サーブレット仕様書、第 3.0 最終版

Java™ Servlet Specification
Version 3.0
Rajiv Mordani
December 2009

日本語翻訳版（訳：Cresc Corp.）
2010 年 5 月初版
2011 年 1 月一部訂正

この資料は 2009 年 12 月付けのサーブレット仕様書の第 3.0 版（Final Version）のまえがき、変更記録等を除いた第 1 章から第 15 章の部分を翻訳したものである。翻訳の精度は保障されていないので、意味が不明な場合は原文のほうを参照いただきたい。またこの翻訳ドキュメントは利用者のインプットをもとにリファインする予定であり、ときどきチェックされることをお願いする。

オリジナルの仕様書と javadoc 等は以下のアドレスから取得できる：
http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html

目次

第1章　　概要（Overview）

1.1　サーブレットとは何か？（What is a Servlet）

サーブレットは Java™ 技術ベースのウェブ・コンポーネントであって、ダイナミックなコンテントを発生させるコンテナによって管理されている。他の Java 技術ベースのコンポーネントと同様に、サーブレットはプラットホームに依存しない Java のクラス群であって、プラットホームに依存しないバイト・コードにコンパイルされ、それが Java 技術対応のウェブ・サーバにダイナミックにロードされ実行される。時にはサーブレット・エンジンとも呼ばれるコンテナはウェブ・サーバの拡張物であって、サーブレットの機能を提供する。サーブレットはサーブレット・コンテナに実装されている要求 / 応答のパラダイムを介してウェブのクライアントたちと関わりあう。

1.2　サーブレット・コンテナとは何か？（What is a Servlet Container?）

サーブレット・コンテナはウェブ・サーバまたはアプリケーション・サーバの一部であって、要求と応答が送信され、MIME ベースの要求をデコードし、MIME ベースの応答を生成するネットワーク・サービスを提供する。サーブレット・コンテナはまたサーブレットをそのライフサイクルにわたって含みまた管理する。

サーブレット・コンテナはホストのウェブ・サーバに組み込まれる、あるいはサーバのネーティブな拡張 API を解してウェブ・サーバ上のアドオンのコンポーネントとしてインストールされよう。サーブレット・コンテナはまたウェブ・対応のアプリケーション・サーバに組み込まれる、あるいはおそらくはインストールされることもあろう。

総てのサーブレット・コンテナは要求と応答の為のプロトコルとして HTTP をサポートしなければならないが、HTTPS (HTTP over SSL) のような付加的な要求 / 応答ベースのプロトコルもサポートされることもある。コンテナが実装しなければならない HTTP 仕様のバージョンは HTTP/1.0 と HTTP/1.1 である。コンテナが RFC2616(HTTP/1.1)で記されたキャッシングのメカニズムを持つ場合は、そのコンテナは RFC2616 に準拠して、クライアントからの要求をサーブレットに渡す前に加工する、サーブレットが生成した応答をクライアントに送信する前に加工する、あるいはサーブレットにわたすことなく要求に対し応答することもある。

サーブレット・コンテナはあるサーブレットが実行する環境上でセキュリティの制約を設置できる。Java Platform, Standard Edition（J2SE, v.1.3 またはそれ以降）、あるいは Java Platform, Enterprise Edition（Java EE, v.1.3 あるいはそれ以降）の環境では、これらの制約は Java プラットホームが規定した許可のアーキテクチャを使って設置されねばならない。例えば、ハイエンドのアプリケーション・サーバではそのコンテナの他のコンポーネントがマイナスの影響を与えないよう Thread オブジェクトの生成を制限することがある。

サーブレット・コンテナが実装される Java プラットホームの最低バージョンは Java SE 6 である。

1.3 事例（An Example）

以下は一般的なイベントのシーケンスである：

1. あるクライアント（例えばウェブ・ブラウザ）があるウェブ・サーバをアクセスし、HTTP 要求をする。
2. その要求はウェブ・サーバによって受信され、サーブレット・コンテナに引き渡される。そのサーブレット・コンテナはそのホストのウェブ・サーバと同じプロセスで動作する、同じホストの別のプロセスで動作する、あるいはそのウェブ・サーバとは別の要求処理をするホスト上のプロセスとして動作する。
3. そのサーブレット・コンテナはそのサーブレットの設定に基づいてどのサーブレットを呼び出すかを判断し、その要求と応答を代表するオブジェクトと共にそれを呼び出す。
4. そのサーブレットは要求オブジェクトを使って誰がリモート・ユーザなのか、どの HTTP POST パラメタがこの要求の一部として送信されたか、及びその他の関連データを見出す。このサーブレットはプログラムされたロジックを実行し、クライアントに送り返すデータを発生させる。このサーブレットは応答オブジェクトを介してクライアントにデータを送り返す。
5. そのサーブレットがその要求の処理を終了したら、このサーブレット・コンテナは応答を適正に送出させ、コントロールをホストのウェブ・サーバに戻す。

1.4 サーブレットの他の技術との比較（Comparing Servlets with Other Technologies）

機能的には、サーブレットはコモン・ゲートウェイ・インターフェイス（CGI）プログラムと Netscape Server API (NSAPI)あるいは Apache Modules のような独自のサーバ拡張との間に位置している。

サーブレットたちは他のサーバ拡張メカニズムに比べて以下のような優位性を持つ：

- 異なった処理モデルが使われているので、これらは CGI スクリプトよりは一般的にずっと高速である。
- これらは多くのウェブ・サーバがサポートしている標準の API を使っている。
- これらは開発し易さとプラットホーム独立を含む Java プログラミング言語の完全な優位性を持っている。
- これらは Java プラットホームで利用できる大きな API のセットにアクセスできる。

1.5 Java プラットホーム法人版との関連（Relationship to Java Platform, Enterprise Edition）

Java プラットホーム法人版 v.5[1] では Java サーブレットの v.3.0 版 API が要求されている。これらに配備されるサーブレット・コンテナとサーブレットたちは Java EE 環境で実行するために Java EE 仕様書で記されている更なる要求を満足しなければならない。

1. http://java.sun.com/javaee/で取得できる Java™ 2 プラットホーム法人版仕様を参照されたい。

1.6 Java サーブレット仕様書の 2.5 版との互換性（Compatibility with Java Servlet Specification Version 2.5）

本節は本仕様のこの版によって生じる互換性問題を記している。

1.6.1 リスナの順番付け（Listener ordering）

本仕様の本リリース前は、リスナたちはランダムな順番で呼び出されていた。サーブレット 3.0 では、リスナが呼び出される順序は、第 8.2.7 節「web.xml、web-fragment.xml、及びアノテーションからの記述子のくみたて」で定義されている。

1.6.2 アノテーションの処理（Processing annotations）

サーブレット 2.5 版では、metadata-complete のみが配備時のアノテーションのスキャンに影響を与えていた。サーブレット 2.5 版ではウェブ・フラグメント(web-fragments)という概念は存在しなかった。しかしながらサーブレット 3.0 版では metadata-complete は配備時の総てのアノテーションとフラグメントのスキャンに影響を与える。この版での記述子はあるウェブ・アプリケーションにたいしどのアノテーションをスキャンするかに影響させては**いけない**。本仕様のある版の実装物は metadata-complete が指定されていない限り、その設定でサポートされている総てのアノテーションをスキャンしなければ**ならない**。

第2章　　Servlet インターフェイス（The Servlet Interface）

Servlet インターフェイスは Java サーブレット API の中心的な抽象化である。総てのサーブレットは直接、あるいはより一般的にはこのインターフェイスを実装したクラスを継承することで、このインターフェイスを実装する。この Servlet インターフェイスを実装した Java サーブレット API の 2 つのクラスが GenericServlet と HttpServlet である。殆どの用途では、開発者たちは自分たちのサーブレットを実装するのに HttpServlet を継承することになる。

2.1　要求処理メソッド群（Request Handling Methods）

ベーシックな Servlet インターフェイスはクライアントの要求を処理するための service メソッドを定義している。このメソッドはサーブレット・コンテナがサーブレットのあるインスタンスに引き渡す各要求ごとに呼び出される。あるウェブ・アプリケーションへの同時要求を処理するには一般的にはウェブの開発者はサーブレットをその service メソッド内である時間に複数のスレッドが処理できるよう設計する必要がある。

一般にはウェブ・コンテナは異なったスレッドでその service メソッドを同時に実行することで同じサーブレットへの同時の要求を処理している。

2.1.1　HTTP 固有の要求処理メソッド群（HTTP Specific Request Handling Methods）

HttpServlet 抽象サブクラスはベーシックな Servlet インターフェイスに更にメソッド群が追加されていて、これらは HTTP ベースの要求を処理するのを支援するために HttpServlet クラスの service メソッドによって自動的に呼び出される。これらのメソッドは以下のようである：

- HTTP GET 要求を処理するための doGet
- HTTP POST 要求を処理するための doPost
- HTTP PUT 要求を処理するための doPut
- HTTP HEAD 要求を処理するための doHead
- HTTP OPTIONS 要求を処理するための doOptions
- HTTP TRACE 要求を処理するための doTrace

一般的に HTTP ベースのサーブレットを開発する際は、サーブレットの開発者は doGet と doPost のメソッドのみ対処することになる。他のメソッドたちは HTTP プログラミングに精通したプログラマのみが対処することになろう。

2.1.2 更なるメソッドたち（Additional Methods）

doPut と doDelete メソッドによりサーブレットの開発者たちはこれらの機能を採用している HTTP/1.1 クライアントをサポートできる。HttpServlet の doHead メソッドは doGet を特化させたもので、doGet メソッドで生成されたヘッダのみをクライアントに返す。doOptions メソッドはこのサーブレットがどの HTTP のメソッドに対応しているかを応答する。doTrace メソッドは TRACE 要求で送信されてきたヘッダの総てのインスタンスたちを含んだ応答を生成する。

2.1.3 条件付き GET のサポート（Conditional GET Support）

HttpServlet インターフェイスは条件付き GET 動作をサポートするための getLastModified メソッドを定義している。条件付き GET 動作はある時間以来修正されているリソースに限って送信することが要求されている。しかるべき状況においては、このメソッドの実装はネットワーク資源の効率的活用に寄与する可能性がある。

2.2 インスタンスの数（Number of Instances）

14 章の「配備記述子」で記されているようにこのサーブレットを含んだ該ウェブ・アプリケーションの配備記述子の一部であるサーブレット宣言が、どのようにサーブレット・コンテナがそのサーブレットのインスタンスを用意するのかをコントロールしている。

分散環境でホストされていないサーブレットの場合は（デフォルト）、サーブレット・コンテナはサーブレット宣言あたりただ一つのインスタンスのみを使わねばならない。しかしながら SingleThreadModel インターフェイスを実装したサーブレットの場合は、そのサーブレット・コンテナは重い要求負荷を処理するために複数のインスタンスをインスタンス化して特定のインスタンスへの要求を直列化できる。

配備記述子で分散可能としてマークされたあるアプリケーションの要素としてあるサーブレットが配備されている場合には、コンテナは Java 仮想マシン (JVM™)[1] あたりサーブレット宣言あたりただ一つのインスタンスを持つことができる。しかしながらある分散可能アプリケーションのなかのそのサーブレットが SingleThreadModel インターフェイスを実装している場合には、そのコンテナはそのコンテナの各 JVM でそのサーブレットの複数のインスタンスをインスタンス化出来る。

1. 「Java 仮想マシン」及び"JVM"は Java(TM) プラットホームの為の仮想マシンを意味する。

2.2.1 単一スレッド・モデルに関する注意（Note About The Single Thread Model）

SingleThreadModel インターフェイスの使用により与えられたサーブレットのインスタンスの service メソッ内ではある時間においてただ一つのスレッドのみが実行することが保障される。コンテナはそのようなオブジェクトをプールする可能性を持っているので、この保証は各サーブレットのインスタンスに対してのみ適用されることを指摘すること重要なことである。HttpSession のような同時にひとつ以上のサーブレットでアクセスできるようなオブジェクトは、SingleThreadModel を実装したものを含む複数のサーブレットによって任意のある時間に取得可能である。開発者はこのインターフェイスを実装するのではなくて他の手段、たとえばインスタンス変数を使わない、あるいはこれらの資源をアクセスするコードのブロックを同期化するなど、を使ってこれらの問題を解決することが推奨される。SingleThreadModel インターフェイスは本仕様のこのバージョンからは廃止対象化されている。

2.3 サーブレットのライフ・サイクル（Servlet Life Cycle）

サーブレットは、如何にロードしインスタンス化され、クライアントからの要求を処理し、そしてサービスから外されるかを定義したきちんと定められたライフ・サイクルによって管理されている。このライフ・サイクルは API のなかで総てのサーブレットが直接的に、または GenericServlet または HttpServlet 抽象クラスを介して間接的に実装しなければならない javax.servlet.Servlet インターフェイスの init、service、及び destroy のメソッドによって表現されている。

2.3.1 ロードとインスタンス化（Loading and Instantiation）

サーブレット・コンテナはサーブレットのロードとインスタンス化の責任を持つ。ロードとインスタンス化はそのコンテナが開始したときに生じるか、あるいはそのコンテナがある要求をサービスするのに必要と判断するまで延期され得る。サーブレット・エンジンが開始したときに、必要とされるサーブレットのクラスはサーブレット・コンテナによってその場所を特定されていなければならない。サーブレット・コンテナは通常の Java のクラス・ロード機能を使ってサーブレット・クラスをロードする。このローディングはローカルなファイ・システム、リモートのファイル・システム、あるいは他のネットワーク・サービスからなされ得る。その Servlet クラスをロードした後は、そのコンテナはそれを使用するためにインスタンス化する。

2.3.2 初期化（Initialization）

そのサーブレットのオブジェクトがインスタンス化された後で、コンテナはクライアントからの要求処理ができる前にそのサーブレットの初期化をしなければならない。初期化はサーブレットが永続的な設定データを読むことができよう、コストがかかる資源（JDBC™ API ベースの接続のような）を初期化出来るよう、及び他のワンタイムの動作を実施できるように、用意されている。コンテナは

ServletConfig インターフェイスを実装した固有（サーブレット宣言あたり）のオブジェクトを持った Servlet インターフェイスの init メソッドを呼ぶことでそのサーブレット・インスタンスを初期化する。

この設定オブジェクトにより、そのサーブレットはウェブ・アプリケーションの設定情報からの名前-値の初期化パラメタにアクセスできる。この設定オブジェクトによりサーブレットはそのサーブレットの実行時環境を記述したオブジェクト（ServletContext インターフェイスを実装した）にアクセスできるようになる。ServletContext インターフェイスに関する更なる情報は第 4 章の「サーブレット・コンテキスト」を見られたい。

2.3.2.1 初期化時のエラー条件（Error Conditions on Initialization）

初期化時にそのサーブレット・インスタンスは UnavailableException または ServletException をスローできる。この場合はそのサーブレットはアクティブなサービスのために置いてはならず、またサーブレット・コンテナによって解放されねばならない。Destroy メソッドは初期化が不成功だったと考えられるので呼び出されない。

初期化失敗時は新しいインスタンスがそのコンテナによってインスタンス化と初期化されても良い。この規則の例外は UnavailableException が使えない最小時間を示しているときで、その場合はそのコンテナは新しいサーブレット・インスタンスを生成し初期化する前にこの期間が過ぎるまで待たねばならない。

2.3.2.2 ツールに関する考察（Tool Considerations）

あるツールがあるウェブ・アプリケーションをロードして内省するときのスタティックな初期化メソッドたちをトリガすることはこの init メソッドを呼ぶこととは区別される。開発者たちは Servlet インターフェイスの init メソッドが呼ばれるまではサーブレットがアクティブなコンテナのランタイム内にあると仮定すべきではない。たとえば、サーブレットは、スタティックな（クラス）初期化メソッドが実行された後でのみでないとデータベースあるいは Enterprise JavaBeans™ コンテナの接続を確立しようとしてはならない。

2.3.3 要求処理（Request Handling）

サーブレットが適正に初期化された後は、サーブレット・コンテナはクライアント要求を処理するためにそれを使用できる。要求は ServletRequest 型の要求オブジェクトで代表されている。該サーブレットは ServletResponse 型のオブジェクトが提供するメソッドたちを呼ぶことで要求に対する応答を埋める。これらのオブジェクトは Servlet インターフェイスの service メソッドにパラメタとして渡される。HTTP 要求の場合は、コンテナによって提供されるこれらのオブジェクトは HttpServletRequest 及び

HttpServletResponse の型である。サーブレット・コンテナによってサービスに置かれたサーブレットのインスタンスはその生存中にどの要求をも処理しないこともあり得ることに注意されたい。

2.3.3.1 マルチスレッド問題（Multithreading Issues）

サーブレット・コンテナはサーブレットの service メソッドを介して同時に要求を送信できる。これらの要求を処理するために、サーブレットの開発者はその service メソッド内で複数のスレッドを同時処理するための適正な備えをしなければならない。

推奨はされていないものの、開発者にとっての代替手段は SingleThreadModel インターフェイスを実装することで、このインターフェイスは service メソッド内に唯一のスレッドのみが存在することを保障することが求められている。サーブレット・コンテナはこの要求を満たすためにあるサーブレットの要求を直列化する、あるいはサーブレットのインスタンスのプールを維持する。そのサーブレットが分散可能とマークされているウェブ・アプリケーションの要素である場合は、コンテナはそのアプリケーションが分散される各 JVM ごとにサーブレットのインスタンスのプールを維持できる。

SingleThreadModel インターフェイスを実装しないサーブレットに対しては、もしその service メソッド（あるいは HttpServlet 抽象クラスの service メソッドに割り当てられる doGet または doPost のようなメソッド）が synchronized という同期化キーワードで定義されている場合は、そのサーブレット・コンテナはインスタンス・プールのアプローチを使うことはできない、しかしそれを介して要求を直列化しなければならない。性能に対する有害な効果があるので、これらのような状況では service メソッド（あるいはそれに割り当てられるメソッド）を同期化しないことが開発者たちに強く勧告されている。

2.3.3.2 要求処理中の例外（Exceptions During Request Handling）

サーブレットはある要求のサービス中に ServletException あるいは UnavailableException のいずれかをスローできる。ServletException はその要求の処理中に何らかのエラーが生じたことをシグナルし、そしてそのコンテナはその要求をクリーン・アップするためのしかるべき処置を取らねばならない。UnavailableException はそのサーブレットが一時的または恒久的のどちらかで要求を処理できないことをシグナルする。

UnavailableException で恒久的に使えないことが示されたときは、サーブレット・コンテナはそのサーブレットをサービスから外し、destroy メソッドを呼び、そしてそのサーブレットのインスタンスを解放しなけらばならない。その要因でコンテナが拒否した要求は、SC_NOT_FOUND (404)応答で戻さねばならない。

もし UnavailableException で一時的に使えないことが示された場合は、サーブレット・コンテナは一時的に使えない期間中そのサーブレットを介しての要求中継をしないことを選択できる。その期間そのコンテナによって拒否された要求は SC_SERVICE_UNAVAILABLE (503)応答ステータス・コードで返されるとともに、それには何時再利用可能になるかを示す Retry-After ヘッダを付す。

コンテナは恒久的と一時的の区別を無視し総ての UnavailableExceptions を恒久的とみなし、UnavailableException をスローしたサーブレットをサービスから外すことを選択できる。

2.3.3.3 非同期処理（Asynchronous processing）

時にはフィルタ及び/あるいはサーブレットが応答発生前にあるリソースあるいはイベントを待たないと、ある要求処理を終了できないことがある。例えば、あるサーブレットは応答を発生させる前に、利用可能な JDBC 接続、リモートのウェブ・サービスからの応答、JMS メッセージ、あるいはアプリケーションのあるイベント、などを待つ必要がある。そのサーブレット内での待機は非効率な使い方である。何故ならそれはスレッドと他の限られたリソースを消費するブロッキング動作であるからである。しばしばデータベースのような遅いリソースが多くのスレッドをアクセス待ちの状態でブロックさせ、ウェブ・コンテナ全体のスレッドの枯渇とサービス品質劣化をもたらす。

サーブレット第 3 版では要求の非同期処理を導入しており、そのスレッドをコンテナに返して他のタスクを実行できるようにしている。その要求で非同期処理が始まると、他のスレッドあるいは折り返し（コールバック）が、応答生成及び complete 呼び出しする、あるいはその要求をディスパッチ(dispatch：引き渡し)し AsyncContext.dispatch メソッドを使ってそのコンテナのコンテキスト内で走れるようにする。非同期処理の一般的なイベントのシーケンスは次のようである：

1. 該要求が受信され、認証等のために通常のフィルタを介してそのサーブレットに渡される。
2. そのサーブレットは要求パラメタ及び/あるいはコンテントを処理し、その要求の内容を判断する。
3. そのサーブレットは例えばリソースあるいはデータ要求を出し、リモートのウェブ・サービスに要求を送信するあるいは JDBC 接続待ちの行列に加わる。
4. そのサーブレットは応答を発生させること無く戻る。
5. ある時間後に、要求されたリソースが使えるようになり、そのイベントを取り扱っているスレッドが、同じスレッド内で処理を続ける、あるいは AsyncContext を使ってそのコンテナ内のあるリソースにその処理をディスパッチ（渡す）して処理を続ける。

15.2.2 節の「ウェブ・アプリケーション環境」及び 15.3.1 節の「EJB™コール内のセキュリティ ID の伝播」のような Java 法人バージョンの機能は、初期要求を実行している、あるいはその要求が AsyncContext.dispatch メソッドによってそのコンテナにディスパッチ（引き渡し）されているときにのみ、スレッドたちが利用できる。Java 法人バージョンの機能たちは、AsyncContext.start(Runnable)メソッドを介して直接応答オブジェクト上で動作している他のスレッドたちが利用できる。

第 8 章で記されている@WebServlet と@WebFilter アノテーションは asyncSupported という属性を持っており、これはデフォルト値が false である boolean である。asyncSupported が true にセットされていると、そのアプリケーションは startAsync を呼ぶ（下記参照）ことで別のスレッドでの非同期処理を開始し、その要求と応答のオブジェクトへの参照をそれに渡し、次にオリジナルのスレッドでそのコンテナから出ることができる。このことはその応答は入ってきたと同じフィルタ（あるいはフィルタのチェイン）をトラバース（逆順で）することを意味する。その応答は AsyncContext 上で complete が呼ばれるまで（下記参照）コミットされない。そのアプリケーションは startAsync を呼んだコンテナが開

始したディスパッチ（引き渡し：dispatch）がそのコンテナに戻る前に、非同期タスクが実行中のときは、要求と応答のオブジェクトへの同時アクセスを処理する責を持つ。

asyncSupported=true であるサーブレットから asyncSupported が false にセットされているサーブレットへのディスパッチは許される。この場合、非同期をサポートしないサーブレットの service メソッドを抜けたらその応答はコミットされ、関心を持っている AsyncListener のインスタンスが通知を受けるように AsyncContext 上で complete を呼ぶのはそのコンテナの責任である。
AsyncListener.onComplete 通知はまた、それを持っていたリソースをクリアしてその非同期タスクを終了させるメカニズムとしてフィルタによって使われるべきである。

同期サーブレットから非同期サーブレットへのディスパッチは許されないことになろう。しかしながらそのアプリケーションが startAsync を呼ぶときまで IllegalStateException のスローの判断は延期される。これによりあるサーブレットが同期あるいは非同期のサーブレットとして機能できる。

そのアプリケーションが待っているその非同期タスクは、初期の要求に使われたスレッドとは別のスレッドで、その応答に直接書き込むことが出来る。このスレッドはフィルタに関しては何も知らない。もしあるフィルタが新しいスレッド内の応答オブジェクトを操作したいときは、そのフィルタはその初期要求が「入ってくる最中」に処理中のときにその応答をラップし、そのラップされた応答をそのチェインの次のフィルタに渡し、そして最終的にはそのサーブレットに渡す。したがってもしその応答がラップされていて（おそらくは複数回、フィルタあたり一度）、そのアプリケーションがその要求を処理し、応答に直接書き込むときは、それは実際は応答のラッパ（複数もあり）に書いていることになり、言い換えれば、その応答に追加された出力はそれでもその応答ラッパによって処理される。あるアプリケーションが別のスレッドである要求から読み出し、その応答に出力を追加するときは、それは実際は要求のラッパ（複数あり）から呼んでいて、応答のラッパ（複数あり）に書いていることになり、したがってそのラッパが意図した入力と出力の操作は引き続き生じ得る。

代替的にそのアプリケーションがそうすることを選択するときは、そのアプリケーションは AsyncContext を使って新しいスレッドからの要求をそのコンテナ内のあるリソースにディスパッチすることができる。これによりそのコンテナの適用範囲内にある JSP のようなコンテント生成技術を使うことができるようになる。

アノテーション属性たちに加えて、以下のようなメソッドとクラスが追加されている：

- ServletRequest
  - public AsyncContext startAsync(ServletRequest req, ServletResponse res). このメソッドはその要求を非同期モードに置き、与えられた要求と応答のオブジェクトと getAsyncTimeout で返されたタイムアウトでその AsyncContext を開始する。ServletRequest 及び ServletResponse パラメタは呼び出しているサーブレットの service、あるいはそのフィルタの doFilter メソッド、あるいはこれらをラップした ServletRequestWrapper または ServletResponseWrapper クラスのサブクラス、に渡されたと同じオブジェクトでなければ**ならない**。このメソッド呼び出しにより、このアプリケーションが service メソッドを出てもその応答がコミットされないことが確保される。戻された AsyncContext 上で AsyncContext.complete が呼ばれたとき、またはその AsyncContext がタイム・アウトしそのタイム・アウトを処理するために結び付けられたリスナが存在しないときに、それはコミットされる。非同期タイム・アウトのためのタイマはその要求とそれに対応した応答がそのコンテナから戻るまでは開始しない。非同期の

スレッドから応答への書き込みは AsyncContext が使用できる。これはまたその応答が閉じてコミットされていないことの通知としてだけにも使える。その要求が非同期動作をサポートしていないサーブレットまたはフィルタの適用範囲内にあるとき、あるいはその応答がコミットし閉じてしまっているとき、あるいは同じ引き渡し（ディスパッチ）中に再度呼ばれたときに、startAsync を呼ぶことは許されない。startAsync 呼び出しで返された AsyncContext は次に更なる非同期処理のために使用できる。返された AsyncContext 上の AsyncContext.hasOriginalRequestResponse()を呼ぶと、渡された ServletRequest および ServletResponse 引数がオリジナルなものである、あるいはアプリケーションが提供したラッパを持っていないとき、以外は false を返す。この要求が非同期モードに置かれた後での下り方向に呼び出されたフィルタたちは、その要求の一部及び/あるいは上り呼び出し中に追加した応答のラッパたちが非同期動作中に存在したままである必要が**あるかも知れず**、それに関連したリソースは解放**しなくても良い**ことに表示として**使うことができる**。あるフィルタの上り呼び出し中に適用される ServletRequestWrapper は、AsyncContext 初期化に使われていて AsyncContext.getRequest()を呼ぶことで返されるその ServletRequest が、該 ServletRequestWrapper を含んでいないときに限り、そのフィルタの下り呼び出しで解放されても良い。同じことが ServletResponseWrapper に対しても適用される。

- public AsyncContext startAsync()は非同期処理のためにオリジナルの要求と応答のオブジェクトを使うという利便性のために用意されている。ラップされた応答に書き込まれたデータが確実に失われないようにしたいときは、このメソッドを呼ぶ前に、これらがラップされている場合は、このメソッドのユーザはこれらの応答をフラッシュしなければ**ならない**ことに注意されたい。

- public AsyncContext getAsyncContext()は startAsync 呼び出しで生成されたあるいは再初期化された AsyncContext を返す。その要求が非同期モードに置かれていないときでの getAsyncContext は違反である。

- public boolean isAsyncSupported()は、その要求が非同期処理に対応していれば true を返しそうでなければ false を返す。その要求が非同期処理に対応（指定されたアノテーションを介してあるいは宣言的に）していないフィルタまたはサーブレットに渡されると直ちに非同期対応しなくなる。

- public boolean isAsyncStarted()はこの要求で非同期処理が開始したときは true を、そうでなければ false を返す。この要求が非同期モードに置かれて以来 AsyncContext.dispatch メソッドたちのひとつを使ってディスパッチされているとき、あるいは AsynContext.complete 呼び出しが呼ばれているときは、このメソッドは false を返す。

- public DispatcherType getDispatcherType()はある要求のディスパッチャ・タイプを返す。ある要求のディスパッチャ・タイプは、コンテナがその要求に適用するのに必要なフィルタを選ぶのに使う。ディスパッチャ・タイプと URL パタンが合ったフィルタのみが適用されることになる。複数のディスパチャ・タイプのために設定されているあるフィルタがそのディスパッチャ・タイプのためにある要求をクエリできるようにすることで、そのフィルタがそのディスパッチャ・タイプに基づいてその要求を異なったかたちで処理で

きる。ある要求の初期のディスパッチャ・タイプは DispatcherType.REQUEST と定められている。RequestDispatcher.forward(ServletRequest, ServletResponse)または RequestDispatcher.include(ServletRequest, ServletResponse)を介してディスパッチされた要求のディスパッチャ・タイプはそれぞれ DispatcherType.FORWARD あるいは DispatcherType.INCLUDE として与えられており、一方 AsyncContext.dispatch メソッドたちのひとつでディスパッチされた非同期要求のディスパッチ・タイプは DispatcherType.ASYNC で与えられる。最後に、コンテナのエラー処理メカニズムによりエラー・ページにディスパッチされた要求のディスパッチャ・タイプは DispatcherType.ERROR で与えられる。

- AsyncContext - このクラスは ServletRequest 上で開始した非同期動作のための実行コンテキストを代表している。AsyncContext は上記のように ServletRequest.startAsync を呼ぶことで生成され初期化される。以下のメソッドが AsyncContext に存在する:

  - public ServletRequest getRequest()は startAsync メソッドたちのひとつを呼ぶことで AsyncContext を初期化するのに使われた要求を返す。

  - public ServletResponse getResponse() は startAsync メソッドたちのひとつを呼ぶことで AsyncContext を初期化するのに使われた応答を返す。

  - public void setTimeout(long timeoutMilliseconds) このメソッドは非同期処理のタイム・アウト発生をミリ秒で設定する。このメソッドの呼び出しはコンテナによって設定されたタイム・アウトを凌駕する。setTimeout 呼び出しでタイム・アウトが指定されていないときは、コンテナのデフォルト値が使われる。0 またはそれ以下の値はその非同期動作が決してタイム・アウトを生じないことを示す。このタイム・アウトは ServletRequest.startAsync メソッドたちのひとつが呼ばれてコンテナが開始したディスパッチがコンテナに戻るまでの期間の AsyncContext に適用される。非同期サイクルが開始したコンテナが開始したディスパッチがコンテナに戻ったあとでこのメソッドが呼ばれているとき、タイム・アウトを設定することは違反となり、IllegalStateException をもたらす。

  - public long getTimeout() - その AsyncContext にかかわるタイム・アウトをミリ秒で取得する。このメソッドはそのコンテナのデフォルトのタイム・アウトまたはもっとも最近の setTimeout メソッド呼び出しで設定されたタイム・アウト値を返す。

  - public void addListener(asyncListener, req, res) – タイム・アウト、エラー、及び完了の通知のためのリスナを ServletRequest.startAsync メソッドのひとつを呼ぶことで開始したもっとも最近の非同期サイクルに登録する。非同期リスナたちはその要求に追加された順に通知を受ける。このメソッドに渡された要求と応答オブジェクトは AsyncListener が通知を受けたときは AsyncEvent.getSuppliedRequest()と AsyncEvent.getSuppliedResponse()で得られるものと全く同じものである。与えられた AsyncListener が登録されて以来ラッピングが発生している可能性があるので、これらのオブジェクトは読み出しまたは書き込みをしてはならないが、それにかかわる何らかのリソースを解放するために使われる可能性がある。コンテナがディスパッチを開始し

てそれにより非同期サイクルが開始したあとで、新しい非同期サイクルが開始する前に、でこのメソッドを呼ぶのは違反であり、IllegalStateException が結果として生じる。

- public <T extends AsyncListener> createListener(Class<T> clazz) – 与えられた AsyncListener クラスをインスタンス化する。返された AsyncListener インスタンスは以下に規定された addListener のメソッドたちのひとつを呼ぶことで AsyncContext に登録される前に更にカスタマイズされても良い。与えられた AsyncListener クラスはそのインスタンス化に使われる引数なしのコンストラクタを**定義しなければならない**。このメソッドはこの AsyncListener に適用できるアノテーションに対応している。

- public void addListener(asyncListener) - タイム・アウト、エラー、及び完了を通知するための与えられたリスナを ServletRequest.startAsync のメソッドたちのひとつを呼ぶことでもっとも最近の非同期サイクルに通知する。もし startAsync(req, res)または startAsync()がその要求で呼ばれているときは、その AsyncListener が通知を受けたときは全く同じ要求と応答のオブジェクトが AsyncEvent から取得できる。その要求と応答はラップされていてもされていなくても良い。非同期リスナたちはそれがその要求に追加されたと同じ順で通知を受ける。コンテナが開始したディスパッチで非同期サイクルが開始してコンテナに戻った後で、及び新しい非同期サイクルが開始する前にこのメソッドを呼ぶのは違反であり、結果として IllegalStateException が発生する。

- public void dispatch(path) – AsyncContext を初期化するのに使われた要求と応答を与えられたパスを持ったリソースにディスパッチする。この与えられたパスは AsyncContext を開始した ServletContext に相対したものとして解釈される。その要求の総てのパス関連クエリのメソッドたちはディスパッチ・ターゲットを**反映したものでなければならず**、一方オリジナルの要求 URL、コンテキスト・パス、パス情報、及びクエリ文字列は、第 9.7.2 章「ディスパッチされた要求パラメタたち」で定義されているように要求の属性から取得できよう。これらの属性はたとえ複数のディスパッチの後であっても常にオリジナルのパス情報を反映しなければならない。

- public void dispatch() - 以下のように AsyncContext を初期化するのに使われた要求と応答をディスパッチするという利便性のためにこのメソッドが用意されている。もし startAsync(ServletRequest, ServletResponse)を介して AsyncContext が初期化され、渡された要求が HttpServletRequest のインスタントである場合は、そのディスパッチは HttpServletRequest.getRequestURI()で戻された URI へのものである。そうでないときはそのディスパッチはそのコンテナによって最後にディスパッチされた要求の URI 向けである。以下の事例コード例 2-1、コード例 2-2、及びコード例 2-3 が異なったケースで何がターゲットの URI なのかを示している。

コード例 2-1

```
REQUEST to /url/A
AsyncContext ac = request.startAsync();
...
ac.dispatch();/url/A への非同期ディスパッチを実施
```

コード例 2-2

```
REQUEST to /url/A
FORWARD to /url/B
getRequestDispatcher("/url/B").forward(request, response);
AsyncContext ac = request.startAsync();
ac.dispatch();/url/A への非同期ディスパッチを実施
```

コード例 2-3

```
REQUEST to /url/A
FORWARD to /url/B
getRequestDispatcher("/url/B").forward(request, response);
AsyncContext ac = request.startAsync(request, response);
ac.dispatch();/url/B への非同期ディスパッチを実施
```

- public void dispatch(ServletContext context, String path) – AsyncContext を初期化するのに使われた要求と応答を与えられた ServletContext 内の与えられたパスをもったリソースにディスパッチする。

- 上記で定義された 3 つのディスパッチ変種たちの総てに対し、これらのメソッド呼び出しはコンテナが管理するスレッドにたいし要求と応答のオブジェクトを渡したら即座に戻り、そのコンテナが管理するスレッドがディスパッチ動作を実行する。この要求へのディスパッチャ・タイプは ASYNC にセットされている。
  RequestDispatcher.forward(ServletRequest, ServletResponse)ディスパッチとは違って、応答バッファとヘッダたちはリセットされないし、たとえ応答が既にコミットされている状態でもディスパッチすることは違反ではない。その要求と応答のコントロールはディスパッチ・ターゲットに委譲され、その応答は ServletRequest.startAsync()または ServletRequest.startAsync(ServletRequest, ServletResponse)が呼ばれていない限り、そのディスパッチ・ターゲットが実行を終了したときに閉じられる。startAsync を呼んだコンテナが開始したディスパッチがそのコンテナに戻る前にこれらのディスパッチのメソッドたちのどれかが呼ばれたときは、その呼び出しはコンテナが開始したディスパッチがそのコンテナに戻るまでは効果を持たない。AsyncListener.onComplete(AsyncEvent), AsyncListener.onTimeout(AsyncEvent)及び AsyncListener.onError(AsyncEvent)の呼び出しもまたコンテナが開始したディスパッチがそのコンテナに戻るまでは待たされることになる。ServletRequest.startAsync メソッドたちのひとつを呼ぶことで開始した非同期サイクルあたり最大ひとつの非同期ディスパッチ動作が存在し得る。
  ServletRequest.startAsync 呼び出しで開始した非同期サイクルあたり最大ひとつの非同期ディスパッチが存在し得る。同じ非同期サイクル内で更なる非同期ディスパッチ動作をおこなわせようとすることは違反で、IllegalStateException をもたらす。そのディスパッチ要求にたいし startAsync は引き続いて呼ばれたときは、これらのディスパッチのメソッドたちのどれもを上記制限のもとで呼び出すことができる。

- そのディスパッチのメソッドたちの実行中に生じたエラーまたは例外は以下のようにそのコンテナによって捕捉されまた処理**されねばならない**:
  i. それに対し AsyncContext がつくられた ServletRequest に登録された AsyncListener の総てのインスタンスに AsyncListener.onError(AsyncEvent)を

呼びだし、AsyncEvent.getThrowable()によって Throwable を捕捉できるようにする。

ii. どのリスナも AsyncContext.complete あるいは AsyncContext.dispatch メソッドたちのどれかを呼んでいないときは、HttpServletResponse.SC_INTERNAL_SERVER_ERROR と等しい状態コードでエラーのディスパッチを行い、RequestDispatcher.ERROR_EXCEPTION 要求属性の値として Throwable が利用できるようにする。

iii. もしマッチしたエラー・ページが見つからない、あるいはそのエラー・ページが AsyncContext.complete()あるいは AsyncContext.dispatch メソッドたちのどれかを呼んでいないときは、そのコンテナは AsyncContext.complete を呼**ばねばならない**。

- public boolean hasOriginalRequestAndResponse() - このメソッドは ServletRequest.startAsync()を呼ぶことでその AsyncContext がオリジナルの要求と応答のオブジェクトで初期化されているか、あるいは ServletRequest.startAsync(ServletRequest, ServletResponse)を呼ぶことでそれが初期化されたか、及び ServletRequest 引数も ServletResponse 引数もアプリケーションが提供したラッパを持っていないかをチェックし、その場合は true を返す。もし AsyncContext が ServletRequest.startAsync(ServletRequest, ServletResponse)を使ってラップされた要求及び/または応答で初期化されているときは、このメソッドは false を返す。この情報はある要求が非同期モードに置かれた後でくだり方向に呼ばれたフィルタたちによって使われているフィルタたちが、それらの上り方向呼び出し中に追加した要求及び/あるいは応答が、非同期動作中に保存する必要があるかあるいは開放できるかを判断するために使える。

- public void start(Runnable r) – このメソッドは指定された Runnable を走らせるために、おそらくは管理されたスレッド・プールから、そのコンテナがあるスレッドをディスパッチさせる。そのコンテナはしかるべきコンテキスト的情報を Runnable に伝搬させることができる。

- public void complete() - もし request.startAsync が呼ばれたとき、次にこの非同期処理を完了させ応答のコミットとクローズを行うためにはこのメソッドを呼ばねばならない。この完了メソッドは非同期処理をサポートしていないサーブレットに要求がディスパッチされている、あるいは AsyncContext.dispatch で呼ばれたターゲットのサーブレットがそれに伴う startAsync 呼び出しをしていないときは、そのコンテナによって呼び出されねばならない。この場合、そのサーブレットの service メソッドを抜けた直後に complete()を呼ぶのはそのコンテナの責任である。もし startAsync が呼ばれなかったときに IllegalStateException がスローされ**ねばならない**。ServletRequest.startAsync()あるいは ServletRequest.startAsync(ServletRequest, ServletResponse)を呼び出した後で、またディスパッチのメソッドたちのひとつを呼び出す前に、何時でもこのメソッドを呼び出すのは構わない。もしこのメソッドが startAsync を呼び出したコンテナが開始したディスパッチがそのコンテナに戻っている前に呼び出されたときは、その呼び出しはそのコンテナが開始したディスパッチがそのコンテナに戻るまで効果を持たない。AsyncListener.onComplete(AsyncEvent)呼び出しもまたコンテナが開始したディスパッチがそのコンテナに戻るまで遅らせられる。

- ServletRequestWrapper（サーブレット要求ラッパ）

  - public boolean isWrapperFor(ServletRequest req)- このラッパが与えられた ServletRequest をラップしているかを再帰的にチェックし、そうであれば true を返し、そうでなければ false を返す。

- ServletResponseWrapper（サーブレット応答ラッパ）

  - public boolean isWrapperFor(ServletResponse res)- 再帰的にこのラッパが与えられた ServletResponse をラップしているかをチェックし、そうであれば true を返し、そうでなければ false を返す。

- AsyncListener（非同期リスナ）

  - public void onComplete(AsyncEvent event) – このメソッドはその ServletRequest で開始した非同期動作の完了をそのリスナに通知するのに使われる。

  - public void onTimeout(AsyncEvent event) – このメソッドはその ServletRequest で開始した非同期動作のタイム・アウトをそのリスナに通知するのに使われる。

  - public void onError(AsyncEvent event) – このメソッドはその非同期動作が完了するのに失敗したことをそのリスナに通知するのにつかわれる。

  - public void onStartAsync(AsyncEvent event) – このメソッドは新しい非同期サイクルが ServletRequest.startAsyn のメソッドたちのひとつを呼ぶことで開始されていることをそのリスナに通知するのに使われる。再初期化されているその非同期動作に対応した AsyncContext は、与えられたイベントの AsyncEvent.getAsyncContext を呼ぶことで取得できる。

- ある非同期動作がタイム・アウトになったイベントに置いては、そのコンテナは以下のステップをとらねばならない：

  - それによって非同期動作が開始された ServletRequest に登録された総ての AsyncListener のインスタンス上の AsyncListener.onTimeout を呼び出す。

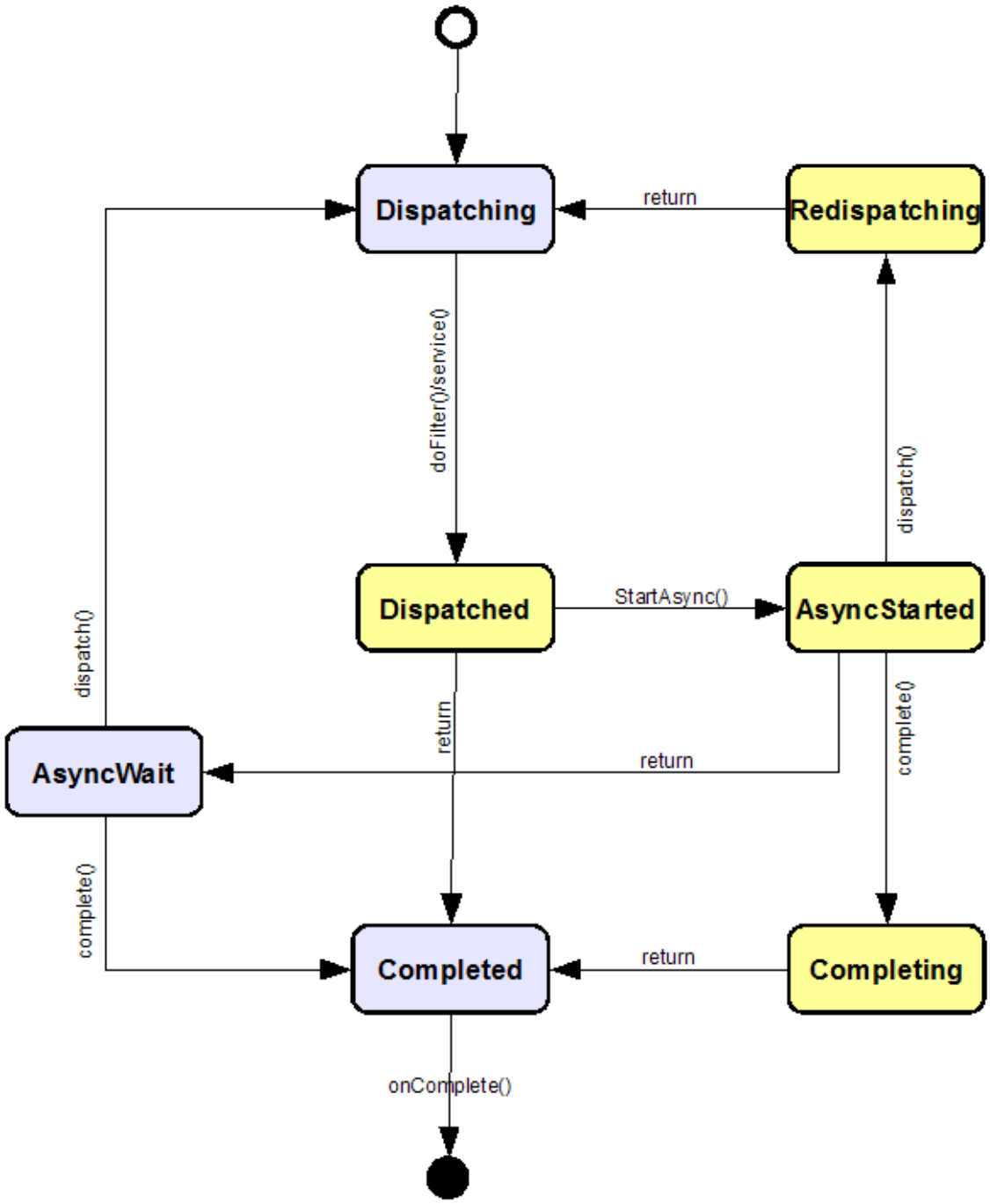
  - どのリスナも AsyncContext.complete()または AsyncContext.dispatch メソッドたちのひとつを呼んでいないときは、HttpServletResponse.SC_INTERNAL_SERVER_ERROR と等しいステータス・コードでエラー・ディスパッチを実行する。

- マッチしたエラー・ページが見つからないとき、あるいはそのエラー・ページが AsyncContext.complete()あるいは AsyncContext.dispatch メソッドたちのどれかを呼んでいないときは、そのコンテナは AsyncContext.complete()を呼ばねばならない。

- それがコンテント発生のために使われており、非同期処理はコンテント発生前になされていなければならないので、JSP のなかでの非同期処理はサポートされない。このケースをどう処理するかはそのコンテナ次第である。一旦総ての同期動作が終了していれば、AsyncContext.dispatch を使った JSP ページへのディスパッチはコンテント生成のために使用できる。

- 図 2-1 はいろんな非同期動作の状態遷移を示したダイヤグラムである。

FIGURE 2-1 State transition diagram for asynchronous operations
図 2-1　非同期動作の状態遷移図



2.3.3.4 スレッド安全（Thread Safety）

startAsync と complete メソッド以外は、要求と応答のオブジェクトの実装物はスレッド安全であることが保障されていない。このことはこれらはその要求処理スレッドの適用範囲内で使われるべきであること、あるいはそのアプリケーションは要求と応答のオブジェクトへのアクセスはスレッド安全であることを確保しなければならないことを意味する。

もしあるアプリケーションが生成したスレッドが要求と応答オブジェクトのようなコンテナが管理しているオブジェクトを使うときは、これらのオブジェクトは章 3.10 及び 5.6 で定められているようにそのオブジェクトのライフ・サイクル内でのみアクセスされねばならない。startAsync 及び complete メソッド以外は、要求と応答のオブジェクトはスレッド安全でないことに注意されたい。もしこれらのオブジェクトが複数のスレッドによってアクセスされるときは、そのアクセスは同期化されるかあるいはスレッド安全を付加、例えば要求の属性にアクセスするためにそのメソッドの呼び出しを同期化する、あるいはスレッド内で応答オブジェクトの為にローカルな出力ストリームを使うなど、したラッパを介してなされるべきである。

2.3.4 サービスの終了（End of Service）

サーブレット・コンテナにはどの時間期間においてもサーブレットをロードしたままにすることは要求されていない。あるサーブレットのインスタンスは数ミリ秒の期間サーブレット・コンテナ内で存続することもあろうし、サーブレット・コンテナのライフタイム（数日、数か月、あるいは数年間にもなり得る）にわたって存続することもあろうし、その間の時間で存続することもある。

サーブレット・コンテナがそのサーブレットをサービスから外すべきだと判断すれば、そのサーブレット・コンテナは Servlet インターフェイスの destroy メソッドを呼び、そのサーブレットが使っている資源を解放し、永続状態を保管できるようにする。例えば、そのコンテナはメモリ資源を節約したいとき、あるいはシャット・ダウン中のときにこれを行う。サーブレット・コンテナがこの destroy メソッドを呼ぶ前に、コンテナはそのサーブレットの service メソッド内で現在走っている総てのスレッドが実行を終える、あるいはサーバが定義した時間期限を超えるのを許さねばならない。

あるサーブレットのインスタンス上で destroy メソッドが一旦呼び出されると、そのコンテナはそのサーブレットのそのインスタンスに他の要求を渡してならない。もしコンテナがそのサーブレットを再度使えるようにする必要が出た場合は、そのコンテナはそのサーブレットのクラスの新しいインスタンスでそれを行わねばならない。

destroy メソッドが終了した後は、そのサーブレット・コンテナはそのサーブレットのインスタンスを開放し、ガーベージ・コレクションの対象にしなければならない。

第3章　　要求（The Request）

要求オブジェクトはクライアント要求の総ての情報をカプセル化している。HTTP プロトコルでは、この情報はその要求の HTTP ヘッダたちとメッセージ・ボディでクライアントからサーバに送信されている。

3.1　HTTP プロトコル・パラメタ（HTTP Protocol Parameters）

そのサーブレットへの要求パラメタはその要求の要素としてサーブレット・コンテナにクライアントから送信される文字列である。その要求が HttpServletRequest オブジェクトの場合で、SRV.3.1.1 の「パラメタが取得できる場合」で定められた条件が満たされているときに、そのコンテナは URI クエリ文字列と POST されたデータからのパラメタたちを集める。これらのパラメタは名前-値のセットとしてストアされる。あるパラメタ名にたいし複数のパラメタ値も存在し得る。

ServletRequest インターフェイスのなかの以下のメソッドがパラメタへのアクセスに使える：

- getParameter
- getParameterNames
- getParameterValues
- getParameterMap

getParameterValues メソッドはあるパラメタ名で集められた総てのパラメタ値を含んだ String オブジェクトの配列を返す。getParameter メソッドで返された値は getParameterValues で返された String オブジェクトの配列の最初の値でなければならない。getParameterMap メソッドはその要求のパラメタの java.util.Map を返し、これにはキーとしての名前とマップ値としてのパラメタ値が含まれている。

クエリ文字列とポスト・ボディからのデータは要求パラメタのセットとして集積される。クエリ文字列データはポスト・ボディ・データの前に示される。例えば、ある要求が a=hello という文字列と a=goodbye&a=world というポスト・ボディでできているときは、結果としてのパラメタのセットは a=(hello, goodbye, world)という順となる。

GET 要求の要素としてのパス・パラメタ（HTTP 1.1 で規定された）はこれらの API では提供されていない。これらは getRequestURI メソッドまたは getPathInfo メソッドで返された String 値を解析されねばならない。

3.1.1　パラメタが取得可能な場合（When Parameters Are Available）

ポスト様式データがパラメタのセットとして集められる前に満たさねばならない条件は以下のようである：

1. その要求が HTTP または HTTPS 要求である。

2. その HTTP メソッドが POST である。
3. コンテントのタイプが application/x-www-form-urlencoded である。
4. そのサーブレットはその要求オブジェクトの getParameter ファミリのメソッドのどれかを最初呼び出した。

これらの条件が満たされず、またポスト形式のデータがそのパラメタ・セットに含まれていないときは、そのポスト・データはそれでもその要求オブジェクトの入力ストリームを介してサーブレットにより取得可能でなければならない。これらの条件が満たされているときは、ポスト様式のデータはその要求オブジェクトの入力ストリームから直接読みだすことはできなくなる。

3.2 ファイルのアップロード（File upload）

もしある要求が multipart/form-data のタイプであってその要求処理が第 8.1.5 の第 5-8.1.5 の “@MultipartConfig” で示された@MultipartConfig を使ってアノテートされているときは、その HttpServletRequest は以下のメソッドたちによりそのマルチパート要求のいろんな部分を取得可能になる：

- public Collection<Part> getParts()
- public Part getPart(String name)

各パートがヘッダたち、それに関連したコンテント・タイプ、及び getInputStream を介したコンテントへのアクセスを提供する。Content-Disposition としての form-data を持っているがファイル名を持っていない要素に関しては、そのパートの文字列値が HttpServletRequest 上の getParameter / getParameterValues メソッドを介して、そのパートの名前を介して取得できるようになる。

3.3 属性（Attributes）

属性はある要求に関連付けされたオブジェクトである。属性は API を介しては表現できない情報を表現するためにコンテナによって設定される、あるいは他のサーブレットと情報を通信する（RequestDispatcher を介して）ためにサーブレットによって設定される。属性は ServletRequest インターフェイスの以下のメソッドによってアクセスされる：

- getAttribute
- getAttributeNames
- setAttribute

ある属性名ではただ一つの属性値が関連付けされる。

“java.”および”javax.”プレフィックスで始まる属性名はこの仕様の定義の為に予約されている。同じように、”sun.”及び”com.sun.”のプレフィックスで始まる属性名は Sun Microsystems による定義のために予約されている。属性セットで置かれる総ての属性はパッケージの名前つけの為の Java プログラミング言語仕様 1 で示されている逆ドメイン名規約に準拠して名前がつけられることが推奨されている。

1. Java プログラミング言語仕様は以下のアドレスで取得できる:
http:// java.sun.com/docs/books/jls

3.4 ヘッダ（Headers）

サーブレットは HttpServletRequest インターフェイスの以下のメソッドを介して HTTP 要求のヘッダたちにアクセスできる:

- getHeader
- getHeaders
- getHeaderNames

getHeader メソッドはそのヘッダの与えられた名前のヘッダを返す。ある HTTP 要求には例えば Cache-Control ヘッダのように同じ名前を持った複数のヘッダを持ち得る。同じ名前で複数のヘッダが存在するときは、その要求のなかの最初のヘッダを返す。getHeaders メソッドでは特定のヘッダ名をもったヘッダ値の総てを String オブジェクトの Enumeration を返すことでアクセスできるようにしている。

ヘッダは int または Date データの String 表現を含むことがある。HttpServletRequest インターフェイスの以下の便利なメソッドによりこれらの様式のひとつの形でヘッダ・データにアクセスできる:

- getIntHeader
- getDateHeader

もし getIntHaeder メソッドがそのヘッダを int に変換できないときは NumberFormatException がスローされる。もし getDateHeader メソッドがそのヘッダを Date オブジェクトに変換できないときは、IllegalArgumentException がスローされる。

3.5 要求パス要素（Request Path Elements）

ある要求をサービスしているあるサーブレットへ導く要求パスは多くの重要な区間で構成されている。以下の要素が要求 URI パスから取得されており、また要求オブジェクトを介して取得できる:

- コンテキスト・パス(Context Path):このサーブレットがその要素である ServletContext と結びつけられたパス・プレフィックス。もしこのコンテキストがウェブ・サーバの URL 名空間のベースにある「デフォルト」のコンテキストである場合は、このパスは空の文字列になる。そうでない場合は、もしそのコンテキストがそのサーバの名前空間のルートでないときは、このパスは'/'文字で始まるが、'/'文字では終了しない。
- サーブレット・パス(Servlet Path):この要求をアクチベートしたマッピングに直接対応したパス区間。このパスは'/'文字で始まるが、例外はその要求が'/*'パタンにマッチしている場合で、その場合は空の文字列となる。

- パス情報(PathInfo):コンテキスト・パスまたはサーブレット・パスの要素でない要求パスの要素である。これは更なるパスが無い時は空(null)、あるいは'/'で始まる文字列のいずれかである。

この情報にアクセスするために HttpServletRequest インターフェイスには以下のようなメソッドが存在する:

- getContextPath
- getServletPath
- getPathInfo

要求 URI とパス要素間の URL エンコーディングの相違を除けば、以下の式が常に成り立つことを指摘することが重要である:

requestURI = contextPath + servletPath + pathInfo

上記のポイントを明確化するために幾つかの事例を示す為に、以下のものを考えてみる:

表 3-1:コンテキスト設定例

コンテキスト・パス	/catalog
サーブレット・マッピング	パタン: /lawn/* サーブレット:LawnServlet
サーブレット・マッピング	パタン: /garden/* サーブレット:GardenServlet
サーブレット・マッピング	パタン: */jsp サーブレット:JSPServlet

以下の振る舞いが観察されることになる:

表 3-2:観察されたパス要素の振る舞い

要求パス	パス要素
/catalog/lawn/index.html	ContextPath: /catalog ServletPath: /lawn PathInfo: /index.html
/catalog/garden/implements/	ContextPath: /catalog ServletPath: /garden PathInfo: /implements/
/catalog/help/feedback.jsp	ContextPath: /catalog ServletPath: /help/feedback.jsp PathInfo: null

3.6 パス変換のためのメソッド(Path Translation Methods)

開発者たちが特定のパスに等価なファイル・システム・パスを取得できる便利なメソッドが API のなかに 2 つ存在する。これらのメソッドは:

- ServletContext.getRealPath
- HTTPServletRequest.getPathTranslated

getRealPath メソッドは String 引数を得てあるパスが対応するローカルなファイル・システム上のファイルを代表する String を返す。GetPathTranslated メソッドはその要求の pathInfo の実際のパスを計算する。

サーブレット・コンテナがこれらのメソッドのための有効なファイル・パスが判断できなかった場合、例えばそのウェブ・アプリケーションがアーカイブから実行されている、ローカルにはアクセスできないリモートのファイル・システム上にある、あるいはデータ・ベース内にある、といった場合は、これらのメソッドは空を返さねばならない。

JAR ファイルの META-INF/リソース・ディレクトリのなかにあるリソースは、そのコンテナが getRealPath()呼び出しがなされたときに自分たちが含んでいる JAR ファイルからそれらを解凍したときに限り検討されるべきで、そのときは解凍された場所を返さ**ねばならない**。

3.7 クッキー（Cookies）

HttpServletRequest インターフェイスはその要求で示されているクッキーの配列を取得するための getCookies メソッドを用意している。これらのクッキーはクライアントがする各要求でクライアントからサーバに送信されるデータである。一般的にはクッキーの要素としてクライアントが送り返してくる唯一の情報はクッキー名とクッキー値である。ブラウザにクッキーを送るときにセットされ得る他のクッキー属性（例えばコメント）は、一般的には返されない。本仕様はまた HttpOnly クッキーであるクッキーを可能としている。HttpOnly クッキーはクライアントに対しこれらのクッキーがクライアント・サイドのスクリプトのコードでは見えなくするべきであることを示している（これはそのクライアントがこの属性を探すことを知っていないかぎりフィルタされない）。HttpOnly クッキー使用はある種のサイトをまたぐスクリプティング攻撃を最小化するのに寄与する。

3.8 SSL 属性（SSL Attributes）

ある要求が例えば HTTPS のような安全化されたプロトコル上で送信されてきた場合は、この情報は ServletRequest インターフェイスの isSecure メソッドによって与えられねばならない。ウェブ・コンテナはサーブレットのプログラマに対し以下のような属性を与えねばならない：

表 3-3：プロトコル属性

属性	属性名	Java の型
暗号化セット	`javax.servlet.request.cipher_suite`	`String`
そのアルゴリズムのビット長	`javax.servlet.request.key_size`	`Integer`

SSL セッション ID	javax.servlet.request.ssl_session_id	String

その要求に関し SSL 認証がされている場合は、それはサーブレットのプログラマに対し javax.servlet.request.X509Certificate 型のオブジェクトの配列で示されねばならないし、javax.servlet.request.X509Certificate の ServletRequest 属性を介してアクセスできねばならない。

この配列の順は信頼度の降順であると定義されている。そのチェインの最初の認証はクライアントがセットしたものであり、次は最初の認証につかわれたもの、以下同様となる。

3.9 国際化（Internationalization）

クライアントは与えられる応答に対しどの言語が好ましいかを示すことがある。この情報は HTTP/1.1 仕様で規定された他のメカニズムとともに Accept-Language ヘッダを使ってクライアントから通信できる。ServletRequest インターフェイスには送信先にとって好ましいロケールを判断するための 2 つのメソッドが用意されている：

- getLocale
- getLocales

getLocale はそのクライアントがコンテントを受け入れる為に望んでいる好ましいロケールを返す。クライアントにとって好ましい言語を判断するためにこの Accept-Language ヘッダをどのように解釈するかに関する更なる情報は RFC 2616 (HTTP/1.1)の 14.4 節を見られたい。getLocales メソッドは Locale の Enumeration を返し、降順に好ましいロケールから始まって、クライアントが受け入れることが可能なロケールを示す。クライアントから特に好ましいロケールを指定していないときは、getLocale メソッドで返されるロケールはそのサーブレット・コンテナのデフォルトのロケールでなければならず、getLocales メソッドで返されるロケールはデフォルトのロケールの単一の Locale 要素の列挙を含まねばならない。

3.10 要求データのエンコーディング（Request data encoding）

現在多くのブラウザが Content-Type ヘッダで char エンコーディング識別子を送信しておらず、HTTP 要求を読むときの文字エンコーディングの判断をオープンな状態にしている。クライアント要求で指定されていない場合は、コンテナが要求リーダをつくり POST データを解析するために使うデフォルトのエンコーディングは"ISO-8859-1”でなければならない。しかしながらクライアントが文字エンコーディングを送っていないことを開発者に示すために、この場合はコンテナは getCharacterEncoding メソッドでは null を返さねばならない。

もしクライアントが文字エンコーディングを設定しておらず、その要求データが上記のデフォルトのエンコーディングでない場合は、破損が起きうる。このような状況を是正するために、ServletRequest インターフェイスには新しいメソッドの setCharacterEncoding(String enc)が追加されている。開発者

たちはこのメソッドを呼ぶことでコンテナが提供する文字エンコーディングをオーバライドできる。
データが読まれてしまったあとでこのメソッドを読んでもエンコーディングに影響を与えない。

3.11 要求オブジェクトの生存期間（Lifetime of the Request Object）

その要素のために非同期処理が可能になっておりその要求オブジェクトで startAsync メソッドが呼び出されたとき以外は、各要求オブジェクトはあるサーブレットの service メソッドの適用範囲内、あるいはフィルタの doFilter メソッドの適用範囲内でのみ有効である。コンテナは一般的に要求オブジェクトのリサイクルをして、要求オブジェクトの生成の為の性能オーバヘッドを回避している。開発者は上記の適用範囲外で要求オブジェクトへの参照を維持することは、それが不確定な結果をもたらす可能性があるので勧められないということに注意しなければならない。

第4章　　サーブレット・コンテキスト（Servlet Context）

4.1　ServletContext インターフェイス（Introduction to the ServletContext Interface）

ServletContext インターフェイスはそのなかでサーブレットが走っているウェブ・アプリケーションをサーブレットからみた環境を定義している。コンテナの提供者はそのサーブレット・コンテナに ServletContext インターフェイスを実装する責を持つ。ServletContext オブジェクトを使うことで、サーブレットはイベントのログ、資源への URL 参照の取得、及びそのコンテキスト内の他のサーブレットがアクセスできる属性の設定と蓄積、ができるようになる。

ServletContext はウェブ・サーバ内の知られたパスとしてルートづけられている。例えば、あるサーブレット・コンテキストは http://www.mycorp.com/catalog に位置できる。/catalog 要求パス（コンテキスト・パス”context path”として知られる）で始まる総ての要求はその ServletContext で関連付けられたウェブ・アプリケーションに渡される。

4.2　ServletContext インターフェイスの適用範囲（Scope of a ServletContext Interface）

あるコンテナに配備された各ウェブ・アプリケーションに関連付けされた ServletContext インターフェイスのインスタンスがひとつ存在する。そのコンテナが多くの仮想マシンに分散された場合では、あるウェブ・アプリケーションは各 JVM ごとにひとつの ServletContext のインスタンスを持つことになる。

あるウェブ・アプリケーションの要素として配備されていないあるコンテナ内のサーブレットたちは、暗示的に「デフォルト」ウェブ・アプリケーションの要素であり、デフォルトの ServletContext を持つ。分散コンテナのなかでは、この「デフォルト」の ServletContext は分散不可でああり、ひとつの JVM のなかでのみ存在しなければならない。

4.3　初期化パラメタ（Initialization Parameters）

以下の ServletContext インターフェイスのメソッドにより、アプリケーション開発者が配備記述子で指定したウェブ・アプリケーションに関連付けされたコンテキスト初期化パラメタにそのサーブレットがアクセスできる:

- getInitParameter

- getInitParameterNames

初期化パラメタはアプリケーションの開発者が設定情報を伝えるのに使われる。一般的な事例はウェブマスタの電子メール・アドレス、あるいは重要なデータを保持しているシステムの名前などである。

4.4 設定法（Configuration methods）

Servlet 3.0 以降 ServletContext に以下のメソッドたちが追加されており、サーブレット、フィルタ、及びそれらをマップするための URL パタンをプログラム的に定義できるようになっている。これらのメソッドは ServletContextListener 実装の contexInitialized メソッドから、あるいは ServletContainerInitializer 実装の onStartup メソッドからのいずれかでそのアプリケーションの初期化中でのみ呼び出すことができる。サーブレットたちとフィルタたちの追加に加えて、あるサーブレットまたはフィルタに対応した Registration オブジェクトのあるインスタンス、あるいはそれらのサーブレットまたはフィルタの総ての Registration オブジェクトのマップ、を検索できる。もし ServletContextListener の contextInitialized に渡されたその ServletContext が web.xml または webfragment. xml で宣言されていない、または@WebListener でアノテートされていないときは、UnsupportedOperationException がサーブレット、フィルタ、及びリスナのプログラム的な設定のために定義された総てのメソッドでスローされ**ねばならない**。

4.4.1 サーブレットのプログラム的な追加と設定（Programmatically adding and configuring Servlets）

あるコンテキストにプログラム的にあるサーブレットが追加できるということはフレームワークの開発者たちには有用なことである。例えばあるフレームワークはこのメソッドを使ってあるコントローラ・サーブレットを宣言できる。このメソッドの戻り値は ServletRegistration あるいは ServletRegistration.Dynamic オブジェクトであり、これはさらに init-params、url-mappings のようなサーブレットの他のパラメタの設定も可能になる。以下に記すようにこのメソッドの 3 つのオーバロードされたバージョンが存在する。

4.4.1.1 addServlet(String servletName, String className)

このメソッドによりそのアプリケーションはプログラム的にあるサーブレットを宣言できる。このメソッドは与えられた名前、及びクラス名をもったそのサーブレットをそのサーブレット・コンテキストに追加する。

4.4.1.2 addServlet(String servletName, Servlet servlet)

このメソッドによりそのアプリケーションはプログラム的にあるサーブレットを宣言できる。このメソッドは与えられた名前、及びサーブレットのインスタンスをもったそのサーブレットをそのサーブレット・コンテキストに追加する。

4.4.1.3 addServlet(String servletName, Class <? extends Servlet> servletClass)

このメソッドによりそのアプリケーションはプログラム的にあるサーブレットを宣言できる。このメソッドは与えられた名前、及びサーブレットのクラスをもったそのサーブレットをそのサーブレット・コンテキストに追加する。

4.4.1.4 <T extends Servlet> T createServlet(Class<T> clazz)

このメソッドは与えられた Servlet クラスをインスタンス化する。このメソッドは@WebServlet を除く Servlets に適用できる総てのアノテーションに対応しなければならない。戻された Servlet インスタンスは上で定義されたように addServlet(String, Servlet)呼び出しを介して ServletContext に登録される前に更にカスタム化されても良い。

4.4.1.5 ServletRegistration getServletRegistration(String servletName)

このメソッドは与えられた名前を持ったサーブレットに対応した ServletRegistration を、またはその名前の ServletRegistration が無い場合は null を返す。もし web.xml または web-fragment.xml で宣言されていない、あるいは javax.servlet.annotation.WebListener でアノテートされていない ServletContextListener の contextInitialized メソッドにその ServletContext が渡されたときには UnsupportedOperationException がスローされる。

4.4.1.6 Map<String, <? extends ServletRegistration> getServletRegistrations()

このメソッドはその ServletContext で登録された総てのサーブレットに対応した名前をキーとした ServletRegistration のオブジェクトのマップを返す。もしその ServletContext で登録されたサーブレットが存在しないときは空のマップが返される。戻された Map は総ての宣言された及びアノテート

されたサーブレットに対応した ServletRegistration オブジェクトたち、及び addServlet メソッドたちのひとつを介して追加された総てのサーブレットに対応した ServletRegistration たちを含んでいる。戻された Map への何らかの変更はその ServletContext に影響を与えては**いけない**。もし web.xml または web-fragment.xml で宣言されていない、あるいは javax.servlet.annotation.WebListener でアノテートされていない ServletContextListener の contextInitialized メソッドにその ServletContext が渡されたときには UnsupportedOperationException がスローされる。

4.4.2 フィルタのプログラム的な付加と設定（Programmatically adding and configuring Filters）

4.4.2.1 addFilter(String filterName, String className)

このメソッドによりアプリケーションはプログラム的にフィルタを宣言できる。このメソッドは与えられた名前とクラス名を持ったフィルタをそのウェブ・アプリケーションに追加する。

4.4.2.2 addFilter(String filterName, Filter filter)

このメソッドによりアプリケーションはプログラム的にフィルタを宣言できる。このメソッドは与えられた名前とフィルタ・インスタンスを持ったフィルタをそのウェブ・アプリケーションに追加する。

4.4.2.3 addFilter(String filterName, Class <? extends Filter> filterClass)

このメソッドによりアプリケーションはプログラム的にフィルタを宣言できる。このメソッドは与えられた名前とサーブレット・クラスのインスタンスを持ったフィルタをそのウェブ・アプリケーションに追加する。

4.4.2.4 <T extends Filter> T createFilter(Class<T> clazz)

このメソッドは与えられた Filter クラスをインスタンス化する。このメソッドは Filters に適用できる総てのアノテーションをサポートしなければならない。返された Filter のインスタンスは上記で定められたように addServlet(String, Filter)への呼び出しを介してそれが ServletContext に登録される前に更

にカスタム化出来る。与えられた Filter クラスはそれをインスタンス化するための引数なしのコンストラクタを定義しなければならない。

4.4.2.5 FilterRegistration getFilterRegistration(String filterName)

このメソッドは与えられた名前を持ったフィルタに対応した FilterRegistration を、あるいはもしその名前の FilterRegistration が存在しないときは null を返す。もし web.xml または web-fragment.xml で宣言されていない、あるいは javax.servlet.annotation.WebListener でアノテートされていない ServletContextListener の contextInitialized メソッドにその ServletContext が渡されたときには UnsupportedOperationException がスローされる。

4.4.2.6 Map<String, <? extends FilterRegistration> getServletRegistrations()

このメソッドは ServletContext に登録された総てのサーブレットに対応した名前をキーとした、ServletRegistration オブジェクトたちのマップを返す。もしその ServletContext に登録されたサーブレットが無い時は空のマップが返される。返されたマップには総ての宣言された及びアノテートされたサーブレットに対応した ServletRegistration オブジェクトたち、及び addServlet メソッドたちのひとつを介して追加された総ての ServletRegistration オブジェクトたちが含まれる。戻された Map への何らかの変更はその ServletContext に影響を与えては**いけない**。もし web.xml または web-fragment.xml で宣言されていない、あるいは javax.servlet.annotation.WebListener でアノテートされていない ServletContextListener の contextInitialized メソッドにその ServletContext が渡されたときには UnsupportedOperationException がスローされる。

4.4.3 リスナのプログラム的な追加と設定（Programmatically adding and configuring Listeners）

4.4.3.1 void addListener(String className)

与えられたクラス名を持ったリスナを ServletContext に追加する。ServletContext で代表されるそのアプリケーションにかかわるクラスローダによって与えられた名前を持ったクラスがロードされ、以下のインターフェイスたちのひとつまたはそれ以上を実装しなければ**ならない**。

- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener

- javax.servlet.http.HttpSessionAttributeListener

もしその ServletContext がその ServletContainerInitializer の onStartup メソッドに渡されたときは、次に与えられた名前を持ったそのクラスはまた上記のインターフェイスたちに加えて javax.servlet.ServletContextListener を実装**しても良い**。このメソッド呼び出しの一部として、そのコンテナは要求されたインターフェイスたちのひとつを確実に実装するために指定されたクラス名を持ったクラスをロードしなければ**ならない**。もし与えられた名前を持ったそのクラスがその呼び出し順序が宣言の順番に対応しているリスナ・インターフェイスを実装している、即ちもしそれが javax.servlet.ServletRequestListener または javax.servlet.http.HttpSessionListener を実装しているときは、新しいリスナはそのインターフェイスのリスナたちの順序リストの終わりに追加されることになる。

4.4.3.2 <T extends EventListener> void addListener(T t)

ServletContext に与えられたリスナを追加する。与えられたリスナは以下のインターフェイスたちのひとつまたはそれ以上のインスタンスでなければならない:

- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener

もしその ServletContext がその ServletContainerInitializer の onStartup メソッドに渡されたときは、次に与えられた名前を持ったそのクラスはまた上記のインターフェイスたちに加えて javax.servlet.ServletContextListener を実装**しても良い**。もし与えられたリスナががその呼び出し順序が宣言の順番に対応しているリスナ・インターフェイスのインスタンスである、即ちもしそれが javax.servlet.ServletRequestListener または javax.servlet.http.HttpSessionListener を実装しているときは、新しいリスナはそのインターフェイスのリスナたちの順序リストの終わりに追加されることになる。

4.4.3.3 void addListener(Class <? extends EventListener> listenerClass)

与えられたクラス・タイプのリスナを ServletContext に追加する。与えられたリスナ・クラスは以下のインターフェイスのひとつまたはそれ以上を実装しなければ**ならない**。

- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener

もしその ServletContext がその ServletContainerInitializer の onStartup メソッドに渡されたときは、次に与えられた名前を持ったそのクラスはまた上記のインターフェイスたちに加えて javax.servlet.ServletContextListener を実装**しても良い**。もし与えられたリスナ・クラスがそのクラスが

その呼び出し順序が宣言の順番に対応しているリスナ・インターフェイスを実装している、即ちもしそれが javax.servlet.ServletRequestListener または javax.servlet.http.HttpSessionListener を実装しているときは、新しいリスナはそのインターフェイスのリスナたちの順序リストの終わりに追加されることになる。

4.4.3.4 <T extends EventListener> void createListener(Class<T> clazz)

このメソッドは与えられた EventListener クラスをインスタンス化する。指定された EventListener クラスは以下のインターフェイスの少なくともひとつを実装しなければならない:

- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener

このメソッドはこの仕様で定義されているように上記リスナ・インターフェイスに適用できる総てのアノテーションをサポートしなければ**ならない**。返された EventListner インスタンスはそれが addListener(T t)呼び出しを介して ServletContext に登録される前に更にカスタマイズされても良い。与えられた EventListner クラスはそれをインスタンス化するために使われる引数なしのコンストラクタを定義しなければ**ならない**。

4.4.3.5 プログラム的に追加されたサーブレット、フィルタ、及びリスナの為のアノテーション処理の要求事項(Annotation processing requirements for programmatically added Servlets, Filters and Listeners)

あるインスタンスをとる addServlet とは別に、あるサーブレットを追加または生成するためにプログラム的な API を使う際は、以下のアノテーションたちが問題のクラスのなかで内省されねばならないし、そのなかで定義されたメタデータは、ServletRegistration.Dynamic / ServletRegistration のなかの API への呼び出しによってオーバライドされない限り、使われ**ねばならない**。
@ServletSecurity, @RunAs, @DeclareRoles, @MultipartConfig.

フィルタとリスナに関してはアノテーションの内省の必要はない。

インスタンスをとるメソッドを介して追加されたもの以外は、プログラム的に追加されたあるいは生成された総ての要素(サーブレット、フィルタ、及びリスナ)は、その要素が Managed Bean(管理された Bean)であるときにのみサポートされる。Managed Bean とはなにかの詳細に関しては Java EE 6 及び JSR 299 の一部として定義されている Managed Bean 仕様を参照されたい。

4.5 コンテキスト属性（Context Attributes）

サーブレットはそのコンテキストにオブジェクト属性を名前でバインドできる。あるコンテキストにバインドされたどの属性も同じウェブ・アプリケーションの要素である他のどのサーブレットで利用できる。ServletContext インターフェイスの以下のメソッドたちがこの機能へのアクセスを可能にしている：

- setAttribute
- getAttribute
- getAttributeNames
- removeAttribute

4.5.1 分散コンテナ内のコンテキスト属性（Context Attributes in a Distributed Container）

コンテキスト属性はそれらが生成された JVM にとってローカルなものである。これにより ServletContext 属性が分散されたコンテナ内で共有されたメモリ・ストアになるのを防いでいる。情報が分散環境で走っているサーブレットたち間で共有される必要がある場合は、その情報はセッションに置かれる（第 7 章、「セッション」を見られたい）、データベースにストアされる、あるいは Enterprise JavaBeans™ にセットされるかしなければならない。

4.6 リソース（Resources）

ServletContext インターフェイスは ServletContext インターフェイスの以下のメソッドを介して HTML、GIF、及び MPEG ファイルのような、ウェブ・アプリケーションの要素である静的なコンテントのドキュメントに対してのみへの直接的なアクセスを提供している：

- getResource
- getResourceAsStream

getResource と getResourceAsStream メソッドはそのコンテントのルートに相対的なそのリソースへのパスを与える“/”が先頭の String を引数としている。これらドキュメントのこの階層はそのサーバのファイル・システムのなか、あるウェブ・アプリケーションのアーカイブ・ファイル内、リモートのサーバ上、あるいは他のどこかの場所に存在し得る。

これらのメソッドはダイナミックなコンテント取得には使われない。例えば、JavaServer Pages™ 仕様に対応したコンテナのなかで、getResource("/index.jsp")のかたちでのメソッド呼び出しは JSP ソース・コードを返し、処理された出力を返さない。ダイナミックなコンテントへのアクセスに関する更なる詳細は、第 9 章の「要求ディスパッチ（転送）」を参照されたい。

そのウェブ・アプリケーションのリソースの全部のリスティングは getResourcePaths(String path)メソッドによってアクセスできる。このメソッドに意味に関する完全な詳細はこの仕様書の API ドキュメンテーションで示されている。

1. The JavaServer Pages™ 仕様書は以下のアドレスから取得できる:
http://java.sun.com/products/jsp

4.7 複数のホストとサーブレット・コンテキスト（Multiple Hosts and Servlet Contexts）

ウェブ・サーバはあるサーバ上でひとつの IP アドレスを共有する複数の論理的なホストをサポートすることがある。この機能は「バーチャル・ホスティング」とも呼ばれている。この場合、各論理的なホストは自分のサーブレット・コンテキストあるいはサーブレット・コンテキストたちのセットを持たねばならない。

4.8 再ローディングに関する考察（Reloading Considerations）

コンテナのプロバイダが開発し易さのためのクラス再ローディングのスキームを実装することが要求されてはいないが、そのような実装は使用するであろう総てのサーブレットとクラスたち[2]を単一のクラス・ローダの適用範囲内に確実にロードするようにしなければならない。この要求はそのアプリケーションが開発者が期待したとおりにふるまうことを保障するために必要なことである。開発支援のために、クラス再ローディングに伴うセッションの終了の監視用に、セッション・バインディング・リスナへの通知の総ての意味をコンテナがサポートしなければならない。

以前の世代のコンテナはあるサーブレットをロードするのに新しいクラス・ローダを生成していて、これはそのサーブレット・コンテキスト内で使われている他のサーブレットたちまたはクラスたちのロードに使われているクラス・ローダとは違っていた。このことがあるサーブレット・コンテキスト内でのオブジェクト参照が予期せぬクラスまたはオブジェクトを差す原因となり、また予期せぬ振る舞いをもたらす原因になっていた。この要求は新しいクラス・ローダのデマンド生成によって引き起こされる問題を防止するために必要なものである。

2. 例外はそのサーブレットが別のクラス・ローダ内で使うかもしれないシステム・クラスたちである。

4.8.1 一時的な作業ディレクトリ（Temporary Working Directories）

各サーブレット・コンテキストごとに一時的な蓄積ディレクトリが必要になる。サーブレット・コンテナは各サーブレット・コンテキストごとにプライベートな一時的なディレクトリを提供し、javax.servlet.context.tempdir コンテキスト属性を介して利用できるようにしなければならない。この属性に結び付けられたオブジェクトたちは java.io.File 型でなければならない。

この要求は多くのサーブレット・エンジン実装物が提供している共通的な利便性を認識したものである。そのサーブレット・コンテナが再スタートしたときにその一時的なディレクトリの内容を維持する

ことは要求されていないが、ひとつのサーブレット・コンテキストの一時的なディレクトリの内容が、そのサーブレット・コンテナ上で走っている他のウェブ・アプリケーションからは不可視であることが要求されている。

第5章　　応答（The Response）

応答のオブジェクトはサーバからクライアントに返すことになる総ての情報をカプセル化している。HTTP プロトコルでは、この情報はその応答の HTTP ヘッダたちまたはメッセージ・ボディ部のいずれかでサーバからクライアントに送信される。

5.1　バッファリング（Buffering）

サーブレット・コンテナは効率上の目的からクライアントへ行く出力をバッファすることが、要求されてはいないが、認められている。一般的にバッファリングを可能としているサーバはそれをデフォルトとしているが、サーブレットがバッファリングのパラメタを指定できるようにしている。

ServletResponse インターフェイスの以下のメソッドたちによりサーブレットがバッファリングのパラメタにアクセスとセットができるようにしている：

- getBufferSize
- setBufferSize
- isCommitted
- reset
- resetBuffer
- flushBuffer

これらのメソッドたちはそのサーブレットが ServletOutputStream を使っていようと Writer を使っていようとバッファリングができるように用意されている。

getBufferSize メソッドは使われているバッファのサイズを返す。もしバッファリングが使われていないときは、このメソッドは 0（ゼロ）の int 値を返さねばならない。

サーブレットは setBufferSize メソッドを使うことで好ましいバッファ・サイズを要求できる。割り当てられたバッファはそのサーブレットが要求したサイズであることは要求されていないが、少なくとも要求されたサイズの大きさでなければならない。これによりコンテナは固定されたサイズのバッファたちを再利用し、もし適切なら要求された以上の大きなバッファを提供できる。このメソッドは ServletOutputStream または Writer によって何らかのコンテントが書き込まれる前に呼ばれねばならない。もし何らかのコンテントが書かれてしまっている、あるいはその応答オブジェクトがコミットされてしまっているときは、このメソッドは IllegalStateException をスローしなければならない。

isCommitted メソッドは何らかの応答のバイトがクライアントに返されたかどうかを示す boolean 値を返す。FlushBuffer メソッドはそのバッファ内のコンテントをクライアントに書き込ませる。

reset メソッドはその応答がコミットされていないときはそのバッファ内のデータをクリアする。reset 呼び出し前にサーブレットが設定したヘッダたちとステータス・コードたちも同じくクリアされねばならない。resetBuffer メソッドは応答がまだコミットされていないときはそのバッファ内のコンテントをクリアするが、ヘッダたちとステータス・コードたちはクリアしない。

もしその応答がコミットされており reset または resetBuffer メソッドが呼ばれたときは、IllegalStateException がスローされる。その応答とそれにかかわるバッファは変化しない。

バッファを使用中は、そのコンテナはいっぱいになったバッファのコンテンツを即座にフラッシュしなければならない。もしこれがクライアントに送られた最初のデータだったときは、その応答はコミットされたと考えられる。

5.2 ヘッダたち（Headers）

サーブレットは HttpServletResponse インターフェイスの以下のメソッド群を使って HTTP 応答のヘッダたちをセットできる：

- setHeader
- addHeader

setHeader は与えられた名前と値を持ったヘッダをセットする。これまでのヘッダは新しいヘッダによって置き換えられる。その名前を持ったヘッダ値が存在するときは、これらの値はクリアされ新しい値で置き換えられる。addHeader メソッドは与えられた名前を持ったセットにあるヘッダ値を追加する。その名前をもったヘッダが既にないときは、新しいセットがつくられる。

ヘッダたちは int または Date オブジェクトを代表するデータを含んでも良い。HttpServletResponse インターフェイスの以下の利便性のためのメソッドたちにより、サーブレットはしかるべきデータ・タイプのための正しいフォーマッティングを使ってヘッダをセットできる：

- setIntHeader
- setDateHeader
- addIntHeader
- addDateHeader

正しくクライアントに送り返されるためには、ヘッダたちはその応答がコミットされる前にセットされねばならない。応答がコミットされた後でセットされたヘッダはサーブレット・コンテナによって無視される。

サーブレットのプログラマたちはサーブレットが生成しているコンテントの為の応答オブジェクト内に Content-Type ヘッダを適正にセットする責任がある。HTTP 1.1 仕様は HTTP 応答のなかにこのヘッダをセットすることを要求していない。サーブレットのコンテナはサーブレットのプログラマがこのタイプをセットしていないときにデフォルトのコンテント・タイプをセットしてはいけない。

コンテナはその実装情報を示すために X-Powered-By という HTTP ヘッダを使うことが勧告されている。フィールド値は"Servlet/3.0"のようにひとつあるいはそれ以上の実装タイプで構成すべきである。オプション的には、そのコンテナともとになっている Java プラットホームの補完的なデータもカッコ内で実装タイプのあとに追加できる。コンテナはこのヘッダを抑制するよう設定できなければならない。以下はこのヘッダの事例である。

- X-Powered-By: Servlet/3.0
- X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish v3 JRE/1.6.0)

5.3 利便性のためのメソッドたち（Convenience Methods）

以下の利便性のためのメソッドたちが HttpServletResponse インターフェイスにある：

- sendRedirect
- sendError

sendRedirect メソッドはクライアントを異なった URL にリダイレクト（差し向ける）為にしかるべきヘッダたちとコンテント・ボディをセットする。相対 URL パスでこのメソッドを呼ぶことは違反ではない、しかしながらコンテナはその相対パスを完全な型の URL に変換してクライアントに送り返さねばならない。もし部分的な URL が与えられており、そしてどんな理由であっても、有効な URL に変換できないときは、このメソッドは IllegalArgumentException をスローしなければならない。

sendError メソッドはエラー・メッセージをクライアントに返すためにしかるべきヘッダたちとコンテント・ボディをセットする。この sendError メソッドにはオプション的に String 引数を提供し、そのエラーのコンテント・ボディに使えるようにできる。

これらのメソッドは応答が既にコミットされていないときにその応答をコミットし、送信してしまうという副作用を持つ。これらのメソッドが呼ばれた後は、クライアントへの更なる出力をそのサーブレットはしてはならない。これらのメソッドが呼ばれた後にその応答にデータが書かれたときは、そのデータは無視される。

もし応答バッファにデータが書かれてしまっていて、まだクライアントに返されていない（言い換えるとその応答はまだコミットされていない）ときは、その応答バッファ内のデータはクリアされ、これらのメソッドでセットされたデータで置き換えられねばならない。その応答がコミットされているときは、これらのメソッドは IllegalStateException をスローしなければならない。

5.4 国際化（Internationalization）

サーブレットは応答のロケールと文字エンコーディングをセットしなければならない。ロケールは ServletResponse.setLocale メソッドを使ってセットされる。このメソッドは繰り返し呼び出し可能であるが、応答がコミットされた後での呼び出しは効果を与えない。もしそのサーブレットがそのページがコミットされる前にそのロケールをセットしないときは、その応答のロケールを判断するのにデフォルトのロケールが使われるが、HTTP の場合の Content-Language ヘッダのようなクライアントとの通信のための指定はされない。

```
<locale-encoding-mapping-list>
      <locale-encoding-mapping>
            <locale>ja</locale>
            <encoding>Shift_JIS</encoding>
      </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

もしその要素が存在しないあるいはマッピングを提供していないときは、setLocale はコンテナ依存のマッピングを使う。setCharacterEncoding、setContentType、及び setLocale のメソッドたちは文字エンコーディングを変更するために繰り返し呼び出すことができる。そのサーブレット応答の getWriter メソッドが呼ばれてしまっているあるいはその応答がコミットされている後でなされた呼び出しは文字エンコーディングには影響を与えない。setContentType の呼び出しはこれまでに setCharacterEncoding も setContentType もどちらも文字エンコーディングをセットしていないときに文字エンコーディングをセットする。もしそのサーブレットが ServletResponse インターフェイスの getWriter メソッドが呼ばれているあるいはその応答がコミットされているより前に文字エンコーディングを指定しなかったときは、デフォルトである ISO-8859-1 が使用される。

もしそのプロトコルがそうする手段を提供しているときは、コンテナはクライアントにそのサーブレット応答のライタに使われているロケールと文字エンコーディングを通信しなければならない。HTTP の場合は、ロケールは Content-Language ヘッダを介して、文字エンコーディングはテキスト・メディア・タイプのための Content-Type の一部として通信されている。サーブレットがコンテント・タイプを指定しないときは、文字エンコーディングは HTTP ヘッダによっては通信されないが、しかしながらそのサーブレット応答のライタを介して書き込まれるテキストのエンコードに使われていることに注意されたい。

5.5 応答オブジェクトのクローズ（Closure of Response Object）

応答がクローズして（閉じて）いるときは、そのコンテナは応答バッファ内に残っている総てのコンテントを即座にクライアントにフラッシュしなければならない。以下のイベントたちがそのサーブレットが要求を満足させた、及び応答のオブジェクトをクローズしなければならないことを示している：

- そのサーブレットの service メソッドの終了。
- その応答の setContentLength メソッドで指定されたコンテント量がゼロ以上でその応答に書き込まれている。
- SendError メソッドが呼ばれた。
- SendRedirect メソッドが呼ばれた。
- AsyncContext の complete メソッドが呼ばれた。

5.6 応答オブジェクトの生存期間（Lifetime of the Response Object）

各応答オブジェクトは、対応した要求オブジェクトがその要素のために非同期処理対応でないかぎり、サーブレットの service メソッドの適用範囲内、またはフィルタの doFilter メソッドの適用範囲内でのみ有効である。もし対応した要求で非同期処理が開始しているときは、その要求オブジェクトは AsyncContext 上の complete メソッドが呼ばれるまでは有効な状態に留まる。コンテナたちは一般的に応答オブジェクト生成のための性能オーバヘッドを回避するために応答オブジェクトをリサイクルしている。開発者は対応した要求上の startAsync が呼ばれていない応答オブジェクトへの参照

を上記の適用範囲外で維持すると不確定な振る舞いをもたらすかもしれないことに注意しなければならない。

第6章　　フィルタリング（Filtering）

フィルタはリソースへの要求とリソースからの応答の双方にあるヘッダ情報とペイロードの動作中での変換を可能とするJava部品である。本章は動的及び静的コンテントをフィルタリングするための軽量のフレームワークを提供しているJava Servlet v.3.0 APIクラスたちとメソッドたちを記述している。本章は如何にあるウェブ・アプリケーションにフィルタたちが設定されるか、そしてそれらの実装の為の規約と意味(semantics)を記述している。サーブレット・フィルタのAPIドキュメンテーションはオンラインで提供されている。フィルタたちの設定のシンタックスは第14章の「配備記述子(Deployment Descriptor)」のなかの配備記述子スキーマで与えられている。読者は本章を読む際はこれらのリソースを参照物として使うべきである。

6.1　フィルタとは何か？（What is a filter?）

フィルタとは再使用可能なコードであってHTTP要求、応答、及びヘッダ情報を変換できるものである。フィルタは一般にはサーブレットがしているような応答をつくるあるいは要求に応答するということはしないで、むしろリソースからの要求を修正または適合化、及びリソースからの応答の修正または適合化を行う。フィルタは動的または静的なコンテント上で機能できる。本章の目的のために、動的と静的なコンテントはウェブ・リソースとして引用されている。フィルタ使用が必要となる開発者が利用できる機能のタイプは以下のようである:

- ある要求が呼び出される前にリソースにアクセスする。
- それが呼び出される前にリソースへの要求を処理する。
- 要求オブジェクトのカスタム化されたバージョンとしてその要求をラップ（包み込む）ことにより要求ヘッダたちとデータを修正する。
- 応答オブジェクトのカスタム化されたバージョンを提供することで応答ヘッダたちと応答データを修正する。
- それの呼び出し後のリソースの呼び出しの傍受。
- あるサーブレット上、サーブレットたちのグループ上、あるいは静的なコンテントの、指定可能な順序でゼロ、1、あるいはそれ以上のフィルタによるアクション。

6.1.1　フィルタリング部品の例（Examples of Filtering Components）

- 認証フィルタ
- ロッギングと監査のフィルタ
- イメージ変換フィルタ
- データ圧縮フィルタ
- 暗号化フィルタ
- トークン化フィルタ

- リソースのアクセス・イベントのトリガをかけるフィルタ
- XML コンテントを変換する XSL/T フィルタ
- MIME-type チェイン・フィルタ
- キャッシングのためのフィルタ

6.2 メインのコンセプト（Main Concepts）

このフィルタリングのモデルのメインのコンセプトが本節で記されている。

アプリケーションの開発者は javax.servlet.Filter インターフェイスを実装し引数を持たないパブリックなコンストラクタを用意することでフィルタを生成する。このクラスは、ウェブ・アプリケーションを構成する静的コンテントとサーブレットたちとともにウェブ・アーカイブ(Web Archive)にパッケージ化される。フィルタは配備記述子の<filter>要素を使うことで宣言される。あるフィルタあるいはフィルタたちの収集物は配備記述子の<filter-mapping> 要素を定義することで設定できる。これはそのサーブレットの論理名によってフィルタたちをある特定のサーブレットにマッピングする、あるいはあるフィルタをある URL パタンにマッピングすることでサーブレットたちのグループと静的なコンテントのリソースにマッピングすることでなされる。

6.2.1 フィルタの生存期間（Filter Lifecycle）

そのウェブ・アプリケーションの配備後、そしてある要求でコンテナがウェブのリソースにアクセスする前に、そのコンテナは以下に記したようなそのウェブ・リソースに適用されねばならないフィルタたちのリストの場所を特定しなければならない。そのコンテナは確実にそのリストにある各フィルタの為のしかるべきクラスのフィルタをインスタンス化し、その init(FilterConfig config)メソッドを呼ばねばならない。このフィルタはそれが適正に機能できないことを示すための例外をスローできる。その例外が UnavailableException のタイプであるときは、そのコンテナはその例外の isPermanent 属性を調べ、しばらくした後でそのフィルタを再試行することを選択できる。

配備記述子のなかの<filter>宣言あたり唯ひとつのインスタンスはそのコンテナの JVM あたりでインスタンス化される。そのコンテナはそのフィルタに、そのフィルタの配備記述子のなかで宣言された config、ウェブ・アプリケーションの為の ServletConfig への参照、及び初期化パラメタたちのセットを用意する。

そのコンテナが着信要求を受けたときは、そのコンテナはそのリストの最初のフィルタ・インスタンスをえらび、その doFilter メソッドを呼び、ServletRequest と ServletResponse、及びそれが使う FilterChain オブジェクトへの参照を渡す。

あるフィルタの doFilter メソッドは一般的には以下のものに従う、あるいは以下のパタンの何らかのサブセットにしたがって実装される:

1. このメソッドはその要求のヘッダたちを調べる。
2. このメソッドは要求ヘッダたちまたはデータを加工するために ServletRequest または HttpSerletRequest のカスタム化された実装でその要求オブジェクトをラップできる。
3. このメソッドは応答ヘッダたちまたはデータを加工するために ServletResponse または HttpSerletResponse のカスタム化された実装で、その doFilter メソッドに渡されたそのオブジェクトをラップできる。
4. このメソッドはそのフィルタ・チェインのなかの次のエンティティを起動できる。次のエンティティは別のフィルタになり、あるいはもし起動しているそのフィルタがこのチェインのために配備記述子のなかで設定されている最後のフィルタのときは、次のエンティティはターゲットのウェブ・リソースになる。次のエンティティへの起動は、その FilterChain オブジェクト上の doFilter メソッドを呼び出し、それが呼ばれたときの、あるいはそれが作ったかもしれないラップされたバージョンの要求と応答の渡しで引き起こされる。このフィルタ・チェインの doFilter の実装は、そのコンテナが用意するものであるが、そのフィルタ・チェインのなかの次のエンティティの場所を特定し、その doFilter メソッドを起動し、しかるべき要求と応答のオブジェクトを渡さねばならない。代替的には、このフィルタ・チェインは次のエンティティを起動させるために呼び出しをしないことでその要求をブロックし、そのフィルタに応答オブジェクトを埋める責任を持たせることができる。
5. フィルタ・チェインのなかの次のフィルタの起動をしたあとで、そのフィルタは応答ヘッダたちを調べることができる。
6. 代りに、そのフィルタは処理中のエラーを示すために例外をスローしたかもしれない。もしそのフィルタがその doFilter 処理中に UnavailableException をスローするときは、そのコンテナはそのフィルタ・チェインに基づいて処理を継続しようとしてはならない。そのコンテナはその例外が恒久的とマークされていないときは、あとでそのチェイン全体を再試行することを選択できる。
7. そのチェインのなかの最後のフィルタが起動された後は、次にアクセスされるエンティティはターゲットのサーブレットまたはそのチェインの最後にあるリソースである。
8. そのコンテナによってあるフィルタがサービスから外され得る前に、そのコンテナはそのフィルタが何らかのリソースを解放し他のクリーンアップ操作ができるようにそのフィルタ上の destroy メソッドを最初に呼び出さねばならない。

6.2.2 要求と応答のラッピング（Wrapping Requests and Responses）

フィルタリングの概念の中心になっているのはフィルタリングのタスクを実施するための振る舞いをオーバライド（凌越）できる為に要求と応答をラップするというコンセプトである。このモデルでは、開発者は要求と応答に関しオブジェクト上の既存のメソッドたちをオーバライドできるだけでなく、あるフィルタまたはそのチェインをたどったターゲットのウェブ・リソースへの特定のフィルタリングのタスクに適した新しい API を提供できる。例えば、その開発者はその応答オブジェクトを拡張してより高度な出力オブジェクトを持たせ、DOM オブジェクトが可能な API などがそのクライアントに書き戻されることを望むかもしれない。

このスタイルのフィルタに対応するためにそのコンテナは以下のような要求を満たさねばならない。あるフィルタがそのコンテナのフィルタ・チェイン実装上の doFilter メソッドを起動するときは、そのコンテナは、それがそのフィルタ・チェインの次のエンティティに、あるいはもしそのフィルタがその

チェインの最後だったらターゲットのウェブ・リソースに渡す要求と応答のオブジェクトが、呼び出しフィルタによって doFilter メソッドに渡されたと同じオブジェクトであることを確保しなければならない。

ラッパ・オブジェクト・アイデンティティの同じ要求が、呼び出し元が要求または応答オブジェクトをラップするときは、RequestDispatcher.forward または RequestDispatcher.include へのサーブレットまたはフィルタからの呼び出しにも適用される。この場合、呼び出されたサーブレットから見た要求と応答のオブジェクトは、呼び出しもとのサーブレットまたはフィルタによって渡されたと同じラッパ・オブジェクトでなければならない。

6.2.3 フィルタ環境（Filter Environment）

初期化パラメタたちのセットが配備記述子のなかの<initparams>要素を使ってあるフィルタに結び付けられることができる。これらのパラメタの名前と値は実行時にそのフィルタの FilterConfig オブジェクト上の getInitParameter 及び getInitParameterNames のメソッドにより取得できる。加えて、FilterConfig はリソースのローディングのため、ロッギング機能のため、及び ServletContext の属性リストのなかの状態のストアのために、そのウェブ・アプリケーションの ServletContext にアクセスできる。フィルタとそのフィルタ・チェインの最後にあるターゲットのサーブレットまたはリソースは同じ起動スレッド内で実行しなければならない。

6.2.4 ウェブ・アプリケーションのなかでのフィルタの設定（Configuration of Filters in a Web Application）

フィルタは、この 8.1.2 章“@WebFilter”で定義されているように@WebFilter アノテーションを介して、あるいは<filter>要素を使って配備記述子のなかで、のいずれかで定義される。この要素のなかでは、プログラマは以下のように宣言する:

- filter-name:そのフィルタをサーブレットまたは URL にマップするのに使用する
- filter-class:コンテナによってそのフィルタ・タイプを特定するのに使われる
- init-params:フィルタの為の初期化パラメタたち

オプション的には、そのプログラマはツール操作のためにアイコン、テキストによる記述、及び表示名を指定できる。コンテナは配備記述子のなかのフィルタ宣言あたりそのフィルタを定めた Java クラスのまさしくひとつのインスタンスをインスタンス化しなければならない。

それゆえ、その開発者が同じフィルタクラスに対し 2 つのフィルタ宣言をした場合は、同じフィルタ・クラスの 2 つのインスタンスがそのコンテナによってインスタンス化されることになる。

以下はフィルタ宣言の例である:

```
<filter-mapping>
      <filter-name>Image Filter</filter-name>
      <servlet-name>ImageServlet</servlet-name>
</filter-mapping>
```

一旦あるフィルタが配備記述子のなかで宣言されたら、アセンブラはそのフィルタが適用されるそのウェブ・アプリケーションのサーブレットたちと静的なリソースたちを定義する為に<filter-mapping>要素を使う。フィルタたちは<servlet-name>要素を使ってあるサーブレットに関連付けられ得る。例えば、以下のコード例では Image Filter というフィルタを ImageServlet servlet にマップしている:

```
<filter-mapping>
      <filter-name>Image Filter</filter-name>
      <servlet-name>ImageServlet</servlet-name>
</filter-mapping>
```

フィルタたちはフィルタ・マッピングの<urlpattern>スタイルを使ってサーブレットたちと静的コンテントのグループに関連付けされ得る。

```
<filter-mapping>
      <filter-name>Logging Filter</filter-name>
      <url-pattern>/*</url-pattern>
</filter-mapping>
```

ここでは、各要求 URL が'/*'なる URL パタンにマッチするので、そのウェブ・アプリケーションのなかの総てのサーブレットと静的コンテント・ページたちにログイン用の Logging Filter が適用される。

<url-pattern>スタイルを使った<filter-mapping>要素を処理するときは、そのコンテナは第 12 章の「サーブレットへの要求のマッピング」で定められたパス・マッピング規則を使ってその<url-pattern>が要求 URI にマッチするかどうかを判断しなければならない。

特定の要求 URI の為に適用されるフィルタのチェインを組み立てる際にそのコンテナが使う順序は以下のとおりである:

1. 先ず、配備記述子のなかで出てくるこれらの要素たちの順と同じ順番で<url-pattern>にマッチするフィルタ・マッピングたち
2. 次に、配備記述子のなかで出てくるこれらの要素たちの順と同じ順番で<servlet-name>にマッチするフィルタ・マッピングたち

あるフィルタ・マッピングが<servlet-name>と<url-pattern>の双方を含んでいるときは、そのコンテナはそのフィルタ・マッピングを複数のフィルタ・マッピングたち(各<servletname>と<url-pattern>にたいしひとつ)に拡張し、<servlet-name>と<urlpattern>要素たちの順番を維持させねばならない。例えば、以下のようなフィルタ・マッピングは:

```
<filter-mapping>
      <filter-name>Multipe Mappings Filter</filter-name>
      <url-pattern>/foo/*</url-pattern>
      <servlet-name>Servlet1</servlet-name>
      <servlet-name>Servlet2</servlet-name>
      <url-pattern>/bar/*</url-pattern>
</filter-mapping>
```

は以下のものと等価である:

```
<filter-mapping>
      <filter-name>Multipe Mappings Filter</filter-name>
      <url-pattern>/foo/*</url-pattern>
</filter-mapping>
<filter-mapping>
      <filter-name>Multipe Mappings Filter</filter-name>
```

```
      <servlet-name>Servlet1</servlet-name>
</filter-mapping>
<filter-mapping>
      <filter-name>Multipe Mappings Filter</filter-name>
      <servlet-name>Servlet2</servlet-name>
</filter-mapping>
<filter-mapping>
      <filter-name>Multipe Mappings Filter</filter-name>
      <url-pattern>/bar/*</url-pattern>
</filter-mapping>
```

フィルタ・チェインの順序に関するこの要求は、ある到来要求を受信したときにそのコンテナが以下のようにその要求を処理することを意味する:

- 12.2 章の「マッピングの為の仕様」の規則にしたがってターゲットのウェブ・リソースを特定する。
- サーブレット名でマッチしたフィルタが存在し、そのウェブ・リソースが<servletname>を持っているときは、そのコンテナは配備記述子で宣言された順でマッチするフィルタたちのチェインを組み立てる。そのフィルタの最後のフィルタが最後の<servlet-name>マッチング・フィルタに対応し、それがターゲットのウェブ・リソースを起動するフィルタになる。
- もし<url-pattern>マッチングを使うフィルタたちが存在し、その<url-pattern>が第 12.2 章の「マッピングの為の仕様」の規則に従って要求 URI とマッチするときは、そのコンテナは配備記述子で宣言された順で<url-pattern>マッチするフィルタたちのチェインを組み立てる。このチェインの最後のフィルタがこの要求 URI のために配備記述子のなかにある最後の<url-pattern>マッチするフィルタである。そのフィルタの最後のフィルタが最後の<servlet-name>マッチング・フィルタを起動する、あるいは何も無ければターゲットのウェブ・リソースを起動するフィルタになる。

高性能のウェブ・コンテナがフィルタ・チェインたちをキャッシュして、これらのコンテナが要求あたりベースで計算しなくても良くすることが期待される。

6.2.5 フィルタたちと RequestDispatcher（Filters and the RequestDispatcher）

Java Servlet 仕様書の第 2.4 版以降新しくなっていることは、要求ディスパッチャの forward()と include()呼び出しで起動されるようフィルタたちを設定できるようになったことである。

配備記述子の新しい<dispatcher>要素を使うことで、開発者はあるフィルタ・マッピングに対し以下のときにそのフィルタが要求に適用されるかを示すことが出来る:

1. その要求がクライアントから直接来ている。これは REQUEST という値を持った<dispatcher>要素によって、あるいはなにも<dispatcher>が存在しないことによって示される。
2. その要求が forward()呼び出しを使って<url-pattern>または<servlet-name>にマッチするウェブ部品を代表する要求ディスパッチャのもとで処理されている。このことは FORWARD という値を持った<dispatcher>要素によって示される。

3. その要求が include()呼び出しを使って<url-pattern>または<servlet-name>にマッチするウェブ部品を代表する要求ディスパッチャのもとで処理されている。このことは INCLUDE という値を持った<dispatcher>要素によって示される。
4. その要求が 10.9 章の「エラー処理」で規定されたエラー・ページのメカニズムによって、<url-pattern>がマッチするエラーのあるリソースに対し処理中である。これは ERROR という値を持った<dispatcher>要素によって示される。
5. その要求が第 2.3.3.3 の「非同期処理」で規定された非同期コンテキスト・ディスパッチのメカニズムによって dispatch 呼び出しであるウェブ部品に対し使われている。これは ASYNC という値を持った<dispatcher>要素によって示される。
6. あるいは上記の 1 から 5 までの何らかの組み合せ。

例えば

```
<filter-mapping>
      <filter-name>Logging Filter</filter-name>
      <url-pattern>/products/*</url-pattern>
</filter-mapping>
```

では Logging Filter は/products/...で始まるクライアント要求で起動されるが、要求ディスパッチャが/products/....で始まっているパスを持っている要求ディスパッチャ呼び出しのもとでは起動されない。

この Logging Filter はその要求の初期のディスパッチと再開要求の双方で起動される。以下のコードでは:

```
<filter-mapping>
      <filter-name>Logging Filter</filter-name>
      <servlet-name>ProductServlet</servlet-name>
      <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

ではこの Logging Filter は ProductServlet へのクライアント要求によってもまた要求ディスパッチャの ProductServlet への forward()呼び出しでも起動されないが、その要求ディスパッチャが ProductServlet で始まる名前をもっている要求ディスパッチャの include()呼び出しで起動される。

以下のコードでは:

```
<filter-mapping>
      <filter-name>Logging Filter</filter-name>
      <url-pattern>/products/*</url-pattern>
      <dispatcher>FORWARD</dispatcher>
      <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Logging Filter は、/products/...で始まるクライアント要求により、そしてその要求ディスパッチャが/products/....で始まるパスを持っている要求ディスパッチャ forward()呼び出しにより、起動される。

最後に、以下に示すコードは特別な'*'というサーブレット名を使っている:

```
<filter-mapping>
      <filter-name>All Dispatch Filter</filter-name>
      <servlet-name>*</servlet-name>
      <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

このコードでは、名前またはパスで得られた総ての要求ディスパッチャにたいする要求ディスパッチャの forward()呼び出しで All Dispatch Filter が起動される。

第7章　　セッション（Sessions）

ハイパー・テキスト転送プロトコル（HTTP）は設計上は状態なし（ステートレス）のプロトコルである。効果的なウェブ・アプリケーションを構築するには、ある特定のクライアントからの要求たちが互いに関連付けられていることが必須である。セッション追跡のための多くの戦略が時とともに発展してきたが、そのプログラマが直接使うには総てが難しいか問題を生じやすいものである。本仕様書はシンプルな HttpSession インターフェイスを定義して、これによりアプリケーションの開発者がどれかのひとつのアプローチの微妙な差に関与することなくユーザのセッションを追跡する為に、サーブレット・コンテナがどの幾つかのアプローチも使うことができるようにしている。

7.1　セッション追跡のメカニズム（Session Tracking Mechanisms）

以下の章で、あるユーザのセッションを追跡するアプローチを記述する。

7.1.1　クッキー（Cookies）

HTTP クッキーを介したセッション追跡が最も使われているセッション追跡のメカニズムであり、総てのサーブレット・コンテナたちがサポートすることが要求されている。

コンテナはクライアントにクッキーを送信する。そのクライアントはそのサーバに返すその後の要求の各々でそのクッキーを返す。このセッション追跡クッキーの標準名は JSESSIONID で、これは総ての 3.0 版対応のコンテナがサポートしなければならない。コンテナたちはコンテナ固有の設定によってこのセッション追跡クッキーの名前をカスタマイズしても良い。

総てのサーブレット・コンテナたちはそのセッション追跡クッキーを HttpOnly としてマークするかどうかを設定できるようにしなければ**ならない**。この確立された設定は、それに対し固有の設定が確立されていない総てのコンテキストたちに適用されねばならない（更なる詳細は javadoc の SessionCookieConfig を見られたい）。

あるウェブ・アプリケーションがそのセッション追跡クッキーにカスタムの名前を設定するときは、そのセッション ID がその URL にエンコードされているときは、同じカスタム化された名前がその URL パラメタの名前として使われることになる（URL 書き換えが可能になっているときは）。

7.1.2　SSL セッション（SSL Sessions）

HTTPS プロトコルで使われている暗号化技術であるセキュア・ソケット層には、あるクライアントからの複数の要求があるセッションの部分であることを明確に特定するメカニズムが組み込まれている。サーブレット・コンテナはあるセッションを判断するのにこのデータを容易に使うことができる。

7.1.3 URL 書き換え（URL Rewriting）

URL 書き換えはセッション追跡の最小公倍数である。あるクライアントがクッキーを受け付けないときは、URL 書き換えはセッション追跡の為のベースとしてそのサーバが使うことになろう。URL 書き換えではセッション ID なるデータが URL パスに追加され、そのデータがその要求をあるセッションに結び付けるためにそのコンテナによって解釈される。このセッション ID は URL 文字列のパス・パラメタとしてエンコードされねばならない。以下はエンコードされたパス情報を含む URL の例である。
http://www.myserver.com/catalog/index.html;jsessionid=1234

URL 書き換えではセッション識別子がログ、ブックマーク、リフェラのヘッダ、キャッシュされた HTML、及び URL バーで見えてしまう。URL 書き換えはクッキーと SSL セッションがサポートされており、また適正である場合のセッション追跡のメカニズムとしては使うべきではない。

7.1.4 セッションの完全性（Session Integrity）

ウェブ・コンテナたちはクッキー使用をサポートしていないクライアントたちからの HTTP 要求にサービスしつつ HTTP セッションをサポートできなければならない。この要求を満足させるために、ウェブのコンテナたちは一般に URL 書き換えのメカニズムをサポートしている。

7.2 セッションの生成（Creating a Session）

それが予期的(prospective)セッションであってまだ確立されていないときに限りあるセッションは新(new)だと考えられる。HTTP は要求-応答ベースのプロトコルであるので、HTTP セッションはそのクライアントがそれに加わる(join)するまでは new だと考えられる。セッション追跡の情報がそのサーバに戻されたときにあるクライアントはあるセッションに加わったことになり、セッションが確立されたことを示す。そのクライアントがあるセッションに加わるまでは、そのクライアントからの次の要求があるセッションの要素だとして認識されるとは仮定できない。

セッションは以下のいずれかが真であるときに new であると考えられる：

- そのクライアントがそのセッションのことを未だ知らない
- そのクライアントがセッションに加わらないことを選択している

これらの条件が、そのサーブレット・コンテナがそれによってある要求を以前の要求に結び付けるメカニズムを持たない状況を定義している。サーブレットの開発者は、あるクライアントがあるセッションに加わっていない、加わることができない、あるいは加わろうとしない状況を処理できるよう自分のアプリケーションを設計しなければならない。

7.3 セッションの適用範囲（Session Scope）

HttpSession オブジェクトはそのアプリケーション（またはサーブレット・コンテキスト）のレベルでの適用範囲でなければならない。セッション確立のために使われるクッキーのような基になっているメカニズムは、別のコンテキストでも同じである可能性があるが、参照されるオブジェクトは、そのオブジェクトの属性も含めて、そのコンテナによってコンテキストたち間で決して共有されてはならない。

この要求を事例で示すと:もしあるサーブレットが別のウェブ・アプリケーション内のあるサーブレットを RequestDispatcher を使って呼び出すときは、その呼ばれたサーブレットのためにつくられていて可視であるセッションは、呼び出しもとのサーブレットにとって可視なものとは違っていなければならない。

加えて、そのセッションがつくられた時点でそれに結び付けられたコンテキストが直接あるいは要求ディスパッチのターゲットしてアクセスされているかどうかにかかわらず、あるコンテキストのセッションたちは、そのコンテキストへの要求により再開可能でなければならない。

7.4 セッションへの属性のバインド（Binding Attributes into a Session）

サーブレットは HttpSession 実装物に対し名前でオブジェクト属性をバインドできる。セッションにバインドされたオブジェクトは同じ ServletContext に属しまた同じセッションの要素であると特定された要求を処理する他のサーブレットも利用できる。

一部のオブジェクトたちはそれがあるセッションに置かれた、あるいは外されたときに通知を必要とするかもしれない。この情報はそのオブジェクトに HttpSessionBindingListener インターフェイスを実装させることで取得できるようになる。このインターフェイスはあるセッションにあるオブジェクトがバインドされた、あるいは外されたことをシグナルする以下のメソッドたちを定義している:

- valueBound
- valueUnbound

valueBound メソッドは HttpSesion インターフェイスの getAttribute メソッドを介してそのオブジェクトが取得できるようになる前に呼ばれねばならない。ValueUnbound メソッドは HttpSesion インターフェイスの getAttribute メソッドを介してそのオブジェクトを取得する必要がもはや無くなったときに呼ばれねばならない。

7.5 セッションのタイムアウト（Session Timeouts）

HTTP プロトコルにおいてはクライアントがもはやアクティブでないときの明示的な終了信号が存在しない。このことは、あるクライアントがもはやアクティブでないことを示すために使うことができる唯一のメカニズムはタイムアウト期間であることを意味する。

セッションのためのデフォルトのタイムアウト期間はそのサーブレット・コンテナによって定められ、HttpSession インターフェイスの getMaxInactiveInterval メソッドによって取得できる。このタイムアウトは HttpSession インターフェイスの setMaxInactiveInterval メソッドを使うことで開発者が変更できる。これらのメソッドで使われているタイムアウト期間は秒で定義されている。定義によって、あるセッションのタイムアウト期間が-1 にセットされているときは、セッションはタイムアウトを決して起こさない。

セッションの無効化はそのセッションを使っている総てのサーブレットが service メソッドを抜けたときに効果を持つ。一旦このセッションの無効化が始まったら、新しい要求はそのセッションを見れるようにしてはならない。

7.6 最後にアクセスした時間（Last Accessed Times）

HttpSession インターフェイスの getLastAccessedTime は、現在の要求以前にこのセッションがアクセスされた最後の時間をサーブレットが判断するのに使うことができる。このセッションの要素であるようなある要求がそのサーブレット・コンテナによって最初に処理されたときにこのセッションはアクセスされたと考えられる。

7.7 セッションの重要な意味（Important Session Semantics）

7.7.1 スレッドの問題（Threading Issues）

要求スレッドたちを実行している複数のサーブレットが同じ時間に同じセッション・オブジェクトへのアクティブなアクセスを持っている可能性がある。コンテナはセッション属性たちを代表している内部データ構造の操作が、スレッド安全なやり方で実行されるようにしなければならない。開発者は属性オブジェクトたち自身へのスレッド安全なアクセスをする責任を持つ。これにより HttpSession オブジェクト内部の属性の集積を同時アクセスから保護し、あるアプリケーションがその集積物を壊す機会を無くすことができる。

7.7.2 分散環境（Distributed Environments）

分散可能とマークされたアプリケーション内では、あるセッションの要素である総ての要求はある時点でひとつの JVM によって処理されねばならない。そのコンテナは setAttribute または putValue メソッドを使って HttpSession クラスのインスタンスに置かれた総てのオブジェクトを適正に処理できなければならない。これらの条件を満たすためには以下のような制約が課される：

- そのコンテナは Serializable インターフェイスを実装したオブジェクトを受け入れねばならない
- そのコンテナは法人 JavaBeans 部品とトランザクションへの参照のような、HttpSession 内に指定された他のオブジェクトたちの蓄積をサポートすることを選ぶことができる。
- セッションの別のコンテナへの移行（migration）はコンテナ固有の機能で処理される。

この分散サーブレット・コンテナは蓄積しているセッションの移行に必要なメカニズムに対応していないオブジェクトたちに対しては IllegalArgumentException をスローしなければならない。分散されたサーブレット・コンテナは Serializable を実装したオブジェクトたちを移行させるに必要なメカニズムをサポートしなければならない。

これらの制約は、開発者が非分散コンテナで遭遇するもの以上の更なる同時実行問題がないことが保障されることを意味する。そのコンテナのプロバイダはあるセッション・オブジェクトとその中身を分散システムの何らかのアクティブなノードからそのシステムの別のノードに移すことができるようにすることで、拡張性と負荷バランスとフェイルオーバのようなサービス品質を確保できる。

分散コンテナたちがサービス品質機能を提供するためにセッションを維持存続あるいは移行するときは、分散コンテナたちは HttpSession とその属性たちを直列化するためのネーティブな JVM 直列化メカニズムを使うことに制約されない。開発者たちはもしそれらを実装するときはセッション属性上の readObject と writeObject をコンテナたちが呼ぶということは保障されないが、これらの属性の Serializable 句が保持されることは保障される。

コンテナたちはあるセッションの移行中に HttpSessionActivationListener を実装するセッション属性たちに通知をしなければならない。コンテナたちはあるセッションの直列化に先立ちリスナたちに受動化の通知を、またあるセッションの非直列化のあとでの活性化の通知をしなければならない。

分散アプリケーションを書いているアプリケーションの開発者たちは、そのコンテナがひとつ以上の JVM で走っているかもしれないので、その開発者はあるアプリケーションの状態をストアするのに静的な変数に依存できないことに注意しなければならない。彼らは法人ビーンまたはデータベースを使ってそのような状態をストアすべきである。

7.7.3 クライアントにとっての意味（Client Semantics）

クッキーあるいは SSL 認証は通常 web ブラウザのプロセスで処理され、そのブラウザの特定のウインドウとは関連づけられてはいないので、クライアントアプリケーションからサーブレットコンテナへの総てのウインドウからの要求は、同じセッションの部分である可能性がある。ポータビリティを最大

とする為に、開発者はクライアントの総てのウインドウが同じセッションに参加しているものと仮定すべきである。

第8章　　アノテーションとプラグ可能性（Annotations and pluggability）

本章ではサーブレット3.0仕様で定義されたアノテーションと、あるウェブ・アプリケーション内で使うためのフレームワークたちやライブラリたちのプラグイン可能化のためのその強化に関して記述する。

8.1　アノテーションとプラグ可能性（Annotations and pluggability）

ウェブ・アプリケーションにおいては、アノテーションを使ったクラスたちは、それらが WEB-INF/classes ディレクトリに置かれているとき、あるいはそれらがそのアプリケーションの WEB-INF/lib 内に置かれた jar ファイルにパッケージされているときのみ、それらのアノテーションが処理される。

そのウェブ・アプリケーションの配備記述子は web-app 要素上に新しい“metadata-complete”属性を含んでいる。この“metadata-complete”属性はそのウェブ記述子が完了しているか、あるいはその jar ファイルのクラス・ファイルたちは配備時にアノテーションたちとウェブ・フラグメントたちがあるかどうかを検査されるべきかどうかを定義している。もし“metadata-complete”が"true"にセットされていると、その配備ツールはそのアプリケーションとウェブ・フラグメントたちのクラス・ファイルたちのなかの何らかのサーブレットのアノテーションを無視しなければ**ならない**。もし metadata-complete 属性が指定されていない、あるいは"false"にセットされていれば、その配備ツールはそのアプリケーションのクラスファイルたちがアノテーションを含んでいるかを調べ、ウェブ・フラグメントたちをスキャンしなければならない。

以下は Servlet 3.0 対応のウェブ・コンテナがサポートしなければ**ならない**アノテーションたちである。

8.1.1　@WebServlet

このアノテーションはあるウェブ・アプリケーションの Servlet 部品を定義するのに使われる。このアノテーションはあるクラス上で指定され、宣言されている Servlet に関するメタデータを含む。このアノテーション上では urlPatterns または value 属性がなければ**ならない**。他の総ての属性はオプショナルで、デフォルトの設定になっている（より詳細は javadocs を見られたい）。このアノテーション上の属性がその URL パタンのみであるときに value を使用し、他の属性も使われているときは urlPatterns 属性を使うことが推奨される。おなしアノテーション上で value と urlPatterns 属性の双方を使わせるのは違反である。もし指定されていないときの Servlet のデフォルトの名前は完全修飾クラス名である。アノテートされたサーブレットは少なくともひとつの配備される URL パタンを指定しなければ**ならない**。もし同じサーブレット・クラスが異なった名前で配備記述子のなかで宣言されているときは、そのサーブレットの新しいインスタンスがインスタンス化されねば**ならない**。4.4.1 章の「サーブレットのプログラム的な追加と設定」で定義されたプログラム的な API を介して同じサーブ

レット・クラスが ServletContext に追加されているときは、@WebServlet を介して宣言される値たちは無視されねばならず、指定された名前を持ったサーブレットの新しいインスタンスがつくられねばならない。

@WebServlet クラスでアノテートされたクラスは、javax.servlet.http.HttpServlet クラスを継承しなければならない。

以下は如何にこのアノテーションが使われるかの例である。

コード例 8-1　@WebServlet アノテーション例

```
@WebServlet(”/foo”)
public class CalculatorServlet extends HttpServlet{
      //...
}
```

以下は如何にこのアノテーションが更に多くの属性を指定するのに使われるかの例である。

コード例 8-2　他のアノテーション属性を指定した@WebServlet アノテーション例

```
@WebServlet(name=”MyServlet”, urlPatterns={"/foo", "/bar"})
public class SampleUsingAnnotationAttributes extends HttpServlet{
      public void doGet(HttpServletRequest req, HttpServletResponse
res) {

      }
}
```

8.1.2　@WebFilter

このアノテーションはあるウェブ・アプリケーション内の Filter を定義する。このアノテーションはあるクラス上で指定され、宣言されているフィルタに関するメタデータを含む。特に指定されてないときのこの Filter のデフォルト名は完全修飾クラス名である。そのアノテーションの urlPatterns 属性、servletNames 属性、あるいは value 属性は指定されねばならない。総てのその他の属性はデフォルト値を持っていてオプショナルである（より詳細は javadocs を見られたい）。このアノテーション上の属性がその URL パタンのみであるときに value を使用し、他の属性も使われているときは urlPatterns 属性を使うことが推奨される。同じアノテーション上で value と urlPatterns の双方の属性が使われるのは違反である。

@WebFilter でアノテートされたクラスは javax.servlet.Filter を実装しなければならない。

以下は如何にこのアノテーションが使われるかの例である。

コード例 8-3@WebFilter アノテーション例

```
@WebFilter(“/foo”)
public class MyFilter implements Filter {
      public void doFilter(HttpServletRequest req,
HttpServletResponse res)
      {
      ...
```

```
        }
}
```

8.1.3 @WebInitParam

このアノテーションは Servlet または Filter に渡さねばならない init パラメタがあればそれを指定するのに使われる。これは WebServlet と WebFilter アノテーションの属性である。

8.1.4 @WebListener

WebListener アノテーションは特定のウェブ・アプリケーションのコンテキスト上でのいろんな操作のためのイベントを取得するためのリスナをアノテートするのに使われる。@WebListener でアノテートされたクラスは以下のインターフェイスたちのひとつを実装していなければならない:

- javax.servlet.ServletContextListener
- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener

事例:

```
@WebListener
public class MyListener implements ServletContextListener{
      public void contextInitialized(ServletContextEvent sce) {
      ServletContext sc = sce.getServletContext();
      sc.addServlet("myServlet", "Sample servlet",
      "foo.bar.MyServlet", null, -1);
      sc.addServletMapping("myServlet", new String[]
      { "/urlpattern/*" });
      }
}
```

8.1.5 @MultipartConfig

ある Servlet 上で指定されているときは、このアノテーションはそれが期待している要求が type mime/multipart であることを示している。対応するサーブレットの HttpServletRequest オブジェクトはいろんな mime 添付物上で繰り返す為に getParts 及び getPart メソッドで mime 添付を利用できるようにしていなければならない。

8.1.6 他のアノテーション/規約（Other annotations / conventions）

これらのアノテーションたちに加えて、第 15.5 章の「アノテーションとリソース注入」で定められた総てのアノテーションたちがこれらの新しいアノテーションたちのコンテキストのなかでも動作する。デフォルトでは総てのアプリケーションは welcome-file-list のリストのなかに index.htm(l)及び index.jsp を持つであろう。記述子はこれらのデフォルトの設定をオーバライド（凌越）するのに使われても良い。

WEB-INF/classes または WEB-INF/lib にあるいろんなフレームワーク jar/class たちから Listener たち、Servlet たちがロードされる順番は、アノテーションが使われているときは不定である。順序付けが重要な場合には、以下の web.xml のモジュール性及び web.xml 及び web-fragment.xml の順序付けの章を見られたい。順序は配備記述子のなかでのみ指定できる。

8.2 プラグ可能性（Pluggability）

8.2.1 web.xml のモジュール性（Modularity of web.xml）

上記で定義したアノテーションたちにより、web.xml の使用をオプショナルなものにしている。しかしながら、デフォルト値たちまたはアノテーションたちでセットされた値たちをオーバライドするために、配備記述子が使われる。これまでどおりに、web.xml 記述子のなかで metadata-complete 要素が true にセットされていれば、クラス・ファイルたちと jar にバンドルされたウェブ・フラグメントたちは処理されない。このことはそのアプリケーションの総てのメタデータが web.xml 記述子で指定されることを意味する。

開発者たちにとってより良いプラグイン化性とより少ない設定のために、この仕様の本バージョン(Servlet 3.0)では、我々はウェブ・モジュール配備記述子フラグメント（ウェブ・フラグメント）の概念を導入している。ウェブ・フラグメントというのはライブラリまたはフレームワーク jar の META-INF ディレクトリのなかで指定されまた含められ得る web.xml の一部または総てのことである。コンテナは以下に定めた規則に従いこの設定を拾い上げ使用する。

ウェブ・フラグメントはウェブ・アプリケーションの論理的な分割(partitioning)で、そのウェブ・アプリケーションのなかで使われているフレームワークが開発者たちに web.xml 内の情報を編集または追加することを要請することなく総ての加工物を定義できるようにするものである。これは web.xml 記述子が使っている要素の殆ど総てを含めることができる。しかしながらこの記述子のトップ・レベルの要素は web-fragment でなければ**ならず**、これに対応する記述子ファイルは呼ばれねば**ならない**。web-fragment.xml と web.xml 間では関連した要素たちの順序はまた異なっている。配備記述子のなかの web-fragments の対応するスキーマは第 14 章を見られたい。

もしあるフレームワークが jar ファイルとしてパッケージ化されており配備記述子の様式でメタデータ情報を持っておれば、その web-fragment.xml 記述子はその jar ファイルの META-INF/ディレクトリのなかになければならない。

もしあるフレームワークがその META-INF/web-fragment.xml があるウェブ・アプリケーションの web.xml を増強させる役を持たせたいときは、そのフレームワークはそのウェブ・アプリケーションの WEB-INF/lib ディレクトリのなかにバンドルされねばならない。そのフレームワークのどの他の形式のリソースたち（例えばクラス・ファイルたち）も、あるウェブ・アプリケーションにとって利用可能とするには、そのフレームワークがそのウェブ・アプリケーションのクラスローダ委譲チェインのどこかで見えることで十分である。言い換えると、あるウェブ・アプリケーションの WEB-INF/lib ディレクトリのなかにバンドルされた JAR ファイルたちのみ、しかしそのクラス・ローディングの委譲チェインのなかよりも上位でないものが、web-fragment.xml の為にスキャンされる必要がある。配備中はそのコンテナは上記で規定された場所をスキャンし、web-fragment.xml を発見し、それらを処理する責任を持つ。単一の web.xml のために現在存在している名前のユニーク性に関する要求はまた web.xml のユニオン及び総ての適用可能な web-fragment.xml ファイルたちにも適用される。

どのライブラリまたはフレームワークが含められ得るかの例を以下に示す。

```
<web-fragment>
      <servlet>
            <servlet-name>welcome</servlet-name>
            <servlet-class>
                  WelcomeServlet
            </servlet-class>
      </servlet>
      <listener>
            <listener-class>
                  RequestListener
            </listener-class>
      </listener>
</web-fragment>
```

上の web-fragment.xml はそのフレームワークの jar ファイルの META-INF/ディレクトリに含められることになる。web-fragment.xml とアノテーションたちからのどの設定が適用されるかの順番は定められていない。あるアプリケーションで順序付けが重要な側面である場合は、どうやって必要な順番を達成するかに関して如何に定められた規則を見られたい。

8.2.2 Web.xml と web-fragment.xml の順序付け（Ordering of web.xml and web-fragment.xml）

この仕様ではアプリケーション設定リソースたちが複数の設定ファイルたち（web.xml と web-fragment.xml）で構成され、そのアプリケーションのなかの幾つかの異なった場所から発見されロードされることを可能としている為、順序付けの問題は対処されねばならない。この章では如何に設定リソースの著者たちが彼らの加工品の順序付け要求を宣言できるかを規定している。

web-fragment.xml は javaee:java-identifierType というタイプのトップ・レベルの<name>要素を持っても良い。もし<name>要素が存在するときは、それは加工物たちの順序付けのためのものとして考えられねばならない（以下に示すように重複名前の例外が適用されない限り）。

アプリケーション設定リソースたちが自分たちの順序の好みを表現できるようにするために 2 つのケースが考えられねばならない。

1. 絶対順序付け：web.xml のなかの<absolute-ordering>要素。

    a. このケースでは、以下のケース 2 で処理されるであろう順序付けの好みは無視されねばならない。

    b. web.xml は web-fragments のどれかが absolute-ordering 要素のなかにリストされる前に処理されねば**ならない**。

    c. <absolute-ordering>の直接の子供であるどの<name>要素も、提示されようとされまいと、これらの名前がつけられたウェブ・フラグメントたちが処理されられねばならない絶対順序付けを示していると解釈されねば**ならない**。

    d. この<absolute-ordering>要素はゼロまたはひとつの<others /> 要素を持って良い。この要素に対する要求されているアクションは以下に記されている。もし<absolute-ordering>要素が<others/>要素を含んでいないときは、<name />要素内で特に指定されていない何らかのウェブ・フラグメントたちは無視されねば**ならない**。含まれていない jar たちはアノテートされたサーブレットのためにスキャンされない。しかしながらもし外されている jar からのあるサーブレット、フィルタ、あるいはリスナが web.xml あるいは排除されていない web-fragment.xml にリストされていたら、metadata-complete で外されていない限りそのアノテーションたちは適用される。TLD ファイルたちのなかに ServletContextListeners が発見されたら、外された jar たちはプログラム的な API たちを使ってフィルタたちとサーブレットたちを設定することはできない。そうしようと試みることは IllegalStateException をもたらす。もし発見された ServletContainerInitializer が外された jar からロードされた場合は、それは無視されよう。外された jar たちは ServletContainerInitializer によって処理されるクラスとしてスキャンされない。

    e. 重複した名前の例外：もし、<absoluteordering>の子供たちを調べているときに、同じ<name>要素を持った複数の子供たちに遭遇したときに、そのような発生の最初のもののみが検討されねばならない。

2. 相対順序付け：web-fragment.xml のなかの<ordering>要素。

    a. web-fragment.xml は<ordering>要素を持っても良い。もしそうなら、この要素はゼロまたはひとつの<before>要素とゼロまたはひとつの<after>要素を含まねばならない。これらの要素たちの意味は以下に説明されている。

    b. 重複した名前の例外：もし、ウェブ・フラグメントたちを調べているときに、同じ<name>要素を持った複数のメンバーたちに遭遇したときに、そのアプリケーションは問題解決のための

情報を含んだ情報としてのエラー・メッセージをログし、配備に失敗しなければならない。例えば、ユーザにとってこの問題を解決するためのひとつの手段は絶対順序付けを使うことで、このケースでは相対順序付けは無視される。

c. この省略されているがわかりやすい例を考えてみよう。MyFragment1、MyFragment2、及び MyFragment3 の 3 つのウェブ・フラグメントがそのアプリケーションの要素であり、また web.xml を含んでいる。

```
web-fragment.xml
<web-fragment>
      <name>MyFragment1</name>
      <ordering><after><name>MyFragment2</name></after></ordering>
      ...
</web-fragment>

web-fragment.xml
      <web-fragment>
      <name>MyFragment2</name>
      ..
</web-fragment>

web-fragment.xml
<web-fragment>
      <name>MyFragment3</name>
      <ordering><before><others/></before></ordering>
      ..
</web-fragment>

web.xml
<web-app>
      ...
</web-app>
```

この場合の処理の順序はつぎのようになる:
web.xml
MyFragment3
MyFragment2
MyFragment1

この例は全部ではないが以下の原則の一部を示している。

- <before>はそのドキュメントがネストされた<name>要素内で指定されたものとマッチする名前を持ったドキュメントの前に順序付けされねばならないことを意味する。
- <after>ははそのドキュメントがネストされた<name>要素内で指定されたものとマッチする名前を持ったドキュメントの後に順序付けされねばならないことを意味する。
- <before>または<after>要素のなかにゼロまたはワンタイムで含められる、あるいは<absolute-ordering>のなかに直接ゼロまたはワンタイムで含められることがある<others/>という特別の要素がある。この<others/>要素は以下のように処理されられねばならない。
  - もし<before>要素がネストされた<others/>要素を含んでいれば、そのドキュメントはストアされているドキュメントたちのリストの最初に移される。<before><others/>を要求している複数のドキュメントがあるときは、それらはソートされたドキュメントたちのリストの最初に置かれるが、そのようなドキュメントたちのグループ内の順序は指定されない。

- もし<after>要素がネストされた<others/>要素を含んでいれば、そのドキュメントはストアされているドキュメントたちのリストの最後に移される。<after><others/>を要求している複数のドキュメントがあるときは、それらはソートされたドキュメントたちのリストの最後に置かれるが、そのようなドキュメントたちのグループ内の順序は指定されない。
- <before>または<after>要素のなかに、もしひとつの<others/>要素が存在しているが、その親の要素内では<name>要素のみではないときは、その親のなかの他の要素たちはこの順序付けプロセスのなかにあると考えねばならない。
- もし<others/>要素が直接<absolute-ordering>要素のなかにある場合には、そのランタイムは<absolute-ordering>セクションのなかで明示的に名前がつけられていないウェブ・フラグメントたちを確実に処理順序のなかのそのポイントに含められるようにしなければならない。

- もしある web-fragment.xml ファイルが<ordering>を持っていないあるいは web.xml が <absolute-ordering>要素を持っていないときは、これらの加工物は何らの順序付け依存性を持っていないとみなされる。
- もしそのランタイムが循環参照を発見したら、情報的なメッセージがログされねばならず、そのアプリケーションは配備に失敗しなければならない。ここでも。そのユーザがとれるアクションは web.xml のなかで絶対順序付けを使うことである。
- これまでの例は web.xml が順序付けのセクションを含んでいるときのケースを示すように拡張できる:

```
web.xml
<web-app>
      <absolute-ordering>
            <name>MyFragment3</name>
            <name>MyFragment2</name>
      </absolute-ordering>
      ...
</web-app>
```

この例では、いろんな要素たちの順序は次のようになる:
web.xml
MyFragment3
MyFragment2

更に幾つかのシナリオ事例が以下に含まれている。これらの総てが相対順序付けを適用していて絶対順序付けを適用していない。

```
Document A:
<after>
      <others/>
      <name>
            C
      </name>
</after>

Document B
<before>
      <others/>
</before>

Document C:
<after>
      <others/>
```

```
</after>

Document D: 順序付けなし

Document E: 順序付けなし

Document F:
<before>
      <others/>
      <name>
            B
      </name>
</before>
```

結果としての順序は web.xml、F、B、D、E、C、A となる。

```
Document <no id>:
<after>
      <others/>
</after>
<before>
      <name>
            C
      </name>
</before>

Document B:
<before>
      <others/>
</before>

Document C: 順序付けなし

Document D:
<after>
      <others/>
</after>

Document E:
<before>
      <others/>
</before>

Document F: 順序付けなし
```

結果としての解析順序は以下のもののひとつになり得る:

- B, E, F, <no id>, C, D
- B, E, F, <no id>, D, C
- E, B, F, <no id>, C, D
- E, B, F, <no id>, D, C
- E, B, F, D, <no id>, C
- E, B, F, D, <no id>, D

```
Document A:
```

```
<after>
      <name>
            B
      </name>
</after>

Document B: 順序付けなし

Document C:
<before>
      <others/>
</before>

Document D: 順序付けなし
```

結果としての解析順序は C,B,D,A である。この解析順序はまた C、D、B、A または C、B、A、D にもなり得る。

8.2.3 web.xml、webfragment.xml、及びアノテーションたちからの記述子の組み立て（Assembling the descriptor from web.xml, webfragment.xml and annotations）

あるアプリケーションにとってリスナたち、サーブレットたち、フィルタたちが呼び出される順序が重要な場合は配備記述子が使われねばならない。また、必要なら、上記で定義された順序付けの要素が使用可能である。上で記したように、リスナたち、サーブレットたち、及びフィルタたちを定義するのにアノテーションを使うときは、それらが起動される順序は指定されない。以下はそのアプリケーションのための最終的な配備記述子を組み立てるときに適用される規則たちのセットである:

1. 関連性を持っている場合は、リスナたち、サーブレットたち、及びフィルタたちの順序は web-flagment.xml または web.xml で指定されねばならない。

2. 順序付けは記述子のなかで定められた順序及び web.xml のなかの absolute-ordering 要素で、あるいはもしあれば web-fragment.xml のなかの ordering 要素で定められた順序に基づくことになる。

    a. ある要求にマッチするフィルタたちは web.xml のなかで宣言されている順番でチェインされる。

    b. サーブレットたちは要求処理時に無精にあるいは配備時に積極的にのどちらかで初期化される。後者の場合は、これらはこれらの load-on-startup 要素で示された順序で初期化される。

    c. 本仕様のこのリリース以前は、コンテキスト・リスナたちはランダムな順番で起動されていた。サーブレット 3.0 版では、以下に示すようにリスナたちは web.xml のなかで宣言された順番で起動される:

i. javax.servlet.ServletContextListener の実装物たちはそれが宣言された順番で自分たちの contextInitialized メソッドが呼ばれ、その逆順で contextDestroyed メソッドが呼ばれる。
ii. javax.servlet.ServletRequestListener の装物たちはそれが宣言された順番で自分たちの requestInitialized メソッドが呼ばれ、その逆順で requestDestroyed メソッドが呼ばれる。
iii. javax.servlet.http.HttpSessionListener の実装物たちはそれが宣言された順番で自分たちの sessionInitialized メソッドが呼ばれ、その逆順で sessionDestroyed メソッドが呼ばれる。
iv. その他のリスナ・インターフェイスの呼び出し順序は指定されない。

3. もしあるサーブレットが web.xml のなかで導入された enabled 要素で利用不可(disabled)になっているときは、そのサーブレットはそのサーブレットのために指定されている url-pattern では使用できない。

4. そのウェブ・アプリケーションの web.xml が web.xml、web-fragment.xml、及びアノテーションたちの間での齟齬を解決するときの最高の優先度を持つ。

5. もし記述子たちのなかで metadata-complete が指定されていないとき、あるいは配備記述子のなかで false にセットされているときは、そのアプリケーションのための有効なメタデータはアノテーションたちと記述子たちのなかで出てくるメタデータをつかって引き出されねばならない。合わせるときの規則は以下に規定されている:

a. ウェブ・フラグメントたちの設定が、あたかもそれが同じ web.xml で指定されているかのごとくメインの web.xml のなかで指定されているものを強化するために使われる。

b. メインの web.xml のなかに追加されるウェブ・フラグメントたちの設定の順序は、第 8.2.2 章の「web.xml と web-fragment.xml の順序付け」で規定されたとおりである。

c. メインの web.xml のなかで true にセットされているときの metadata-complete 属性は、完了とみなされアノテーションたちとフラグメントたちのスキャンは配備時には生じない。absolute-ordering と ordering 要素は存在していても無視される。あるフラグメントで true にセットされているときは、metadata-complete 属性はその特定の jar のなかのアノテーションたちのみのスキャンに適用される。

d. metadata-complete がセットされていない限りウェブ・フラグメントたちはメインの web.xml に合体される。この合体は対応するフラグメント上のアノテーション処理の前に起きる。

e. ウェブ・フラグメントとたちで web.xml を強化している際に以下のものが設定の矛盾と考えられる。
i. 同じ<param-name>だが<param-value>が違っている複数の<init-param>要素たち
ii. 同じ<extension>だが異なった<mime-type>を持った複数の<mime-mapping>要素たち

f. 上記の設定矛盾は以下のように解決される:

i. メインの web.xml とあるウェブ・フラグメント間の設定の矛盾は web.xml の設定が優先することで解決される。
ii. 2 つのウェブ・フラグメント間での設定の矛盾は、矛盾している要素がメインの web.xml にないときは、エラーが起きる。情報的メッセージがログされ、そのアプリケーションの配備は失敗にしなければならない。

g. 上記の矛盾が解決された後で、以下のような更なる規則が適用される:
i. 任意の回数宣言されるかもしれない要素たちは結果としての web.xml のなかで付加的(additive:追加累積可能)である。例えば異なった<param-name>をもった<context-param>パラメタたちは付加的である。
ii. もしある要素がゼロの最小発生で 1 の最大発生を持ったある要素があるウェブ・フラグメントのなかにあって、メインの web.xml のなかに存在しないときは、メインの web.xml はそのウェブ・フラグメントからの設定を引き継ぐ。もしその要素がメインの web.xml とそのウェブ・フラグメントの双方に存在する場合は、メインの web.xml の設定が優先される。例えば、もしメインの web.xml とそのウェブ・フラグメントの双方が同じサーブレットを宣言し、ウェブ・フラグメントのサーブレット宣言が<load-on-startup>要素を指定していて、一方ではメインの web.xml のなかのそれがそうではないとき、合体された web.xml のなかではウェブ・フラグメントからの<load-on-startup>要素が使われる。
iii. もしある要素がゼロの最小発生で 1 の最大発生を持ったある要素が 2 つのウェブ・フラグメントのなかに異なって指定されていて、メインの web.xml には存在しないときはエラーとして考えられる。例えば、もし 2 つのウェブ・フラグメントたちが同じサーブレットを、異なった<load-on-startup>で宣言し、また同じサーブレットがまたメインの web.xml で<load-on-startup>なしで宣言されているときは、エラーが報告されねばならない。
iv. <welcome-file>宣言は付加的である。
v. 同じ<servlet-name>をもった<servlet-mapping>要素たちは付加的である。
vi. 同じ<filter-name>を持った<filter-mapping>要素たちは付加的である。
vii. 同じ<listener-class> をもった複数の<listener>要素たちは単一の<listener>宣言として取り扱われる。
viii. 合併した結果の web.xml は、そのすべてのウェブ・フラグメントたちが<distributable>としてマークされているときに限り<distributable>として考えられる。
ix. あるウェブ・フラグメントのトップ・レベルの<icon>とその子供の要素たち、<display-name>、及び<description>要素たちは無視される。
x. jsp-property-group は付加的である。jar ファイルの META-INF/resources ディレクトリのなかに性的なリソースたちをバンドルするときは、jsp-configy 要素は拡張マッピングではなくて url-pattern を使うことが勧告される。あるフラグメントの更なる JSP リソースたちはもしそれがひとつ存在していればそのフラグメント名とおなじサブ・ディレクトリに置くべきである。これによりあるウェブ・フラグメントの jsp-property-group がそのアプリケーションのメインの docroot のなかにある JSP たち、また META-INF/resources ディレクトしないのあるフラグメントのなかの JSP たちに影響を与えるのが防止される。

h. 総てのリソース参照要素たち(nv-entry、ejb-ref、ejblocal- ref、service-ref、resource-ref、resource-env-ref、message-destination-ref、persistence-context-ref、及び persistence-unit-ref)にたいし、以下の規則が適用される:

i. もし何らかのリソース参照要素があるウェブ・フラグメントに存在していて、メインの web.xml に存在しないときは、そのメインの web.xml はそのウェブ・フラグメントからその値を引き継ぐ。もしその要素がメインの web.xml とウェブ・フラグメントの双方に同じ名前で存在するときは、web.xml が優先する。そのフラグメントの子供の要素のどれも、以下に定められた injection-target 以外は、メインの web.xml に合体されない。例えば、もしメインの web.xml とあるウェブ・フラグメントの双方が同じ <resource-ref-name>を持った<resource-ref>を宣言しているときは、以下に記した <injection-target>を除いてそのウェブ・フラグメントからの子供の要素たちが合体されることなく web.xml からの<resource-ref>が使われる。
ii. もし 2 つのウェブ・フラグメントたちのなかで同じ名前を持ったあるリソース参照要素があって、メインの web.xml にはないときはエラーとして考えられる。例えば、2 つのウェブ・フラグメントたちが同じ<resource-ref-name>要素を持った<resource-ref>を宣言しており、一方メインの web.xml で指定されていないときは、エラーが報告され、そのアプリケーションの配備を失敗としなければ**ならない**。
iii. もしあるリソース参照要素がメインの web.xml に存在しているが<injection-target>要素が指定されておらず、ウェブ・フラグメントが同じ名前で同じリソース参照要素を指定していてまた<injection-target>が指定されているときは、そのフラグメントからの<injection-target>要素たちがメインの web.xml に合体される。しかしながらもしあるリソース参照要素がメインの web.xml に存在し、それが少なくともひとつの <injection-target>要素を指定しているときは、その<injection-target>要素はそのフラグメントからは合体されない。<injection-target>のリストを完全なものにするのはその web.xml の著者の責任である。

i. 上で規定された web-fragment.xml のための合体ルールたちに加えて、リソース参照アノテーションたち
(@Resource、@Resources、@EJB、@EJBs、@WebServiceRef、@WebServiceRefs、@PersistenceContext、@PersistenceContexts、@PersistenceUnit、及び @PersistenceUnits)を使うときに以下のルールたちが適用される。
i. もしあるリソース参照アノテーションがあるクラスに適用されるときは、それはリソース定義と等価であるが、injection-target を定義することとは等価ではない。上記のルールたちはこの場合 injection-target に適用される。もしあるリソース参照アノテーションがあるフィールド上で使われているときはそれは web.xml のなかの injection-target 要素を定義することと等価である。しかしながら記述子のなかに injection-target 要素が無い場合は、フラグメントたちからの injection-target が上記で規定したように web.xml に合体される。もし一方メインの web.xml に injection-target が存在し、同じリソース名を持ったリソース参照アノテーションが存在するときは、それはリソース参照アノテーションのオーバライドとして考えられる。この場合は記述子のなかで規定された injection-target があるので、上記で規定された規則たちはそのリソース参照アノテーションの値をオーバライドすることに加えて適用される。

j. もし同じ名前を持ったある data-source が 2 つのウェブ・フラグメントのなかで指定されていて、メインの web.xml にない場合は、エラーとみなされる。そのアプリケーションは配備に失敗しなければならない。web.xml のなかで指定されているときは、それはウェブ・フラグメントのなかで指定されている値をオーバライドする。以下は異なったケースでの結果を示す例である。

コード例 8-4

```
web.xml -resource-ref 指定なし

Fragment 1
web-fragment.xml
<resource-ref>
      <resource-ref-name="foo">
      ...
      <injection-target>
            <injection-target-class>
                  com.foo.Bar.class
            </injection-target-class>
            <injection-target-name>
                  baz
            </injection-target-name>
      </injection-target>
</resource-ref>

実効的なメタデータは次のようになる。

<resource-ref>
      <resource-ref-name="foo">
      ....
      <injection-target>
            <injection-target-class>
                  com.foo.Bar.class
            </injection-target-class>
            <injection-target-name>
                  baz
            </injection-target-name>
      </injection-target>
</resource-ref>
```

コード例 8-5

```
web.xml
<resource-ref>
      <resource-ref-name="foo">
      ...
      </resource-ref>

Fragment 1
web-fragment.xml
<resource-ref>
      <resource-ref-name="foo">
      ...
      <injection-target>
            <injection-target-class>
                  com.foo.Bar.class
            </injection-target-class>
            <injection-target-name>
                  baz
```

```
            </injection-target-name>
      </injection-target>
</resource-ref>

Fragment 2
web-fragment.xml
<resource-ref>
      <resource-ref-name="foo">
            ...
      <injection-target>
            <injection-target-class>
                  com.foo.Bar2.class
            </injection-target-class>
            <injection-target-name>
                  baz2
            </injection-target-name>
      </injection-target>
</resource-ref>
```

実効的なメタデータは次のようになる。

```
<resource-ref>
      <resource-ref-name="foo">
            ....
      <injection-target>
            <injection-target-class>
                  com.foo.Bar.class
            </injection-target-class>
            <injection-target-name>
                  baz
            </injection-target-name>
      </injection-target>
      <injection-target>
            <injection-target-class>
                  com.foo.Bar2.class
            </injection-target-class>
            <injection-target-name>
                  baz2
            </injection-target-name>
      </injection-target>
</resource-ref>
```

コード例 8-6

```
web.xml
<resource-ref>
      <resource-ref-name="foo">
      <injection-target>
            <injection-target-class>
                  com.foo.Bar3.class
            </injection-target-class>
            <injection-target-name>
                  baz3
            </injection-target-name>
```

```
...
</resource-ref>

Fragment 1
web-fragment.xml<resource-ref>
      <resource-ref-name="foo">
      ...
      <injection-target>
            <injection-target-class>
                  com.foo.Bar.class
            </injection-target-class>
            <injection-target-name>
                  baz
            </injection-target-name>
      </injection-target>
</resource-ref>

Fragment 2
web-fragment.xml<resource-ref>
      <resource-ref-name="foo">
      ...
      <injection-target>
            <injection-target-class>
                  com.foo.Bar2.class
            </injection-target-class>
            <injection-target-name>
                  baz2
            </injection-target-name>
      </injection-target>
</resource-ref>
```

実効的なメタデータは次のようになる。

```
<resource-ref>
      <resource-ref-name="foo">
            <injection-target>
            <injection-target-class>
                  com.foo.Bar3.class
            </injection-target-class>
            <injection-target-name>
                  baz3
            </injection-target-name>
...
</resource-ref>
```

フラグメント 1 及び 2 からの<injection-target>は合体されることはなく、総ての<injection-target>要素を確実にメインの web.xml で指定するのは開発者の責任ということになろう。

k. もしメインの web.xml で<post-construct>要素が指定されておらずウェブ・フラグメントで<post-construct>が指定されていたときは、ウェブ・フラグメントからの<post-construct>要素はメインの web.xml に合体される。しかしながらメインの web.xml で少なくともひとつの<post-construct>要素が指定されているときは、そのフラグメントからの<post-construct> 要素は合体されない。web.xml で<post-construct>のリストを確実に完全なものとするのはその著者の責任ということになろう。

l. もしメインの web.xml で<pre-destroy>要素が指定されておらずウェブ・フラグメントで<pre-destroy>が指定されていたときは、ウェブ・フラグメントからの<pre-destroy>要素はメインの web.xml に合体される。しかしながらメインの web.xml で少なくともひとつの<pre-destroy>要素が指定されているときは、そのフラグメントからの<pre-destroy>要素は合体されない。web.xml で<pre-destroy>のリストを確実に完全なものとするのはその著者の責任ということになろう。

m. web-fragment.xml の処理後に、対応するフラグメントからのアノテーションたちが処理され、次のフラグメントを処理する前にそのフラグメントのための実効的なメタデータが完成される。アノテーションたちの処理には以下の規則が使われる:

n. アノテーションを介して指定される記述子で既に出ていないどのメタデータも実効的な記述子の強化のために使われる。
   i. メインの web.xml またはウェブ・フラグメントで指定された設定はアノテーションたちを介して指定された設定より優先する。
   ii. @WebServlet を介して定義されたサーブレットが記述子を介した値をオーバライドするためには、記述子のなかのサーブレットの名前がアノテーションで指定されたサーブレットの名前(明示的に指定された、あるいはアノテーションを介して指定されていないときはデフォルト名)と一致していなければ**ならない**。
   iii. アノテーションを介して指定されたサーブレットとフィルタたちの init パラメタたちは、その init パラメタがアノテーションを介した名前と正確に一致するときは、記述子によってオーバライドされる。
   iv. url-patterns は、与えられたサーブレット名にたいし記述子で指定されているときは、アノテーションを介して指定された URL パタンをオーバライドする。
   v. @WebFilter を介して定義されたフィルタにたいし、記述子を介した値がオーバライドするためには、記述子のなかのフィルタの名前がアノテーションで指定されたフィルタの名前(明示的に指定された、あるいはアノテーションを介して指定されていないときはデフォルト名)と一致していなければ**ならない**。
   vi. あるフィルタに対し適用される url-patterns は、与えられたフィルタ名にたいし記述子で指定されているときは、アノテーションを介して指定された url patterns をオーバライドする。
   vii. あるフィルタに対し適用される DispatcherTypes は、与えられたフィルタ名にたいし記述子で指定されているときは、アノテーションを介して指定されている DispatcherTypes をオーバライドする。
   viii. 以下の例たちは上記の規則の幾つかを示すものである - あるサーブレットがあるアノテーションを介して宣言され、記述子のなかで対応する web.xml でパッケージ化されている。

```
@WebServlet(urlPatterns=”/MyPattern”, initParams=
{@WebInitParam(name="ccc", value="333")})

public class com.acme.Foo extends HttpServlet
      {
      ...
      }

web.xml
```

```
<servlet>
      <servlet-class>com.acme.Foo</servlet-class>
      <servlet-name>Foo</servlet-name>
      <init-param>
            <param-name>aaa</param-name>
            <param-value>111</param-value>
      </init-param>
</servlet>

<servlet>
      <servlet-class>com.acme.Foo</servlet-class>
      <servlet-name>Fum</servlet-name>
      <init-param>
            <param-name>bbb</param-name>
            <param-value>222</param-value>
      </init-param>
</servlet>

<servlet-mapping>
      <servlet-name>Foo</servlet-name>
      <url-pattern>/foo/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
      <servlet-name>Fum</servlet-name>
      <url-pattern>/fum/*</url-pattern>
</servlet-mapping>
```

アノテーションを介して宣言されたサーブレット名が web.xml で宣言されたサーブレットの名前と一致していないので、このアノテーションは web.xml 内の他の宣言に加えて新しいサーブレット宣言を指定していて、以下のものと等価になる:

```
<servlet>
      <servlet-class>com.acme.Foo</servlet-class>
      <servlet-name>com.acme.Foo</servlet-name>
      <init-param>
            <param-name>ccc</param-name>
            <param-value>333</param-name>
</servlet>

If the above web.xml were replaced with the following

<servlet>
      <servlet-class>com.acme.Foo</servlet-class>
      <servlet-name>com.acme.Foo</servlet-name>
      <init-param>
            <param-name>aaa</param-name>
            <param-value>111</param-value>
      </init-param>
</servlet>

<servlet-mapping>
      <servlet-name>com.acme.Foo</servlet-name>
      <url-pattern>/foo/*</url-pattern>
</servlet-mapping>
```

そうすると実効的な記述子は以下のものと等価になる:

```
<servlet>
      <servlet-class>com.acme.Foo</servlet-class>
      <servlet-name>com.acme.Foo</servlet-name>
      <init-param>
            <param-name>aaa</param-name>
            <param-value>111</param-value>
      </init-param>
      <init-param>
            <param-name>ccc</param-name>
            <param-value>333</param-value>
      </init-param>
</servlet>

<servlet-mapping>
      <servlet-name>com.acme.Foo</servlet-name>
      <url-pattern>/foo/*</url-pattern>
</servlet-mapping>
```

8.2.4 共有ライブラリ/ランタイムのプラグ化性(Shared libraries / runtimes pluggability)

フラグメントたちのサポートとアノテーションたちの使用に加えて、要求のひとつは我々が WEB-INF/lib のなかにバンドルされているものたちをプラグ・インできることだけでなく、ウェブ・コンテナの上に構築される JAX-WS、JAX-RS、及び JSF のようなものをウェブ・コンテナにプラグ・インできることを含めて、フレームワークたちの共有コピーたちをプラグ・インできることである。
ServletContainerInitializer が以下に示すようにそのようなユース・ケースを処理できるようにしている。

ServletContainerInitializer のインスタンスがコンテナ / アプリケーションスタートアップ時にそのコンテナによって jar サービス API を介して検索される。ServletContainerInitializer の実装を提供しているフレームワークは、その jar ファイルの META-INF/services ディレクトリのなかで jar サービス API 毎に ServletContainerInitializer の実装クラスをさし示している javax.servlet.ServletContainerInitializer と呼ばれるファイルをバンドルしなければ**ならない**。

ServletContainerInitializer に加えて HandlesTypes というアノテーションがある。このアノテーションは、値で指定されたクラスでアノテートされた、あるいはあるクラスがそれらのクラスのひとつをそのクラスたちのスーパー・タイプのなかのどこかで継承 / 実装指定す場合、のいずれかであるクラスたちへの関心を示すための ServletContainerInitializer の実装物に付加される。コンテナはこの HandlesTypes アノテーションを使って何時 initializer の onStartup メソッドを呼ぶかを判断する。あるアプリケーションのクラスたちを知らべこれらが ServletContainerInitializer の HandlesTypes アノテーションで指定された基準のどれかに合致しているかを判断する際は、そのコンテナはそのアプリケーションのオプション的な JAR ファイルのひとつまたはそれ以上が欠けているときにクラス・ローディングの問題に面するかもしれない。そのコンテナはこのタイプのクラス・ローディングの失敗でそのアプリケーションが正しく動作できなくなるかどうかを判断する立場にないので、そのコンテナはそれらを無視しなければならないが、一方同時にそれらをログする設定オプションを用意しなければならない。

もし ServletContainerInitializer のある実装物が@HandlesTypes アノテーションを持っていない、あるいは指定された HandlesType のどれにも合致するものが無いときは、その実装物は Set の値が null である各アプリケーションごとに一回呼び出される。これによりイニシアライザはそのアプリケーションがサーブレット / フィルタを初期化する必要があるかどうかを利用できるリソースに基づいて判断できる。

ServletContainerInitializer の onStartup メソッドはそのアプリケーションがリスナの何らかのイベントが作動する前に到来しているときに呼び出される。ServletContainerInitializer の onStartup メソッドはそのイニシアライザが関心を持っているあるいは@HandlesTypes アノテーションを介して指定されたクラスたちのどれかでアノテートされているときクラスを継承 / 実装のどちらかをしている Class たちの Set を得る。

以下の具体的例が如何にこれが機能するかを示している。

JAX-WS のウェブ・サービスのランタイムを取り上げてみる。JAX-WS ランタイムの実装は一般的には war ファイル毎にはバンドルされていない。その実装は ServletContainerInitializer の実装（以下を見られたい）をバンドルし、コンテナはサービス API を使って検索をすることになる（その jar ファイルはその META-INF/services ディレクトリに javax.servlet.ServletContainerInitializer と呼ばれるファイルをバンドルし、このファイルは以下の JAXWSServletContainerInitializer を指すことになる）。

```
@HandlesTypes(WebService.class)
JAXWSServletContainerInitializer implements
ServletContainerInitializer{
      public void onStartup(Set<Class<?>> c, ServletContext ctx)
throws ServletException {
      // ランタイムを初期化しマッピング等を設定するための JAX-WS 固有のコードが
      // ここに置かれる
      ServletRegistration reg = ctx.addServlet("JAXWSServlet",
"com.sun.webservice.JAXWSServlet");
      reg.addServletMapping("/foo");
}
```

このフレームワークの jar はまた war ファイルの WEB-INF/lib ディレクトリにバンドルせれることができる。もし ServletContainerInitializer があるアプリケーションの WEB-INF/lib ディレクトリの内部の JAR ファイルにバンドルされていると、その onStartup メソッドはバンドルしているアプリケーションの startup の時間に一回のみ呼び出される。一方 ServletContainerInitializer が WEB-INF/lib ディレクトリの外部の JAR ファイルにバンドルされていて、それでもランタイムのサービス提供者の検索のメカニズムによって発見可能な場合は、その onStartup メソッドはあるアプリケーションが開始するたびに毎回呼び出される。

ServletContainerInitializer の実装物たちはそのランタイムのサービス検索メカニズムまたはそれと意味的に等価なコンテナ固有のメカニズムによって発見される。いずれの場合でも、絶対順序付けから外されているウェブ・フラグメント JAR ファイルからの ServletContainerInitializer サービスは無視され**ねばならず**、。これらのサービスが発見される順序はそのアプリケーションのクラス・ローディングの委譲モデルに従わねば**ならない**。

8.3 JSP コンテナのプラグ化性（JSP container pluggability）

ServletContainerInitializer とプログラム的な登録機能により、サーブレット・コンテナが web.xml と web-fragment.xml のリソースたちのみの解析の責任を持ちタグ・ライブラリ記述子（TLD：Tag Library Descriptor）の解析を JSP コンテナに委譲することで、サーブレットと JSP のコンテナたち間の責任分離が可能となっている。

これまでは、ウェブ・コンテナが何らかのリスナ宣言にたいし TLD リソースたちをスキャンしなければならなかった。サーブレット 3.0 では、この責任は JSP コンテナに委譲されている。サーブレット 3.0 対応サーブレット・コンテナを組み込んでいるある JSP コンテナは、自分自身で ServletContainerInitializer 実装を用意し、何らかの TLD リソースたちの onStartup メソッドに渡す ServletContext をサーチし、リスナ宣言たち用のこれらのリソースをスキャンし、そして対応するリスナたちを ServletContext に登録する。

加えて、サーブレット 3.0 以前では、何らかの jsp-config 関連の設定に対しあるアプリケーションの配備記述子をスキャンしなければならなかった。サーブレット 3.0 では、サーブレット・コンテナは ServletContext.getJspConfigDescriptor メソッドを介してそのアプリケーションの web.xml 及び web-fragment.xml 配備記述子からの jsp-config 関連設定が可能となっていなければならない。

TLD のなかで発見された及びプログラム的に登録されたどの ServletContextListeners もそれらが提供する機能に制限されている。サーブレット 3.0 で追加されている ServletContext API メソッドをこれらで呼ぶと UnsupportedOperationException が起きる。

加えて、サーブレット 3.0 対応のサーブレット・コンテナは javax.servlet.context.orderedLibs という名前の ServletContext 属性を提供しなければならないが、この属性値（java.util.List<java.lang.String>の型）は ServletContext によって代表されているそのアプリケーションの WEB-INF/lib ディレクトリ内の JAR ファイルたちの名前のリストが含めれており、そのリストはウェブ・フラグメント名で順序付けされているか（ありえる排除はもしフラグメントの JAR ファイルたちが absolute-ordering から除外されている場合である）、そのアプリケーションが絶対または相対順序付けを全く指定していないときは null である。

8.4 アノテーションとフラグメントの処理（Processing annotations and fragments）

ウェブ・アプリケーションたちはアノテーションたちと web.xml / webfragment. xml 配備記述子たちの双方を含むことが出来る。配備記述子が無い、あるいはあっても metadata-complete が true にセットされていない場合は、そのアプリケーションで使われている場合は web.xml、fragment.xml、及びアノテーションたちは処理されなければならない。下表はアノテーションたちと web.xml フラグメントを処理するかしないかを示している：

表 8-1 アノテーションとウェブ・フラグメントの処理要求

配備記述子	metadata-complete	アノテーションとウェブ・フラグメントの処理
web.xml 2.5	Yes	No
web.xml 2.5	no	yes
web.xml 3.0	yes	no
web.xml 3.0	no	yes

第9章　　要求のディスパッチ（Dispatching Requests）

あるウェブ・アプリケーションを構築する際は、ある要求処理を別のサーブレットに渡す、あるいは別のサーブレットの出力を応答に含めることはしばしば有用である。RequestDispatcher インターフェイスはこれを達成する為のメカニズムを提供している。その要求で非同期処理が可能になっている場合は、AsyncContext によりユーザはその要求をサーブレット・コンテナにディスパッチして戻すことが出来る。

9.1　RequestDispatcher の取得（Obtaining a RequestDispatcher）

RequestDispatcher インターフェイスを実装したオブジェクトは以下のメソッドたちにより ServletContext から取得される：

- getRequestDispatcher
- getNamedDispatcher

getRequestDispatcher メソッドはその ServletContext の適用範囲内でのあるパスを記述した String 引数を取る。このパスはその ServletContext のルートに対し相対的なもので、‘/’で始まるか空でなければならない。このメソッドはこのパスを使って、第 12 章「要求のサーブレットへのマッピング」の規則でマッチするサーブレット・パスを使ってサーブレットを検索し、それを RequestDispatcher オブジェクトでラップし、結果としてのオブジェクトを返す。与えられたパス上でサーブレットが見つけられないときは、RequestDispatcher はそのパス上のコンテントを返すものになる。

getNamedDispatcher メソッドはその ServletContext が知っているサーブレットの名前を示す String 引数をとる。もしサーブレットが見つかれば、それは RequestDispatcher でラップされ、そのオブジェクトが返される。与えられた名前に対応するサーブレットが見つからないときは、このメソッドは null を返さねばならない。

現在の要求のパスに相対的な相対パス（ServletContext のルートに対する相対ではない）を使って RequestDispatcher オブジェクトを取得できるようにする為に、ServletRequest インターフェイスには getRequestDispatcher メソッドが用意されている。

このメソッドの振舞いは ServletContext のなかの同じ名前のメソッドと同じである。サーブレット・コンテナは要求オブジェクトの中のこの情報を使って、与えられた現在のサーブレットに対する相対パスを完全なパスに変換する。例えば、’/’がルートであるコンテキストにおいて/garden/tools.html への要求にたいし、ServletRequest.getRequestDispatcher("header.html")を介して得られる要求ディスパッチャは ServletContext.getRequestDispatcher("/garden/header.html")呼び出しとまさしく同じふるまいをする。

9.1.1 要求ディスパッチャ・パスのクエリ文字列（Query Strings in Request Dispatcher Paths）

パス情報を使って RequestDispatcher オブジェクトを作る ServletContext と ServletRequest のメソッドたちでは、そのパスにクエリ文字列情報をオプション的に付加できる。例えば、開発者は以下のコードで RequestDispatcher を取得出来よう:

```
String path = "/raisins.jsp?orderno=5";
RequestDispatcher rd = context.getRequestDispatcher(path);
rd.include(request, response);
```

RequestDispatcher を作るために使われるクエリ文字列で指定されたパラメタたちは、含められたサーブレットに渡される同じ名前をもった他のパラメタよりも優先する。ある RequestDispatcher に集められたパラメタたちは include または forward 呼び出し期間中のみの適用範囲である。

9.2 要求ディスパッチャの使用（Using a Request Dispatcher）

要求ディスパッチャを使うには、サーブレットは RequestDispatcher の include メソッドまたは forward メソッドのどちらかを呼ぶ。これらのメソッドたちへのパラメタたちは、javax.servlet インターフェイスの service メソッドを介して渡される要求と応答の引数、あるいはバージョン 2.3 で導入された要求と応答のラッパ・クラスのサブクラスのインスタンスたちのどちらかである。後者の場合は、ラッパーのインスタンスはそのコンテナが service メソッドに渡した要求と応答のオブジェクトたちをラップしなければならない。コンテナのプロバイダはターゲットのサーブレットへの要求のディスパッチが確実にオリジナルの要求と同じ JVM の同じスレッドのなかで起きるようにしなければならない。

9.3 include メソッド（The Include Method）

RequestDispatcher インターフェイスの include メソッドは何時でも呼ぶことが出来る。この include メソッドのターゲットのサーブレットはその要求オブジェクトの総ての側面にアクセス出来るが、その応答オブジェクトの使用はより制限的なものになっている。

ターゲットのサーブレットはこの応答オブジェクトの ServletOutputStream または Writer に情報を書き、応答バッファの終わりを過ぎてコンテントを書くことであるいは明示的に ServletResponse インターフェイスの flushBuffer メソッドを呼ぶことでのみ応答をコミットできる。ターゲットのターゲットのサーブレットはヘッダをセットしたり、あるいは HttpServletRequest.getSession()と HttpServletRequest.getSession(boolean)のメソッドを除いてその応答のヘッダに影響を与えるメソッドを呼ぶことは出来ない。ヘッダをセットする何らかの試みは無視され、クッキー応答ヘッダを追加する HttpServletRequest.getSession()または HttpServletRequest.getSession(boolean)の呼び出しはその応答がコミットされているときは IllegalStateException をスローしなければならない。

デフォルトのサーブレットが RequestDispatch.include()になっていて、要求されているリソースが存在しないときは、そのデフォルトのサーブレットは FileNotFoundException をスローしなければならない。もしこの例外が捕捉され処理されず、その応答がコミットされていないときは、HTTP 応答のステータス・コードは 500 にセットしなければならない。

9.3.1 インクルードされた要求パラメタたち（Included Request Parameters）

getNamedDispatcher メソッドを使って得られたサーブレットを除いて、RequestDispatcher の include メソッドを使って別のサーブレットから呼ばれたサーブレットは、それが呼ばれたパスへのアクセスを持つ。

以下の要求属性たちがセットされねばならない:

```
javax.servlet.include.request_uri
javax.servlet.include.context_path
javax.servlet.include.servlet_path
javax.servlet.include.path_info
javax.servlet.include.query_string
```

これらの属性は要求オブジェクトの getAttribute メソッドを介してインクルードされたサーブレットからアクセスでき、その値は要求 URI、コンテキスト・パス、サーブレット・パス、パス情報、及びクエリ文字列と等しくなければならない。もしその要求がその後にインクルードされるときは、これらの属性はそのインクルードで置き換えられる。

もしインクルードされたサーブレットが getNamedDispatcher メソッドを使って得られているときは、これらの属性はセットされてはならない。

9.4 forward メソッド（The Forward Method）

RequestDispatcher インターフェイスの forward メソッドは、クライアントに出力がコミットされていないときに限り呼び出しサーブレットによって呼び出される。もし未だコミットされていない出力データが応答バッファ内に存在しているときは、そのコンテントはターゲットのサーブレットの service メソッドが呼ばれる前にクリアされていなければならない。もしその応答がコミットされているときは、IllegalStateException がスローされねばならない。

ターゲットのサーブレットに露出されているその要求のパス情報は、RequestDispatcher を取得するのに使われたパスを反映しなければならない。

これに対する唯一の例外はその RequestDispatcher が getNamedDispatcher で取得されているときである。この場合はその要求オブジェクトのパス情報はオリジナルの要求のそれを反映したものでなければならない。

RequestDispatcher インターフェイスの forwar メソッドが例外なしで戻る前にその応答コンテントはそのサーブレット・コンテナによって送信、コミット、そしてクローズされねばならない。
RequestDispatcher.forward()のターゲットのなかでエラーが生じたときは、その例外は総ての呼び出しのフィルタたちとサーブレットたちに戻り伝播され、最終的にコンテナに戻ることになる。

9.4.1 クエリ文字列（Query String）

要求ディスパッチのこのメカニズムは、要求をフォワードあるいはインクルードするときはクエリ文字列パラメタたちを集める責任を持つ。

9.4.2 フォワードされた要求パラメタたち（Forwarded Request Parameters）

getNamedDispatcher メソッドを使って得られたサーブレットを除いて、RequestDispatcher の forward メソッドを使って別のサーブレットから呼ばれたサーブレットは、それが呼ばれたパスへのアクセスを持つ。

以下の要求属性たちがセットされねばならない：

```
javax.servlet.forward.request_uri
javax.servlet.forward.context_path
javax.servlet.forward.servlet_path
javax.servlet.forward.path_info
javax.servlet.forward.query_string
```

これらの属性の値は、クライアントから受けた要求である呼び出しチェインの最初のサーブレットに渡された要求オブジェクトで呼び出された HttpServletRequest の各々 getRequestURI、getContextPath、getServletPath、getPathInfo、getQueryString のメソッドたちが戻す値と等しくなければならない。

これらの属性は要求オブジェクトの getAttribute メソッドを介してフォワードされたサーブレットからアクセスできる。これらの値は例え複数のフォワードとその後のインクルードが呼ばれるような状況にあっても常にオリジナルの要求の情報を反映しなければならないことに注意のこと。もしフォワードされたサーブレットが getNamedDispatcher メソッドにより取得されたものなら、これらの属性はセットされてはならない。

9.5 エラー処理（Error Handling）

ある要求ディスパッチャのターゲットであるサーブレットがランタイムの例外または ServletException または IOException の型の例外をスローするときは、その例外は呼び出しているサーブレットに伝

搬されねばならない。その他の総ての例外は ServletExceptions としてラップされ、これは伝搬されてはいけないので、その例外のルート要因(root cause)がオリジナルの例外にセットされる。

9.6 AsyncContext の取得（Obtaining an AsyncContext）

AsyncContext インターフェイスを実装しているオブジェクトは、startAsync メソッドたちのひとつを介して ServletRequest から得られることがある。AsyncContext を有している場合は、complete()メソッドを介して、あるいは以下に示す dispatch のメソッドたちのひとつを使うかのいずれかで、AsyncContext を使ってその要求処理を完了させることができる。

9.7 ディスパッチのメソッド（The Dispatch Method）

以下のメソッドたちが AsyncContext からの要求をディスパッチするのに使える:

- dispatch(path)
  この dispatch メソッドはその ServletContext の適用範囲内のパスを記述した String 引数をとる。このパスはその ServletContext のルートの対し相対的で‘/’で始まらねばならない。
- dispatch(servletContext, path)
  この dispatch メソッドは指定された ServletContext の適用範囲内のパスを記述した String 引数をとる。このパスはその ServletContext のルートの対し相対的で‘/’で始まらねばならない。
- Dispatch()
  この dispatch メソッドは引数をとらない。これはパスとしてオリジナルの URI を使う。もしその AsyncContext が startAsync(ServletRequest, ServletResponse)を介して初期化されており、渡された要求が HttpServletRequest のインスタンスである場合は、そのディスパッチは HttpServletRequest.getRequestURI()が戻す URI むけとなる。それ以外ではそのディスパッチはそれがコンテナによって最後にディスパッチされた要求の URI 向けになる。

AsyncContext インターフェイスの dispatch メソッドたちのひとつは、非同期イベント発生を待っているアプリケーションによって呼ばれて良い。もし AsyncContext 上で complete()が呼ばれているときは、IllegalStateException がスローされねばならない。この dispatch メソッドたちの総てが即座に戻りその応答をコミットしない。ターゲットのサーブレットに露出されている要求オブジェクトのパス要素は、AsyncContext.dispatch で指定されたパスを反映したものでなければならない。

9.7.1 クエリ文字列（Query String）

要求ディスパッチのこのメカニズムは、要求をディスパッチするときはクエリ文字列パラメタたちを集める責任を持つ。

9.7.2 ディスパッチされた要求パラメタたち（Dispatched Request Parameters）

AsyncContext の dispatch メソッドを使って呼び出されたサーブレットは、オリジナルの要求のパスへのアクセスを持つ。

以下の要求属性たちがセットされねばならない:

```
javax.servlet.async.request_uri
javax.servlet.async.context_path
javax.servlet.async.servlet_path
javax.servlet.async.path_info
javax.servlet.async.query_string
```

これらの属性の値は、クライアントから受けた要求である呼び出しチェインの最初のサーブレットに渡された要求オブジェクトで呼び出された HttpServletRequest の各々 getRequestURI、getContextPath、getServletPath、 getPathInfo、getQueryString のメソッドたちが戻す値と等しくなければならない。

これらの属性は要求オブジェクトの getAttribute メソッドを介してフォワードされたサーブレットからアクセスできる。これらの値は例え複数のフォワードとその後のインクルードが呼ばれるような状況にあっても常にオリジナルの要求の情報を反映しなければならないことに注意のこと。

第10章　　ウェブ・アプリケーション（Web Applications）

ウェブ・アプリケーションとはあるウェブ・サーバ上での完全なあるアプリケーションを構成するサーブレットたち、HTML ページたち、クラスたち、及びその他のリソースたちを集めたものである。ウェブ・アプリケーションはバンドルされ、また複数のベンダからの複数のコンテナ上で走ることがあり得る。

10.1 ウェブ・サーバ内のウェブ・アプリケーション（Web Applications Within Web Servers）

ウェブ・アプリケーションはウェブ・サーバ内の特定のパスをルートとしている。例えば、あるカタログのアプリケーションは http://www.mycorp.com/catalog に置かれることになる。このプレフィックスで始まる総ての要求はこのカタログのアプリケーションを代表している ServletContext に渡される。

サーブレット・コンテナはウェブ・アプリケーションの自動発生のための規則を確立できる。例えば~user/というマッピングは/home/user/public_html/に置かれたウェブ・アプリケーションにマップするのに使うことができる。

デフォルトとして、あるウェブ・アプリケーションのインスタンスは何時に時点でもひとつの VM 上で走らねばならない。この振る舞いは、そのアプリケーションが配備記述子で“distributable”とマークされているときはオーバライドされることができる。分散可能(distributable)とマークされたアプリケーションは通常のウェブ・アプリケーションで要求されているよりもより制限的な規則のセットに従わねばならない。

10.2 ServletContext との関連（Relationship to ServletContext）

サーブレット・コンテキストはあるウェブ・アプリケーションとある ServletContext 間の 1 対 1 の対応を厳格にとらねばならない。ServletContext のオブジェクトはあるサーブレットから見たそのアプリケーションのビューを提供している。

10.3 ウェブ・アプリケーションの要素たち（Elements of a Web Application）

ウェブ・アプリケーションは以下のアイテムたちで構成される：

- サーブレットたち

- JSP[1] ページたち
- ユーティリティのクラスたち
- 静的ドキュメントたち
- クライアント・サイドの Java Applet、bean、及び class たち
- 上記の総ての要素を結び付ける記述的なメタ情報

1 http://java.sun.com/products/jsp から取得できる JSP の仕様を参照のこと

10.4 配備階層（Deployment Hierarchies）

本仕様ではオープンなファイル・システムのなかで、アーカイブ・ファイルのなかで、あるいは何らかの他の形式で存在し得る、配備とパッケージングの目的に使われる階層的構造を定めている。サーブレットのコンテナはランタイムの表現としてこの構造をサポートすることが勧告されるが拘束されるものではない。

10.5 ディレクトリ構造（Directory Structure）

ウェブ・アプリケーションはディレクトリたちの構造化された階層として存在する。この階層のルートがそのアプリケーションの要素であるファイルたちのドキュメント・ルートになる。例えば、あるウェブ・コンテナのなかで/catalog というコンテキスト・パスを持ったあるウェブ・アプリケーションでは、このアプリケーション階層のベースにある index.html ファイル、あるいは META-INF/resources ディレクトリのなかで index.html を含む WEB-INF/lib のなかの JAR ファイルのなかにある index.html ファイルは、/catalog/index.html からの要求を満足させるために使われる。もし index.html がルート・コンテキストの中、及びこのアプリケーションの WEB-INF/lib ディレクトリのなかの JAR ファイルの META-INF/resources ディレクトリの中の双方に存在する場合は、そのルート・コンテキストの中で得られるファイルが使われねば**ならない**。URL たちをコンテキスト・パスにマッチさせる規則は第 12 章「要求のサーブレットへのマップ」で示されている。あるアプリケーションのコンテキスト・パスがそのウェブ・アプリケーションのコンテントたちの URL 名前空間を決めているので、ウェブのコンテナはこの URL 名前空間と潜在的に矛盾を起こすようなコンテキスト・パスを決めるようなウェブ・アプリケーションを拒否しなければならない。これは例えば、同じコンテキスト・パスを持った第 2 のウェブ・アプリケーションを配備しようとするときに起き得る。要求は大文字と小文字を区別してリソースとのマッチがとられるので、潜在的な矛盾の判断も大文字と小文字を区分してなされねばならない。

“WEB-INF”という名前の特別なディレクトリがアプリケーション階層の中に存在している。このディレクトリはそのアプリケーションのドキュメント・ルートにないこのアプリケーションに関連する総てのものを含む。殆どの WEB-INF ノードはこのアプリケーションのパブリックなドキュメント・トリーの要素ではない。WEB-INF/lib ディレクトリの中に存在する JAR ファイルの METAINF/resources のなかでパッケージ化された静的なリソースと JSP たちを除いて、WEB-INF ディレクトリの中に含まれているどのファイルもコンテナによって直接クライアントにサービスされない。しかしながら WEB-INF ディレクトリの中のコンテンツは ServletContext 上の getResource 及び getResourceAsStream メソッドた

ちによってサーブレットのコードからは可視であり、RequestDispatcher 呼び出しを使って可視化される。したがってアプリケーションの開発者がウェブのクライアントに直接露出させたくないアプリケーション固有の設定情報にサーブレットからアクセスする必要があるときは、その開発者はそれをこのディレクトリに置くことになろう。要求は大文字と小文字を分けてリソース・マッピングとマッチがとられるので、例えば‘/WEB-INF/foo’、‘/WebiNf/ foo’へのクライアント要求は/WEB-INF にあるウェブ・アプリケーションのコンテンツが返されてはならず、何らかのかたちのそれのディレクトリ・リスティングも返してはならない。

WEB-INF ディレクトリの中身は:

- /WEB-INF/web.xml:配備記述子
- /WEB-INF/classes/:サーブレットとユーティリティのクラスたちのためのディレクトリ。このディレクトリの中のクラスたちはアプリケーションのクラス・ローダによって取得可能でなければならない。
- /WEB-INF/lib/*.jar:Java Archive ファイルたちのための領域。これらのファイルは JAR ファイルにパッケージされたサーブレット、ビーン、静的リソース、及び JSP たち、及びこのウェブ・アプリケーションにとって有用な他のユーティリティ・クラスたちを含んでいる。ウェブ・アプリケーションのクラス・ローダはこれらのアーカイブファイルたちのどれからもクラスたちがロードできねばならない。

ウェブ・アプリケーションのクラス・ローダは最初に WEB-INF/classes ディレクトリから、次に WEB-INF/lib の中のライブラリ JAR からクラスたちをロードしなければならない。また、静的リソースたちが JAR ファイルたちにパッケージされている場合を除き、クライアントからの WEB-INF/ディレクトリ内のリソースにアクセスするどの要求も SC_NOT_FOUND(404)応答で返さねばならない。

10.5.1.1 アプリケーション・ディレクトリ構造例(Example of Application Directory Structure)

以下はサンプルのウェブ・アプリケーションの総てのファイルのリストである:

```
/index.html
/howto.jsp
/feedback.jsp
/images/banner.gif
/images/jumping.gif
/WEB-INF/web.xml
/WEB-INF/lib/jspbean.jar
/WEB-INF/lib/catalog.jar!/META-
INF/resources/catalog/moreOffers/books.html
/WEB-INF/classes/com/mycorp/servlets/MyServlet.class
/WEB-INF/classes/com/mycorp/util/MyUtils.class
```

10.6 ウェブ・アプリケーションのアーカイブ・ファイル(Web Application Archive File)

ウェブ・アプリケーションは標準の Java アーカイブ・ツールを使って Web Archive (WAR)フォーマットにパッケージ化されサインされ得る。例えば、問題追跡(issue tracking)のためのあるアプリケーションは issuetrack.war と呼ばれるアーカイブ・ファイルにして配布され得る。そのような様式でパッケージ化したときには、Java アーカイブ・ツールにとって有用な情報を含む META-INF ディレクトリが用意される。このディレクトリはあるウェブ・クライアントの要求に応じてそのコンテナによって直接サービスされてはならないが、その内容は ServletContext の getResource 及び getResourceAsStream 呼び出しを介してサーブレットのコードからは可視である。また、META-INF ディレクトリ内のリソースにアクセスする要求は SC_NOT_FOUND(404)応答で戻されねばならない。

10.7 ウェブ・アプリケーションの配備記述子（Web Application Deployment Descriptor）

ウェブ・アプリケーション配備記述子（第 14 章の「配備記述子」を見られたい）は以下のタイプの設定と配備の為の情報を含んでいる:

- ServletContext の init パラメタ
- セッション設定
- サーブレット / JSP 定義
- サーブレット / JSP マッピング
- MIME タイプのマッピング
- ウェルカム・ファイル・リスト
- エラー・ページ
- セキュリティ

10.7.1 拡張物依存性（Dependencies On Extensions）

一連のアプリケーションが同じコードあるいはリソースを使っているときは、これらのアプリケーションは一般的にそのコンテナの中のライブラリ・ファイルたちとしてインストールされている。しばしばこれらのファイルは移植性を犠牲にすることなく使える共通のあるいは標準の API たちである。ひとつあるいは僅かのみのアプリケーションが使うファイルたちはそのウェブ・アプリケーションの要素としてアクセスするために利用できるようになる。コンテナはこれらのライブラリのためのディレクトリを用意しなければならない。このディレクトリ内に置かれたファイルたちは総てのアプリケーションから利用できなければならない。このディレクトリの場所はコンテナに依存する。サーブレット・コンテナがこれらのライブラリ・ファイルをロードするのに使うクラス・ローダは、同じ JVM 内の総てのウェブ・アプリケーションにとって同じものでなければならない。このクラス・ローダのインスタンスはそのウェブ・アプリケーションのクラス・ローダの親のクラス・ローダたちのチェインのどこかになければならない。

移植性を維持させるために、アプリケーションの開発者たちはあるウェブ・コンテナの中にどのような拡張物がインストールされているかを知る必要があり、コンテナたちは WAR のなかのサーブレットたちがそのようなライブラリたちにどのような依存性を持っているかを知る必要がある。

そのようなコア機能の拡張または拡張たちに依存しているアプリケーションの開発者は、WAR ファイルの中に META-INF/MANIFEST.MF エントリを用意しその WAR が必要としている総ての拡張物をリストしなければならない。このマニフェストのエントリの書式は標準の JAR マニフェスト書式に従わねばならない。そのウェブ・アプリケーションの配備中に、そのウェブ・コンテナはオプション的パッケージ・バージョニング(Optional Package Versioning:
http://java.sun.com/j2se/1.4/docs/guide/extensions/)メカニズムで定められた規則に従ってそのアプリケーションが利用できる拡張物たちの正確なバージョンをつくらねばならない。

ウェブ・コンテナたちはまた WAR の WEB-INF/lib のなかのライブラリ JAR たちのどれかのマニフェスト・エントリで示されている宣言された依存性を認識できなければならない。もしあるウェブ・コンテナがこのやり方で宣言された依存性を満足できないときは、そのコンテナは情報的エラー・メッセージでそのアプリケーションを拒否しなければならない。

10.7.2 ウェブ・アプリケーションのクラス・ローダ（Web Application Class Loader）

WAR のなかのサーブレットをロードするためにコンテナが使うクラス・ローダは、WAR の中にある JAR たちに含まれている任意のリソースを開発者が getResource を使って通常の Java SE セマンティクスに従ってロードできるようになっていなければならない。Java EE ライセンス合意に示されているように、Java EE 製品の要素でないサーブレット・コンテナたちは、Java SE が加工を認めていない java.*及び javax.*名前空間のような Java SE プラットホームのクラスたちをそのアプリケーションがオーバライドさせてはならない。またコンテナにわたってのライブラリ JAR たちの中に存在するクラスたちとリソースたちの好みに基づきその WAR 内にパッケージ化されたクラスたちとリソースたちがロードされるようにそのアプリケーション・クラス・ローダが実装されることが推奨される。また実装は、コンテナの中に配備された各ウェブ・アプリケーションにとって、Thread.currentThread.getContextClassLoader()呼び出しはこの章で指定されている契約を実装した ClassLoader インスタンスを返さねば**ならない**。更に、その ClassLoader インスタンスは配備された各ウェブ・アプリケーション用のものとは別のインスタンスでなければ**ならない**。コンテナは何らかのコールバック(リスナのコールバックを含む)をする前、及び一旦そのコールバックが戻ったらオリジナルの ClassLoader にそれを戻す前に、上記のようにスレッド・コンテキスト ClassLoader をセットすることが要求される。

10.8 ウェブ・アプリケーションの置き換え（Replacing a Web Application）

サーバはそのコンテナを再起動させることなくあるアプリケーションを新しいバージョンで置き換えができるようになっていなければならない。あるアプリケーションが置き換えられたら、コンテナはそ

のアプリケーション内のセッション・データが維持されるようなしっかりした手段を提供しなければならない。

10.9 エラー処理（Error Handling）

10.9.1 要求の属性（Request Attributes）

ウェブ・アプリケーションはエラーが生じたときに、エラー応答のコンテント・ボディを提供するためにそのアプリケーション内の他のリソースが使われるよう指定できなければならない。これらのリソースの指定は配備記述子のなかでなされる。もしそのエラー・ハンドらの場所がサーブレットまたは JSP であったときは：

- そのコンテナが作ったオリジナルのラップされていない要求と応答のオブジェクトがそのサーブレットまたは JSP ページに渡される。
- 要求パスと属性たちがあたかもエラー・リソースへの RequestDispatcher.forward が実行されたかのごとくセットされる。
- 表 10-1 の要求属性がセットされねばならない。

表 10-1 要求の属性とその型

要求の属性	型
javax.servlet.error.status_code	java.lang.Integer
javax.servlet.error.exception_type	java.lang.Class
javax.servlet.error.message	java.lang.String
javax.servlet.error.exception	java.lang.Throwable
javax.servlet.error.request_uri	java.lang.String
javax.servlet.error.servlet_name	java.lang.String

これらの属性によりサーブレットは、ステータス・コード、例外の型、エラー・メッセージ、伝搬した例外オブジェクト、及びエラーが発生したサーブレットにより処理された要求の URI（getRequestURI 呼び出しで決まる）、及びエラーが発生したサーブレットの論理名、に基づいた特化したコンテントを発生できる。

本仕様の第 2.3 版で属性リストに例外オブジェクトが導入されたことで、例外の型とエラー・メッセージ属性が冗長なものになっている。これらはこの API の以前のバージョンとの後方互換性のために残されている。

10.9.1.1 エラー・ページ（Error Pages）

開発者たちがあるサーブレットがエラーを発生させたときにウェブのクライアントに返されるコンテントの体裁をカスタマイズできるように、配備記述子はエラー・ページの記述のリストを定めている。このシンタックスにより、リソースたちの構成が、特定のステータス・コードのための応答に関しsendErrorをサーブレットあるいはコンテナが呼び出した時、あるいはそのコンテナを伝搬する例外またはエラーをそのサーブレットが発生させたときのいずれかで、コンテナによって戻されるようになっている。

もしsendErrorメソッドがその応答で呼ばれるときは、そのコンテナはそのステータス・コードのシンタックスを使うウェブ・アプリケーションのためのエラー・ページ宣言のリストを調べ、一致するものを探す。もし一致するものがあれば、そのコンテナはその場所のエントリで示されたリソースを返す。

サーブレットまたはフィルタは要求処理中に以下の例外をスローしても良い:

- ランタイムの例外またはエラー
- ServletExceptionsまたはそのサブクラス
- IOExceptionsまたはそのサブクラス

ウェブ・アプリケーションはexception-type要素を使って宣言されたエラー・ページを持っても良い。この場合、コンテナはスローされた例外とexception-type要素を使うエラー・ページ定義のリストを比較して例外タイプの一致をとる。一致するものがあればそのコンテナは場所エントリのなかで示されたリソースを返す。クラス階層の中で最も近い一致が適用される。

クラス階層マッチを使って一致するexception-typeを含んでいるエラー・ページ宣言が無く、スローされた例外がServletExceptionまたはそのサブクラスの場合は、そのコンテナはServletException.getRootCauseメソッドで定められたようにラップされた例外を抽出する。エラー・ページ宣言たちで第二のパスがとられ、再度エラー・ページ宣言たちとの一致を試まれるが、この場合はラップされた例外が使われる。

配備記述子の中のexception-type要素を使ったエラー・ページ宣言はそのexception-typeのクラス名に関してユニークでなければならない。同じように、status-code要素を使ったエラー・ページ宣言はそのステータス・コードに関してその配備記述子の中でユニークでなければならない。

記述されたエラー・ページのメカニズムはRequestDispatcherまたはfilter.doFilterメソッドを使って呼び出されたときにエラーが生じるときには介在しない。このように、RequestDispatcherを使ったフィルタまたはサーブレットは発生したエラーを処理する機会を持つ。

あるサーブレットが上記のエラー・ページのメカニズムで処理されないエラーを発生させたときは、そのコンテナはステータス500の応答を確実に送信しなければならない。デフォルトのサーブレットとコンテナは4xx及び5xxのステータス応答を送信する為にsendErrorメソッドを使用し、そのエラーのメカニズムが呼ばれ得るようにする。デフォルトのサーブレットとコンテナは2xx及び3xxのステータス応答を送信する為にsetStatusメソッドを使用し、エラー・ページのメカニズムを呼ばない。

もしそのアプリケーションが第2.3.3.3章の「非同期処理」で記された非同期動作を使っているときは、総てのエラーがアプリケーションが作ったスレッドで処理されるようにするのはそのアプリケーションの責任である。コンテナはAsyncContext.startを介して出されたスレッドからのエラーの面倒をみても良い。AsyncContext.dispatch中に発生したエラーの処理に関しては同じ章の「そのディスパッチのメソッドたちの実行中に生じたエラーまたは例外は以下のようにそのコンテナによって捕捉されまた処理されねばならない」の個所を見られたい。

10.9.2 エラー・フィルタ（Error Filters）

エラー・ページのメカニズムはそのコンテナが作ったオリジナルのラップされていない / フィルタされていない要求と応答のオブジェクトたちで動作している。第 6.2.5 章の「フィルタと RequestDispatcher」で記されたメカニズムがエラー応答が発生される前に適用されるフィルタを指定するのに使えよう。

10.10 ウェルカム・ファイル（Welcome Files）

ウェブ・アプリケーションの開発者たちはそのウェブ・アプリケーションの配備記述子の中にウェルカム・ファイルと呼ばれる部分的 URI たちの順序づけられたリストを定義できる。このリストのための配備記述子のシンタックスはウェブ・アプリケーション配備記述子の図で記されている。

このメカニズムの目的は、ウェブの要素にマップされていない WAR のなかのディレクトリ・エントリに対応するある URI 向けの要求があるときにそのコンテナが URI たちに追加するために使う為の、順序づけられた部分的 URI たちのリストを配備者が指定できるようにすることである。この種の要求は有効な部分要求(valid partial request)として知られる。

この種の機能の使用は以下の共通的な事例で明確にされる: ‘index.html’のウェルカム・ファイルは host:port/webapp/directory/（ここでは‘directory’はサーブレットまたは JSP ページにマップされていない WAR のなかのエントリである）のような URL への要求が、‘host:port/webapp/directory/index.html’としてクライアントに戻される。

もしウェブ・コンテナが有効な部分的要求を受けるときは、そのウェブ・コンテナは配備記述子のなかで指定されたウェルカム・ファイルのリストを調べなければならない。このウェルカム・ファイル・リストは始まりまたは終わりに'/'が無い部分的 URL たちの順序づけられたリストである。そのウェブ・サーバは配備記述子で指定された順序で各ウェルカム・ファイルをその部分的要求に追加し、WAR のなかの静的なリソースがその要求 URI にマップされるかどうかをチェックしなければならない。もしマッチするものが見つからないときは、そのウェブ・サーバは再度配備記述子で指定された順序で各ウェルカム・ファイルをその部分的要求に追加し、そのサーブレットがその要求 URI にマップされているかをチェックしなければ**ならない**。そのウェブ・コンテナはその要求を WAR の中のマッチする最初のリソースに送らねばならない。そのコンテナはその要求をフォワード、リダイレクト、または直接要求から区別できないようなコンテナ固有のメカニズムでその要求をそのウェルカムのリソースに送信しても良い。

上記のやり方でマッチするウェルカム・ファイルが見つからないときは、そのコンテナはそれが適切の見出すやり方でその要求を処理してよい。ある設定においては、このことはディレクトリのリスティングを返すことを意味する、あるいは他の設定では 404 応答を返すことを意味する。

以下のようなウェブ・アプリケーションを考えてみる:

- 配備記述子が以下のようなウェルカム・ファイルたちをリストしている:

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

- WAR の中の静的コンテントは次のようになっている:

```
/foo/index.html
/foo/default.jsp
/foo/orderform.html
/foo/home.gif
/catalog/default.jsp
/catalog/products/shop.jsp
/catalog/products/register.jsp
```

- /foo という要求 URI は/foo/の URI にリダイレクトされる。
- /foo/という要求 URI は/foo/index.html として戻される。
- /catalog という要求 URI は/catalog/という URI にリダイレクトされる。
- /catalog/という要求 URI は/catalog/default.jsp として戻される。
- /catalog/index.html という要求 URI は 404 not found をもたらす
- /catalog/products という要求 URI は/catalog/products/の URI にリダイレクトされる。
- /catalog/products/という要求 URI はもしあれば“default”サーブレットに渡される。もし“default”サーブレットがマップされていないときは、その要求は 404 not found を引き起こすか、shop.jsp と register.jsp を含むディレクトリ・リスティングを引き起こすか、あるいはそのコンテナで決められている他の振る舞いを起こす。“default”サーブレットの定義に関しては第 12.2 章の「マッピングの仕様」を見られたい。
- 上記の静的コンテントの総てはまた JAR ファイルの META-INF/resources ディレクトリの中にパッケージされた上記でリストしたコンテントとともに JAR ファイルにパッケージ化され得る。その JAR ファイルは次にそのウェブ・アプリケーションの WEB-INF/lib ディレクトリの中に含める事ができる。

10.11 ウェブ・アプリケーションの環境（Web Application Environment）

Java EE 技術対応実装の一部ではないサーブレット・コンテナたちは第 15. 2.2 章の「ウェブ・アプリケーション環境と Java EE 仕様」で記されたアプリケーション環境機能を実装することが推奨されるが、これは要求されてはいない。もしそれらがこの環境をサポートするために必要な機能を実装していないときは、それらの機能に依存するアプリケーションの配備にあたっては、そのコンテナは警報を出さねばならない。

10.12 ウェブ・アプリケーションの配備（Web Application Deployment）

あるウェブ・アプリケーションがあるコンテナ内に配備されるときは、そのウェブ・アプリケーションがクライアント要求処理を始める前に、以下のステップがこの順で実施されねばならない。

- 配備記述子の<listener>要素で特定された各イベント・リスナのインスタンスをインスタンス化する。
- ServletContextListener を実装したインスタンス化されたリスナ・インスタンスにたいし、contextInitialized()メソッドを呼ぶ。
- 配備記述子の中の<filter>要素で特定された各フィルタのインスタンスをインスタンス化し、各フィルタ・インスタンスの init()メソッドを呼ぶ。
- load-onstartup 要素値で定められた順で<load-on-startup>要素を含む<servlet>要素で特定された各サーブレットのインスタンスをインスタンス化し、各サーブレット・インスタンスの init()メソッドを呼ぶ。

10.13 web.xml 配備記述子の包含（Inclusion of a web.xml Deployment Descriptor）

ウェブ・アプリケーションは、何らサーブレット、フィルタ、あるいはリスナ部品を持っていないとき、あるいは同じものを宣言するためのアノテーションを使っていないときは、web.xml を含むことは要求されて**いない**。言い換えると、静的ファイルと JSP ページたちのみを含むアプリケーションでは web.xml の提示は要求されていない。

第11章　　アプリケーションのライフサイクルのイベント（Application Lifecycle Events）

11.1　序説（Introduction）

アプリケーション・イベントの機能によりウェブ・アプリケーションの開発者は ServletContext 及び HttpSession 及び ServletRequest のライフサイクルにわたってのより大きなコントロールを持つことができ、より良いコードのファクトリ化(code factorization)ができ、そのウェブ・アプリケーションが使うリソースたちの管理効率を上げることができる。

11.2　イベント・リスナ（Event Listeners）

アプリケーション・イベント・リスナたちはひとつまたはそれ以上のサーブレット・イベント・リスナのインターフェイスを実装したクラスである。これらはそのウェブ・アプリケーションの配備時にそのウェブ・コンテナの中でインスタンス化され登録される。これらは開発者によって WAR のなかで用意される。

サーブレット・イベント・リスナたちは ServletContext、HttpSession、及び ServletRequest オブジェクトの中の状態変化のイベント通知をサポートしている。サーブレット・コンテキストのリスナたちはそのアプリケーションの JVM レベルで保持されているリソースまたは状態を管理するのに使われる。HTTP セッションのリスナたちは同じクライアントまたはユーザからのあるウェブ・アプリケーションになされる一連の要求にかかわる状態またはリソースを管理するのに使われる。サーブレット要求のリスナたちはサーブレット要求のライフサイクルにわたっての状態を管理するのに使われる。非同期リスナたちは非同期処理のタイム・アウトと完了のような非同期のイベントを管理するのに使われる。

各イベント・タイプに対し複数のリスナ・クラスが聞いていることもあり、その開発者は各イベント・タイプのためのリスナ・ビーンズをそのコンテナが呼び出す順序を指定しても良い。

11.2.1　イベントのタイプとリスナ・インターフェイス（Event Types and Listener Interfaces）

イベントのタイプとこれらの監視に使われるリスナ・インターフェイルを以下の表に示す:

表 11-1：サーブレット・コンテキスト・イベント

イベント・タイプ	記述	リスナ・インターフェイス

ライフサイクル	そのサーブレット・コンテキストが作られたばかりでその最初の要求をサービスするのに使用可能になった、あるいはそのサーブレット・コンテキストがシャット・ダウンしようとしている。	javax.servlet.ServletContextListener
属性への変更	そのサーブレット・コンテキストの属性が追加された、除去された、あるいは置き換えられた。	javax.servlet.ServletContextAttributeListener

表 11-2：HTTP セッション・イベント

イベント・タイプ	記述	リスナ・インターフェイス
ライフサイクル	HttpSession が生成された、無効化された、あるいはタイム・アウトを起こした。	javax.servlet.http.HttpSessionListener
属性への変更	HttpSession 上で属性が追加された、除去された、あるいは置き換えられた。	javax.servlet.httpHttpSessionAttributeListener
セッションの移行	HttpSession が能動化された、あるいは受動化された。	javax.servlet.httpHttpSessionActivationListener
オブジェクトのバインド	HttpSession にオブジェクトがバインドされた、あるいは HttpSession からオブジェクトのバインドが外された。	javax.servlet.httpHttpSessionBindingListener

表 11-3：サーブレット要求イベント

イベント・タイプ	記述	リスナ・インターフェイス
ライフサイクル	ウェブ・コンポネトたちにより、あるサーブレット要求の処理が開始された。	javax.servlet.ServletRequestAttributeListener
属性への変更	ServletRequest 上で属性が追加された、除去された、あるいは置き換えられた。	javax.servlet.ServletRequestAttributeListener
非同期イベント	非同期処理のタイムアウト、接続終了、あるいは完了。	javax.servlet.AsyncListener

この API の詳細に関しては API 参照を見られたい。

11.2.2 リスナの使用例（An Example of Listener Use）

このイベントのスキームの使用を示すために、あるデータベースを使用する一連のサーブレットたちを含むシンプルなウェブ・アプリケーションを考えてみよう。開発者はデータベース接続の管理のためのサーブレット・コンテキストのリスナを用意しているとする。

1. そのアプリケーションが起動するときは、そのリスナ・クラスは通知を受ける。そのアプリケーションはそのデータベースにログオンし、そのサーブレット・コンテキストにその接続をストアする。

2. そのアプリケーションの中のサーブレットたちはそのウェブ・アプリケーションが作動している間は必要に応じその接続にアクセスする。
3. そのウェブ・サーバがシャットダウンするとき、あるいはそのアプリケーションがウェブ・サーバから外されたときは、そのリスナ・クラスは通知を受け、データベース接続はクローズされる。

11.3 リスナ・クラスの設定（11.3 Listener Class Configuration）

11.3.1 リスナ・クラスの作動設定（Provision of Listener Classes）

ウェブ・アプリケーションの開発者は javax.servlet の API のひとつまたはそれ以上のリスナ・インターフェイスを実装したリスナのクラスを用意する。各リスナ・クラスは引数を持たないパブリックなコンストラクタを持たねばならない。これらのリスナ・クラスは WEB-INF/classes のアーカイブのエントリのもとで、または WEB-INF/lib ディレクトリの中の JAR のなかで、WAR の中にパッケージ化される。

11.3.2 配備宣言（Deployment Declarations）

リスナのクラスたちはそのウェブ・アプリケーションの配備記述子のなかで listner 要素を使って宣言される。これらはそれらに呼び出される予定の順序でクラス名でリストされる。他のリスナたちと違って、AsyncListener 型のリスナはプログラム的によってのみ（ServletRequest に）登録される。

11.3.3 リスナ登録（Listener Registration）

ウェブ・コンテナは、そのアプリケーションが最初の要求を処理する前に、各リスナ・クラスのインスタンスを生成しそれをイベント通知のために登録する。ウェブ・コンテナはそれらが実装しているインターフェイスに従いまた配備記述子のなかで出てくる順序でそのリスナを登録する。ウェブ・アプリケーションが実行中はリスナたちは自分たちの登録順に呼び出される。

11.3.4 シャットダウン時の通知（Notifications At Shutdown）

アプリケーションのシャットダウン時は、リスナたちは自分たちの宣言の順と逆順で、セッションのリスナたちへの通知がコンテキストのリスナたちへの通知に先行して、通知を受ける。セッションのリスナたちはコンテキストのリスナたちがアプリケーションのシャットダウンの通知を受ける前にセッションの無効化の通知を受けねばならない。

11.4 配備記述子の例（Deployment Descriptor Example）

以下の例は 2 つのサーブレット・コンテキスト・ライフサイクル・リスナたちとひとつの HttpSession のリスナを登録するための配備文法である。com.acme.MyConnectionManager と com.acme.MyLoggingModule の双方が javax.servlet.ServletContextListener を実装し、com.acme.MyLoggingModule はそれに加えて javax.servlet.http.HttpSessionListener を実装しているとしよう。また、その開発者は com.acme.MyConnectionManager がサーブレット・コンテキストのライフサイクル・イベントを com.acme.MyLoggingModule より前に通知を受けることを望んでいるとする。以下はこのアプリケーションの配備記述子である:

```
<web-app>
      <display-name>MyListeningApplication</display-name>
      <listener>
            <listener-
class>com.acme.MyConnectionManager</listenerclass>
      </listener>
      <listener>
            <listener-class>com.acme.MyLoggingModule</listener-
class>
      </listener>
      <servlet>
            <display-name>RegistrationServlet</display-name>
            ...etc
      </servlet>
</web-app>
```

11.5 リスナ・インスタンスとスレッド（Listener Instances and Threading）

コンテナはそのアプリケーションへの最初の要求を実行開始前にウェブ・アプリケーション内のリスナ・クラスたちのインスタンス化を完了することが要求されている。コンテナはそのウェブ・アプリケーションの最後の要求がサービスされるまで各インスタンスの参照を維持しなければならない。

ServletContext と HttpSession のオブジェクトたちの属性の変更は同時に起きても良い。コンテナは結果としての属性リスナ・クラスたちへの通知を同期化することは要求されていない。状態を維持しているリスナ・クラスたちはデータの完全性の責任を持ち、このケースを明示的に処理しなければならない。

11.6 リスナの例外（Listener Exceptions）

リスナ内のアプリケーション・コードは動作中に例外をスローしても良い。一部のリスナ通知はそのアプリケーションのなかの別の部品の呼び出しツリーのもとで生じる。これの事例はセッション属性をセットするサーブレットで、そのセッション・リスナが処理不能（unhandled）の例外をスローするときである。コンテナは処理できない例外は第 10.9 章「エラー処理」で記したエラー・ページのメカニズムで処理されるようにしなければならない。これらの例外に対しエラー・ページが指定されていないときは、そのコンテナはステータス・コード 500 で確実に応答を送り戻さねばならない。

一部の例外はそのアプリケーションの別の部品の呼び出しスタックのもとで生じない。これの例はタイム・アウトを起こしたという通知を受け処理できないという例外をスローする SessionListener、あるいはサーブレット・コンテキストの初期化の通知中に処理不能の例外をスローする ServletContextListener、あるいは要求オブジェクトの初期化または破壊の通知中に処理不能の例外をスローする ServletRequestListener である。この場合、開発者はこれらの例外を処理する機会を持たない。コンテナはそのウェブ・アプリケーションに対する総てのその後の要求に対しアプリケーション・エラーを示すために HTTP ステータス・コード 500 で応答しても良い。

リスナが例外を発生させた後でも通常の処理が起きることを望む開発者たちは、その通知のメソッド内で自分の例外を処理しなければならない。

11.7 分散コンテナ（Distributed Containers）

分散されたウェブ・コンテナたちの場合は、HttpSession のインスタンスの適用範囲はセッション要求をサービスしている特定の JVM となり、ServletContext のインスタンスの適用範囲はそのウェブ・コンテナの JVM である。分散されたコンテナたちはサーブレット・コンテきるとのイベントまたは HttpSession のイベントのいずれかを他の JVM たちに伝搬させることは要求されていない。リスナ・クラスのインスタンスたちの適用範囲は JVM あたりの配備記述子宣言あたりのひとつである。

11.8 セッションのイベント（Session Events）

リスナ・クラスたちは開発者にあるウェブ・アプリケーション内のセッションの追跡手段を提供している。これはしばしばセッション追跡の中で、そのセッションをコンテナがタイム・アウトにした為、あるいはそのウェブ・アプリケーション内のあるウェブ部品が invalidate メソッドを呼んだためにそのセッションが無効になったかどうかを知るのに有用である。この区別はリスナたちと HttpSession のメソッドたちを使って間接的に判断できよう。

第12章　　要求のサーブレットへのマッピング（Mapping Requests to Servlets）

この章で記されているマッピング技術が、ウェブ・コンテナがクライアント要求たちをサーブレットたちにマッピングする際に要求されている[1]。

1. この仕様の 2.5 版以前まではこれらのマッピング技術の使用は要求(requirement)ではなくて推奨（suggestion）であって、サーブレット・コンテナはクライアント要求をサーブレットにマッピングするのに自分たち固有のスキームを持つことが出来ていた。

12.1 URL パスの使用（Use of URL Paths）

クライアント要求を受けると、そのウェブ・コンテナはどのアプリケーションにそれを送るかを判断する。選択されたウェブ・アプリケーションはその要求 URL の最初にマッチする最長のコンテキスト・パスを持っていないければならない。その URL のマッチした部分がサーブレットにマッピングするときのコンテキスト・パスである。

ウェブ・コンテナは次に以下に示したパス・マッピング手順を使ってその要求を処理するサーブレットを見つけねばならない。

サーブレットへのマッピングに使われるパスは要求オブジェクトからの要求 URI からコンテキスト・パスとパス・パラメタを引いたものである。最初にマッチしたものが使われそれ以上のマッチングは試みられない：

1. そのコンテナはその要求のパスとサーブレットのパスとの完全な一致を見つようと試みる。マッチが成功したらそのサーブレットを選択する。
2. そのコンテナは最長パス・プレフィックスのマッチを繰り返し試みる。このことはパス・セパレータとしての’/’文字を使ってパス・ツリーを降りることでなされる。最長のマッチが選択サーブレットを決める。
3. もしその URL パスの最後のセグメントが拡張子（例えば.JSP）を含んでいるときは、そのサーブレット・コンテナはその拡張子のために要求を処理するサーブレットのマッチを取ろうと試みる。拡張子は最後の’.’文字の後の最後のセグメントの部分として定義される。
4. もし前記の 3 つの規則のいずれでもでサーブレット・マッチが取れないときはそのコンテナは要求されたリソースにふさわしいコンテントをサービスしようとする。もし「デフォルト」のサーブレットがそのアプリケーションのために定義されているときは、それが使われる。多くのコンテナはコンテントのサービス用に暗示的なデフォルト・サーブレットを用意している。

コンテナはマッチングのためには大文字小文字を分けた文字列比較を使わねばならない。

12.2 マッピングの仕様（Specification of Mappings）

ウェブ・アプリケーション配備記述子の中で、以下の文法がマッピングを規定するのに使われる:

- ‘/’文字で始まり‘/*’サフックスで終わる文字列がパス・マッピングのために使われる。
- ‘*.’プレフィックスで始まる文字列は拡張子マッピングに使われる。
- 空の文字列("")は特別の URL パタンで、正確にそのアプリケーションのコンテキスト・ルートにマップする、言い換えれば http://host:port/<contextroot>/の要求を意味する。この場合パス情報は’/’でサーブレット・パスとコンテキスト・パスは空の文字列(“”)である。
- ’/’文字のみを含む文字列はそのアプリケーションの「デフォルト」サーブレットを示す。この場合のサーブレット・パスは要求 URI からコンテキスト・パスを引いたもので、パス情報は null となる。
- その他の総ての文字列は正確なマッチングのためにのみ使われる。

12.2.1 暗示的マッピング（Implicit Mappings）

もしそのコンテナが内部 JSP コンテナを持っている場合は、*.jsp 拡張子はそれにマップされ、オン・デマンドで JSP ページの実行が出来る。このマッピングは暗示的マッピングと呼ばれる。もし *.jsp マッピングがそのウェブ・アプリケーションで決められているときは、そのマッピングは暗示的マッピングより優先する。サーブレット・コンテナは明示的なマッピングが優先する限り他の暗示的マッピングをつくることができる。例えば、*.shtml という暗示的なマッピングはそのサーバ上での機能を含むようマッピングできる。

12.2.2 マッピング・セット例（Example Mapping Set）

以下のようなマッピングを考えてみよう:

表 12-1 マップ・セット例

パス・パタン	サーブレット
/foo/bar/*	servlet1
/baz/*	servlet2
/catalog	servlet3
*.bop	servlet4

結果として以下のような振る舞いとなる:

表 12-2 マップ例に適用される到来パス

到来パス	要求処理サーブレット
/foo/bar/index.html	servlet1
/foo/bar/index.bop	servlet1

/baz	servlet2
/baz/index.html	servlet2
/catalog	servlet3
/catalog/index.html	「デフォルト」サーブレット
/catalog/racecar.bop	servlet4
/index.bop	servlet4

/catalog/index.html 及び/catalog/racecar.bop のケースでは、正しいマッチではないため“/catalog”にマップされたサーブレットが使われないことに注意されたい。

第13章　　セキュリティ（Security）

ウェブ アプリケーションは開発者たちによって作られる。開発者たちはそのアプリケーションを次に配備者に与えるとか売るとか、さもなければ移管し、ランタイム環境内にインストールされる。

アプリケーションの開発者たちは配備者たちへのセキュリティ要求と配備システムを伝える。この情報はそのアプリケーションの配備記述子、あるいはそのアプリケーションのコード内のアノテーションを使って宣言的に伝えられる。

本章ではサーブレット・コンテナのセキュリティのメカニズムとインターフェイス及びアプリケーションのセキュリティ要求を伝えるための配備記述子とアノテーション・ベースのメカニズムを記す。

13.1 序説（Introduction）

ウェブ・アプリケーションは多くのユーザがアクセスできるリソースたちを含んでいる。これらのリソースはしばしばインターネットのような保護されていないオープンなネットワークを通過する。そのような環境においては、相当な数のウェブ・アプリケーションがセキュリティ要求を持つことになろう。

品質保障と実装の詳細は異なっているだろうが、サーブレット・コンテナたちはこれらの要求を満たすためのメカニズムとインフラストラクチャを備えており、以下のような特徴を共通して持っている：

- 認証： 通信エンティティ同士が、互いに特定のエンティティに代わって動作していることを証明するためのメカニズム。
- リソースへのアクセス制御： 取得性、保全性、または秘匿性の適用実施のために、ユーザやプログラムの集まりに対して、リソースへの関与を制限するメカニズム。
- データ保全性： その情報が通過する間に、第 3 者によって変更されていないことを証明するメカニズム。
- 機密性またはデータのプライバシー： その情報がそれをアクセスする為にオーソライズされたユーザに対してのみ取得可能とするメカニズム。

13.2 宣言的セキュリティ（Declarative Security）

宣言型セキュリティは、ロール、アクセス制御、認証の要求事項などを含むアプリケーションのセキュリティのモデルまたは要求をそのアプリケーションの外部にフォームとして表記する手法をいう。ウェブ アプリケーションにおいて、配備記述子が宣言的セキュリティの主たる手段となる。

配備者は、そのアプリケーションの論理的なセキュリティの要求をランタイムの環境ごとに固有のセキュリティ ポリシーのある表現にマッピングする。ランタイム時は、サーブレット コンテナはそのセキュリティ ポリシーの表現を使って認証と認可（オーソライズ）を施行する。

このセキュリティ・モデルはそのウェブ・アプリケーションの静的コンテント部分に、及びクライアントから要求されているアプリケーション内のサーブレットとフィルタたちに適用される。このセキュリティ・モデルは、あるサーブレットが静的リソースをまたは forward または include を使ってサーブレットを呼び出すために RequestDispatcher を使っているときには適用されない。

13.3 プログラム的セキュリティ（Programmatic Security）

プログラム的なセキュリティは宣言的なセキュリティだけではそのアプリケーションのセキュリティ・モデルを十分に表現できないときにセキュリティ利用アプリケーションによって使われる。プログラム的なセキュリティは HttpServletRequest インターフェイスの以下のメソッドたちで構成される：

- authenticate
- login
- logout
- getRemoteUser
- isUserInRole
- getUserPrincipal

authenticate メソッドでアプリケーションはユーザ名とパスワードの収集ができる（フォーム・ベースのログインの代替的なものとして）。login メソッドでアプリケーションは制約なしの要求コンテキストの中からコンテナからの要求発信者の認証をさせることができる。

logout メソッドはアプリケーションがある要求の発信者アイデンティティをリセットするために用意されている。

getRemoteUser メソッドはコンテナにより、その要求に結び付けられたリモート・ユーザ（即ち発信者）の名前を返す。

isUserInRole メソッドはその要求に結び付けられているリモート・ユーザ（即ち発信者）が指定されたセキュリティ・ロールの中にいるかどうかを判断する。

getUserPrincipal メソッドはそのリモート・ユーザ（即ち発信者）のプリンシパル名を判断し、そのリモート・ユーザに対応した java.security.Principal オブジェクトを返す。getUserPrincipal で返された Principal 上の getName メソッド呼び出しはそのリモート・ユーザの名前を返す。この API によりサーブレットたちは取得した情報をもとにビジネス・ロジックの判断ができる。もし認証されたユーザがいないときは、getRemoteUser は null を返し、isUserInRole メソッドは常に false を返し、そして getUserPrincipal メソッドは null を返す。

isUserInRole メソッドは String のユーザ・ロール名パラメタを引数とする。securityrole-ref 要素はそのメソッドに渡されるロール名を含む role-name サブ要素を持って配備記述子の中で宣言されていなければならない。security-role-ref 要素はその値がそのユーザがマップされているかもしれないセキュリティ・ロールの名前である role-link サブ要素を含んでいないければならない。そのコンテナはこの呼び出しの戻り値を決めるときに security-role-ref の security-role へのマッピングを使う。

例えば、"FOO"というセキュリティ・ロールの参照を"manager"というロール名のセキュリティ・ロールにマップするときは、そのシンタックスは以下のようになる:

```
<security-role-ref>
      <role-name>FOO</role-name>
      <role-link>manager</role-link>
</security-role-ref>
```

このケースではもし"manager"セキュリティ・ロールに属するあるユーザが呼んだサーブレットがisUserInRole("FOO")なるAPI呼び出しを行ったときは、その結果はtrueとなる。

もしあるsecurity-role要素にマッチするsecurity-role-refが宣言されていないときは、そのコンテナはデフォルトとしてそのウェブ・アプリケーションの為のsecurity-role要素のリストに対しrole-name要素をチェックしなければならない。isUserInRoleメソッドはその発信者があるセキュリティ・ロールにマップされているか判断するのにこのリストを参照する。開発者はこのデフォルトのメカニズムを使うことは、呼び出しをするサーブレットの再コンパイルすることなくそのアプリケーションの中のロール名を変更するという柔軟性が制限するかもしれないことを注意しなければならない。

13.4 プログラム的なアクセス制御アノテーション（Programmatic Access Control Annotations）

本節はサーブレット・コンテナが執行するセキュリティ制約を設定するために用意されているアノテーションたちとAPIたちを定めている。

13.4.1 @ServletSecurityアノテーション（@ServletSecurity Annotation）

@ServletSecurityアノテーションは、そうでなければ可搬的な配備記述子のなかのsecurity-constraint要素を介して宣言的に表現されたか、あるいはServletRegistrationインターフェイスのsetServletSecurityメソッドを介してプログラム的に表現されていたであろうものと等価なアクセス制御制約を定義する為の代替的なメカニズムを提供している。サーブレットのコンテナはjavax.servlet.Servletインターフェイスを実装したクラスたち（及びそのサブクラスたち）上での@ServletSecurityアノテーション使用に対応しなければならない。

```
package javax.servlet.annotation;
@Inherited
@Documented
@Target(value=TYPE)
@Retention(value=RUNTIME)
      public @interface ServletSecurity {
      HttpConstraint value();
      HttpMethodConstraint[] httpMethodConstraints();
}
```

表 13-1：ServletSecurity インターフェイス

要素	記述	デフォルト
value	httpMethodConstraints で返される配列のなかで表現されていない総ての HTTP メソッドに適用される保護を定める HttpConstraint	@HttpConstraint
httpMethodConstraints	HTTP メソッド固有の制約の配列	{}

@HttpConstraint

@HttpConstraint アノテーションは@ServletSecurity アノテーションの中で使われ、それに対し対応する@HttpMethodConstraint が@ServletSecurity アノテーション内で生じない総ての HTTP プロトコル・メソッドに対し適用されるセキュリティ制約を表現する。

```
package javax.servlet.annotation;
@Documented
@Retention(value=RUNTIME)
public @interface HttpConstraint {
      ServletSecurity.EmptyRoleSemantic value();
      java.lang.String[] rolesAllowed();
      ServletSecurity.TransportGuarantee transportGuarantee();
}
```

表 13-2：HttpConstraint インターフェイス

要素	記述	デフォルト
value	rolesAllowed が空の配列を返すとき（のみ）適用されるデフォルトの認可（オーソライズ）のためのセマンティックス	PERMIT
rolesAllowed	認可されたロールたちの名前を含む配列	{}
transportGuarantee	要求が到来している接続で満足されねばならないデータ保護要求	NONE

@HttpMethodConstraint

@HttpMethodConstraint アノテーションは特定の HTTP プロトコルのメッセージ上のセキュリティ制約を表現する為に@ServletSecurity アノテーションの中で使われる。

```
package javax.servlet.annotation;
@Documented
@Retention(value=RUNTIME)
public @interface HttpMethodConstraint {
      ServletSecurity.EmptyRoleSemantic value();
      java.lang.String[] rolesAllowed();
      ServletSecurity.TransportGuarantee transportGuarantee();
```

```
}
```

表 13-3：HttpMethodConstraint インターフェイス

要素	記述	デフォルト
value	その HTTP メソッド名	
emptyRoleSemantic	rolesAllowed メソッドが空の配列を返すとき（のみ）適用されるデフォルトの認可のセマンティックス	PERMIT
rolesAllowed	認可されたロールたちの名前を含む配列	{}
transportGuarantee	要求が到来している接続で満足されねばならないデータ保護要求	NONE

@ServletSecurity アノテーションは（ターゲットとなる）Servlet 実装クラス上で指定され、その値は @Inherited メタ-アノテーションで定められた規則に従ってサブクラスたちによって引き継がれていても良い。たかだかひとつの@ServletSecurity のインスタンスがある Servlet 実装クラス上で生じ、その@ServletSecurity アノテーションは（ターゲットとなる）Java のメソッドで指定されていなければ**ならない**。

@ServletSecurity アノテーション内でひとつまたはそれ以上の@HttpMethodConstraint たちが指定されているときは、各@HttpMethodConstraint は@HttpMethodConstraint 内で指定されたその HTTP プロトコル・メソッドに適用される security-constraint を定める。対応する @HttpMethodConstraint が定義されているもの以外の総ての HTTP プロトコル・メソッドに対して適用される security-constraint を定義する@ServletSecurity の拡大は、@ServletSecurity アノテーション内に定義されている。

可搬配備記述子内で指定された security-constraint 要素たちはその制約内で生じる総ての url-patterns に対し認可（オーソライズ）可能である。

可搬配備記述子の中のある security-constraint が@ServletSecurity でアノテートされたあるクラスにマップされたパタンと正確に一致する url-pattern を含んでいるときは、そのアノテーションはそのパタンに対しサーブレット・コンテナが施行する制約に影響を与えてはならない。

可搬配備記述子の中で metadata-complete=true が定められているときは、@ServletSecurity アノテーションは配備記述子の中でアノテートされたクラスにマップされた url-patterns（マップされたどのサーブレット）のどれにも適用されない。

@ServletSecurity アノテーションは、そのサーブレットが ServletContext インターフェイスの createServlet メソッドでコンストラクトされていない限り、ServletContext インターフェイスの addServlet(String, Servlet)メソッドを使って生成された ServletRegistration の url-patterns には適用されない。

上記のリストの例外として、ある Servlet クラスが@ServletSecurity でアノテートされているとき、そのアノテーションはそのクラスにマップされた総てのサーブレットたちにマップされた総ての url-

patterns に適用されるセキュリティ制約を指定する。あるクラスが@ServletSecurity アノテーションでアノテートされていないときは、そのクラスからマップされたサーブレットに適用されるアクセス・ポリシーは、もしあれば対応する可搬配備記述子のなかの適用可能 security-constrain 要素によって確立される、あるいは ServletRegistration インターフェイスの setServletSecurity メソッドを介してターゲットのサーブレット用にプログラム的に確立されている制約によってそのような要素たちを禁止する。

13.4.1.1 例（Examples）

以下に ServletSecurity の使用例を示す。

コード例 13-1 総ての HTTP メソッドに対し制約なし

```
@ServletSecurity
public class Example1 extends HttpServlet {
}
```

コード例 13-2 総ての HTTP メソッドに対し、auth-constraint（認証制約）なし、秘密(confidential)のトランスポートが要求される

```
@ServletSecurity(@HttpConstraint(transportGuarantee =
TransportGuarantee.CONFIDENTIAL))
public class Example2 extends HttpServlet {
}
```

コード例 13-3 総ての HTTP メソッドに対し、総てのアクセスが拒否される

```
@ServletSecurity(@HttpConstraint(EmptyRoleSemantic.DENY))
public class Example3 extends HttpServlet {
}
```

コード例 13-4 総ての HTTP メソッドに対し、auth-constraint（認証制約）が Role R1 のなかのメンバーシップであることを要求

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "R1"))
public class Example4 extends HttpServlet {
}
```

コード例 13-5 GET と POST を除く総ての HTTP メソッドに対し、制約をかけない；GET と POST のメソッドたちに対しては、auth-constraint（認証制約）が Role R1 のなかのメンバーシップであることを要求；POST に対しては、秘密(confidential)のトランスポートが要求される

```
@ServletSecurity((httpMethodConstraints = {
      @HttpMethodConstraint(value = "GET", rolesAllowed = "R1"),
      @HttpMethodConstraint(value = "POST", rolesAllowed = "R1",
      transportGuarantee = TransportGuarantee.CONFIDENTIAL)
      }))
public class Example5 extends HttpServlet {
}
```

コード例 13-6 GETを除く総てのHTTPメソッドに対し、auth-constraint（認証制約）がRole R1のなかのメンバーシップであることを要求;GETに対しては制約なし

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "R1"),
      httpMethodConstraints = @HttpMethodConstraint("GET"))
public class Example6 extends HttpServlet {
}
```

コード例 13-7 TRACEを除く総てのHTTPメソッドに対し、auth-constraint（認証制約）がRole R1のなかのメンバーシップであることを要求;TRACEに対しては、総てのアクセスが拒否される

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "R1"),
      httpMethodConstraints = @HttpMethodConstraint(value="TRACE",
emptyRoleSemantic = EmptyRoleSemantic.DENY))
public class Example7 extends HttpServlet {
}
```

13.4.1.2 @ServletSecurityのsecurity-constraintへのマッピング（Mapping @ServletSecurity to security-constraint）

本節では@ServletSecurityアノテーションのsecurity-constraint要素のそれと等価な表現とのマッピングを記す。これはそのコンテナの既存のsecurity-constraint施行のメカニズムを使うことを促進するために用意されている。サーブレット・コンテナたちによる@ServletSecurityアノテーションの施行は、そのコンテナによる、この節で示されたマッピングからもたらされるsecurity-constraint要素たちの施行と実効的に等価でなければならない。

@ServletSecurityアノテーションはメソッド依存性のない@HttpConstraintを定義するのに使われており、ひとつまたはそれ以上の@HttpMethodConstraint指定が続く。このメソッド独立の制約は、HTTPメソッド固有の制約が指定されていない総てのHTTPメソッドに対し適用される。

@HttpMethodConstraint要素が含まれていないときは、@ServletSecurityアノテーションはhttp-method要素を含んでいない、したがって総てのHTTPメソッドたちに適切なweb-resource-collectionを含む単一のsecurity-constraint要素に対応している。

以下の例は単一のsecurity-constraint要素として@HttpMethodConstraintアノテーションを含んでいない@ServletSecurityアノテーション表現を示す。対応するサーブレット（登録）で定められたurl-pattern要素はweb-resource-collectionに含まれ、auth-constraintとuser-data-constraintを含むプレゼンスと値は第13.4.1.3節の「@HttpConstraintと@HttpMethodConstraintのXMLへのマッピング」で定められた@HttpConstraintの値のマッピングにより定まることになる。

コード例 13-8　@ServletSecurityの@HttpMethodConstraintなしへのマッピング

```
@HttpMethodConstraint
@ServletSecurity(@HttpConstraint(rolesAllowed = "R1"))
<security-constraint>
      <web-resource-collection>
            <url-pattern>...</url-pattern>
      </web-resource-collection>
```

```
      <auth-constraint>
            <security-role-name>Role1</security-role-name>
      </auth-constraint>
</security-constraint>
```

ひとつあるいはそれ以上の@HttpMethodConstraint 要素が指定されているときは、その method-independent は method-specific 制約のなかで名前を挙げられた HTTP メソッドの各々の http-method-omission 要素を含む web-resource-collection を含む単一の security-constraint に対応する。各@HttpMethodConstraint は対応する HTTP メソッドでの名前を指定した http-method を含む web-resource-collection を含む他の ecurity-constraint に対応する。

以下の例は単一の@HttpMethodConstraint が含まれる@ServletSecurity アノテーションの 2 つの security-constraint 要素たちへのマッピングを示す。対応するサーブレット（登録）で指定された url-pattern 要素たちが両制約の web-resource-collection に含められることになり、どの含まれている auth-constraint 及び user-data-constraint たちのプレゼンスと値は第 13.4.1.3 節の「HttpConstraint と @HttpMethodConstraint の XML へのマッピング」で示されているように結び付けられている @HttpConstraint と@HttpMethodConstraint のマッピングにより決まることになる。

コード例 13-9　含まれている@HttpMethodConstraint への@ServletSecurity のマッピング

```
@ServletSecurity(value=@HttpConstraint(rolesAllowed = "Role1"),
httpMethodConstraints = @HttpMethodConstraint(value = "TRACE",
emptyRoleSemantic = EmptyRoleSemantic.DENY))
<security-constraint>
      <web-resource-collection>
            <url-pattern>...</url-pattern>
            <http-method-omission>TRACE</http-method-omission>
      </web-resource-collection>
      <auth-constraint>
            <security-role-name>Role1</security-role-name>
      </auth-constraint>
</security-constraint>
<security-constraint>
      <web-resource-collection>
            <url-pattern>...</url-pattern>
            <http-method>TRACE</http-method>
      </web-resource-collection>
      <auth-constraint/>
</security-constraint>
```

13.4.1.3　@HttpConstraint と@HttpMethodConstraint の XML へのマッピング（Mapping @HttpConstraint and @HttpMethodConstraint to XML）

本節では@HttpConstraint と@HttpMethodConstraint のアノテーションの値たち（@ServletSecurity 内の仕様のために定義された）のそれに対応する auth-constraint 及び user-data-constraint 表現へのマッピングを記す。これらのアノテーションは可搬的配備記述しないで使われる auth-constraint

及び user-data-constraint 要素たち等価な表現をする為のある共通のモデルを共有している。このモデルは以下の 3 つの要素で構成されている:

- emptyRoleSemantic
  rolesAllowed で指名されたロールが存在しないときに適用される PERMIT（許可）または DENY（拒否）の認可セマンティック。この要素のデフォルト値は PERMIT で、DENY は空でない rolesAllowed リストと組み合わせてはサポートされない。
- RolesAllowed
  認可されたロールたちの名前を含むリスト。このリストが空のときは、その意味は emptyRoleSemantic の値に依存する。“*”というロール名は許されたロールたちのリストのなかに含まれているときは特に意味を持たない。この要素のデフォルト値は空のリストである。
- TransportGuarantee
  NONE（なし）または CONFIDENTIAL（秘密）のどちらかであるデータ保護要求で、要求が到来した接続上でこれは満足されねばならない。この要素は対応した値を持った transport-guarantee を含む user-data-constraint と等価な意味を持つ。この要素のデフォルト値は NONE である。

以下の例は上記の@HttpConstraint モデルと web.xml 内の auth-constraint 及び user-data-constraint 要素たち間の対応を示す:

コード例 13-10

```
emptyRoleSemantic=PERMIT, rolesAllowed={}, transportGuarantee= NONE
```

制約なし。

コード例 13-11

```
emptyRoleSemantic=PERMIT, rolesAllowed={}, transportGuarantee=
CONFIDENTIAL
<user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

コード例 13-12

```
emptyRoleSemantic=PERMIT, rolesAllowed={Role1},
transportGuarantee=NONE
<auth-constraint>
      <security-role-name>Role1</security-role-name>
</auth-constraint>
```

コード例 13-13

```
emptyRoleSemantic=PERMIT, rolesAllowed={Role1},
transportGuarantee=CONFIDENTIAL
<auth-constraint>
      <security-role-name>Role1</security-role-name>
</auth-constraint>
<user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

コード例 13-14

```
emptyRoleSemantic=DENY, rolesAllowed={}, transportGuarantee= NONE
<auth-constraint/>
```

コード例 13-15

emptyRoleSemantic=DENY, rolesAllowed={}, transportGuarantee= CONFIDENTIAL

```
<auth-constraint/>
<user-data-constraint>
     <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

13.4.1.4 ServletRegistration.Dynamic の setServletSecurity（setServletSecurity of ServletRegistration.Dynamic）

SetServletSecurity メソッドは、ある ServletRegistration のために定義されたマッピングに対し適用されるセキュリティ制約たちを定義する為に ServletContextListener 内で使うことが出来る。

```
Collection<String> setServletSecurity(ServletSecurityElement arg);
```

setServletSecurity への javax.servlet.ServletSecurityElement 引数は@ServletSecurity アノテーションの ServletSecurity インターフェイスにその構造とモデルで相似している。従って、第 13.4.1.2 節「@ServletSecurity のセキュリティ制約へのマッピング」で定められたマッピングが、HttpConstraintElement 及び HttpMethodConstraintElement の値を含む SecurityConstraintElement のその等価な security-constraint 表現へのマッピングに相似的に適用される。

setServletSecurity メソッドは可搬配備記述子のなかで既に security-constraint 要素のターゲットとなっている（従ってその呼び出しで影響されなかった）URL パタンのセット（空の可能性あり）を返す。このメソッドは ServletRegistration が取得された ServletContext が既に初期化されている場合には IllegalStateException をスローする。可搬配備記述子のなかで security-constraint が ServletRegistration でマップされたパタンと正確にマッチする url-pattern を含んでいるときは、その ServletRegistration 上での setServletSecurity 呼び出しはそのパタンでサーブレット・コンテナが施行する制約に影響を与えてはならない。

上記を除きまたそのサーブレット・クラスが@ServletSecurity でアノテートされている時を含め、ある ServletRegistration 上で setServletSecurity が呼ばれるときは、その呼び出しはその登録の url-patterns に適用されるセキュリティ制約を確立する。

13.5 ロール（Roles）

セキュリティのロール（role）というのは、アプリケーション開発者またはアセンブラが定めたユーザの論理グルーピングのことである。そのアプリケーションが配備（デプロイ）されたら、これらのロールは配備者により、ランタイムの環境下に、プリンシパルたちとかグループたちなどのセキュリティのエンティティにマッピングされる。

到来した要求に結び付けられたプリンシパルへの宣言型またはプログラム的なセキュリティを、呼び出しプリンシパルのセキュリティの属性にもとづき、サーブレットは実施する。このことは以下の手段たちのどちらかで起きうる：

1. 配備者がセキュリティのロールをその運用環境においてあるユーザ グループにマッピングするとき。呼び出しプリンシパルが属しているユーザ グループが、そのセキュリティの属性から検索される。そのプリンシパルのユーザ グループが運用環境においてそのセキュリティ ロールがマッピングされているユーザ グループと一致するときにのみ、そのプリンシパルはそのセキュリティ ロールの中にいることになる。
2. 配備者があるセキュリティ ロールをあるセキュリティ ポリシー ドメインのプリンシパル名にマッピングしたときは、その呼び出しプリンシパルのプリンシパル名がそのセキュリティ属性から検索される。そのセキュリティ ロールにマッピングされているプリンシパルと同じプリンシパルのときのみ、呼び出しプリンシパルはそのセキュリティ ロールの中にいる。

13.6 認証（Authentication）

ウェブ・クライアントは以下のメカニズムの一つを使ってあるウェブ・サーバに対しあるユーザを認証できる：

- HTTP ベーシック認証 (HTTP Basic Aythentication)
- HTTP ダイジェスト認証 (HTTP Digest Authentication)
- HTTS クライアント認証 (HTTPS Client Authentication)
- フォーム ベース認証 (Form Based Authentication)

13.6.1 HTTP ベーシック認証（HTTP Basic Authentication）

ユーザ名(username)とパスワード(password)をベースにした HTTP ベーシック認証は、HTTP/1.0 仕様で規定されている認証のメカニズムである。ウェブ・サーバはウェブ・クライアントに対しそのユーザの認証を要求する。その要求の一部として、そのウェブ・サーバはそのユーザが認証されることになるレルム（realm：文字列）を渡す。そのウェブ・クライアントはそのユーザからのユーザ名とパスワードを取得し、それらをそのウェブ・サーバに送信する。そのウェブ・サーバは次に指定されたレルム内でそのユーザを認証する。

ベーシック認証は安全な認証プロトコルではない。ユーザのパスワードはシンプルな base64 エンコーディングで送信され、ターゲットとなるサーバは認証を受けない。更なる保護がこれらの危惧の

一部を緩和できる：安全なトランスポートのメカニズム(HTTS)、あるいはネットワーク・レベルでのセキュリティ（IPSEC プロトコルまたは VPN の戦略のような）が一部の配備のシナリオで適用される。

13.6.2 HTTP ダイジェスト認証（HTTP Digest Authentication）

HTTP ベーシック認証と同じように、HTTP ダイジェスト認証はユーザ名とパスワードに基づいてあるユーザを認証する。しかしながら HTTP ベーシック認証とは違って、HTTP ダイジェスト認証ではクライアントはそのパスワード（及び追加データ）の反方向暗号化ハッシュを送信する。パスワードは接続上で送信されないものの、HTTP ダイジェスト認証では認証サーバが期待されるダイジェストを計算することで受信した認証者を認証できるよう認証コンテナが取得できるクリアなテキストのパスワード等価物[1]が必要になる。サーブレット・コンテナたちは HTTP_DIGEST 認証に対応していなければならない。

1. パスワード等価物としては、特定のレルムでユーザとして認証する為だけに使われるものがある。

13.6.3 フォーム・ベースの認証（Form Based Authentication）

ウェブ・ブラウザの組み込み認証メカニズムでは「ログイン画面」の見た目と感じを買えることは出来ない。本仕様では開発者がログイン画面の見た目と感じをコントロールできる必要とされるフォーム・ベースの認証メカニズムを導入している。

ウェブ・アプリケーションの配備記述子にはログインのフォームとエラー・ページのエントリたちが含まれている。このログイン・フォームはユーザ名とパスワード入力用のフィールドを含んでいなければならない。これらのフィールドは各々 j_username 及び j_password という名前が付けられていなければならない。あるユーザがある保護されたウェブのリソースにアクセスを試みるときは、そのコンテナはユーザの認証をチェックする。もしそのユーザが認証されていてそのリソースにアクセスする認可（オーソリティ）を所有しているときは、要求されたウェブのリソースは起動されそれへの参照が返される。

もしそのユーザが認証されていないときは、以下のステップの全部が起きる：

1. そのセキュリティ制約に結び付けられたログイン・フォームがそのクライアントに送信され、その認証のトリガとなった URL パスがそのコンテナによってストアされる。
2. そのユーザはユーザ名とパスワードのフィールドを含むそのフォームを埋めるよう求められる。
3. そのクライアントはそのフォームをサーバに対しポストで返す。
4. そのコンテナはそのフォームからの情報を使ってそのユーザの認証を試みる。
5. もし認証に失敗すれば、フォワードまたはリダイレクトのどちらかを使って、その応答のステータス・コードに 200 がセットされて、エラー・ページが返される。
6. もし認証に成功すれば、認証されたそのユーザのプリンシパルがチェックされ、そのユーザがそのリソースをアクセスする認可されたロール内にいるかどうかがチェックされる。

7. もしそのユーザが認可されれば、そのクライアントはストアされたURLパスを使ってそのリソースにリダイレクトされる。

認証されないユーザに送信されるエラー・ページはその失敗に関する情報が含まれる。

フォーム・ベース認証はユーザのパスワードはシンプルなbase64エンコーディングで送信され、ターゲットとなるサーバは認証を受けないので、ベーシック認証と同じく安全な認証プロトコルではない。更なる保護がこれらの危惧の一部を緩和できる:安全なトランスポートのメカニズム(HTTS)、あるいはネットワーク・レベルでのセキュリティ(IPSECプロトコルまたはVPNの戦略のような)が一部の配備のシナリオで適用される。

HttpServletRequestインターフェイスの認証のメソッドはそのログイン画面の見た目と感じをコントロールする為の代替的手段をアプリケーションに提供している。

13.6.3.1 ログイン・フォームの注意(Login Form Notes)

フォーム・ベースのログインとURLベースのセッション追跡は実施時は問題を起こし得る。フォーム・ベースのログインはセッションがクッキーまたはSSLセッション情報で維持されているときにのみ使われるべきである。

認証が適正に続けられる為に、ログイン・フォームのアクションは常にj_security_checkでなければならない。この制約は、どんなリソース向けでもログインのフォームが機能し、外向きのフォームのアクション・フィールドを指定することをサーバに要求することを回避する為に与えられている。

これは如何にこのフォームがHTMLページにコード化されるかを示したものである:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

もしあるHTTP要求が理由でフォーム・ベースのログインが呼び出されたときは、もし認証が成功したら、その呼び出しを要求されているリソースにリダイレクトするのに使われるように、オリジナルの要求パラメタたちはそのコンテナによって保存されねばならない。もしそのユーザがフォーム・ログインを使っていてHTTPセッションが作られてしまっているときは、そのセッションのタイムアウトまたは無効化(invalidation)は、そのユーザのログアウトをもたらし、その後に続く要求に対しユーザは際認証が要求されることになりる。ログアウトの適用範囲は認証のそれと同じである:例えば、もしそのコンテナがウェブ・コンテナ対応のJava EEのような単一サインオンをサポートしているときは、ユーザはそのウェブ・コンテナ上のどのウェブ・アプリケーションにも再認証が必要になる。

13.6.4 HTTPSクライアント認証(HTTPS Client Authentication)

HTTP（SSL 上の HTTP）を使ったエンド・ユーザ認証は強力な認証メカニズムである。このメカニズムではクライアントが PKC（Public Key Certificate：公開鍵認定）を持っている必要がある。現在、PKC は電子商取引のアプリケーションでは有用なものであり、またブラウザ内からのシングル・サインオンにも有用である。

13.6.5 更なるコンテナによる認証メカニズム（Additional Container Authentication Mechanisms）

サーブレット・コンテナたちは配備されたアプリケーションに代わってそのコンテナが使う為の更なる HTTP メッセージ層の認証メカニズムの統合と設定に使えるパブリックなインターフェイスを用意しなければならない。これらのインターフェイスはコンテナのメーカ以外のパーティたち（アプリケーションの配備者、システム管理者、及びシステム・インテグレータたちを含む）が使うためにオファーされねばならない。

更なるコンテナの認証メカニズムの可搬的実装と統合を促進するために、総てのサーブレットがコンテナたちのための Java 認証 SPI のサーブレット・コンテナ・プロファイル（即ち JSR 196）を実装することが勧告される。この SPI は次のアドレスからダウンロードできる：
http://www.jcp.org/en/jsr/detail?id=196

13.7 認証情報のサーバによる追跡（Server Tracking of Authentication Information）

ランタイム環境の中でロールたちがマップされる基になるセキュリティのアイデンティティたち（ユーザたちとグループたちのような）はアプリケーション固有というよりは環境固有であるので、以下のことが好ましい：

1. ログインのメカニズムとポリシーをそのウェブ・アプリケーションが配備されている環境のプロパティとする、
2. 同じ環境に配備されている総てのアプリケーションがあるプリンシパルを代表する為の同じ認証情報を使えるようにする、及び
3. セキュリティ・ポリシー・ドメインの境界がまたがったときのみユーザたちの再認証を要求する

従って、サーブレットはコンテナ・レベルで（ウェブ・アプリケーション・レベルではなくて）認証情報を追跡することが要求される。これにより同じセキュリティ・アイデンティティに対し許されたコンテナによって管理されている他のリソースたちをひとつのウェブ・アプリケーションがアクセスするのにユーザ認証が出来るようになる。

13.8 セキュリティ制約の指定（Specifying Security Constraints）

セキュリティの制約というのはウェブのコンテントの保護を定義するための宣言的な手段である。あるセキュリティ制約は認可及び / あるいはユーザ・データ制約をウェブ・リソース上の HTTP 操作に結び付ける。配備記述子の中で security-constraint として表現されるセキュリティ制約は以下の要素たちで構成される:

- ウェブ・リソース・コレクション（配備記述子内の web-resource-collection）
- 認可の制約（配備記述子の中の auth-constraint ）
- ユーザ・データ制約（配備記述子の中の user-data-constraint ）

セキュリティ制約が適用される HTTP 操作とウェブのリソースたちはひとつあるいはそれ以上のウェブ・リソース・コレクションで特定される。ウェブ・リソース・コレクションは以下の要素たちで構成される:

- URL パタン（配備記述子の中の url-pattern）
- HTTP メソッド（配備記述子の中の http-method または http-method-omission 要素）

認可制約は、認証の要求を確立し、また制約された要求を実施することが許可された認可のためのロールたちを指名する。ユーザは制約された要求を実施する為に許可される指名されたロールたちの少なくともひとつのメンバーでなければならない。"*"という特別なロール名は配備記述子の中で指定された総てのロール名たちの簡略表記法である。ロールが指名されていない認可の制約は、その制約された要求へのアクセスはどんな状況においても許されないことを意味する。認可制約は以下の要素で構成される:

- ロール名（配備記述子内の role-name ）

ユーザ・データ制約は、その制約された要求が保護されたトランスポート層接続上で受信されねばならないという要求を確立する。要求された保護の強度はそのトランスポート保障の値で定義される。INTEGRAL というトランスポート保障はコンテントの完全性の要求を確立するのに使われ、CONFIDENTIAL というトランスポート保障は秘密性の要求を確立するのに使われる。“NONE”というトランスポート保障はそのコンテナは保護されていないものを含むどの接続上でも受信した制約された要求を受け付けなばならないことを示す。ユーザ・データ制約は以下の要素で構成される:

- トランスポート保障（配備記述子内の transport-guarantee）

ある要求に認可制約が適用されないときには、そのコンテナはユーザに認証を要求することなくその要求を受け付けねばならない。ある要求に対しユーザ・データ制約が適用されないときは、そのコンテナは保護されていないもを含むどの接続上ででも受信した要求を受け付けねばならない。

13.8.1 制約の組み合わせ（Combining Constraints）

制約たちを組み合わせるという目的のために、コレクションのなかに HTTP メソッドが指名されていない、あるいはそのコレクションが特に制約された http-method 要素のなかでその HTTP メソッドを指名している、あるいはそのコレクションがひとつあるいはそれ以上の http-method-omission 要素を含んでいてそのどれもが HTTP メソッドを指名していないときに、HTTP メソッドはある web-resource-collection の中で生じると言われる。

あるURLパタンとHTTPメソッドのペアが複数のセキュリティ制約の組み合わせ（言い換えるとweb-resource- collection内）で起きるときは、その制約（パタンとメソッドで）は個々の制約の組み合わせで定義される。同じパタンと制約が生じる制約たちの組み合わせのルールは以下のようである：

ロールたちを指名するあるいは"*"名でロールたちを暗示する認可制約の組み合わせは、許可されたロールとして個々の制約のなかでのロール名のユニオンをもたらさねばならない。認可制約を含まないセキュリティ制約は、その名前または暗示ロールが認証なしのアクセスを可能とするために、認可制約と組み合わせねばならない。ロールを指名しない認可制約の特別なケースは、その効果をオーバライドしアクセスが排除されるために、他の制約と組み合わせねばならない。

共通のurl-patternとhttp-methodに適用するuser-data-constraintsの組み合わせは、受け入れ可能な接続のタイプとして個々の制約で受け入れられる接続のタイプのユニオンをもたらさねばならない。user-data-constraintを含まないセキュリティ制約は、他のuser-data-constraintと組み合わせて、保護されない接続タイプが受け入れられる接続タイプとなるようにしなければならない。

13.8.2 例（Example）

以下の例は制約たちの組み合わせと、それらを提供される制約の表への変換を示したものである。配備記述子が以下のようなセキュリティ制約を含んでいるとしよう：

```
<security-constraint>
      <web-resource-collection>
            <web-resource-name>precluded methods</web-resource-name>
            <url-pattern>/*</url-pattern>
            <url-pattern>/acme/wholesale/*</url-pattern>
            <url-pattern>/acme/retail/*</url-pattern>
            <http-method-exception>GET</http-method-exception>
            <http-method-exception>POST</http-method-exception>
      </web-resource-collection>
      <auth-constraint/>
</security-constraint>

<security-constraint>
      <web-resource-collection>
            <web-resource-name>wholesale</web-resource-name>
            <url-pattern>/acme/wholesale/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>PUT</http-method>
      </web-resource-collection>
      <auth-constraint>
            <role-name>SALESCLERK</role-name>
      </auth-constraint>
</security-constraint>

<security-constraint>
      <web-resource-collection>
```

```
            <web-resource-name>wholesale 2</web-resource-name>
            <url-pattern>/acme/wholesale/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
            <role-name>CONTRACTOR</role-name>
      </auth-constraint>
      <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
      </user-data-constraint>
</security-constraint>

<security-constraint>
      <web-resource-collection>
            <web-resource-name>retail</web-resource-name>
            <url-pattern>/acme/retail/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
            <role-name>CONTRACTOR</role-name>
            <role-name>HOMEOWNER</role-name>
      </auth-constraint>
</security-constraint>
```

この仮想的な配備記述子は表 13-4 で示されるような制約をもたらすことになる。

表 13-4：セキュリティ制約表

URL パタン	HTTP メソッド	許可されるロールたち	サポートされる接続タイプ
/*	GET、POST を除く総てのメソッド	アクセス不能	制約なし
/acme/wholesale/*	GET、POST を除く総てのメソッド	アクセス不能	制約なし
/acme/wholesale/*	GET	CONTRACTOR SALESCLERK	制約なし
/acme/wholesale/*	POST	CONTRACTOR	CONFIDENTIAL
/acme/retail/*	GET、POST を除く総てのメソッド	アクセス不能	制約なし
/acme/retail/*	GET	CONTRACTOR HOMEOWNER	制約なし
/acme/retail/*	POST	CONTRACTOR HOMEOWNER	制約なし

13.8.3 要求の処理（Processing Requests）

サーブレット・コンテナがある要求を受信したら、そのコンテナは第 12.1 節「URL パスの使用」で定められたアルゴリズムを使い、要求された URI の最も良くマッチする url-pattern で定められている（もしあれば）制約を選択しなければならない。選択される制約が無い場合は、そのコンテナはその要求を受け付けねばならない。そうでない場合はそのコンテナはその要求の HTTP メソッドが選択されたパタンで制約されていないかどうかを判断しなければならない。そうでなければ、その要求は受け付けられねばならない。それ以外の場合はその要求はその url-pattern でその HTTP メソッドに適用される制約を満足しなければならない。その要求が受け付けられそれに対応するサーブレットに回される為には以下のルールの双方が満足されねばならない：

1. その要求が受信された接続の特性は、それらの制約で定められたサポートされる接続タイプの少なくともひとつを満たさねばならない。このルールが満たされないときは、そのコンテナはその要求を拒否しそれを HTTPS ポート[2]にリダイレクトしなければならない。

   [2]：最適化のために、コンテナがアクセスが最終的に受け付けられない（ロールを指名していない認可制約によって）ことが分かる場合は、そのコンテナは禁止(forbidden)されているとしてその要求を拒否し、403 ステータス・コード(SC_FORBIDDEN)を返さねばならない。

2. その要求の認証特性はその制約で定められている認証とロールの要求たちを満足しなければならない。アクセスが受け付けられなかった（ロールを指名していない認可制約によって）為にこのルールが満足されない場合は、その要求は禁止(forbidden)されているとしてその要求を拒否され、403 ステータス・コード(SC_FORBIDDEN)が返されねばならない。もし許可されたロールに対しアクセスが制約されており、その要求が認証されていないときは、その要求は認可されないとして拒否され、401（SC_UNAUTHORIZED）ステータス・コードが返され、認証を引き起こさねばならない。もしアクセスが許可されたロールたちにたいし制約されていてその要求の認証のアイデンティティがこれらのロールのメンバーでない場合は、その要求は禁止(forbidden)されているとしてその要求を拒否され、403 ステータス・コード(SC_FORBIDDEN)がそのユーザに返されねばならない。

13.9 デフォルトのポリシー（Default Policies）

デフォルトとしては、リソースたちのアクセスには認証は必要とされない。認証は、その要求に最も良くマッチする url-pattern を含む（もしあれば）セキュリティ制約が、そのその要求の HTTP メソッドでの auth-constraint（ロールの指名）を課すことと組み合わされているときに、要求される。同じように、保護されたトランスポートは、その要求に適用されるセキュリティ制約がその要求の HTTP メソッド上で user-data-constraint を課す（保護された transport-guarantee で）と組み合わさっていない限り、要求されない。

13.10 ログインとログアウト（Login and Logout）

コンテナは、その要求をサーブレット・エンジンにディスパッチする前に要求の発信者のアイデンティティを確立する。その発信者アイデンティティはその要求の処理にわたって、あるいはそのアプリケーションがその要求での認証、ログイン、あるいはログアウトを成功裏に呼ぶまでは変更されないで留まる。非同期要求に対しては、初期ディスパッチ時に確立された発信者アイデンティティは、全体の要求の処理が関慮するまで、あるいはそのアプリケーションが成功裏にその要求での認証、ログイン、あるいはログアウトを呼ぶまでは、変更されないままで留まる。

ある要求処理中にあるアプリケーションにログインしているということは、その要求上で getRemoteUser または getUserPrincipal を呼ぶことで決まるであろうその要求に結び付けられた有効な非 null の発信者アイデンティティであるということと正確に対応している。これらのメソッドのどちらからかの null の戻り値は、その要求処理に関してはその発信者がまだそのアプリケーションにログインしていないことを意味する。

コンテナたちはログイン状態を追跡するために HTTP Session オブジェクトを生成できる。もしある開発者がユーザが認証されていないのにセッションを生成し、そしてそのコンテナが次にそのユーザを認証するときは、ログイン後開発者のコードにとって可視なセッションは、セッション情報の損失が無いように、ログイン前に作られたと同じセッション・オブジェクトでなければならない。

第14章　　配備記述子（Deployment Descriptor）

本章は配備記述子のウェブ・コンテナ対応のための Java™ サーブレット・バージョン 3.0 の要求事項を規定する。この配備記述子は開発者たち、アプリケーションのアセンブラたち、及び配備者たち間で、あるウェブ・アプリケーションの要素と設定の情報を伝達する。

Java Servlets v.2.4 あるいはそれ以降のバージョンでは、配備記述子は XML スキーマ・ドキュメントに関して規定されている。

本 API の 2.2 バージョンに書かれているアプリケーションのバックワード互換性のために、ウェブ・コンテナたちはまた配備記述子の 2.2 バージョンの対応も要求されている。この API の 2.3 バージョンに対し書かれているアプリケーションとのバックワード互換性のために、ウェブ・コンテナたちはまた配備記述子の 2.3 バージョンの対応も要求されている。2.2 バージョンは http://java.sun.com/j2ee/dtds/web-app_2_2.dtd から取得でき、また 2.3 バージョンは http://java.sun.com/dtd/web-app_2_3.dtd から取得できる。

14.1 配備記述子の要素たち（Deployment Descriptor Elements）

以下のタイプの設定と配備の情報が、総てのサーブレット・コンテナたち向けにウェブ配備記述子のなかでサポートされることが要求されている：

- ServletContext の初期化パラメタたち
- セッションの設定
- サーブレット宣言
- サーブレット・マッピングたち
- アプリケーションのライフサイクル・リスナ・クラスたち
- フィルタの定義とフィルタのマッピングたち
- MIME タイプのマッピングたち
- ウェルカム・ファイル・リスト
- エラー・ページたち
- ロケールとエンコーディングマッピングたち
- ogin-config、security-constraint、security-role、security-role-ref、及び run-as を含むセキュリティの設定

14.2 配備記述子の処理のためのルール（Rules for Processing the Deployment Descriptor）

本節ではあるウェブ・アプリケーションの配備記述子の処理に関してウェブのコンテナたちと開発者たちが注意しなければならない幾つかの一般的ルールをリストアップする。

- ウェブ・コンテナたちは配備記述子の中のテキスト・ノードたちの要素コンテントにたいし、前後につけられた XML 1.0 (http://www.w3.org/TR/2000/WD-xml-2e-20000814) で“S(white space)”として定義されたホワイトスペース（訳者注：スペース、タブ、及び空白行等）を除去しなければならない。
- その配備記述子はそのスキーマに対し有効なものでならねばならない。ウェブ・アプリケーションたちを操作するウェブ・コンテナとツールたちは、ある WAR の有効性をチェックする為の広範なオプションを持っている。これらには内部に持っている配備記述子ドキュメントの有効性のチェックが含まれる。加えて、ウェブ・アプリケーションたちを操作するウェブ・コンテナとツールたちはあるレベルのセマンティックなチェックを用意することが推奨される。例えば、あるセキュリティ制約のなかで参照されているあるロールが配備記述子の中で定義されているセキュリティのロールたちのひとつと同じかどうかをチェックすべきである。非適合ウェブ・アプリケーションの場合は、ツールたちとウェブ・コンテナたちはその開発者に記述的エラー・メッセージを伝えるべきである。ハイエンドのアプリケーション・サーバのメーカたちは、コンテナとは分離したツールの形でこの種の有効性チェックを提供することが推奨される。
- web-app のもとでのサブ要素たちは本仕様のこのバージョンでは順不同で良い。XML スキーマの制約ゆえに、distributable、session-config、welcome-file-list、jsp-config、login-config、及び locale-encoding-mapping-list の要素たちの多重性は「オプショナル」から「0 またはそれ以上」と変更されている。配備記述子がひとつ以上の session-config、jsp-config、及び login-config の要素を含むときは記述的なエラー・メッセージをその開発者に知らせねばならない。複数の発生があるときは elcome-file-list と locale-encodingmapping-list のなかのアイテムたちを集めねばならない。
  Distributable の複数の同時発生は distributable の単一の発生と同じやり方で取り扱わねばならない。
- 配備記述子の中で指定されている URI パスたちは URL デコードされた形であることが前提とされる。URL が CR(#xD)または LF(#xA)文字を含んでいるときは記述的エラー・メッセージでこれを開発者に知らせねばならない。コンテナたちはホワイトスペースを含む URI の総ての文字を保持しなければならない。
- コンテナたちは配備記述子の中でパスたちをカノニカライズ（正規化）することを試みなければならない。例えば、/a/../b という様式のパスは/b.として解釈されねばならない。../ で始まるあるいは../ で始まるパスを解くことはその配備記述子の中の有効なパスではない。
- WAR のルートに相対的なリソースを参照する URI パス、あるいは WAR のルートに相対的なパスのマッピングは、指定されていない限り、/ で始まらねばならない。
- その値が enumerated の型である要素たちの中では、その値は大文字と小文字が区別される。

14.3 配備記述子（Deployment Descriptor）

本使用のこのバージョンの配備記述子は http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd で取得できる。

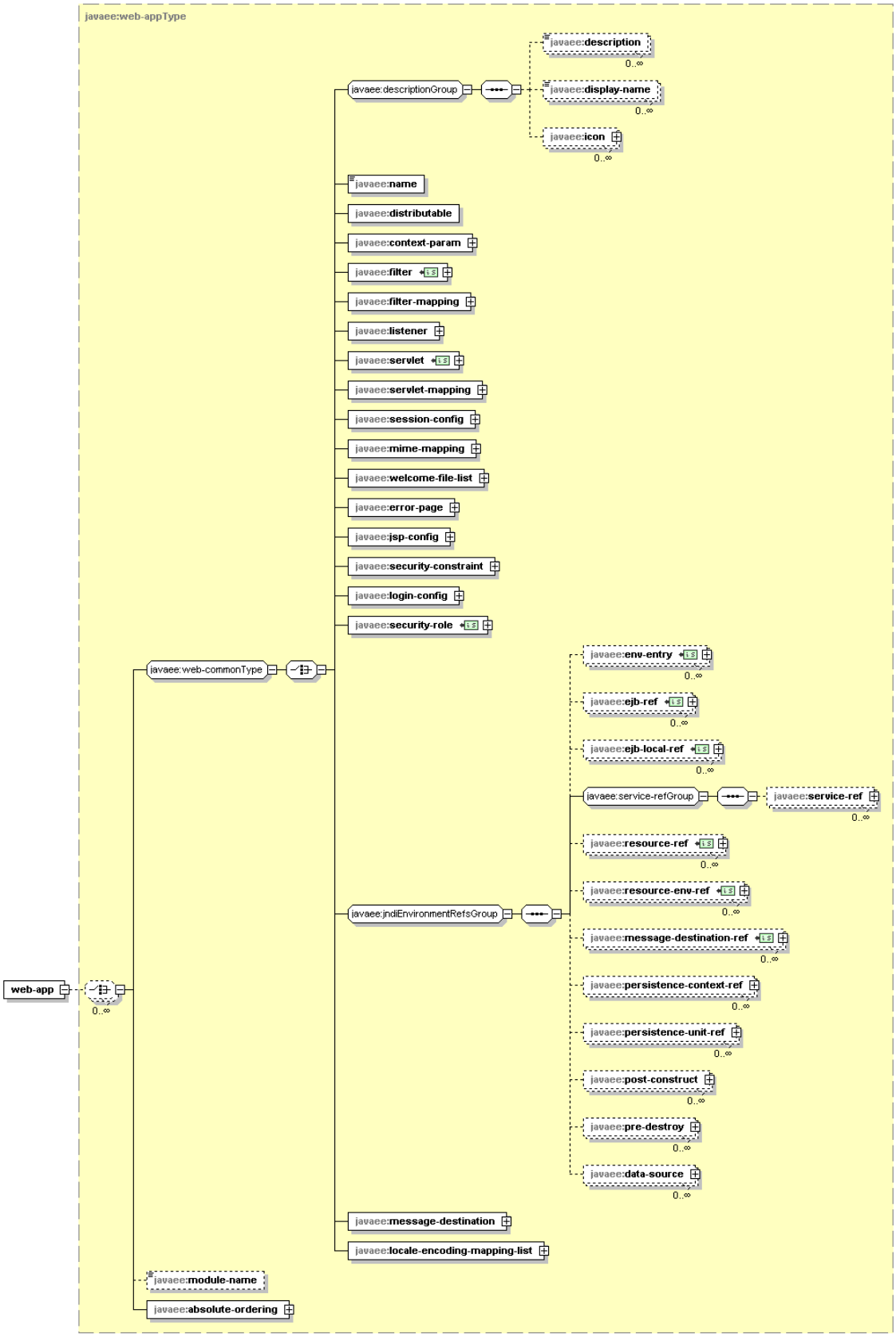
14.4 配備記述子図（Deployment Descriptor Diagram）

本節は配備記述子の要素たちを図示する。これらのダイアグラムには属性たちは示されていない。詳細情報に関しては配備記述子スキーマ（Deployment Descriptor Schema）を見られたい。

This section illustrates the elements in deployment descriptor. Attributes are not shown in the diagrams. See Deployment Descriptor Schema for the detailed information.

1. Web-app（ウェブ・アプリケーション）要素
   web-app 要素はあるウェブ・アプリケーションのルートの配備記述子である。この要素は以下の要素たちを含んでいる。この要素はこの配備記述子がこのスキーマのどのバージョンに対応しているかを指定するために version という属性を持っている。この要素のもとの総てのサブ要素たちは順不同で構わない。

図 14-1： web-app 要素構造


javaee:web-appType
javaee:description
0..∞
javaee:descriptionGroup
javaee:display-name
0..∞
javaee:icon
0..∞
javaee:name
javaee:distributable
javaee:context-param
javaee:filter
javaee:filter-mapping
javaee:listener
javaee:servlet
javaee:servlet-mapping
javaee:session-config
javaee:mime-mapping
javaee:welcome-file-list
javaee:error-page
javaee:jsp-config
javaee:security-constraint
javaee:login-config
javaee:security-role
javaee:web-commonType
javaee:env-entry
0..∞
javaee:ejb-ref
0..∞
javaee:ejb-local-ref
0..∞
javaee:service-refGroup
javaee:service-ref
0..∞
javaee:resource-ref
0..∞
javaee:resource-env-ref
0..∞
javaee:jndiEnvironmentRefsGroup
javaee:message-destination-ref
0..∞
javaee:persistence-context-ref
0..∞
javaee:persistence-unit-ref
0..∞
javaee:post-construct
0..∞
javaee:pre-destroy
0..∞
javaee:data-source
0..∞
web-app
0..∞
javaee:message-destination
javaee:locale-encoding-mapping-list
javaee:module-name
javaee:absolute-ordering

2. description（記述）要素
   description 要素は親の要素を記述するテキストを提供するためのものである。この要素は web-app 要素の下にのみ生じるものではなく、他の複数の要素たちのもとでも生じる。これはオプションとして xml:lang という属性を持ち、この記述にどの言語が使われているかを示す。この属性のデフォルトの値は英語(“en”)である。

3. display-name（表示名）要素
   display-name はツールたちによって表示されることを意図した短い名前を含んでいる。この表示名はユニーク（他に同じものが無い）である必要はない。この要素はオプショナルの言語を指定するための xml:lang という属性を持っている。

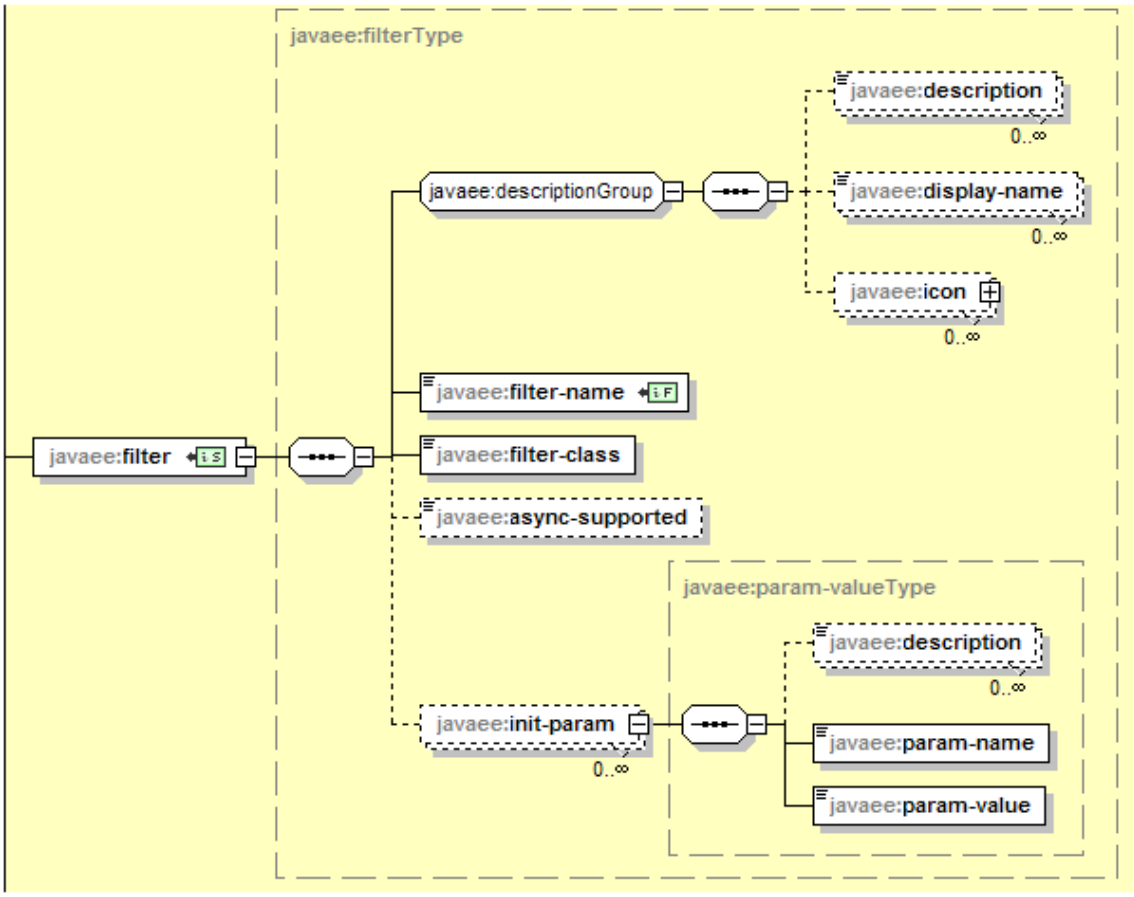
4. icon（アイコン）要素
   icon は small-icon 及び large-icon の要素を含み、これらは GUI ツールの中で親の要素を表示するのに使われる大及び小のサイズの GIF または JPEG イメージを指定する。

5. distributable（分散可能）要素
   distributable はこのウェブ・アプリケーションが分散サーブレット・コンテナの中に配備されるよう適正にプログラムされていることを示す。

6. context-param（コンテキスト・パラメタ）要素
   context-param はウェブ・アプリケーションのサーブレット・コンテキストの初期化パラメタたちの宣言を含む。

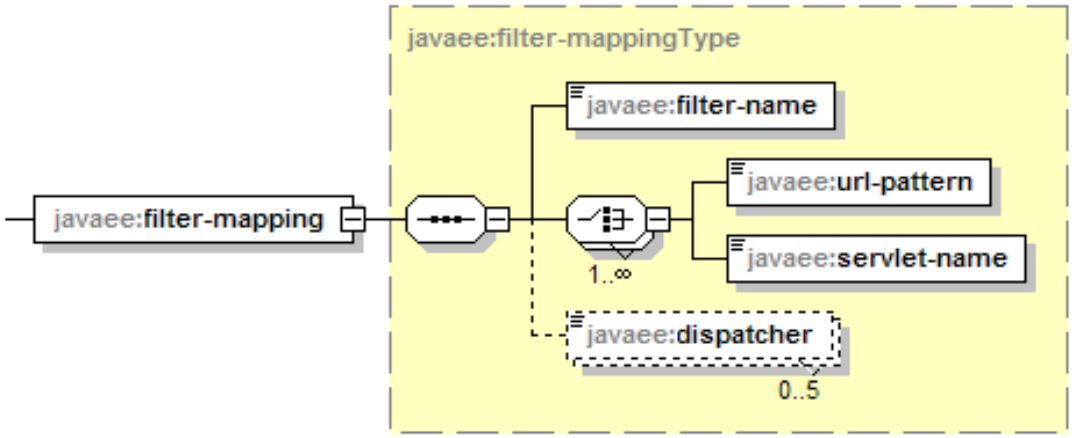
7. filter（フィルタ）要素
   filter はそのウェブ・アプリケーションの中のフィルタを宣言する。このフィルタは filter-mapping 要素の中で参照のための filter-name という値を使ってサーブレットまたは URL パタンのどちらかにマップされる。フィルタは FilterConfig インターフェイスを介してランタイムに配備記述子の中で宣言された初期化パラメタたちのアクセスできる。filter-name 要素はそのフィルタの論理名である。この名前はそのウェブ・アプリケーションのなかでユニーク（他に同じものが無い）でなければならない。filter-name 要素のこの要素のコンテントは空であってはならない。filter-class はそのフィルタの完全修飾クラス名である。init-param 要素はこのフィルタの初期化パラメタとして名前-値のペアを含む。オプショナルな async-supported 要素は、これが指定されていれば、このフィルタは非同期要求処理に対応していることを示す。

図 14-2：filter 要素構造



8. filter-mapping（フィルタ・マッピング）要素
   filter-mapping はそのコンテナがある要求をどのフィルタにどの順序で適用するのかを判断するのに使う。filter-name の値は配備記述子の中のフィルタ宣言たちのひとつでなければならない。マッチした要求は url-pattern または servlet-name のどちらかで指定され得る。

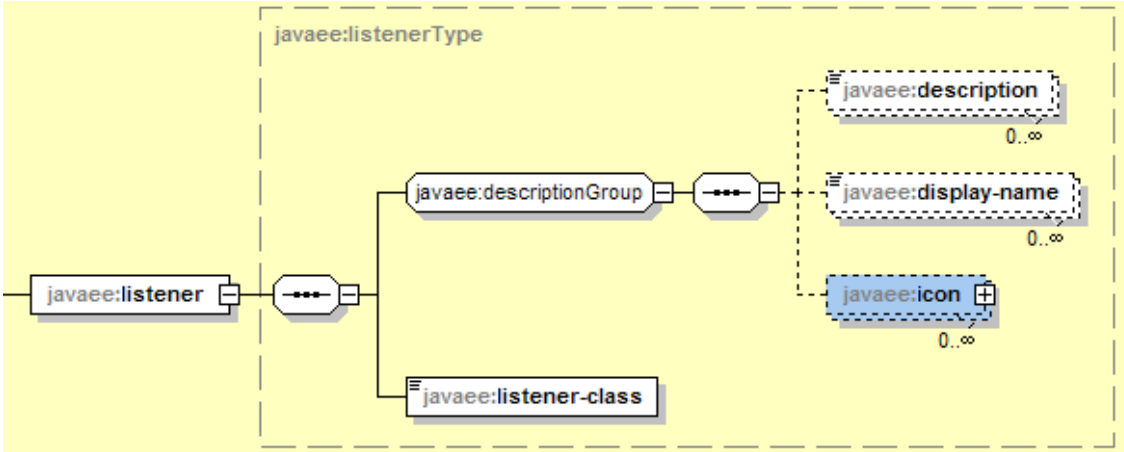
図 14-3：filter-mapping 要素構造



9. listener（リスナ）要素

listener はあるアプリケーション・リスナ・ビーンの配備プロパティを示す。サブ要素の listener-class はそのアプリケーションの中のクラスはウェブ・アプリケーションのリスナ・ビーンとして登録されねばならないことを宣言する。この値はそのリスナの完全修飾クラス名でなければならない。

図 14-4：listner 要素構造

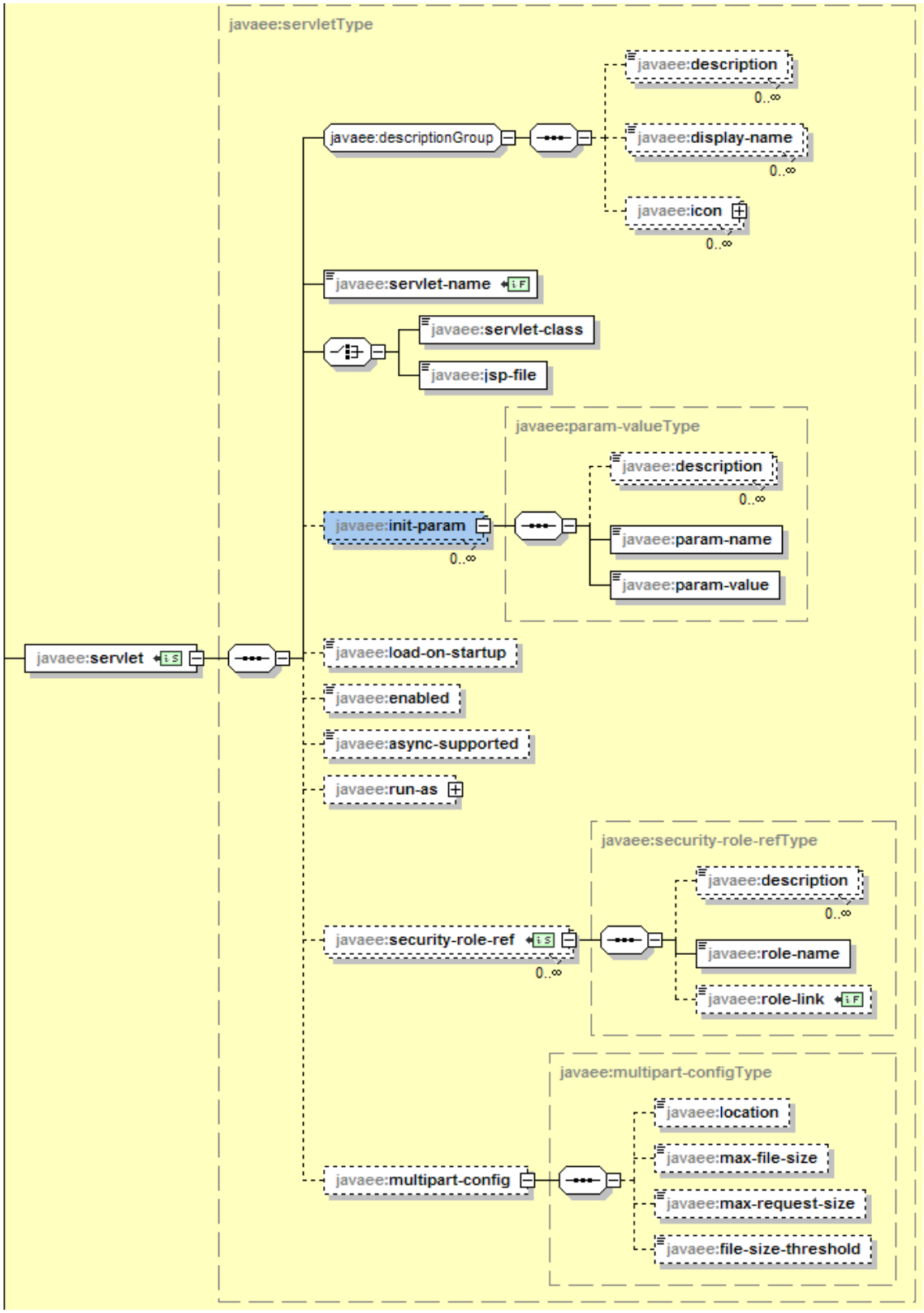


10. servlet（サーブレット）要素

servlet はあるサーブレットを宣言するのに使われる。これはあるサーブレットの宣言的データを含む。jsp-file 要素は“/”で始まるそのウェブ・アプリケーションのある JSP ファイルへの完全パスを含む。もし jsp-file が指定されていて、load-on-startup 要素が存在するときは、その JSP はあらかじめコンパイルされロードされねばならない。servlet-name 要素はそのサーブレットの正規名（カノニカル名）を含む。各サーブレット名はそのアプリケーションの中でユニーク（唯一無二）でなければならない。servlet-name の要素コンテントは空であってはならない。servlet-class はそのサーブレットの完全修飾クラス名を含む。run-as 要素はある部品の実行のために使われるアイデンティティを指定する。これはオプショナルな description と、role-name 要素で指定されたセキュリティ・ロールを含む。 load-on-startup 要素はそのウェブ・アプリケーションのスタートアップ時にこのサーブレットはロードされ（インスタンス化され init()が呼ばれ）ていなければならないことを示す。この要素の要素コンテントはこのサーブレットがロードされる順番を示す整数でなければならない。もしこの値が負の整数の場合は、あるいはこの要素が無い場合は、そのコンテナは何時でも自由にそのサーブレットをロードする。もしこの値が正の整数または()の場合は、そのコンテナはそのサーブレットが配備されるときにそのサーブレットをロードし初期化しなければならない。コンテナは低い整数でマークされたサーブレットたちが高い整数でマークされたサーブレットたちよりも前にロードされることを保障しなければならない。コンテナは同じ load-on-startup 値を持ったサーブレットたちのロード順を選択できる。Security-role-ref 要素はある部品のなかのあるいはある配備部品のコードのなかのセキュリティ・ロール参照を宣言する。これはオプショナルな description、そのコードのなかで使われるセキュリティ・ロール名(role-name)、及びセキュリティ・ロールへのオプショナルなリンク(role-link)で構成される。もしセキュリティ・ロールが指定されていないときは、配備者はしかるべきセキュリティ・ロールを選択しなければならない。オプショナルな async-supported 要素は、それが指定されているときは、そのサーブレットが非同期要求処理に対応できることを示す。もしあるサーブレットがファイル・アップロード（fileupload）機能に対応し mime-multipart 要求処理をサポートして

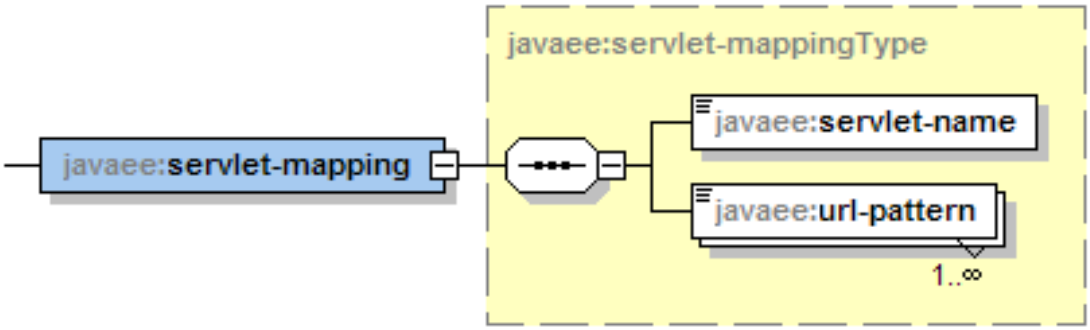
いるときは、同じ設定が記述子の中の multipart-config 要素を介して提供されることができる。multipart-config 要素はファイルたちをストア出来る場所、アップロードされているファイルの最大サイズ、最大要求サイズ、及びその後そのファイルがディスクに書かれるサイズの閾値を指定するのに使われる。

図 14-5：servlet 要素構造



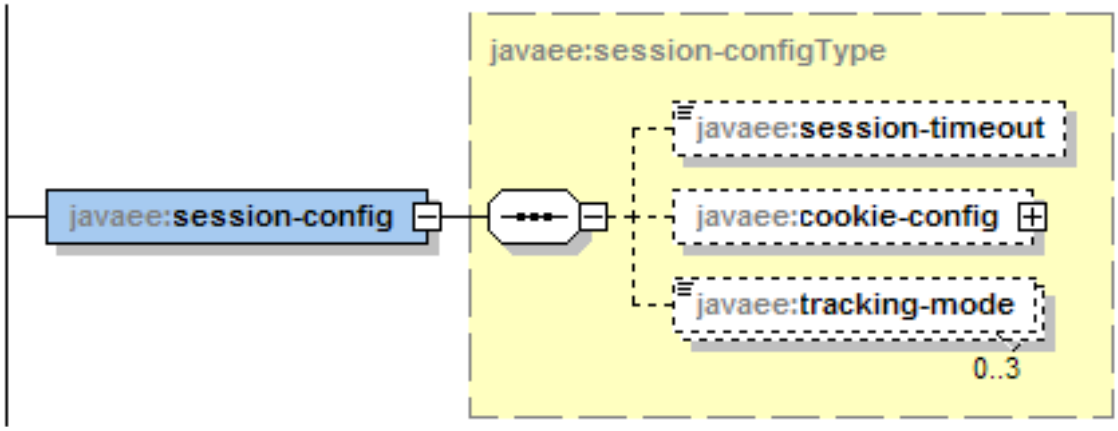
11. servlet-mapping 要素
    servlet-mapping はサーブレットと URL パタンとの間のマッピングを指定する

図 14-6: servlet-mapping 要素構造



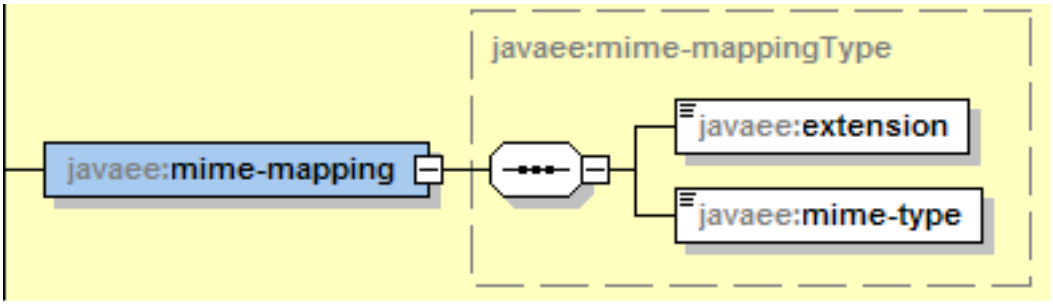
12. session-config（セッション設定）要素
session-config はこのウェブ・アプリケーションの為のセッションのパラメタたちを指定する。session-timeout というサブ要素はこのウェブ・アプリケーションン内で作られる総てのセッションのデフォルトのセッション・タイムアウト時間間隔を定める。指定されたタイム・アウトは分での整数で表現されねばならない。もしこのタイム・アウトが 0 またはそれ以下だったときは、そのコンテナはセッションたちのデフォルトの振る舞いが決してタイム・アウトを起こさないようにしなければならない。もしこの要素が指定されていないときは、そのコンテナは自分のデフォルトのタイム・アウト期間を設定しなければならない。

図 14-7: session-config 要素構造



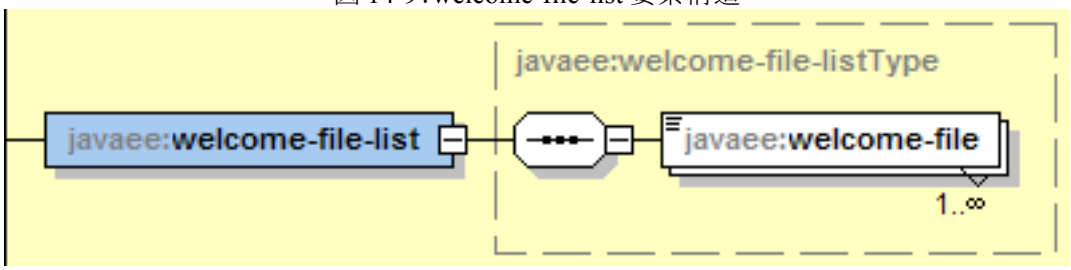
13. mime-mapping（MIME マッピング）要素
mime-mapping はある拡張子(extension)と MIME タイプ間のマッピングを指定する。extension 要素は“txt”のような拡張子を示すある文字列を含む。

図 14-8: mime-mapping 要素構造



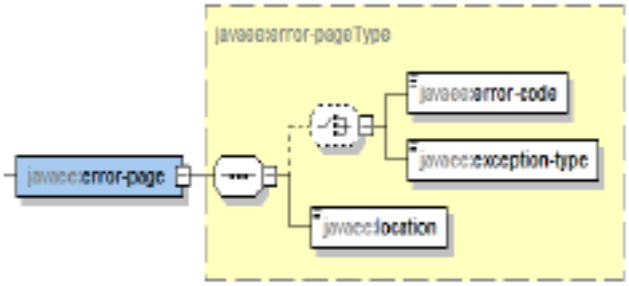
14. welcome-file-list（ウェルカム・ファイル・リスト）要素
welcome-file-list はウェルカム・ファイルたちの順序付けされたリストを含む。welcome-file というサブ要素は index.html のようなデフォルトのウェルカム・ファイルに使われるファイル名を含む。

図 14-9：welcome-file-list 要素構造



15. error-page（エラー・ページ）要素
error-page はエラー・コードまたは例外タイプとそのウェブ・アプリケーション内のあるリソースのパス間のマッピングを含む。exception-type というサブ要素は Java 例外タイプの完全修飾クラス名を含む。location というサブ要素はそのウェブ・アプリケーションのルートに対し相対的なそのウェブ・アプリケーションのなかのリソースの場所を含む。location の値は‘/’で始まっていなければならない。

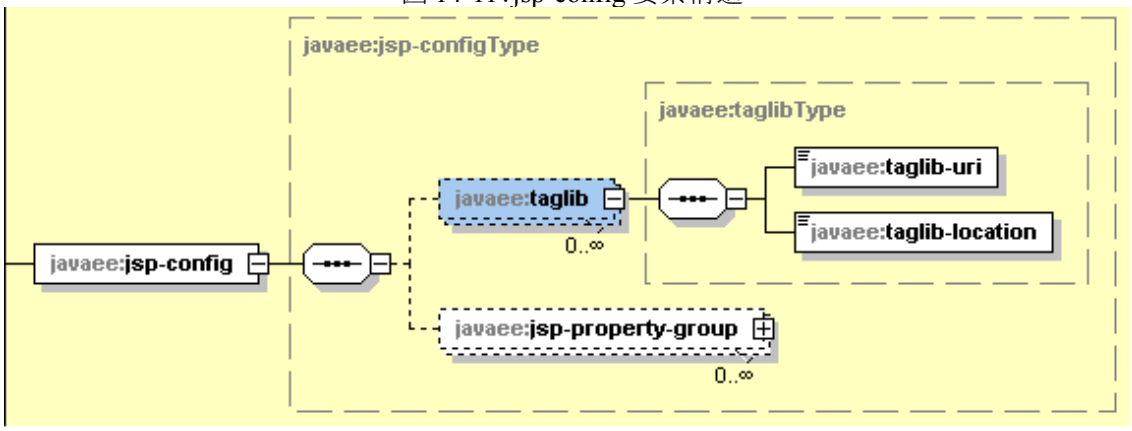
図 14-10： error-page 要素構造



16. jsp-config（JSP 設定）要素
jsp-config はあるウェブ・アプリケーション内の JSP ファイルたちのグローバルな設定情報を与えるために使われる。これは taglib と jsp-property-group の 2 つのサブ要素を持つ。

taglib 要素はそのウェブ・アプリケーションのある JSP ページが使うタグ・ライブラリに関する情報を与えるために使われる。詳細は JavaServer Pages 仕様書の 2.1 版を見られたい。

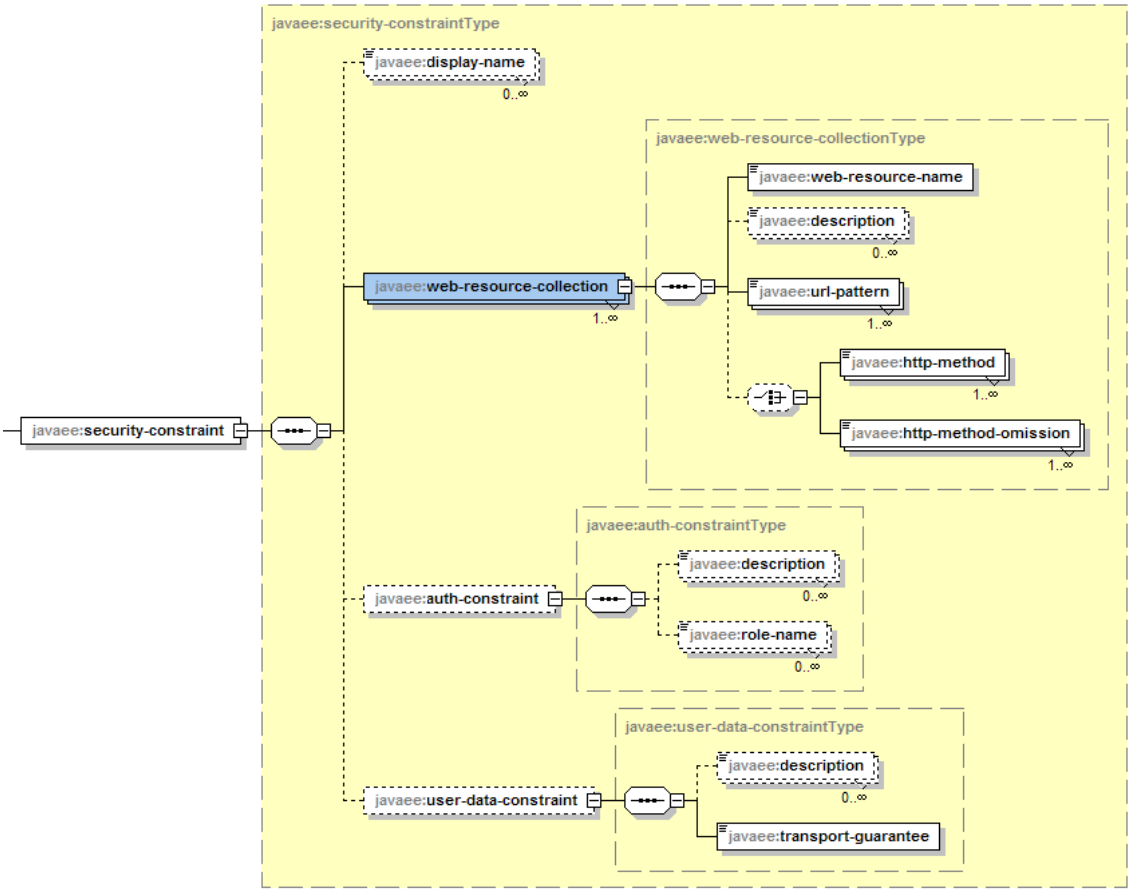
図 14-11 : jsp-config 要素構造



17. security-constraint（セキュリティ制約）要素

security-constraint はセキュリティ制約たちをひとつあるいはそれ以上のウェブ・リソースのコレクションたちと結び付けるのに使われる。web-resource-collection というサブ要素はセキュリティ制約が適用されるあるウェブ・アプリケーション内のリソースたち及びこれらのリソースたちの HTTP メソッドたちのサブセットを特定する。auth-constraint（認証制約）はこのリソース・コレクションにアクセスすることが許可しなければならないユーザ・ロールを指定する。ここで使われている role-name（ロール名）はこのウェブ・アプリケーションのために指定されている security-role 要素たちのひとつの role-name に対応するか、あるいはそのウェブ・アプリケーション内の総てのロールたちを示すコンパクトなシンタックスである"*"という特別に予約されたロール名であるかのいずれかである。"*"とロール名のどちらもないときは、そのコンテナはこれを総てのロールと解釈する。ロールが指定されていないときは、含まれているセキュリティ制約で示されたそのウェブ・アプリケーションのその部分へのアクセスはどのユーザも許されない。コンテナはアクセスを判断するときにはロール名を大文字と小文字を区別して一致をとる。user-data-constraint はそのクライアントとコンテナ間のデータ通信がどのように保護されるべきかを transport-guarantee というサブ要素を介して示す。この transport-guarantee の合法な値は NONE、INTEGRAL、または CONFIDENTIAL のいずれかである。

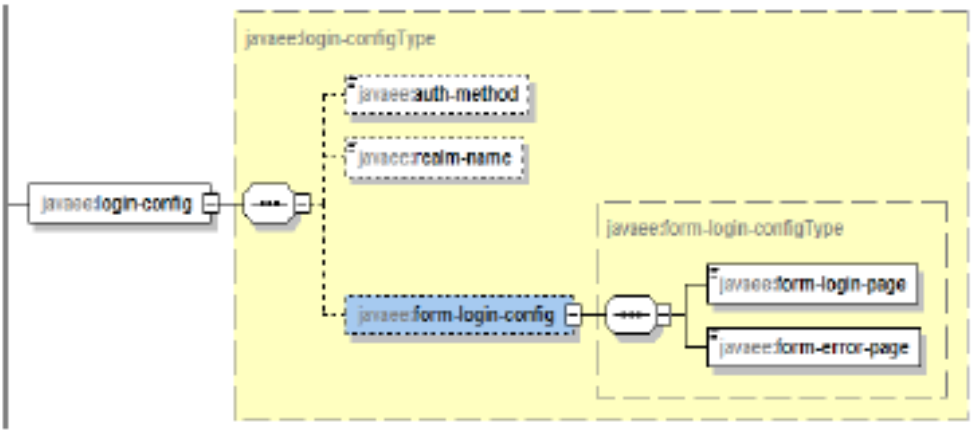
図 14-12： security-constraint 要素構造



18. login-config（ログイン設定）要素

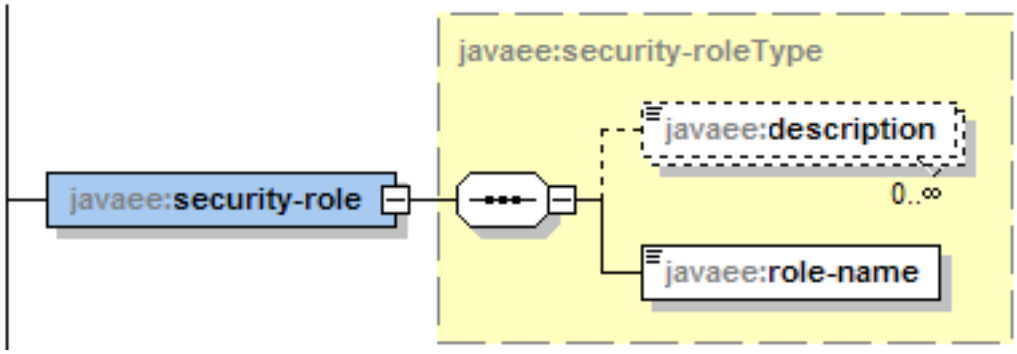
login-config は使われるべき認証手段、このアプリケーションで使われるべきレルム名、及びフォーム・ログインのメカニズムで必要となる属性たち、を設定するのに使われる。auth-method（認証手段）というサブ要素はそのウェブ・アプリケーションに対する認証メカニズムを設定する。これの要素コンテントは BASIC、DIGEST、FORM、CLIENT-CERT、及びコンテナ・メーカ固有の認証のスキームのどちらかでなければならない。realm-name（レルム名）はそのウェブ・アプリケーションのために選択された認証のスキームのために使われるレルム名を示す。form-login-config（フォーム・ログインの設定）は FORM ベースのログインで使われるべきログインとエラーのページを指定する。もし FORM ベースのログインが使われていないときは、これらの要素は無視される。

図 14-13：login-config 要素構造



19. security-role（セキュリティ・ロール）要素
security-role はセキュリティのロールを指定する。role-name というサブ要素はそのセキュリティ・ロールの名前を指定する。この名前は NMTOKEN（名前トークン）のための語彙ルールを満足しなければならない。

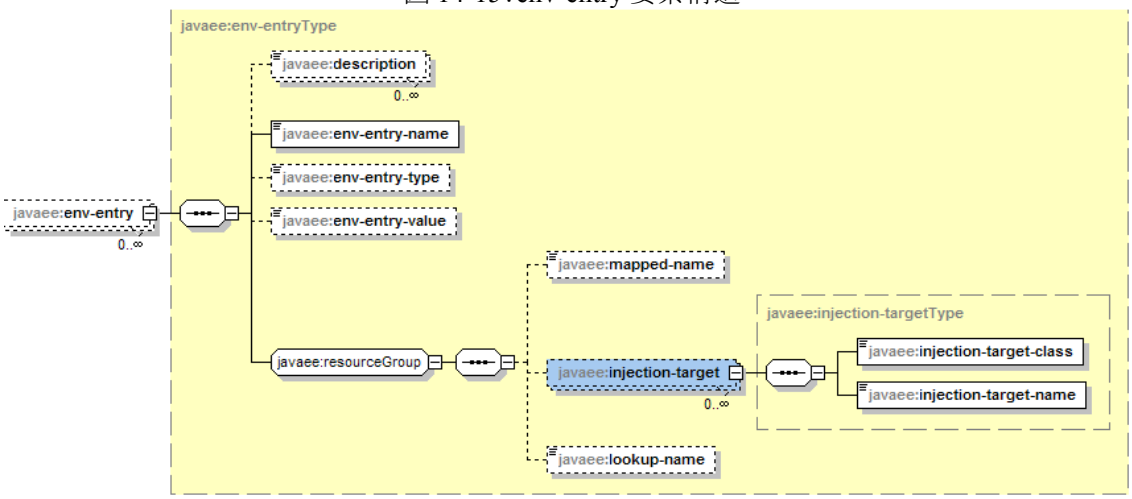
図 14-14：security-role 要素構造



20. env-entry（環境エントリ）要素
env-entry はあるアプリケーションの環境エントリを指定する。サブ要素の env-entry-name（環境エントリ名）はある配備部品の環境エントリの名前を含む。この名前は java:comp/env コンテキストに相対な JDNI 名である。この名前はある配備部品の中ではユニーク（独自）でなければならない。env-entry-type（環境エントリの型）はそのアプリケーションのコードによって想定されている環境エントリ値の完全修飾 Java タイプを含む。この値はパラメタとして単一の String をとる指定された型のコンストラクタにとって有効な String、または java.lang.Character の単一の文字でなければならない。オプショナルな injection-target（注入ターゲット）はフィールドまたは JavaBeans プロパティたちのなかへの指名されたリソースの注入を指定する。injection-target はそのクラス内にあるリソースが注入されるべきクラスとその名前を指定する。 injection-target-class（注入ターゲット・クラス）はその注入のターゲットであるクラスの完全修飾クラス名を指定する。このターゲットは最初に JavaBean プロパ

ティ名として調べられる。もし見つからない場合は、そのターゲットはフィールドの名前として調べられる。指定されたリソースは、そのターゲットのプロパティの set メソッドを呼ぶ、あるいは指名されたフィールドにある値をセットすることで、そのクラスの初期化中にそのターゲットに注入される。もしその環境エントリに injection-target が指定されていると、その enventry-type はオミットされるか、その注入ターゲットの型との一致をとられねば**ならない**。injection-target が指定されていないときは、env-entry-type が必要とされる。

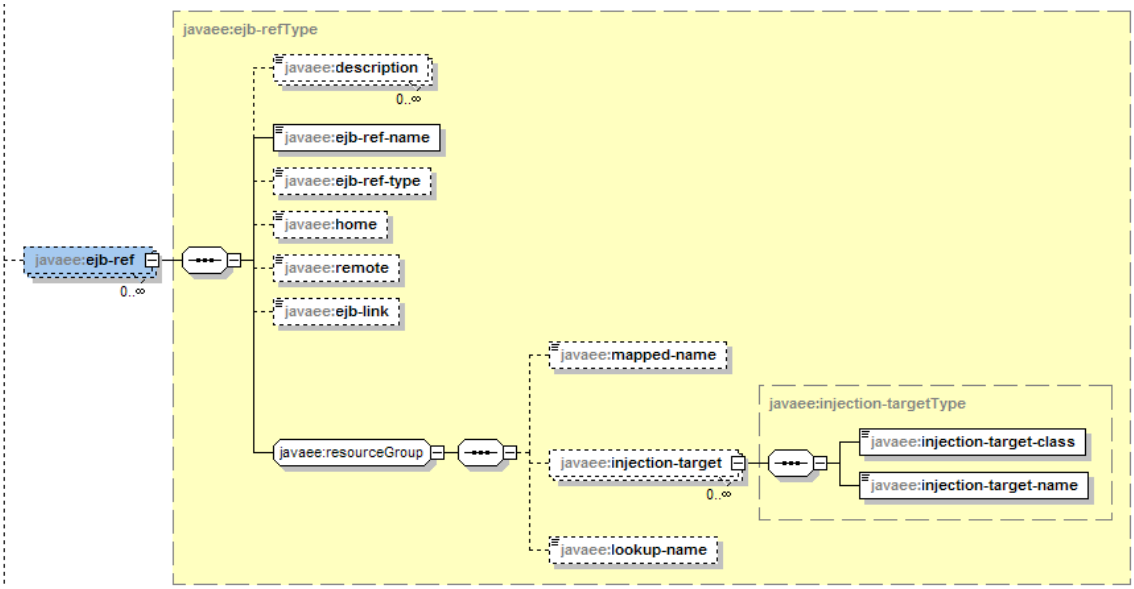
図 14-15：env-entry 要素構造



21. ejb-ref（EJB 参照）要素

ejb-ref はある法人向けビーンのホームへの参照を宣言する。ejb-ref-name（EJB 参照名）はその配備部品のコードの中で使われ、その法人ビーンを参照している。ejb-ref-type（EJB 参照型）は参照される法人ビーンの想定されている型で Entity または Session のどれかである。home（ホーム）は参照される法人ビーンのホーム・インターフェイスの完全修飾名を指定する。ejb-link（EJB リンク）はある EJB 参照がその法人ビーンにリンクしていることを指定する。より詳細に関しては Java プラットホーム法人版第 6 版を見られたい。これらの要素に加えて、injection-target 要素がある部品のフィールドまたはプロパティへの指名された法人ビーンの注入を指定するために使うことができる。

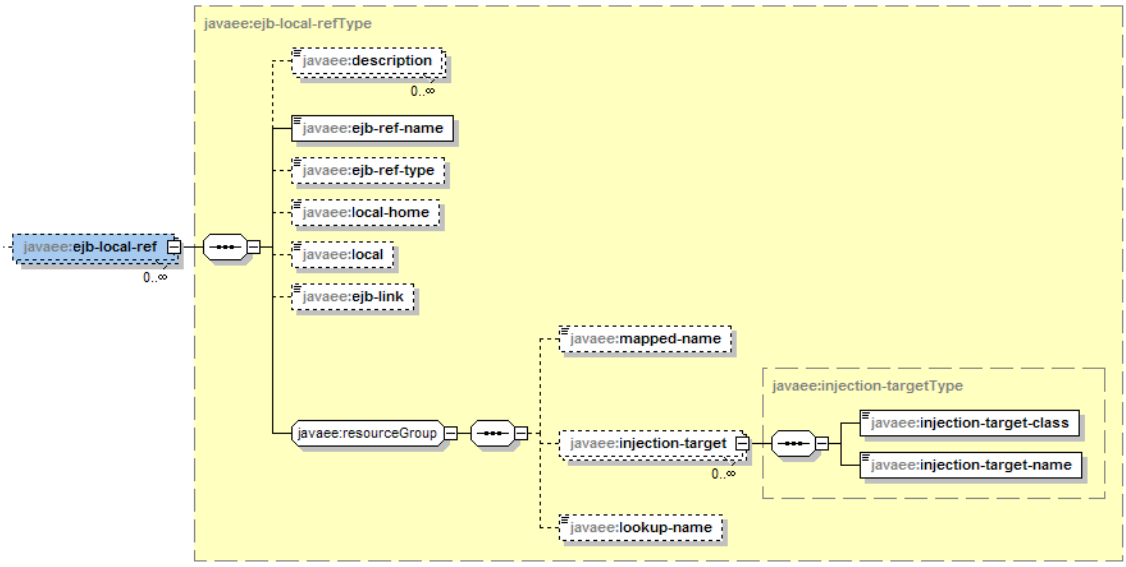
図 14-16： ejb-ref 要素構造



22. ejb-local-ref（EJB ローカル参照）要素

ejb-local-ref はその法人ビーンのローカルなホームへの参照を宣言する。local-home（ローカル・ホーム）はその法人ビーンのローカルなホーム・インターフェイスの完全修飾名を指定する。local（ローカル）はその法人ビーンのローカルなインターフェイスの完全修飾名を指定する。

図 14-17：ejb-local-ref 要素構造

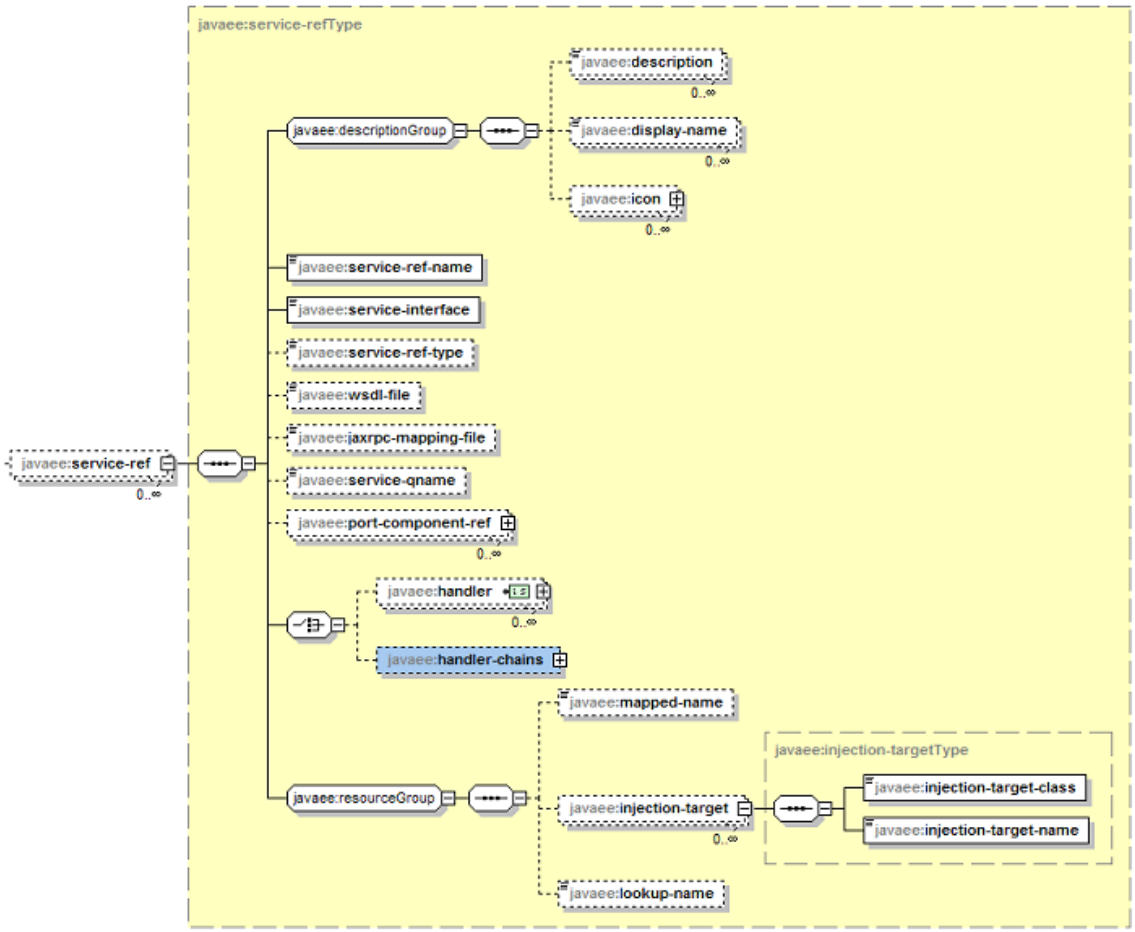


23. service-ref（サービス参照）要素

service-ref はあるウェブ・サービスを宣言する。service-ref-name（サービス参照名）はそのウェブ・サービスを検索するときに使うそのモジュールの中の部品の論理名を宣言する。総てのサービス参照名が/service/で始まることが推奨される。service-interface（サービス・インターフェイス）はそのクライアントが依存する JAX-WS サービス・インターフェイスの完全修飾名を指定する。殆どの場合は、この値は javax.xml.rpc.Service となろう。JAX-WS が

生成した Service Interface クラスもまた指定できる。wsdl-file（WDSL ファイル）は WDSL ファイルの URI の場所を含む。この場所はこのモジュールのルートに相対的である。jaxrpc-mapping-file（Java RPC マッピング・ファイル）はそのアプリケーションが使う Java インターフェイスと wsdl-file のなかの WSDL 記述との間の JAX-WS マッピングを記述するファイルの名前が含まれる。このファイル名はそのモジュール・ファイルのなかでの相対パスである。service-qname（サービス Q 名）要素は参照されている特定の WDSL サービス要素を宣言する。wsdl-file が宣言されていないときはこれは指定されない。 port-component-ref（ポート部品参照）はある WSDL ポートへのサービス・エンドポイント・インターフェイスを見つけるためのそのコンテナへのクライアントの依存性を宣言する。これはオプショナルとして特定のポート部品にこのサービス・エンドポイント・インターフェイスを関連付けする。これは Service.getPort(Class)メソッドでそのコンテナが使う為にのみ使われる。 handler 要素はあるポート部品のためのハンドラを宣言する。ハンドラたちは HandlerInfo インターフェイスを使って init-param の名前-値のペアたちにアクセスできる。もし port-name が指定されていないときは、そのハンドラはそのサービスの総てのポートに関連付けられているとみなされる。詳細に関しては JSR-109 仕様[http://www.jcp.org/en/jsr/detail?id=921]を見られたい。 Java Ee 実装の要素ではないコンテナに関してはこの要素のサポートは要求されない。

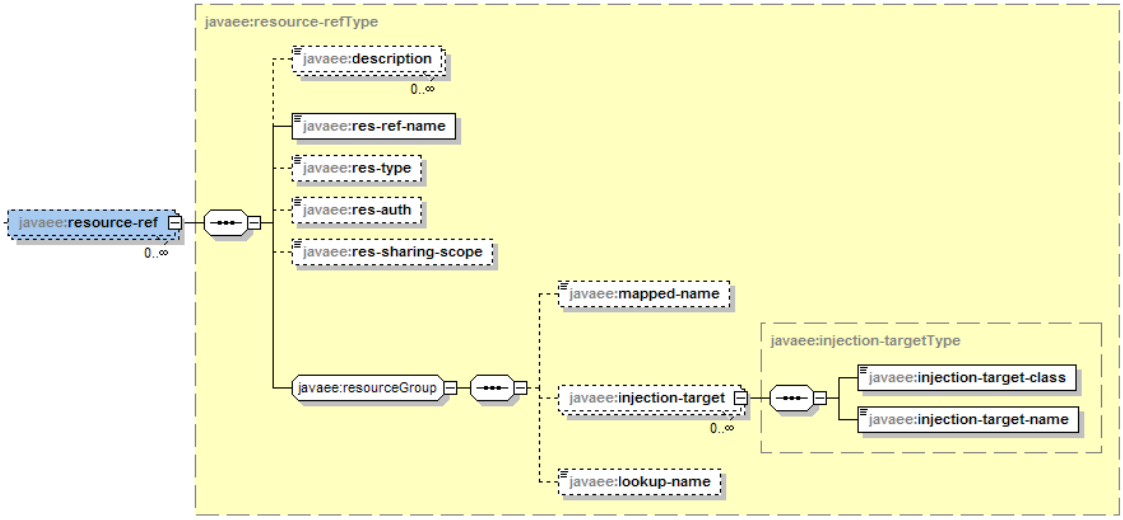
図 14-18： service-ref 要素構造



24. resource-ref（リソース参照）要素

resource-ref はある配備部品の外部リソースへの参照を宣言する。res-ref-name（リソース参照名）はリソース・マネージャ接続ファクトリ参照の名前を指定する。この名前は java:comp/env コンテキストにたいし相対的な JNDI 名である。この名前はある配備ファイルの中ではユニークでなければならない。res-type（リソース型）はそのデータ・ソースの型を指定する。この型はそのデータ・ソースによって実装されると予定される完全修飾 Java 言語のクラスまたはインターフェイスである。 res-auth（リソース認証）はその配備部品のコードがリソース・マネージャに対しプログラム的にサイン・オンするかどうか、あるいはそのコンテナが配備部品の代理としてそのリソース・マネージャにサイン・オンするかどうかを指定する。後者の場合は、そのコンテナは配備者によって与えられた情報を使う。 res-sharing-scope（リソース共有適用範囲）は与えられたリソース・マネージャ接続ファクトリを介して得られ r た接続が共有可能かどうかを指定する。この値は、もし指定されていると、Shareable（共有可能）または Unshareable（共有不能）のどちらかでなければならない。オプショナルな injection-target 要素は指名されたリソースのフィールドあるいは JavaBeans プロパティへの注入を指定するのに使われる。

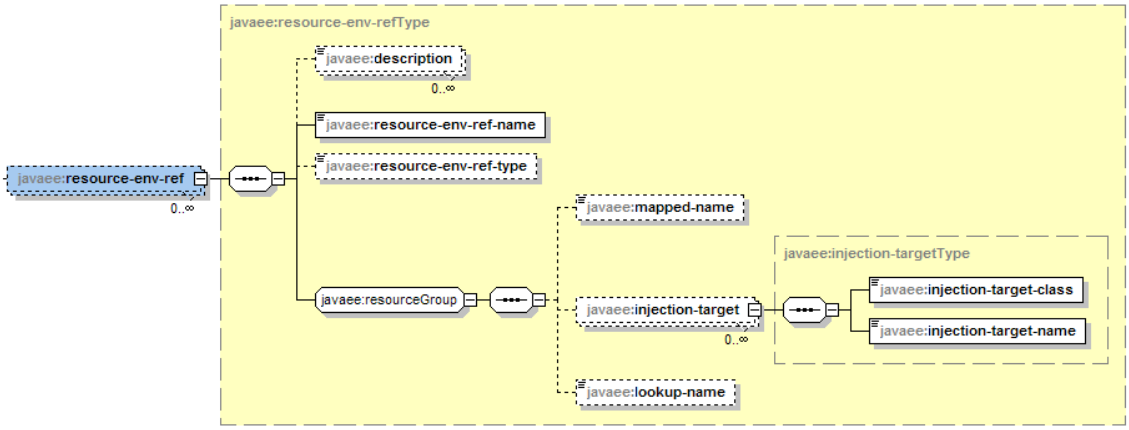
図 14-19： resource-ref 要素構造



25. resource-env-ref（リソース環境参照）要素

resource-env-ref はその配備部品の環境の中のあるリソースに関連付けられた管理されたオブジェクトへのその配備部品の参照を含む。resource-env-ref-name（リソース環境参照名）はそのリソース環境参照の名前を指定する。この値は配備部品のコードの中で使われている環境エントリ名であり、java:comp/env コンテキストに対する相対的な JNDI 名であり、その配備部品内ではユニークでなければならない。 resource-env-ref-type（リソース環境参照型）はそのリソース環境参照の型を指定する。これは Java 言語のクラスまたはインターフェイスの完全修飾名である。オプショナルな injection-target 要素は指名されたリソースのフィールドあるいは JavaBeans プロパティへの注入を指定する。resource-env-ref-type はある注入ターゲットが指定（この場合はターゲットの型が使われる）されていない限り、指定されなければ**ならない**。双方が指定されているときは、この型は注入ターゲットの型と互換性を持たせて割り当てられねば**ならない**。

図 14-20： resource-env-ref 要素構造



26. message-destination-ref(メッセージ宛先参照)要素

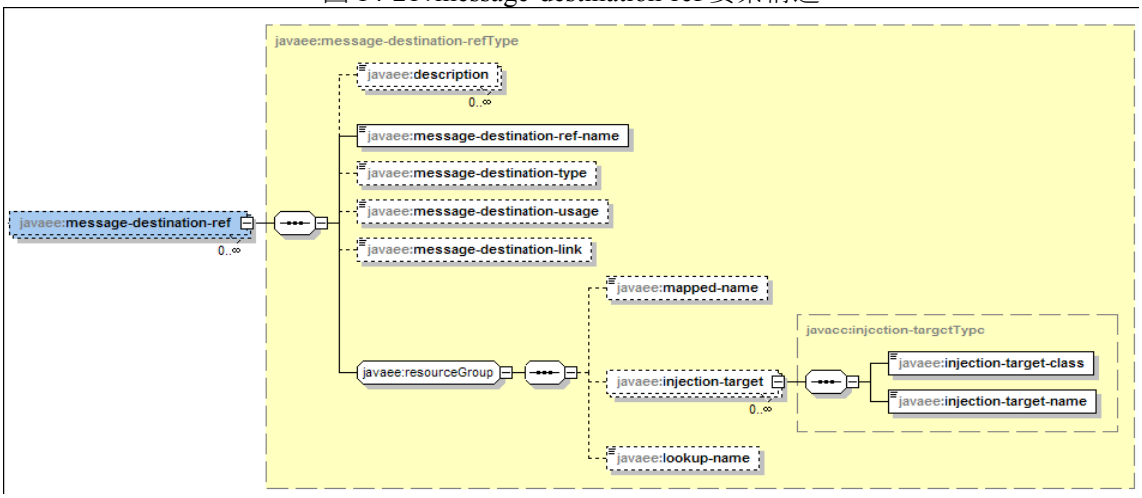
message-destination-ref 要素は配備部品の環境の中のあるリソースに関連付けられたあるメッセージのあて先への配備部品の参照の宣言を含む。この message-destination-ref-name(メッセージ宛先参照名)要素はあるメッセージのあて先参照の名前を指定する;その値は配備部品コードのなかで使われる環境エントリ名である。この名前は java:comp/env コンテキストにたいし相対的な JNDI 名であり、また法人ビーンズ用の ejb-jar のなかであるいはその他の配備ファイルのなかでユニークでなければならない。message-destination-type(メッセージ宛先型)は宛先の型を指定する。この方はその宛先で実装されると想定されている Java インターフェイスにより指定されている。 message-destination-usage(メッセージ宛先使用)はその参照で示されているメッセージの宛先の使用を指定する。この値はメッセージたちがそのメッセージの宛先から消費されるか、宛先向けに作られるか、あるいは双方であるかを示す。アセンブラはこの情報を使ってある宛先のプロデューサたちを消費者たちとリンクする。 message-destination-link(メッセージ宛先リンク)はあるメッセージ宛先参照メッセージまたはメッセージ・ベースのビーンをあるメッセージの宛先にリンクする。アセンブラはこの値をセットして、そのアプリケーション内のプロデューサたちと消費者たち間のメッセージの流れを反映させる。この値は同じ配備ファイルのなかで、あるいは同じ Java EE アプリケーション・ユニット内の別の配備ファイル内の、あるメッセージ宛先の message-destination-name でなければならない。代替的には、この値は参照されたメッセージ宛先を含む配備ファイルを指定するパス名で構成されても良く、その宛先の message-destination-name は追加されまた"#"によってそのパスから分離される。このパス名はそのメッセージの宛先を参照している配備部品を含む配備ファイルに対し相対である。これにより同じ名前を持った複数のメッセージ宛先たちがユニークに特定できる。オプショナルな injection-target 要素は指名されたリソースをフィールドまたは JavaBeans のプロパティたちへの注入を指定する。 message-destination-type は注入ターゲットが指定(その場合はターゲットの型が使用される)されていない限り指定されねばならない。

例:

```
<message-destination-ref>
      <message-destination-ref-name>
            jms/StockQueue
      </message-destination-ref-name>
      <message-destination-type>
            javax.jms.Queue
      </message-destination-type>
```

```
<      message-destination-usage>
            Consumes
      </message-destination-usage>
      <message-destination-link>
            CorporateStocks
      </message-destination-link>
</message-destination-ref>
```

図 14-21：message-destination-ref 要素構造



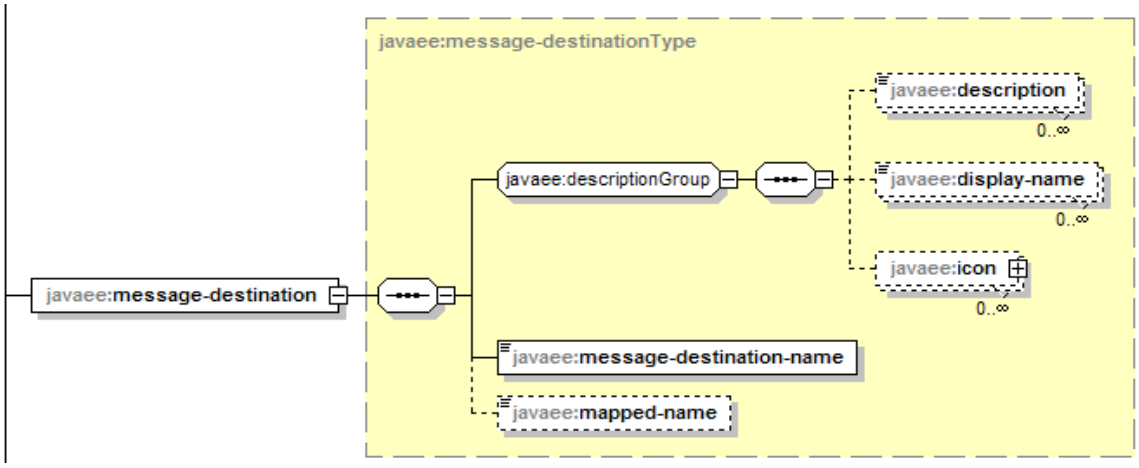
27. message-destination（メッセージ宛先）要素

    message-destination はあるメッセージの宛先を指定する。この要素で記された論理的な宛先は配備者によって物理的な宛先にマップされる。message-destination-name（メッセージ宛先名）要素はあるメッセージ宛先の名前を指定する。この名前はその配備ファイルのなかのメッセージ宛先たちの名前たちのなかでユニークでなければならない。

    例：

```
<message-destination>
      <message-destination-name>
            CorporateStocks
      </message-destination-name>
</message-destination>
```

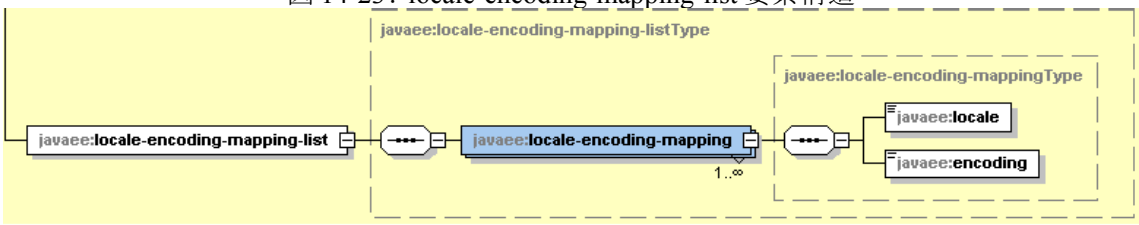
図 14-22：message-destination 要素構造



28. locale-encoding-mapping-list（ロケール・エンコーディング・マッピング・リスト）要素
locale-encoding-mapping-list は locale-encoding-mapping（ロケール・エンコーディング・マッピング）サブ要素で指定されたそのロケールとエンコーディング間のマッピングを含む。
例

```
<locale-encoding-mapping-list>
      <locale-encoding-mapping>
            <locale>ja</locale>
            <encoding>Shift_JIS</encoding>
      </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

図 14-23：locale-encoding-mapping-list 要素構造



14.5 例

以下の幾つかの例は配備記述子の図でリストされた定義たちの使い方を示している。

14.5.1 ベーシックな例

コード例 14-1：ベーシックな配備記述子例

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd”
      version=”2.5”>
      <display-name>A Simple Application</display-name>
      <context-param>
            <param-name>Webmaster</param-name>
            <param-value>webmaster@mycorp.com</param-value>
      </context-param>
      <servlet>
            <servlet-name>catalog</servlet-name>
            <servlet-class>com.mycorp.CatalogServlet
            </servlet-class>
            <init-param>
                  <param-name>catalog</param-name>
                  <param-value>Spring</param-value>
            </init-param>
      </servlet>
      <servlet-mapping>
            <servlet-name>catalog</servlet-name>
            <url-pattern>/catalog/*</url-pattern>
      </servlet-mapping>
      <session-config>
            <session-timeout>30</session-timeout>
      </session-config>
      <mime-mapping>
            <extension>pdf</extension>
            <mime-type>application/pdf</mime-type>
      </mime-mapping>
      <welcome-file-list>
            <welcome-file>index.jsp</welcome-file>
            <welcome-file>index.html</welcome-file>
            <welcome-file>index.htm</welcome-file>
      </welcome-file-list>
      <error-page>
            <error-code>404</error-code>
            <location>/404.html</location>
      </error-page>
</web-app>
```

14.5.2 セキュリティの例

コード例 14-2:セキュリティを使った配備記述子例

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
      version=”2.5”>
```

```
      <display-name>A Secure Application</display-name>
      <servlet>
      <servlet-name>catalog</servlet-name>
            <servlet-class>com.mycorp.CatalogServlet
            </servlet-class>
      <init-param>
            <param-name>catalog</param-name>
            <param-value>Spring</param-value>
      </init-param>
      <security-role-ref>
            <role-name>MGR</role-name>
            <!-- role name used in code -->
            <role-link>manager</role-link>
      </security-role-ref>
      </servlet>
      <security-role>
            <role-name>manager</role-name>
      </security-role>
      <servlet-mapping>
            <servlet-name>catalog</servlet-name>
            <url-pattern>/catalog/*</url-pattern>
      </servlet-mapping>
      <security-constraint>
            <web-resource-collection>
                  <web-resource-name>SalesInfo
                  </web-resource-name>
                  <url-pattern>/salesinfo/*</url-pattern>
                  <http-method>GET</http-method>
                  <http-method>POST</http-method>
            </web-resource-collection>
            <auth-constraint>
                  <role-name>manager</role-name>
            </auth-constraint>
            <user-data-constraint>
                  <transport-guarantee>CONFIDENTIAL
                  transport-guarantee>
            </user-data-constraint>
      </security-constraint>
</web-app>
```

第15章　　他の仕様と関連する要求事項（Requirements related to other Specifications）

本章では他の Java 技術たちを含んだ製品のなかに含められるウェブ・コンテナたちへの要求事項をリストしている。以下の節での何らかの Java EE 参照は、完全な Java EE プロファイルだけではなく Java EE Web Profile のようなサーブレットをサポートしているどのプロファイルにも適用される。プロファイルに関する更なる情報に関しては Java EE プラットホーム仕様を参照されたい。

15.1　セッション（Sessions）

Java EE 実装の一部である分散サーブレット・コンテナたちは、ひとつの JVM から他の JVM に他の Java EE オブジェクトたちを移行させるのに必要なメカニズムをサポートしなければならない。

15.2　ウェブ・アプリケーション（Web Applications）

15.2.1　ウェブ・アプリケーション・クラス・ローダ（Web Application Class Loader）

Java EE 製品の一部であるようなサーブレット・コンテナたちは、Java SE または Java EE が手を加えることを認めていない java.*及び javax.*名前空間の中のように、Java SE または Java EE プラットホームのクラスたちをそのアプリケーションがオーバライドするのを許してはならない。

15.2.2　ウェブ・アプリケーション環境（Web Application Environment）

Java EE ではどのように外部情報が名前付けられているかあるいは編成されているかの明示的な知識なしでリソースたちと外部情報に簡単にアプリケーションたがアクセス出来るように名前付け環境を定義している。サーブレットが Java EE 技術の不可欠の部品のタイプであるので、サーブレットがリソースたちと法人ビーンたちへの参照を取得出来る情報を指定するためにウェブ・アプリケーションの配備記述子のなかで設定がなされている。この情報を含んでいる配備要素たちは以下のようである：

- env-entry
- ejb-ref

- ejb-local-ref
- resource-ref
- resource-env-ref
- service-ref
- message-destination-ref
- persistence-context-ref
- persistence-unit-ref

開発者はこれらの要素を使って、そのウェブ・アプリケーションがランタイムにウェブ・コンテナのなかの JNDI 名前空間に登録されることを必要としているある種のオブジェクトたちを記述する。環境設定に関しての Java EE 環境の要求事項は Java EE 仕様書の第 5 章に記されている。Java EE 技術対応実装の一部であるようなサーブレット・コンテナたちは、このシンタックスに対応することが要求される。より詳細に関しては Java EE を見られたい。この種のサーブレット・コンテナは、そのようなオブジェクトと、そのサーブレット・コンテナが管理するあるスレッド上で行われるときはこれらのオブジェクトになされた呼び出しの、ルックアップ（検索）をサポートしなければならない。この種のサーブレット・コンテナは、開発者が作ったが現在そうすることが要求されていないスレッド上で実施されるときは、この振舞いをサポートしなければならない。そのような要求は本仕様の次の版で追加されよう。開発者たちはアプリケーションが作ったスレッドたちのためのこのような機能は可搬でないので勧められないことに注意しなければならない。

15.2.3 ウェブ・モジュール・コンテキスト・ルート URL の為の JNDI 名（JNDI Name for Web Module Context Root URL）

Java EE プラットホーム仕様書では Java EE アプリケーションのいろんな適用範囲にマップする標準化されたグローバルな JNDI 名前空間と一連の関連した名前空間を定めている。これらの名前空間たちはアプリケーションたちが可搬的に部品たちとリソースたちへの参照を検索する為に使える。本節はそれによってあるウェブ・アプリケーションのベース URL が登録されることが要求される JNDI 名を定義する。

あらかじめ定められているあるウェブ・アプリケーションのコンテキスト・ルートの為の java.net.URL リソース名は以下のシンタックスを持っている:

`java:global[/<app-name>]/<module-name>`!グローバルな名前空間の中のルート 及び `java:app/<module-name>`!アプリケーション固有の名前空間の中のルート

アプリケーション名（ app name ）とモジュール名（module name）の決定の為の規則に関しては EE 8.1.1 節「部品の生成」及び EE 8.1.2 節「アプリケーションの組み立て」を見られたい。<app-name> は webapp が.ear ファイル内にパックされているときにのみ適用可能である。java:app というプレフィックスによりある Java EE アプリケーションのなかで実行中のある部品がアプリケーション固有のある名前にアクセスすることが出来る。 java:app 名はある法人アプリケーションのなかのあるモジュールが同じ法人アプリケーション内の別のモジュールのコンテキスト・ルートを参照できるようにする。<module-name>は java:app url のシンタックスで必要な部分である。

例

上記の URL は次に以下のようにアプリケーションのなかで使われうる:
もしあるウェブ・アプリケーションが myWebApp としてのモジュール名でスタンドアロンで配備されていると、その URL は以下のように別のウェブ・モジュールに注入される:
コード例 15-1

```
@Resource(lookup="java:global/myWebApp!ROOT")
URL myWebApp;
```

myApp という名前の ear ファイルにパッケージされているときは、以下のようにつかえる:
コード例 15-2

```
@Resource(lookup="java:global/myApp/myWebApp!ROOT")
URL myWebApp;
```

15.3 セキュリティ(Security)

本節では EJB、JACC、あるいは JASPIC も含んでいる製品の中に含められているときのウェブ・コンテナたちのための追加のセキュリティ要求事項の詳細を記している。以下の節でこれらの要求を拾い出している。

15.3.1 EJB™呼び出しの中でのセキュリティ・アイデンティティの伝播(Propagation of Security Identity in EJB™ Calls)

セキュリティ・アイデンティティ(またはプリンシパル)は法人ビーン呼び出しで使われるために常に用意されねばならない。ウェブ・アプリケーションから法人ビーンズへの呼び出しのデフォルトのモードは、EJB コンテナに伝搬されるべきあるウェブ・ユーザのセキュリティ・アイデンティティである。他のシナリオたちでは、ウェブ・コンテナたちはそのウェブ・コンテナたちにとったまたはその EJB コンテナたちにとって知らないウェブ・ユーザたちに呼び出しを許すことが要求される:

- ウェブ・コンテナたちはそのコンテナたちに認証されていないクライアントたちのウェブ・リソースたちへのアクセスに対応することが要求される。これはインターネット上のウェブ・リソースたちへのアクセスの共通のモードである。
- アプリケーションのコードは、呼び出し者のアイデンティティに基づくサインオンとデータの課す多くかの単一のプロセサであっても良い。

これらのシナリオでは、ウェブ・アプリケーションの配備記述子は run-as 要素を指定できる。これが指定されている時は、そのコンテナはあるサーブレットからその EJB 層への呼び出しに、run-as 要素で指定されたセキュリティ・ロール名に関し、セキュリティ・アイデンティティを伝搬させねばならない。このセキュリティ・ロール名はそのウェブ・アプリケーション用に定められたセキュリティ・ロール名たちのひとつでなければならない。

Java EE プラットホームの一部として動作しているウェブ・コンテナたちでは、この run-as 要素の使用は、同じ Java EE アプリケーションたちのなかの EJB 部品たちの呼び出しと、他の Java EE アプリケーションたちのなかで配備された EJB 部品たちの呼び出しの双方でサポートされねばならない。

15.3.2 コンテナ認可の要求事項（Container Authorization Requirements）

Java EE 製品あるいは Java Authorization Contracts for Containers (JACC、即ち JSR 115)対応を含む製品においては、総てのサーブレット・コンテナたちは JACC 対応を実装しなければ**ならない**。JACC 仕様は http://www.jcp.org/en/jsr/detail?id=115 からダウンロード可能である。

15.3.3 コンテナ認証の要求事項（Container Authentication Requirements）

Java EE 製品、あるいは The Java Authentication SPI for Containers (JASPIC、即ち JSR 196)対応を含んでいる製品においては、総てのサーブレット・コンテナたちは JASPIC 対応を実装しなければ**ならない**。JACC 仕様は http://www.jcp.org/en/jsr/detail?id=196 からダウンロード可能である。

15.4 配備（Deployment）

本節では Java EE 技術対応コンテナ、及び JSP 及び / あるいはウェブ・サービス対応を含む製品における配備記述子、パッケージング、及び配備記述子処理に関する要求事項の詳細を示す。

15.4.1 配備記述子の要素たち（Deployment Descriptor Elements）

JSP ページ対応または Java EE アプリケーション・サーバの一部になっているウェブ・コンテナたちの要求事項にあわせて、ウェブ・アプリケーション配備記述子の中に以下のような更なる要素たちが存在している。サーブレット仕様のみに対応するコンテナたちはこれらをサポートすることは要求されない。

- jsp-config
- リソース参照宣言のシンタックス(env-entry, ejb-ref, ejb-local-ref, resource-ref, resource-env-ref)
- メッセージ宛先指定のためのシンタックス(message-destination, messagedestination-ref)
- ウェブ・サービスへの参照(service-ref)
- Persistence コンテキストへの参照(persistence-context-ref)

- Persistence Unit への参照(persistence-unit-ref)

これらの要素のシンタックスは現在は JavaServer Pages 仕様書 2.2 版及び Java EE 仕様書で示されている。

15.4.2 JAX-WS 部品のパッケージングと配備（Packaging and Deployment of JAX-WS Components）

ウェブ・コンテナたちは JAX-RPC 及び / あるいは JAX-WS 仕様で規定されたウェブ・サービスのエンドポイントを実装するために書かれた部品たちの稼働をサポートすることを選択してよい。 Java Ee 適合実装物の中に組み込まれたウェブ・コンテナたちは、JAX-RPC 及び JAX-WS ウェブ・サービス部品たちをサポートすることが要求される。本節では JAX-RPC 及び JAX-WS もまたサポートする製品に含まれているときのウェブ・コンテナたちのパッケージングと配備モデルを記す。

JSR-109 [http://jcp.org/jsr/detail/109.jsp]ではウェブ・サービス・インターフェイスをそれに結び付いた WSDL 記述と関連クラスたちをパッケージングするためのモデルを規定している。これは JAX-RPC 及び JAX-WS 対応のウェブ・コンテナたちがこのウェブ・サービスを実装する部品にリンクするためのメカニズムを定めている。JAX-WS または JAX-RPC ウェブ・サービス実装部品は、JAX-WS 及び / または JAX-RPC 仕様で（これは JAX-WS 及び / または JAX-RPC 対応ウェブ・コンテナたちとの契約を定めている）規定されている API たちを使う。これは WAR ファイルにパッケージされる。ウェブ・サービスの開発者は通常の<servlet>宣言を使ってこの部品の宣言を行う。

JAX-WS 及び JAX-RPC 対応のウェブ・コンテナたちは、servlet 要素を使っている HTTP サーブレット部品たち向けと同じシンタックスを使って、エンドポイント実装部品の以下の情報を指定するのにウェブの配備記述子を開発者が使うのに対応しなければならない。子供の要素たちは以下のようにエンドポイント情報を指定するのに使われる：

- servlet-name 要素はその WAR の中の他のウェブ部品たちのなかからこのエンドポイント記述を特定するのに使われる可能性がある論理名を指定する
- servlet-class 要素はこのエンドポイント実装物の完全修飾クラス名を指定する
- description 要素（たち）はその部品を記述しまたツールのなかで表示されるのに使われる
- load-on-startup 要素はそのウェブ・コンテナのなかで他のウェブ部品たちと相対してその部品が初期化される順序を指定する
- security-role-ref 要素は認証されたそのユーザがある論理セキュリティ・ロールの中にいるかどうかをテストするのに使える
- run-as 要素はこの部品による EJB たちのコールに伝搬されるアイデンティティをオーバライドするのに使われる

このウェブ部品のためにその開発者が定めた何らかの初期化パラメタたちはそのコンテナによって無視されて良い。更に、JAX-WS 及び JAX-RPC 対応のウェブ部品は以下の情報を指定するのにこれまでのウェブ部品のメカニズムを引き継ぐ：

- サーブレット・マッピング技術を使ってそのウェブ・コンテナの URL 名前空間にその部品をマッピングする
- セキュリティ制約を使ってウェブ部品たちへの認可の制約を課す

- フィルタ・マッピング技術を使って JAX-WS 及び / あるいは JAX-RPC メッセージの操作をサポートするために低レベルのバイト・ストリームの対応を提供するためにサーブレット・フィルタが使える
- その部品にかかわる何らかの HTTP セッションのタイム・アウト特性
- JNDI 名前空間の中にストアされた Java EE オブジェクトたちへのリンク

上記要求事項の総てが第 8.2 節の「プラグ可能性」で定められたプラグ可能性メカニズムを使って満足させることができる。

15.4.3 配備記述子処理のための規則（Rules for Processing the Deployment Descriptor）

Java EE 技術対応実装の一部であるコンテナたちとツールたちは、その配備記述子が XML スキームに対する構造的正確さを持っていることを確認することが要求される。この確認は勧告であるが、Java EE 技術対応実装の一部でないウェブ・コンテナとツールでは要求されない。

15.5 アノテーションとリソース注入（Annotations and Resource Injection）

J2SE 5.0 及びそれ以降の一部となっている Java メタデータ仕様書（JSR-175：Java Metadata specification）では Java コードのなかで設定データを指定する手段が用意されている。Java コードのなかのメタデータはまたアノテーションとしても呼ばれている。Java EE においては、アノテーションは外部リソースたちによる依存性を宣言し、また Java コード内で設定データを宣言し設定ファイルないでそのデータを規定する必要をなくしている。

本節は Java EE 技術対応のサーブレット・コンテナたちのなかでのアノテーションとリソース注入の振る舞いを記述する。本節は Java EE 仕様書第 5 章の「リソース、名前付け、及び注入」を拡張している。

アノテーションたちは以下のインターフェイスたちを実装した以下のコンテナが管理するクラスたちでサポートされねばならず、ウェブ・アプリケーション配備記述子の中で宣言される、あるいは第 8.1 節の「アノテーションとプラグ可能性」で定められたアノテーションを使って宣言される、あるいはプログラム的に付加される。

表 15-1：アノテーションと依存性注入がサポートされる部品とインターフェイス

部品のタイプ	以下のインターフェイスを実装するクラス
サーブレット	javax.servlet.Servlet
フィルタ	javax.servlet.Filter
リスナ	javax.servlet.ServletContextListener javax.servlet.ServletContextAttributeListener javax.servlet.ServletRequestListener

	javax.servlet.ServletRequestAttributeListener javax.servlet.http.HttpSessionListener javax.servlet.http.HttpSessionAttributeListener javax.servlet.AsyncListener

ウェブ・コンテナたちは上記表 15-1 にリストされたもの以外のクラスたちで生じるアノテーションたちの為のリソース注入を実施することは要求されていない。

参照たちは呼び出されているメソッドたち及びそのアプリケーションで利用できる部品インスタンスの何らかのライフサイクルの前に注入されねばならない。

ウェブ・アプリケーションの中では、リソース注入を使っているクラスたちは、それらが WEB-INF/classes ディレクトリ内に置かれているときに限り、あるいは WEB-INF/lib の中に置かれている jar ファイルの中にパッケージされているときに、自分たちのアノテーションたちが処理される。コンテナたちはオプションとしてそのアプリケーションのクラスパス内のどこかで見つかったクラスたちのリソース注入アノテーションを処理してよい。

ウェブ・アプリケーションの配備記述子は web-app 要素上で metadata-complete という属性を含んでいる。この metadata-complete 属性はその web.xml 記述子が完了しているか、あるいはその配備処理で使われるメタデータの他のソースを検討すべきかどうかを指定する。メタデータは、web.xml ファイル、web-fragment.xml ファイルたち、WEBINF/classes 内のクラス・ファイルたちのアノテーションたち、及び WEB-INF/lib ディレクトリ内の jar ファイルたちのなかのクラスたち上のアノテーションたちで存在する。もし metadata-complete が"true"にセットされていると、その配備のツールは web.xml ファイルのみを調べ、そのアプリケーションのクラス・ファイルたちの中で出てくる @WebServlet、@WebFilter、及び@WebListener のようなアノテーションたちを無視しなければならず、また WEB-INF/lib 内の jar ファイル内にパッケージされている web-fragment.xml 記述子も無視しなければならない。もし metadata-complete 属性が指定されていないあるいは"false"にセットされているときは、その配備ツールは先に指定したメタデータのためのクラス・ファイルたちと web-fragment.xml ファイルたちを調べねばならない。

web-fragment.xml もまた web-fragment 要素上で metadata-complete 属性を含んでいる。この属性はその web-fragment.xml 記述子が与えられたフラグメントの中で完了しているか、あるいは関連する jar ファイル内のクラスたちのアノテーションたちをスキャンすべきかどうかを指定する。もしこの metadata-complete が“true”にセットされているときは、その配備ツールは web-fragment.xml のみを調べ、そのアプリケーションのクラス・ファイルたちの中で出てくる@WebServlet、@WebFilter、及び @WebListener のようなアノテーションたちを無視しなければならない。もし metadata-complete 属性が指定されていないあるいは"false"にセットされているときは、その配備ツールは先に指定したメタデータのためのクラス・ファイルたちを調べねばならない。

以下は Java EE 技術対応ウェブ・コンテナが必要とするアノテーションたちである。

15.5.1 @DeclareRoles アノテーション（@DeclareRoles）

本アノテーションはそのアプリケーションのセキュリティ・モデルを構成するセキュリティ・ロールたちを指定するのに使われる。このアノテーションはクラス上で指定され、アノテートされたクラスのメソッドたちのなかから(即ち isUserInRole を呼ぶことで)テストできるロールたちを指定するのに使われる。@RolesAllowed のなかで使われた結果暗示的に宣言されたロールは@DeclareRoles アノテーションを使って明示的に宣言される必要はない。@DeclareRoles アノテーションは javax.servlet.Servlet インターフェイスまたはそのサブクラスを実装したクラスたちの中でのみ定義されよう。

以下はこのアノテーションがどのように使われるかの例である。
コード例 15-3:@DeclareRoles アノテーション例

```
@DeclareRoles("BusinessAdmin")
      public class CalculatorServlet {
      //...
      }
```

@DeclareRoles ("BusinessAdmin")を宣言することは以下の web.xml で定義されたと等価である。
コード例 15-4:@DeclareRoles web.xml

```
<web-app>
      <security-role>
            <role-name>BusinessAdmin</role-name>
      </security-role>
</web-app>
```

このアノテーションはアプリケーション・ロールたちを他のロールたちと再リンクするのには使われない。そのようなリンキングが必要な場合は、それは関連する配備記述子のなかでしかるべき security-role-ref を定義することで達成される。

アノテートされたクラスから isUserInRole 呼び出しがなされたときは、そのクラスの呼び出しで結び付けられた発信者アイデンティティは isCallerInRole の引数と同じロールのメンバーであるかどうかだテストされる。security-role-ref が role-name の引数のために定義されているときは、その発信者はその role-name にマップされたロールのメンバであるかどうかがテストされる。

@DeclareRoles アノテーションの更なる詳細に関しては、Java™ Platform™ 仕様(JSR 250)の第 2.10 節の「共通アノテーション」を参照されたい。

15.5.2 @EJB アノテーション(@EJB Annotation)

法人 JavaBeans™ 3.0 (EJB)部品たちは@EJB アノテーションを使ってあるウェブ部品から参照されることができる。この@EJB アノテーションは配備記述子の中で ejb-ref または ejb-local-ref 要素を宣言すると等価な動作を提供するものである。対応する@EJB アノテーションを持ったフィールドたちは対応する EJB 部品への参照で流入される。

例:

```
@EJB private ShoppingCart myCart;
```

上のケースでは“myCart”というEJB部品への参照が、その注入を宣言しているクラスが利用可能になる前にプライベートな“myCart”というフィールドの値として注入される。

@EJBアノテーションの振る舞いの更なる詳細はJava™ Platform™ 仕様(JSR 250)の第15.5節を参照されたい。

15.5.3 @EJBsアノテーション(@EJBs Annotation)

@EJBsアノテーションは単一のリソース上でひとつ以上の@EJBの宣言を可能とするものである。

例:
コード例15-5:@EJBsアノテーション例

```
@EJBs({@EJB(Calculator), @EJB(ShoppingCart)})
public class ShoppingCartServlet {
//...
}
```

上記の例ではShoppingCartとCalculatorのEJB部品たちがShoppingCartServletで利用可能になる。ShoppingCartServletはそれでもJNDIを使ってこれらの参照を検索しなければならないが、これらのEJBたちはweb.xmlのなかで宣言される必要はない。

@EJBsアノテーションはEJB 3.0仕様書(JSR220)で更に詳細に記されている。

15.5.4 @Resourceアノテーション(@Resource Annotation)

@Resourceアノテーションはデータ・ソース、Javaメッセージング・サービス(JMS)宛先、あるいは環境エントリのようなリソースへの参照を宣言するのに使われる。このアノテーションは配備記述子の中でのresource-ref、message-destination-refまたはenv-ref、あるいはresource-env-ref要素の宣言と等価である。

@Resourceアノテーションはクラス、メソッド、あるいはフィールド上で指定される。コンテナは@Resourceアノテーションで宣言されたリソースたちへの参照の注入と、それをしかるべきJNDIリソースたちにマッピングすることの責任を持つ。更なる詳細はJava EE仕様の第5章を見られたい。

@Resourceアノテーションの例を以下に示す:
コード例15-6:@Resourceの例

```
@Resource private javax.sql.DataSource catalogDS;
public getProductsByCategory() {
      // get a connection and execute the query
      Connection conn = catalogDS.getConnection();
..
}
```

上記のコード例では、あるサーブレット、フィルタ、あるいはリスナが javax.sql.DataSource 型の catalogDS というフィールドを宣言しており、それに対してそのデータ・ソースへの参照がそのコンテナによりその部品がそのアプリケーションによって使えるようになる前に注入される。データ・ソース JNDI マッピングは“catalogDS”というフィールド名と型(javax.sql.DataSource)から推論される。更に catalogDS というリソースは配備記述子のなかで定義される必要性がもはや無くなる。

@Resource アノテーションのセマンティクスは Java™ Platform™ 仕様書(JSR 250) の第 2.3 節「共通アノテーション」及び Java Ee 仕様書第 5.2.5 節に更に詳細に示されている。

15.5.5 @PersistenceContext アノテーション（@PersistenceContext Annotation）

このアノテーションは参照された永続性ユニットたちのためのそのコンテナが管理するエンティティ・マネージャを指定する。

PersistenceContext アノテーションの例を以下に示す:
コード例 15-7:@PersistenceContext の例

```
@PersistenceContext (type=EXTENDED)
EntityManager em;
```

@PersistenceContext アノテーションの振る舞いは Java Persistence API, Version 2.0 (JSR317)仕様書の第 10.4.1 節に更に詳細に示されている。

15.5.6 @PersistenceContexts アノテーション（@PersistenceContexts Annotation）

PersistenceContexts アノテーションはあるリソース上でひとつ以上の@PersistenceContext の宣言を可能とする。@PersistenceContexts アノテーションの振る舞いは Java Persistence API, Version 2.0 (JSR317)仕様書の第 10.4.1 節に更に詳細に示されている。

15.5.7 @PersistenceUnit アノテーション（@PersistenceUnit Annotation）

@PersistenceUnit アノテーションはあるサーブレット内で宣言された法人 Java ビーンズ部品にエンティティ・マネージャ・ファクトリへの参照を与える。エンティティ・マネージャ・ファクトリは EJB 3.0 仕様書(JSR220)の第 5.10 節で記述されているように別の persistence.xml 設定ファイルで結び付けられる。

@PersistenceUnit の例:

コード例 15-8：@PersistenceUnit 例

```
@PersistenceUnit
EntityManagerFactory emf;
```

@PersistenceUnit アノテーションの振る舞いは Java Persistence API, Version 2.0 (JSR317)仕様書の第 10.4.2 節に更に詳細に示されている。

15.5.8 @PersistenceUnits アノテーション（15.5.8 @PersistenceUnits Annotation）

このアノテーションはひとつ以上の@PersistentUnit アノテーションをあるリソース上で宣言できるようにする。@PersistenceUnits アノテーションの振る舞いは Java Persistence API, Version 2.0 (JSR317) 仕様書の第 10.4.2 節に更に詳細に示されている。

15.5.9 @PostConstruct アノテーション（@PostConstruct Annotation）

@PostConstruct アノテーションは引数を持たないメソッドで宣言され、何らかのチェックされた例外（checked exceptions）をスローしてはいけない。戻り値は void でなければならない。このメソッドはリソース注入が完了した後でおよびその部品のライフサイクル・メソッドが呼び出される前に呼び出されねば**ならない**。

@PostConstruct の例：
コード例 15-9：@PostConstruct 例

```
@PostConstruct
public void postConstruct() {
      ...
}
```

上記の例は@PostConstruct アノテーションを使った例を示す。@PostConstruct アノテーションは依存性注入をサポートする総てのクラスたちによってサポートされねばならず、またたとえそのクラスが注入されるリソースを要求していなくてもサポートされねばならない。もしそのメソッドがチェックされない例外（unchecked exception：バグ等）をスローした時は、そのクラスはサービスを開始してはならず、そのインスタンス上のどのメソッドも呼ばれては**ならない**。

より詳細は Java EE 仕様書の第 2.5 節、及び Java™ Platform™ 仕様書の第 2.5 節「共通アノテーション」を見られたい。

15.5.10 @PreDestroy アノテーション（@PreDestroy Annotation）

@PreDestroy アノテーションはあるコンテナが管理する部品のメソッドで宣言される。このメソッドはそのコンテナが部品を外す前に呼ばれる。
@PreDestroy の例:
コード例 15-10:@PreDestroy 例

```
@PreDestroy
public void cleanup() {
      // オープンなリソースのクリーンアップ
...
}
```

@PreDestroy でアノテートされたメソッドは void を返さねばならず、またチェックされた例外をスローしてはいけない。このメソッドは public、protected、package private、あるいは private であって良い。このメソッドは final であっても良いが static であってはならない。より詳細は JSR 250 の第 2.6 節を参照のこと。

15.5.11 @Resources アノテーション(@Resources Annotation)

Java MetaData 仕様は同じアノテーション・ターゲット上で同じ名前を持った複数のアノテーションを認めていないので、 @Resources アノテーションは複数の@Resource アノテーションたちのコンテナとして機能する。

@Resources アノテーションの例:
コード例 15-11:@Resources アノテーション例

```
@Resources ({
@Resource(name="myDB" type=javax.sql.DataSource),
@Resource(name="myMQ" type=javax.jms.ConnectionFactory)
})
public class CalculatorServlet {
//...
}
```

上の例では JMS 接続ファクトリとデータ・ソースが@Resources アノテーションにより CalculatorServlet が利用可能になっている。@Resources アノテーションのセマンティックスに関しては Java™ Platform™ 仕様書(JSR 250)の第 2.4 節「共通アノテーション」でより詳細に記されている。

15.5.12 @RunAs アノテーション(@RunAs Annotation)

@RunAs アノテーションは配備記述子のなかの run-as 要素と等価である。@RunAs アノテーションは javax.servlet.Servlet インターフェイスを実装したクラスまたはそのサブクラスでのみ定義される。

@RunAs アノテーションの例:

コード例 15-12:@RunAs 例

```
@RunAs("Admin")
public class CalculatorServlet {
@EJB private ShoppingCart myCart;
      public void doGet(HttpServletRequest, req,
HttpServletResponse res) {
      //....
      myCart.getTotal();
      //....
      }
}
      //....
}
```

@RunAs("Admin")ステートメントは web.xml のなかで以下のように定義するのと等価である。

コード例 15-13:@RunAs web.xml 例

```
<servlet>
      <servlet-name>CalculatorServlet</servlet-name>
      <run-as>Admin</run-as>
</servlet>
```

上記の例は myCart.getTotal()メソッドが呼ばれるときに如何にサーブレットが"Admin"というセキュリティ・アイデンティティを EJB 部品に伝搬させるのに@RunAs アノテーションが使われるかを示している。アイデンティティの伝搬の更なる詳細に関しては第 15.3.1 節の「EJB™呼び出しのなかでのセキュリティ・アイデンティティの伝搬」を見られたい。

15.5.13 @WebServiceRef アノテーション(@WebServiceRef Annotation)

@WebServiceRef アノテーションは配備記述子の中の resource-ref 要素と同じように、あるウェブ部品の中であるウェブ・サービスへの参照を提供する。

例:

```
@WebServiceRef private MyService service;
```

この例では"MyService"というウェブ・サービスへの参照がこのアノテーションを宣言しているクラスに注入される。このアノテーションと振舞は JAX-WS 仕様書(JSR 224)第 7 章に更に詳細に記されている。

15.5.14 @WebServiceRefs アノテーション(@WebServiceRefs Annotation)

このアノテーションは単一のリソース上でひとつ以上の@WebServiceRefアノテーションを可能とするものである。このアノテーションと振舞はJAX-WS仕様書(JSR 224)第7章に更に詳細に記されている。

15.5.15 管理されたビーンズとJSR 299の要求事項（Managed Beans and JSR 299 requirements）

管理されたビーンズ(Managed Beans)もサポートする製品においては、実装物はサーブレット、フィルタ、及びリスナとしての管理されたビーンズをサポートしなければならない。 JSR 299もサポートする製品はJSR 299スタイルの管理されたビーンズの使用をサポートしなければならない。またSR-299をサポートする製品においては、実装物は299スタイルの管理されたビーンズをあるアプリケーションのサーブレット、フィルタ、及びリスナのクラスとして使うことをサポートしなければならない。サーブレットのアノテーションたちは直接これらのビーンたちに適用されても良い。JSR-299ではインスタンス化、注入、及び他のサービスに関しこれらのコンテナが管理するビーン・インスタンスの要求事項を規定している。 JSR-299では@Dependent疑似適用範囲(pseudo-scope)、サーブレット、フィルタ、及びリスナたちはその適用範囲内になければ**ならない**と規定している。