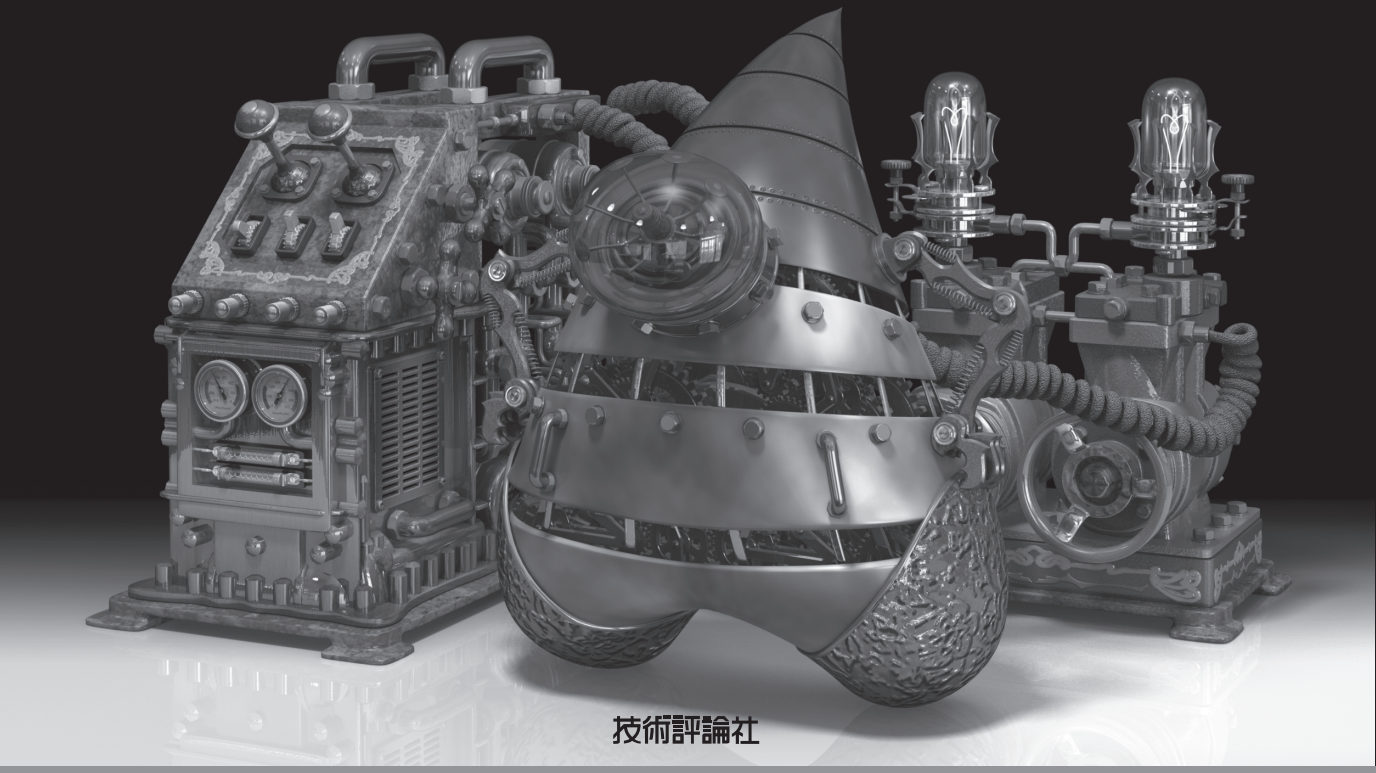
パーフェクト

【改訂2版】

Java

井上 誠一郎／永井 雅人 著



技術評論社

付録A Java EE概論

JavaでWebアプリケーション（以降Webアプリ）開発を行う上で基礎となるJava EEの概要を説明します。本パートのWebアプリ開発の背景知識になります。

A-1 Java EEとは

Java EE（Enterprise Edition）とはサーバプログラム開発のための技術規格を集めたものです。Webアプリ以外の技術規格も含んでいますが、実質上、Webアプリのための技術規格が中心です。

Webアプリ中心の世の中の情勢を受けて、Java EEの中でWebアプリ開発に特化したサブセット規格があります。Webプロファイルと呼びます。

本書執筆時点のJava EEの最新バージョンはJava EE7です。Java EE7のWebプロファイルに含まれる技術規格の一覧を**表A.1**にまとめます。Java EE全体のバージョンと別に、個別規格にも独自バージョンがあるので注意してください。

表A.1　Java EE7のWebプロファイルの技術規格

規格名	バージョン	説明
Java Servlet	3.1	Webアプリの基本API。付録Bで説明します
JavaServer Pages（JSP）	2.3	Webアプリのビュー処理
Standard Tag Library for JavaServer Pages（JSTL）	1.2	JSPファイル内で使う標準カスタムタグ
Expression Language（EL）	3.0	JSPファイル内などで使える簡易言語
Java API for RESTful Web Services(JAX-RS)	2.0	RESTfulなWebアプリ用API。付録Bで説明します
JavaServer Faces（JSF）	2.2	HTMLフォーム処理を中心としたビュー処理
Java API for WebSocket	1.0	WebSocketを扱うAPI
Contexts and Dependency Injection for Java（CDI）	1.1	DI（Dependency Injection）によるコンポーネント化
Dependency Injection for Java	1.0	（CDIで利用する）DIのためのアノテーション規格
Common Annotations for the Java Platform	1.2	共通的なアノテーション規格
Java Persistence（JPA）	2.1	データベースアクセスのORM（Object-relational mapping）
Java Transaction API(JTA)	1.2	データベースのトランザクション処理API
Enterprise JavaBeans（EJB Lite）	3.2	トランザクション処理、並行処理などをコンポーネント化
Bean Validation	1.1	バリデーション処理のAPI
Java API for JSON Processing	1.0	JSONデータを読み書きするAPI

A-1-1　規格と実装

Java EE自体が決めているのは規格のみです。その規格に沿った実装を開発するのは自由です。**表A.1**の個々の規格に対する実装が、商用、オープンソース両方でたくさん存在します。Java EE全体の規格に沿った実装も複数存在します。Java EE全体の規格に沿った実装を一般にJava EEコンテナと呼びます。実行時の役割で呼ぶ場合、アプリケーションサーバとも呼びます。アプリケーションサーバについては後ほど説明します。

A-2　WebアプリとJava EE

A-2-1　Webアプリの構造

本書は読者がHTTPの動作を理解していると想定しています。リクエスト、レスポンス、URLなどの用語を説明なしに使います。必要であれば、HTTPについて別の書籍を参照してください。

Webアプリの動作を概観すると、HTTPリクエストを入力としてHTTPレスポンスを出力するプログラムです。Webアプリを1つのクラスのように見立てると、リクエストURLはメソッド名に相当し、レスポンスが返り値に相当します。Javaのメソッドと HTTPのメソッドを混乱しないために、本パートでは、HTTPのメソッドはGETメソッドやPOSTメソッドのようにすべて大文字で表記します。

A-2-2　Java EEの歴史と概要

シンプルなWebアプリはJava EEがなくても開発可能です。ネットワークプログラミングをすれば、JavaでHTTPサーバを開発可能だからです。

誰もが書くHTTPサーバ処理はフレームワークとして共通化するほうが幸せです。こうしてできた標準規格がサーブレットです。その後、JSPが登場して、サーブレットを補強しました。これらを基礎として発展してきたのが今のJava EEです。

サーブレットとJSP周辺技術（JSTLとEL）は、Java EEの中でもシンプルで低レイヤの規格です。HTTPサーバ機能とこれらの規格を実装したサーバをサーブレットコンテナと呼びます。Java EE規格すべてを実装したJava EEコンテナより小さくて軽い特徴があります（コラム参照）。

サーブレット以後、Java EEの規格が徐々に増えてきました。**表A.1**の中で特筆すべき規格がEJBとCDIです。

EJBはかつてJava EEの中心規格でした。EJBはリモートアクセス可能なEJBコンテナを配置して分散処理を可能にします。EJBコンテナは背後のデータベースアクセス処理、トランザクション処理、並行処理、非同期処理などの複雑な処理をアプリ開発者から隠蔽します。

現在のJava EEでは、EJBの機能が他の規格に徐々に委譲されています。たとえば、データベースアクセス処理はJPA、トランザクション管理はJTA、という具合です。またWeb中心の世界では、並列Webアプリで分散化するのが主流なので、EJBコンテナでの分散処理は徐々に廃れています。このような背景の下、Webプロファイルにはリモート分散処理部分抜きのEJB Liteが残っています。

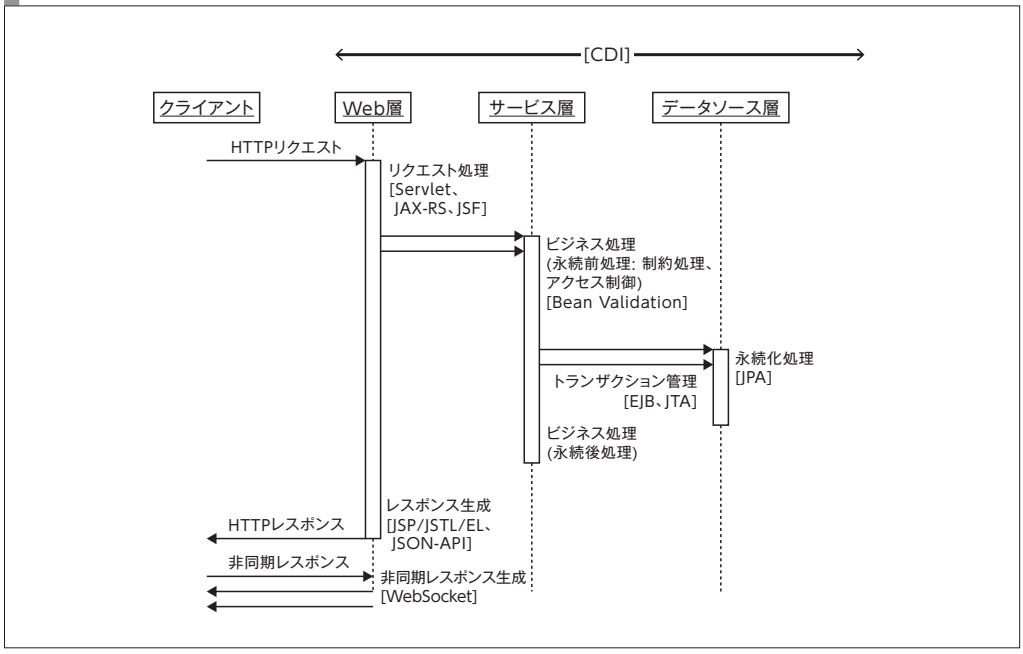
EJBに代わりJava EEの中心規格になりつつあるのがCDIです。後述しますが、Java EE全体はコンテナアーキテクチャで徹底されています。しかし徹底しているのは思想だけで手法は異なります。つまり、それぞれの規格がそれぞれにコンテナとして振る舞い、オブジェクト管理をしてきたのがJava EEの歴史でした。

CDIはばらばらのオブジェクト管理を統一して、Java EEに一貫したコンテナアーキテクチャを提供します。

■Java EEの役割

表A.1だけでは各規格の役割がわかりづらいので、一般的なWebアプリの動作シーケンスとの関連を示します（**図A.1**）。

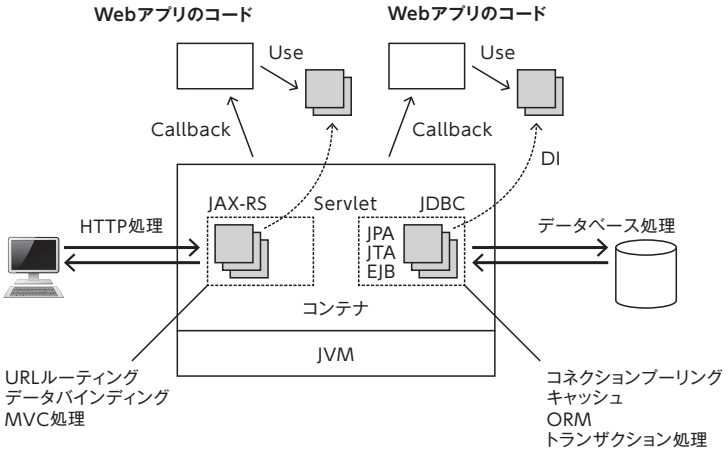
図A.1　Webアプリのシーケンスと Java EE



A-2-3　コンテナアーキテクチャ

Java EEを理解する上で重要な概念がコンテナアーキテクチャです。**図A.2**の「Webアプリのコード」の部分がアプリ開発者の書くコードで、それ以外はコンテナのコードだと考えてください。アプリ開発者の書くコードは、コンテナから呼び出されます（コールバック）。そして、コンテナ提供機能を利用するコードになります。

図A.2　コンテナとアプリの関係の図



コンテナアーキテクチャを実行環境の視点とプログラム開発の視点の両方から説明します。

普通のJavaプログラムの場合、生成したクラスファイル群をjavaコマンドで実行します。一方、

C O L U M N

Spring FrameworkとJava EE

本文でサーブレットコンテナは小さくて軽いと説明しました。しかしサーブレットコンテナだけでは大規模Webアプリ開発が不便なのも事実です。このため、開発を支えるいくつかのWebフレームワークが存在します。もっとも代表的なWebフレームワークはSpring Framework（以下Spring）です。SpringはJava EEに大きな影響を与えたフレームワークで、表A.1とほぼ同じ機能を独自に持ちます。

紛らわしいことに、Springはすべてが独自路線ではなく、Java EEのAPI規格を準拠する部分も多々あります。基本アーキテクチャはSpringとJava EEはほぼ同じなので、APIが同じになるとほとんど違いがなくなりつつあります。

Java EEアプリの場合、javaコマンドで実行するプログラムの実体はコンテナです。開発者の作ったWebアプリのクラスファイル群は、コンテナがロードして実行します。コンテナはWebアプリに共通的な機能を提供します。共通処理の1つがHTTP処理なので、通常、コンテナはHTTPサーバとして機能します。

プログラム開発の視点で見ると、オブジェクト生成の責務をコンテナに任せるアーキテクチャになります。Webアプリはコンテナがライフサイクル管理するオブジェクトを受け取り、それらを利用してコンテナの機能を活用します。

コンテナの説明から、フレームワーク、JavaBeans、DI（本誌「**18章　リフレクション**」参照）を思い起こす人もいるでしょう。これら根底に流れる思想は似ているからです。これは偶然ではなく、大規模ソフトウェア開発のアーキテクチャの一定の進化の方向性を示唆しています。

次にコンテナアーキテクチャにとって重要な2つの概念を紹介します。コンテナ管理のオブジェクトをどう探すか（JNDIとDI）とどう整理するか（コンテキストとスコープ）です。

■JNDIとDI

コンテナが管理するオブジェクトをWebアプリから利用する手段が必要です。主に2つの手段があります。

1つはJNDI（Java Naming and Directory Interface）です。Java EEの世界ではJNDIで管理する対象物をリソースと呼びます(注1)。コンテナは名前付きでリソースを管理します。アプリ開発者はJNDI名でリソースを引けば利用できます。JNDIのAPIは標準ライブラリのjavax.namingパッケージで提供されます。

最近のJava EEではアノテーションAPIの利用が広がっています。コンテナ機能を利用したい場合、自作クラスのフィールド変数やメソッドの引数にアノテーションを付与します。こうすると、実行時にコンテナが必要なオブジェクトを生成して、アノテーションを付与された変数にオブジェクト参照を代入してくれます。技法としてはDIと呼びます。コンテナ管理のオブジェクト（Managed Beanと呼ぶ場合もあります）をWebアプリのコードから簡易に利用できます。

JNDIの利用にもjavax.annotation.Resourceアノテーションを利用可能です。アノテーションを使うと、JNDI利用コードは、ほとんどDIと区別がつかなくなります(注2)。

■ コンテキストとスコープ

コンテナはたとえるとオブジェクトの貯蔵庫です。実装パターンの用語を使うとレジストリやレポジトリなどと呼びます。大規模なアプリケーションで有効なアーキテクチャです。

ただ、大きな単一の貯蔵庫のままでは、管理オブジェクトが増えると整理しづらくなります。下手をすると、形を変えたグローバル変数になりかねません。

（注1）「リソース」は多義的な用語なので注意してください。本書だけでもtry-with-resources文のリソース、JNDIのリソース、JAX-RSのリソースという用語を使います。それぞれ別のコンテキストなので別の意味になります。

（注2）Resourceアノテーション利用例は「**付録D　データベース**」を参照してください。

管理オブジェクトを整理する仕組みとして、寿命と可視性で分類するスコープと呼ぶ仕組みがJava EEにあります。CDIの用語を使うと、スコープ管理されたオブジェクトをコンテキストオブジェクトと呼びます。紛らわしいことに、サーブレットの世界のスコープとCDIの世界のスコープは別々に定義されていて、用語の定義が異なります。それぞれの詳細は次章以降で説明しますが、どちらのスコープも、管理オブジェクトを分類する仕組みである点は共通しています。

A-2-4　アプリケーションサーバ

代表的なアプリケーションサーバを**表A.2**にまとめます。

表A.2　代表的なWebアプリケーションサーバ

名称	配布元	説明
Tomcat	Apache Software Foundation（ASF）	代表的なオープンソースのサーブレットコンテナ
Apache TomEE	Apache Software Foundation（ASF）	Tomcatを使うJava EEコンテナ
Jetty	Mort Bay Consulting	オープンソースのサーブレットコンテナ。他のアプリケーションサーバの1コンポーネントになっていることも多い
GlassFish	Oracle	オープンソースのJava EEコンテナ。 Java EEの参照実装
WebLogic	Oracle	商用のJava EEコンテナ
JBoss	RedHat	商用のJava EEコンテナ
WildFly	RedHat	JBossのオープンソース版
WebSphere	IBM	商用のJava EEコンテナ
Resin	Caucho	オープンソースのサーブレットコンテナ

■ GlassFish

本書はJava EEコンテナとしてGlassFish4を使います。ただしWebアプリを開発して動かすまでの最小限の説明に限定します。またインストール方法も説明しません。GlassFish4の詳細について知りたい場合は別の書籍を参照してください。

サーブレットおよびJSPはJava EEのサブセットなので、Java EEコンテナは必然的にサーブレットコンテナになります。サーブレットに説明を限定する場合はサーブレットコンテナ、他のJava EEの機能を使う場合はJava EEコンテナと用語を使い分けます。とは言え、実体としてのGlassFishは同じものなので、使う機能だけでの区別です。

付録B サーブレットとJAX-RS

Java EEでWebアプリを作るシンプルな規格がサーブレットAPIです。サーブレットAPIの使い方を通じてWebアプリの構造を理解してください。サーブレットAPIより新しい規格がJAX-RSです。JAX-RSを使うと、より高度なWebアプリをより簡単に作成可能です。サーブレットAPIと比較しながらJAX-RSの理解を進めてください。

B-1 Webアプリ開発の準備

B-1-1 GlassFish4の準備

GlassFish4（以下GlassFish）をインストールしたディレクトリを「$GLASSFISH」と表記します。たとえば「/opt/glassfish4/bin/asadmin」の代わりに「$GLASSFISH/bin/asadmin」と表記します。

GlassFishを動かす上で最低限知っておくべきコマンドはasadminです。第1引数でサブコマンドを指定します。次の2つはGlassFishの起動と停止のサブコマンド例です。

```
$ $GLASSFISH/bin/asadmin start-domain
$ $GLASSFISH/bin/asadmin stop-domain
```

他のサブコマンドは次のように調べられます。

```
$ $GLASSFISH/bin/asadmin list-commands
```

asadminコマンドでGlassFishの各種管理が可能です。コマンドラインツールではなく、対話的に管理できる管理コンソールも利用可能です。デフォルト設定では、Webブラウザで「http://localhost:4848/」にアクセスすると管理コンソール画面にアクセスできます。別のPCからアクセスする場合は、ホスト名（localhost）の部分を読み替えてアクセスしてください。以下、類似のアクセスURLでも事情は同じです。

B-1-2 Webアプリ開発の構成管理

現場の開発の構成管理は大掛かりになる傾向にあります（コラム参照）。多人数で開発を行うにはきちんとした構成管理が必要だからです。ただ、Webアプリ開発の学習に限れば簡易な構成管理で充分です。このため、本書は必要最小限の構成管理を紹介します。

構成管理に必要な最小工程は「ビルド」と「デプロイ」の2つです。

ビルドとはソースファイルをコンパイルする工程です。Javaで言えばソースファイルからクラスファイルを生成する工程です。最近のWebアプリの場合、JavaScriptやCSSファイルの変換や圧縮なども含みます。

ビルドの次工程がデプロイ（配備）です。デプロイとは、作成したプログラムをコンテナがロードできるようにする工程です。デプロイ対象ファイルはクラスファイルだけではありません。Webアプリには画像ファイル、HTMLファイル、CSSファイルなど、実行ファイル以外の構成要素があるからです。

現実の製品開発では、デプロイすべきファイル群をアーカイブ（1つのファイルにまとめる）するのが普通です。アーカイブするとファイルの扱いが簡単になるからです。またファイル数が減るのでファイルのロードが速くなる利点もあります。本章はwarと言うアーカイブ形式を使います。

原始的なデプロイ手法はファイルコピーです。コンテナがロードできる所定のディレクトリにファイルをコピーしてコンテナを起動すればロード可能だからです。コンテナによっては管理ツールによるデプロイ手法も提供します。GlassFishのデプロイ方法は後述します。

B-1-3　自作WebアプリをGlassFishで動かす工程

本章は下記の手順でWebアプリを開発します。

① mvnコマンドで開発ソースツリーの雛形を作成
② ソースコードを記述
③ mvnコマンドでビルド
④ asadminコマンドでデプロイ
⑤ 動作確認

C O L U M N

構成管理とは

構成管理とはプログラムを書いて実際に動かすまでの一連の作業の管理と考えてください。

広義には、製品出荷までの一連の作業（ソースコード管理、バグ管理、プロジェクト管理など）、および出荷後の保守運用まで含めた作業管理を指しますが、本章の場合、プログラムを書いてGlassFish上で動かすまでの作業に限定します。

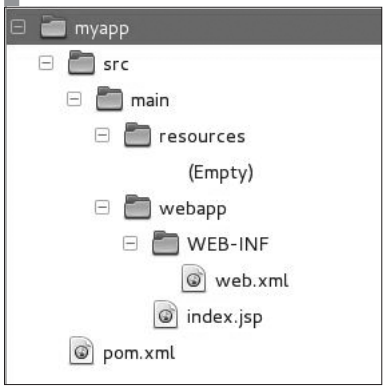
■ mvnコマンドで開発ソースツリーの雛形を作成

ビルドのためにmaven(注1)というツールを使います。紙幅の都合でmavenの詳細は説明しません。本書を読む上では、mavenのコマンド名がmvn、設定ファイルがpom.xmlの2点のみを覚えておけば充分です。

次のようにmvnコマンドを入力していくつかの質問に答えると開発ソースツリーの雛形を生成できます（**図B.1**）。

```
$ mvn archetype:generate -DarchetypeArtifactId=maven-archetype-webapp
```

図B.1　mvnが生成する開発ソースツリー雛形



質問に答える代わりに次のようにすべてをコマンドライン引数でも指定できます。

```
$ mvn archetype:generate -DgroupId=com.app -DartifactId=myapp -DarchetypeArtifactId=maven-
archetype-webapp -Dversion=1.0-SNAPSHOT -DinteractiveMode=false
```

上記のmyappの部分には任意のWebアプリ名、com.appの部分には任意のグループ名を指定してください。2つを合わせたものが（mavenの）プロジェクト名になります。以降、mavenにほとんど依存しない説明をするので、プロジェクト名は意識せずWebアプリ名だけを意識すれば充分です。

■ ソースコードを記述

mvnコマンドの-DarchetypeArtifactId=maven-archetype-webapp引数で開発ソースツリーの雛形が決まります。本章で作りたいWebアプリにぴったりの雛形ではないので微調整が必要です。

（注1）　http://maven.apache.org

自動で生成されたファイルのうち次の2つのファイルを手動で修正してください。開発ソースツリーのトップディレクトリを $WEBAPP と表記しています。書き換えの意味はコメントを参照してください。

- $WEBAPP/pom.xmlの書き換え（リストB.1とリストB.2）
- $WEBAPP/src/main/webapp/WEB-INF/web.xmlの書き換え（リストB.3）。もしくはweb.xmlを削除

リストB.1　pom.xmlの追記項目（既存<dependencies>内に<dependency>要素を追記）

```
<dependencies>
  <!-- Java EE（Webプロファイル）を使うための設定を追記 -->
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId> <!-- (注2) -->
    <version>7.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

リストB.2　pom.xmlの追記項目（既存<build>内に<plugins>要素を追記）

```
<build>
  <finalName>myapp</finalName>   <!-- Webアプリ名（mvnにより自動生成済みの行）-->
  <!-- 以下を追記 -->
  <plugins>
    <plugin>
      <!-- デフォルトのJavaコンパイラバージョンが5なので、7に変更(注3) -->
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
    <plugin>
      <!-- web.xmlなしを許可する設定（web.xmlの存在意義は後述します）-->
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.4</version>
      <configuration>
```

（注2）　Webプロファイルではないjava EEを使う場合、javaee-web-apiの代わりにjavaee-apiにしてください。javaee-web-apiはjavaee-apiのサブセットです。

（注3）　本書執筆時点のGlassFish4の安定版に対応するJava言語のバージョンは、Java8ではなくJava7です。

```
        <failOnMissingWebXml>false</failOnMissingWebXml>
      </configuration>
    </plugin>
  </plugins>
</build>
```

リストB.3　web.xmlの書き換え（mvnコマンド生成のweb.xmlは古いサーブレットAPI用なので）

```
<!-- サーブレットAPI 3.1を使うために全体を書き換え -->
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/
javaee/web-app_3_1.xsd"
         version="3.1">
</web-app>
```

■ mvnコマンドでビルド

開発ソースツリーのトップディレクトリで次のコマンドを実行するとビルドできます。Webアプリの名前がmyappであればtargetディレクトリの直下にmyapp.warファイルができます。

```
$ mvn package
```

前述の開発ソースツリー自動生成直後にビルド可能です。デプロイと動作確認も可能なので、何かソースコードを書き始める前に動作確認してください(注4)。

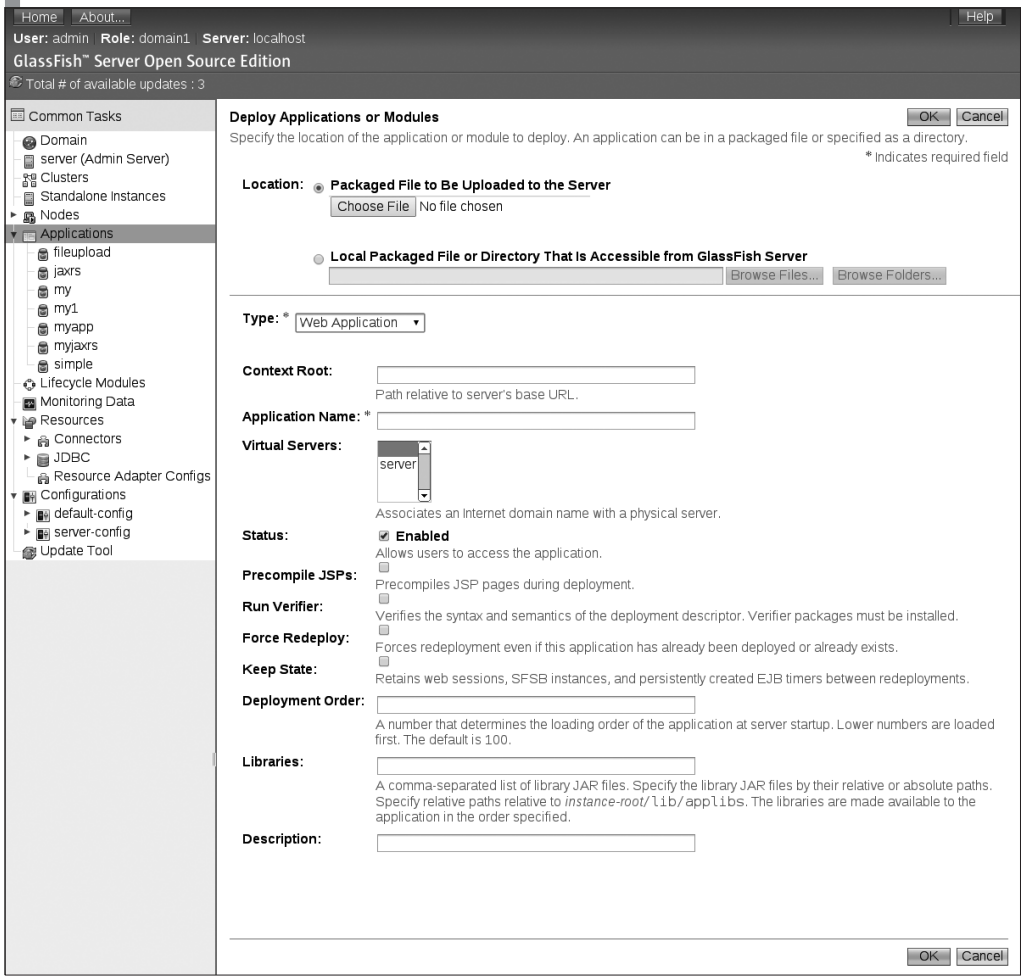
■ デプロイと動作確認

GlassFishの場合、主に次の3つのデプロイ方法があります。

- GlassFishの管理コンソール画面からデプロイ（図B.2）
- warファイルを手動コピー（コピー先ディレクトリは $GLASSFISH/glassfish/domains/domain1/autodeploy/）
- asadminコマンドを使うデプロイ

（注4）　ソフトウェア開発はツールやフレームワークが増え環境が複雑化する傾向にあります。動作確認をして土台に問題がないことを確認してから次に進む癖をつけておくと、はまる可能性を減らせます。

図B.2　GlassFishの管理コンソール



本章はasadminを使います。asadminコマンドで次のようにデプロイ可能です。引数にビルドで生成したwarファイルを指定してください。

```
$ $GLASSFISH/bin/asadmin deploy --force=true target/myapp.war
```

デプロイされているかは次のコマンドで確認可能です。

```
$ $GLASSFISH/bin/asadmin list-applications
```

■ 動作確認

Webブラウザで以下のURLにアクセスすると、デプロイしたWebアプリ（アプリ名がmyappの場合）にアクセスできます。

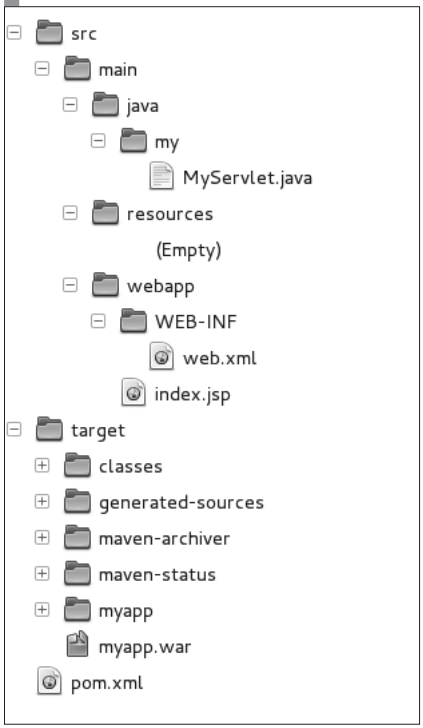
```
http://localhost:8080/myapp/
```

mvnで自動生成したデフォルト状態ではこのURLにアクセスすると「$WEBAPP/src/main/webapp/index.jsp」を表示する設定になっています。気になる場合は、index.jspファイルを変更して、ビルド、デプロイ、動作確認をしてみてください。

B-1-4　簡単なサーブレットアプリ

サーブレットAPIを使うWebアプリをサーブレットアプリと呼びます。簡単なサーブレットアプリの例を示します（**リストB.4**と**図B.3**）。**リストB.3**のweb.xmlの書き換えをしないとうまく動かないので注意してください。古いweb.xmlが@WebServletというアノテーションを認識できないからです。後述しますが書き換える代わりに、web.xmlファイルを削除しても動作します。

図B.3　ファイルの配置場所



リストB.4　サーブレットアプリの例

```
package my; // パッケージ名は任意

// 紙幅の節約のため、後述のコードから類似のimport文の記述を省略していきます
import java.io.*;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;

@WebServlet("/my")
public class MyServlet extends HttpServlet { // クラス名は任意
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        PrintWriter out = resp.getWriter();
        out.print("<html><head><title>hello servlet</title></head>");
        out.print("<body><p>hello, servlet</p></body></html>");
    }
}
```

Webブラウザから以下のURLにアクセスして hello, servlet の画面が現れれば成功です（ビルドとデプロイを忘れないでください）。

http://localhost:8080/myapp/my

■ サーブレットクラスの読解

リストB.4のように、HttpServletクラスを拡張継承した自作クラスをサーブレットクラスと呼びます。最小のWebアプリはサーブレットクラス1つあれば開発できます。

サーブレットクラスを直感的に説明すると、あるリクエストURLに対するHTTPアクセスを受けた場合にサーブレットコンテナから呼ばれるエントリポイントのクラスです。@WebServletアノテーションがリクエストURLとサーブレットクラスを結びつけます。

doGetメソッドは基底クラスのHttpServletクラスが持つメソッドです。HTTPのGETメソッドに対する処理を記述するエントリポイントです。自作サーブレットクラスでdoGetメソッドをオーバーライドして独自処理を記述できます。

doGetメソッドにはHttpServletRequestとHttpServletResponseの2つの引数オブジェクトが渡ってきます。前者がHTTPリクエストを表現するオブジェクト、後者がHTTPレスポンスを表現するオブジェクトです。直感的には前者を入力、後者を出力と考えてください。どちらのオブジェクトも、オブジェクト生成の責務はサーブレットコンテナです。

リストB.4は、HttpServletResponseオブジェクトから出力ストリームを取得してHTTPレスポンスのHTML文字列を直接書き込んでいます。現実のWebアプリはここまで単純ではありませんが、最小のサーブレットクラスとしてはこれで動作します。

■ リクエストURLの構造

http://localhost:8080/myapp/my を分解してみます。localhostがホスト名で8080がポート番号です。これが1つのコンテナ（OSレベルでのJavaプロセス）に対応すると考えてください。myapp がWebアプリ名に相当します。1つのコンテナ上で複数のWebアプリが稼働可能なので、Webアプリ名はそれらを区別する識別子と考えてください。

リクエストURLのWebアプリ名相当の部分をコンテキストパスと呼びます。コンテキストパス以降はWebアプリ内でのパスです。http://localhost:8080/myapp/my の my 以降のパスです。Webアプリ内のパスとサーブレットクラスの対応づけを次に説明します。

B-1-5　URLマッピング

特定のリクエストURLとサーブレットクラスを関連づける必要があります。Webブラウザから該当URLにアクセスを受けたコンテナは、そのサーブレットクラスのメソッドに処理を委譲します。このように、リクエストURLから処理サーブレットクラスを決めることをURLマッピングやURLルーティングと呼びます。

URLマッピングは次のいずれかで設定可能です。

- アノテーション（@WebServlet）
- web.xmlの<servlet-mapping>
- プログラマブルAPI（ServletContextクラスのaddServletメソッド）

本書はアノテーションによる方法とweb.xmlによる方法を説明します。

■ @WebServletアノテーション

@WebServletアノテーションをサーブレットクラスに付与すると、そのサーブレットクラスを特定のリクエストURLに関連付けられます。@WebServletのvalue要素もしくはurlPatterns要素でURLを指定します。URLは複数指定可能です。後述するようにワイルドカード的なURL指定も可能です。

value要素とurlPatterns要素の意味は同じなので使い分けの基準は開発者次第です。他のアノテーション要素がなく要素名を省略可能な場合はvalue要素を使い、他の要素と併記する場合は意味がわかりやすいurlPatterns要素を使うのが慣例です。下記に例を示します。

```
@WebServlet("/")                                  // value要素でURLマッピング
@WebServlet({"/foo", "/bar"})                     // 複数のURLを指定可能
@WebServlet(urlPatterns="/")                      // urlPatterns要素でもURLマッピング可能
@WebServlet(urlPatterns="/", name="my")           // name要素でサーブレット名を指定可能
@WebServlet(urlPatterns={"/foo", "bar"}, asyncSupported=true)  // 他のアノテーション要素と併記する
場合、urlPatterns要素を推奨
```

■ web.xmlを使うURLマッピング

web.xmlでURLマッピングする例を**リストB.5**に示します。

クラスとURLの関連づけは2段階になります。まずweb.xml内の<servlet>要素で、完全修飾名で記述したサーブレットクラスにサーブレット名を対応づけます。次に<servlet-mapping>要素でサーブレット名とURLパターンを対応づけます。サーブレット名はweb.xml内で一意であれば任意につけられる名前です。

リストB.5　web.xmlの例

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/
javaee/web-app_3_1.xsd"
         version="3.1">
  <servlet>  <!-- my.MyServletクラスにmyServletという名前をつける -->
    <servlet-name>myServlet</servlet-name>
    <servlet-class>my.MyServlet</servlet-class>
  </servlet>
  <servlet-mapping> <!-- myServletと/myというURLを関連づける -->
    <servlet-name>myServlet</servlet-name>
    <url-pattern>/my</url-pattern>
  </servlet-mapping>
</web-app>
```

■ URLパターンの文法

@WebServletおよびweb.xmlに記述できるURLのパターンの文法を**表B.1**に示します。URLの記述はcase sensitive（大文字小文字を区別）です。

表B.1　URLパターンの文法

記述	説明	例
パス	完全一致パスのルール	/my
/*で終端するパス	前方一致パスのルール	/my/*
*.拡張子	拡張子によるマッピングルール	*.jsp
/の1文字	デフォルトルール	/

URLにマッチするサーブレットクラスは次の優先順で決まります。

① 完全一致
② 拡張子ルール一致
③ 最長の前方一致
④ デフォルト
⑤ コンテナ（GlassFishなど）のデフォルト動作

■ GlassFishのデフォルトURLマッピング

GlassFishは下記のデフォルトURLマッピングの規則を持ちます。

- **ファイルの拡張子が.jspのURLの場合、JSPサーブレットに処理を委譲**
- **拡張子が.jsp以外であれば、URLのマッピングパスをファイルシステムのファイルにマッピングして、ファイルの中身をそのままHTTPレスポンスにする**
- **マッピング先がファイルシステムのディレクトリにマッピングされる場合、ディレクトリ内のindex.html、index.htm、index.jspにマッピングする**

対応先ファイルシステムのルートディレクトリは $WEBAPP/src/main/webapp/ ディレクトリです。上記動作を確認するにはこのディレクトリ下にファイルを配置して、ビルドとデプロイをしてください。

mvnコマンドで開発ソースツリーの雛形を生成直後、特別なURLマッピングの指定をしなくても「http://localhost:8080/myapp」にアクセスしてindex.jspが表示される理由は、上記規則で説明できます。

■ WEB-INFとMETA-INFディレクトリ

WEB-INFとMETA-INFの2つのディレクトリの下のファイルは、URLのパターンがマッチしてもファイルの中身を返さない決まりになっています。一般にWEB-INFの下には、web.xmlファイル、クラスファイル（jarファイル）などを配置します。META-INFディレクトリの下にはコンテナ固有の設定ファイルなどを配置します。これら設定ファイルをWebブラウザから見られたり、クラスファイルをダウンロードされない仕組みになっています。

逆に言うと、この2つのディレクトリ以外に配置したファイルは、設定次第でダウンロードできる可能性があります。設定に気を使うよりも仕組みに依存して安全さを確保してください。

■ web.xmlファイル

web.xmlファイルはサーブレットの規格で決まっている設定ファイルです(注5)。URLマッピン

(注5)　web.xmlの肥大化を防ぐため、web-fragment.xmlに分割可能です。

グなどを設定します。

昔のサーブレットではweb.xmlファイルの存在が必須でしたが、サーブレット3.0で@WebServletアノテーションが導入され、ファイルの存在が必須ではなくなりました。

B-2 サーブレット

B-2-1 サーブレットAPIの概要

サーブレットAPIを提供するパッケージは次の2つです。

- javax.servlet
- javax.servlet.http

javax.servletパッケージはHTTPに依存しないクラスやインターフェースを提供します。javax.servlet.httpパッケージはHTTPに依存したクラスやインターフェースを提供します。ただ、普通にWebアプリを開発する際はあまり違いを気にする必要はありません。

B-2-2 サーブレットクラス

サーブレットクラスとは、HttpServlet抽象基底クラスを拡張継承した具象クラスです。Webアプリ開発者が自作のサーブレットクラスを作成します。**リストB.4**で実例を見ました。フレームワークが抽象基底クラスを用意し、開発者が拡張継承して具象クラスを作成する技法は多くのフレームワークで一般的な構造です。

説明のため、以降、サーブレットクラスのオブジェクトを「サーブレットオブジェクト」と呼ぶことにします。

サーブレットオブジェクトの生成はサーブレットコンテナ（フレームワーク）の役割です。Webアプリ内でサーブレットオブジェクトの生成を明示的に行うのは、（たとえできたとしても）禁止です。サーブレットオブジェクトの生成はサーブレットコンテナの専売特許と心得てください。

このように、コンテナにオブジェクト（インスタンス）生成を丸投げする感覚は重要なので覚えておいてください。この感覚はサーブレットのみならず、JAX-RSでもCDIでも（本書では深く扱いませんが）JPAでも重要です。

■ doメソッドのオーバーライド

開発者は、自作のサーブレットクラス内でdoGetやdoPostのように接頭辞doのついた名前のメソッドをオーバーライドします。本書では便宜上これらをdoメソッドと呼びます。

doメソッドはHTTPのメソッドに対応しています。リクエストがGETメソッドであればdoGetメソッドが呼ばれる関係です。本書ではdoGetとdoPostのメソッドのみを使います。

```
// HttpServletのdoメソッド（一部）
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException
```

サーブレットコンテナはサーブレットオブジェクトに対してdoメソッドを呼びます。開発者が具象クラスでdoメソッドをオーバーライドしていれば、サーブレットコンテナがそのメソッドをコールバックする関係になります。

doメソッドをオーバーライドしていない場合、該当HTTPメソッドをサポートしていない旨のエラーをWebブラウザに返します。つまりHTTPエラーを返す実装が抽象基底クラスのデフォルト実装です。適切な実装でオーバーライドして必要なHTTPメソッドをサポートしてください。

doメソッドのパラメータ引数はHttpServletRequestオブジェクトとHttpServletResponseオブジェクトです。この2つのオブジェクト生成もコンテナが行います。doメソッドに実装すべき基本動作はHttpServletRequestオブジェクトからリクエスト情報を読み取り、HttpServletResponseオブジェクトにレスポンス情報を書き出すことです。

後述する適切なMVCアーキテクチャの下ではレスポンス生成処理をJSPなどのビュー処理として分離するのが普通です。この流儀に従うdoメソッドは、HttpServletRequestオブジェクトからリクエスト情報を読み取り、JSPなどのビュー処理にレスポンス処理を丸投げ（委譲）する構造になります。サーブレットの世界では処理の丸投げを「フォワード」と呼びます。フォワード処理は後ほど説明します。

■ initメソッドのオーバーライド

doメソッド以外にオーバーライドすることのあるメソッドはinitメソッドです。サーブレットコンテナはサーブレットオブジェクト生成後にinitメソッドをただ1度呼びます。各サーブレットオブジェクトはただひとつしか生成されないので、initメソッドをオーバーライドすると起動時に1度だけ行いたい処理を書けます。

■ サーブレットクラスのインスタンスと同期処理

サーブレットコンテナは、サーブレットクラスごとのオブジェクトをただ1つだけ生成します（複数オブジェクトを生成するモードも可能ですが非推奨です）。

サーブレットコンテナ（APサーバ）は、同時に複数のクライアントから接続を受けます。これら並行リクエスト処理に別々のスレッドを割り当てます。つまり、複数のWebブラウザが同じURLに同時にアクセスすると、サーバ上では、複数のスレッドが同時に同じサーブレットオブジェ

クトのdoメソッドを呼び出します。仮にサーブレットクラスにインスタンスフィールドがある場合、同期処理（排他制御）が必要になります。

仮にインスタンスフィールドがあれば、と説明しました。実際には、原則、サーブレットクラスにインスタンスフィールドを持たせるべきではありません。代わりにリクエストごとに持つべき状態はHttpServletRequestオブジェクトに持たせます。またセッションごと（直感的にはログインユーザごと）に持つべき状態はHttpSessionオブジェクトに持たせます。状態の持たせ方については後述します。

doメソッドの引数に渡ってくるHttpServletRequestオブジェクトとHttpServletResponseオブジェクトはリクエストごとに独立して生成されます（これらの生成の責務もコンテナです）。この2つのオブジェクトは他のスレッドと共有しません（してはいけません）。このため、仮にこれらにインスタンスフィールドがあっても同期処理は不要です。可能な限りこれら2つのオブジェクトに状態を寄せると、サーブレットアプリの並行処理の複雑さを回避できます。

上記指針を守ると、同期処理が必要になるのはHttpSessionオブジェクトおよびアプリケーション全体で共有するキャッシュなどに限定できます。複雑な並行処理を統制するには、同期処理が必要な部分とそうでない部分を意識的に分離する必要があります。並行処理は本質的に難しいので、指針を守って少しでも複雑さを減らしてください

B-2-3　リクエスト処理

HttpServletRequestオブジェクトからHTTPリクエストの情報を取得する処理をリクエスト処理と呼びます。HTTPリクエストから得られる主な情報と取得メソッドを**表B.2**に示します。

表B.2　リクエストに対する操作

リクエストの構成要素	対応メソッド
リクエストURL	getRequestURLなど
クエリパラメータ	getParameterやgetParameterValuesなど
リクエストボディ（ポストデータ）	getInputStreamやgetReaderなど
リクエストヘッダ	getHeaderやgetHeadersなど

■ リクエストURL

次のリクエストURLを見てください。

```
http://localhost:8080/myapp/doJob/extra?id=foobar&x=y
```

上記は次のように分解されます。

http://localhost:8080/コンテキストパス/サーバパス/拡張パス…?クエリパラメータ

このURLを前提にメソッドの具体的な返り値と説明を**表B.3**にまとめます。

表B.3　リクエストURLに関係する代表的なメソッド

メソッド名	説明	返り値の具体例
getContextPath	コンテキスパス	/myapp
getServletPath	サーバパス	/doJob
getPathInfo	拡張パス	/extra
getQueryString	クエリパラメータ	id=foo&x=1&y=2
getRequestURI	URLのパス部分	/myapp/doJob/extra
getRequestURL	クエリパラメータを除くURL全体	http://localhost:8080/myapp/doJob/extra

■ クエリパラメータ

クエリパラメータはURLの?文字以降に現れる文字列です。リンク先URLで普通に記述する場合もあれば、入力フォームの入力値がクエリパラメータになる場合もあります。

後者を補足します。Webブラウザで利用者からの入力を受けつけるには一般にフォームと呼ばれるHTML要素を使います。テキスト入力領域やチェックボックスなどで馴染みがあるでしょう。

フォームの入力項目をサーバに送信する時、GETメソッドもしくはPOSTメソッドのどちらで送信するかを選択できます(コラム参照)。GETメソッドで送信するとフォームの入力項目はリクエストURLのクエリパラメータになります。たとえば次のHTMLを見てください。

```
// HTMLフォームの例
<form method="GET" action="/doJob>
  <input name="title" type="text"/>  <!-- フィールド名が title -->
  <input name="submit" type="submit" value="Submit" />
</form>
```

C O L U M N

GETとPOSTのメソッドの使い分けの指針

GETとPOSTのメソッドの使い分けの指針は次のようになります。

GETメソッドは名前のとおりWebサーバから情報を得る場合に使うのが原則です。一方、POSTメソッドはWebサーバ上の状態を変更するために使うのが原則です。

HTTPメソッドをクラスのメソッドに見立てると、問い合わせ処理にGETメソッドを使い、コマンド処理にPOSTメソッドを使うと説明できます。

たとえば座標をフォームに入力して地図を表示する場合、情報を得る処理なのでGETメソッドを使うのが適切です。この時のURLは「http://maps.foo.com?x=100&y=200」のようになり、URLをリンクとして使えます。

GETメソッドとPOSTメソッドの使い分けの例外がログイン処理などパスワード送信処理です。パスワードをGETメソッドで送ると、ブラウザの履歴やサーバ側のアクセスログにパスワードが残ってしまうからです。

利用者が入力領域にfooを入力して送信すると、リクエストURLのクエリパラメータはtitle=fooになります。便宜上、title=fooのtitleに相当する値をクエリ名、fooに相当する値をクエリ値と呼びます。クエリ名は大文字小文字を区別します（case-sensitive）。

■ クエリパラメータ用メソッド

クエリパラメータ全体の文字列はgetQueryStringメソッドで取得できます。しかし、通常、クエリパラメータ全体を取得する必要はありません。代わりに**表B.4**のメソッドを使います(注6)。これらのメソッドを使うと、クエリパラメータをキー（クエリ名）とバリュー（クエリ値）のペアとして扱えて便利だからです。また、通常クエリパラメータにはURL固有のエンコード処理が施されていますが、**表B.4**のメソッドを使うとデコード処理を隠蔽できる利点もあります。

同じクエリ名に対して複数のクエリ値が存在しえます。リクエストURL的には「http://localhost:8080/appname?title=foo&title=bar」のような場合です。この場合、クエリ名titleに対して、クエリ値がfooとbarの2つになります。

クエリ値は常に文字列で得られます。意味的に数値を送信する場合もネットワーク上は文字列になるのでメソッドの返り値の型はStringです。必要に応じて数値に変換するのは開発者の責任です（後述するJAX-RSを使うと型変換を任せられます）。

表B.4　クエリパラメータ取得メソッド

メソッド定義	説明
String getParameter(String name)	クエリ名からクエリ値を取得。存在しない場合、null
String[] getParameterValues(String name)	クエリ名から複数のクエリ値を取得。存在しない場合、null
Enumeration<String> getParameterNames()	クエリ名の一覧を取得。クエリが1つも存在しない場合、空のEnumeration。返り値の型はSet<String>相当
Map<String,String[]> getParameterMap()	キーがクエリ名、値がクエリ値の集合のマップを取得

■ フォームのPOSTデータ（サブミットデータ）

HTMLフォームの入力項目はPOSTメソッドでも送信できます（POSTメソッドのほうが一般的です）。HTTPの観点で見ると、POSTデータはヘッダ部と分かれたボディ部で送信されます。

この場合でも、**表B.4**のクエリパラメータを取得するメソッドを使えます。サーブレットコンテナが内部でHTTPボディ部を解析して、クエリ名とクエリ値のペアにするからです。つまり、フォーム送信に関しては、HttpServletRequestがGETメソッドとPOSTメソッドの違いを隠蔽します。

■ HTTPのボディ部

HTTPのボディ部はフォーム送信に限定するものではありません。HTTPボディ部の現実的な形式は次の3パターンです。括弧内に通称を併記します。

（注6）　Enumerationは古いクラスです（本誌「**6章　コレクションと配列**」参照）。Collectionsクラスのlistクラスメソッドで List型に変換してください。

- HTMLフォームから送るPOSTデータ（フォームデータ）
- ファイルアップロードによるPOSTデータ（ファイルアップロードデータ）
- その他（Web APIでのJSONやXMLでのデータ送信。あるいは任意のバイト列データ）

一般的なWebブラウザが送信できるHTTPボディ部の形式はフォームデータとファイルアップロードデータの2種です。それ以外の形式はWebサービスなど他の用途で使います。

前節で説明したようにフォームデータはHttpServletRequestが内部的にクエリパラメータとして解釈するので、**表B.4**のメソッドでデータを取得できます。ファイルアップロードデータはHttpServletRequestのgetPartメソッドで取得可能です。その他の形式は、開発者が自分でHTTPボディ部を解釈する必要があります。

■ HTTPのボディ部のI/Oストリーム処理

HTTPボディ部のデータを読み取るには、HttpServletRequestのgetInputStreamメソッドもしくはgetReaderメソッドでI/Oストリームオブジェクトを取得する必要があります。I/Oストリームの詳細は「ファイルとネットワーク」の章を参照してください。バイトI/Oストリームとして扱う場合はgetInputStreamメソッド、文字I/Oストリームとして扱う場合はgetReaderメソッドを使います。

```
// リクエストボディ用I/Oストリームの取得メソッド
ServletInputStream getInputStream() throws IOException  // ServletInputStreamクラスはInputStreamクラスの拡張継承クラス
BufferedReader getReader() throws IOException
```

HTTPボディ部のデータ長を取得するgetContentLengthメソッドがあります。ただしHTTPの構造上、データ長が不明な場合があります(注7)。データ長が不明な場合、getContentLengthメソッドは-1を返します。確実にHTTPボディ部を読み取るには、データの終端まですべて読み取る処理が必要です。

理屈上はHTTPボディ部のデータを解析する処理を自分で書けます。しかしあまり現実的ではないので、後述するJAX-RSや別のライブラリに解析処理を任せることを推奨します。

■ リクエストヘッダ

リクエストヘッダ情報の読み取りメソッドを**表B.5**にまとめます。

（注7） HTTPではContent-Lengthヘッダで HTTPボディ長を指定できます。しかし、Content-Lengthヘッダは必須ではないので、全体長が不明な場合があります。

表B.5　リクエストヘッダ取得メソッド

メソッド定義	説明
String getHeader(String name)	ヘッダ名からヘッダ値を取得。ヘッダが存在しない場合、null
int getIntHeader(String name)	ヘッダ名からヘッダ値をintで取得。ヘッダが存在しない場合、-1。intに変換できない場合、NumberFormatException例外が発生
long getDateHeader(String name)	ヘッダ名からヘッダ値を時刻のエポック値で取得。ヘッダが存在しない場合、-1。エポック値に変換できない場合、IllegalArgumentException例外が発生
Enumeration<String> getHeaderNames()	ヘッダ名の一覧を取得。ヘッダが1つも存在しない場合、空のEnumeration。返り値の型はSet<String>相当
Enumeration<String> getHeaders(String name)	ヘッダ名から複数のヘッダ値を取得。ヘッダが存在しない場合、空のEnumeration。返り値の型はList<String>相当

ヘッダ名には大文字小文字の区別がありません(case-insensitive)。"Referer"でも"referer"でも同じように動作します。

リクエストヘッダの読み取りはフレームワークに隠蔽することが多く、実際のWebアプリで使う機会はそれほど多くありません。個々のリクエストヘッダの有無がWebブラウザに依存するので、リクエストヘッダに依存する仕様は汎用性に欠けるためです。

B-2-4　レスポンス処理

HttpServletResponseオブジェクトを使うレスポンス出力処理をレスポンス処理と呼びます。

HTTPレスポンスは、「レスポンスステータス」「レスポンスヘッダ」「レスポンスボディ」の3つの構成要素からなります。HttpServletResponseはそれぞれに対応する出力メソッドを持ちます(**表B.6**)。

表B.6　レスポンスの構成要素とHttpServletResponseの対応メソッド

レスポンスの構成要素	対応メソッド
レスポンスステータス	setStatusなど
レスポンスヘッダ	setHeaderやaddHeaderやsetContentTypeなど
レスポンスボディ	getOutputStreamやgetWriterなど

■ レスポンスステータス

レスポンスステータスは"200 OK"や"404 Not Found"などで知られる文字列です。200や404の数字の部分がステータスコードです。HTTPの規格で数値と意味と説明文字列が決まっています。

setStatusメソッドでレスポンスコードをセットできます。引数にステータスコードを与えます。ステータスコードはHttpServletResponseのクラスフィールドで定数定義されています。下記に一部を引用します。

```
// ステータスコードの定数定義（一部抜粋）
public static final int SC_OK = 200;
public static final int SC_MOVED_TEMPORARILY = 302;
public static final int SC_UNAUTHORIZED = 401;
public static final int SC_FORBIDDEN = 403;
public static final int SC_NOT_FOUND = 404;
```

実際のWebアプリでsetStatusメソッドを使うべき場面はあまりありません。なぜならsetStatusメソッドを呼ばない場合、自動的に200の成功ステータスコードになるからです。

表B.7のような、200以外のステータスコードを返すために特別に用意されたメソッドがあります。

表B.7　ステータスコードを変更するHttpServletResponseのメソッド

メソッド名	説明
sendError	引数で指定したステータスコードでエラーページを返す
sendRedirect	リダイレクト処理を行う

sendErrorメソッドで（コンテナの）デフォルトエラーページを返せます。しかしsendErrorメソッドの濫用は避けるべきです。なぜならデフォルトのエラーページを返しても利用者に利することはほとんどないからです。たとえばフォーム入力で入力値が足りない場合、デフォルトエラーページを返すのではなく、利用者に適切なフィードバック（不正な入力項目のあったフィールドを明示するなど）のある画面を返すほうが望ましいはずです。

sendRedirectは「**B-2-6　リダイレクト処理**」で説明します。

■ レスポンスヘッダ

レスポンスヘッダはヘッダ名とヘッダ値のペアで指定します。リクエストヘッダ同様、レスポンスヘッダの処理もフレームワークで暗黙に処理することがほとんどです。Webブラウザがレスポンスヘッダをどう解釈するかWebブラウザ依存が大きく、レスポンスヘッダの利用は汎用性が低いからです。

比較的利用頻度が高いメソッドがsetContentTypeメソッドです。レスポンスボディのフォーマットを指示するContent-Typeヘッダの値を指定できるメソッドです。HTML以外のフォーマットでレスポンスを返す場合、指定するほうが利用者の利便性が上がります。

■ レスポンスボディ

レスポンスボディの送信はI/Oストリームに対する出力で行います。バイトI/Oストリームと文字I/Oストリームのそれぞれの取得メソッドがあります

```
// レスポンスボディ用I/Oストリームの取得メソッド
PrintWriter getWriter() throws IOException
ServletOutputStream getOutputStream() throws IOException
```

getWriterメソッドの使い方は**リストB.4**で示しました。出力I/Oストリームへの書き込みはそのままレスポンスボディとして送信されます。直感的には、I/OストリームにHTML文字列を書き込むとそれがWebブラウザの画面に表示されます。

ただし、適切なMVCアーキテクチャに従うサーブレットアプリでは、レスポンスボディ部の出力処理をサーブレットクラス自身が行うべきではありません。JSPなどのビュー処理に任せる（フォワードする）のが定石です。

B-2-5　フォワード処理

フォワード処理とは、他のサーブレットクラスやJSPに処理を丸投げ（委譲）することです。

他のサーブレットオブジェクトに処理を委譲するには、そのサーブレットオブジェクトのdoメソッドを直接呼べばいいと思うかもしれません。しかし、このようなコードは決して書いてはいけません。サーブレットオブジェクトから別のサーブレットオブジェクトの参照は禁止と考えてください。代わりにフォワード処理で処理を委譲します。

■ フォワード処理の方法

フォワード処理するにはRequestDispatcherオブジェクトを使います。RequestDispatcherオブジェクトは**表B.8**の3つの手段で取得できます。

表B.8　RequestDispatcherオブジェクトの取得メソッド

取得メソッド	説明
ServletContextオブジェクトのgetRequestDispatcherメソッド	URLのパスからRequestDispatcherオブジェクトを取得。一般的な取得手段
HttpServletRequestオブジェクトのgetRequestDispatcherメソッド	URLのパスからRequestDispatcherオブジェクトを取得。上記のラッパーメソッド
ServletContextオブジェクトのgetNamedDispatcherメソッド	サーブレット名からRequestDispatcherオブジェクトを取得

フォワード処理の典型的な処理は下記になります。

① getServletContextメソッドでServletContextオブジェクトを取得
② ServletContextオブジェクトからgetRequestDispatcherメソッドでRequestDispatcherオブジェクトを取得
③ RequestDispatcherオブジェクトに対してforwardメソッドを呼ぶ

フォワード処理の例を示します。**リストB.6**はURLパスからRequestDispatcherオブジェクトを取得します。この場合のフォワード先サーブレットオブジェクトはURLマッピングで決まります。**リストB.7**はサーブレット名からRequestDispatcherオブジェクトを取得ます。この場合、サーブレット名で決まります。

forwardメソッドは内部的に（間接的に）フォワード先サーブレットオブジェクトのdoメソッドを呼び出します。

リストB.6　getRequestDispatcherを使うフォワード処理

```
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        リクエスト処理など
        getServletContext().getRequestDispatcher("/Another").forward(req, resp); // レスポンス生
成処理を委譲
    }
}
```

リストB.7　getNamedDispatcherを使うフォワード処理

```
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        リクエスト処理など
        getServletContext().getNamedDispatcher("my").forward(req, resp); // レスポンス生成処理を委譲
    }
}
```

これで、サーブレットオブジェクトから別のサーブレットオブジェクトへフォワード処理が可能ですが、残念ながら悪い習慣です。サーブレットクラス間に依存関係を持たせることは避けるべきだからです。フォワード処理はサーブレットクラスからJSPのみに限定してください。**リストB.6**のようにURLマッピングを使うと、JSPへのフォワードが可能です。こういう事情があるので、実際に使うのはgetNamedDispatcherではなくgetRequestDispatcherのほうです。

■ インクルード処理

フォワード処理と似た機能にインクルード処理があります。フォワード処理が処理のすべてを丸投げするのに対し、インクルード処理は処理の一部を任せる機能です。

HTML内のヘッダやフッタのような共通部分のレスポンス生成をインクルード処理に任せることが一般的です。ただ、何度も書いているようにサーブレットクラスの中からレスポンス生成を直接行うのは勧めないので、必然的にインクルード処理も非推奨になります。このため本書では説明を省略します。

B-2-6　リダイレクト処理

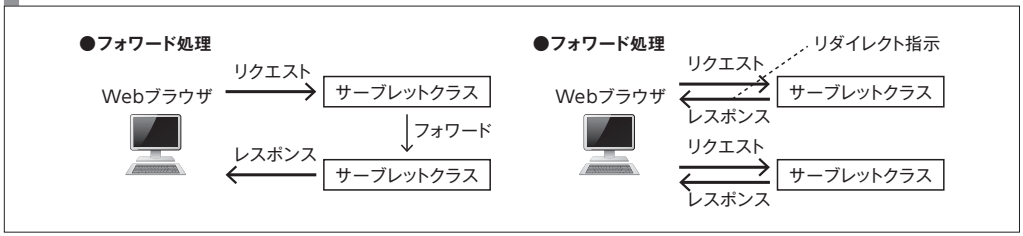
リダイレクト処理とは、特別なステータスコードを返すレスポンスのことです。リダイレクト用のステータスコードを受けたWebブラウザは、指定されたURLにもう1度リクエストしなおします。Webブラウザのリダイレクト動作は自動で動くため、Webブラウザの利用者は通常のレスポンスとリダイレクトによる別URLのレスポンスの区別がつきません(Webブラウザのアドレス欄を見れば気づきますが)。つまり、元のリクエスト処理の中から別リクエスト処理に切り替える効果があります。

doメソッドからリダイレクト処理をするにはHttpServletResponseオブジェクトのsendRedirectメソッドを呼びます。引数にリダイレクト先のURLを指定します。

■ リダイレクト処理とフォワード処理

リダイレクト処理とフォワード処理は時々混同されますがまったく別の仕組みです。フォワード処理とリダイレクト処理の違いを**図B.4**で示します。

図B.4　リダイレクト処理とフォワード処理



リダイレクト処理をするとHTTPの通信が余計に発生します。サーバ側の負荷も少しかかり、利用者の体感する速度も若干低下します。この速度低下を嫌い、リダイレクト処理を避けて、フォワード処理でなるべく代替しようと考える人がいます。残念ながらこれは誤った代用です。フォワード処理とリダイレクト処理はそもそも目的が異なるからです。リダイレクト処理を適切に使わないWebアプリはURLを軽んじた設計になります。具体例を次節で示します。

■ リダイレクト処理を使うべき時

ブログや掲示板などのWebアプリを例に考えます。文書作成画面で「保存」ボタンを押して文書を保存した後、利用者に文書一覧画面を見せたいとします。動作上はリダイレクトでもフォワードでも同じ効果を得られます。

ここで動作効率を考えてフォワード処理にするのは間違いです。「保存」ボタンを押した時のHTTPリクエストは一般にPOSTメソッドであり、この時のリクエストURLは保存処理に対応するURLです。仮にフォワード処理で文書一覧画面を返してしまうと、Webブラウザは、保存処

理のURLに対して文書一覧画面が表示される状態になります。たいした問題に感じないかもしれませんが、利用者が文書一覧画面のつもりでこのURLをブックマークしたり、あるいはURLをリンクとして共有すると問題になります。

特別な設計（残念ながらおかしな設計です）をしていない限り、保存用URLに直接アクセスしても文書一覧画面を表示しないはずです。つまり利用者から見ると、不正なブックマークやリンクです。

この問題を避けるためにリダイレクトを使います。保存処理の後、文書一覧を表示するためのリクエストURLにリダイレクトします。Webブラウザは文書一覧画面用のURLにリダイレクトアクセスして文書一覧を表示します。リダイレクト処理は利用者には見えない処理なので、利用者からすれば、文書保存後に直接文書一覧画面に遷移したように見えます。この時Webブラウザに残っているURLは適切なものです。ブックマークもリンクも適切に動作します。

B-2-7　状態管理

MVCアーキテクチャに従うと、サーブレットクラスからJSPなどへフォワードするコードが一般的になります。この時、サーブレットクラス内の何らかの処理結果をフォワード先に渡す必要が生じます。

一般にメソッド間で状態を引き渡すには引数を使います。しかしフォワード処理は間接的にメソッドを呼ぶ構造上、直接引数を渡せません。そもそも呼び出し先がJSPだとすると引数で渡すという概念自体が希薄になります。

任意の引数を渡せない問題に対して、サーブレットプログラミングでは「呼び出し元と呼び出し先の間で共有可能なオブジェクトに属性を持たせる」という手法を使います。

属性は概念的にはマップのような機能です。文字列をキーとして任意のオブジェクトを値として持てます。つまりMap<String,Object>相当です。マップと区別するため、属性の場合、キーバリューではなく属性名と属性値と呼ぶことにします。

サーブレットとJSPで共有可能オブジェクトの1つがHttpServletRequestです。HttpServletRequestは次の属性用メソッドを持ちます。マップ同様、同じ属性名でsetAttributeを呼ぶと元の属性値を上書きします。

```
// HttpServletRequestの属性用メソッド
void setAttribute(String name, Object o);
Object getAttribute(String name);
```

サーブレットクラスからJSPにフォワードすると、両者は同じHttpServletRequestオブジェクトを参照できます。フォワード元のサーブレットクラスはsetAttributeメソッドを使い任意のオブジェクトを属性としてセットしておきます。フォワード先のJSPが同じ属性名でgetAttributeメソッドを呼び出すと、属性値として事前にセットされたオブジェクトを取得できます。お互いの間で使う属性名を取り決めておけば、任意のオブジェクトをいくつでも共有可能

です(注8)。

属性値の型はObjectなので任意のオブジェクトを指定できます。属性値にはJavaBeansオブジェクトもしくは文字列をキーとしたマップオブジェクトが便利です。なぜなら、JSPのELで簡易にアクセス可能だからです。例は「**付録D　Webアーキテクチャ**」で紹介します。

■ 属性を持つオブジェクト

HttpServletRequestのように属性を持てるオブジェクトが他に3種類あります。便宜上、これらを属性用コンテナと呼びます。HttpServletRequestを含めて**表B.9**にまとめます。

表B.9　属性用コンテナ

コンテナ型	doメソッド内での取得方法	JSPのスコープ
HttpServletRequest	引数で渡ってくる	リクエスト
HttpSession	HttpServletRequestのgetSessionメソッドで取得	セッション
ServletContext	getServletContextメソッドで取得	アプリケーション
PageContext	なし	ページ

属性用コンテナの違いはサーブレットの世界ではスコープの違いとして認識されます。スコープは可視範囲と生存期間に関係します。スコープの広い順にアプリケーション、セッション、リクエスト、ページになります。スコープが広いほど可視範囲が大きく生存期間が長くなります。スコープが広いほどグローバル変数に近くなるので、必要な範囲で最小スコープの属性コンテナを使うべきです。

この規則に例外があります。ページスコープです。他の属性用コンテナと異なり、サーブレットクラスとJSPの間の状態共有には使えません。ページスコープのPageContextは、JSPの中だけで有効な属性用コンテナだからです。

■ 属性とスコープ

ページスコープ以外の残りの3つに限定して説明を続けます。

HttpServletRequestは3つの中でもっとも小さいスコープです。HttpServletRequestオブジェクトはHTTPリクエストごとに生成されるオブジェクトなので、生存期間も短く他に与える影響も最小です。また複数スレッドで共有しないので同期処理を考える必要もありません。このため、サーブレットクラスとJSPの間だけで共有する状態は、HttpServletRequestの属性で引き渡してください。

HttpSessionを使うセッションスコープにはセッションの理解が必要です。現時点では、セッションとはログイン中のユーザに紐付くものだと理解してください。ユーザに紐付く状態管理はHttpSessionの属性で管理します。詳しくは「**B-4　セッション管理**」で説明します。

(注8)　属性を使う状態引き渡しには弱点もあります。属性名が文字列なので打ち間違いをコンパイルエラーで検出できません。

ServletContextを使うアプリケーションスコープは、事実上、サーブレットアプリの中のグローバル変数です。グローバル変数と同じ理由で利用は推奨しません。

スコープという概念は後述するJAX-RSやCDIにも存在する概念です。スコープで状態の可視範囲と生存期間をうまく統制するのが複雑なWebアプリ開発のひとつの肝になります(注9)。

B-3 JAX-RS

B-3-1 JAX-RSとは

JAX-RSはサーブレットAPIとほぼ同等の領域をカバーする新しいAPIです。

JAX-RSはRESTfulなWebアプリを実現する規格です。RESTfulとはREST風という意味で、RESTとはRepresentational State Transferの略です。RESTはWebアプリのアーキテクチャを説明する用語の1つで、Webアプリ開発の視点で見ると、URL設計に強く影響を与える用語です。

REST以前のURL設計は、URLをメソッド名のように考える方式が主流でした。前章でリクエストURLをメソッド名に例えたようにです。

一方、RESTfulなWebアプリの場合、URLはオブジェクト名やデータ名に相当すると見立てます。RESTの文脈では、サーバ上のサービスやデータをリソースと呼び、URLはリソースを指し示す参照と見立てるからです。Webアプリは公開URLで各種リソースへの操作を提供するプログラムになります。操作するメソッド名に相当するのが、GETやPOSTなどのHTTPメソッドです。GET、POST以外に、PUT、DELETEの4つが基本的な操作になります。

本書はRESTの専門書ではないので、RESTの詳細にはこれ以上踏み込みません。代わりに、サーブレットAPIより高度なWeb処理APIという視点でJAX-RSを紹介します（コラム参照）。なお、GlassFishのデフォルトJAX-RS実装であるJerseyを使います。

C O L U M N

サーブレットのURLマッピングの無効化

JAX-RSを使うとサーブレットでURLマッピングをする必要がなくなります。この場合、サーブレットの@WebServletアノテーション検出処理を無効化すると少し無駄が減ります。web.xmlファイルのweb-app要素にmetadata-complete="true"属性を指定します。ただ、他のサーブレットアノテーション（フィルタ用の@WebFilterなど）も無効になるので注意してください。

（注9） 大規模なアプリの状態管理や状態共有は難しいため、適切な設計が重要です。可視範囲と生存期間に名前をつけて明確に管理するスコープという概念は、Webアプリに限らず使える設計技法です。

B-3-2　自作JAX-RSアプリをGlassFishで動かす工程

簡単な自作JAX-RSアプリを作ってGlassFishで動かしてみます。Webアプリ名を「myjaxrs」にします。

サーブレットアプリの「自作Webアプリを GlassFishで動かす工程」と同じ手順で、開発ソースツリーの雛形作成、続いてpom.xmlとweb.xmlの書き換えを実施してください（web.xmlは消してもかまいません）。

次に、**リストB.8**のファイルを「$WEBAPP/src/main/java/my/MyJax.java」に配置、**リストB.9**のファイルを「$WEBAPP/src/main/java/my/AppConfig.java」に配置してください。その後、mvnコマンドによるビルド、デプロイをサーブレットアプリと同じ手順で実行します。

リストB.8　JAX-RSアプリの例

```
package my; // パッケージ名は任意（後述のコード例では記述を省略）

// 紙幅の節約のため、後述のコードから類似のimport文の記述を省略していきます
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped  // なくても動きます（後述）
@Path("my")
public class MyJax { // クラス名は任意
    @GET
    @Path("hello")
    @Produces(MediaType.TEXT_HTML)
    public String hello() {
        String html = "<html><head><title>hello JAX-RS</title></head>"
                    + "<body><p>hello, JAX-RS</p></body></html>";
        return html;
    }
}
```

リストB.9　Applicationクラスの継承クラス

```
package my; // パッケージ名は任意

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/")
public class AppConfig extends Application { // クラス名は任意（クラスの中身は空でよい）
}
```

Webブラウザから「http://localhost:8080/myjaxrs/my/hello」のURLにアクセスしてhello, JAX-RSの画面が現れれば成功です。

■JAX-RSアプリの読解

リストB.8のMyJaxクラスのように、@Pathアノテーションを付与したクラスをリソースクラスと呼びます(注10)。リソースクラスのメソッドのうち、@GETや@POSTのアノテーションを付与したメソッドをリソースメソッドと呼びます。リソースメソッドにも@Pathアノテーションを付与可能です(必須ではありません)。

リソースメソッドは、HTTPリクエスト処理のエントリポイントです。サーブレットクラスのdoメソッドと同じ役割のメソッドと考えてください。サーブレットの場合、doGetやdoPostのようにメソッド名が抽象基底クラスで決まっていますが、リソースメソッドの名前は開発者が任意に決められます。

メソッド名が任意である代わりに、メソッドの役割をアノテーションで規定します。リソースクラスの@Pathとリソースメソッドの@Pathでリソースメソッドに紐づくリクエストURLが決まります。また、@GETや@POSTなどのアノテーションでHTTPのメソッドを規定します。

リソースメソッドの@Pathはリソースクラスの@Pathからの相対パスになります。つまり、**リストB.8**の場合、リソースクラスの@Path("my")とリソースメソッドの@Path("hello")からmy/helloというURLのパスが決まります。

JAX-RSのリクエストURLを決める要素はもう1つあります。**リストB.9**のようなクラスです。Applicationクラスを継承して、かつ@ApplicationPathを付与したクラスです。クラス名は自由に付けられます。このクラスの直感的な理解はJAX-RSのグローバル設定クラスです。**リストB.9**はクラスの中身が空ですが、JAX-RSの初期化コードをクラス内に記述可能です。

@ApplicationPathの要素がJAX-RS全体のURLのルートパスになります。**リストB.9**の場合"/"を指定しているので、すべてのURLがJAX-RSの対象になります。仮に@ApplicationPath("/rest")とすると、/restで始まるパスのURLのみがJAX-RSの処理対象になります。そして、**リストB.8**のリソースメソッドのURLパスは/rest/my/helloになります。

ここまでの情報でリソースメソッドに対応するリクエストURLが決まります。Webアプリ名がmyjaxrsだとすると、JAX-RSのルートパスが"/"、リソースメソッドのパスが"my/hello"なので、これらを合わせて「http://localhost:8080/myjaxrs/my/hello」で**リストB.8**のリソースメソッドが呼ばれます。

リストB.8のリソースメソッドにはもう1つアノテーションがあります。@Produces(MediaType.TEXT_HTML)です。これでレスポンスのフォーマットを規定できます。HTML以外にJSONやXMLなどを指定できます。

(注10) リソースクラスは、ルートリソースクラスとサブリソースクラスに分類できます。サブリソースはルートリソースから処理を委譲されるリソースクラスです。本書はサブリソースの説明を割愛するので、本書で扱うリソースクラスはすべてルートリソースと考えてください。

リソースメソッドの返り値の型と引数は一定の制約の下でいくつかのバリエーションがあります。完全にメソッドの型が決まっているサーブレットのdoメソッドと異なる部分です。具体例は本章の説明を通じて説明します。

■ @Pathの詳細

@Pathアノテーションのvalue要素にはURLパスを指定します。正規表現も指定可能です。たとえば@Path(".*")にすると任意のパスにマッチします。

正規表現が入ると、1つのリクエストURLに対して複数のリソースメソッドがマッチする可能性があります。どの@Path指定が選択されるかは少し複雑です。直感的には@Pathのvalue要素のリテラル（正規表現の特殊文字以外と考えてください）の一致長が長いほど選択されると考えてください。少し複雑になる場合は動作確認するほうが簡単です。

@Pathのvalue要素には正規表現以外にもう1つ特殊な表記があります。URIテンプレートと呼ばれる表記です。パス文字列の一部を{}で囲む表記です。たとえば@Path("users/{id}")のように表記します。{}で囲った部分をリソースメソッド内でプログラム的に参照できます。使い方は後ほど@PathParamと合わせて説明します。

■ リソースクラスのスコープ

デフォルトでは、JAX-RSコンテナはリクエストごとにリソースクラスのインスタンスを生成します。この動作をリクエストスコープと呼びます。これはサーブレットクラスと異なる動作なので注意してください。元々のJAX-RSは、リソースクラスのオブジェクトにレスポンス用の状態を持たせる方針だからです。

しかし、本書はリソースクラスに、HTTP処理のエントリポイントの役割のみを持たせる設計方針を取ります。HTTP処理のエントリポイントと、状態を持つオブジェクトを別にしたほうが見通しが良くなるからです（後述するフォームビーンなどに状態管理を寄せます）。

この設計方針に従うと、リソースクラスに状態を持たせるべきではありません。つまり可変なインスタンスフィールドを持たせません。この場合、リクエストごとのオブジェクト生成は無駄なので、**リストB.8**のように@ApplicationScopedを付与して、リソースクラスのオブジェクトをシングルトンにするのを勧めます（コラム参照）。

B-3-3　リクエスト処理

リクエスト処理に使えるJAX-RSアノテーションを**表B.10**にまとめます。リソースメソッドは一定の制約の下で任意の引数を持てると書きました。これらのアノテーションの付与が制約になります。

これらのアノテーションをリソースメソッドの引数に付与すると、リソースメソッドがコンテナから呼ばれた時に、自動的に値が入ってきます（**リストB.10**）。

これらのアノテーションさえあれば、リソースメソッドは、任意の引数名および任意の数の引数を持てます。引数の型は、文字列型、基本型、文字列型から変換可能な任意の型の3パターンあります。

文字列型から変換可能な型とは、引数が文字列1つのコンストラクタを持つクラス、文字列1つの引数を持つvalueOfまたはfromStringメソッドを持つ型のいずれかです。これ以外に独自に変換処理（ParamConverter）を開発者が用意することも可能です。**表B.10**のアノテーションで多値を得られるので、List<String>、Set<String>、SortedSet<String>などのコレクション型の引数も可能です。

表B.10　リクエスト処理用のJAX-RSアノテーション

アノテーション	説明
@QueryParam	リクエストURLのクエリパラメータ値
@PathParam	リクエストURLのパス値。パスの位置はURIテンプレートで指定する（前述の@Pathの説明を参照）
@HeaderParam	リクエストヘッダ値
@CookieParam	リクエストのCookieヘッダ値（Cookieヘッダは、名前と値のペアの集合）
@MatrixParam	リクエストURLの最後にセミコロン文字（;）で区切った後に並べる名前と値のペアの集合の値

C O L U M N

シングルトンオブジェクトのためのアノテーション

Java EEには、シングルトンオブジェクトにするためのクラス用アノテーションがいくつかあります。

@javax.ejb.SingletonはEJBのアノテーションです（EJB Liteでも利用可能）。EJBを使うと、EJBコンテナに同期処理とデータベースのトランザクション制御を隠蔽できます。同期処理の隠蔽とは、簡単に言うとsynchronizedコード相当の処理をEJBコンテナに任せることです。しかし、本文で述べたように、リソースクラスにインスタンスフィールドを持たせない方針の場合、同期処理はそもそも不要です。同様に、リソースクラスからデータベースアクセスをする予定もないので、トランザクション制御も余計な機能です。

@javax.ejb.StatelessもEJBのアノテーションです（同じくEJB Liteでも利用可能）。Statelessは名前のとおりインスタンスフィールドを持たない前提のアノテーションです。このため同期処理の隠蔽機能はありません。しかしトランザクション制御の隠蔽はあります。

結論すると、EJB用アノテーションは目的をかなえますが、余計なオーバーヘッドがあるのでリソースクラスには推奨しません。

@javax.inject.Singletonは、@javax.enterprise.context.ApplicationScopedの代わりに使っても大きな弊害はなく、目的を達成できます。ただ、@javax.ejb.Singletonと紛らわしいという理由で、本書は@ApplicationScopedを利用します。

リストB.10　JAX-RSのリクエスト処理

```
import javax.ws.rs.QueryParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.HeaderParam;
import javax.ws.rs.CookieParam;
import javax.ws.rs.MatrixParam;

// http://localhost:8080/myjaxrs/my/hello/foo/bar;m=baz?q=keyword
// でアクセスしたと仮定すると、リソースメソッドの各引数は下記コメントのような値になる
@ApplicationScoped
@Path("/my")
public class MyJax {
    @GET
    @Path("hello/{id}/{trailing:.+}")
    @Produces(MediaType.TEXT_HTML)
    public String hello(@QueryParam("q") String param,  //=> "keyword"
                        @PathParam("id") String id,     //=> "foo"
                        @PathParam("trailing") String trail, //=> "baz"
                        @MatrixParam("m") String matrix,     //=> "baz"
                        @HeaderParam("User-Agent") String userAgent,   //=> "User-Agent"リクエスト
ヘッダの値
                        @CookieParam("JSESSIONID") String sessionId) { //=> Cookieヘッダの
JSESSIONID値
        メソッド内は省略
    }
}
```

リソースクラスをリクエストスコープ動作（リクエストごとにインスタンス生成）にすると、**表B.10**のアノテーションをリソースクラスのフィールドやJavaBeansプロパティに付与できます。なお、プロパティに付与とは、ゲッターあるいはセッターメソッドへの付与を意味します。

前述したように本書はリソースクラスをシングルトンオブジェクト（アプリケーションスコープ動作）として使う方針なので、この例は省略します。

■ フォーム処理

実アプリでもっとも良く使うリクエスト処理はHTMLフォームの入力項目の処理です。

リストB.11のような簡易なフォーム生成のJSPファイルを用意します。JSPの詳細は説明しませんが、ここでは生成されるHTMLの雰囲気だけイメージしてください（**図B.5**）。このJSPファイルをform.jspという名前で「$WEBAPP/src/main/webapp/form.jsp」に配置します。後述するように、JSPをリソースクラスからのフォワード先にする場合、直接URLで叩かせないためにWEB-INFディレクトリ下に配置するのが普通です。しかし今回は簡易に試すために直接JSPファイルをURLで叩きます。

サーブレットの世界で考えると、「$WEBAPP/src/main/webapp/form.jsp」に配置すると、「http://localhost:8080/myjaxrs/form.jsp」でアクセスできます（Webアプリ名はmyjaxrsという前提です）。しかし**リストB.9**の設定の場合、/myjaxrs/ 以下のすべてのリクエストURLがJAX-RS用のリクエストに見なされてJSPファイルにアクセスできません。これを回避するため、AppConfigクラスの@ApplicationPath()を@ApplicationPath("/rest")に変更してください（**リストB.12**）。これで「http://localhost:8080/myjaxrs/form.jsp」でアクセスできるようになります。

リストB.11　簡易フォーム用のJSP

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ page session="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head><title>edit</title></head>
<body>
<form action="rest/my/hello" method="POST">
  <p>タイトル: <input id="title" name="title" size="30" type="text" value="デフォルト値" /></p>
  <p>名前: <input id="name" name="name" size="30" type="text" value="デフォルト値" /></p>
  <p><input id="submit" name="submit" type="submit" value="送信" /></p>
</form>
</body>
</html>
```

図B.5　リストB.11のJSPにアクセスした画面

リストB.12　Applicationクラスの継承クラス（本節のみの設定とします。次節以降は再び"/"に戻します）

```
@ApplicationPath("/rest")  // リストB.9からの変更点
public class AppConfig extends Application { // クラス名は任意
}
```

form.jspの構造を説明します。formタグのaction属性の値がフォームデータ送信時のURLで、method属性の値がHTTPメソッドです。そしてtitleとnameという名前の2つのフィールド値を送信します。フォームのポスト先URLは「http://localhost:8080/myjaxrs/rest/my/hello」で、このURLに対応するリソースクラスが**リストB.13**です。

リストB.13は2つのフィールド値をリソースメソッドの引数で受けとります。引数の@FormParamアノテーションの結果です。

リストB.13のもう1つのポイントはリダイレクトを使うレスポンスです。リダイレクトにする意味はサーブレットでの「**B-2-6　リダイレクト処理**」の説明を参照してください。JAX-RSでのリダイレクト処理の詳細は後述します。

リストB.13　JAX-RSのフォーム処理

```
import java.net.URI;
import javax.servlet.http.HttpServletRequest;
import javax.ws.rs.POST;
import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;

@ApplicationScoped
@Path("my")
public class MyJax {
    @POST
    @Path("hello")
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)  // この行はなくても動作します
    public Response hello(@FormParam("title") String title, @FormParam("name") String name, @
Context HttpServletRequest req) {
        (name と title を使うコード例は省略)
        URI uri = URI.create(req.getContextPath() + "/index.jsp"); // リダイレクト先URL
        return Response.seeOther(uri).build();                     // リダイレクト処理(後述)
    }
}
```

デフォルトのGlassFishの場合、フォームに日本語を入力すると適切に**リストB.13**のString型で扱えません。日本語を扱うには「$WEBAPP/src/main/webapp/WEB-INF/glassfish-web.xml」に**リストB.14**の内容のファイルを作成してください。

リストB.14　フォーム処理で日本語を使うための設定ファイル(glassfish-web.xml)

```
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1
Servlet 3.0//EN"
"http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app>
    <parameter-encoding default-charset="UTF-8"/>
</glassfish-web-app>
```

■ フォームビーン

実アプリでは、フォームの入力項目を使って内部オブジェクトの生成をしばしば行います。HTMLフォームと対応するオブジェクトをフォームビーン、フォームオブジェクト、バッキングビーンなどと呼んだりします。単なる慣習なのでどの用語を使っても本質は変わりませんが、本章ではフォームビーンと呼びます(注11)。

リソースメソッドの@FormParam引数からフォームビーンを自前でnewする実装も可能です。ただフォーム上の入力フィールドの数が増えると、@FormParam引数の数が増えていきますし、たくさんの引数を渡すオブジェクト生成のコードは面倒です。こういった定型処理はフレームワークに隠蔽すべきです。

JAX-RSを使うとフォームの入力値からフォームビーンを自動で生成できます。いくつかの手法があるので**表B.11**にまとめます。

表B.11　HTMLフォームの送信データからフォームビーンを生成する手段

フォームビーン	説明
リソースクラス自身	リソースクラスをリクエストスコープにして、フィールドもしくはプロパティに@FormParamアノテーションを付与
MultivaluedMap<String,String>	リソースメソッドの引数をMultivaluedMap型にすると、フォーム送信データのキーバリューになる
Form	リソースメソッドの引数をForm型にする。FormオブジェクトのasMapメソッドでMultivaluedMapオブジェクト取得可能
自作クラス（@BeanParam）	リストB.18参照

本書が推奨する方式は@BeanParamの利用です（**リストB.15**）。フォームビーン用のクラスを**リストB.16**のように自分で定義します。

フォームビーンクラスのフィールドあるいはJavaBeansプロパティに@FormParamを付与します。そしてリソースメソッドにフォームビーン型の引数を@BeanParam付きで宣言します。これでHTMLフォームの入力処理でフォームビーンオブジェクトを自動生成できます。

リストB.15　@BeanParamでフォームビーンを扱うリソースクラス

```
@ApplicationScoped
@Path("my")
public class MyJax {
    @POST
    @Path("hello")
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public Response hello(@BeanParam MyDTO form, @Context HttpServletRequest req) {
        // リソースメソッドが呼ばれた時点で、引数のformオブジェクトがコンテナにより自動生成され、フォーム
入力値でtitleおよびnameフィールドに値がセットされる
        String title = form.getTitle();
```

（注11）　ここではフォーム送信（サブミット処理）のリクエスト内容からフォームビーンを生成する説明をします。逆方向の、内部オブジェクトから表示フォームや編集フォームのレスポンス内容を生成する処理は後述するフォワード処理の中で説明します。

```
        String name = form.getName();
        以下省略
    }
}
```

リストB.16　リストB.15で使うフォームビーンクラス

```
import javax.ws.rs.FormParam;

public class MyDTO {
    public MyDTO() {}   // 引数なしのコンストラクタが必要（アクセス制御は何でも良い）

    @FormParam("title") private String title;
    @FormParam("name") private String name;

    public String getTitle() { return title; }
    public String getName() { return name; }
}
```

■JSON形式でポストされたリクエストの受信処理

Web APIなどでは、HTMLフォームでのデータ送信の代わりにJSON形式でのデータ送信を使う場合が多々あります。

JSON形式のリクエストを受け取るJAX-RSアプリは**リストB.17**のように書けます。リソースメソッドに@Consumes(MediaType.APPLICATION_JSON)を付与して、引数に任意のJavaBeansオブジェクトを指定します。

JSONデータに対応する内部オブジェクト用のクラス例を**リストB.18**に示します。このクラスには、引数なしのコンストラクタ、JSONのプロパティに対応するJavaBeansプロパティ（セッター、ゲッター）が必要です。この条件を守れば、他のフィールドやメソッドに特別な制約はありません。

リストB.17　JSON形式で送られたリクエストを受け取るリソースクラス（MyDTOはリストB.18）

```
@ApplicationScoped
@Path("my")
public class MyJax {
    @POST
    @Path("hello")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.TEXT_PLAIN)
    public String hello(MyDTO dto) {
        return "OK; " + dto;
    }
}
```

リストB.18　JSONデータに対応するビーンクラス（リストB.17とリストB.19で利用）

```
public class MyDTO {
    public MyDTO() {}   // 引数なしのコンストラクタが必要（アクセス制御はpublicでなくても良い）

    private String title;
    private String name;

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @Override
    public String toString() {
        return "DTO: " + title + ", " + name;
    }
}
```

リストB.17のJAX-RSアプリのURLをcURLコマンドでアクセスする例を下記に示します（AppConfigクラスのアノテーションは@ApplicationPath("/")の前提です）。

```
$ curl -X POST -d '{"title":"foo", "name":"bar"}' -H "Content-Type: application/json" http://
localhost:8080/myjaxrs/my/hello
OK; DTO: foo, bar
```

B-3-4　リソースクラスからサーブレットAPIの利用

リソースクラスのコード内からサーブレットAPIを呼び出せます。このために@Contextアノテーションを使います。@Contextを付与すれば、サーブレットAPI用のクラスのインスタンスをリソースクラス内に宣言できます。これらのインスタンス生成はコンテナで自動的に行われ、リソースメソッド内で自由に利用できます。リソースメソッドの引数に@Contextを使う例は**リストB.13**を見てください。

リソースクラスをリクエストスコープにすれば、リソースクラスのフィールドやプロパティとして@Contextを付与したサーブレットAPIオブジェクトも利用可能です（くどいですが本書はこの方針を取りません）。

B-3-5　レスポンス処理

HTMLやプレーンテキストのレスポンスを返すには、リソースメソッドの返り値の型をString型にして、メソッドに@Produces(MediaType.TEXT_HTML)や@Produces(MediaType.TEXT_

PLAIN）を指定します。具体例は**リストB.8**や**リストB.10**を参照してください。

実アプリではサーブレットアプリ同様、JSPなどビュー処理へのフォワードを推奨します。JAX-RSのフォワード処理については後述します。

■JSON形式のレスポンス

Web APIではJSON形式のレスポンスをしばしば使います。**リストB.19**のようにリソースメソッドに@Produces(MediaType.APPLICATION_JSON)を付与して、返り値の型を任意のJavaBeansクラスにします。JavaBeansクラスには、引数なしのコンストラクタ、ゲッターおよびセッターメソッドが必要です。**リストB.19**は**リストB.18**を再利用したと仮定します。

リストB.19　JSONデータのレスポンスを返すリソースクラス（MyDTOはリストB.18）

```
@ApplicationScoped
@Path("my")
public class MyJax {
    @GET
    @Path("hello")
    @Produces(MediaType.APPLICATION_JSON)
    public MyDTO hello() {
        return new MyDTO("foo", "bar");
    }
}
```

リストB.19のJAX-RSアプリにcURLコマンドでアクセスした例を示します。

```
$ curl http://localhost:8080/myjaxrs/my/hello
{"name":"bar","title":"foo"}
```

内部的には、オブジェクトからJSONへの変換が暗黙に起きています。変換ライブラリはMOXy、Jacksonなど使えます。GlassFish4のデフォルト実装はMOXyです。

■XML形式のレスポンス

XML形式のレスポンスを返すには**リストB.20**のようにリソースメソッドに@Produces(MediaType.APPLICATION_XML)を付与して、返り値の型を任意のJavaBeansクラスにします。便宜上、このクラスをXML用クラスと呼びます。

XML用クラスには@XmlRootElementアノテーションを付与します（**リストB.21**）。@XmlRootElementのname要素で生成XMLのルート要素名を指定します。

XML用クラスには引数なしコンストラクタとゲッターメソッドが必要です。セッターメソッドは必須ではないので**リストB.21**では省略しています。XML用クラスのフィールドまたはJavaBeansプロパティに@XmlElementを付与します。@XmlElementのname属性で対応するXMLの要素名を指定できます。name属性がない場合、プロパティ名がXMLの要素名になります。

リストB.20　XMLデータのレスポンスを返すリソースクラス（MyDTOはリストB.21）

```
@ApplicationScoped
@Path("my")
public class MyJax {
    @GET
    @Path("hello")
    @Produces(MediaType.APPLICATION_XML)
    public MyDTO hello() {
        return new MyDTO("foo", "bar");
    }
}
```

リストB.21　XMLデータに対応するJavaBeansクラス（XML用クラス）

```
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "base")
public class MyDTO {
    private MyDTO() {}   // 引数なしのコンストラクタが必要（アクセス制御は何でも良い）

    public MyDTO(String title, String name) {
        this.title = title;
        this.name = name;
    }
    private String title;
    private String name;

    @XmlElement(name = "subject")  // name属性で、JavaBeansのプロパティ名と異なる名前のXMLの要素名
にできる
    public String getTitle() { return title; }

    @XmlElement                    // XMLの要素名はJavaBeansのプロパティ名になる
    public String getName() { return name; }
}
```

リストB.20のJAX-RSアプリにcURLコマンドでアクセスした例を示します。

```
$ curl http://localhost:8080/myjaxrs/my/hello
<base>
  <name>bar</name>
  <subject>foo</subject>
</base>
```

B-3-6　フォワード処理

「リソースクラスのスコープ」で述べたように、本書はリソースクラスをHTTP処理のエントリポイントの役割に限定します。MVCアーキテクチャの用語を使うとコントローラの役割です。レスポンス生成処理（MVCで言うビュー処理）は分離して、リソースクラスから処理をフォワードします。ビュー処理にはJSPを使う方針とします。

残念ながらJAX-RSの規格自体にはリソースクラスからJSPなどへフォワードする仕組みがありません。ただし、JAX-RS実装のJerseyは、独自実装でJSPなどのビュー処理へフォワードする仕組みがあります。本書はそれを利用します。

リソースクラスからJSPファイルへフォワード処理するには、次の2つの設定ファイルの書き換えが必要です。

- $WEBAPP/src/main/webapp/WEB-INF/web.xmlの書き換え（リストB.22）
- $WEBAPP/pom.xmlの書き換え（リストB.23）

リストB.22　JerseyでJSPへフォワード処理するためのweb.xmlの書き換え（注12）

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/
javaee/web-app_3_1.xsd"
         version="3.1">
  <!-- Jerseyクラスをフィルタに設定 -->
  <filter>
    <filter-name>jersey</filter-name> <!-- 任意のフィルタ名 -->
    <filter-class>org.glassfish.jersey.servlet.ServletContainer</filter-class>

    <!-- JAX-RSアノテーションを読むパッケージを限定可能（Jersey固有機能） -->
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>my</param-value> <!-- 任意のパッケージ名を ; 文字で区切って指定可能 -->
    </init-param>
    <init-param>
      <param-name>jersey.config.server.provider.scanning.recursive</param-name>
      <param-value>false</param-value>
    </init-param>

    <!-- JSPファイルの相対パスを設定 -->
    <init-param>
      <param-name>jersey.config.server.mvc.templateBasePath.jsp</param-name>
```

（注12）フィルタとして設定する必要があるのはJerseyの制限事項です（Jersey2のMVCでJSPを使う場合の制限事項）。将来的にはリストB.22のweb.xmlの書き換えは不要になるかもしれません。

```
      <param-value>/WEB-INF/jsp/</param-value>
    </init-param>

    <init-param>
      <param-name>jersey.config.server.provider.classnames</param-name>
      <param-value>org.glassfish.jersey.server.mvc.jsp.JspMvcFeature</param-value>
    </init-param>
  </filter>

  <!-- フィルタをURLパターンと結び付ける -->
  <filter-mapping>
    <filter-name>jersey</filter-name> <!-- 上記で指定したフィルタ名 -->
    <url-pattern>/*</url-pattern>     <!-- URLパターン（@ApplicationPathの指定は無効になります） -->
  </filter-mapping>
</web-app>
```

リストB.23　JerseyでJSPへフォワード処理するためのpom.xmlの書き換え（既存<dependencies>内に<dependency>要素を追記）

```
<dependencies>
  <dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-mvc-jsp</artifactId>
    <version>2.0</version>
    <type>jar</type>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

JSPへフォワード処理するリソースクラスは**リストB.24**のように書きます。ポイントはリソースメソッドの返り値の型です。（Jersey固有の）Viewableクラスにします。

Viewableオブジェクト生成の責務はリソースメソッドにあります。Viewableのコンストラクタの第1引数にJSPファイルのパス、第2引数にJSPへ渡すコンテキストオブジェクト（モデル）を渡します。モデルは任意のクラスのオブジェクトです。JSPから参照する利便性のため、JavaBeansクラスにするのが一般的です（**リストB.25**）。

JSPファイルのパスは**リストB.22**で指定した「/WEB-INF/jsp/」からの相対パスになります。**リストB.24**のフォワード先JSPファイルの開発ソースツリー上のパスは「/WEB-INF/jsp/view.jsp」です。フォワード先のJSPファイルは直接URLで叩けないようにWEB-INFディレクトリの下に配置するのが定石です。

リストB.24　JSPへフォワードするリソースクラス（MyModelはリストB.25）

```
import org.glassfish.jersey.server.mvc.Viewable; // 他のimport文の記述は省略

@ApplicationScoped
@Path("my")
public class MyJax {
    @GET
    @Path("hello")
    public Viewable hello() {
        MyModel model = new MyModel("foo", "bar");
        return new Viewable("/view.jsp", model);
    }
}
```

リストB.25　リストB.24で使うモデル用クラス

```
public class MyModel {
    public MyModel(String title, String name) {
        this.title = title;
        this.name = name;
    }
    private final String title;
    private final String name;

    public String getTitle() { return title; }
    public String getName() { return name; }
}
```

簡単のためプレーンテキストを生成するJSPの例を示します（**リストB.26**）。ここでのポイントは it という識別子です。Jersey固有の仕様ですが、リソースクラスからフォワードされた先のJSP内でモデルオブジェクトを参照するために識別子 it を使います。it.titleなどでモデルオブジェクトのJavaBeansプロパティの値にアクセスできます。かつ ${...} の形式で値を参照できます。

リストB.26　リストB.24からフォワードされるJSPファイル（view.jsp）

```
This is a forwarded JSP.
${it.title}, ${it.name}
end.
```

リストB.24のJAX-RSアプリにcURLコマンドでアクセスした例を示します。

```
$ curl http://localhost:8080/myjaxrs/my/hello
This is a forwarded JSP.
foo, bar
end.
```

B-3-7　リダイレクト処理

JAX-RSアプリでリダイレクトする実例は**リストB.13**で紹介しました。リダイレクトするには、リソースメソッドの返り値の型をResponseクラスにします。

Responseオブジェクト生成の責務はリソースメソッドです。リダイレクトのためのResponseオブジェクトは、seeOtherメソッドでResponse.ResponseBuilderオブジェクトを生成後、buildメソッドを呼んで生成できます。

seeOtherメソッドの引数にはリダイレクト先URLを指定します。**リストB.13**はサーブレットAPIでURIオブジェクトを生成しましたが､JAX-RSのUriInfo オブジェクトを使う例を**リストB.27**に示します。@ApplicationPath("/rest")のようにJAX-RSのルートパスを "/" 以外にした場合、UriInfoを使ってJAX-RSのベースURL（ルートパスを含むURL）を取得するほうが適切です。

リストB.27　リダイレクト処理するリソースメソッド

```
public Response submit(@Context UriInfo uriInfo) {
    URI uri = uriInfo.getBaseUriBuilder().path("/my/hello").build();
    return Response.seeOther(uri).build();
}
```

B-4　セッション管理

Webアプリの「セッション管理」とは、端的に説明すると、HTTPリクエストがどの利用者からのリクエストかを判別するための仕組みです。

B-4-1　セッション管理が必要な理由

HTTPという通信プロトコルは、1つのリクエストに対し1つのレスポンスが返り、これが1つの単位になります。同じ利用者が同じWebブラウザから同じサーバに新しいリクエストを投げても、前のリクエストと次のリクエストを結びつける情報は（原則は）存在しません。

低レイヤ（TCP/IP）のレベルで見ると、同じPCからのリクエストは同じIPアドレスからのリクエストになります。しかし、同じIPアドレスからのリクエストを同じ利用者からのリクエストと見なすには、2つの理由で無理があります。

1つはNATやプロクシサーバの存在です。WebブラウザとWebサーバの間にNATやプロクシサーバが存在しえます。この場合、サーバには、異なるPCからのリクエストが同じIPアドレスからのリクエストに見えます。

2つ目は、利用者が同じPCを使い続けても、IPアドレスが変わる可能性です。たとえばノートPCや携帯端末を使い移動する場合が1例です。また物理的な移動がなくても、DHCPなどの

動的なIPアドレスの割り当てで利用者のPCのIPアドレスが変わることもあります。

このため、HTTPの世界ではIPアドレスで利用者を区別できません(注13)。

B-4-2　セッション管理の仕組み

セッション管理の基本的なアイディアは、個々のHTTPリクエストにどのWebブラウザ発信かを区別するマークをつける部分です。そのマークを元にサーバ側でリクエストがどのWebブラウザから来たかを判別します。Webアプリ側がマークごとに保持する状態をセッションと呼びます。

セッションに利用者の情報を格納すれば、リクエストと利用者の紐付けが可能です。こうしてリクエストごとに利用者に応じたレスポンスを返せます。

リクエストを区別するマークには、クッキーヘッダの値（以下クッキー）もしくは特別なクエリパラメータを使います。サーブレット規格がクエリパラメータ名をjsessionidとしているので、後者を通称jsessionidパラメータと呼びます。

Webアプリは、これらクッキーもしくはjsessionidパラメータの値を参照して、そのリクエストがどのセッションに属するかを判別します。クッキー、jsessionidパラメータの値のセット方法は後述します。

B-4-3　クッキーによるセッション管理

クッキーの実体はヘッダの1つです。クッキーに関連するヘッダは2つあります。Cookieという名前のリクエストヘッダと、Set-Cookieというレスポンスヘッダです。

Cookieヘッダには他のリクエストヘッダと異なる点があります。Webブラウザがヘッダの値を記憶する点です。Webブラウザは相手サーバごとのクッキー値を記憶します。リクエスト先が同じサーバであれば、記憶しているクッキー値を必ずクッキーヘッダに載せて送ります(注14)。

Cookieヘッダの値で区別できるのはリクエスト元のWebブラウザです。つまり別の利用者が同じWebブラウザを使うと利用者の区別がつきません。企業や学校のPCでも充分に危険ですし、ネットカフェなど不特定多数の利用者が同一のPCを使う環境ではセキュリティリスクになります。またクッキーの値を1度でも盗まれると完全に利用者のなりすましができます。

このリスクを防ぐために、クッキー値にはWebアプリが発行する予測困難な値を使い、かつ寿命を短くします。一般にこの値をセッションIDと呼びます。セッションIDはCookieヘッダ値であると同時に、サーバ側で保持される値でもあります。

（注13）HTTPのkeep-alive機能を使うと、TCP/IPのレベルで接続を維持し続けてHTTPリクエストを投げられます。この場合、同じPCからのHTTPリクエストをTCP/IPレベルで判別可能です。しかし、keep-aliveはHTTPの必須機能ではない点、および同じWebブラウザから複数のHTTPリクエストを同時に投げる場合を考慮すると、同一利用者を判別するために依存はできません。

（注14）厳密には相手サーバとリクエストURLのパスまで含めて対応するクッキー値を記憶します。詳細はHTTPの書籍などを参照してください。

利用者がログアウトをした時あるいは利用者から一定時間リクエストがない時、サーバ側でセッションIDをクリアします。サーバ側のセッションIDの有効期間を限定することで、クッキー値つまりセッションIDを盗まれた時のリスクを減らします(注15)。この件は「セッションタイムアウト値」で説明します。

■ クッキー値のセット方法

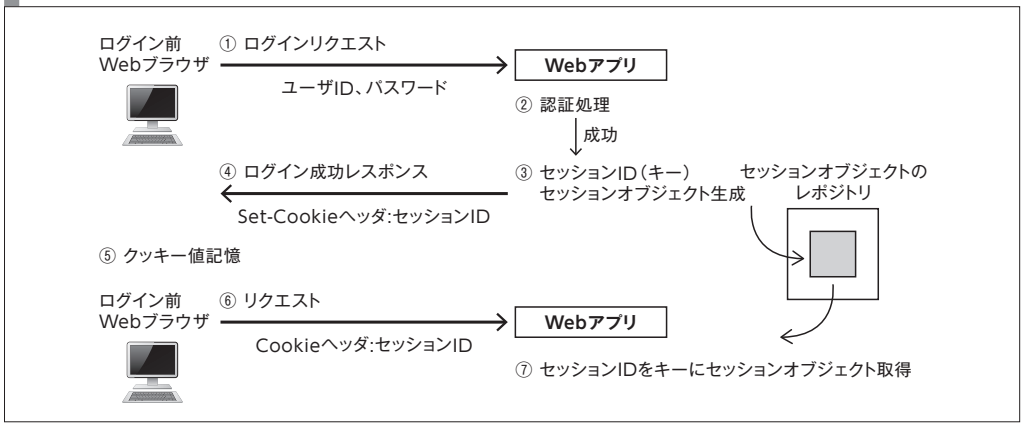
技術的にはCookieヘッダの値をセットする手段は2つ存在します。サーバからのレスポンスのSet-Cookieヘッダを使う方法とJavaScriptを使う方法です。Webアプリのセッション管理では一般に前者を使います。セッションIDの生成がサーバ側の責任だからです。

WebブラウザはサーバからSet-Cookieレスポンスヘッダを受け取ると、Set-Cookieヘッダの値をクッキー値として記憶し、次以降のリクエストのCookieヘッダの値として使います。

■ セッションオブジェクト

通常のWebアプリ開発ではCookieヘッダやSet-Cookieヘッダを直接意識する必要はありません。またセッションIDの値も直接意識しません。代わりにセッションオブジェクト(HttpSessionオブジェクト)を意識します(**図B.6**)。

図B.6　クッキーを使うセッション管理の概念図



プログラムの中でセッションオブジェクト(HttpSessionオブジェクト)取得のAPIを呼んだ時の動作は、リクエスト送信元に対応するセッションが既に存在するか否かで動作が変わります。

セッションIDの有効期間をサーバ側で制御しても、セキュリティリスクはゼロになりません。

(注15) セッションIDの盗難をセッションIDハイジャックと呼びます。内部的には、セッションIDをキーにして、(メモリ上の)レポジトリからHttpSessionオブジェクトを探し出します。

Webアプリはセッション IDをキーにセッションオブジェクトを引けるようになります。セッション IDの値はユニークであればいいので、通常、ランダムに生成します。

サーブレットコンテナは明示的にセッションオブジェクトを破棄するか、もしくはセッションタイムアウト値に達しない限り、生成したセッションオブジェクトを維持します。

既に対応セッションがある場合、セッションオブジェクト取得のAPIは、受信したCookieヘッダの値（セッション ID）から対応するセッションオブジェクトを取得します。

■ セッションオブジェクト取得API

HttpSessionオブジェクト取得のAPIは、HttpServletRequestのgetSessionメソッドです。下記のように2つの形式があります。後者は単なる前者の省略呼び出しで、2つの動作の意味は同じです。

```
// HttpSessionオブジェクトの取得API
HttpSession getSession(boolean create)
HttpSession getSession()      // 上記の引数createにtrueを渡す動作
```

getSessionメソッドを呼ぶと既存HttpSessionオブジェクトを取得できます。内部的には、セッション IDをキーにして、HttpSessionオブジェクトの（メモリ上の）レポジトリから探し出します。

既存HttpSessionオブジェクトがない場合、引数createの値でgetSessionメソッドの動作は変わります。

引数createにtrueを渡すと、新規にセッション IDとHttpSessionオブジェクトを生成して、HttpSessionオブジェクトを返します。同時に、HttpSessionオブジェクトを内部的なレポジトリに記憶し、レスポンスのSet-Cookieヘッダにセッション IDを載せます。

引数createにfalseを渡すと、既存HttpSessionオブジェクトがなければnullを返す動作になります。

ログイン処理と組み合わせたgetSessionメソッドの使い方は後ほど「ユーザ認証」の節で説明します。

C O L U M N

セッション管理とユーザ管理

HTTPのレベルで見ると、セッションに紐付いているのはログインユーザではなくWebブラウザです。HTTPに（後述する）フォーム認証済みのユーザという概念がないためです。ユーザ管理と独立したセッション管理をする場合もありますが、本書はユーザ管理がある（ログイン必須の）Webアプリを前提にセッション管理を説明します。

B-4-4 jsessionidクエリパラメータによるセッション管理

クッキーを使えないWebブラウザがあります。クッキーの代わりにjsessionidパラメータでもセッション管理可能です。HTTPリクエストのURLに必ずセッションIDのクエリパラメータが載るようにします。

クッキーを使うセッション管理で、開発者がクッキーの具体的な値を意識する必要がないように、jsessionidパラメータを使う場合も開発者は値を意識する必要はありません。リクエストURLにjsessionidパラメータがあれば、サーブレットコンテナが自動的にセッションIDとして認識するからです。つまり、セッション管理に関して、クッキーとjsessionidパラメータの利用は透過に扱えます。

jsessionidパラメータを使うセッション管理には、クッキーを使う方法と1つだけ違う点があります。Webアプリが実行中に生成するリクエストURLにjsessionidパラメータを明示的に付与しなければいけない点です。クッキーの場合、セッションオブジェクトを生成するだけで暗黙にクッキーヘッダの読み書きをしてくれますが、クエリパラメータの場合は自前でURLを変更する必要があります。

生成URLにjsessionidを付与するためのAPIが用意されています。HttpServletResponseオブジェクトのencodeURLメソッドです。

```
// HttpServletResponseのencodeURLメソッドの定義
public String encodeURL(String url)
```

jsessionidパラメータのないURL文字列をencodeURLメソッドの引数に渡すと、jsessionidパラメータ（パラメータ名がjsessionid、パラメータ値がセッションID）を付与したURL文字列を返します。

JSTL内で同様のURL書き換えを行うには次のc:urlタグを使います。

```
// JSTLでのURL書き換え（jsessionidパラメータの付与）
<c:url value="url"/>
```

B-4-5 セッションタイムアウト

サーバ側でセッションIDを無効にするまでの時間を「セッションタイムアウト値」と呼びます。

セッションタイムアウト値が長いと、長期間、同じセッションIDを使い続けるので、セッションIDを盗まれる確率が高まります。一方、セッションタイムアウト値が短いと安全になりますが、Webアプリの利用者がすぐにセッションタイムアウトで再ログインを促され、利便性を落とします。利便性とセキュリティリスクの兼ね合いでセッションタイムアウト値を決める必要があります。

セッションタイムアウト値の設定は、web.xmlに次の記述をします。

```
// web.xmlのセッションタイムアウト値の設定
<session-config>
  <session-timeout>30</session-timeout>  <!-- 30分 -->
</session-config>
```

B-4-6　セッション管理の周辺

■ 複数台構成のWebアプリ

パフォーマンスや可用性の向上のため、Webアプリを複数台構成で稼働する場合があります。セッションオブジェクト（HttpSessionオブジェクト）はWebアプリごとのメモリ上の実体なので、同じ利用者からのリクエストが別のWebアプリに割り振られてしまうと、再ログイン状態になってしまいます。

これは実用に耐えないので、Webアプリを振り分けるサーバを配置して対応します(注16)。同じセッションIDのリクエストを常に同じWebアプリが処理するように振り分けます。同じセッションIDが同じWebアプリに張り付くことから、セッションスティッキーなどと呼びます。

■ セッションと状態管理

「状態管理」で説明したように、セッションスコープの属性はHttpSessionオブジェクトの属性になります。この属性にサイズの大きなオブジェクトを格納するのは悪い習慣です。メモリ使用量の限界で同時ログイン可能な利用者の数が低く抑えられてしまうからです。

セッションオブジェクトを小さく保つには、利用者に結び付くオブジェクトは別途キャッシュで管理して、これらのキーのみをセッションオブジェクトで管理します。キャッシュの適切な廃棄処理でメモリ使用量を制御します。

■ クッキーの扱い

本書では説明を省略しますが、セッションID以外の値をクッキー値に含められます。しかしクッキーに利用者の個人情報を載せるのは厳禁です。クッキーの値は秘密情報になりえないことを覚えておいてください。

重要な値をクッキーにそのまま載せるのは厳禁ですが、サーバ側で暗号化したクッキー値であれば、セッション管理の1つの選択肢になります。

こうしてクッキー値のみでセッション管理をすると、先ほど紹介したセッションスティッキーなどの仕組みが不要になります。サーバ側に状態を持つ必要がなくなるからです。このような構成のWebアプリをステートレスと呼びます。

（注16）多くの場合、ロードバランサがこの役割を兼ねます。

暗号化して安全にしても、クッキー値に情報を載せすぎるのは別の問題を発生させます。全リクエストに載るCookieヘッダが通信のオーバーヘッドになるからです。また、そもそもクッキー値にはWebブラウザごとに最大長が決まっています。

これらの制約の下でのステートレスWebアプリの現実解の1つの形は、サーバ側の状態をWebアプリの外側に配置したデータストア（キャッシュなど）に持たせ、これら状態を問い合わせるキーを暗号化クッキー値に載せる手法です。

■ ユーザ認証

多くのWebアプリはセッションをWebブラウザとではなくログインユーザと紐付けます。利用者にログインしてもらいユーザを識別する処理を認証と呼びます。

既に説明したHttpServletRequestのgetSessionメソッドを使うと、ログイン処理およびユーザが既にログイン済みかをチェックするロジックを実現できます。

利用者にユーザ名とパスワードをフォームに入力してもらう前提とします。ログイン処理の詳細は省略しますが、ログイン処理を行うloginメソッドがあると仮定します。loginメソッドはログインに成功すると内部的なユーザIDの数値を返し、失敗するとnullを返す仕様とします。この仕様を前提にすると、**リストB.28**のようなコードを書けます。ログインに成功すると、HttpSessionオブジェクトを新規生成し、セッションスコープにユーザIDを記録します。

リストB.28　ログイン処理を呼び出すコード例

```
Integer id = login(name, password); // ログイン成功するとユーザIDを返す
if (id != null) { // ログイン成功
    HttpSession session = req.getSession(true); // HttpSessionオブジェクト生成
    session.setAttribute("id", id);             // セッションスコープにユーザIDを記録
    resp.sendRedirect(ログイン成功時の画面);
} else {          // ログイン失敗
    resp.sendRedirect(ログイン画面); // リトライ
}
```

リストB.29のような処理で認証済みかをチェックします。HttpSessionオブジェクトの存在チェックをして、セッションスコープにユーザIDがあるかを確認します。このような処理を認証が必要なすべてのURLに対して実施します。普通のリクエスト処理の前に割り込む必要があるので、フィルタなど共通的な場所で実施する必要があります。

リストB.29　認証済みであればtrue、認証済みでなければfalseを返す処理

```
boolean isAuthenticated(HttpServletRequest req) {
    HttpSession session = req.getSession(false);
    return session != null && session.getAttribute("id") != null;
}
```

付録C  データベース

Webアプリの多くはデータ保存のためにデータベースを利用します。データベースの中でも一般的なのがRDBです。最初にJavaでデータベースを扱う基本APIであるJDBCを説明します。その後、Java EEコンテナにデータベース管理を任せて、データベース処理をデータソースとして抽象化する説明をします。

C-1 RDB概論

C-1-1 RDBとは

厳密さを犠牲にして直感的な用語でRDB（リレーショナルデータベース）の概略を説明します(注1)。

本章を読む上では、RDBをただのデータの容物と考えてください。

RDBは複数のテーブルを構成要素とするデータベースです。各テーブルは行と列で構成されます。テーブルの行をレコードと呼び、列をカラムと呼びます。RDBに馴染みがなければ、テーブルはExcelの表のようなものを想像してください。Excelの表は利用者が画面上から読み書きするのが普通ですが、RDBのテーブルは一般にプログラムから読み書きします。

RDBの特徴はテーブルの読み書き操作を演算として定義する点です。あらゆるテーブルへの操作結果がテーブルになります(注2)。テーブル同士に対する加算や乗算に相当する処理もあります。特にテーブル同士の乗算に相当する結合処理は重要です。

テーブルに対する操作は、一般にSQLと呼ばれる言語で行います。Javaプログラムからテーブルを操作する時もSQLを使います。

個々のJavaプログラムがRDBのデータベースを直接操作してもかまいませんが、一般にRDBを扱う専門プログラムを別途用意します。そんな専門プログラムをRDBMS（RDB管理システム）と呼びます。Webアプリなどのプログラムは、RDBMSと通信してRDB処理をします。

（注1）　本書はRDBやSQLに関して一定の知識がある読者を想定しています。必要な知識はRDBの専門書を読んで補ってください。
（注2）　数値同士の加算の結果が必ず数値になるように、テーブルへの操作結果は必ずテーブルになります。

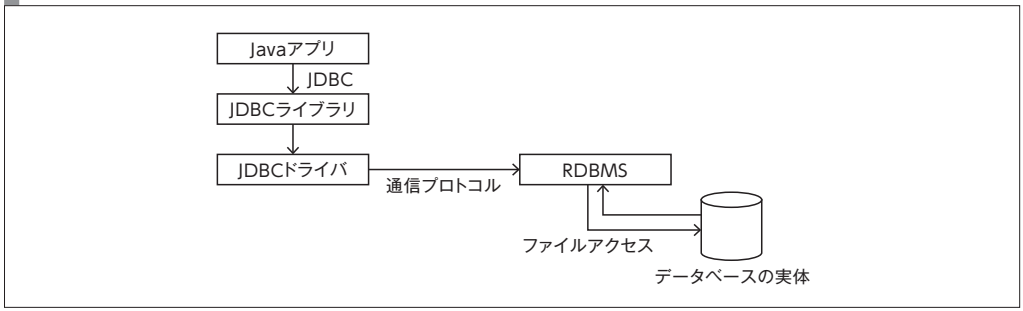
C-1-2 JDBCとは

JavaプログラムからRDBのテーブルへのアクセス方法やSQLをコマンドとして送信する仕組みを定めたAPI規格がJDBCです。

JDBCのAPIのほとんどはインターフェースです。各RDBMSごとにJDBCのインターフェースを実装したクラスライブラリが存在します。このようなクラスライブラリをJDBCドライバと呼びます（**図C.1**）。

開発者は利用するRDBMSに応じて適切なJDBCドライバを自分のプログラムから使えるように設定します。JDBCのインターフェースに依存するJavaコードを書いていれば、JDBCドライバを変更してもコードを変更する必要がなくなります。つまり、JDBCのAPIおよび標準的なSQLを使って書いたJavaコードは、RDBMSの違いを意識する必要がなくなります(注3)。

図C.1 RDBMSとJDBCドライバ



C-1-3 Java DB（Derby）

本書はRDBMSの実装としてJava DBを使います。Java DBはJDK6以降に標準で付属するRDBMSです。Java（JDK）をインストールすると特別なインストールなしに使い始められます(注4)。またGlassFishにもJava DBが付属しています。

Java DBは、組み込み型とクライアントサーバ型の両方のモードで動作可能です。組み込み型にすると、Javaプログラムによってロードされた JDBCドライバがデータベースの実体であるファイルを直接読み書きします。一方、クライアントサーバ型を使うと、Javaプログラムが（別途動作する）Java DBサーバと通信します。サーバを別マシンで動かす場合も多いため、クライアントサーバ型をネットワーク型と呼ぶこともあります。

（注3） 現実は、RDBMSごとにSQLに微妙な方言があるのが実状です。JDBCはSQLを生で書くAPIなので、RDBMSの違いを完全には隠蔽できません。JPAを使うとSQLの違いを完全に隠蔽できますが、今度はRDBMS固有のリッチな機能を使えなくなる欠点があります。

（注4） Java DBはApache Software Foundationが開発元のDerbyをJDKに標準添付したものです。

本書の説明ではクライアントサーバ型を使います。ただし、JDBCのAPIが2つのモードの違いを隠蔽するので、組み込み型モードに変更してもJavaコードの変更は不要です。

C-1-4　データベースの準備

■ 初期化の手順

RDBを使うプログラムを開発するには、データの容物であるデータベースとテーブルを作る必要があります。最初にデータベースを作り、次にデータベースの中にテーブルを作成します。

データベースを使う準備は最低限次のようになります。

① データベースの作成
② データベース内にテーブルを作成（必要に応じて複数のテーブルを作成）
③（必要であれば）初期データをテーブルに書き込む

上記すべてをJavaプログラムから行うことも可能です。しかし本書ではJava DBの付属コマンドijで実施します。ijコマンドは、Java（JDK）のインストールパスを$JAVA_HOMEと表記すると、$JAVA_HOME/db/bin/ij に存在します。

以降のコード例を動かすため、次のように環境変数DERBY_HOMEを設定してください。

```
// Unix系OS
$ export DERBY_HOME=$JAVA_HOME/db

// Windows
% set DERBY_HOME=%JAVA_HOME%¥db
```

Java DBをクライアントサーバ型で使う場合、サーバ用のプロセスを起動する必要があります。環境変数DERBY_HOMEを設定した場合、$DERBY_HOME/bin/startNetworkServerコマンドで起動できます。Java DBサーバプロセスは実行カレントディレクトリ以下にデータベースファイルを作成するので、書き込み可能なディレクトリで下記を実行してください。

```
// Unix系OS
$ $DERBY_HOME/bin/startNetworkServer -h 0.0.0.0

// Windows
$ %DERBY_HOME%¥bin¥startNetworkServer -h 0.0.0.0
```

引数なしでサーバを起動すると同一マシンのプロセスからのみ接続を受けつける状態で起動します。ネットワーク経由で別マシンからの接続を受けつけるには、上記のようにコマンドライン引数-hを使います。また待ち受けポート番号をデフォルトの1527番から変更したければ、コマ

ンドライン引数-pで指定します。

Java DBサーバプロセスを起動後、ijコマンドでサーバプロセスに接続します。

```
// Unix系OSでのijコマンド起動
$ $DERBY_HOME/bin/ij

// Windowsでのijコマンド起動
% %DERBY_HOME%¥bin¥ij
```

ijコマンドを実行するとコマンド待ち状態になります。**図C.2**の ij> の後の入力例を入力してください。これでデータベースの初期化を実施できます。

図C.2　ijコマンドのコマンドプロンプトでの操作例

```
// 暗黙にデータベースを作成（データベース名はmydb。任意の名前を指定可能）
ij> connect 'jdbc:derby://localhost:1527/mydb;create=true;user=app;password=app';
// テーブルを作成（articleテーブルとcommentテーブル）
ij> CREATE TABLE article (id bigint primary key generated by default as identity, title
varchar(256), body long varchar, updated_at timestamp);
ij> CREATE TABLE comment (id bigint primary key generated by default as identity, article_id
bigint not null, title varchar(256), body long varchar, updated_at timestamp, constraint fk
foreign key (article_id) references article(id));
// レコードをテーブルに挿入
ij> INSERT INTO article VALUES (default, 'title1', 'body1', current_timestamp);
ij> INSERT INTO article VALUES (default, 'title2', 'body2', current_timestamp);
// 終了
ij> exit;
```

■ijコマンドの内部命令

ijコマンドの内部命令の一覧はコマンドプロンプトからhelp;を実行すると表示できます。便利な命令を**表C.1**に紹介します。

表C.1　ijコマンドの便利な命令

命令名	説明	具体例
connect	データベースに接続	本文のコマンド例を参照
describe	テーブル定義などを表示	describe article;
show	テーブル一覧などを表示	show tables;
run	命令やSQLを記述したファイルを実行	run 'filename';

■ データベース作成と接続

ijのconnect命令はデータベースに接続する命令です。connect命令にはデータベース接続URLを引数で渡します。接続URLはRDBMSごとに命名規則が決まっています（**表C.2**）。詳しくは個々のRDBMSのマニュアルを調べてください。

表C.2　データベースの接続URLのフォーマット

RDBMS名	データベースURL
クライアントサーバ型Java DB	jdbc:derby://ホスト名（:ポート番号）/データベース名（;オプション）
組み込み型Java DB	jdbc:derby:データベース名（;オプション）
MySQL	jdbc:mysql://ホスト名（:ポート番号）/データベース名

普通のconnect命令は、存在しないデータベース名を指定すると接続に失敗します。

URL文字列に;create=trueオプションを付与すると、データベースが存在しない場合にデータベースを新規に作成します。つまり、connect命令はデータベース接続と同時に、データベース作成命令も兼ねています。なお、既に存在するデータベースに;create=trueオプションでconnectしても、既存データベースはそのままです。このため、常に;create=trueオプション付きでconnectしても問題ありません。

user=appとpassword=appのオプションは認証用オプションです。特別な設定をしないJava DBは認証なしでアクセス可能なので、半分飾りです。しかし、user=appオプションの指定値がデータベース内でテーブルを分類するスキーマに対応するので、忘れずに指定してください。

■ テーブル作成

CREATE TABLE命令はテーブル作成を指示するSQLです。**図C.2**の例ではarticleという名前のテーブルを作成しています。次のようなカラム（列）を持つテーブルを定義しています。

```
|id|title|body|updated_at|
```

INSERT命令はテーブルにデータ（レコード）を挿入するSQLです。**図C.2**のINSERT命令で次のような状態になると考えてください。

```
|id|title |body  |updated_at              |
|1 |title1|body1 |2014-06-29 16:19:55.757|
|2 |title2|body2 |2014-06-29 16:20:55.029|
```

このようにデータベース初期化処理を別途実施しておくと、データベースを使うJavaプログラムのコードが単純になります。

C-2 JDBC

C-2-1 JDBCの概要

JDBCはJavaプログラムからRDBを使う時の標準APIです。java.sqlとjavax.sqlの2つのパッケージで提供しています。

JDBCは主に次の機能を提供します。

- RDBMSへの接続
- RDBのレコード検索（問い合わせ）
- RDBのレコード更新

JDBCを使うコードの典型的な構成を下記に示します。

① JDBCドライバをロード（データソースオブジェクトを使う間接的ロードも含む）
② java.sql.Connectionオブジェクトを生成（ファクトリメソッド経由）
③ Connectionオブジェクトからjava.sql.Statementオブジェクト取得
④ Statementオブジェクトのメソッド呼び出しでSQLを実行。この時、SQL文字列を引数などでStatementオブジェクトに渡す
⑤ 上記メソッドの返り値で検索結果のResultSetオブジェクトを取得
⑥ ResultSetオブジェクトから必要な検索結果を取得
⑦ 必要なクローズ処理を実施

C-2-2 RDBMSへの接続

自作JavaプログラムでRDBを使う最初の1歩は、JDBCドライバのロードです。

JDBCドライバをロードする手段は、直接ロードする方法とデータソースオブジェクト（javax.sql.DataSource）を経由する方法の2種類存在します。

データソースオブジェクトを使うと、JDBCドライバのロードとConnectionオブジェクトの管理を隠蔽できます。Java EEアプリ開発などコンテナを使う場合、データソースオブジェクトの利用が一般的です(注5)。

JDBCドライバをロード後、java.sql.Connectionオブジェクトを取得する必要があります。取得方法は後ほど説明します。

ConnectionオブジェクトはJDBCのAPI呼び出しのエントリポイントです。ネットワークプ

(注5) データソースオブジェクトの利用は「**C-3 Webアプリとデータベース処理**」で説明します。

ログラミングでデータ通信の端点としてソケットが存在するように、JDBCプログラミングでRDBMSとやりとりする端点がConnectionオブジェクトです。

Connectionオブジェクトの内部動作は、クライアントサーバ型と組み込み型で異なります。前者は通信をし、後者はファイルへの直接の読み書きをするからです。しかし、この辺の実装詳細を気にせず、Connectionオブジェクトはデータベース接続を抽象化した操作を提供します。

■JDBCドライバのロード

Java DBのJDBCドライバの実体はderby.jarとderbyclient.jarです。前者が組み込み型、後者がクライアントサーバ型用のドライバです。

自作のJavaプログラムを動かす時、クラスパスにJDBCドライバファイルが存在すれば自動的にロードするようになっています(注6)。内部的には、DriverManagerクラスのgetConnectionクラスメソッド呼び出し時に自動ロードします。

JDBCドライバの自動ロード機能はJDBC4.0以降の機能です。JDBCドライバの自動ロードが使える条件は、Java6以降のJavaを使い、JDBC4.0以降に対応したJDBCドライバを使っている場合です。

JDBC3.0以前は明示的にJDBCドライバをロードする必要がありました。

Java5以前のJavaもしくは古いJDBCドライバを使っている場合のために、JDBCドライバを明示的にロードするコードを紹介します(**リストC.1**)。Classクラスのリフレクション機能を使います。やや奇妙なコードですが、これが昔のJDBCドライバのロード方法の慣習でした。

リストC.1　JDBCドライバの明示的なロード(Java6以降は不要なコード)

```
// クライアントサーバ版Derby(Java DB)のJDBCドライバの明示的なロード
Class.forName("org.apache.derby.jdbc.ClientDriver")

// 組み込み型Derby(Java DB)のJDBCドライバの明示的なロード
Class.forName("org.apache.derby.jdbc.EmbeddedDriver")
```

■Connectionオブジェクトの取得

Connectionオブジェクトのファクトリメソッドに相当するのが、java.sql.DriverManagerクラスのgetConnectionメソッドです(**リストC.2**)。

getConnectionメソッドの第1引数はデータベースの接続URLです(**表C.2**参照)。第2、第3の引数がRDBMSへの接続ユーザIDとパスワードです。それぞれ文字列で渡します。**図C.2**に合わせて"app"を渡していますが、認証設定をしていないのでnullを渡しても動作します(注7)。

(注6)　jarファイルのサービスプロバイダ機能を使い自動的にJDBCドライバをロードします。

(注7)　RDBMSの多くは認証ユーザIDを内部的な分類単位に兼用します。この分類単位はしばしばスキーマと呼ばれます。デフォルト設定のJava DBにユーザ認証はありませんが、スキーマという仕組みはあります。このため、データベース接続時に(認証に不要であっても)ユーザIDを渡すほうが何かと困りません。

getConnectionメソッドは接続URLからロード済みのJDBCドライバを検索します。ロード済みでなければJDBCドライバをロードします。該当するJDBCドライバをロード後、RDBMSに対して接続をして、Connectionオブジェクトを返します。

リストC.2　JDBCドライバのロードとConnectionオブジェクト取得の雛型

```
import java.sql.*;// 後述のコード例では紙幅の節約のため、import文の記述を省略します

public class MyDerby {
    public static void main(String... args) {
        try (Connection conn = DriverManager.getConnection("jdbc:derby://localhost:1527/mydb",
"app", "app");
             Statement stmt = conn.createStatement()) {
            [JDBC利用コードをここに書きます]
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

リストC.2のコードの実行時、JDBCドライバのファイルをクラスパスに見つける必要があります。たとえば次のようにjavaコマンドの-cpオプションでクラスパスを指定してください。

```
$ java -cp "$DERBY_HOME/lib/derbyclient.jar:." MyDerby
あるいは
$ java -cp "$DERBY_HOME/lib/*:." MyDerby　（コラム参照）
```

■ Connectionオブジェクトの破棄（クローズ）

Connectionオブジェクトは、使い終わった後にcloseメソッドでクローズします。ConnectionクラスはAutoCloseableインターフェースを実装しているので、Java7以降であればtry-with-

C O L U M N

Java6以降のクラスパスの指定方法

Java6以降、クラスパスの指定にワイルドカード指定ができます。次のように環境変数CLASSPATHもしくはjavaのコマンドライン引数の-cpに指定してコンパイルおよび実行ができます。

```
$ CLASSPATH='.:lib/*' java Main
あるいは
$ java -cp '.:lib/*' Main
```

resources文を使えます。Java6以前であればfinally節のイディオムで確実にクローズしてください。クローズを忘れるとRDBMSとの接続を占有したままになり大きな問題になるからです。

後述するStatementクラスとResultSetクラスもAutoCloseableインターフェースを実装しています。JDBC用のクラスは原則try-with-resources文でクローズすると覚えてください。

■ SQLException

JDBC処理で起きる例外はSQLExceptionクラスおよびその継承クラスです。検査例外なので必ず捕捉してください。

C-2-3　問い合わせ処理

JDBCによる問い合わせ処理の説明をします。条件に合致するレコード群をRDBから取得する処理です。問い合わせ処理はクエリ処理とも言います。

問い合わせ処理をするには、ConnectionオブジェクトからStatementオブジェクト（PreparedStatementやCallbaleStatementなど派生型も含む）を取得する必要があります。StatementオブジェクトはConnectionオブジェクトのcreateStatementメソッドで取得します。このメソッドをStatementオブジェクトのファクトリメソッドと考えてください。

問い合わせSQLを実行するにはStatementオブジェクトのexecuteメソッドもしくはexecuteQueryメソッドを呼びます。executeメソッドを使う必要性はほとんどないので、本書の説明はexecuteQueryメソッドに限定します。

executeQueryメソッドの引数にSQLを文字列のまま渡すとSQLを実行できます。SQLの実行結果はResultSetオブジェクトで受け取ります。

リストC.2の雛形コードに、問い合わせ処理を書き足したコード例を示します（**リストC.3**）。

リストC.3　JDBCを使う問い合わせ処理

```
public class MyDerby {
    public static void main(String... args) {
        try (Connection conn = DriverManager.getConnection("jdbc:derby://localhost:1527/mydb",
"app", "app");
             Statement stmt = conn.createStatement()) {
            String sql = "SELECT id, title, body FROM article ORDER BY updated_at DESC";
            try (ResultSet rs = stmt.executeQuery(sql)) {
                while (rs.next()) {
                    int id = rs.getInt("id");
                    String title = rs.getString("title");
                    String body = rs.getString("body");
                    System.out.println(id + " " + title + " " + body);
                }
            }
```

```
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

StatementのexecuteQueryメソッドの定義は次のようになっています。

```
ResultSet executeQuery(String sql) throws SQLException
```

引数に渡すSQLが不正な文字列であってもコンパイル時にチェックできません。実行時に例外が起きるだけです。SQLの妥当性を検証するのは開発者の責任です。

返り値のResultSetは、問い合わせ結果を表現するオブジェクトです。

■ ResultSetオブジェクト

ResultSetクラスは概念的にはList<Row>です(注8)。Row型は実際には存在しませんが、テーブルの1つの行を表す型だと考えてください。Row型は概念的にはList<Column>です。Column型も実際には存在しませんが1つの列を表す型だと考えてください。Column型は概念的には、列名と列型と列値の3つのフィールドを持つクラスに相当します。

現実のResultSetはここまで構造化されていませんが、概念上はこのようなものだと考えてください。

ResultSetは問い合わせ結果すべてをメモリに読み込むわけではありません。代わりに、ResultSetオブジェクトは内部にカーソルを持ちます。カーソルは特定の行を指し、カーソルを進めることで行単位の読み込みをします。カーソルを前に戻すことも可能ですが、非効率になるので一般的には非推奨です。

カーソルは最初にResultSetの先頭行の1つ前を指します。ResultSetのnextメソッドでカーソルが1行進みます。ResultSetをコレクションに見立てると、nextメソッドはイテレータを進める操作に相当します。

ResultSetオブジェクトは、カーソルが現在指している行の列値を取り出すメソッドを提供します。列値を取り出すメソッドの定義を一部引用します。便宜上、これらをgetメソッドと呼びます。

```
// ResultSetのgetメソッドの定義（一部）
String getString(int columnIndex) throws SQLException
String getString(String columnLabel) throws SQLException
int getInt(String columnLabel) throws SQLException
int getInt(int columnIndex) throws SQLException
```

（注8） 原則論で言うとRDBの行同士の間に順序の概念はありません。つまり概念上はSet<Row>のほうが適切です。しかし実用上、ResultSetはList<Row>に近いデータ型です。

getメソッドの引数には列インデックス値もしくは列名を指定します。列インデックス値は1から始まる整数です（0からではないので注意してください）。

メソッド名からわかるように列値の型ごとにメソッドが存在します。開発者は列値の型に合う適切なメソッドを呼ぶ責任があります。RDBの型とJavaの型の対応関係は後述します。

複雑なSQLの場合、列名は曖昧になる危険があります。たとえば次のSQLを考えます。

```
String sql = "SELECT article.id, comment.id, article.title, comment.title FROM article INNER
JOIN comment ON article.id = comment.article_id";
ResultSet rs = stmt.executeQuery(sql);
```

列名は先頭から順に"id"、"id"、"title"、"title"になります（ResultSetオブジェクトのgetMetaDataメソッドで確認可能です）。同名の列名の場合、getメソッドは先に見つけた列値を返します。列名が曖昧な場合、SQLで別名を定義してください。**リストC.4**のようにすると、列名が先頭から順に"aid"、"cid"、"atitle"、"ctitle"となり曖昧さがなくなります。

リストC.4　列名の曖昧さを回避

```
String sql = "SELECT article.id aid, comment.id cid, article.title atitle, comment.title ctitle
FROM article INNER JOIN comment ON article.id = comment.article_id";
ResultSet rs = stmt.executeQuery(sql);
int aid = rs.getInt("aid");
int cid = rs.getInt("cid");
String atitle = rs.getString("atitle");
String ctitle = rs.getString("ctitle");
```

カーソルを行番号で移動できる手段も存在しますが利用は推奨しません。なぜなら、RDBのテーブルは原則として行番号を意識すべきではないからです。nextメソッドで1行ずつ進めて処理してください。nextメソッドの返り値の型はbooleanです。最終行でnextメソッドを呼ぶと偽が返ります。この性質を利用して、返り値が真の間nextメソッドを呼ぶwhileループにするのが定石です。

ResultSetに対して更新処理（行削除を含む）を行う手段も存在します。しかし本書では説明を省略します（SQLによる更新を推奨します）。

■ ResultSetとDTO

ResultSetオブジェクトをそのまま使い、アプリケーションコード内でデータを運ぶオブジェクト（いわゆるDTO）に、原理上はできます。しかしこれは良い習慣ではありません。クローズするまで、ResultSetオブジェクトはデータベース内の余計なリソースを消費するからです。

またResultSetは汎用データ型すぎてアプリにとって必ずしも扱いやすいデータ形式ではありません。ResultSetオブジェクトを取得した場合、速やかにアプリ側で定義した適切なデータ型

へ変換して、ResultSetオブジェクトをクローズしてください(注9)。

必要な行の絞りこみをSQLで行うか、Java側のコードで行うかは難しい問題です。一般的にはSQLで絞りこんで小さなResultSetを得るほうが効率的です。ただし例外もありますし、またSQLだけで完全に絞りこむのが不可能な場合もあります(SQLからJava側のオブジェクトを参照したりメソッドを呼んだりはできないと考えてください)。現代の大規模アプリが抱える最大の難問の1つです。

■ クローズ処理

StatementもResultSetも、AutoCloseableインターフェースを実装してクローズ処理を持ちます。

Connectionオブジェクトと違いStatementのクローズ処理は必須ではありません。クローズしなくても通常のGCで回収されるからです。しかし早めにクローズするとメモリ効率が上がるので、可能であればcloseメソッドを呼んでください。

ResultSetオブジェクトのクローズも必須ではありません。ResultSetオブジェクトはStatementオブジェクトごとに1つだけ存在し、Statementオブジェクトが破棄されると一緒に破棄されるからです。しかしResultSetもクローズして損はないのでtry-with-resources文でクローズする癖をつけてください。

■ RDBの型とJavaの型

RDBの列には型があります。RDBの型とJavaの型は必ずしも1対1に対応しませんが、JDBCが型の違いをなるべく隠蔽します。主な型の対応表を**表C.3**に示します。

表C.3　Javaの型とRDBの型の対応表

Javaの型	RDBの主な型	ResultSetのメソッド
String	CHAR、VARCHAR、LONGVARCHAR	getString
byte[]	BINARY、VARBINARY、LONGVARBINARY	getBytes
boolean	BIT	getBoolean
short	SMALLINT	getShort
double	DOUBLE、FLOAT	getDouble
int	INTEGER	getInt
long	BIGINT	getLong
float	REAL	getFloat
java.sql.Date	DATE	getDate
java.sql.Time	TIME	getTime
java.sql.Timestamp	TIMESTAMP	getTimestamp
java.sql.Blob	BLOB	getBlob
java.sql.Clob	CLOB	getClob

(注9)　この説明は「データベースアクセスとアプリ側のドメインロジックをなるべく疎に保つほうが良い」という思想の下に書いています。これは1つの思想であり、絶対的真理ではない点を付記しておきます。

JDBCは可能な範囲で列値の型変換を試みます。たとえば**リストC.3**のrs.getInt("id")をrs.getString("id")に書き換えても、数値から文字列への型変換が起きて問題なく動作します。

一方、rs.getString("title")をrs.getInt("title")に書き換えると、列値がたまたま"123"の文字列であれば数値への型変換が成功しますが、通常の文字列であれば実行時に例外が起きます。この動作に依存するコードはたまたま動くだけなので、潜在バグと考えてください。

RDBの列値にはnullがあります。Javaのnullに近い存在で、空文字とも0ともfalseとも違う列値です。しかし、getIntなど数値を返すgetメソッドはnull値を0に、getBooleanはnull値をfalseに自動変換します。この自動変換は潜在バグを隠す危険があるので注意してください。列値がnullであるかを確認するには、ResultSetのwasNullメソッドを使います。

■ 集約処理

RDBには集約処理と呼ばれる問い合わせ処理があります。結果の行数や列値の合計を求めたりできます。JDBCは集約処理の結果もResultSetで扱います。

たとえば問い合わせ結果の行数を知りたいとします。次のようにSQLのcount集約関数で計算できます。getメソッドに列名を渡す場合、別名をつけてください。

リストC.5　データベースの集約処理

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT count(*) cnt FROM article");
rs.next(); // 直感的ではないですが、結果行数取得のためにnext()呼び出しが必要です
System.out.println(rs.getInt("cnt")); // =>行数を表示
// System.out.println(rs.getInt(1));  // 上記は列インデックスでも動きます
```

C-2-4　更新処理

RDBの更新処理は挿入（insert）、変更（update）、削除（delete）の3つあります。JDBCは同じAPIで3つの違いを隠蔽します。

更新処理にはStatementオブジェクトのexecuteUpdateメソッドを使います。executeUpdateメソッドの定義は次のようになっています。

```
int executeUpdate(String sql) throws SQLException
```

引数に更新SQLを文字列のまま渡します。返り値は更新SQLによって影響を受けた行数です。たとえば削除操作であれば削除された行数が返ります。

具体例を**リストC.6**に示します。小文字のtで始まるtitleを一括して"T"に変更する更新処理です。

リストC.6　データベースの更新処理

```
Statement stmt = conn.createStatement();
String sql = "UPDATE article SET title = 'T' WHERE title LIKE 't%'";
int ret = stmt.executeUpdate(sql);
System.out.println("ret = " + ret);  // 変更のあった行数
```

テーブル作成やテーブル削除などのSQLもexecuteUpdateメソッドに渡せます。テーブル内のデータ（レコード）に対する操作とテーブルそのものに対する操作はレベルの異なる操作ですが、JDBCは同じAPIで違いを隠蔽します。

C-2-5　PreparedStatement

PreparedStatementという仕組みがあります。あらかじめパラメータ付きSQLをRDBMSに記憶させておき、変更の可能性のあるパラメータ値を後から送って実行する仕組みです。

PreparedStatementを使う理由は主に次の2つです。

- **SQLを使いまわすことによる効率性の向上**
- **エスケープ処理をRDBMSに任せることによる脆弱性の回避（SQLインジェクション対策）**

PreparedStatementの使い方を説明する前に、SQLインジェクションを説明します。

■ SQLインジェクション

Webアプリでは、利用者の入力（リクエストのクエリパラメータなど）を使ってSQL文を組み立てる必要のある場合があります。**リストC.7**のように文字列の結合でSQLを組み立てたとします。

リストC.7　文字列結合によるSQL組み立て

```
String title = req.getParameter("title"); // HttpServletRequest reqを前提
Statement stmt = conn.createStatement();
String sql = "DELETE FROM article WHERE title = '" + title + "'";
stmt.executeUpdate(sql);
```

クエリパラメータtitleの値が適切であれば、適切なSQL文字列になり、想定どおりの行だけを削除できます。しかし、Webブラウザの利用者はどんな値でもリクエストに送信できるので、いつも想定どおりのSQLになるとは限りません。たとえば、クエリパラメータtitleの値が "' or title is null or title is not null or title = '" という文字列だったとします。この文字列を使って組み立てたSQLは事実上すべての行を削除してしまいます。

このようなクエリパラメータを送りつけて危険なSQLを発行させる攻撃手法をSQLインジェクションと呼びます。

SQLインジェクションを防ぐには適切なエスケープ処理が必要です。たとえば、'（シングルクォート文字）を''（シングルクォート文字2つ）にエスケープするなどです。しかし、このような手動エスケープ処理は非効率で、かつエスケープ洩れの発生可能性が非常に高くなります。

PreparedStatementを使うと、手動のエスケープ処理の代わりにRDBMSにエスケープ処理を任せられます。RDBMSは実行対象のSQL構文を完全に把握した上でエスケープするので、原理上、漏れが発生しません。

PreparedStatementを使うとほとんどのSQLインジェクションを防止できます。しかし残念ながらすべてではありません。代表的な例がIN句です。IN句は実用上よく使うSQLの構文ですが、並べるパラメータ数を確定できないのでPreparedStatementと相性が悪い構文です（パラメータ数を確定できればPreparedStatementを利用可能です）。この場合、自前でエスケープ処理をする必要があります。

■ PreparedStatementの使い方

PreparedStatementオブジェクトは、ConnectionオブジェクトのprepareStatementメソッドで取得できます。prepareStatementメソッドの引数にSQL文字列（の雛形）を渡します。この引数にエスケープ処理は行われないので注意してください。未検証の入力値をそのまま使ってはいけません。

引数のSQL文字列にはバインドパラメータ（プレースホルダー）を持たせられます。具体例を**リストC.8**に示します。バインドパラメータは?文字で指定します。

リストC.8　PreparedStatementの利用例

```
public class MyDerbyInsert {
    public static void main(String... args) {
        String sql = "UPDATE article SET title=?, body=?, updated_at=current_timestamp WHERE
id=?"; // バインドパラメータを含むSQL文字列（の雛形）
        try (Connection conn = DriverManager.getConnection("jdbc:derby://localhost:1527/mydb",
"app", "app");
             PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setString(1, "title1");
            stmt.setString(2, "body1");
            stmt.setInt(3, 1);
            stmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

バインドパラメータ（?文字）の部分に実値を当てはめて最終的なSQL文字列を組み立てるのがPreparedStatementを使うコードの肝です。この実値による置換の時、適切なエスケープ処理を

自動的に行うので、SQLインジェクションの心配が不要になります。

バインドパラメータの置換には、PreparedStatementオブジェクトのsetIntやsetStringメソッドを使います。便宜上、これらのメソッドをsetメソッドと呼びます。

setメソッドの第1引数にはバインドパラメータのインデックス値を指定します。1から始まる整数です。たとえば、**リストC.8**のsetInt(1, "title")は、1番目の?文字を文字列"title1"で置き換えます。

バインドパラメータにセットした実値は自動的にエスケープされます。パラメータ値が文字列であれば文字列はシングルクォート文字で囲まれ、文字列中のシングルクォート文字はエスケープされます。LIKE演算子を使う場合、LIKE演算子の特殊文字('%'と'_')はそのままの意味を持つので注意してください。LIKE演算子を使う例を**リストC.9**に示します。

すべてのバインドパラメータに実値をセット後、executeQueryもしくはexecuteUpdateでSQLを実行します。SQL実行後の処理はStatementを使う場合と同じです。たとえば問い合わせ系のSQLであれば、結果をResultSetで得られます。

バインドパラメータをインデックス値ではなく文字列で指定するメソッドは存在しません。名前指定を可能にする実装は上位APIで提供されています(Spring JDBCやJPAなど)。

リストC.9　LIKE演算子の利用例

```
String sql = "SELECT id, title, body, updated_at FROM article WHERE title like ?";
PreparedStatement stmt = conn.prepareStatement(sql);
stmt.setString(1, "t%"); //=> '%'はSQLのLIKE演算子の特殊文字で、そのままSQLに渡ります
ResultSet rs = stmt.executeQuery();
(ResultSetを使うコードは省略)
```

PreparedStatementはバインドパラメータと共に使うことが普通ですが、バインドパラメータがなくてもPreparedStatementを使えます。何度も同じSQLを呼ぶ場合、高速化する可能性があるので利用を検討してください。

■ addBatch処理

多くのWebアプリでSQL処理時間が全体の応答性能のボトルネックになりがちです。このため、現場では少しでもSQL処理時間を減らすための工夫が求められます。

適切に記述すれば、多くのSQLの実行時間は1つ当たりで見れば数ミリ秒以下です。しかしSQLの実行数が増えると処理時間は増えていきます。

SQL処理時間を削減する工夫の1つにaddBatch処理があります。複数の更新SQLをまとめて実行できる仕組みです。個々のSQL処理時間は変わらないので、重いSQLを書いた場合の効果は薄いですが、軽めのSQLを多数実行する場合、ネットワーク処理などのオーバーヘッド軽減により速度改善効果があります。

addBatch処理はStatementとPreparedStatementのどちらにでも使えます。**リストC.10**にPreparedStatementのaddBatch利用例を示します。

リストC.10　addBatch処理の例

```
public class MyDerbyBatch {
    public static void main(String... args) {
        String sql = "INSERT INTO article (id, title, body, updated_at) VALUES (default, ?, ?,
current_timestamp)";
        try (Connection conn = DriverManager.getConnection("jdbc:derby://localhost:1527/mydb",
"app", "app");
             PreparedStatement stmt = conn.prepareStatement(sql)) {
            for (int i = 0; i < 1000; i++) {
                stmt.setString(1, "title" + i);
                stmt.setString(2, "body" + i);
                stmt.addBatch();
            }
            stmt.executeBatch(); // まとめて実行
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

C O L U M N

ORMとJPA

ORMとはObject Relational Mappingの略です。RDBのレコードの読み書きを、可能な限り透過に、Javaオブジェクトの操作に対応させる実装パターンです。

ORMの実装方針にはいくつかの流派がありますが、もっとも一般的なORMは、RDBテーブル内のレコード（行）をオブジェクトに対応づける方法を取ります。レコードを構成するカラム（列）はオブジェクトのフィールドに対応づけます。レコードの各列の型や名前はテーブル定義で決定します。テーブル定義はJava側のクラス定義に対応します。

クラス側のidプロパティとテーブル側のidカラムで、オブジェクトとレコードを1対1に対応づけます。単純な仕組みですがほとんどのORMライブラリが採用するベストプラクティスです。

ORMライブラリは、オブジェクトの読み書きを自動的にRDBMS上のレコードの読み書きに変換します。裏側ではJDBCのSQL呼び出しをしますが、ORMライブラリがJDBCを隠蔽します。

著名なORM実装のHibernateなどをベースにしてORMのAPIが標準化されました。それがJPA（Java Persistence API）です。

本書執筆時点で有名なJPAの実装は、Hibernate、EclipseLink（GlassFishに付属）、OpenJPAです。

C-2-6 トランザクション処理

一連のまとまったRDB処理に意味がある場合があります。たとえば、あるレコードを読み込み、その値を使い別のレコードを更新する場合などです。一連の処理を中途半端に中断したり、途中の半端な状態が他の接続から見えたりすると、データの一貫性を失う危険があります。

中途半端な中断を許さず、かつ中途半端な状態を他から見えなくした一連の処理を「トランザクション処理」と呼びます。RDBMSはトランザクションをまとめて一貫性のある形で、かつ独立（分離）した形で実行できます。

RDBMSのトランザクション処理は複数のSQL処理からなる場合もあります。コード上、複数のSQL処理を実行したとしても、トランザクション処理にしていれば中途半端に中断されることも割り込まれることもありません。

トランザクション処理は、最終的に成功か失敗のどちらかで終わります。途中まで成功という状態にはなりません。

トランザクションは、開始と終了を明示する必要があります。終了の明示をコミット（commit）と呼びます。コミットが成功すると一連の処理すべてが完全な形で成功します。

明示的なトランザクションの中止も可能です。これをロールバック（rollback）と呼びます。ロールバックをすると、トランザクション中のSQL実行がすべてなかったことになります。

コミットに失敗した場合も自動的にロールバックになり、トランザクション中のSQL実行がすべてなかったことになります。

JDBCのデフォルト動作は、SQL1つ1つが1トランザクション処理になるモードです。この動作をオートコミットモードと呼びます。この場合、明示的にコミットしなくても、1つ1つのSQLが成功か失敗かのどちらかで終わります(注10)。

複数のSQL実行をトランザクション処理にまとめるには、オートコミットモードを無効にする必要があります。かつ最後にコミットしてトランザクション処理の終わりを明示する必要があります。

トランザクション処理コードの典型的な構造を示します（**リストC.11**）。トランザクションは暗黙に開始します。終了はcommitメソッドを呼んで明示します。

リストC.11　トランザクション処理の雛形

```
try (Connection conn = コネクション取得処理) {
    conn.setAutoCommit(false);
    try (StatementやPreparedStatementの生成処理) {
        Connectionを使うJDBC更新処理
        conn.commit();
    } catch (Exception e) {
```

（注10）当たり前に聞こえるかもしれませんが、SQL実行1つでもトランザクション処理単位になっているのは、RDBMSが持つべき重要な性質です。

```
        conn.rollback();
        throw e;
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

C-3 Webアプリとデータベース処理

C-3-1 コンテナとデータソースオブジェクト

Java EEアプリでデータベースを使う場合、コンテナにJDBCドライバのロードおよびデータベース接続を任せるのが定石です。

コンテナが管理するデータベース接続は、Webアプリからデータソースオブジェクト(DataSource)として見えます。DataSourceオブジェクトの取得方法は後述します。

WebアプリケーションのコードはDataSourceオブジェクトからConnectionオブジェクトを取得します。この時、DataSourceオブジェクトのgetConnectionメソッドを呼びます。DataSourceオブジェクトからConenctionオブジェクトを取得した後の処理は、今まで説明してきたJDBC処理と違いがありません。

■ コネクションプーリング

Webアプリでは一般に、複数のWebブラウザからの接続それぞれにスレッドを生成します。

各スレッドがConnectionオブジェクトを生成すると、WebアプリとRDBMSの間に多数の接続が発生します。接続が増えすぎると、Webアプリ側とRDBMS側の双方でメモリ消費が大きくなります。また、接続と破棄を繰り返すのは実行効率の点で好ましくありません。

しかし、1つのConnectionオブジェクトを複数スレッドで共有する解決方法は取れません。なぜなら、Connectionオブジェクトには同期処理がなく、複数スレッドでの同時利用を想定していないからです。

この問題に対処するために、コネクションプーリングという技法があります。コネクションプーリングを使うコンテナは、RDBMSとの複数接続を最初に確立し、それらの接続を維持し続けます。

WebアプリがDataSourceオブジェクトのgetConnectionメソッドを呼ぶと、新しい接続を確立する代わりに維持済みの接続(Connectionオブジェクト)を返します。このConnectionオブジェクトをクローズすると、接続を破棄する代わりにConnectionオブジェクトはプールに返されます。こうして接続(Connectionオブジェクト)を再利用し続けます。

C-3-2 手順

GlassFishにはJava DBが付属しています。GlassFishでJava DBを使うWebアプリ（Java EEアプリ）を開発する手順は下記になります。

① GlassFishからJava DBプロセスを起動
② GlassFish管理のコネクションプールを作成
③ GlassFish管理のJDBCリソースを作成（JNDI名を決める必要あり）
④ Webアプリのコード内からJNDI名でDataSourceオブジェクト取得
⑤ DataSourceオブジェクトからConnectionオブジェクト取得
⑥ Connectionオブジェクト取得以降は、通常のJDBCプログラミング

なお、Java DB以外のRDBMSを使う場合、そのRDBMSのJDBCドライバをGlassFishがロードできるクラスパスに配置する必要があります。JDBCドライバの配置以外の基本的な考え方は同じです。

■Java DBプロセスの起動

GlassFishからJava DBプロセスを起動、停止するには下記のように実行します。

```
$ ~/glassfish4/bin/asadmin start-database --dbport 9000
$ ~/glassfish4/bin/asadmin stop-database --dbport 9000
```

引数の--dbport 9000は待ち受けポート番号の指定です（コラム参照）。後述するコネクションプール作成時に、このポート番号の指定を忘れないようにしてください。ijコマンドでJava DBに接続する場合は下記のようになります。

```
ij> connect 'jdbc:derby://localhost:9000/mydb;create=true;user=app;password=app';
```

C-3-3 GlassFishの設定

明示的にJDBCドライバをロードする場合、**リストC.3**のように接続URLやユーザIDなどの接続情報をJavaコード上に記述しました。コンテナに接続を任せる場合、これらの情報はコンテナに与える設定値になります。

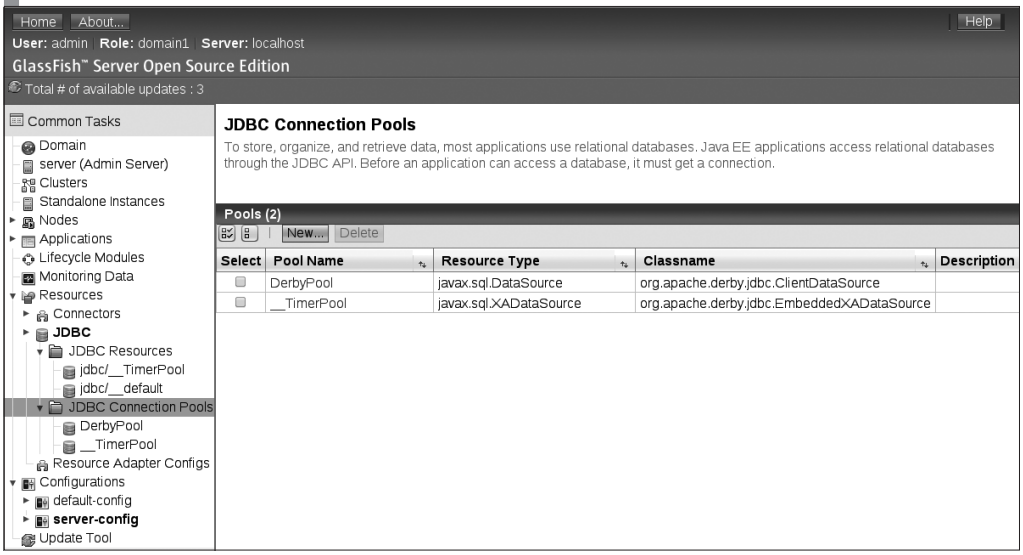
asadminコマンドでも設定可能ですが、説明のしやすさを優先して管理コンソールを使う設定方法を説明します。

手順は**図C.3**から**図C.7**になります。まず**図C.3**～**図C.5**の手順でコネクションプールを作成します。実質上、データベース接続のための設定と考えてください。

図C.6と**図C.7**がJDBCリソースの作成です。先ほど作成したコネクションプールをJNDIに登

録する作業と考えてください。これでJavaコードからJNDI名でデータソースを取得できるようになります。

図C.3　コネクションプールの作成開始



C O L U M N

Javaのセキュリティポリシーとポート番号

Java DBのデフォルトの待ち受けポート番号は1527です。新しいJavaでは、このポート番号のままJava DBを起動しようとすると下記のエラーで起動に失敗します。

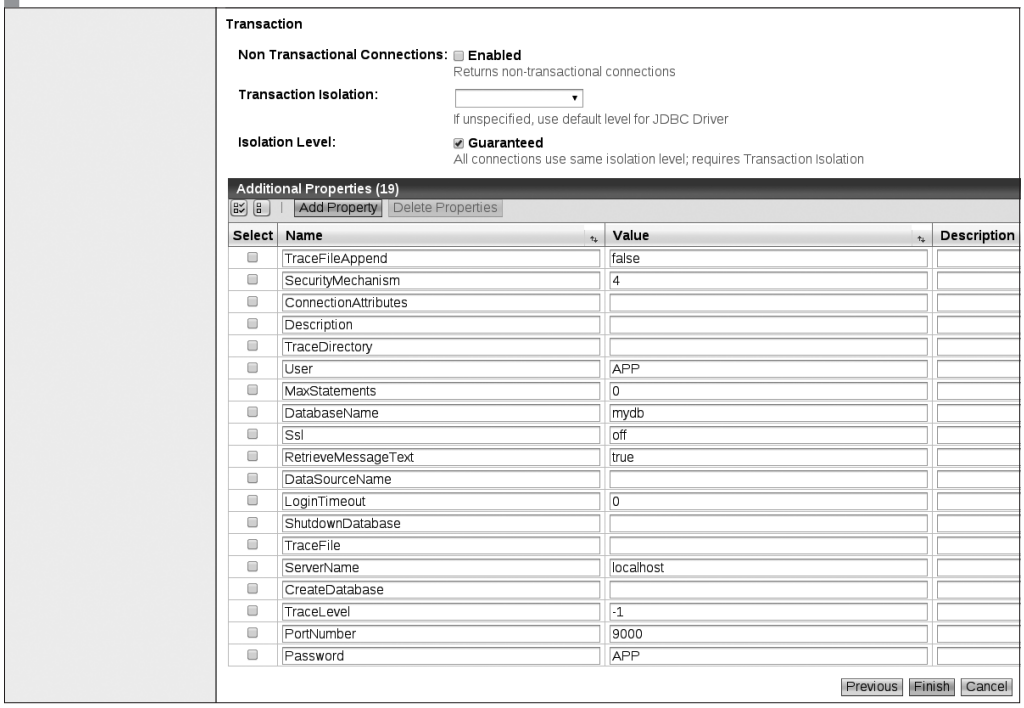
```
$ ./asadmin start-database
java.security.AccessControlException: access denied ("java.net.SocketPermission"
"localhost:1527" "listen,resolve")
```

このエラーは、待ち受けポート番号に使えるポート番号の範囲に特権を必要とするセキュリティポリシーを示しています。Javaセキュリティポリシーの設定での対処も可能ですが、本書はポート番号の変更で対処します。仮に9000番でもエラーになる場合、(たとえばLinuxなら) 下記ファイルを見てください。2つの数値の間のポート番号を指定すれば起動するはずです。調べ方が不明な場合、適度に大きい数値をポート番号に指定すれば起動するはずです。

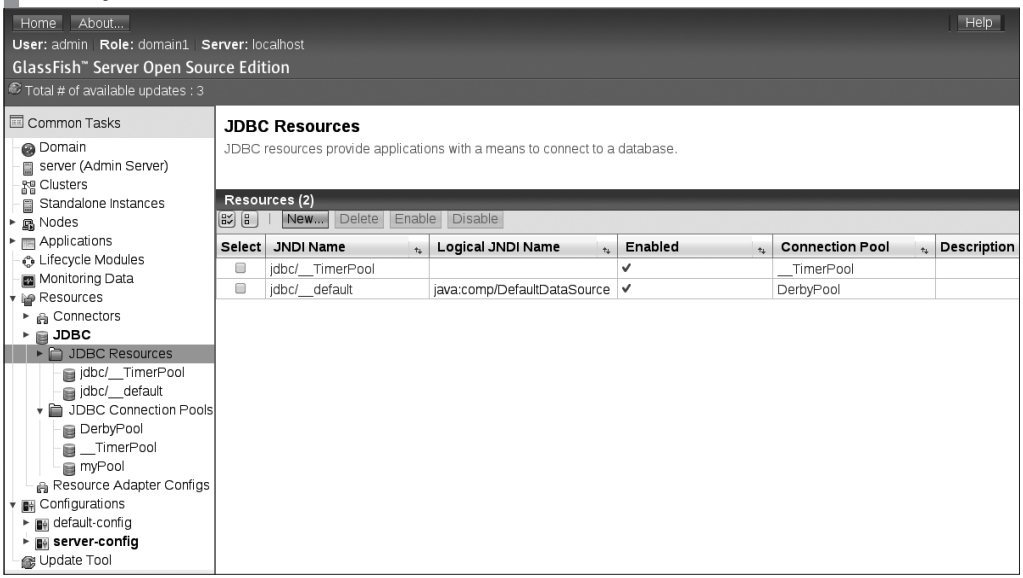
```
$ cat /proc/sys/net/ipv4/ip_local_port_range
9000    65500
```

図C.4　コネクションプールの作成（ステップ1）

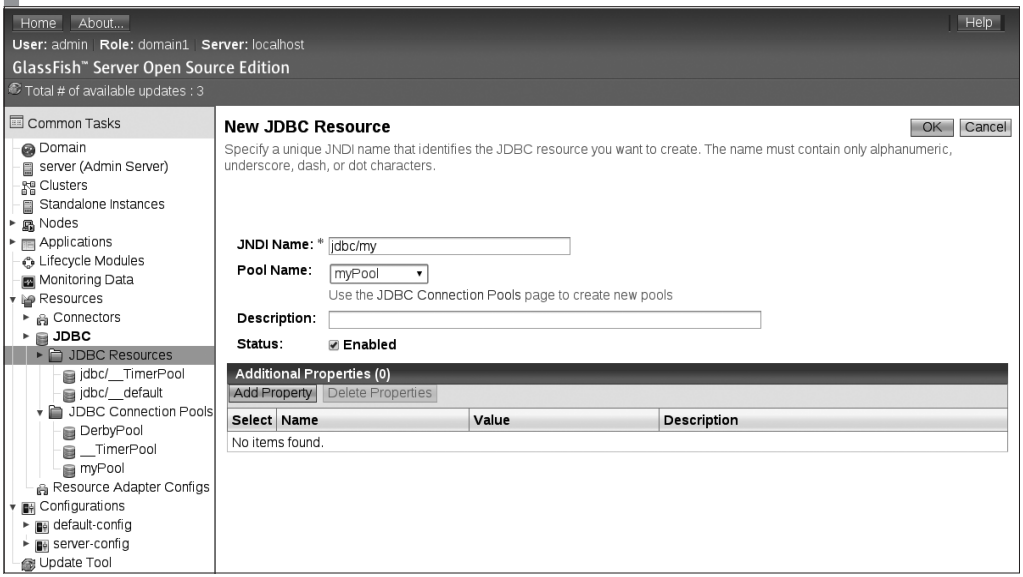
図C.5　コネクションプールの作成（ステップ2）



図C.6　JDBCリソースの作成（ステップ1）



図C.7　JDBCリソースの作成（ステップ2）



データソースを使うJAX-RSアプリを**リストC.12**に示します。サーブレットアプリでも同じように書けます。

JNDIで管理されたDataSourceオブジェクトは@Resourceアノテーションで取得できます。属性としてJNDI名を渡す必要があります。**リストC.12**の"jdbc/my"は、**図C.7**のJDBCリソース作成時につけた名前に対応します。

DataSourceオブジェクトのgetConnectionメソッドでConnectionオブジェクトを取得して、JDBCのAPIを使ってデータベースアクセス可能です。

リストC.12　データソースを使うJAX-RSアプリ

```
import javax.annotation.Resource;
import javax.ws.rs.WebApplicationException;

@ApplicationScoped
@Path("/my")
public class MyController {
    @Resource(lookup="jdbc/my")
    private DataSource ds;

    @GET
    @Path("list")
    @Produces(MediaType.TEXT_PLAIN)
    public String list() {
        StringBuilder text = new StringBuilder();
        try (Connection conn = ds.getConnection();
             Statement stmt = conn.createStatement()) {
            String sql = "SELECT title, body FROM article ORDER BY updated_at DESC";
            try (ResultSet rs = stmt.executeQuery(sql)) {
                while (rs.next()) {
                    text.append("title: " + rs.getString("title") + ", ");
                    text.append("body: " + rs.getString("body") + System.lineSeparator());
                }
            }
        } catch (Exception e) {
            throw new WebApplicationException(e);
        }
        return text.toString();
    }
}
```

リストC.12は、リクエスト処理のエントリポイント、データベース処理、レスポンス生成処理すべての役割が1つのクラスにまとまっています。規模が大きくなった場合、コードの複雑さを抑制するため役割を分割したほうが良いでしょう。具体例は次章で説明します。

C-3-4　Java EEのトランザクション管理

Java EEにはトランザクション管理のための規格が2つあります。JTAとEJBです。JTAはトランザクション管理専用の規格で、EJBはJTAよりも包括的な規格です。EJB（本書で扱うのはEJB Lite）の利用例は次章に譲り、ここではJTAを紹介します。本節で説明の大部分はEJBにも当てはまります。

JTA（EJBも）はトランザクション管理をコンテナに委譲します。コンテナに管理を委譲すると下記の利点を得られます。

- **JPAなどキャッシュを含めたトランザクション管理が可能になる**
- **複数のデータソースにまたがる分散トランザクション管理が可能になる**
- **後述する宣言的トランザクションを使うと、入れ子になったトランザクション処理の伝播を簡易に扱える**

JTAはプログラマブルAPIとアノテーションAPIの2つを手段を提供します。

プログラマブルAPIを利用するには、javax.transaction.UserTransactionインターフェースの機能を使います。トランザクション管理オブジェクトは内部的にJNDIで管理されるので、@Resourceアノテーションで UserTransactionオブジェクトを取得可能です。

トランザクション開始時にUserTransactionオブジェクトのbeginメソッドを呼び、コミット時にcommitメソッド、ロールバック時にrollbackメソッドを呼びます。やや作為的ですが利用例を**リストC.13**を示します。

リストC.13　JTAのトランザクション処理の例

```
import javax.transaction.UserTransaction;

@ApplicationScoped
@Path("/my")
public class MyController {
    @Resource(lookup="jdbc/my")
    private DataSource ds;

    @Resource
    private UserTransaction ut;

    @POST
    @Path("update")
    @Produces(MediaType.TEXT_PLAIN)
    public String update(@FormParam("id") List<Integer> idList) {
        String sql = "UPDATE article SET updated_at=current_timestamp WHERE id=?";
        try (Connection conn = ds.getConnection();
             PreparedStatement stmt = conn.prepareStatement(sql)) {
```

```
            ut.begin();
            for (Integer id : idList) {
                stmt.setInt(1, id);
                if (stmt.executeUpdate() != 1) { // 存在しないレコード更新の場合、トランザクション中止
(ロールバック)
                    throw new WebApplicationException("no row found");
                }
            }
            ut.commit();
        } catch (Exception e) {
            try {
                ut.rollback();
            } finally {
                return "NG";
            }
        }
        return "OK";
    }
}
```

リストC.13にcURLコマンドでアクセスする例を示します(Webアプリ名はmydbです)。

```
$ curl -X POST -d 'id=1' -d 'id=2' http://localhost:8080/mydb/my/update
```

リストC.13は、存在しないレコードのidをPOSTするとトランザクションをロールバックする動きにしています。確認してみてください。

JTAのアノテーションAPIはjavax.transaction.Transactionalアノテーションです。アノテーションをクラスもしくはメソッドに付与します。アノテーションAPIを使うトランザクション処理を宣言的トランザクション処理と呼びます。

@Transactionalアノテーションを付与すると、明示的なbegin、commit、rollback呼び出しは不要になります。メソッド呼び出しがコミット開始になり、メソッドを正常終了すると自動的にコミット扱いになります。メソッド内で実行時例外が発生してメソッドを抜けた場合にロールバックします。明示的指定をしない限り、検査例外でメソッドを抜けた場合はロールバックではなくコミットになります。これらの規則は次章で紹介するEJBのアノテーションと同じ動作です。

付録D Webアーキテクチャ

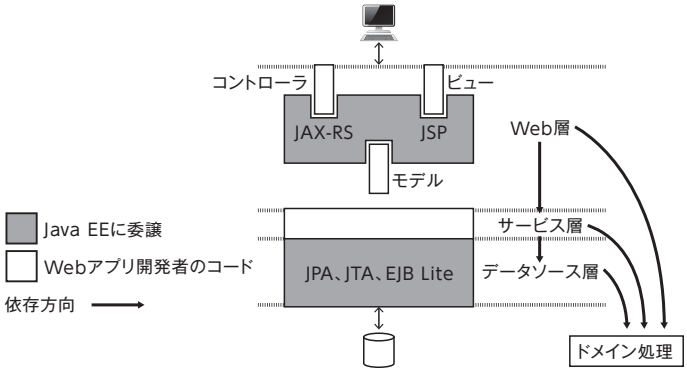
Webアプリ開発のアーキテクチャを代表的なベストプラクティスの視点で紹介します。最後に本パートのここまでで説明したJava EEを使い、データベースアクセスのある簡単なWebアプリの全体像を説明します。

D-1 アーキテクチャ

D-1-1 レイヤアーキテクチャ

Webアプリのコードを**図D.1**のようなレイヤ構成に整理する設計技法があります(注1)。レイヤアーキテクチャと呼びます。

図D.1 Webアプリのレイヤアーキテクチャ



レイヤアーキテクチャのポイントは、複雑さを層に隠蔽、シンプルな境界の提供、レイヤ間の依存方向を直下の層への1方向のみに限定、の3つです。たとえばデータソース層にはデータベースアクセスの複雑さを押し込め、Web層にはWeb処理の複雑さを押し込めます。

ある規模以上のWebアプリでは、3層レイヤと別にドメイン処理を区別するほうが現実的です(**図D.1**の右側)(注2)。ドメイン処理は、他から依存される安定コードとして存在します。

(注1)　レイヤアーキテクチャの用語の使い方は言語や文化によって様々です。宗教論争になるので本書は深入りしません。
(注2)　レイヤ構成にこだわりすぎる弊害は、レイヤ間で受け渡すデータオブジェクト(DTO)の変換処理の過剰実装です。

D-1-2 MVCアーキテクチャ

図D.1のWeb層には、MVCアーキテクチャとして知られる設計のベストプラクティスがあります。

MVCアーキテクチャとは、モデル（M）、ビュー（V）、コントローラ（C）の3つの機能分割を意識した技法です。元々はSmalltalkを由来とするGUIプログラミングで醸成されたプラクティスです。Webアプリケーションの世界へ適用、改造され、現在主流のアーキテクチャになっています。

「**1章　概論**」で述べたようにプログラミングの大原則の1つに分割統治があります。実装レベルの分割統治もあれば、大きな視点での役割の分割もあります。

MVCアーキテクチャは非常に大きな視点での役割の分割です。当然、たかが3つに分類するだけで全体の複雑さは劇的に軽減しません。その意味ではMVCアーキテクチャはおおざっぱな分割にすぎません。

分割統治という概念を頭で理解しても、プログラムの分割にどこから手をつけていいのか途方にくれるというのが一般的な開発者の姿です。最初はおおざっぱな分割から始めるのが簡単です。この時、MVCというベストプラクティスは有効です。どこから手をつけていいかわからずに立ち止まっていては何も始まりませんが手を動かすと物事が動き始めるからです。

D-1-3 MVCの最初の一歩

■ ビュー処理の分離と依存方向の管理

MVCアーキテクチャの最初のポイントはビューの分離です。

ビューとはソフトウェアの機能の中で見た目に関わる部分のことです。いわゆるユーザインターフェース機能です。Webアプリに限定するとビューは次の2つの機能で説明できます。1つは出力HTMLを生成する機能。もう1つは利用者からの入力（リクエスト）を処理する機能です。

Webアプリの中で見た目（ビュー）は要求がもっとも変わりやすい部分です。要求が変わるとコードに変更が必要です。プログラミングの原則の1つが、変わりやすい部分と変わりにくい部分の分離です。そして変わりやすい部分への依存を減らします。このため、ビュー処理を他から分離して、他の部分からビューへの依存をなくす（減らす）のは理にかなっています。

MVCアーキテクチャで最低限達成すべきは、ビューの分離と、他の部分からビューへの依存をなくすことです。

■ コントローラの分離とフレームワーク

Web層の役割は利用者の操作に応じた画面遷移です。利用者の操作（コード上はHTTPリクエストとして見えます）に応じて、必要な内部処理を呼び出し、内部処理の結果をビュー生成出力に渡します。このようなWeb層の処理の交通整理を行う役割を抽出します。この役割にコントロー

ラと名前をつけて分離します。

Web層をMVCの3つにおおざっぱに分割すると、ビューとコントローラ以外の役割をモデルと呼びます。

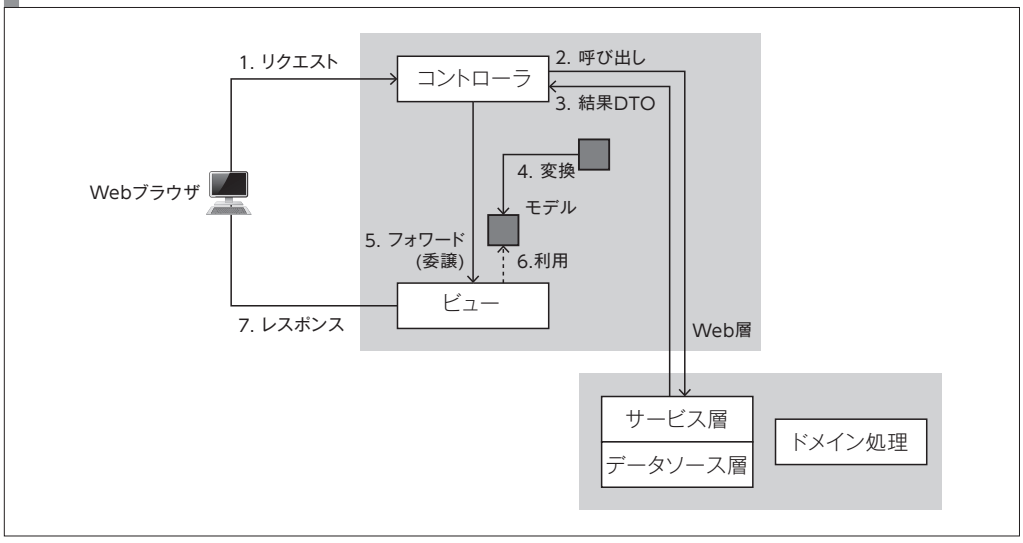
分割と依存方向の管理がMVCアーキテクチャの肝です(注3)。絶対に守るべき指針が、ビュー処理からモデルへの一方向の依存関係です。ビュー処理はモデルから情報を得て結果を生成しますが、モデルは決してビュー処理に依存してはいけません。

この依存関係を徹底した1つの方法論が、モデルを単に値を引くだけのデータにする設計です。Spring FrameworkのSpring MVCはこの設計方針を採用しています。コントローラからビュー処理に渡すオブジェクトを「モデル」と呼びます(**図D.2**)。

「モデル」という用語にはいくつかの流儀がありますが、本書はSpring MVCの用語の使い方に従います(注4)。

依存関係の話を続けると、モデルおよびビューはコントローラを知らない関係であるべきです。

図D.2　MVCアーキテクチャ



■ MVCフレームワーク

コントローラの役割分離と依存関係の整理はMVCフレームワークへの道につながります。なぜならコントローラの処理の多くはアプリ固有性が低いからです。

(注3)　レイヤの説明と類似性を感じるのは偶然ではなく、分割、依存管理、命名という設計の原則が根源にあるからです。

(注4)　Spring MVCのモデルの用語が気に入らない人は、プログラム全体の安定した構造を表現する「ドメインモデル」と、MVCのビュー処理に渡される「ビューモデル」の用語を使い分けてください。

本書はSpring MVC相当のアーキテクチャをJAX-RSで開発します。つまりJAX-RSをMVCフレームワークとして使います。

JAX-RSをMVCフレームワークとして使うとコントローラの役割の大半をJAX-RSに任せて、URLルーティングなどを（アノテーションで）宣言的に記述できます。

D-2 CRUDアプリの具体例

D-2-1 概要

本節ではarticleテーブルに対して、データ作成、データ表示、データ更新、データ削除を行うコードを示します。一般にこの4操作をCRUD（create、read、update、delete）と呼びます。CRUDはデータベースを扱うアプリケーションで基本となる操作です。

本節で作るWebアプリの最終的な開発ソースツリーを示しておきます（**図D.3**）。

図D.3　本節で開発するWebアプリのソースツリー

前節のレイヤアーキテクチャに当てはめると、ArticleControllerクラス（JAX-RSリソースクラス）はWeb層のコントローラです。

ArticleServiceクラスはサービス層です。本CRUDアプリにドメイン処理（入力バリデーション処理、アクセス制御、ワークフロー処理、複雑な検索処理、外部連携処理など）はほぼなく、データベースアクセスの結果を整形して返すだけなので、ArticleServiceはデータソース層を内包しています。ArticleクラスはWeb層のモデル兼データベースレコードのマッピング対象クラスです。**図D.1**のようにレイヤの外側にあるドメインモデルと見立てます。

D-2-2　準備

本パートの説明の手順を再利用します。改めて手順を書くと下記になります。

① mvnコマンドで開発ソースツリーの雛形を作成（「サーブレットとJAX-RS」章を参照）
② JAX-RSを使うための設定ファイル更新（「サーブレットとJAX-RS」章を参照）
③ データベースおよびテーブルを準備（「データベース」章を参照）
④ GlassFishのデータソースを準備（「データベース」章を参照）

Webアプリ名をmycrudにします。mvnコマンド例を下記に示します。

```
$ mvn archetype:generate -DgroupId=com.app -DartifactId=mycrud -DarchetypeArtifactId=maven-
archetype-webapp -Dversion=1.0-SNAPSHOT -DinteractiveMode=false
```

「**付録C　データベース**」の説明に従い、mydbデータベース、articleテーブルを作成してください。GlassFish上にデータソースを作成してJNDI名をjdbc/myにしてください。

D-2-3　文書一覧表示

aritcleテーブルのレコードすべてを表示する画面機能を開発してみます。

リストD.1はコントローラクラスです。この文書一覧を表示するにはhttp://localhost:8080/mycrud/article/のURLにアクセスします。

リストD.1　コントローラ役割のJAX-RSリソースクラス

```
package my; // パッケージ名は任意（後述のコードでは記述を省略）
// 紙幅の節約のため、一度説明したimport文の記述は省略します
import javax.inject.Inject;

@ApplicationScoped
@Path("/article")
public class ArticleController {
    @Inject
    private ArticleService articleService;
```

```
    @GET
    @Path("/")
    @Produces(MediaType.TEXT_HTML)
    public Viewable list() {
        try {
            List<Article> articles = articleService.fetchArticles();
            return new Viewable("/list.jsp", articles);
        } catch (RuntimeException e) {
            throw new WebApplicationException(e);
        }
    }
}
```

■ @Inject

今まで説明していないアノテーションが@Injectです。直感的にはDIのインジェクト位置を指示するアノテーションと考えてください。

@InjectはCDIのアノテーションです。CDIコンテナが、アノテーションを付与したフィールド変数にオブジェクト参照を代入します。代入対象のオブジェクトの生成およびライフサイクル管理もCDIコンテナがすべて面倒を見ます。

インジェクト対象のコードが**リストD.2**です。メソッドは「**付録C　データベース**」で説明したコードの応用なので説明を省略します。articleテーブルの1レコードをArticleオブジェクトに対応づけています。Articleクラスのコードは**リストD.3**を参照してください。後で使うために@FormParamアノテーションを使っていることを除けば説明不要のクラスです。

リストD.2　JAX-RSリソースクラスにインジェクトされるクラス

```
import javax.ejb.Stateless;

@Stateless
public class ArticleService {
    @Resource(lookup="jdbc/my")
    private DataSource ds;

    public List<Article> fetchArticles() {
        List<Article> articles = new ArrayList<>();
        try (Connection conn = ds.getConnection();
             Statement stmt = conn.createStatement()) {
            String sql = "SELECT id, title, body, updated_at FROM article ORDER BY updated_at
DESC";
            try (ResultSet rs = stmt.executeQuery(sql)) {
                while (rs.next()) {
                    articles.add(new Article(rs.getInt("id"), rs.getString("title"),
                                             rs.getString("body"), rs.getDate("updated_at")));
                }
```

```
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
        return articles;
    }
}
```

■ @Stateless

リストD.2の特徴は@Statelessアノテーションです。@Statelessアノテーションの付与には2つの目的があります。1つは@Injectでインジェクト対象可能にする目的です。もう1つはEJB Liteの機能により、宣言的トランザクションを可能にします。

前者の意味を説明します。@Injectでインジェクト可能なオブジェクトはCDIコンテナ管理のオブジェクトである必要があります。このために使えるアノテーションがいくつかありますが、一般に@StatelessのようなEJBのアノテーションか、もしくはCDIのアノテーションを使います。アノテーションは次節の管理ビーンで説明します。

宣言的トランザクションの意味は「**付録C　データベース**」で説明したJTAの@Transactionalと同じと考えてください。

■ 管理ビーン

表D.1、**表D.2**、**表D.3**のアノテーションをクラスに付与すると、そのクラスのオブジェクトがJava EEコンテナ（EJBコンテナとCDIコンテナを総称してこう呼ぶことにします）の管理オブジェクトになります。管理ビーン（Managed Bean）と呼ぶ場合もあります。管理ビーンは@Injectでインジェクト可能になります。

管理ビーンに付与するアノテーションの使い分けの基準は、コンテナによる宣言的トランザクションや同期処理が不要であればCDIアノテーション、必要であればEJBアノテーション、が一般的です(注5)。

■ スコープ

スコープの概要は「**付録A　Java EE概論**」の説明を読んでください。管理ビーンはスコープを持ちます。

リストD.1と**リストD.2**の@Inject関係から、管理ビーンの具体的なスコープを考えてみます。

ArticleControllerクラスはWebアプリ全体でシングルトンオブジェクトになります（@ApplicationScopedの効果）。このフィールド参照にインジェクトされるオブジェクトのスコー

（注5）　この使い分けの基準はJTAの@Transactional利用が一般的になると少し変わる可能性があります。オブジェクトをステートレスにして同期処理を不要と割り切れば、欲しいのは宣言的トランザクションだけになり、CDIとJTAでEJBを代替可能だからです。

プは理論上2パターンしかありえません。1つはアプリケーションスコープです。この場合、別のJAX-RSリソースクラスに@Injectしても、同一のArticleServiceオブジェクトがインジェクトされます。

もう1パターンは、@Inject先のJAX-RSリソースオブジェクトと1対1に対応するようにオブジェクト生成するパターンです。この場合、別のJAX-RSリソースクラスに@Injectすると、別のArticleServiceオブジェクトを生成してインジェクトします。管理ビーンがアノテーションに応じてどのパターンになるかは表D.1～表D.3の第3列を参照してください。

表D.1　インジェクト対象クラスに付与するEJBのアノテーション（javax.ejbパッケージ）

アノテーション	説明	@ApplicationScopedのJAX-RSリソースクラスに@Injectした時の挙動
@Singleton	シングルトンオブジェクト（状態の有無は問わない）。コンテナがトランザクション管理および同期処理	複数JAX-RSリソースオブジェクトから単一オブジェクトへの参照になる
@Stateless	状態（可変フィールド）を持たない前提のオブジェクト。コンテナがトランザクション管理	JAX-RSリソースオブジェクトごとにオブジェクト生成
@Stateful	状態（可変フィールド）を持つ前提のオブジェクト。コンテナがトランザクション管理および同期処理	JAX-RSリソースオブジェクトごとにオブジェクト生成

表D.2　インジェクト対象クラスに付与するCDIのアノテーション（javax.enterprise.contextパッケージ）

アノテーション	説明	@ApplicationScopedのJAX-RSリソースクラスに@Injectした時の挙動
@RequestScoped	HTTPリクエストごとにオブジェクト生成	GlassFish4では不正な動作（※1）
@SessionScoped	HTTPセッションごとにオブジェクト生成	デプロイ時にエラー
@ApplicationScoped	シングルトンオブジェクト生成	複数JAX-RSリソースオブジェクトから単一オブジェクトへの参照になる
@Dependent	@Injectされる側のオブジェクトと同じスコープでオブジェクト生成	JAX-RSリソースオブジェクトごとにオブジェクト生成
@ConversationScoped	開発者が開始と終了を明示できるスコープでオブジェクト生成	デプロイ時にエラー

※1　シングルトンのJAX-RSリソースオブジェクトのフィールドから参照されるオブジェクトはリクエストスコープになりえません。このためデプロイ時にエラーであるべきですが、GlassFish4ではエラーにならず動作してしまいます。ただし、インジェクト対象オブジェクトは@ApplicationScopedと同じスコープで動作します。

表D.3　インジェクト対象クラスに付与する疑似アノテーション

アノテーション	説明	@ApplicationScopedのJAX-RSリソースクラスに@Injectした時の挙動
javax.inject.Singleton	シングルトンオブジェクト。同期処理は開発者の責任	複数JAX-RSリソースオブジェクトから単一オブジェクトへの参照になる

リストD.3　フォームビーン兼データベースレコードのマップ先DTOクラス

```
public class Article {
    public Article() {}

    @FormParam("id") private int id;
    @FormParam("title") private String title;
```

```
    @FormParam("body") private String body;
    @FormParam("updated_at") private Date updated_at;

    public int getId() {
        return id;
    }
    public String getTitle() {
        return title;
    }
    public String getBody() {
        return body;
    }
    public Date getUpdated_at() {
        return updated_at;
    }

    public Article(int id, String title, String body, Date updated_at) {
        this.id = id;
        this.title = title;
        this.body = body;
        this.updated_at = updated_at;
    }
}
```

■ ビュー処理

リストD.1はビュー処理をJSPにフォワードします。フォワード先のJSPファイルを**リストD.4**に示します。モデルオブジェクト（articleのリスト）を${it}という記法で参照できる仕組みさえわかれば、理解できると思います(注6)。

C O L U M N

@Injectの類似アノテーション

@Injectの類似アノテーションがあるのでまとめておきます。@ResourceはJNDIで管理されるオブジェクトをインジェクトします。@PersistenceContextはJPAのEntityManagerオブジェクトをインジェクトします。@EJBはEJBの世界の@Injectと同じ役割のアノテーションです。@InjectはEJBのオブジェクトをインジェクト可能なので、@EJBは（事実上）時代遅れのアノテーションです。

（注6）　JSP内の${ }で囲まれた部分はEL（式言語）という記法です。ELのarticle.titleをJavaコードに読み替えると、JavaBeansプロパティアクセスのarticle.getTitle()呼び出しになると理解してください。

リストD.4　list.jsp（リストD.2のフォワード先）

```
// 紙幅の節約のため、後述のコードからpageディレクティブの記述を省略します
<%@ page contentType="text/html; charset=utf-8" %>
<%@ page session="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head><title>list</title></head>
<body>
  <table>
    <tr>
      <th>Title</th>
      <th>Body</th>
    </tr>

<form>
<c:forEach var="article" items="${it}">  <!-- ${it} の要素変数が article -->
  <tr>
    <td><c:out value="${article.title}"/></td>
    <td><c:out value="${article.body}"/></td>
    <td><a href="${article.id}">Show</a></td>
    <td><a href="${article.id}/_update">Edit</a></td>
    <td><button formaction="${article.id}/_delete" formmethod="POST">Delete</button></td>
  </tr>
</c:forEach>
</form>

  </table>
<hr>
<a href="create">Create</a>
</body>
</html>
```

D-2-4　文書表示

文書を表示する機能の実装を見ます。コントローラクラスは**リストD.5**です。**リストD.4**の Show がリンクです。データベーステーブルのidカラムの値を文書IDと見なして /mycrud/article/{id} をパスとするリクエストURLで文書表示とします。

コントローラクラスのshowメソッドはArticleServiceクラスのfetchArticleメソッドを呼び、結果をモデルとしてshow.jspビューに渡します。ArticleServiceクラスのfetchArticleメソッドとshow.jspは文書一覧表示関連のコードから類推可能なので省略します。

リストD.5　JAX-RSリソースクラスの文書表示コントローラメソッド

```
@ApplicationScoped
@Path("/article")
public class ArticleController {
    リストD.1に下記メソッドを追記

    @GET
    @Path("{id}")
    @Produces(MediaType.TEXT_HTML)
    public Viewable show(@PathParam("id") int id) {
        try {
            Article article = articleService.fetchArticle(id);
            return new Viewable("/show.jsp", article);
        } catch (RuntimeException e) {
            throw new WebApplicationException(e);
        }
    }
}
```

D-2-5　文書編集と文書作成

編集機能には2つの処理があります。編集画面を表示する処理と、編集結果をサブミットする処理です。

編集画面を表示するリンクは、**リストD.4**のEditです。**リストD.6**のeditメソッドが対応するコントローラメソッドです。

編集画面のJSPファイルはedit.jspです（**リストD.7**）。編集のサブミット先のURLは表示URLと同じにしてHTTPメソッドをPOSTにしています。

POSTメソッドのコントローラメソッドは**リストD.6**のupdateメソッドです。コメントにもありますが、RESTfulなURL設計を厳守するのであればPOSTメソッドではなくPUTメソッドのほうが適切です。今回はWebブラウザのフォームから（JavaScriptを使わずに）サブミットするためにPOSTメソッドを使っています。POSTでの代用の目印として、URLの末尾を _update にしています。

updateメソッドの引数は@BeanParam Article articleです。**リストD.3**の@FormParam効果により、HTMLのフォームの入力項目からフォームビーンが自動生成されて引数に渡ってきます。詳細は「**付録B　サーブレットとJAX-RS**」で説明しました。

updateメソッドから呼ぶArticleServiceクラスのupdateArticleメソッドは**リストD.7**を参照してください。

updateメソッドは保存処理の後、文書一覧画面にリダイレクトします。ここでリダイレクトする意味は「**付録B　サーブレットとJAX-RS**」の説明を参照してください。

リストD.6　JAX-RSリソースクラスの文書編集コントローラメソッド

```
@ApplicationScoped
@Path("/article")
public class ArticleController {
    リストD.1とリストD.5に下記メソッドを追記

    @GET
    @Path("{id}/_update")
    @Produces(MediaType.TEXT_HTML)
    public Viewable edit(@PathParam("id") int id) {
        try {
            Article article = articleService.fetchArticle(id);
            return new Viewable("/edit.jsp", article);
        } catch (RuntimeException e) {
            throw new WebApplicationException(e);
        }
    }

    @POST
    @Path("{id}/_update")    // PUTメソッドの代替URL
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public Response update(@PathParam("id") int id, @BeanParam Article article, @Context UriInfo
uriInfo) {
        try {
            articleService.updateArticle(id, article);
            URI uri = uriInfo.getBaseUriBuilder().path("/article/").build();
            return Response.seeOther(uri).build();
        } catch (RuntimeException e) {
            throw new WebApplicationException(e);
        }
    }
}
```

リストD.7　文書編集処理（リストD.6から呼ばれる）

```
@Stateless
public class ArticleService {
    リストD.2に下記メソッドを追記

    public void updateArticle(int id, Article article) {
        String sql = "UPDATE article SET title=?, body=?, updated_at=current_timestamp WHERE
id=?";
        try (Connection conn = ds.getConnection();
             PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setString(1, article.getTitle());
            stmt.setString(2, article.getBody());
            stmt.setInt(3, id);
```

```
            stmt.executeUpdate();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

リストD.8　edit.jsp（リストD.6のフォワード先）

```
<html>
<head><title>edit</title></head>
<body>
  <form action="" method="post">
    <p>Title: <input id="title" name="title" size="30" type="text" value="<c:out value="${it.
title}"/>" /></p>
    <p>Body: <textarea cols="40" id="body" name="body" rows="20"><c:out value="${it.body}"/></
textarea></p>
    <p><input id="submit" name="submit" type="submit" value="Update" /></p>
  </form>
</body>
</html>
```

文書作成機能は編集機能から類推可能な実装なので説明を省略します。

D-2-6　文書の削除

文書削除のコントローラメソッドは**リストD.9**のdeleteメソッドです。RESTfulなURL設計に厳密に従うのであればDELETEメソッドを使いますが、編集操作のPUTメソッドと同じ理由でPOSTメソッドで代用します。代用を明示するためURLパスに _delete を付与します（DELETEメソッドを使うのであればURLのパスに _delete がないほうがRESTfulです）。

deleteメソッドは文書削除後、文書一覧画面にリダイレクトします。文書削除処理の本体はArticleServiceクラスのdeleteArticleメソッドは類推可能なので省略します。

リストD.9　JAX-RSリソースクラスの文書削除コントローラメソッド

```
@ApplicationScoped
@Path("/article")
public class ArticleController {
    リストD.1、リストD.5、リストD.6に下記メソッドを追記

    @POST
    @Path("{id}/_delete")    // DELETEメソッドの代替
    public Response delete(@PathParam("id") int id, @Context UriInfo uriInfo) {
        try {
```

```
            articleService.deleteArticle(id);
            URI uri = uriInfo.getBaseUriBuilder().path("/article/").build();
            return Response.seeOther(uri).build();
        } catch (RuntimeException e) {
            throw new WebApplicationException(e);
        }
    }
}
```