPageInputStream.read(readBuf);
  } catch (IOException e) {
    moreToRead = false;
    numBytesRead = -1;
  }
  if (numBytesRead > 0) {
    totalBytesRead += numBytesRead;
    webPageDataBuffer.append(new String(readBuf, 0,
     numBytesRead));
  } else {
    moreToRead = false;
  }
}
...
webpageDataBuffer.setLength(totalBytesRead);
...
```

5 次の行に示すように、Web ページのバッファーを String に変換します。

```
String webPageData = webPageDataBuffer.toString();
```

6 次の例に示すように、抽出するターゲットデータ用に区切り文字列を定義します。

```
label[3] = "Year: ";
predetails[3] = "year¥" VALUE=¥"";
postdetails[3] = "¥">";
```

この例では、次の String の出現を開始区切り文字列として検索します。

```
year" VALUE="
```

この例では、¥"> を検索し、それを終了区切り文字列に設定します。

ターゲット Web サイトの次の行が検索に一致します。

```
<INPUT TYPE="HIDDEN" NAME="year" VALUE="1999">
```

1999 は predetails 区切り文字列と postdetails 区切り文字列で区切られます。

7 次の例に示すように、区切り文字列に一致する String を検索して、Web ページの String 表現を解析します。

```
...
int preStringLoc;
int postStringLoc;
preStringLoc = WebPageData.indexOf(predetails[i]);
postStringLoc = WebPageData.indexOf(postdetails[i], preStringLoc);
if (preStringLoc == -1 || postStringLoc == -1) {
  details[i] = "該当なし";
} else {
  details[i] = WebPageData.substring(preStringLoc +
     predetails[i].length(), postStringLoc);
}
...
```

8 次の例に示すように、区切り文字列に一致する目的のデータをプリントします。

```
...
for (int i = 0; i < details.length; i++) {
  out.println(label[i] + details[i]);
  out.println("<BR>");
}
...
```

サンプルサーブレットを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

第 7 章
フィルタ

Java Servlet API 2.3 仕様では、前処理および後処理のための HTTP リクエストおよびレスポンスオブジェクトへのアクセスを提供するサーブレットフィルタが導入されました。この章では、フィルタの使用方法について説明します。また、後半に、フィルタに関する外部情報リソースを掲載します。

目 次

フィルタの概要

フィルタは、サーバーに送信される前にリクエストオブジェクトを処理したり、サーバーからクライアントに戻される間にレスポンスオブジェクトを処理したりします。また、フィルタは、チェーン内で呼び出すこともできるため、チェーン内のフィルタ間でリクエストやレスポンスをやり取りできます。フィルタを使用すると、次のタスクを実行できます。

- Web アプリケーションコンポーネントからフロー制御ロジックを排除します。
- サーバーがクライアントからリクエストオブジェクトを受信する前に、リクエストオブジェクトの評価や変更を行います。
- クライアントがサーバーからレスポンスオブジェクトを受信する前に、レスポンスオブジェクトの評価や変更を行います。
- レスポンスのコンテンツを変更します。

フィルタの順番と用途は、Web アプリケーションの設定ファイルで定義されており、Web アプリケーション自体にはコンパイルされません。その結果、アプリケーションを再コンパイルせずに、フィルタを配列し直したり、フィルタの追加や削除を行ったりすることができます。このような Web アプリケーションとの疎結合性により、フィルタは、開発チームの多数のメンバーが行う多様なタスクを行う場合に便利です。

フィルタはしばしば、次のような形態で実装されています。

- リクエストディスパッチャー
- ユーザーのオーセンティケータおよびオーソライザ
- リクエストおよびレスポンスロガーおよびオーディタ
- フォームの検証
- 画像コンバータ
- データ圧縮および解凍ツール
- データ暗号化および復号化
- レスポンス出力トークナイザー
- リソースアクセスのトリガー
- XML トランスフォーマー
- コンテンツローカライザ

この章のフィルタサンプルをコンパイルする場合は、クラスパスに <JRun のルートディレクトリ >/lib/jrun.jar ファイルを含める必要があります。次に例を示します。

```
>javac -classpath .;c:/jrun4/lib/jrun.jar *.java -d c:/jrun4/servers/
        default/deploy/default-app/web-inf/classes/
```

この章のほとんどのサンプルは、`Filter` インターフェイスのすべてのメソッドをオーバーライドする必要がないように、`GenericFilter` クラスを拡張します。`GenericFilter` コードについては、184 ページの「汎用フィルタクラスの作成」を参照してください。

デフォルトの JRun サーバーでは、リクエストごとのページの総実行時間を表示するフィルタが含まれています。詳細については、240 ページの「実行時間の表示」を参照してください。

Filter インターフェイスについて

すべてのフィルタは javax.servlet.Filter インターフェイスを実装します。フィルタ API には、オーバーライドする必要がある 3 つのメソッドがあります。次の表で、これらのメソッドについて説明します。

メソッド	説明
void init(FilterConfig config)	JRun は、init メソッドを一度呼び出してフィルタを初期化します。その後フィルタを呼び出しても、サーバーを再起動しないかぎり init メソッドは呼び出されません。この init メソッドは、初期化パラメータにアクセスする可能性がある FilterConfig オブジェクトを定義します。
void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)	ほとんどの作業は doFilter メソッドで実行されます。doFilter では、リクエストおよびレスポンスオブジェクトに作用し、複数のフィルタが呼び出されます。使用するフィルタの数にかかわらず、フィルタはフィルタチェーンの一部です。フィルタチェーンは、次のフィルタに制御を渡したり、ターゲットリソースに制御を渡したりするときにフィルタが使用するオブジェクトです。 doFilter メソッドは、リクエストおよびレスポンスオブジェクトのデータおよびヘッダーを検証して変更できます。 このメソッドは FilterChain オブジェクトを使用してチェーン内の次のフィルタを呼び出します。その後も、リクエストおよびレスポンスのデータおよびヘッダーを再び検証して変更できます。
void destroy()	JRun は、フィルタが不要になったことを示すために、通常はシャットダウン時に destroy メソッドを一度呼び出します。destroy メソッドは通常、FilterConfig オブジェクトを廃棄することによってフィルタが使用するリソースを解放します。

簡単なフィルタサンプル

次のコードは、フィルタがオーバーライドする必要がある 3 つのメソッドを示しています。このサンプルでは、フィルタチェーンを呼び出すだけです。

```
import javax.servlet.*;

public class SimpleFilter implements Filter {

 public void init(FilterConfig filterConfig) {
  this.filterConfig = filterConfig;
 }

 public void doFilter(ServletRequest request, ServletResponse
        response, FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {
  chain.doFilter(request,response);
 }

 public void destroy() {
  this.filterConfig = null;
 }
}
```

FilterConfig オブジェクトについて

JRun は、FilterConfig オブジェクトを使用してフィルタを初期化します。FilterConfig オブジェクトは、次のメソッドを提供します。これらのメソッドによって、初期化パラメータおよび ServletContext オブジェクトにアクセスできます。

- `getFilterName()`
- `getInitParameter()`
- `getInitParameterNames()`
- `getServletContext()`

FilterConfig オブジェクトは、フィルタの `init` メソッド内で定義します。その後、`doFilter` メソッド内で、`init` メソッドで指定されている初期化パラメータを使用できます。

```
private FilterConfig filterConfig = null;
private String version;

public void init(FilterConfig filterConfig) {
 this.filterConfig = filterConfig;
 this.version = filterConfig.getInitParameter("version");
 System.out.println(" フィルタが初期化されました：" +
        filterConfig.getFilterName() + "version " +
        filterConfig.getInitParameter("version"));
 }

public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {
 version = getVersion(request);
 chain.doFilter(request,response);
}

public String getVersion(ServletRequest request) {
 return this.version;
}
```

フィルタの初期化パラメータは、web.xml デプロイメントディスクリプタのフィルタ定義内で定義します。詳細については、185 ページの「フィルタの定義」を参照してください。

あるいは、フィルタの `destroy` メソッド内で FilterConfig オブジェクトを廃棄します。コードは次のとおりです。

```
public void destroy() {
 this.filterConfig = null;
}
```

FilterChain オブジェクトについて

JRun は、FilterChain オブジェクトをフィルタの `doFilter` メソッドに渡します。各フィルタは、次のフィルタまたはターゲットリソースに制御を渡しますが、下流のフィルタが処理を終了すると、制御は最終的にチェーンに戻ります。スタックをワインドおよびアンワインドする方法としては、チェーン内の複数のフィルタを使用することができます。

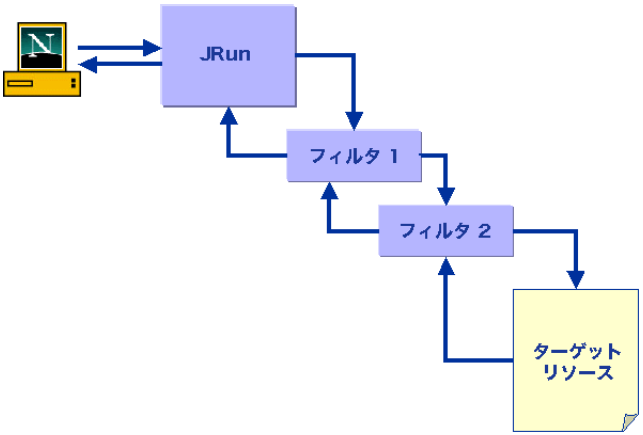
各フィルタは、リクエストの処理を終了すると、制御をチェーンに返します。それ以降、処理は次のフィルタ (存在する場合)、前のフィルタ、またはターゲットリソースで実行されます。

フィルタには、制御をいつチェーンに返すかを指定したり、制御を戻すかどうかを指定したりするロジックを含めることができます。最後のフィルタがチェーンを呼び出し、チェーン内にこれ以上フィルタがないと、リクエストはターゲットリソースに渡されます。

フィルタのロジックによって制御がチェーンに戻されないようになっている場合、チェーンは中断され、レスポンスはフィルタチェーンを通り抜けてクライアントに戻されます。チェーン内の下流にある他のフィルタ（存在する場合）は無視されます。

チェーンを中断したフィルタは、他のリソースにリクエストを転送できます。したがって、フィルタによってはリクエストを一度も受け取らない場合があります。フィルタのチェーンは、RequestDispatcher と、`forward` または `include` を使用して中断できます。送信されたリクエストは、コンテナマッピングがあってもチェーン内のフィルタに送られることはなく、クライアントに戻されます。

次の図にこの処理を示します。



FilterChain オブジェクトには次の 1 つのメソッドがあります。

```
doFilter(ServletRequest request, ServletResponse response)
```

リクエストとレスポンスをチェーン内の次のフィルタに渡すには、`doFilter` メソッドを使用します。現在のフィルタがチェーン内の最後のフィルタか、または唯一のフィルタである場合、リクエストおよびレスポンスはターゲットリソースに渡されます。ターゲットリソースが処理を終了すると、リクエストとレスポンスはチェーンのフィルタに戻されます。このフィルタは、リクエストとレスポンスを、チェーン内の前のフィルタ（存在する場合）に渡していきます。レスポンスがチェーン内の最初のフィルタまたは唯一のフィルタに達すると、レスポンスが渡されます。

フィルタはターゲットリソースにアクセスし、レスポンスを返すために、まず、フィルタの `doFilter` メソッド内でチェーンの `doFilter` メソッドを呼び出す必要があります。コードは次のとおりです。

```
public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {
 chain.doFilter(request,response);

}
```

汎用フィルタクラスの作成

すべてのフィルタに javax.servlet.Filter インターフェイスが実装されているので、各フィルタ内では、そのインターフェイスのメソッドをオーバーライドする必要があります。ほとんどのフィルタは init および destroy メソッド内でフィルタ固有のアクションを実行することはありません。したがって、これらのメソッドをオーバーライドする汎用クラスを使用すると、不要なコードを記述せずに済みます。

Java のオブジェクト指向性を利用するには、Filter インターフェイスを実装し、すべてのメソッドをオーバーライドする汎用クラスを作成してください。次に、フィルタで、新しく作成した汎用クラスを拡張し、必要なメソッドについてだけカスタマイズロジックを指定します。ほとんどの場合、doFilter メソッドだけをオーバーライドします。

次のコードは、javax.servlet.Filter インターフェイスを実装する GenericFilter クラスを示しています。フィルタはこのクラスを拡張し、必要なメソッドだけをオーバーライドします。

```
package jrunsamples.filters;

import javax.servlet.*;

public class GenericFilter implements Filter {
 public FilterConfig filterConfig;
 public void doFilter(ServletRequest request,ServletResponse response,
        FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {
  chain.doFilter(request, response);
 }

 public void destroy() {
  this.filterConfig = null;
 }

 public void init(FilterConfig filterConfig) {
  this.filterConfig = filterConfig;
 }
}
```

GenericFilter を次のように拡張することによって、簡単なフィルタサンプルを実装できます (181 ページの「簡単なフィルタサンプル」を参照)。

```
package jrunsamples.filters;

import javax.servlet.*;

public class SimpleFilter extends GenericFilter {

 public void doFilter(ServletRequest request, ServletResponse
        response, FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {
  // フィルタロジック
  chain.doFilter(request,response);
  // フィルタロジック
 }
}
```

フィルタの追加と削除

Web アプリケーションに新規フィルタを追加するには、フィルタを定義し、リソースにマッピングする必要があります。これは、Web アプリケーションのデプロイメントディスクリプタ (web.xml ファイル) を使用して行います。このセクションでは、フィルタを定義し、その実行順番を指定するためのオプションについて説明します。

フィルタは web.xml ファイル内で定義します。したがって、Web アプリケーションを再コンパイルせずに、フィルタの順番の変更、初期化パラメータの定義、フィルタの追加および削除を行うことができます。この疎結合アーキテクチャにより、フィルタの設定タスクとアプリケーションの記述タスクを分離できます。アプリケーションとその基盤リソースは、リクエストとレスポンスを処理するフィルタとは別個に動作します。

web.xml ファイル内では、フィルタの他の設定情報を定義できます。詳細については、188 ページの「初期化パラメータへのアクセス」を参照してください。

フィルタの定義

web.xml ファイルではまず、**web-app** ブロック内に名前とクラスによってフィルタを定義します。次のサンプルは web.xml ファイルのフィルタ定義部分です。

```
<web-app>
...
<filter>
 <filter-name>filter_name</filter-name>
 <filter-class>filter_class</filter-class>
</filter>
...
</web-app>
```

たとえば、TimingFilter を定義するには、次のコードを使用します。

```
<filter>
 <filter-name>TimingFilter</filter-name>
 <filter-class>jrunsamples.filters.TimingFilter</filter-class>
</filter>
```

また、フィルタブロック内に初期化パラメータを定義することもできます。詳細については、188 ページの「初期化パラメータへのアクセス」を参照してください。

フィルタのマッピング

フィルタをマッピングするリソースを定義するには、**url-pattern** 要素または **servlet-name** 要素を使用します。さまざまなフィルタ名を使用して同じフィルタを複数回宣言できます。そのため、URL マッピングとサーブレット名を組み合わせることによって 1 つのフィルタを複数のリソースにマッピングできます。

URL パターンへのフィルタのマッピング

web.xml ファイル内で URL パターンを使用してフィルタを定義できます。シンタックスは次のとおりです。

```
<web-app>
...
<filter>
 <filter-name>filter_name</filter-name>
 <filter-class>filter_class</filter-class>
</filter>
...
<filter-mapping>
 <filter-name>filter_name</filter-name>
 <url-pattern>pattern_for_resource</url-pattern>
</filter-mapping>
...
</web-app>
```

たとえば、TimingFilter を welcome.jsp ファイルにマッピングするには、次のコードを使用します。

```
<filter>
 <filter-name>TimingFilter</filter-name>
 <filter-class>jrunsamples.filters.TimingFilter</filter-class>
</filter>

<filter-mapping>
 <filter-name>TimingFilter</filter-name>
 <url-pattern>/welcome.jsp</url-pattern>
</filter-mapping>
```

`url-pattern` は、web.xml ファイル内でサーブレットをリソースにマッピングする際に使用する要素と同じなので注意してください。

URL マッピングを web.xml ファイル内で使用した場合と同様に、`url-pattern` 要素内でワイルドカードを使用することによって複数のリソースにフィルタをマッピングできます。たとえば、Web アプリケーションのコンテキストにおいて、フィルタをすべての JSP にマッピングするには、パターンを次のように定義します。

```
<url-pattern>/*.jsp</url-pattern>
```

サーブレット、JSP、およびスタティックコンテンツすべてにフィルタをマッピングするには、パターンを次のように定義します。

```
<url-pattern>/*</url-pattern>
```

指定したサーブレットへのフィルタのマッピング

フィルタは、指定したサーブレットにマッピングすることもできます。次のシンタックスは、フィルタを指定したサーブレットにマッピングします。

```
<web-app>
...
<servlet>
 <servlet-name>servlet_name</servlet-name>
 <servlet-class>servlet_class</servlet-class>
</servlet>
```

```
...
<filter>
 <filter-name>filter_name</filter-name>
 <filter-class>filter_class</filter-class>
</filter>
...
<filter-mapping>
 <filter-name>filter_name</filter-name>
 <servlet-name>servlet_name</servlet-name>
</filter-mapping>
...
</web-app>
```

チェーン内のフィルタの順番指定

JRun は、リクエスト URI が web.xml ファイル内で一致する URL パターンとサーブレットに従って、各リクエストのフィルタの実行順番を指定します。JRun はまず、**url-pattern** の一致候補を検証します。web.xml ファイル内ではこの順番で url-pattern の一致候補が並んでいます。url-pattern の後には **servlet-name** の一致候補が続きます。web.xml ファイル内ではこの順番で servlet-name の一致候補が並んでいます。

次のサンプルでは、フィルタ 1 およびフィルタ 3 は URL マッピングを使用してリソースにマッピングされ、フィルタ 2 はサーブレット名マッピングを使用します。したがって、JRun は、フィルタ 1、フィルタ 3、フィルタ 2 の順番でサーブレットを呼び出します。

```
<web-app>
...
<filter>
 //フィルタ 1 の定義
</filter>

<filter>
 //フィルタ 2 の定義
</filter>

<filter>
 //フィルタ 3 の定義
</filter>

<filter-mapping>
 //フィルタ 1 のマッピング
 <url-mapping>...</url-mapping>
</filter-mapping>

<filter-mapping>
 //フィルタ 2 のマッピング
 <servlet-name>...</servlet-name>
</filter-mapping>

<filter-mapping>
 //フィルタ 3 のマッピング
 <url-mapping>...</url-mapping>
</filter-mapping>
...
</web-app>
```

初期化パラメータへのアクセス

フィルタは、Web アプリケーションの web.xml ファイルから初期化パラメータを抽出できます。これにより、Web アプリケーションを再コンパイルせずに、処理中に設定値を変更できます。たとえば、データソースまたはログファイルのロケーションは、ハードコードではなく初期化パラメータで設定することができます。

フィルタ定義内で初期化パラメータを使用しても、サーブレットの場合と同様に動作します。`FilterConfig` クラスは、`ServletConfig` クラスと同じ名前を共有するメソッドを使用してこれらのパラメータにアクセスします。

```
getInitParamter()
getInitParameterNames()
```

web.xml ファイル内で初期化パラメータの名前と値を定義するには、`init-param` 要素を使用します。シンタックスは次のとおりです。

```
...
<filter>
 <filter-name>filter_name</filter-name>
 <filter-class>filter_class</filter-class>
 <init-param>
  <param-name>parameter_name</param-name>
  <param-value>parameter_value</param-value>
 </init-param>
</filter>
...
```

フィルタごとに任意の数の初期化パラメータを指定できます。たとえば、次のように指定できます。

```
<filter>
 <filter-name>InitParamsFilter</filter-name>
 <filter-class>jrunsamples.filters.InitParamsFilter</filter-class>
 <init-param>
  <param-name>message</param-name>
  <param-value>Drink your Ovaltine.</param-value>
 </init-param>
 <init-param>
  <param-name>answer</param-name>
  <param-value>42</param-value>
 </init-param>
</filter>
```

フィルタの doFilter メソッドの次のコードサンプルは、初期化パラメータのリストを取得し、標準出力でリストをプリントします。

```
...
Enumeration initParams = filterConfig.getInitParameterNames();
if (initParams == null) {
 System.out.println(" フィルタ定義内に初期化パラメータがありません ");
 chain.doFilter(request, response);
} else {
 while (initParams.hasMoreElements()) {
 String name = (String) initParams.nextElement();
 String value = filterConfig.getInitParameter(name);
 System.out.println(name + ":"+ value);
  }
}
...
```

簡単なフィルタサンプル

簡単なフィルタサンプルとしては TimingFilter があります。このフィルタは、JRun がリクエストの処理を開始する時刻をミリ秒単位で表示します。リクエストがチェーンから戻ると、JRun は再び時刻を表示します。

このフィルタは、doInit および doDestroy メソッドがオーバーライドされないように、GenericFilter インターフェイスを拡張します。

```
package jrunsamples.filters;

import javax.servlet.*;
import javax.servlet.http.*;

public class TimingFilter extends GenericFilter {
 public void doFilter(ServletRequest request, ServletResponse
        response, FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {
  long bef = System.currentTimeMillis();
  System.out.println(" フィルタの開始：" + bef);
  chain.doFilter(request,response);
  long aft = System.currentTimeMillis();
  System.out.println(" フィルタの終了：" + aft);
 }
}
```

Web アプリケーションの web.xml デプロイメントディスクリプタで、フィルタおよびマッピングは次のように定義されています。

```
<filter>
 <filter-name>TimingFilter</filter-name>
 <filter-class>jrunsamples.filters.TimingFilter</filter-class>
</filter>

<filter-mapping>
 <filter-name>TimingFilter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

このアプリケーションコンテキスト内でリソースを呼び出すと、次のようなメッセージがシステムコンソールにプリントされます。

```
11/20 14:42:47 info JSPServlet：init
フィルタの開始： 1006285367836
フィルタの終了： 1006285367956
```

ラッパーの使用

フィルタは、ServletContext を使用してリクエストおよびレスポンスオブジェクトのヘッダーを表示できます。また、フィルタは、このオブジェクトの属性にアクセスすることもできます。ただし、リクエストまたはレスポンスの実際のコンテンツを変更するには、フィルタは、リクエストまたはレスポンスのコンテンツストリームをクライアントに戻される前に阻止できなければなりません。これは、次のラッパークラスを使用して実行します。

```
javax.servlet.http.ServletRequestWrapper(ServletRequest req)
javax.servlet.http.HttpServletRequestWrapper(HttpServletRequest req)
javax.servlet.http.ServletResponseWrapper(ServletResponse resp)
javax.servlet.http.HttpServletResponseWrapper(HttpServletResponse resp)
```

デフォルトでは、これらのラッパークラスは、すべてのメソッド呼び出しを、メソッドの取り出し元であるクラスに変更せずに渡します。ラッパーを使用すると、ラップされたオブジェクトが公開するすべてのメソッドをオーバーライドすることができます。

デフォルトでは、`HttpServletRequestWrapper` クラスは、すべてのメソッド呼び出しを、このクラスがラップしている HttpServletRequest オブジェクトに渡します。リクエストメソッド (`isUserInRole` や `getRemoteUser` など) をオーバーライドするには、ラッパーを作成し、それらのメソッドをラッパー内に実装し、ラッパー上でこれらのオーバーライドするメソッドを呼び出します。

デフォルトでは、`HttpServletResponseWrapper` クラスは、このクラスがラップしている HttpServletResponse オブジェクトにすべてのメソッド呼び出しを渡します。レスポンスメソッド (`getWriter` や `getLocale` など) をオーバーライドするには、ラッパー上のこれらのオーバーライドされたメソッドを呼び出します。

リクエストやレスポンスがラップされているかどうかにかかわらず、チェーン内の他のフィルタは、このオブジェクトを、他のリクエストまたはレスポンスと同様に処理します。チェーン内の他のフィルタは、オブジェクトがラップされているかどうかを判断することはできません。

オブジェクトがラッパーであるかどうかを判断するには、`instanceOf` 演算子を使用してください。

リクエストの処理

フィルタでは、リクエストオブジェクトのヘッダーまたは ServletContext を直接処理できます。`ServletRequestWrapper` クラスまたは `HttpServletRequestWrapper` クラスを拡張するオブジェクト内にリクエストをラップすることによって、リクエストメソッドをオーバーライドすることもできます。ラッパークラスの使用方法の詳細については、190 ページの「ラッパーの使用」を参照してください。

このセクションでは、リクエストの一般的な処理方法について説明します。

リクエストヘッダーの処理

リクエストオブジェクトには、フィルタがさまざまな方法で使用できるヘッダー情報が含まれています。これらのヘッダーはクライアントブラウザによって生成されます。リクエストヘッダーを調べることによって、フィルタは次のタスクを実行できます。

- トラブルシューティング情報の提供
- 基本的な認証スキームの実行
- 詳細なログの生成
- フォームベース入力の検証
- セッションデータの操作
- エンコードおよび MIME タイプの変更

フィルタは通常、次のリクエストヘッダーを使用します。

- Accept-Encoding
- Accept-Language
- Authorization
- Host
- User-Agent

ヘッダー情報のロギング

フィルタの一般的なタスクは HTTP リクエストおよびレスポンスヘッダーをログファイルや標準出力に出力することです。この機能はリクエストおよびレスポンスのデバッグおよび解析に役立ちます。

次のフィルタコードサンプルでいくつかの方法を紹介します。

- `FilterConfig` インターフェイスを使用して ServletContext を取得します。
- `getHeader` および `getHeaderNames` メソッドにアクセスするために、ServletRequest および ServletResponse オブジェクトを、HttpServletRequest および HttpServletResponse オブジェクトとしてキャストします。そのため、フィルタは、`javax.servlet.*`. だけでなく、`javax.servlet.http.*` もインポートする必要があります。

- Enumeration オブジェクトを使用してヘッダーを取得します。そのため、java.util.* をインポートする必要があります。

```
package jrunsamples.filters;

import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class HeaderFilter extends GenericFilter {

 public void doFilter(ServletRequest req, ServletResponse resp,
        FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {
  ServletContext context = filterConfig.getServletContext();

  chain.doFilter(req, resp);

  HttpServletRequest request = (HttpServletRequest)req;

  context.log("****** リクエストヘッダー ******");
  context.log(" リクエスト :" + request.getRequestURI());
  Enumeration headers = request.getHeaderNames();
  if (headers == null) {
  } else {
   while (headers.hasMoreElements()) {
    String name = (String) headers.nextElement();
    String value = request.getHeader(name);
    context.log(name + "=" + value);
   }
  }

  HttpServletResponse response = (HttpServletResponse)resp;

  context.log("****** レスポンス情報 ******");
  String charencode = response.getCharacterEncoding();
  int bufsize = response.getBufferSize();
  context.log(" 文字エンコード：" + charencode);
  context.log(" バッファーサイズ：" + bufsize);
 }
}
```

Web アプリケーションの web.xml デプロイメントディスクリプタで、フィルタを定義し、そのマッピングを追加します。次に例を示します。

```
<filter>
 <filter-name>HeaderFilter</filter-name>
 <filter-class>jrunsamples.filters.HeaderFilter</filter-class>
</filter>

<filter-mapping>
 <filter-name>HeaderFilter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

Web アプリケーション内のリソースをリクエストする際に、このフィルタは、次のような出力を生成します。

```
11/21 08:04:34 info JSPServlet:init
11/21 08:04:34 info ****** リクエストヘッダー ******
11/21 08:04:34 info リクエスト :/welcome.jsp
11/21 08:04:34 info Accept=image/gif, image/x-xbitmap, image/jpeg,
        image/pjpeg,application/vnd.ms-powerpoint, application/
        vnd.ms-excel, application/msword, */*
11/21 08:04:34 info Cookie=JSESSIONID=1887861006284883609
11/21 08:04:34 info Connection=Keep-Alive
11/21 08:04:34 info Host=localhost:8100
11/21 08:04:34 info Accept-Encoding=gzip, deflate
11/21 08:04:34 info User-Agent=Mozilla/4.0 (compatible; MSIE 5.5;
        Windows NT 5.0)
11/21 08:04:34 info Accept-Language=en-us
11/21 08:04:34 info ****** レスポンス情報 ******
11/21 08:04:34 info 文字エンコード :ISO-8859-1
11/21 08:04:34 info バッファサイズ : 8192
```

リクエストパラメータに基づいたリクエストの転送

フィルタはリクエストパラメータにアクセスできます。リクエストパラメータは通常、<FORM> ブロック内のリクエスト側ページによって設定されるか、セッションオブジェクト内の値から取り出されます。リクエストパラメータは、リクエストクエリ文字列 (GET リクエストなど) に含めたり、リクエスト本体 (POST リクエストなど) に含めることができます。

フィルタは通常、チェーン内の次のフィルタにリクエストをチャレンジなしで渡します。ただし、これは必須動作ではありません。リクエストを処理する際、フィルタは、リクエストを他のリソースに転送したり、独自のレスポンスを生成したりすることができます。この機能は、セキュリティチェックによるアクセスの阻止、ユーザーの転送、ブラウザタイプに基づいてリクエストの転送を行う場合に便利です。

フィルタチェーンを中断するには、RequestDispatcher を使用し、リクエストを転送することによって行います。サーブレットまたはフィルタの doFilter メソッド内でリクエスト転送を呼び出した後、チェーン内で、そのリクエストの前または後にフィルタが処理されることはありません。これは、既にリクエストを前処理したフィルタが対象となります。

次のコードサンプルは、リクエストパラメータ "username" が "nick" でない場合にリクエストを転送します。

```
...
public void doFilter (ServletRequest request, ServletResponse
        response, FilterChain chain) throws IOException,
        ServletException {
 if (request.getParameter("username").equals("nick")) {
  ...// 通常のフィルタ処理
 } else {
  forwardToErrorPage(request, response);
 }
 chain.doFilter(request, response);
}
public void forwardToErrorPage (ServletRequest req, ServletResponse
        res) {
 System.out.println(" エラーページに転送されました ");
 try {
  RequestDispatcher rd = req.getRequestDispatcher("/Error.jsp");
  rd.forward(req, res);
 } catch (Exception e) {
  System.out.println(e.toString());
 }
}
...
```

User-Agent に基づいたリクエストの転送

リクエスト転送のその他の例としては、リクエスト側のブラウザのタイプに基づいた方法があります。リクエストオブジェクトの User-Agent ヘッダーを調べて、ブラウザのタイプを検出し、非互換ブラウザを転送することができます。ヘッダーに直接アクセスするには、ServletRequest オブジェクトを HttpServletRequest としてキャストする必要があります。たとえば、次のように記述します。

```
...
public void doFilter(ServletRequest req, ServletResponse resp,
        FilterChain chain) throws IOException, ServletException {
 HttpServletRequest request = (HttpServletRequest)req;
 String agent = (String) request.getHeader("User-Agent");
 System.out.println("agent=" + agent);
 if (agent.startsWith("Mozilla") || agent.startsWith("Microsoft")) {
  chain.doFilter(req, resp);
 } else {
  try {
   RequestDispatcher rd = request.getRequestDispatcher("/Non-Com/");
   rd.forward(req, resp);
  } catch (Exception e) {
   System.out.println(e.toString());
  }
 }
 chain.doFilter(req,resp);
}
...
```

フィルタを使用した基本的な認証の指定

認証は、ユーザーの資格 (ユーザ名とパスワード) を収集し、それらをシステム内で検証する処理です。認証では通常、資格を、データベースまたは LDAP サーバーなどの一部のユーザーレポジトリと照合し、ユーザーの ID を確認する必要があります。リクエストヘッダーを評価することによって、フィルタを使用して、Web アプリケーションリソースに基本的なホストベース認証を提供できます。

次のフィルタサンプルは、受信リクエストの IP アドレスを検証し、ローカルホスト (127 で始まる IP アドレス) からのリクエストだけがチェーンを通過できるようにして実現するホストベース認証です。フィルタは RequestDispatcher を使用して、非認証 IP アドレスからのリクエストをエラーページに転送します。

リクエストヘッダーはなりすまされる可能性があるため、この方法は本格的な認証方法ではありません。しかし、セキュリティ戦略全体の一部を担うものとして有効です。

```
package jrunsamples.filters;

import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HostAuthFilter extends GenericFilter {

 public void doFilter(ServletRequest request, ServletResponse
        response, FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {

  String remoteaddr = request.getRemoteAddr();
  System.out.println("remoteaddr = " + remoteaddr);

  if (remoteaddr.startsWith("127.")) {
   System.out.println(" ホストからのリクエストは認証されました ");
   chain.doFilter(request, response);
  } else {
   RequestDispatcher rd = request.getRequestDispatcher("/Error.jsp");
   System.out.println(" ホストからのリクエストは認証されませんでした ");
   rd.forward(request, response);
  }

 }
}
```

Web アプリケーションの web.xml デプロイメントディスクリプタで、フィルタを定義し、そのマッピングを追加します。次に例を示します。

```
<filter>
 <filter-name>HostAuthFilter</filter-name>
 <filter-class>jrunsamples.filters.HostAuthFilter</filter-class>
</filter>

<filter-mapping>
 <filter-name>HostAuthFilter</filter-name>
 <url-pattern>/startpage.html</url-pattern>
</filter-mapping>
```

ローカルホストから Web アプリケーション内の startpage.html をリクエストすると、このフィルタは、次のようなメッセージを出力します。

```
remoteaddr = 127.0.0.1
ホストからのリクエストは認証されました
```

認証条件と一致しない別のホスト（この場合は無関係のホスト）からリソースをリクエストすると、このフィルタは、リクエストを Error.jsp に転送し、次のようなメッセージを出力します。

```
remoteaddr = 10.1.116.192
ホストからのリクエストは認証されませんでした
```

フィルタを使用した基本的な承認の指定

承認は、アクセス制御リストなどを使用することによって、ユーザーが所定のリソースにアクセスできるようにする処理です。サーブレットエンジンは承認を使用する前にユーザーを認証します。リソース表示の承認がユーザーに与えられていない場合、サーブレットコンテナ（エンジン）はアクセスを許可しません。

フィルタは、リクエストヘッダーへのアクセスを利用して、リクエストがターゲットリソースに送信される前にユーザーに承認が与えられているかどうかを調べます。フィルタを、Web コンテナセキュリティの他のメソッドと組み合わせることによって、基本的なセキュリティシステムを作成できます。

次の手順は、Authorization（承認）ヘッダーを調べるフィルタの設定方法です。ヘッダーが設定されると、ユーザーはログインしてリクエストを続行することができます。ユーザーがログインできない (Authorization リクエストヘッダーが設定されていない）と、フィルタはリクエストをログインページに転送します。

リクエストヘッダーはなりすまされる可能性があるため、この方法は本格的な承認方法ではありません。しかし、セキュリティ戦略全体の一部を担うものとして有効です。

基本的な承認チェックを目的としたフィルタを設定するには

1　次のフィルタをコンパイルします。

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class AuthFilter extends GenericFilter {
 public void doFilter(ServletRequest request, ServletResponse
      response, FilterChain chain) throws java.io.IOException,
      javax.servlet.ServletException {
```

```
  HttpServletRequest req = (HttpServletRequest)request;
  String authorization = req.getHeader("Authorization");
  if (authorization != null) {
   System.out.println("承認されたアクセス：" + req.getRemoteUser());
   chain.doFilter(request,response);
  } else {
   System.out.println("承認されなかったアクセス：" +
     req.getRemoteUser());
   RequestDispatcher rd = request.getRequestDispatcher("/
     login.html");
   rd.forward(request, response);
  }
 }
}
```

2 フィルタとマッピングを web.xml ファイルに追加します。次のように指定します。

```
<filter>
 <filter-name>AuthFilter</filter-name>
 <filter-class>jrunsamples.filters.AuthFilter</filter-class>
</filter>
...
<filter-mapping>
 <filter-name>AuthFilter</filter-name>
 <url-pattern>protected_resource_mapping</url-pattern>
</filter-mapping>
```

3 新規ユーザーを追加し、そのユーザーに <JRun のルートディレクトリ>/*server_name*/SERVER-INF/jrun-users.xml ファイル内のロールを割り当てます。次に例を示します。

```
<user>
 <user-name>nick</user-name>
 <password>danger</password>
</user>
<role>
 <role-name>manager</role-name>
 <user-name>nick</user-name>
</role>
```

4 web.xml ファイルで、ターゲットリソースに対するセキュリティ制約を追加します。これによって、ユーザーがそのページをリクエストすると、ユーザーにログインをリクエストします。最も簡単なサンプルでは BASIC 認証を使用します。この認証では、クライアントのブラウザが、ユーザー名とパスワードを入力するようにユーザーに要求します。クライアントの入力が認証されると、以後のリクエストの Authorization ヘッダーが設定されます。

5 JRun サーバーを再起動します。

6 web.xml ファイル内にセキュリティ設定値を追加します。たとえば、次のように指定します。

```
<login-config>
  <auth-method>BASIC</auth-method>
 </login-config>
 <security-constraint>
  <web-resource-collection>
   <web-resource-name>Protected Page</web-resource-name>
   <url-pattern>/login.html</url-pattern>
   <http-method>GET</http-method>
   <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
   <role-name>manager</role-name>
  </auth-constraint>
 </security-constraint>
```

リクエストの変更

フィルタには、リクエストデータを検証して処理中に変更する機能があります。この機能は、フォーム入力を検証したり、リクエスト側クライアントがアクセスできない内部設定を行ったりする場合に便利です。

フィルタ内でリクエストオブジェクトを変更したり、リクエストがターゲットリソースに送られる前にユーザーのステートを変更したりするには、さまざまな方法があります。これらの方法は次のものによって実装されます。

- `request.setAttribute` メソッド
- セッションオブジェクト
- リクエストラッパー

次のセクションでは、これらのメソッドの一部のサンプルについて説明します。

リクエストを変更するためのリクエスト属性の設定

`request.setAttribute` メソッドを使用すると、フィルタがターゲットリソースにリクエストを渡す前に、リクエストの属性の追加または変更を行うことができます。

また、String オブジェクトだけでなく、他のオブジェクトも属性として設定することができます。フィルタの次のコード抜粋では、リクエストの 2 つの属性、すなわち String オブジェクトと Integer オブジェクトを設定します。

```
...
String sVeggies = "ブロッコリーとほうれんそう";
Integer x = new Integer(6);
//chain.doFilter() の前に呼び出す必要があります。
request.setAttribute("vegetables",sVeggies);
request.setAttribute("some_number",x);
chain.doFilter(request, response);
...
```

JSP 内の次の式は、これらの属性の値を取得します。`getAttribute` は汎用オブジェクトを返します。したがって、String のみを返す `request.getParameter` メソッドとは異なり、結果を希望のオブジェクトタイプにキャストする必要があります。

次の場合、vegetables 属性の値は String にキャストされ、x は Integer にキャストされます。

```
<%= (String)request.getAttribute("vegetables") %>
<%
 Integer x = (Integer)request.getAttribute("some_number");
 out.println(x.intValue() * 30);
%>
```

セッションオブジェクトを使用して複数のリクエストにわたる変数を渡す方法をお勧めします。コントローラサーブレットまたは RequestDispatcher でリクエストを処理しない場合は、リクエストオブジェクトに対して setAttribute を使用する方法は使用しないでください。

セッションの操作

セッションオブジェクトはリクエストオブジェクトよりも大きなスコープを持っています。リクエスト属性は、現在リクエストされているリソースでのみ使用できます。セッション属性は複数のリクエストにわたって継続します。フィルタを使用すると、セッションを作成し、セッションオブジェクト内の値を設定し、クライアントがリクエストするすべてのページからこれらの値にアクセスすることができます。

ターゲットリソースではなく、フィルタ内でセッションオブジェクトを作成し、これにデータを挿入する場合は、Web コンポーネントの作業を分離しておくと便利です。これにより、コードを再利用しやすくなります。これは、Front Controller などのデザインパターン戦略でしばしば使用される手法です。

次のフィルタはセッションオブジェクトを作成し (まだセッションオブジェクトが存在しない場合)、リクエストパラメータの値を名前として保存し、この名前をセッション属性として設定し、チェーン経由でリクエストをターゲットリソースに渡すことによってリクエストの処理を終了します。

```
public class SessionFilter extends GenericFilter {
 public void doFilter(ServletRequest request, ServletResponse
        response, FilterChain chain) throws java.io.IOException,
        javax.servlet.ServletException {

  // セッションオブジェクトにアクセスできるように、必ず、ServletRequest を
        HttpServletRequest にキャストしてください。
  HttpServletRequest req = (HttpServletRequest)request;

  // 新規セッションを作成します。
  HttpSession session = req.getSession(true);
  String name = request.getParameter("name");
  session.setAttribute("name", name);

  // 制御をターゲットリソースに渡します。
  chain.doFilter(req, response);
 }
}
```

文字エンコードの設定

複数の文字セットおよびロケールをサポートしている多言語 Web アプリケーションでは、リクエストの文字エンコードを設定する必要があります。ただし、リクエストパラメータは、コンテナのデフォルトの文字エンコードを使用して解析されることがあります。解析エラーを防ぐには、フィルタを使用してリクエストの文字エンコードを設定してください。

たとえば次のように、リクエストを渡す前に、`setCharacterEncoding` メソッドを呼び出す必要があります。

```
...
public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
 request.setCharacterEncoding("ISO-8859-1");
 chain.doFilter(request, response);
}
...
```

ロケール対応 Web アプリケーションの操作の詳細については、31 ページの第 3 章「国際化対応とローカリゼーション」を参照してください。

レスポンスの処理

ターゲットリソースにアクセスした後に、レスポンスヘッダーを追加または更新することによってレスポンスオブジェクトを変更できます。また、ServletResponseWrapper または HttpServletResponseWrapper のいずれかを継承したオブジェクト内にレスポンスをラップすることによって、リクエストメソッド（request.getWriter メソッドなど ）をオーバーライドすることもできます。

レスポンス内のデータの解析

レスポンスをカスタマイズするには、リクエストラッパークラスを使用してください。ラッパークラスの詳細については、190 ページの「ラッパーの使用」を参照してください。このセクションでは、ラッパークラスを使用してレスポンスオブジェクトをカスタマイズするフィルタのサンプルを紹介します。

次の例は、レスポンス出力を読み取り、特定の文字列を置き換えます。このサンプルは、"Macromedia" または "Yahoo" という単語が発生するたびに株式表示記号および株式情報のハイパーリンクを付加します。

以下のアクティビティを次のサンプルフィルタコードで記述します。

- ServletResponse を HttpServletResponse としてキャストし、CharWrapper 内にレスポンスオブジェクトをラップします。
- キーワード "Macromedia" または "Yahoo" を分離するように設定されているデフォルトの区切り記号を使用して、ラップされたレスポンスをトークンにします。
- レスポンスがクライアントに戻される前にフィルタが変更できるように、CharArrayWriter へのレスポンスを阻止します。
- レスポンスのコンテンツの長さを再設定します。

chain.doFilter 呼び出しで、フィルタは、レスポンスオブジェクトではなく新規ラッパーを渡します。

```
package jrunsamples.filters;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public final class ParseFilter extends GenericFilter {

 public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

  ServletContext context = filterConfig.getServletContext();
  PrintWriter out = response.getWriter();

  HttpServletResponse resp = (HttpServletResponse)response;
  CharWrapper wrapper = new CharWrapper(resp);

  chain.doFilter(request, wrapper);

  String temp = wrapper.toString();

  StringTokenizer st = new StringTokenizer(temp," ¥t¥n¥r");

  while (st.hasMoreTokens()) {
   String word = (String) st.nextToken();
   if (word.equals("Macromedia")) {
```

```
    word = word + " (<A HREF=¥"http://finance.yahoo.com/
        q?s=macr¥">MACR</A>)";
   } else if (word.equals("Yahoo")) {
    word = word + " (<A HREF=¥"http://finance.yahoo.com/
        q?s=yhoo¥">YHOO</A>)";
   }
   out.write(" " + word + " ");
  }
  response.setContentLength(temp.length());
  out.close();
 }
}
```

クラス CharWrapper は、HttpServletResponseWrapper を拡張します。このクラスは、次のアクティビティを示しています。

- CharArrayWriter を使用して、getWriter メソッドをコンストラクタでオーバーライドします。オーバーライドできるのは、getWriter ではなく getOutputStream です。両方をオーバーライドすることはできません。
- toString メソッドをオーバーライドして、toString 呼び出しを阻止し、CharArrayWriter を使用してそれらを出力します。

```
// Sun の "The Essentials of Filters" の CharResponseWrapper クラスに基づく
package jrunsamples.filters;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CharWrapper extends HttpServletResponseWrapper {
 private CharArrayWriter cawout;

 public CharWrapper(HttpServletResponse response){
  super(response);
  cawout = new CharArrayWriter();
 }

 public String toString() {
  return cawout.toString();
 }

 public PrintWriter getWriter(){
  return new PrintWriter(cawout);
 }
}
```

Web アプリケーションの web.xml デプロイメントディスクリプタで、フィルタを定義し、そのマッピングを追加します。次に例を示します。

```
<filter>
 <filter-name>ParseFilter</filter-name>
 <filter-class>jrunsamples.filters.ParseFilter</filter-class>
</filter>

<filter-mapping>
 <filter-name>ParseFilter</filter-name>
 <url-pattern>/macromedia.html</url-pattern>
</filter-mapping>
```

サーブレットの出力を変更するフィルタの関連サンプルについては、204 ページの「リソース」にリストしたリンクを参照してください。

リソース

Web には、この章では網羅しきれなかった高度なフィルタリングテクニックの優れたサンプルなど、フィルタを習得するための情報リソースが豊富にあります。フィルタの詳細については、次のリソースを参照してください。

リソース	アドレス
Java Developer Connection:Technical Tips	http://developer.java.sun.com/developer/JDCTechTips/2001/tt0626.html#tip2
Sun:The Essentials of Filters	http://java.sun.com/products/servlet/Filters.pdf
JavaWorld:Jason Hunter, "Filter Code with the Servlet 2.3 Model"	http://www.javaworld.com/javaworld/jw-06-2001/jw-0622-filters.html
OpenSymphony:SiteMesh	http://www.opensymphony.com/sitemesh
O'Reilly OnJava:Stephanie Fesler, "Writing Servlet 2.3 Filters"	http://www.onjava.com/pub/a/onjava/2001/05/10/servlet_filters.html?page=1
DevwebPro:James McGovern, "Servlet Filters 101"	Part 1:http://www.devwebpro.com/2001/0906.html Part 2:http://www.devwebpro.com/2001/0907.html Part 3:http://www.devwebpro.com/2001/0914.html
O'Reilly OnJava:Satya Kontineni, "Learning Servlet Filters"	http://www.onjava.com/pub/a/onjava/2001/08/28/filters.html

第 8 章
アプリケーションのライフサイクルイベント

Java Servlet API 2.3 仕様の主な新機能の 1 つに、サーブレットイベントリスナの追加があります。この章では、Web アプリケーションでのイベントリスナの使用方法について説明します。

目 次

イベントリスナの概要

Servlet 2.3 仕様は、Web アプリケーション用のリスナクラスについて規定しています。イベントリスナはイベントハンドラとも呼ばれ、特定のイベントの発生時に JRun が呼び出すコールバックメソッドが用意されています。イベントリスナを使用すると、Web アプリケーションのさまざまな要素を管理し、ServletContext および HttpSession オブジェクト内のステートの変化を追跡できます。

イベントの発生時にエンジンがリスナ内の適切なメソッドを排除しないように、イベントリスナをサーブレットエンジンに登録する必要があります。この作業は、Web アプリケーションの web.xml デプロイメントディスクリプタ内で行います。

ServletContext および HttpSession オブジェクトのアクティビティを監視するサーブレットイベントリスナには 2 つの基本タイプがあります。これらのタイプにはそれぞれ、リスナインターフェイスと属性リスナインターフェイスがあります。

サーブレットオブジェクト	関連付けられているリスナインターフェイス
ServletContext	`ServletContextListener` `ServletContextAttributeListener`
HttpSession	`HttpSessionListener` `HttpSessionAttributeListener`

基本リスナインターフェイスには、オブジェクトの変更をリスンするメソッドが用意されています。属性リスナインターフェイスメソッドは、新規属性の追加、属性の値の変更、またはそれらのオブジェクトに関連付けられている属性の削除をリスンします。

イベントの例は次のとおりです。

- 新規 HttpSession オブジェクトの作成
- HttpSession オブジェクトの廃棄
- 新規 ServletContext オブジェクトの作成
- ServletContext オブジェクトの廃棄
- HttpSession および ServletContext オブジェクトの属性の追加、削除、および変更

イベントリスナは、次のような多数の一般的なタスクに使用します。

- ロギング
- セッションの管理
- アプリケーションサーバーリソースの追跡

サーブレットイベントリスナの詳細については、Servlet 2.3 仕様の「Application Lifecycle Events」の章を参照してください。

イベントリスナの作成

イベントリスナは java.util.EventListener インターフェイスを拡張します。イベントリスナを作成するには、リスンするインターフェイスの 1 つを実装する必要があります。各リスナには、引数なし public コンストラクタも必要です。コンストラクタを指定しないと、コンパイラは引数なし public コンストラクタを使用します。イベントリスナをコンパイルするとき、クラスパスに <JRun のルートディレクトリ >/lib/jrun.jar を指定する必要があります。

WEB-INF/classes または WEB-INF/lib ディレクトリの Web アプリケーションがリスナクラスにアクセスできるようにする必要があります。JAR または WEB-INF/classes ディレクトリ構造のサブディレクトリまたはパッケージにリスナクラスを含めることができます。

イベントリスナの定義

イベントリスナは、Web アプリケーションの web.xml デプロイメントディスクリプタ内で定義する必要があります。このシンタックスは次のとおりです。

```
<listener>
 <listener-class>listener_class</listener-class>
</listener>
```

次の例では、SessionWatcher フィルタを定義します。

```
<listener>
 <listener-class>jrunsamples.events.SessionWatcher</listener-class>
</listener>
```

サーブレットの定義がある場合は、その前にリスナの定義を指定する必要があります。リスナは、その前に定義されたサーブレットに対しては呼び出されません。JRun は起動時にリスナを登録し、シャットダウン時に登録を解除します。

該当するイベントが発生すると、JRun は web.xml ファイルに指定されている順にイベントリスナを呼び出します。

ServletContext イベントのリスン

ServletContext には、Web アプリケーションですべてのサーブレットが共有するリソースにユーザーがアクセスできるようにする API があります。ServletContext オブジェクトには次の 2 つのイベントリスナインターフェイスがあります。

- `ServletContextListener`
- `ServletContextAttributeListener`

JRun は ServletContextEvent オブジェクトを ServletContext に渡し、ServletContextAttributeEvent オブジェクトを ServletContextAttribute イベントに渡します。JRun は、イベントオブジェクトを介して ServletContext および ServletContext の属性にアクセスできます。

ServletContext イベントが発生すると、リスナインターフェイスが実装しているメソッドをトリガーできます。次の表に、`ServletContext` インターフェイスのメソッドと、JRun にこれらのメソッドを呼び出させるイベントを示します。

メソッド	このメソッドに通知するイベント
`ServletContextListener` インターフェイス	
`contextInitialized`	JRun が ServletContext を作成する JRun サーバーを起動する JRun サーバーを再起動する
`contextDestroyed`	JRun が ServletContext を廃棄する JRun サーバーを停止する JRun サーバーを再起動する
`ServletContextAttributeListener` インターフェイス	
`attributeAdded`	ServletContext オブジェクトに属性を追加する JRun サーバーを起動する JRun サーバーを再起動する `context.setAttribute` を呼び出すことによって属性を追加する
`attributeReplaced`	ServletContext オブジェクトの属性の値を変更する `context.setAttribute` を呼び出すことによって属性を変更する
`attributeRemoved`	ServletContext 属性を削除する JRun サーバーを停止または再起動する `context.removeAttribute` を呼び出すことによって属性を削除する

リスナを使用して、データベース接続のロギングや作成などのタスクを実行する ServletContext リスナイベントメソッドに開始または終了コードを入れることができます。

ServletContextListener について

JRun は、Web アプリケーションがリクエストを受け取る準備が整い、新規 ServletContext が作成されたとき、および Web アプリケーションがコンテナから削除され、ServletContext が廃棄されたときに ServletContextListener に通知します。これらのイベントは通常、JRun サーバーの起動、停止、および再起動時に発生します。

ServletContextListener サーブレットには次の 2 つのメソッドがあります。

- `contextInitialized(ServletContextEvent event)`
- `contextDestroyed(ServletContextEvent event)`

ServletContextEvent オブジェクトには次の 1 つのメソッドがあります。
`getServletContext()`

ServletContextAttributeListener について

ServletContext の属性には、Web アプリケーション内のすべてのサーブレットからアクセス可能です。ServletContext の属性は、RequestDispatcher を使用して頻繁に関連付けられます。

JRun は、ServletContext の属性が追加または削除されるか、または属性の値が変更されると ServletContextAttributeListener に通知します。

このリスナは 3 つのメソッドから構成されています。

- `attributeAdded(ServletContextAttributeEvent event)`
- `attributeReplaced(ServletContextAttributeEvent event)`
- `attributeRemoved(ServletContextAttributeEvent event)`

ServletContextAttributeEvent オブジェクトは ServletContextEvent オブジェクトから `getServletContext` メソッドを継承します。ServletContext の属性にアクセスできるメソッドが他に 2 つあります。

- `getName()`
- `getValue()`

ServletContextListener のロギング例

次の例は、ServletContextEvent オブジェクトで使用可能な唯一のメソッド `getServletContext` の使用方法を示しています。このメソッドを使用すると、ServletContext オブジェクトを取得し、ServletContext のすべてのメソッドにアクセスできます。

この単純な ServletContext イベントリスナは `ServletContextListener` インターフェイスのメソッドを実装します。これらのいずれかのイベントが発生すると、ログファイルに行が書き込まれます。

```
public final class ContextLoggingListener implements
        ServletContextListener {
 public void contextInitialized(ServletContextEvent event) {
  event.getServletContext().log("ServletContext initialized");
 }
 public void contextDestroyed(ServletContextEvent event) {
  event.getServletContext().log("ServletContext destroyed");
 }
}
```

このイベントリスナは web.xml ファイルで定義されます。たとえば、次のコードは ContextLoggingListener を定義します。

```
<listener>
 <listener-class>jrunsamples.events.ContextLoggingListener</
        listener-class>
</listener>
```

ServletContextListener へのファイルアクセス例

この例では、ServletContext オブジェクトから初期化パラメータを取得し、ServletContext に別の属性を設定します。

```
...
public final class ContextLoggingListener implements
        ServletContextListener {
 public void contextInitialized(ServletContextEvent event) {
  private static final char c = File.separatorChar;
  try {
   System.out.println("ServletContext initialized");
   File baseDir =
        (File) event.getServletContext().getInitParameter("baseDir");
   String fullpath = baseDir.getAbsolutePath() + c + "temp.txt";
   event.getServletContext().setAttribute("fullpath", fullpath);
   File f = new File(fullpath);
   if (!f.exists()) {
    ... // ファイル操作ロジック
   } else {
    ... //
   }
  catch (Exception e) {
   e.printStackTrace();
  }
 }
...
```

ServletContextAttributeListener の例

次の例では、ServletContextAttributeListener のメソッドを実装します。これらのいずれかのイベントが発生すると、標準出力に行が出力されます。この例では、ServletContextAttributeEvent オブジェクトに用意されている便利な getValue メソッドの使用方法も示しています。

```
public final class ContextLoggingAttributeListener implements
        ServletContextAttributeListener {
 public void attributeAdded(ServletContextAttributeEvent event) {
  System.out.println("ServletContext attribute added");
  Enumeration enum = event.getServletContext().getAttributeNames();
  while (enum.hasMoreElements()) {
  String attrname = (String)enum.nextElement();
  System.out.println(attrname + "=" + event.getValue(attrname));
 }
 public void attributeReplaced(ServletContextAttributeEvent event) {
  System.out.println("ServletContext attribute replaced");
 }
 public void attributeRemoved(ServletContextAttributeEvent event) {
  System.out.println("ServletContext attribute removed");
 }
}
```

HttpSession イベントのリスン

HttpSession オブジェクトには、ステートを簡単かつ強力に管理できるようにする API が用意されています。HttpSession オブジェクトにはイベントリスナが 3 つあります。

- `HttpSessionListener`
- `HttpSessionAttributeListener`
- `HttpSessionActivationListener`

ServletContext リスナの場合と同様に、JRun はイベントオブジェクトをセッションリスナ内の各メソッドに渡します。リスナはこれらのオブジェクトを介して Session オブジェクト、Session オブジェクトの属性、および ServletContext オブジェクトにアクセスできます。

HttpSession イベントが発生すると、これらのインターフェイスが実装しているメソッドをトリガーできます。次の表に、これらのインターフェイスのメソッドと、JRun にこれらのメソッドを呼び出させるイベントを示します。

メソッド	このメソッドに通知するイベント
HttpSessionListener インターフェイス	
`sessionCreated`	JRun が新規 HttpSession オブジェクトを作成します。 `request.getSession(true)` を呼び出します。
`sessionDestroyed`	HttpSession がタイムアウトになります。 `session.invalidate` を呼び出します。
HttpSessionAttributeListener インターフェイス	
`attributeAdded`	HttpSession オブジェクトに属性を追加します。 `session.setAttribute` を呼び出して属性を追加します。
`attributeReplaced`	HttpSession オブジェクトの属性の値を変更します。 属性に対して `session.setAttribute` を呼び出します。
`attributeRemoved`	HttpSession 属性を削除します。 JRun サーバーを停止または再起動します。 `session.removeAttribute` を呼び出します。
HttpSessionActivationListener インターフェイス	
`sessionWillPassivate`	JRun が HttpSession オブジェクトをパッシベートします。
`sessionDidActivate`	JRun が以前にパッシベートされた HttpSession オブジェクトをアクティブにします。

HttpSessionListener について

HttpSessionListener を使用して、ロギングまたはパーシスタンスコードをアプリケーションに入れることができます。このリスナは、アプリケーションにログインしているユーザーの数を追跡する際にも役立ちます。この例では、セッションが作成されるたびにカウンタを 1 つ増やし、セッションが廃棄されるたびに 1 つ減らします。

次の行のように、セッションを作成するリソースがリクエストされると、JRun はセッションを作成します。

```
HttpSession session = request.getSession(true);
```

次の行のように、セッションがタイムアウトになるか、明示的に廃棄されると、JRun はセッションを廃棄します。

```
session.invalidate();
```

クライアントからのリクエストなしに MaxInactiveInterval が経過すると、セッションはタイムアウトになります。JRun がタイムアウトになるまでのデフォルト設定は 1800 秒 (30 分) です。セッション設定の詳細については、『JRun 管理者ガイド』を参照してください。

セッションオブジェクトは Cookie オブジェクトとして頻繁にインスタンス化されます。それによって、Cookie オブジェクトとしてセッションでオペレーションを実行できますが、制限もあります。たとえば、Cookie の MaxAge が -1 に設定されている場合は、ブラウザを閉じたときにクライアント上で Cookie が廃棄されます。サーバーはクライアントが閉じたことを検出できないため、`HttpSessionListener.sessionDestroyed` イベントは通知されません。

JRun は、セッションオブジェクトが最初に作成または廃棄されたときに、次のメソッドを使用して HttpSessionListener に通知します。

- `sessionCreated(HttpSessionEvent event)`
- `sessionDestroyed(HttpSessionEvent event)`

HttpSessionListenerEvent オブジェクトには次の 1 つのメソッドがあります。

`getSession()`

HttpSessionAttributeListener について

JRun は、Session オブジェクトの属性が追加または削除されるか、または Session オブジェクトの属性の値が変更されると、次のメソッドを使用して HttpSessionAttributeListener に通知します。

- `attributeAdded(HttpSessionBindingEvent event)`
- `attributeRemoved(HttpSessionBindingEvent event)`
- `attributeReplaced(HttpSessionBindingEvent event)`

Session オブジェクトの属性は、HttpSession オブジェクトのみがアクセス可能です。ServletContext のオブジェクト名に合わせるには、イベントオブジェクト HttpSessionBindingEvent の名前を HttpSession*Attribute*Event とする方が適切です。

HttpSessionBindingEvent オブジェクトは HttpSessionEvent オブジェクトから `getSession` メソッドを継承します。セッション属性にアクセスできるメソッドが他に 2 つあります。

- `getName()`
- `getValue()`

HttpSessionListener のロギング例

単純な HttpSession イベントリスナの例を次に示します。このイベントリスナは HttpSessionListener のメソッドを実装します。これらのいずれかのイベントが発生すると、標準出力に行が出力されます。セッションが作成されると、このリスナはセッション ID と作成日を標準出力に出力します。

```
...
public final class SessionLogger implements HttpSessionListener {
 public void sessionCreated(HttpSessionEvent event) {
  System.out.println("作成されたセッション");
  HttpSession session = event.getSession();
  String sid = session.getId();
  long time = session.getCreationTime();
  System.out.println("新規セッション ID：" + sid);
  System.out.println("作成されたセッション @ " + time);
 }
 public void sessionDestroyed(HttpSessionEvent event) {
  System.out.println("セッション " + event.getSession().getId() + " 廃棄");
 }
}
```

データベースでのセッション情報の保管

HttpSession リスナの用途の 1 つに、セッションを保管および追跡するためのコードの集中化があります。すべてのデータベースコードをリスナに入れ、サーブレットと JSP を保管すると、追跡情報の詳細とは別にセッションオブジェクトを操作できます。

さまざまなイベントの呼び出しを予測できるため、このセットアップは、Sun の『Core J2EE Patterns』に記載されている Intercepting Filter パターン (Decorator Filter パターンとも呼ばれる) に似ています。

サンプルの JRun サーバーでは、次のリスナを実装します。

```
package jrunsamples.events;

import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public final class SessionWatcher implements HttpSessionListener {

 public void sessionCreated(HttpSessionEvent event) {
//Session オブジェクトを介して ServletContext オブジェクトにアクセスすることもできます。
  String datasource_name = (String) event.getSession().
        getServletContext().getAttribute("datasource_name");
  HttpSession session = event.getSession();
  String sid = session.getId();
  long time = session.getCreationTime();
  // セッション ID、作成日、およびセッション変数の値をデータベースに追加します。
  String sqlstmt = "INSERT INTO SESSIONS VALUES(?,?,?)";
  try {
   InitialContext ctx = new InitialContext();
```

```
   DataSource ds = (DataSource) ctx.lookup(datasource_name);
   Connection conn = ds.getConnection();
   PreparedStatement ps = conn.prepareStatement(sqlstmt);
   ps.setString(1,sid);
   ps.setLong(2,time);
   ps.setInt(3, 1); // デフォルト値は 1 に設定されています。
   ResultSet rs = ps.executeQuery();
  } catch (Exception e) {
   e.printStackTrace();
  } finally {
   try {
    ps.close();
    conn.close();
   } catch (Exception e) {
    e.printStackTrace();
   }
  }
 }
}
```

HttpSessionAttributeListener の例

次の単純な HttpSession イベントリスナの例では、HttpSessionAttributeListener のメソッドを実装します。これらのいずれかのイベントが発生すると、標準出力に行が出力されます。属性の値が変更されると、このイベントリスナはセッション属性のすべてを標準出力に出力します。

```
public final class SessionAttrLogger implements
        HttpSessionAttributeListener {
 public void attributeAdded(HttpSessionEvent event) {
  System.out.println("追加されたセッション属性");
 }
 public void attributeRemoved(HttpSessionEvent event) {
  System.out.println("削除されたセッション属性");
 }
 public void attributeReplaced(HttpSessionEvent event) {
  System.out.println("変更されたセッション属性");
  HttpSession session = event.getSession();
  Enumeration enum = session.getAttributeNames();
  while (enum.hasMoreElements()) {
   String attrname = (String)enum.nextElement();
   System.out.println(attrname + "=" +
        session.getAttribute(attrname));
  }
 }
}
```

HttpSessionActivationListener について

JRun は、セッションがアクティブになるかパッシベートされると、HttpSessionActivationListener に通知します。

パッシベートは、JRun がセッションを無効にしないが、リソースを解放するためにセッションをディスクにバックアップした場合や、" ミラーリング " された JRun サーバーに障害が発生し、別の JRun サーバーがそのセッションを引き受けた場合に発生します。これは、セッションパーシスタンスとも呼ばれます。

アクティブ化は、セッションをパッシベートされたクライアントがリクエストし、セッションをディスクから取得するか、新規 JRun サーバーにロードする必要がある場合に発生します。

次の単純な HttpSession イベントリスナの例では、**HttpSessionActivationListener** インターフェイスのメソッドを実装します。これらのいずれかのイベントが発生すると、標準出力に行が出力されます。

```
...
 public void sessionWillPassivate(HttpSessionEvent event) {
  System.out.println("Session was passivated");
 }
 public void sessionDidActivate(HttpSessionEvent event) {
  System.out.println("Session was activated");
 }
...
```

第 9 章
Web アプリケーションの最適化

この章では、Web アプリケーションのレスポンス時間とスループットを改善するためにコード内で使用できる具体的な手法を中心に説明します。さらに、Web アプリケーションのパフォーマンスとスケーラビリティを調整および監視するために JRun に組み込まれているツールについても説明します。

第 2 章の情報とともにこの章の情報を利用して、Web アプリケーションのパフォーマンスと扱いやすさを向上させてください。

目 次

サーブレットの最適化

このセクションでは、Web アプリケーションのサーブレットコンポーネントのコードを改善するための具体的な手法について説明します。Web アプリケーションの要素を変更する予定がある場合は、デザインパターンの使用も検討する必要があります。デザインパターンの詳細については、11 ページの第 2 章「デザインパターン」を参照してください。

init メソッドでのスタティックデータのキャッシュ

可能な場合は、クライアントがリクエストするたびに Web アプリケーションがスタティックデータをダイナミックに生成するのではなく、キャッシュするようにしてください。サーブレットを使用して、`init` メソッドでスタティックデータをキャッシュすることができます。

サーブレットが web.xml ファイルに登録されており、`load-on-startup` 要素が 1 に設定されていると、JRun はサーバーの起動時に `init` メソッドを呼び出します。そうでない場合は、クライアントが最初にサーブレットをリクエストしたときに (最初のみ)、JRun は `init` メソッドを呼び出します。

次の例はサーブレットの `init` メソッドを示しています。このメソッドは、アプリケーションの web.xml ファイルから初期化パラメータを取得し、サーブレットの `doGet` メソッド内で後で使用されるフッターを作成します。

ServletConfig オブジェクトを介して `init` メソッドを呼び出すことも、介さずに呼び出すこともできます。ただし、この場合は、初期化パラメータにアクセスしてフッターを作成できるように、ServletConfig を使用してメソッドを呼び出します。

```
public class InitOverride extends HttpServlet {

  String name;
  String email;
  String copyright;
  String footer;

  public void init (ServletConfig config) throws ServletException {
    super.init(config);
    name = config.getInitParameter("name");
    email = config.getInitParameter("email");
    copyright = config.getInitParameter("copyright");
   footer = ("<HR><FONT FACE=¥"arial,helvetica,sans-serif¥"
        SIZE=-2>Copyright " + copyright + ".Brought to you by:<A
        HREF=¥"mailto:"+ email + "¥">" + name + "</A></FONT>");
  }

  public void doGet ( HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    response.setContentType("text/html");
...
    out.println(footer);
    out.println( "</body></html>" );
  }
}
```

次の行は、初期化パラメータを含めた web.xml ファイル内のサーブレット定義を示しています。

```
<servlet>
 <servlet-name>InitOverride</servlet-name>
 <servlet-class>InitOverride</servlet-class>
 <init-param>
  <param-name>name</param-name>
  <param-value>Nick Danger</param-value>
 </init-param>
 <init-param>
  <param-name>email</param-name>
  <param-value>ndanger@sandstonebuilding.com</param-value>
 </init-param>
 <init-param>
  <param-name>copyright</param-name>
  <param-value>2000, 2001, 2002</param-value>
 </init-param>
 <load-on-startup>1</load-on-startup>
</servlet>
```

ServletContext オブジェクトでのスタティックデータのキャッシュ

ServletContext オブジェクトはオブジェクトを属性として保管します。アプリケーション内のすべてのサーブレットが同じコンテキストにアクセスするため、ハッシュテーブルなどの構造のようなオブジェクトをキャッシュすることによって、アプリケーションの処理負荷を軽減できます。さらに、オブジェクトをサーブレットの `init` メソッド内のコンテキストに保管すると、メソッドは 1 回しか実行されません。

サーブレットの `init` メソッドを使用してオブジェクトを作成し、ServletContext 内に保管し、`ServletContext.getAttribute` への呼び出しを使用してサーブレットの本文内のオブジェクトにアクセスします。データベーステーブルの場合は、ResultSet オブジェクトを属性として保管できません。配列などのオブジェクトとしてデータの要素を変更する必要があります。

次の例では、サーブレットの `init` メソッド内で ResultSet オブジェクトを反復処理し、コンテキストの `setAttribute` メソッドを使用して ServletContext 内に Hashtable オブジェクトを保管します。サーブレットは `doGet` メソッド内で ServletContext を介して Hashtable オブジェクトにアクセスし、リクエストパラメータを使用して適切な出力行を取得します。

```
public void init () throws ServletException {
...
ResultSet rs = stmt.executeQuery(sqlstmt);
Hashtable h = new Hashtable();
while (rs.next()) {
  h.put(rs.getString("ABBR"),rs.getString("LONGNAME"));
  }
getServletContext().setAttribute("statelist", h);
...
}

public void doGet (HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
  ...
```

```
  Hashtable h = (Hashtable) getServletContext().
        getAttribute("statelist");
  String state = (String) h.get(request.getParameter("state"));
  out.println(request.getParameter("state") + " is an abbreviation for
        " + state);
  ...
}
```

JSP で作業を行う場合は、ServletContext オブジェクトとして暗黙に宣言されている application オブジェクトを使用します。

println の代わりに print メソッドを使用

PrintWriter オブジェクトにデータを送信する際は、次の例のように `println` メソッドではなく `print` メソッドを使用します。

```
PrintWriter out = response.getWriter();
//out.println("これは効率のよくない出力です。");
out.print("これは効率のよい出力です。");
```

Java では、`print` メソッドの方が `println` メソッドよりも効率のよい出力メソッドです。`println` は内部で入力を取り込み、`print` に送信します。`println` と `print` の使用方法の違いは、`println` が文字列の最後に改行文字を挿入することです。しかし、クライアントで HTML のソースを表示しないかぎり、出力のレンダリング方法は変わりません。

HttpSession オブジェクトによるステート管理

ステート管理の場合、非表示フィールド、Cookie、および URL リライティングではなく HttpSession オブジェクトを使用します。HttpSession API を使用する際に、JRun はクライアントとサーバー間でセッションデータ自体ではなくセッション ID だけを渡します。これは、セッションオブジェクトの API により実装の詳細が非表示になっているため、他のメソッドよりも使用し易くなっています。

JSP 開発者の場合は、暗黙のセッションオブジェクトを使用します。

HttpSession オブジェクトを使用したステート管理の詳細については、151 ページの「セッションの操作」を参照してください。

出力のフラッシュ

ページ全体のロードが終了する前にクライアントにページの一部が表示されるように、定期的にデータをフラッシュします。それによってアプリケーションの全体的なレスポンス速度が速くなるわけではありませんが、ユーザーはページがより速やかに処理されたように感じます。

この手法は、ダウンロードに時間がかかるグラフィックスが多く含まれているセクションや多くの処理が必要なセクションがある場合に有効です。

次に例を示します。

```
PrintWriter out = response.getWriter();
out.print(header);
out.flush(); //ヘッダーをフラッシュします。
...
```

```
out.print("/images/large_graphic.gif");
out.flush(); // 本文内の容量の大きいグラフィックをフラッシュします。
...
out.print(footer);
out.flush(); // フッターをフラッシュします。
```

この手法は、暗黙の out オブジェクトとともに JSP ページに使用します。

レスポンスオブジェクトのバッファーサイズの拡張

サーブレットはレスポンスオブジェクトのバッファー内のコンテンツをロードします。バッファーが満杯の場合、サーブレットはクライアントへのソケット接続を作成し、バッファーをフラッシュします。ソケットおよびネットワークトラフィックの数を減らすには、バッファーの容量を大きくします。

サーブレットのレスポンスバッファーのデフォルト値は 4 KB です。レスポンスサイズがこのサイズに到達すると、JRun はバッファーをフラッシュします。次の例のように `setBufferSize` メソッドを使用して、バッファーに大きい値を設定し、必要なフラッシュ回数を減らします。

```
response.setBufferSize(8192);
```

バッファーのサイズを確認するには、次の例のように `getBufferSize` メソッドを使用します。

```
out.print("バッファーサイズは " + response.getBufferSize() です。);
```

PrintWriter オブジェクトのバッファーサイズの拡張

PrintWriter のバッファーサイズを拡張すると、JRun によるレスポンスデータのフラッシュ回数が減少します。その結果、オープンソケットや接続の数が減少します。

次の例のように ByteArrayOutputStream オブジェクトを使用して PrintWriter のバッファーサイズを大きくします。

```
...
response.setContentType("text/html");
ByteArrayOutputStream bos = new ByteArrayOutputStream(8192);
PrintWriter out = new PrintWriter(bos, true);
out.print("<html><head><title>Using BytaArrayOutputStream</title></
        head>");
out.print("<body>");
...
out.print( "</body></html>" );
response.setContentLength(bos.size());
bos.writeTo(response.getOutputStream());
...
```

ServletContext.log への呼び出し制限

ServletContext.log メソッドへの呼び出しにより、パフォーマンスが低下する場合があります。可能な範囲内でこのメソッドへの呼び出しを制限し、System.out.println への呼び出しを使用します。

JSP の最適化

このセクションでは、JSP のレスポンス速度を向上させる方法について説明します。

JSP は最初にリクエストされたときにサーブレットにコンパイルされるので、その他の方法についてはサーブレットの最適化手法を参照してください。詳細については、218 ページの「サーブレットの最適化」を参照してください。

JSP のプリコンパイル

デプロイ前にコンパイルされるサーブレットとは異なり、JSP ページは最初にリクエストされたときに JRun によってコンパイルされます。そのため、明らかな遅延が発生し、ユーザーによってはレスポンス速度が遅いと感じる可能性があります。JRun には、JSP をプリコンパイルするオプションがあります。

JSPC コンパイラは、Web サーバーのコンテキストの外部で JSP をコンパイルするときに使用するコマンドラインツールです。JSPC コンパイラを使用すると、JSP のリクエスト時に JRun を使用して JSP をコンパイルするのではなく、コマンドラインから明示的に JSP をコンパイルできます。JSPC が <JRun のルートディレクトリ >/bin ディレクトリに置かれている場合は IBM jikes コンパイラを使用します。そうでない場合は、Sun javac コンパイラを使用します。

JSP をプリコンパイルする場合、これらのファイルの変更に対する JRun の自動検出機能を使用不可にしてください。詳細については、222 ページの「変更検出の無効化」を参照してください。

メモ：JSPC コンパイラは、JRun での JSP のコンパイルに使用されるコンパイラと同じコンパイラです。唯一異なる点は、JSPC コンパイラはコマンドラインから起動することです。

JSPC コンパイラの使用方法の詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

変更検出の無効化

デフォルトでは、JRun が Web コンポーネントファイル (サーブレットや JSP など) を巡回し、最後の巡回時から変更があった場合にそれを検出します。この処理は、ホットデプロイや自動検出による変更を有効にするものです。ただし、このサイクルはパフォーマンスを低下させるため、運用環境では使用不可にしてください。222 ページの「JSP のプリコンパイル」の説明にあるとおり、JSP をプリコンパイルする場合は特に不要です。

次の例に示すように、default-web.xml ファイルで JSPServlet の `translationDisabled` 初期化パラメータを true に設定します。

```
<servlet>
   <servlet-name>JSPServlet</servlet-name>
   <servlet-class>jrun.jsp.JSPServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
   <init-param>
     <param-name>keepGenerated</param-name>
     <param-value>false</param-value>
   </init-param>
   <init-param>
     <param-name>translationDisabled</param-name>
     <param-value>true</param-value>
   </init-param>
 </servlet>
```

ServletContext オブジェクトでのスタティックデータのキャッシュ

ServletContext オブジェクトはオブジェクトを属性として保管します。ハッシュテーブルなどの構造のようなオブジェクトをキャッシュすることによって、アプリケーションの処理負荷を軽減できます。

JSP での暗黙のアプリケーションオブジェクトは、その JSP の ServletContext です。サーブレット内で実行する場合のように、ServletContext を取得するためのメソッドを実行する必要はありません。たとえば、次のコードによって JSP 内の ServletContext オブジェクトに属性を設定します。

```
application.setAttribute("statelist", qt);
```

JRun タグライブラリの **jrun:sql** タグを使用して QueryTable を取得する場合は、QueryTable オブジェクトをアプリケーションの属性として保管できます。ResultSet を Hashtable などの直列化可能なオブジェクトに変換する必要はありません。

ServletContext オブジェクトは、JSP に対してサーブレットの場合と同様に動作します。詳細については、219 ページの「ServletContext オブジェクトでのスタティックデータのキャッシュ」を参照してください。

セッションの無効化

JSP がリクエストされるたびに、JRun は HttpSession オブジェクトを作成して、固有のクライアントごとにステートを維持します。JSP では、セッションデータは 暗黙のセッションオブジェクトとしてアクセス可能です。JSP では、セッションはデフォルトで有効になっています。

各セッションオブジェクトはサーバー上に保管されているため、システムリソースを少し使用します。さらに、セッションが有効になっている場合は、クライアントとサーバー間のトラフィックにセッション ID が含まれている必要があります。それによって、さらに小さいオーバーヘッドが発生します。

セッションを無効にするには、**page** ディレクティブを使用します。次の例のように、page ディレクティブの **session** 属性を false に設定します。

```
<%@ page session="false" %>
```

レスポンスオブジェクトのバッファーサイズの拡張

レスポンスオブジェクトのバッファーサイズを大きくして、ソケットの作成を減らします。

JSP レスポンスバッファーのデフォルトサイズは 8 KB です。レスポンスサイズが少なくともこのサイズに到達すると、JRun はバッファーをフラッシュします。次の例のように、バッファーに大きい値を設定し、必要なフラッシュ回数を減らします。

```
<%@ page buffer="16kb" %>
```

page ディレクティブは JSP ページ全体とそれに含まれているファイルに適用されます。

バッファーのサイズを確認するには、次の例のように、暗黙のレスポンスオブジェクトの **getBufferSize** メソッドを使用します。

```
バッファーサイズは <%= response.getBufferSize() %> です。
```

include の正しい使用方法

スタティックページを含める際は、できるだけ include アクションではなく include ディレクティブを使用します。include ディレクティブは、コンパイル時 (JSP の場合は最初にページがリクエストされたとき) に含まれたファイルの内容を挿入します。一方、include アクションは、実行時に (ページがリクエストされるたびに) 含まれたファイルをフェッチします。include アクションの方がサーバーサイドでより多くの処理が必要です。

たとえば、スタティックファイル header.html がある場合は、次の例のように include ディレクティブを使用してファイルを含めます。

```
<%@ include file="/static-pages/header.html" %>
```

次の例のように include アクションを使用して、スタティックファイルをインクルードしないでください。

```
<jsp:include page="/static-pages/header.html" flush="true" />
```

SSI (Server-Side Include：サーバーサイドインクルード) と include は同じタスクを実行しますが、JSP 内では SSI タグレットを使用しないでください。JRun では SSI をサポートしなくなりました。

出力をフラッシュしない

前バージョンの JRun (および以前の仕様に付属のサーブレットエンジン) では、include アクションの実行後にバッファー出力をフラッシュする必要がありました。include アクションでは、必要な flush 属性を true に設定する必要がありました。JRun 4 では、この属性が false に設定されていて、自由に変更することができます。属性を false に設定することで、出力バッファーのフラッシュを遅らせることができます。ページのサイズにもよりますが、これで JSP のパフォーマンスが向上します。

次のコードは、include アクションの flush 属性が false に設定されていることを示しています。

```
<jsp:include page="myadditionalpage.jsp" flush="false" />
```

setProperty ショートカットの使用

JavaBeans は、JSP によって使用されるステート情報などのデータを保管します。フォーム変数を処理し、各入力フィールドを bean プロパティとして保管する作業は単調で、更新が難しく、ミスが発生する可能性があります。

フォームフィールドを JavaBean プロパティに設定する一般的な setProperty タグは次のとおりです。

```
<jsp:setProperty name="myBean" property="first_name" value="<%=
        request.getParameter("first_name") %>" />
```

JSP には、すべての bean プロパティを 1 つのタグで設定できるショートカットが用意されています。

jsp:setProperty タグがすべてのリクエストパラメータを反復処理し、パラメータ名および値のタイプが bean プロパティ名およびタイプと一致するように、property 属性を * に設定します。一致したプロパティはそれぞれ、対応するパラメータの値に設定されます。setProperty タグ内には value パラメータを指定しません。

property ワイルドカードの使用例を次に示します。

```
<jsp:setProperty name="myBean" property="*" />
```

パラメータ値を空の文字列 ("") に設定する場合、対応するプロパティは変更されません。リクエストパラメータに対応する bean プロパティがない場合、そのパラメータは無視されます。

jsplnit でのスタティックデータのキャッシュ

クライアントが最初に JSP をリクエストしたとき、JRun は **jspInit** メソッドを呼び出します。このメソッドは、JSP ソースコードが変更された場合、または JRun サーバーが再起動した場合のみ再び呼び出されます。**jspInit** メソッドはサーブレットの **init** メソッドに相当します。

JSP の **jspInit** メソッドをオーバーライドし、スタティックデータをこのメソッドに保管できます。一般的な例としては、データベースコネクションプールを作成し、初期化するタスクがあります。その他、アプリケーションのスコープに入っている初期化パラメータを取得するといったタスクも **jspInit** メソッド内で処理できます。

次のコードサンプルは、**jspInit** メソッド内の初期化パラメータを取得し、後で本文内で使用する JSP ページを示しています。

```
<%!
 String email;
 public void jspInit() {
  System.out.println("in jspInit()");
  javax.servlet.ServletConfig servletConfig = getServletConfig();
  email = servletConfig.getServletContext().getInitParameter("email");
}
%>
<HTML><BODY>
Email addy is:<%= email %>
</BODY></HTML>
```

次の例のように、web.xml ファイルの **context-param** ブロック内の初期化パラメータを定義します。

```
<web-app>
...
<context-param>
 <param-name>email</param-name>
 <param-value>webmaster@macromedia.com</param-value>
</context-param>
...
</web-app>
```

アプリケーションのスコープに入っている bean によるキャッシュ

bean のスコープを **application** に設定することは、JRun にとって現在のアプリケーションの bean のインスタンスが 1 つしかないことを意味します。

たとえば、データベースのステートおよびその短縮形のリストを選択し、その結果をアプリケーションのスコープに入っている bean にプロパティとして保管すると、ステートのフルネームをルックアップするたびにデータベースにクエリを実行する必要はありません。

その結果、ユーザーがサーバーを再起動するまで、JRun はデータベースクエリを 1 回しか実行しません。次の例のように、bean にアクセスする任意の JSP は、bean のプロパティで取得できるようになります。

```
<jsp:usebean id="statelist" class="jrunsamples.optimize.StateList"
        scope="application" />
<jsp:setProperty name="statelist" property="*" />
```

bean のプロパティの 1 つであるステートにアクセスするには、次の行を使用します。

```
<jsp:getProperty name="statelist" property="MA" />
```

bean のスコープを **session**、**page**、または **request** に変更することもできます。bean のスコープを session に設定する場合は、**page** ディレクティブ内の **sessions=true** をオーバーライドしないでください。

Web アプリケーション環境の最適化

このセクションでは、Web アプリケーションに合わせて JRun の動作環境を改善する方法について説明します。

セッション設定の最適化

Cookie はサーバーに保管され、クライアントとサーバー間でやり取りされるセッション ID で参照されます。Cookie はメモリに保管されるため、各セッションはシステムリソースを少し使用します。

次の表で、リソースの負荷を軽減するために変更できるセッション設定を説明します。

設定	説明
セッションの最大数	ディスクにスワップする前に JRun がメモリでキャッシュするセッションの数。JRun がセッションオブジェクトを直列化しないようにするには、メモリでキャッシュするセッションの数を増やします。 この値は、web.xml ファイルで設定できます。
使用不可になるまでの最大時間 (有効期限)	JRun が、アクティブでないセッションオブジェクトを廃棄するまでの時間。デフォルトは 1800 秒 (30 分) です。使用されていないセッションを早めにクリアするには、この値を小さくします。 ステート管理メカニズムとして Cookie を使用している場合、jrun-web.xml ファイルで (`cookie.setMaxAge` メソッドを使用して) プログラムまたは宣言で Cookie の有効期限を設定できます。

セッション設定の変更方法の詳細については、151 ページの「セッションの操作」を参照してください。

ホットデプロイの無効化

ホットデプロイ機能は、コンポーネント構造に対する変更を検出すると、実行中の Web コンポーネントを自動的に再起動 (リデプロイ) します。

ホットデプロイが有効になっていると、JRun は、すべてのリソースの日付 / タイムスタンプを定期的に確認して、ファイルが変更されたかどうかを調べます。この継続的なサイクルによってシステムリソースが少し使用されます。

ホットデプロイを無効にするには、次の属性を JRun のルートディレクトリ /servers/ サーバー名 /SERVER-INF/jrun.xml ファイル内の `jrun.deployment.DeployerService` サービスに追加します。

```
<attribute name="hotDeploy">false</attribute>
```

メモ：運用サーバー上では、ホットデプロイを常に無効にする必要があります。

スレッドプールの管理

JRun と外部 Web サーバー間のコネクタの構成には、並行処理設定の最適化が含まれます。並行処理は、HTTP リクエストをプールし、分散する方法を定義します。これらの設定値を変更することによって、各 JRun サーバーによって処理されるスレッドとリクエストの数を制限できます。実際に、その JRun サーバーのトラフィックを調節できます。

たとえば、Web アプリケーションの平均レスポンス時間が RMI-CORBA データベースという 3 段階のトランザクションが原因で遅れるのであれば、新規リクエストを拒否せずにスループットを維持するために、より多くのリクエストを受け入れるようにキューを大きくする必要があります。この場合は、最大同時リクエスト数を予想されるリクエスト数よりも大きい値にします。最大同時リクエスト数の設定値は、リソースの安全弁の役割を果たします。

メモ: " 同時リクエスト " と " 同時ユーザー " の概念は異なります。1000 人の同時ユーザーをサポートする必要がある Web サイトでは、1000 人のユーザーがそれと同数のリクエストを一度に作成する可能性は小さいため、1000 個の同時リクエストをサポートする必要はありません。

次の表で、Web サーバーへの接続に関する並行処理を設定できる設定値を説明します。

名前	説明	デフォルト
ActiveHandlerThreads	ハンドラースレッドプールの初期サイズ。	10
Backlog	新規リクエストが拒否されるまでに受け入れられる同時リクエストの数 (旧 Maximum Concurrent Requests)。	1000
MaxHandlerThreads	新規リクエストがキューに入れられるまでに受け入れられる同時リクエストの数。 アプリケーションに複数のデフォルト同時ユーザーが必要で、CPU の処理能力が十分ある場合は、MaxHandlerThreads 値を大きくして、パフォーマンスを向上させることができます。	20
MinHandlerThreads	プール内のハンドラースレッドの最小数。 Web サイトでトラフィックの増加が急激に発生する環境では、トラフィックが急増した場合にスレッドのグループを作成しなくても済むように、MinHandlerThreads を大きい値に設定します。 また、MinHandlerThreads 値を、予期される安定したステートの負荷の同時リクエスト数に設定することもできます。たとえば、Web サイトで常時 80 個の同時リクエストが発生する場合は、MinHandlerThreads は 80 に設定します。	1
Timeout	アイドルスレッドタイムアウト、すなわちスレッドが廃棄されるまでにアイドル状態で維持される時間 (秒)。	300

並行処理設定を変更するには、次の例のように新しい値を使用して、jrun.xml ファイル内の適切なサービスにプロパティを追加します。

```
<service class="jrun.servlet.http.WebService" name="WebService">
 ...
 <attribute name="minHandlerThreads">80</attribute>
 ...
</service>
```

JRun Web サーバーの設定を編集するには、WebService の属性を変更します。外部 Web サーバーのコネクタ設定を編集するには、ProxyService の属性を変更します。

メモ：デフォルトの JRun 並行処理設定を変更する前に、トラフィックパターンが実際に監視できることを確認します。設定値を変更してしまうと、リソースを消耗する可能性があります。Web アプリケーションの負荷テストの詳細については、244 ページの「その他のリソース」のリソースを参照してください。

さまざまな JVM の使用

オブジェクトを再利用してキャッシュすると、JVM ガーベッジコレクションおよびオブジェクト割り当てに関連するメモリと CPU サイクルを節約できます。オブジェクト割り当てには CPU サイクルとメモリが必要です。未リファレンスオブジェクトのガーベッジコレクションでは、ガーベッジコレクタのスレッドの実行中に JVM 内の他のすべてのスレッドを保留する必要があり、アプリケーションの処理速度が低下します。

選択した JVM により、Web アプリケーションのパフォーマンスに大幅な差が生じる可能性があります。Sun Microsystems の HotSpot Server VM および Appeal の JRockit では、世代間ガーベッジコレクションのような高度なアルゴリズムを使用しており、ガーベッジコレクションスレッドの実行時間が短縮されています。

さまざまな環境で負荷テストを実行して、どのオペレーティングシステムと JVM の組み合わせが特定の Web アプリケーションに最適かを調べる必要があります。

JVM のヒープサイズの拡張

JVM のメモリ設定を変更すると、JRun サーバーのパフォーマンスを向上させることができます。ただし、調整する前に、アプリケーションに必要なメモリの容量を知っておく必要があります。アプリケーションで負荷テストを行い、ヒープサイズを表示します。

JVM の最小 (または初期) ヒープサイズを、負荷テスト時に記録した最大値 (メモリ消費量) に設定します。また、最大ヒープサイズ値は、使用可能なメモリの合計から他の必要なリソースを引いた容量よりも小さい値に設定しますが、できるだけ最大の値を選択します。ご使用の JDK 内のデフォルト値 (通常は 128 MB) に最大ヒープサイズが設定されている場合に、その最大値を越えると、"OutOfMemory" エラーが発生する可能性があります。

最大ヒープサイズ値を設定するには、JMC の [JVM 設定] パネルの [最大ヒープサイズ (MB)] フィールドを編集します。[JVM 設定] パネルの [VM 引数] フィールドにコマンドライン引数を追加することによって他の設定を変更できます。

次の表で、設定可能な引数を説明します。これらの引数は、ご使用の JVM によって異なります。

引数	説明
-Xms< 値 >	Java ヒープサイズの初期 (最小) 値
-Xmx< 値 >	Java ヒープサイズの最大値
-Xss< 値 >	Java スレッドスタックサイズの値

パフォーマンスを最大限に高めるには、最大ヒープサイズと最小ヒープサイズを同じ値に設定します。それによって、JRun プロセスでヒープサイズを増やさずに済みます。

詳細については、JVM のドキュメントを参照してください。

ロギングによる負担の減少

ログファイルは、デバッグや Web アプリケーションの利用状況の追跡には便利なツールですが、実質的なログファイルの生成に必要なオーバーヘッドが著しい場合があります。このような理由から、運用アプリケーションではロギングを無効にし、その代わりに Web サーバーのログを情報源として使用することを検討してみてください。

JRun でロギングを無効にするには、次の例に示すように、/JRun のルートディレクトリ / servers/ サーバー名 /SERVER-INF/jrun.xml ファイルで LoggerService 属性を false に設定します。

```
...
<service class="jrunx.logger.LoggerService" name="LoggerService">
...
  <attribute name="errorEnabled">false</attribute>
  <attribute name="warningEnabled">false</attribute>
  <attribute name="infoEnabled">false</attribute>
  <attribute name="debugEnabled">false</attribute>
  <attribute name="metricsEnabled">false</attribute>
...
</service>
...
```

デフォルトでは、冗長なデータベースロギングは使用不可になっています。Web アプリケーションの開発中にこのロギングを有効にした場合、再び無効にするには、jrun-resources.xml ファイル内の適切なデータソースを編集し、デバッグ要素の値を false に設定します。

JDBC の最適化

このセクションでは、データベースへのアクセス時のコーディング作業を効率化する方法について説明します。

データベースコネクションプールの使用

コネクションプールを使用すると、データベース接続はリクエストごとに作成され廃棄されるのではなく、いったん借用されて、再利用可能なオブジェクトのプールに戻されます。JRun には、独自のコネクションプールメカニズムが組み込まれており、デフォルトで、データソースを JMC に追加するたびに有効になります。

JRun のデータソースを使用していない場合は、独自にカスタマイズしたコネクションプールのメカニズムを実装する必要があります。

JMC 内では影響を受けないオプションの設定を使用して、JRun のデータソースを操作できます。これらのすべての設定はオプションです。次の表で、これらの設定について説明します。

パラメータ	説明	デフォルト
native-results	キャッシュ、スクロールおよび更新が可能な JRun ResultSet 実装を使用するには true に設定します。 その下にある JDBC ドライバによって返される ResultSet を使用するには false に設定します。	true
initial-connections	プールのインスタンス化時に JRun が作成する接続の数を設定します。	1
pool-statements	呼び出しのたびに PreparedStatements を再作成せずにプールしておくには、true に設定します。	true
minimum-size	JRun がプール内で保持するオブジェクトの最小数を設定します。	0
maximum-size	JRun がプール内で保持するオブジェクトの最大数を設定します。	2147483647
maximum-soft	プールの最大サイズに到達したが、リクエストがまだオブジェクト上で待機している場合に、コネクションプールで新しい緊急のオブジェクトが作成されるようにするには、true に設定します。それによってプールのサイズは一時的に大きくなりますが、スキマーがアクティブになると、プールは自動的に許容可能なサイズに縮小します。 オブジェクトが使用可能になるまでリクエストを待機させるには、false に設定します。 スキマーの詳細については、skimmer-frequency の設定を参照してください。	true
connection-timeout	JRun が接続を廃棄するまでの休止状態を維持する時間 (秒) を設定します。	1200
user-timeout	接続が自動的にプールに戻されるまでにユーザーが接続を維持する時間 (秒) を設定します。	20

パラメータ	説明	デフォルト
skimmer-frequency	プールスキマーがリープサイクルの間で待機する時間 (秒) を設定します。スキマーは各リープサイクルですべての接続 (チェックインした接続とチェックアウトした接続の両方) を評価して、それらの接続がタイムアウトになった場合に自動的にプールに戻すか、あるいは廃棄するかを決めます。	420
shrink-by	1 つのリープサイクルでプールから削除できるオブジェクトの最大数を設定します。 JRun は、スキマーによってプールが縮小されるたびにこの値を確認します。そのため、JRun のコネクションプールメカニズムによってピーク時にプールが急激に縮小されるのを防ぎます。	5
debugging	冗長なロギング情報を有効にするには、true に設定します。	false
cache-enabled	クエリ /ResultSet のキャッシュを有効にするには、true に設定します。	false
cache-size	キャッシュできるクエリ /ResultSet のペアの最大数を設定します。	5
cache-refresh-interval	キャッシュがデータベースからその ResultSet をリロードする間隔 (秒) を設定します。	30
remove-on-exceptions	SQLException がある場合にプールから接続を削除するには、true に設定します。	false

コネクションプールの設定を変更するには、jrun-resources.xml ファイル内の適切なデータソースのプロパティを編集します。たとえば、次の設定により、デフォルトの `skimmer-frequency` 値の 420 をオーバーライドし、100 に置き換えます。

```
<datasource>
...
 <skimmer-frequency>100</skimmer-frequency>
...
</datasource>
```

PreparedStatement の使用

データベースにクエリを実行すると、JRun はクエリをコンパイルし、データベース接続を介してデータベースに送信します。頻繁に使用するクエリには PreparedStatement を使用できます。PreparedStatement はプリコンパイルされ、サーバーに保管されています。PreparedStatement には ? を使用して変数を任意の数だけ指定できます。これらの変数は実行時にクエリに挿入されます。

次のコード例は、3 つの変数を使用して PreparedStatement を作成する方法を示しています。

```
String sqlstmt = "INSERT INTO SESSIONS VALUES(?,?,?)";
try {
 InitialContext ctx = new InitialContext();
 DataSource ds = (DataSource) ctx.lookup(dsName);
 conn = ds.getConnection();
 ps = conn.prepareStatement(sqlstmt);
 ps.setString(1,sid);
 ps.setLong(2,time);
 ps.setInt(3,1);
 rs = ps.executeQuery();
} catch (SQLException sqle) {
...
}
...
```

PreparedStatement オブジェクトの使用方法の詳細については、Java 2 API のドキュメントを参照してください。

Connection、Statement、および ResultSet オブジェクトを閉じる

Connection、Statement、および ResultSet オブジェクトを使用したら、これらのオブジェクトを閉じて、要求されたリソースを解放し、使用されていない接続をプールに戻す必要があります。次の例のように finally ブロックを追加します。

```
...
} finally {
 try {
  ps.close();
  rs.close();
 } catch (Exception e) {
  e.printStackTrace();
 }
}
```

フェッチおよび行数の制限

ResultSet および Statement オブジェクトには、次のようなデータベースオペレーションの微調整に役立つ多数のメソッドが用意されています。

- Statement.setMaxRows
- setFetchSize
- setFetchDirection

setFetchSize および setMaxRows メソッドは、開発者の間でもよく混同されます。これらのメソッドは ResultSet または Statement オブジェクト上で呼び出すことができます。これらのメソッドの違いを理解するには、まずデータベースがクエリの結果を返すときの動作を理解する必要があります。次のセクションでは、データベースにクエリを実行する方法、および setFetchSize メソッドと setMaxRows メソッドの違いについて説明します。

データベースクエリの概要

データベースに対してクエリを実行すると、データベースはクエリを処理し、結果データをそのキャッシュに保管します。ResultSet オブジェクトは、データベースのキャッシュ内にフェッチサイズと同数のレコードへのリファレンスを保持します。つまり、ResultSet オブジェクトはデータベースへのオープン接続にすぎず、返された行のデータは含んでいません。

JRun サーバーとデータベース間の接続を確立する JDBC ドライバが、データベースキャッシュから行の一部を取得し、それを処理するクライアントサイド (JRun サーバーがデータベースのクライアントに相当) に返します。ドライバは、`getFetchSize` と同数の行を取得します。

クライアントがドライバによって提供された行の処理を終了したら、JDBC ドライバは次の `getFetchSize` 行をあらかじめフェッチし、同様の処理を続けます。

setMaxRows の使用

データベースがクエリから返す行の数を制限するには、次の例のように `setMaxRows` を使用します。

```
Statement stmt = conn.createStatement();
stmt.setMaxRows(10);
ResultSet rs = stmt.executeQuery("SELECT * from SESSIONS");
```

データベースが返す行の数を制限するには、SQL ステートメントで `LIMIT` コマンドを使用することもできます。次の例は、100 行に制限された ResultSet を示しています。

```
rs = ps.executeQuery("SELECT * FROM tablename LIMIT 100");
```

setFetchSize の使用

JDBC ドライバがデータベースキャッシュから取得して JRun に返す行の数を設定するには、`setFetchSize` を使用します。データに多くの処理を行う必要がある場合は、小さい値に設定します。

メモ：`setFetchSize` メソッドは、データをフェッチする方法についてのデータベースへのヒントにすぎません。すべてのデータベースドライバでこのメソッドをサポートしているとはかぎりません。

適切な `setFetchSize` を決めるには、フロントエンドでのユーザー操作と、ユーザーがクエリの結果を操作する方法について検討します。たとえば、ユーザーがユーザーインターフェイスで一回に 10 行を操作できる場合は、フェッチサイズを 10 に設定します。

setFetchDirection の使用

`setFetchDirection` の処理時に最後の行を最初に返すには、setFetchDirection メソッドを使用します。それによって、大量のデータを操作する際は特に、処理速度が速くなります。

たとえば、日付別に保管されているトランザクションの ResultSet がある場合に、直前のトランザクションを最初のフェッチで取得できます。

メモ: `setFetchDirection` メソッドは、データをフェッチする方法についてのデータベースへのヒントにすぎません。すべてのデータベースドライバでこのメソッドをサポートしているとはかぎりません。

次のコードは、フェッチ方向を反転させる例を示しています。

```
ResultSet rs = null;
Connection conn = null;
String sql = "SELECT * FROM employee";
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
rs.setFetchDirection(ResultSet.FETCH_REVERSE);
rs = stmt.executeQuery(sql);
```

JDBC ドライバのテスト

すべての JDBC ドライバが一様に動作するとはかぎりません。さまざまな負荷をかけてさまざまなデータベースで各種ドライバの動作をテストして、Web アプリケーションに最適な組み合わせを調べます。

JDBC と ODBC bridge は、絶対に必要な場合にのみ使用してください。

JDBC ドライバの詳細については、Sun の Web サイト http://industry.java.sun.com/products/jdbc/drivers をご覧ください。

スタティックデータのキャッシュ

データベースにアクセスする最も速い方法は、データベース自体にまったくアクセスしないことです。可能な場合は、データベースへの呼び出しによってスタティックデータをダイナミックに取得するのではなく、キャッシュするようにしてください。サーブレットおよび JSP の `init` メソッドおよび `jspInit` メソッド内で、初期化時にデータベースへのアクセスを 1 つのクエリに制限します。さらに、結果をキャッシュし、残りのリクエストがそのキャッシュに入ったデータにアクセスできるようにします。詳細については、次のセクションを参照してください。

218 ページの「init メソッドでのスタティックデータのキャッシュ」

225 ページの「jspInit でのスタティックデータのキャッシュ」

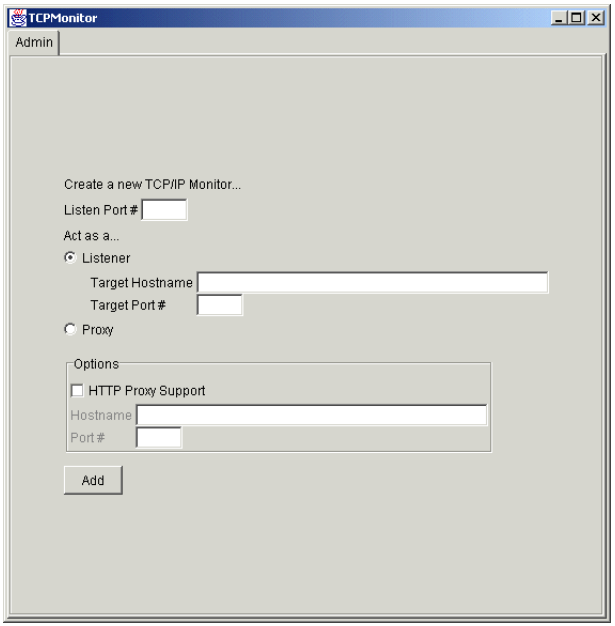
TCPMonitor の使用

TCPMonitor は、HTTP トラフィックのリクエストおよびレスポンスフローを監視できる Swing ベースのアプリケーションです。SOAP トラフィックのリクエストおよびレスポンスフローも監視できます。TCPMonitor は、JRun で以前に使用されていたスニファサービスの代わりとして機能します。

TCPMonitor を実行するには

1 Windows や Unix のようなプラットホームでは、JRun のルートディレクトリ /bin ディレクトリのスニファユーティリティを起動して TCPMonitor を実行します。

TCPMonitor メインウィンドウが表示されます。



2 次の表の説明に従って、メインウィンドウに値を入力します。

フィールド	説明
ポート # のリスン	受信した接続を監視するためのローカルポート番号 (8123 など) を入力します。サーバーが実行されている通常のポートをリクエストするのではなく、このポートをリクエストします。TCPMonitor はリクエストを阻止し、ターゲットポートに転送します。
リスナ	JRun で TCPMonitor をスニファサービスとして使用するには、[リスナ] を選択します。
プロキシ	TCPMonitor にプロキシサポートを有効にするには、[プロキシ] を選択します。
ターゲットホスト名	受信した接続の転送先のターゲットホストを入力します。 たとえば、samples JRun サーバーで実行されているサービスを監視する場合、ホスト名は `localhost` となります。
ターゲットポート #	TCPMonitor の接続先のターゲットマシンのポート番号を入力します。たとえば、samples JRun サーバーで実行されているサービスを監視する場合、デフォルトのポート番号は 8200 となります。
HTTP プロキシサポート	TCPMonitor にプロキシサポートを設定する場合のみ、このチェックボックスをオンにします。

コマンドラインで TCPMonitor を呼び出す際にリスンポート番号、ターゲットホスト名、およびターゲットポート番号をオプションで指定できます。TCPMonitor のシンタックスは次のとおりです。

```
java org.apache.axis.utils.tcpmon [ リスンポート ] [ ターゲットホスト ]
    [ ターゲットポート ]
```

3 [追加] をクリックして、このプロファイルを TCPMonitor セッションに追加します。

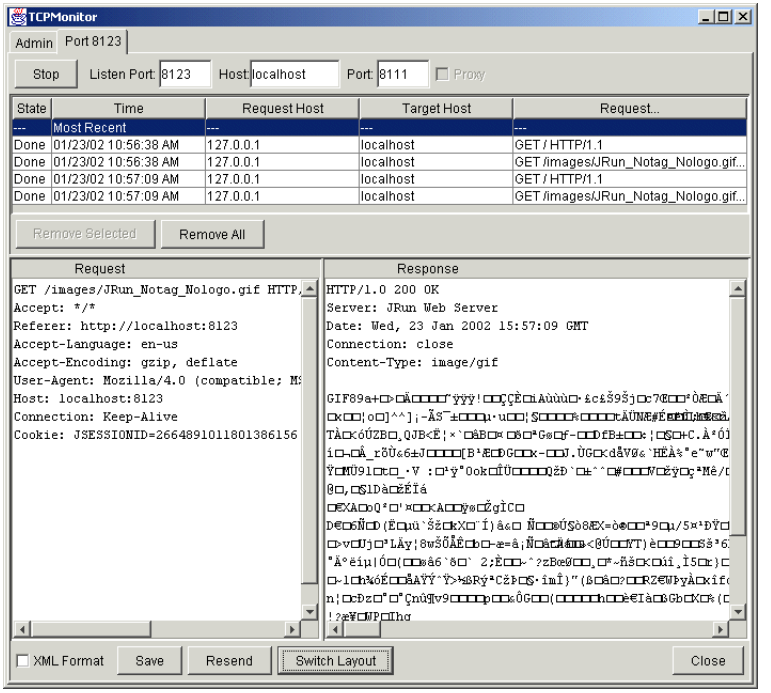
新しくトンネルした接続のタブが表示されます。

4 [新規] タブを選択します。ポートが競合している場合は、[リクエスト] パネルに警告が表示されます。

5 この TCPMonitor セッションで定義したリスンポートを使用してページをリクエストします。たとえば、[ポート # のリスン] に「8123」と入力した場合は、ブラウザに次の URL を入力します。

```
http://localhost:8123/
```

現在のリクエストおよびレスポンスの情報が表示されます。



各接続のリクエストが [リクエスト] パネルに表示され、レスポンスが [レスポンス] パネルに表示されます。TCPMonitor では、リクエストおよびレスポンスのすべての組み合わせのログが保持され、ユーザーは上部パネルのエントリを選択することによって特定の組み合わせを表示できます。

6 後で表示するために結果をファイルに保存するには、[保存] をクリックします。保存する必要がない古いリクエストが表示された上部パネルをクリアするには、[選択されたものを削除] および [すべて削除] をクリックします。

7 現在表示しているリクエストを再送し、新規レスポンスを表示するには、[再送信] をクリックします。再送する前にリクエストを [リクエスト] パネルで編集し、さまざまなリクエストの結果をテストできます。

8 ポートを変更するには、[停止] をクリックし、ポート番号を変更して [起動] をクリックします。

9 他のリスナを追加するには、[Admin] タブをクリックし、前の説明に従って値を入力します。

10 この TCPMonitor セッションを終了するには、[閉じる] をクリックします。

メソッドタイミング機能の使用

アプリケーションのパフォーマンスを向上させるための実用的な戦略では、実行時間の大部分を占めるアプリケーションの領域を識別する必要があります。最も深刻な問題を最適化することによって、アプリケーションの総合的なパフォーマンスを大幅に改善できます。

JRun には、サーブレット内の個々のメソッドのパフォーマンスを測定するためのメソッドタイミング機能が用意されています。作成したメソッドだけでなく、EJB、サードパーティライブラリ、およびヘルパクラスのメソッドの実行時間も測定できます。

メソッドタイミングの機能

SERVER-INF/jrun.xml ファイルの InstrumentationService でタイミング パラメータの値を指定することによって、メソッドタイミングがアプリケーションをどのように監視するかを設定します。

メソッドタイミング機能には、2 つのメソッドタイミングが用意されています。

- **現在のメソッドタイミング**　アプリケーションの一部であるメソッドの実行を監視します。
- **メソッド呼び出しタイミング**　呼び出されたメソッドタイミングとも呼ばれています。アプリケーションメソッド内で呼び出されたメソッドの実行を監視します。

メソッドにおけるパフォーマンスの問題を検出するための一般的な戦略では、メソッドタイミングに別の設定を使用して、アプリケーションを 2 度実行します。

- 最初の実行では、現在のメソッドタイミングを使用して、アプリケーションの一部であるすべてのメソッドの実行統計を生成します。このパスの結果には、実行時間の大部分を占めるメソッドを分離する目的があります。
- 2 回めは、メソッド呼び出しタイミングを使用して、最初のパスによって識別されたメソッド呼び出しの実行統計を生成します。このパスの結果を使用すると、ボトルネックの発生場所を判断できるはずです。

たとえば、サーブレットの doGet メソッドは他のメソッドを呼び出します。

```
public void doGet {
...
  ...xyz()...
...
}
```

doGet メソッドの現在のメソッドタイミングはメソッドの実行所要時間を生成します。この実行時間には、xyz の所要時間も含まれます。メソッド呼び出しタイミングは、メソッド xyz の実行所要時間を生成します。この結果は、実行時間が主に doGet メソッド内にあるか、または特にメソッド xyz にあるかを示します。

メソッドタイミングの設定

メソッドタイミングは、jrun.xml ファイルで定義する MethodInstrumentor サービスを使用して制御します。このサービスを使用して、次の項目を指定できます。

- **除外するクラス** excludeCallsTo 属性で制御します。デフォルトでは、java.*、javax.*、および sun.* パッケージのメソッドはタイミングを計測できません。
- **タイミングを計測するメソッドを含んでいるクラス** className 属性で制御します。
- **直接サブクラスのタイミングを計測するかどうか** directSubclasses 属性で制御します。この設定は、javax.servlet.http.HttpServlet などのクラスには有効です。一般に、このクラスを拡張してサーブレットを定義します。
- **現在のメソッドタイミングの有効化** instrumentMethods 属性で制御します。true または false を指定します。
- **タイミングを計測するメソッドの指定** instrumentMethod 属性で制御します。instrumentMethod 属性の行を必要な数だけ使用してメソッド名を指定します。すべてのメソッドのタイミングを計測するには、アスタリスク (*) を指定します。
- **メソッド呼び出しタイミングの有効化** instrumentCalls 属性で制御します。true または false を指定します。
- **タイミングを計測するメソッド呼び出しの指定** instrumentCallsTo 属性で制御します。instrumentCallsTo 属性の行を必要な数だけ使用してメソッド名を指定します。すべてのメソッド呼び出しのタイミングを計測するには、アスタリスク (*) を指定します。
- **出力オプション** outputToStandardLogger および outputToRequestThread 属性で制御します。true または false を指定します。

計測の詳細のプリント

JRun ロガーは、パフォーマンス出力を処理します。デフォルトでは、JRun はログファイルに出力を書き込みますが、jrun.xml ファイルの LoggerService メカニズムを設定することで、クライアントまたは別のファイルに出力を書き込むように指示することもできます。

実行時間の表示

JRun には、InstrumentationService で Web 出力を定義するメソッドの実行時間に関する統計を収集する JRunTimingFilter という機能があります。JRunTimingFilter は、JRunStatistics サーブレットを呼び出し、タイミング情報を表示するターゲットリソースの一番下にテーブルを作成します。

次の行は、default-web.xml ファイル内の JRunTimingFilter の定義を示しています。

```
<filter>
  <filter-name>JRunTimingFilter</filter-name>
  <filter-class>jrun.servlet.filters.TimingFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>JRunTimingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

デフォルトでは url-pattern のマッピングが /* なので、JRunTimingFilter は、有効にするとリクエストごとに呼び出されます。

次の表のように、JRunTimingFilter には 2 つの初期化パラメータがあります。

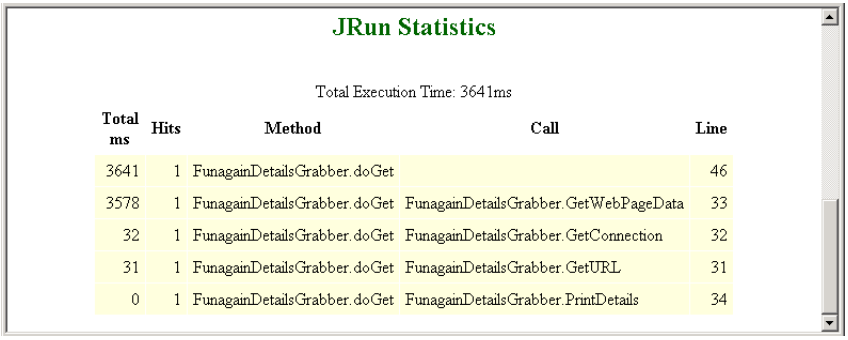
初期化パラメータ	説明
statsPage	レスポンス後にインクルードする統計ページの名前を定義します。デフォルトは /JRunStatistics です。
mimeTypes	JRun により JRunTimingFilter で適用されるレスポンス MIME タイプのカンマ区切りリストを定義します。MIME タイプの最後には、ワイルドカード (*) が 1 つずつあります。デフォルトは text/* です。

デフォルトでは、JRunTimingFilter は無効になっています。フィルタの定義およびマッピングを有効にするには、これらのコメントを解除する必要があります。

次の行は、default-web.xml ファイル内の JRunStatistics サーブレットの定義を示しています。この設定は編集しないでください。

```
<servlet>
  <servlet-name>JRunStatistics</servlet-name>
  <jsp-file>/jrunx/instrument/Results.jsp</jsp-file>
</servlet>
<servlet-mapping>
  <servlet-name>JRunStatistics</servlet-name>
  <url-pattern>/JRunStatistics</url-pattern>
</servlet-mapping>
```

JRun では、JRunTimingFilter を有効にすると、次の例に示すように、ページの実行までの時間と InstrumentationService で処理するメソッドがページの一番下に挿入されます。



JRun Statistics

Total Execution Time: 3641ms

Total ms	Hits	Method	Call	Line
3641	1	FunagainDetailsGrabber.doGet		46
3578	1	FunagainDetailsGrabber.doGet	FunagainDetailsGrabber.GetWebPageData	33
32	1	FunagainDetailsGrabber.doGet	FunagainDetailsGrabber.GetConnection	32
31	1	FunagainDetailsGrabber.doGet	FunagainDetailsGrabber.GetURL	31
0	1	FunagainDetailsGrabber.doGet	FunagainDetailsGrabber.PrintDetails	34

InstrumentationService の設定の詳細については、240 ページの第 9 章「メソッドタイミングの設定」を参照してください。

メッセージの形式

タイミングを計測するメソッドが呼び出されるたびに、JRun は出力先に 2 つのメッセージを送信します。最初のメッセージには、識別情報、そのメソッドタイミングの開始を示すテキスト、およびメソッドの開始時間が記載されます。2 番めのメッセージには、同じ識別情報とメソッドタイミングの終了を示すテキストのほかに、メソッドの終了時間および経過時間 (ミリ秒) が記載されます。

現在のメッセージタイミング形式

現在のメソッド呼び出しに関するメッセージの形式には、次のコンポーネントが含まれます。

```
currentTimeMillis, loglevel, type, className, hashCode, methodName,
        methodType, elapsed
```

次の表で、これらの値について説明します。

値	説明
currentTimeMillis	メッセージの日付と時刻。時刻はミリ秒単位で記録されます。
loglevel	タイミングメッセージのレベル。info、warning、error、または debug。
type	メソッドの開始または終了を表す文字列。
className	メソッドが含まれているクラスの名前。
hashCode	ハッシュコードは、マルチスレッド環境で役に立つ、固有オブジェクトリファレンスです。2 つのスレッドが同じサーブレットで同時に機能している場合、メソッドタイミングメッセージはインターリーブされます。このため、ハッシュコードを使用すると、メッセージを区別できます。
methodName	メソッドの名前
methodType	void を示す V など、戻り値のタイプを含むメソッドの署名。
elapsed	メソッド呼び出しのミリ秒単位の経過時間 (EXIT メソッドのみ)。

doGet メソッドの開始または終了によって生成されるタイミングメッセージの例を次に示します。

```
01/13 10:57:37 info METHOD ENTER SimpleServlet 7126423
        doGet(javax.servlet.http.HttpServletRequest,javax.servlet.htt
        p.HttpServletResponse) 11

01/13 10:57:37 info METHOD EXIT SimpleServlet 7126423
        doGet(javax.servlet.http.HttpServletRequest,javax.servlet.htt
        p.HttpServletResponse) 25 0
```

メソッド呼び出しタイミング

呼び出されたメソッド呼び出しメッセージの形式には、次のコンポーネントが含まれています。

```
currentTime loglevel type,className hashCode,methodName,
methodType,callClassName,callMethodName,callMethodType,line,elapsed
```

このメッセージの形式には、現在のメソッド呼び出しメッセージのすべてのコンポーネントと次の追加コンポーネントが含まれます。追加コンポーネントは太字で示されています。

その他の値については次の表で説明します。

値	説明
callClassName	タイミングを計測するメソッドのクラス名
callMethodName	現在のメソッドによって呼び出された、呼び出されたメソッドの名前 (methodName)
callMethodType	呼び出されたメソッドの署名 (callMethodName)
line	呼び出されたメソッド (callMethodName) を含む、メソッド (methodName) のステートメントの行番号

例

次のセクションでは、メソッドタイミングの使用例について説明します。

メソッドタイミングのデフォルト設定の有効化

jrun.xml ファイルのデフォルトのメソッドタイミング設定はサーブレットに対して有効に機能します。

現在のメソッドタイミングを有効にするには

1 `instrumentMethods` 属性を true に設定します。

2 1 つ以上のサーブレットを実行します。

3 JRun サーバーのログファイルを表示します。

EJB タイミング設定の有効化

EJB タイミングを有効にするには、jrun.xml ファイルに次の変更を行う必要があります。

- タイミングを計測する EJB クラスの `className` 属性 (`javax.ejb.SessionBean` など) を追加します。
- `instrumentMethods` および `instrumentCalls` 属性を使用してタイミング機能を有効にします。
- `instrumentMethod` および `instrumentCallsTo` 属性を使用して、タイミングを計測するメソッドおよび呼び出されたメソッドを指定します。

その他のクラスの設定の有効化

JRun メソッドタイミング機能を使用して、JRun クラスローダーによって処理される任意のクラスのタイミングを計測できます。その中には、サーブレット、JSP、EJB、ヘルパークラス、および JavaBeans が含まれます。JavaBean などのクラスのメソッドタイミングを有効にするには、jrun.xml ファイルに次の変更を行う必要があります。

- タイミングを計測するクラスの `className` 属性 (`helpers.DataObject` など) を追加します。
- `instrumentMethods` および `instrumentCalls` 属性を使用してタイミング機能を有効にします。
- `instrumentMethod` および `instrumentCallsTo` 属性を使用して、タイミングを計測するメソッドおよび呼び出されたメソッドを指定します。

その他のリソース

Web アプリケーションのパフォーマンスおよびスケーラビリティを向上させるために次の参考文献が役立つ場合があります。

リソース	格納場所
『Java 2 Performance and Idiom Guide』	Craig Larman、Rhett Guthrie、Prentice Hall、1999 年 8 月
『Java Platform Performance: Strategies and Tactics』	Steve Wilson、Jeff Kesselman、Addison-Wesley Pub Co. 刊、2000 年 5 月
『Java Performance and Scalability Volume 1』	Dov Bulka、Addison-Wesley 刊、2000 年 5 月
『Miscrosoft Web Application Stress Tool』	http://webtool.rte.microsoft.com/
TCPMonitor のドキュメント	http://xml.apache.org/axis/
Java セキュリティマネージャの使用	http://developer.java.sun.com/developer/TechTips/2000/tt0124.html#tip2

パート III
JSP プログラミング

パート III では、JRun による JSP と カスタムタグプログラミングについて説明します。次の章で構成されています。

第 10 章
JSP プログラミングテクニック

この章では、JRun コンテナ内で実行される JavaServer Page (JSP) のプログラミングテクニックを説明します。サーバーサイド Java プログラミングと J2EE スイートの基礎知識があることを前提としています。JSP の経験がまったくない場合は、『JRun 入門』をお読みください。

目 次

JSP 入門

JSP を利用することによって、HTML とスクリプトコードの組み合わせを含んでいるテキストファイルからサーブレットを作成できます。クライアントが JSP をリクエストすると、ページがサーブレットに変換されます。JSP のスクリプト部分を使用して、ダイナミックコンテンツを作成し、クライアントに返送できます。さらに、JSP からサーブレットと EJB コンポーネントにアクセスできます。

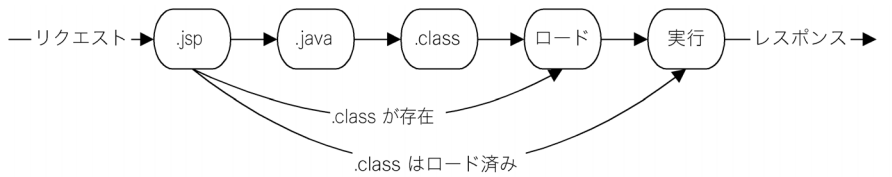
JSP はテンプレートデータ (通常、テキストと HTML タグ) と JSP 要素の組み合わせです。JSP 要素は、サーブレットに変換されて Web サーバーで実行されます。JSP 仕様では、次の 3 種類の JSP 要素が定義されています。

- **ディレクティブ**要素 JSP ファイルからサーブレットを作成するためのプロパティを表します。
- **スクリプト**要素。オブジェクトを操作し、計算を実行します。
- **アクション**。オブジェクトの使用、修正、または作成や、ページの出力ストリームへの書き込みを行います。

JSP ライフサイクル

JSP をサーブレットに変換する処理は、ページ変換と呼ばれています。最初に JSP がリクエストされると、JRun はファイルを解析して、Java ソースコードファイルを出力します。Java ソースコードファイルはサーブレットクラスファイルにコンパイルされます。次に、そのサーブレットクラスファイルがロードされ、実行されます。

次の図は、JRun が JSP のリクエストを受け取ったときに実行する手順を示しています。



次の手順は、JSP がリクエストされたときに JRun が実行するアクションを示しています。

1 JRun は JSP (.jsp ファイル) を解析して、Java ソースコード (.java ファイル) を作成します。

2 JRun は Java ソースコードをサーブレットクラス (.class ファイル) にコンパイルします。

3 JRun はサーブレットの .class ファイルをメモリにロードします。

4 クライアントがサーブレットをマッピングするようにリクエストすると、JRun はそのサーブレットを実行します。

JRun はサーブレットからの出力をクライアントに返します。JSP のデフォルト出力は、`text/html; charset=ISO-8859-1` の MIME タイプです。このタイプでは、クライアントに直接送信できるように出力を設定します。

これ以後 JSP がリクエストされたときに、JSP が最後に解析されて以降修正されていない場合は、JRun は手順 4 だけを実行します。これは、最初のリクエストの後、サーブレットがメモリに残っているからです。デフォルトでは、JSP ファイルが最後に解析された後で修正された場合、JRun は再コンパイルします。

JSPC コンパイラ使用の詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

JSP .java ファイルの保存

JSP ページを表すサーブレットのソースコードを表示することも可能です。JSP ページの XML 表現は、JRun がサーブレットを作成する間一時的にメモリ内に維持されるだけなので、表示できません。

デフォルトでは、JRun は、JSP のソースコードから生成されたサーブレットを保持できません。このセクションでは、Java ファイルを保持するように JRun を設定し、それらを表示する方法について説明します。

生成されたファイルを保持する方法については、57 ページの「サーブレットのソースコードの表示」を参照してください。

JSP の保管

JSP を Web サーバーのドキュメントのルートディレクトリに保管できます。たとえば、IIS に接続されている JRun を使用している場合、デフォルトでは、このディレクトリは c:¥Inetpub¥wwwroot です。デフォルトの JRun サーバーとして JRun Web Server を使用している場合、ディレクトリは <jrun のルート ディレクトリ > ¥servers¥default-ear¥default-war¥ です。

ただし、異なる JSP 保管方法が使用されている可能性もあります。たとえば、JRun を IIS に接続しながら、WAR ファイル内のアプリケーションルートディレクトリに JSP を保管することができます。

変数の宣言

他のプログラム言語と同様に、JSP でも変数を宣言できます。次の例に示すように、変数を定義でき、またその割り当てを変更できます。

```
<html><head><title> 変数の使用 </title></head><body>
<% int myVar = 5; %>
<b> <% out.println ("Value of myVar:" + myVar); %> </b>
<p>
<%
  myVar = 2;
  out.println("Value of myVar again:" + myVar);
%>
</body></html>
```

myVar 変数へのアクセスは、この変数を宣言した JSP 内でのみ可能です。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

ページ内で変数を再度割り当てることはできますが、名前自体は 1 回しか宣言できません。次の例で、変数の不適切な使用方法を示します。この例では、myVar が最初の Java ブロック内で int として宣言され、3 番めの Java ブロックで再び int として宣言されているため、コンパイラエラーが発生します。

```
<html><head><title> 変数の使用 </title></head><body>
<% int myVar = 5; %>
<b> <% out.println ("Value of myVar:" + myVar); %> </b><P>
```

```
<%
  int myVar = 2;  // ここに間違いがあります
  out.println("Value of myVar again:" + myVar);
%>
</body></html>
```

JSP への条件ロジックの追加

スクリプトレット内で **if** ステートメントを使用すると、JSP で HTML を条件付きで出力できます。次の例は、JSP の条件ステートメントを示しています。

```
<html><head><title> 口座の残高 </title></head><body>
<% double accountBalance = 1.00; %>
<P> お客様の口座の残高：<% out.println( accountBalance ); %> <br>
<% if(accountBalance <= 1.00) { %>
  <b> 仕事を探してください。</b> <br>
<% } %>
</body></html>
```

この例は、変数 accountBalance の値を出力します。accountBalance が 1 ドル以下の場合は、次のステートメントで「仕事を探してください」と提案します。accountBalance を増減させることによって、ステートメントを修正できます。

条件ステートメントは、ブロック **<% } %>** を使用して **if** ステートメントを閉じます。このシンタックスで HTML で「** 仕事を探してください。
**」と表示されるのは、条件が true の場合です。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

パラメータと属性の使用

多くの JSP オブジェクトはメソッドを含んでおり、パラメータや属性としてオブジェクト内に保管されているデータにアクセスします。JSP オブジェクトを使用するときは、パラメータを使用する場合と属性を使用する場合を知っている必要があります。

パラメータ

パラメータは、常に String として JSP オブジェクト内に保管されます。パラメータの主な用途は、クライアントのリクエストの中でクライアントからサーバーにデータを渡すことです。

たとえば、クライアントがフォームを送信するときは、すべてのフォームデータが request オブジェクト内の名前 / 値のペアとしてサーバーに送信されます。名前はパラメータ名に対応します。値はパラメータ値を含む文字列です。JSP 内では、request オブジェクトの getParameter メソッドを使用してパラメータにアクセスします。

次の例では、request オブジェクトを使用して JSP への HTTP リクエストに含まれている 2 つのパラメータ fName および lName の値を取得し、次に out オブジェクトを使用してこれらの値をクライアントに渡します。

```
<%
  String firstName = request.getParameter("fName");
  String lastName = request.getParameter("lName");
  out.println("Welcome " + firstName + " " + lastName);
%>
```

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

属性

通常は、JSP、JavaBean、サーブレットなどのサーバーサイドコンポーネント間で情報をやり取りするために使用します。たとえば、ある JSP から別の JSP を呼び出す場合、呼び出し側の JSP は、request オブジェクトや session オブジェクト内の属性として、情報をデスティネーションページに渡すことができます。

属性は名前 / 値のペアとして保管されます。名前は属性名に対応し、値は `java.lang.Object` のインスタンスとして保管されます。これがパラメータと属性の主な違いです。パラメータは常に String として保管され、属性は Java オブジェクトとして保管されます。

たとえば、JSP の session オブジェクトに属性 `fName` と属性 `lName` が含まれる場合、次のコードを使用するとそれらの属性にアクセスできます。

```
<%
  String firstName = (String) session.getAttribute("fName");
  String lastName = (String) session.getAttribute("lName");
  out.println("Welcome " + firstName + " " + lastName);
%>
```

この例では、`getAttribute` の戻り値を String にキャストします。このキャストが必要になるのは、`getAttribute` が常に `java.lang.Object` のタイプのオブジェクトを返すからです。キャストは、返されたオブジェクトをデスティネーションの形式 (この場合は String) に変換します。

属性を使用すると、サーバーサイドのアプリケーションをより柔軟に開発できます。なぜなら、パラメータを使用する場合と違って、String 以外のオブジェクトを保管および取得できるからです。属性を使用すると、どのようなタイプのオブジェクトでも保管および取得でき、それらのオブジェクトをアプリケーションのコンポーネントに渡すことができます。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

相対 URL の指定

JSP 要素は相対 URL を使用して、JSP、サーブレット、ページ内のその他の Web コンポーネントを参照できます。参照元 JSP 内で URL を指定する方法に応じて、参照元 JSP を含んでいるアプリケーションまたは参照元 JSP のロケーションに対して相対的に指定できます。

たとえば、myErrorPage.jsp という URL では、参照元 JSP のロケーションに対して相対的な myErrorPage.jsp が参照されます。この場合、JRun は、参照元ページと同じディレクトリにある myErrorPage.jsp を検索します。

たとえば、../myErrorPage.jsp という URL では、参照元 JSP のロケーションに対して相対的な myErrorPage.jsp が参照されます。この場合、JRun は、参照元ページの親ディレクトリにある myErrorPage.jsp を検索します。

/errorPages/myErrorpage.jsp という URL には、リファレンスの先頭にスラッシュがあります。リファレンスの先頭に "/" を付けると、JRun は、参照元 JSP を含んでいるアプリケーションのルートに対して相対的に myErrorpage.jsp を検索します。JSP はすべて、1 つのアプリケーションに含まれます。アプリケーションを設定する手順の一部として、/ にマッピングされるディレクトリを定義する必要があります。

別の JSP の呼び出し

JSP から別の JSP を呼び出すことによって、複雑なアプリケーションの作成に使用できるモジュラー JSP を開発できます。ある JSP を別の JSP から呼び出すときは、JSP で次のアクションのいずれかを使用します。

- **jsp:include**　デスティネーションページを完全に実行してから、呼び出し側ページに制御を戻します。
- **jsp:forward**　デスティネーションページに制御を渡して終了します。この場合、デスティネーションページは呼び出し側ページに制御を戻しません。

`jsp:include` および `jsp:forward` アクションの詳細については、263 ページの「アクションについて」を参照してください。

メモ：デスティネーションページをロードした後に変更した場合、JRun はそのページが呼び出されたときにそのページを再変換します。ただし、デスティネーションページが修正された場合、JRun は呼び出し側 JSP を再コンパイルしません。

デフォルトでは、JRun は JSP からクライアントに送信された出力データをバッファーに入れます。`jsp:include` アクションと `jsp:forward` アクションのいずれかを呼び出すと、JRun によって JSP の出力バッファーがフラッシュされます。出力バッファーの詳細については、252 ページの「JSP 出力のバッファリング」を参照してください。

JSP 出力のバッファリング

JRun は JSP からクライアントに送信される出力データをバッファリングします。バッファーを使用しているため、レスポンスヘッダー情報およびその他の出力はバッファーがフラッシュされるまではクライアントに送信されません。このフラッシュは、次のいずれかが発生したときに行われます。

- バッファーが満杯になったとき。
- out オブジェクトの `flush` を呼び出したとき。
- `flush` 属性が true になっており (デフォルト)、`jsp:include` アクションを使用して他の JSP を呼び出したとき。
- `jsp:forward` アクションを呼び出した。
- response オブジェクトの `redirect` メソッドを使用してリクエストを転送したとき。

リクエストを転送する場合、転送元 JSP が設定した Cookie は破棄されず、クライアントに送信されます。バッファリングを無効にしていてもリクエストを転送できますが、それはクライアントにまだ何も出力を書き込んでいない場合に限られます。

バッファリングの影響より、HTTP ヘッダーに依存するオペレーションは、flush メソッドが実行されヘッダーがクライアントに送信されるまで無効です。

`page` ディレクティブを使用してバッファーサイズを設定し、ページのバッファリングを有効または無効にします。詳細については、257 ページの「page ディレクティブ」を参照してください。

タグライブラリの使用

JSP には、JSP タグと、一般的には HTML であるテンプレートテキストが含まれ、また、オプションとしてタグライブラリにあるカスタムタグが含まれます。タグライブラリを使用すれば、タグ開発者は、自分の JSP、社内の他のユーザー、または顧客が使用するカスタムタグを実装することによって、使用できるタグのセットを拡張できます。

JSP で使用するタグライブラリを宣言するには、JSP 内で `taglib` ディレクティブを使用します。`taglib` ディレクティブの一部として、タグライブラリへのパスと、ページ内で使用するタグ接頭辞を指定して、ライブラリを識別します。たとえば、次のステートメントでは、接頭辞 `myTags` によって参照されるタグライブラリが定義されます。

```
<%@ taglib uri="/myApp/appTags" prefix="myTags" />
```

`taglib` ディレクティブの後で、接頭辞 `myTags` を使用してタグライブラリ内のタグを参照できます。次のステートメントでは、タグライブラリ内の `coolTag` を使用します。

```
<myTags:coolTag>
...
</myTags:coolTag>
```

タグライブラリが見つからない場合、JRun では致命的な変換エラーが発生します。JSP 内で接頭辞 `myTag` を使用する別のタグライブラリを定義した場合もエラーが発生します。

`taglib` ディレクティブ使用の詳細については、261 ページの「taglib ディレクティブ」を参照してください。

独自のカスタムタグライブラリ作成の詳細については、285 ページの第 11 章「Java のカスタムタグ」を参照してください。

JSP の基本シンタックス

次の表で、JSP を作成するときのシンタックスの基本的な注意点を説明します。

シンタックス	説明
テンプレートテキスト	JSP では、テンプレートテキストは JSP 要素の外側にあるので、JRun によって解釈されることはありません。テンプレートテキストは、修正されずにクライアントに直接返されます。JSP の HTML テキストは、テンプレートテキストと見なされます。
空白文字	JSP ファイルでは、テンプレートコードに含まれる空白文字はすべて、JSP ファイルに入力されているとおりにクライアントに返されます。
開始タグと終了タグ	開始タグと終了タグ、そしてこれらのタグに囲まれた本文を持つ JSP 要素。これらのタグは両方とも同じファイル内に入れる必要があります。開始タグと終了タグを別々のファイル内に入れることはできません。 たとえば、JSP スクリプトレットのシンタックスは <% *scriptlet* %> です。開始タグ (<%) と終了タグ (%>) は、両方とも同じファイル内になければなりません。

シンタックス	説明
属性値	すべての JSP 要素の属性値を引用符や二重引用符で囲む必要があります。たとえば、次の例の **page** ディレクティブは、**contentType** を **text/plain** に設定します。 `<%@ page contentType = "text/plain" %>` 属性値自体に同じタイプの引用符 (引用符や二重引用符) が含まれている場合は、埋め込まれた引用符の前にエスケープ文字 (¥) を付けます。エスケープ文字の次に来る文字は、JSP パーサーでは無視されます。属性の引用符には、次のエスケープシーケンスを使用します。 • ' は ¥' とエスケープします。 • " は ¥" とエスケープします。 また、引用符に HTML 文字エンティティを使用できます。たとえば、二重引用符の代わりとして属性値に文字リファレンス **"** を挿入します。HTML 文字エンティティ使用の詳細については、42 ページの「HTML エンティティの使用」を参照してください。
文字のエスケープ	属性値の中で引用符をエスケープできるだけでなく、さらに、JSP の他の領域にある次のような文字をエスケープできます。 • リテラルの **%>** は **%¥>** でエスケープします。 • リテラルの **<%** は **<¥%** でエスケープします。

コメント	JSP で使用できるコメントには 2 種類あります。
	• **コンテンツコメント** JSP ページの出力に含まれます。ユーザーは、ページのソースを表示すると、ブラウザでこのコメントを読むことができます。コンテンツコメントのシンタックスは次のとおりです。
	`<!-- クライアントが表示可能なコメント -->`
	• **JSP コメント** JSP ページの出力に含まれません。これらのコメントが表示されるのは、JSP のソースコードや JRun が生成するサーブレットの中だけです。ユーザーは、ページのソースを表示しても、このタイプのコメントを見ることはできません。JSP コメントのシンタックスは次のとおりです。
	`<%-- ソースコードにだけ存在するコメント %>`
	
	また、スクリプトレットに挿入したコメントもクライアントには表示されません。たとえば次のとおりです。
	`<%`
	`  /* 次のコードでユーザー ID を取得します */`
	`  String sqlstmt = "SELECT userID FROM users WHERE _`
	`  lastname = request.getParameter("lastname")";`
	`%>`

スクリプト要素について

スクリプト要素によって、JSP に含まれるコードが定義されます。コードは、Java 言語シンタックスを使用して書く必要があります。

次の表で、JRun でサポートされている 3 つのスクリプト要素を示します。

要素	目的	シンタックス
宣言	変数などの、ページ全体で使用される定義を作成します。	*<%! declaration %>*
スクリプトレット	このページで使用されるスクリプトコードが含まれます。	*<% script code %>*
式	ページ出力をクライアントに送信する前にサーバーで評価されるステートメントを定義します。	*<%= expression %>*

宣言

宣言により、1 つの JSP 内のページ全体にわたる定義を行うことができます。一般的に、宣言を使用して、JSP で使用する変数やメソッドを定義します。宣言をしても、クライアントに出力が書き込まれることはありません。

宣言のシンタックスは次のとおりです。

<%! declaration(s) %>

次の宣言では変数と関数が定義されます。

```
<%!
  private String foo = "42";
  public String getFoo() { return this.foo; }
%>
```

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

スクリプトレット

scriptlet 要素は、JSP で使用されるスクリプトコードを指定します。有効なコードをスクリプトレット要素の本文内で指定できます。

スクリプトレットを使用して、JSP の出力ストリームにデータを書き込むことができます。その後、この情報は HTTP レスポンスとともにクライアントに返されます。通常、このデータは HTML テキスト形式で記述されています。

スクリプトレット内のコードは、application、session、request、response、out などの、JSP 用に定義された暗黙的オブジェクトに完全にアクセスできます。

スクリプトレットは Java 言語シンタックスで書く必要があります。スクリプトレットのシンタックスは次のとおりです。

```
<% script code %>
```

Java コードが組み込まれた例は次のとおりです。

```
<%
  String greeting = request.getParameter("Greeting");
  out.println(greeting);
%>
```

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

式

式は、ページ出力をクライアントに送信する前にサーバーで評価されるステートメントです。式の結果のデータタイプは String です。

式から、application、session などの、JSP 用に定義された暗黙的オブジェクトに完全にアクセスできます。

式のシンタックスは次のとおりです。

```
<%= expression %>
```

式の要素を使用した例は次のとおりです。

```
<html><head><title> 口座の残高 </title></head><body>
<% double accountBalance =1.00; %>
<P> お客様の口座の残高：<%= accountBalance %> <br>
<% if(accountBalance <= 1.00) { %>
  <b> 仕事を探してください。</b> <br>
<% } %>
</body></html>
```

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

ディレクティブについて

ディレクティブで、JSP と結果として求められるサーブレットのプロパティを設定します。ディレクティブは、コンパイル時の JSP 解析中に評価される前処理要素です。

ディレクティブによって定義される情報の例には、JSP の出力 MIME タイプ、ページが使用するタグライブラリ、ページに含まれるクラスがあります。

すべての JSP ディレクティブの基本シンタックスは次の形式です。

```
<%@ directive %>
```

JSP では、次の表に示す 3 種類のディレクティブがサポートされています。次のセクションでは、これらのディレクティブについて説明します。

ディレクティブ	目的	シンタックス
page	ページ全体にわたる属性を定義します。	<%@ page *attribute*="*value*" ... %>
include	JSP にテキストを挿入します。	<%@ include file = "*path*" ... %>
taglib	JSP にタグライブラリを含めます。	<%@ taglib uri="*tagLibraryURI*" prefix="*tagPrefix*" %>

式は実行時に評価されるため、ディレクティブ内では使用できません。したがって、次のディレクティブの例は、式 <%= myVar%> をコンパイル時に評価できないので、正しくありません。

```
<%@ page import="<%= myVar%>" %>
```

さらに、ディレクティブは解析時に評価され、何も出力しないので、ディレクティブを使用してクライアントに情報を返すことはできません。

次のセクションでは、JSP がサポートするディレクティブを説明します。

page ディレクティブ

page ディレクティブでは、JSP 全体に対して 1 つ以上の属性を定義できます。

page ディレクティブのシンタックスは次のとおりです。

```
<%@ page attribute = "value" ... %>
```

ここで、

```
attribute = language | import | contentType | session | buffer |
        autoflush | isThreadSafe | info | errorPage | isErrorpage |
        extends
```

value は、引用符または二重引用符で囲まれた文字列リテラル。

1 つの JSP で複数の page ディレクティブを使用できます。ただし、import 属性を使用している場合を除き、1 つの属性は 1 回しか参照できません。複数の page ディレクティブを指定したために属性が重複すると、JRun は JSP 内の page ディレクティブの最初のインスタンスだけを認識します。つまり JRun では、最初の属性以上の定義は無視されます。

import 属性を含む page ディレクティブは複数指定できます。コンパイルされたページでは、import 属性で参照されたファイルがすべてインポートされます。

たとえば、次のディレクティブでは JSP の出力 MIME タイプは HTML に、スクリプト言語は Java に設定されます。

```
<%@  page contentType="text/html" language = "java" %>
```

(language 属性を定義する場合は、java に設定する必要があります。)

次の表で、page ディレクティブの属性を説明します。

属性	説明
language	ファイルで使用されるスクリプト言語を定義します。デフォルトは Java ですが、Java 以外の値は使用できません。JRun は、JSP での JavaScript の使用をサポートしません。

import	コンパイルされたページがインポートするパッケージを、カンマ区切りリストで指定します。次に例を示します。
	`<%@  page import = "java.io.*,java.util.Hashtable" %>`
	
	JRun は次のクラスを暗黙的にインポートします。
	`java.lang.*`
	`javax.servlet.*`
	`javax.servlet.jsp.*`
	`javax.servlet.http.*`

contentType	MIME タイプを定義します。またオプションで、JSP によって生成されるレスポンスの文字セットを定義します。デフォルトの MIME タイプは text/html で、デフォルトの文字セットは ISO-8859-1 です。
	この属性のシンタックスは次のとおりです。
	`<%@  page contentType = "TYPE; charset = CHARSET" %>`
	
	TYPE では、出力 MIME タイプを指定します。またオプションの CHARSET で、文字セットを指定します。次に例を示します。
	`<%@  page contentType = "text/plain" %>`
	
	JSP の文字セット変更の詳細については、31 ページの第 3 章「国際化対応とローカリゼーション」を参照してください。

session	ページがセッションの一部としてアクセスされるかどうかを指定します。 true に設定すると、ページの JSP session オブジェクトが初期化されます。 false に設定すると、ページがセッションの一部ではなく、session オブジェクトを使用できないことが指定されます。 デフォルトは true です。

属性	説明
buffer	ページ出力で使用されるバッファリングモデルを指定します。サイズ指定に文字列 kb を入れる必要があります。たとえば、次のディレクティブはバッファーサイズを 16 キロバイトに設定します。 `<%@ page buffer = "16kb" %>` none に設定すると、自動バッファリングが無効になります。つまり、ページ出力はすべてクライアントに直接書き込まれます。 バッファーサイズを指定した場合、出力は指定されたサイズ以下のサイズでバッファリングされます。**autoFlush** 属性の値に応じて、バッファーの内容が自動的にフラッシュされるか、または、オーバーフローが発生したときに例外が発生します。デフォルト値は 8 キロバイトです。 バッファーサイズ設定の詳細については、223 ページの「レスポンスオブジェクトのバッファーサイズの拡張」を参照してください。
autoFlush	true に設定すると、バッファーが一杯になったときに出力が自動的にフラッシュされます。 false に設定すると、バッファーが一杯になったときに例外が発生します。 デフォルトは true です。**buffer** 属性を none に設定している場合は、**autoFlush** を false に設定できません。
isThreadSafe	ページごとに実装されるスレッドセーフのレベルを指定します。 false に設定すると、複数のリクエストを処理するためにページのインスタンスが複数作成されます。これらのページインスタンスはそれぞれ 1 つのスレッドを持ちます。 true に設定すると、複数の未解決のクライアントリクエストが、このページに同時にディスパッチされます。これによって、JSP は、ページの共有ステートへのアクセスを正しく同期させることができます。 デフォルトは true です。
info	JSP によって実行時に使用される文字列を指定します。この文字列へは、**application.getServletInfo** メソッドを使用してアクセスできます。
isErrorPage	別の JSP でキャッチされない例外をこの JSP で処理するように指定します。 true に設定すると、JSP exception オブジェクトがインスタンス化されます。JRun は、例外発生元 JSP からのエラーに基づいて、exception オブジェクトを定義します。 false に設定すると、この exception オブジェクトを使用できなくなります。JSP の本文からこのオブジェクトを参照すると、致命的な変換エラーが発生します。 デフォルト値は false です。 JSP のエラーキャッチメカニズムの詳細については、281 ページの「エラーの処理」を参照してください。

属性	説明
errorPage	ページでキャッチされなかった例外がエラー処理のために送信される JSP への URL を指定します。デスティネーションページは、page ディレクティブの isErrorPage 属性を true に設定する必要があります。 呼び出されると、デスティネーション JSP の exception オブジェクトに例外へのリファレンスが入ります。 autoFlush を true に設定し、例外を返したページからの出力データがすべてフラッシュされた場合は、エラーページへ例外を返そうとしても、失敗する可能性があります。 JSP のエラーキャッチメカニズムの詳細については、281 ページの「エラーの処理」を参照してください。
extends	JSP がサブクラス化する JSP ベースクラスを指定します。次に例を示します。 <%@ page extends = "com.myPackage.AServletImplementation" %> デフォルトでは、サーブレットは JRun で javax.servlet.http.HttpServlet クラスのサブクラスとして作成されます。したがって、サーブレットのベースクラスを再定義する特定の理由がある場合にかぎり、この属性を使用します。
pageEncoding	JSP をコンパイルするときに使用するエンコーディングタイプを指定します。 次の例は、日本語文字セットでページをエンコードします。 <% page contentType="text/html; charset=UTF-8" pageEncoding="EUC-JP" %> ページのエンコーディングは、エディタや IDE でページを保存するときに使用したエンコーディングタイプに一致しなければなりません。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

include ディレクティブ

include ディレクティブは、変換時にテキストを JSP ファイルに挿入します。このディレクティブによって、挿入元 JSP のコンテンツと挿入先 JSP のコンテンツの両方が 1 つの JSP に入るようになります。

include ディレクティブのシンタックスは次のとおりです。

<%@ include file = "*path*" %>

JRun は、指定されたファイルのコンテンツを、このディレクティブのある位置で JSP ファイルに挿入します。

path の先頭がスラッシュ (/) の場合、パスは JSP のアプリケーションに相対的になります。パスの先頭がスラッシュでない場合、JRun は、パスが、変換中の JSP のパスに相対的であると見なします。パスの詳細については、251 ページの「相対 URL の指定」を参照してください。

JSP で include ディレクティブを使用すると、JSP が 1 つ作成されます。これは、1 つの JSP を別の JSP にインクルードした場合でも同様です。JRun は 1 つの JSP とそれに対応する 1 つのサーブレットを作成します。このサーブレットは、元の 2 つの JSP のコンテンツを含んでいます。

JRun は、インクルードされたすべてのファイルについて、実行時に依存チェックを実行します。インクルードしたファイルがメモリにロードされた後で、インクルードされたファイルが変更された場合、インクルードしたファイルは次のリクエスト時に再変換されます。

次の例で、ヘッダー情報を含む別の JSP をインクルードします。

```
<html><%@ include file="my_header.jsp" %>
<%-- 残りの JSP --%>
...
</body></html>
```

my_header.jsp ファイルには次の情報が含まれています。

```
<head><title> ご挨拶 </title></head><body>
<center><table width=80%><tr>
<td><img src="logo.gif" width=200 height=21 alt=" インクルードのテスト "
        border="0"><P></td>
<td><H1><font color="#336699"> ご挨拶 </font></h1></td>
</tr></table></center>
```

JSP request オブジェクトを使用して、インクルードされた JSP に情報を渡すことができます。たとえば、次のように、入力としてページのタイトルを定義する属性を取るヘッダー JSP を作成できます。

```
<html><%request.setAttribute("title", " ご挨拶 "); %>
<%@ include file="my_header.jsp" %>
<%-- 残りの JSP --%>
...
</body></html>
```

こうするとヘッダーファイルは、次のようにしてこの属性にアクセスできます。

```
<html><head><title><%= request.getAttribute("title")%></title>...
```

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

taglib ディレクティブ

taglib ディレクティブを使用して、JSP によってインクルードされるタグライブラリを宣言します。タグライブラリには JSP で使用可能なカスタムタグが含まれています。

taglib ディレクティブのシンタックスは次のとおりです。

```
<%@ taglib uri="path" prefix="tagPrefix" %>
```

次の例で、タグ coolTag を含むタグライブラリを定義します。このディレクティブの後でこのタグライブラリにあるタグを参照するには、接頭辞 myTags を使用します。接頭辞 myTags を使用する JSP に別のタグライブラリを定義すると、エラーが発生します。

```
<%@ taglib uri="myApp/appTags" prefix="myTags" />
<myTags:coolTag>
...
</myTags:coolTag>
```

次の表で、taglib ディレクティブの属性を説明します。

属性	説明
uri	タグライブラリの場所を、相対パス位置、または web.xml ファイルへのルックアップキーとして指定します。JRun は、まず web.xml ファイルをチェックし、path がルックアップキーであるかどうかを判断します。 uri が web.xml ファイルへのルックアップキーである場合は、JRun によって、web.xml ファイル内のキーと、関連するタグライブラリが検索されます。たとえば、アプリケーションの web.xml ファイルで次のように指定すると、myTagLib と名付けられたルックアップキーと、タグライブラリの関連する位置が定義されます。 <taglib> <taglib-uri>myTagLib</taglib-uri> <taglib-location>/WEB-INF/tlibs/myTagLib.tld</ taglib-location> </taglib> uri がタグライブラリを識別する相対パスで、スラッシュ (/) で始まっている場合、この uri は JSP のアプリケーションに関連付けられています。uri がスラッシュで始まっていない場合、このパスは 変換されている JSP に関連付けられているパスと見なされます。 詳細については、251 ページの「相対 URL の指定」を参照してください。
prefix	ライブラリ内のカスタムタグを識別する接頭辞文字列を定義します。接頭辞 jsp、jspx、java、javax、servlet、sun、sunw は予約されています。空の接頭辞は使用できません。

独自のカスタムタグライブラリ作成の詳細については、285 ページの第 11 章「Java のカスタムタグ」を参照してください。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

アクションについて

アクションで、オブジェクトの使用、修正、または作成や、ページの出力ストリームの修正を行います。このセクションでは、JRun でサポートされているすべてのアクションについて説明します。次の表で、JSP のアクションを説明します。

アクション	目的
jsp:useBean	JavaBeans のインスタンスを定義します。
jsp:setProperty	bean にある 1 つ以上のプロパティ値を設定します。
jsp:getProperty	bean プロパティの値を、文字列として out オブジェクトに自動的に書き込みます。
jsp:include	ある JSP から別の JSP を呼び出します。呼び出しが完了すると、デスティネーションページから呼び出し側ページに制御が戻ります。
jsp:forward	ある JSP から別の JSP を呼び出します。この呼び出しにより、呼び出し側ページの実行が終了します。
jsp:param	ある JSP から別の JSP にリクエストを転送するときに、名前 / 値のペアとしてパラメータを HTTP リクエストに追加します。
jsp:plugin	クライアントのブラウザでアプレットまたは他のプラグ可能コンポーネントを呼び出します。

jsp:useBean

jsp:useBean アクションにより、JSP で JavaBean がインスタンス化されます。インスタンス化が完了すると、JSP ファイル内の bean を参照できるようになります。次の例では、タイプ com.myco.myapp.MyBean の myBean と名付けられた bean を定義します。

```
<jsp:useBean id="myBean" class="com.myco.myapp.MyBean" />
```

jsp:useBean アクションのシンタックスは次のとおりです。

```
<jsp:useBean id="name" scope="page|request|session|application"
        typeSpec />
```

ここで、typespec には次のいずれかが入ります。

```
class="className" |
class="className" type="typeName" |
beanName="beanName" type=" typeName" |
type="typeName"
```

type または class を指定する必要があります。class を指定した場合は、beanName は指定できません。type と class の両方を指定する場合は、class を type に割り当てる必要があります。

属性 beanName は a.b.c の形式で表された bean の名前です。これには、a/b/c.ser の形式でクラスまたはリソース名を指定できます。

次のシンタックスを使用して、本文を jsp:useBean アクションに指定できます。

```
<jsp:useBean id="name" scope="page|request|session|application"
        typeSpec >
  body
</jsp:useBean>
```

JRun によって、bean の本文が呼び出されます。一般的に、本文には、新規作成された bean を修正するスクリプトレットまたは jsp:setProperty アクションが含まれます。

次の表で、jsp:useBean アクションの属性を説明します。

属性	説明
id	指定されたスコープで bean を識別するために使用される名前と、bean のスクリプト変数名を指定します。指定する名前では大文字と小文字を区別し、Java の変数のネーミング規則に準拠する必要があります。
scope	オプションで、bean を使用できるスコープを定義します。次のスコープを指定できます。 • **page** この bean は現在のページで使用できます。これがデフォルトです。 • **request** この bean は、getAttribute メソッドを使用して、現在のページにある request オブジェクトから使用できます。現在のクライアントのリクエストが完了すると、リファレンスは破棄されます。 • **session** この bean は、getValue メソッドを使用して、現在のページにある session オブジェクトから使用できます。現在のセッションが無効になると、リファレンスは破棄されます。 page ディレクティブの session 属性を false に設定している場合は、scope を session に設定できません。 • **application** この bean は、getAttribute メソッドを使用して、現在のページにある application オブジェクトから使用できます。
class	bean の実装を定義する認識可能なクラス名を設定します。クラス名では大文字と小文字が区別されます。 class 属性と beanName 属性を省略する場合、オブジェクトは、指定されたスコープ内にある必要があります。

属性	説明
beanName	java.beans.Beans クラスの instantiate メソッドで認識されるように bean の名前を指定します。 beanName 属性は実行時の式を利用できます。
type	指定されている場合、type はスクリプト変数のタイプを表します。bean は指定されたタイプへのインスタンスである必要があります。 この属性を使用すると、スクリプト変数のタイプを、指定された実装クラスのスクリプト変数に関連付けながら区別することができます。このタイプはクラス自体、クラスのスーパークラス、または指定されたクラスによって実装されたインターフェイスでなければなりません。 type 属性は、デフォルトで class 属性の値になります。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

jsp:setProperty

jsp:setProperty アクションは、bean の 1 つ以上のプロパティの値を設定します。このアクションを使用する前に、jsp:useBean を使用して bean を定義しておく必要があります。

jsp:setProperty アクションのシンタックスは次のとおりです。

```
<jsp:setProperty name="beanName" prop_expr />
```

prop_expr 変数は次の形式のいずれかです。

```
property="*" |
property="propertyName"|
property="propertyName" param="parameterName"|
property="propertyName" value="propertyValue"
```

propertyValue 変数は、リテラル文字列または式である必要があります。

次の例では、user と名付けられた bean のプロパティに値を設定します。

```
<jsp:setProperty name="user" property="user" param="username" />
```

次の例では、式を使用してプロパティを設定します。

```
<jsp:setProperty name="results" property="row" value="<%= i + 1 %>" />
```

単純なインデックス付きプロパティは、setProperty を使用して設定できます。インデックス付きプロパティの値には、配列を割り当てる必要があります。

次の表で、jsp:setProperty アクションの属性を説明します。

属性	説明
name	jsp:useBean アクションまたは他の要素によって定義された bean の名前。bean のインスタンスには、設定するプロパティが含まれている必要があります。jsp:useBean アクションは、同じファイル内の jsp:setProperty アクションも前に呼び出す必要があります。
property	設定する bean のプロパティの名前。 *propertyName* を * に設定すると、現在のリクエストパラメータに対して jsp:setProperty が繰り返し実行され、パラメータ名と値タイプが bean のプロパティ名とタイプと比較されます。一致したプロパティはそれぞれ、対応するパラメータの値に設定されます。パラメータの値が空の文字列 ("") の場合、対応するプロパティは修正されません。 前に示した *prop_expr* の先頭から 3 つのフォームはそれぞれ、文字列として表される値を bean のプロパティに割り当てます。ただし、bean のプロパティのデータタイプが String 以外の場合は、JRun によりタイプ変換が行われます。 前に示した 4 番めの形式 ***prop_expr*** は、オブジェクトを bean のプロパティに割り当てます。この場合、オブジェクトは自動的に割り当て先 bean プロパティのデータタイプに変換されます。
param	リクエストパラメータの名前。パラメータの値を bean プロパティに渡します。アクションは param 属性と value 属性を持つことはできません。 param を省略した場合、JRun は、リクエストパラメータ名と bean プロパティ名が同じであると見なします。 param が request オブジェクトに設定されていない場合、または値が空の文字列 ("") の場合、jsp:setProperty 要素は機能しません。
value	プロパティに割り当てられる値。アクションは param 属性と value 属性を持つことはできません。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

jsp:getProperty

jsp:getProperty アクションは、bean プロパティの値を文字列として out オブジェクトに書き込みます。このアクションを使用する前に、bean を定義しておく必要があります。

jsp:getProperty アクションのシンタックスは次のとおりです。

```
<jsp:getProperty name="name" property="propertyName" />
```

次の例では、user bean の name プロパティを書き込みます。

```
<jsp:getProperty name="user" property="name" />
```

次の表で、jsp:getProperty アクションの属性を説明します。

属性	説明
name	このプロパティで使用される bean インスタンス名を指定します。このアクションは、bean が見つからなかった場合に例外を生成します。
property	出力する値のプロパティの名前

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

jsp:include

jsp:include アクションは、スタティックおよびダイナミックなリソースを現在のページに挿入します。挿入が完了すると、呼び出し側 JSP で処理が再開されます。

jsp:include アクションのシンタックスには、次のいずれかの形式を使用します。

```
<jsp:include page="path" flush="true|false"/>
```

または

```
<jsp:include page="path" flush="true|false">
  <jsp:param name="paramName" value="paramValue" /> ...
</jsp:include>
```

デフォルトでは、JSP からクライアントに送信される出力データはバッファリングされます。バッファーを使用しているため、レスポンスヘッダー情報およびその他の出力はバッファーがフラッシュされるまではクライアントに送信されません。インクルードするページの出力がバッファリングされている場合、バッファーはインクルード前にフラッシュされます。このフラッシュのために、インクルードされるページはレスポンスヘッダーを設定できません。したがって、インクルードされたページでは setCookie などのメソッドを使用できません。デフォルトは false です。

メモ: バッファーを無効にするか、または JSP の出力バッファーのサイズを設定するには、JSP ディレクティブを使用します。page ディレクティブの詳細については、257 ページの「page ディレクティブ」を参照してください。

次の例では、HTML ページをインクルードします。

```
<jsp:include page="/templates/copyright.html"/>
```

次の表で jsp:include アクションの属性を説明します。

属性	説明
page	インクルードするファイルのパスを指定します。 パスの詳細については、251 ページの「相対 URL の指定」を参照してください。
flush	true に設定すると、このインクルードの前にバッファーがフラッシュされます。デフォルト値は false です。

jsp:include は、jsp:param アクションを取ることができます。このアクションにより、目的の JSP により受信される HTTP リクエストにパラメータを追加できます。詳細については、270 ページの「jsp:param」を参照してください。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

jsp:forward

jsp:forward アクションによって、現在のページと同じアプリケーションで JSP やサーブレットが呼び出されます。jsp:forward によって、現在の JSP の実行が終了されます。

JRun でページ出力がバッファリングされている場合は、jsp:forward アクションが呼び出されるか、または response オブジェクトの redirect メソッドを使用してリクエストが転送されたときに、JRun によってバッファーがクリアされます。JRun は、転送元 JSP によって設定された Cookie を破棄せずに、クライアントに送信します。

JRun でページ出力がバッファリングされておらず、何らかのデータがその出力に書き込まれている場合に jsp:forward を使用すると、IllegalStateException が投げられます。

メモ：バッファーを無効にするか、または JSP の出力バッファーのサイズを設定するには、JSP ディレクティブを使用します。page ディレクティブの詳細については、257 ページの「page ディレクティブ」を参照してください。

jsp:forward アクションのシンタックスでは、次の形式のいずれかを使用します。

```
<jsp:forward page="path" />
```

または

```
<jsp:forward page="path">
  <jsp:param name="paramName" value="paramValue" /> ...
</jsp:forward>
```

次の例で、ある JSP から別の JSP を呼び出す方法を示します。

```
<% String whereTo = "/templates/" + someValue; %>
<jsp:forward page='<%= whereTo %>' />
```

次の表で、jsp:forward アクションの属性を説明します。

属性	説明
page	呼び出されるファイルのパスを指定します。パスの詳細については、251 ページの「相対 URL の指定」を参照してください。

jsp:forward の 2 番めの形式は jsp:param アクションへの追加です。このアクションにより、目的の JSP により受信される HTTP リクエストにパラメータを追加できます。詳細については、270 ページの「jsp:param」を参照してください。

次の例は、ある JSP から別の JSP を呼び出す方法を示しています。

デスティネーション JSP に属性を渡すには

1 新規テキストファイルを作成して、次のように入力します。

```
<% request.setAttribute("Greeting", "Hello World"); %>
<jsp:include page="secondpage.jsp" flush="true"/>
```

2 Web サーバーのドキュメントのルートに JSP としてファイルを保存します。

3 2 番めのファイルを作成して、次のように入力します。

```
ご挨拶 :<%= request.getAttribute("Greeting")%>
```

4 2 番めのページを JSP として Web サーバーのドキュメントのルートに保存します。

5 ブラウザを開いて、最初のページをリクエストします。

この例では、あるページから別のページにデータを渡します。属性の値は String に限定されません。どのようなタイプのオブジェクトでも属性として渡すことができます。

また、jsp:param アクションを使用して、Greeting をパラメータとして渡すこともできます。この場合、パラメータはデスティネーション JSP に文字列として渡されます。この処理は、デスティネーション JSP が HTTP POST メソッドを使用してリクエストされた後で情報を受け取る場合と同様です。

request オブジェクト内でパラメータを使用して情報を 2 番めのページに渡すことによって 2 番めのページを直接クライアントのリクエストにレスポンスできるようにするか、または別の JSP から呼び出すことができるように、2 番めのページを作成できます。

次の例では、jsp:param アクションを使用して、Greeting パラメータを渡します。

```
<jsp:include page="page1.jsp" flush="true">
  <jsp:param name="Greeting" value="Hello World" />
</jsp:include>
```

インクルードされるページでは、次のように、request.getParameter メソッドを使用してパラメータにアクセスします。

```
ご挨拶 :<%= request.getParameter("Greeting") %>
```

呼び出し側ページと同様に、デスティネーション JSP も、JSP out オブジェクトを使用してクライアントにデータを返すことができます。ただし、呼び出し側ページの出力がバッファリングされている場合、JRun は、呼び出し前にバッファーをフラッシュします。このフラッシュのために、デスティネーションページはレスポンスヘッダーを設定できません。

また、デスティネーションページは、request オブジェクトを使用して呼び出し側ページに情報を返すこともできます。たとえば、デスティネーションページは何らかの値を決定して、setAttribute メソッドを使用してそれを request オブジェクトに書き込むことができます。デスティネーションページから呼び出し側ページに制御が戻されたとき、呼び出し側ページは request オブジェクトの getAttribute メソッドを使用してその情報にアクセスできます。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

jsp:param

jsp:param アクションにより、ある JSP から別の JSP にリクエストが転送されるときに、パラメータが名前 / 値のペアとして HTTP リクエストに追加されます。このアクションと併用できるのは、jsp:include、jsp:forward、および jsp:plugin アクションだけです。

jsp:include または jsp:forward とともに jsp:param を使用すると、デスティネーションページにより、オリジナルのリクエストパラメータを持つオリジナルの HTTP リクエストと、jsp:param で指定された新規パラメータがすべて受信されます。jsp:param により、リクエストに既に入っているパラメータが追加された場合、新規パラメータ値は既存の値の前に入ります。

たとえば、リクエストにパラメータ myParm=a が含まれているときに、jsp:param を使用して myParm=b を追加すると、転送されたリクエストには myParm=b, a が含まれます。ここで、新規パラメータがリストの先頭に追加されていることに注意してください。

新規パラメータのスコープは、jsp:include や jsp:forward のデスティネーション JSP です。つまり、インクルードされたページからオリジナルの JSP に返された後、新規パラメータと値がリクエストから削除されます。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

jsp:plugin

jsp:plugin アクションを使用して、クライアントブラウザでアプレットを呼び出すことができます。このアクションで、適切なクライアントブラウザ依存のコンストラクト (OBJECT または EMBED) を含む HTML テキストが生成されます。この構成体により、Java プラグインがダウンロードされ、続いてアプレットや JavaBean が実行されます。

JRun では、リクエスト元ユーザーエージェントに応じて、この要素が object タグまたは embed タグのいずれかで置き換えられ、レスポンスの出力ストリームに書き込まれます。

jsp:plugin アクションのシンタックスは次のとおりです。

```
<jsp:plugin
  type="bean|applet"
  code="objectCode"
  codebase="objectCodebase"
  { align="alignment" }
  { archive="archiveList" }
  { height="height" }
  { hspace="hspace" }
  { jreversion="jreversion" }
  { name="componentName" }
  { vspace="vspace" }
  { width="width" }
  { nspluginurl="url" }
  { iepluginurl="url" } >
  { <jsp:params>
    { <jsp:param name="paramName" value="paramValue" /> }+
  </jsp:params> }
  { <jsp:fallback> arbitrary_text </jsp:fallback> } >
</jsp:plugin>
```

中括弧 ({}) で囲まれた要素はオプションです。

次の例では、クライアントにある MyPlugin.class を呼び出します。

```
<jsp:plugin type=applet code="MyPlugin.class" codebase="/html" >
  <jsp:params>
    <jsp:param name="myplugin" value="Greetings"/>
  </jsp:params>
  <jsp:fallback>
    <p> unable to load Plugin </p>
  </jsp:fallback>
</jsp:plugin>
```

次の表で jsp:plugin アクションの属性を説明します。

属性	説明
type	コンポーネントのタイプを bean またはアプレットとして指定します。
code	アプレットを含むクラスファイル名、またはクラスへのパスを指定します。
codebase	archive 属性で指定された相対パスを解決するために使用されるベースパスを指定します。この値を省略すると、jsp:plugin を呼び出した JSP ファイルのパスが使用されます。
align	bean またはアプレットの場所を指定します。align に次の値を指定すると、周囲のテキストとの関係でオブジェクトの位置が決まります。 • bottom　オブジェクトの最下部が、現在のベースラインに対して垂直に揃えられます。これがデフォルトです。 • middle　オブジェクトの中心が、現在のベースラインに対して、垂直に揃えられます。 • top　オブジェクトの最上部が、現在のテキストラインの最上部に対して垂直に揃えられます。 • left および right　イメージが現在の左マージンまたは右マージンに移動します。
archive	オブジェクトのリソースを含むアーカイブを表す URI を、スペースで区切られたリストとして指定します。一般的に、アーカイブをあらかじめロードしておくと、オブジェクトのロード時間を削減できます。相対 URI として指定されたアーカイブは、codebase 属性に相対的であると解釈されます。
height	デフォルトの bean またはアプレットの高さを書き換えて、指定された値を使用します。この値は次の形式で指定できます。 • ピクセルを表す整数値 (N) • 使用可能な垂直スペースのパーセント値 (N%) • 使用可能な垂直スペースの部分 (N*)
hspace	bean またはアプレットの左右に挿入される余白の量を指定します。この値は次の形式で指定できます。 • ピクセルを表す整数値 (N) • 使用可能な水平スペースのパーセント値 (N%) • 使用可能な水平スペースの部分 (N*)

属性	説明
jreversion	コンポーネントが動作するために必要な JRE 仕様のバージョン番号を表します。デフォルトは 1.1 です。
name	アプレットの名前を指定します。これにより、このアプレットを JavaScript から参照したり、同じページにある別のアプレットを参照できるようになります。
vspace	bean またはアプレットの上下に挿入される余白の量を指定します。この値は次の形式で指定できます。 • ピクセルを表す整数値 (N) • 使用可能な垂直スペースのパーセント値 (N%) • 使用可能な垂直スペースの部分 (N*)
title	アプレットに関する記述情報を指定します。
width	bean やアプレットについて、デフォルトのオブジェクト幅を書き換え、指定された値を使用します。この値は次の形式で指定できます。 • ピクセルを表す整数値 (N) • 使用可能な水平スペースのパーセント値 (N%) • 使用可能な水平スペースの部分 (N*)
nspluginurl	Netscape Navigator 用 JRE プラグインをダウンロードできる URL を指定します。デフォルトは実装によって異なります。
iepluginurl	Internet Explorer 用 JRE プラグインをダウンロードできる URL を指定します。デフォルトは実装によって異なります。
jsp:param	アプレットコンポーネントまたは JavaBeans コンポーネントにパラメータを設定します。詳細については、270 ページの「jsp:param」を参照してください。
jsp:fallback	プラグインが起動できなかったときに、クライアントブラウザにより表示されるコンテンツを指定します。プラグインは起動できても、アプレットコンポーネントまたは JavaBeans コンポーネントが見つからなかったり、起動できなかったりした場合は、プラグイン固有のメッセージがユーザーに送られます。

HTML 関連の属性の詳細については、HTML 4.0 仕様 (http://www.w3.org/TR/REC-html40/) をご覧ください。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

JSP オブジェクトについて

デフォルトでは、JRun は JSP 内で使用するいくつかの暗黙的 JSP オブジェクトを作成します。これらの JSP オブジェクトは、JSP 仕様、Java サーブレット API、またはコア Java ライブラリによって定義されたオブジェクトのインスタンスです。これらのオブジェクトには、どの JSP からもアクセスできます。オブジェクトへのアクセスとは、オブジェクトメソッドを呼び出して、これらのメソッドに保管されているデータにアクセスすることです。

次の表で、各 JSP オブジェクトを説明し、それらのオブジェクトに対応するサーブレット API オブジェクトのタイプを定義します。

JSP オブジェクト	サーブレット API オブジェクトのタイプ	説明
application	javax.servlet.ServletContext	サーブレット設定オブジェクトから取得したサーブレットコンテキスト
config	javax.servlet.ServletConfig	JSP の ServletConfig
exception	java.lang.Throwable	エラーページが呼び出される原因となるキャッチされていない例外
out	javax.servlet.jsp.JspWriter	JSP の出力ストリームに書き込みを行うオブジェクト
pageContext	javax.servlet.jsp.PageContext	JSP に対するページコンテキスト
request	javax.servlet.HttpServletRequest	クライアントのリクエスト
response	javax.servlet.HttpServletResponse	クライアントへのレスポンス
session	javax.servlet.http.HttpSession	リクエスト元クライアントのために作成された session オブジェクト

JSP オブジェクトは、Java サーブレット API からオブジェクトとして実装されます。JSP オブジェクトにより、これらの Java サーブレット API オブジェクトにアクセスするための簡易メカニズムが使用できます。

各オブジェクトとそのメソッドの詳細については、Java サーブレット API の HTML バージョンのドキュメントを参照してください。JSP オブジェクト名ではなく、表にリストされている Java サーブレット API オブジェクトタイプを使用します。

JSP オブジェクトへのアクセス

オブジェクトを作成しないように JSP に明示的に指示しないかぎり、JRun によって JSP が呼び出されると、前のセクションの表にリストされているすべての JSP オブジェクトが作成されます。たとえば、**page** ディレクティブの **session** 属性を false に設定できます。この場合、JRun はそのページの session オブジェクトをインスタンス化しません。

メモ：セッショントラッキングを有効にした場合だけ、session オブジェクトが作成されます。exception オブジェクトが作成されるのは、JSP で **page** ディレクティブを使用して、**isErrorPage** を true に設定している場合だけです。詳細については、257 ページの「page ディレクティブ」を参照してください。

JSP オブジェクトを JSP で呼び出すときは、次のシンタックスを使用します。

object_name.<メソッド>(<変数 ...>)

このオブジェクトのメソッドを呼び出す前に、オブジェクトをインスタンス化する必要はありません。オブジェクトはすべて **page スコープ**です。つまり現在のページのステートメントからアクセスできます。

application オブジェクト

application オブジェクト使用すると、指定されたアプリケーションのすべてのユーザー間で情報を共有できます。アプリケーション内のどの JSP からも、application オブジェクトにアクセスできます。J2EE では、JSP ベースのアプリケーションは、仮想ディレクトリとそのサブディレクトリにあるすべての .jsp ファイルとして定義されます。

次の表で、application オブジェクトの一般的なメソッドを説明します。

メソッド	説明
getAttribute (String *name*)	指定された名前を持つ属性を返します。この名前を持つ属性が存在しない場合は、null を返します。
getAttributeNames	アプリケーションオブジェクト内にある属性名をすべて返します。
getInitParameter (String *name*)	初期化パラメータの値を返します。パラメータが存在しない場合は、null を返します。
getInitParameterNames	初期化パラメータの名前を返します。
getServerInfo	JRun サーブレットエンジンの名前とバージョン番号を返します。

次の例では、application オブジェクトに 2 つのパラメータを設定します。

```
<%
  application.setAttribute("appName", "Application Object Example");
  application.setAttribute("counter","0");
%>
```

次の例に示すように、このアプリケーションのこれ以降のページで、これらの設定にアクセスできます。

```
<HTML><BODY>
<h2>Display the default application settings</h2>
<%
  String appName = (String) application.getAttribute("appName");
  String counter = (String) application.getAttribute("counter");
%>
このアプリケーションの名前： "<%= appName %>"<BR>
The counter value = <%= counter %>
</BODY></HTML>
```

application オブジェクトの初期化パラメータを設定できます。クライアントが最初にアプリケーションをリクエストしたときに、JRun は初期化パラメータを application オブジェクトに書き込みます。パラメータにアクセスするには、**application.getInitParameter** メソッドを使用します。

一般的なテクニックは、次の例に示すように、データソースの名前を初期化パラメータとして web.xml ファイル内に設定することです。

```
<web-app>
...
<context-param>
  <param-name>datasourcename</param-name>
  <param-value>fred</param-value>
</context-param>
...
</web-app>
```

次の例では、アプリケーション全体の初期化パラメータにアクセスします。

```
...
<% String ds = (String)application.getInitParameter("datasource"); %>
try {
  DataSource ds = (DataSource) application.lookup(ds);
  dbConnection = ds.getConnection();
...
}
...
```

初期化パラメータへのアクセスの詳細については、129 ページの「初期化パラメータの使用」を参照してください。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

config オブジェクト

config オブジェクトは、サーブレットが初期化されるときに、設定情報をサーブレットに渡します。サーブレットがアクセスできる設定情報は、初期化パラメータを表す名前 / 値のペアのセットと、サーブレットが実行されているコンテキストを表す ServletContext オブジェクトです。

次の表で、config オブジェクトの一般的なメソッドを説明します。

メソッド	説明
`getInitParameter (String name)`	初期化パラメータの値を返します。パラメータが存在しない場合は、null を返します。
`getInitParameterNames`	各サーブレットの初期化パラメータ名を返します。
`getServletName`	サーブレット名を返します。

初期化パラメータへのアクセスの詳細については、129 ページの「初期化パラメータの使用」を参照してください。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

exception オブジェクト

exception オブジェクトはすべてのエラーと例外を表します。`page` ディレクティブの `isErrorPage` 属性を使用して、エラーページとして宣言したページ上の exception オブジェクトにアクセスできます。

次の表で、exception オブジェクトの一般的なメソッドを説明します。

メソッド	説明
`getMessage`	エラーメッセージが含まれている文字列を返します。
`printStackTrace`	標準エラーに exception オブジェクトとそのバックトレースを書き込みます。
`toString`	exception オブジェクトについて説明した文字列を返します。

exception オブジェクト使用の詳細については、281 ページの「エラーの処理」を参照してください。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

out オブジェクト

out オブジェクトは、JSP の出力ストリームに書き込みを行うオブジェクトを定義します。

次の表で、out オブジェクトの一般的なメソッドを説明します。

メソッド	説明
clear	クライアントにコンテンツを書き込まずに、出力バッファーをクリアします。バッファーが既にフラッシュされている場合、メソッドは例外を投げます。
clearBuffer	コンテンツをクライアントに書き込まずに、出力バッファーをクリアします。
flush	クライアントにコンテンツを書き込んで、出力バッファーをフラッシュします。
getBufferSize	バッファーのサイズをバイト単位で返します。出力がバッファリングされていない場合は、0 を返します。
getRemaining	バッファー内の空き容量をバイト単位で返します。
isAutoFlush	出力バッファーが自動的にフラッシュされる場合は true を返します。
newLine	出力に改行文字を書き込みます。
print	改行文字を付けずに、値を出力に書き込みます。
println	改行文字を付けて、値を出力に書き込みます。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

pageContext オブジェクト

pageContext オブジェクトは、JSP にローカルな情報を保管します。各 JSP には、ユーザーがページに入ったときに JRun によって作成される JSP 自体の pageContext オブジェクトがあり、ユーザーがそのページを離れたときに、このオブジェクトは JRun によって破棄されます。pageContext オブジェクトのメソッドを使用して、JSP の情報にアクセスしたり、他のアクションを実行できます。

次の表で、pageContext オブジェクトの一般的なメソッドを説明します。

メソッド	説明
findAttribute(String *name*)	page、request、session (有効である場合)、および application スコープ内の属性値を返します。属性が見つからない場合は、null を返します。
getAttribute(String *name*)	page スコープ内で、指定された名前に関連付けられているオブジェクトを返します。オブジェクトが見つからない場合は、null を返します。
removeAttribute (String *name*)	指定された名前を持つ属性を削除します。
setAttribute (String *name*, java.lang.Object *attribute*)	関連付けられた名前を持つオブジェクトを pageContext オブジェクトに書き込みます。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

request オブジェクト

request オブジェクトは、HTTP リクエスト中にクライアントから Web サーバーに渡される値を取り出します。

次の表で、request オブジェクトの一般的なメソッドを説明します。

メソッド	説明
`getCookies`	リクエストとともにクライアントから送信されたすべての Cookie を返します。
`getHeader(String name)`	リクエストヘッダーの値を文字列として返します。
`getAttribute(String name)`	指定された属性値を返します。この属性が存在しない場合は、null を返します。
`getAttributeNames`	リクエストに含まれる属性名をすべて返します。
`getHeaderNames`	リクエストに含まれるすべてのヘッダー名を返します。
`getHeaders(String name)`	指定されたリクエストヘッダーの値をすべて返します。
`getMethod`	リクエストの作成に使用された HTTP メソッドに対する `GET`、`POST`、または `PUT` を返します。
`getParameter(String name)`	リクエストに含まれるパラメータの値を返します。パラメータが存在しない場合は、null を返します。
`getParameterNames`	リクエストに含まれるパラメータ名をすべて返します。
`getParameterValues (String name)`	指定されたパラメータの値をすべて返します。
`getQueryString`	リクエストからクエリ文字列を返します。
`getRequestURI`	リクエストの URL のうち、プロトコル名からクエリ文字列までの部分を返します。
`getServletPath`	リクエストの URL のうち、サーブレットを呼び出す部分を返します。
`setAttribute(String name, java.lang.Object o)`	属性と、関連する値をリクエストに書き込みます。

JSP request オブジェクトを使用して、HTTP リクエストの一部としてクライアントから JSP に送信されたデータにアクセスします。たとえば、リクエストには HTML フォームから JSP に渡されたデータを含めることができます。フォーム データは HTTP リクエストの名前 / 値のペアとして JSP に送信されます。この情報にアクセスするには、request オブジェクトとそのメソッドを使用します。

また、パラメータをそのページのリクエスト URL の一部として JSP に渡すこともできます。たとえば、次の URL を使用して、JSP をリクエストし、そのリクエストと併せて 2 つのパラメータを渡すことができます。

http://localhost/my.jsp?fName=Bob&lName=Smith

次の例に示すように、request.getParameter メソッドを使用して、フォームまたはリクエスト URL から JSP に渡されたパラメータを取得します。

```
<%
  String firstName = request.getParameter("fName");
  String lastName = request.getParameter("lName");
  out.println("Welcome " + firstName + " " + lastName);
%>
```

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

response オブジェクト

response オブジェクトはデータをクライアントに送信します。

次の表で、response オブジェクトの一般的なメソッドをリストします。

メソッド	説明
addCookie(Cookie *cookie*)	レスポンスに Cookie を書き込みます。
addHeader(String *name*, String *value*)	ヘッダーを名前 / 値のペアとしてレスポンスに書き込みます。ヘッダーが存在する場合、この既存のヘッダー値に値が追加されます。
containsHeader (String *name*)	指定されたレスポンスヘッダーが既に設定されている場合は、true を返します。
sendError (int *status_code*)	指定されたステータスコードを含むエラーレスポンスをクライアントに送信します。
setHeader(String *name*, String *value*)	指定された名前と値で、レスポンスヘッダーを設定します。このヘッダーが既に存在している場合、この値でヘッダーの現在の値を置き換えます。

session オブジェクト

session オブジェクトは、特定のユーザーセッションに関する情報を保管します。デフォルトでは、JRun は Cookie を使用してセッションをトラッキングしますが、URL リライティングまたは非表示フォームフィールド、またはカスタムメソッドを使用して、セッションをトラッキングできます。Cookie 自体には、サーバーサイドの session オブジェクトへのリクエストに一致する ID 番号 (セッション ID と呼ばれます) だけが含まれています。URL リライティングまたは非表示フォームフィールドを使用した場合、セッション ID は Cookie の値としてではなく、追加パス情報またはフォームフィールドとして渡されます。

ユーザーがアプリケーション内で別のページに移動しても、session オブジェクトに保管された変数は破棄されず、ユーザーセッションが継続している間は、保持されます。クライアントがリクエストでセッション ID を送り続けているかぎり、JRun はそのクライアントのユーザー特有のセッションデータにアクセスできます。

まだセッションを開始していないユーザーからアプリケーションのページがリクエストされた場合、JRun は session オブジェクトを作成します。セッションが期限切れになった場合や放棄された場合は、Web サーバーにより session オブジェクトが廃棄されます。セッション使用の詳細については、151 ページの「セッションの操作」を参照してください。

次の表は、session オブジェクトの一般的なメソッドをリストします。

メソッド	説明
getAttribute (String *name*)	指定された名前を持つオブジェクトを返します。オブジェクトが見つからない場合は、null を返します。
getAttributeNames	このセッションに含まれるオブジェクト名をすべて返します。
getCreationTime	グリニッジ標準時で 1970 年 1 月 1 日午前 0 時を基準に、セッションが作成された時刻をミリ秒単位で返します。
getId	セッションに対する固有の ID を返します。
getLastAccessedTime	このセッションに関連付けられている、最後にクライアントリクエストが行われた時刻を返します。時刻は、グリニッジ標準時で 1970 年 1 月 1 日午前 0 時を基準に、ミリ秒単位で返されます。
getMaxInactiveInterval	クライアントアクセス間で、JRun がこのセッションを開いた状態にしておく最長時間を秒単位で返します。
removeAttribute (String *name*)	セッションから属性と値を削除します。
setAttribute(String *name*, java.lang.Object *value*)	属性と、関連する値をセッションに書き込みます。

ある JSP が session オブジェクトに情報を書き込み、その後、クライアントからリクエストされた別の JSP がそれにアクセスできます。たとえば、ユーザーのショッピングカートに ID を割り当てておくと、クライアントがカートの内容を追加、削除、または修正したときにはいつでも、Web サイトからショッピングカートに関する情報にアクセスできます。この ID を session オブジェクトに保管できます。

次の例では、session オブジェクトに属性を設定します。

```
<%
  String firstName = request.getParameter("fName");
  String lastName = request.getParameter("lName");
  session.setAttribute("fName", firstName);
  session.setAttribute("lName", lastName);
  out.println("Welcome " + firstName + " " + lastName);
%>
```

次の例に示すように、同じセッション内の別の JSP がこの属性にアクセスできます。

```
<%
  String firstName = (String) session.getAttribute("fName");
  String lastName = (String) session.getAttribute("lName");
  out.println("あなたの注文は準備できました。" + firstName + " " + lastName);
%>
```

getAttribute がタイプ java.lang.Object のオブジェクトを返すので、getAttribute の戻り値をキャストする必要があります。キャストは、getAttribute から返されたオブジェクトをデスティネーションの形式 (この場合は文字列) に変換します。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

エラーの処理

JSP には強固なエラー処理能力が備わっています。try-catch ブロックなど、Java コードのあらゆるエラー処理テクニックを使用できます。JSP 用に特別に設計されたテクニックも使用できます。

JSP のエラーは、次に示す JSP ライフサイクルの 2 つのポイントで発生します。

- JRun が JSP ソースファイルを Java クラスファイルに変換するとき。このエラーはコンパイル時に発生します。
- JSP がリクエストを処理するとき。このエラーは実行時に発生します。

コンパイル時エラーとランタイムエラーの違いを理解する必要があります。

- **コンパイル時エラー**　コンパイル時エラーは、クライアントが最初に JSP を呼び出すときに発生します。JRun は JSP を XML 表示に変換し、次に、XML 表示からサーブレットをコンパイルします。この処理のどこかでエラーが発生した場合、コンパイル時エラーが投げられます。このエラーは JSP の外でキャッチする必要があります。ただし、JSP をテストしないか、またはプリコンパイルしない場合は、このエラーをまったくキャッチできないことがあります。
- **ランタイムエラー**　ランタイムエラーは、JSP がコンパイルに成功した後で、たとえば、存在しないファイルをインクルードしようとしたときなど、無効なステートメントを含むコードを処理しようとしたときに発生します。JSP がコンパイルに成功したことがわかっており、エラーをキャッチして出力を調べれば良いだけなので、ランタイムエラーは簡単に処理できます。

ランタイムエラーのキャッチ

ランタイムエラー情報には次の方法でアクセスできます。

- JSP 暗黙的 exception オブジェクトを使用
- `javax.servlet.error` 属性を使用

このセクションでは、これらの方法について説明します。

exception オブジェクトの処理

ランタイムエラーは、エラーを生成した JSP 内でキャッチし処理できます。次の例に示すように、`page` ディレクティブを使用してエラーページを指定することもできます。このページは例外を処理する別の JSP です。

```
<%@ page errorPage="errhand.jsp"%>
<html><head><title> エラーを発生させるページ </title></head><body>
<%
 int zero = 0;
 int x = 42/zero;
%>
</body></html>
```

キャッチされない例外が JSP から投げられると、その JSP によって、その例外とクライアントリクエストが `errorPage` 属性で指定されたページに転送されます。

エラーページとして使用する JSP は、JSP のディレクティブを使用して `isErrorPage` 属性を true に設定する必要があります。JSP がエラーを生成し、エラーページに転送するとき、エラーページの JSP exception オブジェクトを、生成されたエラーに設定します。

次のコードに示すように、exception オブジェクトを使用して、エラー、エラーメッセージ、スタックトレースの簡単な説明にアクセスできます。

```
<%@ page isErrorPage="true" %>
<html><head><title> エラーハンドラ </title></head><body>
大変です。500 エラーページ
<p><table border=1>
<tr>
  <td><strong> エラーメッセージ </strong></td>
  <td><%= exception.getMessage() %></td>
</tr>
<tr>
  <td><strong> 例外を文字列に変換 </strong></td>
  <td><%= exception.toString() %></td>
</tr>
<tr>
  <td><strong> スタックトレース </strong></td>
  <td>
<%
  exception.printStackTrace(new java.io.PrintWriter(out));
%>
</td></tr>
</table></body></html>
```

page ディレクティブの詳細については、257 ページの「page ディレクティブ」を参照してください。exception オブジェクトの詳細については、276 ページの「exception オブジェクト」を参照してください。

エラー属性の処理

page ディレクティブ内でエラーページを指定しなかった場合は、キャッチされない例外が発生すると、JRun は、エラーステータスコード 500 (サーバーエラー) をクライアントに返します。

次の例に示すように、このタイプのエラーは、web.xml ファイル内の error-page 要素を使用してキャッチできます。

```
<error-page>
  <error-code>500</error-code>
  <location>/errhand.jsp</location>
</error-page>
```

実行時にエラーが投げられると、JRun は、exception 属性を request オブジェクトに設定し、そのオブジェクトを web.xml ファイル内で定義したエラー処理ページに転送します。

次の表で、javax.servlet.error で始まる名前を持つエラー属性を説明します。

属性	説明
message	エラーメッセージを表す文字列。
status_code	エラーのステータスコードを表す整数。
exception_type	投げられた例外のタイプ。
request_uri	例外を投げたページのリクエスト URI を表す文字列。
servlet_name	例外を投げたサーブレットの名前を表す文字列。

次の例では、エラー属性を出力する HTML テーブルを表示します。

```
<html><head><title> エラーハンドラ </title></head><body>
<H2> 大変です。500 エラーページ </H2>
<p><table border=1>
<tr>
  <td><b> ステータスコード </b></td>
  <td><%= request.getAttribute("javax.servlet.error.status_code") %></td>
</tr>
<tr>
  <td><b> 例外のタイプ </b></td>
  <td><%= request.getAttribute("javax.servlet.error.exception_type")
        %></td>
</tr>
<tr>
  <td><b> メッセージ </b></td>
  <td><%= request.getAttribute("javax.servlet.error.message") %></td>
</tr>
<tr>
  <td><b> 例外 </b></td>
  <td><%= request.getAttribute("javax.servlet.error.exception") %></td>
</tr>
<tr>
  <td><b>URI</b></td>
  <td><%= request.getAttribute("javax.servlet.error.request_uri") %></td>
</tr>
<tr>
  <td><b> サーブレット名 </b></td>
  <td><%= request.getAttribute("javax.servlet.error.servlet_name") %></
        td>
</tr>
</table>
</body></html>
```

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

コンパイル時エラーのキャッチ

JSP を JSP ソースファイルから Java クラスファイルに変換する処理は、まず JRun がこのファイルへの最初のリクエストを受け取ったときに発生します。その後の変換は、ページが最後に変換されてから JSP ソースコードファイルが修正されたと JRun が判断したときに行われます。

変換が失敗すると、クライアントリクエストが失敗し、対応するエラーが投げられます。たとえば、JRun が変換エラーを検出すると、JRun からエラーステータスコード 500 (サーバーエラー) を返します。

ユーザーがページをリクエストするまで、使用中のページにコンパイル時エラーがあることに開発者が気付かないことがあります。その JSP は正しくコンパイルされないので、開発者がコンパイル時エラーを投げるようにした場合でも、`page` ディレクティブ内の `errorPage` 属性でそのエラーは処理されません。そのかわりに JRun がコンパイラエラーを投げます。500 エラーはやはりそのページをリクエストしているクライアントに返されますが、エラー処理の方法は異なります。

JSP でコンパイル時エラーをキャッチするには、次の例に示すように、web.xml ファイル内で `error-page` 要素を使用します。

```
<error-page>
  <error-code>500</error-code>
  <location>/exception500.jsp</location>
</error-page>
```

コンパイル時の JSP エラーを防止するには JSP をプリコンパイルします。JSP のプリコンパイルの詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

サンプルの JSP を表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

第 11 章
Java のカスタムタグ

この章では、JSP (JavaServer Pages) でカスタムタグを使用する方法および Java でタグハンドラを作成する方法について説明します。また、JRun には JSP でタグハンドラを作成する機能もあります。詳細については、315 ページの第 12 章「JSP カスタムタグのコーディング」を参照してください。

目 次

カスタムタグとタグライブラリの概要

JavaServer Pages の仕様書には、タグライブラリに関するフレームワークが記述されています。タグライブラリでは、JSP プログラマ用のタグの作成に使用できる強力な機能が提供されます。

タグライブラリは、タグまたはカスタムタグと呼ばれる 1 つ以上のアクションから構成され、関連する Java タグハンドラクラスでコーディングされている処理が、それぞれのタグによって実行されます。

Java 開発者は、各カスタムタグの機能 (属性を含む) を定義し、タグハンドラをコーディングし、さらにタグライブラリディスクリプタ (TLD) ファイルで各カスタムタグを定義します。TLD ファイルでは、タグへの本文のビルトイン可能性や必須属性など、その他の情報も定義します。JSP 開発者は、タグハンドラクラスへのポインタを JSP に含め、そのタグを JSP タグであるかのように使用します。

アプリケーションデプロイ担当者は、web.xml、TLD、および TEI (Tag Extra Information) ファイルを使用して、Web アプリケーションにおけるタグの動作を設定します。

タグの基本

タグの基本的な構造は、JSP ページで使用される HTML 構成として理解する必要があります。次の 3 行は、HTML 仕様の一部であるタグを示しています。

```
<A HREF="http://www.hamsteak.com">
 ホームページに移動するには、ここをクリックしてください。
</A>
```

1 行めは開始タグ (<A>) を示し、タグの動作を設定する属性がオプションで含まれています (この場合、属性は HREF です)。2 行めは、タグでラップするテンプレートまたは本文を示しています。このテキストはタグの本文と呼びます。3 行めは終了タグを示しています。これは開始タグに似ていますが、終了スラッシュがあり、属性を含みません。

タグの中には本文を持たないものもあります。これらはスタンドアローンタグと呼びます。スタンドアローンタグの例には、<IMG> や <HR> タグがあります。スタンドアローンタグを閉じるには、次の例のように、左不等号の前にスペースとスラッシュを追加します。

```
<HR />
<IMG SRC="gaffer.gif" width=100 height=100 />
<HR />
```

スタンドアローンタグには本文がないので、空のタグと呼ばれることもあります。

タグライブラリの利点

システムデザイナーは、タグライブラリを使用して Web アプリケーションのデザインパターンを実装することがよくあります。これによって、開発者は JSP での制限を決め、JSP からビジネスロジックを抽出し、ヘルパークラスに挿入することができます。また、JSP プログラマが理解しやすいページレベルのプログラムを作成します。

タグライブラリの利点は次のとおりです。

- 関連機能のカプセル化
- JSP からのスクリプトレットの削除
- 一般的なタグベースメタファーを使用したビジネスロジック機能へのアクセスの提供
- 反復されるビジネスロジックの削減または削除
- Java 経験の少ない JSP 開発者でもタグライブラリへのアクセスが可能

- タグハンドラ開発者によるページ出力時の条件の制限が可能

Java 開発者と JSP 開発者とでは、カスタムタグライブラリとの対話が次のように異なります。

- Java 開発者は、タグライブラリ内のクラスやサポートファイルのコーディング、文書化、およびパッケージを実施します。
- JSP 開発者の視点から見ると、タグライブラリは、特定のタイプの処理の実行時に使用するカスタムタグが格納されているライブラリです。JSP 開発者は、TLD ファイルの格納場所、ライブラリ内のカスタムタグの名前、タグ属性、スクリプト変数、スクリプト変数の使用方法、およびスクリプト変数のスコープを知っておく必要があります。

カスタムタグライブラリの例

タグの中には、Web アプリケーションの JSP コンポーネントのデザインパターンを実行するために使用されるものがあります。それ以外のタグは、多くのスクリプトレットコードに必要な複雑なオペレーションを単純化するために使用されます。独自のカスタムタグを作成する前に、他の開発者が作成したタグライブラリの使用を検討してください。一般的な機能の多くは、カスタム JSP タグとして既に実装されています。

次の表で、既存のカスタムタグライブラリの例を説明します。

タグライブラリ	説明
JSP 標準タグライブラリ (JSTL)	JSTL は、JSP で使用するためのカスタムタグのコアセットを提供します。このライブラリには、データベースへのアクセス、XML ドキュメントの処理、JSP の国際化対応を行うためのタグが含まれています。 詳細については、http://java.sun.com/products/jsp/taglibraries.html をご覧ください。
JRun タグライブラリ	JRun タグライブラリは本来 JRun 3.0 で出荷されています。このライブラリはあまり使用されていませんが、JSTL に影響を与えた多くのタグを提供します。
Struts タグライブラリ	一般的な MVC (Model-View-Controller) デザインパターンを実装するためのタグを提供するカスタムタグライブラリです。 詳細については、http://jakarta.apache.org/struts/index.html をご覧ください。 JRun での Struts の使用方法の詳細については、27 ページの「Struts について」を参照してください。
Dreamweaver UltraDev のカスタムタグライブラリエクステンション (CTLX)	CTLX は、UltraDev デザイン環境でカスタムタグライブラリを使用するための機能を提供します。 詳細については、http://jakarta.apache.org/taglibs/doc/ultradev4-doc/index.html をご覧ください。
Regexp	JSP で Perl 正規表現シンタックスをエミュレートするタグを提供します。詳細については、http://jakarta.apache.org/taglibs/doc/regexp-doc/intro.html をご覧ください。

多くのオープンソースタグライブラリの詳細およびリンクについては、次のリソースを参照してください。

- http://jakarta.apache.org/taglibs/index.html
- http://jsptags.com/tags/

タグライブラリの使用

JSP 開発者にとって、JSP の作成に利用できるツールのリストにタグライブラリを追加することは簡単です。このセクションでは、カスタムタグを使用して、従来の JSP シンタックスおよび JSP XML シンタックスで JSP を作成する方法について説明します。

JSP でのカスタムタグの使用

カスタムタグライブラリのタグを使用する前に、Web アプリケーションを設定する必要があります。

Web アプリケーションにカスタムタグをインストールおよび使用するには

1 タグライブラリディスクリプタ (TLD) ファイルを Web アプリケーションの /WEB-INF ディレクトリに移動します。

2 タグライブラリの JAR ファイルを Web アプリケーションの /WEB-INF/lib ディレクトリに移動します。

3 web.xml ファイルの `taglib` 要素にあるタグライブラリの TLD ファイルを指定します。

次の例は、JRun タグライブラリの web.xml ファイルのエントリを示しています。

```
<taglib>
  <taglib-uri>jruntags</taglib-uri>
  <taglib-location>/WEB-INF/jruntags.tld</taglib-location>
</taglib>
```

4 カスタムタグを呼び出す各 JSP ページの上部に、次の例に示す taglib ディレクティブを追加します。

```
<%@ taglib uri="jruntags" prefix="jrun" %>
```

`taglib` ディレクティブは、タグライブラリの URI を識別し、タグ名とともに使用してライブラリのタグを呼び出す接頭辞を指定します。

次の行は、taglib ディレクティブのシンタックスを示しています。

```
<%@ taglib uri="taglib_URI" prefix="tag_prefix" %>
```

接頭辞には任意の値を指定できますが、そのタグライブラリの各タグに使用する必要があります。

5 JSP でカスタムタグを使用するには、`prefix:tagname` 規則を使用してカスタムタグをコーディングします。

次の例は、`tag` 接頭辞を `test` として定義し、WEB-INF ディレクトリにあるタグライブラリの TLD ファイルを指定する `taglib` ディレクティブを示しています。

```
<%@ taglib prefix="test" uri="/WEB-INF/DocSamples.tld" %>
<html>
  <body>
    <h1> 簡単なカスタムタグ </h1>
    <%-- この例では、開始タグと終了タグを組み合わせます。 --%>
    <test:hello/>
  </body>
</html>
```

JSP XML シンタックスでのカスタムタグの使用

JSP XML シンタックスを使用して JSP を作成する JSP 開発者は、`jsp:root` 要素の `xmlns` 属性を使用して、カスタムタグライブラリの URI を宣言する必要があります。次の例は、JRun タグライブラリを追加する方法と `jrun` に接頭辞を設定する方法を示しています。

```
<jsp:root xmlns:jsp="http://java.sun.com/jsp_1_2" xmlns:jrun="/
        WEB-INF/jruntags.tld" versi
on="1.2">
```

詳細については、47 ページの第 4 章「JRun と XML」を参照してください。

タグハンドラのオーサリング

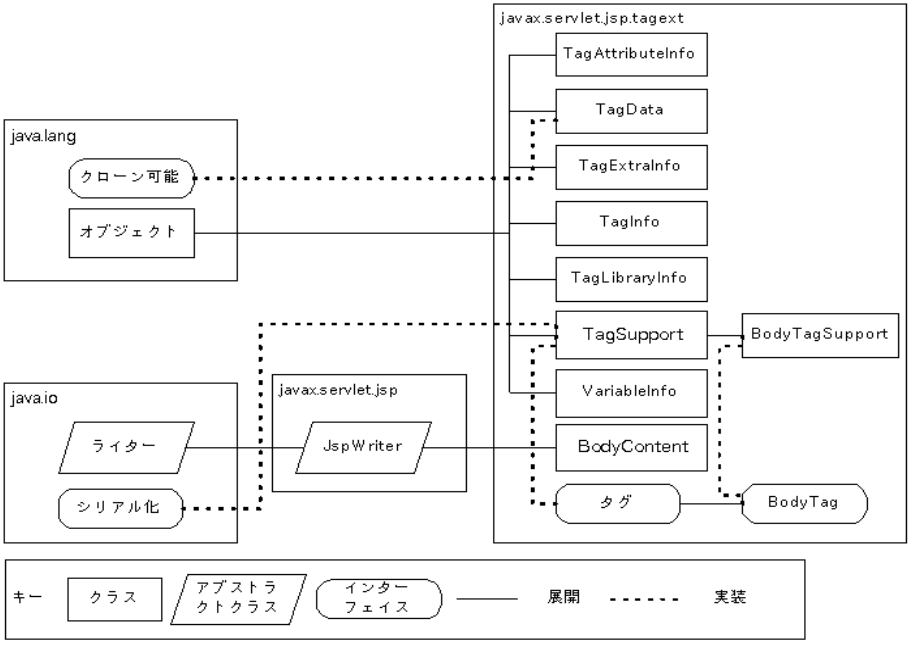
Java 開発者は、タグハンドラの開発時に、次のタスクを行う責任があります。

- タグハンドラのコーディング
- TEI クラスのコーディング (オプション)
- サポートクラスのコーディング (オプション)
- TLD ファイルの作成
- ライブラリ内のカスタムタグの名前、タグ属性、スクリプト変数、スクリプト変数の使用方法、およびスクリプト変数のスコープの文書化
- タグライブラリ用 JAR ファイルの作成
- Web アプリケーション用 web.xml ファイルの `taglib` 要素の作成

次のセクションでは、クラスとインターフェイス、およびタグハンドラの作成方法について説明します。

クラスとインターフェイス

カスタムタグやタグライブラリのコーディングには、`javax.servlet.jsp.tagext` パッケージのクラスとインターフェイスを使用します。次の図は、これらのクラスとインターフェイスの関係を示しています。



次の表で、プライマリクラスおよびインターフェイスについて説明します。

クラス / インターフェイス	説明
Tag インターフェイス	関連する開始タグと終了タグによって呼び出される基本的な開始メソッドと終了メソッドを定義します。これらのメソッドには、doStartTag と doEndTag が含まれます。
IterationTag インターフェイス	Tag インターフェイスを拡張します。カスタムタグを使用して本文テキスト上でループ処理を行う場合に使用する追加メソッドを定義します。このメソッドは doAfterBody と呼びます。
BodyTag インターフェイス	IterationTag インターフェイスを拡張します。カスタムタグを使用して本文テキストと対話する場合や、必要に応じて doInitBody メソッド などの結果の変更を行う場合に使用する追加メソッドを定義します。
TagSupport クラス	Tag インターフェイスを実装します。これは、本文テキストと対話しないタグハンドラに応じて拡張可能なオプションのヘルパークラスです。
BodyTagSupport クラス	BodyTag インターフェイスを実装します。これは TagSupport のサブクラスです。これは、本文テキストと対話するタグハンドラに応じて拡張可能なオプションのヘルパークラスです。

各インターフェイスにより、前のインターフェイスが拡張されます。その結果 IterationTag インターフェイスには、Tag のメソッドだけでなく追加の機能も含まれます。BodyTag インターフェイスには、他の 2 つのインターフェイスに含まれるすべてのメソッドと追加機能が含まれます。

タグライブラリのプログラミングで使用するクラスとインターフェイスの詳細については、JRun/docs ディレクトリにある JavaDocs を参照してください。

使用するインターフェイスの決定

javax.servlet.jsp.tagext パッケージを拡張する各インターフェイスは、その前にパッケージを拡張します。その結果、IterationTag インターフェイスには Tag インターフェイスのすべての機能と追加メソッドが、BodyTag インターフェイスには IterationTag インターフェイスのすべての機能と追加メソッドが含まれます。

カスタムタグハンドラのあらゆる実装で、BodyTag インターフェイスは排他的に使用され、機能やパフォーマンスに影響を及ぼすことはありません。これにより、最大限の柔軟性と複雑さが同時に実現されます。Tag インターフェイスと IterationTag インターフェイスが含まれることにより、あまり多くのメソッドを使用しない、より明解な実装を実現できます。

BodyTag を使用する利点の 1 つに、インターフェイスを変更せずに Web アプリケーションに機能を追加できる点があります。Tag インターフェイスを使用する場合は、タグハンドラが使用するインターフェイスを変更しないかぎり、タグハンドラを使用して本文テキストのループ処理はできません。IterationTag インターフェイスを使用する場合は、インターフェイスを変更しないかぎり、タグハンドラを使用して本文コンテンツのバッファリングや変更はできません。

簡単なタグハンドラのコーディング

簡単なタグハンドラでは、次の処理を行います。

- Tag インターフェイスを拡張します。
- TagSupport クラスを実装します。
- doStartTag をオーバーライドします。
- オプションで、TagSupport または BodyTagSupport から継承された doEndTag メソッドをオーバーライドします。
- 本文テキストとの対話は行いません。

TagSupport を拡張すると、doStartTag メソッドおよび doEndTag メソッドは、次の戻り値 (定数として定義済み) を使用します。

- doStartTag は、次のいずれかの値を返します。
  - EVAL_BODY_INCLUDE　開始タグと終了タグの間の本文テキスト (JSP コードを含む) を受け入れます。ただし、doEndTag メソッドでは本文テキストを使用できないことに注意してください。
  - SKIP_BODY　本文テキストを無視します。開始タグと終了タグの間にあるテキストは評価せず、表示しません。
- doEndTag は、次のいずれかの値を返します。
  - EVAL_PAGE　ページの評価を続行します。
  - SKIP_PAGE　ページの残りを無視します。

次のタグハンドラは、doStartTag メソッドと doEndTag メソッドから HTML を出力します。この HTML は JSP ページのタグの内容を形成します。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class SimpleTag extends TagSupport {
  /**
  * タグが開始されたら実行します。
  */
public int doStartTag() throws JspException {
  try {
    pageContext.getOut().print("<h2>Hello from doStartTag()</h2>");
    // タグの本文でテキストを使用できるようにします。
    return EVAL_BODY_INCLUDE;
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}

/**
* 終了タグを実行します。
*/
```

```
public int doEndTag() throws JspException {
  try {
    pageContext.getOut().print("<h2>Hello from doEndTag()</h2>");
    // ページの評価を継続します。
    return EVAL_PAGE;
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}
}
```

このカスタムタグハンドラをコンパイルして JRun 環境に追加すると、次の行を JSP に追加することによって、このタグを呼び出すことができます。

```
<prefix:simpletag />
```

タグハンドラの保存

コンパイル済みのタグハンドラクラスは、次のいずれかの格納場所に保存します。

- **WEB-INF/classes**　このディレクトリは、開発の予備段階やテストの実施中にタグハンドラクラスを保存するのに適しています。ただし、タグライブラリをパッケージするときは、事前にタグハンドラを WEB-INF/lib 内の JAR ファイルに保存しておく必要があります。
- **WEB-INF/lib**　パッケージングやデプロイを行う場合は、タグハンドラ、TEI クラス、ドキュメント、およびその他の関連ファイルを WEB-INF/lib 内の JAR ファイルに保存します。

推奨されている TLD ファイルの配置方法など、その他のパッキング情報については、314 ページの「タグライブラリのパッケージング」を参照してください。

タグハンドラの使用

JSP でタグハンドラのクラスを使用するには、web.xml ファイルに `taglib` 要素を追加し、TLD ファイルでタグを定義し、JSP の TLD の URI にリンクする必要があります。

web.xml ファイルへの `taglib` 要素の追加の詳細については、314 ページの「タグライブラリのパッケージング」を参照してください。

TLD ファイルの作成の詳細については、294 ページの「TLD ファイルの作成」を参照してください。JSP の TLD へのリンクの詳細については、288 ページの「タグライブラリの使用」を参照してください。

TLD ファイルの作成

TLD は、タグライブラリを記述する XML 形式のテキストファイルです。JRun では TLD ファイルを使用して、taglib ディレクティブを含んでいる、JSP 内のカスタムタグの動作を定義します。

taglib 要素は、TLD ファイルのルートです。次の表で、TLD ファイルに含まれているサブ要素について説明します。

taglib サブ要素	説明
tlibversion	タグライブラリのバージョン。
jspversion	オプション。タグライブラリに必要な JSP のバージョン。
shortname	デフォルトのショートネーム。
displayname	オプション。開発環境で使用するタグライブラリ名。
smallicon	オプション。タグライブラリを表す 16 × 16 ピクセルのアイコンへの相対パス。GUI 環境で使用。
largeicon	オプション。タグライブラリを表す 32 × 32 ピクセルのアイコンへの相対パス。GUI 環境で使用。
uri	オプション。タグライブラリを固有に識別する URI。
info	オプション。タグライブラリの使用情報。
tag	カスタムタグ情報。TLD ファイルは、1 つ以上の tag 要素を含むことができます。それぞれの tag 要素には、次のサブ要素が含まれています。 • name　カスタムタグ名。 • tagclass　タグハンドラのクラス名。 • teiclass　TEI ファイルのクラス名。 • bodycontent　本文のコンテンツタイプを識別します。有効な値は、tagdependent (SQL ステートメントなどのタグ依存本文コンテンツ)、jsp (JSP および HTML 本文コンテンツ)、および empty (本文コンテンツ使用不可) です。empty を指定した場合、カスタムタグの本文は空です。 • info　オプション。カスタムタグ使用情報。 • attribute　オプション。属性情報。tag 要素は、attribute 要素を持つ場合と持たない場合があります。タグ内での属性の使用については、296 ページの「属性の使用」を参照してください。 • variable　オプション。タグ要素内のスクリプト変数を定義します。詳細については、306 ページの「スクリプト変数の使用」を参照してください。

taglib サブ要素	説明
listener	タグライブラリにイベントリスナを登録します。listener 要素は、listener-class サブ要素を指定する必要があります。イベントリスナの詳細については、205 ページの第 8 章「アプリケーションのライフサイクルイベント」を参照してください。
validator	このタグの validator クラスを識別します。validator 要素には、次のサブ要素が含まれています。 • validator-class　必須。タグライブラリを検証するクラスの完全修飾名。 • init-params　オプション。バリデータが使用する 1 つ以上の初期化パラメータを指定します。 • description　オプション。バリデータを記述します。 タグライブラリの検証の詳細については、313 ページの「タグライブラリの検証」を参照してください。

taglib ディレクティブには、TLD ファイルを指定する uri 属性が含まれています。taglib ディレクティブの詳細については、288 ページの「タグライブラリの使用」を参照してください。

カスタムタグは、TLD ファイル内で tag 要素とそのサブ要素によって定義します。tag 要素の主な目的は、JSP ファイルで使用されているカスタムタグ名をタグハンドラのクラスファイルに関連付けることです。JSP 1.1 仕様に記述されているメソッドを使用してカスタムタグでスクリプト変数を作成する場合は、そのタグの TEI クラスファイル名を指定する teiclass 要素を使用する必要もあります。詳細については、309 ページの「JSP 1.1 でのスクリプト変数の使用」を参照してください。

tag 要素を使用して属性を定義することもできます。カスタムタグ属性の詳細については、296 ページの「属性の使用」を参照してください。

TLD ファイルの例

次の TLD ファイルでは、292 ページの「簡単なタグハンドラのコーディング」で示した SimpleTag クラスに対してタグを定義しています。

```
<?xml version="1.0" ?>
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.2</jspversion>
  <shortname>JRun Doc Samples</shortname>
  <tag>
    <name>hello</name>
    <tagclass>SimpleTag</tagclass>
    <bodycontent>JSP</bodycontent>
  </tag>
  <tag>
    <name>goodbye</name>
    <tagclass>SimpleTag</tagclass>
    <bodycontent>JSP</bodycontent>
  </tag>
</taglib>
```

属性の使用

属性を受け入れるタグハンドラをコーディングできます。属性はタグに代替値またはランタイム値を与えます。属性または属性を設定する式は、JSP で使用するときにタグに記述します。たとえば、<A> (アンカー) タグでは、次の行に示すように HREF が属性です。

```
<A HREF="http://www.hamsteak.com"> ここをクリックしてください。</A>
```

このセクションでは、タグハンドラの属性を処理する方法と、TLD ファイルを使用して属性に追加機能を提供する方法について説明します。

JSP での属性のコーディング

カスタムタグの一部として属性をコーディングします。次の例は、page および flush 属性を持つ test:include タグの呼び出しを示しています。

```
<html>
  <body>
    <%@ taglib prefix="test" uri="test.tld" %>
    <h1> パラメータを持つカスタムタグのテスト </h1>
    <test:include page="/includedText.htm" flush="true" />
  </body>
</html>
```

属性の使用

属性の機能を有効にするには、タグハンドラで次のことを定義する必要があります。

- 変数にアクセスするための、各属性のオブジェクトスコープ変数。
- 各属性の setter (必要な場合は getter) メソッド。setter メソッドの名前は、set で始まり、その後に先頭が大文字の変数名を付けます。たとえば、変数名が foo であれば、setFoo メソッドとなります。

必要であれば、TLD ファイルまたは TEI クラスを使用して、属性の使用方法と動作をカスタマイズできます。

- **TLD ファイル**　必須属性を指定する場合や、属性で JSP 実行時の式を利用する場合は、TLD ファイル内で属性を定義します。詳細については、298 ページの「TLD ファイルでの属性の定義」を参照してください。
- **TEI クラス**　属性をスクリプト変数として使用する場合や、isValid メソッドをオーバーライドして検証を有効にする場合は、TEI クラス内で属性を定義します。詳細については、309 ページの「TEI クラスのコーディング」を参照してください。

TLD ファイルの指定と TEI クラスファイルには、強力な検証機能があります。ただし、属性に必要な機能は、bean に類似した setter メソッドで、タグハンドラクラスのクラススコープ変数と対話することだけです。

attribute 要素は必須ではありませんが、次のような場合に便利です。

- **Required**　属性が必須の場合は、attribute 要素に次のコードを含めます。

  ```
  <required>true</required>
  ```

- **実行時の式**　属性をランタイムスクリプトレット式によって計算できる場合は、attribute 要素に次のコードを含めます。

  ```
  <rtexprvalue>true</rtexprvalue>
  ```

簡単な属性の例

次のタグ属性の使用を有効にするには

```
...
<test:hello username="Joe"/>
...
```

タグハンドラは、次のコードを実装する必要があります。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestParms extends BodyTagSupport {
  // 属性と同じ名前を持つオブジェクトスコープの変数
  String username;

  // JSP コンパイラによって呼び出される setVariablename メソッド
public void setUsername(String username) {
  this.username = username;
}

// オプションの getter メソッド
public String getUsername() {
  return username;
}
...
```

JSP でタグが呼び出されると、username 変数が開始タグに設定され、この変数を後続の他のタグハンドラのメソッドで使用できます。

完全な属性の例

サンプルのタグハンドラの次の完全な例では、属性を使用して、JSP の include アクション要素によって提供される機能をエミュレートしています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import javax.servlet.*;
import java.io.IOException;

public class TestInclude extends TagSupport {
  // デフォルトがないので必須です。
  String page;
  // デフォルトは true です。
  String flush = "true";
  // setter メソッドです。

 public void setPage(String page) {
  this.page = page;
 }

 public void setFlush(String flush) {
  // 小文字で保存します。
  this.flush = flush.toLowerCase();
 }
```

```
 public int doStartTag() throws JspException {
  // 本文テキストを無視します。
  return SKIP_BODY;
 }

  // doEndTag がすべての作業を行います。
 public int doEndTag() throws JspException {
  try {
    ServletContext sc = pageContext.getServletContext();
    RequestDispatcher rd = sc.getRequestDispatcher(page);
    if (rd !=null) {
      // アクセスのリクエストとレスポンス
      ServletRequest request = pageContext.getRequest();
      ServletResponse response = pageContext.getResponse();
      // 必要な場合は、バッファーを一括フラッシュします。
      if (flush.equals("true")) {
        pageContext.getOut().flush();
      }
      // ファイルをインクルードします。
      rd.include(request, response);
      return EVAL_PAGE;
    }
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  } catch (Exception e) {
    pageContext.getServletContext().log("Error with " + page, e);
  }
  return EVAL_PAGE;
 }
}
```

TLD ファイルでの属性の定義

属性はタグクラスハンドラで定義します。ただし、TLD ファイルで **attribute** 要素を使用して、属性に追加の設定を行うことができます。TLD ファイルでの属性の設定はオプションですが、設定することによって属性の必要条件を指定でき、属性の値として実行時の式を使用できるようにすることも、できないようにすることもできます。

次の表で、**attribute** 要素のサブ要素について説明します。

サブ要素	説明
name	属性名です。
required	属性が必須かどうかを示します。このサブ要素は、true または false に設定します。true の場合、JSP プログラマは JSP でタグを呼び出すときに属性を含める必要があります。
rtexprvalue	カスタムタグがこの属性の値として実行時の式を使用できるかどうかを示します。このサブ要素は、true または false に設定します。

次の TLD エントリの例では、必須属性とオプション属性を設定します。

```
<?xml version="1.0" ?>
<taglib>
  <tlibversion>0.0</tlibversion>
  <jspversion>1.0</jspversion>
  <shortname>test</shortname>
  <tag>
    <name>include</name>
    <tagclass>TestInclude</tagclass>
    <bodycontent>empty</bodycontent>
    <attribute>
      <name>page</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>flush</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>
  <tag>
    <name>hello</name>
    <tagclass>HelloTag</tagclass>
    <teiclass>HelloTEI</teiclass>
    <bodycontent>JSP</bodycontent>
  </tag>
</taglib>
```

本文コンテンツとの対話

Tag インターフェイス、IterationTag インターフェイス、および BodyTag インターフェイスを使用すると、テンプレートテキスト、JSP スクリプト要素、およびネストしたカスタムタグをカスタムタグの本文に含めることができます。カスタムタグが本文コンテンツとの対話を行わない場合は、TagSupport クラスを拡張し、doStartTag メソッドで EVAL_BODY_INCLUDE を返します。

ただし、カスタムタグが本文コンテンツのループ処理または変更を行う必要がある場合は、次のクラスのいずれかを拡張します。

- TagSupport　IterationTag を実装します。doAfterBody メソッドを提供します。タグハンドラが本文コンテンツのループ処理のみを行う必要がある場合は、このクラスを拡張します。
- BodyTagSupport　BodyTag を実装します。doInitBody メソッドと doAfterBody メソッドを提供します。タグハンドラが本文コンテンツを変更する必要がある場合は、このクラスを拡張します。doStartTag メソッドは、EVAL_BODY_INCLUDE と SKIP_BODY の他に EVAL_BODY_BUFFERED を返すことができます。

メモ：TLD ファイル内でカスタムタグの bodycontent 要素に empty を指定すると、本文コンテンツを無効にできます。タグハンドラは、doStartTag メソッドで SKIP_BODY を返すことによって、本文コンテンツを無視できます。

BodyContent オブジェクトについて

BodyContent オブジェクトは JspWriter のサブクラスです。JspWriter は、JSP の out 変数のために内部的に使用されるライターです。BodyContent オブジェクトは、bodyContent 変数を介して、doInitBody、doAfterBody、および doEndTag で使用できます (BodyContent オブジェクトと bodyContent 変数の、大文字の使用方法の違いに注意してください)。このオブジェクトのコンテンツは、doEndTag メソッドで元の JspWriter と統合できます。

BodyContent オブジェクトには、出力を書き出すために使用するメソッドのほかに、このオブジェクトのコンテンツの読み取り、クリア、および取り出しを行うメソッドが含まれています。たとえば、bodyContent.getString を使用すると、ライターのコンテンツを取り出せるだけでなく、必要であれば、そのコンテンツを元の JspWriter と統合する前に変更できます。

メモ：bodyContent 変数を使用する前に、その値が null でないことを確認します。

doInitBody メソッドについて

doStartTag メソッドが EVAL_BODY_BUFFERED を返した場合、JRun は doInitBody メソッドを呼び出します。このメソッドは、必要に応じて本文コンテンツの初期化に使用します。次にタグハンドラが本文を処理し、JRun が doAfterBody メソッドを呼び出します。

doAfterBody メソッドについて

doAfterBody メソッドは、SKIP_BODY または EVAL_BODY_AGAIN を返します。EVAL_BODY_AGAIN を返した場合、JRun はループに戻って本文を再実行します。これにより、リストやデータベースの結果セットなどの反復データ全体を繰り返し処理できます。

簡単な例

次の例は、doInitBody と doAfterBody メソッドの使用方法を示しています。また、bodyContent 出力を出力ストリームに統合する方法も示しています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestBody extends BodyTagSupport {

 public int doStartTag() throws JspException {
  try {
    pageContext.getOut().print("<h2>doStartTag() を使用しています。
        </h2>");
   return EVAL_BODY_BUFFERED;
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
 }

 public void doInitBody() throws JspException {
  try {
    // これは、doStartTag や doEndTag のライターとは違うことに
    // 注意してください。
    bodyContent.print("<h2>doInitBody() を使用しています。</h2>");
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
 }

 public int doAfterBody() throws JspException {
  try {
    // これは、doStartTag や doEndTag のライターとは違うことに
    // 注意してください。
    bodyContent.print("<h2>doAfterBody() を使用しています。</h2>");
    // return IterationTag.EVAL_BODY_AGAIN; // これを使用してループ処理します。
    return SKIP_BODY;
  } catch(IOException ioe) {
     throw new JspException(ioe.getMessage());
  }
 }

public int doEndTag() throws JspException {
  try {
    // bodyContent ライターから元のライターに書き込みます。
    pageContext.getOut().print(bodyContent.getString());
    //// バッファーが大きい場合は、前の行よりも
```

```
    //// 次のコードの方がより効率的です。
    //// 元の（囲まれている）ライターを取得します。
    // JspWriter jOut = bodyContent.getEnclosingWriter();
    //// 前のライターに本文の出力を追加します。
    //bodyContent.writeOut(jOut);
    // ここで元のライターに戻ります。
    pageContext.getOut().print("<h2>doEndTag() を使用しています。</h2>");
    return EVAL_PAGE;
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
 }
}
```

ループの例

doAfterBody が EVAL_BODY_AGAIN を返すようにコーディングすることによって、カスタムタグの本文を繰り返し実行するループを作成できます。

メモ：カスタムタグとループで使用されるスクリプト変数のスコープは、TLD ファイルまたは TEI クラスによって制御されます。詳細については、306 ページの「スクリプト変数の使用」を参照してください。

次の JSP の例は、タグハンドラを呼び出しています。

```
<html>
  <body>
  <%@ taglib prefix="test" uri="DocSamples.tld" %>
    <h1> ヘッダーのループ </h1>
    <table border="1">
      <tr>
        <th> 名前 </th>
        <th> 値 </th>
      </tr>
      <test:enumloop thisEnum="<%= request.getHeaderNames() %>">
        <tr>
          <% String header =
        (String)pageContext.getAttribute("nextElement");%>
          <td><%= header %></td>
          <td><%= request.getHeader(header) %></td>
        </tr>
      </test:enumloop>
    </table>
  </body>
</html>
```

次のタグハンドラは、Enumeration タイプの属性を受け入れて、Enumeration オブジェクト内のそれぞれの名前と値 (この例では HTTP ヘッダー) をループ処理します。

```
import java.util.Enumeration;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;
```

```
public class TestBodyLoopHeaders extends BodyTagSupport {
  Enumeration thisEnum;

  public void setThisEnum(Enumeration passedEnum) {
    this.thisEnum = passedEnum;
  }

  public int doStartTag() throws JspException {
    return EVAL_BODY_INCLUDE;
  }

  public void doInitBody() throws JspException {
    if (thisEnum.hasMoreElements()) {
      pageContext.setAttribute("nextElement",
        thisEnum.nextElement());
    }
  }

  public int doAfterBody() throws JspException {
    if (thisEnum.hasMoreElements()) {
      pageContext.setAttribute("nextElement",
        thisEnum.nextElement());
      return EVAL_BODY_AGAIN;  // ループ
    }else {
      return SKIP_BODY;
    }
  }

  public int doEndTag() throws JspException {
    try {
      // bodyContent ライターから元のライターに書き込みます。
      pageContext.getOut().print(bodyContent.getString());
      return EVAL_PAGE;
    } catch(IOException ioe) {
      throw new JspException(ioe.getMessage());
    }
  }
}
```

ネストしたタグハンドラのコーディング

Tag インターフェイスまたは BodyTag インターフェイスのどちらを実装している場合でも、カスタムタグをネストできます。タグをネストすると、ネストしたタグに関するタグハンドラは、findAncestorWithClass メソッドによって親クラスへのリファレンスを取得できます。ネストしたタグハンドラは、このリファレンスを親クラスにキャスティングすることによって、親クラス内のメソッドを呼び出せます。たとえば、親クラスはネストしたタグハンドラで出力ストリームを書き出すメソッドを実装する場合があります。

次の JSP の例は、bodyparent という親タグと bodynest というネストしたタグを使用しています。

```
<html>
  <body>
  <%@ taglib prefix="test" uri="DocSamples.tld" %>
    <h1> ネストしたカスタムタグのテスト </h1>
      <test:bodyparent name="Johnson">
```

```
        <test:bodynest name="Lorna"/>
        <test:bodynest name="Gretchen"/>
        <test:bodynest name="Brian"/>
      </test:bodyparent>
  </body>
</html>
```

次のコードは、親タグに関するサンプルタグハンドラを示したものです。このサンプルには、ネストしたタグに関するタグハンドラで出力ストリームを更新するときに呼び出すことができるメソッドが含まれています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestBodyParent extends TagSupport {
  String name;

public void setName(String name) {
  this.name = name;
}

public int doStartTag() throws JspException {
  try {
    // まず、親の名前をプリントします。
    pageContext.getOut().print("<h1> 親：" + name + "</h1>");
    return EVAL_BODY_INCLUDE;
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}

public int doEndTag() throws JspException {
  try {
    // グループの後にルーラを追加します。
    pageContext.getOut().print("<hr>");
    return EVAL_PAGE;
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}

public void setNestedName(String name) throws JspException {
  try {
    // ネストされている名前をプリントします。
    pageContext.getOut().print("<p> ネスト：" + name);
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}
}
```

次のコードは、ネストしたタグに関するサンプルタグハンドラを示したものです。このサンプルでは、親のタグハンドラのメソッドを呼び出して、出力ストリームを更新します。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestBodyNest extends TagSupport {
  private String name;
  private TestBodyParent parent = null;

public void setName(String name) {
  this.name = name;
}

public int doStartTag() throws JspException {
  // 親へのリファレンスを検索して保存します。
  Tag t = findAncestorWithClass(this, TestBodyParent.class);
  if (t == null) {
    throw new JspException("TestBodyNest must be in TestBodyParent.");
  } else {
    parent = (TestBodyParent)t;
    return EVAL_BODY_INCLUDE;
  }
}

public int doEndTag() throws JspException {
  // 名前を親にコピーします。
  parent.setNestedName(name);
  return EVAL_PAGE;
}
}
```

スクリプト変数の使用

スクリプト変数は、ユーザーがページをリクエストするときに JSP で使用できる変数です。宣言、スクリプトレット、または式の中のスクリプト変数にアクセスできます。

カスタムタグでスクリプト変数を定義できます。スクリプト変数は、JSP 内のスクリプトレットや他のカスタムタグで使用できます。スクリプト変数は、JSP の pageContext オブジェクトにオブジェクトとして保管されます。その結果、スクリプト変数を Strings および Integers として設定できるだけでなく、データベース接続または他の直列化可能なオブジェクトとしても設定できます。

カスタムタグにスクリプト変数を設定するには、2 つの方法があります。

- JSP 1.1 では、TEI ファイルで変数を定義し、TEI クラスを実装し、タグハンドラでスクリプト変数を定義する必要があります。
- JSP 1.2 では、タグハンドラで変数を定義し、スクリプト変数を設定する TLD ファイルに `variable` 要素を追加する必要があります。

どちらの方法でも、JSP でカスタムタグスクリプト変数を使用可能にできますが、JSP 1.2 の方法の方が簡単に実装できます。ただし、多くの既存のカスタムタグライブラリでは JSP 1.1 の方法を使用しているので、JSP 1.1 の方が柔軟性はあります。

このセクションでは、カスタムタグ内でスクリプト変数を使用する 2 つの方法を説明します。

メモ: タグ属性 (ID 属性など) を使用してスクリプト変数を定義できますが、タグ属性とスクリプト変数は直接は関係ありません。

JSP 1.2 でのスクリプト変数の使用

JSP 1.2 では、JSP ページのカスタムタグのスクリプト変数を使用する簡単な方法を導入しています。次のタスクを実行する必要があります。

- タグハンドラにスクリプト変数を定義します。
- TLD ファイルに `variable` 要素を追加します。

次のセクションでは、これらのタスクについて説明します。

タグハンドラでのスクリプト変数の有効化

タグハンドラは pageContext オブジェクトにスクリプト変数を追加します。タグハンドラでは、スクリプト変数に指定されているスコープに応じて、これらの変数をさまざまな方法で定義します。さらに、本文テキストをループ処理するタグハンドラでは、`doAfterBody` メソッドによって、スクリプト変数を更新またはリセットできます。

次の例は、タグハンドラがスクリプト変数を設定する方法を示しています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class HelloTag extends BodyTagSupport {
  String to;

public void setTo(String to) {
  this.to = to;
}
```

```
public int doStartTag() throws JspException {
  try {
    pageContext.getOut().print("Hello " + to);

    // AT_BEGIN として定義されるので、ここで定義します。
    // doInitBody 内で定義し、ループの場合は
    // doAfterBody 内で変更またはリセットできます。
    pageContext.setAttribute("foo", "foo");
    return EVAL_BODY_INCLUDE;
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}

public void doInitBody() throws JspException {
  // NESTED として定義されるので、ここで定義します。
  // doStartTag 内で定義し、ループの場合は
  // doAfterBody 内で変更またはリセットできます。
  pageContext.setAttribute("baz", "baz");
}

public int doEndTag() throws JspException {
  try {
    // このタグハンドラによって BodyTag が実装されるので
    // (BodyTagSupport を拡張することによって )、
    // 本文のライターと元のライターを統合する必要があります。
    pageContext.getOut().print(bodyContent.getString());

    // AT_END として定義されるので、
    // doEndTag まで自分で定義する必要はありません。
    pageContext.setAttribute("bar", "bar");
    return EVAL_PAGE;
  } catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}
}
```

TLD ファイルでのスクリプト変数の追加

tag 要素の variable サブ要素にスクリプト変数を定義します。次の表で、variable 要素のサブ要素について説明します。

tag のサブ要素	説明
name-given	スクリプト変数名。JSP で変数にアクセスする際に使用します。
name-from-attribute	スクリプト変数の代替名。変数のトランスレート時に名前を設定します。
variable-class	オプション。変数の完全修飾クラス。 たとえば、java.lang.Double。 デフォルトは java.lang.String です。

tag のサブ要素	説明
declare	オプション。新しい変数を作成するかどうかを決定する Boolean 設定。デフォルトは true です。
scope	スクリプト変数のスコープ。次の 3 つの値のいずれかに設定できます。 • **AT_BEGIN**　スコープを AT_BEGIN に設定すると、カスタムタグの本文か、タグの本文の外にあるタグの後の JSP ページのいずれかで発生する任意の宣言、式、またはスクリプトレットでスクリプト変数を使用可能にできます。 • **AT_END**　スコープを AT_END に設定すると、カスタムタグの後の JSP で発生する任意の宣言、式、またはスクリプトレットでスクリプト変数を使用可能にできます。このタグはタグの本文内では使用できませんが、その後で宣言されます。 • **NESTING**　スコープを NESTING に設定すると、JSP 内のカスタムタグの本文でのみスクリプト変数を使用可能にできます。このメソッドによって最少量のリソースが使用されます。

次の例は、カスタムタグのスクリプト変数を定義する TLD ファイルを示しています。

```
<?xml version="1.0" ?>
<taglib>
...
  <tag>
...
    <variable>
      <name-given>foo</name-given>
      <name-from-attribute>foo</name-from-attribute>
      <variable-class>java.lang.String</variable-class>
      <declare>true</declare>
      <scope>AT_BEGIN</scope>
    </variable>
    <variable>
      <name-given>bar</name-given>
      <name-from-attribute>bar</name-from-attribute>
      <variable-class>java.lang.String</variable-class>
      <declare>true</declare>
      <scope>AT_END</scope>
    </variable>
```

```
    <variable>
      <name-given>baz</name-given>
      <name-from-attribute>baz</name-from-attribute>
      <variable-class>java.lang.String</variable-class>
      <declare>true</declare>
      <scope>NESTED</scope>
    </variable>
...
  </tag>
</taglib>
```

JSP 1.1 でのスクリプト変数の使用

JSP 1.1 で JSP ページのカスタムタグのスクリプト変数を使用するには、次のタスクを実行する必要があります。

- タグハンドラの pageContext オブジェクトにスクリプト変数を追加します。詳細については、306 ページの「タグハンドラでのスクリプト変数の有効化」を参照してください。
- TEI クラスでスクリプト変数を定義します。詳細については、309 ページの「TEI クラスのコーディング」を参照してください。

このセクションで説明した JSP 1.1 の方法でスクリプト変数を使用すれば、TEI クラスを拡張することによってタグを検証できます。このようなタグの検証が不要な場合は、306 ページの「JSP 1.2 でのスクリプト変数の使用」で説明した JSP 1.2 の方法でスクリプト変数を使用します。

TEI クラスのコーディング

カスタムタグでスクリプト変数を作成する場合、Java 開発者は TEI ファイルも作成する必要があります。TEI ファイルは、JSP コードで使用するスクリプト変数とそのスコープを定義する Java クラスです。TEI ファイルを使用して、トランスレート時に属性を検証することもできます。

TEI ファイルは、`TagExtraInfo` クラスを拡張する Java クラスです。
`getVariableInfo` メソッドには、次の署名があります。

```
public VariableInfo[] getVariableInfo(TagData tagData) { }
```

`tagData` パラメータには、属性の名前 / 値のペアが格納されています。この名前 / 値のペアを使用してスクリプト変数を定義します。たとえば、`useBean` タグは、`id` 属性を使用してスクリプト変数を作成します。

`getVariableInfo` メソッド内で、VariableInfo オブジェクトの配列を作成し、1 つのスクリプト変数につき 1 つの VariableInfo オブジェクトをその配列に挿入します。次の表で、VariableInfo オブジェクトのコンストラクタが取るパラメータについて説明します。

パラメータ	説明
`varName`	スクリプト変数名を指定する String 型パラメータ。
`className`	スクリプト変数のクラスを指定する String 型パラメータ。

パラメータ	説明
declare	コンストラクタで新規の変数を宣言するかどうかを指示する Boolean 型パラメータ。
scope	スクリプト変数のスコープを指定する Int 型パラメータ。次の表に示すように AT_BEGIN、NESTED、または AT_END を指定します。

同期化とは、ページコンテキストからオブジェクトを取り出して、それをスクリプト変数に割り当てる操作です。次の表で、getVariableInfo メソッドで指定した変数に関するスコープ、使用方法、および同期化について説明します。

スコープ	JSP での使用方法	JSP の同期化	設定およびリセットされる場所
AT_BEGIN	タグの本文内と、JSP の残りの部分	このスクリプト変数は、本文を反復するたびにリセットされます。	doInitBody と doAfterBody (BodyTag を実装している場合)。それ以外の場合は doStartTag または doEndTag。
NESTED	タグの本文内	このスクリプト変数は、本文を反復するたびにリセットされます。	doInitBody と doAfterBody (BodyTag を実装している場合)。それ以外の場合は、doStartTag。
AT_END	JSP の残りの部分	この変数は、本文の実行が完了した後でリセットされます。	doEndTag の後

スクリプト変数の使用例

次の例は、3 つのスクリプト変数を定義する TEI クラスに関する Java コードを示しています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class HelloTEI extends TagExtraInfo {
public VariableInfo[] getVariableInfo(TagData tagData) {
  VariableInfo[] vars = new VariableInfo[3];

  // ページの本文と残りの部分で使用できます。
  vars[0] = new VariableInfo("foo", "java.lang.String", false,
        variableInfo.AT_BEGIN);

  // ページの残りの部分のカスタムタグの後ろで使用できます。
  vars[1] = new VariableInfo("bar", "java.lang.String", true,
        VariableInfo.AT_END);
```

```
  // タグ本文でのみ使用できます。
  vars[2] = new VariableInfo("baz", "java.lang.String", true,
        VariableInfo.NESTED);
  return vars;
}
}
```

次の JSP の例では、TEI クラスで定義したスクリプト変数を使用しています。

```
<%@ taglib prefix="test" uri="test.tld" %>
<% String foo; %>

<test:hello to="World">
  <%-- baz は NESTED です（本文でのみ使用可能）。--%>
  <%= baz %>
  <%-- foo は AT_BEGIN です（本文とそれ以外でも使用可能）。--%>
  <%= foo %>
</test:hello>
<%= foo %>
<%-- bar は AT_END です（本文の後ろでのみ使用可能）。--%>
<%= bar %>
```

属性の検証

TEI クラスは、isValid メソッドをオーバーライドしてタグ特有の属性の検証を実施できます。トランスレート時に、JRun から isValid メソッドに TagData インスタンスが渡されます。

isValid メソッドは、TagData.getAttribute を呼び出して、その値にアクセスできます。TLD ファイル内の属性の定義で実行時の式を有効にしている場合は、TagData.REQUEST_TIME_VALUE オブジェクトを調べることができます。このオブジェクトは、カスタムタグの呼び出しが実行時の式を使用していることと、検証が不可能であることを示します。実行時の式の詳細については、298 ページの「TLD ファイルでの属性の定義」を参照してください。

次のクラスの例では、バージョン属性の値が 5 より小さいことを確認します。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestIsValidTEI extends TagExtraInfo {
  // この例では、スクリプト変数が 1 つであると想定しています
  // （この例とは無関係）。

public VariableInfo[] getVariableInfo(TagData tagData) {
  VariableInfo[] vars = new VariableInfo[1];
  vars[0] = new VariableInfo("foo", "java.lang.String", true,
        VariableInfo.AT_BEGIN);
  return vars;
}
```

```
public boolean isValid(TagData data) {
  Object version = data.getAttribute("version");
  // この属性では実行時の式が使用できるので、
  // REQUEST_TIME_VALUE をチェックする必要があります。
  if (version != null && version != TagData.REQUEST_TIME_VALUE) {
    int iVersion = Integer.parseInt((String)version);
    // バージョンは 5 以下です。
    if (iVersion > 5) {
      return false;
    }else {
      return true;
    }
  }else {
    return false;
  }
}
}
```

タグライブラリの検証

タグライブラリのバリデータクラスについては JSP 1.2 で紹介しました。タグライブラリの検証は次の理由から比較的簡単に実装できます。

- JSP は、サーブレットにコンパイルされる前に XML ビューに変換されます。

  JRun で JSP ページの XML ビューが作成される方法については、56 ページの「XML ビューについて」を参照してください。
- XML では形式の整ったコードを使用する必要があります。

タグライブラリのバリデータを使用するには

1 次のセクションで説明するように、バリデータクラスを作成します。

2 `validator` サブ要素を TLD ファイルの `taglib` 要素に追加します。この要素はバリデータのクラスファイルを指定します。

  `validator` サブ要素の使用の詳細については、294 ページの「TLD ファイルの作成」を参照してください。

3 Web アプリケーションの /WEB-INF/classes ディレクトリにバリデータクラスをパッケージします。

4 タグライブラリを使用する JSP をリクエストします。

初めて JSP ページをリクエストしてコンパイルする場合、JRun では JSP ページのタグライブラリのバリデータが呼び出されます。その後、Web アプリケーションの /WEB-INF/jsp ディレクトリにある JSP ファイルを変更するか、または JSP クラスファイルを削除して、ページを再検証します。

バリデータの作成

タグライブラリのバリデータは `javax.servlet.jsp.tagext.TagLibraryValidator` クラスを拡張します。次の行に示すように、クラスに `tagext` パッケージを含める必要があります。

```
import javax.servlet.jsp.tagext.*;
```

`TagLibraryValidator` クラスは独自の検証は行いませんが、オーバーライド可能なメソッドを示して検証を行います。バリデータに機能を与えるには、`TagLibraryValidator` クラスの次のメソッドをオーバーライドします。

- `ValidationMessage[] validate(java.lang.String prefix, java.langString uri, PageData page)`
- `setInitParameters(java.util.Map map)`
- `getInitParameters()`
- `void release()`

TLD ファイルにバリデータを追加すると、JRun はそのバリデーションクラスのメソッドを呼び出し、値を渡します。JRun は、JSP ページの各ディレクティブに対して `validate` メソッドを 1 回呼び出します。他のメソッドは、ページのリクエストごとに 1 回だけ呼び出されます。

ほとんどのタグバリデータは `validate` メソッドをオーバーライドします。ページに無効なタグが含まれていなければ、このメソッドは `null` を返します。それ以外の場合は、`validate` メソッドは ValidationMessage オブジェクトの配列を返します。

validate メソッドが使用した PageData オブジェクトには、次のメソッドがあります。

getInputStream()

getInputStream メソッドは、JSP ページの XML ビューの入力ストリームを返します。

setInitParameter メソッドと getInitParameter メソッドは、TLD ファイルの validator 要素の init-params サブ要素に設定された初期化パラメータと対話します。

release メソッドはリソースを解放する場所を提供します。

タグライブラリのパッケージング

アプリケーション開発者は、タグライブラリを JSP 開発者に配信します。デプロイ可能なタグライブラリは、TLD ファイル、タグハンドラ、TEI クラス、タブライブラリのバリデータ、および JAR ファイルへの他のサポートファイルを含んでいる JAR ファイルです。

メモ：JAR ファイルを別の場所に保管し、Web アプリケーションの仮想マッピングをその場所に確立することもできます。この場合は、複数の Web アプリケーションで 1 つのタグライブラリを共有できます。ただし、Web アプリケーションをデプロイ用にパッケージングするときは、その JAR ファイルを /WEB-INF/lib に移動します。詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

タグライブラリ JAR ファイルを持つ次のドキュメントを、次の項目とともに含めます。

- 各タグの名前
- 属性のステータス (必須またはオプション)
- 実行時の式を受け入れる属性のステータス
- 本文コンテンツを使用できるタグと、使用できないタグの指示
- 必須の親タグ、ネストしたタグ、または本文コンテンツ

JSP 仕様では、TLD ファイルを、タグライブラリの JAR ファイルの一部として /META-INF ディレクトリに保管することを推奨しています。JAR ファイルに TLD ファイルを保管しない場合は、TLD ファイルが直接表示されないように、WEB-INF ディレクトリの下に保管してください。

web.xml ファイル内で taglib 要素を定義するには

1 先頭の taglib 要素をコーディングします。

```
<taglib>
```

2 taglib-uri 要素をコーディングします。

この要素では、JSP 開発者が JSP taglib ディレクティブで使用している名前を指定します。

```
<taglib-uri>/eisTaglib</taglib-uri>
```

3 taglib-location 要素をコーディングします。

この要素は、TLD ファイルへのパスを指定します。

```
<taglib-location>/WEB-INF/tldHome/eisTaglib.tld</taglib-location>
```

4 終了を示す taglib 要素をコーディングします。

```
</taglib>
```

JSP 開発者がこの例のタグライブラリを参照するときは、次の taglib ディレクティブを使用します。

```
<%@ taglib prefix="eis" uri="/eisTagLib" %>
```

第 12 章
JSP カスタムタグのコーディング

JavaServer Pages の仕様書には、Java で作成されたタグハンドラを使用した、タグライブラリに関するフレームワークが記述されています。この章では、JSP でカスタムタグハンドラをコーディングする方法について説明します。

目 次

JSP カスタムタグの概要

関連する機能のセットのカプセル化、ロジックからプレゼンテーションの分離、および JSP の操作性の強化を行うには、カスタムタグを使用します。JRun は、20 個以上のカスタムタグのコレクションである JRun タグライブラリを使用して、エンジン内でこの機能をサポートする最初の製品の 1 つでした。

初めに考えられたように、カスタムタグハンドラは Java で書かれており、Tag および BodyTag インターフェイスを基にした 1 つまたは複数のメソッドを実装して、タグが呼び出されたときに適切な処理を実行します。さらに、Java 開発者は、タグライブラリディスクリプタ (TLD) ファイルに要素をコーディングし、タグ拡張情報 (TEI) クラスをコーディングして機能を追加する必要があります。カスタムタグのコーディングには、Java、カスタムタグ API、および JSP による開発経験が必要です。Java ベースのカスタムタグハンドラのコーディングについては、JSP の仕様書または 285 ページの第 11 章「Java のカスタムタグ」を参照してください。

JRun サーバータグ (JST) は、JSP にカスタムタグハンドラを実装しています。Java コードは必要ありません。JST テクノロジーでは従来の JSP シンタックスが使用されているので、JSP プログラマーはカスタムタグを利用できます。JST を使用すると、Java で書かれたカスタムタグに比べて短期間でアプリケーションを開発できます。ページが JST ページであることを示すには、ページの名前に拡張子 .jst を付けます。

JST と Java ベースのタグハンドラには、JSP とサーブレットの場合と同じ関係があります。つまり、サーブレットは Web アプリケーションプログラミングに対して Java ベースの強力なソリューションを提供し、JSP はその機能を損ねることなく操作性を向上させる、あるレベルのアブストラクションを実装します。また、JRun によって JSP がサーブレットにトランスレートされるように、JST ページは Java ベースのタグハンドラにトランスレートされます。このテクノロジーによって生成されたタグハンドラクラスは、TLD ファイルを指定するだけで、JSP カスタムタグをサポートするすべてのサーブレットエンジンに移植可能です。

JST とカスタムタグの比較

JST を使用すると Java のコーディングから解放されますが、JST ページをコーディングするには、基本的なカスタムタグの概念を認識している必要があります。次の表は、JST とカスタムタグハンドラの違いを要約したものです。

機能	カスタムタグハンドラ	JST
カスタムタグの実装	タグハンドラをコーディングし、TLD ファイルのタグ、TEI クラス、およびデプロイメントディスクリプタを定義します。	ファイルを .jst 形式で保存します。
呼び出しポイント	この機能を実装するには、`doStart`、`doEnd`、および `doAfterBody` メソッドをコーディングします。	<tag method= "START|END|AFTER_BODY" %> ディレクティブにより制御されるメソッドの機能が、自動的に実装されます。他のメソッドは、宣言内でメソッドを Java でコーディングすることによって実装できます。
本文コンテンツとの対話	この機能を実装するには、`BodyTagSupport` クラスを拡張し、`doAfterBody` メソッドをコーディングし、`bodyContent` 変数によって本文テキストにアクセスします。	`<%@ tag type="LOOPING|BUFFERED" %>` ディレクティブをインクルードすることによって、このタイプの処理を実行します。
戻りコード	戻り定数を使用します。	`returnCode` という暗黙の変数を使用します。
属性	TLD ファイルで各タグの属性を宣言します。	`tagAttribute` ディレクティブによって属性を宣言します。
スクリプト変数	TEI クラスまたは TLD ファイルでスクリプト変数を宣言します。	`tagVariable` ディレクティブによってスクリプト変数を宣言します。また、`isValid` メソッドを定義する宣言を使用して、属性の検証を実行することもできます。

次の表は、Java ベースのカスタムタグハンドラと JSP で書かれたカスタムタグの違いを要約したものです。

Java ベースのカスタムタグハンドラ	JSP ベースのカスタムタグ
タグハンドラ Java ソースファイル	JSP シンタックスによる JST ファイル
`doStartTag` メソッド	`<%@ tag method="START" %>`
`doEndTag` メソッド	`<%@ tag method="END" %>`
`doAfterBody` メソッド	`<%@ tag method="AFTER_BODY" %>`
`TagSupport` クラスから継承します。	`<%@ tag type="TAG" %>`
`BodyTagSupport` クラスから継承します。	`<%@ tag type="LOOPING|BUFFERED" %>`
タグハンドラ内に複数のメソッドを実装します。	最初のメソッドは `<% tag method="method-type" %>` を使用して実装します。 その他のメソッドは、宣言に囲まれた Java ベースのメソッドを使用して実装します。

TLD ファイルで属性を定義します。	`<%@ tagAttribute name="name"`
	`type="type"`
	`required="true|false"`
	`rtexpr="true|false"`
	`setter="true|false"`
	`default="default attribute value" %>`

TEI クラスまたは TLD ファイルでスクリプト変数を定義します。	`<%@ tagVariable (id="id"|name="name")`
	`type="type"`
	`scope="AT_BEGIN|AT_END|NESTED" %>`

これらのトピックの詳細については、285 ページの第 11 章「Java のカスタムタグ」を参照してください。

この章の後続のセクションでは、JST のコーディング開始に役立つ情報について説明します。

JST の使用

JST を使用するには、拡張子 .jst を持つ JSP ファイルを保存します。JRun では、ページを、**end** タグで呼び出されるカスタムタグとして処理します。これによって、コードの再利用に便利なメカニズムを提供します。ただし、JST テクノロジーには、JSP の仕様書に概説されている次の機能があります。

- 開始タグと終了タグでの呼び出し
- 属性
- 本文コンテンツとの対話
- ループ
- スクリプト変数

JSP でカスタムタグを作成するには、JSP の標準的なディレクティブについて、修正されたシンタックスまたは新しいシンタックスを使用する必要があります。次のセクションでは、新しいシンタックスについて説明します。

- 319 ページの「tag ディレクティブの使用」
- 321 ページの「JST での属性の使用」
- 322 ページの「JST でのスクリプト変数の使用」
- 323 ページの「URI の定義」

tag ディレクティブの使用

tag ディレクティブを使用して、次のタイプの情報を宣言します。

- **Method**　JST ファイルが **begin** タグで呼び出されるか、**end** タグで呼び出されるか、または本文コンテンツを反復した後で呼び出されるかを指定します。**method** 属性を持つ **tag** ディレクティブを指定します。
- **Type**　タグと本文コンテンツが対話するかどうかを指定します。**type** 属性を持つ **tag** ディレクティブで指定します。

method 属性を持つ tag ディレクティブ

JST がいつ呼び出されるかを示すには、method 属性を持つ tag ディレクティブをコーディングします。次のシンタックスを使用します。

```
<%@ tag method="method" %>
```

次の表で、tag ディレクティブ内の method 属性の値について説明します。

値	説明
START	開始タグが検出されたときに JST ページを呼び出します。START の許容可能な戻りコードは、EVAL_BODY_INCLUDE および SKIP_BODY です。
END	終了タグが検出されたときに JST ページを呼び出します。END の許容可能な戻りコードは、EVAL_PAGE および SKIP_PAGE です。
AFTER_BODY	本文の後に JST ページを呼び出します。AFTER_BODY の許容値は、本文のループに使用する EVAL_BODY_AGAIN、および SKIP_BODY です。AFTER_BODY と END の違いは、許容可能な戻り値にあります。また、AFTER_BODY を指定した場合は、<%@ tag type="LOOPING|BUFFERED" %> も指定する必要があります。

tag ディレクティブはオプションです。デフォルトは END です。

メソッドおよび戻りコードの詳細については、JSP の仕様書または 285 ページの第 11 章「Java のカスタムタグ」を参照してください。

type 属性を持つ tag ディレクティブ

JST が本文コンテンツと対話するかどうかを示すには、type 属性を持つ tag ディレクティブをコーディングします。次のシンタックスを使用します。

```
<%@ tag type="type" %>
```

次の表で、type 属性の値について説明します。

値	説明
LOOPING	JST は、BodyTag インターフェイスを実装するカスタムタグにトランスレートされます。BODY_TAG を使用すると、本文コンテンツと対話してループできます。LOOPING は、本文を反復しますが本文コンテンツを返しません。
BUFFERED	JST は、BodyTag インターフェイスを実装するカスタムタグにトランスレートされます。BODY_TAG を使用すると、本文コンテンツと対話してループできます。BUFFERED は、本文コンテンツをループして返します。

<%@ tag method="AFTER_BODY" %> を指定するときに、このディレクティブをコーディングします。

JST での属性の使用

JST ページに渡される属性を定義するには、次のシンタックスを使用して tagAttribute ディレクティブをコーディングします。

```
<%@ tagAttribute name="name" type="class" required="true|false"
                 rtexpr="true|false" setter="true|false"
                 default="default attribute value" %>
```

次の表で、tagAttribute ディレクティブの属性について説明します。

属性	説明
name	必須。属性名を指定します。
type	オプション。たとえば Integer など、属性の Java クラス名を指定します。インポートステートメントをコーディングしていない場合や、クラスが java.lang でない場合は、完全修飾クラス名を指定します。デフォルトは String です。
required	オプション。属性が必須かどうかを示します。true または false を指定します。デフォルトは false です。
rtexpr	オプション。属性に実行時の式を含めることができるどうかを示します。true または false を指定します。デフォルトは false です。
setter	オプション。JRun によって get*AttributeName* メソッドを自動的に作成する場合は true、カスタマイズされた get*AttributeName* メソッドを指定する場合は false を指定します。デフォルトは true です。
デフォルト	オプション。属性のデフォルト値を指定します。

属性の例

次の例は、handle という名前の属性の宣言を示しています。

```
<%@ tagAttribute name="handle" type="String" required="true"
        rtexpr="true" %>
```

この tagAttribute ディレクティブによって、JST ページで次の処理が実行されます。

- handle と呼ばれる専用のインスタンス変数を作成します。
- setHandle(java.lang.String) と呼ばれるメソッドを作成します。
- TagLibraryInfo.getTag メソッドによって返される TagInfo オブジェクトに属性を組み込みます。

JST でのスクリプト変数の使用

スクリプト変数は、ユーザーがページをリクエストする際に JSP で使用できる変数です。スクリプト変数は、宣言、スクリプトレット、または式で利用できます。JST でスクリプト変数を定義するには、tagVariable ディレクティブをコーディングします。次のシンタックスを使用します。

```
<%@ tagVariable (id="attribute"|name="name") type="class"
        scope="scope" %>
```

次の表で、tagVariable ディレクティブの属性について説明します。

属性	説明
id	スクリプト変数の名前を値として持つタグ属性の名前を指定します。id 属性または name 属性のいずれかを指定する必要があります。両方を指定することはできません。
name	name 属性ではスクリプト変数の名前を指定します。id 属性または name 属性のいずれかを指定する必要があります。両方を指定することはできません。
type	オプション。たとえば Integer などのスクリプト変数の Java クラス名を指定します。インポートステートメントをコーディングしていない場合や、クラスが java.lang でない場合は、完全修飾クラス名を指定します。デフォルトは String です。
scope	オプション。スクリプト変数の範囲を指定します。次のいずれかを指定できます。 • AT_BEGIN　タグの本文内と JSP の残りの部分でスクリプト変数を使用できます。デフォルトは AT_BEGIN です。 • AT_END　JSP の残りの部分でスクリプト変数を使用できます。 • NESTED　タグの本文内でスクリプト変数を使用できます。

scope 属性を使用する際の追加事項については、JSP 1.1 のメソッドを使用している場合は 309 ページの「TEI クラスのコーディング」、JSP 1.2 のメソッドを使用している場合は 306 ページの「JSP 1.2 でのスクリプト変数の使用」を参照してください。

スクリプト変数の例

次の例では、tagVariable ディレクティブを使用して、filename という名前のスクリプト変数を作成します。この変数は、タグの本文内と、JSP の残りの部分で使用できます。

```
<%@ tagVariable name="filename" type="String" scope="AT_BEGIN" %>
```

URI の定義

JST を使用する JSP をコーディングするときは、Java ベースのカスタムタグを使用する JSP のコーディングの場合と同様に、必要なのは taglib ディレクティブを指定することです。唯一の違いは、uri 属性で、JAR ファイルの名前ではなく、.jst ファイルが保存されているディレクトリを指定することです。

次のシンタックスを使用します。

```
<%@ taglib prefix="prefix" uri="directory_containing_jst_pages" %>
```

次の表で、taglib ディレクティブの属性について説明します。

属性	説明
prefix	必須。JSP 開発者が、uri 属性で指定されたディレクトリから JST を呼び出すときに使用する接頭辞を指定します。
uri	必須。JST ページが含まれているディレクトリの名前を指定します。これは Web アプリケーションのルートを基にしています。

URI の例

次の例は、myjst という接頭辞を使用して、JST ページが web_app_root/jstdir ディレクトリにあることを指定します。

```
<%@ taglib prefix="myjst" uri="/jstdir" %>
```

JST の例

このセクションでは、JST の例をいくつか説明します。このセクションの例を使用する場合は、次のことを行う必要があります。

- JST ファイルに拡張子 .jst を付けて、Web アプリケーションのルートディレクトリに保存します。
- JSP に拡張子 .jsp を付けて、Web アプリケーションのルートディレクトリに保存します。
- JSP ページの **taglib** ディレクティブにある **URI** 属性の値には、スラッシュ (/) を使用します。**URI="/"** は、JRun が Web アプリケーションのルートディレクトリにあるタグを検索することを示しています。
- JST をテストするには、Web ブラウザで JSP ページをリクエストします。

簡単な例

このセクションでは、JST の簡単な例を示します。

JST ページ

次のコードをアプリケーションのルートディレクトリに simple.jst として保存します。

```
<%-- simple.jst --%>
<p>JRun サーバーのタグです。</p>
```

JSP

次のコードを拡張子 .jsp を持つページとしてアプリケーションのルートディレクトリに保存します。

```
<%@ taglib prefix="t" uri="/" %>
<t:simple/>
```

属性との対話

このセクションでは、属性と対話する JST の呼び出し方法について説明します。

JST ページ

サンプルの JST にエラーメッセージが表示されます。前のページに戻るボタンと、2 つの属性が表示されます。

次のコードを Web アプリケーションのルートディレクトリに message.jst として保存します。

```
<%-- message.jst --%>
<%@ tagAttribute name="messagetext" type="String" required="true"
      rtexpr="true"%>
<%@ tagAttribute name="messagetype" type="String" required="true"
      rtexpr="true"%>
<html>
<body>
<form>
<h1><%= messagetype %></h1>
```

```
<p><%= messagetext %></p>
<p>  <INPUT TYPE="button" VALUE="Back"
        onClick="history.back()">
</form>
</body>
</html>
<%
out.flush();
out.close();
%>
```

JSP

次のコードを拡張子 .jsp を持つページとしてアプリケーションのルートディレクトリに保存します。

```
<%@ taglib prefix="t" uri="/" %>
<%
  // userPassword 変数を設定していると想定します。
  if(userPassword.length() == 0) {
  // ページをテストするには、前の行をコメントに変え、
  // 次の行をコメントからコードに戻します。
  // if(0 == 0) {
%>
<t:message messagetype="Validation Error" messagetext="Enter password">
</t:message>
<% } %>
...
```

本文コンテンツとの対話

このセクションでは、本文コンテンツと対話する JST の例を示します。

JST ページ

次のコードを Web アプリケーションのルートディレクトリに bodyinteract.jst として保存します。

```
<%-- bodyinteract.jst --%>
<%-- 列挙のインポート --%>
<%@ page import="java.util.Enumeration" %>
<%-- 本文コンテンツを調べるには method="AFTER_BODY" を使用する必要があります。--%>
<%@ tag method="AFTER_BODY" %>
<%-- また、本文コンテンツを調べるには、type="BUFFERED" または TYPE="LOOPING"
     を使用する必要があります。--%>
<%@ tag type="BUFFERED" %>
<% // 本文コンテンツを取得します。
  String body = bodyContent.getString();
  // 本文の長さを取得します。本文には、HTML 文字も含まれているので注意してください。
  int bclength = body.length();
```

```
  callerPageContext.getOut().print("<hr>HTML for body contains " +
        bclength + "characters.");
  // ループしないで、本文コンテンツを 1 回だけ調べてください。
  returnValue=SKIP_BODY; %>
```

JSP

次のコードを拡張子 .jsp を持つページとしてアプリケーションのルートディレクトリに保存します。

```
<html>
<body>
<%@ taglib prefix="t" uri="/" %>
<%@ page import="java.util.*" %>
<h1> 本文コンテンツとの対話 </h1>
<t:bodyinteract>
 <hr>
 <p>First line
 <br>Second line
 <br>Third line
</t:bodyinteract>

</body>
</html>
```

ループ

このセクションでは、ループを使用する JST ページの例を示します。

JST ページ

次のコードを Web アプリケーションのルートディレクトリに loop.jst として保存します。

```
<%-- loop.jst --%>
<%-- 列挙のインポート --%>
<%@ page import="java.util.Enumeration" %>
<%-- ループさせるには method="AFTER_BODY" を使用する必要があります。--%>
<%@ tag method="AFTER_BODY" %>
<%-- ループさせるには type="BUFFERED" または TYPE="LOOPING" を使用する必要があり
        ます。--%>
<%@ tag type="LOOPING" %>
<%-- 渡されたオブジェクトの属性を宣言します。--%>
<%@ tagAttribute name="thisEnum" type="Enumeration" required="true"
        rtexpr="true"%>
<%@ tagAttribute name="var" type="String" required="true"
        rtexpr="true"%>
<%-- スクリプト変数を定義します。id 属性は var 属性の値から
     スクリプト変数の名前を受け取ります。--%>
<%@ tagVariable id="var" scope="NESTED" %>
```

```
<%
  if(thisEnum.hasMoreElements()) {
    // jst には独自の pageContext があるので、
    // callerPageContext を使用して呼び出し側 JSP 内で設定します。
    callerPageContext.setAttribute("header", thisEnum.nextElement());
    returnValue=EVAL_BODY_AGAIN; // ループ
  } else {
    returnValue=SKIP_BODY;
  }
%>
<%-- この例では、複数のメソッドの実装方法も示します。
     JRun では以前のコードは doAfterBody メソッドに変換されます。
   この例では、doAfterBody に加えて実行する
   doStartTag メソッドのコーディング方法を示します。--%>
<%!
public int doStartTag() throws JspException {
  if(thisEnum.hasMoreElements()) {
    // jst には独自の pageContext があるので、
  // callerPageContext.setAttribute を使用して呼び出し側 JSP 内で設定します。
    callerPageContext.setAttribute("header", thisEnum.nextElement());
    // Java ベースのメソッドでは、returnValue を設定する代わりに return を使用します。
    return EVAL_BODY_TAG; // 本文テキストの使用を可能にします。
  } else {
    return SKIP_BODY;
  }
}
%>
```

JSP

次のコードを拡張子 .jsp を持つページとしてアプリケーションのルートディレクトリに保存します。

```
<%-- loop.jsp --%>
<html>
<body>
<%@ taglib prefix="t" uri="/" %>
<%@ page import="java.util.*" %>
<h1> ヘッダーのループ </h1>
<table border="1">
  <tr>
    <th>Name</th>
    <th>Value</th>
  </tr>
<t:loop var="header" thisEnum="<%= request.getHeaderNames() %>">
   <tr>
     <td><%= header %></td>
     <td><%= request.getHeader(header) %></td>
   </tr>
</t:loop>
</table>
</body>
</html>
```

スクリプト変数の使用

このセクションでは、JST でのスクリプト変数の使用例を示します。

JST ページ

次のコードを Web アプリケーションのルートディレクトリに scriptingvar.jst として保存します。

```
<%-- scriptingvar.jst --%>
<%@ tagVariable name="myName" scope="AT_BEGIN" %>
<%-- 呼び出し側ページのコンテキストに値を保存します。--%>
<% callerPageContext.setAttribute("myName", "Christopher"); %>
<p><b>Inside the tag:</b> このタグは、myName スクリプト変数を設定します。
          これは、後で JSP の呼び出しに使用します。
```

JSP

次のページに拡張子 .jsp を付けてアプリケーションのルートディレクトリに保存します。

```
<html>
<body>
<%@ taglib prefix="t" uri="/" %>
<h1> スクリプト変数の使用 </h1>
<p>Before the tag.
<t:scriptingvar/>
<p>After the tag.
<p> myName 変数はカスタムタグで設定されていましたが、scope="AT_BEGIN" となっている
         ため、ページの後半で利用可能になります。
<p>myName is <%= myName %>
</body>
</html>
```

高度な使用方法

このセクションでは、JST の高度な使用方法について説明します。

複数のハンドラの実装

デフォルトでは、JST ページでは doStart、doEnd、または doAfterBody のいずれかのハンドラを実装します。ただし、Java で追加のハンドラをコーディングして宣言で定義すると、JST ページにハンドラを追加できます。例については、326 ページの「ループ」を参照してください。

ファイル拡張子 .jst の再マッピング

JSP ベースのカスタムタグには、.jst 以外のファイル拡張子を使用できます。拡張子を変更するには、JRun サーバー用の default-web.xml ファイルにある JSTServlet の servlet-mapping 要素を修正します。

次の例では、JST 接尾辞を .jtg に設定します。

```
<servlet-mapping>
  <servlet-name>JSTServlet</servlet-name>
  <url-pattern>*.jtg</url-pattern>
</servlet-mapping>
```

JST へのリクエストのマッピング

設定ファイルのリクエストをマッピングする際、サーブレットや他のファイルと同様に JST を使用できます。たとえば、exeption500.jst ファイルを エラーコード 500 (サーバーエラー) に相当するエラーにマッピングするために、Web アプリケーションの web.xml ファイルにあるマッピングの例を使用します。

```
<error-page>
  <error-code>500</error-code>
  <location>/exception500.jst</location>
</error-page>
```

次の例のように、JSP にある page ディレクティブの errorPage 属性を用いて JST をマッピングできます。

```
<%@ page errorPage="/exception500.jst" %>
...
```

再帰呼び出しの使用

JST は再帰呼び出しをサポートしています。これによって、JST 自身を呼び出しても、無限ループに入ることはありません。次のコードは、基本的なダンプ処理を行う再帰呼び出しタグの例を示したものです。

```
<%-- recursiveTag.jsp で使用 --%>
<%@ taglib uri="." prefix="jst" %>
<%@ page import="java.util.*" %>
<%@ tagAttribute name="var" required="true" type="java.lang.Object"
        rtexprvalue="true" %>
<%
 String theClass=var.getClass().getName();
 if (theClass.trim() == "java.util.Vector") {
  out.print("<table border=1><tr><td colspan=3>" + theClass +
        "</td></tr>");
  out.print("<tr><th></th><th> タイプ </th><th> 値 </th></tr>");
  Vector v=(Vector) var;
  Enumeration e=v.elements();
  while (e.hasMoreElements()) {
   out.print("<tr>"); %>
   <jst:_dump var="<%= e.nextElement() %>" />
   <%out.print("</tr>");
  }
  out.print("</table>");
 } else {
  out.print("<td></td><td><strong>" + theClass + "</strong></td><td>" +
        var.toString()  + "</td>");
 }
%>
```

パート IV
EJB プログラミング

パート IV では、JRun による Enterprise JavaBean (EJB) プログラミングについて説明します。次の章で構成されています。

第 13 章
EJB の概要

この章では、JRun 特有の Enterprise JavaBeans (EJB) 機能の他に、EJB の基本的な要素と概念を説明します。

目 次

EJB の基礎

EJB には、分散型でコンポーネントベースのアプリケーションを作成するためのコンポーネントアーキテクチャがあります。これは、J2EE (Java 2 Enterprise Edition) アーキテクチャの重要な部分であり、EJB 仕様の機能だけを使用して書かれた EJB は、いずれの J2EE アプリケーションサーバーでも使用できます。

「エンタープライズ」は、この略語の中で重要な言葉です。EJB はエンタープライズレベルのアプリケーション用アーキテクチャであり、EJB プログラミングモデルは、EJB API (Application Programming Interface : アプリケーションプログラミングインターフェイス) を理解している経験豊富なサーバーサイド Java 開発者に最も適しています。ただし、EJB 仕様に対応するには、アプリケーションサーバーがさまざまな EJB コンテナサービスを提供する必要があります。これらのサービスを使用することにより、ビジネスロジックの開発に専念できます。さらに JRun では、スタブレスオープンディレクトリデプロイ、エンタープライズデプロイウィザード、XDoclet サポートなどの機能を提供して、EJB 開発処理が簡素化されています。

EJB のパーツ

次の表で、EJB の構成を説明します。

パーツ	説明
ホームインターフェイス	EJB のインスタンスの作成、削除、配置などの EJB のライフサイクルオペレーションを管理するメソッドを提供します。ホームインターフェイスには 2 つのタイプがあります。 • リモートホーム　リモートクライアントによって使用されます。 • ローカルホーム　同じ JRun サーバーで実行中のクライアントによって使用されます。
コンポーネントインターフェイス	EJB クライアントに適用されるビジネスメソッドを定義します。コンポーネントインターフェイスには 2 つのタイプがあります。 • リモート　リモートクライアントによって使用されます。 • ローカル　同じ JRun サーバーで実行中のクライアントによって使用されます。
bean 実装	ビジネスロジックを実行するメソッドが含まれます。また、必要に応じて EJB 開発者が実装するコールバックメソッドも含まれます。
デプロイメントディスクリプタ	EJB とそれに必要なサービスを説明する宣言セマンティクスを指定します。

EJB の使用については、345 ページの第 14 章「EJB プログラミングテクニック」を参照してください。

EJB クライアント

EJB クライアントは次のようにリモートアクセスとローカルアクセスを使用できます。

- **リモート**　クライアントは、別の Java 仮想マシン (JVM) 内のアプリケーションサーバーの外で実行されます。
- **ローカル**　クライアントは、EJB コンテナと同じアプリケーションサーバーのインスタンス (JVM) で実行されます。

アプリケーションデプロイ担当者は、クライアントがアクセスするすべての EJB のためのコンパイル済みインターフェイスが、クライアントのクラスパスに含まれていることを確認する必要があります。

メモ：元の EJB 仕様では、クライアントが bean とそのメソッドにアクセスするのに、リモートメソッド呼び出し (RMI) を使用する必要がありました。EJB の優れた使用方法が整備されるにつれて、EJB クライアントと EJB コンテナを同じアプリケーションサーバーのインスタンスに配置する方法でパフォーマンスを最適化するデザインパターンが現れました。しかし、このようなクライアントはやはり RMI のオーバーヘッドに制約されたため、多くのアプリケーションサーバーが、同じ場所に配置されたクライアントを最適化するオプションを提供しました。最新の EJB 仕様では、アプリケーションサーバーは、ローカルとリモートの両方のインターフェイスをサポートする必要があります。

リモート bean

リモート EJB はクライアントのさまざまな JVM で実行されます。EJB がリモートで実行されることによって、ネットワークの任意の場所に配置されたクライアントからアクセスできます。

次のように bean のリモートインターフェイスを定義することによって、リモート能力を有効化できます。

- **リモートホーム**　create、remove、および find (エンティティ beans のみ) などの EJB のライフサイクルメソッドを定義します。次の例に示すように、リモートホームインターフェイスは、`javax.ejb.EJBLocalHome` を拡張します。

```
...
public interface SimpleHome extends EJBHome {
  // リモートインターフェイスが返されます。
  public Simple create() throws RemoteException, CreateException;
}
...
```

- **リモート**　EJB のビジネスメソッドを定義します。次の例に示すように、リモートインターフェイスは、`javax.ejb.EJBObject` を拡張します。

```
...
public interface Simple extends EJBObject {
   public String getMessage() throws RemoteException;
}
...
```

- **例外**　リモートインターフェイスで定義されたメソッドは、`java.rmi.RemoteException` を返す必要があります。

次の例に示すように、ejb-jar.xml ファイルで EJB を定義する場合は、**remote** 要素と **remote-home** 要素も入れる必要があります。

```
...
<session>
  <display-name>Simple</display-name>
  <ejb-name>Simple</ejb-name>
  <home>SimpleHome</home>
  <remote>Simple</remote>
  <ejb-class>SimpleBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
</session>
   ...
```

リモートとローカルの EJB クライアントのコーディングの詳細については、345 ページの第 14 章「EJB プログラミングテクニック」を参照してください。

ローカル bean

ローカル EJB はクライアントと同じサーバーで実行されます。ローカルで実行されることにより、EJB が RMI 呼び出しのオーバーヘッドを回避します。

典型的なシナリオは、1 つ以上のローカルエンティティ bean に対してクライアントとして動作するステートレスセッション bean に関係があります。

次のように、bean のローカルインターフェイスを定義することによって、ローカル能力を有効化できます。

- **ローカルホーム**　create、remove、および find (エンティティ beans のみ) などの EJB のライフサイクルメソッドを定義します。次の例に示すように、ローカルホームインターフェイスは **javax.ejb.EJBLocalHome** を拡張します。

  ```
  ...
  public interface SimpleLocalHome extends EJBLocalHome {
    public SimpleLocal create() throws CreateException;
  }
  ...
  ```

- **ローカル**　EJB のビジネスメソッドを定義します。次の例に示すように、ローカルインターフェイスは **javax.ejb.EJBLocalObject** を拡張します。

  ```
  ...
  public interface SimpleLocal extends EJBLocalObject {
     public String getMessage();
  }
  ...
  ```

- **例外**　ローカルインターフェイスで定義されたメソッドは、**java.rmi.RemoteException** を返しません。

EJB のためにリモートインターフェイスとローカルインターフェイスを実装できますが、これは一般的ではありません。

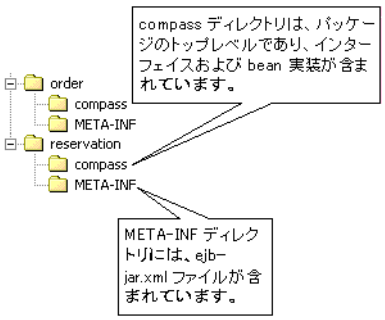
次の例に示すように、ejb-jar.xml ファイルで EJB を定義する場合は、**local** 要素と **local-home** 要素も入れる必要があります。

```
...
<session>
  <display-name>SimpleLocal</display-name>
  <ejb-name>SimpleLocal</ejb-name>
  <local-home>SimpleLocalHome</local-home>
  <local>SimpleLocal</local>
  <ejb-class>SimpleLocalBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
</session>
   ...
```

リモートとローカルの EJB クライアントのコーディングの詳細については、345 ページの第 14 章「EJB プログラミングテクニック」を参照してください。

デプロイメントディスクリプタ

EJB デプロイメントディスクリプタは XML 形式のテキストファイルで、1 つ以上の EJB に必要なコンポーネント、機能、サービスを定義する要素が含まれます。多くの開発者は、テキストエディタを使用して手作業でコーディングすることによって、デプロイメントディスクリプタを管理しています。さらに、JRun では、エンタープライズデプロイウィザードと XDoclet サポートが提供され、デプロイメントディスクリプタ要素を自動的に生成できます。次の例に示すように、EJB デプロイメントディスクリプタには ejb-jar.xml という名前を付け、インターフェイスと bean の実装に関連付けられている META-INF ディレクトリに保管する必要があります。



詳細については、345 ページの第 14 章「EJB プログラミングテクニック」を参照してください。JRun 特有のデプロイメントディスクリプタを使用して、JRun 特有の機能を実装することもできます。詳細については、第 14 章および『JRun アセンブルとデプロイガイド』を参照してください。

コンテナサービス

EJB 仕様では、少ないコーディングで EJB が使用できる高度な機能を提供するさまざまなサービスを EJB コンテナがサポートできなければなりません。

次の表で、EJB コンテナサービスを説明します。

サービス	説明	メカニズム
ライフサイクル管理	JRun では、サーバーの現在の負荷に比例して、利用可能な bean インスタンス数が自動的に増減されます。	自動
セッション管理	JRun では、パッシベートおよびアクティブ化コールバックメソッドによってステートフルセッション bean のステートを維持し、エンティティ bean コールバックメソッドによってエンティティ bean のステートを維持します。	開発者がコーディングするコールバックメソッド
トランザクション	トランザクションとは、1 つのまとまりとして実行する必要がある作業単位です。トランザクションが成功するためには、その作業単位のすべての要素が成功する必要があります。EJB には 2 つのタイプのトランザクション管理があります。 • **CMT (Container-managed transactions**：コンテナ管理トランザクション） CMT を使用すると、bean に代わってコンテナがトランザクションを開始してコミットまたはロールバックします。 • **BMT (Bean-managed transactions**：bean 管理トランザクション） BMT を使用すると、トランザクションを完全に制御できます。トランザクションはクライアントから管理できます。つまり、セッション bean の場合は bean 自体から管理できます。	デプロイメントディスクリプタ
セキュリティ	EJB では認証およびロール方式のアクセス制御をサポートしています。JRun 4 の特徴は、JAAS (Java Authentication and Authorization Service) をベースにしたセキュリティシステムです。	デプロイメントディスクリプタ

メモ：JRun EJB コンテナは EJB サーバーで実行されます。ただし、このマニュアルでは、コンテナとサーバーの両方に対して、**EJB コンテナ**という用語を使用しています。

EJB タイプ

EJB には、次の 3 つのタイプがあります。

- セッション bean
- エンティティ bean
- メッセージ駆動型 bean

これらの機能については次のページで説明します。

セッション bean

セッション bean を使用して、ビジネスロジックを管理します。セッション bean には、次の 2 つのタイプがあります。

- ステートレスセッション bean
- ステートフルセッション bean

ステートレスセッション bean

ステートレスセッション bean では、会話ステートが維持されません。ただし、ステートレスセッション bean は、セッション / エンティティのファサードパターンで利用される場合は特に有効です。この場合、セッション bean のメソッドによって、1 つ以上のエンティティ bean にある 1 つ以上のメソッドを呼び出して、1 つの単位の作業が実行されます。

詳細については、345 ページの第 14 章「EJB プログラミングテクニック」を参照してください。

ステートフルセッション bean

ステートフルセッション bean では、会話ステートが維持されます。クラスタ環境で JRun を使用すると、JRun は、クラスタ全体でステートフルセッション bean ステートを維持することによってフェイルオーバーをサポートします。

詳細については、345 ページの第 14 章「EJB プログラミングテクニック」を参照してください。

エンティティ bean

エンティティ bean は、サーバーがシャットダウンするまで存続するオブジェクトを表します。エンティティ bean のインスタンスを表現するデータは通常、関係データベースのテーブルの行に保管されています。このデータベースには、JDBC データソースからアクセスします。テーブルは、複数のデータベースに及ぶ場合もあります。

パーシスタンスには、次の 2 つのタイプがあります。

- **BMP (Bean-managed persistence：bean 管理パーシスタンス)**　データベースアクセスおよびコールバックメソッドのステートメントの更新をコーディングすることによって、エンティティ bean の実装がパーシスタンスを管理します。
- **CMP (Container-managed persistence：コンテナ管理パーシスタンス)**　デプロイメントディスクリプタに作成された仕様をコンテナが使用して、データベースアクセスを実行し、自動的にステートメントを更新します。

BMP

BMP では、コールバックメソッドの適切なデータベースの更新をコーディングすることによって、開発者がパーシスタンスを管理します。たとえば、`ejbUpdate` メソッドにはデータベースを更新するコードがあり、`ejbFindByPrimaryKey` メソッドにはプライマリキーを使用してデータベース行を見つけるコードがあります。

詳細については、samples JRun サーバーの compass-ear ディレクトリにある `Order` EJB を参照してください。

CMP

CMP では、ライフサイクルのある時点でデータベースと bean を自動的に同期することによって、コンテナがパーシスタンスを管理します。CMP を使用すると、bean 実装コーディングがより簡単になるので、ビジネスロジックに焦点を合わせることができます。

EJB 2.0 仕様では、アプリケーションサーバーで EJB 1.1 CMP および EJB 2.0 CMP がサポートされている必要があります。

EJB 2.0 CMP のサポート

EJB 2.0 仕様は CMP に対する主要な変更を特徴としており、次の機能が含まれています。

- EJB 実装クラスはアブストラクトクラスとして定義されています。
- パーシスタンスは、EJB クエリ言語 (EJBQL) を使用して管理されます。
- リレーションシップではエンティティ bean 間の関係を維持できます。
- CMP bean では、`finder` メソッドだけではなく、`select` メソッドも使用できます。

CMP の拡張が EJB 2.0 の主な特徴になっています。このマニュアルのリソースおよび例を使用するだけでなく、EJB 2.0 に関する業界誌も参照してください。このマニュアルの序章にはこれらの本のリストが記載されています。

EJB 2.0 サポートの詳細については、第 14 章を参照してください。

EJB 1.1 CMP のサポート

JRun 4 の EJB 1.1 CMP のサポートは、JRun の以前のバージョンの CMP とは異なります。以前は、ejb-jar.xml ファイルの環境エントリでパーシスタンス情報を指定していました。JRun 4 では、jrun-ejb-jar.xml ファイルの要素でパーシスタンス情報を指定します。これらの要素で、store、load、findByPrimaryKey などの適切なパーシスタンスアクションのパーシスタンス管理方法を EJB コンテナに通知します。

EJB 1.1 サポートの詳細については、第 14 章を参照してください。

メモ: JRun では、JRun 3.1 スタイルの CMP bean のオートデプロイもサポートしています。

メッセージ駆動型 bean

MDB (message-driven bean：メッセージ駆動型 bean) は JMS メッセージリスナとしての役割を果たします。MDB にはリモート、リモートホーム、ローカル、またはローカルホームインターフェイスがない点で、セッション bean およびエンティティ bean とは異なります。MDB は、bean 実装があるという点で他の bean タイプと同じであり、ejb-jar.xml ファイルで定義されます。トランザクション、セキュリティ、ライフサイクル管理などの EJB 機能を利用できます。単純な JMS メッセージコンシューマに対する MDB の主な利点は、コンテナで複数の MDB インスタンスをインスタンス化して複数のメッセージを同時に処理できることです。

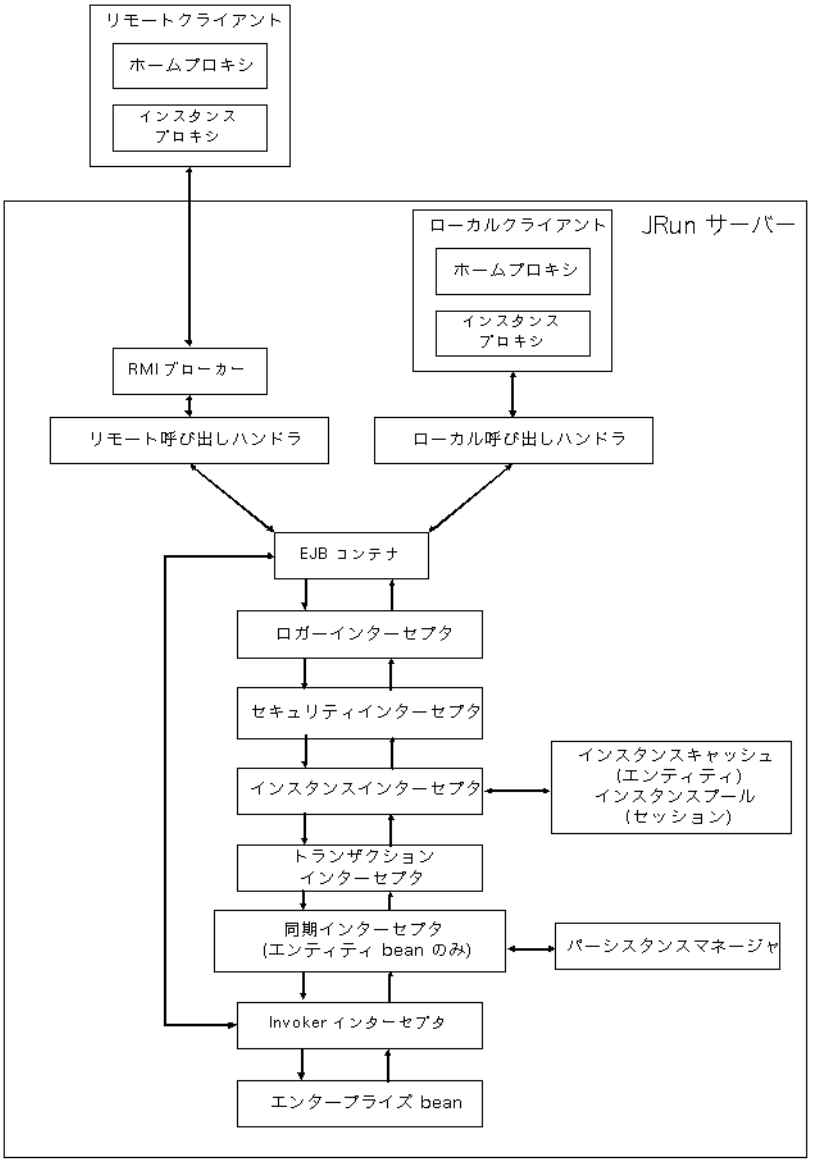
MDB の詳細については、第 14 章を参照してください。

JRun での EJB の使用

このセクションでは、JRun の EJB アーキテクチャに固有の機能について説明します。

JRun EJB アーキテクチャ

次の図に示すように、JRun EJB アーキテクチャの特徴は、それぞれが特定の領域を管理する一連のインターセプタです。



スタブレスデプロイ

JRun EJB の主な機能の 1 つは、スタブレスデプロイです。JRun の EJB のデプロイツールはありません。EJB をコンパイルして JAR ファイルにパッケージし (オプション)、コンパイルされた EJB インターフェイスをクライアントにコピーし、その場所をクライアントのクラスパスに定義します。JRun インスタンスプロキシでは、以前はスタブで管理されていた機能を処理します。

デプロイオプション

JRun には、bean の開発やデプロイを容易にする柔軟な EJB デプロイモデルが用意されています。次の表で、このデプロイモデルの機能を説明します。

デプロイ機能	説明
ホットデプロイ	ホットデプロイを有効にすると、JAR ファイルまたはディレクトリ構造をデプロイディレクトリにコピーすることによって EJB をデプロイできます。デフォルトでは、JRun サーバーのルートディレクトリがホットデプロイ用に設定されます。この機能をテストするには、samples JRun サーバーを起動し、EJB を <JRun のルートディレクトリ >/servers/samples ディレクトリにコピーします。 ホットデプロイを有効にすると、JRun デプロイヤは EJB デプロイメントディスクリプタへの変更を検出し、bean を自動的にリデプロイします。 また、Sun リファレンス実装および JRun 3.1 を含む他のアプリケーションサーバー用に作成された EJB もデプロイされます。
JAR やオープンディレクトリからの実行	JRun では、JAR ファイルまたはオープンディレクトリ構造から EJB を実行できます。JRun デプロイヤは、META-INF/ejb-jar.xml ファイルを検出すると、ファイルに指定されている EJB をデプロイします。オープンディレクトリ構造から実行できると、従来の EJB デプロイ処理とリデプロイ処理で必要だった多くの手順を省略できるので、EJB 開発サイクルでは特に有効です。

JRun EJB デプロイメントディスクリプタ

ejb-jar.xml ファイルでの標準設定に加えて、JRun では、JRun EJB デプロイメントディスクリプタで JRun 特有の設定を行うことができます。ほとんどの J2EE アプリケーションサーバーには、サーバー特有のディスクリプタが含まれます。jrun-ejb-jar.xml が JRun EJB デプロイメントディスクリプタです。jrun-ejb-jar.xml ファイルには、次のタイプの情報が含まれます。

- JNDI ロケーション
- CMP 1.1 仕様
- クラスタリング設定
- タイムアウト指定
- コミットオプション
- ステートレスセッション bean のインスタンスプールサイズ

jrun-ejb-jar.xml ファイルの詳細については、JRun ドキュメントのホームページで利用できるオンラインディスクリプタドキュメントを参照してください。

EJB クラスタリング

クラスタの一部である JRun サーバーで EJB を実行すると、ロードバランスおよびフェイルオーバーが有効になり、性能および信頼性が高まります。

JRun は、デフォルトで EJB クラスタリングを有効にします。EJB クラスタリングを無効にするには、jrun-ejb-jar.xml ファイルの **`cluster-home`** 要素と **`cluster-object`** 要素を false に設定します。

メモ：EJB クラスタリングは、ローカル bean では有効化されません。

詳細については、『JRun 管理者ガイド』を参照してください。

XDoclet

JRun は XDoclet と統合できます。XDoclet は、EJB、Web アプリケーション、および JSP タグライブラリのコードおよびデプロイメントディスクリプタを生成するオープンソースツールです。多くのアプリケーションサーバーでは、基本的な XDoclet タグで記述されている XDoclet および EJB のサポートを追加しています。XDoclet タグはサーバー間で移植可能です。

特に EJB では、XDoclet を使用して次のタスクを実行できます。

- 入力としてビジネスメソッドを含んでいるアブストラクトクラスを使用して、bean 実装に必要な EJB コールバックメソッドを生成します。
- bean　実装クラスからリモート、ローカル、ホーム、およびローカルホームインターフェイスを生成します。
- デプロイメントディスクリプタを生成します。

XDoclet が実行する処理を制御するには、bean 実装で jrun.xml 属性と JavaDoc スタイルのコメントを組み合わせて使用します。

XDoclet を使用した EJB の開発の詳細については、345 ページの第 14 章「EJB プログラミングテクニック」を参照してください。

エンタープライズデプロイウィザード

JRun エンタープライズデプロイウィザードによって、EJB の開発およびデプロイ処理が簡素化されています。Swing ベースのグラフィカルユーザーインタフェースを使用すると、あらゆるタイプの EJB を作成したり、既存の EJB のデプロイメントディスクリプタを編集して JAR ファイルにパッケージし、JRun にデプロイしたりすることができます。特に、エンタープライズデプロイウィザードのオブジェクト / 関連マッピング機能では、エンティティ bean の開発処理を簡素化できます。

エンタープライズデプロイウィザードは、スタンドアローンのツールとして実行されるか、Borland JBuilder、Sun Forte、または IntelliJ IDEA のプラグインとして実行されます。

エンタープライズデプロイウィザードを起動するには、<JRun のルートディレクトリ >/bin ディレクトリの **`jrunwizard`** を実行します。

詳細については、エンタープライズデプロイウィザードのオンラインヘルプを参照してください。

第 14 章
EJB プログラミングテクニック

この章では、一般的な EJB プログラミングテクニックのサンプルコードを紹介します。

目 次

JRun での EJB の使用

JRun で EJB をコーディングする場合、コーディングの他にデプロイやクライアントアクセスについても考慮する必要があります。

EJB のコーディング

EJB は次のコンポーネントで構成されています。

- **ホームまたはローカルホームインターフェイス**　ホームインターフェイスは `javax.ejb.EJBHome` を拡張し、ローカルホームインターフェイスは `javax.ejb.EJBLocalHome` を拡張します。
- **コンポーネントインターフェイス**　リモートコンポーネントインターフェイスは `javax.ejb.EJBObject` を拡張し、ローカルコンポーネントインターフェイスは `javax.ejb.EJBLocalObject` を拡張します。
- **bean 実装**　セッション bean は `javax.ejb.SessionBean` を実装し、エンティティ bean は `javax.ejb.EntityBean` を実装します。
- **デプロイメントディスクリプタ**　ejb-jar.xml ファイルは、基本的な bean、アセンブリ、リファレンスの他に、EJB 2.0 CMP エンティティ bean のパーシスタンス情報を定義します。また、このファイルはデプロイメントディスクリプタとも呼ばれます。
- **JRun デプロイメントディスクリプタ**　`jrun`-ejb-jar.xml ファイルは、JRun 特有の EJB コンテナ情報の他に、EJB 2.0 CMP エンティティ bean のパーシスタンス情報を定義します

メモ： 1 つの EJB に対して、リモートとローカルの両方のインターフェイスを有効にできます。ただし、この用法は一般的ではないのでお勧めしません。

J2EE 互換のアプリケーションサーバーとして、EJB 仕様に記述されている既存の bean をデプロイすることができます。また、JRun は EJB を開始するにあたって次の方法をサポートします。

- **XDoclet**　`XDoclet` サービスが EJB ディレクトリに対して有効で、XDoclet スタイルのコメントを使用して bean 実装クラスをコーディングした場合、JRun はインターフェイスディスクリプタおよびデプロイメントディスクリプタを生成します。詳細については、391 ページの「EJB と XDoclet」を参照してください。
- **JRun エンタープライズデプロイウィザード**　EJB を開始するときに使用します。オプションで、必要なファイルのテンプレートを生成します。このツールは、特に EJB 1.1 スタイル と EJB 2.0 スタイルの両方のエンティティ bean の CMP 設定を生成する際に役立ちます。詳細については、390 ページの「エンタープライズデプロイウィザード」を参照してください。

この章では、EJB の一般的なコーディング例を多数く取り上げていきます。XDoclet など JRun 特有の機能についても説明します。EJB プログラミングの詳細については、EJB 2.0 を解説している業界紙を参照してください。このマニュアルの序章にはこれらの本のリストが記載されています。

EJB のデプロイ

JRun の自動的なデプロイは、開発サイクル時における開発処理を簡素化します。JRun サーバーの jrun.xml ファイルで DeoplyService の `deployDirectory` 属性を 1 つ以上指定している場合は、自動デプロイが有効になります

JRun サーバーには、オートデプロイディレクトリが 1 つ以上存在する場合としない場合があります。デフォルトでは、各 JRun サーバーのルートディレクトリに含まれるすべてのディレクトリでオートデプロイが有効です。

サーバーの起動時、EJB ディレクトリとオートデプロイディレクトリに含まれる EJB JAR ファイルが、JRun によって自動的にデプロイされます。ディレクトリまたは JAR ファイルには、有効な META-INF/ejb-jar.xml ファイルの他に、コンパイル済みのインターフェイスや実装が含まれている必要があります。ejb-jar.xml ファイルや jrun-ejb-jar.xml ファイルを変更した場合は、JRun はホットデプロイを使用して、EJB コンテナに再デプロイします。

JRun では、自動デプロイされた EAR ファイルまたはエンタープライズアプリケーションのディレクトリで見つかった EJB も自動的にデプロイされます。EAR ファイルで使用する場合は、web.xml ファイルの **`ejb-ref`** 要素を使用してサーブレットから EJB へのアクセスを定義します。

EJB へのアクセス

EJB クライアントでは、次のようなさまざまな形式が使用されます。

- JSP および Servlet クライアント
- リモート Java クライアント (アプリケーション、アプレット、またはスタンドアローンオブジェクト)
- EJB クライアント (すなわち、ある EJB が別の EJB のメソッドを呼び出します。)

コンパイル済みの EJB インターフェイスがクライアントのクラスパスに存在する必要があります。サーブレットの場合は、このインターフェイスが、WEB-INF/classes (JAR ファイル内ではない)、WEB-INF/lib (JAR ファイルの一部として)、または *jrun_server*/SERVER-INF/lib (JAR ファイルの一部として) に存在することがあります。)

メモ：リモート Java クライアントは、EJB にアクセスするときに、コマンドラインでセキュリティポリシーファイルを指定する必要があります。たとえば、次のコマンドラインを使用します。
java -Djava.security.policy=<JRun のルートディレクトリ >/lib/jrun.policy MyEJBClient

EJB クライアントでは、EJB のホームオブジェクトとリモートオブジェクトへのアクセスに JNDI (Java Naming and Directory Interface：Java ネーミングディレクトリインターフェイス) が使用されます。EJB にアクセスするには、次のように **`InitialContext.lookup`** メソッドを実行すると、クライアントはホームオブジェクトを探します。

- リモート Java クライアントは、jrun-ejb-jar.xml ファイルの **`jndi-name`** 要素 (これはデフォルトで ejb-jar.xml ファイルの **`ejb-name`** 要素になります) を使用して検索を実行します。
- サーブレットと EJB クライアントは **`jndi-name`** の値を使用できます。しかしデプロイメントディスクリプタで **`ejb-ref`** 要素や **`ejb-local-ref`** 要素を定義し、J2EE ENC (Environment Naming Context) を使用して EJB のホームインターフェイスを検索する方法もあります。J2EE ENC は、java:comp/env のルート下にすべてのリファレンス名をバインドします。

  たとえば、web.xml で、**`ejb-ref`** を ejb/Orders のために定義した場合、サーブレットクライアントは java:comp/env/ejb/Orders で **`InitialContext`** の検索を実行できます。このテクニックを使用すると、最も移植しやすいクライアントコードが作成できます。詳細については、357 ページの「ローカル EJB」を参照してください。

クライアントコードは、リモートオブジェクトやローカルオブジェクトを通じて EJB メソッドを呼び出します。

リモートクライアント

Swing アプリケーションのようなリモートクライアント、他の JRun サーバー上のサーブレット、他の JRun サーバー上の EJB では、次の例に示すように、InitialContext コンストラクタにプロパティを渡して bean を検索します。

```
...
try {
  // 1:  // コンテキストを取得します。
  Properties props = new Properties();
  // サーバーの SERVER-INF/jndi.properties ファイルにある
  // java.naming.factory.initial プロパティと比較します。
  props.put(Context.INITIAL_CONTEXT_FACTORY,
    "jrun.naming.JRunContextFactory");
  // サーバーの SERVER-INF/jndi.properties ファイルにある
  // java.naming.provider.url のプロパティと比較します。
  //  コンマ区切りのリストも使用できます。
  props.put(Context.PROVIDER_URL,
    "localhost:2918");
  Context ctx = new InitialContext(props);

  // 2: 特定の EJBHome を検索して絞り込みます。
  // java:Simple の場合もあります。
  Object o = ctx.lookup("Simple");

  SimpleHome home = (SimpleHome)
    PortableRemoteObject.narrow(o, SimpleHome.class);

  // 3: 特定の EJBObject を作成します。
  Simple test = home.create();

  // 4: EJB でメソッドを実行します。
  String s = test.getMessage();
  // s を使用してテストします。この例ではサーブレットを想定しています。
  out.println("<br>EJB Message:" + s);
  } // try を終了
catch (Exception e) {
  e.printStackTrace();
  } // catch を終了
...
```

クラスタで実行するときは、オプションで、PROVIDER_URL でコンマ区切りのサーバーとポートのリストを指定できます。これは、InitialContext の作成が最初のサーバーで失敗した場合に、次のサーバーにフェイルオーバーすることで、接続成功に役立ちます。EJB クラスタリングの詳細については、『JRun 管理者ガイド』を参照してください。

ローカルクライアント

サーブレットや他の EJB などのローカルクライアントでは、次の例に示すように、空の InitialContext オブジェクトを作成して bean を検索します。

```
...
try {
  // 1:  // コンテキストを取得します。
  Context ctx = new InitialContext();

  // 2: 特定の EJBHome を検索して絞り込みます。
  // java:Simple の場合もあります。
  Object o = ctx.lookup("Simple");
  // この bean にローカルホームインターフェイスが含まれている場合は、
  // PortableRemoteObject.narrow の呼び出しの代わりにキャストを使用します。
  SimpleHome home = (SimpleHome)
    PortableRemoteObject.narrow(o, SimpleHome.class);

  // 3: 特定の EJBObject を作成します。
  Simple test = home.create();

  // 4: EJB でメソッドを実行します。
  String s = test.getMessage();
...
```

ステートレスセッション bean

ステートレスセッション bean は、単一メソッド呼び出しの範囲内でビジネスロジックを管理します。各呼び出し間のステートは維持しません。ステートレスセッション bean の連続的な呼び出しは、bean のインスタンスプールからの個別のインスタンスによって処理できます。

ホームインターフェイス

ステートレスセッション bean のホームインターフェイスは、リモートまたはローカルのホームインターフェイスを返す create メソッドを定義します。create メソッドにはパラメータがなく、RemoteException（リモートホームインターフェイスのみ）および CreateException を返します。

次のコードは、ステートレスセッション bean のホームインターフェイスを示しています。

```
import java.rmi.*;
import javax.ejb.*;

public interface SimpleHome extends EJBHome {
  //  create メソッドは、Simple を返します。
  public Simple create() throws RemoteException, CreateException;
}
```

コンポーネントインターフェイス

ステートレスセッション bean のコンポーネントインターフェイスは、クライアントから呼び出されるビジネスメソッドを定義します。リモートコンポーネントインターフェイスで定義されるメソッドの場合、常に RemoteException を返す必要があり、また、すべてのコンポーネントインターフェイスのメソッドが、bean 実装内の対応するビジネスメソッドから返された例外を返す必要があります。

次のコードは、ステートレスセッション bean のリモートコンポーネントインターフェイスを示しています。

```
import java.rmi.*;
import javax.ejb.*;

public interface Simple extends EJBObject {
   public String getMessage() throws RemoteException;
}
```

bean 実装

ステートレスセッション bean の bean 実装には、setSessionContext、必要なコールバックメソッド、およびビジネスメソッドが含まれます。リモート bean のメソッドはすべて RemoteException を投げる必要があり、ejbCreate メソッドは RemoteException および CreateException を投げる必要があります。

次のコードは、ステートレスセッション bean の bean 実装を示しています。

```
import java.rmi.*;
import javax.ejb.*;

public class SimpleBean implements SessionBean {
  private SessionContext context;
  // これはビジネスメソッドです。
  public String getMessage() throws RemoteException {
   String thisMessage = "Habari Yako";
   return thisMessage;
  }
  // 次はコールバックメソッドで、
  // SessionBean インターフェイスにあります。
  public void ejbCreate() throws RemoteException, CreateException { }
  public void ejbRemove() throws RemoteException { }
  public void ejbActivate() throws RemoteException { }
  public void ejbPassivate() throws RemoteException { }

  // コンテキスト変数を設定します。
  public void setSessionContext(SessionContext context) throws
        RemoteException {
    this.context = context;
  }
}
```

EJB デプロイメントディスクリプタ

ステートレスセッション bean の基本的なデプロイメントディスクリプタには、session 要素内に要素が含まれます。後で、アプリケーションアセンブラにより、assembly-descriptor 要素内に追加仕様が作成されます。

次のサンプルは、ステートレスセッション bean の基本的なデプロイメントディスクリプタの要素を示しています。

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
        JavaBeans 2.0//EN" "http://java.sun.com/j2ee/dtds/
        ejb-jar_2_0.dtd">

<ejb-jar>
<description>EJB の簡単なテスト </description>
<enterprise-beans>
<session>
  <display-name>Simple</display-name>
  <ejb-name>Simple</ejb-name>
  <home>SimpleHome</home>
  <remote>Simple</remote>
  <ejb-class>SimpleBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
  </session>
</enterprise-beans>
<assembly-descriptor>
</assembly-descriptor>
</ejb-jar>
```

メモ：デプロイメントディスクリプタは、既存のものをコピーするか、直接コーディングするか、XDoclet サービスを使用するか、またはエンタープライズデプロイウィザードを使用して生成できます。

JRun EJB デプロイメントディスクリプタ

jrun-ejb-jar.xml ファイルには、JRun 特有のデプロイ設定が含まれます。詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

次のサンプルは、ステートレスセッション bean の JRun EJB デプロイメントディスクリプタを示しています。

```
<?xml version="1.0"?>
<!DOCTYPE jrun-ejb-jar PUBLIC '-//Macromedia, Inc.//DTD jrun-ejb-jar
        4.0//EN' 'http://jrun.macromedia.com/dtds/jrun-ejb-jar.dtd'>

<jrun-ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>Simple</ejb-name>
      <jndi-name>Simple</jndi-name>
    </session>
  </enterprise-beans>
</jrun-ejb-jar>
```

サンプルクライアント

次のサーブレット例は、ステートレスセッション bean へのリモートクライアントアクセスを示しています。

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.rmi.*;
import java.rmi.*;
import java.util.*;
import javax.naming.*;
import javax.ejb.*;
...
  try {
   // 1: コンテキストを取得します。
       Properties props = new Properties();
       props.put(Context.INITIAL_CONTEXT_FACTORY,
          "jrun.naming.JRunContextFactory");
       props.put(Context.PROVIDER_URL,
          "remoteservername:2918");
       Context ctx = new InitialContext(props);
       // ローカルクライアントは、空の InitialContext を使用できます。
       // Context ctx = new InitialContext();
       out.println("<br> 新規 Context の取得 ");

    // 2: 特定の EJBHome を検索して絞り込み、キャストします。
        out.println("<br>Simple の作成 ");
        Object o = ctx.lookup("Simple");
        SimpleHome home = (SimpleHome)
               PortableRemoteObject.narrow(o, SimpleHome.class);

    // 3: 特定の EJBObject を作成します。
      Simple test = home.create();
      if (test != null)
                out.println("<br> リモート EJBObject の完成 ");
       else
         out.println("<br> 返された null の作成 ");
      // 4: EJB でメソッドを実行します。
      String s = test.getMessage();
      out.println("<br>EJB メッセージ:" + s);
  } // try を終了
  catch (Exception e) { out.println("<br>" + e); e.printStackTrace();
    } // catch を終了
...
```

ステートフルセッション bean

ステートフルセッション bean は、メソッド呼び出し間のステートを維持します。したがって、クライアントは、変数やオブジェクトインスタンスがメソッド呼び出し間で確実に継続されるという前提で一連のメソッドを呼び出すことができます。

ホームインターフェイス

ステートフルセッション bean のホームインターフェイスは、次の例に示すように、1 つ以上の `create` メソッドを定義します。

```
package flashgateway.samples.ejb;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface SampleStatefulEjbHome
        extends EJBHome {
  public SampleStatefulEjb create()
           throws CreateException, RemoteException;
  public SampleStatefulEjb create(String initialMessage)
           throws CreateException, RemoteException;
}
```

コンポーネントインターフェイス

ステートフルセッション bean のコンポーネントインターフェイスは、クライアントから呼び出されるビジネスメソッドを定義します。リモートコンポーネントインターフェイスで定義されるメソッドの場合は、常に `RemoteException` を返す必要があり、また、すべてのコンポーネントインターフェイスのメソッドが、bean 実装内の対応するビジネスメソッドから返された例外を返す必要があります。

次のコードは、ステートフルセッション bean のリモートコンポーネントインターフェイスを示しています。

```
package flashgateway.samples.ejb;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

/**
 * SampleStatefulEjb は、カスタムの計算に使用できるリソースで、
 * 前に実行した計算を記憶できます。
 */
public interface SampleStatefulEjb
        extends EJBObject {
```

```
    public String getMessage()
        throws RemoteException;
    public int getBiggerInt(int input)
            throws RemoteException;
    public String getBiggerString(String input)
            throws RemoteException;
}
```

bean 実装

次のコードは、ステートフルセッション bean の bean 実装を示しています。

```
package flashgateway.samples.ejb;

import javax.ejb.EJBContext;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

/**
 * サンプルのステートフル EJB 実装です。
 */
public class SampleStatefulEjbBean implements SessionBean {

    private String message;
    private int lastCalculation;
    private int count = 0;

    transient EJBContext context;

    public void ejbCreate() {
        message = "ステートフル EJB からこんにちは";
    }

    public void ejbCreate(String initialMessage) {
        message = initialMessage;
    }

    public int getBiggerInt(int input) {
        return input + 1;
    }

    public String getBiggerString(String input) {
        count++;
        message = "ハロー " + input;
        return getMessage();
    }

    public String getMessage() {
        return message + " (count=" + count + ")";
    }
```

```
    public void ejbRemove() throws RemoteException {   }
    public void ejbActivate() throws RemoteException {  }
    public void ejbPassivate() throws RemoteException {  }

    public void setSessionContext(SessionContext context) {
        context = context;
    }
    public SessionContext getSessionContext() {
        return (SessionContext) context;
    }
}
```

EJB デプロイメントディスクリプタ

次のサンプルは、ステートフルセッション bean の基本的なデプロイメントディスクリプタの要素を示しています。

```
...
<session>
      <display-name>Flash Sample Stateful Session EJB</display-name>
      <ejb-name>FlashSampleStatefulEJB</ejb-name>
      <home>flashgateway.samples.ejb.SampleStatefulEjbHome</home>
      <remote>flashgateway.samples.ejb.SampleStatefulEjb</remote>
      <ejb-class>flashgateway.samples.ejb.SampleStatefulEjbBean
        </ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Bean</transaction-type>
</session>
...
```

JRun EJB デプロイメントディスクリプタ

次のサンプルは、ステートフルセッション bean の JRun EJB デプロイメントディスクリプタを示しています。

```
...
<session>
      <ejb-name>FlashSampleStatefulEJB</ejb-name>
      <jndi-name>FlashSampleStatefulEJB</jndi-name>
      <cluster-home>False</cluster-home>
      <cluster-object>False</cluster-object>
      <timeout>900</timeout>
</session>
...
```

サンプルクライアント

上記のステートレスセッション bean のサンプルコードは、samples JRun サーバーの Flash Gateway アプリケーションにあります。クライアントを表示するには、Flash MX を使用して <JRun のルートディレクトリ >/servers/samples/flashsamples-ear/webapp/movies/EjbExample.fla を開きます。

ローカル EJB

ローカル EJB により、クライアントは、EJB コンテナと同じアプリケーションサーバーのインスタンスで実行されます。これは、クライアントが、PortableRemoteObject.narrow メソッドの代わりに簡単なキャストを使用して、ローカルホームインターフェイスにアクセスできることを意味します。

メモ：EJB のクラスタリングは、ローカル EJB では無効です。

この機能はセッション bean の機能ですが、エンティティ bean でも有効です。すべてのエンティティ bean をローカルとして定義することを検討してください。

ヒント：一般的なデザインパターンでは、リモートクライアントが、同一サーバー上に共存するローカルエンティティ bean に対してクライアントとして動作するセッション bean を呼び出します。

ローカルホームインターフェイス

ローカルホームインターフェイスは、create メソッドを定義します。create メソッドは、RemoteException を返しません。

次のコードは、ローカルホームインターフェイスを示しています。

```
import javax.ejb.*;

// EJBLocalHome を拡張します。
// ローカルインターフェイス (SimpleLocal) を返します。
public interface SimpleLocalHome extends EJBLocalHome {
  public SimpleLocal create() throws CreateException;
}
```

コンポーネントインターフェイス

ローカルコンポーネントインターフェイスは、ローカルで呼び出すことができるビジネスメソッドを定義します。

次のコードは、ローカルコンポーネントインターフェイスを示しています。

```
import javax.ejb.*;

// EJBLocalObject を拡張します。
public interface SimpleLocal extends EJBLocalObject {
   public String getMessage();
}
```

bean 実装インターフェイス

ローカルでアクセスする bean の bean 実装は、リモートアクセスの実装と同じです。サンプルコードについては、350 ページの「ステートレスセッション bean」を参照してください。

EJB デプロイメントディスクリプタ

ローカル EJB のデプロイメントディスクリプタをコーディングする場合は、local-home 要素と local 要素ではなく、home 要素と remote 要素を使用します。

次の例は、ローカル EJB のデプロイメントディスクリプタの要素を示しています。

```
...
<session>
  <display-name>SimpleLocal</display-name>
  <ejb-name>SimpleLocal</ejb-name>
  <local-home>SimpleLocalHome</local-home>
  <local>SimpleLocal</local>
  <ejb-class>SimpleLocalBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
</session>
...
```

JRun EJB デプロイメントディスクリプタ

次の例は、ローカル bean の JRun EJB デプロイメントディスクリプタを示しています。

```
<?xml version="1.0"?>
<!DOCTYPE jrun-ejb-jar PUBLIC '-//Macromedia, Inc.//DTD jrun-ejb-jar
        4.0//EN' 'http://jrun.macromedia.com/dtds/jrun-ejb-jar.dtd'>
<jrun-ejb-jar>
  <enterprise-beans>
     <session>
      <ejb-name>SimpleLocal</ejb-name>
      <jndi-name>SimpleLocal</jndi-name>
    </session>
  </enterprise-beans>
</jrun-ejb-jar>
```

サンプルクライアント

次のサーブレットクライアントでは、JNDI を通じてローカル EJB を検索するさまざまな方法を示します。

- J2EE ベンダーフリーテクニック
- JRun 固有テクニック

J2EE ベンダーフリーテクニック

web.xml ファイルと jrun-web.xml ファイルで **ejb-local-ref** 要素を定義すると、最も移植しやすいサーブレットクライアントコードが作成できるので、複数の J2EE ベンダーのコンテナで実行するアプリケーションにお勧めします。

- **web.xml file　ejb-local-ref** 要素は、次のサンプルに示すように、EJB リファレンス名などの EJB 情報を定義します。

  ```
  ...
  <ejb-local-ref>
    <ejb-ref-name>ejb/SimpleLocal</ejb-ref-name>
  ```

```
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>SimpleLocalHome</local-home>
    <local>SimpleLocal</local>
    <ejb-link>SimpleLocal</ejb-link>
  </ejb-local-ref>
  ...
```

- **jrun-web.xml file**　ejb-local-ref 要素は、次の例に示すように、web.xml ファイルの ejb-ref-name 要素を JNDI 名にマッピングします。

```
...
<ejb-local-ref>
  <ejb-ref-name>ejb/SimpleLocal</ejb-ref-name>
  <jndi-name>SimpleLocal</jndi-name>
</ejb-local-ref>
...
```

これらの要素を定義すると、次のサーブレットコードで示されているように、サーブレットクライアントは、実際のベンダー固有 JNDI ロケーションの代わりに、J2EE ENC (つまり、java:comp/env) を使用して、InitialContext.lookup メソッドを実行します。

```
...
try {
  // 1：コンテキストを取得します。
  Context ctx = new InitialContext();

  // 2：ENC (java:comp/env) を使用して検索します。
  Object o = ctx.lookup("java:comp/env/ejb/SimpleLocal");
  // ローカル bean にキャストを使用します。
  SimpleLocalHome home = (SimpleLocalHome)o;
  // 3：特定の EJBObject を作成します。
  SimpleLocal test = home.create();
...
```

JRun 固有テクニック

ejb-local-ref 要素を使用しないクライアントは、次の例に示すように、JNDI 名を指定し、InitialContext.lookup メソッドで local/ 接頭辞を使用します。

```
...
try {
  // 1：コンテキストを取得します。
  Context ctx = new InitialContext();

  // 2：jrun-ejb-jar.xml ファイルの jndi-name 要素に SimpleLocal しかなくても、
  // local/ 接頭辞を使用して検索します。
  Object o = ctx.lookup("local/SimpleLocal");
  // ローカル bean にキャストを使用します。
  SimpleLocalHome home = (SimpleLocalHome)o;
  // 3：特定の EJBObject を作成します。
  SimpleLocal test = home.create();
...
```

メモ:クライアントコードでこのオプションを使用すると、J2EE アプリケーションサーバー間で移植できなくなります。しかし、これは JRun だけで実行予定のアプリケーションには役立ち、JRun でのアプリケーションのプロトタイプ作成がすばやく簡単になります。

セッションエンティティファサード

一般的なデザインパターンには、セッション bean によるすべてのエンティティ bean アクセスのルーティングが含まれます。EJB クライアントは、セッション bean でメソッドを呼び出し、セッション bean は、必要に応じて、メソッドを 1 つ以上のエンティティ bean メソッドに転送します。これは、セッションエンティティファサードと呼びます。

セッションエンティティファサードのデザインパターンには、セッションエンティティ bean 通信にローカルアクセスを使用することによるパフォーマンスの簡略化など、多くの利点があります。

この例では、363 ページの「BMP エンティティ bean」に示すエンティティ bean でメソッドを呼び出すセッション bean を示します。ただし、デプロイメントディスクリプタは示しません。

ホームインターフェイス

次のコードは、セッション bean ファサードで使用するステートレスセッション bean のリモートホームインターフェイスを示しています。

```
package compass;

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

コンポーネントインターフェイス

次のコードは、セッションエンティティ bean ファサードで使用するステートレスセッション bean のリモートコンポーネントインターフェイスを示しています。

```
package compass;

public interface ReservationRemote extends javax.ejb.EJBObject {
  public int reserve(String custId, int tripId, double amount,
        String ccType, String ccNumber,
    String ccExpiration) throws java.rmi.RemoteException;
}
```

bean 実装

セッションエンティティファサードのセッション bean 実装は、エンティティ bean を検索し、ビジネスロジックの必要に応じてそのメソッドを呼び出します。

次のコードは、エンティティ bean を呼び出して予約処理の一部として予約内容を作成するセッション bean の bean 実装を示しています。

```
package compass;

import javax.ejb.*;

public class ReservationBean implements SessionBean {
...
public int reserve(String custId, int tripId, double amount,
        String ccType, String ccNumber, String ccExpiration) {

    int orderId=0;

    // InitialContext をインスタンス化します
    javax.naming.InitialContext ctx=null;
    try {
      ctx = new javax.naming.InitialContext();
    } catch (Exception e) {
      log(e);
    }
...
    // エンティティ bean の create メソッドを呼び出すことによって、
        新規の予約内容を作成します。
    try {
      Object obj = ctx.lookup("Order");
      // この例で、ローカルインターフェイスを使用することもできます。
      OrderHomeRemote home =
        (OrderHomeRemote)javax.rmi.PortableRemoteObject.narrow(obj,
        OrderHomeRemote.class);
      OrderRemote order = home.create(custId, tripId, ccType, ccNumber,
        ccExpiration);
      orderId = order.getOrderId();
      log("Order created:"+orderId);
    } catch (Exception e) {
    }
...
    return orderId;
...
```

サンプルクライアント

次の JSP 例は、セッションエンティティファサードへのクライアントアクセスを示しています。

```
<%@ page import="compass.*" %>
<jsp:useBean id="trip" scope="session" class="compass.TripBean"/>
<html>
<head>
<title>確認</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<link rel="stylesheet" href="master.css" type="text/css">
</head>
<body>
<%@ include file="header.jsp"%>
<br>
<img src="images/tripConfirmation.gif">
<br><br>
<jsp:include page="tripsummary.jsp" flush="true"/>
<br>
<%
  String type = request.getParameter("type");
  String number = request.getParameter("number");
  String expiration = request.getParameter("expiration");
  String userId = (String) session.getAttribute("userId");
  int tripId = trip.getTripId();
  double price = trip.getPrice();

  try {
    javax.naming.InitialContext ctx = new
        javax.naming.InitialContext();
    Object obj = ctx.lookup("Reservation");
    ReservationHomeRemote home =
        (ReservationHomeRemote)javax.rmi.PortableRemoteObject.narrow
        obj, ReservationHomeRemote.class);
    ReservationRemote reservation = home.create();
    int id = reservation.reserve(userId, tripId, price, type, number,
        expiration);
%>
     ありがとうございます。予約内容が確認されました。<br><br>
    予約番号：<%= id %><br>
<%
  } catch (Exception e) {
%>
     申し訳ありません。この旅行は予約できませんでした。
    <br><%= e.getMessage() %>
<%
  }
%>
<%@ include file="footer.htm"%>
</body>
</html>
```

BMP エンティティ bean

BMP エンティティ bean では、フィールドやパーシスタンスロジックの getter メソッドや setter メソッドをコーディングします。

bean 実装で検出されたコールバックメソッドのパーシスタンスロジックを指定します。パーシスタンスロジックは、次に示すように、データを作成、取り出し、更新、保存または削除します。

- **ejbCreate**　データストアにインスタンス (通常はデータベースの行) を追加します。
- **ejbStore**　現在の値を持つデータストアを更新します。
- **ejbLoad**　データストアからインスタンスを取り出します。
- **ejbRemove**　データストアからインスタンスを削除します。
- **ejbFindByPrimaryKey**　データストアからインスタンスを取り出し、そのプライマリキーを戻します。
- **ejbFindBy***Xxx*　カスタムの取り出しロジックを使用して、データストアからインスタンスを 1 つ以上取り出す場合と取り出さない場合があります。

EJB コンテナは、bean ライフサイクルのさまざまなポイントでこれらのメソッドを呼び出すことにより、データソースのパーシスタンスや一貫性を調整します。

メモ：EJB コンテナは、CMP bean にこれらのメソッドを自動的に実装します。詳細については、374 ページの「CMP エンティティ bean (1.1 仕様)」および 383 ページの「CMP エンティティ bean (2.0 仕様)」を参照してください。

ホームインターフェイス

BMP エンティティ bean のホームインターフェイスは、1 つ以上の `create` メソッド、1 つの `findByPrimary` キーメソッド、およびオプションでカスタム finder メソッドを定義します。

次のコードは、BMP エンティティ bean のリモートホームインターフェイス、および各メソッドから返される例外を示しています。

```
package compass;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface OrderHomeRemote extends javax.ejb.EJBHome {
  public OrderRemote create(String customerId, int tripId, String
        ccType, String ccNumber,
    String ccExpiration) throws CreateException, RemoteException;
  public OrderRemote findByPrimaryKey(Integer pk) throws
        FinderException, RemoteException;
  public java.util.Collection findByCustomer(String customerId) throws
        FinderException, RemoteException;
}
```

コンポーネントインターフェイス

BMP エンティティ bean のコンポーネントインターフェイスは、クライアントから呼び出されるビジネスメソッドを定義します。リモートコンポーネントインターフェイスで定義されるメソッドは、常に RemoteException を投げる必要があり、また、すべてのコンポーネントインターフェイスのメソッドは、bean 実装内のビジネスメソッドから投げられた例外を投げる必要があります。

次のコードは、BMP エンティティ bean のコンポーネントインターフェイスを示しています。

```
package compass;

import java.rmi.RemoteException;

public interface OrderRemote extends javax.ejb.EJBObject {
  public int getOrderId() throws RemoteException;
  public String getUserId() throws RemoteException;
  public void setUserId(String customerId) throws RemoteException;
  public int getTripId() throws RemoteException;
  public void setTripId(int tripId) throws RemoteException;
  public java.sql.Date getOrderDate() throws RemoteException;
  public String getCCType() throws RemoteException;
  public void setCCType(String ccType) throws RemoteException;
  public String getCCNumber() throws java.rmi.RemoteException;
  public void setCCNumber(String ccNumber) throws RemoteException;
  public String getCCExpiration() throws RemoteException;
  public void setCCExpiration(String ccType) throws RemoteException;
}
```

bean 実装

次のコードは、BMP エンティティ bean の bean 実装を示しています。

```
package compass;

import java.sql.*;
import javax.ejb.EJBException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import javax.ejb.RemoveException;

public class OrderBean implements javax.ejb.EntityBean {

  public javax.ejb.EntityContext context;
  // これらのフィールドは、データベーステーブルの列に対応します。
  public int orderId;
  public String userId;
  public int tripId;
  public String ccType;
  public String ccNumber;
  public String ccExpiration;
  public java.sql.Date orderDate;
```

```
public int getOrderId() {
  // orderID はプライマリキーです。
  return ((Integer) context.getPrimaryKey()).intValue();
}
public String getUserId(){
  return userId;
}
public void setUserId(String userId){
  this.userId = userId;
}
public int getTripId(){
  return tripId;
}
public void setTripId(int tripId){
  this.tripId = tripId;
}
public java.sql.Date getOrderDate(){
  return orderDate;
}
public String getCCType(){
  return ccType;
}
public void setCCType(String ccType){
  this.ccType = ccType;
}
public String getCCNumber(){
  return ccNumber;
}
public void setCCNumber(String ccNumber){
  this.ccNumber = ccNumber;
}
public String getCCExpiration(){
  return ccExpiration;
}
public void setCCExpiration(String ccExpiration){
  this.ccExpiration = ccExpiration;
}
// データベースに行を作成します。
public java.lang.Integer ejbCreate(String userId, int tripId, String
      ccType, String ccNumber, String ccExpiration) throws
      CreateException {
```

```
// log() はこのクラスの最後で定義されます。
log("ejbCreate()");

this.userId = userId;
this.tripId = tripId;
this.ccType = ccType;
this.ccNumber = ccNumber;
this.ccExpiration = ccExpiration;

Connection connection=null;
PreparedStatement ps=null;
Statement stmt=null;
ResultSet rs=null;

try {
  javax.naming.InitialContext ctx = new
    javax.naming.InitialContext();
  javax.sql.DataSource ds = (javax.sql.DataSource)
    ctx.lookup("compass");
  connection=ds.getConnection();

  // 新規プライマリキーを自動生成します。
  stmt = connection.createStatement();
  stmt.executeUpdate("UPDATE sequence SET lastkey=lastkey+1 WHERE
    sequence_id='order'");
  rs = stmt.executeQuery("SELECT lastkey FROM sequence WHERE s
    equence_id='order'");
  rs.next();
  orderId=rs.getInt(1);
  log("order:"+orderId);

  // 予約内容を挿入します。
  ps = connection.prepareStatement("INSERT INTO orders (order_id,
    user_id, trip_id, cc_type, cc_number, cc_expiration) VALUES
    (?,?,?,?,?,?)");
  ps.setInt(1, orderId);
  ps.setString(2, userId);
  ps.setInt(3, tripId);
  ps.setString(4, ccType);
  ps.setString(5, ccNumber);
  ps.setString(6, ccExpiration);

  if (ps.executeUpdate() == 1) {
    log("INSERT order succeeded");
  } else {
    log("INSERT order failed");
    throw new CreateException("INSERT order failed");
  }
```

```
    return new Integer(orderId);
  } catch (Exception e) {
    log(e);
    throw new EJBException(e.getMessage());
  } finally {
      try {
        rs.close();
        ps.close();
        connection.close();
      } catch (SQLException e) {
        log(e);
      }
  }
}

public void ejbPostCreate(String userId, int tripId, String ccType,
      String ccNumber, String ccExpiration) {
  // 実装されていません。
}

// データベースから行を取り出します。
public void ejbLoad(){
  log("ejbLoad()");

  orderId = ((Integer) context.getPrimaryKey()).intValue();
  log("Primary key:"+orderId);

  Connection connection = null;
  PreparedStatement ps=null;
  ResultSet rs=null;

  try {
    javax.naming.InitialContext ctx = new
      javax.naming.InitialContext();
    javax.sql.DataSource ds = (javax.sql.DataSource)
      ctx.lookup("compass");
    connection=ds.getConnection();

    ps = connection.prepareStatement("SELECT * FROM orders WHERE
      order_id=?");
    ps.setInt(1, orderId);

    rs = ps.executeQuery();
```

```
    if (rs.next()) {
      log("SELECT order succeeded");
      userId = rs.getString("user_id");
      tripId = rs.getInt("trip_id");
      orderDate = rs.getDate("order_date");
      ccType = rs.getString("cc_type");
      ccNumber = rs.getString("cc_number");
      ccExpiration = rs.getString("cc_expiration");
    } else {
      log("SELECT order failed");
      throw new EJBException("SELECT order failed");
    }
  } catch (Exception e) {
    throw new EJBException(e.getMessage());
  } finally {
      try {
        rs.close();
  ps.close();
        connection.close();
      } catch (Exception e) {
        log(e);
      }
  }
}

// データベースに行を保管します。
public void ejbStore(){
  log("ejbStore()");

  Connection connection = null;
  PreparedStatement ps=null;

  try {
    javax.naming.InitialContext ctx = new
      javax.naming.InitialContext();
    javax.sql.DataSource ds = (javax.sql.DataSource)
      ctx.lookup("compass");
    connection=ds.getConnection();

    ps = connection.prepareStatement("UPDATE orders SET user_id=?,
      trip_id=?, cc_type=?, cc_number=?, cc_expiration=?WHERE
      order_id=?");
    ps.setString(1, userId);
    ps.setInt(2, tripId);
    ps.setString(3, ccType);
    ps.setString(4, ccNumber);
    ps.setString(5, ccExpiration);
    ps.setInt(6, orderId);
```

```
    if (ps.executeUpdate() == 1) {
      log("UPDATE order succeeded");
    } else {
      log("UPDATE order failed");
      throw new EJBException("UPDATE order failed");
    }
  } catch (Exception e) {
    log(e);
    throw new EJBException(e.getMessage());
  } finally {
    try {
      ps.close();
      connection.close();
    } catch (Exception e) {
      log(e);
    }
  }
}

// データベースから行を削除します。
public void ejbRemove() throws RemoveException {

  log("ejbRemove()");

  Connection connection = null;
  PreparedStatement ps = null;

  try {
    javax.naming.InitialContext ctx = new
      javax.naming.InitialContext();
    javax.sql.DataSource ds = (javax.sql.DataSource)
      ctx.lookup("compass");
    connection=ds.getConnection();

    ps = connection.prepareStatement("DELETE FROM orders WHERE
      order_id=?");
    ps.setInt(1, orderId);

    if (ps.executeUpdate() == 1) {
      log("DELETE order succeeded");
    } else {
      log("DELETE order failed");
      throw new RemoveException("DELETE order failed");
    }

  } catch (Exception e) {
      log(e);
      throw new EJBException(e.getMessage());
  } finally {
    try {
      ps.close();
      connection.close();
```

```
    } catch (Exception e) {
      log(e);
    }
  }
}

public void ejbPassivate() {  }
public void ejbActivate() {  }

public void setEntityContext(javax.ejb.EntityContext context) {
  this.context = context;
}

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

// 行を取り出してプライマリキーを返します。
// 戻り値のタイプが PK のクラスであることに注意します。
// CMP の場合、このメソッドは void を返します。
public Integer ejbFindByPrimaryKey(Integer pk) throws
      FinderException {

  log("ejbFindByPrimaryKey()");

  orderId = pk.intValue();

  Connection connection = null;
  PreparedStatement ps=null;
  ResultSet rs=null;

  try {
    javax.naming.InitialContext ctx = new
      javax.naming.InitialContext();
    javax.sql.DataSource ds = (javax.sql.DataSource)
      ctx.lookup("compass");
    connection=ds.getConnection();

    ps = connection.prepareStatement("SELECT count(*) FROM orders
      WHERE order_id=?");
    ps.setInt(1, orderId);

    rs = ps.executeQuery();
    if (rs.next()) {
      throw new javax.ejb.ObjectNotFoundException("Invalid Primary
      Key");
    }
  } catch (Exception e) {
    throw new EJBException(e.getMessage());
  } finally {
    try {
      rs.close();
```

```
      ps.close();
      connection.close();
    } catch (Exception e) {
      log(e);
    }
  }

  return pk;
}

// 顧客の予約内容を検索します。
public java.util.Collection ejbFindByCustomer(String userId) throws
      FinderException {

  log("ejbFindByCustomer():"+userId);

  Connection connection = null;
  PreparedStatement ps=null;
  ResultSet rs=null;

  java.util.Vector list = new java.util.Vector();

  try {
    javax.naming.InitialContext ctx = new
      javax.naming.InitialContext();
    javax.sql.DataSource ds = (javax.sql.DataSource)
      ctx.lookup("compass");
    connection=ds.getConnection();

    ps = connection.prepareStatement("SELECT order_id FROM orders
      WHERE user_id=? ORDER BY order_date desc");
    ps.setString(1, userId);

    rs = ps.executeQuery();

    while (rs.next()) {
      Integer pk = new Integer(rs.getInt(1));
      list.addElement(pk);
    }
```

```
    } catch (Exception e) {
      throw new EJBException(e.getMessage());
    } finally {
      try {
        rs.close();
        ps.close();
        connection.close();
      } catch (Exception e) {
        log(e);
      }
    }
    return list;
  }

  private void log(Object message) {
    System.out.println("-> "+new java.util.Date()+" "+this+"
        "+message);
  }
}
```

EJB デプロイメントディスクリプタ

BMP エンティティ bean の基本的なデプロイメントディスクリプタには、**entity** 要素内に要素が含まれます。後で、アプリケーションアセンブラにより、**assembly-descriptor** 要素内に追加仕様が作成されます。

次のサンプルは、BMP エンティティ bean のデプロイメントディスクリプタ内の要素を示しています。

```
...
  <entity>
      <ejb-name>Order</ejb-name>
      <home>compass.OrderHomeRemote</home>
      <remote>compass.OrderRemote</remote>
      <ejb-class>compass.OrderBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <security-identity><use-caller-identity/></security-identity>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>Order</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
...
```

JRun EJB デプロイメントディスクリプタ

次の例は、BMP エンティティ bean の JRun EJB デプロイメントディスクリプタを示しています。

```
...
    <entity>
      <ejb-name>Order</ejb-name>
      <jndi-name>Order</jndi-name>
    </entity>
...
```

サンプルクライアント

次のステートレスセッション bean の例は、リモートの BMP エンティティ bean へのクライアントアクセスを示しています。

```
...
// 新規の予約内容を作成します。
try {
  Object obj = ctx.lookup("Order");
  OrderHomeRemote home =
        (OrderHomeRemote)javax.rmi.PortableRemoteObject.narrow(obj,
        OrderHomeRemote.class);
  OrderRemote order = home.create(custId, tripId, ccType, ccNumber,
        ccExpiration);
  orderId = order.getOrderId();
  log("Order created:"+orderId);
}
  catch (Exception e) {
    log(e);
  }
  // トランザクションが引き続き有効かどうかをチェックします。
  // 有効でない場合、例外を返してクライアントに通知します。
  if (context.getRollbackOnly()) throw new
        javax.ejb.EJBException("Order failed");

    return orderId;
...
```

BMP の追加検討事項

フィールドを変更するエンティティ bean のビジネスメソッドの呼び出しの後では必ず、デフォルトで、EJB コンテナは `ejbStore` メソッドを呼び出します。jrun-ejb-jar.xml の `alwaysDirty` 要素を true に設定することによって、フィールドの状態にかかわらず、EJB コンテナ `ejbStore` メソッドを呼び出すようにすることができます。

`alwaysDirty` 要素は、CMP bean にも適用されます。

CMP エンティティ bean (1.1 仕様)

EJB 2.0 では、アプリケーションサーバーが EJB 2.0 スタイルの CMP だけではなく、EJB 1.1 スタイルの CMP もサポートする必要があります。JRun EJB 1.1 CMP を使用する場合は、jrun-ejb-jar.xml ファイルでパーシスタンスロジックを指定します。

ヒント：エンタープライズデプロイウィザードを使用すると、パーシスタンスロジックを生成できます。詳細については、390 ページの「エンタープライズデプロイウィザード」を参照してください。

ホームインターフェイス

エンティティ bean のホームインターフェイスには、任意の数の引数を持つ **create** メソッドが含まれている場合と含まれていない場合があります。各 **create** メソッドには、bean 実装の同じ引数に一致する **ejbCreate** メソッドが 1 つ必要です。エンティティ bean には、少なくとも **findByPrimaryKey** の finder メソッドが含まれている必要があります。これは、bean 実装の **ejbFindByPrimaryKey** メソッドに対応しています。リモートホームインターフェイスメソッドは **RemoteException** を投げる必要があり、またすべての **create** メソッドは **CreateException** を投げる必要があります。

次のコードは、CMP 1.1 エンティティ bean のリモートホームインターフェイスを示しています。

```
package samples.cmp11;
import javax.ejb.FinderException;
import java.util.Collection;
import java.rmi.RemoteException;

public interface EmployeeHome extends javax.ejb.EJBHome {
  public Employee create(java.lang.String employeeId, java.lang.String
        firstName,
         java.lang.String lastName, java.lang.String phone)
           throws java.rmi.RemoteException, javax.ejb.CreateException;
  public Employee findByPrimaryKey(java.lang.String pk)
           throws java.rmi.RemoteException, javax.ejb.FinderException;
  public Collection findAll()
           throws java.rmi.RemoteException, javax.ejb.FinderException;
  public Collection findByLastName(String lastName)
           throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

コンポーネントインターフェイス

次のコードは、CMP 1.1 エンティティ bean のリモートコンポーネントインターフェイスを示しています。

```
package samples.cmp11;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Employee extends EJBObject {
  public String getEmployeeId() throws RemoteException;
  public void setEmployeeId(String employeeId) throws RemoteException;
  public String getFirstName() throws RemoteException;
  public void setFirstName(String firstName) throws RemoteException;
  public String getLastName() throws RemoteException;
  public void setLastName(String lastName) throws RemoteException;
  public String getPhone() throws RemoteException;
  public void setPhone(String phone) throws RemoteException;
}
```

bean 実装

CMP 1.1 エンティティ bean の一般的な必要条件は次のとおりです。

- `javax.ejb.EntityBean` を実装する必要があります。
- public である必要があります。
- abstruct ではいけません。

エンティティ bean の bean 実装には、`setEntityContext`、必要なコールバックメソッドおよびビジネスメソッドが含まれます。

次のコードは、CMP 1.1 エンティティ bean のリモートの bean 実装を示しています。

```
package samples.cmp11;

import javax.ejb.EntityContext;
import javax.ejb.RemoveException;
import javax.ejb.FinderException;
import java.util.Collection;
import java.rmi.RemoteException;

public class EmployeeBean implements javax.ejb.EntityBean {

    private EntityContext context;
    // フィールドは、データベースの列に対応しています。
    public String employeeId;
    public String firstName;
    public String lastName;
    public String phone;
```

```
    public String ejbCreate(String employeeId, String firstName,
        String lastName, String phone) {
        this.employeeId = employeeId;
        this.firstName = firstName;
        this.lastName = lastName;
        this.phone = phone;
        return null;
    }

    public void ejbPostCreate(String employeeId, String firstName,
        String lastName, String phone) {
    }

    public String getEmployeeId() {
        return employeeId;
    }
    public void setEmployeeId(String employeeId) {
        this.employeeId = employeeId;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
    public void setEntityContext(EntityContext context) {
        this.context = context;
    }
    public void unsetEntityContext() {
        this.context = null;
    }
    // CMP は、これらのコールバックメソッドを自動処理します。
    public void ejbActivate() {  }
    public void ejbPassivate() {  }
    public void ejbLoad() {  }
    public void ejbStore() {  }
    public void ejbRemove() throws RemoveException {  }
}
```

EJB デプロイメントディスクリプタ

CMP 1.1 エンティティ bean のデプロイメントディスクリプタには、**entity** 要素内に要素が含まれます。jrun-ejb-jar.xml ファイルのパーシスタンスアクションを指定する必要もあります。また、アプリケーションアセンブル担当者は、**assembly-descriptor** 要素内に追加仕様を作成できます。

次のサンプルは、CMP 1.1 エンティティ bean のデプロイメントディスクリプタ内の基本要素を示しています。

```
...
  <entity>
    <ejb-name>Employee</ejb-name>
    <ejb-class>samples.cmp11.EmployeeBean</ejb-class>
    <home>samples.cmp11.EmployeeHome</home>
    <remote>samples.cmp11.Employee</remote>
    <persistence-type>Container</persistence-type>
    <primkey-field>employeeId</primkey-field>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>True</reentrant>
   <cmp-version>1.x</cmp-version>
    <cmp-field>
        <field-name>employeeId</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>firstName</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>lastName</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>phone</field-name>
    </cmp-field>
</entity>
...
```

JRun EJB デプロイメントディスクリプタ

jrun-ejb-jar.xml ファイルには、CMP パーシスタンスのアクションなど、JRun 特有のデプロイ設定が含まれます。各アクション (作成、ロード、保管など) は、必要に応じてデータベースにアクセスする SQL ステートメントに対応します。次のようにデータソースを指定します。

- **メソッド特有のデータソース**　EJB コンテナは、まず **source** 要素でパーシスタンスアクション用に指定されたデータソースを検索します。
- **デスクリプタワイドデータソース**　メソッド特有データソースが指定されていない場合、EJB コンテナは最上位の **source** 要素を使用します。
- **デフォルトデータソース**　データソースが指定されていない場合、EJB コンテナはデフォルトのデータソースを使用します。デフォルトデータソースは、JRun サーバーの起動時にバインドされる最初のデータソースです。

次のサンプルは、EJB 1.1 CMP エンティティ bean の JRun EJB デプロイメントディスクリプタを示しています。

```
...
<entity>
      <ejb-name>Employee</ejb-name>
      <jndi-name>Employee</jndi-name>
      <jdbc-mappings>
        <jdbc-mapping>
          <name>create</name>
          <statement>
           <action>INSERT INTO employees (employee_id , first_name ,
              last_name, phone) VALUES ( ?, ? , ? , ?)</action>
            <source>samples</source>
            <params>
              <param>
                 <name>employeeId</name>
                 <type>VARCHAR</type>
              </param>
              <param>
                 <name>firstName</name>
                 <type>VARCHAR</type>
              </param>
              <param>
                 <name>lastName</name>
                 <type>VARCHAR</type>
              </param>
              <param>
                 <name>phone</name>
                 <type>VARCHAR</type>
              </param>
            </params>
          </statement>
        </jdbc-mapping>
        <jdbc-mapping>
          <name>load</name>
          <statement>
            <action>SELECT employee_id, first_name, last_name, phone
              FROM employees WHERE employee_id=?</action>
            <source>samples</source>
            <params>
              <param>
                 <name>employeeId</name>
                 <type>VARCHAR</type>
              </param>
            </params>
            <fields>
              <field>employeeId</field>
              <field>firstName</field>
              <field>lastName</field>
              <field>phone</field>
            </fields>
          </statement>
```

```
</jdbc-mapping>
<jdbc-mapping>
  <name>remove</name>
  <statement>
    <action>DELETE FROM employees WHERE employee_id=?</action>
    <source>samples</source>
    <params>
      <param>
         <name>employeeId</name>
         <type>VARCHAR</type>
      </param>
    </params>
  </statement>
</jdbc-mapping>
<jdbc-mapping>
  <name>store</name>
  <statement>
    <action>UPDATE employees SET first_name=?, last_name=?,
      phone=?WHERE employee_id=?</action>
    <source>samples</source>
    <params>
      <param>
         <name>firstName</name>
         <type>VARCHAR</type>
      </param>
      <param>
         <name>lastName</name>
         <type>VARCHAR</type>
      </param>
      <param>
         <name>phone</name>
         <type>VARCHAR</type>
      </param>
      <param>
         <name>employeeId</name>
         <type>VARCHAR</type>
      </param>
    </params>
  </statement>
</jdbc-mapping>
<jdbc-mapping>
  <name>findAll</name>
  <statement>
    <action>SELECT employee_id FROM employees</action>
    <source>samples</source>
    <fields>
      <field>employeeId</field>
    </fields>
  </statement>
</jdbc-mapping>
<jdbc-mapping>
  <name>findByLastName</name>
```

```
            <statement>
              <action>SELECT employee_id FROM employees WHERE
                last_name=?1</action>
              <source>samples</source>
              <params>
                <param>
                   <name>lastName</name>
                   <type>VARCHAR</type>
                </param>
              </params>
              <fields>
                <field>employeeId</field>
              </fields>
            </statement>
          </jdbc-mapping>
          <jdbc-mapping>
            <name>findByPrimaryKey</name>
            <statement>
              <action>SELECT employee_id FROM employees WHERE
                employee_id=?</action>
              <source>samples</source>
              <params>
                <param>
                   <name>employeeId</name>
                   <type>VARCHAR</type>
                </param>
              </params>
              <fields>
                <field>employeeId</field>
              </fields>
            </statement>
          </jdbc-mapping>
        </jdbc-mappings>
      </entity>
...
```

サンプルクライアント

サンプルクライアントを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

JRun CMP 1.1 の追加検討事項

JRun EJB コンテナは、次の機能によって CMP 1.1 開発の負担を軽減します。

- テーブルの自動作成と自動削除
- SQL の自動生成
- 1 つのパーシスタンスアクションによる複数の SQL ステートメント実行
- パーシスタンスアクションの強制

テーブルの自動作成と自動削除

jrun-ejb-jar.xml ファイルで **create-table** 要素が true に設定されている場合、JRun は、**createTable jdbc-mapping** 要素で指定されたとおりにテーブル作成 DDL (Data Definition Language：データ定義言語) を実行します。また、JRun は **delete-table** 要素と **deleteTable jdbc-mapping** 要素を使用して、テーブル削除ロジックを実行します。この機能は、常に正式なデータベーステーブルのコピーを使用してテストを開始できるので、開発サイクルの段階で役立ちます。次の jrun-ejb-jar.xml の抜粋は、テーブルの自動作成と自動削除のステートメントを示しています。

```
...
<create-table>true</create-table>
<delete-table>true</delete-table>
...
<jdbc-mapping>
  <name>createTable</name>
  <statement>
    <action>CREATE TABLE public.EntityTestBeanTable (id INTEGER, value
        VARCHAR(32))</action>
    <source>samples</source>
    <params />
    <fields />
  </statement>
</jdbc-mapping>
<jdbc-mapping>
  <name>deleteTable</name>
  <statement>
    <action>DROP TABLE public.EntityTestBeanTable</action>
    <source>samples</source>
    <params />
    <fields />
</statement>
</jdbc-mapping>
...
```

create-table 要素が true に設定されていて、**createTable jdbc-mapping** 要素がファイルにない場合、JRun は「SQL の自動生成」で説明するようにデフォルトの DDL を生成します。

SQL の自動生成

基本的なメソッドステートメント (**create**、**load**、**store**、**remove**、**findByPrimaryKey**、**createTable**、および **deleteTable**) のいずれかに jrun-ejb-jar.xml ファイルの **jdbc-mapping** が含まれていない場合、JRun は、ejb-jar.xml ファイルで指定されている EJB 名と CMP フィールドに基づいて、デフォルトの SQL 実装を作成します。この機能を使用するには、EJB 名がデータベースのテーブル名に一致し、**cmp-field** 要素がそのデータベーステーブルの列名に一致している必要があります。

メモ：テーブルを作成および削除する場合、EJB コンテナは、create-table および delete-table 要素が true に設定されている場合にのみ SQL を生成します。

1 つのパーシスタンスアクションで複数の SQL ステートメントを実行

JRun では、各 jdbc-mapping 要素に対して複数の statement 要素を指定できます。この機能を使用して、各パーシスタンスアクションによる複数のデータベースへのアクセスなど、カスタマイズしたパーシスタンス処理を実行できます。EJB コンテナは、jdbc-mapping 要素に定義された順序でステートメントを実行します。

次の例は、jdbc-mapping 要素に関連付けられた statement 要素のコーディング方法を示しています。

```
...
<jdbc-mapping>
  <name>create</name>
  <statement>
    <action>INSERT INTO employee( empid , lastname, firstname)
      VALUES( ?, ? , ? , ? , ?)</action>
    <source>samples</source>
    <params>
      <param>
        <name>empid</name>>
        <type>VARCHAR</type>
      </param>
      <param>
        <name>lastname</name>
        <type>VARCHAR</type>
      </param>
      <param>
        <name>firstname</name>
        <type>VARCHAR</type>
      </param>
    </params>
    <fields />
  </statement>
  <statement>
...
  </statement>
</jdbc-mapping>
...
```

CMP エンティティ bean (2.0 仕様)

CMP 2.0 には、パーシスタンスの機能の大部分を EJB コンテナに移す高度なパーシスタンスモデルが用意されています。パーシスタンスモデルには、コードへの影響を最小限に抑えながら実際のデータの複雑さを EJB が反映できるようにする、リレーションシップの宣言仕様が含まれています。

エンティティ bean の開発者は、EJB クエリ言語 (EJB-QL) を用いて ejb-jar.xml ファイルの基本的なパーシスタンス情報を指定するだけで、後は CMP 2.0 を使用してビジネスロジックに集中することができます。さらに、エンタープライズデプロイウィザードのようなツールによって、デプロイメントディスクリプタの作成および管理を簡単に行うことができます。

このトピックでは、基本的な CMP 2.0 の例と JRun 特有の CMP 2.0 の使用方法および機能について説明します。CMP 2.0、EJB-QL、およびその他の EJB 2.0 機能の詳細については、EJB 2.0 を解説している業界誌を参照してください。このマニュアルの序章にはこれらの本のリストが記載されています。

ヒント：エンタープライズデプロイウィザードを使用すると、パーシスタンスロジックを生成できます。詳細については、390 ページの「エンタープライズデプロイウィザード」を参照してください。

ホームインターフェイス

エンティティ bean のホームインターフェイスには、任意の数の引数を持つ `create` メソッドが含まれている場合と含まれていない場合があります。各 `create` メソッドには、bean 実装の引数と同じ引数を持つ、一致する `ejbCreate` メソッドが 1 つ必要です。エンティティ bean には、少なくとも `findByPrimaryKey` の finder メソッドが含まれている必要があります。これは、bean 実装の `ejbFindByPrimaryKey` メソッドに対応しています。リモートホームインターフェイスメソッドは `RemoteException` を投げる必要があり、またすべての `create` メソッドは `CreateException` を投げる必要があります。

次のコードは、EJB 2.0 CMP エンティティ bean のリモートホームインターフェイスを示しています。

```
package samples.cmp20;

import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import java.util.Collection;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface EmployeeHome extends EJBHome {
  public Employee create(String employeeId, String firstName, String
        lastName, String phone) throws RemoteException,
        CreateException;
    public Employee findByPrimaryKey(String employeeId) throws
        FinderException, RemoteException;
    public Collection findAll() throws FinderException,
        RemoteException;
    public Collection findByLastName(String lastName) throws
        FinderException, emoteException;
}
```

コンポーネントインターフェイス

次のコードは、2.0 CMP エンティティ bean のリモートコンポーネントインターフェイスを示しています。

```
package samples.cmp20;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Employee extends EJBObject {
  public String getEmployeeId();
  public void setEmployeeId(String userId);
  public String getFirstName();
  public void setFirstName(String firstName);
  public String getLastName();
  public void setLastName(String lastName);
  public String getPhone();
  public void setPhone(String phone);
}
```

bean 実装

EJB 2.0 CMP エンティティ bean の一般的な必要条件は次のとおりです。

- `javax.ejb.EntityBean` を実装する必要があります。
- public である必要があります。
- abstract である必要があります。
- CMP フィールドの変数を定義しません。
- abstract の setter メソッドと getter メソッドを各 CMP フィールドに定義する必要があります。

エンティティ bean の bean 実装には、`setEntityContext`、必要なコールバックメソッドおよびビジネスメソッドが含まれます。

次のコードは、CMP 2.0 エンティティ bean のリモートの bean 実装を示しています。

```
package samples.cmp20;

import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.EJBException;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
import javax.ejb.FinderException;

public abstract class EmployeeBean implements EntityBean {
  private EntityContext context;

  public void ejbLoad() throws RemoteException {  }
  public void ejbStore() throws RemoteException {  }
```

```
  public void setEntityContext(EntityContext context) throws
        RemoteException {
    this.context = context;
  }

  public void unsetEntityContext() throws RemoteException,
        EJBException {
    this.context = null;
  }

  public void ejbRemove() throws RemoteException {  }
  public void ejbActivate() throws RemoteException {  }
  public void ejbPassivate() throws RemoteException {

  public String ejbCreate(String employeeId, String firstName, String
        lastName, String phone) throws RemoteException,
        CreateException {
    setEmployeeId(employeeId);
    setFirstName(firstName);
    setLastName(lastName);
    setPhone(phone);
    return null;
  }

  public void ejbPostCreate(String employeeId, String firstName,
        String lastName, String phone) {
  }

  // CMP フィールドのメソッドです。abstract である必要があります。
  public abstract String getEmployeeId();
  public abstract void setEmployeeId(String userId);
  public abstract String getFirstName();
  public abstract void setFirstName(String firstName);
  public abstract String getLastName();
  public abstract void setLastName(String lastName);
  public abstract String getPhone();
  public abstract void setPhone(String phone);
}
```

EJB デプロイメントディスクリプタ

CMP 2.0 エンティティ bean のデプロイメントディスクリプタには、entity 要素内に要素が含まれます。また、EJB-QL を使用して、デフォルト以外のすべてのメソッドに対して query 要素を定義します。

次の例は、CMP 2.0 エンティティセッション bean のデプロイメントディスクリプタ内の基本要素を示したものであり、findAll と findByLastName メソッドの query 要素が含まれています。

```
...
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
        JavaBeans 2.0//EN" "http://java.sun.com/j2ee/dtds/
        ejb-jar_2_0.dtd">
<ejb-jar>
   <enterprise-beans>
      <entity>
         <ejb-name>Employee</ejb-name>
         <ejb-class>samples.cmp20.EmployeeBean</ejb-class>
         <home>samples.cmp20.EmployeeHome</home>
         <remote>samples.cmp20.Employee</remote>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.String</prim-key-class>
         <reentrant>True</reentrant>
         <primkey-field>employeeId</primkey-field>
         <abstract-schema-name>employeeschema</abstract-schema-name>
         <cmp-version>2.x</cmp-version>
         <cmp-field>
            <field-name>employeeId</field-name>
         </cmp-field>
         <cmp-field>
            <field-name>firstName</field-name>
         </cmp-field>
         <cmp-field>
            <field-name>lastName</field-name>
         </cmp-field>
         <cmp-field>
            <field-name>phone</field-name>
         </cmp-field>
         <query>
            <query-method  >
               <method-name>findAll</method-name>
               <method-params />
            </query-method>
            <return-type-mapping>Local</return-type-mapping>
            <ejb-ql>SELECT OBJECT(o) FROM employeeschema AS o</ejb-ql>
         </query>
         <query>
            <query-method  >
               <method-name>findByLastName</method-name>
               <method-params>
                  <method-param>java.lang.String</method-param>
```

```
            </method-params>
          </query-method>
          <return-type-mapping>Local</return-type-mapping>
          <ejb-ql>SELECT OBJECT(o) FROM employeeschema AS o WHERE
       o.lastName = ?1</ejb-ql>
        </query>
      </entity>
   </enterprise-beans>
</ejb-jar>
...
```

JRun EJB デプロイメントディスクリプタ

jrun-ejb-jar.xml ファイルには、JNDI 名やデータソースなど、JRun 特有のデプロイ設定が含まれます。

次のサンプルは、EJB 2.0 CMP エンティティ bean の JRun EJB デプロイメントディスクリプタを示しています。

```
...
<jrun-ejb-jar>
  <source>compass</source>
  <enterprise-beans>
    <entity>
      <ejb-name>Employee</ejb-name>
      <jndi-name>Employee</jndi-name>
    </entity>
  </enterprise-beans>
...
```

サンプルクライアント

サンプルクライアントを表示するには、samples JRun サーバーを起動し、ブラウザで http://localhost:8200/techniques を開きます。

JRun CMP 2.0 の追加検討事項

このセクションでは、次のように JRun 4 EJB コンテナ特有の開発とデプロイの検討事項を説明します。

- テーブルの自動作成
- データソースの検討事項
- データベースの検討事項
- 詳細情報

テーブルの自動作成

JRun EJB コンテナは、デプロイメントディスクリプタによって暗黙に指定されるデータベーステーブルを自動的に生成します。JRun は、各 CMP 2.0 bean に対して `abstract-schema-name` 要素と同じ名前のテーブルを生成します。また、サポートテーブルも自動的に生成します。これにより、JRun は開発者の対話を最小限にとどめて CMP 2.0 エンティティ beans をデプロイでき、また、EJB の移植性も保証されます。

現在 JRun CMP 2.0 実装では、既存のテーブルとの統合は行われません。既存のデータにアクセスできるエンティティ bean を作成するには、374 ページの「CMP エンティティ bean (1.1 仕様)」で説明した CMP 1.1 を使用します。

データソースの検討事項

CMP 2.0 の場合、JRun EJB コンテナは次のデータソースを使用します。

- **jrun-ejb-jar.xml source 要素**　EJB コンテナは ejb-jar.xml ファイルにあるすべてのエンティティ bean をデプロイするために指定の JRun データソースを使用します。データソースは、JRun サーバーの SERVER-INF/jrun-resources.xml ファイルで定義する必要があります。
- **DefaultDataSource**　`source` 要素が ejb-jar.xml ファイルに指定されていない場合、JRun は ejb-jar.xml ファイルにあるすべてのエンティティ bean をデフォルトのデータソースを使用してデプロイします。デフォルトのデータソースは、JRun サーバーの起動時にバインドされる最初のデータソースです。

別々の ejb-jar.xml ファイルで定義された 2 つの異なる CMP 2.0 EJB には同一の `abstract-schema-name` が存在できるので、JRun で個別のテーブルが生成されるように、ejb-jar.xml ファイルごとにデータソースを定義する必要があります。

メモ：テーブル名の重複を最小限に抑えるには、ターゲットのデータソースに、ejb-jar.xml ファイルの abstract-schema-name 要素と同じ名前のテーブルが存在しないようにします。

SQL Server と JRun JDBC ドライバの併用

SQL Server データソースと JRun のデータベースドライバを使用する場合は、次の jrun-resources.xml ファイルの例に示すように、JRun のデータソース URL に `SelectMethod=cursor` を追加します。

```
<url>jdbc:macromedia:sqlserver://
        host:1433;databasename=JRun;SelectMethod=cursor
        </url>
```

データベースの検討事項

次の表で、JRun EJB コンテナが CMP 2.0 エンティティ bean 用に生成するテーブルを説明します。

表	説明
abstract-schema-name	JRun は、ejb-jar.xml ファイルに各 `abstract-schema-name` 要素のテーブルを生成します。このテーブルには、cmp-field 要素に対応する列、および JRun 内部でインスタンスとの関連付けに使用される列 (*abstract-schema-name*$JRunID) があります。
*abstract-schema-name*_view	データ取り出しの最適化に使用します。
jrun_id	内部 ID の生成に使用します。
jrun_insts	生成されたすべてのインスタンスのタイプとインスタンス ID が含まれます。
jrun_props	cmp-field プロパティのメタデータが含まれます。
jrun_types	デプロイされた各 CMP 2.0 エンティティ bean のエントリが 1 つずつ含まれます。

開発サイクルの過程で、JRun CMP 2.0 のデータベーステーブルおよびデータのクリーンアップが必要になることがあります。生成したすべてのテーブルとデータをクリアするには、次のユーティリティを使用します。

```
java -classpath classpath jrunx.persistence.greylock.InitDatabase
        driver url user パス
```

クラスパスには、<JRun のルートディレクトリ >/lib/jrun.jar と、指定したデータベースドライバのクライアントクラスのための JAR ファイルが含まれている必要があります。

パラメータは、次のとおりです。

- **driver**　データベースドライバの完全修飾クラス名です。
- **url**　JRun EJB テーブルを含むデータベースの URL です。
- **user**　データベースのユーザー名です。
- **pass**　データベースのパスワードです。

次の例は、サンプルのデータベースのテーブルをクリーンアップします。

```
java -classpath c:/jrun4b3/lib/jrun.jar;c:/jrun4b3/pointbase/lib/
        pbclient42re.jar
        jrunx.persistence.greylock.InitDatabase
        com.pointbase.jdbc.jdbcUniversalDriver
        jdbc:pointbase:server://127.0.0.1:9192/samples PBPUBLIC PBPUBLIC
```

詳細情報

JRun CMP 2.0 の詳細については、次のリソースを参照してください。

- **samples JRun サーバー**　http://localhost:8200 からアクセスできる samples の JRun サーバーには、CMP 2.0 を使用したさまざまな例が紹介されています。
- **ディスクリプタドキュメント**　ドキュメンテーションのホームページ　(<JRun　のルートディレクトリ >/docs/dochome.htm) で利用できる JRun デプロイメントディスクリプタのオンラインドキュメントには、jrun-ejb-jar.xml ファイルで利用できる要素の詳細情報が紹介されています。

このマニュアルのリソースおよび例を使用するだけでなく、EJB 2.0 に関する業界誌も参照してください。このマニュアルの序章にはこれらの本のリストが記載されています。

エンタープライズデプロイウィザード

JRun エンタープライズデプロイウィザードは、既存の EJB デプロイメントディスクリプタの編集、JAR ファイルへのパッケージ化、および JRun へのデプロイなどに使用する Swing ベースのツールです。特に、エンタープライズデプロイウィザードのオブジェクトリレーショナルマッピング機能では、エンティティ bean の開発処理を簡素化できます。

エンタープライズデプロイウィザードはスタンドアローンツールとして実行するか、または IDE のプラグインとして実行できます。

エンタープライズデプロイウィザードを開始するには、<JRun のルートディレクトリ >/bin/jrunwizard.exe (Windows の場合) または <JRun のルートディレクトリ >/bin/jrunwizard (UNIX の場合) を実行します。

IDE にエンタープライズデプロイウィザードをインストールするには、コンソールウィンドウで <JRun のルートディレクトリ >/lib ディレクトリを開き、次のコマンドを実行します。

```
java -jar jrunwizard-installer.jar
```

現在サポートされている IDE のリストについては、リリースノートを参照してください。

エンタープライズデプロイウィザードは、コンテキスト対応のオンラインヘルプシステムです。使用方法の詳細については、オンラインヘルプを参照してください。

EJB と XDoclet

JRun は XDoclet と統合できます。XDoclet は、EJB、Web アプリケーション、および JSP タグライブラリのコードおよびデプロイメントディスクリプタを生成するオープンソースツールです。

特に EJB の場合、XDoclet を使用して次のものを生成できます。

- 入力としてビジネスメソッドを含むアブストラクトクラスを使用する bean 実装に必要な EJB コールバックメソッド
- bean 実装クラスのリモート、ローカル、ホーム、およびローカルホームインターフェイス
- デプロイメントディスクリプタ

Using XDoclet with EJBs

bean 実装内で jrun.xml 属性と JavaDoc スタイルのコメントを組み合わせて使用することによって、XDoclet が実行する処理を制御できます。

EJB と XDoclet を併用するには

1. XDoclet コメントを含んでいる bean 実装クラスを作成します。XDoclet のコメントは、@ マークで始まります。XDoclet で EJB のコールバックメソッド (たとえば、`ejbPassivate`) を含む下位クラスを生成する場合は、クラスをアブストラクトタイプとして宣言します。
2. JRUN サーバーの jrun.xml ファイルにある `XDocletService` セクション内の `ejbSourceFiles` 属性がサポートしているネーミング規則を使用して、EJB に名前を付けるか、または、EJB のファイル名をカバーする `ejbSourceFiles` 属性を追加します。デフォルトのネーミング規則は `*Bean.java` です。
3. XDoclet サポートが有効になっているディレクトリにクラスを保存します。JRun サーバーの jrun.xml ファイルの `XDocletService` セクションの `watchedEJBDirectory` 属性でディレクトリを指定することで、EJB に対する XDoclet サポートを有効にします。

   デフォルトのディレクトリは *server_root*/default-ear/default-ejb です。

   このディレクトリは、デフォルトではコメントとして処理されます。XDoclet が jrun.xml ファイルの `watchedEJBDirectory` 属性を監視できるようにするには、この属性のコメントを無効にする必要があります。JRun が監視するディレクトリを無制限に追加することもできます。

ソースファイルを保存する場合、JRun は、下位の bean 実装クラス、関連付けられたインターフェイス、およびデプロイメントディスクリプタを生成します。ディレクトリが auto-deploy ディレクトリの場合、JRun は EJB もデプロイします。

JRun は、ejb-jar.xml ファイルに要素を生成する基本的な XDoclet タグ、および jrun-ejb-jar.xml ファイルに要素を生成する JRun 特有の XDoclet タグをサポートします。

基本的な XDoclet タグ

基本的な XDoclet タグは、`@ejb:` という接頭辞で始まります。これらのタグは、基本的な XDoclet のアーキテクチャの一部です。タグとそのパラメータのリストなどの詳細については、XDoclet の Web サイト (http://xdoclet.sourceforge.net/) をご覧ください。

次のテーブルでは、基本的な XDoclet タグを説明します。

タグ	説明	使用方法
@ejb:bean	EJB に関する基本的な情報を提供します。パラメータには、type、name、primkey-field、および cmp-version が含まれます。	すべて (ただし、すべてのパラメータがすべての EJB タイプに適用されるわけではありません) クラスレベルタグ
@ejb:home	ホームインターフェイス情報を提供します。パラメータには、extends および remote-class が含まれます。	エンティティ bean およびセッション bean クラスレベルタグ
@ejb:interface	コンポーネントインターフェイス (リモートまたはローカル) 情報を提供します。パラメータには、extends、remote-class、および local-class が含まれます。	エンティティ bean およびセッション bean クラスレベルタグ
@ejb:finder	ホームインターフェイスの finder メソッドを定義する情報を提供します。パラメータには、signature および role-name が含まれます。	エンティティ bean クラスレベルタグ
@ejb:select	ホームインターフェイスの select メソッドを定義する情報を提供します。パラメータには、signature および query が含まれます。	エンティティ bean (CMP 2.0 のみ) クラスレベルタグ
@ejb:pk	プライマリキーを定義する情報を提供します。パラメータには、class、package および extends が含まれます。	エンティティ bean クラスレベルタグ
@ejb:env-entry	環境エントリを定義する情報を提供します。パラメータには、name、type および value が含まれます。	すべて クラスレベルタグ
@ejb:resource-ref	リソースリファレンスを定義します。パラメータには、res-name、res-type、および res-auth が含まれます。	すべて クラスレベルタグ
@ejb:resource-env-ref	リソース環境リファレンスを提供します。パラメータには、name および type が含まれます。	すべて クラスレベルタグ
@ejb:security-role-ref	セキュリティロールリファレンスを提供します。パラメータには、role-name および role-link が含まれます。	エンティティ bean およびセッション bean クラスレベルタグ

タグ	説明	使用方法
@ejb:transaction	トランザクションの動作を指定します。次の値を受け取る type パラメータを持ちます。 • NotSupported • Supports • Required • RequiresNew • Mandatory • Never	すべて クラスレベルタグ
@ejb:permission	リモートおよびホームインターフェイスのメソッドへのアクセスを指定します。	エンティティ bean およびセッション bean クラスレベルタグ
@ejb:security-identity	呼び出し側のセキュリティ ID または特定の run-as ID のどちらを使用するかを定義します。パラメータには description、run-as、および use-caller-identity が含まれます。	すべて クラスレベルタグ
@ejb:interface-method	ローカルまたはリモートインターフェイスにメソッドを表示するかどうかを指定します。view-type パラメータを持ちます。	セッション bean およびエンティティ bean メソッドレベルタグ
@ejb:home-method	home メソッドおよびそれを表示するインターフェイスを識別します。	セッション bean およびエンティティ bean メソッドレベルタグ
@ejb:create-method	`ejbCreate` メソッドを識別します。	セッション bean およびエンティティ bean メソッドレベルタグ
@ejb:transaction	メソッドのトランザクション動作を指定します。	セッション bean およびエンティティ bean メソッドレベルタグ
@ejb:permission	特定のメソッドに対するアクセスを指定します。	セッション bean およびエンティティ bean メソッドレベルタグ

JRun 特有の XDoclet タグ

JRun 特有の XDoclet タグは、**`@jrun:`** という接頭辞で始まります。次の表で、JRun 特有の XDoclet タグのリストを説明します。

タグ	説明	使用方法
@jrun:always-dirty	エンティティ bean のフィールドに変更がない場合でも、トランザクションの最後にデータソースと強制的に同期させることができます。true または false を指定します。	エンティティ bean クラスレベルタグ
@jrun:cluster-home	ホームオブジェクトをクラスタリングするかどうか指定します。デフォルトは true です。この要素を使用して、特定の bean のデフォルトの動作を無効にできます。true または false を指定します。	セッション bean およびエンティティ bean クラスレベルタグ
@jrun:cluster-object	EJB オブジェクトをクラスタリングするかどうか指定します。デフォルトは true です。この要素を使用して、特定の bean のデフォルトの動作を無効にできます。true または false を指定します。	セッション bean およびエンティティ bean クラスレベルタグ
@jrun:commit-option	EJB 2.0 仕様のセクション 10.5.9 および 12.1.9 のコミットオプションを指定します。有効な値は A、B、および C です。	エンティティ bean クラスレベルタグ
@jrun:ejb-local-ref	bean 開発者が提供した ejb-ref-name とその JNDI 名の間のマッピングを指定します。デプロイ担当者が実際の JNDI 名を提供します。パラメータには ejb-ref-name および jndi-name が含まれます。	すべて クラスレベルタグ
@jrun:ejb-ref	bean 開発者が提供した ejb-ref-name とその JNDI 名の間のマッピングを指定します。デプロイ担当者が実際の JNDI 名を提供します。パラメータには ejb-ref-name および jndi-name が含まれます。	すべて クラスレベルタグ
@jrun:instance-pool	StatelessSessionBean インスタンスプールの最大サイズおよび最小サイズパラメータを指定します。このタグには、maximum-size および minimum-size パラメータが必要です。	ステートレスセッション bean クラスレベルタグ

タグ	説明	使用方法
@jrun:jdbc-mappings	ejb-jar.xml ファイルで宣言されていない JRun 特有の パーシスタンス情報を指定します。create、load、store、find および remove の各メソッドに使用する SQL を指定します。	エンティティ bean (CMP 1.1 のみ) クラスレベルタグ
@jrun:jndi-name	bean またはリソースがバインドされる JNDI 名を指定します。	すべて クラスレベルタグ
@jrun:message-driven-destination =	MDB のトピックまたはキュー名を指定します。name パラメータを持っています。	メッセージ駆動型 bean クラスレベルタグ
@jrun:message-driven-subscription	永続サブスクリプション処理に JRun が使用するユーザー ID を指定します。client-id パラメータを持ちます。	メッセージ駆動型 bean クラスレベルタグ
@jrun:resource- env-ref	bean 開発者が提供した resource-env-name とその JNDI 名の間のマッピングを指定します。デプロイ担当者が実際の JNDI 名を提供します。パラメータには resource-env-ref-name、jndi-name および mdb-destination が含まれます。	すべて クラスレベルタグ
@jrun:resource-ref	bean 開発者が提供した resource-name とその JNDI 名の間のマッピングを指定します。デプロイ担当者が実際の JNDI 名を提供します。パラメータには resource-ref-name、jndi-name、user および password が含まれます。	すべて クラスレベルタグ
@jrun:timeout	ステートフルセッション bean がパッシベートされるまでのアイドル時間を秒単位で指定します。	ステートフルセッション bean クラスレベルタグ
@jrun:tx-domain	bean のトランザクションが実行されるトランザクションドメイン名を指定します。パラメータには、name、action および source が含まれます。	すべて クラスレベルタグ
@jrun:jdbc-mapping	作成、ロードなど CMP 1.1 の操作情報を指定します。パラメータには、name、action および source が含まれます。	エンティティ bean (CMP 1.1 のみ) メソッドレベルタグ

タグ	説明	使用方法
@jrun:jdbc-mapping-field	SELECT ステートメントが返す列に対応する特定の EJB 変数フィールドを指定します。	エンティティ bean (CMP 1.1 のみ) メソッドレベルタグ
@jrun:jdbc-mapping-param	作成したステートメントの疑問符 (?) に対応するパラメータ名とタイプを指定します。パラメータには、name および type が含まれます。	エンティティ bean (CMP 1.1 のみ) メソッドレベルタグ

JRun 特有の XDoclet タグの多くは、jrun-ejb-jar.xml ファイルの要素にマッピングされます。このファイルの詳細については、JRun ドキュメンテーションのホームページで利用できるディスクリプタのオンラインドキュメントを参照してください。XDoclet を使用するオンラインの EJB の例については、samples JRun サーバーを参照してください。

XDoclet を使用した EJB の例

次の例は、XDoclet のサポートを有効にする CMP 1.1 エンティティ bean を示しています。

```
package samples.cmp11.xdoclet;

import javax.ejb.EntityContext;
import javax.ejb.RemoveException;
import javax.ejb.FinderException;
import java.util.Collection;
import java.rmi.RemoteException;

/**
 * @ejb:bean type="CMP"
 *           name="Employee"
 *    primkey-field="employeeId"
 *           cmp-version="1.x"
 * @ejb:home remote-class="samples.cmp11.xdoclet.EmployeeHome"
 * @ejb:interface remote-class="samples.cmp11.xdoclet.Employee"
 * @ejb:finder signature="java.util.Collection findAll()"
 * @ejb:finder signature="java.util.Collection findByLastName(String
       lastName)"
 * @ejb:pk class="java.lang.String"
 * @ejb:transaction type="Required"
 *
 * @jrun:jndi-name Employee
 * @jrun:jdbc-mappings
 */
```

```
public class EmployeeBean implements javax.ejb.EntityBean {

    private EntityContext context;
    public String employeeId;
    public String firstName;
    public String lastName;
    public String phone;

    /**
     * @ejb:create-method
     * @jrun:jdbc-mapping name="create"
     *                 source="samples"
     *                 action="INSERT INTO employees (employee_id ,
     *                         first_name , last_name, phone) VALUES
     *                         ( ? , ? , ? , ?)"
     * @jrun:jdbc-mapping-param name="employeeId" type="VARCHAR"
     * @jrun:jdbc-mapping-param name="firstName" type="VARCHAR"
     * @jrun:jdbc-mapping-param name="lastName" type="VARCHAR"
     * @jrun:jdbc-mapping-param name="phone" type="VARCHAR"
     */
    public String ejbCreate(String employeeId, String firstName,
        String lastName, String phone) {
        this.employeeId = employeeId;
        this.firstName = firstName;
        this.lastName = lastName;
        this.phone = phone;
        return null;
    }

    public void ejbPostCreate(String employeeId, String firstName,
        String lastName, String phone) {
    }

    /**
     * @ejb:interface-method
     * @ejb:persistent-field
     * @ejb:pk-field
     */
    public String getEmployeeId() {
        return employeeId;
    }

    /**
     * @ejb:interface-method
     */
    public void setEmployeeId(String employeeId) {
        this.employeeId = employeeId;
    }
```

```
/**
 * @ejb:interface-method
 * @ejb:persistent-field
 */
public String getFirstName() {
    return firstName;
}

/**
 * @ejb:interface-method
 */
public void setFirstName(String firstName) {
    this.firstName = firstName;
}

/**
 * @ejb:interface-method
 * @ejb:persistent-field
 */
public String getLastName() {
    return lastName;
}

/**
 * @ejb:interface-method
 */
public void setLastName(String lastName) {
    this.lastName = lastName;
}

/**
 * @ejb:interface-method
 * @ejb:persistent-field
 */
public String getPhone() {
    return phone;
}

/**
 * @ejb:interface-method
 */
public void setPhone(String phone) {
    this.phone = phone;
}

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

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

public void ejbActivate() {  }

public void ejbPassivate() {  }

/**
 * @jrun:jdbc-mapping name="load" source="samples"
 *                 action="SELECT employee_id, first_name,
                           last_name, phone FROM employees WHERE
                           employee_id=?"
 * @jrun:jdbc-mapping-param name="employeeId" type="VARCHAR"
 * @jrun:jdbc-mapping-field employeeId
 * @jrun:jdbc-mapping-field firstName
 * @jrun:jdbc-mapping-field lastName
 * @jrun:jdbc-mapping-field phone
 */
public void ejbLoad() {  }

/**
 * @jrun:jdbc-mapping name="store" source="samples"
 *                 action="UPDATE employees SET first_name=?,
                          last_name=?, phone=?WHERE employee_id=?"
 * @jrun:jdbc-mapping-param name="firstName" type="VARCHAR"
 * @jrun:jdbc-mapping-param name="lastName" type="VARCHAR"
 * @jrun:jdbc-mapping-param name="phone" type="VARCHAR"
 * @jrun:jdbc-mapping-param name="employeeId" type="VARCHAR"
 */
public void ejbStore() {  }

/**
 * @jrun:jdbc-mapping name="remove"
 *                 source="samples"
 *                 action="DELETE FROM employees WHERE
                           employee_id=?"
 * @jrun:jdbc-mapping-param name="employeeId" type="VARCHAR"
 */
public void ejbRemove() throws RemoveException {
}

/**
 * @jrun:jdbc-mapping name="findByPrimaryKey" source="samples"
 *                 action="SELECT employee_id FROM employees WHERE
                           employee_id=?"
 * @jrun:jdbc-mapping-param name="employeeId" type="VARCHAR"
 * @jrun:jdbc-mapping-field employeeId
 */
public void findByPrimaryKey(String pk) {
}
```

```
    /**
     * @jrun:jdbc-mapping name="findAll" source="samples"
     *                action="SELECT employee_id FROM employees"
     * @jrun:jdbc-mapping-field employeeId
     */
    public Collection findAll() {
        return null;
    }

    /**
     * @jrun:jdbc-mapping name="findByLastName" source="samples"
     *                action="SELECT employee_id FROM employees WHERE
                              last_name=?1"
     * @jrun:jdbc-mapping-param name="lastName" type="VARCHAR"
     * @jrun:jdbc-mapping-field employeeId
     */
    public Collection findByLastName(String lastName) {
    this.lastName = lastName;
        return null;
    }
}
```

メッセージ駆動型 bean

MDB (Message-driven bean：メッセージ駆動型 bean) は、JMS メッセージを使用するためにデザインされる EJB です。これは、直接呼び出すことがないため、リモート、ローカル、リモートホーム、またはローカルホームのインターフェイスがありません。ただし、ejb-jar.xml および jrun-ejb-jar.xml ファイルに MDB を定義する必要があります。また、デプロイメントディスクリプタで指定したキューおよびトピックが存在することを確認する必要もあります。

MDB 実装

MDB 実装クラスは、MessageDrivenBean と MessageListener の両方を実装します。MessageListener インターフェイスには、onMessage というコールバックメソッドが含まれます。これは、メッセージが受信されるとコンテナから呼び出されます。

次のコードは、MDB 実装を示しています。

```
import javax.ejb.*;
import javax.jms.MessageListener;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.JMSException;

public class SimpleMDBBean implements MessageDrivenBean,
        MessageListener {
    MessageDrivenContext ctx;

    /* NO-ARGS CONSTRUCTOR */
    public SimpleMDBBean() {
    System.out.println("In MDBTest:constructor");
    }

    /* onMessage callback method; part of MessageListener interface */
    public void onMessage(Message message) {
    try {
    TextMessage tMessage = (TextMessage)message;
      //実際の環境では、ここで do more を実行します。
     System.out.println("In MDBTest.onMessage(), message = " +
         tMessage.getText() );
    }
    catch(JMSException e) {
      System.out.println("JMS Exception:" + e.getMessage());
    }
    } // onMessage を終了
```

```
    /* Required EJB callback methods */
    public void setMessageDrivenContext(MessageDrivenContext ctx)
                                        throws EJBException {
    System.out.println("In MDBTest.setMessageDrivenContext()");
    this.ctx = ctx;
    }
   public void ejbCreate() {  }
   public void ejbRemove() {  }
}
```

EJB デプロイメントディスクリプタ

ejb-jar.xml ファイルでは、MDB で使用するキューまたはトピックを定義します。また、EJB コンテナで使用するその他情報も指定します。

次のサンプルは、MDB のデプロイメントディスクリプタの基本的要素を示しています。

```
...
  <message-driven>
      <ejb-name>RKN Test Message-Driven Bean</ejb-name>
      <ejb-class>SimpleMDBBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
      <security-identity>
        <description></description>
        <run-as>
          <role-name>everyone</role-name>
        </run-as>
      </security-identity>
      <resource-ref>
        <res-ref-name>MDBQCF</res-ref-name>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Unshareable</res-sharing-scope>
      </resource-ref>
      <resource-env-ref>
        <resource-env-ref-name>jms/queue/queue1</
        resource-env-ref-name>
       <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
      </resource-env-ref>
      <message-driven-destination>
        <destination-type>javax.jms.Queue</destination-type>
        <subscription-durability>Durable</subscription-durability>
      </message-driven-destination>
  </message-driven>
...
```

JRun EJB デプロイメントディスクリプタ

jrun-ejb-jar.xml ファイルには、MDB の JNDI キューの接続ファクトリ情報など、JRun 特有のデプロイ設定が含まれます。

次のサンプルは、MDB の JRun EJB デプロイメントディスクリプタを示しています。

```
...
<message-driven>
      <ejb-name>RKN Test Message-Driven Bean</ejb-name>
      <jndi-name>RKN Test Message-Driven Bean</jndi-name>
      <resource-ref>
        <res-ref-name>MDBQCF</res-ref-name>
        <jndi-name>jms/QueueConnectionFactory</jndi-name>
        <user>admin</user>
        <password>admin</password>
      </resource-ref>
      <resource-env-ref>
        <resource-env-ref-name>jms/queue/queue1</
        resource-env-ref-name>
        <jndi-name>jms/queue/queue1</jndi-name>
      </resource-env-ref>
      <message-driven-subscription>
        <client-id>MDBSubscriber</client-id>
      </message-driven-subscription>
    </message-driven>
...
```

その他のファイルと設定

JRun JMS および MDB を使用するには、JMC から JMS デスティネーション (キューおよびトピック) を指定する必要があります。これらの設定は、SERVER-INF/jrun-resources.xml ファイルに保存され、テキストエディタで自由に変更できます。

JMS デスティネーションの作成の詳細については、JMC オンラインヘルプを参照してください。JMS のその他の設定については、『JRun 管理者ガイド』を参照してください。

サンプル JMS センダー

次のコードは、サンプルの MDB 実装への簡単なサーブレットセンダーを示しています。

```
...
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;

public class MySender extends HttpServlet {
  // キュー属性を設定します。
  // ユーザー ID とパスワード
  private String _user = new String("admin");
  private String _password = new String("admin");
  // JMC で定義したキュー名と一致させます。
  // この情報は、SERVER-INF/jrun-resources.xml ファイルでも表示できます。
```

```
  private String _queue = new String("queue1");
  // これは、リモートホスト名と JNDI ポート番号を連結したものです。
  private String _providerurl = new String("localhost:2918");
  private String _ctxtFactory = new String
        ("jrun.naming.JRunContextFactory");
  // JMS で使用するオブジェクトを設定します。
  private Context _context = null;
  private QueueConnection _connection = null;
  private QueueSession _session = null;
  private TextMessage _message = null;
  private QueueSender _sender = null;

public void init(ServletConfig config) throws ServletException {
  super.init(config);

  // JNDI で設定を行います。
  try {
  Properties p = new Properties();
  p.put(Context.SECURITY_PRINCIPAL, _user);
  p.put(Context.SECURITY_CREDENTIALS, _password);
  p.put(Context.PROVIDER_URL, _providerurl );
  p.put( Context.INITIAL_CONTEXT_FACTORY, _ctxtFactory);

    // 同一サーバーのコンテキストにアクセスする場合は、空のコンストラクタ
    // になることがあるので、EJB では Web アプリケーションの認証情報が
    // 使用されます。
    _context = new InitialContext(p);

    final QueueConnectionFactory factory =
      (QueueConnectionFactory)_context.lookup
        ("jms/QueueConnectionFactory");
    // ファクトリを使用して QueueConnection (anonymous) を作成します。
    _connection = factory.createQueueConnection();
    // QueueConnection を使用して QueueSession を作成します。
    _session = _connection.createQueueSession(false,
      Session.AUTO_ACKNOWLEDGE);
    // QueueSession を使用して QueueSender を作成します。
    _sender = _session.createSender((Queue)_context.lookup
      ("java:jms/queue/queue1"));
    // TextMessage オブジェクトを作成します。
    _message = _session.createTextMessage();
  }
  catch(NamingException e) {
    System.out.println("Naming Exception:" + e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
}
```

```
// doGet は、送信するメッセージ（非表示）を受け取るフォームを表示します。
public void doGet(HttpServletRequest req,
  HttpServletResponse res) throws IOException, ServletException {
...
// doPost は、フォームを読み込んでメッセージを送信します。
public void doPost(HttpServletRequest req,
  HttpServletResponse res) throws IOException, ServletException {
  // フォームを読み込みます。
  // メッセージはすべて、ここで送信されます。
  // 設定の内容については、init メソッドを参照してください。
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();

  int priority = Message.DEFAULT_PRIORITY;
  int delivery = Message.DEFAULT_DELIVERY_MODE;
  String text = new String("Blank Message");

  String[] attrArray = req.getParameterValues("thisMessage");
  // 呼び出しフォームには、thisMessage 用の値が 1 つしかないと想定します。
  if(attrArray != null) {
    text = attrArray[0];
  }
   try {
        System.out.println("just before setText");
        _message.setText(text);
        // キューに送信します。
        _sender.send(_message);
    } // try を終了
    catch(JMSException e) {
      System.out.println("JMS Exception:" + e.getMessage());
    }
...
public void destroy() {
  // これらは if != null のチェックでラップしたほうがよいでしょう。
  try {
    _sender.close();
    _session.close();
    _connection.close();
    _context.close();
  }
  catch(NamingException e) {
    System.out.println("Naming exception in destroy: " +
      e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
...
```

トランザクション管理

コンピューティングとネットワークの機能が拡張されるにつれて、分散 2 フェーズコミットのようなツールを含めてトランザクション管理プログラミングはより複雑になってきました。しかし、EJB には、トランザクション管理を簡素化するアーキテクチャが用意されています。bean のタイプに応じて、次のトランザクション管理モデルを使用します。

- **BMT (Bean-maneged transactions：bean 管理トランザクション)** bean は、Java Transaction API (JTA) のメソッドを使用して、bean 独自のトランザクションを管理します。このオプションは、セッション beans と MDB でのみ使用できます。
- **CMT (Container-managed transactions：コンテナ管理トランザクション)** EJB コンテナは、各 bean の ejb-jar.xml ファイルに作成された仕様に基づいて、トランザクションを管理します。トランザクションは EJB コンテナによって開始され、bean は、`setRollbackOnly` と `getRollbackOnly` メソッドを使用してトランザクションのコンテキストと対話します。

BMT、CMT、JTA、およびトランザクションプログラミングの詳細については、EJB 2.0 を解説している業界紙を参照してください。このマニュアルの序章にはこれらの本のリストが記載されています。

EJB のログの作成

JRun EJB からログメッセージを書き出す場合、次のオプションがあります。

- **Nonportable** JRun 特有のメソッドとサービスを使用します。これらのメソッドは、JRun サーバーのイベントログに書き出します。
- **Portable** `System.out print` メソッドを使用します。これらのメソッドは、(JRun 起動時に使用する場合) コンソールに書き込出します。

JRun 特有のメソッド

次の JRun 特有のテクニックは、JRun サーバーのイベントログに書き出します。

- `logInfo` メソッド
- logger サービス

これらのテクニックは、JRun のイベントログにシンプルなインターフェイスを提供します。ただし、他の J2EE アプリケーションサーバーには移植できません。

logInfo メソッドの使用

EJB は、コンテナから logger インスタンスを取得して、次のコード例に示すように、ロガーの `logInfo` メソッドを使用できます。

セッション bean

```
import jrun.ejb.SessionInstanceContext;
...
SessionInstanceContext sessionInstanceContext =
    (SessionInstanceContext) sessionContext;
sessionInstanceContext.getEJBContainer().getLogger().
     logInfo("hi");
```

エンティティ bean

```
import jrun.ejb.EntityInstanceContext;
...
EntityInstanceContext entityInstanceContext =
    (EntityInstanceContext) entityContext;

  entityInstanceContext.getEJBContainer().getLogger().
     logInfo("hi");
```

メッセージ駆動型 bean

```
import jrun.ejb.MessageDrivenInstanceContext;
...
MessageDrivenInstanceContext mdbInstanceContext =
    (MessageDrivenInstanceContext) messageDrivenContext;
mdbInstanceContext.getEJBContainer().getLogger().logInfo("hi");
```

logger サービスの使用

次のコード例に示すように、JNDI を使用して LoggerService を検索できます。

```
...
  try {
    Properties p = new Properties();
    p.put(Context.SECURITY_PRINCIPAL, "userid");
    p.put(Context.SECURITY_CREDENTIALS, "password");
    p.put(Context.INITIAL_CONTEXT_FACTORY,
        "jrun.naming.JRunContextFactory");
    p.put(Context.PROVIDER_URL, "localhost:2918");
    InitialContext ctx = new InitialContext(p);
    LoggerService ls = (LoggerService) ctx.lookup("jrun:service/
        LoggerService");
    ls.logInfo("Message logged using LoggerService");
...
```

移植可能なメソッドの使用

EJB のメッセージを書き出す際のシンプルで移植可能なオプションは、
System.out.println のような System.out print メソッドの 1 つを使用する方法です。

このテクニックは移植可能ですが、特にすべてのログレベルが有効になっている場合は、メッセージのコンソール出力が、スレッドロガーの出力と混ざってしまう可能性があります。

パート V
JMS プログラミング

パート V では、JRun による Java Message Service (JMS) プログラミングについて説明します。次の章で構成されています。

第 15 章
JMS の概要

この章では、Java Message Service (JMS) の基本的な概念について説明します。

目 次

JMS の導入

Java Message Service (JMS) 仕様では、メッセージを作成および送受信する一般的な方法を Java 開発者に提供する社内メッセージングミドルウェアシステムを構築するための API を定義します。JMS は、移植可能でなメッセージベースのビジネスアプリケーションをサポートします。JMS は MDB の基盤でもあります。

JRun は、JMS の完全サポートや、SonicMQ など外部の JMS プロバイダのサポートとビルトインの JMS プロバイダをシームレスに統合します。

この章では、JRun で提供される JMS サービスの使用方法について説明します。JMS をよく理解している方を対象としています。JMS の詳細については、http://java.sun.com で入手できる JMS 仕様を参照してください。

メッセージのタイプ

JRun JMS 実装では、Sun JMS バージョン 1.02b 仕様の ポイントツーポイント (キューベース) とパブリッシュ/ サブスクライブ (トピックベース) の同期および非同期メッセージングがサポートされています。メッセージにはパーシスタンスを指定できるので、サーバーのシャットダウン時にメッセージは失われません。

トピックベースのメッセージングでは、永続サブスクライブが使用可能です。これにより、サブスクライバがアクティブでないときに生成されるメッセージを含め、生成されるすべてのメッセージがクライアントにより確実に受信されるようになります。

用語集

この章では、プロデューサという用語はメッセージを送信するクライアントを表します。また、コンシューマという用語はメッセージを受信するクライアントを表します。ただし、コンシューマ、プロデューサは両方ともクライアントと呼びます。

ポイントツーポイントメッセージングでは、プロデューサはセンダー、コンシューマはレシーバーと呼びます。パブリッシュ / サブスクライブのメッセージングでは、プロデューサはパブリッシャ、コンシューマはサブスクライバと呼びます。

メッセージを生成または処理するために、クライアントは、サーバーへの接続を確立し、Connection を呼び出して、Session を作成します。クライアントはサーバーと対話し、以前に確立された `Session` オブジェクトを使用してメッセージを生成または処理します。ポイントツーポイントメッセージングとパブリッシュ / サブスクライブでは、`Connection` オブジェクトと `Session` オブジェクトのカスタマイズされた子オブジェクトが使用されます。

メッセージコンポーネント

メッセージは次の部分からなります。

- **ヘッダー**　メッセージの識別とルーティングのために、クライアントとサーバーの両方で使用される情報。
- **プロパティ**　追加のヘッダープロパティ。プロパティにはアプリケーション特有のプロパティ、標準プロパティ、サーバー特有のプロパティがあります。
- **本文**　メッセージの本文。メッセージ本文には、テキスト、オブジェクト、バイトなど、定義済みタイプのいずれかを使用できます。

メッセージヘッダーフィールド

JRun では、JMS メッセージヘッダーフィールドがサポートされていて、これらのフィールドは JMS メッセージ受信者に送信されます。JRun でサポートされている JMS メッセージヘッダーフィールドは、次の表のとおりです。

フィールド	内容	設定者
JMSDestination	メッセージの送信先を表す Destination オブジェクトが含まれています。	JRun
JMSDeliveryMode	配送モードが含まれています。有効な値は DeliveryMode.PERSISTENT と DeliveryMode.NON_PERSISTENT です。	JRun
JMSMessageID	固有のメッセージ ID が含まれています。	JRun
JMSTimestamp	メッセージが JRun に送信された時刻が含まれています。	JRun
JMSCorrelationID	レスポンスと、関連するリクエストをリンクするアプリケーション特有の文字列が含まれています。	JRun
JMSReplyTo	応答の送信先となる Destination オブジェクトが含まれています。応答は必須ではありませんが、Destination オブジェクトがこのフィールドに含まれているのは応答が期待されていることを示しています。	JRun
JMSRedelivered	メッセージを再配送するかどうかを表す boolean が含まれています。コンシューマにより、JMSRedelivered が true に設定されたメッセージが受信された場合、このメッセージは以前配送されたが、コンシューマが受信を認めていなかったと考えられます。	JRun
JMSType	メッセージタイプを表す String が含まれています。	クライアント
JMSExpiration	メッセージの有効期限を指定する long が含まれています。JRun では、クライアントにより指定された Time-to-Live の値に送信時間の GMT を加えて、この時刻を設定します。	JRun
JMSPriority	メッセージの優先順位が含まれます。優先順位は 0 (最下位) ～ 9 (最上位) の間で指定します。	JRun
JMSXGroupID	メッセージのグループに使用する ID を指定します。	クライアント

フィールド	内容	設定者
JMSXGroupSeq	メッセージのグループに使用するシーケンスを指定します。	クライアント
JMSXRcvTimestamp	メッセージの配送時刻です。	JRun

ヘッダーフィールドにアクセスするには、Message インターフェイスのメソッドを使用します。Message インターフェイスは、TextInterface や MapInterface などのコンテンツ特有のメッセージインターフェイスにより拡張されます。

メッセージプロパティ

JRun は JMS 仕様で定義されているオプションの JMSX 接頭辞付きのメッセージプロパティをサポートしません。しかし、Message オブジェクトメソッドを使用してプロパティを取得および設定できます。たとえば、次のコードの一部分を使用して、メッセージを送信する前にプロパティを設定できます。

```
...
try {
  // ユーザー ID のプロパティを設定します。thisUser String 変数を想定します。
  if(message != null) {
    message.setStringProperty("UserID", thisUser);
    message.setText(text);
    // キューに送信します。メッセージは 5 分間存続します。
    sender.send(_message, delivery, priority, 5 * 60 * 1000);
  }else {
    // 1 つのサーブレットまたは JSP ページでの使用方法を想定します。
    out.println("<H1> メッセージは null でした </H1>");
  }
}
...
```

次のコードの一部分を使用して、メッセージを受信したときにプロパティを取り出すことができます。

```
final TextMessage message = (TextMessage)(_receiver.receiveNoWait());
// すべてのプロパティを取得します。
Enumeration e = message.getPropertyNames();
if(!e.hasMoreElements()) {
  // 1 つのサーブレットまたは JSP ページでの使用方法を想定します。
  out.println("<h1> プロパティがありません </H1>");
}
while(e.hasMoreElements()) {
  String prop = (String)e.nextElement();
  out.print("<p> " + prop);
  // すべてのプロパティが Strings であると想定します。
  out.println(":" + message.getStringProperty(prop));
}
```

メッセージ本文のタイプ

JMS 1.0.2b 仕様には、メッセージ本文の形式について記載されています。これらの形式は、`Message` を拡張したインターフェイスにより定義されます。

次の表で、JMS メッセージ本文インターフェイスの概要を説明します。

インターフェイス	説明	コメント
StreamMessage	Java プリミティブ値のストリームが含まれています。	このタイプは値の挿入と読み込みが順次に行われます。
MapMessage	名前 / 値のペアのセットが含まれます。名前は String オブジェクト、値は Java プリミティブタイプです。	これらには、列挙によって連続的にアクセスするか、名前によってランダムにアクセスします。
TextMessage	String オブジェクトを 1 つ含んでいます。	TextMessage はテキストメッセージまたは XML 形式のデータを持つメッセージで使用します。
ObjectMessage	直列化可能な Java オブジェクトが含まれています。	いずれかの JDK 1.2 Collection クラスを使用できます。
BytesMessage	未解釈のバイト ストリームが含まれています。	通常、このタイプの本文は使用しません。

メッセージ本文インターフェイスの使用方法を含む JMS のプログラミングについては、417 ページの第 16 章「JMS プログラミングテクニック」で説明します。

第 16 章
JMS プログラミングテクニック

この章では、JMS クライアントやサーバーをプログラムするときに使用するテクニックを説明します。キュー (ポイントツーポイント) とトピック (パブリッシュ / サブスクライブ) も説明します。

目 次

ポイントツーポイントのプログラミング

ポイントツーポイントメッセージングは、キューベースのメカニズムです。メッセージは特定のキューに送られます。センダーはメッセージをキューに追加し、レシーバーはキューからメッセージを抽出します。

ポイントツーポイントメッセージングソリューションを実装する場合は、次のコードを作成します。

- Sender
- Receiver

ポイントツーポイントセンダーのコーディング

センダーは次のオブジェクトを使用してメッセージをキューに追加します。

- `javax.naming.Context`
- `javax.jms.QueueConnectionFactory`
- `javax.jms.QueueConnection`
- `javax.jms.QueueSession`
- `javax.jms.QueueSender`
- 415 ページの「メッセージ本文のタイプ」で指定されている `javax.jms.Message` オブジェクトの子オブジェクトの 1 つ。

センダークラスは、`javax.jms` クラスのいずれも拡張しません。このクラスは明示的に他のクラスを拡張したり、暗黙的に `java.lang.Object` を拡張したりできます。

次のコードを実行するには、queue1 と名付けたキューを JMC 内に定義する必要があります。

センダーのコードを作成するには

1 次のパッケージをインポートします。

```
import java.util.*;
import javax.jms.*;
import javax.naming.*;
```

2 次のように、JMS で使用されるオブジェクトの変数を宣言します。

```
private Context _context = null;
private QueueConnection _connection = null;
private QueueSession _session = null;
private TextMessage _message = null;
private QueueSender _sender = null;
```

3 アプリケーション特有のメソッドを使用して、ユーザーとキューの情報を取得します。わかりやすくするために、この例では各値をハードコーディングしています。

```
// ユーザー ID とパスワード
private String _user = new String("admin");
private String _passwprd = new String("admin");
// キューの名前を JMC 全体で一致させます。
// この情報は、SERVER-INF/jrun-resources.xml ファイルでも表示できます。

private String _queue = new String("queue1");
// これは、ホスト名と JNDI ポート番号を連結したものです。
private String _providerurl = new String("localhost:2918");
private String _ctxtFactory = new String
     ("jrun.naming.JRunContextFactory");
```

4 メッセージを送信する前に、JMS 変数を作成して、値を挿入します。

```
// JNDI を使用して設定します。
try {
  Properties p = new Properties();
  p.put(Context.SECURITY_PRINCIPAL, _user);
  p.put(Context.SECURITY_CREDENTIALS, _password);
  p.put(Context.PROVIDER_URL, _providerurl );
  p.put( Context.INITIAL_CONTEXT_FACTORY, ctxtFactory);

  // 同じサーバー上のコンテキストにアクセスする場合は、
  // これは空のコンストラクタでも構いません。
  _context = new InitialContext(p);

    // JNDI から QueueConnectionFactory を取得します。
    log("just before QueueConnectionFactory context lookup");

  final QueueConnectionFactory factory =
    (QueueConnectionFactory)_context.lookup
      ("jms/QueueConnectionFactory");
  // ファクトリを使用して、QueueConnection (anonymous) を作成します。
  _connection = factory.createQueueConnection();
  // QueueConnection を使用して QueueSession を作成します。
  _session = _connection.createQueueSession(false,
  Session.AUTO_ACKNOWLEDGE);
  // QueueSession を使用して QueueSender を作成します。
  // queue1 を探します。
  log("just before createSender");
  _sender = _session.createSender((Queue)_context.lookup
    ("jms/queue/queue1"));
  // TextMessage オブジェクトを作成します。
  _message = _session.createTextMessage();
  }
  catch(NamingException e) {
    System.out.println("Naming Exception:" + e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
```

5 メッセージを送信します。このサーブレット例では、フォームフィールドからメッセージを取得します。

```
int priority = Message.DEFAULT_PRIORITY;
int delivery = Message.DEFAULT_DELIVERY_MODE;

String text = new String("Blank Message");

String[] attrArray = req.getParameterValues("thisMessage");
// 呼び出しフォームには、thisMessage 用の値が 1 つしかないと想定します。
if(attrArray != null) {
  text = attrArray[0];
}
try {
  _message.setText(text);
  // キューに送信します。メッセージは 5 分間存続します。
  _sender.send(_message, delivery, priority, 5 * 60 * 1000);
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

6 完了したら、JMS オブジェクトを閉じます。

```
// これらは if != null のチェックでラップしたほうがよいでしょう。
try {
  _sender.close();
  _session.close();
  _connection.close();
  _context.close();
  }
  catch(NamingException e) {
    System.out.println("Naming exception in destroy: " +
      e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
```

ポイントツーポイントレシーバーのコーディング

レシーバーは、次のオブジェクトを使用してキューからメッセージを取り出します。

- `javax.naming.Context`
- `javax.jms.QueueConnectionFactory`
- `javax.jms.QueueConnection`
- `javax.jms.QueueSession`
- `javax.jms.QueueReceiver`
- 415 ページの「メッセージ本文のタイプ」で指定されている `javax.jms.Message` オブジェクトの子オブジェクトの 1 つ。

レシーバーが非同期メッセージを使用する場合は、`MessageListener` インターフェイスと `onMessage` メソッドを実装する必要があります。レシーバークラスは、`javax.jms` クラスのいずれも拡張しませんが、このクラスは明示的に他のクラスを拡張したり、暗黙的に `java.lang.Object` を拡張したりできます。

メッセージ駆動型 bean スタイルメッセージレシーバーについては、401 ページの「メッセージ駆動型 bean」を参照してください。

レシーバーのコードを作成するには

1 次のパッケージをインポートします。

```
import java.util.*;
import javax.jms.*;
import javax.naming.*;
```

2 次のように、JMS で使用されるオブジェクトの変数を宣言します。

```
private Context _context = null;
private QueueConnection _connection = null;
private QueueSession _session = null;
private TextMessage _message = null;
private QueueSender _sender = null;
private QueueReceiver _receiver = null;
```

3 メッセージを受信する前に、JMS 変数を作成して、値を挿入します。

```
try {
  String providerurl="localhost:2918";
  Properties env = new Properties();
  env.put(Context.PROVIDER_URL, providerurl );
  env.put( Context.INITIAL_CONTEXT_FACTORY,
     "jrun.naming.JRunContextFactory" );
  _context = new InitialContext(env);

  // JNDI から QueueConnectionFactory を取得します。
  final QueueConnectionFactory factory =
    (QueueConnectionFactory)_context.lookup
      ("java:comp/env/jms/QueueConnectionFactory");
  // ファクトリを使用して QueueConnection を作成します。
  _connection = factory.createQueueConnection();
  // QueueConnection を使用して QueueSession を作成します。
  _session = _connection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
  // QueueSession を使用して QueueReceiver を作成します。
  // queue1 を探します。
  _receiver = _session.createReceiver((Queue)_context.lookup
      ("jms/queue/queue1"));
  // TextMessage オブジェクトを作成します。
  _message = _session.createTextMessage();
}
catch(NamingException e) {
  System.out.println("Naming Exception:" + e.getMessage());
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

4 キューからメッセージを取り出す onMessage メソッドをコーディングします。

```
...
public void onMessage(Message message) {
  try {
    TextMessage textMessage = (TextMessage) message;
    String text = textMessage.getText();
    System.out.println(text);
  } catch(JMSException jmse){ jmse.printStackTrace(); }
}
...
```

パブリッシュ / サブスクライブのプログラミング

パブリッシュ / サブスクライブはブロードキャストメカニズムの 1 つで、メッセージはトピックにパブリッシュされ、トピックのサブスクライバに自動的に配信されます。トピックは階層的に構成できます。こうすると最上位レベルのサブスクライバはすべてのメッセージを受信しますが、サブトピックのサブスクライバはサブトピックのメッセージだけを受信します。

パブリッシャのコーディング

パブリッシャは、次のオブジェクトを使用してメッセージをトピックに追加します。

- javax.naming.Context
- javax.jms.TopicConnectionFactory
- javax.jms.TopicConnection
- javax.jms.TopicSession
- javax.jms.TopicPublisher
- 415 ページの「メッセージ本文のタイプ」で指定されている javax.jms.Message オブジェクトの子オブジェクトの 1 つ。

パブリッシャクラスは、javax.jms クラスのいずれも拡張しません。このクラスは明示的に他のクラスを拡張したり、暗黙的に java.lang.Object を拡張したりできます。

次のコードを実行するには、JMC を使用して topic1 と名付けたトピックを定義する必要があります。

パブリッシャのコードを作成するには

1 次のパッケージをインポートします。

```
import java.util.*;
import javax.jms.*;
import javax.naming.*;
```

2 次のように、JMS で使用されるオブジェクトの変数を宣言します。

```
private Context _context = null;
private TopicConnection _connection = null;
private TopicSession _session = null;
private TextMessage _message = null;
private TopicPublisher _publisher = null;
```

3 アプリケーション特有のメソッドを使用して、ユーザーとトピックの情報を取得します。わかりやすくするために、この例では各値をハードコーディングしています。

```
// ユーザー ID とパスワード
private String _user = new String("admin");
private String _password = new String("admin");
// トピックの名前を JMC 全体で一致させます。
// この情報は、SERVER-INF/jrun-resources.xml ファイルでも表示できます。
private String _topic = new String("topic1");
// これは、リモートホスト名と JNDI ポート番号を連結したものです。
private String _providerurl = new String("localhost:2918");
private String _ctxtFactory = new String
     ("jrun.naming.JRunContextFactory");
```

4 メッセージを送信する前に、JMS 変数を作成して、値を挿入します。

```
// JNDI で設定を行います。
try {
  Properties p = new Properties();
  p.put(Context.SECURITY_PRINCIPAL, _user);
  p.put(Context.SECURITY_CREDENTIALS, _password);
  p.put(Context.PROVIDER_URL, _providerurl );
  p.put( Context.INITIAL_CONTEXT_FACTORY, ctxtFactory);

  // 同じサーバー上のコンテキストにアクセスする場合は、
  // これは空のコンストラクタでも構いません。
  _context = new InitialContext(p);

// Get TopicConnectionFactory from JNDI.
  final TopicConnectionFactory factory =
    (TopicConnectionFactory)_context.lookup
      ("jms/TopicConnectionFactory");
  // ファクトリを使用して TopicConnection を作成します。
  _connection = factory.createTopicConnection();
  // TopicConnection を使用して TopicSession を作成します。
  _session = _connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
  // TopicSession を使用して TopicSender を作成します。
  // topic1 を探します。
  _publisher = _session.createPublisher((Topic)_context.lookup
    ("java:jms/topic/topic1"));
  // TextMessage オブジェクトを作成します。
  _message = _session.createTextMessage();
  }
  catch(NamingException e) {
    System.out.println("Naming Exception:" + e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
```

5 メッセージをパブリッシュします。このサーブレット例では、フォームフィールドからメッセージを取得します。

```
int priority = Message.DEFAULT_PRIORITY;
int delivery = Message.DEFAULT_DELIVERY_MODE;
String text = new String("Blank Message");

String[] attrArray = req.getParameterValues("thisMessage");
// 呼び出しフォームには、thisMessage 用の値が 1 つしかないと想定します。
if(attrArray != null) {
   text = attrArray[0];
}
try {
  System.out.println("setText の直前 ");
  _message.setText(text);
  // トピックをパブリッシュします。
  System.out.println(" パブリッシュの直前 " );
  // トピックをパブリッシュします。メッセージは 5 分間存続します。
 _publisher.publish(_message, delivery, priority, 5 * 60 * 1000);
  System.out.println(" パブリッシュの直後 ");
  } // try を終了
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
}
```

6 完了したら、JMS オブジェクトを閉じます。

```
// これらは if != null のチェックでラップしたほうがよいでしょう。
try {
  _publisher.close();
  _session.close();
  _connection.close();
  _context.close();
  }
  catch(NamingException e) {
    System.out.println("Naming exception in destroy: " +
      e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
```

サブスクライバのコーディング

JMS では、関連するトピックに対してトピックがパブリッシュされると、これらがサブスクライバに渡されます。次のオブジェクトを使用して、サブスクライバをコーディングします。

- javax.naming.Context
- javax.jms.TopicConnectionFactory
- javax.jms.TopicConnection
- javax.jms.TopicSession
- javax.jms.TopicSubscriber
- 415 ページの「メッセージ本文のタイプ」で指定されている javax.jms.Message オブジェクトの子オブジェクトの 1 つ。

サブスクライバが非同期メッセージを使用する場合は、MessageListener インターフェイスと onMessage メソッドを実装する必要があります。サブスクライバクラスがサブスクライブしているトピックにメッセージがパブリッシュされると、JRun では onMessage メソッドが呼び出されます。

サブスクライバクラスは、javax.jms クラスのいずれも拡張しませんが、明示的に他のクラスを拡張したり、暗黙的に java.lang.Object を拡張したりできます。

パブリッシャのコードを作成するには

1 次のパッケージをインポートします。

```
import java.util.*;
import javax.jms.*;
import javax.naming.*;
```

2 MessageListener インターフェイスを実装するクラス宣言を作成します。

```
public class MyTopicSubscriber implements MessageListener {
```

3 次のように、JMS で使用されるオブジェクトの変数を宣言します。

```
private Context jndi = null;
private TopicSession pubSession = null;
private TopicSession subSession = null;
private TopicPublisher publisher = null;
private static TopicConnection connection = null;
private String userName = null;
```

4 アプリケーション特有のメソッドを使用して、ユーザーとトピックの情報を取得します。わかりやすくするために、この例では各値をハードコーディングしています。

```
// トピックの名前を JMC 全体で一致させます。
// この情報は、SERVER-INF/jrun-resources.xml ファイルでも表示できます。
private String topicName = new String("topic1");
// これは、リモートホスト名と JNDI ポート番号を連結したものです。
private String providerurl = new String("localhost:2918");
private String factoryName = new String
     ("jrun.naming.JRunContextFactory");
Properties env = new Properties();
env.put(Context.PROVIDER_URL, providerurl );
env.put( Context.INITIAL_CONTEXT_FACTORY,
     "jrun.naming.JRunContextFactory" );
  jndi = new InitialContext(env);
```

```
TopicConnectionFactory conFactory =
  (TopicConnectionFactory)jndi.lookup(factoryName);
connection=conFactory.createTopicConnection();
TopicSession pubSession =
  connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
TopicSession subSession =
  connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);

// JMS トピックをルックアップします。
Topic thisTopic = (Topic)jndi.lookup(topicName);

// JMS メッセージリスナを設定します。
TopicSubscriber subscriber=subSession.createSubscriber(thisTopic);
subscriber.setMessageListener(this);

// このサンプルはパブリッシュでも使用します。
publisher=pubSession.createPublisher(thisTopic);

// アプリケーションを初期化します。
set(connection, pubSession, subSession, publisher, username);

// JMS 接続を開始し、メッセージが配送されるようにします。
connection.start();
}
catch(NamingException e) {
  System.out.println("Naming Exception:" + e.getMessage());
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

5 トピックからメッセージを取り出す onMessage メソッドをコーディングします。

```
...
public void onMessage(Message message) {
  try {
    TextMessage textMessage = (TextMessage) message;
    String text = textMessage.getText();
    System.out.println(text);
  } catch(JMSException jmse){ jmse.printStackTrace(); }
}
...
```

receive と receiveNoWait の使用

receive メソッドや receiveNoWait メソッドを使用することにより、onMessage コールバックメソッド内でメッセージを待つ代わりに、メッセージを直接取り出すことができます。これらのメソッドはキューとトピックに対して機能します。これらのメソッドは両方とも、Message タイプのオブジェクトを返します。このオブジェクトは、適切なメッセージタイプにキャストする必要があります (TextMessage など)。次のように、これらのメソッドを使用します。

- **receive**　メッセージを待機します。オプションでタイムアウト値を指定できます。
- **receiveNoWait**　メッセージをチェックして戻ります。

receive メソッドと receiveNoWait メソッドは MessageConsumer インターフェイスで定義されており、トピック (TopicSubscriber を通して) とキュー (QueueReceiver を通して) で利用できます。

これらのメソッドを使用するために、MessageListener インターフェイスの実装、setMessageListener メソッドのコーディング、または onMessage メソッドの実装は行いません。代わりに、次の例に示すように、接続開始後に、receive か receiveNoWait のいずれかをコーディングします。

```
...
topicConnection.start();
...
TextMessage mess = (TextMessage)sub.receiveNoWait();
...
```

このテクニックは、MessageListener インターフェイスを実装できないサーブレットや MDB 以外の EJB で特に役立ちます。

パート VI
Web サービスのプログラミング

パート VI では、JRun による Web サービスプログラミングについて説明します。次の章で構成されています。

第 17 章
JRun Web サービスの概要

この章では、JRun の Web サービスサポートの概要を説明します。

目 次

JRun Web サービスについて

JRun を使用すると、インターネットのどこからでも利用できるビジネス機能である Web サービスをパブリッシュし、使用できます。Web サービスは、XML と HTTP などの標準インターネットプロトコルを使用して、プラットフォームやロケーションに依存しないコンピューティングを提供します。以前には互換性のなかったアプリケーションを言語、プラットフォーム、またはオペレーティングシステムにかかわりなく Web 上で相互運用できるようにすることによって、Web サービスから新しいビジネスチャンスが生み出され、企業はビジネス上のリレーションシップの変化に対応できます。たとえば、Microsoft .NET コンポーネントは EJB (Enterprise JavaBeans) などの J2EE コンポーネントと双方向に通信することができます。

JRun を使用すれば、既存の Java コードを Web サービスとしてパブリッシュしたり、新しいコードを特別に Web サービスとして作成できます。また、リモート Web サービスが Microsoft .NET などの Java 以外のプラットフォーム上にあっても、そのサービス上のメソッドを呼び出すことができるオブジェクトベースまたはタグベースのクライアントを作成できます。

JRun Web サービスのパブリッシュは、Java ソースファイルの名前の変更、または Web アプリケーションに Java クラスおよび Web サービスのデプロイメントディスクリプタを含めることと同様に簡単です。パブリッシュされた Web サービスは、URL を介してクライアントが利用できます。サービスの呼び出しに必要な情報をクライアントに提供するために、WSDL (Web Services Description Language：Web サービス記述言語) ドキュメントをリクエスト時に生成できます。ユーザー独自の Web サービスクライアントの場合は、WSDL ドキュメントからローカル Web サービスプロキシを生成できます。

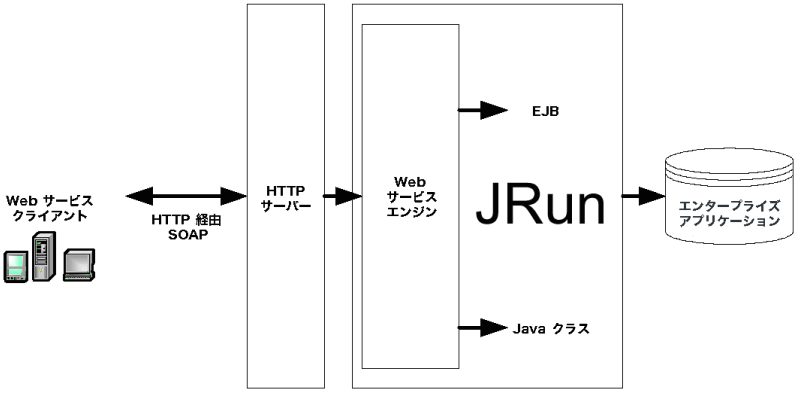
JRun Web サービスの実装は、Apache Software Foundation の SOAP (Simple Object Access Protocol：単一オブジェクトアクセスプロトコル) エンジンの第 3 世代である Apache Axis 上に構築します。

Web サービスのプラットフォーム

Web サービスのプラットフォームの 3 つの主要コンポーネントは次のとおりです。

- SOAP
- WSDL
- UDDI (Universal Description, Discovery, and Integration)

次の簡単な図は、JRun 内での Web サービスオペレーションを示しています。



SOAP による Web サービスの呼び出し

SOAP には、Web サービスリクエストおよびレスポンスの送信と受信のための標準 XML シンタックスが用意されています。

通常は HTTP を使用して SOAP メッセージを送信しますが、SMTP やその他のプロトコルを使用して送信することもできます。通常のシナリオでは、次のように動作します。

1 サービスクライアントは、インターネット上で HTTP を使用して、SOAP を使用している Web サービスの オペレーション (メソッド) を呼び出します。オペレーション呼び出しの指示は、HTTP リクエスト内の SOAP でエンコードされた XML ドキュメントとして送信されます。

2 Web サーバーが HTTP リクエストを受け取り、それを SOAP エンジンに渡します。

3 SOAP エンジンは、SOAP でエンコードされた XML ドキュメントを読み取り、該当するオペレーションをバックエンドシステム上に呼び出します。Web サービスのバックエンドでビジネスロジックを提供するオブジェクトは、Java クラス、EJB、.NET コンポーネント、または他のタイプのプログラムです。

4 SOAP エンジンは、オペレーションの結果を含んでいる HTTP レスポンスを、SOAP でエンコードされた XML ドキュメント内に生成します。

5 HTTP レスポンスはクライアントに返送されます。

SOAP リクエストとレスポンスのペアのサンプルを表示するには、473 ページの第 22 章「SOAP の監視」を参照してください。SOAP の詳細については、次の URL にある W3C の SOAP 1.1 ノートをご覧ください。

http://www.w3.org/TR/SOAP/

WSDL による Web サービスの記述

WSDL ドキュメントとは、Web サービスの目的、配置場所、およびアクセス方法を記述している XML ファイルのことです。

WSDL ドキュメントは、呼び出すことができるオペレーションとそれに関連するデータタイプを記述しています。

Axis では Web サービスから WSDL ドキュメントを生成でき、その WSDL ドキュメントを URL にパブリッシュしてクライアントに情報を提供できます。Axis WSDL2Java ツールを使用すると、クライアントサイドおよびサーバーサイドの Java コードを WSDL ドキュメントから生成することができます。また、Java2WSDL ツールを使用すると、Java コードから WSDL を生成できます。詳細については、457 ページの「WSDL ドキュメントの操作」を参照してください。

WSDL の詳細については、次の URL にある W3C の WSDL ノートをご覧ください。

http://www.w3.org/TR/wsdl

UDDI による Web サービスの検出

UDDI には、Web サービスクライアントが特定の機能を持つ Web サービスをダイナミックに見つける方法が用意されています。UDDI クエリはサービスプロバイダを見つけるときに使用します。UDDI レスポンスには、ビジネス連絡先情報、業務カテゴリ、および Web サービスを呼び出す方法に関する技術的な詳細情報が含まれています。UDDI の詳細については、次の URL をご覧ください。

http://www.uddi.org/about.html.

Axis SOAP エンジン

Axis は優れたアーキテクチャを持つ SOAP エンジンで、次の特徴があります。

- **高速性**。Axis は、DOM (Document Object Model：ドキュメントオブジェクトモデル) 解析を使用する SOAP の実装よりも処理速度を向上させるために、SAX (Simple API for XML) を使用しています。DOM 解析では、XML ドキュメント構造に基づく内部ツリーの作成を必要とする、ツリーベースのナビゲーションモデルを使用しています。SAX 解析では、XML ドキュメントの内部表現を必要としない、イベントベースのナビゲーションモデルを使用しています。SAX パーサーは XML データをスキャンし、ドキュメントの特定の部分が見つかったときにハンドラ関数を呼び出します。
- **拡張性**。Axis を拡張して、カスタムヘッダーの処理やシステム管理などの追加機能を提供します。
- **コンポーネント指向アーキテクチャ**。再利用可能なメッセージハンドラクラス (再利用が可能でハンドラチェーンに結合できるプロセッサ) を定義します。チェーンは、共通処理パターンを実装するときに使用できるメッセージハンドラの再利用可能なネットワークです。
- **トランスポートフレームワーク**。単純なアブストラクションを使用して、SMTP、FTP、メッセージ指向ミドルウェアなどのさまざまなプロトコルに対応する SOAP 用の トランスポート、センダー、およびリスナを設計します。エンジンの中核部分は、トランスポートから独立しています。
- **宣言文による設定**。Web アプリケーションの WEB-INF ディレクトリにある WSDD (Web Service Deployment Descriptor：Web サービスデプロイメントディスクリプタ) ファイル server-config.wsdd を編集することによって、Axis の特定の Web サービスまたはカスタム機能を設定します。

メッセージハンドラと Web サービス間のリレーションシップの定義

次の 2 つのセクションでは、Axis WSDD ファイルで標準 Axis メッセージハンドラを定義する方法と、そのメッセージハンドラを使用し、その WSDD ファイルで特定の Web サービスを定義する方法を説明します。

Axis メッセージハンドラの定義

Axis WSDD ファイルの次の行では、`RPCDispatcher` と呼ばれる標準 Axis リクエストハンドラを定義しています。このハンドラは `org.apache.axis.providers.java.RPCProvider` クラスを使用して、Java クラスバックエンドを使用する RPC (Remote Procedure Call：リモートプロシージャコール) Web サービスに対して行われたリクエストを処理します。

```
<handler name="RPCDispatcher"
type="java:org.apache.axis.providers.java.RPCProvider"/>
```

Web サービスの定義

Axis WSDD ファイルの次のセクションでは、RPC Web サービスを定義しています。サービス とはターゲットとなるチェーンのことで、リクエストハンドラ、プロバイダ、およびレスポンスハンドラを含むことができます。name="SampleLoanService" 属性は、Web サービスの名前を示します。provider="java:RPC" 属性は、Web サービスが RPCDispatcher リクエストハンドラを使用することを示します。最初の parameter の値は、どのクラスのパブリックメソッドが Web サービスに使用できるかを示します。2 番めの parameter の値は、Web サービスのバックエンドの実装が SampleJavaClass と呼ばれる Java クラスであることを示します。

```
<service name="SampleLoanService" provider="java:RPC">
   <parameter name="methodName" value="calculate"/>
   <parameter name="className" value="samples.Loan"/>
</service>
```

Axis の詳細については、<JRun のルートディレクトリ >/docs/html/thirdparty/axis ディレクトリの『Axis User Guide』を参照してください。

第 18 章
Web サービスのパブリッシュ

この章では、Web サービスのパブリッシュに関連する概念とタスクについて説明します。

目 次

Web サービスのバックエンド選択

次のいずれのエンティティも JWS (JRun Web Service：JRun Web サービス) としてパブリッシュできます。

- JWS ファイル
- Java クラスファイル
- ステートレスセッション bean

このセクションでは、Web サービスのバックエンドに関する一般情報について説明します。

JWS ファイル

JWS ファイルとは、最も基本的なタイプの Web サービスバックエンドです。サービスクラスのソースコードがあり、カスタムデータタイプのマッピングや、Axis がリクエストを処理する方法のカスタム設定をサービスが必要とない場合に適しています。JWS ファイルは .jws 拡張子を持つ Java ソースファイルです。これを Web サービスとしてダイナミックにデプロイできます。JSP と同様に、JWS ファイルも自動的にコンパイルされます。

ソースファイルを Web からアクセス可能な Web アプリケーションディレクトリ (WEB-INF または META-INF 以外) に配置し、ファイルの拡張子を .jws に変更することによって、Java クラスを Web サービスとしてダイナミックにデプロイできます。Axis はクラスを自動的にコンパイルし、SOAP 呼び出しを Java メソッド呼び出しに変換します。JWS は基本的な Web サービスには適していますが、Java と XML 間のカスタムデータタイプのマッピングは使用できません。また、ハンドラが Axis のリクエスト処理方法を制御するように設定することもできません。カスタムデータタイプのマッピングおよび Axis ハンドラの詳細については、<JRun のルートディレクトリ >/docs/html/thirdparty/axis ディレクトリの『Axis User Guide』を参照してください。

Java クラスファイル

サービスクラスのソースコードがない場合、またはサービスクラスが基本データタイプでないオペレーション (メソッド) パラメータを取る場合、Java クラスは Web サービスバックエンドに適しています。

Java クラスをデプロイするには、Web アプリケーションの server-config.wsdd ファイルに `java:RPC` のプロバイダ値を指定する RPC スタイルの `service` 要素を作成する必要があります。次に例を示します。

```
<service name="AxisSampleJavaClassService" provider="java:RPC">
  <parameter name="methodName" value="*"/>
  <parameter name="className" value="SampleJavaClass"/>
</service>
```

次の表で、PRC スタイルの service 要素パラメータを説明します。

パラメータ	説明
methodName	許可されるメソッド。すべてのメソッドの場合は *、複数のメソッドの場合はスペースで区切ったリストを使用します。
className	バックエンド実装クラスの名前。
allowedRoles	(オプション) サービスにアクセスできるロールのカンマで区切ったリスト。サービス認証が有効な場合に使用します。 詳細については、467 ページの「Web サービスのセキュリティ」を参照してください。

複雑なデータタイプのマッピングが必要な Web サービスを使用する場合は、Axis の bean マッピング機能を利用できます。Axis は、get および set アクセッサの標準 JavaBeans パターンに従っている Java クラスを自動的に直列化および直列化の解除を行うことができます。オペレーションパラメータが JavaBeans オブジェクトに対応しているかぎり、bean マッピングを server-config.wsdd ファイルに含め、そのタイプのオブジェクトを特定の XML QName にマッピングでき、自動的に直列化および直列化の解除を行うことができます。

Web サービスの WSDL ドキュメントからクライアントプロキシコードを生成するときは、WSDL2Java ツールが必要な JavaBeans を自動的に生成します。bean マッピングの詳細については、478 ページの第 23 章「Axis bean シリアライザによるデータタイプマッピング」を参照してください。

ステートレスセッション bean

CMT (Container-managed transactions: コンテナ管理トランザクション) を提供する場合、またはセッション bean がエンティティ bean のファサードとして動作する場合、ステートレスセッション bean は Web サービスバックエンドに適しています。

ステートレスセッション bean をデプロイするには、Web アプリケーションの server-config.wsdd ファイルに Java:EJB のプロバイダ値を指定する service 要素を作成する必要があります。次に例を示します。

```
<service name="AxisSampleLoanEjbService" provider="java:EJB">
  <parameter name="allowedMethods" value="calculate"/>
  <parameter name="jndiPassword" value="AxisPassword"/>
  <parameter name="beanJndiName"
        value="java:sample_ws.SampleLoanEjbHome"/>
  <parameter name="homeInterfaceName" value="ejbeans.SampleLoanHome"/>
  <parameter name="jndiUser" value="AxisUser"/>
 </service>
```

次の表で、service 要素パラメータを説明します。

パラメータ	説明
allowedMethods	(オプション) 許可されるメソッド。すべてのメソッドの場合は *、複数のメソッドの場合はスペースで区切ったリストを使用します。
jndiPassword	(オプション) bean にアクセスするパスワード。パススルーセキュリティにのみ必要です。
jndiURL	bean の JNDI URL。この値は、<JRun のルートディレクトリ >/servers/<JRun のサーバー >/SERVER-INF/jndi.properties ファイル内の java.naming.provider.url の値と一致する必要があります。
jndiContextClass	bean の JNDI コンテキストクラス。
beanJndiName	bean の JNDI 名。
homeInterfaceName	bean のホームインターフェイス名。
jndiUser	(オプション) bean にアクセスするユーザー名。ユーザー名とパスワードを入力する必要のないパススルーセキュリティにのみ必要です。

クラスベースの Web サービスでは、複雑なデータタイプのマッピングが必要な EJB ベースのサービスを使用する場合に、Axis の bean マッピング機能を利用できます。

Web サービスのパッケージングおよびパブリッシュ

このセクションでは、Web サービスのパッケージングおよびパブリッシュに関する一般情報について説明します。

JRun での Web サービスのパブリッシュは簡単です。正しく設定された Axis デプロイメントディスクリプタ server-config.wsdd を標準 Web アプリケーションの WEB-INF ディレクトリに含めて Web アプリケーションをデプロイするのと同じです。アプリケーションのアセンブルおよびデプロイの詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

Web サービスをパブリッシュするには

1 ターゲットの Java クラスまたはステートレスセッション bean のバックエンドに基づいて、server-config.wsdd と呼ばれる Axis の Web サービスデプロイメントディスクリプタ (WSDD) を作成します。

メモ：JWS Web サービスに WSDD ファイルは必要ありません。

JRun samples サーバーの JRun Web サービスのサンプルアプリケーションには、WSDD スニペットおよび完全な WSDD 例が用意されています。一般設定情報に加え、server-config.wsdd ファイルには Web サービスバックエンドクラスまたは EJB の `service` 要素が含まれている必要があります。

2 Web アプリケーションをアセンブルします。

詳細については、『JRun アセンブルとデプロイガイド』を参照してください。

3 Web アプリケーションの WEB-INF ディレクトリに server-config.wsdd ファイルをコピーします。

4 次の手順のいずれかを行います。

- JWS Web サービスをアセンブルする場合は、Web アプリケーションのルートディレクトリ、あるいは WEB-INF または META-INF 以外のサブディレクトリに JWS ファイルをコピーします。
- EJB ベースのサービスではなくクラスベースの Web サービスをアセンブルする場合は、ターゲットの Java クラスファイルを Web アプリケーションの WEB-INF/classes ディレクトリにコピーします。
- EJB ベースの Web サービスをアセンブルする場合は、Web アプリケーションおよび EJB を含んでいるエンタープライズアプリケーションをアセンブルします。

5 クラスベースのサービス用の Web アプリケーションまたは EJB ベースのサービス用のエンタープライズアプリケーションをデプロイします。展開したディレクトリまたはアーカイブファイルに J2EE コンポーネントをデプロイできます。

第 19 章
Web サービスクライアントの作成

この章では、Web サービスクライアントの作成に関する概念とタスクを説明します。

目 次

Web サービスクライアントの作成

Web サービスオペレーションを呼び出すプロキシクライアントまたはダイナミッククライアントを作成できます。

プロキシクライアントは、特定の Web サービスのために WSDL ファイルから生成されたローカルプロキシ上にメソッドを呼び出します。プロキシオブジェクトは、リモート Web サービスとの対話を処理します。プロキシの生成には WSDL2Java ツールを使用します。WSDL2Java ツールの詳細については、457 ページの「WSDL ドキュメントの操作」を参照してください。

ダイナミッククライアントは、Axis クライアント API を直接呼び出すことによってオペレーションを呼び出す JSP または Java クラスです。また、JRun には Web サービスタグライブラリが用意されています。これは、Axis クライアント API へのラッパーとして動作する JSP カスタムタグライブラリです。

プロキシクライアント

このセクションでは、JSP ベースのクライアントから Web サービスプロキシを呼び出す方法を説明します。Web サービスプロキシを生成する方法の詳細については、457 ページの第 20 章「WSDL ドキュメントの操作」を参照してください。

JSP ベースのプロキシクライアントの作成および使用

プロキシコードを作成したら、Web サービスプロキシオブジェクトをインスタンス化してそのメソッドを呼び出すことによってターゲット Web サービスオペレーションを呼び出すプロキシクライアントを作成する必要があります。JSP ベースのプロキシクライアントのサンプルおよびそのソースコードは、samples JRun サーバーの Web サービスサンプルアプリケーション内にあります。

JSP ベースのプロキシクライアントを作成するには

1 プロキシファクトリおよびプロキシをインスタンス化します。次に例を示します。

```
<%
proxy.SampleServiceService stubFactory =
new proxy.SampleServiceService();
proxy.Sample stub = stubFactory.getSampleService();
%>
```

2 プロキシのメソッドを呼び出し、結果をページにプリントします。次に例を示します。

```
<%= stub.getResult(3)%>
```

ダイナミッククライアント

JSP ベースおよびオブジェクトベースのダイナミッククライアントは、Axis クライアント API 内のメソッドを呼び出して Web サービスオペレーションを呼び出します。Axis クライアント API の詳細については、<JRun のルートディレクトリ >/docs/html/thirdparty/axis directory の『Axis User Guide』を参照してください。

JSP スクリプトレットベースのダイナミッククライアントの使用

このセクションでは、JSP スクリプトレットベースのダイナミッククライアントに使用するデザインパターンを説明します。クライアントのサンプルおよびそのソースコードは、samples JRun サーバーの Web サービスサンプルアプリケーション内にあります。サンプルクライアントには、パスワード保護された Web サービス用のものもあります。

JSP スクリプトレットベースのダイナミッククライアントを作成するには

1 次のインポートステートメントを追加します。

```
<%@page import=
"org.apache.axis.client.Call,
org.apache.axis.client.Service,org.apache.axis.encoding.XMLType,
javax.xml.rpc.ParameterMode" %>
```

2 エンドポイント URL の変数、つまり SOAP メッセージの送信先を割り当てます。次に例を示します。

```
<% String endpoint =
"http://nagoya.apache.org:5049/axis/servlet/AxisServlet"; %>
```

3 Axis の `Service` および `Call` オブジェクトをインスタンス化します。これらのオブジェクトは、呼び出すサービスに関するメタデータを保管します。

```
<%
Service service = new Service();
Call call = (Call) service.createCall();
```

4 Web サービスのエンドポイントアドレスを設定します。次に例を示します。

```
call.setTargetEndpointAddress(new java.net.URL(endpoint));
```

5 呼び出す Web サービスオペレーション (メソッド) の名前を設定します。次に例を示します。

```
call.setOperationName("echoString");
```

6 SOAP メッセージの本文で使用するネーム空間を設定します。次に例を示します。

```
call.setProperty(Call.NAMESPACE, "http://soapinterop.org/");
%>
```

7 パラメータ名を設定します。次に例を示します。

```
call.addParameter("testParam", XMLType.XSD_STRING,
Call.PARAM_MODE_IN);
```

メモ：このコード例は、定数 XMLType.XSD_STRING をパラメータのデータタイプとして指定します。Axis データタイプの詳細については、455 ページの「Axis ビルトインデータタイプ」を参照してください。

8 戻りタイプを設定します。次に例を示します。

```
call.setReturnType( XMLType.XSD_STRING );
```

9 Web サービスを呼び出し、結果をページにプリントします。例を次に示します。

```
<%=call.invoke(new Object[] {"Hello!"})%>
```

例

次の JSP には、前の手順で示したすべての要素が含まれています。

```
%@page import="org.apache.axis.client.Call,
org.apache.axis.client.Service,org.apache.axis.encoding.XMLType,
javax.xml.rpc.ParameterMode" %>
<% String endpoint =
"http://nagoya.apache.org:5049/axis/servlet/AxisServlet";
Service service = new Service();
Call call = (Call) service.createCall();
call.setTargetEndpointAddress(new java.net.URL(endpoint));
call.setOperationName("echoString");
call.setProperty(Call.NAMESPACE, "http://soapinterop.org/");
call.addParameter("testParam", XMLType.XSD_STRING,
Call.PARAM_MODE_IN); call.setReturnType( XMLType.XSD_STRING );%>
<%=call.invoke(new Object[] {"Hello!"})%>
```

オブジェクトベースのダイナミッククライアントの使用

このセクションでは、オブジェクトベースのダイナミッククライアントのために使用するデザインパターンを説明します。オブジェクトベースのダイナミッククライアントでは、それに対応するスクリプトレットベースのものと同じコードを使用しますが、ただし、シンタックスは標準 Java シンタックスを使用します。

オブジェクトベースのダイナミッククライアントを作成するには

1 次のインポートステートメントを追加します。

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import javax.xml.rpc.ParameterMode;
```

2 エンドポイント URL の変数、つまり SOAP メッセージの送信先を割り当てます。次に例を示します。

```
String endpoint = "http://nagoya.apache.org:5049/axis/servlet
/AxisServlet";
```

3 Axis の Service および Call オブジェクトをインスタンス化します。これらのオブジェクトは、呼び出すサービスに関するメタデータを保管します。

```
Service service= new Service();
Call call = (Call) service.createCall();
```

4 Web サービスのエンドポイントアドレスを設定します。次に例を示します。

```
call.setTargetEndpointAddress( new java.net.URL(endpoint) );
```

5 呼び出す Web サービスオペレーション (メソッド) の名前を設定します。

次に例を示します。

```
call.setOperationName( "echoString" );
```

6 SOAP メッセージの本文で使用するネーム空間を設定します。次に例を示します。

```
call.setProperty( Call.NAMESPACE, "http://soapinterop.org/" );
```

7 パラメータ名を設定します。次に例を示します。

```
call.addParameter("testParam",
org.apache.axis.encoding.XMLType.XSD_STRING,Call.PARAM_MODE_IN);
```

8 戻りタイプを設定します。次に例を示します。

```
call.setReturnType( XMLType.XSD_STRING );
```

9 Web サービスを呼び出します。次に例を示します。

```
String ret = (String) call.invoke( new Object[] { "Hello!" } );
```

10 結果をコンソールにプリントします。次に例を示します。

```
System.out.println("Sent 'Hello!', got '" + ret + "'");
```

例

次のクライアントには、前の手順で示したすべての要素が含まれています。このクライアントは、Axis の配布用サンプルおよび <JRun のルートディレクトリ >/docs/html/thirdparty/axis ディレクトリの『Axis User Guide』に含まれています。

```
package samples.userguide.example1;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
package samples.userguide.example1;
import org.apache.axis.encoding;
public class TestClient
{
   public static void main(String [] args) {
       try {
           String endpoint =
                    "http://nagoya.apache.org:5049/axis/servlet/
        AxisServlet";
           Service  service = new Service();
           Call     call    = (Call) service.createCall();
          call.setTargetEndpointAddress( new java.net.URL(endpoint) );
           call.setOperationName( "echoString" );
          call.setProperty( Call.NAMESPACE, "http://soapinterop.org/"
        );
```

```
            call.addParameter("testParam", XMLType.XSD_STRING,
          Call.PARAM_MODE_IN);
            call.setReturnType( XMLType.XSD_STRING );
            String ret = (String) call.invoke( new Object[] { "Hello!"
       } );
            System.out.println("Sent 'Hello!', got '" + ret + "'");
        } catch (Exception e) {
            System.err.println(e.toString());
        }
    }
}
```

JSP タグベースのダイナミッククライアントの使用

Web サービスタグライブラリを使用すると、JSP カスタムタグを使用して Axis クライアント API にアクセスできます。サービスのエンドポイント URL およびオペレーションの名前とパラメータがわかっていれば、タグライブラリを使用してオペレーションを呼び出すことができます。

JSP タグベースのダイナミッククライアントのサンプルおよびそのソースコードは、samples JRun サーバーの Web サービスサンプルアプリケーション内にあります。Web サービスのタグライブラリを使用するには、<JRun のルートディレクトリ >/servers/samples/ws-ear/wsc-war/WEB-INF/lib ディレクトリから Web アプリケーションの WEB-INF/lib ディレクトリへ webservicetag.jar ファイルをコピーします。JRun samples をインストールしていない場合は、samples を含んでいるカスタムインストールを実行する必要があります。

このセクションでは、JSP タグベースのダイナミッククライアントのデザインパターンを説明します。ライブラリ内の各タグの説明は、450 ページの「Web サービスタグライブラリリファレンス」を参照してください。

基本的なデータタイプのみを使用するサービスオペレーションの呼び出しは、**invoke** タグ内のオペレーションの指定、およびネストされた **param** タグ内のオペレーションパラメータ値の指定だけなので簡単です。

複雑なデータタイプを使用するサービスでは、**beanmapping** タグを使用して、Axis にビルトインされた JavaBeans シリアライザとユーザーが定義した bean との間でタイプをマッピングできます。詳細については、478 ページの第 23 章「Axis bean シリアライザによるデータタイプマッピング」を参照してください。

inout パラメータ、すなわち Web サービスによって変更できるパラメータを使用するサービスの場合は、**inout** のパラメータモードを指定できます。

デフォルトでは、**invoke** タグは RPC エンコードの SOAP エンコードを使用します。.NET Web サービスが一般的に使用しているドキュメントリテラルや、ドキュメントエンコード、またはラップされたドキュメントリテラルの SOAP エンコードスタイルも使用できます。

JSP タグベースのクライアントを作成するには

1 ページの先頭に、次の行を加えます。

```
<%@ taglib prefix="web" uri="WEB-INF/lib/webservicetag.jar" %>
```

この行は、JSP の taglib ディレクティブです。これによって、Web サービスタグライブラリのタグが **"web"** 接頭辞を使用し、タグライブラリが Web アプリケーションの WEB-INF/lib ディレクトリの webservicestag.jar ファイルにあることを指定します。

2 (オプション) Web サービスオペレーションパラメータの変数を割り当てます。

次のサンプルコードでは、通貨の両替レートを変換する Microsoft SOAP Web サービスで使用する通貨コードの変数を設定します。USD は米ドルを表し、BRL はブラジルのレアルを表します。

```
<% String str1="BRL";
String str2="USD";%>
```

3 invoke タグを使用して、Web サービスオペレーションを呼び出します。

次の例では、デフォルト RPC エンコードの SOAP エンコードスタイルを使用する通貨変換 Web サービスを呼び出します。

```
<web:invoke
url="http://www.itfinity.net:8008/soap/exrates/default.asp"
namespace="http://www.itfinity.net/soap/exrates/exrates.xsd"
resulttype ="double"
operation="GetRate"
result="myresult">
</web:invoke>
```

この invoke タグには、次の情報があります。

- Web サービスのエンドポイント URL
- Web サービスの XML ネーム空間
- 予期される戻り値のデータタイプ
- 呼び出すオペレーションの名前
- オペレーション結果を保管する変数の名前

4 invoke タグの本文 (</web:invoke> の直前) のオペレーションに対して、適切な param タグを指定します。

次の例では、通貨変換 Web サービスの GetRate オペレーションの from および to の値を指定します。name の値に大文字を正しく使用していることを確認します。

```
<web:param name="fromCurr" value="<%=str1%>"/>
<web:param name="ToCurr" value="<%=str2%>"/>
```

5 オペレーション結果をページにプリントします。

次は、GetRate オペレーションが返した両替レートをプリントする例です。

```
<%= pageContext.getAttribute("myresult") %>
```

例

次の JSP には、前の手順で示したすべての要素が含まれています。

```
<%@ taglib prefix="web" uri="webservicetag" %>
<% String str1="BRL";
String str2="USD";%>
<web:invoke
url="http://www.itfinity.net:8008/soap/exrates/default.asp"
namespace="http://www.itfinity.net/soap/exrates/exrates.xsd"
operation="GetRate"
resulttype="double"
result="myresult">
<web:param name="fromCurr" value="<%=str1%>"/>
<web:param name="ToCurr" value="<%=str2%>"/>
</web:invoke>
<%= pageContext.getAttribute("myresult") %>
```

Web サービスタグライブラリリファレンス

次のセクションでは、Web サービスタグライブラリのタグの使用方法、シンタックス、および属性を説明します。

Web サービスのタグライブラリを使用するには、<JRun のルートディレクトリ >/servers/samples/ws-ear/wsc-war/WEB-INF/lib ディレクトリから Web アプリケーションの WEB-INF/lib ディレクトリへ webservicetag.jar ファイルをコピーします。JRun samples をインストールしていない場合は、samples を含んでいるカスタムインストールを実行する必要があります。

invoke タグ

説明 invoke タグは、Web サービスタグライブラリの主要タグです。このタグを使用して、サービスのエンドポイント URL およびオペレーション名がわかっている Web サービスオペレーションを呼び出します。タグには、ユーザー名やパスワードなどの補足情報を提供するオプション属性があります。invoke タグの本文には、オペレーションパラメータを指定する 1 つ以上の param タグを含めます。

シンタックス

```
<web:invoke
  url=" サービスのエンドポイント URL"
  operation=" 呼び出すオペレーションの名前 "
  result=" 結果の変数名 "
  [use="literal または encoded"]
  [style="rpc、document、または wrapped"]
  [resultType=" オペレーション結果のデータタイプ "]
  [namespace=" オペレーションのネーム空間 "]
  [soapaction=" オペレーションに対する SOAP アクション "]
  [resultTypeNamespace="resulttype のネーム空間 "]
  [scope=" 結果変数の JSP スコープ "]
  [username=" 認証用のユーザー名 "]
  [password=" 認証用のパスワード "]
</web:invoke>
```

属性

属性	必須またはオプション	説明
url	必須	java.lang.String Web サービスのエンドポイント URL です。
operation	必須	java.lang.String 呼び出す Web サービスオペレーションの名前です。
result	必須	java.lang.String Web サービスオペレーション結果を保管するプロパティの名前です。
use	オプション	java.lang.String literal SOAP、または encoded SOAP。デフォルトは encoded です。 Web サービスがリテラルの SOAP を使用するのかエンコードの SOAP を使用するのかを判断するには、WSDL ドキュメントの soap:body 要素の use 属性をチェックします。ほとんどの .NET Web サービスはリテラル SOAP エンコードを使用します。

属性	必須またはオプション	説明
style	オプション	java.lang.String `rpc`、`document` または `wrapped` の SOAP スタイル。デフォルトは、`rpc` です。 Web サービスが rpc スタイルを使用するのか document スタイルを使用するのかを判断するには、WSDL ドキュメントの `soap:operation` 要素の `style` 属性をチェックします。 wrapped スタイルはオペレーション名で名付けられた唯一の要素にオペレーションのパラメータをラップするドキュメントスタイルです。これは、.NET Web サービスが一般的に使用するスタイルです。
resultType	オプション	java.lang.String Web サービスオペレーション結果の XML データタイプです。この値は Web サービスの WSDL ドキュメント内の戻りタイプから取得します。詳細については、455 ページの「Axis ビルトインデータタイプ」を参照してください。
namespace	オプション	java.lang.String Web サービスオペレーションのネーム空間です。
soapaction	オプション	java.lang.String オペレーションに対する SOAP アクションです。
resultTypeNamespace	オプション	java.lang.String 結果タイプのネーム空間です。この値は Web サービスの WSDL ドキュメント内の戻りタイプのネーム空間から取得します。
scope	オプション	java.lang.String このタグによって返されるオブジェクトのデフォルトの JSP スコープです。有効な値は、page、request、session、および application です。デフォルトは `page` です。
username	オプション	java.lang.String Web サービスが認証を要求したときのユーザー名です。
password	オプション	java.lang.String Web サービスが認証を要求したときのパスワードです。

param タグ

説明 invoke タグの本文の 1 つ以上の param タグを使用して、Web サービスオペレーションのパラメータを指定します。

シンタックス

```
<web:param
   value=" オペレーションパラメータ値 "
   [name=" オペレーションパラメータ名 "]
   [mode=" オペレーションパラメータモード "]
   [namespace=" オペレーションパラメータのネーム空間 "]
/>
```

属性

属性	必須またはオプション	説明
value	必須	データタイプは関連するオペレーションパラメータによって異なります。 Web サービスオペレーションパラメータの値です。
name	オプション	java.lang.String Web サービスオペレーションパラメータの名前です。
mode	オプション	java.lang.String オペレーションパラメータのモードです。有効な値は inout、in、および out です。WSDL ファイルに含まれているパラメータのモードが入力パラメータ (in)、出力パラメータ (out)、または両方 (inout) のいずれであるかを確認することによって、パラメータのモードを判断します。データを入力と出力の両方に渡す inout パラメータには、mode 属性が必要です。
namespace	オプション	java.lang.String Web サービスオペレーションパラメータのネーム空間です。

beanmapping タグ

説明 パラメータが基本データタイプでない場合、**beanmapping** タグを **invoke** タグとともに使用して、Web サービスオペレーションパラメータを Java クラスにマッピングします。Java クラスには、パラメータに対して Web サービスが送受信を行うデータの取得および設定のための標準 JavaBeans の get および set アクセッサが含まれている必要があります。

beanmapping タグは、Axis のビルトイン JavaBeans シリアライザを使用して、Java と XML 間のパラメータ値の直列化や直列化の解除を行います。このタグはサーバー上で定義されたネーム空間を使用する必要があります。bean マッピングを使用する Web サービスとクライアントのサンプルは、Samples JRun サーバーの Web サービスサンプルアプリケーション内にあります。bean マッピングの詳細については、478 ページの第 23 章「Axis bean シリアライザによるデータタイプマッピング」と <JRun のルートディレクトリ >/docs/html/thirdparty/axis ディレクトリの『Axis User Guide』を参照してください。

```
<web:beanmapping
class="com.macromedia.myclass"
namespace="http://macromedia.com/xsd"
type="myclass"
[scope="結果変数の JSP スコープ"]
/>
```

属性

属性	必須またはオプション	説明
class	必須	java.lang.String bean マッピングに使用する、JavaBeans の完全修飾 Java クラス名です。
namespace	必須	java.lang.String bean マッピングが割り当てられるオペレーションパラメータのネーム空間です。
type	必須	java.lang.String JavaBeans クラスのスキーマタイプで、通常は JavaBeans 名です。
scope	オプション	java.lang.String このタグによって返されるオブジェクトのデフォルトの JSP スコープです。有効な値は、page、request、session、および application です。デフォルトは **page** です。 ページより広い範囲を指定すると、Web アプリケーションはグローバル bean マッピング (アプリケーション全体にわたる bean マッピング) を確立でき、その他の invoke タグを bean マッピングの指定なしで使用できます。

Axis ビルトインデータタイプ

次の表で、Axis に組み込まれた XML データタイプ、割り当てられた Axis 定数、およびそれらに相当する Java データタイプを説明します。

接頭辞 xsd は次の XML ネーム空間 URI を表します。

http://www.w3.org/2001/XMLSchema

接頭辞 SOAP-ENC は次の XML ネーム空間 URI を表します。

http://schemas.xmlsoap.org/soap/encoding

XML データタイプ (接頭辞は完全修飾のネーム空間を表します)	XML データタイプの Axis 定数	Java データタイプ
xsd:string	XSD_STRING	java.lang.String
xsd:boolean	XSD_BOOLEAN	boolean
xsd:double	XSD_DOUBLE	double
xsd:float	XSD_FLOAT	float
xsd:int	XSD_INT	int
xsd:integer	XSD_INTEGER	java.math.BigInteger
xsd:long	XSD_LONG	long
xsd:short	XSD_SHORT	short
xsd:byte	XSD_BYTE	byte
xsd:decimal	XSD_DECIMAL	java.math.BigDecimal
xsd:base64Binary	XSD_BASE64	byte[]
xsd:hexBinary	XSD_HEXBIN	byte[]
xsd:QName	XSD_QNAME	javax.xml.rpc.namespace.QName
xsd:dataTime	XSD_DATE	java.util.date
SOAP-ENC:base64	SOAP_BASE64	byte[]
SOAP-ENC:string	SOAP_STRING	java.lang.String
SOAP-ENC:boolean	SOAP_BOOLEAN	boolean
SOAP-ENC:double	SOAP_DOUBLE	double
SOAP-ENC:float	SOAP_FLOAT	float
SOAP-ENC:int	SOAP_INT	int
SOAP-ENC:long	SOAP_LONG	long
SOAP-ENC:short	SOAP_SHORT	short
SOAP-ENC:byte	SOAP_BYTE	byte
SOAP-ENC:integer	SOAP_INTEGER	java.math.BigInteger

XML データタイプ (接頭辞は完全修飾のネーム空間を表します)	XML データタイプの Axis 定数	Java データタイプ
SOAP-ENC:decimal	SOAP_DECIMAL	java.math.BigDecimal
SOAP-ENC:Array	SOAP_ARRAY	ビルトインデータタイプの配列
SOAP-ENC:Map	SOAP_MAP	java.util.HashMap
SOAP-ENC:Element	SOAP_ELEMENT	org.w3c.dom.Element
SOAP-ENC:Vector	SOAP_VECTOR	java.util.Vector

メモ：Web サービスタグライブラリが xsd ネーム空間内で resulttype のあるタグを呼び出すために、resulttypenamespace 属性に完全修飾の xsd ネーム空間 URI を含める必要はありません。代わりに `resulttype ="double"` のようにコロンの後にテキストを指定すれば呼び出すことができます。

ダイナミッククライアントでデータタイプを指定している場合は、次のように Axis 定数を使用できます。

```
call.addParameter("testParam", XMLType.XSD_STRING, Call.PARAM_MODE_IN);
call.setReturnType( XMLType.XSD_STRING );
```

第 20 章
WSDL ドキュメントの操作

この章では、Axis ベースの Web サービスの WSDL ドキュメントの生成方法と、WSDL ドキュメントからのクライアントサイドおよびサーバーサイド Web サービスコードの生成方法について説明します。

目 次

概要

WSDL ドキュメントとは、Web サービスの目的、配置場所、およびアクセス方法を記述している XML ファイルのことです。ユーザーが呼び出すオペレーションおよびそれに関連するデータタイプが記述されています。

JRun を使用して次のタスクを実行できます。

- パブリッシュ済み Web サービスから WSDL ドキュメントを生成します。
- Java インターフェイスまたはクラスから WSDL ドキュメントを生成します。
- WSDL ドキュメントから Web サービスのスケルトンおよびプロキシを生成します。

WSDL の詳細については、次の URL にある W3C の WSDL ドキュメントをご覧ください。

http://www.w3.org/TR/wsdl

パブリッシュ済み Web サービスからの WSDL の生成

最後に `?WSDL` パラメータを持つサービスのエンドポイント URL にアクセスすることによって、パブリッシュ済み Web サービスの WSDL ドキュメントを生成できます。次に例を示します。

- JWS サービス

  http://localhost:8200/ws/AxisSampleJavaClass.jws?WSDL
- クラスベースまたは EJB ベースのサービス

  http://localhost:8200/ws/services/AxisSampleJavaClassService?WSDL

  ここで、AxisSampleJavaClassService は、Web アプリケーションの server-config.wsdd ファイル内のこの Web サービスの `service` 要素に指定されている名前です。
- Secured Web サービス
- http://AxisUser:AxisPassword@localhost:8200/ws/services/AxisSampleAuthService?WSDL

メモ：WSDL ドキュメントは Internet Explorer では表示されますが、Netscape Navigator ではドキュメントが返されても空白の Web ページが表示されます。

WSDL の URL にアクセスしたら、ブラウザの [名前を付けて保存] コマンドを使用して、ファイルに wsdl 拡張子を付けて保存します。URL に WSDL ドキュメントをパブリッシュして、ユーザーがクライアントからサービスを呼び出すときに必要な情報を提供できます。

Java インターフェイスまたはクラスからの WSDL の生成

Java2WSDL ツールを使用して、Web サービスの基礎として使用する Java インターフェイスまたはクラスの WSDL ドキュメントを生成できます。Java2WSDL で WSDL ドキュメントを生成したら、WSDL2Java ツールを使用して、適切なプロキシおよびスケルトンコードを生成し、Web サービスのデプロイメントディスクリプタ情報を生成できます。詳細については、462 ページの「WSDL ドキュメントからの Java コードの生成」を参照してください。

メモ：クラスをデバッグ情報とともにコンパイルする場合、Java2WSDL はデバッグ情報を使用してメソッドパラメータ名を取得します。

Java インターフェイスまたはクラスから WSDL ドキュメントを生成するには

- 次のいずれかの方法で Java2WSDL ツールを実行します。

  ```
  cd <JRun のルートディレクトリ>¥bin
  java2wsdl -o sample.wsdl -l"http://localhost:8111/services/MyService"
  -n "urn:Sample1" -p"samples.sample1" "urn:Sample1"
  samples.Sample1.MyServiceClass
  ```

  または

  ```
  java -classpath "<JRun のルートディレクトリ>/lib/jrun.jar;
  <JRun のルートディレクトリ>/lib/webservices.jar"
  org.apache.axis.wsdlgen.Java2WSDL -o sample.wsdl
  -l"http://localhost:8111/services/MyService" -n "urn:Sample1"
  -p "samples.sample1"="urn:Sample1" samples.Sample1.MyServiceClass
  ```

  ここで、

  - -o は出力 WSDL ドキュメント名を指定します。
  - -l はサービスの場所を指定します。
  - -n は WSDL ドキュメントのターゲットネーム空間を指定します。
  - -p はパッケージからネーム空間へのマッピングを指定します。マッピングは複数ある場合があります。指定されたクラス (この場合は MyServiceClass) には Web サービスのインターフェイスが含まれています。

Java2WSDL のコマンドラインスイッチ

次の二重ダッシュ (–) またはダッシュ (-) バージョンのコマンドラインスイッチオプションを使用できます。

スイッチ	説明
--help、 -h	使用方法をプリントします。
--output、 -o <WSDL ファイル名>	出力される実装 WSDL ドキュメントの名前を示します。このオプションを指定すると、Java2WSDL はインターフェイスおよび 実装 WSDL ドキュメントを生成します。このスイッチを使用する場合は、Java2WSDL は --outputWSDLMode スイッチを無視します。
--location、 -l	サービスの場所の URL を示します。スラッシュまたはバックスラッシュの後の名前は、--service 値で書き換えられないかぎりサービスポートの名前です。サービスポートアドレスの場所の属性は指定された値に割り当てられます。
--service、 -s	サービス名を示します。指定されていない場合は、サービス名は --location 値から生成されます。WSDL バインディング、サービス、およびポート要素の名前は、サービス名から生成されます。
--namespace、 -n <ターゲットネーム空間>	WSDL ドキュメントのターゲットネーム空間名を示します。
--PkgToNS、 -p <パッケージ>=< ネーム空間>	パッケージとネーム空間のマッピングを示します。ネーム空間を持たないパッケージがあると、Java2WSDL は適切なネーム空間名を生成します。このスイッチは何回でも指定できます。
--methods、 -m <メソッド名>	インターフェイスクラスで指定されたメソッドのみを WSDL ドキュメントにエクスポートします。メソッドのリストはスペースかカンマで区切る必要があります。このオプションを指定しないと、インターフェイスクラスで宣言されたすべてのメソッドが WSDL ドキュメントにエクスポートされます。
--all、 -a	Java2WSDL は拡張クラスを検証して、WSDL ドキュメントにエクスポートするメソッドのリストを作成します。
--outputWSDLMode、 -w <モード>	生成する WSDL のタイプを示します。使用できる値は次のとおりです。 • All (デフォルト) インターフェイスと実装 WSDL コンストラクトの両方を含む WSDL ドキュメントを生成します。 • Interface インターフェイスコンストラクトを含んでいる WSDL ドキュメントを生成します (サービス要素なし)。 • Implementation 実装を含んでいる WSDL ドキュメントを生成します。インターフェイス WSDL は –locationImport スイッチを使用してインポートされます。

スイッチ	説明
--locationImport、 -L <url>	実装 WSDL ドキュメントを生成するときに、インターフェイス WSDL ドキュメントの場所を示します。
--namespaceImpl、 -N <ネーム空間>	実装 WSDL ドキュメントのネーム空間を示します。
--outputImpl、 -O <WSDL ファイル名>	指定された実装 WSDL ファイル名を出力します。このスイッチを使用する場合は、Java2WSDL は --outputWSDLMode スイッチを無視します。
--factory、 -f <クラス>	エキスパートスイッチによって Java2WSDL が拡張およびカスタマイズされます。
--implClass、 -i <impl-class>	Java2WSDL はメソッドパラメータ名を使用して WSDL メッセージ部分の名前を生成します。メッセージ名は <class-of-portType> クラスファイルのデバッグ情報から取得します。クラスファイルがデバッグ情報なしにコンパイルされたり、<class-of-portType> がインターフェイスである場合は、メソッドパラメータ名はありません。このような場合は、--implClass スイッチを使用して、メソッドパラメータ名を取得する代替クラスを与えます。<impl-class> となるのは、実際の実装クラス、スタブクラス、またはスケルトンクラスです。
--exclude、 -x <メソッド名>	WSDL ファイルへのエクスポートからスペースやカンマ区切りのメソッドリストを除外します。
--stopClasses、 -c <クラス名>	スペース区切りやカンマ区切りのクラス名リストで、--all スイッチを使用したときのクラス継承検索が停止されます。これにより、Java2WSDL は拡張クラスを調べて、WSDL ドキュメントにエクスポートするメソッドのリストを判断します。

WSDL ドキュメントからの Java コードの生成

WSDL2Java ツールは、Web サービスの WSDL ドキュメントの情報に基づいて Web サービスプロキシおよびスケルトンのコードを生成します。**プロキシ**とは、実際の Web サービスと同じインターフェイスを持つローカルオブジェクトのことです。これを使用すると、Web サービスがローカルオブジェクトであるかのように呼び出すことができます。WSDL2Java は、RPC エンコードされていてドキュメントリテラルで、しかもドキュメントエンコードされた SOAP エンコードスタイルを指定する WSDL ドキュメントからプロキシを生成します。

スケルトンとは、新規 Web サービス実装のテンプレートのことです。スケルトンを生成しておけば、後は独自のコードを追加するだけで実装を完成できます。

WSDL2Java のコマンドラインスイッチの詳細については、465 ページの「WSDL2Java のコマンドラインスイッチ」を参照してください。

Web サービスプロキシの生成

このセクションでは、Web サービスプロキシを作成する方法について説明します。JSP ベースのクライアントからのプロキシの呼び出しについては、444 ページの「プロキシクライアント」を参照してください。

Web サービスのプロキシコードを生成するには

1 Web サービスの WSDL ドキュメントを取得します。

Axis ベースの Web サービスの WSDL ドキュメント生成方法の詳細については、458 ページの「パブリッシュ済み Web サービスからの WSDL の生成」を参照してください。

2 コマンドプロンプトで、次のいずれかの方法で WSDL2Java ツールを実行します。

```
cd <JRun のルートディレクトリ>¥bin
wsdl2java -v -o <出力ディレクトリのローカルパス>/<ファイル名>.wsdl
```

または

```
java -classpath "<JRun のルートディレクトリ>/lib/jrun.jar;<JRun のルートディレクトリ>/lib/webservices.jar"
org.apache.axis.wsdl.WSDL2Java -v -o <出力ディレクトリのローカルパス>/<ファイル名>.wsdl
```

ここで、

- -v によってツールは情報メッセージを表示します。
- -o は出力ディレクトリ名を指定します。
- *localpath/your_file*.wsdl は WSDL ドキュメントのディレクトリおよびファイル名を表します。

メモ：Axis ベースの Web サービスでは、手順 2 のコマンドラインの URL として <ローカルパス>/<ファイル名>.wsdl を使用する代わりに、458 ページの「パブリッシュ済み Web サービスからの WSDL の生成」で説明しているように、WSDL 引数を持つ Web サービスのエンドポイント URL を使用できます。

生成したプロキシコードの確認

WSDL2Java は次のプロキシファイルを生成します。

ファイル	説明
< サービス名 >Locator.java	このクラスはプロキシのインスタンスを取得するファクトリの役割をします。これは、< サービス名 >.java インターフェイスを実装します。これは WSDL ファイルのサービス名から生成されます。 このクラス名は、WSDL ドキュメント内のサービス名に接頭辞 Locator を付けたものです。WSDL ファイルに 2 つ以上のサービスがリストされている場合は、サービスごとに 1 つのクラスが生成される必要があります。このクラス内の `get` メソッドを使用して、`web service` インターフェイスを実装するスタブオブジェクトを取得します。
< サービス名 >.java	これは、< サービス名 >Locator.java クラスが実装するファクトリインターフェイスです。 このインターフェイス名は、WSDL ドキュメント内のサービス名から付けられます。
< サービスポート名 >.java (エンコードされた WSDL の場合) < サービスバインド名 >.java (リテラル WSDL の場合)	このインターフェイスには、各 Web サービスオペレーションのメソッド署名が含まれています。 このインターフェイス名は、WSDL ドキュメント内のポート名やバインド名から付けられます。
<サービスバインド名>Stub.java	これは、Web サービスプロキシクラスです。これは、Axis クライアント API を使用して Web サービス呼び出しを行う < サービスポート名 >.java インターフェイスを実装します。 このクラス名は、サービスバインド名に接頭辞 Stub を付けたものです。

複雑なデータタイプを指定する WSDL ドキュメントでは、WSDL2Java はこれらのタイプを表す JavaBeans クラスを生成します。Axis が JavaBeans を使用する方法の詳細については、478 ページの「Axis bean シリアライザによるデータタイプマッピング」を参照してください。

WSDL ドキュメントでは、入力と出力の両方に使用する in-out パラメータ、および出力のみに使用する out パラメータも指定できます。Java では inout および out パラメータを直接サポートしていないので、必要な場合は、WSDL2Java は**ホルダー**クラスを生成します。これを使用するとサービスはパラメータを変更し、クライアントに返すことができます。ホルダークラスを必要とする各クラスに対して、WSDL2Java は同じベース名を持つホルダークラスをそれぞれ生成します。たとえば、MyClass というクラスは、MyClassHolder というホルダークラスを持ちます。

Web サービススケルトンの生成

デフォルトで WSDL2Java が生成するプロキシコードに加えて Web サービススケルトンコードを生成するには、-s または --skeleton スイッチを使用します。

このセクションでは、Web サービススケルトンを作成する方法について説明します。

Web サービススケルトンコードを生成するには

1 Web サービスの WSDL ドキュメントを取得します。Axis ベースの Web サービスの WSDL ドキュメント生成方法の詳細については、458 ページの「パブリッシュ済み Web サービスからの WSDL の生成」を参照してください。

2 コマンドプロンプトで、次のいずれかの方法で WSDL2Java ツールを実行します。

```
cd <JRun のルートディレクトリ>¥bin
wsdl2java -v -o -s <出力ディレクトリのローカルパス>/<ファイル名>.wsdl
```

または

```
java -classpath "<JRun のルートディレクトリ>/lib/jrun.jar:
     <JRun のルートディレクトリ>/lib/webservices.jar"
org.apache.axis.wsdl.WSDL2Java -v -o -s
<出力ディレクトリのローカルパス>/<ファイル名>.wsdl
```

ここで、

- -v によってツールは情報メッセージを表示します。
- -o は出力ディレクトリ名を指定します。
- -s によってツールはスケルトンコードを生成します。
- <ローカルパス / ファイル名 >.wsdl は WSDL ドキュメントのディレクトリおよびファイル名を表します。

メモ：Axis ベースの Web サービスでは、手順 2 のコマンドラインの URL として <ローカルパス >/< ファイル名 >.wsdl を使用する代わりに、458 ページの「パブリッシュ済み Web サービスからの WSDL の生成」で説明しているように WSDL 引数を持つ Web サービスのエンドポイント URL を使用できます。

生成したスケルトンコードおよびデプロイ情報の確認

WSDL2Java は次のスケルトンファイルを生成します。

ファイル	説明
<サービス名 >SoapBindingSkeleton.java	これは、Web サービスとしてデプロイするクラスです。
<サービス名 >SoapBindingImpl.java	これは、独自の実装コードを追加する実装クラステンプレートです。
deploy.wsdd	このファイルには、Web サービスのデプロイメントディスクリプタである server-config.wsdd ファイルにコピーアンドペーストができる WSDD service 要素が含まれています。

複雑なデータタイプを使用する Web サービスでは、WSDL2Java はこれらのタイプを表す JavaBeans クラスを生成します。Axis が JavaBeans を使用する方法の詳細については、478 ページの「Axis bean シリアライザによるデータタイプマッピング」を参照してください。

WSDL ドキュメントは、入力と出力の両方に使用する inout パラメータ、および出力のみに使用する out パラメータを指定できます。Java では inout および out パラメータを直接サポートしていないので、必要な場合は、WSDL2Java はホルダークラスを生成します。これを使用するとパラメータは入力値と出力値の両方を含むことができます。ホルダークラスを必要とする各クラスに対して、WSDL2Java は同じベース名を持つホルダークラスをそれぞれ生成します。たとえば、MyClass というクラスは、MyClassHolder というホルダークラスを持ちます。

WSDL2Java のコマンドラインスイッチ

WSDL2Java ツール用の次の二重ダッシュ (--) またはダッシュ (-) バージョンのコマンドラインスイッチを使用できます。

スイッチ	説明
`--help、-h`	使用方法をプリントします。
`--verbose、` `-v`	ツールが生成するものに関する情報メッセージを生成時にプリントします。
`--skeleton、-s`	新規 Web サービスのスケルトンコードを生成します。
`--skeletonDeploy、` `-S<true または false>`	スケルトンをデプロイするか、実装クラスをデプロイするかを指定します。この値が false の場合は、WSDL2Java は実装クラスを使用します。
`--NStoPkg、` `-N <argument>=<value>`	Java パッケージとネーム空間をマッピングします。 デフォルトでは、パッケージ名は WSDL ドキュメントのネーム空間文字列から生成されます。固有のネーム空間マッピングごとに --NStoPkg 引数を使用して、独自のマッピングを提供できます。たとえば、"urn:AddressFetcher2" という WSDL ドキュメントにネーム空間があり、このネーム空間内のオブジェクトから生成したファイルをパッケージ samples.addr に配置する場合は、次のスイッチを WSDL2Java に指定します。 --NStoPkg urn:AddressFetcher2=samples.addr WSDL ドキュメントに多くのネーム空間がある場合は、それらすべてのマッピングをリストするのに時間がかかる場合があります。コマンドラインを簡潔にするために、WSDL2Java はデフォルトのパッケージ (パッケージなし) 内にある NStoPkg.properties と呼ばれるファイル内でもマッピングを検証します。このファイルのエントリは、--NStoPkg コマンドラインスイッチの引数と同じ形式です。たとえば、上記のようにコマンドラインスイッチを指定する代わりに、同じ情報を NStoPkg.properties に与えることができます。 urn¥:AddressFetcher2=samples.addr (プロパティファイルではコロンをエスケープする必要があります。) 指定されたマッピングのエントリがコマンドラインとプロパティファイルの両方にある場合は、コマンドラインのエントリが優先されます。

スイッチ	説明
--package、 -p <パッケージ名>	すべての名前空間とパッケージのマッピングを変更して、代わりに指定されたパッケージ名を使用します。
--output、 -o <ディレクトリ>	指定された出力ディレクトリにファイルを生成します。
--deployScope、 -d <引数>	生成された WSDD ファイル (deploy.wsdd) に、Application、Request、または Session のスコープを追加します。 このスイッチを使用しない場合は、deploy.xml ファイルにスコープタグが現れず、AXIS のランタイムデフォルト設定は Request スコープとなります。
--testCase、-t	クライアントサイド JUnit テストケースを生成します。
--noImports、-n	コマンドラインに表示されるドキュメント専用のコードを生成します。デフォルトの動作では、すべての WSDL ドキュメントのファイル、直接のファイル、およびインポートされたファイルが生成されます。
--all、 -a	参照されなかった要素を含むすべての要素のコードを生成します。
--debug、-D	デバッグ情報を表示します。

第 21 章
Web サービスのセキュリティ

この章では、Web サービスでの JRun 認証および SSL の使用方法について説明します。

目 次

JRun Web サービスの認証の設定

<JRun のルートディレクトリ >/servers/< サーバー名 >/SERVER-INF/default-web.xml ファイルには、次の AxisServlet 定義が含まれています。**use-servlet-security** パラメータは、Axis が JRun サーブレットセキュリティメカニズムを使用するかどうかを決めます。このパラメータはデフォルトで true に設定されています。

```
<servlet>
    <servlet-name>AxisServlet</servlet-name>
    <display-name>Apache-Axis Servlet</display-name>
    <servlet-class>org.apache.axis.transport.http.AxisServlet</servlet-class>
    <init-param>
         <param-name>use-servlet-security</param-name>
         <param-value>true</param-value>
     </init-param>
 </servlet>
```

default-web.xml ファイル内で Web サービス認証が有効な場合でも、Web アプリケーションの server-config.wsdd ファイルおよび Web サービスクライアントが認証を使用するように設定する必要があります。468 ページの「JRun の認証を使用するための Axis の設定」の手順に従ってください。

JRun の認証を使用するための Axis の設定

Web サービス認証を使用するように Axis を設定するには、次の手順を実行します。

Web サービス認証を設定するように Axis を設定するには

1 server-config.wsdd ファイル内のサービスの **service** セクションに、Axis 認証および許可ハンドラを使用するリクエストフローを追加します。

次に例を示します。

```
<requestFlow name="checks">
<handler type=
     "java:org.apache.axis.handlers.SimpleAuthenticationHandler" />
<handler type=
     "java:org.apache.axis.handlers.SimpleAuthorizationHandler" />
</requestFlow>
```

2 サービスの **service** セクションに、1 つ以上のカンマで区切った認証ロール (デフォルトでは <JRun のルートディレクトリ >/servers/< サーバー名 >/SERVER-INF/jrun-users.xml 内に設定されている) を含んでいる **allowedRoles** パラメータを追加します。指定されたロールのみがサービスにアクセスできます。次に例を示します。

```
<parameter name="allowedRoles" value="AxisRole"/>
```

Web サービスクライアントが認証を使用できるように設定する

このセクションでは、認証を使用するプロキシおよびダイナミッククライアントをコーディングする方法について説明します。また、Web アプリケーションのデプロイメントディスクリプタ web.xml をクライアント認証に使用できるように設定する方法についても説明します。

認証を使用するためのプロキシクライアントのコーディング

プロキシクライアントにおいて認証を使用するには、次の例で太字で示しているようにユーザー名とパスワードを設定できます。

```
proxy.SampleJavaClass client = new proxy.SampleJavaClass();
proxy.SampleJavaClassPortType stubI =
client.getSampleJavaClassPort();
proxy.SampleAuthServiceSoapBindingStub stub =
(proxy.SampleAuthServiceSoapBindingStub) stubI;
stub._setProperty("user.id","AxisUser");
stub._setProperty("user.password","AxisPassword");
```

認証を使用するためのダイナミッククライアントのコーディング

オブジェクトベースまたはスクリプトレットベースのダイナミッククライアントにおいて認証を使用するには、次の例で太字で示しているようにユーザー名とパスワードを設定できます。

```
Service service = new Service();
Call call = (Call) service.createCall();
call.setTargetEndpointAddress(new java.net.URL(url));
call.setProperty(Call.NAMESPACE, serviceName);
call.setOperationName("calculate");
call.setProperty("http.auth.username", "user");
call.setProperty("http.auth.password", "password");
```

Web サービスのタグライブラリを使用する JSP において認証を使用するには、invoke タグにユーザー名とパスワードの値を指定します。詳細については、450 ページの「Web サービスタグライブラリリファレンス」を参照してください。

クライアント認証を使用するための web.xml ファイルの設定

セキュリティを使用するように Web アプリケーションを設定するには、Web アプリケーションの WEB-INF ディレクトリ内の web.xml ファイルに `security-constraint` および `login-config` セクションを追加する必要があります。次の例は、`security-constraint` および `login-config` セクションを示しています。`url-pattern` および `role-name` の値を Web サービスに対応した値で置き換えます。`url-pattern` は Web サービスの URL の全部または一部を表しています。これにより、その Web サービスにのみ認証を適用できます。クライアント内に Web サービスの URL を指定するときは、別個の URL パターンを持つように、サービス名を含んでいる完全な URL を使用してください。`role-name` は server-config.wsdd ファイルに指定された `allowedRoles` 値と一致します。

```
<security-constraint>
    <web-resource-collection>
      <url-pattern>/services/SampleAuthService*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>AxisRole</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

JRun の Web サービスで SSL を使用する

Web サービスとクライアント間で安全なプラベート通信を行うために HTTP over SSL (HTTPS) を使用します。JRun の Web サーバーまたは外部 Web サーバーを通じて SSL を使用するように設定した JRun では、Web サービスクライアントは HTTPS を介して JRun Web サービスと通信できます。さらに、JRun 内で実行中の Web サービスクライアントは、わずかな設定で HTTPS を使用して Web サービスと通信できます。

SSL を使用するように JRun を設定する

SSL を使用するように JRun を設定する方法については、JMC (JRun 管理コンソール) のオンラインヘルプを参照してください。必要な SSL ハンドシェークを設定した Web サービスクライアントは、HTTPS を使用して JRun Web サービスでメソッドを呼び出すことができます。

SSL を使用するための Web サービスクライアントの設定

この例では、サーバーからの SSL 証明書のコピーを持っていることを前提としています。証明書のコピーを取得するには、HTTPS を介して Web ブラウザを通じてサーバーに接続し、証明書をファイルにエクスポートします。

SSL 証明書のコピーを入手したら、トラストストアに入れる必要があります。トラストストアでは、クライアントが信頼するサーバーからの証明書を保持します。Sun の JRE (Java Runtime Environment：Java ランタイム環境) に含まれている keytool ユーティリティを使用してトラストストアを作成し、サーバーの証明書をインポートできます。

SSL 証明書を作成してインポートするには

1 コマンドプロンプトで、<JRun のルートディレクトリ>/lib ディレクトリに変更します。

2 次のコマンドを入力します。

```
<Java のホームディレクトリ>bin/keytool -import -file <ファイルパス/
    証明書名>.cer
-alias JRun -keystore trustStore
```

3 パスワード入力のプロンプトが表示されたら、トラストストアのパスワードを入力します。

4 JRun サーバーの jrun.xml ファイル内の次の vmArgs 属性を設定します。
javax.net.ssl.keyStorePassword の値には、手順 3 で入力したパスワードを指定します。

```
<service class="jrunx.launcher.LauncherInfo" name="LauncherInfo">
<attribute name="vmArgs">
-Djava.protocol.handler.pkgs=
com.sun.net.ssl.internal.www.protocol
-Djavax.net.ssl.trustStore={path_to_JRun_lib}¥trustStore
-Djavax.net.ssl.keyStorePassword=<パスワード>
</attribute>
...
```

これらの手順を完了すると、HTTPS を使用してクライアントから Web サービスを呼び出すことができます。次の例は、HTTPS を使用するように設定した Web サービスの **invoke** タグを示しています。**url** 値のプロトコル部分は **https** である必要があります。また、ポート番号は JRun JMC で SSL を設定したときにポート番号として指定したものと一致する必要があります。

```
<web:invoke
namespace="AxisSampleJavaClassService"
url="https://localhost:443/ws/services"
operation="SampleService"
result="myresultInt"
scope="page">
<web:param name="input" value="3"/>
</web:invoke>
```

コマンドラインプログラムなどの JRun 外で動作する Web サービスクライアントを使用するには、Sun の JSSE (Java Secure Socket Extension) を設定する必要があります。JSSE とは、SSL を処理するときに JRun が内部で使用するものです。JSSE の設定手順については、次の URL にある Sun のマニュアルを参照してください。

http://java.sun.com/products/jsse/index.html

第 22 章
SOAP の監視

この章では、クライアントが Web サービスを呼び出したときに生成される SOAP リクエストおよびレスポンスを監視する方法を説明します。

目 次

SOAP リクエストおよびレスポンスの監視

Axis TCPMonitor アプリケーションを使用すると、SOAP エンベロープを含むものを含めて、HTTP リクエストとレスポンスを監視できます。基本的な SOAP リクエストとレスポンスの組み合わせを表示するには、475 ページの「例：SOAP リクエストとレスポンス」を参照してください。

TCPMonitor の Web サービスクライアントを準備するには、クライアントのサービス URL を変更して、ローカルホスト、および tcpmon で設定したリスンポート番号を使用します。次の手順を参照してください。TCPMonitor で、接続をサービスホストに転送するためのターゲットホスト名およびターゲットポート番号を設定します。

次の手順を実行すれば、TCPMon をすばやく使用できます。詳細については、236 ページの第 9 章「TCPMonitor の使用」を参照してください。

TCPMonitor を実行するには

1 コマンドウィンドウを開いて、<JRun のルートディレクトリ >/bin ディレクトリに移動します。

2 次のコマンドを入力します。

```
sniffer
```

プログラムを使用するには

1 [ポート # のリスン] フィールドに、「8080」のように、接続要求を監視するローカルポート番号を入力します。

2 [ターゲットホスト名] フィールドに、接続要求を転送するターゲットホストの名前を入力します。

たとえば、デフォルトの JRun サーバー上で実行中のサービスを監視している場合のホスト名は `localhost` です。

3 [ターゲットポート #] フィールドに、トンネルするターゲットマシンのポート番号を入力します。

たとえば、デフォルトの JRun サーバーで実行中のサービスを監視している場合のデフォルトのポート番号は 8100 です。

4 [追加] をクリックします。

新しくトンネルした接続のタブが表示されます。各 HTTP 接続のリクエストが [リクエスト] パネルに表示され、レスポンスが [レスポンス] パネルに表示されます。TCPMonitor では、すべてのリクエストおよびレスポンスのペアのログが保持され、ユーザーは上部パネルのエントリを選択することによって特定のペアを表示できます。また、エントリを削除したり、結果をファイルに保存して後で表示したりすることもできます。

[再送信] ボタンを選択すると、現在表示しているリクエストが TCPMonitor によって再送され、新しいレスポンスが記録されます。これは、再送する前にリクエストウィンドウ内の XML を編集し、SOAP サーバー上で別の XML の効果をテストできるので、特に便利です。

例：SOAP リクエストとレスポンス

次の例は、指定された利率、元金、月数を元に、月々のローンの支払いを計算する Web サービスオペレーションを呼び出したときに生成される SOAP リクエストとレスポンスを示しています。

SOAP リクエスト

この例は、元金、月数、および利率を元に、ローンの支払いを計算する Web サービスからの実際の SOAP リクエストです。SOAP エンベロープ本文 (SOAP-ENV:Body) には、オペレーション名 (calculate) およびオペレーションパラメータ (principal、months、および rate) が含まれています。

```
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://
        schemas.xmlsoap.org/soap/
encoding/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV=
"http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <calculate>
   <principal xsi:type="xsd:double">10000.0</principal>
   <months xsi:type="xsd:int">12</months>
   <rate xsi:type="xsd:float">0.08</rate>
  </calculate>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP レスポンス

SOAP エンベロープ本文の SOAP レスポンスには結果値が含まれています。

```
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <calculateResponse>
   <calculateResult xsi:type="xsd:double">1326.9497228077948</
        calculateResult>
  </calculateResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

第 23 章
Web サービスデータタイプのマッピング

この章では、Web サービスデータタイプのマッピングについて説明します。

目 次

Axis bean シリアライザによるデータタイプマッピング

Web サービスオペレーションによって基本データタイプ以外のオブジェクトが取得または返送される状況に対応するために、Axis では、標準 JavaBeans パターンの get および set アクセッサに従うすべての Java クラスの直列化または直列化の解除を行うことができます。必要に応じて、この機能をサービスまたはクライアント、あるいはその両方に使用できます。

メモ：WSDL2Java を使用して、複雑なタイプを使用するオペレーションのプロキシまたはスケルトンを生成すると、XML データタイプに対応する Java クラスも生成されます。プロキシ内のコードによって、Axis 内のタイプマッピングが設定されます。スケルトンの場合、タイプマッピングは生成された deploy.wsdd ファイル内に含まれています。

bean マッピングの詳細については、<JRun のルートディレクトリ >/docs/html/thirdparty/axis ディレクトリの『Axis User Guide』を参照してください。

bean マッピングを手動で設定するには

1 Web サービスまたはクライアント、あるいはその両方で bean マッピングが必要かどうかを検証します。

たとえば、ユーザー固有の Axis ベースのサービスを使用する場合、または制御が及ばないサービスのためにクライアントを作成する場合があります。

次のクラスベースの Web サービスでは、`userid` を指定された `getUser` メソッドは `User` オブジェクトを返すので、bean マッピングが必要です。

```
import java.util.ArrayList;
public class UserService {
  private ArrayList users = new ArrayList();
  public UserService() {
    User user1 = new User(0, "John", "Doe");
    User user2 = new User(1, "Jane", "Doe");
    users.add(user1);
    users.add(user2);
  }
    public User getUser(int userid) {
    return (User)users.get(userid);
  }
  public static void main(String args[]) {
    UserService myService = new UserService();
    User myUser = (User)myService.getUser(0);
    System.out.println(myUser.getFirstname());
  }
}
```

2 Web サービスが要求または返送するデータの取得または設定を行うための標準 JavaBeans アクセッサを持つ Java クラスを作成します。

次の JavaBeans は、手順 1 で導入された User を表します。

```
public class User {
  // private 変数
  private int userid;
  private String firstname;
  private String lastname;
  public User() {}
  public User(int userid, String firstname, String lastname) {
    this.userid = userid;
    this.firstname = firstname;
    this.lastname = lastname;
  }
  public int getUserid() {
    return userid;
  }
  public void setUserid(int userid) {
    this.userid = userid;
  }
  public String getFirstname() {
    return firstname;
  }
  public void setFirstname(String firstname) {
    this.firstname = firstname;
  }
  public String getLastname() {
    return lastname;
  }
  public void setLastname(String lastname) {
    this.lastname = lastname;
  }
}
```

3 Web サービスがユーザー独自の Axis ベースのサービスであり、サーバーサイドで JavaBeans を使用している場合は、beanMapping 要素を Web サービスの server-config.wsdd ファイルに追加します。

UserService サービスでは、server-config.wsdd ファイル内の次の beanMapping 要素を使用します。

```
<beanMapping qname="ns:local" xmlns:ns="http://finduser.org/xsd"
languageSpecificType="java:<パッケージ名>.User"/>
```

4 bean を使用するクライアントを作成します。

次の JSP は、UserService Web サービスのクライアントです。このクライアントは、Web サービスタグライブラリの invoke タグを使用してそのサービスの getUser オペレーションを呼び出します。また beanmapping タグを使用して User データタイプを JavaBeans クラスの User にマッピングします。

```
<%@ taglib prefix="web" uri="webservicetag" %>
<%@ page import="<パッケージ名>.User"%>
<web:beanmapping class="<パッケージ名>.User" namespace="http://
     finduser.org/xsd" type="User"/>
<web:invoke
  namespace="http://finduser.org/"
  url="http://localhost:8100/services"
  operation="getUser"
  result="myresult"
  scope="page">
  <web:param name="userid" value="<%= new Integer(1) %>"/>
</web:invoke>
<% User thisUser = (User)pageContext.getAttribute("myresult",
     pageContext.PAGE_SCOPE); %>
<%= thisUser.getFirstname() %> <%= thisUser.getLastname() %>
```

Web サービスタグライブラリの詳細については、448 ページの第 19 章「JSP タグベースのダイナミッククライアントの使用」を参照してください。

パート VII
プログラミングについてのその他のトピック

パート VII では、JRun によるプログラミングのその他のトピックについて説明します。次の章で構成されています。

第 24 章
Flash と JRun の併用

Macromedia JRun 4 に含まれている Flash のネイティブコネクティビティを使用することにより、JRun の開発者は Macromedia Flash MX のデザイナーと協力して、JRun アプリケーション向けのダイナミックな Flash ユーザーインターフェイスを作成できます。この章では、Flash と JRun を併用する方法を説明します。

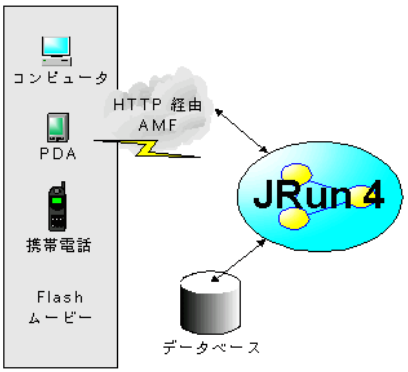
目 次

Flash と JRun の併用について

JRun 4 に含まれている Flash のネイティブコネクティビティを使用することにより、JRun の開発者は Macromedia Flash MX のデザイナーと協力して、Java アプリケーション向けのダイナミックな Flash ユーザーインターフェイス (UI) を作成できます。Flash UI を作成するには、UI コードとビジネスロジックコードを分離する必要があります。Flash MX で UI コントロールを作成し、JRun でビジネスロジックを作成します。

Flash UI と従来の HTML UI では、どちらもインターネット通信に HTTP が使用されています。ただし、HTML ページの通信には通常 GET および POST の HTTP メソッドが使用されますが、JRun と Flash との通信では、HTTP FORM メソッドを通して AMF (Action Message Format) が使用されます。AMF は、JRun と Flash との間に、各種のデータタイプや増分結果セットなどをサポートする、豊富なインターフェイスを提供します。

次の図は JRun と Flash の関係を簡単に示したものです。



JRun アプリケーション向けの Flash UI の開発は、次の 2 つの部分からなります。

1 **サーバーサイドの開発。**詳細については、485 ページの「Flash と JRun の併用」を参照してください。

2 **クライアントサイドの開発。**詳細については、Macromedia の Web サイトにある Flash Remoting ドキュメントを参照してください。

Flash と JRun の併用

JRun で作成した機能は、Java メソッドを作成するだけで Flash に適用できます。JRun は、次の Java アプリケーションタイプと Flash のコネクティビティをサポートします。

- Java クラス (ステートレス)
- JavaBean オブジェクト (ステートフル)
- Enterprise JavaBeans (EJB) のステートレスセッション、ステートフルセッション、および エンティティ bean
- Java Management Extensions (JMX)

Java アプリケーションを Flash で利用するには、クラスファイルを SERVER-INF ディレクトリに保存する必要があります。

次の表に、Java データタイプとそれらに対応する ActionScript をリストします。

Java データタイプ	ActionScript データタイプ
number	number (プリミティブデータタイプ)
boolean	Boolean (プリミティブデータタイプ)
string	string
map	ActionScript (AS) オブジェクト、サービス関数に渡される唯一の引数としての AS オブジェクト、または AS XML オブジェクト
null	null
array	array
date	date オブジェクト

Flash と通信する JavaBean の作成

Flash と通信するには、`flashgateway` パッケージの一部としてファイルを宣言します。public メソッドは、ActionScript 関数として自動的に Flash ムービーで利用できます。

Flash と通信する JavaBean を作成するには

1 Java ファイルを作成し、WEB-INF ディレクトリの samples フォルダに FlashExampleBean.java として保存します。

2 Java コードが次のように保存されるように、ファイルを修正します。

```
// パッケージに名前を付けます。
package flashgateway.samples;

import java.util.Date;
import java.io.Serializable;

// クラスを宣言します。
public class FlashExampleBean
        implements Serializable {
  private String message;
  private int count;
```

```
public FlashExampleBean()
  {
        message = "こんにちは JavaBean です。";
        count = 0;
  }
  // testBoolean メソッドを宣言します。
  public boolean testBoolean(boolean b)
  {
    return b;
  }
  // testDate メソッドを宣言します。
  public Date testDate(Date d)
  {
    return d;
  }
  // setMessage メソッドを宣言します。
  public void setMessage(String message)
  {
    this.message = "はい " + message;
  }
  // getMessage メソッドを宣言します。
  public String getMessage() {
        count++;
        return message + " (count=" + count + ")";
    }
  // getCount メソッドを宣言します。
  public int getCount()
  {
    count++;
    return count;
  }
  //setCount を宣言します。
  public void setCount(int count)
  {
        this.count = count;
    }
}
```

3　ファイルを保存します。

JavaBean のメソッドが FlashExampleBean ActionScript 関数にトランスレートされます。次の表は、JavaBean で利用できるメソッドと、対応する ActionScript の関数を示したものです。

JavaBean のメソッド	ActionScript 関数
testBoolean	FlashExampleBean.testBoolean
testDate	FlashExampleBean.testDate
setMessage	FlashExampleBean.SetMessage
getMessage	FlashExampleBean.getMessage
setCount	FlashExampleBean.setCount
getCount	FlashExampleBean.getCount

ActionScript の考察

ActionScript で JavaBean のメソッドにアクセスする場合は、getService ActionScript 関数に、flashgateway.samples.FlashExampleBean のような完全修飾名を指定します。

フレームレベルで適用された次の ActionScript は、JavaBean のメソッドを呼び出します。

```
#include "NetServices.as"
#include "NetDebug.as"
// ------------------------------------------------
// ユーザーとの対話イベントハンドラ
// ------------------------------------------------
function runExample()
{
  setMessage();
  getMessage();
  testBoolean();
  testDate();
}
function setMessage()
{
  flashtestService.setMessage(messageInput.text);
}
function getMessage()
{
  flashtestService.getMessage();
}
function testBoolean()
{
  if (trueRadio.GetState())
  {
    flashtestService.testBoolean(true);
  }
  else
```

```
  {
    flashtestService.testBoolean(false);
  }
}
function testDate()
{
  flashDate = new Date();
  dateInput.text = "" + flashDate;
  flashtestService.testDate(flashDate);
}
// ------------------------------------------------
// サーバーから受信するデータハンドラ
// ------------------------------------------------
function setMessage_Result( result )
{
}
function getMessage_Result( result )
{
  messageOutput.text = result;
}
function setMessage_Status( result )
{
  messageOutput.text = result.details;
}
function getMessage_Status( result )
{
  messageOutput.text = result.details;
}
function testBoolean_Result(result)
{
  boolOutput.text = "result:" + result;
}
function testBoolean_Status(result)
{
  boolOutput.text = "ステータス：" + result.details;
}
function testDate_Result(result)
{
  flashDate = result;
  dateOutput.text = " " + flashDate;
}
function testDate_Status(result)
{
  dateOutput.text = "ステータス：" + result.details;
}
// ------------------------------------------------
// アプリケーションを起動します。
// ------------------------------------------------
if (inited == null)
{
  // このコードは 1 度だけ実行します。
  inited = true;
```

```
  // デフォルトゲートウェイ URL を設定します（オーサリングで使用するため）。
    NetServices.setDefaultGatewayUrl("http://localhost:8200/
        flashservices/gateway");
  // ゲートウェイに接続します。
    gatewayConnnection = NetServices.createGatewayConnection();
  // 自身の HTTP セッションの JavaBean へのリファレンスを取得します（初めて新規のリ
        ファレンスを作成します）。
    flashtestService =
        gatewayConnnection.getService("flashgateway.samples.FlashJava
        Bean", this);
  flashDate = new Date();
  messageInput.text = "[Enter a Message]";
  dateInput.text = "" + flashDate;
}
```

Flash による EJB アプリケーションへのアクセス

他の EJB とまったく同じように Salsa 用の EJB を作成します。Salsa を通じて示された EJB メソッドにアクセスするには、**getService** 関数で JNDI EJB 名を使用します。次の ActionScript の例では、Flash ムービーでユーザーが行う選択に応じて、2 種類の EJB を呼び出します。

```
function getBiggerString()
{
  if (useStateless.selected)
  {
    flashStatelessEJB.getBiggerString( stringInput.text );
    stringOutput.text = "[ ステートレス EJB を呼び出し中 ...]";
  }
  else
  {
    flashStatefulEJB.getBiggerString( stringInput.text );
    stringOutput.text = "[ ステートフル EJB を呼び出し中 ...]";
  }
}
```

EJB アプリケーションと Flash を併用する場合、ActionScript 関数は EJBHome メソッドと EJBObject メソッドにマッピングされます。

ActionScript の考察

次の ActionScript の例では、EJB メソッドを呼び出し、結果を Flash に表示します。

```
#include "NetServices.as"
#include "NetDebug.as"
// ------------------------------------------------
// ユーザーとの対話イベントハンドラ
// ------------------------------------------------
function runExample()
{
  // 最初に選択された EJB タイプへのリファレンスを取り出します。
  getEJBs();
}
// ------------------------------------------------
// ビジネスメソッド
// ------------------------------------------------
function getBiggerString()
{
  if (useStateless.selected)
  {
    flashStatelessEJB.getBiggerString( stringInput.text );
    stringOutput.text = "[ ステートレス EJB を呼び出し中 ...]";
  }
  else
  {
    flashStatefulEJB.getBiggerString( stringInput.text );
    stringOutput.text = "[ ステートフル EJB を呼び出し中 ...]";
  }
}
function getBiggerInt()
{
  if (useStateless.selected)
  {
    flashStatelessEJB.getBiggerInt( (new
        Number(numInput.getSelectedItem().data)).valueOf() );
    stringOutput.text = "[ ステートレス EJB を呼び出し中 ...]";
  }
  else
  {
    flashStatefulEJB.getBiggerInt( (new
        Number(numInput.getSelectedItem().data)).valueOf() );
    stringOutput.text = "[ ステートフル EJB を呼び出し中 ...]";
  }
}
// ------------------------------------------------
// サーバーから受信するデータハンドラ
// ------------------------------------------------
function getBiggerString_Result ( result )
{
  stringOutput.text = result;
}
```

```
function getBiggerInt_Result ( result )
{
  numOutput.text = result;
}
function create_Result( result )
{
  if (useStateless.selected)
  {
    flashStatelessEJB = result;
  }
  else
  {
    flashStatefulEJB = result;
  }

  getBiggerString();
  getBiggerInt();
}
// このメソッドは、create を呼び出し中にエラーが発生した場合に呼び出されます。
function create_Status( result )
{
  stringOutput.text = result.details;
}
function getEJBs()
{
  // EJB へのリファレンスを取得します。
  // ステートレス：
  if (useStateless.selected)
  {
    flashstatelessHome =
        gatewayConnection.getService("FlashSampleStatelessEJB", this);
    flashstatelessHome.create();
  }
  // ステートフル：
  else
  {
    if (flashstatefulHome == undefined)
    {
    flashstatefulHome =
        gatewayConnection.getService("FlashSampleStatefulEJB", this);
      flashstatefulHome.create( "stateful message" );
    }
    else
    {
      getBiggerString();
      getBiggerInt();
    }
  }

}
```

```
// ------------------------------------------------
// アプリケーションを起動します。
// ------------------------------------------------
if (inited == null)
{
  // このコードは 1 度だけ実行します。
  inited = true;
  // デフォルトゲートウェイ URL を設定します（オーサリングで使用するため）。
    NetServices.setDefaultGatewayUrl("http://localhost:8200/
        flashservices/gateway");
  // ゲートウェイに接続します。
    gatewayConnection = NetServices.createGatewayConnection();
  // カスタムの認証と認可用のセキュリティ情報を設定します。
  //gatewayConnection.setCredentials("Flash", "Flashpass");
  stringInput.text = "[Enter a Word]";
  numInput.setSelectedIndex(0);
}
```

JMX と Flash の併用

Flash を使用して JMX を通して JRun アプリケーションを呼び出すには、MBean メソッドを呼び出します。ActionScript の **getService** 関数で MBean オブジェクト名を使用して、JRun の JMX オブジェクトに接続します。

ActionScript の考察

次の ActionScript の例では、MBean メソッドを呼び出し、結果を Flash に表示します。

```
#include "NetServices.as"
#include "NetDebug.as"
// ------------------------------------------------
// ユーザーとの対話イベントハンドラ
// ------------------------------------------------
function runExample()
{
  jrunDeployerMBean.getServerName();
  jrunDeployerMBean.getEARs();
}
// ------------------------------------------------
// サーバーから受信するデータハンドラ
// ------------------------------------------------
function getServerName_Result(result)
{
  serverName.text = result;
}
function getEARs_Result(result)
{
  numDeployed.text = result.length;
}
function onStatus(result)
{
  serverName.text = "status triggered";
```

```
}
// ------------------------------------------------
// アプリケーションを起動します。
// ------------------------------------------------
if (inited == null)
{
  // このコードは 1 度だけ実行します。
  inited = true;
  // デフォルトゲートウェイ URL を設定します（オーサリングで使用するため）。
    NetServices.setDefaultGatewayUrl("http://localhost:8200/
        flashservices/
        gateway");
  // ゲートウェイに接続します。
    gatewayConnnection = NetServices.createGatewayConnection();
  // リモート MBean へのリファレンスを取得します。
  jrunDeployerMBean =
        gatewayConnnection.getService("DefaultDomain:service=
        DeployerService", this);
}
```

索引

T

X

あ