改訂サーブレット　チュートリアル
（サーブレット 3.0 及び Tomcat 7 対応）

2010 年 12 月（初版）
2011 年 1 月（一部追加改訂）
2011 年 4 月（第 15 章追加）
2011 年 6 月（一部追加改訂）
株式会社　クレス

サーブレット・チュートリアルを株式会社クレスのサイトにアップロードして以来 9 年間が経過した。その間サーブレットそのものも発展し（当時は 2.2 だったが現在は 3.0 にバージョン・アップされている）、また Struts などのフレームワークによるウェブ・アプリケーション開発が普及しており、開発者がサーブレットの API を使うことも少なくなってきている。また多くのサーブレットのチュートリアルもインターネット上で多数出現している。

しかしながら今回改訂版を作成することにしたのは、単なるサーブレットの基本と使い方を習得するためだけのチュートリアルではなく、もっとその基本動作をきちんと説明し、開発者が自分が開発したアプリケーションは実際はどのようにそのもとになっているプラットホームやネットワーク上で動作しているかが理解できる教材の必要性を感じているからである。従ってこのチュートリアルでは JSP、Struts のようなフレームワーク、及びデータベースなどは取り扱っていない。

開発者たちは優れた開発環境やフレームワークをもとにアプリケーションを開発しているが、そのテスト段階や実際のサービス段階での問題や障害に面したときには、そのような知識が威力を発揮しよう。

従ってこの資料は、既にサーブレットの開発経験があるソフトウエア技術者たちや、より詳しい知識を積み上げたいと考えている学生たちを対象にしている。読者は少なくとも Java プログラミング言語と HTML の基礎知識が必要である。

改定前のチュートリアルは開発環境として IBM の VisualAge for Java を使っていたが、今回はその発展形であり、また広く普及している Eclipse を使用した。またサーブレット・コンテナとしてはサーブレット 3.0 対応の Tomcat の 7.0 版（現時点ではベータ版）を使用しているので、サーブレット 3.0 の理解にも役立とう。OS は Windows である。

サーブレット 3.0 の仕様書の邦訳はこのサイトにアップロードされているので参考にされたい。

このチュートリアルに添付されている tutorial.war、security.war、及び async_request.war という教材アプリケーションのインポートは、「添付 WAR ファイルの Eclipse へのインポート」の節を参考されたい。

この資料は作成途中のものであり、その後追加改定される予定なので、注意されたい。またこの資料に含まれているプログラムを商用に使うことに対しては、当社はその責を負わない。

目次

第 1 章　サーブレット(Servlet)概要.....11
1.1 節　サーブレットとはなにか？.....11
1.2 節　歴史.....14
1.3 節　Servlet 3.0.....15
1.4 節　サーブレットの名前.....16
1.5 節　サーブレットのライフサイクル.....16
1.5.1　サーブレットのライフサイクルのイベント処理.....16
1.5.2　リスナ・クラスの定義.....17
1.5.3　サーブレットのエラーの処理.....18
1.6 節　情報の共有.....18
1.6.1　適用範囲を持ったオブジェクト（スコープ・オブジェクト）の使用.....18
1.6.2　共有資源への同時アクセス制御.....18
1.6.3　共有資源と分散環境.....19
1.7 節　サーブレットの初期化.....19
1.7.1　init メソッドに関する注意.....19
1.7.2　@WebInitParam アノテーション.....20
1.8 節　Service メソッドの記述.....20
1.8.1　要求からの情報の取得（より詳細は「要求オブジェクト」の章で述べる）.....20
1.8.2　応答の組み立て（より詳細は「応答オブジェクト」の章で述べる）.....21
1.9 節　要求と応答のフィルタリング.....22
1.9.1　フィルタのプログラム.....23
1.9.2　カスタム化された要求と応答のプログラミング.....23
1.9.3　フィルタ・マッピングの指定.....24
1.10 節　他のウェブ・リソースの呼び出し.....25
1.10.1　他のリソースを応答に含める.....25
1.10.2　別のウェブ・コンポネントに制御を渡す.....25
1.11 節　ウェブ・コンテキストへのアクセス.....26
1.12 節　クライアント状態の維持.....26
1.12.1　セッションへのアクセス.....26
1.12.2　セッションにオブジェクトを結び付ける.....27
1.12.3　セッション管理.....27
1.12.4　セッション追跡.....27
1.13 節　サーブレットの終了.....27
1.13.1　サービス要求の追跡.....28
1.13.2　メソッドたちにシャットダウンを通知する.....29
1.13.3　長時間がかかるメソッドを礼儀正しいものにする.....29
1.14 節　アノテーション.....30
1.14.1　@WebServlet アノテーション.....30
1.14.2　@WebFilter アノテーション.....30
1.14.3　@WebListener アノテーション.....31

1.14.4 @WebInitParam アノテーション.....31
1.14.5 @MultipartConfig アノテーション.....31
第 2 章 HTTP.....33
2.1 節 HTTP の歴史.....33
2.1.1 HTTP/1.1 とその特徴.....34
2.2 節 ウェブ・ブラウザ.....36
2.2.1 歴史.....37
2.3 節 HTTP メッセージの構成.....38
2.4 節 要求行.....39
2.4.1 HTTP メソッド.....39
2.4.2 要求 URI.....40
2.4.3 プロトコル・バージョン.....45
2.4.4 GET 要求と POST 要求との相違.....45
2.5 節 ステータス行.....46
2.5.1 ステータス・コードと理由句.....46
2.6 節 ヘッダ行.....50
2.6.1 一般ヘッダ.....51
2.6.2 要求ヘッダ.....51
2.6.3 応答ヘッダ.....52
2.7 節 メッセージ・ボディ.....54
2.7.1 MIME エンコーディング.....54
2.7.2 チャンク転送方式.....55
第 3 章 開発環境(Eclipse)とそのインストール.....57
3.1 節 Java EE 6 SDK のインストール.....57
3.2 節 サーブレット 3.0 対応 Eclipse（WTP 3.2）.....58
3.3 節 Eclipse のインストール.....59
第 4 章 Apache Tomcat.....63
4.1 節 Apache Tomcat 概説.....63
4.2 節 歴史.....63
4.3 節 Tomcat の構成要素.....65
4.3.1 Catalina.....65
4.3.2 Coyote.....66
4.3.3 Jasper.....66
4.4 節 Tomcat におけるウェブ・アプリケーションの配備.....66
4.4.1 背景.....66
4.4.2 標準ディレクトリ・レイアウト.....67
4.4.3 共有ライブラリ・ファイル.....67
4.4.4 ウェブ・アプリケーション配備記述子.....68
4.4.5 Tomcat のコンテキスト記述子.....68
4.4.6 Tomcat での配備.....68
4.5 節 ソース・コードの構成.....69
4.5.1 ドキュメント構造.....70

4.5.2 ソース・コードのコントロール....71
4.5.3 BUILD.XML 設定ファイル....72
4.6 節 開発プロセス....73
4.6.1 開発のための Ant と Tomcat の初回設定....74
4.6.2 プロジェクト・ソース・コードのディレクトリを作る....74
4.6.3 ソース・コードとページのエディット....75
4.6.4 ウェブ・アプリケーションのビルド....76
4.6.5 ウェブ・アプリケーションのテスト....76
4.7 節 Tomcat のクラス・ローダ....77
4.7.1 概説....77
4.7.2 クラス・ローダ定義....78
4.7.3 XML パーサと JSE 5 80
第 5 章 Tomcat 7 のインストールと設定....81
5.1 節 環境変数の設定....81
5.2 節 Tomcat のバージョン 7 版をインストール....82
5.3 節 tomcat7 と tomcat7w....85
5.4 節 Tomcat 7 の設定....85
5.4.1 自動再ロード....85
5.4.2 管理者ユーザ名とパスワードの設定....86
5.5 節 DOS レベルでの開発環境の確立....86
5.5.1 新しいアプリケーション(tutorial)をつくる....86
5.5.2 マニュアル・コンパイル用バッチファイルの作成....89
5.5.3 HelloWorld サーブレットの作成とコンパイル....89
5.5.4 HelloWorld サーブレットの実行....90
5.5.5 簡単な JSP の実験....91
5.5.6 自動再ロード機能の確認....92
第 6 章 Eclipse で Tomcat 7 を使う....94
6.1 節 新しいウェブ・アプリケーション(tutorial)を作る....94
6.2 節 新しいサーブレットを作成する....96
6.2.1 ソース・コードの為のパッケージ(basic_package)を用意する....96
6.2.2 新規サーブレット(HelloWorld)を作成する....97
6.2.3 ソース・コードの入力....98
6.3 節 Tomcat 7 の開始と停止....99
6.3.1 Eclipse に Tomcat 7 のことを知らせる....99
6.3.2 Tomcat 7 の開始と停止....100
6.3.3 サーブレットのテスト....101
6.3.4 自動再ロード機能の確認....102
6.4 節 ロギング・ファサードのインストール....103
6.4.1 SLF4J とは....103
6.4.2 SLF4J のパッケージのダウンロードとインストール....104
6.4.3 動作確認....105
第 7 章 要求オブジェクト....107

7.1 節　ブラウザが出す HTTP 要求.....107
7.1.1　MINA プロキシのプログラムで HTTP 要求メッセージを調べる.....107
7.1.2　ブラウザによる相違.....109
7.2 節　HTTP 要求メッセージのデータ取得のためのメソッドたち.....111
7.2.1　要求行に関する情報取得のためのメソッドたち.....112
7.2.2　ヘッダ行に関する情報取得のためのメソッドたち.....113
7.2.3　ボディ部のデータを取り出す為のメソッドたち.....114
7.3 節　HttpRequestDump サーブレットによる要求メッセージの把握.....116
7.3.1　HttpRequestDump のコード.....116
7.3.2　HttpRequestDump を使ってみる.....120
7.3.3　text/plain の問題点（text/plain はブラウザによって異なった結果をもたらす）.....123
7.3.4　URL エンコードされた POST データのストリームやリーダによる読み出し.....125
7.3.5　java.net の URL デコーダの問題.....128
7.4 節　この章のまとめ.....129
第 8 章　応答オブジェクト.....130
8.1 節　Tomcat が送信する HTTP 応答メッセージ.....130
8.1.1　Tera Term による確認.....131
8.2 節　HTTP 応答メッセージの組み立てのためのメソッドたち.....133
8.2.1　ステータス行のためのメソッドたち.....134
8.2.2　ヘッダ行をセットする為のメソッドたち.....134
8.2.3　ボディ部のためのメソッドたち.....138
8.3 節　エラー応答.....139
8.3.1　sendError と setStatus.....140
8.3.2　100 番台のステータス応答.....143
8.3.3　200 番台のステータス応答.....144
8.3.4　300 番台のステータス応答.....144
8.3.5　400 番台のステータス応答.....144
8.3.6　500 番台のステータス応答.....146
8.4 節　リダイレクション(Redirection).....146
8.4.1　sendRedirect メソッド.....146
8.4.2　RedirectionTest サーブレットによる確認.....147
8.5 節　ボディ部への書き込み.....148
8.5.1　PrintWriter による書き込み.....149
8.5.2　ServletOutputStream による書き込み.....152
8.6 節　ブラウザにバイナリ・データを送る.....154
8.6.1　ブラウザに各種ファイルを送る.....154
8.6.2　Office のデータを送る.....156
8.6.3　ブラウザのウィンドウで開けるドキュメント.....156
8.6.4　ファイル・タイプと MIME タイプ.....156
8.7 節　圧縮転送.....160
8.8 節　この章のまとめ.....163
第 9 章　セッション.....164

9.1 節　セッションとは.....164
9.2 節　サーブレットにおけるセッション管理の仕組み.....165
9.2.1　HTTP におけるセッション管理の選択肢.....165
9.2.2　セッション管理のメカニズム.....169
9.2.3　HttpSession オブジェクト.....170
9.3 節　セッションの管理の為のメソッドたち.....171
9.3.1　HTTP 要求のセッションの状態を知る、及び新たなセッションの生成のためのメソッドたち.....171
9.3.2　セッションの情報取得と設定の為のメソッドたち.....172
9.3.3　セッションの属性のセットと取得の為のメソッドたち.....173
9.3.4　セッション ID を応答メッセージに含める為のメソッドたち.....173
9.3.5　イベントとリスナの為のメソッドたち.....174
9.3.6　サーブレット 3.0 でのセッション追跡関係の強化.....175
9.3.7　セッション管理以外の目的でクッキーを使う.....176
9.4 節　簡単なサーブレットでセッション管理のメカニズムを確認する.....180
9.4.1　SimpleSessionTest.java のコード.....180
9.4.2　クッキーを受け付けなくしたブラウザでのセッション確立の確認.....184
9.4.3　ブラウザがクッキーを受け付ける状態でのセッションの確立と維持の確認.....190
9.4.4　ブラウザにおけるクッキーの保持.....192
9.4.5　クッキーをセッションに使わないように Tomcat を設定する.....194
9.5 節　セッション管理のスレッド対応.....197
9.6 節　セッションのイベント処理.....198
9.6.1　メッセージ受付サーブレット.....199
9.6.2　Message クラス.....204
9.6.3　メッセージ問合せサーブレット.....205
9.6.4　MessageManager クラス.....208
9.6.5　言伝サービスの実験.....211
9.7 節　この章のまとめ.....211
第 10 章　スレッド安全.....213
10.1 節　Java における排他制御.....214
10.1.1　同期化(Synchronization).....214
10.1.2　ThreadLocal.....216
10.1.3　Java.util.concurrent パッケージ.....217
10.1.4　Java.util.concurrent.locks パッケージ.....219
10.1.5　Java.util.concurrent.atomic パッケージ.....220
10.2 節　複数スレッドの同時通過の確認.....220
10.3 節　SingleThreadModel.....222
10.4 節　サーブレット内での同期ブロック.....226
10.5 節　スレッド安全でない API への対策.....227
10.5.1　同期メソッドによる対策.....227
10.5.2　同期ブロックによる対策.....228
10.5.3　ThreadLocal を使ってスレッド毎に DateFormat のオブジェクトを持つ方法.....228

10.6 節　デッドロック、枯渇、及びライブロック.....229
10.6.1　デッドロック.....230
10.6.2　タイムアウトによる対策.....231
10.6.3　ウォッチドッグ・タイマ.....232
10.7 節　データベースとスレッド安全.....236
10.8 節　この章のまとめ.....237
第 11 章　サーブレット・コンテキスト.....239
11.1 節　ServletConfig、ServletContext、及び Servlet 間の関係.....240
11.2 節　ServletContext のメソッドたち.....242
11.2.1　初期化パラメタに関するメソッドたち.....242
11.2.2　データ共有の為のメソッドたち.....242
11.2.3　プログラム的なコンテキストへの追加と設定.....243
11.2.4　コンテキスト・パス取得.....245
11.2.5　他のアプリケーションやリソースのアクセス関連メソッドたち.....246
11.2.6　要求ディスパッチャ関連メソッドたち.....247
11.2.7　その他のメソッドたち.....247
11.3 節　要求のフォワードとインクルード.....248
11.3.1　include メソッド.....248
11.3.2　forward メソッド.....249
11.3.3　クエリ文字列の追加.....250
11.3.4　サーブレット間通信.....250
11.3.5　sendRedirect との相違.....250
11.4 節　オブジェクト共有.....251
11.5 節　簡単なサーブレットによる確認.....251
11.5.1　ForwardTest サーブレット.....252
11.5.2　ForwardTarget サーブレット.....254
11.5.3　ForwardTestForm.html.....257
11.5.4　テスト結果例.....257
11.6 節　自動再ロードの問題.....261
11.6.1　自動再ロードで static な要素も再ロードされる.....261
11.6.2　何が起きるのか.....262
11.6.3　対策.....263
11.7 節　この章のまとめ.....264
第 12 章　サーブレット・フィルタ.....265
12.1 節　サーブレット・フィルタの概念.....265
12.2 節　サーブレット・フィルタの為のメソッドたち.....267
12.2.1　Filter インターフェイス.....267
12.2.2　FilterChain インターフェイス.....268
12.2.3　FilterConfig インターフェイス.....268
12.3 節　要求と応答のラップ.....269
12.3.1　汎用応答ラッパ.....269
12.3.2　汎用応答ラッパとそれを使ったフィルタ例のプログラム・コード.....271

12.3.3 PrePost フィルタの実行例.....273
12.4 節 フィルタ・チェイン.....275
12.4.1 アプリケーションの中でのフィルタの設定.....276
12.5 節 フォワードとインクルードの為のフィルタ.....278
第 13 章 非同期処理(Asynchronous Processing).....280
13.1 節 基本的なコンセプト.....280
13.1.1 問題.....280
13.1.2 非同期処理.....281
13.1.3 より具体的なコード例.....283
13.1.4 一般的な非同期処理のシーケンス.....285
13.2 節 非同期動作の為のクラスとメソッドたち.....287
13.2.1 AsyncContext クラス.....287
13.2.2 アノテーション属性.....287
13.2.3 ServletRequest のなかのメソッドたち.....288
13.2.4 非同期リスナ.....288
13.3 節 非同期サーブレットのサンプル.....289
13.3.1 ロング・ポール：株価問合せのサーブレット.....290
13.3.2 ストリーミング：Java EE 6 SDK のチャットのサンプル・アプリケーション.....293
第 14 章 ファイル・アップロード.....299
14.1 節 ブラウザが multipart 送信するときの HTTP 要求メッセージ.....299
14.1.1 ブラウザによる相違.....301
14.1.2 日本語のパラメタ名とファイル名の扱い.....302
14.2 節 新しい API.....302
14.2.1 @MultipartConfig アノテーション.....302
14.2.2 新しいメソッドとクラス.....303
14.2.3 ファイル・アップロードの為の基本的なコード.....304
14.3 節 サンプル・アプリケーション.....304
14.3.1 FileUploadServlet.java.....306
14.3.2 FileUploadTest.jsp.....307
第 15 章 セキュリティ.....309
15.1 節 サーブレットにおけるセキュリティ.....309
15.1.1 認証と認可.....310
15.1.2 レルム、ロール、プリンシパル、クレデンシャル.....311
15.1.3 宣言的(Declarative)セキュリティとプログラム(Programmatic)的セキュリティ.....313
15.2 節 認証メカニズム.....314
15.2.1 HTTP ベーシック認証.....315
15.2.2 HTTP ダイジェスト認証.....318
15.2.3 フォーム・ベース認証.....330
15.2.4 TLS (SSL)と CLIENT-CERT 認証.....341
15.3 節 配備記述子による設定.....345
15.3.1 レルム設定.....345
15.3.2 セキュリティ制約の設定.....347

15.3.3 ログイン設定.....349
15.4 節 アノテーションによる指定.....349
15.5 節 プログラム的なセキュリティ.....351
15.5.1 プログラム的なログインのテスト.....352
15.5.2 プログラム的な認証のテスト.....355
15.6 節 TLS (SSL) 設定.....357
15.6.1 配備記述子による TLS 設定.....358
15.6.2 証明書キーストアを用意する.....358
15.6.3 server.xml の SSL コネクタの設定.....360
15.6.4 DOS 上での Tomcat 7 への SSL アクセス.....360
15.6.5 認証局からの証明書の取得.....361
15.6.6 自己証明書の作成とインポート.....365
15.6.7 Eclipse 上での動作確認.....366
15.7 節 SSL 接続上のサーブレット.....366
15.7.1 SSL セッション追跡モードのセット.....367
15.7.2 SslSessionTest サーブレット.....368
15.7.3 SSL 接続上の認証.....373
第 16 章 参考資料.....378
16.1 節 サーブレット 3.0 の主要クラスのメソッド一覧.....378
16.1.1 サーブレット API の主要クラスとインターフェイス.....378
16.1.2 要求関係.....380
16.1.3 応答関係.....386
16.1.4 セッション関係のクラス、インターフェイス、及びメソッド.....393
16.1.5 コンテキスト関係のインターフェイス、クラス、及びメソッドたち.....397
16.1.6 フィルタ関係.....413
16.1.7 非同期処理関係.....420
16.1.8 ファイル・アップロード関係.....427
16.1.9 セキュリティ関係.....429
16.2 節 MINA プロキシのダウンロードと実行.....441
16.2.1 MINA_Proxy のプロジェクトとパッケージの作成.....441
16.2.2 MINA ライブラリのダウンロードと登録.....442
16.2.3 SLF4J のパッケージのダウンロード.....443
16.2.4 Proxy のプログラムのインポート.....443
16.2.5 プロキシの開始.....444
16.2.6 サーブレットへのプロキシ経由でのアクセス.....445
16.3 節 添付 WAR ファイルの Eclipse へのインポート.....447
16.4 節 Eclipse 上の tutorial アプリケーションの Tomcat 7 への配備.....449
16.4.1 WAR 形式のエクスポート.....449
16.4.2 Tomcat 7 の起動.....449
16.4.3 DOS 上での WAR ファイルの作成.....450
16.5 節 Eclipse 上での security アプリケーションのインストール法.....452
16.5.1 server.xml ファイルの設定.....452

16.5.2 tomcat-users.xml ファイルの設定.....453
16.5.3 証明書のインストールとインポート.....453
16.6 節 Tomcat 7 における SSL 設定.....454
16.6.1 クイック・スタート.....454
16.6.2 SSL 概説.....454
16.6.3 証明書.....455
16.6.4 SSL を稼動させる為の一般的なポイント.....455
16.6.5 設定.....456
16.6.6 認証局からの証明書をインストールする.....459
16.6.7 トラブル・シューティング.....460
16.6.8 自分のアプリケーション内でセッション追跡に SSL を使用する.....461
16.6.9 その他のポイント.....462

第1章　サーブレット(Servlet)概要

本章の1.3節以降は主として、Oracle社のJava EE チュートリアルのサーブレットの部分の筆者による翻訳をベースにしている。ただしウェブ・コンテナとサーブレット・コンテナとは区別して記述してある。読者はこの章を先ず一読されたい。詳細は後で、実習をもとに、なるべく分かりやすく記述する予定である。

この章の参考資料は以下のようである:

- 英語版のサーブレット3.0の仕様書とAPI Javadocのダウンロード:
  1. http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html のページをアクセス
  2. Specificationの"To view the specification for evaluation"の Download をクリック
  3. 仕様書はservlet-3_0-final-spec.pdfのリンクをクリックしてダウンロード
  4. APIのJavadocはservlet-3_0-final-javadoc.zipのリンクをクリックして、適当なフォルダに展開し、自分のブラウザでindex.htmlをアクセス
  5. なお、Apache Org.ではServlet 3 - Tomcat 7として以下のアドレスでAPIのJavadocを公開している:
     http://tomcat.apache.org/tomcat-7.0-doc/servletapi/
- サーブレット3.0仕様書の日本語版:
  http://www.cresc.co.jp/tech/java/Servlet_Specifications/servlet_3-0_about.htm
- Oracleのサーブレット技術の英文サイト:
  http://www.oracle.com/technetwork/java/index-jsp-135475.html
- 英語版Java EE6のチュートリアルのダウンロード:
  http://download.oracle.com/javaee/6/tutorial/doc/bnafi.html
- 英文Wikipedia:
  http://en.wikipedia.org/wiki/Java_Servlet
- 富士通のJ2EE入門のサーブレットの章
  http://jp.fujitsu.com/solutions/sdas/technology/j2ee/02-servlet.html

1.1節　　サーブレットとはなにか？

サーブレットは動的(ダイナミック)なウェブ・アプリケーションのためのJavaベースのプログラムである。これはHTTP要求に応答する為の、JavaのサーブレットAPIを満たすJavaのクラスともいえる。ソフトウエアの開発者はこのサーブレットを使用して、Javaのプラットホームを使ってウェブ・サーバに動的なコンテントを付加する。生成されたコンテントは一般的にはHTMLであるが、XMLのような他の形式のものでもあり得る。

サーブレットは、CGI (Common Gateway Interface)やASP.NET(.NET用Active Server Pages)のような非Javaの動的なウェブ・コンテント技術への、Javaベースの対抗馬でもある。サーブレットはHTTPクッキー、あるいはURL書き換えを使って、サーバの多くのトランザクションにわたってセッション変数としてその状態を維持できる。

javax.servlet及びjavax.servlet.httpパッケージに含まれているservlet APIはサーブレット・コンテナ(ウェブ・コンテナでもあり得る)とサーブレットの関わり合いを決めている。サーブレットと関わり合うサーブレット・コンテナは本質的にウェブ・サーバの要素である。サーブレット・コンテナはサーブレットのライフサイクルの管理、URLと特定のサーブレットとのマッピング、及びそのURLをアクセスしたものが正しいアクセス権を持っていることを確保すること、などの責任を持つ。

サーブレット仕様書では、サーブレット・コンテナは上記の責任を果たすためのサーブレット・エンジンと、管理対象となるサーブレットたちの実行環境（即ちサーブレット、JSP、及び関連クラスのオブジェクト、あるいは静的なドキュメント等が置かれる領域）を含めたものとして使われている。サーブレット仕様書の 1.1 及び 1.2 節では次のように書かれている：

> 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、動的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。
>
> 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。

Tomcat に代表されるサーブレット・エンジンは、単体としてウェブ・サーバとして機能させる（スタンド・アロン）、あるいは Apache のようなウェブ・サーバに付加して（アドオン）使用される。Java EE サーバで代表されるサーバは、ウェブ・サーバのエンジンがサーブレット・エンジンの機能を拡張機能として含めている。

下図はウェブ・サーバとサーブレット・コンテナとの関係を示したものである。

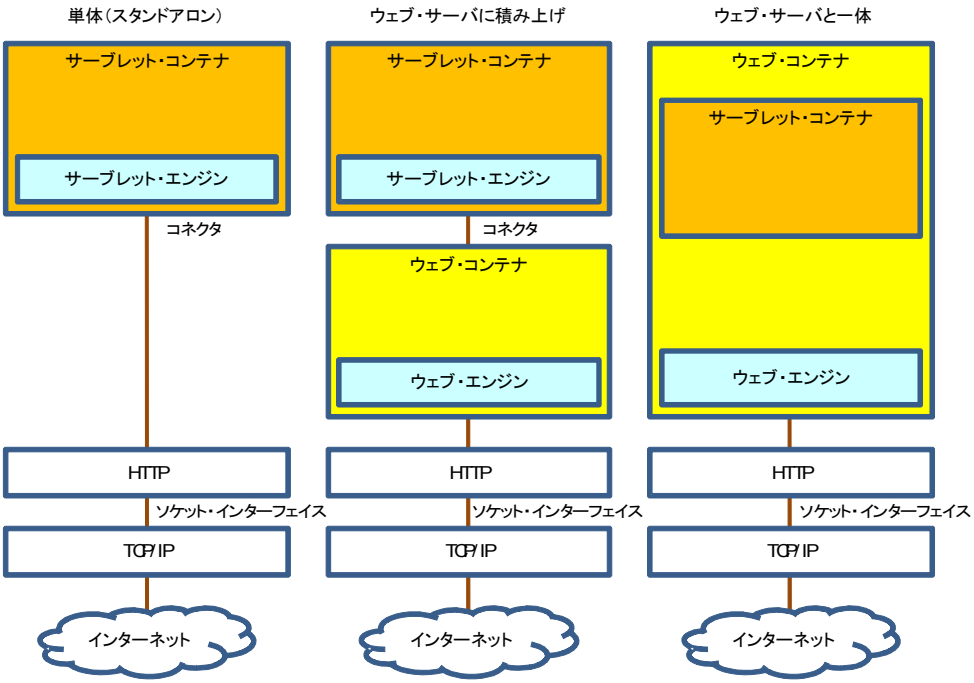


図 1-1：ウェブ・サーバとサーブレット・コンテナとの関係

ウェブ・サーバはインターネットの HTTP プロトコルをベースにしたネットワーク・アプリケーションである。HTTP は、

インターネットの TCP/IP をベースにしたアプリケーション・プロトコルである。TCP/IP と上位アプリケーションは、一般的にはソケット・インターフェイスが使われている。ウェブ・サーバは HTTP プロトコルでクライアントから送信されてきた要求（Request：リクエスト）を受け付け、必要な処理をして応答（Response：レスポンス）をクライアントに送り返す。

サーブレット仕様書では、一般的なイベントのシーケンス事例を次のように書いている：

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。

このようにサーブレットというのはある要求を受け、その要求に基づいて応答を生成させるオブジェクトである。ベーシックな javax.servlet パッケージでは、サーブレットにおける要求と応答を表現する Java のオブジェクトたち、及びその設定パラメタたちと実行環境を反映させるオブジェクトたちを定義している。javax.servlet.http パッケージでは、ジェネリックな servlet 要素たちの HTTP 固有のサブクラスたちを定義しており、これにはウェブ・サーバとあるクライアント間の複数の要求と応答を追跡する為のセッション管理オブジェクトたちが含まれる。サーブレットたちは、あるウェブ・アプリケーションとしてひとつの WAR ファイルにパッケージされ得る。

サーブレットたちはまた、JavaServer Pages (JSP)コンパイラによって自動的に生成される、あるいはその代わりに HTML 生成の為の WebMacro あるいは Apache Velocity のようなテンプレート・エンジンを使って自動的に生成される。またサーブレットたちは、モデル-ビュー-コントローラ(MVC)のパタンである「モデル 2」と呼ばれるパタン（モデルは EJB、画面表示は JSP、制御はサーブレットで構成）で JSP とともによく使用される。

JSP は HTML に Java を組み合わせて動的なウェブ・コンテントを生成する技術である。これはサーブレットの高レベルの抽象化ともいえよう。サーブレットが生成する応答は一般には HTML のページである。しかしリッチで見栄えが良いページは、静的なコンテントの比率が大きい。従って、アプリケーション開発にはむしろ静的なコンテント生成のためのツールで効率良く作成した HTML に、Java による動的なコンテントを組み合わせることが有用である。各 JSP ページはクライアントから最初に呼ばれたときは JSP エンジンによってサーブレットに変換され、サーブレット・コンテナがこれを実行する。JSP エンジンはサーブレット・エンジンに組み入れられていることが多い。例えば Apache Tomcat には Apache Jasper という JSP エンジンが組み込まれている。

サーブレット・コンテナは以上のようにサーブレット技術の核になっており、これをまとめると次のようである：

- サーブレット・コンテナはサーブレット、JSP あるいはその他のコンテナ内のリソースを管理する
- ウェブ・サーバがあるサーブレットの為の HTTP 要求を受けたら、そのウェブ・サーバはその要求をサーブレット・コンテナに渡し、そのサーブレット・コンテナが service（及びそれが展開された doGet、doPost...）メソッドを呼び出すことでそのサーブレットにその要求を渡す
- サーブレット・コンテナは以下の事柄を実施する
  - 通信のサポート：ウェブ・サーバとの通信（スタンドアロンの場合は TCP/IP 網経由で直接クライアントと通信）
  - ライフサイクル管理：サーブレット（あるいは JSP）のロード、インスタンス化、初期化、呼び出し、削除の面倒をみる

- マルチスレッド対応：各サーブレット要求ごとに新しいスレッドを生成（あるいは割当て）
- 宣言的セキュリティ：サーブレットでハード・コードすることなく、XML 配備記述子でセキュリティが設定できる。セキュリティの変更に際しても再コンパイルの必要がない
- JSP 対応：JSP をサーブレットに変換する

- サーブレットがその要求に対する情報を取得し、またクライアントに情報（応答）を送信できるように、サーブレット・コンテナは要求と応答のオブジェクトを生成する

サーブレットの CGI に対する優位性は次のようである：
サーブレットはこれまでの CGI に比べてより高速性能であるとともに使いやすく、また強力である。これまでの Java で書かれた CGI スクリプトは性能面では劣っている。HTTP 要求が発生したときに、各 CGI スクリプトの呼び出し毎に新しいプロセスが生成されている。このプロセス生成のオーバヘッドが、特にそのスクリプトが比較的高速で動作する（プロセス生成がある CGI スクリプトの実行よりも時間がかかる）ときには、システムにとって負担になる。Java サーブレットではそのウェブ・サーバのプロセスのなかで別々の Java スレッドによって各要求が処理されるようにし、HTTP デモンによる別々の子プロセス生成（フォーキング）を無くすことで、この問題を解決している。加えて、同時の CGI 要求により、その要求数分その CGI スクリプトがメモリ上でコピーされまたロードされる。しかしながらサーブレットでは、要求分のスレッドが存在するが、サーブレットのクラスのコピーはただひとつメモリ上にコピーされるだけである。

1.2節　歴史

サーブレットの仕様書は Sun Microsystems 社（現在は買収されて Oracle となっている）によって作られている。バージョン 1.0 は 1997 年 6 月に最終化されている。

バージョン 2.3 からは、Java Community Process. JSR 53 のもとで開発とリリースがされており、このときは Servlet 2.3 と JavaServer Page 1.2 の双方がリリースされた。

その後 JSR 154 がサーブレット 2.4 と 2.5 の仕様書をまとめた。

2010 年の時点では、現在のバージョンは 3.0 となっていて、これは JSR 315 がまとめたものである。

表 1-1：サーブレットのバージョンの経過

サーブレット API のバージョン	リリース	プラットホーム	重要な変更
Servlet 3.0	2009 年 12 月	JavaEE 6, JavaSE 6	プラグ容易性、開発の容易性、非同期サーブレット、セキュリティ、ファイルのアップロード
Servlet 2.5	2005 年 9 月	JavaEE 5, JavaSE 5	JavaSE 5 が必要、アノテーション対応
Servlet 2.4	2003 年 11 月	J2EE 1.4, J2SE 1.3	web.xml で XML Schema 使用
Servlet 2.3	2001 年 8 月	J2EE 1.3, J2SE 1.2	フィルタの追加
Servlet 2.2	1999 年 8 月	J2EE 1.2, J2SE 1.2	J2EE の要素となる、war ファイルの独立したウェブ・アプリケーションたちの導入

Servlet 2.1	1998 年 11 月	指定なし	最初の公式な仕様、RequestDispatcher と ServletContext が追加される
Servlet 2.0		JDK 1.1	Java Servlet Development Kit 2.0 の要素
Servlet 1.0	1997 年 6 月	規定なし	

1.3節　Servlet 3.0

サーブレット 3.0 版は 2009 年 12 月にその仕様書が最終リリースされている最新のバージョンである。従って商用に導入されているのは未だ限定的である。この仕様書の邦訳は当社のサイトにアップロードされているので、参照されたい。

このバージョンはこれまでの 2.2、2.3、2.4、2.5 といったマイナー・バージョン部のアップデートから 3.0 とメージャー・バージョン部のアップデートへと大きく改版されている。Web2.0 時代の強力なコンテナ技術という意味もあってかサーブレット 3.0(Servlet 3.0)と一般的に呼ばれている。サーブレット 3.0 は仕様書の第 15 章でみられるように、Java EE 6 プラットホームの要素でもあることを意識して開発されている。Java EE 6 は 2009 年 12 月にリリースされている。サーブレット 3.0 対応の Tomcat は 7.0 で、2010 年 8 月にベータ版がリリースされている。ベータ版であるので、商用に使うのは避けたほうが良いが、近い将来に安定版になろう。従って、技術者たちはサーブレット 3.0 を良く理解しておくべきであろう。

サーブレット 3.0 の主たる改善ポイントは以下のようである:

- 非同期処理(servise を呼んだスレッドが時間がかかる処理を待たないで帰ることが出来るメカニズム。クライアントからの要求で生成またはプールから取り出されたスレッドの数が、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、あるいはチャットなどアプリケーションのあるイベント、などを待つことで増大し、VM のスタック・メモリ使用量が増大やスレッドの枯渇を起こすだけでなく、サーバ全体の処理能力を落とし、サービス品質(QoS)を劣化させてしまうのを防止する)
- 開発の容易性(EoD とも呼ばれる。サーブレット、フィルタ、リスナ、及びセキュリティ制約を指定するアノテーションが用意されており、web.xml を使わなくてもアプリケーションの設定が可能になっている)
- フレームワーク親和性(ウェブ・アプリケーションの多くは Apache の Wicket、JSF(Java ServerFaces)、Struts などのフレームワークを使って開発されている。フレームワークを使用する際は web.xml の配備記述子に面倒な登録を必要としている。その為 web.xml が複雑で大きなものとなってしまう。サーブレット 3.0 では web.xml をモジュール化して、フレームワークが自分たちの jar ファイルの中にそれを含めることを可能にしている。またプログラム的な設定を可能とする ServletContext のなかの API 及びアノテーションも用意されている。その為にウェブ・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:フラグメントは片割れという意味)の概念が導入されている)
- 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 パタンがプログラム的(動的)に定義可能になった
- ファイル・アップロードが強化された

1.4節　サーブレットの名前

サーブレットには 3 つの名前が存在する:

1. 完全パス名:パッケージ名とクラス名の双方を含む「完全修飾」名である
2. パブリックな URL 名:クライアントがそのサーブレットを呼ぶときに使用する外部に公開する名前である
3. 配備名:これはサーバに配備者(管理者)がそのサーブレットを配備するときに使う内部用の名前であり、XML で書かれた配備記述子(web.xml)の以下の要素たちで使われる:
    - <servlet>:「配備名」を「完全パス名」にマップする
        - <servlet-name>:<servlet>を特定の<servlet-mapping>に結び付ける(クライアントからは見えない)
        - <servlet-class>:".class"拡張子を含まないそのサーブレットの完全修飾名
    - <servlet-mapping>:「配備名」を「パブリックな URL 名」にマップする
        - <servlet-name>:<servlet>に含まれている<servlet-name>と一致をとる
        - <url-pattern>:クライアントが見て使う偽名(パブリックな URL 名)である

サーブレット名をマッピングすることで、そのアプリケーションの柔軟性とセキュリティが改善される。クライアントからはそのアプリケーションの内部構造が見えないようにし、またサーブレットを変更したときにもマッピングの一部の変更だけで良く、変更したサーブレットへの各参照を変更しなくて良くなる。

1.5節　サーブレットのライフサイクル

サーブレットのライフサイクルは次のようなステップで構成される:

1. スタート・アップ時(あるいは最初にそのサーブレットに対する HTTP 要求が到来したとき)に、サーブレット・コンテナがそのサーブレット・クラスをロードする
2. コンテナは引数なしのコンストラクタを呼ぶ
3. コンテナは init()メソッドを呼ぶ。このメソッドはこのサーブレットを初期化し、そのサーブレットが要求の対するサービスを行う前に呼ばれねばならない。あるサーブレットのライフサイクル中に init()メソッドは一度のみ呼び出される
4. 初期化後は、そのサーブレットはクライアントからの要求を受けうけることが出来る。各要求はその要求のスレッドによってサービスされる。コンテナは各要求に対しそのサーブレットの service()メソッドを呼び出す。この service()メソッドは、要求の種類を判断し、しかるべき要求処理のメソッドにそれを振り分ける。サーブレットの開発者はこれらのメソッドの為のしかるべき実装を行わねばならない。実装されていないメソッドへの要求に対しては、親のクラスのメソッドが呼び出され、一般にはエラーがクライアントに返される
5. 最後に、そのコンテナはそのサーブレットのサービス止めるために destroy()メソッドを呼ぶ。このメソッドは init()と同じく、そのサーブレットのライフサイクル中に一度だけ呼び出される。

1.5.1　サーブレットのライフサイクルのイベント処理

リスナのオブジェクトを定義することで、あるサーブレットのライフサイクル内でのイベントを監視しまた反応することが出来る。それらのオブジェクトのメソッドたちは、ライフサイクルのイベントが発生したときに呼び出される。これらのリスナ・オブジェクトを使うには、このリスナ・クラスを定義し、指定しなければならない。イベント・リスナに関

しては、サーブレット仕様書の第 11 章に詳しく説明されているので、詳細はそちらを参照されたい。

1.5.2 リスナ・クラスの定義

あるリスナ・インターフェイスの実装物としてあるリスナ・クラスを定義することになる。下表は監視可能なイベントと、それに対応した実装すべきインターフェイスのリストである。あるリスナのメソッドが呼ばれたら、そのメソッドにはそのイベントに対するしかるべき情報を含んだイベントが渡される。たとえば、HttpSessionListener インターフェイスのメソッドには HttpSession を含んだ HttpSessionEvent イベントが渡される。

表 1-2：イベントとリスナ

オブジェクト	イベント	リスナ・インターフェイスとイベントのクラス
ウェブ・コンテキスト	初期化と廃棄	javax.servlet.ServletContextListener 及び ServletContextEvent
	属性の追加、除去、あるいは置き換え	javax.servlet.ServletContextAttributeListener 及び ServletContextAttributeEvent
セッション	生成、無効化、活性化、受動化、及びタイムアウト	javax.servlet.http.HttpSessionListener、及び javax.servlet.http.HttpSessionActivationListener、及び HttpSessionEvent
	属性の追加、除去、あるいは置き換え	javax.servlet.http.HttpSessionAttributeListener 及び HttpSessionBindingEvent
要求	あるサーブレット要求がウェブ・コンポネントたちによって処理開始された	javax.servlet.ServletRequestListener 及び ServletRequestEvent
	属性の追加、除去、あるいは置き換え	javax.servlet.ServletRequestAttributeListener 及び ServletRequestAttributeEvent

ここで、ウェブ・コンポネントとは、サーブレット、あるいは JavaServer Faces あるいは/及び JSP を使って作られたウェブ・ページ、のことを言う。

そのウェブ・アプリケーションのコンテキスト上でいろんな操作の為にイベントを得る為にあるリスナを定義するには、@WebListener アノテーションを使う。@WebListener アノテーションがついたクラスは、以下のインターフェイスのうちのひとつを実装しなければならない。

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

たとえば、以下のコードはこれらのインターフェイスの 2 つを実装したあるリスナを定義している：

```
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener()
public class SimpleServletListener implements ServletContextListener,
       ServletContextAttributeListener {
    ...
```

1.5.3　サーブレットのエラーの処理

サーブレットの実行中には任意の数の例外が生じ得る。例外が生じたときには、サーブレット・コンテナは以下のメッセージを含むデフォルトのページを生成する:
「サーブレット例外が発生しました(A Servlet Exception Has Occurred)」
しかしながら、プログラマは与えられた例外に対し特定のエラー・ページをそのコンテナが返すように指定することも可能である。

1.6節　　情報の共有

他の殆どのオブジェクトと同じく、ウェブ・コンポネント(サーブレット、あるいは JavaServer Faces あるいは/及び JSP を使って作られたウェブ・ページ、のこと)は通常そのタスクを達成するために他のオブジェクトとともに動作する。その為には幾つかの手段がある。プライベートなヘルパ・オブジェクトの使用(例えば JavaBeans コンポネント)、パブリックな適用範囲(スコープ)の属性であるオブジェクトを共有する、データベースを使用する、そして他のウェブ・リソースを呼び出す、などの手段である。

サーブレットにおいては、あるウェブ・コンポネントが他のウェブのリソースを呼び出すことが出来るメカニズム(インクルードとフォワード)を備えている。(「他のウェブ・リソースの呼び出し」の節で別途説明)

1.6.1　適用範囲を持ったオブジェクト(スコープ・オブジェクト)の使用

リスナ・クラスの定義の節でわかるように、ウェブ・コンテキスト、セッション、及び要求のオブジェクトに属性という形で、サーブレットを含むオブジェクト間での情報の共有が可能である。ウェブ・コンポネント間で情報を共有するには、これらの 4 種類のスコープ・オブジェクトの属性として保持されるオブジェクトを利用する。その適用範囲を示すクラスの「[get|set]属性」のメソッドたちを使って、これらの属性にアクセスが出来る。

下表はこれらのスコープ・オブジェクト(オブジェクトを収容できるという意味でスコープ・コンテナとも呼ばれる)のリストである:

表 1-3:スコープ・オブジェクト

スコープ・オブジェクト	クラス	アクセス可能コンポネント
ウェブ・コンテキスト	javax.servlet. ServletContext	あるウェブ・コンテキスト内のウェブ・コンポネント
セッション	javax.servlet. http.HttpSession	そのセッションに属する要求を処理するウェブ・コンポネント
要求	javax.servlet. ServletRequest のサブタイプ	その要求を処理するウェブ・コンポネント
ページ	javax.servlet. jsp.JspContext	そのオブジェクトを生成する JSP ページ

1.6.2　共有資源への同時アクセス制御

複数スレッドベースのサーバにおいては、共有資源への同時アクセスが生じ得る。セッション及びコンテキストの

スコープ・オブジェクトの属性以外にも、共有資源にはメモリ内データ（インスタンス変数やクラス変数など）と、ファイル、データベース接続、及びネットワーク接続のような外部オブジェクトたちがある。

同時アクセスは幾つかの状況において生じ得る:

- 複数のウェブ・コンポネントがウェブ・コンテキスト内にストアされているオブジェクトにアクセスする
- 複数のウェブ・コンポネントがあるセッション内にストアされているオブジェクトにアクセスする
- あるウェブ・コンポネント内の複数のスレッドがインスタンス変数にアクセスする

ウェブ・コンテナは、各要求を処理する為に一般的には要求ごとにひとつのスレッドを生成する。あるサーブレットのインスタンスが確実にただひとつの要求のみを処理するようにするために、サーブレットは SingleThreadModel インターフェイスを実装することが出来る。もしあるサーブレットがこのインスタンスを実装しているときは、このサーブレットの service()メソッドを 2 つのスレッドが同時に実行することは無くなる。サーブレット・コンテナはこのサーブレットの単一のインスタンスへのアクセスを同期化する、あるいはウェブ・コンポネントのインスタンスのプールを用意して新しい要求は空いているインスタンスに振り向ける、といったことでこの保証を実現できる。このインターフェイスはスタチックなクラス変数、あるいは外部オブジェクトのような共有資源へのウェブ・コンポネントたちによるアクセスで生じる同期化問題を解決することにはならないことに注意が必要である。

共有資源が同時にアクセスされ得る場合には、各々のスレッドの動作結果が意図されたことと一致しなくなる可能性が出る。従って、同期技術を使ってスレッド対策をとる必要が出る。

1.6.3　共有資源と分散環境

配備記述子で<distributable>と指定されたウェブ・アプリケーションでは、ある要求は幾つかの JVM のどれかで処理されることになる。従って、HttpSession のインスタンスは複数の JVM 上に置かれ、それらの同期が必要になる。従って、セッション、及び要求にバインドされるオブジェクトのクラスは、Serializable インターフェイスを実装しておかねばならない。なお「サーブレット・コンテキスト」の章の「オブジェクト共有」の節で示すように、**コンテキストは JVM 固有のものであり、その属性は分散環境下では利用できない**ことに注意しなければならない。

1.7節　　サーブレットの初期化

そのサーブレットのクラスをサーブレット・コンテナがロードしてインスタンス化したあとで、クライアントからの要求をそのサーブレットに渡す前に、サーブレット・コンテナはそのサーブレットの初期化を行う。このプロセスをカスタマイズして、そのサーブレットが初期化データを読み出す、リソースを初期化する、あるいはその他のオンラインの動作をおこなう、などを行うためには、開発者は Servlet インターフェイスの init メソッドをオーバライドする。そのサーブレットが初期化のプロセスを終了できないときは、UnavailableException または ServletException がスローされる。

初期化パラメタの設定と取得は、サーブレット・コンテキストのオブジェクトを介することになる。これらのメソッドに関しては、「サーブレット・コンテキスト」の章の「初期化パラメタに関するメソッド」の項に示されている。

1.7.1　init メソッドに関する注意

init(ServletConfig)メソッドは javax.Servlet インターフェイスで定義されており、サーブレットに対しそのサーブレットがサービスに置かれることを示す為にコンテナが呼び出す。このメソッドは javax.servlet.GenericSrevlet 抽象ク

ラスで実装されている。GenericServlet でのこの実装では、ServletConfig オブジェクトを super.init(ServletConfig) でプライベートな変数としてストアして、あとで getServletConfig()でこのオブジェクトに含まれている初期設定等を読み出せるようにすることを想定している。

開発者がこの init(ServletConfig)メソッドをオーバライドしたときにそのメソッドのなかで super.init(config)を呼ばないときは、この ServletConfig オブジェクトをメンバ変数としてストアしあとでその変数をサーブレットからアクセスするようにしなければならない。この場合は getServletConfig()メソッドではそれは取得できなくなる。

これに対して GenericServlet 抽象クラスでは引数を持たない init()が定義されている。この場合は super.init(config)を呼ばなくても、getServletConfig()メソッドで ServletConfig オブジェクトを取得できるようになる。

1.7.2 @WebInitParam アノテーション

このアノテーションは Servlet または Filter に渡さねばならない初期化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。

使い方は次のようである:

```
@WebServlet(value="/hello",
       initParams = {
           @WebInitParam(name="foo", value="Hello "),
           @WebInitParam(name="bar", value=" World!")
       })
```

そのサーブレットは getInitParameter("foo")のようにその値を読みだすことが可能になる。

1.8節 Service メソッドの記述

サーブレットが提供するサービスは、ある GenericServlet の service メソッド、ある HttpServlet オブジェクトの doMethod メソッド(ここで Method は Get、Delete、Options、Post、または Trace)、あるいは Servlet インターフェイスを実装したあるクラスで用意されている他の何らかのプロトコル固有のメソッドたち、において実装されている。service メソッドという言葉は、クライアントにサービスを提供するサーブレットのクラスのなかのメソッドのことを称している。

service メソッドの一般的なパタンは、要求から情報を取り出し、外部リソースにアクセスし、次にその情報をもとにして応答を作り上げるということになる。HTTP サーブレットの場合は、応答を作り上げるための正しい手続きは、最初に応答から出力ストリームを取得し、次に応答のヘッダ部分を埋め、最後にボディ部分のコンテントをその出力ストリームに書き込む、ことである。**応答ヘッダたちはその応答がコミットされる(ネットワークに送出される)前に必ずセットされねばならない**。その応答がコミットされた後でヘッダ部分をセットしたり付加したりしようとすると、サーブレット・コンテナはそれを無視する。

1.8.1 要求からの情報の取得(より詳細は「要求オブジェクト」の章で述べる)

要求にはクライアントとサーブレット間で渡されるデータが含まれている。総ての要求は ServletRequest インターフェイスを実装している。このインターフェイスには、以下の情報をアクセスする為のメソッド群が定められている:

- パラメタたち、一般的にはクライアントとサーバ間の情報を伝えるのに使われる
- オブジェクトのかたちの属性、一般的にはサーブレット・コンテナとサーブレットまたはともに動作するサーブレットたち間で情報を渡すのに使われる
- その要求を通信する為に使われたプロトコル、及びその要求に関わっているクライアントとサーバに関する情報
- ローカリゼーションに関わる情報

また要求から入力ストリームを取得し、またマニュアルでデータを分析できる。文字データを読み出すときは、その要求の getReader メソッドで返される BufferedReader オブジェクトを使用する。バイナリ・データを読み出すときは getInputStream で返される ServletInputStream を使用する。

HTTP サーブレットは HTTP 要求オブジェクト HttpServletRequest が渡され、これには要求 URL、HTTP ヘッダ、クエリ文字列、等々が含まれている。

HTTP 要求 URL は以下の要素で構成される:

http://[host]:[port][request-path]?[query-string]

ここに、host はホスト名、port は TCP ポート番号、request-path は要求パス、query-string はクエリ文字列である。

要求パスは更に以下の要素で構成されている:

- コンテキスト・パス:これは斜線 (/) で連結されたそのサーブレットのウェブ・アプリケーションのコンテキスト・ルートである
- サーブレット・パス:その要求が起動させるコンポーネント・エイリアスに対応したパス区間である
- パス情報:コンテキスト・パスの要素でもサーブレット・パスの要素でもない要求パスの部分

例えばコンテキスト・パスが/catalog で、表 1-4 のようなエイリアスのときは、如何に URL が分析されるかをその下の表 1-5 で示す。

表 1-4:エイリアス(別名)

パタン	サーブレット
/lawn/*	LawnServlet
/*.jsp	JSPServlet

表 1-5:要求パス要素

要求パス	サーブレット・パス	パス情報
/catalog/lawn/index.html	/lawn	/index.html
/catalog/help/feedback.jsp	/help/feedback.jsp	null

要求のサーブレットのマッピングに関しては、サーブレット 3.0.仕様書の第 12 章を見られたい。

クエリ文字列は、パラメタ名と値のセットで構成されている。個々のパラメタは getParameter メソッドを使って要求から取り出される。クエリ文字列の生成には 2 つの手段がある:

- クエリ文字列が明示的にウェブ・ページの中に出現している
- GET HTTP メソッドをもったフォームのかたちで出されたときにクエリ文字列が URL に付加される

1.8.2 応答の組み立て(より詳細は「応答オブジェクト」の章で述べる)

応答にはサーバとクライアント間で渡されるデータが含まれる。総ての応答は ServletResponse インターフェイスを実装している。このインターフェイスは、以下のことが可能なメソッド群を用意している:

- クライアントにデータを送信する為に使う出力ストリームの取得。文字データを送信するには、この応答

の getWriter メソッドで返される PrintWriter を使用する。MIME のボディ部をもった応答でバイナリ・データを送信するには、getOutputStream メソッドで返される ServletOutputStream を使用する。バイナリとテキストが混ざったデータを送信する(マルチパート応答で)ときは、ServletOutputStream を使用して文字の区間はマニュアルで設定する。

- 応答によって送り返されるコンテントのタイプ(例えば text/html)は、setContentType(String) をつかって指定する。このメソッドはその応答がコミットされる(ネットワークに送出される)前に呼ばねばならない。コンテント・タイプ名のレジストリは Internet Assigned Numbers Authority (IANA)が管理していて、http://www.iana.org/assignments/media-types/ を参照のこと。
- 出力を setBufferSize(int)メソッドでバッファリングするかどうかを指定する。デフォルトでは、出力ストリームに書き込まれたコンテントは即座にクライアントに送信される。バッファリングをすることで、クライアントに応答が送信される前にコンテントをすべて書き込むことが出来るので、サーブレットはしかるべきステータス・コードとヘッダをセットしたり、別のウェブのリソースに転送する時間を得ることが出来る。このメソッドは何らかのコンテントが書き込まれる、あるいは応答がコミットされる前に呼び出されねばならない。
- ロケールと文字エンコーディングのようなローカリゼーションの情報をセットする。

javax.servlet.http.HttpServletResponse という HTTP 応答のオブジェクトは、以下のような HTTP ヘッダを表現するフィールドを有している:

- ステータス・コード、これらは要求に対応できない理由を、あるいはリダイレクトされたことを示すのに使われる
- クッキー(cookie)、これはクライアント側でアプリケーション固有の情報をストアするのに使われる。場合によってはクッキーはユーザのセッションを追跡する為に識別子保持のために使われる

1.9節　要求と応答のフィルタリング

フィルタの詳細は、この資料の「フィルタ」の章、及びサーブレット3.0 仕様書の第6 章に詳しく記載されているので、より正確な内容はそちらを参照されたい。

フィルタというのは要求または応答(あるいは双方)のヘッダ部とボディ部を変換できるオブジェクトである。フィルタとウェブ・コンポネントとの相違は、フィルタは通常はそれ自身は応答を作らないことである。フィルタは応答を作らない代わりにウェブ・リソースの類に「貼り付け」される機能を持っている。

従って、フィルタはそれに対しフィルタとして機能しているウェブ・リソースへの依存性を持ってはならない;これはひとつ以上のウェブ・リソースのタイプで構成される。

フィルタが実行できる主たるタスクは次のようである:

- 要求を調べそれに従ってアクションをとる
- 要求と応答の対がこれ以上通過するのを阻止する
- 要求のヘッダとデータに手を加える。これはカスタム化された要求のバージョンをわたすことでなされる
- 応答のヘッダとデータに手を加える。これはカスタム化された応答のバージョンを渡すことでなされる
- 外部リソースと関わり合う

フィルタの応用としては、認証、ロギング、イメージ変換、データ圧縮、暗号化、ストリームのトークン化、XML 変換、等々がある。

あるウェブ・リソースがゼロ、ひとつ、あるいはそれ以上のフィルタたちによってある順番でフィルタリングされるようにそのリソースを設定できる。このチェインは、そのウェブ・コンポネントを含むアプリケーションが配備されるとき

に指定され、サーブレット・コンテナがそのコンポネントをロードするときにインスタンス化される。

要約すると、フィルタ使用に際のタスクとしては:

- そのフィルタをプログラムする
- カスタム化された要求と応答をプログラムする
- 各ウェブ・リソースにたいしそのフィルタ・チェインを指定する

1.9.1 フィルタのプログラム

フィルタのための API は javax.servlet パッケージの Filter、FilterChain、及び FilterConfig インターフェイスで用意されている。フィルタは Filter インターフェイスを実装することで作ることが出来る。

このインターフェイスのなかで最も重要なメソッドは doFilter で、これには要求、応答、及びフィルタ・チェインのオブジェクトが渡される。このメソッドは以下のアクションをとることが出来る:

- 要求ヘッダを調べる
- もしそのフィルタが要求ヘッダまたは要求データを加工したいときは、要求オブジェクトをカスタマイズする
- もしそのフィルタが応答ヘッダまたは応答データを加工したいときは、応答オブジェクトをカスタマイズする
- そのフィルタ・チェインの次のものを呼び出す。もしそのフィルタがターゲットのウェブ・コンポネントまたは静的リソースで終わるフィルタ・チェインの最後のフィルタである場合は、次のものはそのチェインの最後にあるリソースになり、そうでない場合には次のフィルタは WAR のなかで設定されている次のフィルタになる。そのフィルタはそのチェインのオブジェクト上の doFilter メソッドを呼ぶ(自分が呼ばれたときの要求と応答をそのまま渡す、あるいはそれが作ったラップされた要求と応答のバージョンを渡す)ことで、次のものを呼び出す。別のアクションとしては、そのフィルタは次のものを呼び出さないでその要求をブロックすることもできる。後者の場合は、そのフィルタは自分で応答を作り上げねばならない
- そのチェインの次のフィルタを呼び出した後で、応答のヘッダを調べる
- 処理中のエラーは、それを示すために例外をスローする

doFilter に加えて、init と destroy のメソッドも実装しなければならない。init メソッドはそのフィルタインスタンス化されたときにサーブレット・コンテナが呼び出す。初期化パラメタをフィルタに渡したいときは、init に渡された FilterConfig オブジェクトからそれを取得できる。

1.9.2 カスタム化された要求と応答のプログラミング

フィルタが要求あるいは応答を加工する多くの手段が存在する。例えば、フィルタはその要求に属性を付加したり、あるいは応答にデータを挿入することが出来る。

応答に加工するフィルタの場合は、通常その応答をクライアントに返す前にその応答を捕捉しなければならない。その為には、その応答を発生させるサーブレットに対し代役のストリームを渡さねばならない。その代役のストリームにより、そのサーブレットがそれを終わらせたときにオリジナルの応答のストリームを閉じてしまうことを防止し、そのフィルタがサーブレットの応答に手を加えることを可能とする。

代役のストリームをサーブレットに渡すために、そのフィルタはこの代役のストリームを返すために getWriter または getOutputStream メソッドをオーバライドする応答ラッパ(response wrapper)を生成する。このラッパがフィルタ・チェインの doFilter メソッドに渡される。ラッパのメソッドたちはデフォルトではラップされた要求または応答オブジェクトを呼ぶようになっている。

要求のメソッドたちをオーバライドするには、ServletRequestWrapper あるいは HttpServletRequestWrapper を継承

したオブジェクトの中の要求をラップする。応答のメソッドたちをオーバライドするには、ServletResponseWrapper あるいは HttpServletResponseWrapper を継承したオブジェクトの中の要求をラップする。

1.9.3　フィルタ・マッピングの指定

フィルタたちをどのようにウェブ・リソースたちにマッピングするかを判断するのに、サーブレット・コンテナはフィルタ・マッピング(filter mappings)を使用する。フィルタ・マッピングは名前によってあるウェブ・コンポネントとの、あるいは URL パタンによってウェブ・リソースとのマッチングをとっている。フィルタたちは WAR に含まれているフィルタ・マッチング・リストのなかでフィルタ・マッチングが出現する順番で呼び出される。ある WAR に対するフィルタ・マッピング・リストは配備記述子の中で指定され、これは NetBeans や Eclipse のような IDE によって、あるいは他のエディタで行うことになる。

ハンド・コーディングによる XML 指定では、以下のステップでウェブ・アプリケーション配備記述子の XML を直接編集することで、フィルタの宣言、マッピング、及び制約を行うことが出来る。

1. Eclipse のエディタで web.xml を開く
2. display-name 要素の直後 filter 要素を追加する。filter 要素は、そのフィルタの名前、そのフィルタの実装クラス、及び初期化パラメタを含む
3. filter-mapping 要素を使って名前または URL パタンを使ってそのフィルタとウェブ・リソースとのマッピングを行う
4. dispatcher オプションを使ってそのフィルタが要求に対しどのように適用されるかを指定する

もしあるウェブ・アプリケーションへの要求のログをとりたいときには、ヒット・カウンタ・フィルタを/*という URL パタンにマッピングすれば良い。

あるフィルタを複数のウェブ・リソースにマップする、あるいは複数のフィルタをあるウェブ・リソースにマップすることも可能であり、その例を下図に示す。この例では、フィルタ F1 が S1、S2、及び S3 の 3 つのサーブレットにマッピングされており、フィルタ F2 はサーブレット S2 に、そしてフィルタ F3 が S1 と S2 の 2 つのサーブレットのマッピングされている。

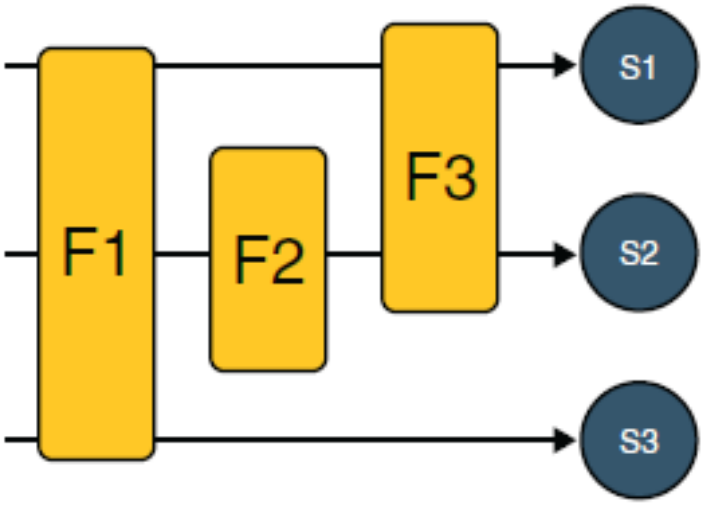


図 1-2：フィルタのマッピング例

1.10節　他のウェブ・リソースの呼び出し

この節の詳細は、このチュートリアルの「サーブレット・コンテキスト」の章、あるいはサーブレット仕様書の第9章の「要求のディスパッチ」を参照されたい。

ウェブ・コンポネントたちは間接的と直接的の2つのやり方で他のウェブ・リソースを呼び出すことが出来る。ウェブ・コンポネントはクライアントに返されるコンテントのなかの別のウェブ・コンポネントを示すURLを埋め込むことで、別のウェブ・リソースを間接的に呼び出す。また、ウェブ・コンポネントは実行中に直接的に別のリソースを直接的に呼び出すことが出来る。そのウェブ・コンポネントは別のリソースのコンテントを含める(include)する、あるいはその要求を別のリソースに引き渡す(forward)ことが出来る。

あるウェブ・コンポネントを走らせているサーバ上で取得可能なリソースを呼び出しには、最初にgetRequestDispatcher("URL")メソッドを使って、あるRequestDispatcher(要求ディスパッチャ)オブジェクトを取得しなければならない。RequestDispatcherオブジェクトは要求とウェブ・コンテキストのどちらからでも取得できるが、これらの2つの方法はその振る舞いには少し相違がある。要求は相対パス(即ち"/"で始まらないもの)をとり得るが、ウェブ・コンテキストの場合は絶対パスを必要とする。そのリソースが得られない場合、あるいはそのサーバがそのタイプのリソースのためのRequestDispatcherを実装していない場合は、getRequestDispatcherはnullを返す。サーブレットはそのような条件に対処できるように備えていなければならない。

1.10.1　他のリソースを応答に含める

あるウェブ・コンポネントから返される応答のなかに別のリソース(例えばバナー・コンテントあるいは著作権情報)を含める(includeする)ことは有用である。別のリソースを含める為には、あるRequestDispatcherオブジェクトのincluneメソッドを呼ぶ:

```
include(request, response);
```

もしそのリソースが静的なものであるときは、このincludeメソッドでプログラム的なサーバ・サイド・インクルートがなされることになる。そのリソースがウェブ・コンポネントの場合は、このメソッドは含める側のサーブレットからの応答の中に実行結果を含める。インクルードされたウェブ・コンポネントは要求オブジェクトへのアクセスを持つが、その応答オブジェクトに対し出来ることが制限される:

- そのウェブ・コンポネントは応答のボディ部分に書き込むこと、及び応答をコミットすることが出来る
- そのウェブ・コンポネントはヘッダをセットできないし、その応答のヘッダ部に影響を与えるメソッド(例えばsetCookie)を呼ぶことはできない

1.10.2　別のウェブ・コンポネントに制御を渡す

アプリケーションによっては、ひとつのウェブ・コンポネントをある要求の前処理の為に使い、別のコンポネントに応答を生成させたい場合もあろう。例えば、ある要求の部分処理を行って、次にその要求の内容によって別のコンポネントに制御を渡したいこともあろう。

制御を別のウェブ・コンポネントに移すためには、RequestDispatcherオブジェクトのforwardメソッドを呼び出す。その要求がフォワードされたら、要求URLはフォワードされたページのパスにセットされる。オリジナルのURIとその構成要素は要求の属性の

```
javax.servlet.forward.[request-uri|context-path|servlet-path|path-info|
query-string]
```

としてストアされる。

この forward メソッドは別のリソースに対し、ユーザに対し応答する責任を持たせるために使われねばならない。既にそのサーブレット内で ServletOutputStream または PrintWriter のオブジェクトをアクセスしてしまっているときは、このメソッドは使用できない。そうすれば IllegalStateException がスローされる。

1.11節　ウェブ・コンテキストへのアクセス

ウェブ・コンテキストの詳細は「サーブレット・コンテキスト」の章で述べてある

そのなかでウェブ・コンポネントたちが実行するコンテキストは、ServletContext インターフェイスを実装したオブジェクトである。このウェブ・コンテキストを取得するには、getServletContext メソッドを使用する。ウェブ・コンテキストにより、以下のものたちへのアクセスが可能となる:

- 初期化パラメタ
- そのウェブ・コンテキストに関わるリソース
- オブジェクトとして貼り付けられている属性
- ロギング機能

ロッギングで使われているカウンタのアクセスのためのメソッドたちは、同時に実行しているサーブレットたちによる不適正な操作を防ぐために同期化されている。フィルタはこのコンテキストの getAttribute メソッドを使って counter オブジェクトを取得している。このカウンタのインクリメントされた値はログに記録される。

1.12節　クライアント状態の維持

セッションの更なる詳細は、本チュートリアルの「セッション」の章に記されている。

多くのアプリケーションでは、あるクライアントからの一連の要求が互いに結び付けられていることが求められる。例えば、あるウェブ・アプリケーションでは、その要求たちにわたってユーザのショッピング・カートの状態を記録できる。HTTP は状態なし（ステートレス：即ち要求/応答で終了し、前回の状態は記憶されない）のプロトコルであるので、ウェブ・ベースのアプリケーションではそのような状態（セッションと呼ばれる）を維持しなけえばならない。状態維持が必要なアプリケーションに対応するように、Java サーブレット技術ではセッション管理の為の API が用意されており、セッションの実装に幾つかのメカニズムが使えるようにしている。

1.12.1　セッションへのアクセス

セッションは HttpSession オブジェクトで表現されている。セッションにアクセスするには request オブジェクトの getSession メソッドを呼ぶ。このメソッドはその要求に関わる現在のセッションを返す、あるいはその要求がセッションを持っていないときは、セッションを生成する。

1.12.2 セッションにオブジェクトを結び付ける

オブジェクトの形の属性を、名前でセッションに結び付けることが出来る。そのような属性は、同じウェブ・コンテキストに属し、同じセッションの一部である要求を処理しているどのウェブ・コンポネントからもアクセスできる。

1.12.2.1 セッションに結び付けられたオブジェクトの通知

サーブレットのライフサイクルの節で説明したように、アプリケーションはウェブ・コンテキストとセッションのリスナたちにサーブレットのライフサイクルのイベントを通知出来る。また以下のようなセッションとの結び付けに関わるイベントを通知できる:

- そのオブジェクトがあるセッションに付加された、あるいは外された。この通知を受けるには、そのオブジェクトは javax.servlet.http.HttpSessionBindingListener インターフェイスを実装しなければならない。
- そのオブジェクトが貼り付けられていたセッションが受動化あるいは能動化された。セッションは VM 間を移動したとき、あるいは継続的蓄積にストアされたあるいは呼びもどされたときに受動化あるいは能動化される。この通知を受けるには、そのオブジェクトは javax.servlet.http.HttpSessionActivationListener インターフェイスを実装しなければならない。

1.12.3 セッション管理

HTTP ではクライアントがもはやセッションが不要になったと知らせる手段がないので、各セッションはそのリソースが利用可能なタイムアウトを有している。このタイムアウト期間はセッションの[get|set]MaxInactiveInterval メソッドを使ってアクセスできる。また配備記述子を使ってタイムアウト期間をセットできる。タイムアウト整数で指定し、分単位となる。

アクティブなセッションがタイムアウトにならないようにするには、一般には service メソッドを使ってそのセッションをアクセスする。なぜならこのメソッドはセッションのタイムアウトのライブ・カウンタをリセットするからである。

あるクライアントとの関わり合いが終了したときには、そのセッションの invalidate メソッドを使ってサーバ・サイドのそのセッションを無効化し、セッション・データ(存在すれば)を除去する。

1.12.4 セッション追跡

サーブレット・コンテナは、あるユーザにあるセッションを結び付けるために幾つかの手段を使うが、そのすべてがクライアントとサーバ間の識別子を渡す。この識別子はクッキーとしてクライアント上に保持される、あるいはウェブ・コンポネントがクライアントに返す各 URL に含めることが出来る。

アプリケーションが session オブジェクトを使うときは、クライアントがクッキーをオフにしたときはアプリケーションが URL 書き換えを行ってセッション追跡が確実に可能なようにしなければならない。その為には、サーブレットが返す総ての URL に対し、応答の encodeURL(URL)メソッドを呼ぶ。このメソッドはクッキーが使えないときに限り URL にセッション ID を含める、そうでないときは URL を変更しないで返す。

1.13節 サーブレットの終了

サーブレット・コンテナがあるサーブレットを外すべきと判断したとき(たとえば、メモリ資源を再割り当てしたいとき、あるいはシャットダウン中のとき)は、そのコンテナはServlet インターフェイスのdestroy メソッドを呼ぶ。このメソッドの中でサーブレットが使用しているリソースを開放し、継続中の状態があればそれを保存することになる。destroy メソッドはinit メソッドで生成されたデータベース・オブジェクトを開放する。

サーブレットが除去されるときは、サーブレットの総てのservice メソッドは終了されねばならない。そのためサーバは総てのサービス要求が返された、あるいはサーバ固有の猶予時間後(どちらかが最初に発生した後)においてのみdestroy メソッドを呼ぶ。もし自分が設計したサーブレットが実行に長時間がかかる操作を有しているときは(即ちサーバの猶予時間よりも操作に時間がかかるときは)、destroy が呼ばれても動作を継続させることが可能である。その為には、どのクライアントのスレッドも要求の処理を継続できるようにしなければならない。以下のことをどのように行うかをこの後で説明する。

- どれだけの数のスレッドがservice メソッドで現在走っているのかを追跡する
- destroy メソッドに対し、長時間実行中のスレッドに対しシャットダウンを通知させ、それらのスレッドが完了するのを待つことで、クリーンなシャットダウンを行うようにする
- 長時間実行中のスレッドに対し定期的にポーリングをかけ、シャットダウンをチェックし、必要なら動作を停止、クリーンアップ、及びリターンさせる

1.13.1 サービス要求の追跡

サービス要求たちを追跡するためには、実行中のservice メソッドの数をカウントするフィールドを自分のサーブレットのクラスに含める。このフィールドは、インクリメント、デクリメント、及び値を返すためのアクセス・メソッドを同期化させねばならない。

```
public class ShutdownExample extends HttpServlet {
	private int serviceCounter = 0;
	...
	// Access methods for serviceCounter
	protected synchronized void enteringServiceMethod() {
		serviceCounter++;
	}
	protected synchronized void leavingServiceMethod() {
		serviceCounter--;
	}
	protected synchronized int numServices() {
		return serviceCounter;
	}
}
```

service メソッドはこのメソッドに入ったときに毎回サービス・カウンタをインクリメントしなければならず、このメソッドを出たときに毎回デクリメントしなければならない。これはプログラマが自分のHttpServlet のサブクラスがservice メソッドをオーバライドする数少ない事例である。以下のように、新しいメソッドはsuper.service を呼んで、オリジナルのservice メソッドの機能(HTTP メソッドによる振分け)を保持しなければならない:

```
protected void service(HttpServletRequest req,
	HttpServletResponse resp)
	throws ServletException,IOException {
		enteringServiceMethod();
		try {
		super.service(req, resp);
		}
		finally {
		leavingServiceMethod();
		}
}
```

1.13.2 メソッドたちにシャットダウンを通知する

確実にクリーンなシャットダウンをさせるために、開発者が作った destroy メソッドは、総てのサービス要求が完了するまでは、共有リソースを開放してはならない。それを行うことの要素はサービス・カウンタをチェックすることである。もうひとつの要素は長時間は知っているメソッドたちに、シャットダウンする時間だということを通知することである。この通知の為にはもうひとつのフィールドが必要になる。このフィールドは通常のアクセス・メソッドを持っていなければならない:

```
public class ShutdownExample extends HttpServlet {
	private boolean shuttingDown;
	...
	// シャットダウンの為のアクセス・メソッドたち
	protected synchronized void setShuttingDown(boolean flag) {
		shuttingDown = flag;
	}
	protected synchronized boolean isShuttingDown() {
		return shuttingDown;
	}
}
```

以下はクリーンなシャットダウンを行う為の、これらのフィールドを使った destroy メソッドの例である:

```
public void destroy() {
	/* 未だサービス・メソッドが存在しているかどうかをチェックする /*
	/* 走っている、それらに停止するよう告げる */
	if (numServices() > 0) {
	setShuttingDown(true);
	}
	/* それらのサービス・メソッドが停止するのを待つ */
	while(numServices() > 0) {
	try {
	Thread.sleep(interval);
	} catch (InterruptedException e) {
		}
	}
}
```

1.13.3 長時間がかかるメソッドを礼儀正しいものにする

クリーンなシャットダウンを実現する為の最後のステップは、長い時間がかかる可能性があるメソッドたちを礼儀正しく振舞わせることである。長い時間がかかる可能性があるメソッドたちは、シャットダウンを通知しているフィールドの値をチェックし、また必要なら自分たちの作業に割り込みをかけるようにしなければならない:

```
public void doPost(...) {
	...
	for(i = 0; ((i < lotsOfStuffToDo) &&
	!isShuttingDown()); i++) {
	try {
	partOfLongRunningOperation(i);
	} catch (InterruptedException e) {
	...
		}
	}
}
```

1.14節　アノテーション

*サーブレット3.0 では、サーブレット、フィルタ、及びリスナを配備記述子(web.xml)を使わなくてもアノテーションによって配備出来るようになっている。これはEoD (Ease of Development : 簡単開発) 化の一環である。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" 属性が含まれている。もし"metadata-complete" が"true" にセットされていると、サーブレット・コンテナ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サーブレットの何らかのアノテーションを無視する。従って**アノテーションが有効である為には、配備記述子の"metadata-complete" が必ず"false" にセットされていなければならないことに注意**されたい。*

1.14.1　@WebServlet アノテーション

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。

以下は属性のリストである。詳細は Javadoc を見て頂きたい:

- name : そのサーブレットの名前
- description : そのサーブレットの記述
- value : そのサーブレットの URL パタン
- urlPatterns : そのサーブレットの URL パタンたち
- initParams : そのサーブレットの初期化パラメタたち
- loadOnStartup : そのサーブレットをスタートアップ時にロードする順位
- asyncSupported : そのサーブレットが非同期処理に対応することを宣言
- smallIcon : そのサーブレットの為の小さなアイコン
- largeIcon : そのサーブレットの為の大きなアイコン

一般的な記述は次のようである:

最もシンプルな記述:

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

より多くの属性を使った記述:

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

   }
}
```

1.14.2　@WebFilter アノテーション

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。

@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。

以下は属性のリストである。詳細は「サーブレット・フィルタ」の章の「アノテーションによる設定」の節、及び Javadoc を見て頂きたい:

- filterName:そのフィルタの名前
- description:そのフィルタの記述
- displayName:そのフィルタの表示名
- initParams:そのフィルタの初期化パラメタたち
- servletNames:ターゲットとなるサーブレットの名前
- value:そのフィルタが適用される URL パタン
- urlPatterns:そのフィルタが適用される URL パタンたち
- dispatcherTypes:そのフィルタが適用するディスパッチャのタイプ
- asyncSupported:そのフィルタが非同期処理を行うことを宣言

以下はその記述例である:

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

1.14.3 @WebListener アノテーション

リスナに関しては「リスナ・クラスの定義」の項で既に述べてある。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/*" });
   }
}
```

1.14.4 @WebInitParam アノテーション

このアノテーションは「サーブレットの初期化」の節で既に述べてある。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。

1.14.5 @MultipartConfig アノテーション

これは「ファイル・アップロード」の章で説明してあるので、そちらを見て頂きたい。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。

第2章　HTTP

ウェブ・サーバやウェブ・アプリケーションは、インターネットのHTTP (Hyper Text Transfer Protocol)というアプリケーション・プロトコルに依存している。従ってサーブレットの理解には、HTTP 及びそれが運ぶHTML テキストの知識が不可欠である。HTTP に関しては、日本語の優れた技術情報のサイトが存在するので、詳細はそちらを参照されたい。ここではサーブレット開発にとって重要な事項を重点的に概説したい。

HTTP はその名の通り HTML で記述されたハイパー・テキストをウェブ・サーバとウェブ・ブラウザ間で転送する為のプロトコルである。しかしながら HTTP は当初のハイパー・テキストのためだけではなく、多くの他のタスクにも使用できる。HTTP そのものはその下位層の TCP と違って、状態を持たない(ステートレス)のプロトコルであり、クライアントが要求をサーバに送信し、サーバが応答を返し要求されたデータ(HTML ドキュメントのような)を、あるいはエラー・コードを返し、その後接続を閉じる(HTTP/1.0 の場合)。一旦応答を返したら、サーバはそのクライアントのことを何も記憶しない。これは当初はサーバが静的なドキュメントのみを返すシステムを想定し、また当時は貧弱だったネットワーク上の輻輳を回避することを意図していたからである。

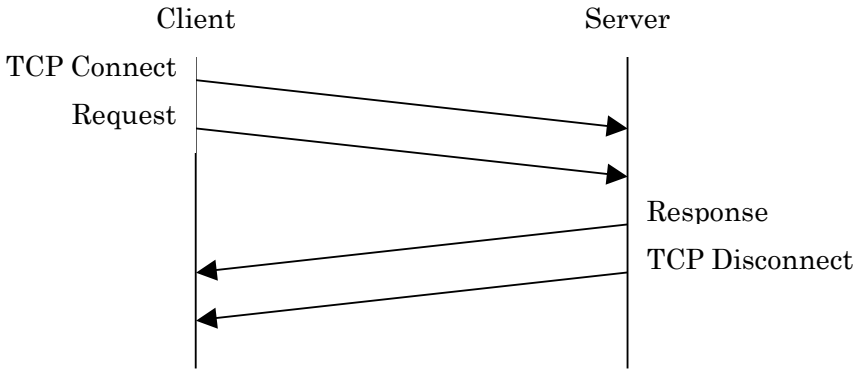


図 2-1:HTTP/1.0 での HTTP 交信

参考資料

- 日本語の HTTP に関する技術サイト:
  http://www.studyinghttp.net/
- RFC (HTTP/1.1):
  http://tools.ietf.org/html/rfc2616
- RFC (HTTP/1.1 旧版)日本語訳:
  http://lab.moyo.biz/translations/rfc/rfc2068-ja.xsp
- RFC (HTTP/1.0):
  http://tools.ietf.org/html/rfc1945
- 分かりやすくて簡便な英文チュートリアル:
  http://www.tutorialspoint.com/http/index.htm
- 当社が約 10 年前にアップロードした解説:
  http://www.cresc.co.jp/tech/java/Servlet_Tutorial/Att_01.htm

2.1節　HTTP の歴史

インターネットが学術研究コミュニティの世界からビジネスと消費者の世界に転換する契機となったのは HTTP の出現によるものだと言えよう。それまでのインターネットのアプリケーションは研究者や学生たちが SMTP（電子メール）や FTP（ファイル転送）を使って情報を交換していた。イギリス人の技術者でコンピュータ科学者の Sir Timothy John "Tim" Berners-Lee（ティム・バーナーズ・リーと一般的に呼ばれている）が 1980 年に当時インターネットの最大のノードだった CERN（セルン：欧州原子核研究機構）で、その前に Ted Nelson と Douglas Engelbart が開発していたハイパーテキストのコンセプト導入を提案、1989 年 3 月に WWW (World Wide Web)を発明・提案したとされている。1990 年 12 月 25 日に、CERN の学生だった Robert Cailliau とともに最初に HTTP によるインターネット上でのクライアント・サーバ間データ交換に成功している。WWW の最初のサイト (Info.cern.ch)は従って CERN で、1991 年 8 月 6 日に稼働している。

最初の HTTP 仕様は存在しないが、HTTP/0.9 とも呼ばれている。HTTP/0.9 では、メソッドは GET のみのシンプルなものであり、HTTP ヘッダも存在していなかった。サーバからの応答は常に HTML テキストのページだった。彼はまた 1994 年にアメリカの MIT で World Wide Web Consortium (W3C)を設立している。W3C は HTTP 及び HTML を中心にした標準化の中心になっている。

最初の公式な HTTP 仕様書は MIT のコンピュータ科学研究所(MIT LCS)の Tim Berners-Lee ほかが提案した 1996 年 5 月付の RFC 1945 で、これは HTTP/1.0 と呼ばれる。HTTP/1.0 では転送されるデータに関するメタデータを含む MIME（インターネット・メールの SMTP で使われているエンコーディング）ライクのメッセージのフォーマットを可能としたこと、及び要求/応答の意味の修正などがされている。

たまたまその少し前に、筆者はある通信機器メーカでキャンパス・ネットワークをやっていた部門の担当を命じられ、WWW の潜在性に注目し、自社のサイトを立ち上げようとしたが、インターネット上で社員が情報を公開することに社内で相当の抵抗があり、調整と規則づくりに苦労した。結局日本で最初の 10 社には入れず、残念な思いをした。

翌年 1997 年 1 月に同じく LCS から HTTP/1.1 としての RFC 2068 が提案されている。この提案は更新されて 1999 年 6 月に RFC 2616 という標準案になっていて、これが現在の世界標準となっている。HTTP/1.0 では階層的プロキシ、キャッシング、継続した接続、あるいはバーチャル・ホストなどが十分配慮されていなかった。更には、HTTP/1.0 と称する不十分な実装のアプリケーションが増えて互換性の問題が出てしまい（仕様策定が後追いの形になってしまった）、新たなバージョンで統一させ互換性を確保し、また互いのアプリケーションが相手の能力をわかるようにしようとした。

2.1.1 HTTP/1.1 とその特徴

HTTP/1.1 では効率的なネットワーク使用が出来るようになっている。以前のバージョンでは要求/応答のペアが終わると TCP 接続が閉じてしまっていた。HTTP/1.1 ではキープ・アライブのメカニズムが導入されており、ひとつの TCP 接続を複数の要求が再利用できるようになっている。そのように接続を切らない(persistent connection)方法は、手間がかかる TCP 接続確立・解放の手順を減らすことにより、高速な情報転送とネットワークの負荷の低減を狙ったものである。更にはチャンク形式の転送エンコーディング(chunked transfer encoding)が導入されており、サーバからのデータはバッファリングによる一括ではなく、ストリーム形式で送信され、クライアント側ではより高速な表示ができる。また HTTP パイプライン(HTTP pipelining)により、クライアントはある応答が返る前に別の要求を続けて送信できる。

下図はこれらの転送方式の相違を図示したものである：

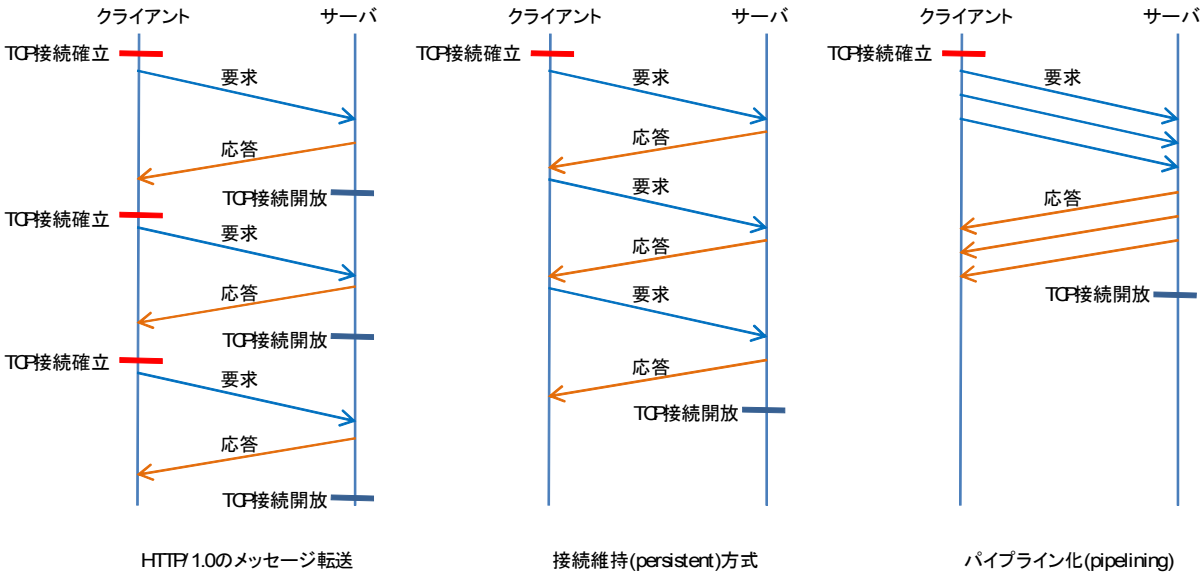


図 2-2：HTTP メッセージ転送方式

継続した接続は HTTP 1.1 ではデフォルトとなっており、これを使うために特別必要とするものはない。単に接続をオープンとして、複数の要求を直列に（パイプライン化するという）送信し、送信したと同じ順序で応答を読み出せば良い。そのためには、クライアント側は各応答の正確なバイト長を読みとり、各応答を正しく分離することに注意しなければならない。

もしクライアントがその要求メッセージに”Connection: close”なるヘッダを含めた場合には、それに対応した応答の直後に接続が解放される。継続した接続をサポートしない系の場合、あるいはその要求が該接続の最後であることがクライアント側で分かった場合には、必ずこのヘッダを挿入しなければならない。同様に、応答にこのヘッダが含まれていた場合には、該応答の後にサーバは接続を解放するので、クライアントは該接続でそののちに要求メッセージを送出してはならない。

サーバは、全ての応答が送られてしまう前に接続を切ってしまう場合があるかもしれない。そのときのため、クライアントは自分が出した要求と応答をきちんと追跡して、不足があれば要求を再送出する。再送の際は、接続が「継続」であることを確認するまでは要求をパイプライン化してはならない。サーバが継続した接続をサポートしていない（HTTP 1.0 のように）ことが分かったら絶対パイプライン化してはならない。

HTTP/1.1 の主な改善点を一覧にすると下表のようになる：

表 2-1：HTTP/1.1 の改善点

項目	内容
ホスト名識別	同じ IP アドレスで複数のホストが存在する場合に対応 例えば： GET / HTTP/1.1 Host: www.cresc.co.jp
コンテントのネゴシエーション	単一のリソースで複数のバージョンがありえるとき（例えば英語と日本語、HTML と PDF など）に対応 サーバが起動する場合とクライアントのエージェントが起動する場合がある
TCP 接続維持	ひとつの TCP 接続で複数の要求/応答を行う Connection: close ヘッダが要求に含まれるまで接続を維持
チャンク形式転送	応答をバッファリングしてコンテント長を指定することをしないで、行単位でボディ部分

	をクライアントに送信して、クライアントは順次それを処理することで高速表示を行う
バイト範囲指定	要求ドキュメントのうちの必要な部分をバイト指定することで、送信データ数を減らす 例えば: Range: bytes=500-999
プロキシとキャッシュ対応	これらの機能を使うためのヘッダを追加 例えば: last-modified: ...
その他	• 新たなステータス・コードを追加 • OPTIONS, TRACE, DELETE, PUT のメソッドを追加 • ダイジェスト認証 • いろんなヘッダの追加 • message/http と multipart/byteranges のメディア・タイプの定義

出典:http://www.apacheweek.com/features/http11

2.2節　ウェブ・ブラウザ

ウェブ・ブラウザ(web browser)は典型的な HTTP クライアントであり、インターネットの訪問者が HTML ドキュメントを見るのを支援し、ウェブ・サーバからあるいはファイル・システムからのコンテントを表示するソフトウエア・アプリケーションである。ブラウザは最も一般的に使用されているユーザ・エージェントである。現在多くのブラウザが普及しているが、特に Microsoft Internet Explorer、Mozilla Firefox、Google Chrome、Apple Safari などが大きな世界市場を占めている。リンクされたドキュメントの最大のネットワーク化された全体のことを World Wide Web (WWW)と呼ぶ。

下図は現在のブラウザの世界市場シェアである:

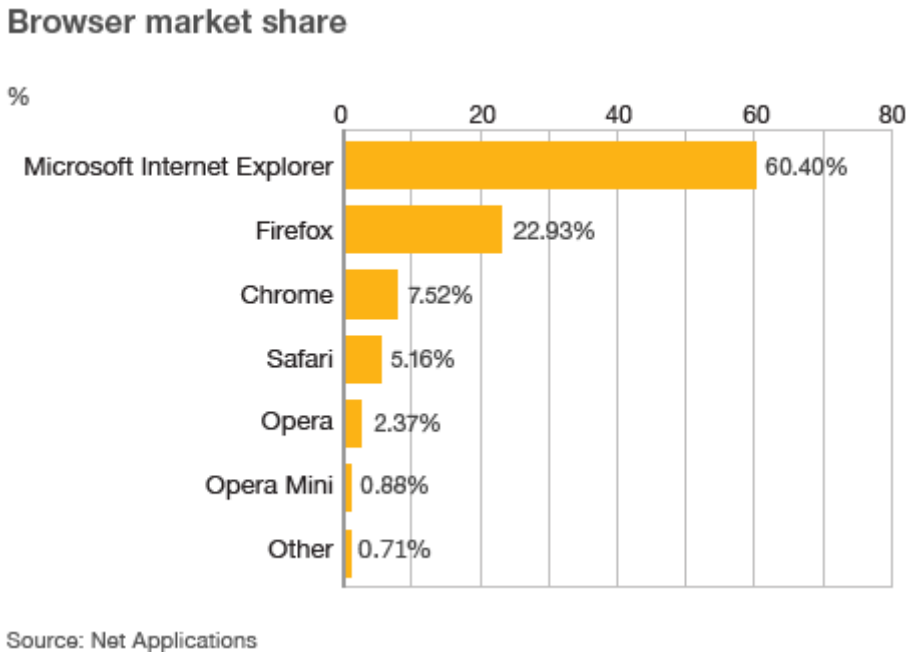


図 2-3:世界のブラウザ市場シェア

2010 年 8 月時点、出典:http://marketshare.hitslink.com/browser-market-share.aspx?qprid=1

ブラウザは HTTP を介してウェブ・サーバに情報を出したり、逆にウェブ・ページをウェブ・サーバから引き出す。ウェブ・ページは URL (uniform resource locator)によって場所が特定される。URL は http:で始まる。多くのブラウザはまたいろんな URL のタイプとそれに関わるプロトコルに対応しており、例えば ftp:は FTP (file transfer protocol)、gopher:では Gopher、https:では HTTPS (HTTP の SSL 暗号のバージョン) である。

ウェブ・ページとして一般的に受け付けられるファイル・フォーマットは HTML (hyper-text markup language)であり、MIME コンテント・タイプを使って HTTP プロトコルのなかでそれが特定される。殆どのブラウザは HTML に加えて JPEG、PNG、及び GIF のような他の技術フォーマットに対応している。プラグインにより更に他のフォーマットに対応するように拡張可能である。HTTP コンテント・タイプと URL プロトコル使用の組み合わせにより、ウェブ・ページの設計者たちはイメージ、アニメーション、ビデオ、サウンド、およびストリーミング・メディアをウェブ・ページに埋め込むことが出来る。あるいはウェブ・ページを介してそれらにアクセス可能なようにできる。

2.2.1 歴史

Tim Berners-Lee の 1991 年の WorldWideWeb という最初のウェブ・ブラウザの以前から、ウェブ・ブラウザの歴史が始まっている。Ted Nelson と Douglas Engelbart によるハイパー・テキストのコンセプトがそのもとになっていることは先に述べた。

1993 年 9 月リリースの NCSA の Mosaic ウェブ・ブラウザ(最初のグラフィカル・ブラウザのひとつ)がウェブの使用の爆発的拡大の引き金となっている。筆者も当時この Mosaic を見てウェブの潜在性を認識したひとりである。

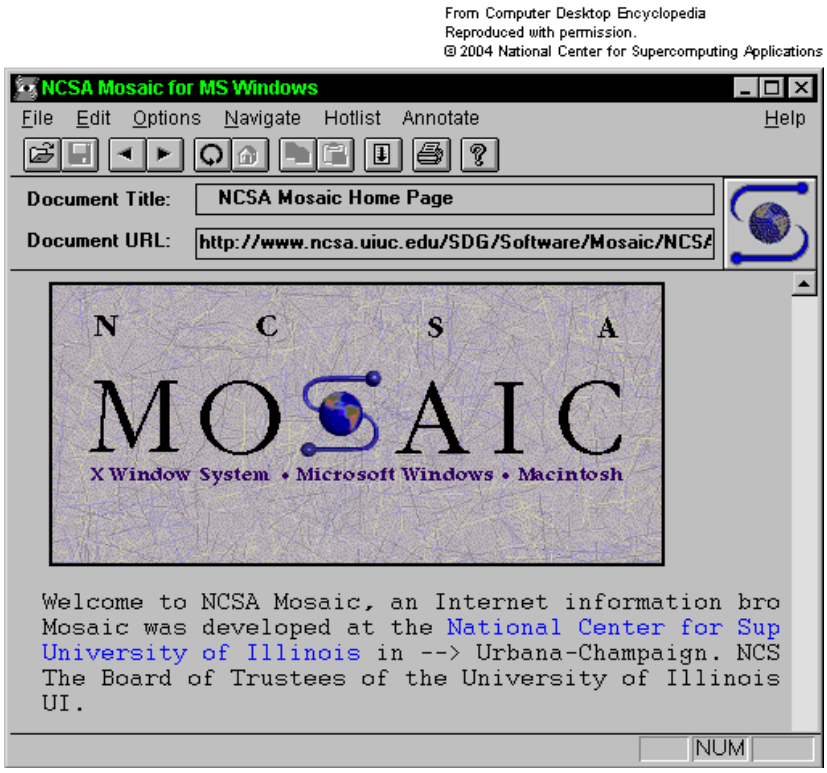


図 2-4:国立スーパー・コンピュータ応用研究所(NCSA)の Mosaic ブラウザ画面

当時 NCSA の Mosaic チームのリーダだった Marc Andreessen がその後 Netscape という会社をたちあげ、1994 年に Netscape Navigator(日本では良くネスケなどと呼んだ)リリース、これが世界で最も普及したブラウザになり、ピーク時には世界のウェブ使用の 90%を占めるまでになった。

Microsoft 社は当時はインターネットにはあまり積極的ではなく、たち遅れたが、1995 年 8 月に Mosaic の影響を大きく受けた Internet Explorer のバージョン 1.0 をリリースしたが、これは Spyglass, Inc.から買ったものだった。しかしこれがブラウザの市場の両社の戦争の始まりになっている。Microsoft は OS の支配力を活かしてこの市場を次第に支配していくことになる。2002 年では Internet Explorer のシェアは 95%のピークに達している。

Opera はノルウェイの Opera Software ASA が開発したもので、1996 年に最初にリリースされている。しかし大きな市場シェアを占めるに至らなかったが、モバイル機器向けの Opera Mini はモバイル・ブラウザの市場では現在かなりのシェアを占めている。

1998 年に Netscape はオープン・ソースのモデルで Internet Explorer に対抗する為に Mozilla Foundation の基礎を作っている。このブラウザが現在の FireFox のもとになっている。

携帯電話機向けのブラウザは、日本のメーカーやフィンランドの世界のトップのメーカーの Nokia が開発したものが市場を占めていた。しかしこれまでの機能電話機に加え、新たな発想、即ち PC の携帯電話機化という大きなタッチ・スクリーンをもったスマートホンは、PC とほぼ同機能を持ったブラウザが搭載され、Apple の iPhone では Safari、Google の Android では Android ブラウザなどがシェアを伸ばしている。アメリカでは iPhone の人気が高く、スマートホン・ブラウザでは 2010 年 4 月時点では 64%のシェアだと Apple が主張している。

2.3節　HTTP メッセージの構成

ここでいう「メッセージ」とは HTTP 通信の基本単位であり、仕様で規定された文法にもとづくバイト列で構成され、TCP 接続上で送信されるものである。クライアントからサーバへのメッセージは「HTTP 要求メッセージ」、サーバからのメッセージは「HTTP 応答メッセージ」という。

HTTP 要求メッセージも HTTP 応答メッセージも同じような構成になっている。

図 2-5：HTTP メッセージの構成

各メッセージは手紙と同じように、ヘッダ部とボディ部で構成されている。そのような構造は電子メールの SMTP などいろんな TCP/IP 網のアプリケーションで良く使われている。各メッセージの最初はスタート行と呼び、要求 HTTP メッセージの場合は要求行、応答 HTTP メッセージの場合はステータス行と呼ぶ。ヘッダ部分は複数のテキスト行で構成され、各行は CR(復帰)と LF(改行)という昔のテレタイプ時代の 2 つのコードで終端されている。ヘッダ部分の各行はヘッダ行と呼ばれ、最後は空白(ブランク)行で識別される。メッセージ・ボディは送信されるデータの本体であり、これはテキスト行だけではなく、画像などのバイト列でも良い。いろんなデータ形式をどのようにエンコードしてメッセージ・ボディとしているかは、Transfer-Encoding のヘッダ行で相手に知らせる。受け手はこのヘッダ行をもとにボディ部のデータをもとのデータに復元する。

ヘッダ部の文字は 7 単位の US-ASCII (American Standard Code for Information Interchange)コードが使用される。昔の端末は 7 単位(プラス 1 ビットのパリティ)しか扱えないものが多く、そのような端末でも扱えるようにこのコードが使われている。

2.4節　　要求行

クライアントあるいはブラウザが HTML ドキュメントのような何らかのリソースをサーバに要求するときは、HTTP メソッド、要求 URI (Uniform Resource Indicator)、プロトコル・バージョン、そしてオプショナルとしてのヘッダ情報を指定する。これらの情報の最初の 3 つが要求行に含まれる。サーバ側はその要求を処理し、応答を返す。応答にはステータス情報、応答ヘッダたち、及び応答データのような情報が含まれる。

要求行は、たとえば次のようなテキストになる:

```
GET http://www.cresc.co.jp:80/tech/java/Servlet_Tutorial/index.htm HTTP/1.1
```

ここで"GET"が HTTP メソッド、"http://www.cresc.co.jp:80/tech/java/Servlet_Tutorial/index.htm"が要求 URI、そして"HTTP/1.1"がプロトコル・バージョンである。要求 URI には後述のようにクエリ文字列が付加される場合がある。

RFC 2616 では Backus-Naur Form (BNF)形式で次のように表現されている:

```
Request-Line    = Method SP Request-URI SP HTTP-Version CRLF
```

ここに、SP はスペース、CRLF は CR コードと LF コードのペアである。

2.4.1　HTTP メソッド

HTTP では、指定したリソースに関してなされるべきアクション要求を示す 9 つのメソッド(「動詞」とも呼ばれる)が定められている。このリソースが何を意味するかは、既に存在するデータあるいは動的に生成されるデータに関わらず、サーバの実装に依存する。一般的にそのリソースはサーバ上のファイル、あるいはサーバ上に駐在するプログラムの出力となる。通常のサーブレットでは、GET と POST の要求のみが扱われる。

表 2-2:HTTP メソッド一覧

HEAD	GET 要求に対応するものと同じ応答だが、応答のボディを含まないものを要求する。このメソッドはコンテント全体を送信させないで応答ヘッダたちの中に書かれているメタ情報を取得するのに有用である。
GET	要求 URI で指定したリソースにあたるものを要求する。GET を使った要求(及び他の一部の

	要求)は、取得以外のアクションをとってはならない。要求 URI がデータ作成プロセスを示す場合は、応答に含められるデータは、そのプロセスで作成されたものとなり、そのプロセスのソースのテキストではない。POST のように処理するデータを指定したリソースに送る場合は、クエリ文字列として要求行に含める。
POST	処理するデータを(たとえば HTML フォームから)指定したリソースに対し送る。このデータはその要求メッセージのボディ部に含められる。その結果は新しいリソースをもたらすか、あるいは既存のリソースの更新をもたらす。
PUT	指定したリソースに相当するものをアップロードする。
DELETE	指定したリソースを削除する。
TRACE	受け取った要求をそのまま返し、クライアントが介在するサーバたちによって出した要求に何らかの変更あるいは追加がなされているかどうかが分かるようにする。
OPTIONS	サーバが指定された URL に対し対応する HTTP メソッドを返す。これは特定のリソースではなくて'*'を要求することでウェブ・サーバの機能をチェックするのに使える。
CONNECT	要求する接続を透過的な TCP/IP トンネルに変える。通常は非暗号化の HTTP プロキシを介して SSL 暗号化通信(HTTP)をする為に使われる。
PATCH	あるリソースに対し部分的な変更を適用するのに使われる。

2.4.2 要求 URI

HTTP での要求 URI (Uniform Resource Indicator)は、絶対的なものと、ベース URI からの相対的なものとがあり得るが、それはそのクライアントのコンテキスト(現在そのリソースへの階層のどこにいるか)次第である。絶対 URI はスキーム名とその後に続く':'があることで分かる。URI の仕様は別途 RFC 2396 で規定されており、HTTP ではそのうち「URI 参照」(URI-reference)、「絶対 URI」(absoluteURI)、「相対 URI」(relativeURI)、「ポート」(port)、「ホスト」(host)、「絶対パス」(abs_path)、「相対パス」(rel_path)、及び「権限」(authority)が使われている。

HTTP プロトコルでは URI の長さに対してはあらかじめ制限をかけてはいない(昔の HTTP/1.0 では 128 バイトに制限されていた)。

URI の構文の BNF 表現は次のようになる:

```
        URI             = ( absoluteURI | relativeURI ) [ "#" fragment ]
; URI には絶対 URI と相対 URI があり、"#"で続くフラグメントは省略可能
        absoluteURI     = scheme ":" *( uchar | reserved )
; 絶対 URI はスキーム、":"、そのあとに uchar あるいは reserved が繰り返される
        relativeURI     = net_path | abs_path | rel_path
; 相対パスはネットワーク・パス、絶対パス、あるいは相対パスである
        net_path        = "//" net_loc [ abs_path ]
; ネットワーク・パスは"//"のあとにネットワークの場所、そのあとの絶対パスは省略可能
        abs_path        = "/" rel_path
; 絶対パスは"/"のあとに相対パスが続く
        rel_path        = [ path ] [ ";" params ] [ "?" query ]
; 相対パスはパスのあとに";"の後に続くパラメタたち、そして"?"の後に続くクエリ文字列で構成されるが、これらの 3 要素は省略可能
        path            = fsegment *( "/" segment )
; パスは fsegment のあとに"/"のあとに続くセグメントの繰り返しで構成される
        fsegment        = 1*pchar
; fsegment は 1 文字以上の印刷可能文字のならび
        segment         = *pchar
```

```
;セグメントは印刷可能文字のならび
        params          = param *( ";" param )
; パラメタたちは";"で区切られたひとつ以上のパラメタで構成
        param           = *( pchar | "/" )
; パラメタは印刷可能文字あるいは"/"の並び
        scheme          = 1*( ALPHA | DIGIT | "+" | "-" | "." )
; スキームはアルファベット文字、数字文字、"+"、"-"、あるいは"･"の文字の並び
        net_loc         = *( pchar | ";" | "?" )
; ネットワークの場所は印刷可能文字、";"、あるいは"?"の文字の並び
        query           = *( uchar | reserved )
; クエリ文字列は非予約文字、エスケープ文字、あるいは予約文字の並び
        fragment        = *( uchar | reserved )
; フラグメントは非予約文字あるいはエスケープ文字、あるいは予約文字の並び
        pchar           = uchar | ":" | "@" | "&" | "=" | "+"
; 印刷可能文字とは非予約文字、あるいは":"、"@"、"&"、"="、あるいは"+"
        uchar           = unreserved | escape
        unreserved      = ALPHA | DIGIT | safe | extra | national
; 非予約文字とはアルファベット文字、数字文字、安全な文字、追加文字、各国文字
        escape          = "%" HEX HEX
; エスケープ文字とは"%"のあとにその文字コードの16進表記が続いたもの
        reserved        = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"
; 予約文字
        extra           = "!" | "*" | "'" | "(" | ")" | ","
; 追加文字
        safe            = "$" | "-" | "_" | "."
; 安全な文字
        unsafe          = CTL | SP | <"> | "#" | "%" | "<" | ">"
; 安全でない文字
        national        = <ALPHA, DIGIT, reserved, extra, safe, unsafeを除くすべての OCTET>
```

これは読者には分かりにくいかも知れないが、特に文字列に関しては重要な意味を持っている。

2.4.2.1 URL エンコーディング

インターネット上のメール(SMTP)や HTTP などのメッセージは、ヘッダ部に宛先やその他メッセージの制御に関わる情報が載せられる。このヘッダは途中の（ヘッダが解釈される）ゲートウエーを幾つか中継され、端から端のノードに伝達されるので、これらのノードに理解できるコードと文字で表現されなければならない。ヘッダ部に日本語のような 2 バイトの文字が入ると、ノードはこれを 1 バイトずつ解釈しようとする。そのときにそれらのバイトのどれかがノードにとって特別の意味を持つバイトであったら、正しい結果が得られなくなってしまう。更に昔はネットワークによっては各バイトの一番上のビット(MSB)を欠落させる伝送ノードが存在していた。従ってどのような文字であってもそのような制約のなかで安全に且つ透過的に伝送されることが必要だった。具体的には 7 ビットで且つ「安全な」ASCII(American Standard Code for Information Interchange)文字セットからなる文字列に変換してインターネットを通すということになる。ヘッダ行に置いては RFC 2047 による表現が使われるが、我々が良く使用するクッキー文字列やクエリ文字列でははは、そのような仕組みとして URL エンコードが使われる。

URL エンコードというのは、もともと HTTP ヘッダ部のスタート行に置かれる URL 部分に 2 バイト文字や制御文字と紛らわしい文字が入るのを防止するために考えられたからそう呼ばれている。しかし送られる情報をすべて「見える」文字列に変換するのは都合が良いことが多く、メッセージのボディ部分の伝達にも使われている。ボディ部分の変換にはもうひとつ MIME(Multi-Purpose Internet Mail Extensions)のエンコーディングがある。これ

はマルチメディア情報の転送に良く使われる。

HTTP の仕様も 8 ビット文字を対象にしており、我々が一般的に使う 2 バイト文字を想定していない。しかも前述のように、8 ビットとはいえ一番上のビット(MSB)がゼロの文字(即ち ASCII 文字)が主体で、MSB が 1 の文字は national という表現で、クエリ文字列とネットワークの場所にしか使えない。しかもそのような文字は欧米の端末では一般に表示されないし、時にはそのような文字を処理できない介在ゲートウェイも存在しないとも限らない。したがって**印刷できない文字、安全でない文字、あるいは予約文字をクエリ文字列やクッキー文字列に含めるときは、エスケープ文字に変換**しなければならない。これは 2 バイト文字をバイト列にしたときでも言えることである。RFC 2396 ではクエリ文字列では";"、"/"、"?"、":"、"@"、"&"、"="、"+"、","、及び"$"が予約文字とされている。

この予約文字をエスケープ変換するかどうかで問題が起きることは、2002 年に当社のサイトにアップロードした JavaScript における URL エンコードの処理というメモで詳しく示されている。つまり JavaScript の encodeURI というメソッドでは予約文字をエスケープ変換しないが、Java の URLEncoder.encode(String,"UTF-8")というメソッドでは予約文字をエスケープ変換するので、注意が必要である。なお encode というメソッドでは、スペース(SP)は"+"に置き換えられる。

したがって、クエリ文字列での URL エンコードの手順は以下のようになる

- 日本語のように 2 バイトの文字は 1 バイト毎にとりだして ASCII 文字とみなして以下の変換を行う。
- 名前と値にある「安全でない」文字、「予約」文字、及び「印刷できない」文字は"%xx"というエスケープ文字列に変換する。"xx"はその文字の ASCII 値を 16 進表示したものである。そのような文字には=; &, %, +やプリントできない文字、MSB(最上位ビット)が 1 の文字を含む。
- 全ての ASCII のスペース文字を+に変換する。
- 名前と値を=と&でつないでひとつの文字列にする。例えば
  name1=value1&name2=value2&name3=value3

この文字列が POST 要求メッセージのボディ部分、あるいは GET 要求のクエリ文字列、あるいはクッキーのヘッダ行としてはめ込まれる。

下表は、java.net.URLEncoder.encode(String, enc)がエスケープしない文字である(enc は"UTF-8"など文字セットの名前)。太字且つ斜め文字の部分がエスケープしない文字、但しスペース(SP)は"+"に置き換え且つそれはエスケープしない文字である。

表 2-3:URLEncoder がエスケープ変換しない文字

Char	Decimal	Hex	Char	Decimal	Hex
NUL	0	0	SOH	1	1
STX	2	2	ETX	3	3
EOT	4	4	ENQ	5	5
ACK	6	6	BEL	7	7
BS	8	8	HT	9	9
NL	10	a	VT	11	b
NP	12	c	CR	13	d
SO	14	e	SI	15	f
DLE	16	10	DC1	17	11
DC2	18	12	DC3	19	13
DC4	20	14	NAK	21	15
SYN	22	16	ETB	23	17
CAN	24	18	EM	25	19
SUB	26	1a	ESC	27	1b
FS	28	1c	GS	29	1d
RS	30	1e	US	31	1f
SP	***32***	***20***	!	33	21

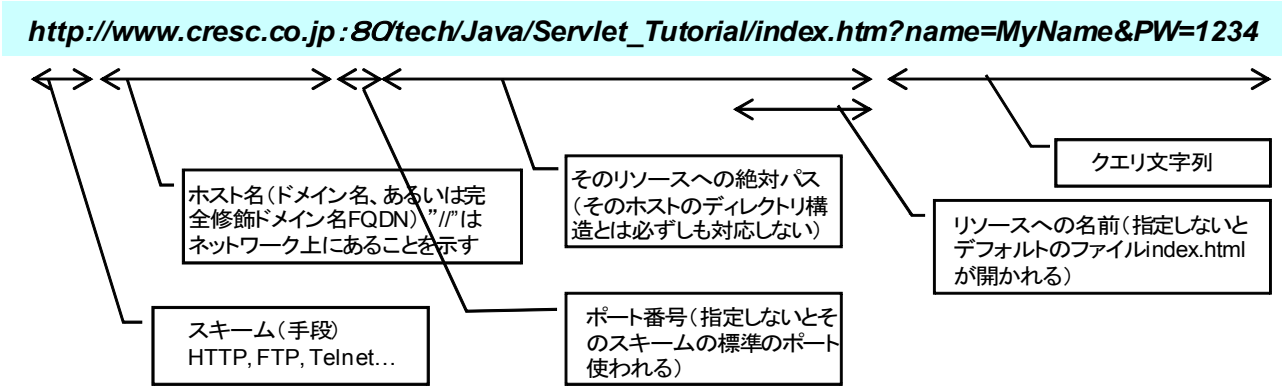
″	34	22	#	35	23
$	36	24	%	37	25
&	38	26	'	39	27
(	40	28	)	41	29
*	42	2a	+	43	2b
,	44	2c	-	45	2d
.	46	2e	/	47	2f
0	48	30	1	49	31
2	50	32	3	51	33
4	52	34	5	53	35
6	54	36	7	55	37
8	56	38	9	57	39
:	58	3a	;	59	3b
<	60	3c	=	61	3d
>	62	3e	?	63	3f
@	64	40	A	65	41
B	66	42	C	67	43
D	68	44	E	69	45
F	70	46	G	71	47
H	72	48	I	73	49
J	74	4a	K	75	4b
L	76	4c	M	77	4d
N	78	4e	O	79	4f
P	80	50	Q	81	51
R	82	52	S	83	53
T	84	54	U	85	55
V	86	56	W	87	57
X	88	58	Y	89	59
Z	90	5a	[	91	5b
¥	92	5c	]	93	5d
^	94	5e	_	95	5f
`	96	60	a	97	61
b	98	62	c	99	63
d	100	64	e	101	65
f	102	66	g	103	67
h	104	68	i	105	69
j	106	6a	k	107	6b
l	108	6c	m	109	6d
n	110	6e	o	111	6f
p	112	70	q	113	71
r	114	72	s	115	73
t	116	74	u	117	75
v	118	76	w	119	77
x	120	78	y	121	79
z	122	7a	{	123	7b
|	124	7c	}	125	7d
~	126	7e	DEL	127	7f

Java.net.URLDecoder.decode(String, enc)は、逆の操作を行う。なお、応答オブジェクトの章で説明するように、**URLEncoder.encode 及び URLDecoder.decode はブラウザで一般的に使われている URL エンコードとは完全な互換性を持っていない**ことに注意が必要である。

2.4.2.2　HTTP URL

HTTP プロトコルでネットワーク上のリソースを特定するには"http"スキームが使われる。ここでは、このスキームに

おける文法と意味を簡単に示す。先ず、HTTP の URL (Uniform Resource Locator)の例を下図に示す:



HTTP: Hyper Text Transfer Protocol, FTP: File Transfer Protocol, FQDN: Fully Qualified Domain Name

図 2-6:HTTP URL の例

これを BNF 表記をすれば次のようになる:

http_URL = "http:" "//" host [":" port] [abs_path ["?" query]]

即ち、HTTP の URL は"http:"、"//"、ホスト名、そしてオプショナルとして":"に続くポート番号、オプショナルとして絶対パスの順で表現され、絶対パスにはオプショナルとして"?"に続くクエリ文字列(名前と値を"="でつないだペアをオプショナルに"$"で複数連結させたもの)がつく。ポートのところが空あるいは指定されていないときは HTTP のデフォルトのポートの 80 であるとみなされる。その意味は、指定されたリソースはそのサーバのそのポートでの TCP 接続で聞いているサーバ上にあり、そのリソースの要求 URI は絶対パスで指定されたものである、ということになる。

なお、絶対パスにはリソースを含めない人も多いので、注意されたい。その場合はパスは"/"で始まり、パスが空でないときは"/"で終わる。

URL に IP アドレスを使うことは極力避けるよう RFC 1900 で記されている。URL の中に絶対パスが存在しないときは、そのリソースのための要求 URI としては"/"(サーバ・ルートを示す)で与えられねばならない。

サーバとの間にプロキシが介在し、そのプロキシが FQDN(完全修飾ドメイン名:ホスト名、ドメイン名(サブドメイン名)などすべてを省略せずに指定した記述形式)でないホスト名を受けたときは、そのプロキシはそのホスト名にドメインを付加しても良い。FQDN を受けた場合はそのプロキシはそのホスト名を変えてはならない。

クエリ文字列は名前と値のペアを"&"で連結させる:

```
?name1=value1&name2=value&name3=value3 ….
```

2.4.2.3 URI 比較

合致しているかどうかを判断する為に 2 つの URL を比較するときは、クライアントは URI 全体にわたって大文字と小文字を区別してバイトごとに比較をしなければならないが、以下の 4 つの場合は例外となる:

- ポートが空または与えられていないときは、その URI 参照の為のデフォルト・ポートだと見なす
- ホスト名の比較には大文字と小文字を区別してはならない

- スキーム名の比較には大文字と小文字を区別してはならない
- 空の絶対パスは"/"の絶対パスだとみなす

予約("reserved")と安全でない("unsafe")セットは、そのエスケープ変換(""%" HEX HEX")と等しい。例えば、以下のURIは皆同じものを示す:

```
http://abc.com:80/~smith/home.html
http://ABC.com/%7Esmith/home.html
http://ABC.com:/%7esmith/home.html
```

2.4.3 プロトコル・バージョン

HTTPバージョンのBNF表記は次のようである:

```
HTTP-Version   = "HTTP" "/" 1*DIGIT "." 1*DIGIT
```

しかしながら現在はHTTP/1.0とHTTP/1.1のふたつのバージョンのみが存在する。これまで当社のサイトにアップロードされていたサーブレットのチュートリアルは、2001年頃に作成したもので、あの時点ではHTTP/1.0が州で、HTTP/1.1対応のブラウザやサーバのほうが少なかった。現在はHTTP/1.0のアプリケーションは殆ど存在していないので、HTTP/1.0対応はあまり気にする必要が無くなっている。しかしHTTP/1.0の知識はまだ必要であろう。

後述のように、**HTTP/1.1バージョンの総ての要求メッセージのヘッダには、Host要求ヘッダ・フィールドが伴っていなければならない。**

HTTP/1.0ではGET、POST、及びHEADのメソッドが定義されていたが、HTTP/1.1では更にOPTIONS、PUT、TRACE、DELETE、及びCONNECTの5つが追加されている。

2.4.4 GET要求とPOST要求との相違

GET要求メソッドは非常にシンプルで最も頻繁に使われるメソッドである。GET要求はHTMLドキュメント、イメージ、CSSファイルのような静的なドキュメントをサーバから取得するのに使われる。GET要求は静的情報を得る為に作られてはいるが、この要求メソッドは要求URIの最後にクエリ文字列を追加することで、動的な情報を取得するのにも使用可能である。HTMLフォームではGETはデフォルトの要求メソッドになっている。

POST要求メソッドは一般的に動的なリソースへのアクセスに使われる。POST要求は大量のデータをサーバに送信するのに使われる。POST要求はその名のとおり、サーバにファイルをアップロードしたり、あるいは直列可能(serializable)Javaオブジェクト、あるいは生のバイト・データを送信することさえ可能である。GETと違って、POST要求は総てのデータをHTTP要求のボディ部の要素として送信するので、ブラウザのアドレス・バーには表示されず、ブックマークもできない。サーバに送られるデータはユーザには見えないことになる。POST要求は一般的にHTTPのフォーム・パラメタをサーバに送信したり、ファイルをアップロードするのに使われる。

POST要求では:

- 要求とともにデータの集まりをメッセージ本体に入れる。そのために通常メッセージ本体を記述するヘッダ、例えばContent-Type:やContent-Length:などのヘッダが追加されている。
- 要求URIは検索すべきリソースを意味するのではなく、通常、送出中のデータが処理されるべきプログラムを指定する。
- HTTP応答はプログラムからの出力であり、静的なファイルではない。

GET と POST の要求には幾つかの相違がある。GET 要求は要求パラメタを要求 URL の最後にクエリ文字列として付して送信されるが、POST 要求では要求パラメタは HTTP 要求のボディ部として送られる。GET 要求はクエリ文字列として送信されるので、パラメタはブラウザのアドレス・バーに表示されるし、ユーザはブックマークだ可能である。このことは情報の重要度によってはセキュリティの問題を起こす可能性がある。GET 要求のもうひとつの問題は、一部のサーバでは、URL の長さを 240 文字に制限していることで、あまりにクエリ文字列に追加するパラメタが多すぎると、この制限を超えることで、その要求が受け付けられなくなる可能性が出る。特に日本語などでは、URL エンコードによりクエリ文字列が簡単に大きくなってしまう。POST 要求ではデータがボディとして送られるので、長さの制限はない。但し、HTTP/1.1 では URL の長さに対しては何ら制限をかけていない。

最も一般的な POST の使用法は、HTML フォームのデータをサーブレットや CGI スクリプトに提出することである。HTML の**フォーム・データをサーバに渡すときは、GET がデフォルトになっているので、<form action="POST">のようにアクションを指定する必要がある**。この場合、Content-Type:ヘッダは通常 application/x-www-form-urlencoded となり、Content-Length:ヘッダは URL エンコードされたフォームデータのバイト長となる。CGI の場合はこのメッセージ本体を STDIN 経由で受領し、デコードする。以下に POST による典型的なフォームデータ提出 (submission) 例を示す。

```
POST /path/script.cgi HTTP/1.0
      From: terry@cresc.co.jp
      User-Agent: HTTPTool/1.0
      Content-Type: application/x-www-form-urlencoded
      Content-Length: 32

Home=Cosby&faborite+flavor=flies
```

この場合、(Home, Cosby)、(faborite flavor, flies)という 2 つの名前と値のペアが script.cgi に送信される。

2.5節　　ステータス行

応答メッセージの開始行はステータス行であり、プロトコル・バージョンの後に数字のステータス・コードと、それに関わるテキスト・フレーズが付いており各要素はスペース文字で分離されている最後の CR と LF のシーケンス以外に CR と LF を使うことは許されない。

BNF 記述は次のようになる:

```
      Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

プロトコル・バージョンは要求行のプロトコル・バージョンの節で説明したとおりである。

2.5.1　ステータス・コードと理由句

ステータス・コードは 3 桁の 10 進数で、サーバがその要求を解釈し満足させようとした結果を示す。理由句はステータス・コードの短いテキストによる記述を意図したものである。ステータス・コードは自動システムで認識され、理由句はユーザである人が見るために作られている。仕様では、クライアントがこの理由句を調べることは要求されていない。

ステータス・コードの最初の桁は応答のクラスを示している。後の 2 桁はカテゴリ化されていない。最初の桁は次

の 5 つである:

表 2-4:ステータス・コードの種類

1xx	情報 (Informational)	要求を受信した、プロセス継続中
2xx	成功 (Success)	そのアクションは正しく受信し、理解し、受け付けた
3xx	リダイレクション (Redirection)	その要求を完了させるには更なるアクションが必要
4xx	クライアント・エラー (Client Error)	その要求に文法上の誤りがあるか、その要求を満足させられない
5xx	サーバ・エラー (Server Error)	明らかに有効な要求だが、満足させることに失敗した

個々のステータス・コードと対応する理由句を以下に示す。この理由句は仕様書上は勧告であり、各国向けに変更しても構わない。

表 2-5:HTTP ステータス・コードと理由句

100	Continue	継続	クライアントは要求を継続できる。サーバが要求の最初の部分を受け取り、まだ拒否していないことを示す。 例として、クライアントが Expect: 100-continue ヘッダをつけた要求を行い、それをサーバが受理した場合に返される。
101	Switching Protocols	プロトコル切替え	サーバは要求を理解し、遂行のためにプロトコルの切替えを要求している。
102	Processing	処理中	WebDAV の拡張ステータス・コード。処理が継続されて行われていることを示す。
200	OK	OK	要求は成功し、応答とともに要求に応じた情報が返される。 ブラウザでページが正しく表示された場合は、ほとんどがこのステータスコードを返している。
201	Created	作成	要求は完了し、新たに作成されたリソースの URI が返される。 例:PUT メソッドでリソースを作成する要求を行ったとき、その要求が完了した場合に返される。
202	Accepted	受理	要求は受理されたが、処理は完了していない。 例:PUT メソッドでリソースを作成する要求を行ったとき、サーバがその要求を受理したものの、リソースの作成が完了していない場合に返される。 バッチ処理向け。
203	Non-Authoritative Information	オーソライズ出来ない情報	オリジナルのデータではなく、ローカルやプロキシ等からの情報であることを示す。
204	No Content	コンテントなし	要求を受理したが、返すべき応答エンティティが存在しない場合に返される。 例:POST メソッドでフォームの内容を送信したが、ブラウザの画面を更新しない場合に返される。
205	Reset Content	コンテントのリセット	要求を受理し、ユーザ・エージェントの画面をリセットする場合に返される。 例:POST メソッドでフォームの内容を送信した後、ブラウザの画面を初期状態に戻す場合に返される。
206	Partial Content	部分的コンテント	部分的内容。部分的 GET 要求を受理したときに、返される。 例:ダウンロードツール等で分割ダウンロードを行った場合や、レジュームを行った場合に返される。
207	Multi-Status	複数ステータス	WebDAV の拡張ステータスコード。
226	IM Used	IM 使用	HTTP のなかでデルタ・エンコーディングの拡張ステータスコード。

300	Multiple Choices	複数の選択肢	要求したリソースが複数存在し、ユーザやユーザー・エージェントに選択肢を提示するときに返される。 具体例として、W3C の http://www.w3.org/TR/xhtml11/DTD/xhtml11.html
301	Moved Permanently	恒久的に移動	要求したリソースが恒久的に移動されているときに返される。Location ヘッダに移動先の URL が示されている。 例としては、ファイルではなくディレクトリに対応する URL の末尾に/を書かずにアクセスした場合に返される。具体例として、 http://www.w3.org/TR
302	Found	発見	要求したリソースが一時的に移動されているときに返される。Location ヘッダに移動先の URL が示されている。 元々は Moved Temporarily（一時的に移動した）で、本来は要求したリソースが一時的にその URL に存在せず、別の URL にある場合に使用するステータス・コードであった。しかし、例えば掲示板や Wiki などで投稿後にブラウザを他の URL に転送したいときにもこのコードが使用されるようになったため、302 は Found になり、新たに 303,307 が作成された。
303	See Other	他を参照のこと	要求に対する応答が他の URL に存在するときに返される。Location ヘッダに移動先の URL が示されている。 要求したリソースは確かにその URL にあるが、他のリソースをもって応答とするような場合に使用する。302 の説明で挙げたような、掲示板や Wiki などで投稿後にブラウザを他の URL に転送したいときに使われるべきコードとして導入された。
304	Not Modified	未更新	要求したリソースは更新されていないことを示す。 例として、If-Modified-Since ヘッダを使用した要求を行い、そのヘッダに示された時間以降に更新がなかった場合に返される。
305	Use Proxy	プロキシを使用のこと	応答の Location ヘッダに示されるプロキシを使用して要求を行わなければならないことを示す。
306	(Unused)	未使用	将来のために予約されている。ステータス・コードは前のバージョンの仕様書では使われていたが、もはや使われておらず、将来のために予約されているとされる。
307	Temporary Redirect	一時的リダイレクト	要求したリソースは一時的に移動されているときに返される。Location ヘッダに移動先の URL が示されている。 302 の規格外な使用法が横行したため、302 の本来の使用法を改めて定義したもの。
400	Bad Request	不正な要求	定義されていないメソッドを使うなど、クライアントの要求がおかしい場合に返される。
401	Unauthorized	認証が必要	Basic 認証や Digest 認証などを行うときに使用される。 たいていのブラウザはこのステータスを受け取ると、認証ダイアログを表示する。
402	Payment Required	支払いを要す	未実装。
403	Forbidden	禁止されている	リソースにアクセスすることを拒否された。要求はしたが処理できないという意味。 アクセス権がない場合や、ホストがアクセス禁止処分を受けた場合などに返される。 例: 社内（イントラネット）からのみアクセスできるページに社外からアクセスしようとした。
404	Not Found	見つからず	リソースが見つからなかった。 単に、アクセス権がない場合などにも使用される。
405	Method Not Allowed	許可されていないメソッド	許可されていないメソッドを使用しようとした。 例: POST メソッドの使用が許されていない場所で、POST メソッドを使用した場合に返される。

406	Not Acceptable	受理不可	Accept 関連のヘッダに受理できない内容が含まれている場合に返される。 例: サーバは英語か日本語しか受け付けられないが、リクエストの Accept-Language:ヘッダに zh（中国語）しか含まれていなかった。 例: サーバは application/pdf を送信したかったが、リクエストの Accept: ヘッダに application/pdf が含まれていなかった。 例: サーバは UTF-8 の文章を送信したかったが、リクエストの Accept-Charset:ヘッダには、UTF-8 が含まれていなかった。
407	Proxy Authentication Required	プロキシ認証要す	プロキシの認証が必要な場合に返される。
408	Request Timeout	要求がタイムアウト	要求が時間以内に完了していない場合に返される。
409	Conflict	矛盾	要求は現在のリソースと矛盾するので完了出来ない。
410	Gone	消滅した	ファイルは恒久的に移動した。どこに行ったかもわからない。404 Not Found と似ているが、こちらは二度と復活しない場合に示される。
411	Length Required	長さが必要	Content-Length ヘッダがないのでサーバがアクセスを拒否した場合に返される。
412	Precondition Failed	前提条件で失敗	前提条件が偽だった場合に返される。 例：要求の If-Unmodified-Since:ヘッダに書いた時刻より後に更新があった場合に返される。
413	Request Entity Too Large	要求エンティティが大きすぎる	要求エンティティがサーバの許容範囲を超えている場合に返す。 例：アップローダの上限を超えたデータを送信しようとした。
414	Request-URI Too Long	要求 URI が大きすぎる	URI が長過ぎるのでサーバが処理を拒否した場合に返す。 例：画像データのような大きなデータを GET メソッドで送ろうとし、URI が何 10kB にもなった場合に返す（上限はサーバに依存する）。
415	Unsupported Media Type	対応していないメディア・タイプ	指定されたメディアタイプはサポートされていない場合に返す。
416	Requested Range Not Satisfiable	要求したレンジは範囲外	実ファイルのサイズを超えるデータを要求した。 たとえば、リソースのサイズが 1024Byte しかないのに、1025Byte を取得しようとした場合などに返す。
417	Expectation Failed	期待するヘッダに失敗	その拡張はレスポンスできず、またはプロキシサーバが次に到達するサーバがレスポンスできないと判断している。 具体例として、Expect:ヘッダに 100-continue 以外の変なものを入れた場合や、そもそもサーバが 100 Continue が扱えない場合に返す。
422	Unprocessable Entity	処理できないエンティティ	WebDAV の拡張ステータスコード。
423	Locked	ロックされている	WebDAV の拡張ステータスコード。リクエストしたリソースがロックされている場合に返す。
424	Failed Dependency	依存関係で失敗	WebDAV の拡張ステータスコード。
426	Upgrade Required	アップグレード必要	Upgrading to TLS Within HTTP/1.1 の拡張ステータスコード。
500	Internal Server Error	サーバ内部エラー	サーバ内部にエラーが発生した場合に返される。 例として、CGI として動作させているプログラムに文法エラーがあったり、設定に誤りがあった場合などに返される。
501	Not Implemented	実装されていない	実装されていないメソッドを使用した。 例として、WebDAV が実装されていないサーバに対して WebDAV で使用するメソッド（MOVE や COPY）を使用した場合に返される。
502	Bad Gateway	不正なゲートウェイ	ゲートウェイ・プロキシサーバは不正な要求を受け取り、これを拒否した。
503	Service Unavailable	サービス利用不可	サービスが一時的に過負荷やメンテナンスで使用不可能である。 例として、アクセスが殺到して処理不能に陥った場合に返される。

504	Gateway Timeout	ゲートウェイ・タイムアウト	ゲートウェイ・プロキシサーバは URI から推測されるサーバからの適切なレスポンスがなくタイムアウトした。
505	HTTP Version Not Supported	サポートしていない HTTP バージョン	その要求がサポートされていない HTTP バージョンである場合に返される。
506	Variant Also Negotiates		Transparent Content Negotiation in HTTP で定義されている拡張ステータスコード。
507	Insufficient Storage	容量不足	WebDAV の拡張ステータスコード。リクエストを処理するために必要なストレージの容量が足りない場合に返される。
509	Bandwidth Limit Exceeded	帯域幅制限超過	そのサーバに設定されている帯域幅(転送量)を使い切った場合に返される。
510	Not Extended	拡張できない	An HTTP Extension Framework で定義されている拡張ステータスコード。

出典:http://ja.wikipedia.org/wiki/HTTP_status_codes

2.6節　　ヘッダ行

あるヘッダは長くなったときに複数の行で構成できるので、仕様書ではヘッダ・フィールドという言葉を使っている。

ヘッダ・フィールドは、一般ヘッダ(general-header)、要求ヘッダ(request-header)、応答ヘッダ(response-header)及びエンティティ・ヘッダ(entity-header)のフィールドたちで構成される。

ヘッダ行は名前(フィールド名)と値(フィールド値)を":"(コロン)でつないで構成される。この行が長くなった場合は複数の行にわたることが可能だが、その場合は継続行の頭は SP または HT で始まらねばならない。総てのリニア・ホワイトスペース(LWS:画面上では何も表示されない文字のつながり。具体的にはスペース、タブ文字、改行文字のつながり)は単一の SP とみなされる。

例えば、以下の二つのヘッダは等価である:

```
Header1: some-long-value-1a, some-long-value-1b
header1:     some-long-value-1a,
             some-long-value-1b
```

フィールド名は大文字と小文字の区別はされない。

ヘッダ行の BNF 表記は次のようになっている:

```
message-header = field-name ":" [ field-value ]
field-name     = token
field-value    = *( field-content | LWS )
field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, separators, and quoted-string>
```

OCTETS、TEXT、separators、quoted-string などの意味は仕様書の 2.2 節を見られたい。

フィールド名が異なるヘッダ行の順序は重要ではないが、一般ヘッダ、要求または応答ヘッダ、そしてエンティティ・ヘッダの順に送るのが良い慣例(good practice)とされている。

あるメッセージの中に同じフィールド名を持ったヘッダ・フィールドが存在する可能性があるのは、そのヘッダ行

のフィールド値全体がカンマで分離されたリストで定義されているとき(言い換えると , #(values))に限る。カンマで区切ったフィールド値を追加することで、複数のヘッダ・フィールドをひとつの”フィールド名:フィールド値”のペアに統合させることが可能である。その場合はヘッダ・フィールドの順序は意味を持つ。

2.6.1 一般ヘッダ

一般ヘッダは、Cache-Control、Connection、Date、Pragma、Trailer、Transfer-Encoding、Upgrade、Via、及びWarning がある。これらは要求及び応答メッセージの双方で適用可能なヘッダである。

しかしながら、Connection というヘッダ・フィールドは、HTTP/1.1 の要求ヘッダとしてのみ使われる。Upgrade もそうである。また Trailer と Transfer-Encoding いうヘッダ・フィールドは、応答ヘッダで使われる。

従って、これらのヘッダ・フィールドは、次からの節の中で含めて説明する。そのほうが、クライアント側及びサーバ側での開発に便利であろう。

2.6.2 要求ヘッダ

下表は要求ヘッダの一覧である:

表 2-6:要求ヘッダ一覧

フィールド名 (アルファベット順)	記述	例
Accept	受け付け可能なコンテント・タイプ(Content-Type)たち	`Accept: text/plain`
Accept-Charset	受け付け可能な文字セットたち	`Accept-Charset: utf-8`
Accept-Encoding	受け付け可能なエンコーディングたち	`Accept-Encoding: <compress | gzip | identity>`
Accept-Language	応答のために受け付け可能な言語たち	`Accept-Language: en-US`
Accept-Ranges	あるリソースのためのレンジ要求をサーバが受け付けることを示すことを可能とする。HTTP/1.1 では bytes というレンジしか存在しない	`Accept-Ranges: bytes`
Authorization	要求しているリソースの、HTTP 認証のためのレルムに対するユーザエージェントの認証情報を含む信用証明	`Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`
Cache-Control	要求/応答のチェインのなかの総てのキャッシング・メカニズムたちに従わねばならない指示たちを指定するのに使われる	`Cache-Control: no-cache`
Connection	どのタイプの接続をそのユーザ・エージェントが求めているかを示す Connection: close が要求または応答ヘッダ・フィールドに存在するときは、その接続が現在の要求/応答が終わったあとは「継続的」でないとみなさねばならない。継続した要求に対応していないアプリケーションは、各メッセージに必ず"close"接続オプションを含めねばならない	`Connection: close`
Cookie	以前にサーバから次の表の `Set-Cookie` で送られた HTTP クッキー	`Cookie: $Version=1; Skin=new;`
Content-Length	要求ボディの長さをオクテット(8 ビット・バイト)で示す	`Content-Length: 348`
Content-Type	その要求のボディ(POST と PUT 要求で使われる)の MIME タイプ	`Content-Type: application/x-www-form-urlencoded`
Date	そのメッセージが送信された日付と時刻	`Date: Tue, 15 Nov 1994 08:12:31 GMT`
Expect	クライアントが特定のサーバ行為を要求していることを示す	`Expect: 100-continue`
From	その要求をしているユーザの電子メール・アドレス	`From: user@email.com`

Host	そのサーバ(バーチャル・ホスティングでの)のドメイン名、**HTTP/1.1 では必須ヘッダ**	Host: en.wikipedia.org
If-Match	クライアントが出したエンティティとサーバ上の同じエンティティが一致しているときにのみ、そのアクションを実施 これはその前に更新したとき以来変更されていないときにのみあるリソースを更新するような OUT のようなメソッドで使われる	If-Match: "737060cd8c284d8af7ad3082f209582d"
If-Modified-Since	コンテントに変更がないときに *304 Not Modified* を返せるようにする	If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
If-None-Match	コンテントに変更がないときに *304 Not Modified* を返せるようにする そのリソースからひとつ以上のエンティティを取得していたクライアントが、それらのエンティティのいずれも最新であることを確認するためのもの	If-None-Match: "737060cd8c284d8af7ad3082f209582d"
If-Range	クライアントがそのキャッシュにあるエンティティの部分的コピーを持っていて、そのキャッシュにエンティティ全部の最新コピーを欲しいときに、条件つき GET(If-Unmodified-Since と If-Match のどれかか双方で)のこの Range 要求ヘッダを使う あるいはそのエンティティに変更がない場合は抜けている個所の送信を要求する	If-Range: "737060cd8c284d8af7ad3082f209582d"
If-Unmodified-Since	指定した時刻以来そのエンティティに変更がされていないときに限り応答を送信することを要求	If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT
Max-Forwards	プロキシあるいはゲートウェイを介してそのメッセージが転送できる回数の上限を指定	Max-Forwards: 10
Pragma	要求-応答チェインのどこかでいろんな効果を与える実装固有のヘッダ。現在は"no-cache"のみ	Pragma: no-cache
Proxy-Authorization	あるプロキシへの接続のための認証の信用証明	Proxy-Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Range	あるエンティティの部分のみを要求。bytes は 0 から始まる	Range: bytes=500-999
Referer	現在要求しているページへのリンクが張ってあったそれまでのウェブ・ページの相対あるいは絶対 URI	Referer: http://en.wikipedia.org/wiki/Main_Page
TE	そのユーザ・エージェントが受け付けたい転送エンコーディング:応答ヘッダとして同じ値の Transfer-Encoding が使用可能であることに加えて、チャンクの最後のゼロ・サイズのチャンクのあとに、更に追加ヘッダ(トレーラ)を受け付けることをサーバに通知する	TE: trailers, deflate;q=0.5
Upgrade	サーバに対し別のプロトコルにアップグレードすることを求める	Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11
User-Agent	そのユーザ・エージェントのユーザ・エージェント文字列	User-Agent: Mozilla/5.0 (Linux; X11)
Via	サーバに対しその要求がどのプロキシ経由で送信されたかを知らせる	Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
Warning	エンティティ・ボディ内に問題があるときの警告	Warning: 199 Miscellaneous warning

出典:http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

2.6.3 応答ヘッダ

下表は応答ヘッダの一覧である:

表 2-7:応答ヘッダ一覧

フィールド名	記述	例

（アルファベット順）		
Accept-Ranges	このサーバはどんな部分コンテント・レンジのタイプに対応しているのかを示す bytes と none が定められている	Accept-Ranges: bytes
Age	そのオブジェクトがプロキシのキャッシュ内にいた時間を秒で示す	Age: 12
Allow	あるリソースに対して有効なアクション。*405 Method not allowed* のために使われる	Allow: GET, HEAD
Cache-Control	サーバからクライアントまでの間の総てのキャッシングのメカニズムたちにこのオブジェクトはキャッシュ出来るかを指定	Cache-Control: no-cache
Content-Encoding	そのデータに使われているエンコーディング	Content-Encoding: gzip
Content-Language	そのコンテントの言語	Content-Language: da
Content-Length	応答ボディ内の長さをオクテット（8 ビット・バイト）で示す	Content-Length: 348
Content-Location	返されたデータの代替的場所	Content-Location: /index.htm
Content-Disposition	ある MIME タイプにたいし"File Download"のダイヤログを発生させる	Content-Disposition: attachment; filename=fname.ext
Content-MD5	その応答のコンテントの Base64 エンコードされた MD5 サム	Content-MD5: Q2hlY2sgSW50ZWdyaXR5IQ==
Content-Range	この部分メッセージは全体のボディ・メッセージのどこに位置にあるのかを示す	Content-Range: bytes 21010-47021/47022
Content-Type	このコンテントの MIME タイプ	Content-Type: text/html; charset=utf-8
Date	このメッセージが送信された日付と時間	Date: Tue, 15 Nov 1994 08:12:31 GMT
ETag	あるリソースの特定のバージョンの識別子、しばしばメッセージ・ダイジェスト	ETag: "737060cd8c284d8af7ad3082f209582d"
Expires	その応答が期限切れとみなされる日日と時間を与える	Expires: Thu, 01 Dec 1994 16:00:00 GMT
Last-Modified	要求されたオブジェクトが最後に修正された日と時間	Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT
Location	回送（リダイレクション）で、あるいは新しいリソースが作られているときに使われる	Location: http://www.w3.org/pub/WWW/People.html
Pragma	要求-応答チェインのどこかでいろんな効果を与える実装固有のヘッダ。現在は"no-cache"のみ	Pragma: no-cache
Proxy-Authenticate	プロキシにアクセスする為に認証を要求	Proxy-Authenticate: Basic
Refresh	回送（リダイレクション）で、あるいは新しいリソースが作られているときに使われる このリフレッシュは 5 秒後に回送を開始する これは Netscape 他が導入した独自の非標準のヘッダだが、殆どのブラウザが対応している	Refresh: 5; url=http://www.w3.org/pub/WWW/People.html
Retry-After	あるエントリが一時的に取得できないとき、クライアントに指定した時間後に再試行することを指示	Retry-After: 120
Server	このサーバの名前	Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Set-Cookie	HTTP クッキー	Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1
Trailer	このフィールド値は、チャンク形式の転送エンコーディングのメッセージのトレーラ部分に指定されたヘッダ・フィールドが入っていることを示す	Trailer: Max-Forwards
Transfer-Encoding	そのユーザの為のエンティティを安全に転送する為に使われているエンコーディングの形式 現在定義されているメソッドは：chunked, compress, deflate, gzip, identity.	Transfer-Encoding: chunked

Vary	下り方向にあるプロキシたちに今後の要求ヘッダたちをどのように一致判断するかを示して、もとのサーバからの新鮮なものを要求する代わりにキャッシュされた応答が使えるかどうかを判断できるようにする	`Vary: *`
Via	クライアントに対しその応答がどのプロキシを介して送られているかを知らせる	`Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)`
Warning	エンティティ・ボディ内に問題がある場合のための一般警告	`Warning: 199 Miscellaneous warning`
WWW-Authenticate	要求されているエンティティにアクセスする為に使わねばならない認証スキームを知らせる	`WWW-Authenticate: Basic`

出典：http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

なお、**サーブレットの仕様書では、サーブレット・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とも記されている。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)
```

2.7節　メッセージ・ボディ

HTTP メッセージのメッセージ・ボディは要求または応答に関わるエンティティのボディ伝達のために使われる。メッセージ・ボディは、エンティティそのもの、あるいは Transfer-Encoding ヘッダ・フィールドで指定された転送コーディングがされたエンティティである。

```
      message-body = entity-body
                   | <entity-body encoded as per Transfer-Encoding>
```

メッセージ・ボディがどのようなときに許されるかは、要求と応答で相違がある：

要求の中にメッセージ・ボディが存在することは、その要求のメッセージ・ヘッダの中に Content-Length あるいは Transfer-Encoding ヘッダ・フィールドが存在することで識別される。メッセージ・ボディが許されない要求メソッドをもつ要求には、メッセージ・ボディを含めてはならない。

応答メッセージの場合は、メッセージにボディが含まれるかどうかは、要求メソッドと応答ステータス・コードの双方に依存する。HEAD 要求に対する総ての応答はメッセージ・ボディを含めてはならない。総ての 1xx (informational)、204 (no content)、及び 304 (not modified)ステータス応答メッセージはメッセージ・ボディを含めてはならない。それ以外の総ての応答は、長さがゼロである場合があるにしろ、メッセージ・ボディを含む。

2.7.1　MIME エンコーディング

MIME（マイムと発音）エンコーディング(MultiPurpose Internet Mail Extensions encoding)は、その名のとおりインターネット・メールで画像などのバイナリ・データを添付送信する為に開発されたエンコーディングで、バイナリ・データを 7 ビットの文字コード列に変換するものである。これは URL エンコーディングでも説明したように、歴史的なインターネットの伝送系の制約からと、文字に変換することによるチェックのし易さからによる。HTTP もそのメッセージ形式はインターネット・メールの SMTP に似ており、このエンコーディングが使われることが多い。

HTTP そのものは MIME 対応プロトコルではないが、HTTP/1.1 メッセージでは MIME-Version ヘッダ・フィールドを含めて、そのメッセージにはどの MIME のバージョンが使われているかを示すことが出来る。

MIME は RFC 2065 で規定されている。

```
Content-Transfer-Encoding: mechanism
```

メカニズムには、"7bit"、"8bit"、"binary"、"quoted-printable"、及び"base64"が指定できるが、一般的なのは base64 である。これは 3 オクテット (24 ビット) を 6 ビットずつ 4 つに分割し、各 6 ビットの値に対してそれぞれ US-ASCII の 64 文字(英字 52 文字、数字 10 文字、「+」、「/」)を割り当てる。効率は落ちる(データ量が 33%増える)が、確実な手段である。

2.7.2 チャンク転送方式

これは HTTP/1.1 固有の方式である。

サーバ側で応答メッセージの全体をまとめるのに時間がかかる(データベース・アクセスなど)場合は、応答メッセージが全部まとまってから送信を開始するのではなく、出来たものから順にチャンク(細切れ)としてクライアントに送信したほうが、サーバは全体のバイト長を知る前に応答を送信できるので、動的に作成されるページを高速にかえすことができる。ブラウザ側では一括表示処理を行うよりも早めに届いたものから表示できるので、ユーザに与える印象は格段に違うことになる。

チャンク転送コーディングされたボディは BNF 表記では次のようになる:

```
      Chunked-Body   = *chunk
                     last-chunk
                     trailer
                     CRLF
チャンク・ボディは複数のチャンク、最終チャンク、オプションのトレーラ、そして空白行からなる
      chunk          = chunk-size [ chunk-extension ] CRLF
                     chunk-data CRLF
      chunk-size     = 1*HEX
      last-chunk     = 1*("0") [ chunk-extension ] CRLF
各チャンクは16進表示のサイズ行、チャンク・データ行で構成される。最後のチャンクのサイズはゼロである
      chunk-extension= *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
      chunk-ext-name = token
      chunk-ext-val  = token | quoted-string
      chunk-data     = chunk-size(OCTET)
      trailer        = *(entity-header CRLF)
```

例えば、チャンク転送を使った応答メッセージは次のようになる:

```
      HTTP/1.1 200 OK
      Date: Fri, 31 Dec 1999 23:59:59 GMT
      Content-Type: text/plain
      Transfer-Encoding: chunked
      空白行
      1a; ignore-stuff-here
      abcdefghijklmnopqrstuvwxyz
      10
      1234567890abcdef
      0
      some-footer: some-value
      another-footer: another-value
```

```
空白行
```

最後のフッタの後に空白行が必要なことに注意されたい。テキストデータ全体は”abcdefghijklmnopqrstuvwxyz1234567890abcdef”であり、そのバイト長は 10 進で 42 (16 進の 1a+10) である。フッタは、あたかもそれが本体の前のヘッダ部分にあるごとく、ヘッダと同じように処理されなければならない。チャンク・データにはどのようなバイナリデータを含めても構わないし、この例よりもっと長くても構わない。

比較のために、チャンクド・エンコーディングを使用しない場合の前事例と等価な応答メッセージを示す:

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Content-Length: 42
some-footer: some-value
another-footer: another-value
空白行
abcdefghijklmnopqrstuvwxyz1234567890abcdef
空白行
```

第3章　開発環境(Eclipse)とそのインストール

本チュートリアルでは、Java アプリケーション開発環境として広く普及しているEclipse を採用している。Eclipse に関しては、当社のもうひとつのチュートリアルである「Java によるTCP プログラミング」の第1 章で詳しく説明してあるので、そちらを読んで頂きたい。ここではサーブレット開発のために必要な事項のみを説明する。

Eclipse は 1990 年代中頃出現した Java ベースの IDE たちのひとつである IBM の VisualAge for Java を初期コードベースとしている。10 年前に出した当社のチュートリアルの最初のバージョンは VisualAge for Java を使用したものだった。当時は IBM は Eclipse の広範な普及にはサード・パーティたちの活発なエコシステムが不可欠だと考えていたが、同社のビジネス・パートナたちはあまり積極的でなかったため 2001 年 11 月にオープン・ソースのライセンシングと運用モデルを採用することを決定している。その結果 IBM と他の 8 組織からなる Eclipse のコンソーシアムを設立している。その後同社は、より IBM から切り離された組織にすることを決め、その結果独立した Eclipse Foundation という非営利組織が 2004 年 1 月に設立され、現在に至っている。

3.1節　Java EE 6 SDK のインストール

JRE だけでなく、SDK を含めてインストールする場合の方法を示す。ここではサーブレット開発のために現時点で最新の Java EE 6 SDK をインストールすることにする。これは 2009 年 12 月 10 日にリリースされたもので、サーブレット 3.0 に対応している。特にウェブ・アプリケーション向けの軽量なプロファイルの「ウェブ・プロファイル(Web Profile)」が新たに用意されている。これはウェブ・アプリケーションに必要な API セットを選択し軽量化したものである。しかしこれは、単純な Servlet コンテナだけではなく、トランザクション管理、セキュリティー、永続化も重要な要素として含められており、EJB Lite や JPA、JTA といった API も含まれているのが特徴である。**Java EE 6 SDK をインストールするにはあらかじめ Java SE SDK のインストールが必要**である。

なお Windows 用の jdk1.6.0_21 及びそれ以降では、OutOfMemoryError を生じるというバグがある。従ってそれ以前のバージョンの JDK 6 を使うか、3.3 節で示すように、eclipse.ini ファイルの最後に-XX:MaxPermSize=256m という行を追加する必要がある。

1. 最初に Java SE のダウンロードのページから最新の JDK 6 のアップデートをダウンロードする(Download JDK の赤いボタンをクリックする)
2. ログオンのダイヤログが出たら skip をクリックしてとばす。ダウンロードするファイルは、例えば jdk-6u20-windows-i586.exe という 78.5MB の実行ファイルで、これを適当なフォルダにダウンロードする。
3. エクスプローラでこれを開いて実行させる。インストーラが開始する。インストール先は C:\Program Files\Java\jdk1.6.0_20\などになる。
4. JDK がインストールされたら、「環境変数の設定」の節でも示しているように、DOS コマンドでのアクセスを容易にするために環境変数とパスを設定する。これはイ)マイコンピュータを右クリック→プロパティ→詳細設定→環境変数をクリックして変数名：JAVA_HOME、変数値：C:\Program Files\Java\jdk1.6.0_20\を入力。ロ)システム環境変数→Path を選択→編集をクリックしてその文字列の最後に”;C:\Program Files\Java\jdk1.6.0_20\bin”を追加する。ハ)システム環境変数→新規で、変数名 CLASSPATH、変数値 C:\Program Files\Java を登録する。これにより DOS のレベルで任意のディレクトリからこの SDK のコマンドやクラスが完全パスを指定しなくても用意にアクセスできるようになる。たとえばアクセサリのなかのコマンド・プロンプトのウィンドウで以下のように java コマンドを実行することでこれが確認できよう。現在の

ディレクトリが C:\Documents and Settings\であるにも関わらず、パス指定なしで java という DOS コマンドのアクセスが可能になっている:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\****>java -version
java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02)
Java HotSpot(TM) Client VM (build 16.3-b01, mixed mode, sharing)

C:\Documents and Settings\****>
```

JDK をインストールすればウェブ・プロファイル (GlassFish) がインストールできる。ちなみに GlassFish という名前は"transparent development"あるいは" see-through"を意味し、オープン・ソースの透明性を強調して名付けられたと言われる。

5. Java EE のダウンロードのページから Java EE 6 SDK (Web Profile)のダウンロードを選択する。これは java_ee_sdk-6-web-windows という 49MB の実行ファイルであり、これを実行する。
6. インストール・ディレクトリは C:\glassfishv3 とする。Oracle ではこのディレクトリのことを as-install-parent と呼んでいる。また GlassFish がインストールされる C:\glassfishv3\glassfish ディレクトリを as-install ディレクトリと呼んでいる。
7. インストール中にサーバの管理設定を要求されるので、自分のユーザ名とパスワードを入力する。なお管理ポート番号は 4848、HTTP ポート番号は 8080 のままで良い。また更新(インストーラが更新ツールをダウンロードし設定ができるようにする)の設定では、ファイアウオールを使っていない場合はプロキシ・ホストとポートを入力するが、通常はブランクのままで良い。更新ツールのインストールと更新ツールを有効にするの 2 つのチェックボックスをチェックのままとする。
8. JDK の選択はインストーラが選択したもの C:\Program Files\Java\jdk1.6.0_20 とする。
9. 製品の登録をスキップすればインストールが終了する。
10. Oracle ではエンタープライズ・サーバのインストールが終了したら **as-install-parent\bin および as-install\bin の 2 つを PATH に追加することを推奨**している。

3.2節　サーブレット 3.0 対応 Eclipse（WTP 3.2）

Eclipse のウェブ・ツールのプロジェクトである WTP が 2010 年 6 月 23 日に 3.2 版をリリースしている。このバージョンは、Servlet 3.0、JPA 2.0、JSF 2.0、EJB 3.1、及び JAX-RS 1.1 技術からなる Java EE 6 を使ったアプリケーションの開発、実行、及びデバッグに対応している。無論下図のように Servlet 3.0 対応の Tomcat 7 もサポートしている。但し Tomcat 7 は現在ベータ版の状態で、頻繁に修正されているので、ユーザはなるべく最新の WTP をダウンロードして使用することをお勧めする。

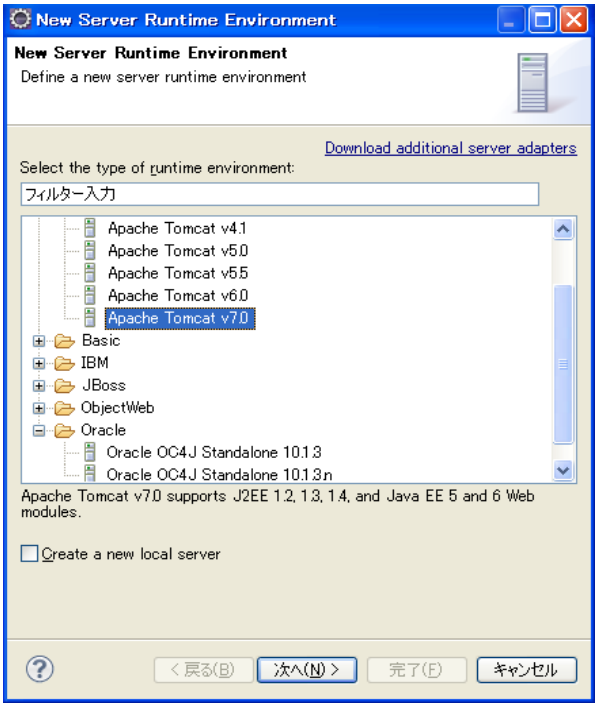


図 3-1 : Eclipse の Tomcat 7 対応

WTP 3.2 は個別にダウンロードできるが、それよりも最新の Eclipse のバージョンをインストールしたほうが手っ取り早い。それ以前のバージョンの Eclipse を使っている人は、現在の Workspace を安全なところに保管して、Eclipse の最新版を再インストールする。

3.3節　　Eclipse のインストール

いよいよ Eclipse のインストールに入る。これに関しては多くのサイトがネット上で存在するのでそちらも参照して頂きたい。例えば九州大学の記事は分かりやすくて推奨される。

なお **Windows 用の jdk1.6.0_21 及びそれ以降を使うと、OutOfMemoryError を生じるというバグがある。**これは 2010 年 7 月の JDK 1.6.0_21 からベンダ名が Sun から Oracle に変わったので、Eclipse のローンチャが Sun のものと認識しなくなり、正確な PermGen サイズを使えなくなっている為である。従ってそれ以前のバージョンの JDK 6 を使うか、以下のように秀丸などのテキスト・エディタを使って eclipse.ini ファイルの最後に"-XX:MaxPermSize=256m"という行を追加する必要がある。

```
-startup
plugins/org.eclipse.equinox.launcher_1.1.0.v20100507.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_1.1.0.v20100503
-product
org.eclipse.epp.package.jee.product
--launcher.defaultAction
openFile
--launcher.XXMaxPermSize
256m
-showsplash
```

```
org.eclipse.platform
--launcher.XXMaxPermSize
256m
--launcher.defaultAction
openFile
-vmargs
-Dosgi.requiredJavaVersion=1.5
-Xms40m
-Xmx512m
-XX:MaxPermSize=256m
```

1. これまで以前のバージョンの Eclipse を使っていた人は、エクスプローラを使ってあらかじめそのバージョンの workspace のフォルダとその内容を安全な場所に保管する。

2. Eclipse のダウンロードのページから最新の Java EE 開発者向けのバージョンをダウンロードする。この資料作成時では Helios のパッケージで、Windows 向けは 32 ビットと 64 ビットの OS 用があるので注意されたい。これは 32 ビット用は eclipse-jee-helios-win32.zip という 206MB もの圧縮ファイルであり、適当な自分のフォルダ、例えばデスクトップのダウンロードのフォルダにダウンロードする。

3. このファイルをエクスプローラで選択右クリックして「すべて展開」を選択し、C:\eclipse-jee-helios-win32 というフォルダに展開させる。そしてこのフォルダ内に展開されている eclipse というフォルダを C:に移動させる。即ち C:\eclipse が Eclipse のフォルダになり、このフォルダには eclipse.exe という実行ファイルが含まれている。

4. 次に日本語の表示が必要な人は Galileo の言語パックをダウンロードすることになる。このプロジェクトのダウンロードのサイトから Galileo を選択すると言語ごとのパック zip ファイルのリストが出てくる。しかし日本語のところにあるこれらのファイルをひとつずつダウンロード・展開するのは面倒である。従ってそのベースとなっている Eclipse Galileo RC3 (3.5.0) 日本語化言語パック (サードパーティ版)を Sourceforge のサイトからダウンロードすることにする。このページに記されているようにまず自分の適当なフォルダに NLpackja-eclipse-SDK-3.5RC3-blancofw20090531.zip をダウンロードし、次にこのファイルをエクスプローラで選択右クリックして「すべて展開」を選択し、C:\eclipse\dropins ディレクトリ以下に展開する。なお Galileo の日本語パックのもう一つのベースである Pleiades (プレアデス)のものもあるので自分でどちらが良いか判断すればよい。

5. Eclipse を起動させる:これはエクスプローラで C:\eclipse\eclipse.exe というアプリケーションを選択ダブル・クリックすればよい。なおデスクトップ上にショートカットを作って配置するほうが便利である。

6. ワークスペース・ラウンチャが開きワークスペースの場所を聞いてくるので、C:\eclipse\workspace などと指定する。以前のバージョンの Eclipse を使っていた人は、保管していた workspace の場所を指定する。なお「この選択をデフォルトとして、今後この質問をしない」というところはチェックしないほうが一般には良いとされている。

7. 下図のようなウェルカム画面が出てくるので、右上の workbench を選択して開く。

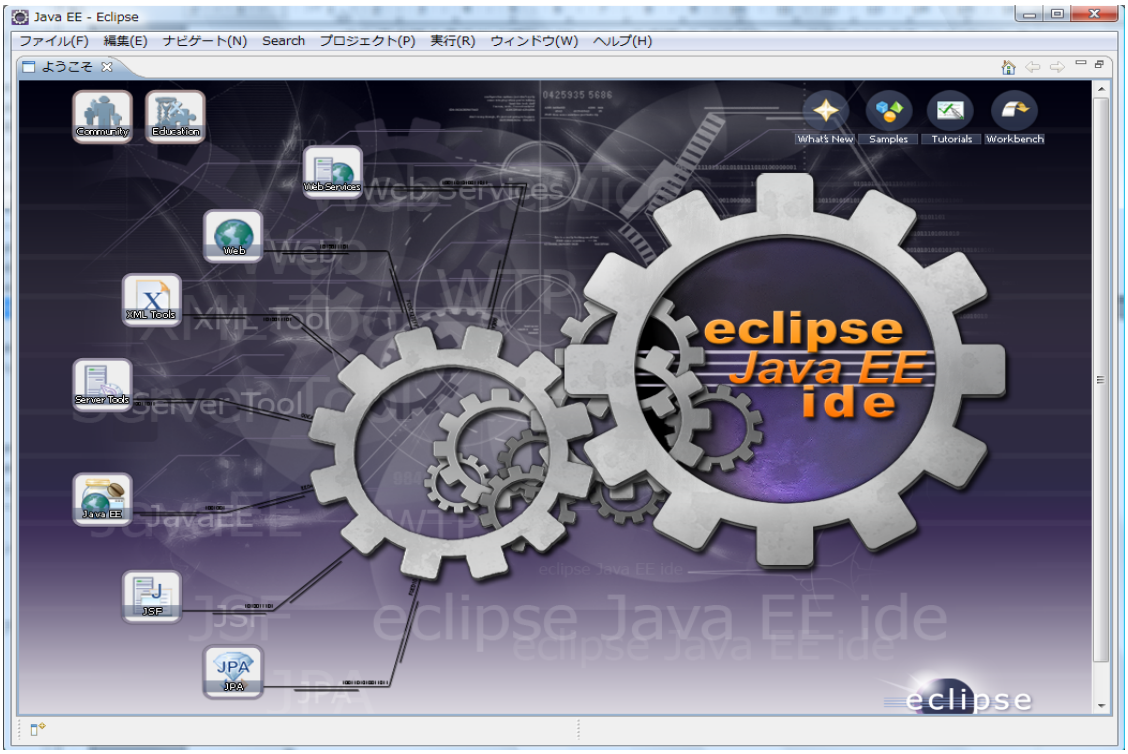


図 3-2：Eclipse のウェルカム画面

8. 最初のワークベンチ・ウィンドウは下図のようである。

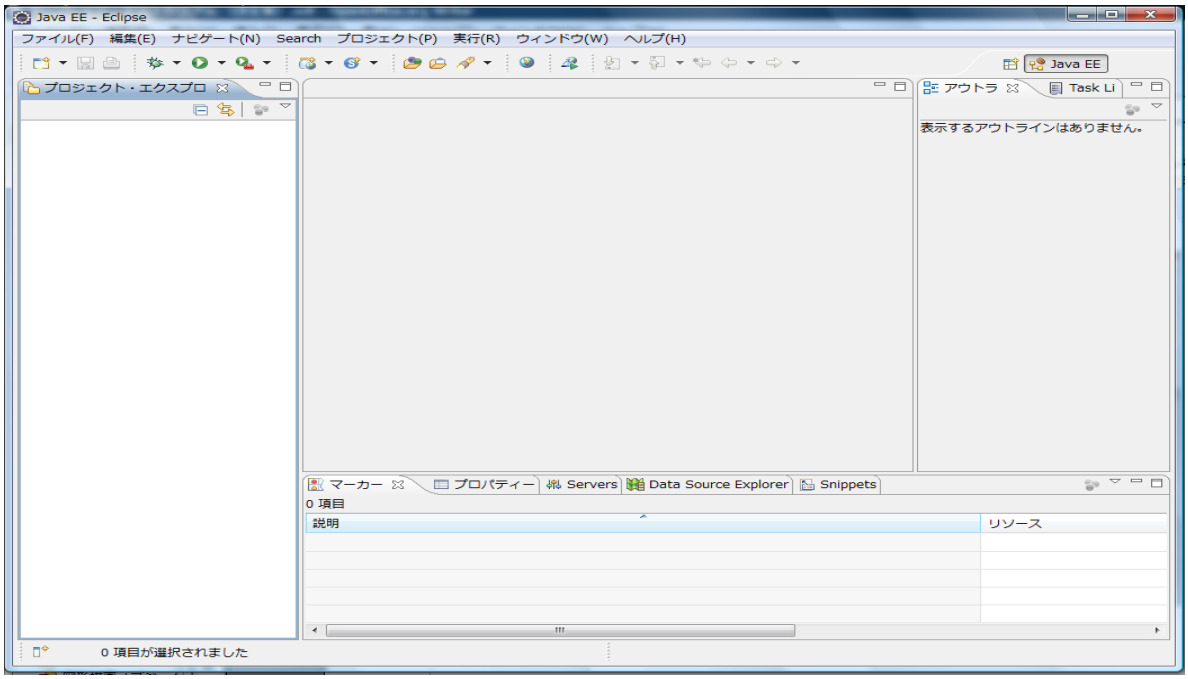


図 3-3：最初のワークベンチのウィンドウ

9. JRE の確認と指定：これはメニュー・バーの「ウィンドウ(Window)」→「設定(Preferences)」→「Java」→「インストール済みの JRE(Installed JRE)」で「検索(S)」で C:\Program Files\java を選択し、見つかったものから選択する。通常は Eclipse が選択したもののままで良いが、**JRE ではなくて Java SDK を使うことが強く推奨されている（特にウェブ・アプリケーションの場合）**。なぜなら JDK には Java ライブラリのソース・コードが含まれており、デバッグがやりやすいからである。これまでの手順に従えば、C:\Program Files\Java\jdk1.6.0_20 が見つかるはずであり、これをチェックする。

10. クラスパス変数の確認と指定：メニュー・バーの「ウィンドウ(Window)」→「設定(Preferences)」→「一般」→「ワークスペース」を開き、「自動的にビルド(B)」が選択されていることを確認する。次にこの設定のペインの「Java」→「ビルド・パス(Build Path)」を選択しソース及び出力フォルダを「プロジェクト(P)」とする。次に「Java」→「エディタ(Editor)」を選択し、「入出力中に問題をレポート(P)」をチェックする。そして「OK」をクリックして設定を終了する。

第4章　Apache Tomcat

4.1節　Apache Tomcat 概説

注：この節は主として Wikipedia の内容をベースにしている。

一般に使われるサーブレット・コンテナとしては、Apache の Tomcat と Oracle の OC4J (Oracle Containers For Java：正式には Oracle Application Server Containers for J2EE)、及びサード・パーティの製品たちがある。ここでは、その中でも長い歴史を持つ Tomcat を中心にして説明する。OC4J を使った場合も必要に応じて説明する。

図 4-1：Tomcat のロゴ

Tomcat のプロジェクトは、Java Servlet 及び Java Server Pages 仕様書の参照実装物の為のものとして、当時の Sun Microsystems 社内でスタートしている。Java コミュニティ・プロセス(JCP：Java Community Process)のサイトでこれらの仕様の詳細が取得できる。

Tomcat のコード・ベースは 1999 年に Sun から Apache Software Foundation (ASF)に寄贈されており、最初の Apache Tomcat のリリースはバージョン 3.0 だった。それ以来多くの Sun や他の組織たちからのボランティアにより、産業に広く受け入れられるところとなり、健全な発展と活発なコミュニティ活動がなされている。

Apache Tomcat (Jakarta Tomcat あるいは単に Tomcat とも呼ばれている) は従って、ASF が開発しているオープン・ソースのサーブレット・コンテナであって、「ピュア Java」のコードが走る HTTP ウェブ・サーバ環境を提供している。

Tomcat は、C で実装されている HTTP ウェブ・サーバである Apache ウェブ・サーバと混同してはならない。これらの 2 つのサーバは一緒にバンドルされていない。しかし Tomcat は Apache ウェブ・サーバにコネクタを使って載せることができる。つまり Apache ウェブ・サーバが動的な処理を要する要求を Tomcat に渡すように出来る。

Apache Tomcat は、設定と管理のためのツールを持っているが、XML 設定ファイルを編集することで設定することも可能である。

4.2節　歴史

Tomcat は当時 Sun Microsystems のソフトウエア技術者だった James Duncan Davidson (ジェームス・デーヴィッドソン) によるサーブレット参照実証としてたち上げられている。彼はその後このプロジェクトのオープン・ソースかと Apache Software Foundation (ASF)への寄贈に寄与している。Tomcat のプロジェクトをオープン・ソース化したことの副効果として、Apache Ant というビルド・オートメーションのツールが開発されている。

Davidson は当初このプロジェクトがオープン・ソース化されることを望み、また多くのオープン・ソースのプロジェクトが O'Reilly 社の動物がカバーに書かれている書籍で解説されていたので、このプロジェクトも動物の名前にすることを望んだ。それで彼は自活できる何かを意味する動物ということで Tomcat と名付けた。O'Reilly のタイトルには既に雄猫(tomcat)が使われていたが、結局 O'Reilly 社は Tomcat の書籍をユキヒョウ(snow leopard)のカバーで出版し、彼が望んだ動物のカバーの希望は実現された。

表 4-1 : Tomcat のバージョン

バージョン	リリース日	対応サーブレット仕様	対応 JSP 仕様	最低 Java バージョン	記述
3.0.x. (最初のリリース)	1999	2.2	1.1	1.1	寄贈された Sun Java Web Server コードと ASF との合体
3.3.2	March 9, 2004	↑	↑	↑	最新の 3.x リリース
4.1.31	October 11, 2004	2.3	1.2	1.3	
4.1.36	March 24, 2007	↑	↑	↑	
4.1.39	December 3, 2008	↑	↑	↑	
4.1.40	June 25, 2009	↑	↑	↑	最新の 4.x リリース
5.0.0	October 9, 2002	2.4	2	1.4	
5.0.23		↑	↑	↑	
5.0.24	May 9, 2004	↑	↑	↑	
5.0.28	August 28, 2004	↑	↑	↑	
5.0.30	August 30, 2004	↑	↑	↑	
5.5.0	August 31, 2004	2.4	2.0	1.4	
5.5.1	September 7, 2004	↑	↑	↑	
5.5.4	November 10, 2004	↑	↑	↑	
5.5.7	January 30, 2005	↑	↑	↑	
5.5.9	April 11, 2005	↑	↑	↑	
5.5.12	October 9, 2005	↑	↑	↑	
5.5.15	January 21, 2006	↑	↑	↑	
5.5.16	March 16, 2006	↑	↑	↑	
5.5.17	April 28, 2006	↑	↑	↑	
5.5.20	September 1, 2006	↑	↑	↑	
5.5.23	March 2007	↑	↑	↑	
5.5.25	September 2007	↑	↑	↑	
5.5.26	February 2008	↑	↑	↑	
5.5.27	September 8, 2008	↑	↑	↑	
5.5.28	September 4, 2009	↑	↑	↑	
5.5.30	July 9, 2010	↑	↑	↑	
5.5.31	September 16, 2010	↑	↑	↑	最新の 5.x リリース

6.0.0	December 1, 2006	2.5	2.1	1.5	
6.0.10	March 1, 2007	↑	↑	↑	
6.0.13	May 15, 2007	↑	↑	↑	
6.0.14	August 13, 2007	↑	↑	↑	
6.0.16	February 7, 2008	↑	↑	↑	
6.0.18	July 31, 2008	↑	↑	↑	
6.0.20	June 3, 2009	↑	↑	↑	
6.0.24	January 21, 2010	↑	↑	↑	
6.0.26	March 11, 2010	↑	↑	↑	
6.0.28	July 9, 2010	↑	↑	↑	
6.0.29	July 22, 2010	↑	↑	↑	最新の安定(stable)バージョン
7.0.0 beta	June 29, 2010	3.0	2.2	1.6 (EE 6)	Servlet 3.0、JSP 2.2 及び EL 2.2 に対応した最初の Apache Tomcat のリリース
7.0.2 beta	August 20, 2010	↑	↑	↑	現時点での最新ベータ版

4.3節　Tomcat の構成要素

Tomcat バージョン 7.x は、サーブレット・コンテナの Catalina（カタリーナ）、HTTP コネクタの Coyote（コヨーテ）、及び JSP エンジンの Jasper 2（ジャスパー 2）が主たる構成要素になっている。ユーザが使えるようにバンドルされている API は次のようになっている：

- annotations-api.jar (アノテーションのパッケージ)
- catalina.jar (Tomcat Catalina 実装)
- catalina-ant.jar (Tomcat Catalina Ant タスク)
- catalina-ha.jar (ハイ・アベーラビリティ・パッケージ)
- catalina-tribes.jar (グループ通信)
- el-api.jar (EL 2.2 API)
- jasper.jar (Jasper 2 コンパイラとランタイム)
- jasper-el.jar (Jasper 2 EL 実装)
- ecj-3.6.jar (Eclipse JDT Java コンパイラ)
- jsp-api.jar (JSP 2.2 API)
- servlet-api.jar (Servlet 3.0 API)
- tomcat-api.jar (Catalina と Jasper が共有しているインターフェイスたち)
- tomcat-coyote.jar (Tomcat コネクタとユーティリティのクラスたち)
- tomcat-dbcp.jar (Commons DBCP に基づいた改名されたデータベース接続プール)

4.3.1　Catalina

Catalina が Tomcat のサーブレット・コンテナである。Catalina は Oracle 社のサーブレットと JavaServer Pages (JSP)の仕様を実装している。Catalina の開発者は Sun の上級スタッフ技術者で Struts なども開発した Craig McClanahan（クレイグ・マクラナハン）だった。

4.3.2 Coyote

Coyote はウェブ・サーバあるいはアプリケーション・コンテナの為の HTTP/1.1 プロトコルに対応した Tomcat の HTTP コネクタのコンポネントである。Coyote は、サーバ上の特定の TCP ポート上の到来接続を聴き、その要求を Tomcat Engine に渡し、そのエンジンがその要求を処理し、応答を要求しているクライアントに送り返すようにする。

4.3.3 Jasper

Jasper は Tomcat の JSP エンジンである。Tomcat 7.x でも Jasper 2 が使われており、これは Oracle 社の JavaServer Pages 2.0 仕様書を実装したものである。Jasper は JSP ファイルを構文解析し、それらを Catalina が処理できるサーブレットとして Java コードにコンパイルする。Jasper は JSP ファイルに変更が出たことを検出し、それを再コンパイルする。

Jasper 2 は以前の Jasper にたいし次のような重要な機能が追加されている:

- JSP タグ・ライブラリのプーリング ： JSP ファイル内の各タグ・マークアップはタグ・ハンドラ・クラスによって処理されている。タグ・ハンドラ・オブジェクトたちはプールされ、JSP サーブレット全体の中で再利用される
- バックグラウンドでの JSP コンパイル ： 手直しされた JSP Java コードを再コンパイルしている間でも、古いバージョンはサーバ要求できる。新しい JSP サーブレットが再コンパイル終了したら、古い JSP サーブレットは削除される
- インクルードされたページに変更が出たときは JSP を再コンパイル ： ランタイム中にページの挿入とインクルードが出来る。JSP は JSP ファイルの変更により再コンパイルされるだけでなく、インクルードされたページの変更でも再コンパイルされる
- JDT Java コンパイラ ： Jasper 2 は Ant と Javadoc の代わりに Eclipse の JDT (Java Development Tools) の Java コンパイラを使用できる

4.4節 Tomcat におけるウェブ・アプリケーションの配備

注:この節は、Apache のドキュメンテーションのアプリケーション開発者向けの「配備」のページの翻訳を中心にしている。

4.4.1 背景

ソフトウエア開発者が作成したソース・コードを何処に置くかを説明する前に、ウェブ・アプリケーションのランタイムにおける構成を説明したほうが有用である。サーブレット API 仕様書の 2.2 版前までは、サーバのプラットホームたち間での一貫性が殆どなかった。しかしながら第 2.2 版において、ウェブ・アプリケーション・アーカイブ(Web Application Archive)を標準のフォーマットとすることが規定された。

ウェブ・アプリケーションは標準のレイアウトにおけるディレクトリとファイルの階層として定義されている。そのような階層はそのアンパック(unpacked)された、すなわち各ディレクトリとファイルが別々にそのファイル・システム内に存在する形式において、あるいはウェブ・アーカイブ（あるいは WAR）ファイルとして知られるパックされた(packed)形式でアクセスできる。前者の形式は開発過程においてはより有用だが、後者はそのアプリケーション

をインストールする為に配布するのに使用される。

ウェブ・アプリケーションの階層のトップ・レベルのディレクトリはまた、そのアプリケーションのドキュメント・ルートでもあり、そこにはそのアプリケーションのユーザ・インターフェイスを構成するHTMLファイルとJSPページたちが置かれる。システム管理者がそのアプリケーションをあるサーバにインストールするときは、その管理者はそのアプリケーションにコンテキスト・パスを割り当てる（詳細は後述）。従って、そのシステム管理者がそのアプリケーションに/catalogというコンテキスト・パスを割り当てたとすると、/catalog/index.htmlを参照している要求URIは、そのドキュメント・ルートからindex.htmlを呼び出すことになる。

4.4.2 標準ディレクトリ・レイアウト

規定されているフォーマットでのウェブ・アプリケーション・アーカイブ・ファイルの作成を奨励する為には、WARフォーマット自身で規定されていると同じ構成で、そのウェブ・アプリケーションの「実行」ファイル（即ち、そのアプリケーションを実行するのにTomcatが実際使うファイルたち）を配置するほうが都合が良い。その為には、以下の中身をそのアプリケーションの「ドキュメント・ルート」ディレクトリに置くことになる。

- *.html, *.jsp, etc. ：そのアプリケーションにとって、クライアントのブラウザにとって可視でなければならないHTMLとJSP、及びその他のファイル（例えばJavaScript、スタイルシート、及びイメージ）たち。大規模なアプリケーションではこれらのファイルを更にサブディレクトリの階層に分割することが選択されるかもしれないが、小規模のアプリケーションでは、これらのファイルをひとつのディレクトリのみにまとめたほうがずっとシンプルである。

- /WEB-INF/web.xml ：これはそのアプリケーションのアプリケーション配備記述子(Web Application Deployment Descriptor)である。これはそのアプリケーションが構成されているサーブレット他のコンポネントたちを記述し、また初期化パラメタたち、及びそのサーバが行ってほしいコンテナが管理するセキュリティ制約を記述したものである。このファイルは以下の節で更に詳しく説明する。

- /WEB-INF/classes/ ：このディレクトリはそのアプリケーションが必要とするJavaのクラス・ファイルたち（JARとして一緒になっていない）を含める。これらのクラス・ファイルはサーブレットのクラスと非サーブレットのクラスの双方が含まれる。これらのクラスたちが幾つかのパッケージになっているときは、その階層構造は/WEB-INF/classes/の下で反映されていなければならない。例えば、あるサーブレット・クラスがcom.mycompany.mypackage.MyServletという名前だったときは、そのファイルは/WEB-INF/classes/com/mycompany/mypackage/MyServlet.classとして置かれていなければならない。

- /WEB-INF/lib/ ：このディレクトリには、サード・パーティのクラス・ライブラリ、あるいはJDBCドライバなどのように、そのアプリケーションにとって必要なJavaのクラス・ファイルたち（及び関連するリソースたち）を含むJARファイルたちが含められる。

そのアプリケーションをTomcatに（あるいは2.2版以降対応のどのサーバにでも）インストールすると、WEB-INF/classes/ディレクトリ内のクラスたち、及びWEB-INF/lib/内にあるJARファイルのなかの総てのクラスたちは、そのウェブ・アプリケーション内での他のクラスたちから可視となる。従って、必要なライブラリのクラスたちの総てをこれらの場所のひとつに含めてしまうと（但しサード・パーティのライブラリでは再配布権のライセンスのチェックが必要）、そのウェブ・アプリケーションのインストールが簡単化される--即ちシステムのクラス・パスの調整（あるいはグローバル・ライブラリ・ファイルをそのサーバにインストールすること）が不要になる。

4.4.3 共有ライブラリ・ファイル

殆どのサーブレット・コンテナと同様に、Tomcatはライブラリの JARファイル（あるいはアンパックされたクラスたち）をインストールするメカニズムを備えており、それらを総てのインストール済みのアプリケーションから見えるよ

うにしている(ウェブ・アプリケーションそのものの中に含める必要がない)。どのように Tomcat がそのようなクラスたちを見つけ供給させているかは、クラス・ローダの節で説明する。Tomcat インスタレーションの中での共有されるコードの置き場所として一般的に使われる場所は**$CATALINA_HOME/lib** である。このディレクトリに置かれた JAR ファイルたちはウェブ・アプリケーションたち、及び内部 Tomcat コードの双方から可視である。これはアプリケーションと内部 Tomcat の双方が使う(たとえば JDBCRealm などで)のに必要とする JDBC ドライバを置くには良い場所である。

標準の Tomcat インスタレーションには、以下のようないろんなインストール済みのライブラリ・ファイルたちが含まれている:

- サーブレットと JavaServer Pages を書くのに不可欠な Servlet 3.0 及び JSP 2.2 の API たち
- JAXP (version 1.2) APIs 対応の XML パーサ、これによりアプリケーションは XML ドキュメントの DOM ベース、あるいは SAX ベースの処理ができる

4.4.4 ウェブ・アプリケーション配備記述子

前述のように、/WEB-INF/web.xml ファイルにはそのアプリケーションのウェブ・アプリケーション配備記述子が含まれている。そのファイル名拡張子がしめしているように、このファイルは XML ドキュメントであり、サーバが知らねばならないそのアプリケーションに関する総てのこと(そのアプリケーションが配備されたときにシステム管理者が割り当てるコンテキスト・パスを除く)を規定している。

配備記述子の文法と意味に関してはサーブレット 3.0 仕様書の第 14 章に記されている。配備記述子を作成編集できる開発ツールがそのうち用意される予定である。一方、原点になるものとして、ベーシックな web.xml ファイルが用意されている。このファイルには、各要素の目的を記述したコメントが入っている。

注意) サーブレット 2.x の仕様書にはドキュメント・タイプ記述子(DTD : Document Type Descriptor)が記されており、Tomcat はそのウェブ・アプリケーションの/WEB-INF/web.xml ファイルを処理する際に、そこで定められている規則を適用している。特に記述要素たち(たとえば<filter>、<servlet>、及び<servlet-mapping>のような)を記述する際は、DTD で決められている順番に記述しなければならない。

4.4.5 Tomcat のコンテキスト記述子

/META-INF/context.xml というファイルはロガー、データ・ソース、セッション・マネージャ設定などの Tomcat 固有の設定のオプションを指定するのに使うことが出来る。このファイルは以前は server.xml として使われていたものに相当するが、現在はこのファイル名は使わないことが推奨されている(ただし現在も使用可能である)。1この XML ファイルは Context 要素を含んでいなければならない。この要素は、Tomcat 設定ドキュメンテーションがその Context 要素に関する情報を含んでいる Host に対応する Host 要素の子供であるか如くみなされる。

4.4.6 Tomcat での配備

殆どの相対パスが解決されるベース・ディレクトリのことを言うのに、ここでは$CATALINA_BASE という変数名を使う。CATALINA_BASE ディレクトリを用意することで Tomcat を複数のインスタンス用に設定することをしていない場合には、$CATALINA_BASE には Tomcat をインストールしたディレクトリである$CATALINA_HOME の値がセットされる。

あるウェブ・アプリケーションが実行されるには、そのアプリケーションはサーブレット・コンテナ上に配備されねばならない。これは開発中であってもそうである。実行環境を揃えるために Tomcat を使う方法についてここで述べ

る。ウェブ・アプリケーションは以下のアプローチのひとつによって Tomcat によって配備される:

- 解凍されたディレクトリ階層を$CATALINA_BASE/webapps/のサブディレクトリとしてコピーする。Tomcat は選択されたサブディレクト名に基づいてそのアプリケーションへのコンテキスト・パスを割り当てる。この方法は開発中はもっとも簡単で早いので、我々が組み立てる build.xml ファイルで使用することにする。

- ウェブ・アプリケーションのアーカイブ(WAR)ファイルを$CATALINA_BASE/webapps/にコピーする。Tomcat が開始するときに、Tomcat は自動的に WAR ファイルを解凍し、そのアプリケーションを実行する。このアプローチは一般的に、サード・パーティの会社が提供している、あるいは社内の開発者たちが用意した付加的アプリケーションを既存の Tomcat のインスタレーションにインストールするのに一般的に使われよう。
  注)このアプローチを採用し、あとで自分のアプリケーションを更新したいときは、その WAR ファイルとともに、Tomcat が作った解凍されたディレクトリを削除し、次に新しい WAR ファイルをコピーして Tomcat を再起動しないと、変更がきちんと反映されない。

- ウェブ・アプリケーションの配備と廃止をするのに、Tomcat のマネージャ(Manager)というウェブ・アプリケーションを使う。Tomcat はデフォルトが/manager というコンテキスト・パス上に配備されるウェブ・アプリケーションを含めるが、これにより実行中の Tomcat サーバを再起動させることなくアプリケーションの配備と廃止が出来る。Manager ウェブ・アプリケーションの詳細に関しては管理者ドキュメンテーションを見ていただきたい。

- 自分のビルド・スクリプトの中で"Manager" Ant タスクを使う。Tomcat は Ant ビルド・ツールの為のカスタム・タスク定義のセットを含めていて、これにより Manager ウェブ・アプリケーションのコマンドたちの実行を自動化できる。これらのタスクは Tomcat 配備ツール(Tomcat Deployer)が使う。

- Tomcat 配備ツール(Tomcat Deployer)を使用する。Tomcat は Ant のタスクたちをバンドリングするパッケージ化されたツールを有しており、配備される前にそのウェブ・アプリケーションの一部である JSP たちを自動的に前もってコンパイルするのに使われる。

自分のアプリケーションを Tomcat 以外のサーブレット・コンテナ上に配備することは、各コンテナに依存することになろうが、サーブレット API 仕様(2.2 版以降)対応の総てのコンテナは、WAR ファイルを受け付けることが要求されている。他のコンテナたちは解凍されたディレクトリ構想を受け付ける(Tomcat が受け付けているように)こと、あるいは共有ライブラリ・ファイルの為のメカニズムを用意すること、は要求されていないが、これらの機能は一般的に使えるようになっていることに注意されたい。

4.5節　ソース・コードの構成

注)この部分はアプリケーション開発者向けのドキュメント(Source Organization)の翻訳がベースになっている。

> *殆どの相対パスが解決されるベース・ディレクトリのことを言うのに、ここでは$CATALINA_BASE という変数名を使う。CATALINA_BASE ディレクトリを用意することで複数のインスタンス用に Tomcat を設定することをしていない場合には、$CATALINA_BASE には Tomcat をインストールしたディレクトリである$CATALINA_HOME の値がセットされる。*

4.5.1 ドキュメント構造

推奨すべきことは、ソース・コードを含むディレクトリ階層を、配備可能なアプリケーションを含むディレクトリ階層とを分離させることである。分離することの利点は次のとおりである:

- そのアプリケーションの「実行可能」バージョンが混ざっていないと、ソース・ディレクトリのコンテンツの管理、移動、及びバック・アップがより容易になる。
- ソース・ファイルのみからなるディレクトリ上で管理するのに、ソース・コードのコントロールがより容易である。
- 配備階層が分離している場合は、そのアプリケーションのインストール可能配布物を構成するファイルたちの選択がずっと容易になる。

以下に示すように Ant 開発ツールを使うと、そのようなディレクトリの生成と処理が苦も無くできる。

あるアプリケーションのソース・コードを含むのに使う実際のディレクトリとファイル階層はかなり自由にできる。しかしながら以下の構成が一般的に適用可能であることがわかっており、以下に説明するサンプルの build.xml 設定ファイルで想定しているものである。これらの総てのコンポネントは、そのアプリケーションのトップ・レベルのプロジェクトのソース・ディレクトリのもとに存在する:

- docs/ - そのアプリケーションのドキュメンテーションで、その開発チームが使っているどのフォーマットでも良い
- src/ - サーブレット、ビーンズ、及びその他のクラスを発生させるそのアプリケーション固有の Java ソース・ファイルたち。これらのソース・コードがパッケージたちで構成されている(強く推奨されている)場合は、そのパッケージの階層はこのディレクトリの下のディレクトリ構造に反映されていなければならない。
- web/ - アプリケーションのクライアントたちからアクセスできるそのウェブ・サイトの静的コンテント(HTML ページ、JSP ページ、JavaScript ファイル、CSS スタイル・シート・ファイル、及びイメージ)。このディレクトリはそのウェブ・アプリケーションのドキュメント・ルートになり、このディレクトリのなかのどのサブディレクトリも、これらのファイルたちをアクセスする為に必要な要求 URI に反映される。
- web/WEB-INF/ - そのアプリケーションに要求される特別な設定ファイルたちで、それらにはウェブ・アプリケーション配備記述子(サーブレット仕様書で規定されている web.xml)、そのアプリケーションの開発者が作ったカスタムのタグ・ライブラリのためのタグ・ライブラリ記述子(tag library descriptors)たち、及びそのウェブ・アプリケーション内に含めたいその他のリソース・ファイルたちが含まれる。このディレクトリはそのドキュメント・ルートのサブディレクトリであるかに見えるが、サーブレット仕様書はこのディレクトリ(あるいはその中のどのファイルも)のコンテンツが直接クライアント要求にたいしサービスされることを禁じている。従って、この場所は重要な設定情報(例えばデータベース接続のためのユーザ名とパスワード)をストアするには都合のよい場所ではあるが、そのアプリケーションがきちんと動作することが要求される。

そのアプリケーションの開発過程においては、一時的に 2 つの更なるディレクトリが生成される:

- build/ - デフォルトのビルド(Ant)を実行中は、このディレクトリにはこのアプリケーションのためのウェブ・アプリケーション・アーカイブ(WAR)のなかの正確なイメージが入る。Tomcat では、$CATALINA_BASE/webapps ディレクトリにコピーするか、あるいはマネージャ(Manager)ウェブ・アプリケーションを介してインストールするかして、このような圧縮されていないディレクトリ内のかたちでアプリケーションを配備することが出来る。後者のアプローチは開発過程においては非常に有用であり、以下にこれを説明する。

- dist/ - Ant dist target コマンドを実行中にこのディレクトリが作られる。これはそのウェブ・アプリケーションのバイナリ配布物の正確なイメージをつくるもので、開発者が準備したライセンス情報、ドキュメンテーション、及び README ファイルたちが含まれる。

これら 2 つのディレクトリは、開発者たちのソース・コードのコントロール・システムによってアーカイブさせてはならないことに注意されたい。何故ならこれらは開発過程において必要に応じ削除や再生成されるからである。その為、変更にたいする恒久的な記録を残そうとして、これらのディレクトリ内のソース・ファイルを編集してはならない。何故なら次回ビルドが行われるとこれらの変更は失われてしまうからである。

外部依存性

そのアプリケーションが外部のプロジェクトあるいはパッケージからの JAR ファイルたち（あるいは他のリソース）を必要とする場合にはどうしたら良いだろうか？一般的な事例は、そのアプリケーションが動作する為に、そのアプリケーションに JDBC ドライバを含めねばならない場合である。

この問題に対し開発者たちによってそのアプローチが異なっている。一部の人たちはこれらの JAR ファイルを必要としている各アプリケーション毎に、ソース・コードのコントロール・アーカイブに、依存している JAR ファイルたちのコピーを記録することを奨励するだろう。しかしながらこのことは、いろんなアプリケーションで同じ JAR を使っている場合、特にその JAR ファイルの異なったバージョンにアップデートしなければならなくなったときなど、に大きな管理上の問題を引き起こす可能性がある。

従って Tomcat のマニュアルでは、そのアプリケーションのソース・コントロール・アーカイブの内部に依存しているパッケージのコピーをストアしないことを推奨している。そうではなくて、外部依存物たちはそのアプリケーションのビルドのプロセスの一部として統合されるべきである。そうすれば、開発システム管理者がインストールしたどの場所からでもその JAR ファイルたちのしかるべきバージョンを常に選択でき、依存している JAR ファイルのバージョンに変更が出た都度自分のアプリケーションを更新することを心配しなくても良くなる。

サンプルとしての Ant build.xml ファイルの中で、これらのファイルに変更が生じたときに build.xml にを手を加えることなく、コピーされるファイルの場所を設定できるビルド属性をどのように指定するかを示したい。その開発者が使うこのビルド属性はアプリケーション単位でカスタム化出来るし、その開発者のホーム・ディレクトリ内にストアされた「標準の」ビルド属性としてデフォルト化できる。

多くの場合、開発システム管理者は既に Tomcat の lib ディレクトリに必要な JAR ファイルたちをインストールしてしまっている。もしこれが終わっていれば何もする必要がない　ー　サンプルの build.xml ファイルは自動的にコンパイルのためのこれらのファイルを含む classpath を作る。

4.5.2　ソース・コードのコントロール

前に述べたように、Concurrent Version System (CVS)のようなソース・コード・コントロール・システムの管理のもとで、そのアプリケーションを構成しているソース・ファイルの総てを置くことが強く推奨される。そうすることを選択した場合は、そのソースの階層の各ディレクトリとファイル（生成されたファイルは除く）は登録されセーブされねばならない。バイナリ・フォーマットのファイル（イメージあるいは JAR ライブラリたち）を登録するときは、そのことを確実にソース・コード・コントロール・システムに示す必要がある。

ソース・コード・コントロール・システムにおいて、開発プロセスによって生成された build/及び dist/ディレクトリにある内容をストアしないことは既に推奨してある。CVS に対しこれらのディレクトリを無視するよう指示する簡単な方法は、以下の内容の.cvsignore（頭のピリオドに注意）という名前のファイルをトップ・レベルのソース・ディレクトリに作ることである：

```
build
dist
build.properties
```

ここにある build.properties の意味は、「開発プロセス」の節で説明する。

開発者たちが使っているソース・コード・コントロール環境の詳細な手順の説明はこのマニュアルの範囲を超えているが、コマンド行 CVS クライアントを使っているときには以下のステップがとられる:

- ソース・コードのステータスをソース・リポジトリ内にストアされているものにリフレッシュし、自分のプロジェクトのソース・ディレクトリに移り、cvs update -dP を実行する。
- ソース・コードの階層に新しいサブディレクトリを作るときは、cvs add {サブディレクトリ名}のようなコマンドで CVS にそれを登録する。
- 最初に新しいソース・コードのファイルを作るときは、それを含むディレクトリに移り、cvs add {ファイル名}のようなコマンドで新しいファイルを登録する。
- あるソース・コード・ファイルが不要になったときは、それを含んでいるディレクトリに移りそれを削除する。次に、cvs remove {ファイル名}のようなコマンドで CVS からその登録を外す。
- ソース・ファイルの作成、修正、及び削除中は、それらの変更はサーバのリポジトリ内にはまだ反映されない。これらの変更を現在のステートとして保管するときは、そのプロジェクト・ソース・ディレクトリに移り cvs commit を実行する。終わったばかりの変更に対する短い記述を書くようプロンプトで促されるが、それは更新されたソース・ファイルの新しいバージョンとともにストアされる

他のソース・コード・コントロール・システムのように、CVS も多くの更なる特徴を持っている(特定のリリースを構成するファイルたちにタグを付すことができる、及び後で合体出来る複数の開発のブランチに対応できることなど)。

4.5.3 BUILD.XML 設定ファイル

Java のソース・コード・ファイルたちのコンパイルと配備階層の生成を管理する為に Ant ツールを使う方法を以下に説明する。Ant は、必要な処理ステップを指定している通常 build.xml と呼ばれるビルド・ファイルのコントロールのもとで動作する。このファイルはそのソース・コードの階層のトップ・レベルのディレクトリにストアされており、そのソース・コード・コントロール・システムに登録されていなければならない。

Make ファイルと同じように、この build.xml ファイルはオプション的な開発活動(例えばそれに関わる Javadoc ドキュメンテーションの作成、配備のホーム・ディレクトリを消去して最初からそのプロジェクトの構築をする、あるいは WAR ファイルを作ってそのアプリケーションの配布を可能にする)をサポートする幾つかの「ターゲット」を用意している。良く組み立てられた build.xml ファイルには、開発者が使うよう設計されたターゲット対内部で使われるターゲットを記述した内部ドキュメンテーションを含むことになろう。そのプロジェクトのドキュメンテーションを表示するよう Ant に依頼するには、build.xml ファイルを含んでいるディレクトリに移って、次のようにタイプする:

```
ant -projecthelp
```

作業を順調に進めれるように、ベーシックな build.xml ファイルが用意されており、開発者はそれをカスタマイズし、自分のアプリケーションの為のプロジェクト・ソース・ディレクトリにインストールできる。このファイルには実行できるいろんなターゲットを記述したコメントたちが含まれている。簡単にいえば、以下のようなターゲットが一般的に用意されている:

- clean – このターゲットは既存の build と dist のディレクトリを削除し、最初からこれらの再構築が出来るようにする。これにより、総ての影響を受けるクラスたちの再コンパイルがされないことによるランタイム時の

問題をもたらすことになるソース・コードの修正が、未だなされていない状態であることが保障される。

- compile – このターゲットは前回コンパイルが起きたとき以来変更が起きていないどのソース・コードをもコンパイルするのに使われる。結果としてのクラス・ファイルたちはそのビルド・ディレクトリの WEB-INF/classes というサブディレクトリに作られ、これはまさしくウェブ・アプリケーションがそうあることを必要としている構造である。このコマンドは開発中は頻繁に実行されるので、通常これは「デフォルト」のターゲットになっており、シンプルな ant コマンドでこれが実行される。

- all – このターゲットは compile ターゲットが続く clean ターゲットを走らせるためのショートカットである。従って、アプリケーション全体を再コンパイルされることが保障され、互換性がない変更を気がつかないうちにしてしまうことが起きないことが確保される。

- javadoc – このターゲットはこのウェブ・アプリケーション内の Java クラスたちの Javadoc API ドキュメントを生成する。サンプルの build.xml は、その app 配布物に API ドキュメンテーションを含むことをユーザが望んでいると想定しており、これは dist ディレクトリのサブディレクトリ内にこの docs を生成する。通常はコンパイル毎に Javadoc を生成する必要はないので、このターゲットは通常は dist ターゲットの依存物であるが、compile ターゲットの依存物ではない。

- dist – このターゲットはそのアプリケーションのための配布ディレクトリをつくり、そのディレクトリには必要とされるドキュメンテーション、Java クラスたちの Javadoc、そのアプリケーションをインストールするシステム管理者に渡されるウェブ・アプリケーション・アーカイブ(WAR)ファイル、が含められる。このターゲットはまた配備ターゲットにも依存するので、そのウェブ・アプリケーション・アーカイブはまた配備時に含められる外部依存物たちも拾い上げる。

Tomcat を使ってそのウェブ・アプリケーションをインタラクティブに開発しまたテストする為に、以下の付加的なターゲットが定められている:

- install – 現在走っている Tomcat に対し、開発中のそのアプリケーションが実行とテストのためにすぐに使えることを知らせる。このアクションでは Tomcat の再スタートが必要でないが、Tomcat が次回再スタートした後はこれは無効になる。

- reload – 一旦そのアプリケーションがインストールされたのちに、変更を行い compile ターゲットを使って再コンパイルを続けることができる。Tomcat は自動的に JSP ページになされた変更を認識するが、サーブレットまたは JavaBean のクラスは認識しない　－　このコマンドは Tomcat に対し現在インストールされているアプリケーションを再スタートするように告げ、そのような変更が認識されるようにする。

- remove - 開発とテストが終了したら、オプション的に Tomcat に対しこのアプリケーションをサービスから外すよう告げることが出来る。

開発とテストのターゲットを使うときは、さらにワンタイムのセットアップが必要になるが、それは次節に記されていいる。

4.6節　　開発プロセス

注)この部分はアプリケーション開発者向けのドキュメント(Development Processes)の翻訳がベースになっている。Apache Tomcat のマニュアルは、IDE ではなくて CVS (ソース・コード・コントロール)と Apache Ant (ビルダ)を使う

ことを前提に書かれている。多くの人たちはEclipse のようなIDE を利用しているので、その場合はこの節は読む必要は無い。

アプリケーション開発にはいろんな様式をとり得るが、ここでは Tomcat を使ったウェブ・アプリケーション作成のためのかなり汎用的なプロセスを提案している。以下のセクションではそのコードの開発者として実施することになるコマンドとタスクを示している。複数のプログラマによる開発においても、しかるべきソース・コード・コントロール・システムを使っており、ある時間にそのアプリケーションのどの部分を誰が作業しているかのチーム内規則を持っている限り、おなしベーシックなアプローチが有効である。

以下に記されたタスクは、ソース・コード・コントロールとして CVS が使われており、また既にしかるべき CVS リポジトリにアクセスするよう設定されていることを想定している。そのことを行うための手順はこの説明の範囲外である。別のソース・コード・コントロールが使われている場合は、そのシステムのための対応するコマンドを使わねばならない。

4.6.1 開発のための Ant と Tomcat の初回設定

Manager という Tomcat のためのウェブ・アプリケーションと関わり合うための特別な Ant タスクを活用する為には、以下のタスクを 1 回行う必要がある（どれだけ多くのアプリケーションを開発する予定かということに関わりなく）。

- Ant のカスタム・タスクを設定する。この Ant カスタム・タスクの実装物は、$CATALINA_HOME/lib/catalina-ant.jar という名前の JAR ファイルで、開発者の Ant インスタレーションの lib ディレクトリにコピーしないといけない。

- ひとりあるいはそれ以上の Tomcat ユーザを定める。Manager ウェブ・アプリケーションはセキュリティ制約のもとで動作し、ユーザはログインが必要で、またその人物にセキュリティ・ロール・マネージャを割当てる必要がある。そのようなユーザをどのように設定するかは Tomcat の conf/server.xml でどのレルムを設定したかに依存する。詳細は Apache Tomcat のレルム設定のページを見られたい。任意の数のユーザ（あるユーザ名とパスワードを持った）にマネージャ・ロールを指定できる。

4.6.2 プロジェクト・ソース・コードのディレクトリを作る

最初のステップは新しいプロジェクト・ソース・コードのディレクトリを作り、これから使うことになる build.xml と build.properties ファイルをカスタマイズすることである。ディレクトリ構造はこれまでの節で記述してあるが、出発点としてサンプル・アプリケーションを使うこともできる。

プロジェクト・ソースのディレクトリを作り、それを CVS リポジトリのなかで定義する。これは次のような一連のコマンドでなされる。ここに{project}はそのプロジェクトが CVS レポジトリにストアされる際の名前であり、{username}は開発者のログイン・ユーザ名である。

```
 cd {my home directory}
mkdir myapp <-- 「プロジェクト・ソース・ディレクトリ」だとする
cd myapp
mkdir docs
mkdir src
mkdir web
mkdir web/WEB-INF
cvs import -m "Initial Project Creation" {project} \
   {username} start
```

ここでこれが正しく CVS 内に生成されたかを確認する為に、その新しいプロジェクトのチェックアウトを実施する：

```
 cd ..
mv myapp myapp.bu
cvs checkout {project}
```

次に、開発のために使われる build.xml スクリプトの初期バージョンを生成とチェックインする必要がある。迅速かつ簡単にスタートする為に自分の build.xml をこのベーシックな build.xml ファイルをベースにするか、あるいは最初からコードを書くかする。

```
 cd {my home directory}
cd myapp
emacs build.xml      <-- 実際のエディタを使いたいなら：
cvs add build.xml
cvs commit
```

CVS commit を行うまでは変更は自分の開発ディレクトリでローカルな状態にある。コミットすることでこれらの変更は同じ CVS リポジトリを共有しているそのチームの他の開発者たちから可視となる。

次のステップは build.xml スクリプト内で名付けられている Ant の属性をカスタマイズすることである。これは自分のプロジェクトのトップ・ディレクトリ内に build.properties という名前のファイルを作ることでなされる。サポートされる属性たちはサンプル build.xml スクリプト内のコメントの中にリストされている。最小限、Tomcat がインストールされている catalina.home 属性と、マネージャ・アプリケーションのユーザ名とパスワードが一般的に必要である。結局次のようなものになろう：

```
# このアプリケーションをインストールする為のコンテキスト・パス
app.path=/hello

# Tomcat 7 インスタレーション・ディレクトリ
catalina.home=/usr/local/apache-tomcat-7.0

# Manager webapp のユーザ名とパスワード
manager.username=myusername
manager.password=mypassword
```

一般的に、CVS リポジトリ内に build.properties ファイルをチェックすることは、それは各開発者の環境によって決まっているので、開発者は望まないと思われる。

次に、ウェブ・アプリケーション配備記述子の初期バージョンを作成する。これはベーシックの web.xml ファイルをベースにできるし、最初から書くこともできる。

```
cd {my home directory}
cd myapp/web/WEB-INF
emacs web.xml
cvs add web.xml
cvs commit
```

これは単にサンプルの web.xml でしかないことに注意されたい。配備記述子の完全な定義はサーブレット仕様書を見られたい。

4.6.3 ソース・コードとページのエディット

編集/ビルド/テストのタスクは、一般的に開発とメンテナンス中における最も一般的な作業であろう。以下の一般原則が適用される。前節のソース・コードの構成のところで記したように、新しく作られたソース・ファイルたちはそのプロジェクト・ソース・ディレクトリの下のしかるべきサブディレクトリに置かれねばならない。

他の開発者たちが行った作業を反映させるために自分の開発ディレクトリをリフレッシュしたいときは位置でも、CVS に対しそうするよう指示できる:

```
cd {my home directory}
cd myapp
cvs update -dP
```

新規ファイルを作成するには、しかるべきディレクトリに移り、ファイルを作成し、それを CVS に登録する。現状のコンテンツで良いとき(ビルドとテストが成功した後で)は、その新しいファイルをリポジトリにコミットする。例えば、新しい JSP ページを作るには:

```
cd {my home directory}
cd myapp/web        <-- 最終的な先はドキュメント・ルート
emacs mypage.jsp
cvs add mypage.jsp
... そのアプリケーションのビルドとテスト...
cvs commit
```

パッケージとして構成されている Java ソース・コードは、そのパッケージ名に一致したディレクトリ階層で構成されねばならない(src/サブディレクトリ下で)。例えば、com.mycompany.mypackage.MyClass.java という名前の Java クラスは、src/com/mycompany/mypackage/MyClass.java というファイルでストアされねばならない。新しいサブディレクトリを作ったときは常にそれを CVS に登録することを忘れないこと。

既存のソース・ファイルを編集するには、一般にはまず編集とテストを行い、次に総てが問題なければ変更したファイルをコミットする。CVS は修正しようとするファイルを「チェックアウト」あるいは「ロック」することを要求するよう設定されている可能性があるが、これは一般的には使われていない。

4.6.4 ウェブ・アプリケーションのビルド

そのアプリケーションのコンパイルの準備が出来たら、以下のコマンドたちを実行する(一般的にはプロジェクトのソース・ディレクトリをセットしたシェル・ウィンドウを開き、最後のコマンドだけが必要になるようにしたいと考えられるだろう):

```
cd {my home directory}
cd myapp        <-- 通常ここでウィンドウをオープンにしたままにする
ant
```

Ant ツールは build.xml ファイルの中のデフォルトの"compile"ターゲットを実行するが、これは新規あるいは更新された Java コードがあればコンパイルする。"build clean"後の最初のコンパイルである場合は、これは総てを再コンパイルする。

アプリケーション全体を再コンパイルさせるときは、次のようにする:

```
cd {my home directory}
cd myapp
ant all
```

これは変更のチェックのまえに、Javac の条件チェックが捕まえない微妙な問題を持ち込まないようにすることは非常に良い慣習である。

4.6.5 ウェブ・アプリケーションのテスト

そのアプリケーションをテストする為に Tomcat のもとにそれをインストールすることになる。最も迅速なやり方はサンプル build.xml に含まれているカスタムの Ant タスクたちを使うことである。これらのコマンドを使うときは次のようなパタンになろう:

- 必要なら Tomcat を開始させる。Tomcat が既に走っていないときは、それを通常のやり方で開始させる必要がある。
- 自分のアプリケーションをコンパイルする。Ant コンパイル・コマンド(あるいはデフォルトである ant だけ)を使用する。コンパイル・エラーがないことを確認する。
- そのアプリケーションをインストールする。このことは Tomcat に対し app.path ビルド属性で定義されているコンテキスト上でそのアプリケーションを開始させることを告げることになる。
- そのアプリケーションをテストする。ブラウザあるいは他のテスト・ツールを使って、そのアプリケーションの機能と動作をテストする。
- 必要に応じ修正と再ビルドを行う。変更が必要だとわかったときは、それらの変更はオリジナルのソース・ファイル内で行い、出力のビルト・ディレクトリで行わない。その後 ant 再コンパイル・コマンドを実行する。これによりその変更が後で保管できる(cvs commit を使って)ようになる　ー　出力ビルド・ディレクトリは削除され、必要に応じ再生成される。
- そのアプリケーションを再ロードする。Tomcat は JSP ページの変更は自動的に認識するが、サーブレットまたは JavaBeans のクラスたちは、そのアプリケーションが再ロードされるまでは古いバージョンを使い続ける。ant reload コマンドをかけることで、これを実行させることが出来る。
- 終わったらそのアプリケーションを削除する。このアプリケーションの作業を完了したら、ant remove コマンドをかけることでそれをライブの実行から外すことが出来る。

テストが完了した後でソース・コード・リポジトリに対し自分が行った変更をコミットすることを忘れてはならない。

4.7節　Tomcat のクラス・ローダ

Tomcat だけでなく、サーブレット・コンテナに置かれるオブジェクトは、親子関係でつながっている複数のクラス・ローダによって生成されている。従ってこれを良く理解しないと、後述のように Static なオブジェクトの適用範囲などで問題を起こすことになる。

Tomcat や J2EE のフレームワークでは幾つかのクラス・ローダが階層化されて使われている。これらはやはり Java 2 の委譲パタン(delegation pattern)が採用されている。以下は Tomcat 6 の階層である。Tomcat 5 からはかなり簡素化されている(Server と Shared という階層がない)ので、注意が必要である。

4.7.1　概説

J2SE 2 環境においては、クラス・ローダは前述のように親子関係のツリーで構成されている。通常はあるクラス・ローダがあるクラスまたはリソースをロードするよう要求されたら、そのクラス・ローダは先ずその親のクラス・ローダ

にその要求を委譲する。そのあとで親のクラス・ローダが要求されたクラスまたはリソースを見つけられなかったときのみ自分のリポジトリ内の検索を行う。ウェブ・アプリケーションのクラス・ローダは後述のようにこれからは少し異なるが、基本原則は変わっていない。

Tomcat が開始したときに、下図のような親子関係を持ったクラス・ローダたちのセットを生成する。この図では上に位置するクラス・ローダがその下のクラス・ローダの親の関係になる:

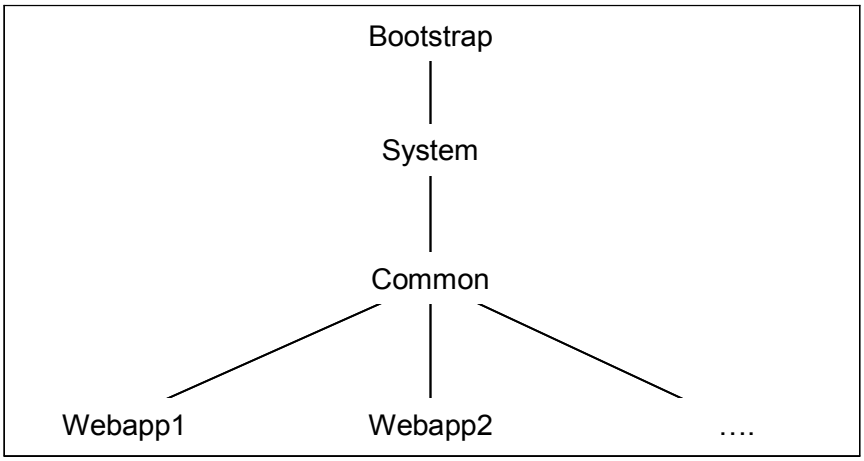


図 4-2 : Tomcat 7 のクラス・ローダ階層

これらの各クラス・ローダの特性(それらが可視にするクラスとリソース源)については、次の項で述べる。

4.7.2 クラス・ローダ定義

前記の図で示したように、Tomcat は初期化されるときに以下のクラス・ローダたちを生成する:

表 4-2 : Tomcat 7 のクラス・ローダたち

Bootstrap	このクラス・ローダは JVM が提供しているベーシックなランタイムのクラスたちに加えて、システム拡張ディレクトリ($JAVA_HOME/jre/lib/ext)にある JAR ファイルたちからのクラスたち、を含む。 注:一部の JVM ではひとつ以上のクラス・ローダとして実装している、あるいはクラス・ローダとして全く見えないようになっていることもある。
System	通常は一般の CLASSPATH の環境変数の内容で初期化される。しかしながら Tomcat では代わりに以下のリポジトリから System クラス・ローダを構築する: • $CATALINA_HOME/bin/bootstrap.jar - Tomcat 6 サーバを初期化するための main()メソッドとそれに属するクラス・ローダ実装クラスを含む • $CATALINA_HOME/bin/tomcat-juli.jar – Jakarta commons logging API と名前が変えられたパッケージと java.util.logging LogManager このクラス・ローダは通常は一般の CLASSPATH の環境変数の内容から初期化される。そのようなクラスたち総ては Tomcat の内部クラスたちから、及びウェブ・アプリケーションたちかの双方から可視である。しかしながら、標準の Tomcat のスタートアップのスクリプト($CATALINA_HOME/bin/catalina.sh あるいは%CATALINA_HOME%\bin\catalina.bat)は CLASSPATH 環境変数のコンテンツそのものを完全に無視しており、その代り以下のリポジトリから System クラス・ローダを構築する: • $CATALINA_HOME/bin/bootstrap.jar - Tomcat サーバを初期化するための main()メソッドと、それが依存しているクラス・ローダ実装クラスを含む • $CATALINA_HOME/bin/tomcat-juli.jar – Jakarta commons logging API と名前が変えられたパッケージと java.util.logging LogManager
Common	このクラス・ローダは Tomcat の内部クラスと総てのウェブ・アプリケーションから見える付加的クラスが含まれる。通常はアプリケーションのクラスはここに置くべきではないと解説に書かれている。このクラス・ローダからは$CATALINA_HOME/lib にある総てのパックされていないクラスとリソー

	ス、及び Jar ファイルの総てのクラスとリソースが可視である。デフォルトとしては以下のものがある: annotations-api.jar – JEE のアノテーションのクラス catalina.jar – Tomcat 6 の Catalina サーブレット・コンテナ部分の実装 catalina-ant.jar - Tomcat Catalina Ant タスク catalina-ha.jar – ハイ・アベーラビリティ(高可用性)パッケージ catalina-tribes.jar – グループ通信パッケージ el-api.jar - EL 2.1 API. jasper.jar - Jasper 2 コンパイラとランタイム jasper-el.jar - Jasper 2 EL 実装 jasper-jdt.jar - Eclipse JDT 3.2 Java コンパイラ jsp-api.jar - JSP 2.1 API. servlet-api.jar - Servlet 2.5 API. tomcat-coyote.jar - Tomcat コネクタとユーティリティのクラス tomcat-dbcp.jar - Commons DBCP に基づいて名前が変えられたデータベース接続プール tomcat-i18n-**.jar – 他の言語の為のリソース・バンドルを含むオプションの Jar たち。デフォルトのバンドルはまたここの Jar に含められるので、メッセージの国際化が不要な場合はこれらは安全に外すことが出来る。
WebappX	各アプリケーションが自分のプライベートなクラスの為に持つローダと考えればよい。このクラス・ローダは単一の Tomcat 7 インスタンスに配備された各ウェブ・アプリケーション毎につくられる。WEB-INF/classes のディレクトリにある総てのパックされていないクラスとリソース、加えてウェブ・アプリケーションのアーカイブの為の/WEB-INF/lib にある Jar ファイルにある総てのクラスとリソースが含まれているウェブ・アプリケーションから可視になるが、それ以外には可視にはならない。

注意しなければいけないのは、親のローダがロードしたインスタンスからは子供のローダがロードしたインスタンスを直接的にアクセスできないことである。その逆は可能である。

WebappX のローダは通常の委譲モデルと違い、要求されたクラス・ファイルがまず自分のローカルなレポジトリを探し、無ければ上位のクラス・ローダにこれを委譲する。これはサーブレット仕様書(例えば 3 版の 10.7.2)に次のように書いてあるからである:
「アプリケーションのクラス・ローダは、WAR のなかにパッケージされたクラスとリソースのほうを、コンテナにわたるライブラリの Jar たちよりも先にロードするよう実装されることが勧告される。」

このクラス・ローダは Servlet 仕様書の第 3 版の 10.7.2 節のウェブ・アプリケーション・クラス・ローダに従い、デフォルトの Java 2 の委譲モデルとは異なっている。このウェブ・アプリケーションの WebappX のクラス・ローダからのあるクラスのロードするという要求が処理されるとき、このクラス・ローダ検索の委譲をしないで、まずローカルなレポジトリを調べる。例外があって、JRE ベース・クラスの部分であるクラスはオーバライドできない。いくつかのクラス(JDK 1.4 以降の XML パーサ・コンポーネントなど)については、J2SE 1.4 の Endorsed 機能が使える。最後に、Servlet API のクラスを含む Jar ファイルすべてをこのクラス・ローダは無視する。Tomcat 6 の他のクラス・ローダはすべて通常の委譲パターンに従う。

従って、ウェブ・アプリケーション側から見れば、クラスあるいはリソースのローディング動作は、以下のリポジトリたちを以下の順で検索する:

- その JVM の Bootstrap クラスたち
- System クラス・ローダ・クラス(上記)
- そのウェブ・アプリケーションの/WEB-INF/classes
- そのウェブ・アプリケーションの/WEB-INF/lib/*.jar
- $CATALINA_HOME/lib
- $CATALINA_HOME/lib/*.jar

4.7.3 XML パーサと JSE 5

多くの変更がなされているなかで、JSE 5 リリースでは JAXP API と Xerces のあるバージョンを JRE のなかにパッケージ化している。このことが自分の XML パーサを使うアプリケーションに影響を与えている。

Tomcat のこれまでのバージョンまでは、総てのウェブ・アプリケーションが使っているパーサを変更するには、$CATALINA_HOME/common/lib ディレクトリ内の XML パーサを変更するだけで良かった。しかしながら JSE 5 上では、通常のクラス・ローダの委譲プロセスが、常に JDK 内の実装物を選択してしまうので、このテクニックは効かなくなってしまう。

JDK 1.5 では「推奨規格標準オーバーライド機構(Endorsed Standards Override Mechanism)」と呼ばれるメカニズムに対応しており、これは JCP の外部で作られた API（言い換えると W3C からの DOM 及び SAX）の交換を可能としている。これはまた XML パーサ実装の更新にも使える。詳細は http://java.sun.com/j2se/1.5/docs/guide/standards/index.html を参照されたい。

Tomcat はそのコンテナを開始させるコマンド行に

```
-Djava.endorsed.dirs=$JAVA_ENDORSED_DIRS
```

をセットして、このシステム属性を含めることでこの機能を活用している。

第5章　Tomcat 7 のインストールと設定

Apache Tomcat プロジェクトは 2010 年 8 月 11 日に 7.0.2 ベータ版をリリースしている。このバージョンは Servlet 3.0、JSP 2.2、及び EL 2.2 のバージョンの仕様書に対応している。この 7.0.2 ベータ版では、これまで問題にされていたメモリー・リークの問題が解決されている。7.0.x バージョンは現在頻繁に修正されているので、ユーザは常に最新のバージョンをダウンロードして使用することをお勧めする。Tomcat 7 は今後数年で最も一般的に使われるサーブレット・コンテナになると予想される。

5.1節　環境変数の設定

Tomcat の Windows 版では、インストーラは J2SE 6 JRE のベース・パスを判断するのに、レジストリまたは JAVA_HOME 環境変数を使用するので、環境変数を最初に設定する。

なおその際、Tomcat を使ったサーブレットの DOS レベルでの開発に便利なように、その他の環境変数も設定しておくとよい。

- CATALINA_BASE は Tomcat が相対パスを解析する為のベース・ディレクトリとなる。
- CLASSPATH は DOS のレベルで JSP やサーブレットのコンパイルをするのに必要である。
- PATH は、DOS のレベルで Java の実行環境にアクセスするのに便利である。
- TOMCAT_HOME も良く引用されるので、設定しておくと良い。

スタート＞設定＞コントロールパネル＞システム＞詳細＞環境設定で、たとえば下記のように必要なパラメタを設定する(JDK の場所やバージョンは自分の環境に合わせる。但し **Tomcat 7 は Java SE 6 以降が必要**である)：

表 5-1：設定する環境変数

JAVA_HOME	C:\Program Files\Java\jdk1.6.0_20
TOMCAT_HOME	C:\tomcat7
CATALINA_HOME	C:\tomcat7
CLASSPATH	C:\tomcat7\lib\servlet-api.jar;c:\tomcat7\lib\tomcat-api.jar;c:\tomcat7\lib\jsp-api.jar;c:\tomcat7\lib\el-api.jar;c:\tomcat7\lib\annotations-api.jar
PATH	C:\Program Files\Java\jdk1.6.0_20\bin

classpath あるいは PATH が既に何かが設定されているときは、そのあとに";"でつないで上の表の内容を追加する。また**環境変数名は大文字と小文字は区別されない**。

注意)Tomcat が複数のインスタンスとして設定されるときは、各インスタンスごとに CATALINA_BASE を用意することになる。通常は複数インスタンスで設定することはないので、CATALINA_BASE は CATALINA_HOME とおなじになる。
注意)PATH は Windows の OS が実行ファイルを見つけるのに使われる。現在のディレクトリではないディレクトリの実行ファイルをファイル名だけで呼び出すには、そのディレクトリへの PATH が切られていなければならない。現在の PATH がどのように設定されているかは、アクセサリのコマンド・プロンプト画面で path コマンドを実行して確認できる。一方 CLASSPATH は JVM がプログラムを走らせるときに Jar ファイルたちを探す場所である。

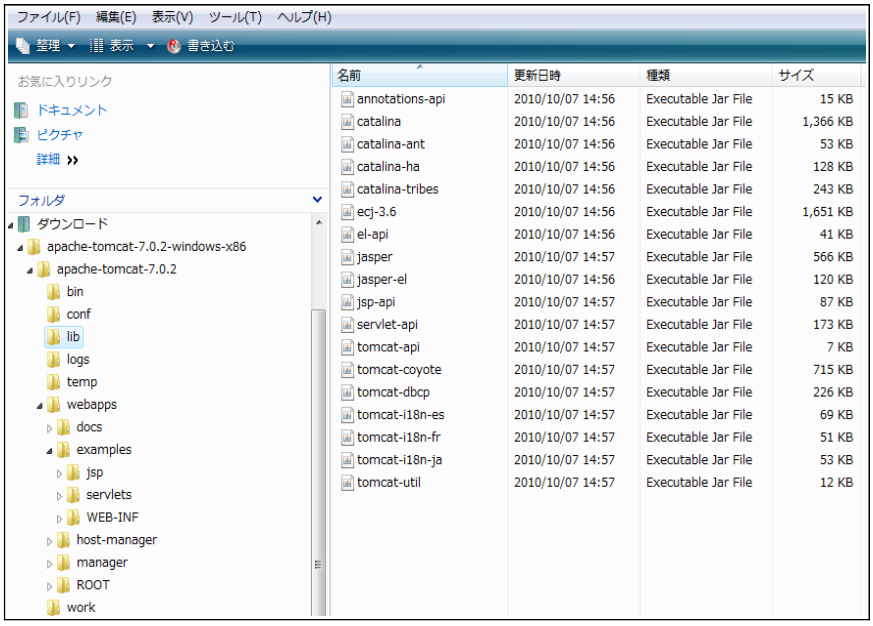
5.2節　　Tomcat のバージョン 7 版をインストール

Tomcat 7 は現在頻繁に更新されているので、常に最新版をインストールすることをお勧めする。ここでは圧縮ファイルをダウンロードしてインストールする方法を説明するが、これ以外に 32-bit/64-bit Windows Service Installer を選択して apache-tomcat-7.0.xx.exe という自己解凍ファイルをダウンロードして実行させる方法もある（この場合は次の節で示すように Windows サービスとして登録される）。

1. Tomcat 7.0.2 (Beta)のダウンロードのページで、最新のバイナリ配布の 32-bit Windows zip (pgp, md5)を選択する（64 ビットの Windows の場合は 64-bit Windows zip (pgp, md5)を選択）。

2. この Zip 圧縮されたファイルを「ダウンロード」などのフォルダに保管し、エクスプローラでこれを「総て展開」する

図 5-1：展開された Tomcat 7



3. apache-tomcat-7.0.2 というフォルダを c:の下に移し、そのフォルダの名前を tomcat7 に変更する

4. エクスプローラで c:\tomcat7\bin\startup.bat というバッチ・ファイルをダブルクリックして起動させると、次のような表示がされ、Tomcat が起動するはずである：

```
2010/10/07 15:55:46 org.apache.catalina.core.AprLifecycleListener init
情報: Loaded APR based Apache Tomcat Native library 1.1.20.
2010/10/07 15:55:46 org.apache.catalina.core.AprLifecycleListener init
情報: APR capabilities: IPv6 [true], sendfile [true], accept filters [false], ra
ndom [true].
2010/10/07 15:55:52 org.apache.coyote.http11.Http11AprProtocol init
情報: Coyote HTTP/1.1 を http-8080 で初期化します
2010/10/07 15:55:52 org.apache.coyote.ajp.AjpAprProtocol init
情報: Initializing Coyote AJP/1.3 on ajp-8009
2010/10/07 15:55:52 org.apache.catalina.startup.Catalina load
```

```
情報: Initialization processed in 6755 ms
2010/10/07 15:55:52 org.apache.catalina.core.StandardService startInternal
情報: サービス Catalina を起動します
2010/10/07 15:55:52 org.apache.catalina.core.StandardEngine startInternal
情報: Starting Servlet Engine: Apache Tomcat/7.0.2
2010/10/07 15:55:52 org.apache.catalina.startup.HostConfig deployDirectory
情報: Web アプリケーションディレクトリ docs を配備します
2010/10/07 15:55:52 org.apache.catalina.startup.HostConfig deployDirectory
情報: Web アプリケーションディレクトリ examples を配備します
2010/10/07 15:55:53 org.apache.catalina.startup.HostConfig deployDirectory
情報: Web アプリケーションディレクトリ host-manager を配備します
2010/10/07 15:55:53 org.apache.catalina.startup.HostConfig deployDirectory
情報: Web アプリケーションディレクトリ manager を配備します
2010/10/07 15:55:53 org.apache.catalina.startup.HostConfig deployDirectory
情報: Web アプリケーションディレクトリ ROOT を配備します
2010/10/07 15:55:53 org.apache.coyote.http11.Http11AprProtocol start
情報: Coyote HTTP/1.1 を http-8080 で起動します
2010/10/07 15:55:53 org.apache.coyote.ajp.AjpAprProtocol start
情報: Starting Coyote AJP/1.3 on ajp-8009
2010/10/07 15:55:53 org.apache.catalina.startup.Catalina start
情報: Server startup in 1408 ms
```

5. 自分のブラウザのアドレスに http://localhost:8080/を入力すると、次のようなページを Tomcat が返すはずである:

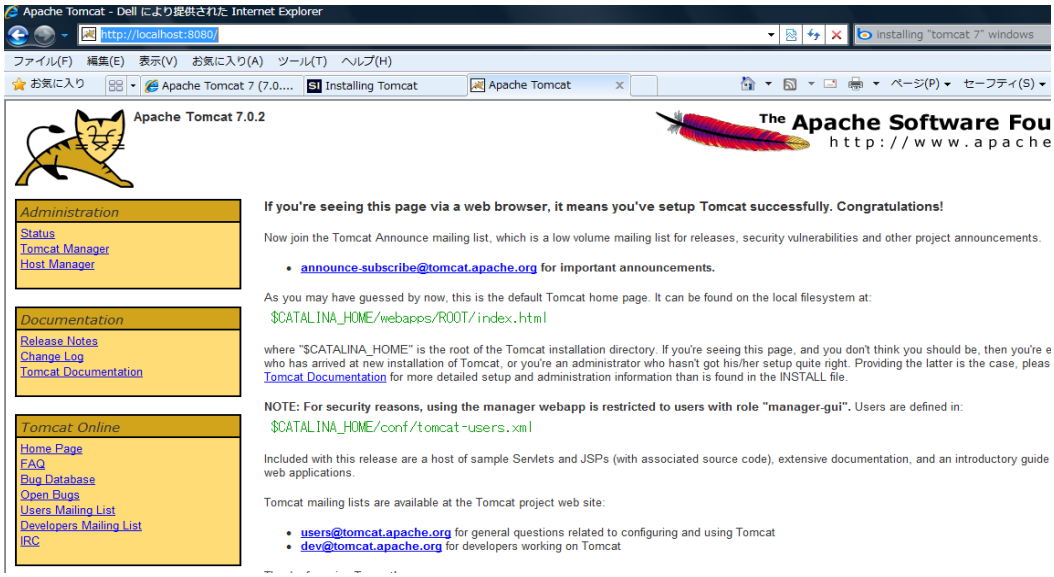


図 5-2:Tomcat 7 のウェルカム画面

これは Tomcat が TOMCAT_HOME/webapps/docs にある index.htm を送り返してくれているからである。ウェルカム画面が表示されない場合は、ブラウザがローカル・アドレスにたいしプロキシをバイパスするように設定されているかを確認する。localhost のポート番号が 8080 なのは、Coyote HTTP/1.1 を http-8080 で初期化してあるからである。

6. エクスプローラで c:\tomcat7\bin\shutdown.bat というバッチ・ファイルを実行させると、Tomcat は終了する。C:\tomcat7\logs\catalina.***のファイルには、次のようなログが記録される。

```
情報: A valid shutdown command was received via the shutdown port. Stopping the Server instance.
2010/10/07 16:15:07 org.apache.coyote.http11.Http11AprProtocol pause
情報: Coyote HTTP/1.1 を http-8080 で一時停止します
2010/10/07 16:15:07 org.apache.coyote.ajp.AjpAprProtocol pause
情報: Pausing Coyote AJP/1.3 on ajp-8009
2010/10/07 16:15:08 org.apache.catalina.core.StandardService stopInternal
```

```
情報: サービス Catalina を停止します
```

なお、startup と shutdown の BAT ファイルは、下図のようにコマンド・プロンプトを使って起動させることが出来る。この場合は画面でわかるように、ディレクトリを c:\tomcat7\bin に移して、startup (あるいは shutdown) を実行する。Tomcat が起動すると、Tomcat のコンソールが表示される。:

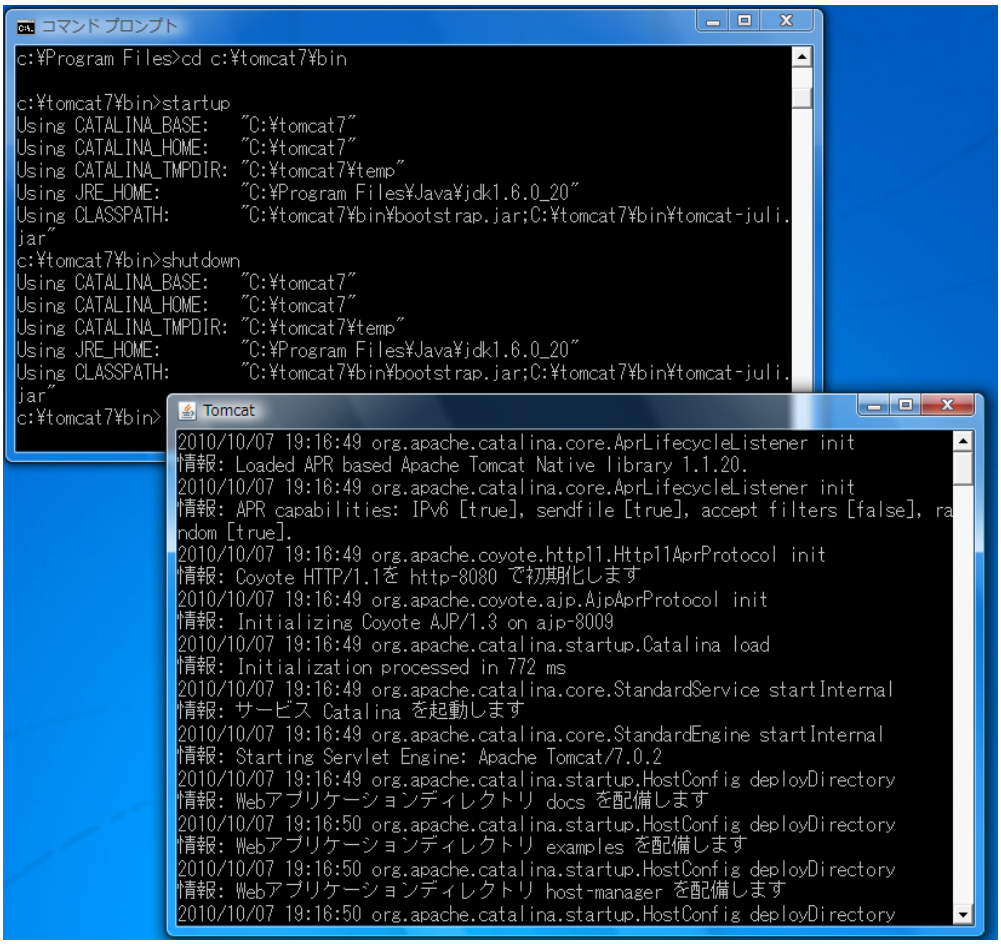


図 5-3 :コマンド・プロンプトからの Tomcat の起動と終了

7. 同梱されているサンプル・サーブレットと JSP を試してみる。
   TOMCAT_HOME\webapps\examples の下の\servlet と\jsp のフォルダにはサンプルのサーブレットと JSP がいろいろ揃えてあるので、各自試してみると良い。例えば:
   1. http://localhost:8080/examples/servlets/servlet/HelloWorldExample (Hello World と表示)
   2. http://localhost:8080/examples/servlets/servlet/RequestInfoExample (HTTP 要求の内容を表示)
   3. http://localhost:8080/examples/servlets/servlet/RequestHeaderExample (要求ヘッダを表示)
   4. http://localhost:8080/examples/servlets/servlet/CookieExample (クッキーの送信実験)
   5. http://localhost:8080/examples/servlets/servlet/SessionExample (セッションとその設定の実験)
   6. http://localhost:8080/examples/servletToJsp (サーブレットから JSP の呼び出し)

   どうしてこのような呼び出しでこれらのサーブレットが呼び出されるのであろうか？それは C:\tomcat7\webapps\examples\WEB-INF\web.xml という配備記述子にそう指定されているからである。配備記述子に関しては別途解説するが、これは examples というアプリケーションに関する各サーブレットや JSP のサーブレット定義及びサーブレット・マッピングの記述がそのような指定しているからである。

5.3節　tomcat7とtomcat7w

これらはTomcatのWindowsサービス化のためのプログラムである。Tomcatのアプリケーションは一般には開発はEclipseで行われ、配備はLinux機上でなされることが多いので、このチュートリアルではこれに関する詳細な説明は省略する。

関心のある読者はネット上の記事などを参考にされたい(apache-tomcat-7.0.xx.exeを使ってTomcatをインストールすると、最初からサービス登録される)。

WindowsサービスとしてTomcatを走らせることは、配備環境に於いて以下のような利点がある:

- コンピュータ起動時での信頼性がある自動開始: リモートでメンテ後に再起動させるときに、そのサーバが稼働するかを心配することなく再起動できる
- ログインなしでのサーバの起動: データ・センタにおいては、Tomcatを起動させる為だけにログインを要するのは妥当ではない。実際に、アクティブなモニタがつながっていないようなブレード・サーバ上でTomcatが走ることが多い。WindowsサービスはSystemが管理しており、アクティブなユーザなしで開始できる。
- セキュリティ: 管理者(administrator)としてTomcatを開始させると、ウェブ・アプリケーションのどれかにセキュリティの脆弱性があったときにサーバ全体が攻撃の対象になり得る。WindowsサービスとしてTomcatを稼働させるときは、特別のシステム・アカウントとのもとで稼働でき、それにより他のユーザ・アカウントと隔離しまた保護できる。

Tomcat7.exeはTomcatをWindowsサービスとして走らせる為の実際のサービス・ラッパ・プログラムであり、これがインストールされた後は完全にバックグラウンドで走行する。これはTomcatサービスの詳細な設定が出来るコマンド行パラメタを受け付ける。

Tomcat7wは簡単なグラフィカルなツールで、これがインストールされるとこのサービスの監視と設定が出来る。このプログラムはシステム・トレー上に表示されるので、アクセスが容易である。このツールを使うと、Tomcat7のコマンド行パラメタの多くが設定できる。またTomcatサービスの開始と停止をマニュアルで行うのにも使うことが出来る。

5.4節　Tomcat 7の設定

5.4.1　自動再ロード

TOMCAT_HOME/conf/context.xmlファイルを編集することで、自動再ロード(Auto reload)機能をオンにすることが出来る。自動再ロードはその都度Tomcatを再起動しなくても新しいクラス・ファイルがロードされたらこれを必要に応じコンテナにロードしてくれるので、学習や開発用には非常に便利な機能である。但しこの機能は取扱いに注意が必要であり、またオーバヘッドがかかるので、実際の運用時には使わないようにすべきである。「コンテキスト」の章で述べるように、自動再ロードを活かしたままだと、そのクラスをどこに置くかによっては**static なオブジェクトの static 性が保持されない**問題があるので、十分な注意が必要である。また、コンテキストに影響を与えるような変更があったら、サーブレット・コンテキストはこれに対応できない。

なお、**JSP の場合は、変更に対して Tomcat の再起動は必要ない**。また、次の章で示すように、**Eclipse からはサーブレットの自動再ロードが可能**になっている。

1. TOMCAT_HOME/conf/server.xml を秀丸等のテキスト・エディタで開き、終りのほうに次のような 2 行を追加して上書きする:

```
      <!-- Enable auto-reloading for context "tutorial" -->
      <Context docBase="tutorial" path="/tutorial" reloadable="true"/>
    </Host>
  </Engine>
 </Service>
</Server>
```

このようにすれば、TOMCAT_HOME\webapps\tutorial\WEB-INF\classes 内のクラス・ファイルがコンテナ上でインスタンス化されているときに、このファイルが更新されれば、自動的に更新されたファイルがインスタンス化される。

5.4.2 管理者ユーザ名とパスワードの設定

TOMCAT_HOME/conf/tomcat-users という XML ファイルのユーザ名とパスワードを以下のようにコメントを外して設定する(*****のところは自分のユーザ名とパスワード)

```
-->
<tomcat-users>
<user name="*****" password="*****" roles="admin,manager" />
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
```

5.5節 DOS レベルでの開発環境の確立

Eclipse などの IDE が普及する前は、開発者はテキスト・エディタでサーブレットを書き、JDK の javac コマンドでそれをコンパイルしていた。そのような開発法はもう使われることはないが、開発の基本を知る上で学習しておくことが好ましい。ここでは ROOT 上、すなわちデフォルトのアプリケーションとしてのサーブレットをこの開発法で実習する。ここではテキスト・エディタとしてよく使われているフリー・ソフトの秀丸を使用しているが、Windows のメモ帳のようなものでも構わない。

5.5.1 新しいアプリケーション(tutorial)をつくる

TOMCAT_HOME の下は次のようなディレクトリになっている:

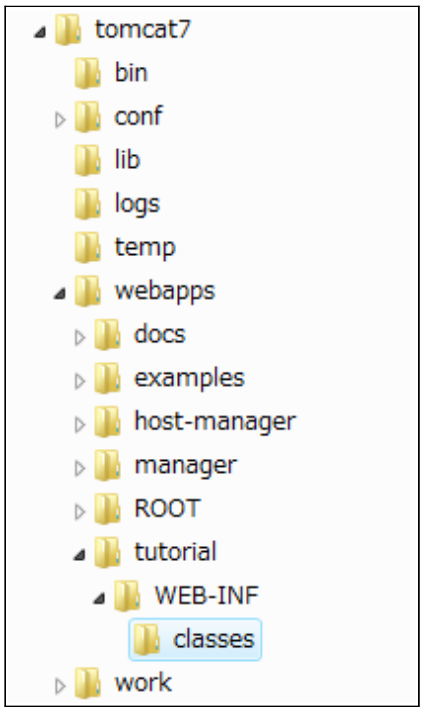


図 5-4：Tomcat のディレクトリ構造

アプリケーションの開発者からみて、Tomcat のディレクトリのなかで重要なものは下図のようである：

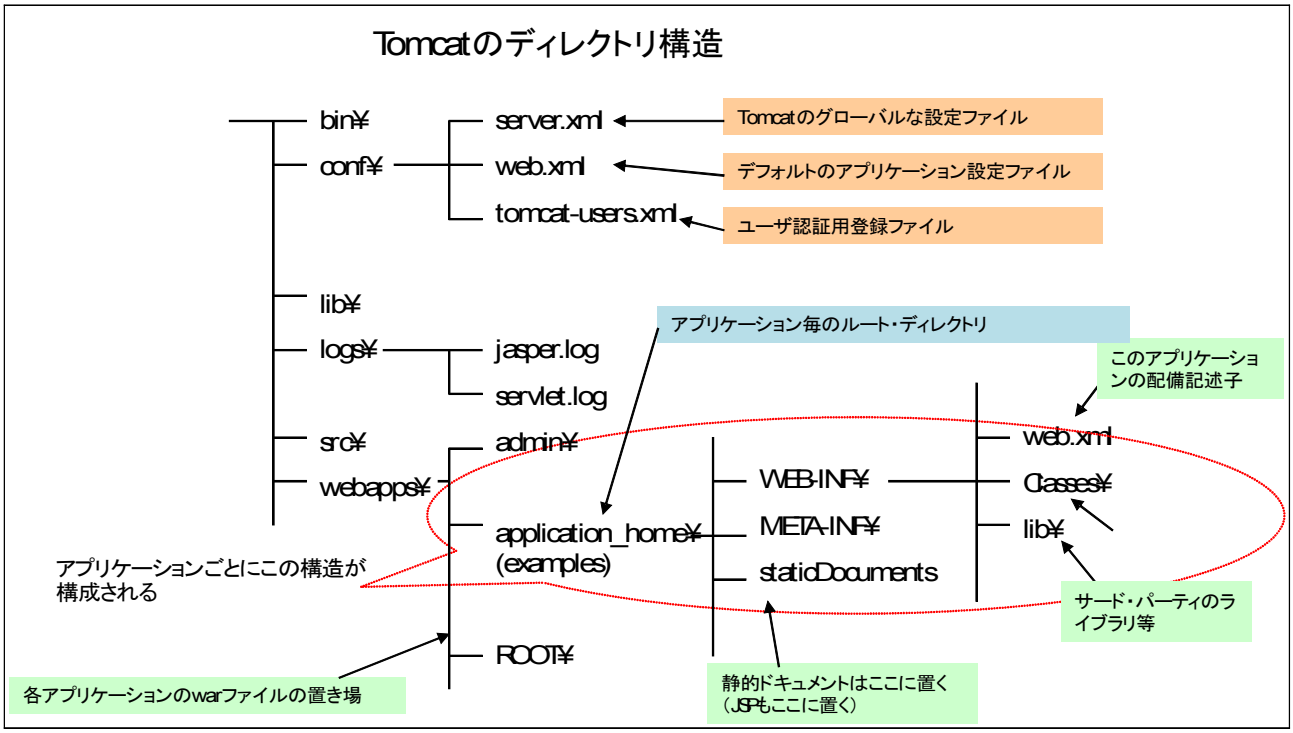


図 5-5：Tomcat のディレクトリ構造

各ディレクトリのより詳細な内容は次のようである：

表 5-2：Tomcat のディレクトリ

ディレクトリ	内容
bin	Tomcat の開始、停止、実行のための、実行 Jar ファイル、startup.bat、shutdown.bat、

	及び*.sh ファイル
conf	Tomcat 設定のための*.policy、*.properties、及び*.xml ファイル
lib	Tomcat 実行に必要な*.jar ファイル
logs	Tomcat のログたち
temp	一時作業領域
webapps	これが開発者が開発したウェブ・アプリケーションの置き場となる。各アプリケーションはその名前のフォルダを持つ(ここでは YourWebApp と表現する)。ウェブ・アプリケーションの標準ディレクトリは第 4 節の「標準ディレクトリ・レイアウト」に従うことになる。なお ROOT というディレクトリが存在するが、これは「デフォルト・アプリケーション」というひとつのアプリケーションであり、クライアントは簡単な URL でアクセスできるが、これは推奨されていない。
webapps/YourWebApp/	これがいわゆるドキュメント・ルートになる。この下はツリー構造になる。*.html、*.jsp、*.gif、*.png、*.jpg、*.svg など、そのアプリケーションにとって、クライアントのブラウザにとって可視でなければならない HTML と JSP、及びその他の**静的ドキュメント**のファイル(例えば JavaScript、スタイルシート、及びイメージ)たち。
webapps/YourWebApp/WEB-INF/web.xml	そのアプリケーションのウェブ・アプリケーション配備記述子。これはそのアプリケーションを構成するサーブレット及びその他のコンポネンツ、及び初期化パラメタたち、及び行使させたいセキュリティ制約などを記述した*.xml ファイルである。
webapps/YourWebApp/WEB-INF/classes	そのアプリケーションに必要な Java クラス・ファイル(及びそれに関わるリソース)が入り、*.jar でまとめられていないサーブレット及び非サーブレットのクラスが含まれる。これにはカスタムのタグを処理する為の Java ソースも含む。これらのコードがパッケージ化されているときは、/WEB-INF/classes/のなかでそれがサブディレクトリとして反映されていなければならない。例えば com.cresc.tutorial.MyServlet という名前の Java クラスは、/WEB-INF/classes/com/cresc/tutorial/MyServlet.class に保管されねばならない。
webapps/YourWebApp/WEB-INF/jsp	*.tld taglib 記述子ファイルたち
webapps/YourWebApp/WEB-INF/tags	*.tag ファイルたち
webapps/YourWebApp/WEB-INF/lib	そのアプリケーションにとって必要な Java クラス・ファイルたち(及びそれに関わるリソースたち)を含めた*.jar ファイルで、サード・パーティのクラス・ライブラリあるいは JDBC ドライバたちなど。
work	作業用で、Tomcat が*.jsp ファイルから生成した*.java と*.class ファイルが入る。

なお必須事項として:

- ウェブ・アプリケーションは WEB-INF(小文字は不可)というフォルダを持っていなければならない
- ウェブ・アプリケーションは WEB-INF フォルダ内に web.xml という配備記述子を持っていなければならない

ことに注意されたい。

そのようなことを念頭に置いて、本チュートリアルでは以下の手順で、"tutorial"というアプリケーションを用意し、そのなかでサーブレットの基本を確認してゆくことにしたい。

1. エクスプローラを使って前図のように、webapps の下に tutorial、更にその下に WEB-INF、更にその下に classes というフォルダを作る。

2. WEB-INF のなかに以下のような内容の web.xml ファイルを置く

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0"
  metadata-complete="false">
<!-- Note: metadata-complete should not set to true to utilze @WevServlet annotation -->

    <description>
      Cresc Servlet and JSP Tutorial.
    </description>
    <display-name>Cresc Servlet and JSP Tutorial</display-name>

    <!-- Define servlets that are included in the tutorial application -->
<!--
    <servlet>
      <servlet-name>HelloWorld</servlet-name>
      <servlet-class>HelloWorld</servlet-class>
    </servlet>

    <servlet-mapping>
       <servlet-name>HelloWorld</servlet-name>
       <url-pattern>/HelloWorld</url-pattern>
    </servlet-mapping>
-->

</web-app>
```

これは Tomcat に同梱されている examples というアプリケーションに含まれている web.xml から必要なものだけを抽出したものである。ここで servlet 及び servlet-mapping という要素は、現在はコメント・アウトしてあるが、将来この要素を使う場合の参考のために含めてある。重要な事項は metadata-complete="false"という指定で、examples では"true”だったものである。我々の教材ではその都度このファイルを変更しなくても良いように @WebServlet というアノテーションを使っているが、**metadata-complete="true"のままだと、アノテーションが無視され、Tomcat が 404 エラーを返してしまう。**

5.5.2 マニュアル・コンパイル用バッチファイルの作成

秀丸を使って以下の内容のテキストファイルを作り、これを jc.bat という名前で c:\tomcat7\bin のフォルダにおく:

```
cd %TOMCAT_HOME%\webapps\tutorial\WEB-INF\classes
javac %1.java
cd %TOMCAT_HOME%\bin
```

これは DOS のバッチ・ファイルで、TOMCAT_HOME\webapps\tutorial\WEB-INF\classes に置かれたサーブレット・クラスの Java ファイルをコンパイルして、同じディレクトリに置くものである。このバッチファイルの引数はそのサーブレットの名前である。

5.5.3 HelloWorld サーブレットの作成とコンパイル

秀丸等のテキスト・エディタを使って次のような HelloWorld というサーブレットのコードを作り、これを TOMCAT_HOME\webapps\tutorial\WEB-INF\classes に HelloWorld.java という名前で置く(Tomcat をダウンロー

ドしたままだと tutorial、WEB-INF、及び classes のフォルダが含まれていないので、このフォルダを追加のこと)。このコードをコピーしてテキスト・エディタに貼り付けるのが手っ取り早い。

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

@WebServlet("/HelloWorld")
public class HelloWorld extends HttpServlet
{
	private static final long serialVersionUID = 1L;
	public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException
	{
	res.setContentType("text/html; charset=Windows-31J");
	PrintWriter out = res.getWriter();
	out.println("<HTML>");
	out.println("<HEAD><TITLE>Hello World</TITLE></HEAD>");
	out.println("<BODY>");
	out.println("<BIG>Hello World from 自分の名前</BIG>");
	out.println("</BODY></HTML>");
	}
}
```

コマンド・プロンプトのプログラムで、c:\tomcat7\bin のディレクトリから jc HelloWorld を実行してみる。以下のように正しくコンパイルができることを確認する。誤りがあれがメッセージが出るので、指摘された箇所を訂正し、何もメッセージを出すことなくコンパイルが終了することを確認する

```
c:\>cd tomcat7\bin
c:\tomcat7\bin>jc HelloWorld
c:\tomcat7\bin>cd C:\tomcat7\webapps\tutorial\WEB-INF\classes
C:\tomcat7\webapps\ROOT\WEB-INF\classes>javac HelloWorld.java
C:\tomcat7\webapps\ROOT\WEB-INF\classes>cd C:\tomcat7\bin
C:\tomcat7\bin>
```

このサーブレットは文字列をブラウザに返すだけという最もシンプルなもののひとつである。このコードのポイントを幾つか説明すると:

- @WebServlet("/HelloWorld") ： これはそのコンテキスト・ルートに"/HelloWorld"を追加することでこのサーブレットが呼び出されるサーブレット・パスになることを指定するアノテーションであり、サーブレット第 3 版で新しく追加された機能である。これは第 3 版の簡単開発 (EOD：Ease of Development) という特徴のひとつである。長い配備記述子(Web.xml ファイル)にサーブレットの登録（名前とマッピング）をしなくても良いので便利なアノテーションである。**但し配備記述子で metadata-complete 属性が true になっていると無視される**ので注意のこと。
- serialVersionUID = 1L; ： シリアライズ可能クラスは long 型の static final serialVersionUID を必要とする為、明示的に指定しないと Eclipse などでは警告がでる。
- doGet(HttpServletRequest req, HttpServletResponse res) ： これは HttpServet クラスにある同じ名前のメソッドをオーバライドする。GET 要求に対するそのアプリケーションのサービスの実装。
- res.setContentType("text/html; charset=Windows-31J") ： HTTP 応答メッセージの content-type ヘッダを用意するとともに、コンテナに対し文字を Windows-31J に変換して応答メッセージを作ることを指示する。
- out = res.getWriter(); ： HTTP 応答メッセージのボディ部分にテキストを書き込むための Writer を取得する。サーブレットはここにクライアントのブラウザが表示する為の HTML テキストを書き込む。

5.5.4 HelloWorld サーブレットの実行

前記のように HelloWorld.java というサーブレットを書いて、コンパイルしたらこれをブラウザからアクセスして実行させる。

1. コマンド・プロンプトで c:\tomcat7.bin のディレクトリに移り、startup(CR)を入力して Tomcat を開始させる。

2. http://localhost:8080/tutorial/HelloWorld または http://127.0.0.1:8080/tutorial/HelloWorld で皆さんのサーブレットが呼び出せることを確認。また近くの人と互いのサーブレットが呼び出せることも確認する。（自分のコンピュータの IP アドレスは「スタート」>>「ファイル名を指定して実行」を開いて cmd と入力後「OK」を押しコマンド画面を開いて ipconfig /all(Enter)を入力して調べる）

図 5-6：マニュアルによる HelloWorld の実行

5.5.5 簡単な JSP の実験

ここでは StaticJspExample.jsp という JSP を書いて、実行させてみよう。

秀丸等で、以下のようなテキストの StaticJspExample.jsp というファイルを作り、TOMCAT_HOME\webapps\tutorial のフォルダに置く

```
<%@ page language="java" pageEncoding="Windows-31J" contentType="text/html;
charset=Windows-31J" %>
<HTML>
<HEAD><TITLE>Static JSP Sample</TITLE>
<META HTTP-EQUIV="cache-control" CONTENT="no-cache">
<META HTTP-EQUIV="content-type" CONTENT="text/html; charset=”Windows-31J"></HEAD>
<BODY><H1>静的 JSP のサンプル</H1>
Hello from 皆さんの名前
</BODY></HTML>
```

この JSP をブラウザから http://localhost:8080/tutorial/StaticJspExample.jsp または http://127.0.0.1:8080/tutorial/StaticJspExample.jsp をアクセスして、皆さんの JSP が下図のように呼び出せることを確認。また近くの人と互いの JSP が呼び出せることも確認する。エラーがでたら指摘された箇所を修正する。

図 5-7：JSP の実験

追加実験：

1. TOMCAT_HOME\webapps\ROOT\のディレクトリにこのファイルを置いたらどのように呼び出せばよいか？
2. Tomcat が生成したサーブレットや class ファイルは何処に存在するか？
3. このファイルを StaticJspExample.JSP と大文字の拡張子をつけてしまうと何が起きるか各自調べること
4. このファイルの名前を StaticJspExample.htm としたら何が起きるか？
5. 文字化け対策：<%@ page language="java" pageEncoding="Windows-31J" contentType="text/html; charset=Windows-31J" %>というページ・ディレクティブを削除すると何が起きるか（このディレクティブを挿入することを習慣づけることをお勧めする）？

回答：

1. http://localhost:8080/StaticJspExample.jsp で呼び出せる（デフォルト・アプリケーション）
2. C:\tomcat7\work\Catalina\localhost\tutorial\org\apache\jsp\StaticJspExample_jsp.java
3. エラーになる
4. http://localhost:8080/tutorial/StaticJspExample.htm で呼び出せる
5. 文字化けが起きる

5.5.6 自動再ロード機能の確認

この環境では、Tomcat を再起動をすることなく、サーブレットの変更を行うことが可能である。例えば秀丸などのエディタで HelloWorld.java の「石原伸晃」のところを「菅直人」と変更して、再コンパイルし、このサーブレットを呼び出したブラウザ上のアドレスの横の「更新」をクリックすれば、その変更が直ちに反映されていることが分かる。このように、自動再ロードの機能は開発段階では非常に有用ではあるが、あとで説明する static な要素が Webapps のクラス・ローダで新しく作られてしまうという問題があるので、注意しなければならない。

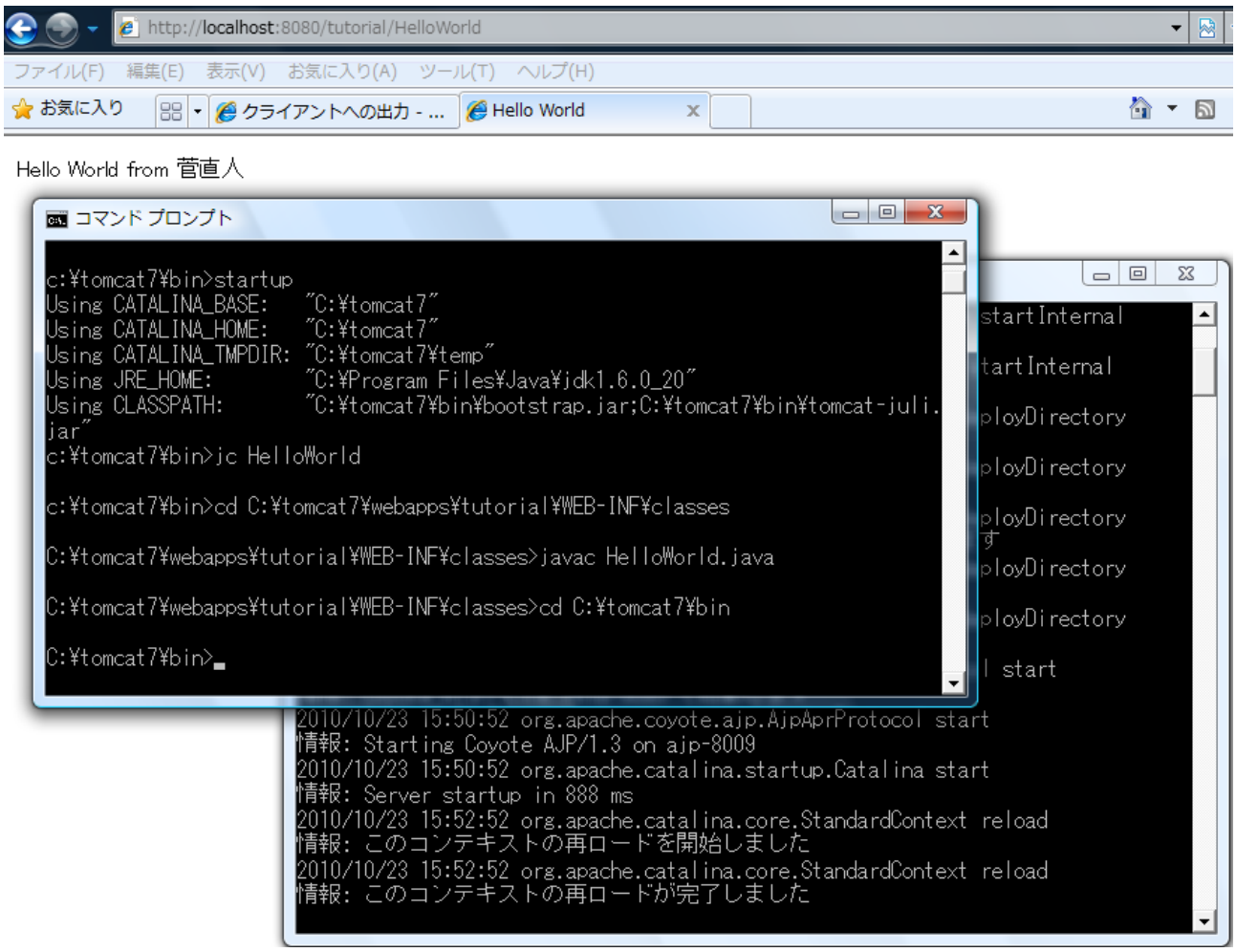


図 5-8：DOS レベルでの自動再ロードの確認

上図は以下のような過程を示したものである：

- コマンド プロンプト画面を見ると、最初に startup コマンドで Tomcat 7 を開始させている。その後、ブラウザから HelloWorld がアクセスされた後で、HelloWorld.java を変更し、jc コマンドでこれを再コンパイルしている。
- Tomcat の画面を見ると、startup 完了後、ブラウザから HelloWorld がアクセスされた後で、「このコンテキストの再ロードが開始しました」及び「このコンテキストの再ロードが完了しました」のメッセージが表示されている。
- その後ブラウザからこのサーブレットを再アクセスすると「Hello World from 菅直人」と内容が更新されている。

第6章　Eclipse で Tomcat 7 を使う

本章では、Eclipse を使ってサーブレットを開発し、Tomcat 7 でそれを実行させるまでの手順を示す。

その前に以下のことが終わっている必要がある:

1. Tomcat 7 を含め、総てのサーブレット 3 対応のサーブレット・コンテナは Java 6 が必要である。それ以前の Java のバージョンを使っている人は、バージョン・アップが必要である。このチュートリアルを学習している人は、第 3 章「開発環境(Eclipse)とそのインストール」で、既に Java EE6 SDK をインストールしているはずである。なお、JDK 1.6.0_21 及びそれ以降の Windows バージョンに限り、Eclipse でメモリ・オーバを起こしクラッシュするという Eclipse の「PermGen 領域エラー」問題を起こすバグがある。従って、これらのバージョンは使わない(即ちそれ以前のバージョンを使う)か、バグ・フィックス操作、即ち次のように、eclipse.ini の最後の行に-XX:MaxPermSize=256m という行を追加する必要がある。

2. Eclipse 3.6 (Helios)がインストールされていること。このチュートリアルを学習している人は、第 3 章「開発環境(Eclipse)とそのインストール」で、既にそのインストールと設定が終わっているはずである。それ以前のバージョンの Eclipse を使っている人は、Workspace を適当なところに保管して、Eclipse の最新版を再インストールする。

3. Tomcat 7 がインストールされていること。これは前章の「Tomcat 7 のインストールと設定」で詳しく説明した。少なくとも 5.2 節までに従って、インストールと設定がされている必要がある。

6.1節　新しいウェブ・アプリケーション(tutorial)を作る

Eclipse ではひとつのプロジェクトがウェブ・アプリケーションに対応する。従ってここではこのチュートリアルで使うサーブレットは tutorial というプロジェクトにまとめることにする。

1. Eclipse のパッケージ・エクスプローラ上で右クリック、「新規(New)」→「プロジェクト(Project...)」で「新規プロジェクト(New Project)」のウィザードを開く。

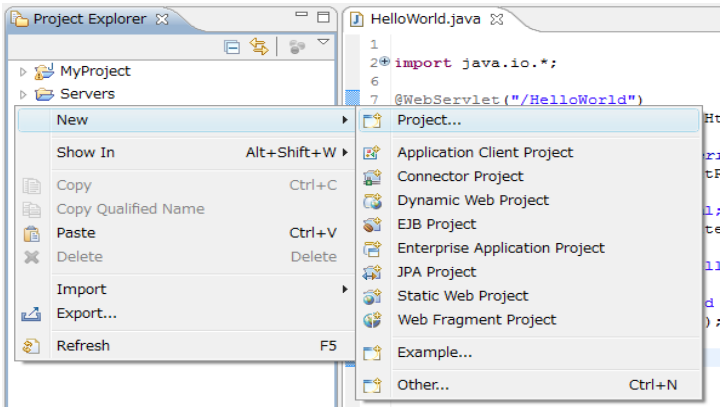


図 6-1:新規プロジェクト作成開始

2. ここでは「ダイナミック・ウェブ・プロジェクト(Dynamic Web Project)」を選択し「次へ(Next>)」をクリックする。

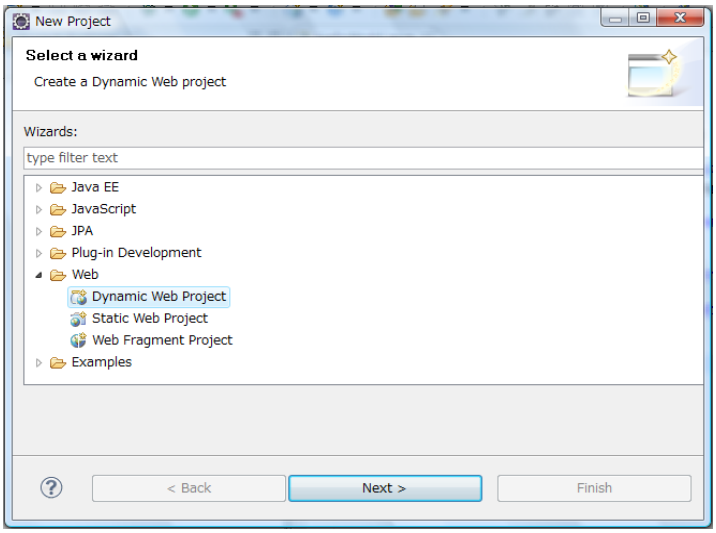


図 6-2：プロジェクトのタイプの指定

3. 「新規ダイナミック・ウェブ・プロジェクト(New Dynamic Web Project)」のペイン上で「ターゲット・ランタイム(Target runtime)」の個所で"Apache Tomcat v7.0"を選択、「プロジェクト名(Project name:)」には”tutorial”を入力する。その他はデフォルトのままとする。もしここで警告が出たら、3.3 節のとおり JRE ではなくて JDK が選択されているかを確認する。

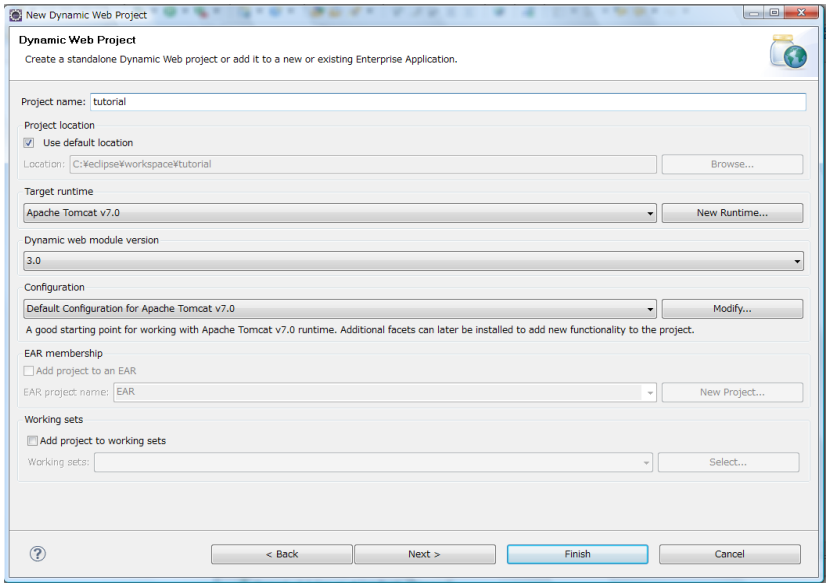


図 6-3：プロジェクト名の入力

4. 「終了(Finish)」をクリックすれば、パッケージ上に下図のように tutorial というプロジェクトが構成される。

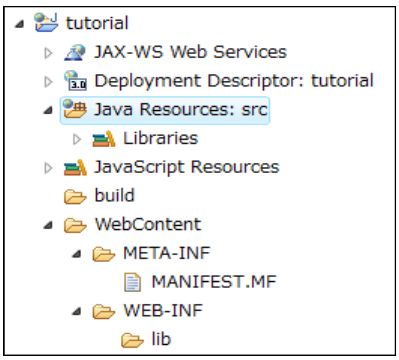


図 6-4：tutorial パッケージの構成

ここでこれらの構成要素のなかの収容なものを簡単に説明する：

- WebContent：この部分が実際の Tomcat の環境への配備対象になる。従ってここがこのアプリケーションのドキュメント・ルートとなり、「Tomcat のディレクトリ」の表に示したように、静的なドキュメント等が置かれる。
- WebContent/WEB-INF ：ここには web.xml という配備記述子が置かれることになるが、サーブレットの Java コードに@WebServlet が入っていればこのファイルは不要にできる。
- WebContent/WEB-INF/lib ：このアプリケーションで必要な外部ライブラリの Jar ファイルが入る
- Java Resoueces: src ：ここにはクラス、ビーンズ、サーブレットのソース・コードが入るが、通常それらが入ったパッケージが入る。パッケージ化しないデフォルト・パッケージはウェブ・アプリケーションでは使用しないことが強く推奨されている。これらのリソースがそのウェブ・プロジェクトに追加されると、それらは自動的にコンパイルされ、生成されたクラス・ファイルは WEB-INF/classes ディレクトリに追加される。そのディレクトリのソース・コードは、特に指定しな限り WAR ファイルには含められない。
- WebContent/WEB-INF/classes ：ここにはコンパイラは生成したクラス・ファイルが入るので、直接*.class ファイルを追加してはならない（コンパイラが削除する）。

6.2節　　新しいサーブレットを作成する

tutorial というプロジェクトが出来たので、今後はこのプロジェクトで幾つかのサーブレットを作りながら、サーブレットの基礎を学習することにする。

6.2.1　ソース・コードの為のパッケージ(basic_package)を用意する

1. 下図のようにパッケージ・ナビゲータ上の”Java Resources: src”を右クリック、「新規(new)」→「プロジェクト(Project...)」を選択する：

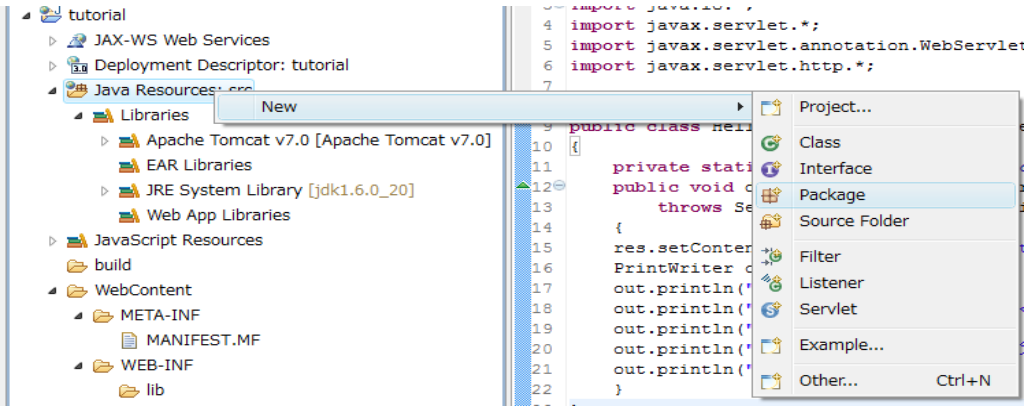


図 6-5：新規パッケージ作成開始

2. 「新規 Java パッケージ」のペインで「名前(Name:)」にパッケージ名”basic_package”を入力し、「完了(Finish)」をクリックする。そうすると”Java Resources: src”のフォルダの下にこの名前のフォルダが生成される。

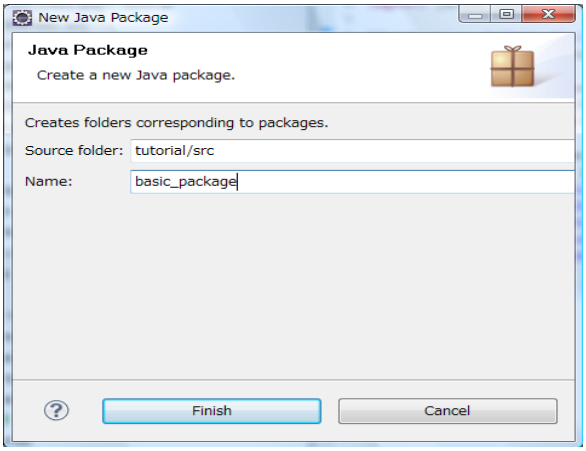


図 6-6：パッケージ名の入力

6.2.2 新規サーブレット(HelloWorld)を作成する

1. パッケージ・ナビゲータ上で新しく作った basic_package を右クリック、「新規(new)」→「クラス(Class)」を選択する：

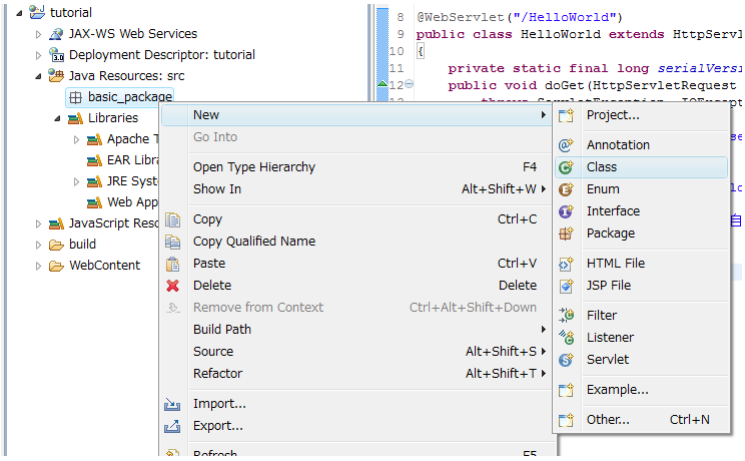


図 6-7：HelloWorld サーブレット作成開始

2. 「新規 Java クラス(New Java Class)」のペインで、サーブレット名とスーパー・クラスを入力する。スーパー・クラスの入力には ctrl＋スペースで候補をリストアップさせ、選択できる。

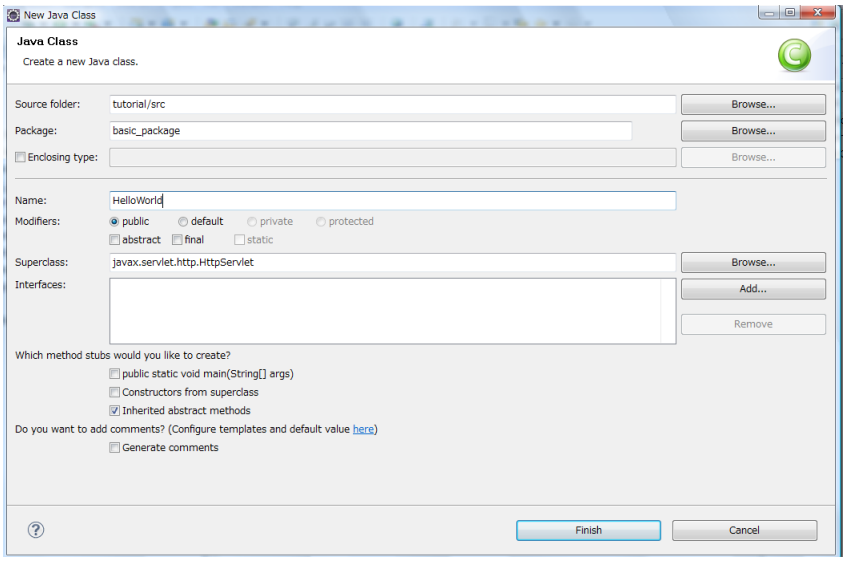


図 6-8：新規サーブレット名とスーパー・クラスの入力

そうすると、下図のように HelloWorld というサーブレットが basic_package の下に作られ、またそのテキストがエディタに表示される：

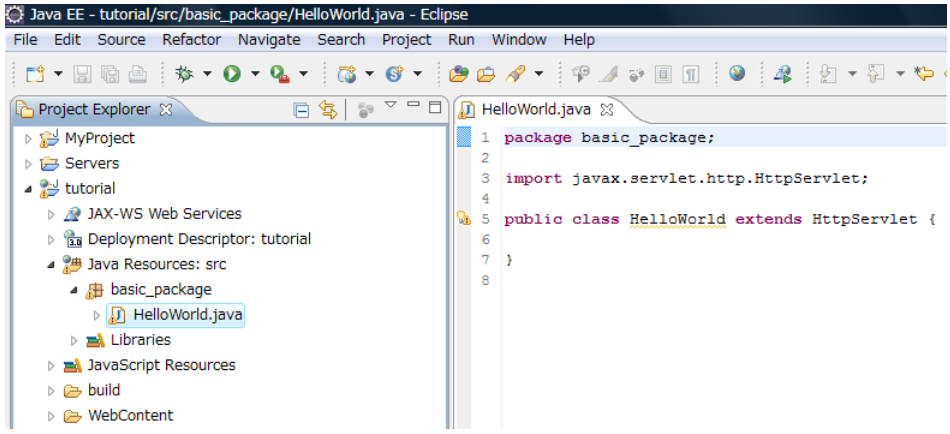


図 6-8：作成された HelloWorld.java とエディタ表示

6.2.3 ソース・コードの入力

1. 上図のエディタ上で、以下のようにテキストを入力する。20 行目の「自分の名前」の部分は各自の名前を漢字で置き換える(コピー/ペーストを使えば手っ取り早い)。これは前章の「DOS レベルでの開発環境の確立」の節の「HelloWorld サーブレットの作成とコンパイル」の項で示したコードとおなじである。

```
package basic_package;

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

```
@WebServlet("/HelloWorld")
public class HelloWorld extends HttpServlet
{
	private static final long serialVersionUID = 1L;
	public void doGet(HttpServletRequest req, HttpServletResponse res)
		throws ServletException, IOException
	{
	res.setContentType("text/html; charset=Windows-31J");
	PrintWriter out = res.getWriter();
	out.println("<HTML>");
	out.println("<HEAD><TITLE>Hello World</TITLE></HEAD>");
	out.println("<BODY>");
	out.println("<BIG>Hello World from 自分の名前</BIG>");
	out.println("</BODY></HTML>");
	}
}
```

2. エラーなく入力が終了したら、エディタが面を右クリックして「保管(Save)」をクリックする。そうすると下図のように、「配備記述子(Deployment Descriptor: tutorial)」/「サーブレット・マッピング(Servlet Mappings)」に
/HelloWorld -> HelloWorld
というマッピングが表示される。これはコンパイラが@WebServlet("/HelloWorld")というアノテーションを解釈して、自動生成したものである。

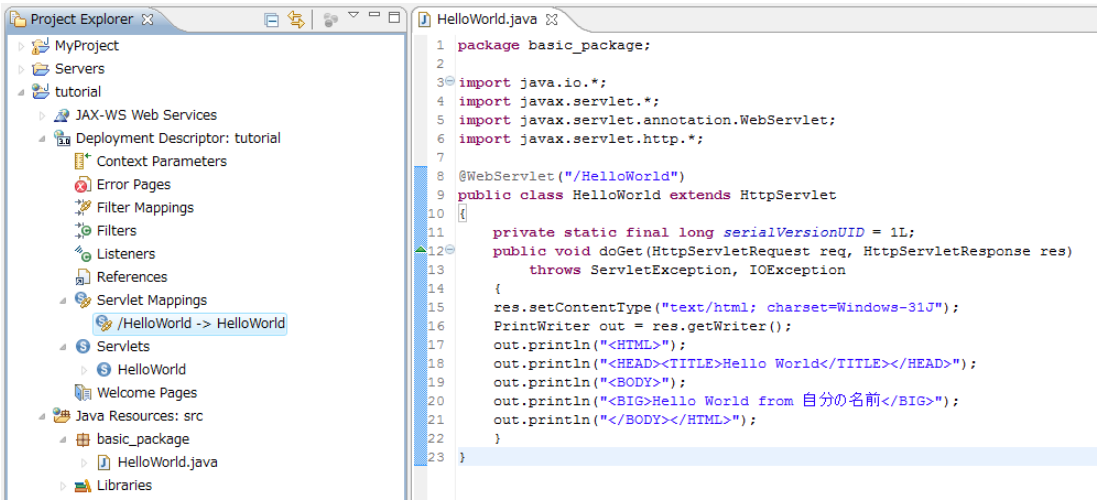


図 6-10：HelloWorld サーブレットの作成完了

6.3節　Tomcat 7 の開始と停止

Eclipse では簡単に Tomcat を動作させ、開発したウェブ・アプリケーションをテストできる。

6.3.1　Eclipse に Tomcat 7 のことを知らせる

1. 「ウィンドウ(Window)」→「表示ビュー(Show View)」→「サーバ(Servers)」で右下のビューに「サーバ(Servers)」のタブを追加する。

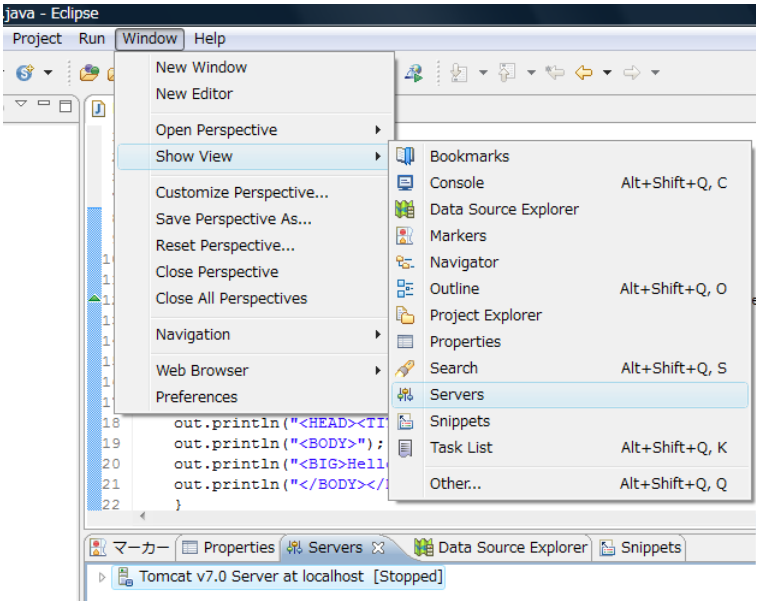


図 6-11：Servers のタブを追加

2. 「サーバ(Servers)」のタブをクリックしたペイン上で右クリック→「新規(New)」→「サーバ(Server)」→”Apache”→”Tomcat v7.0”→「完了(finish)」で、Tomcat 7 を追加する。

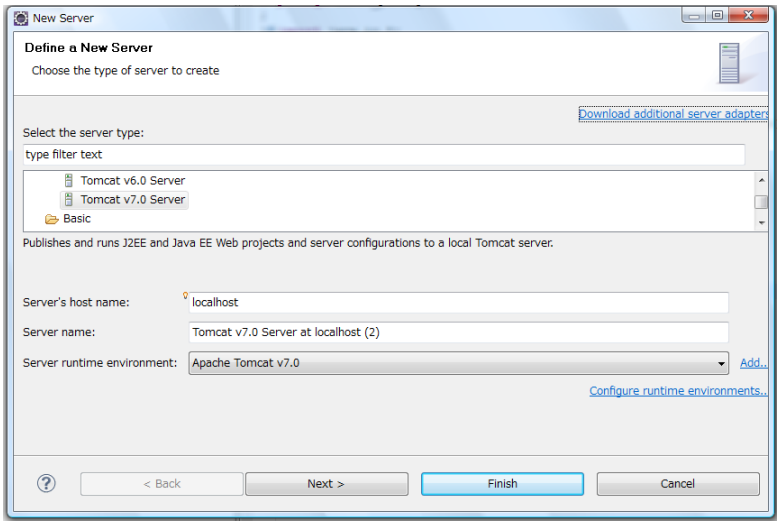


図 6-12：Tomcat 7 をサーバに追加

6.3.2　Tomcat 7 の開始と停止

これは簡単で、”Tomcat v7.0 Server at localhost”の個所を右クリックすると、停止中であれば「開始(Start)」動作中であれば「停止(Stop)」が選択できるので、それをクリックすれば良い。あるいはそのビューの右上に実行あるいは停止のアイコンが表示されるので、それをクリックしても良い。

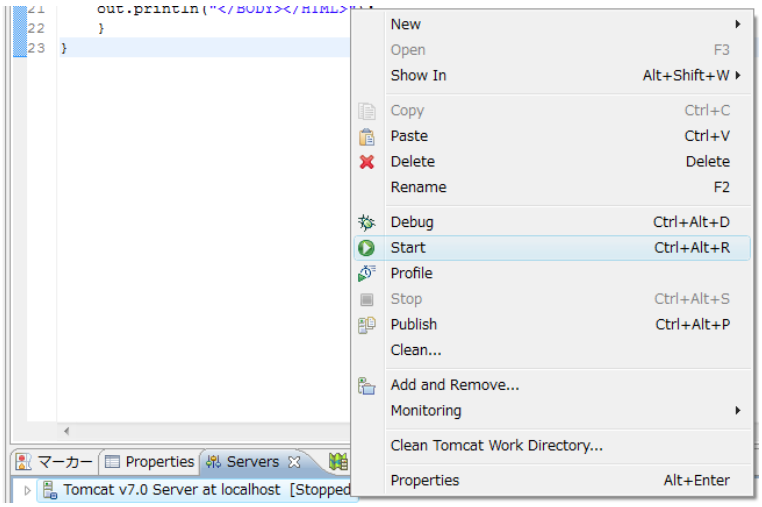


図 6-13：Tomcat 7 の開始と停止

なお、Eclipse 上で複数あるプロジェクト（アプリケーション）のなかからそのアプリケーションを Tomcat 7 で実行したいときには、プロジェクト・エクスプローラ上でそのプロジェクトを右クリック→「実行(R)」→Run on Server で Tomcat が起動する。この場合は、自分のブラウザからもアクセスできるが、エディット画面には Eclipse が持っているブラウザが起動するので、そこからもそのアプリケーションをアクセスできる。

もうひとつの実行プロジェクトの選択法は、”Tomcat v7.0 Server at localhost”の個所を右クリックして、「Add and Remove...」から起動されたいアプリケーションを Available から Configured に移すことである。

6.3.3　サーブレットのテスト

Tomcat が開始すれば、作成したサーブレットが使える。なお、第 5 章での実習で、DOS レベルで同じ tutorial という名前のアプリケーションが作られている場合は、そのアプリケーションの名前（即ち C:\tomcat7\webapps\tutorial というフォルダの名前）を変更しておいたほうが良い。

1. 自分の PC のブラウザから http://localhost:8080/tutorial/HelloWorld をアドレスに入力してアクセスする。下図は Internet Explorer でアクセスした例である：

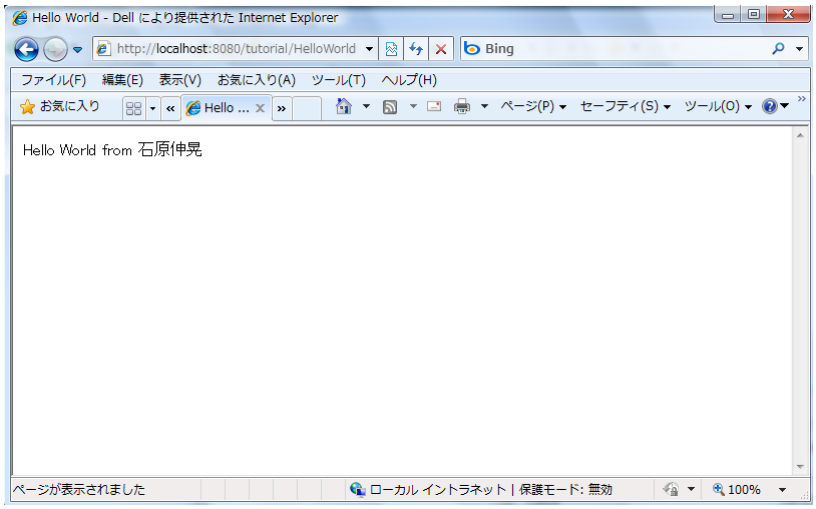


図 6-14：Internet Explorer からのアクセス

2. **Eclipse にも内部ブラウザ機能が付いている**ので外部のブラウザを使わなくても確認が出来る。即ちパッ

ケージ・エクスプローラ上で HelloWorld.java を右クリック→「実行(Run As)」→「サーバ上で実行(Run as Server)」を選択→サーバ上で実行(Run on Server)のペイン上で「完了(Finish)」をクリックすれば、下図のようにエディット画面上に実行結果が表示される。

なお、Eclipse ではテストするブラウザを選択できる。「ウィンドウ(Window)」→「ウェブ・ブラウザ(Web Browser)」で内部以外にも自分の PC がインストールしているブラウザを選択できる。

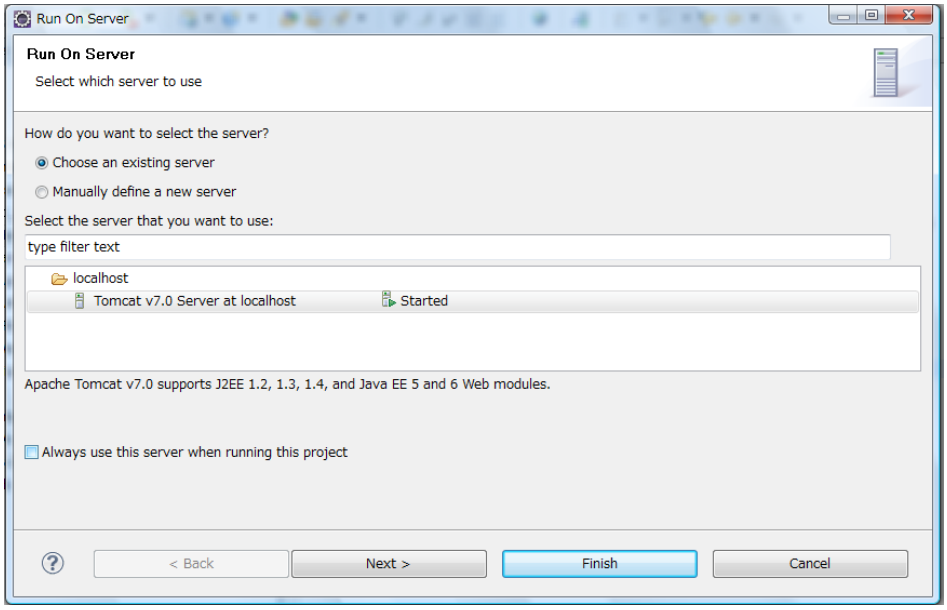


図 6-15：Eclipse 上でのサーブレットの実行開始

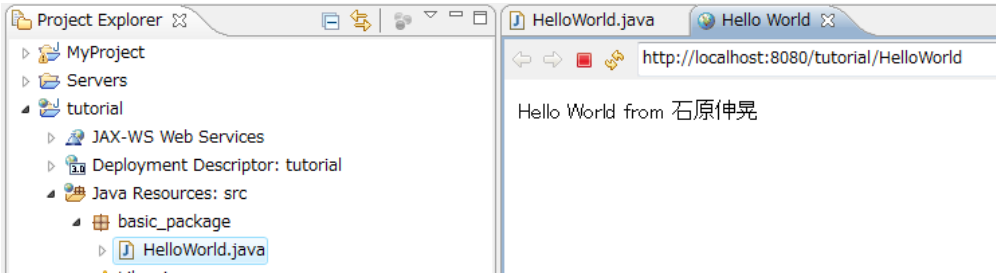


図 6-16：内部ブラウザによるサーブレットの実行結果

6.3.4　自動再ロード機能の確認

「Tomcat 7 の設定」の節の「自動再ロード」の項で示したと同じく、この環境でも Tomcat を再起動をすることなく、サーブレットの変更を行うことが可能である。例えば編集画面上で「石原伸晃」のところを「菅直人」と変更して「保管(Save)」をし、このサーブレットを呼び出したブラウザ上のアドレスの横の「更新」をクリックすれば、その変更が直ちに反映されていることが分かる。このように、自動再ロードの機能は**開発段階では非常に有用**ではあるが、あとで説明する **static な要素が Webapps のクラス・ローダで新しく作られてしまう**という問題があるので、注意しなければならない。また、**コンテキストに影響を与えるような変更があったときは、再起動が必要**になる。

6.4節　　ロギング・ファサードのインストール

今回のチュートリアルでは、Apache Foundation が標準的に使っている log4j というロガーを含むいろんなロガーに対応できるファサードの SLF4J が使えるような環境を用意する。

ロギングは自分が開発したアプリケーションのいろんな段階で重要なツールとなる:

- 開発段階でのデバッギング
- 生産中でのヘルプによるバグ診断
- セキュリティの目的でのアクセスのトレース
- 統計用のデータ作成

等々

本チュートリアルではロギングに関しては詳細な説明はしないが、読者が実際の開発業務を担当するときにも、使われているいろんなロギング・システムに対応できるようにファサードを使い、実用時にもそのままこのチュートリアルで学習したソフトウエアが活用できるようにする。

なお、javax.servlet.GenericServlet 及び ServletContext には以下のようなログのメッセージがある:

- public void log(java.lang.String msg)
  メッセージ文字列を出力する
- public void log(java.lang.String message, java.lang.Throwable t)
  与えられた例外のスタック・トレースとメッセージ文字列を出力する

なお、出力ログのファイルの名前とタイプはコンテナに依存する。必要ならこれらも利用されたい。これを継承している HttpServlet でもこれらのメソッドが使える。

6.4.1　SLF4J とは

ロギング・システムは Java では java.util.logging が一般的なロギング・システムであるが、Apache Foundation では、log4j というロギング・システムを標準的に採用している。log4j は Struts でも使われているので、なじみの人も多いだろう。ここでは開発者たちが自分のロギング・システムを使い易いように、Simple Logging Facade for Java (SLF4J)というファサードを採用する。ファサードは、デザイン・パタンでなじみのもので、その名の通り窓口的なユーティリティで、このファサードにログを書き込むことでいろんなロギング・システムに対応できる。Java では java.util.logging が一般的なロギング・システムであるが、SLF4J では log4j という Apache では一般的なパッケージが用意されており、開発の途中で java.util.logging から log4j にソース・コードを全く変更しないで移行できる点が魅力的である。

下図はこのパッケージの構成である。自分のシステムにあわせてバインドのための jar をライブラリに含めれば良い。

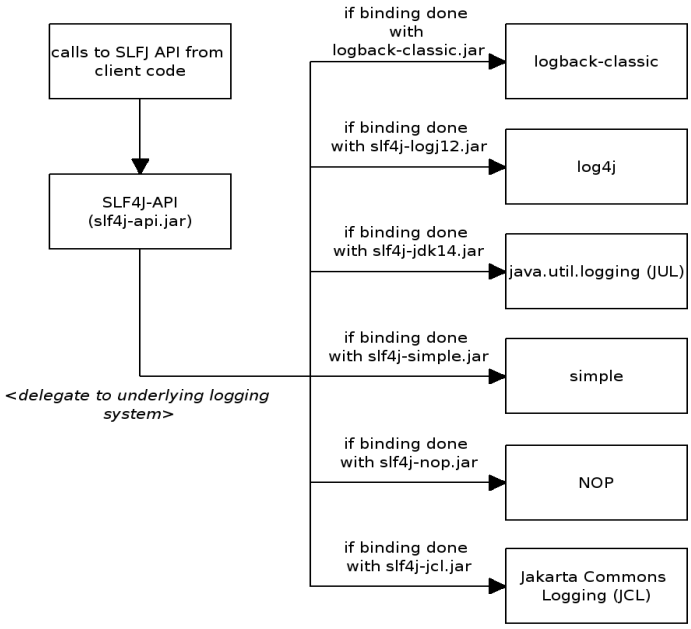


図 6-17：ロギング・システムのバインド

SLF4J バインドとしては以下のものが用意されている：

- slf4j-log4j12-1.6.1.jar
  広く利用されているロギング・フレームワークである log4j version 1.2 用のバインド。この場合は自分のクラス・パス上に log4j.jar も置かねばならないことに注意
- slf4j-jdk14-1.6.1.jar
  JDK 1.4 ロギングとも呼ばれている java.util.logging 用のバインド
- slf4j-nop-1.6.1.jar
  総てのロギングを無視する NOP 用のバインド
- slf4j-simple-1.6.1.jar
  Simple 実装のためのバインドで、総てのイベントを System.err に送り出す。INFO レベル以上の総てのメッセージが印刷される。このバインドは小規模のアプリケーションには有用かもしれない。
- slf4j-jcl-1.6.1.jar
- Jakarta Commons Logging 向けのバインド。このバインディングでは SLF4J ロギングの総てを JCL に委譲する

6.4.2 SLF4J のパッケージのダウンロードとインストール

1. SLF4J のダウンロードのサイトから適当な圧縮形式（例えば ZIP の slf4j-1.6.1.zip）のファイルを適当なディレクトリにダウンロードする。
2. 次に展開したもののメインのフォルダ（例えば SLF4J-1.6.1）を C:SLF4J と C ディレクトリに移す。
3. ウェブ・アプリケーション・ライブラリ(Web App Library)」を追加するには、WebContent\WEB-INF\lib にライブラリを追加したほうが手っ取り早い。Eclipse は自動的にそれを Web App Library としてライブラリに追加する。
   1. パッケージ・エクスプローラ上の tutorial\WebContent\WEB-INF\lib を右クリックし、「インポート(import...)」をクリック
   2. インポートのペインで「一般(General)」→「ファイル・システム(File System)」を選択し「次へ(Next)」ボタンをクリック
   3. のライブラリにダウンロードした jar ファイルたち、即ち c:slf4j の中の以下の jar を選択してインポートする：

slf4j-api-1.6.1
slf4j-simple-1.6.1

4. こうすると下図のように自動的に tutorial\src\Libraries\Web App Libraries のなかにこれらの jar が登録される:

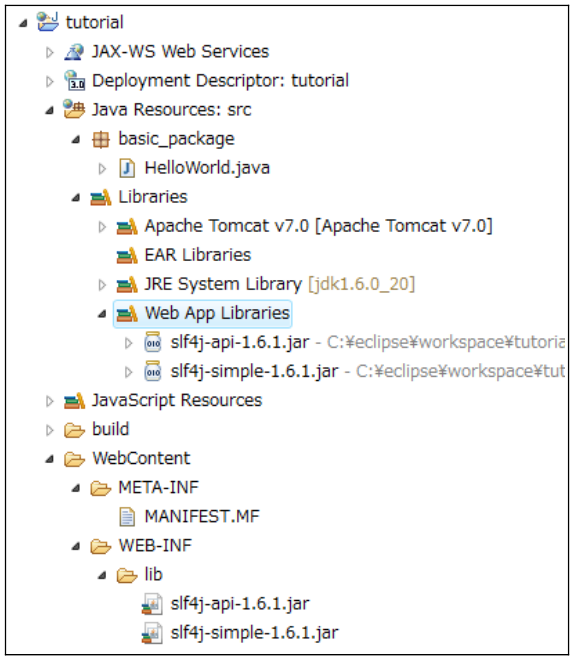


図 6-18:SLF4J ライブラリの追加

6.4.3 動作確認

HelloWord.java のコードに次のようにログ出力のためのコードを追加する:

```
package basic_package;

import java.io.*;
import java.util.Calendar;
import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


@WebServlet("/HelloWorld")
public class HelloWorld extends HttpServlet
{
	private static final long serialVersionUID = 1L;
	private Logger log = LoggerFactory.getLogger(HelloWorld.class);

	public void init(){
	log.info("HelloWorld instanciated at: " +
Calendar.getInstance().getTime().toString());
	}

	public void doGet(HttpServletRequest req, HttpServletResponse res)
		throws ServletException, IOException
	{
	log.info("Request from " +req.getRemoteAddr()+ " accepted");
	res.setContentType("text/html; charset=Windows-31J");
	PrintWriter out = res.getWriter();
	out.println("<HTML>");
```

```
        out.println("<HEAD><TITLE>Hello World</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("<BIG>Hello World from 石原伸晃</BIG>");
        out.println("</BODY></HTML>");
        }
}
```

1. 最初に、このサーブレットのための log という名前の Logger オブジェクトを生成している
2. オーバライドされた init メソッドのなかで、このサーブレットがインスタンス化された時刻をログ出力している(これはサーブレットのライフサイクルの解説のところで使用する)
3. オーバライドされた doGet メソッドでは、クライアントからの要求が到来したらその時の時刻をログ出力している

このサーブレットをブラウザからアクセスすると、次のようなログがコンソールに出力される:

```
0 [http-8080-exec-3] INFO basic_package.HelloWorld - HelloWorld instanciated at: Fri Oct 15
13:34:19 JST 2010
0 [http-8080-exec-3] INFO basic_package.HelloWorld - Request from 0:0:0:0:0:0:0:1 accepted
```

Tomcat がたちあがっても、このサーブレットはインスタンス化されず、クライアントからの最初の要求到来でインスタンス化されていることに注意されたい。

第7章　要求オブジェクト

サーブレット・コンテナは、クライアントからのHTTP 要求をHttpServletRequest 型のオブジェクトにカプセル化してサーブレットに渡している。従ってこのオブジェクトはHTTP 要求そのものではない。この章では、この要求オブジェクトはソフトウエア開発者にどのような情報及びメカニズムを提供しているかを、主としてHTTP 要求メッセージとの対応を中心にして説明する。それ以外のこのオブジェクトが持っている重要な機能

- *セッション管理*
- *情報の共有*
- *フィルタ*
- *ログイン機能*
- *非同期処理等*

は、別途章を分けて説明する。

7.1節　　ブラウザが出す HTTP 要求

最初に、ブラウザがHelloWorld というサーブレットをアクセスしたとき、どのようなHTTP メッセージが交換されるのだろうか？あとでHttpRequestDump というサーブレットで要求オブジェクトの内容を知る方法をお教えするが、まずTCP レベルでどのようなHTTP メッセージ交換がされるのかを、プロキシのプログラムを使って調べることとする。

7.1.1　MINA プロキシのプログラムで HTTP 要求メッセージを調べる

以下は当社の TCP/IP プログラミングのチュートリアルの 6.6 節「プロキシのためのフィルタ(ProxyFilter)」でも紹介した MINA プロキシをダウンロードし、引数”12345 localhost 8080”ではしらせ、Tomcat を開始させ、Internet Explorer から http://localhost:12345/tutorial/HelloWorld をアクセスしたときの結果である。MINA プロキシのインストールと使い方は参考資料「MINA プロキシのダウンロードと実行」の節を見られたい：

要求メッセージ：

```
GET /tutorial/HelloWorld HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, application/x-ms-application,
application/vnd.ms-xpsdocument, application/xaml+xml, application/x-ms-xbap,
application/x-shockwave-flash, application/vnd.ms-excel, application/msword,
application/vnd.ms-powerpoint, */*
Accept-Language: ja
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; SLCC1; .NET
CLR 2.0.50727; Media Center PC 5.0; MDDC; .NET CLR 3.5.30729; .NET CLR 3.0.30729;
OfficeLiveConnector.1.5; OfficeLivePatch.1.3; .NET4.0C)
Accept-Encoding: gzip, deflate
Host: localhost:12345
Connection: Keep-Alive
```

応答メッセージ：

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
```

```
Content-Type: text/html;charset=Windows-31J
Content-Length: 111
Date: Fri, 15 Oct 2010 06:12:45 GMT

<HTML>
<HEAD><TITLE>Hello World</TITLE></HEAD>
<BODY>
<BIG>Hello World from ???´?L?W</BIG>
</BODY></HTML>
```

MINA プロキシのプログラムに興味がある人は、下図のように上記チュートリアルを参考にして、Eclipse 上でインポートして走らせていただきたい。

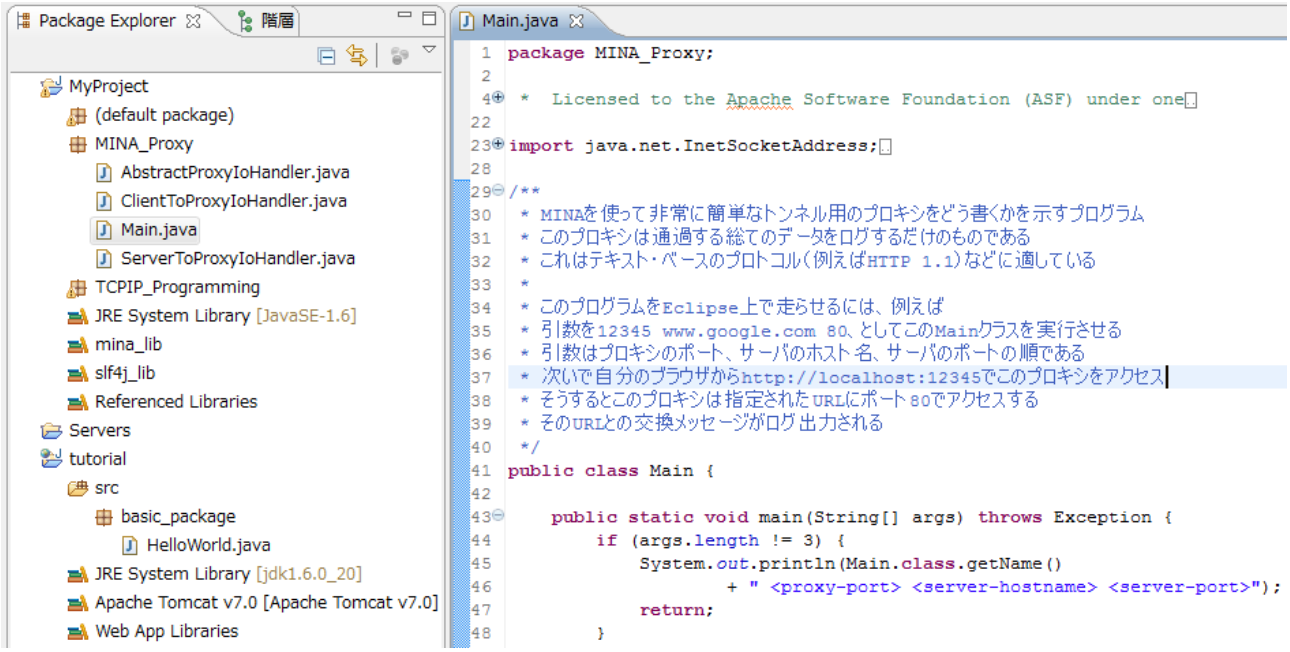


図 7-1：MINA_Proxy の組込み

Internet Explorer のブラウザと Tomcat のコンテナが送信したこれらの HTTP メッセージの意味は、第 2 章「HTTP」の説明を読めば理解されよう。

要求メッセージ:

001 GET /tutorial/HelloWorld HTTP/1.1

HTTP メソッドは GET、要求 URI は/tutorial/HelloWorld、HTTP バージョンは HTTP/1.1

002 Accept: image/gif, image/jpeg, image/pjpeg, application/x-ms-application, application/vnd.ms-xpsdocument, application/xaml+xml, application/x-ms-xbap, application/x-shockwave-flash, application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, */*

このブラウザはこのような形式のリソースを受け付ける。特に多くの Microsoft 社のドキュメント形式を受け付けている

003 Accept-Language: ja

このブラウザは日本語を受け付ける

004 User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; SLCC1; .NET CLR 2.0.50727; Media Center PC 5.0; MDDC; .NET CLR 3.5.30729; .NET CLR 3.0.30729; OfficeLiveConnector.1.5; OfficeLivePatch.1.3; .NET4.0C)

このブラウザのユーザ・エージェントは Mozilla/4.0 である。Mozilla は NetScape 社のブラウザを意味したが、Microsoft がこれを偽装したことから Mozilla/x.x が始まっている

005 Accept-Encoding: gzip, deflate

GZIP とデフレート圧縮したボディを受け付ける

006 Host: localhost:12345

ホスト指定、HTTP/1.1 では必須なヘッダ（ここではプロキシが host になっている）

007 Connection: Keep-Alive

TCP 接続を切らないことを伝えるヘッダ

008

空白行はヘッダ部の終わりを意味する

応答メッセージ：

001 HTTP/1.1 200 OK

HTTP バージョンは HTTP/1.1、ステータス・コードは 200、その意味は OK

002 Server: Apache-Coyote/1.1

サーバは Apache-Coyote/1.1 (Tomcat の HTTP コネクタ)

003 Content-Type: text/html;charset=Windows-31J

ボディ部は text/html で文字エンコーディングは Windows-31J（サーブレットが res.setContentType メソッドで指定した）

004 Content-Length: 111

ボディ部分の長さは 111 バイト（CR と LF を含む）

005 Date: Fri, 15 Oct 2010 06:12:45 GMT

日付ヘッダ

006

空白行はヘッダ部分の終わりを意味する

007 <HTML>

008 <HEAD><TITLE>Hello World</TITLE></HEAD>

009 <BODY>

010 <BIG>Hello World from ???´?L?W</BIG>

011 </BODY></HTML>

これらの行はボディ部の HTML テキストで、サーブレットが作成した

7.1.2 ブラウザによる相違

2.2 節で示したように、PC 用のブラウザの世界シェアは、Internet Explorer、Firefox、Chrome、及び Safari が占めている。これらのブラウザが出す HTTP 要求はどのような相違があるのだろうか？Internet Explorer（バージョン 8.0）は既に説明したが、このブラウザを下図のように「HTTP 1.1 を使用する」及び「プロキシ接続で HTTP 1.1 を使用する」のふたつのチェックを外してみると、次のような HTTP 要求メッセージを出力する：

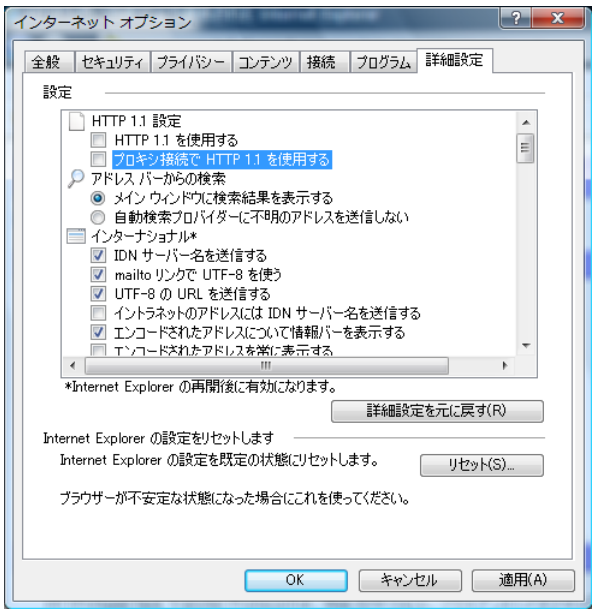


図 7-2：Internet Explorer を HTTP/1.0 のモードにする

Internet Explorer（バージョン 8.0）を HTTP/1.0 にしたときの HTTP 要求メッセージ

```
GET /tutorial/HelloWorld HTTP/1.0
Accept: image/gif, image/jpeg, image/pjpeg, application/x-ms-application,
application/vnd.ms-xpsdocument, application/xaml+xml, application/x-ms-xbap,
application/x-shockwave-flash, application/vnd.ms-excel, application/msword,
application/vnd.ms-powerpoint, */*
Accept-Language: ja
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; SLCC1; .NET
CLR 2.0.50727; Media Center PC 5.0; MDDC; .NET CLR 3.5.30729; .NET CLR 3.0.30729;
OfficeLiveConnector.1.5; OfficeLivePatch.1.3; .NET4.0C)
Host: localhost:12345
Connection: Keep-Alive
```

ここで注目すべきことは、HTTP/1.0 で要求していながら Host: localhost:12345 及び Connection: Keep-Alive というヘッダがつけられていることである。

以下 Firefox、Chrome、及び Safari が出す HTTP 要求メッセージを紹介すると以下のようになる：

Mozilla Firefox 3.6 の HTTP 要求メッセージ

```
GET /tutorial/HelloWorld HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja; rv:1.9.2.3) Gecko/20100401
Firefox/3.6.3 ( .NET CLR 3.5.30729; .NET4.0C)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
```

Google Chrome 6.0 の HTTP 要求メッセージ

```
GET /tutorial/HelloWorld HTTP/1.1
Host: localhost:12345
Connection: keep-alive
Accept:
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0
```

```
.5
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US) AppleWebKit/534.3 (KHTML,
like Gecko) Chrome/6.0.472.63 Safari/534.3
Accept-Encoding: gzip,deflate,sdch
Accept-Language: ja,en-US;q=0.8,en;q=0.6
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.3
```

Apple Safari 5.0 の HTTP 要求メッセージ

```
GET /tutorial/HelloWorld HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja-JP) AppleWebKit/533.18.1 (KHTML,
like Gecko) Version/5.0.2 Safari/533.18.5
Accept:
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0
.5
Accept-Language: ja-JP
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

これらの 3 つのブラウザは同じ Mozilla/5.0 を採用している。また Chrome と Safari はかなり類似していることが分かる。また総てのブラウザが Accept-Encoding: gzip, deflate の圧縮を受け付けるが、**Internet Explorer を HTTP/1.0 にしたときは圧縮転送を受け付けない**ことに注意のこと。

7.2節　HTTP 要求メッセージのデータ取得のためのメソッドたち

本節では HttpServletRequest インターフェイスのオブジェクトが持っている HTTP 要求メッセージに関するデータ取得の為のメソッドたちを中心に説明する。HttpServletRequest 及びそのスーパー・クラスである ServletRequest の持っている総てのメソッドは、参考資料としてこのチュートリアルの終わりに記しているので、そちらを参照されたい。

HTTP 要求メッセージの詳細は「HTTP」の章で既に説明してある。このメッセージに含まれている情報を取り出すには豊富なメソッドたちが用意されている。これらのメソッドたちは次のように分類されよう:

- 要求行(開始行)に関する情報を取得
- ヘッダ行に関する情報を取得
- 要求行のクエリ文字列、あるいはボディ部分に含められている要求パラメタたちを取得
- ボディ部に含まれている要求パラメタやその他のデータ取得のためのストリーム、リーダ、あるいは Part を取得

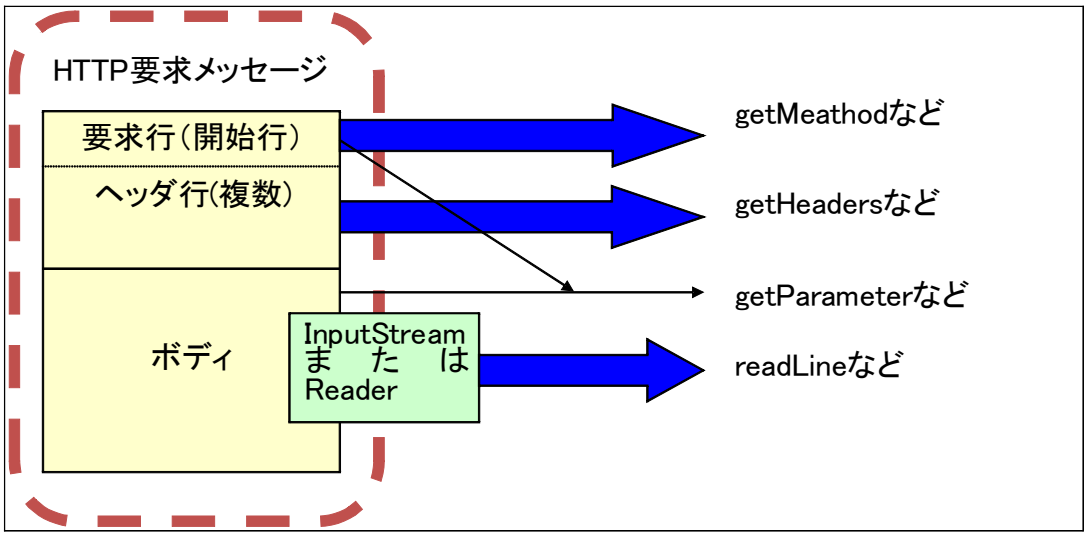


図 7-3：要求メッセージ・データ取得のメソッドたち

7.2.1　要求行に関する情報取得のためのメソッドたち

初期要求行の内容を知るにはどうしたらよいのであろうか？それには以下のメソッドを使用する：

表 7-1：初期要求行の情報取得の為のメソッドたち

メソッド	内容
getRemoteAddr	String 型でクライアントまたはプロキシの IP アドレスを取得
getRemoteHost	String 型でクライアントの完全修飾名または IP アドレスを取得
getRemotePort	int でクライアントまたはプロキシのポート番号を取得
getLocalAddr	String 型で受信したインターフェイスの IP アドレスを取得
getLocalName	String 型で受信したインターフェイスのホスト名を取得
getLocalPort	int で受信したインターフェイスのポート番号を取得
getServerName	String 型でサーバ名を取得
getServerPort	int でサーバの TCP ポート番号を取得
getScheme	String 型でこの要求を作るのに使われた http, https, あるいは ftp などのスキームを取得
isSecure	boolean で HTTPS などのセキュアなチャンネルを介して送られてきたかを知る
getMethod	String 型で HTTP メソッドを取得
getRequestURI	String 型で要求 URI を取得
getRequestURL	String 型でそのクライアントが使った URL を再構築して取得
getPathInfo	クライアントが送信した URL に関わる追加パス情報を取得
getPathTranslate	String 型でサーブレット名の後でクエリ文字列の前の追加的パス情報を実際のパスに変換して取得
getContextPass	String 型で要求 URI のなかのコンテキストを示す部分を取得
getServletPass	String 型でこのサーブレットを呼んでいるこの URL の部分を取得
getQueryString	String 型で URL エンコードされたクエリ文字列を取得（例え

メソッド	内容
	ば”http://localhost:8080/HttpRequestDump?MyName=Terry&Yesterday=Wednesday”というURLの場合は”MyName=Terry&Yesterday=Wednesday”なる文字列が返される)
getProtocol	String型でこの要求が使っているプロトコルをHTTP/1.1のように取得

このように、非常に多くのメソッドがあるが、これらのメソッドは実際のサーブレットでは頻繁に使われることはない。

なお、これらのメソッドで得られる要求URLの要素を下図に示す。これとHTTP仕様上の定義とは一部異なるので注意されたい。

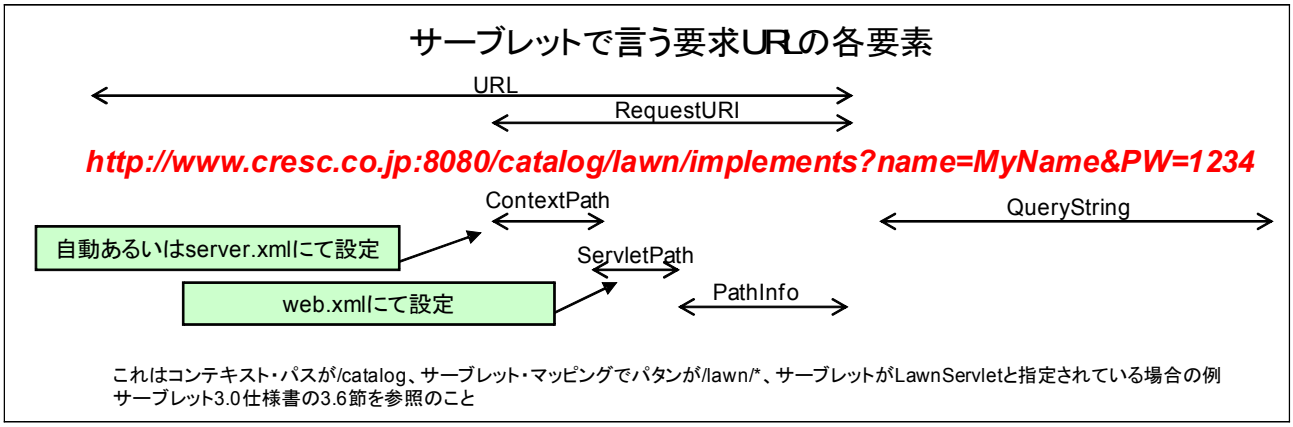


図 7-4：要求URLと各メソッドで取り出す要素

requestURI = contextPass + servletPath + pathInfo の関係があること、queryStringには"?”文字が含まれないことにも注意されたい。

7.2.2　ヘッダ行に関する情報取得のためのメソッドたち

ヘッダ行は2.6節の「ヘッダ行」で説明したように多くのものが存在する。開発者たちが良く利用するヘッダは次のようなものであろう：

- Accept　クライアントが受理可能なMIMEタイプ
- Accept-Charset　ISO-8859-1などクライアントが処理可能な文字セット
- Connection　TCPソケットを接続したままにするか否か(keep-aliveまたはclose)
- From　要求者のe-mailアドレス(Spiderが使う)
- Host　元のURLにあるホストとポート(HTTP/1.1では必須)
- If-Modified-Since　これより新しいバージョンのコンテントを送信要求
- Pragma　no-cacheという値はプロキシにキャッシュしないでいつも要求を転送するよう指示
- Referer　参照しているウェブ・ページのURL
- User-Agent　ブラウザまたはクライアント
- Content-Length　ボディ部にあるPOSTデータのバイト長
- Content-Type　application/x-www-form-urlencodedなどPOSTデータのエンコーディング

ヘッダ行の内容を知るにはどうしたらよいのであろうか？それには以下のメソッドを使用する：

表 7-2：ヘッダ行の情報取得の為のメソッドたち

メソッド	内容
getHeader	String型の名前を指定して値をString型で取得

メソッド	内容
getHeaders	列挙型でヘッダの全てを取得
getHeaderNames	列挙型でヘッダ名の全てを取得
getIntHeader	名前を指定して int 値で整数型ヘッダを取得
getDateHeader	名前を指定して long 値で日付型ヘッダを取得

注意することは、ヘッダ名はブラウザやサーブレット・コンテナでは大文字と小文字を区別せず処理されるが、**ヘッダ値は HTTP メッセージ上の文字列がそのまま返されるので、大文字と小文字を区別なく扱うのは、サーブレット側の責任**である。

7.2.3 ボディ部のデータを取り出す為のメソッドたち

ボディ部のデータは POST 要求メッセージのなかでコンテントに関するヘッダ行に基づきエンコードされている。たとえば:

- Content-Type : application/x-www-form-urlencoded
- Content-Type: text/html;charset=windows-31j
- Transfer-Encoding: chunked
- Content-Length : 123

などのヘッダが良く目にするボディ部分を記述しているヘッダ行である。コンテント・タイプは特に指定しない限りデフォルトとしての URL エンコーディングが適用される。**URL エンコーディングは GET 要求のクエリとの整合性を持たせる為にも、好ましいエンコーディングである。**URL エンコーディングについては第 2 章の「URL エンコーディング」の項を参照のこと。

7.2.3.1 getParameterNames と getParameterValues について

はじめに GET 要求のクエリ文字列、及び POST 要求のボディ・データから要求パラメタたちを取得する、便利ではあるが特殊なメソッドについて説明する必要がある。

表 7-3 : GET と POST に関わらずに要求パラメタ取得する為のメソッドたち

メソッド	内容
getParameterNames	String の列挙型でこの要求に含まれている要求パラメタの名前たちを取得
getParameter	String 型でこの要求に含まれている与えられた名前を持ったパラメタの値を取得。このメソッドを使うときにはその名前のパラメタがただひとつの値しか持っていないことをあらかじめ確認しておく必要がある。(どうしてこのメソッドの名前が getParameterValue になっていないのか疑問、この名前は混乱を与える)
getParameterValues	String の配列で与えられた名前のパラメタの値たちを取得

これらのメソッドは、クエリ文字列と同様に、POST データも名前と値ともに URL エンコードされていることが前提である。application/x-www-form-urlencoded は FORM のデフォルトのエンコーディングである。URL エンコード以外でエンコードされている POST 要求パラメタをこれらのメソッドでアクセスしても正しい情報が得られないことを後で実験で確認することにする。URL エンコーディングについては第 2 章の「URL エンコーディング」の項を参照のこと。

7.2.3.2　リーダまたはストリーム経由でボディ部データを読み出す

下表はボディ部をリーダまたはストリームを介して読み出すためのメソッドである。

表 7-4：POST 要求パラメタ取得する為のメソッドたち

メソッド	内容
getInputStream	ServletInputStream 型のバイト読み出し用のストリームを取得する
getReader	BufferedReader 型の文字データ読み出し用のリーダを取得する。ヘッダ行で指定された文字エンコードで変換を行う
ServletInputStream.readLine	バイト読み出しのストリームでありながら byte 配列に NL 文字が来るまでその行を読み出す
ServletInputStream.read	byte 配列にバイト・データを読み込む

リーダとストリームは同時使用は出来ないことに注意されたい。また getParameter や getParameterValues で既にデータを読み出した後では、これらのストリームやリーダを使って読みだそうとしても、もとのバッファは既に空になっていて、エラーが返されることになる。その逆も同じである。

7.2.3.3　ファイル・アップロード

サーブレット 3.0 からはファイル・アップロードのためのメソッドたちとクラスが追加されている。この機能に就いては別途説明するが、追加されたメソッドとクラスのみを下表に示す：

表 7-5：ファイル・アップロードののために追加されたメソッド

メソッド	内容
getParts	Part 型の Collection で、multipart/form-data エンコーディングで送られた Part 要素の総てを取得
getPart	与えられた名前の Part を取得
Part#getInputStream	このパートのコンテントを InputStream として取得
Part#getContentType	このパートのコンテント・タイプを取得
Part#getName	このパートの名前を取得
Part#getSize	このファイルのサイズを返す
Part#write	このアップロードされたアイテムをディスクに書き込むための利便性のためのメソッド
Part#delete	ファイル・アイテムのための蓄積を、それに関わる一時的ディスク・ファイルを含めて削除する
Part#getHeader	指定された MIME ヘッダの値を String として返す
Part#getHeaders	与えられた名前の Part ヘッダの値たちを返す
Part#getHeaderNames	この Part のヘッダ名たちを返す

7.3節　HttpRequestDump サーブレットによる要求メッセージの把握

前節で紹介したメソッド群をどのように使って必要な情報を取得するかに関しては、次のような HttpRequestDump というサーブレットがその見本となろう。このサーブレットは、HttpServletRequest クラスが持つ要求メッセージ情報を取得する為の総てのメソッドがどのように使われ、どのような情報を提供するかを示している。

7.3.1　HttpRequestDump のコード

先ずこのサーブレットのコードを示す。このコードはサーブレットとなっているが、簡単に皆さんのサーブレットに追加できるし（dumpRequest というメソッド）、フィルタとして必要に応じ付加するようにも出来る。このようなコードは、開発段階で開発者たちが、自分のアプリケーションにクライアントがどのような要求メッセージを送ってきているのか、を確認する上でも有用なものである。既に述べた MINA プロキシも TCP 上でのメッセージを確認するのに有用ではあるが、このコードの出力からも、実際の HTTP 要求メッセージを推定できる。

```
package basic_package;

import java.io.*;
import java.nio.ByteBuffer;
import java.util.*;

import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

/**
 * HttpRequestDump は、HTTP 要求オブジェクトをクライアントに返すサーブレット
 * December 2010, CRESC Corps.
 */
@WebServlet("/HttpRequestDump")
public class HttpRequestDump extends HttpServlet {
   private static final long serialVersionUID = 4874068692136631 45L;

   /**
    * 到来 HTTP GET 要求の処理
    */
   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res);
   }
   /**
    * 到来 HTTP POST 要求の処理
    */
   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res);
   }
   /**
    * 本サーブレット情報の文字列を返す
    */
   @Override
   public String getServletInfo() {
      return "HttpRequestDump, Version 2.0 by Cresc";
   }
   /**
    * ダンプ処理
    */
   public void performTask(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {

      res.setContentType("text/html; charset=windows-31j");//応答ヘッダ Content-Type 追加
```

```
      PrintWriter out = res.getWriter();        //出力バッファ取得
      dumpRequest(req, res, out);               //ダンプ処理のメソッドを呼ぶ
      out.println("</PRE></BODY></HTML>");      //HTML フッタの出力
      out.flush();                              //バッファの明示的吐き出し
      out.close();                              //バッファの明示的クローズ
   }
   /**
    * ダンプ処理の実体
    * @throws IOException
    */
   public void dumpRequest(HttpServletRequest req, HttpServletResponse res,
         PrintWriter out) throws IOException {
      // HTML ヘッダの出力
      out.println("<HTML><HEAD><TITLE>");
      out.println("要求オブジェクトの情報");
      out.println("</TITLE></HEAD>");
      // HTML ボディの出力
      out.println("<BODY><H1>要求オブジェクトの情報</H1><PRE>");
      out.println("getAuthType: " + req.getAuthType());
      out.println("getCharacterEncoding: " + req.getCharacterEncoding());
      out.println("getContentLength: " + req.getContentLength());
      out.println("getContentType: " + req.getContentType());
      out.println("getContextPath: " + req.getContextPath());
      out.println("getLocalAddr: " + req.getLocalAddr());
      out.println("getLocale: " + req.getLocale().toString());
      out.println("getLocalName: " + req.getLocalName());
      out.println("getLocalPort: " + req.getLocalPort());
      out.println("getMethod: " + req.getMethod());
      out.println("getPathInfo: " + req.getPathInfo());
      out.println("getPathTranslated: " + req.getPathTranslated());
      out.println("getProtocol: " + req.getProtocol());
      out.println("getQueryString: " + req.getQueryString());
      out.println("getRemoteAddr: " + req.getRemoteAddr());
      out.println("getRemoteHost: " + req.getRemoteHost());
      out.println("getRemotePort: " + req.getRemotePort());
      out.println("getRemoteUser: " + req.getRemoteUser());
      out.println("getRequestURI: " + req.getRequestURI());
      out.println("getRequestedSessionId: " + req.getRequestedSessionId());
      out.println("isRequestedSessionIdValid: " + (new
Boolean(req.isRequestedSessionIdValid())).toString());
      out.println("isRequestedSessionIdFromCookie: " + (new
Boolean(req.isRequestedSessionIdFromCookie())).toString());
      out.println("isRequestedSessionIdFromURL: " + (new
Boolean(req.isRequestedSessionIdFromURL())).toString());
      out.println("getScheme: " + req.getScheme());
      out.println("isSecure: " + (new Boolean(req.isSecure())).toString());
      out.println("getServerName: " + req.getServerName());
      out.println("getServerPort: " + req.getServerPort());
      ServletContext context = getServletContext();
      out.println("getRealPath: " + context.getRealPath(req.getRequestURI()));
      out.println("getServletPath: " + req.getServletPath());
      out.println("getContextPath: " + req.getContextPath());
      out.println("isAsyncSupported: " + (new Boolean(req.isAsyncSupported())).toString());
      out.println("isAsyncStarted: " + (new Boolean(req.isAsyncStarted())).toString());

      out.println();
      out.println("要求パラメタ(Request Parameters):");
      //注意：setCharacterEncoding メソッドは GET 要求には効かないので使わないことを奨励
      Enumeration<?> paramNames = req.getParameterNames();
      while (paramNames.hasMoreElements()) {
         String name = (String) paramNames.nextElement();
         String[] values = req.getParameterValues(name);
         try{
            out.println("  "+new String(name.getBytes("8859_1"), "Windows-31J")+":");
         }catch(UnsupportedEncodingException theException){
            out.println("UnsupportedException の例外が発生");
         }
         for (int i = 0; i < values.length; i++) {
            try{
               out.println("  "+i+")  "+new String(values[i].getBytes("8859_1"), "Windows-31J"));
            }catch(UnsupportedEncodingException theException){
               out.println("UnsupportedException の例外が発生");
            }
```

```
        }
    }

    out.println();
    out.println("要求ヘッダ(Headers):");
    Enumeration<?> headerNames = req.getHeaderNames();
    while (headerNames.hasMoreElements()) {
       String name = (String) headerNames.nextElement();
       String value = req.getHeader(name);
       try{
          String correctName = new String(name.getBytes("8859_1"), "Windows-31J");
          String correctValue = new String(value.getBytes("8859_1"), "Windows-31J");
          out.println("  " + correctName + " : " + correctValue);
       }catch(UnsupportedEncodingException theException){
          out.println("UnsupportedExceptionの例外が発生");
       }
    }

    out.println();
    out.println("属性(Attributes):");
    Enumeration<?> attributeNames = req.getAttributeNames();
    while (attributeNames.hasMoreElements()) {
       String name = (String) attributeNames.nextElement();
       Object value = req.getAttribute(name);
       out.println("  " + name + " : " + value.toString());
    }

    out.println();
    out.println("クッキー(Cookies):");
    Cookie[] cookies = req.getCookies();
    if ((cookies != null) && (cookies.length > 0)) {
       for (int i = 0; i < cookies.length; i++) {
          String name = cookies[i].getName();
          String value = cookies[i].getValue();
          out.println("  " + name + " : " + value);
       }
    }

    out.println();
    out.println("ロケール(Locales):");
    Enumeration<?> locales = req.getLocales();
    while (locales.hasMoreElements()) {
       Locale locale = (Locale) locales.nextElement();
       out.println("  " + locale.getDisplayLanguage());
    }

    int cl = req.getContentLength();
    if ( cl >= 0){
       out.println();
       out.println("ボディ部(Hexa Dump of the Body Data):");
       ServletInputStream sis = req.getInputStream();
       byte[] bodybytes = new byte[cl];
       int bs = sis.read(bodybytes, 0, cl);
       if (bs == -1){
          out.println("ボディ部は既に読みだされています!!");
       } else {
          ByteBuffer bb = ByteBuffer.wrap(bodybytes, 0, bs);
          int linemax = bs / 16;
          byte[] bytes = new byte[16];
          for (int line =0; line < linemax; line++){
             bb.get(bytes, 0, 16);
             out.print(toHexString(bytes));
             out.print(" : ");
             out.println(toPrintableString(bytes));
          }
          byte[] remainingbytes = new byte[bs % 16];
          bb.get(remainingbytes, 0, bs % 16);
          String linestr = toHexString(remainingbytes);
          out.print(linestr);
          out.print(" : ");
          out.println(toPrintableString(remainingbytes));
       }
    }
    out.println();
```

```
      out.println("***  以上  ***");
   }

   public static String toHexString( byte[] bytes ){
      StringBuffer sb = new StringBuffer( bytes.length * 3 );
      for( int i = 0; i < bytes.length; i++ ){
         sb.append( toHex(bytes[i] >> 4) );
         sb.append( toHex(bytes[i]) );
         sb.append( " " );
      }
      return sb.toString();
   }

   private static char toHex(int nibble){
      final char[] hexDigit =
      {'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
      return hexDigit[nibble & 0xF];
   }

   public static String toPrintableString( byte[] bytes ){
      bytes = toPrintable(bytes);
      StringBuffer sb = new StringBuffer( bytes.length);
      for( int i = 0; i < bytes.length; i++ ){
         sb.append( (char)bytes[i] );
      }
      return sb.toString();
   }

   public static byte[] toPrintable(byte[] bytes){
      for (int i = 0; i < bytes.length; i++ ){
         if (bytes[i] < 31 || bytes[i] > 127){
            bytes[i] = 46;
         }
      }
      return bytes;
   }

}
```

このサーブレットのコードは、サーブレットを書く際の幾つかのポイントを示している:

1. **doGetも doPostも同じメソッド(performTask)をアクセスし、対等に扱う。**これは昔からIBMが推奨していた書き方である。GETでもPOSTでもどちらでも良いようなアプリケーションは、柔軟性があると言える。また、送るときは厳格に受けるときには寛容にという昔からのインターネットのエチケット(Netiketともいう)に沿ったものである。なお、doPostメソッドをオーバライドしていないと、POST要求が来たときにTomcatは次のようなエラー・メッセージを返す:
   「HTTPステータス400 – HTTPのPOSTメソッドは、このURLではサポートされていません。」
   これは親切すぎともいえよう。
   但しこの方法はRESTfulなウェブ・サービスの場合のようにGETとPOSTの使い方を厳格に区別したアプリケーションでは適用されないことに注意されたい。
2. getServletInfoメソッドをオーバライドして、サーブレット名、作成者、作成日などを記入するのは良い習慣である。
3. **POST要求パラメタもURLエンコーディングを使う**こと。でないと、getParameterやgetParameterValuesでパラメタが取得できなくなる。デフォルトのURLエンコーディングであれば、POSTもGETも対等に扱うことが出来る。
4. **Tomcatは要求パラメタ(名前と値とも)はISO 8859_1で文字エンコーディングされていると想定**して、URLエンコードされたバイト列をJavaのUnicodeに変換している。残念ながら日本ではWindows-31Jが最も多用されているエンコーディングなので、new String(name.getBytes("8859_1"), "Windows-31J")などと変換しなおす必要がある。そうではなくて、setCharacterEncodingメソッドで前もってコンテナに対し8859_1ではなくてWindows-31Jだと指示することが出来るが、ただしこれはPOSTデータにのみ適用される。どうしてクエリ文字列に適用されないのかわからないが、GETとPOSTを対等に使えるようにするという趣旨からは、**setCharacterEncodingメソッドは使わないことをお勧めする。**

7.3.2 HttpRequestDump を使ってみる

1. はじめにこのサーブレットを呼び出すための HTML ファイルを Eclipse の tutorial\WebContent\WEB-INF に置くことにする。
2. パッケージ・エクスプローラ上で tutorial\WebContent\を右クリック→「新規(new)」→「HTML ファイル(HTML File)」をクリックし「新しいファイル(New File)」のペインでファイル名に RequestTestForm1.html を記入して「完了(Finish)」をクリックする。
3. 編集エリアにこのファイルの中身を次のように入力(コピー/ペーストで可)して保管(Save)する:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-31j">
<title>Request Message Test HTML (1)</title>
</head>

<body>
<H1>サーブレットに要求パラメタとしてテキスト・ボックスの内容をわたす</H1>
<form method="post" action="http://localhost:8080/tutorial/HttpRequestDump">
<!--
この行の代わりに
<form method="post" action="http://localhost:8080/tutorial/HttpRequestDump"
enctype="text/plain">
を使うとどうなるかも試すこと
-->
<textarea rows="4" cols="40" name="サブミット・データ"></textarea><br>
<input type="submit" value="Submit using POST">
</form>
<br>
<form method="get" action="http://localhost:8080/tutorial/HttpRequestDump">
<textarea rows="4" cols="40" name="ゲット・データ"></textarea><br>
<input type="submit" value="Submit using GET">
</form>
</body>

</html>
```

4. 次に HttpRequestDump というサーブレットを tutorial\Java Resources: src\basic_package に置く。これは HelloWord でやったと同じ手順を使えば良い。
5. パッケージ・ナビゲータ上で新しく作った basic_package を右クリック、「新規(new)」→「クラス(Class)」を選択する。
6. 「新規 Java クラス(New Java Class)」のペインで、サーブレット名 HttpRequestDump とスーパー・クラスを入力する。スーパー・クラス(HttpServlet)の入力には ctrl＋スペースで候補をリストアップさせ、選択できる。
7. そうすると HttpRequestDump というサーブレットが basic_package の下に作られ、またそのテキストがエディタに表示される。
8. エディタ上で前項の HttpRequestDump.java のコードをカット/ペーストで入力し、エラーが出なければ保管(Save)する
9. この状態で Tomcat7 の開始と停止の項で示した手順で Tomcat を起動させる。
10. 自分のブラウザで **http://localhost:8080/tutorial/RequestTestForm1.html** というアドレスを入力してこのファイルをアクセスする。
11. そうすると次のような画面が表示される(これは Google Chrome を使った例)ので、適当なテキストを入力し"Submit using POST”のボタンをクリックする。

図 7-5：テスト用 HTML ファイルから POST データを入力

12. そうするとブラウザには次のような HttpRequestDump サーブレットの応答が表示される：

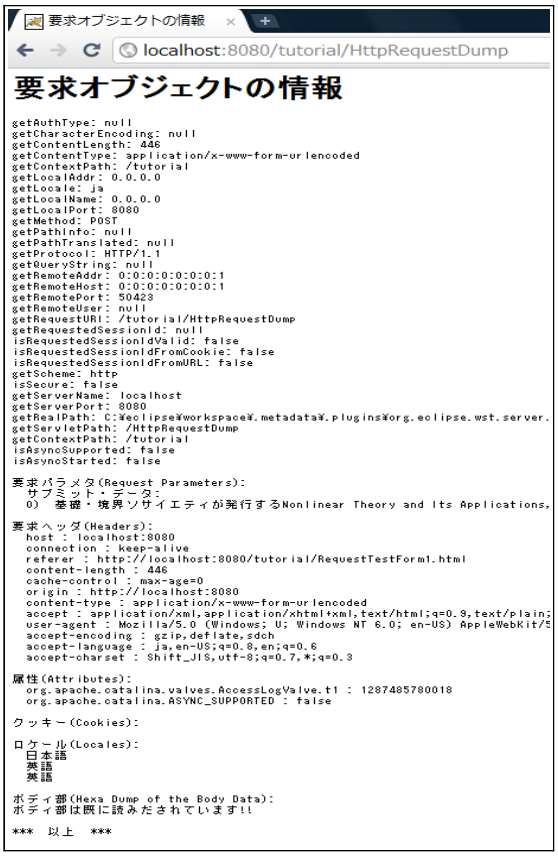


図 7-6：HttpRequestDump の応答例

この図は細かくて見づらいので、テキストとして行番号をつけてその意味を説明することにする：

```
001 要求オブジェクトの情報
002
003 getAuthType: null
004 getCharacterEncoding: null
005 getContentLength: 446
006 getContentType: application/x-www-form-urlencoded
007 getContextPath: /tutorial
008 getLocalAddr: 0.0.0.0
009 getLocale: ja
010 getLocalName: 0.0.0.0
011 getLocalPort: 8080
012 getMethod: POST
```

```
013 getPathInfo: null
014 getPathTranslated: null
015 getProtocol: HTTP/1.1
016 getQueryString: null
017 getRemoteAddr: 0:0:0:0:0:0:0:1
018 getRemoteHost: 0:0:0:0:0:0:0:1
019 getRemotePort: 50423
020 getRemoteUser: null
021 getRequestURI: /tutorial/HttpRequestDump
022 getRequestedSessionId: null
023 isRequestedSessionIdValid: false
024 isRequestedSessionIdFromCookie: false
025 isRequestedSessionIdFromURL: false
026 getScheme: http
027 isSecure: false
028 getServerName: localhost
029 getServerPort: 8080
030 getRealPath:
C:\eclipse\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\tutorial\tutori
al\HttpRequestDump
031 getServletPath: /HttpRequestDump
032 getContextPath: /tutorial
033 isAsyncSupported: false
034 isAsyncStarted: false
035
036 要求パラメタ(Request Parameters):
037   サブミット・データ:
038   0)  基礎・境界ソサイエティが発行するNonlinear Theory and Its Applications, IEICEが、2010年10月1日
に創刊されました。非線形科学・工学 分野をカバーする年4回発行の査読付き英文論文誌 (on-line journal) です.
039
040 要求ヘッダ(Headers):
041   host : localhost:8080
042   connection : keep-alive
043   referer : http://localhost:8080/tutorial/RequestTestForm1.html
044   content-length : 446
045   cache-control : max-age=0
046   origin : http://localhost:8080
047   content-type : application/x-www-form-urlencoded
048   accept :
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
049   user-agent : Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US) AppleWebKit/534.3 (KHTML, like
Gecko) Chrome/6.0.472.63 Safari/534.3
050   accept-encoding : gzip,deflate,sdch
051   accept-language : ja,en-US;q=0.8,en;q=0.6
052   accept-charset : Shift_JIS,utf-8;q=0.7,*;q=0.3
053
054 属性(Attributes):
055   org.apache.catalina.valves.AccessLogValve.t1 : 1287485780018
056   org.apache.catalina.ASYNC_SUPPORTED : false
057
058 クッキー(Cookies):
059
060 ロケール(Locales):
061   日本語
062   英語
063   英語
064
065 ボディ部(Hexa Dump of the Body Data):
066 ボディ部は既に読みだされています!!
067
068 ***  以上  ***
```

- 3-34 行は、基本的にアルファベット順に使えるメソッドの戻り値を出力している
- 36-38 行は、getParameterNames と getParameterValues を使って URL エンコードされたクエリ文字列または POST ボディに含められている要求パラメタの名前と値（ひとつの名前で複数の値を持ち得る）を取得して、表示している。この場合は要求パラメタはひとつで、名前が「サブミット・データ」、値はひとつだけで「基礎・境界ソサイエティが発行する...」というテキストである
- 40-52 行は要求ヘッダを到来順に出力している。この場合は content-length : 446（ボディ部分のバイト数

は 446)、content-type : application/x-www-form-urlencoded (コンテントのエンコーディングは URL エンコーディング)であることが分かる
- 54-56 行は、この要求オブジェクトに既に 2 つのオブジェクトが属性として貼り付けられていることを示している。これらはサーブレット・エンジンがログに付加する名前、及び非ブロック動作は不可であるという情報である
- 58 行はこの要求に含まれているクッキーであるが、今回の応答には含まれていない
- 60-63 行は、このブラウザが対応している言語を示す
- 65-66 行はボディ部のヘキサ・ダンプであるが、この場合は既に getParameterNames と getParameterValues でこのデータが呼び出されてしまっているので、出力はできない

要求パラメタ、要求ヘッダ、属性、クッキー、ロケール、ボディ部のバイト・データの読み出しなどの方法は、このプログラムのコードが参考になろう。

また、ヘキサ・ダンプは、昔 8 ビットや 16 ビットのマイコンをやっていた人たちには懐かしい形式である。各バイトを 16 進数で表示し、その行の右側には 7 ビット ASCII 文字として表示(印刷できない文字は".”で表示)してある。これにより具体的にどのようなバイト列としてこのメッセージのボディ部が送られてきているのかを知ることが出来る。この部分のコードは、ネットワークの下位層の担当者たちには有用であろう。

7.3.3 text/plain の問題点(**text/plain はブラウザによって異なった結果をもたらす**)

それでは、POST 要求で、ボディ・データのエンコーディングをデフォルトの URL エンコーディングから text/plain に変更したらどうなるであろうか？

1. Eclipse の編集画面で:
   <form method="post" action="http://localhost:8080/tutorial/HttpRequestDump">
   の行を
   <form method="post" action="http://localhost:8080/tutorial/HttpRequestDump" **enctype="text/plain"**>
   と変更して、保管(Save)する。
2. ブラウザで http://localhost:8080/tutorial/RequestTestForm1.html の画面に戻り、**再読み込みをクリック**して、前回と同じテキストを入力して、"Submit using POST”のボタンをクリックする。
3. そうすると今度は、次のような結果が得られる:

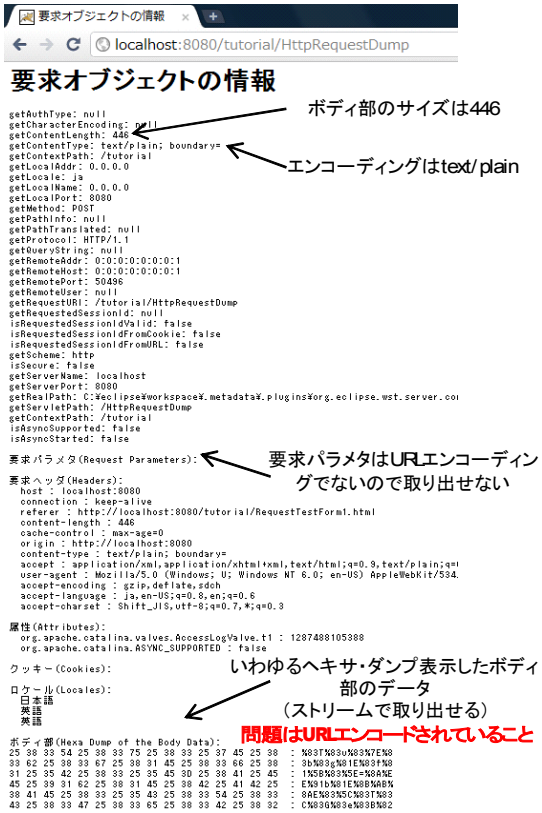


図 7-7：ボディ部のヘキサ・ダンプ

今回はヘッダ上は URL エンコーディングではないので、getParameterNames 及び getParameterValues でパラメタは呼び出されてはいないので、ストリームで読み出し、そのバイト・データを 16 進表示することが出来る。

ところが Google Chrome (6.0.472.63) では、なんとヘッダ行で content-type : text/plain; boundary=としていながら、ボディ部のエンコーディングは URL エンコードのままになっている！これは明らかにバグであろう。おなじことは Apple の Safari でもいえる。

```
ボディ部(Hexa Dump of the Body Data):
25 38 33 54 25 38 33 75 25 38 33 25 37 45 25 38  : %83T%83u%83%7E%8
33 62 25 38 33 67 25 38 31 45 25 38 33 66 25 38  : 3b%83g%81E%83f%8
31 25 35 42 25 38 33 25 35 45 3D 25 38 41 25 45  : 1%5B%83%5E=%8A%E
45 25 39 31 62 25 38 31 45 25 38 42 25 41 42 25  : E%91b%81E%8B%AB%
38 41 45 25 38 33 25 35 43 25 38 33 54 25 38 33  : 8AE%83%5C%83T%83
43 25 38 33 47 25 38 33 65 25 38 33 42 25 38 32  : C%83G%83e%83B%82
25 41 41 25 39 34 25 41 44 25 38 44 73 25 38 32  : %AA%94%AD%8Ds%82
25 42 37 25 38 32 25 45 39 4E 6F 6E 6C 69 6E 65  : %B7%82%E9Nonline
61 72 2B 54 68 65 6F 72 79 2B 61 6E 64 2B 49 74  : ar+Theory+and+It
73 2B 41 70 70 6C 69 63 61 74 69 6F 6E 73 25 32  : s+Applications%2
43 2B 49 45 49 43 45 25 38 32 25 41 41 25 38 31  : C+IEICE%82%AA%81
41 32 30 31 30 25 39 34 4E 31 30 25 38 43 25 38  : A2010%94N10%8C%8
45 31 25 39 33 25 46 41 25 38 32 25 43 39 25 39  : E1%93%FA%82%C9%9
31 6E 25 38 41 25 41 37 25 38 32 25 42 33 25 38  : 1n%8A%A7%82%B3%8
32 25 45 41 25 38 32 25 44 43 25 38 32 25 42 35  : 2%EA%82%DC%82%B5
25 38 32 25 42 44 25 38 31 42 25 39 34 25 46 31  : %82%BD%81B%94%F1
25 39 30 25 46 43 25 38 43 25 36 30 25 38 39 25  : %90%FC%8C%60%89%
43 38 25 38 41 77 25 38 31 45 25 38 44 48 25 38  : C8%8Aw%81E%8DH%8
41 77 2B 25 39 35 25 41 41 25 39 36 25 45 43 25  : Aw+%95%AA%96%EC%
38 32 25 46 30 25 38 33 4A 25 38 33 6F 25 38 31  : 82%F0%83J%83o%81
25 35 42 25 38 32 25 42 37 25 38 32 25 45 39 25  : %5B%82%B7%82%E9%
39 34 4E 34 25 38 39 25 46 31 25 39 34 25 41 44  : 94N4%89%F1%94%AD
25 38 44 73 25 38 32 25 43 43 25 38 44 25 42 38  : %8Ds%82%CC%8D%B8
25 39 33 25 43 37 25 39 35 74 25 38 32 25 41 42  : %93%C7%95t%82%AB
25 38 39 70 25 39 35 25 42 36 25 39 38 5F 25 39  : %89p%95%B6%98_%9
35 25 42 36 25 38 45 25 38 46 25 38 31 69 6F 6E  : 5%B6%8E%8F%81ion
2D 6C 69 6E 65 2B 6A 6F 75 72 6E 61 6C 25 38 31  : -line+journal%81
6A 25 38 32 25 43 35 25 38 32 25 42 37 2E  : j%82%C5%82%B7.
```

Mozilla Firefox(3.6.3)では更におかしな結果が得られる。サイズは 145 バイトでしかない！以下に示すように、どうやら 2 バイト文字を 1 バイトでしか送ってきていない。日本ではあまり使われていないようだが、世界的にはかなりのシェアを占めているブラウザなので、困ったものである:

```
ボディ部(Hexa Dump of the Body Data):
B5 D6 DF C3 C8 FB C7 FC BF 3D FA 0E FB 83 4C BD  : .........=....L.
B5 A4 A8 C6 A3 4C 7A 4C 59 8B 4E 6F 6E 6C 69 6E  : .....LzLY.Nonlin
65 61 72 20 54 68 65 6F 72 79 20 61 6E 64 20 49  : ear Theory and I
74 73 20 41 70 70 6C 69 63 61 74 69 6F 6E 73 2C  : ts Applications,
20 49 45 49 43 45 4C 01 32 30 31 30 74 31 30 08  :  IEICEL.2010t10.
31 E5 6B 75 0A 55 8C 7E 57 5F 02 5E DA 62 D1 66  : 1.ku.U.~W_.^.b.f
FB E5 66 20 06 CE 92 AB D0 FC 59 8B 74 34 DE 7A  : ..f ......Y.t4.z
4C 6E FB AD D8 4D F1 87 D6 87 8C 08 6F 6E 2D 6C  : Ln...M......on-l
69 6E 65 20 6A 6F 75 72 6E 61 6C 09 67 59 2E 0D  : ine journal.gY..
0A  : .
```

Internet Explorer ではそのような問題はなく、次のような結果が得られる。URL エンコーディングではないので、218 バイトのサイズで済んでいる:

```
ボディ部(Hexa Dump of the Body Data):
83 54 83 75 83 7E 83 62 83 67 81 45 83 66 81 5B  : .T.u.~.b.g.E.f.[
83 5E 3D 8A EE 91 62 81 45 8B AB 8A 45 83 5C 83  : .^=...b.E...E.\.
54 83 43 83 47 83 65 83 42 82 AA 94 AD 8D 73 82  : T.C.G.e.B.....s.
B7 82 E9 4E 6F 6E 6C 69 6E 65 61 72 20 54 68 65  : ...Nonlinear The
6F 72 79 20 61 6E 64 20 49 74 73 20 41 70 70 6C  : ory and Its Appl
69 63 61 74 69 6F 6E 73 2C 20 49 45 49 43 45 82  : ications, IEICE.
AA 81 41 32 30 31 30 94 4E 31 30 8C 8E 31 93 FA  : ..A2010.N10..1..
82 C9 91 6E 8A A7 82 B3 82 EA 82 DC 82 B5 82 BD  : ...n...........
81 42 94 F1 90 FC 8C 60 89 C8 8A 77 81 45 8D 48  : .B.....`...w.E.H
8A 77 20 95 AA 96 EC 82 F0 83 4A 83 6F 81 5B 82  : .w .......J.o.[.
B7 82 E9 94 4E 34 89 F1 94 AD 8D 73 82 CC 8D B8  : ....N4.....s....
93 C7 95 74 82 AB 89 70 95 B6 98 5F 95 B6 8E 8F  : ...t...p..._....
81 69 6F 6E 2D 6C 69 6E 65 20 6A 6F 75 72 6E 61  : .ion-line journa
6C 81 6A 82 C5 82 B7 2E 0D 0A  : l.j.......
```

以上のことから、**text/plain エンコーディングによる POST 送信は、ブラウザによって異なる結果が得られるので使わないほうが良い**ということになる。

7.3.4 URL エンコードされた POST データのストリームやリーダによる読み出し

URL エンコードされたデータはストリームやリーダで読み出せないという訳ではない。ストリームやリーダで読み出す前に getParameterNames 及び getParameterValues で読みだされていなければ、それは可能である。HttpRequestDump で、要求パラメタの出力のブロックを全部コメント・アウトすると、次のような結果が得られる:

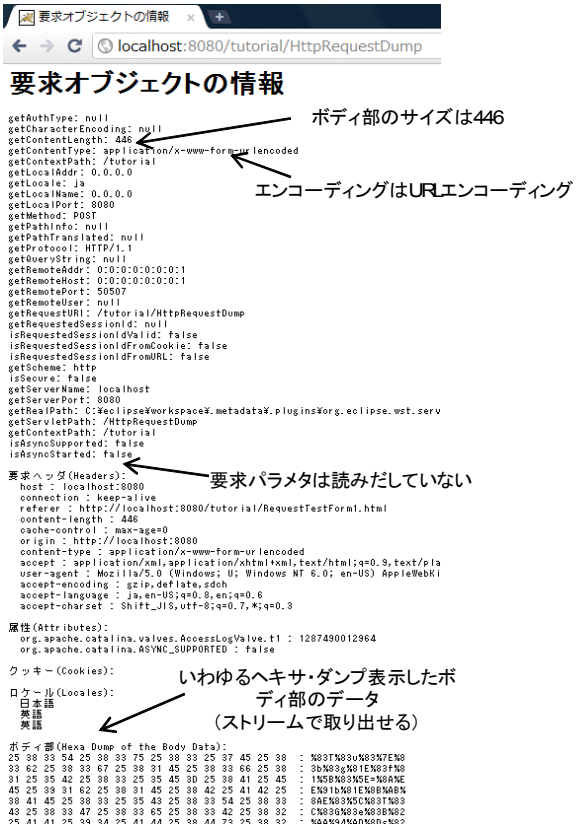


図 7-8：URL エンコードされたデータのストリームによる取得

これはストリームの read メソッドによる方法だが、以下に ServletInputStream で独自な唯一のメソッドである readLine で読みだす RequestBodyDumpViaStream といういささか長い名前のサーブレットを紹介しよう：

```
package basic_package;

import java.io.*;
import java.net.URLDecoder;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.CharacterCodingException;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;

import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

/**
 * RequestBodyDumpViaStream は、InputStream で要求パケットのボディを読み出し、
 * コンソールに表示する。
 */
@WebServlet("/RequestBodyOutViaStream")
public class RequestBodyOutViaStream extends HttpServlet {
   private static final long serialVersionUID = 8712298963391974602L;
   /**
    * 到来 HTTP GET 要求処理
    */
   @Override
   public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
      performTask(request, response);
   }
   /**
    * 到来 HTTP POST 要求処理
    */
   @Override
   public void doPost(HttpServletRequest request, HttpServletResponse response)
```

```
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(request, response);
   }
   /**
    * 本サーブレット情報の文字列を返す。
    */
   @Override
   public String getServletInfo() {
      return "RequstBodyDumpViaStream, Version 2.0 by Cresc";
   }
   /**
    * 到来要求処理
    */
   public void performTask(HttpServletRequest request, HttpServletResponse response) {
      try
      {
         // Insert user code from here.
         ServletInputStream sis = request.getInputStream();//バイト入力用ストリームを取得
         byte[] bytes = new byte[1024];      //バイトの配列を用意する
         int len = (sis.readLine(bytes,0,1024));//一行読み出し（URLには改行符号は入らない）
         String is="";
         for (int i = 0; i < len; i++) {  //これを文字列に変換（上８ビットはゼロ）
            is =is +(char)bytes[i];
         }
         // URLDecoderは、Windows-31Jの文字列だとしてUnicodeの文字列に変換するが、
         // ブラウザのURLエンコードとの互換性がないので注意のこと
//       String os = URLDecoder.decode(is, "Windows-31J");
         String os = urlDec(is, "Windows-31J");

         response.setContentType("text/html; charset=Windows-31J");
         PrintWriter out = response.getWriter();
         out.println("<HTML>");
         out.println("<HEAD><TITLE>要求ボディからの要求データ取得</TITLE></HEAD>");
         out.println("<BODY>");
         out.println("URLエンコードされたボディ・データ　：　" + is);
         out.println("<BR>");
         out.println("URLデコードされたボディ・データ　：　" + os);
         out.println("</BODY></HTML>");
      }
      catch(IOException theException) {
         theException.printStackTrace();
      }
   }
   public String urlDec(String s, String enc) throws UnsupportedEncodingException,
CharacterCodingException{
      final String hexDigit = "0123456789ABCDEF";
      ByteBuffer bb = ByteBuffer.allocate(s.length() * 2);
      for( int i = 0; i < s.length(); i++ ){
         if(s.charAt(i) == '+'){          // スペース変換
            bb.put((byte)' ');
         }
         else if (s.charAt(i) == '%'){    // シフト・バイト変換
            i++; int j = hexDigit.indexOf(s.charAt(i)) * 16;
            i++; j = j + hexDigit.indexOf(s.charAt(i));
            bb.put((byte)(j & 0xFF));
         }
         else {
            bb.put((byte)s.charAt(i));    // 無変換
         }
      }
      bb.flip();
      Charset cs = Charset.forName(enc); // 文字コードの選択
      CharsetDecoder dec = cs.newDecoder(); // バイトから文字へのデコーダの生成
      CharBuffer cb = dec.decode( bb );   // データ読み出しの為のbyte[]からchar[]への変換
      return cb.toString();
   }
}
```

このサーブレットを tutorial\Java Resources: src\basic_package に置き、これまで使ってきた RequestTestForm1.html のフォーム指定行を：

```
<form method="post" action="http://localhost:8080/tutorial/RequestBodyOutViaStream">
```

と変更して試すことができる。

前回の画面と同じテキストを入れて Submit Using POST のボタンをクリックすると、以下のような結果が返されてくる:

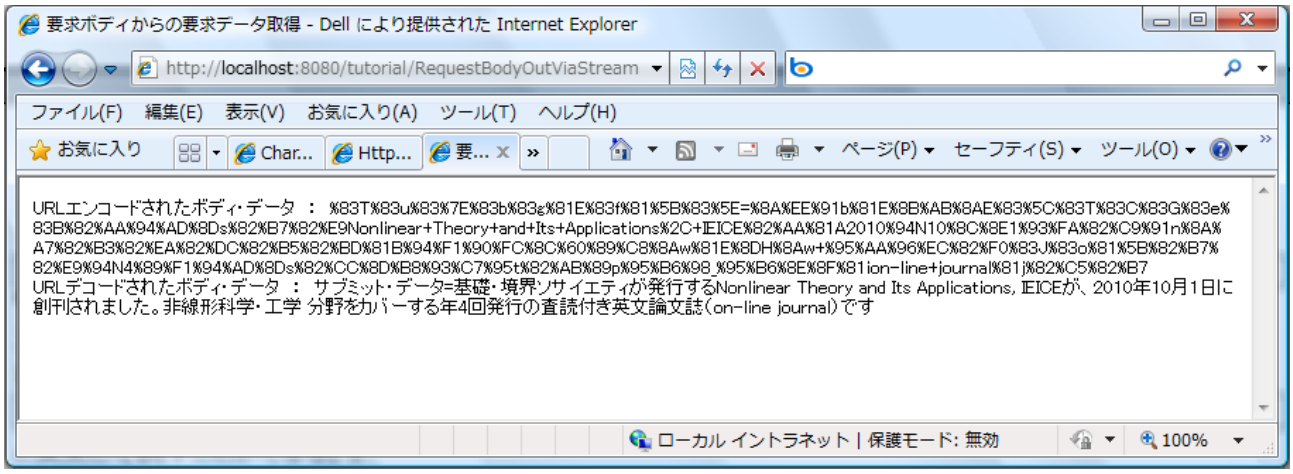


図 7-9:URL エンコードされたボディ・データの読み出しとデコード

ちなみに、このコードにはコメント・アウトされた URLDecoder#decode を使う行が入っている。このコメント・アウトを 1 行ずらすことで、この URLDecoder#decode メソッドが行っているデコードが使えないことが確認されよう。これは次の項で示すように、ブラウザで使われている URL エンコードと、Java.net が使っている URL エンコードが異なっているからである。

従って、**ストリームで読み出す場合は、このコードにある urlDec というメソッドを代わりに使う必要がある。**

7.3.5 java.net の URL デコーダの問題

ここで注意しなければならないことは、ブラウザの URL エンコードは java.net の URL エンコードとは異なっていることである。従って**ブラウザから送られてきた URL エンコードされた文字列は java.net.decode というメソッドでは正しくデコードされない**。これがときどき互換性問題を起こす。従ってブラウザからの URL エンコードされたバイト列をデコードするには、どちらにも対応できるメソッドが必要になる。

表 7-6:ブラウザと java.net の URL エンコードの相違

ブラウザの URL エンコード	java.net の URL エンコード
指定された指定された文字コードで 16 ビット及び 8 ビット(1 バイト文字)からなる文字列に変換する。その文字列を 8 ビットのバイト列として取り扱い、各バイトを ASCII 文字とみなして次のルールを適用する: • 英数字文字の「a」から「z」、「A」から「Z」、および「0」から「9」は元のまま残す • 特殊文字の「.」、「-」、「*」、および「_」は元のまま残す • 空白文字「 」をプラス記号「+」に変換する • ほかのすべてのバイトは安全でなく、各バイトが 3 文字から成る文字列「%xy」で表現される。ここで、xy はそのバイトを 2 桁の 16 進数とし	Java のユニコードを指定された文字コードで 16 ビット及び 8 ビット(1 バイト文字)からなる文字列に変換する。そしてその各文字に対して、次のルールを適用する: • 英数字文字の「a」から「z」、「A」から「Z」、および「0」から「9」は元のまま残す • 特殊文字の「.」、「-」、「*」、および「_」は元のまま残す • 空白文字「 」をプラス記号「+」に変換する • ほかのすべての文字は安全でなく、1 つまたは 2 つのバイトに変換される。次に、各バイトが 3 文字から成る文字列「%xy」で表現される。ここで、xy はそのバイトを 2 桁の 16 進数とし

て表現したもの	て表現したもの このようにして得られた String の各文字は上 8 ビットがゼロとなっているので、ネットワークに出すときは下 8 ビットだけを送出することになる。

従ってブラウザの URL エンコーディングでは、漢字一文字は%83T とエスケープしたバイト(%83)とエスケープしない文字(T)の組み合わせでエンコードされることがあるが、Java.net.URLEncoder の URL エンコードでは漢字のような 2 バイト文字は%81%45 のように、必ずエスケープした表現のバイト 2 つとしてエンコードされる。どちらも間違いではないが、ブラウザが使っているエンコードのほうが全体のバイト数が少なくて済む。**問題は Java.net.URLDecoder#decode が Java.net.URLEncoder の URL エンコードのみにしか対応していない**ことである。

前記のコードにある **urlDec メソッドは、URLEncoder#encode メソッドでエンコードした文字列も正しくデコードできる。早くこの java.net.URLDecoder#decode というメソッドを修正して欲しいものである。この** urlDec というメソッドは、新しい ByteBuffer や Charset / CharsetDecoder などが使われているので、比較的コンパクトに出来ている。

7.4節　この章のまとめ

HTTP 要求処理にあたってのポイントは次のようである(「HttpRequestDump のコード」の項で説明したポイントも参考のこと):

1. POST 要求も GET 要求も等しく受け付けるようにすることが好ましい(但し REST ベースウェブ・サービスのように明示的に区別している場合は除く)。
2. dumpRequest メソッド及びプロキシが、デバッグに有効なツールとなる。
3. GET と POST を対等に使えるようにするという趣旨からは、setCharacterEncoding メソッドは使わない
4. POST 要求パラメタも URL エンコーディングを使う
5. POST 要求で text-plain エンコードは使ってはならない。
6. ブラウザの URL エンコードと、Java.net の URLEncoder / URLDecoder とは異なっているので注意が必要である。ストリームで読み出す場合は、この章で使った urlDec というメソッドはどちらのエンコードでも受け付ける。

第8章　応答オブジェクト

サーブレット・コンテナは、クライアントに送り返すHTTP 応答をHttpServletResponse 型のオブジェクトにカプセル化してサーブレットに渡している。従ってこのオブジェクトはHTTP 応答そのものではない。この章では、この応答オブジェクトはソフトウエア開発者にどのような情報及びメカニズムを提供しているかを、主としてHTTP 応答メッセージとの対応を中心にして説明する。

8.1節　Tomcat が送信する HTTP 応答メッセージ

最初に Tomcat はどのような HTTP 応答メッセージを送信しているのかを確認しよう。TCP 上での Tomcat の出力を調べるには、「ブラウザが出す HTTP 要求」の節で説明した MINA プロキシが非常に有効である。既にその項では Tomcat の応答が示されているが、それを再掲すれば次のようなものだった:

応答メッセージ:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=Windows-31J
Content-Length: 111
Date: Fri, 15 Oct 2010 06:12:45 GMT

<HTML>
<HEAD><TITLE>Hello World</TITLE></HEAD>
<BODY>
<BIG>Hello World from ???´?L?W</BIG>
</BODY></HTML>
```

各行の意味は次のようだった:

001 HTTP/1.1 200 OK
　　HTTP バージョンは HTTP/1.1、ステータス・コードは 200、その意味は OK
002 Server: Apache-Coyote/1.1
　　サーバは Apache-Coyote/1.1 (Tomcat の HTTP コネクタ)
003 Content-Type: text/html;charset=Windows-31J
　　ボディ部は text/html で文字エンコーディングは Windows-31J (サーブレットが res.setContentType メソッドで指定した)
004 Content-Length: 111
　　ボディ部分の長さは 111 バイト(CR と LF を含む)
005 Date: Fri, 15 Oct 2010 06:12:45 GMT
　　日付ヘッダ
006
　　空白行はヘッダ部分の終わりを意味する
007 <HTML>
008 <HEAD><TITLE>Hello World</TITLE></HEAD>
009 <BODY>
010 <BIG>Hello World from ???´?L?W</BIG>
011 </BODY></HTML>

これらの行は、Tomcat 7 が付加した Server:、Content-Length:、及び Date ヘッダ行を除き、我々が作成したサーブレットが HttpServletResponse オブジェクトを使って Tomcat 7 に作らせたものである。次項からは、HTTP 応答メッセージ、及び HTTP 応答メッセージの要素（ステータス行、ヘッダ行、及びボディ）をセットする為のメソッドたちを説明することにする。

8.1.1 Tera Term による確認

MINA プロキシのプログラムを使うのが面倒な人は、Telnet という TCP のアプリケーションを使うことも可能である。このチュートリアルの初版を書いたときは、開発したサーブレットを TCP のレベルで確認するのに Telnet は重宝されたものである。

Telnet は IP ネットワークにおいて、遠隔地（リモート）にあるサーバを端末から操作できるようにする仮想端末ソフトウェア（端末エミュレータ）、またはそれを可能にするプロトコルのことを指し、RFC 854 で規定されている歴史的なアプリケーションのひとつである。Window は DOS レベルでこれを既に実装しているが、Tera Term（テラターム）は、Windows 向けのオープンな Telnet プログラムで、より広く利用されている。

1. 最初に Tomcat 7 の TCP 接続タイムアウト時間を 200 秒（単位は mS）長くする。通常はこの時間は 20 秒に設定されているが、Telnet で操作するにはこれでは短すぎる。
   1. Eclipse で Tomcat 7 を走らせている場合は：
      1. プロジェクト・エクスプローラ上で Servers\Tomcat v7.0 Server at localhost-config\server.xml を開く
      2. エディタで次のようにコネクタの設定を変更する：

```
<Connector connectionTimeout="200000" port="8080" protocol="HTTP/1.1"
redirectPort="8443"/>
```

図 8-1：Eclipse で TCP 接続タイムアウト値を長くする

      3. これを保管する。

   2. DOS のレベルで Tomcat 7 を走らせている場合は、メモ帳などのテキスト・エディタで Tomcat 7 の設定ファイル（TOMCAT_HOME\conf\server.xml）の TCP 接続タイムアウト値を 200 秒（単位は mS）に変更、上書き保存する：

```
<Connector port="8080" protocol="HTTP/1.1"
          connectionTimeout="200000"
          redirectPort="8443" />
```

2. Tera Term のダウンロードのページから実行ファイル（例えば teraterm-4.66.exe）をダウンロードしてインストールする

3. このプログラムを実行して、「設定」→「端末」で端末を次のように設定する

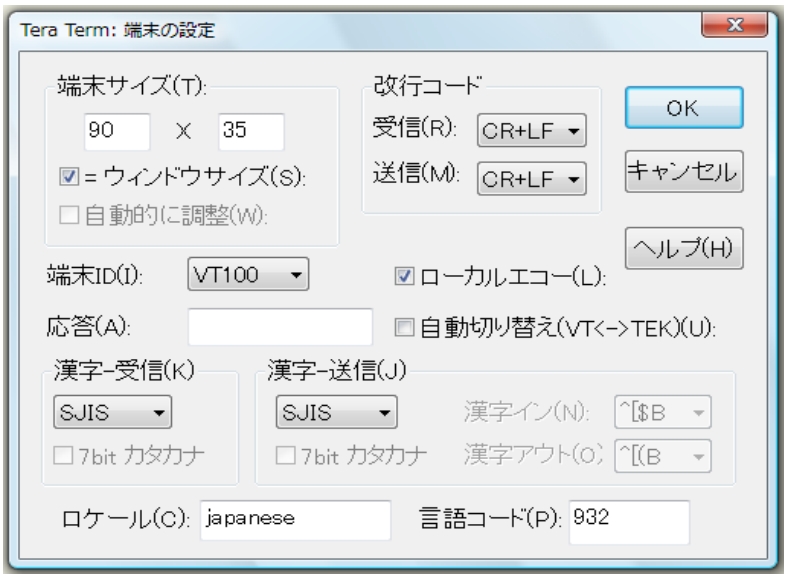


図 8-1：Tera Term の端末設定

4. 「設定」→「TCP/IP の設定」で、「Telnet」はチェック、「自動的にウインドウを閉じる」はチェックを外す。また「ポート」は 8080 にする。

5. 「設定」→「設定の保存」でこの設定を保存する。

6. Eclipse 上で、あるいは DOS のレベルで Tomcat 7 を開始させる。

7. 「ファイル」→「新しい接続」で次のように接続する。

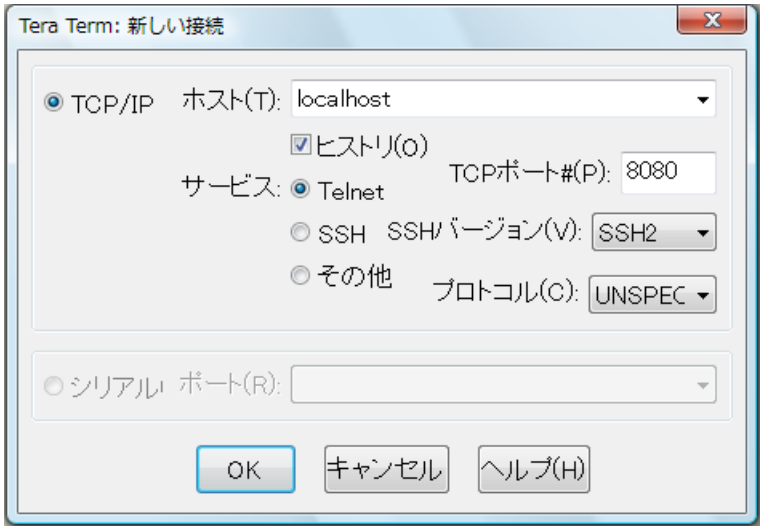


図 8-2：Tera Term による TCP 接続

8. 接続が取れたら一行単位でサーバと通信をする。下図はその例である：
空白行を含めて最初の 3 行がキーボードから入力した HTTP 要求メッセージであり、そのあとは HelloWorld サーブレットが返した応答メッセージである。

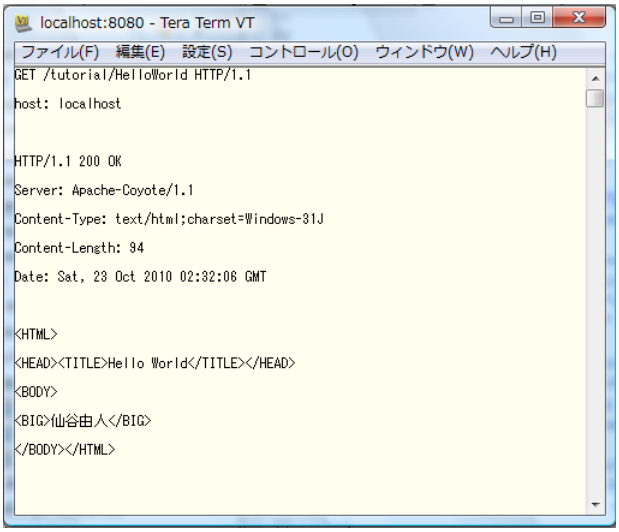


図 8-3：Tera Term によるサーブレットのアクセス

9. TCP 接続の終了は Tomcat のタイムアウトでも起きるが、「ファイル」→「接続断」で切ることも可能である。Tera Term はきちんとした TCP 接続解放手順をとっている。

8.2節　HTTP 応答メッセージの組み立てのためのメソッドたち

本節では HttpServletResponse インターフェイスのオブジェクトが持っている HTTP 応答メッセージ組み立ての為のメソッドたちを中心に説明する。HttpServletResponse 及びそのスーパー・クラスである ServletResponse の持っている総てのメソッドは、参考資料としてこのチュートリアルの終わりに記しているので、そちらを参照されたい。

HTTP 応答メッセージの詳細は「HTTP」の章で既に説明してある。このメッセージの各部分にデータをセットする為に豊富なメソッドたちが用意されている。これらのメソッドたちは次のように分類されよう：

- ステータス行（開始行）をセットする
- ヘッダ行をセットする
- ボディ部に含める応答データ書き込みのためのストリーム、あるいはライタを取得する

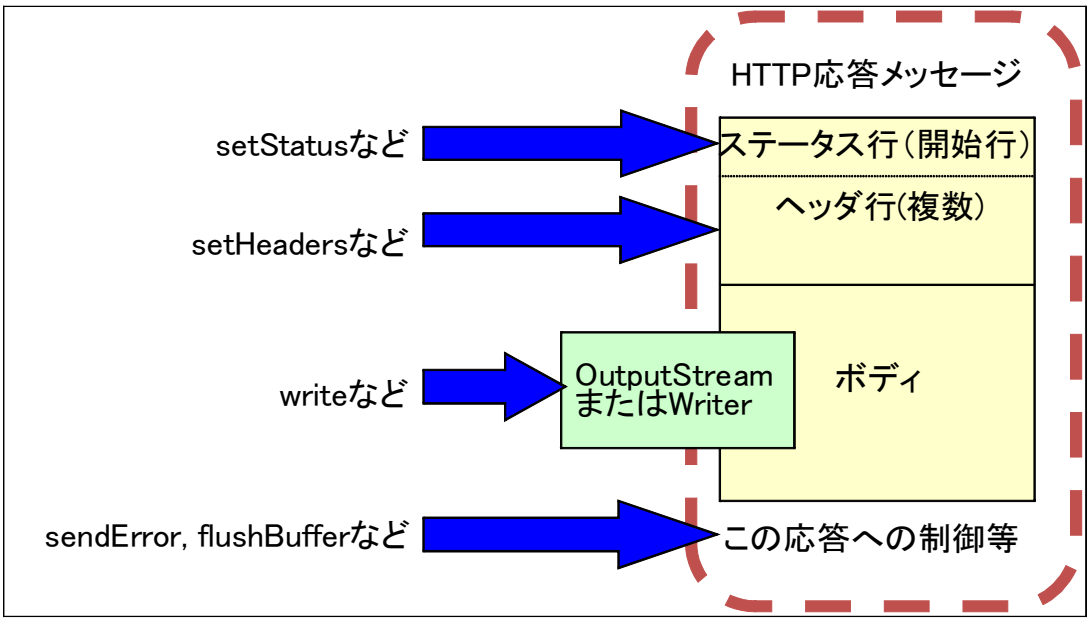


図 8-4:応答メッセージ・データ作成の為のメソッドたち

8.2.1 ステータス行のためのメソッドたち

ステータス行は「ステータス行」の節で説明したように、プロトコル・バージョン、ステータス・コード、及び理由句で構成されている。この行は通常はコンテナが生成するので、開発者が意識することはない。

しかしながらエラーでは、コンテナがエラーを検出する場合と、アプリケーションが検出する場合がある。

表 8-1:ステータス行の為のメソッドたち

メソッド	内容
sendError	エラー・コード(及びエラー・メッセージ)を指定してクライアントにエラー応答を送信する。エラー・メッセージが与えられていない場合は、コンテナは自分のエラー・ページのメカニズムを使ってその応答を送信する。ステータス・コードは必ずしもエラーを意味するものではないが、結果はエラー・ページが出力される。
sendRedirect	リダイレクト先を指定して、その要求をリダイレクト先に転送するが、クライアントへは 302 (Found) のステータス・コードの応答を送信する。
setStatus	200、300、400、及び 500 番台のエラーでないステータス・コードをセットする。表示画面はサーブレット側で用意する。

エラー・コードの番号と意味は、HttpServletResponse でフィールドとして定義されているので、そちらを見られたい。また「ステータス行」の節でも表にしてあるので、それも見られたい。

サーブレットが認知しないうちにコンテナがエラー画面を返してしまうエラーは、例えば次のようなものがある:

- サーブレットを実行中に例外がスローされた(500): この場合は、Tomcat はそれを受けてスタック・トレースをクライアントに返すのは皆さんが良く経験する動作である。JSP においても、errorPage="/error_page.jsp"などをページ・ディレクティ部にしていると、例外が起きれば error_page.jsp がクライアントに返される。
- HTTP 要求が正しくない(400, 404, 505 など)

8.2.2 ヘッダ行をセットする為のメソッドたち

以下はヘッダ行を設置するメソッドたちの表であるが、セッション管理関係は別の章で説明するので、この表には含めていない。

表 8-2：ヘッダ行の為のメソッドたち

メソッド	内容
setContentType	その応答がまだコミットされていないならば、そのクライアントに送られる応答のコンテント・タイプをセットする。与えられるコンテント・タイプには、例えば text/html;charset=UTF-8 のように文字エンコーディング指定が含まれていて良い。この応答の文字エンコーディングは、getWriter が呼ばれる前にこのメソッドが呼ばれているときにのみこのコンテント・タイプからセットされる。 このメソッドは文字エンコーディングを変えるために繰り返し呼ぶことができる。このメソッドは getWriter が呼ばれたあと、またはこの応答がコミットされた後では効果をもたらさない。この応答がコミットされた後あるいは getWriter が呼ばれた後でこのメソッドが呼ばれているときは、この応答の文字エンコーディングをセットしない。
setCharacterEncoding	これはヘッダ行をセットしないが、参考のためにここに含めた。 例えば UTF-8 のような、そのクライアントに送信される応答の文字エンコーディング（MIME charset）をセットする。setContentType(java.lang.String)あるいは setLocale(java.util.Locale)で既に文字エンコーディングがセットされているときには、このメソッドはそれを優越する。setContentType(java.lang.String)を text/html の String で呼び、このメソッドを UTF-8 の String で呼ぶことは、text/html; charset=UTF-8 の String で setContentType を呼ぶことと等価である。 このメソッドは文字エンコーディングを変えるために繰り返し呼ぶことができる。このメソッドは getWriter が呼ばれたあと、またはこの応答がコミットされた後では効果をもたらさない。エンコーディングを指定しないと、デフォルトの ISO-8859-1 が使われる。
setContentLength	与えられた整数値を持った content-length ヘッダをセットする
setLocale	その応答のロケールをセットする。setCharacterEncoding または setContentType で文字エンコーディングが使われていないときは、このロケールに基づいて適切な文字エンコーディングが選択される
setDateHeader	与えられた名前と値を持った date ヘッダを設定する。既に date ヘッダがセットされているときは、それを上書きする
addDateHeader	与えられた名前と値を持った date ヘッダを設定する。既に date ヘッダがセットされているときは、date ヘッダが追加される
setHeader	与えられた名前と値を持ったヘッダを設定する。既にその名前のヘッダがセットされているときは、それを上書きする
addHeader	与えられた名前と値を持ったヘッダを設定する。既にその名前のヘッダがセットされているときは、このヘッダは追加される
setIntHeader	与えられた名前と整数値を持ったヘッダを設定する。既にその名前のヘッダがセットされているときは、それを上書きする
addIntHeader	与えられた名前と整数値を持ったヘッダを設定する。既にその名前のヘッダがセットされているときは、このヘッダは追加される
containsHeader	指定した名前のヘッダが設定されているかどうかをブール値で返す

これらのメソッドに対応した get メソッドも用意されている。またこの表の 3 つのメソッド、すなわち setContentType、setContentLength、及び setLocale はヘッダ行のみならず、ボディ部にも影響を与える。

基本的に、ヘッダを書き込むメソッドは、getWriter でボディ部にデータを書き込む機構を取得する前に呼ばねばならない。HTTP/1.1 ではチャンクド転送で、データをボディ部に書き込むと直ちにそれがネットワークに出力(コミットされるという)され、ヘッダ部はその時点では送信されてしまっていて、もはや書き込むことはできない。これはクッキーを使っているセッションにも言えることである。また setContentType は Writer に書き込む際のエンコーディングとして使われるので、getWriter の前でなければならない。一方 OutputStream でデータを送るときのエンコーディングはサーブレット側で総て行わねばならない。

これらのメソッドは実際に応答メッセージにどのように影響するのかを知るには、「要求オブジェクト」の章の「プロキシのプログラムで HTTP 要求メッセージを調べる」の項で説明したプロキシのプログラムが威力を発揮する。MINA プロキシ(Main.java)を引数"12345 localhost 8080"ではしらせ、Tomcat を開始させ、HelloWorld.java の一部を変更させながら Internet Explorer から http://localhost:12345/tutorial/HelloWorld をアクセスすると、次のような結果が得られよう:

8.2.2.1 デフォルトの応答ヘッダ

ヘッダになにもセットしない(即ち res.setContentType("text/html; charset=Windows-31J");の行をコメント・アウトする)と、次のような応答になる:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Length: 90
Date: Tue, 26 Oct 2010 04:02:22 GMT

<HTML>
<HEAD><TITLE>Hello World</TITLE></HEAD>
<BODY>
<BIG>????</BIG>
</BODY></HTML>
```

即ちヘッダ行には Content-Length 以外はボディ部に関するヘッダ行は何も追加されておらず、漢字も正しくエンコードされない。

8.2.2.2 setLocale の効果

上記の状態で、getWriter の行の前に、res.setLocale(Locale.JAPAN);という行を追加すると、次のような応答が返る:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Language: ja-JP
Content-Length: 90
Date: Tue, 26 Oct 2010 04:20:22 GMT

<HTML>
<HEAD><TITLE>Hello World</TITLE></HEAD>
<BODY>
<BIG>????</BIG>
</BODY></HTML>
```

即ち Content-Language: ja-JP というヘッダが追加されるだけで、エンコーディングには影響を与えていない。仕様書上は Locale とエンコーディングのマッピングはコンテナに依存するとなっており、Tomcat 7 は現時点ではマッピングはされていない。

8.2.2.3　setCharacterEncoding の効果

代わりに res.setCharacterEncoding("Windows-31J");という行を使ってみると次のようになる:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Length: 94
Date: Tue, 26 Oct 2010 04:30:19 GMT

<HTML>
<HEAD><TITLE>Hello World</TITLE></HEAD>
<BODY>
<BIG>???J?R?l</BIG>
</BODY></HTML>
```

即ち、ヘッダ行は追加されないが、エンコーディングは正しく 8 バイトに変換されており、ブラウザ上でも正しく日本語表示される。

8.2.2.4　setContentType の効果

これは既にオリジナルの HelloWorld で確認されているとおりで、ヘッダ行として

```
Content-Type: text/html;charset=Windows-31J
```

がセットされているとともに、指定された文字セットでエンコーディングがされている。

問題は、getWriter を呼んだあとでこのメソッドを呼んでも効果がないということである。この行を getWriter の行の後に移してみると、次のようになる:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 90
Date: Tue, 26 Oct 2010 04:48:19 GMT

<HTML>
<HEAD><TITLE>Hello World</TITLE></HEAD>
<BODY>
<BIG>????</BIG>
</BODY></HTML>
```

無論 Windows-31J のエンコーディングは適用されないが、面白いことに charset=ISO-8859-1 とデフォルトのエンコーディングと置き替えたヘッダがセットされることである。

8.2.2.5　setContentLength の効果

オリジナルの状態で、res.setContentLength(90);と本来の長さよりも 4 つ少ない値をセットしたらどうなるだろうか？

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=Windows-31J
Content-Length: 90
Date: Tue, 26 Oct 2010 05:20:41 GMT

<HTML>
<HEAD><TITLE>Hello World</TITLE></HEAD>
<BODY>
<BIG>???J?R?l</BIG>
</BODY></HTM
```

このように、実際の長さよりも **setContentLength で指定した値のほうが優先**され、中途半端な HTML テキストが送信される。それでは逆に実際よりも長い値をセットしたらどうなるだろうか？答えは Content-Length: 90 というヘッダがセットされるが、サーブレットを抜けるところで（あるいは明示的なフラッシュで）バッファがフラッシュされ

ることで正常な長さのテキストがクライアントに流される。

8.2.3 ボディ部のためのメソッドたち

ボディ部にデータをセットするにはテキスト・データに適した PrintWriter を使うか、バイナリ・データに適した ServletOutputStream を使うかする。ServletOutputStream というのは、OutputStream を継承したクラスであるが、テキストも容易に書き込めるように、PrintWriter の持っている print 及び println というメソッド群が用意されているのが特徴であるが、それは 1 バイト文字セットの ISO 8859-1 に限られる。従って日本語を扱うときは文字コード変換が必要である。

表 8-3:ボディ部のためのメソッドたち

メソッド	内容
getWriter	この応答にバイナリ・データを書き込むのに適した ServletOutputStream を返す。サーブレット・コンテナはバイナリ・データに対してはエンコードしない。 ServletOutputStream の flush()を呼ぶことでその応答はコミットされる。このメソッドまたは getWriter()のどちらかがボディ部書き込みに呼ばれ、双方共は呼ばれない。
getOutputStream	クライアントに文字テキストを送信できる PrintWriter オブジェクトを返す。この PrintWriter は getCharacterEncoding()で返される文字エンコーディングを使う。この応答の文字エンコーディングが未だ getCharacterEncoding で示したように指定されていないとき(言い換えれば getCharacterEncoding がデフォルトの ISO-8859-1 を単に返す場合)は、getWriter はそれを ISO-8859-1 とする。 PrintWriter の flush()を呼ぶことでその応答はコミットされる。このメソッドまたは getOutputStream()のどちらかがボディ部書き込みに呼ばれ、双方共は呼ばれない。
setCharacterEncoding	例えば UTF-8 のような、そのクライアントに送信される応答の文字エンコーディング(MIME charset)をセットする。setContentType(java.lang.String)あるいは setLocale(java.util.Locale)で既に文字エンコーディングがセットされているときには、このメソッドはそれを優越する。setContentType(java.lang.String)を text/html の String で呼び、このメソッドを UTF-8 の String で呼ぶことは、text/html; charset=UTF-8 の String で setContentType を呼ぶことと等価である。 このメソッドは文字エンコーディングを変えるために繰り返し呼ぶことができる。このメソッドは getWriter が呼ばれたあと、またはこの応答がコミットされた後では効果をもたらさない。
setContentLength	前項で既に説明済みだが、このメソッドは Content-Length ヘッダ行もセットする。 この応答のコンテント・ボディ部の長さをセットする。
setContentType	前項で既に説明済みだが、このメソッドはヘッダ行もセットする。 その応答がまだコミットされていないならば、そのクライアントに送られる応答のコンテント・タイプをセットする。与えられるコンテント・タイプには、例えば text/html;charset=UTF-8 のように文字エンコーディング指定が含まれていて良い。この応答の文字エンコーディングは、getWriter が呼ばれる前にこのメソッドが呼ばれているときにのみこのコンテント・タイプからセットされる。 このメソッドは文字エンコーディングを変えるために繰り返し呼ぶことができる。このメソッドは getWriter が呼ばれたあと、またはこの応答がコミットされた後では効果をもたらさない。 この応答がコミットされた後あるいは getWriter が呼ばれた後でこのメソッドが呼ばれているときは、この応答の文字エンコーディングをセットしない。
setBufferSize	その応答のボディのための好ましいバッファ・サイズをセットする。サーブレット・コンテナは少なくとも要求された大きさのバッファを使用する。使用されている実際のバッファ・サイズ

メソッド	内容
	は getBufferSize で知ることが出来る。バッファが大きければそれだけ多くのコンテントが実際に送信される前に書き込むことが出来るので、サーブレットはしかるべきステータス・コードとヘッダをセットする時間的余裕が大きくなる。小さなバッファだとサーバのメモリが少なくて済み、クライアントはより迅速に受信開始を始める。 このメソッドは何らなかの応答ボディが書き込まれる前に呼ばれなければならない。コンテントが既に書かれている、あるいはその応答オブジェクトが既にコミットされているときは、このメソッドは IllegalStateException をスローする。
flushBuffer	このバッファのコンテントをクライアントに書き込ませる。このメソッド呼び出しで応答は自動的にコミットされ、それはステータス・コードとヘッダたちが書き込まれることを意味する。
resetBuffer	ヘッダまたはステータス・コードをクリアすることなく、その応答内の内部で使われているバッファのコンテントをクリアする。この応答が既にコミットされているときは、このメソッドは IllegalStateException をスローする。
isCommited	その応答がコミットされているかどうかを示すブール値を返す。コミットされた応答では既にステータス・コードとヘッダたちが書き込まれてしまっている。
reset	ステータス・コードとヘッダたちを含めてそのバッファ内に存在するデータの総てをクリアする。この応答が既にコミットされているときは、このメソッドは IllegalStateException をスローする。

8.3節　エラー応答

クライアントに返すエラー応答（ここで言うエラー応答には必ずしもエラーを意味するものでないものも含まれるが）は、サーブレットあるいはフィルタがその要求中に発生させた（意図的な場合を含めて）例外がトリガとなってコンテナが送信する場合と、サーブレットまたはフィルタがその要求処理の過程で意図的に生成してコンテナにそれを送信させる場合がある。

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

これらの例外はコンテナがそれを受けて自分が持っているエラー・ページ（あるいは配備記述子で指定されているときはそのページ）をクライアントに返す。その場合はコンテナは ServletException.getRootCause メソッドで例外を抽出し表示する。

サーブレット仕様書では以下のように書かれている：

> 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。

8.3.1 sendError と setStatus

HTML でエラー情報をクライアントに返したいとき（指定されたファイルが存在しないとか、フォーム入力データに誤りがあるなど）は、まず setStatus メソッドを呼んだ後に PrintWriter にその HTML メッセージを書き込む。一方 sendError メソッドはこれを一度にやってくれる。このメソッドはこのメソッドで指定したメッセージ HTML 化し、404 や 403 などのエラー応答としてくれる。**仕様書上では 400 及び 500 番台のエラー・コードは sendError を、200 及び 300 番台のエラー・コードは setStatus を使うことが想定されている。**

8.3.1.1 実験のための HTML とサーブレット

以下にこの 2 つの方法を簡単な HTML ページとサーブレットで実験してみよう。実験に使うコードは：

- ステータス・コードと使用するメソッドを指定して上の ErrorCodeTestServlet サーブレットを呼ぶ ErrorCodeTest.html という HTML ファイル
- HTML ページから指定されたステータス・コードを指定されたメソッドでクライアントに返す ErrorCodeTestServlet というサーブレット

である。

ErrorCodeTest.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-31j">
<title>エラー・コードのテスト</title>
</head>
<body>
<form method="post" action="http://localhost:8080/tutorial/ErrorCodeTestServlet">
<select name = "select" size="5">
<option value = "SC_CONTINUE" >100: SC_CONTINUE</option>
<option value = "SC_SWITCHING_PROTOCOLS" >101: SC_SWITCHING_PROTOCOLS</option>
<option value = "SC_OK " >200: SC_OK</option>
<option value = "SC_CREATED" >201: SC_CREATED</option>
<option value = "SC_ACCEPTED" >202: SC_ACCEPTED</option>
<option value = "SC_NON_AUTHORITATIVE_INFORMATION" >203: SC_NON_AUTHORITATIVE_INFORMATION</option>
<option value = "SC_NO_CONTENT" >204: SC_NO_CONTENT</option>
<option value = "SC_RESET_CONTENT" >205: SC_RESET_CONTENT</option>
<option value = "SC_PARTIAL_CONTENT" >206: SC_PARTIAL_CONTENT</option>
<option value = "SC_MULTIPLE_CHOICES" >300: SC_MULTIPLE_CHOICES</option>
<option value = "SC_MOVED_PERMANENTLY" >301: SC_MOVED_PERMANENTLY</option>
<option value = "SC_FOUND" >302: SC_FOUND</option>
<option value = "SC_MOVED_TEMPORARILY" >302: SC_MOVED_TEMPORARILY</option>
<option value = "SC_SEE_OTHER" >303: SC_SEE_OTHER</option>
<option value = "SC_NOT_MODIFIED" >304: SC_NOT_MODIFIED</option>
<option value = "SC_USE_PROXY" >305: SC_USE_PROXY</option>
<option value = "SC_TEMPORARY_REDIRECT" >307: SC_TEMPORARY_REDIRECT</option>
<option value = "SC_BAD_REQUEST" >400: SC_BAD_REQUEST</option>
<option value = "SC_UNAUTHORIZED" >401: SC_UNAUTHORIZED</option>
<option value = "SC_PAYMENT_REQUIRED" >402: SC_PAYMENT_REQUIRED</option>
<option value = "SC_FORBIDDEN" >403: SC_FORBIDDEN</option>
<option value = "SC_NOT_FOUND" >404: SC_NOT_FOUND</option>
<option value = "SC_METHOD_NOT_ALLOWED" >405: SC_METHOD_NOT_ALLOWED</option>
<option value = "SC_NOT_ACCEPTABLE" >406: SC_NOT_ACCEPTABLE</option>
<option value = "SC_PROXY_AUTHENTICATION_REQUIRED" >407: SC_PROXY_AUTHENTICATION_REQUIRED</option>
<option value = "SC_REQUEST_TIMEOUT" >408: SC_REQUEST_TIMEOUT</option>
<option value = "SC_CONFLICT" >409: SC_CONFLICT</option>
<option value = "SC_GONE" >410: SC_GONE</option>
<option value = "SC_LENGTH_REQUIRED" >411: SC_LENGTH_REQUIRED</option>
<option value = "SC_PRECONDITION_FAILED" >412: SC_PRECONDITION_FAILED</option>
<option value = "SC_REQUEST_ENTITY_TOO_LARGE" >413: SC_REQUEST_ENTITY_TOO_LARGE</option>
<option value = "SC_REQUEST_URI_TOO_LONG" >414: SC_REQUEST_URI_TOO_LONG</option>
<option value = "SC_UNSUPPORTED_MEDIA_TYPE" >415: SC_UNSUPPORTED_MEDIA_TYPE</option>
<option value = "SC_REQUESTED_RANGE_NOT_SATISFIABLE" >416:
SC_REQUESTED_RANGE_NOT_SATISFIABLE</option>
```

```
<option value = "SC_EXPECTATION_FAILED" >417: SC_EXPECTATION_FAILED</option>
<option value = "SC_INTERNAL_SERVER_ERROR" >500: SC_INTERNAL_SERVER_ERROR</option>
<option value = "SC_NOT_IMPLEMENTED" >501: SC_NOT_IMPLEMENTED</option>
<option value = "SC_BAD_GATEWAY" >502: SC_BAD_GATEWAY</option>
<option value = "SC_SERVICE_UNAVAILABLE" >503: SC_SERVICE_UNAVAILABLE</option>
<option value = "SC_GATEWAY_TIMEOUT" >504: SC_GATEWAY_TIMEOUT</option>
<option value = "SC_HTTP_VERSION_NOT_SUPPORTED" >505: SC_HTTP_VERSION_NOT_SUPPORTED</option>
</select>
<p>
<input type="radio"name="applyMethod" value="sendErr1"checked="checked"/>Use sendError(sc)
<input type="radio"name="applyMethod" value="sendErr2"/>Use sendError(sc, msg)
<input type="radio"name="applyMethod" value="setStatus"/>Use setStatus(sc)
<br><br>
<input type="submit" value="Submit">
</p>
</form>
</body>
</html>
```

このHTMLファイルは、Eclipseのプロジェクト・エクスプローラ上でtutorial\WebContentのフォルダ内に作成する。

ErrorCodeTestServlet.java

```
package basic_package;

import java.io.IOException;
import java.io.PrintWriter;
import java.lang.reflect.Field;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/ErrorCodeTestServlet")
public class ErrorCodeTestServlet extends HttpServlet
{

   private static final long serialVersionUID = 1L;

   @Override
   public String getServletInfo() {
      return "ErrorCodeTestServlet, Version 1.0, Sept. 2010, by Cresc"; }

   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException{
      performTask(req,res); }

   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res); }

   public void performTask(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
   {
      String requestedErrorCode = req.getParameter("select");
      try {
         // get the Field
         Field theField = HttpServletResponse.class.getField(requestedErrorCode); // the Field
         // convert to status code in int value
         int sc = 0;
         sc = theField.getInt(sc);
         // generate the output message
         String msg = "ErrorCodeTestServlet からのメッセージです.." + sc + ": " + requestedErrorCode;
         String applyMethod = req.getParameter("applyMethod");
/**         // this portion is for debugging
         System.out.println("field value : " + theField.toString());
         System.out.println("sc value : " + sc);
         System.out.println("message text : " + msg);
         System.out.println("applyMethod : " + applyMethod);
*/
         // generate HTTP response message according to the applyMethod
```

```
        res.setContentType("text/html; charset=windows-31j");
        if (applyMethod.equals("setStatus")){
           res.setStatus(sc);
        }
        else if(applyMethod.equals("sendErr1")) {
           res.sendError(sc);
        }
        else if(applyMethod.equals("sendErr2")) {
           res.sendError(sc, msg);
        }
        PrintWriter out = res.getWriter();
        out.println("<HTML><head><title>");
        out.println("エラー・コードのテスト");
        out.println("</title></head>");
        out.println("<body><h1>エラー・コードのテスト</h1><pre>");
        out.println("選択されたステータス・コード：  " + requestedErrorCode) ;
        out.println("選択されたステータス・コードの番号：  " + sc);
        out.println("選択されたメソッド：  " + applyMethod);
        out.println("</pre></body></HTML>");
        out.flush();
        out.close();
     } catch (Exception e) {
        System.out.println("performTask error occured: " + e.toString());
        e.printStackTrace();
     }
   }
}
```

このサーブレットは tutorial\src\basic_package のなかに作成する。

8.3.1.2　簡単な実験例

この実験では、最初に C:\eclipse\workspace\tutorial\WebContent\ErrorCodeTest.html をブラウザで開くと、以下のような画面が表示される:

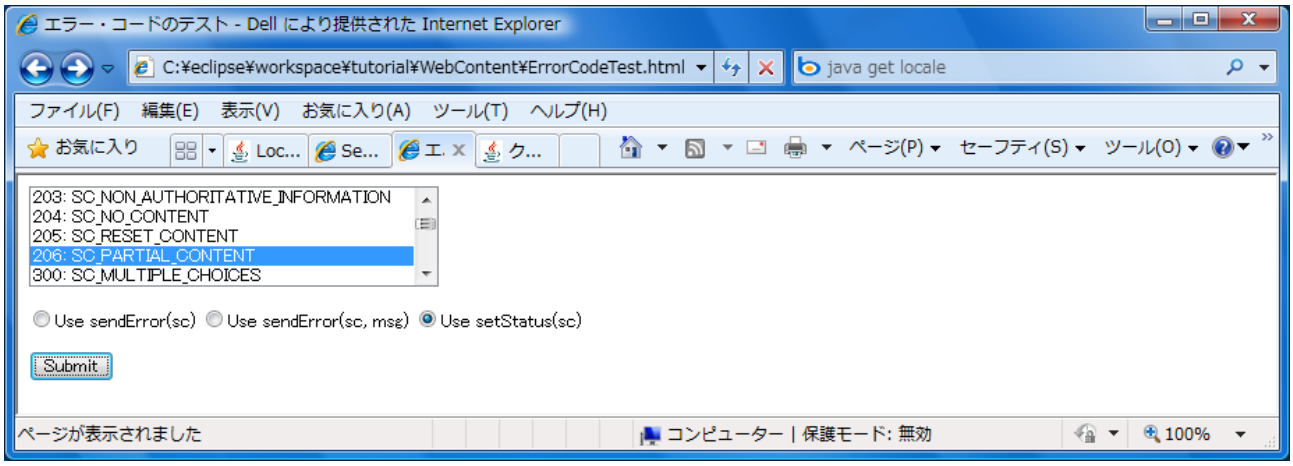


図 8-5 : ErrorCodeTest.html を開く

選択メニューでは HttpServletResponse にあるステータス・コードの総てからひとつを選択できるようになっている。その下にはどのメソッドを使うのかを選択するラジオ・ボタンがある。ステータス・コードと使用メソッドを選択したら Submit ボタンを押せば、この要求が ErrorCodeTestServlet に送信される。下図は SC_PARTIAL_CONTENT を setStatus を使うよう要求したときの応答である。

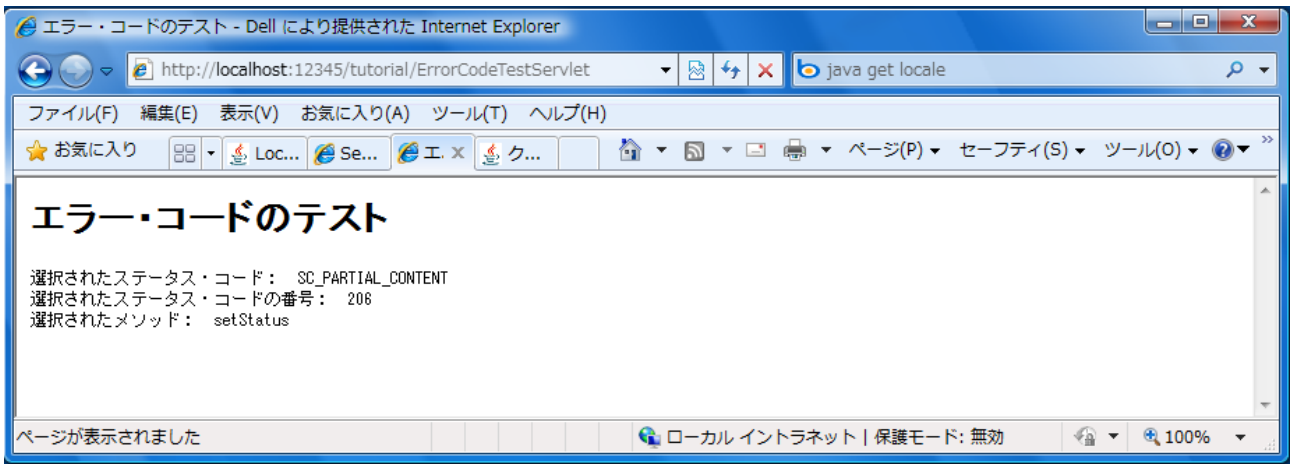


図 8-6：実行結果

この場合にはステータス行には SC_PARTIAL_CONTENT がセットされ、ボディ部分にはサーブレットが作成した HTML テキストがセットされて、クライアントに返される。本当にそのように返されているのだろうか？これも MINA プロキシのプログラムを使えば確認することが出来る。ErrorCodeTest.html の FORM タグの ACTION の URL のポート番号を 8080 から 12345 に変更して、MINA プロキシを走らせて、同じように送信すると、MINA プロキシは以下のような応答メッセージをログ出力する：

MINA プロキシが報告した応答メッセージ

```
HTTP/1.1 206 Partial Content
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=windows-31j
Transfer-Encoding: chunked
Date: Tue, 26 Oct 2010 07:39:37 GMT

100
<HTML><head><title>
?G???[?E?R?[?h???e?X?g
</title></head>
<body><h1>?G???[?E?R?[?h???e?X?g</h1><pre>
?I?????????X?e?[?^?X?E?R?[?h?F?@SC_PARTIAL_CONTENT
?I?????????X?e?[?^?X?E?R?[?h???????F?@206
?I??????????\?b?h?F?@setStatus
</pre></body></HTML>

0
```

このように、ステータス行は HTTP/1.1 206 Partial Content と正しく設定されている。ボディ部分はチャンクド形式になっている。このステータス行の場合は、ブラウザは応答にある HTML テキストをそのまま表示する。

このツールを使えば、開発者はあるステータス応答をしたいときに、それがコンテナがどう理解し、また各社のブラウザがどう対応するかを確認することが出来る。

8.3.2 100 番台のステータス応答

1xx のステータス・コードは情報(Informational)というカテゴリになり、要求を受信し、プロセス継続中である状態で出されるものである。

1. 100 (continue)： この場合はどのメソッドを使ってこの応答を送っても、ブラウザは次の応答を待ち続ける。タイムアウトでそのブラウザはその応答のボディ部を表示する(Internet Explorer の場合)か、何も表示しない(Chrome、Firefox の場合)か、ページが開けないと表示(Safari の場合)する。
2. 101 (switching protocols)： これも 100 応答と同じである。

8.3.3　200 番台のステータス応答

2xx のステータス・コードは成功(Success)というカテゴリになり、そのアクションは正しく受信し、理解し、受け付けた状態で出されるものである。

仕様書上では 400 及び 500 番台のエラー・コードは sendError を、200 及び 300 番台のエラー・コードは setStatus を使うことが想定されている。

1. 200 (OK)：これは、どういうわけだか java.lang.NoSuchFieldException: SC_OK がスローされる。クライアントにはボディ部分が空のメッセージが返される。従って、このステータス・コードはサーブレットからは使用できない。
2. 201 (created)：setStatus を使うと正常な HTML テキストがボディ部にセットされる。sendError メソッドを使うとクライアントにはボディ部分が空のメッセージが返される。
3. 202 (accepted)：201 応答と同じような結果になる。
4. 203 (non-authoritable information)：201 応答と同じような結果になる。
5. 204 (no content)：201 応答と同じような結果になる。
6. 205 (reset content)：どのメソッドを使ってもクライアントにはボディ部分が空のメッセージが返される。ブラウザはもとの URL 画面のままになる。
7. 206 (partial content)：201 応答と同じような結果になる。

8.3.4　300 番台のステータス応答

3xx のステータス・コードはリダイレクション(Redirection)のカテゴリになり、その要求を完了させるには更なるアクションが必要な状態で出されるものである。

1. 300 (multiple choices)：201 応答と同じような結果になる。setStatus を使うと正常な HTML テキストがボディ部にセットされる。SendError メソッドを使うとクライアントにはボディ部分が空のメッセージが返される。
2. 301 (moved permanently)：この場合は sendError ではクライアントにはボディ部分が空のメッセージが返され、setStatus では所定の HTML テキストがボディ部に入る。ブラウザは、Internet Explorer の場合はどのメソッドでも「Internet Explorer ではこのページは表示できません」と表示されるが、Firefox、Chrome、及び Safari では sendError では画面には何も表示されず、setStatus では所定の HTML テキストが表示される。
3. 302 (found)：301 応答と同じような結果となる。
4. 303 (see other)：301 応答と同じような結果となる。
5. 304 (not modified)：この場合は sendError ではクライアントにはボディ部分が空のメッセージが返され、setStatus では所定の HTML テキストがボディ部に入る。ブラウザは、Internet Explorer、Firefox 及び Safari の場合は画面には何も表示されず、Chrome の場合は「ウェブページが見つかりません。」と表示される。
6. 305 (use proxy)：300 応答と同じような結果となる。
7. 307 (temporary redirect)：301 応答と同じような結果となる。

8.3.5　400 番台のステータス応答

4xx のステータス・コードはクライアント・エラー(Client Error)のカテゴリになり、その要求に文法上の誤りがあるか、その要求を満足させられない状態のときに出される。

400 番台の応答に対して、各社のブラウザの処理が異なる場合が多いので、注意が必要である。

1. 400 (bad request)： sendError では Tomcat が出したエラー・ページが出力される。setStatus では Internet Explorer では「Web ページが見つかりません」と表示され、Internet Explorer、Firefox 及び Safari では所定の HTML テキストが表示される。
2. 401 (unauthorized)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入る。どのブラウザも送信されてきた画面をそのまま表示する。
3. 402 (payment required)： 401 応答と同じような結果となる。
4. 403 (forbidden)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入る。ブラウザは Internet Explorer では sendError の場合は Tomcat が出したページを表示し、setStatus では「Web サイトによってこのページの表示を拒否されました」と表示される。Firefox、Chrome、及び Safari では、sendError の場合は Tomcat が出したページを表示し、setStatus ではサーブレットが作った HTML を表示する。
5. 404 (not found)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入る。ブラウザは Internet Explorer では sendError の場合は Tomcat が出したページを表示し、setStatus では「Web ページがみつかりません」と表示する。Firefox、及び Safari では、sendError の場合は Tomcat が出したページを表示し、setStatus ではサーブレットが作った HTML を表示する。Chrome では sendError の場合は Tomcat が出したページを表示し、setStatus では「エラー: このリンクは無効です。」と表示する。
6. 405 (method not allowed)： 401 応答と同じような結果となる。
7. 406 (not acceptable)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入る。ブラウザは、Internet Explorer では sendError の場合は Tomcat が出したページを表示し、setStatus では「Internet Explorer でこの Web ページの形式を読み取ることはできません」と表示する。Firefox、Chrome、及び Safari では、sendError の場合は Tomcat が出したページを表示し、setStatus ではサーブレットが作った HTML を表示する。
8. 407 (proxy authentication required)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入る。Internet Explorer、Firefox、及び Safari は受信した画面をそのまま表示するが、Chrome では setStatus の場合は「このウェブサイトはご利用いただけません。」と表示する。
9. 408 (request timeout)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入る。ブラウザは、Internet Explorer では sendError の場合は Tomcat が出したページを表示し、setStatus では「Web サイトはアクセス過多により Web ページを表示できません」と表示する。Firefox ではこのメッセージを受信したら再接続を試み、タイムアウトを起こして「接続がリセットされました」と表示する。Chrome と Safari では受信した画面をそのまま表示する。
10. 409 (conflict)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入る。ブラウザは、Internet Explorer では 408 応答と同様に、sendError の場合は Tomcat が出したページを表示し、setStatus では「Web サイトはアクセス過多により Web ページを表示できません」と表示する。Firefox、Chrome、及び Safari では受信した画面をそのまま表示する。
11. 410 (gone)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入る。ブラウザは、Internet Explorer では 408 応答と同様に、sendError の場合は Tomcat が出したページを表示し、setStatus では「Web ページは存在しません」と表示する。Firefox、Chrome、及び Safari では受信した画面をそのまま表示する。
12. 411 (length required)： 401 応答と同じような結果となる。
13. 412 (precondition failed)： 401 応答と同じような結果となる。
14. 413 (request entity too large)： 401 応答と同じような結果となる。
15. 414 (requested URI too long)： 401 応答と同じような結果となる。
16. 415 (unsupported media type)： 401 応答と同じような結果となる。

17. 416 (requested range not satisfiable)： 401 応答と同じような結果となる。
18. 417 (expectation failed)： 401 応答と同じような結果となる。

8.3.6 500 番台のステータス応答

5xx のステータス・コードはサーバ・エラー(Server Error)のカテゴリになり、明らかに有効な要求だが、満足させることに失敗した状態のときに出される。

1. 500 (internal server error)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入って送信される。ブラウザは、Internet Explorer では sendError の場合は Tomcat が出したページを表示し、setStatus では「Web サイト側でページを表示できません」と表示する。Firefox、Chrome、及び Safari では受信した画面をそのまま表示する。
2. 501 (not implemented)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入って送信される。ブラウザは、Internet Explorer では sendError の場合は Tomcat が出したページを表示し、setStatus では「Web サイトで Web ページを表示できません」と表示する。Firefox、Chrome、及び Safari では受信した画面をそのまま表示する。
3. 502 (bad gateway)： sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入って送信される。ブラウザ側では、Internet Explorer、Firefox、Chrome、及び Safari ともに受信した画面をそのまま表示する。
4. 503 (service unavailable)： 502 応答と同じような結果になる。
5. 504 (gateway timeout)： 502 応答と同じような結果になる。
6. 505 (HTTP version not supported)： 501 応答と同様、sendError ではクライアントには Tomcat が出したエラー・ページが送信され、setStatus では所定の HTML テキストがボディ部に入って送信される。ブラウザは、Internet Explorer では sendError の場合は Tomcat が出したページを表示し、setStatus では「Web サイトで Web ページを表示できません」と表示する。Firefox、Chrome、及び Safari では受信した画面をそのまま表示する。

8.4節 リダイレクション(Redirection)

リダイレクションとは、HTTP 応答メッセージに 300 番台のステータス・コードと、Location ヘッダを付してクライアントに返し、その応答を Location で示した URL に送ることを指示することである。「要求のフォワードとインクルード」の節で示すサーブレットからサーブレットに要求を引き継ぐ方法と違って、この方法は HTTP 応答のステータス・コードで引き継ぎをブラウザに指示している。従って、ブラウザ経由で転送先のサーブレットに渡せる情報は制限される。**渡せる情報は Location ヘッダ行経由でのみ**であり、以下のものとなる：

- 要求 URI
- クエリ文字列
- URL 書き換えによるセッション ID

8.4.1 sendRedirect メソッド

リダイレクションは、setStatus メソッドでステータス・コードをセットし、setHeader で Location ヘッダをセットすることで実現できるが、HttpServletRequest には sendRedirect メソッドが用意されている。このメソッドは、指定されたリダ

イレクト先 URL を使ってそのクライアントに一時的リダイレクト応答を送信し、バッファをクリアする。バッファはこのメソッドがセットしたデータで置き換えられる。このメソッドを呼ぶとステータス・コードは SC_FOUND 302 (Found)にセットされる。このメソッドは**相対 URL も**受け付ける;サーブレット・コンテナはその応答をクライアントに送信する前にその相対 URL を絶対 URL に変換しなければならない。もし引数の location が'/'で始まっていないときは、コンテナはそれは現在の要求 URI に相対だと解釈する。引数の location が'/'で始まっているときは、コンテナはそれはサーブレット・コンテナのルートに対し相対だと解釈する。このメソッドは、応答が既にコミットされているときは IOException 例外をスローする。

8.4.2 RedirectionTest サーブレットによる確認

それでは実際にこのメソッドを使ったリダイレクションで、クライアントとの間でどのようなメッセージが交換されるのだろうか？

以下はそのテストの為の簡単なサーブレットのコード(処理部分のみ)である。

```
/**
 * RedirectionTest は、リダイレクト応答を返すテストの為の簡単なサーブレット
 * December 2010, CRESC Corps.
 */

@WebServlet("/RedirectionTest")
public class RedirectionTest extends HttpServlet {
   private static final long serialVersionUID = 1L;

   public void performTask(HttpServletRequest req, HttpServletResponse res){
      req.getSession();
      String url = "/tutorial/HttpRequestDump";
      if (req.getQueryString() != null) url = url + "?" + req.getQueryString();
//    url = "http://localhost:8080" + url;  // 絶対 URL 指定の実験
      try {
         res.sendRedirect(res.encodeRedirectURL(url));
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
```

このサーブレットは HttpRequestDump サーブレットにリダイレクトするものであるが:

- 相対 URL に加えて、絶対 URL で sendRedirect メソッドを呼ぶ実験が出来る
- 要求にクエリ文字列があったら、それを Location に含める
- Cookie に対応しないブラウザにも対応できるよう、URL 書き換えを行う(URL 書き換えは「セッション」の章で述べる)

このコードでは GET 要求による要求パラメタをセットできるが、POST 要求による要求パラメタを URL 書き換えでセットすることもできよう。従って**このサーブレットは、クエリ経由で自分が持っている簡単な情報を要求パラメタとしてリダイレクト先のサーブレットに渡すことも可能**である。

このサーブレットを Tera Term でアクセスすると次のような応答を返していることが判る:

```
GET /tutorial/RedirectionTest?name=aaa HTTP/1.1
host: localhost:8080


HTTP/1.1 302 Moved Temporarily
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=119F4D1597D9D37AFC7F42B6555528F6; Path=/tutorial; HttpOnly
Location:
http://localhost:8080/tutorial/HttpRequestDump;jsessionid=119F4D1597D9D37AFC7F42B6555528F6?name=aaa
Content-Length: 0
Date: Tue, 28 Dec 2010 04:59:44 GMT
```

- 最初の 3 行（空白行も含む）は送信データである
- 応答メッセージのステータス行は HTTP/1.1 302 Moved Temporarily と 302 を返している
- Location ヘッダ行が追加されており、これにはセッション ID が付加されている
- 更にこの行には、クエリ文字列で指定した要求パラメタが、クエリ文字列として追加されている

従って、HttpRequestDump サーブレットは、
/tutorial/HttpRequestDump;jsessionid=119F4D1597D9D37AFC7F42B6555528F6?name=aaa として呼ばれる。

なお tutorial アプリケーションには、RedirectionTest サーブレットをアクセスする RedirectionTestForm.html というファイルが用意されている。http://localhost:8080/tutorial/RedirectionTestForm.html でこれを開き、GET 要求で正しくデータが HttpRequestDump サーブレットに渡されることを確認して頂きたい。以下はそのテキストである：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-31j">
<title>Insert title here</title>
</head>

<body>
<H1>サーブレットに要求パラメタとしてテキスト・ボックスの内容をわたす</H1>
<form method="post" action="http://localhost:8080/tutorial/RedirectionTest">
<textarea rows="4" cols="40" name="サブミット・データ"></textarea><br>
<input type="submit" value="Submit using POST">
</form>
<br>
<form method="get" action="http://localhost:8080/tutorial/RedirectionTest">
<textarea rows="4" cols="40" name="ゲット・データ"></textarea><br>
<input type="submit" value="Submit using GET">
</form>
</body>

</html>
```

必要ならポート番号を 12345 にし、プロキシを稼働させて、交信されている実際の HTTP メッセージを確認すると良い。

8.5節　　ボディ部への書き込み

「HTTP 応答メッセージの組み立てのためのメソッドたち」の節で示したように、応答オブジェクトから取得したボディ部分には PrintWriter あるいは ServletOutputStream を使ってボディ部にデータを書き込むことになる。ヘッダを含めて書き込まれたデータはサーブレット・コンテナが持っているバッファに蓄積される。これは先入れ先出し（FIFO: First In First Out）のバッファだと考えれば良い（実際はヘッダ部分は分離している）。このバッファの内容は、チャンク形式の場合（HTTP/1.1）は完全に蓄積が終わる前にネットワークに送出されてゆく。これを**コミット(commit)**された状態だという。即ちネットワークに出て行ってしまうので取り消しが効かないという意味である。このバッファの内容は、コミット前であれば、resetBuffer メソッドでボディ部を、reset メソッドでヘッダ部とボディ部総てをクリアすることが出来る。flushBuffer はヘッダ部を含めてバッファ内のデータ総てをネットワークに送出する。**デフォルトのバッファ・サイズは Tomcat では 8192 バイト**である。

一般的にはボディ部にデータを書くには次のような手順になる：

1. セッションを含めて必要なヘッダ行はセットしておく

2. PrintWriter を使うときは、SetContentType メソッドでボディ部の形式(text/html、URL エンコーディング、あるいはその他のエンコーディングなのか)と使用言語(Windows-31J、EUC-JP、UTF-8、あるいはその他の文字セットなのか)を指定する。
3. HTTP ボディ部の形式がテキストの場合は出力用に getWriter メソッドを使って PrintWriter のオブジェクトを、バイナリ・データの場合は getOutputStream メソッドを使って ServletOutputStream のオブジェクトを取得する。
4. 出力用バッファにコンテントを書き込む。
5. ServletOutputStream を使ってバイナリデータをセットする場合は、必要な場合は、setContentLength メソッドで HTTP ボディ部の長さを指定する。
6. 出力バッファ(その前に何らかのバッファを使っていればそれを含めて)のフラッシュとクローズを行う。但しこの作業は、スレッドがサーブレットから出る時に自動的になされるが、明示的に記すことは良い習慣である。

8.5.1 PrintWriter による書き込み

PrintWriter を使って HTML テキストをバッファに書き込むのは極めて簡単であるのは、これまで見てきた幾つかのサーブレットのコードを見ればお分りだろう。下図はその仕組みをまとめたものである:

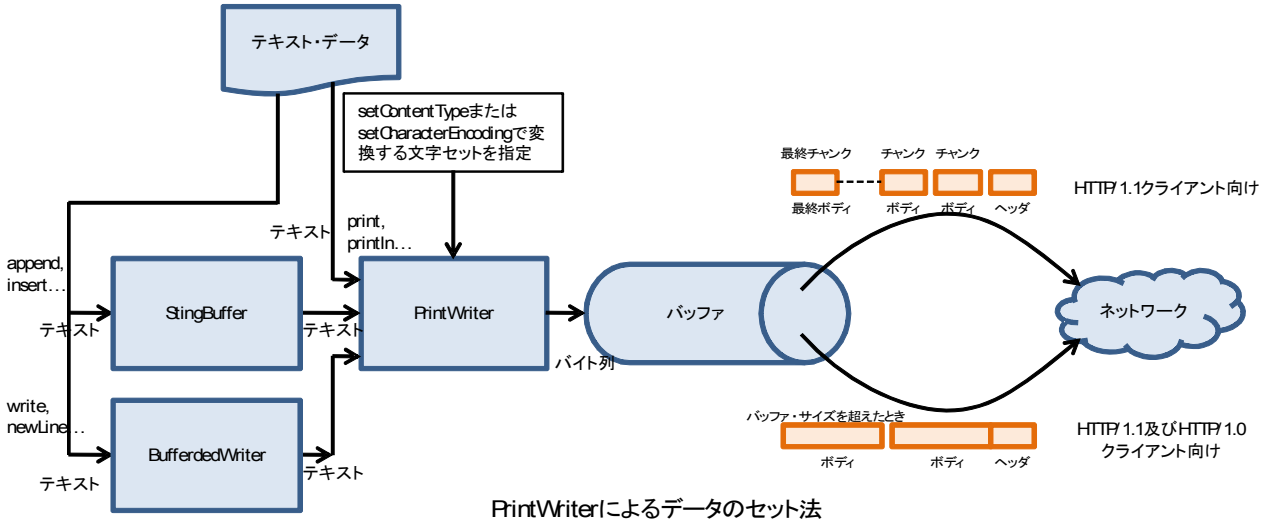


図 8-7:PrintWriter を使ったテキスト・データの応答のし方

サーブレット・エンジンはネットワークに HTTP 応答を送信する為の応答ごとのバイト列のバッファを持っている。その内容を必要に応じチャンクとして、あるいはひとつまたは複数の TCP パケットからなるメッセージとして送信するが、それはサーブレット側からは見ることはできない。但しネットワークに送信され始めたというコミット情報は得られる。テキスト・データからバイト列に変換するのが PrintWriter である。応答オブジェクトから取得したこの PrintWriter は、setContentType あるいは setCharacterEncoding でセットされた文字セットとして Unicode のテキストをバイト列に変換する。PrintWriter は print、println などのメソッドでテキストを書き込むが、その都度コンバータを通してバッファにセットするので効率が悪い。その為、後述のように StringBuffer や BufferedWriter を使う手段もある。この場合は、StringBuffer では append、BufferedWriter では write などのメソッドでテキストを書き込むことになる。以下その詳細を記す。

8.5.1.1 文字エンコーディング

ServletResponse.getWriter メソッドで得られる PrintWriter は、ServletResponse.setContentType メソッドであるいは

setCharacterrEncoding メソッドでセットした文字セットでバイト列に変換してバッファに書き込む。文字セットを指定しないと、デフォルトの 1 バイト文字セットの ISO 8859-1 の文字列だとみなされ、そのままバイト列としてバッファに送られる。従って日本語のような 2 バイト文字を書き込もうとすると、それは 2 つの文字として扱われる。従って、**日本語の場合は setContentType あるいは setCharacterEncoding で文字セットを指定することが必須である。**

なお、setContenType では以下の事項を指定することになる:

- ボディ部の形式:text/html、text/plain、あるいはその他のエンコーディング。**ブラウザはこのメソッドでセットされた Content-Type ヘッダ行を見で、そのコンテントをどう処理するか判断する。**どのエンコーディングを受け付けるかはブラウザに依存する。殆どの場合は text/html が使われる。
- 使用言語:Windows-31J、EUC-JP、UTF-8 (Java の内部文字セットの Unicode をバイト列に変換する一般的な手段)、あるいはその他の文字セット。**Windows-31J でも UTF-8 でもブラウザは対応するが、UTF-8 だと日本語文字が多い場合はデータ量が大きくなる。**

setContentType メソッドと setCharacterEncoding メソッドで異なる文字セットを指定したらどうなるか？この場合は setCharacterEncoding で指定した文字セットが優越する。またどちらのメソッドも getWriter を呼んだ後では無効である。

8.5.1.2　バッファとチャンク転送

比較的短い HTML テキストを送信する場合に、HTTP/1.1 および HTTP/1.0 クライアントに対しどのような応答メッセージを送っているのだろうか？

- バッファ・サイズ以内のデータ量では、チャンクではなく、Content-Length: 367 などと **Content-Length ヘッダを付けて**送信する。

HTML テキストの最後まで行かないうちに ServletResponse.flushBuffer をかけたらどうなるだろうか？

- HTTP/1.0 クライアントに対しては、それまでの内容をひとつの TCP メッセージとしてネットワークに送信し、isCommitted は true になり、その後入力したデータは TCP 接続を維持したまま、別の TCP メッセージとしてネットワークに送信される。終了後 TCP 接続が切れる。**Content-Length ヘッダは付けられない。**
- HTTP/1.1 クライアントに対しては、それまでの内容をチャンクとしてネットワークに送信し、isCommitted は true になり、その後入力したデータは別のチャンクとしてネットワークに送信される。

いずれの場合にも、ブラウザは正常に入力された HTML 画面を表示する。

このバッファのデフォルト・サイズは Tomcat では 8192 バイトである。これを超えるデータを入力したらどうなるであろうか？

- HTTP/1.0 クライアントに対しては、途中からその内容をひとつの TCP メッセージとしてネットワークに送信開始数するとともに isCommitted は true になり、その後入力したデータは TCP 接続を維持したまま、別の TCP メッセージとしてネットワークに送信される。終了後 TCP 接続が切れる。**Content-Length ヘッダは付けられない。**
- HTTP/1.1 クライアントに対しては、16 進で 2000 (つまり 8192 バイト) のサイズのチャンクとして送信される。最初のチャンクが送られた時点で isCommitted は true になる。

いずれの場合にも、ブラウザは正常に入力された HTML 画面を表示する。

これらのテスト結果は、HelloWorld のコードに一部を:

```
        out.println("<HTML>");
        out.println("<HEAD><TITLE>Hello World</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("<BIG>バッファ・サイズを超えるデータのテスト</BIG>");
        for (int i=0; i<100; i++){
                out.println("<BR>" + i +" Recently, several young people have taken their own lives
after being bullied for being gay or perceived as being gay by their peers. The President joins the
nationwide campaign to put a stop to these tragedies."
```

```
);
                out.println("<br>isCommitted: "+res.isCommitted());
        }
        out.println("</BODY></HTML>");
        }
```

のように変換し、HTTP/1.0 あるいは HTTP/1.1 モードで Internet Explorer から http://localhost:12345/tutorial/HelloWorld と、MINA プロキシのプログラム経由でこのサーブレットをアクセスすると確認できる。

従ってプログラマは**バッファのサイズを心配する必要はあまり無い**と言えよう。

8.5.1.3 StringBuffer や BufferdWriter による効率化

PrintWriter では print あるいは println メソッドごとに文字セットのデコーダが呼び出され、バッファにセットするので効率が悪い。その為 StringBuffer には append メソッドを、BufferedWriter には write メソッド使ってテキストをため込んで、それを一括して PrintWriter に渡すほうが効率的である。但しその場合は println で改行文字を入れるようなことはできず、改行はその都度挿入しなければならない。**JSP コンテナが使っている JspWriter は BuuferedWriter と PrintWriter の機能を一部エミュレートしている。**

以下は HelloWorld で BufferedWriter を使う例である:

```
        res.setContentType("text/html; charset= Windows-31J");
        PrintWriter out = res.getWriter();
        BufferedWriter bw = new BufferedWriter(out);
        bw.write("<HTML><HEAD><TITLE>Hello World</TITLE></HEAD>");
        bw.write("<BODY><BIG>Hello from 与謝野 馨</BIG>");
        bw.write("</BODY></HTML>");
        bw.close();    // flush and close the BufferdWriter
        }
```

BufferWriter のデフォルトのサイズは 8192 文字である。コンストラクタでサイズを指定できるが、しかしそれは気にする必要がなく、**バッファのサイズを超えた場合は自動的にフラッシュされる。**心配なひとは小さなサイズを指定して実験されると良い。

以下は HelloWorld で StringBuffer を使う例である:

```
        res.setContentType("text/html; charset= Windows-31J");
        PrintWriter out = res.getWriter();
        StringBuffer sb = new StringBuffer();
        sb.append("<HTML><HEAD><TITLE>Hello World</TITLE></HEAD>");
        sb.append("<BODY><BIG>Hello from 与謝野 馨</BIG>");
        sb.append("</BODY></HTML>");
        out.println(new String(sb));
        out.close();   // flush and close the PrintWriter
        }
```

StringBuffer はサイズを指定しないと 16 文字のサイズになるが、これは自動的に拡大される。どこまで蓄積可能かはメモリ・サイズに依存するが、通常は数 100kB でも問題にならない。

8.5.1.4 バッファリングを使うかどうか

例えば StringBuffer に 100kB の文字をためてそれを PrintWriter に渡しても、そのようなサイズだとネットワークの帯域の制限を受けることになる。一般には print あるいは println を頻繁に使う中規模のコンテント、あるいは同時アクセスのクライアント数が大きいアプリケーションではバッファリングは有効であろう。

もうひとつ検討しなければならないのは、チャンク転送の目的のひとつは、ページ全部のデータを分割して送ることで、ブラウザ側での処理を早めることである。従ってコンテナがチャンク化するよりも大きなデータ量を蓄積す

るのはその目的に沿わなくなる。このことは、サーブレットの処理が JDBC 接続待ちだとか、外部ウェブ・サービスからの応答を待つなどの待ち時間が発生する場合には、特に注意しなければならない。

もっと大きなコンテントの場合は、ネットワークの帯域をカバーする為に、後で説明する圧縮して送信する方法がある。

8.5.2 ServletOutputStream による書き込み

HttpOutputStream を使ってテキストをバッファにセットする場合は、文字エンコーディングはサーブレット側の責任になる。従って文字コンバート機能を持った PrintWriter や OutputStreamWriter が必要になるし、効率を上げたい場合は更に BufferedWriter のようなバッファが必要になる。

一方バイト・データをセットする場合は、ServletOutputStream に直接バイト列を書き込んで構わない。但し HTTP/1.0 クライアントには正しいコンテント・サイズを教えてやるために、ByteArrayOutputStream が必要になる。下図はその流れの概要を示したものである。

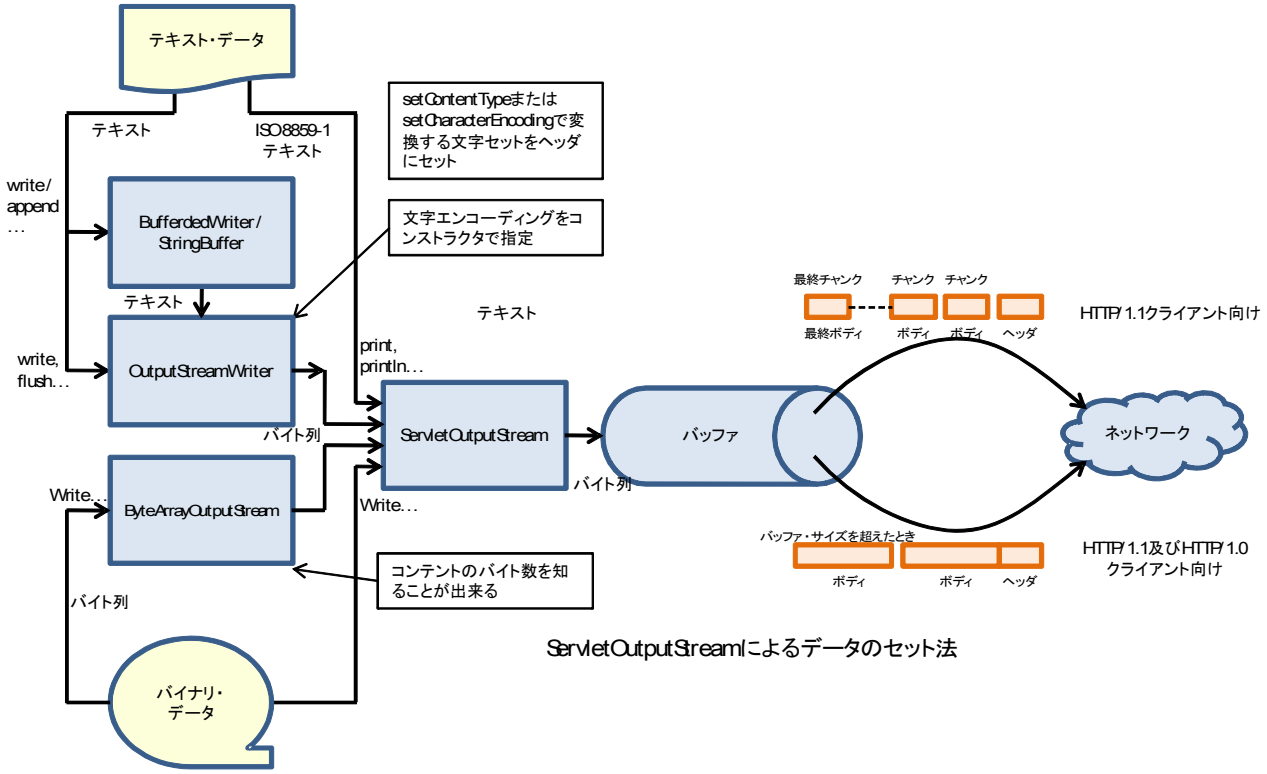


図 8-8 : ServletOutputStream を使ったデータのセット法

8.5.2.1 ServletOutputStream とは

ServletOutputStream はバイト列を入力する為の OutputStream である。ServletOutputStream は OutputStream を継承しているが、print 及び println という PrintWriter のメソッドが更に用意されている。これは PrintWriter のために書かれたコードを ServletOutputStream 用に簡単に移せるという点では便利なものではあるが、残念ながらこれらのメソッドは Java が使っている 2 バイトの Unicode の中で上のバイトがゼロである文字、即ち ISO 8859-1 以外は受け付けない。ちなみに HelloWorld サーブレットで PrintWriter out = res.getWriter();の行を ServletOutputStream out = res.getOutputStream();に変更し、out.println("<BIG>Hello from 与謝野 馨</BIG>"); などと日本語を入れると、Tomcat は次のようなエラー・ページを返す:

```
HTTP ステータス 500 -
type 例外レポート
メッセージ
説明 The server encountered an internal error () that prevented it from fulfilling this request.
例外
java.io.CharConversionException: ISO 8859-1 の文字ではありません: 与
        javax.servlet.ServletOutputStream.print(ServletOutputStream.java:77)
        javax.servlet.ServletOutputStream.println(ServletOutputStream.java:187)
        basic_package.HelloWorld.performTask(HelloWorld.java:66)
        basic_package.HelloWorld.doGet(HelloWorld.java:26)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:621)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
注意 原因のすべてのスタックトレースは、Apache Tomcat/7.0.2 のログに記録されています
Apache Tomcat/7.0.2
```

ちなみに英文だけのテキストだと例外は発生せず、正常な画面が表示される。従って我々のような **2 バイト文字を使っている場合には、この ServletOutputStream にテキストを入力することはできない**。また、**次のように PrintWriter を取得できるが、エンコーディングはやはりデフォルトのコードになり、それなら最初から PrintWriter を応答から取得するほうが良い**。

```
new PrintWriter(res.getOutputStream(),true);
```

8.5.2.2 テキスト・データの書き込み

日本語のテキストを書き込むにはテキストから文字セット変換してバイト列に変換するものが必要であり、OutputStreamWriter がそれに適している。OutputStreamWriter のコンストラクタには文字セットが指定できる。Javadoc は次のように書いている:

> OutputStreamWriter は、文字ストリームからバイトストリームへの橋渡しの役目を持ちます。バイトストリームに書き込まれた文字は、指定された charset を使用してバイトに符号化されます。使用される文字セットは、名前で指定することも、明示的に渡すことも、またはプラットフォームのデフォルトの文字セットをそのまま使うこともできます。
> write()メソッドを呼び出すたびに、指定された文字に対してエンコーディングコンバータが呼び出されます。結果として得られるバイトは、バッファーに蓄積されてから基本となる出力ストリームに書き込まれます。このバッファーのサイズは指定できますが、ほとんどの場合、デフォルトのサイズで十分です。write()メソッドに渡される文字はバッファーに入らないので注意してください。
> 最大限に効率化するには、コンバータを頻繁に呼び出さないようにするために BufferedWriter の内部に OutputStreamWriter をラップすることを考慮してください。例を示します。
> Writer out = new BufferedWriter(new OutputStreamWriter(System.out));

従って、その都度コード・コンバータを呼ばなくして効率を上げるために、BufferedWriter でラップするとか、StringBuffer に一度ため込むことが好ましい。

OutputStreamWriter だけで済ました場合の HelloWorld.java のコードの書き込み部分は次のようになる:

```
        res.setContentType("text/html; charset= Windows-31J");
        ServletOutputStream sos = res.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter( sos, "Windows-31J");
        osw.write("<HTML>");
        osw.write("<HEAD><TITLE>Hello World</TITLE></HEAD>");
        osw.write("<BODY>");
        osw.write("<BIG>Hello from 与謝野 馨</BIG>");
        osw.write("</BODY></HTML>");
        osw.close();
```

BufferedWriter でラップした場合の HelloWorld.java のコードの書き込み部分は次のようになる:

```
        res.setContentType("text/html; charset= Windows-31J");
        ServletOutputStream sos = res.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter( sos, "Windows-31J");
```

```
        BufferedWriter bw = new BufferedWriter(osw);
        bw.write("<HTML>");
        bw.write("<HEAD><TITLE>Hello World</TITLE></HEAD>");
        bw.write("<BODY>");
        bw.write("<BIG>Hello from 与謝野 馨</BIG>");
        bw.write("</BODY></HTML>");
        bw.close();
        osw.close();
```

テキストから文字変換して ServletOutputStream にバイト列を渡すには、java 1.4 から導入されているより高度な処理が出来る java.nio.CharBuffer、java.nio.charset.CharsetEncoder、及び java.nio.ByteBuffer を使うことも可能である。但しこれはやや複雑になるので説明しない。

8.5.2.3　バイト・データの書き込み

PrintWriter 経由であろうと HttpOutputStream 経由であろうと、HTTP/1.1 クライアント向けのチャンクしないデータ量の HTTP 応答メッセージの Content-Length ヘッダは、コンテナが判断してセットする。チャンクで送る場合は Content-Length ヘッダはつかない。問題は HTTP/1.0 クライアントである。この場合は、バッファ・サイズを超える場合は、コンテナは長さを知ることが出来ない。従って Content-Length ヘッダはサーブレットがセットしてやらねばならない。java.io.ByteArrayOutputStream はそのような目的に適している。この出力ストリームはデータが書き込まれるに従って、バッファーは自動的に大きくなってゆく。size メソッドで現在のバイト数を知ることが出来るし、writeTo(OutputStream out)メソッドでこれを一括 OutputStream に吐き出すことが出来る。

現在は HTTP/1.0 に設定されたブラウザは極めてまれになっているので、サーブレットは HTTP/1.0 クライアントには対応しないのもひとつの解決策である。

8.6節　　ブラウザにバイナリ・データを送る

前項で示した java.io.ByteArrayOutputStream を使った具体的な例を示す。

8.6.1　ブラウザに各種ファイルを送る

サーブレットからファイルをクライアントに送るプログラムを示す。これは Eclipse 上で、tutorial/WebContent のフォルダに置いた jpegimage.jpg というイメージ・ファイルをこの SendJpegBinary というサーブレットがクライアントに送信するというものである。

このプログラムでは、Internet Explorer のようにブラウザが image/jpeg を Accept ヘッダ行に含めているかをチェックしているが、**実際はどのブラウザでも受け付けるので、if(true)のコメントアウトされた行を活かして、その上の行をコメントアウトすれば、いろんなブラウザで確認できよう。**

このプログラムでは、HTTP/1.0 クライアントにも対応するよう res.setContentLength(baos.size());という行で Content-Length ヘッダを付加している。これを付けないと Tomcat は HTTP/1.1 クライアント向けとしてチャンク形式でクライアントに送信する。

なお、この場合は何も ByteArrayOutputStream を使わなくても sos.write(bytearray);のように、そのままバイト配列から直接 ServletOutputStream にファイル・データを一括送りこめ、またバイト配列はファイルのバイト数であるの

でその前に res.setContentLength(bytearray.length);をセットできる。しかしながら今回は後で説明する圧縮操作を前提にしているので、このような構成になっている。

```
package basic_package;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/SendJpegBinary")
public class SendJpegBinary extends HttpServlet
{

   private static final long serialVersionUID = 1L;

   @Override
   public String getServletInfo() {
      return "SendJpegBinary servlet, Version 1.0, Oct. 2010, by Cresc"; }

   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException{
      performTask(req,res); }

   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res); }

   public void performTask(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
   {
      String accept = req.getHeader("accept");
      if((accept != null) && (accept.toLowerCase().indexOf("image/jpeg") != -1))
//    if(true)
      {
         ServletOutputStream sos = res.getOutputStream();
         ByteArrayOutputStream baos = new ByteArrayOutputStream();
         ServletContext context = getServletContext();
         String fname = context.getRealPath(req.getRequestURI());
         fname = fname.replace("\\tutorial\\SendJpegBinary", "\\jpegimage.jpg") ;
         FileInputStream fis = null;
         try {
            fis = new FileInputStream(fname);
         }catch (IOException noSuchFile){
            throw new FileNotFoundException("No such file found");
         }
         File file = new File(fname);
         byte bytearray[] = new byte[(int)file.length()];
         int fsize = fis.read(bytearray);
         baos.write(bytearray, 0, fsize);
         res.setContentLength(baos.size());
         res.setContentType("image/jpeg");
         res.setHeader("Content-Disposition", "attachment; filename=\"jpegimage.jpg\"");
         baos.writeTo(sos);
         baos.close();
      }
      else res.sendError(res.SC_NOT_ACCEPTABLE, "Your browser does not accept image/jpeg");
   }
}
```

下図はこのサーブレットを Internet Explorer で呼び出した例である:

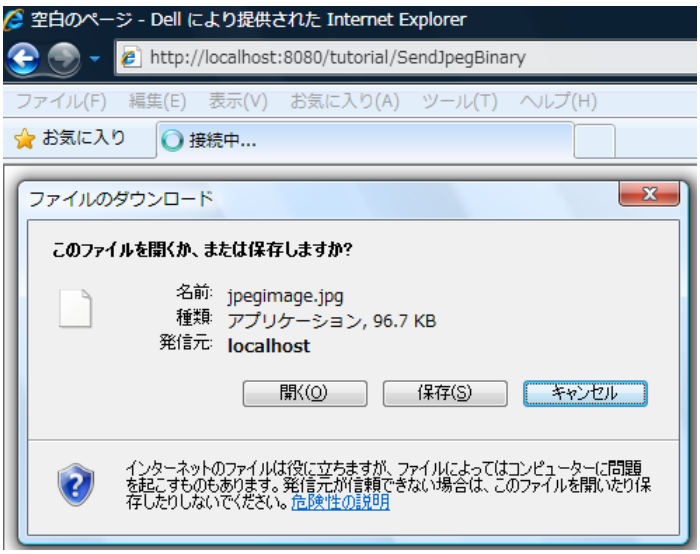


図 8-9：SendJpegBinary の呼び出し

ここでは以下の 2 つの行で送られるデータの内容をブラウザに連絡している：

```
res.setContentType("jpeg/image");
res.setHeader("Content-Disposition", "attachment; filename=\"jpegimage.jpg\"");
```

これらのヘッダ行は、ファイルのダウンロードで頻繁に使われるものである。

8.6.2　Office のデータを送る

Microsoft の Office のアプリケーションのファイルを Internet Explorer 上で表示出来るようにする場合が多いかもしれない。これまでの解説の具体的な応用として、次節に Office 等のドキュメントを圧縮してブラウザに送信するプログラムを紹介するが、その前に注意しなければならない事項を示す。

8.6.3　ブラウザのウィンドウで開けるドキュメント

特に Vista などでは、Windows Internet Explorer 7 または Windows Internet Explorer 8 で Microsoft Office Word 2007、Microsoft Office Excel 2007、または Microsoft Office PowerPoint 2007 のドキュメントを開こうとすると、Internet Explorer と同じウィンドウ内ではドキュメントが開かれないことがある。代わりに、Word 2007、Excel 2007、または PowerPoint 2007 の新しいウィンドウが開いてドキュメントが表示される。これは Microsoft の説明によればセキュリティなどの理由からそのような仕様にしたとのだという。どうしても Internet Explorer と同じウィンドウ内でこれらのファイルを開きたいときは、Microsoft のサポート・ページ http://support.microsoft.com/kb/927009/ja にその方法が示されているので、それを参照されたい。

Internet Explorer とともに Office がインストールされている PC であっても、その Office が旧バージョンである可能性もあるので、**ブラウザに送る Office ドキュメントは、以前の形式でドキュメント・ルートにストアすることが好ましい**。例えば、アメリカの連邦通信委員会のサイト www.fcc.gov/でアップロードされている発表の word ドキュメントは、以前の.doc 形式になっている。

8.6.4　ファイル・タイプと MIME タイプ

Internet Explorer が受け付ける Content-Type（即ち application/vnd.ms-powerpoint、application/vnd.ms-excel、または application/msword）を **Content-Type ヘッダにセットしてやらないと、Internet Explorer はこれを正しく表示しない**ので注意が必要である。これは下表からすると.doc、.dot、.xls、.xlt、.xla、.ppt、.pot、.pps、及び.ppa の

ファイルに限定されるということであるが、実際は Office 2007 のドキュメントも開くことができる。Internet Explorer が自分のウィンドウ内で開けないドキュメントはダウンロード扱いとなる。

しかしながら Accept ヘッダでブラウザからで提示されていないタイプであっても、**setContentType で正しい MIME タイプをブラウザに教えてやるのは良い習慣**である。

Microsoft Office と OpenOffice の各ドキュメント形式に対する MIME タイプは下表のとおりである:

表 8-4:Microsoft Office と OpenOffice の各ドキュメント形式に対する MIME タイプ

拡張子	MIME タイプ
Microsoft	
.doc	application/msword
.dot	application/msword
.docx	application/vnd.openxmlformats-officedocument.wordprocessingml.document
.dotx	application/vnd.openxmlformats-officedocument.wordprocessingml.template
.docm	application/vnd.ms-word.document.macroEnabled.12
.dotm	application/vnd.ms-word.template.macroEnabled.12
.xls	application/vnd.ms-excel
.xlt	application/vnd.ms-excel
.xla	application/vnd.ms-excel
.xlsx	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
.xltx	application/vnd.openxmlformats-officedocument.spreadsheetml.template
.xlsm	application/vnd.ms-excel.sheet.macroEnabled.12
.xltm	application/vnd.ms-excel.template.macroEnabled.12
.xlam	application/vnd.ms-excel.addin.macroEnabled.12
.xlsb	application/vnd.ms-excel.sheet.binary.macroEnabled.12
.ppt	application/vnd.ms-powerpoint
.pot	application/vnd.ms-powerpoint
.pps	application/vnd.ms-powerpoint
.ppa	application/vnd.ms-powerpoint
.pptx	application/vnd.openxmlformats-officedocument.presentationml.presentation
.potx	application/vnd.openxmlformats-officedocument.presentationml.template
.ppsx	application/vnd.openxmlformats-officedocument.presentationml.slideshow
.ppam	application/vnd.ms-powerpoint.addin.macroEnabled.12
.pptm	application/vnd.ms-powerpoint.presentation.macroEnabled.12
.potm	application/vnd.ms-powerpoint.template.macroEnabled.12
.ppsm	application/vnd.ms-powerpoint.slideshow.macroEnabled.12
OpenOffice	
.odt	application/vnd.oasis.opendocument.text
.ott	application/vnd.oasis.opendocument.text-template
.oth	application/vnd.oasis.opendocument.text-web
.odm	application/vnd.oasis.opendocument.text-master
.odg	application/vnd.oasis.opendocument.graphics
.otg	application/vnd.oasis.opendocument.graphics-template
.odp	application/vnd.oasis.opendocument.presentation

.otp	application/vnd.oasis.opendocument.presentation-template
.ods	application/vnd.oasis.opendocument.spreadsheet
.ots	application/vnd.oasis.opendocument.spreadsheet-template
.odc	application/vnd.oasis.opendocument.chart
.odf	application/vnd.oasis.opendocument.formula
.odb	application/vnd.oasis.opendocument.database
.odi	application/vnd.oasis.opendocument.image
.oxt	application/vnd.openofficeorg.extension

次の DocToMime クラスはこの表、及び良く利用される MIME タイプに基づいて、ドキュメント名から MIME タイプを得るものである。DocToMime.getMimeType というメソッドは、拡張子付きのファイル名を引数として渡すと、その MIME タイプが String で返される。対応する MIME タイプがない場合は null を返す。

なお javax.servlet.ServletContext インターフェイスにも getMimeType(java.lang.String file)という同じ名前のメソッドがあり、これも使える。これはこのサーブレット・コンテナに登録されている(web.xml 配備記述子で指定)MIME タイプのみを返す。残念ながら **c:\tomcat7\conf\web.xml には OpenOffice の正確な MIME タイプは含まれているが、競合する Microsoft Office ドキュメント形式に対する MIME タイプは記されていない。**

web.xml に Microsoft Office のドキュメントの MIME タイプを追加して、javax.servlet.ServletContext#getMimeType(java.lang.String file)メソッドを使うように後述の GetStaticDocument サーブレットを変更するのは、読者の宿題とする。

```
package basic_package;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class DocToMime {

   /**
    * <p> Get MIME type from file name (including extension) </p>
    * @param fileName; File name including extension
    * @returns; MIME type for the file name, or null if no corresponding MIME type available
    */

   static String getMimeType(String fileName){
      String mimeType = null;
      int dotPosition = fileName.lastIndexOf(".");
      if (dotPosition != -1){
         String theExtension =fileName.substring(dotPosition).toLowerCase();
         for (int i = 0; i < extensions.length; i++){
            if (extensions[i].equals(theExtension)) mimeType = mimeTypes[i];
         }
      }
      return mimeType;
   }

   public static String[] extensions = {
      ".doc", ".dot", ".docx", ".dotx", ".docm", ".dotm", ".xls", ".xlt", ".xla", ".xlsx", ".xltx",
      ".xlsm", ".xltm", ".xlam", ".xlsb", ".ppt", ".pot", ".pps", ".ppa", ".pptx", ".potx",
      ".ppsx", ".ppam", ".pptm", ".potm", ".ppsm", ".odt", ".ott", ".oth", ".odm", ".odg" ,
      ".otg", ".odp", ".otp", ".ods", ".ots", ".odc", ".odf", ".odb", ".odi", ".oxt",
      ".html", ".htm", ".xml",  ".rtf", ".mid", ".wav",  ".gif", ".jpg", ".png", ".tiff",
      ".bmp", ".wmf", ".avi", ".mpeg", ".ps", ".pdf",
   };

   public static String[] mimeTypes = {
      "application/msword",
      "application/msword",
      "application/vnd.openxmlformats-officedocument.wordprocessingml.document",
      "application/vnd.openxmlformats-officedocument.wordprocessingml.template",
      "application/vnd.ms-word.document.macroEnabled.12",
```

```
      "application/vnd.ms-word.template.macroEnabled.12",
      "application/vnd.ms-excel",
      "application/vnd.ms-excel",
      "application/vnd.ms-excel",
      "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
      "application/vnd.openxmlformats-officedocument.spreadsheetml.template",
      "application/vnd.ms-excel.sheet.macroEnabled.12",
      "application/vnd.ms-excel.template.macroEnabled.12",
      "application/vnd.ms-excel.addin.macroEnabled.12",
      "application/vnd.ms-excel.sheet.binary.macroEnabled.12",
      "application/vnd.ms-powerpoint",
      "application/vnd.ms-powerpoint",
      "application/vnd.ms-powerpoint",
      "application/vnd.ms-powerpoint",
      "application/vnd.openxmlformats-officedocument.presentationml.presentation",
      "application/vnd.openxmlformats-officedocument.presentationml.template",
      "application/vnd.openxmlformats-officedocument.presentationml.slideshow",
      "application/vnd.ms-powerpoint.addin.macroEnabled.12",
      "application/vnd.ms-powerpoint.presentation.macroEnabled.12",
      "application/vnd.ms-powerpoint.template.macroEnabled.12",
      "application/vnd.ms-powerpoint.slideshow.macroEnabled.12",
      "application/vnd.oasis.opendocument.text",
      "application/vnd.oasis.opendocument.text-template",
      "application/vnd.oasis.opendocument.text-web",
      "application/vnd.oasis.opendocument.text-master",
      "application/vnd.oasis.opendocument.graphics",
      "application/vnd.oasis.opendocument.graphics-template",
      "application/vnd.oasis.opendocument.presentation",
      "application/vnd.oasis.opendocument.presentation-template",
      "application/vnd.oasis.opendocument.spreadsheet",
      "application/vnd.oasis.opendocument.spreadsheet-template",
      "application/vnd.oasis.opendocument.chart",
      "application/vnd.oasis.opendocument.formula",
      "application/vnd.oasis.opendocument.database",
      "application/vnd.oasis.opendocument.image",
      "application/vnd.openofficeorg.extension",
      "text/html",
      "text/html",
      "text/xml",
      "text/richtext",
      "audio/mid",
      "audio/wav",
      "image/gif",
      "image/jpeg",
      "image/png",
      "image/tiff",
      "image/bmp",
      "image/x-wmf",
      "video/avi",
      "video/mpeg",
      "application/postscript",
      "application/pdf"
   };

   /**
    * <p>
    * Test the getMimeType method
    * Console in the file name
    * </p>
    */
   public static void main(String[] args) {
      BufferedReader ci = new BufferedReader(new InputStreamReader(System.in));
      while (true) {
         try {
            System.out.println(getMimeType(ci.readLine()));
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
   }

}
```

8.7節　　圧縮転送

バイナリ・データをブラウザに送るための解説のまとめとして、Eclipse 上の WebContent 即ちそのアプリケーションのドキュメント・ベースに置かれた各種のファイルをブラウザからアクセスするサーブレットを紹介しよう。なお圧縮伝送に関しては、サーブレット・フィルタの章で述べる圧縮フィルタも良く使われる。

このアプリケーションは、

```
http://localhost:8080/tutorial/GetStaticDocument?pptdata.pptx
```

のように、GetStaticDocument というサーブレットを呼んで、クエリ文字列として与えた pptdata.pptx のようなファイルを、可能であれば **GZIP 圧縮**して送信させるものである。

このプログラムは基本的には「ブラウザに各種ファイルを送る」の項で示した SendJpegBinary を高度化したものである。

先ずそのコードを示そう:

```
package basic_package;

import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.zip.GZIPOutputStream;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


@WebServlet("/GetStaticDocument")
public class GetStaticDocument extends HttpServlet
{

   private static final long serialVersionUID = 1L;

   @Override
   public String getServletInfo() {
      return "GetStaticDocument servlet, Version 1.0, Oct. 2010, by Cresc"; }

   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException{
      performTask(req,res); }

   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res); }

   public void performTask(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
   {
      String fileName = null;
      try
      {
         FileInputStream fis;
         ServletOutputStream sos;
         ByteArrayOutputStream baos;
         GZIPOutputStream gzipos = null;
```

```
        //Accept ヘッダでブラウザが GZIP 対応か調べる
        boolean gzipAvailable = false;
        String acceptEncodings = req.getHeader("Accept-Encoding");
        if ((acceptEncodings != null)
              && (acceptEncodings.toLowerCase().indexOf("gzip") != -1)) {
           gzipAvailable = true;
        }

        //クエリ文字列からのファイル名をドキュメント・ルート内で探す
        fileName = req.getQueryString();
        ServletContext context = getServletContext();
        String fname = context.getRealPath(req.getRequestURI());
        fname = fname.replace("\\tutorial\\GetStaticDocument", "\\" + fileName);
        fis = new FileInputStream(fname);      //ファイルを開く

        //ブラウザにコンテント・タイプを知らせる
        String mimeType = DocToMime.getMimeType(fileName);
        if (mimeType != null) res.setContentType(mimeType);

        //ファイル読み出しの準備
        sos = res.getOutputStream();           //エンジンへの出力ストリーム
        baos = new ByteArrayOutputStream();       //バイト数を知るためのバイトバッファ
        if(gzipAvailable) gzipos = new GZIPOutputStream(baos);  //GZIP 圧縮のエンコーダ

        //ファイルの読み出しとバイト配列バッファへの転送
        int totalBytesRead = 0;
        byte[] byteArray = new byte[8192];        //8192 バイトのバイトバッファを用意
        while (true) {
           int bytesRead = fis.read(byteArray);   //バッファの最大容量まで読み込み
           if (bytesRead == -1) break;            //EOF に達したかのチェック
           if (gzipAvailable) gzipos.write(byteArray, 0, bytesRead);  //バッファから GZIP 出力ストリーム
へ
           else baos.write(byteArray, 0, bytesRead);
           totalBytesRead = totalBytesRead + bytesRead;
        }
        fis.close();                              //ファイルの内容を全て読み出した
        if (gzipAvailable) gzipos.close();        //CRC を含めバイトバッファに流しだす

        //ヘッダをセットする。必要なら次の行を活かしてダウンロードのダイヤログを開かせる
//      res.setHeader("Content-Disposition", "attachment; filename=\"" + fileName + "\"");
        if (gzipAvailable)
           res.setHeader("Content-Encoding", "gzip");  //ブラウザに GZIP 圧縮であることを知らせる
        res.setContentLength(baos.size());            //HTTP ボディのバイト長を通知

/**     //テスト用
        System.out.println("File: " + fileName);
        System.out.println("Content type: " + mimeType);
        System.out.println("Original byte size: " + totalBytesRead);
        System.out.println("Compressed byte size: " + baos.size());
         */

        //サーブレット・コンテナのバッファに書き込む
        baos.writeTo(sos);                        //ボディを出力ストリームに移す
        baos.close();                             //明示的にバイトバッファを閉じる
        sos.close();                              //明示的に出力ストリームをフラッシュして閉じる
        res.flushBuffer();                        //明示的にコンテナのバッファをフラッシュ
     }

     //エラー処理
     catch(FileNotFoundException e){
        e.printStackTrace();
        res.setContentType("text/html; charset=Windows-31J");
        PrintWriter out = res.getWriter();
        out.println("<HTML>");
        out.println("<HEAD><TITLE>見つかりません</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("<BIG>ドキュメント\"" + fileName +"\"が見つかりません。</BIG>");
        out.println("</BODY></HTML>");
```

```
        }
        catch(Exception e) {
           e.printStackTrace();
           try{
              res.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR, e.getMessage());
           } catch (IOException anotherException){anotherException.printStackTrace();}
        }
    }
}
```

以下に、このコードのポイントを記す:

- gzipos = new GZIPOutputStream(baos);という具合に、バイト配列出力ストリームに GZIP (”ジージップ”と発音) 圧縮フィルタを付加していることである。この圧縮方式は HTTP プロトコルでは転送データ量を減らし高速にブラウザが表示できるようにするために良く使用されており、ブラウザの殆どがこれに対応している。圧縮はテキストが多いドキュメントでは大きな効果が得られる。JPEG のような既に圧縮されているドキュメントでは効果は得られない。

- ここではクライアントからの Accept-Encoding ヘッダに gzip が含まれているかをチェックしている。しかしながら、GZIP は現在殆どのブラウザが対応しているので、総てを GZIP 圧縮しても問題はない。少なくとも IE、Firefox、Chrome、Safari の総てで問題は生じない。但し携帯電話機のブラウザは調べていない。なお IE では「ファイルのダウンロード」のダイヤログが出て「開く」あるいは「保存」をクリックすると、再度このサーブレットをアクセスする場合があるが、その場合には Accept-Encoding ヘッダに gzip が含まれない。このとき toLowerCase()を使って小文字に変換していることに注意されたい。**HTTP ヘッダに関しては、ヘッダ名は大文字と小文字は区別されないが、ヘッダ値はメッセージ上の文字列がそのままで取り扱われる。**ブラウザによっては GZIP をいう値をセットしているかもしれない。

- ブラウザにはそのコンテントの MIME タイプを次のように知らせる。
  String mimeType = DocToMime.getMimeType(fileName);
  if (mimeType != null) res.setContentType(mimeType);
  ブラウザはこの情報をもとに、このコンテントをどのように扱うかを判断する。

- SendJpegBinary のコードでは File.length()の長さのバイト配列をファイル読み出しのバッファとして用意していたが、それでは大きなファイルの場合はメモリを消費してしまう。このコードでは、Tomcat のデフォルトのバッファ・サイズと同じ 8192 バイトのバイト配列に読み出しながら、そのブロックでバイト配列出力バッファ(ByteArrayOutputBuffer)にそのまま、あるいは圧縮して移す方式にしている。

- res.setHeader("Content-Disposition", "attachment; filename=\"" + fileName + "\"");というコメント・アウトされた行は、SendJpegBinary のコードで示したように、必ずブラウザがダウンロードのダイヤログを開かせるもので、**Office ドキュメントが主体である場合以外はなるべくこれは利用したほうが良い。**

- 圧縮する場合は res.setHeader("Content-Encoding", "gzip");でブラウザにコンテントが GZIP 圧縮されていることを通知する。**Transfer-Encoding と混同しないよう注意**すること。

- res.setContentLength(baos.size());でコンテント長を指定している。これは HTTP/1.0 クライアントにも対応するようにチャンク転送をさせない為である。

- コメント・アウトされたブロックを活かすと、圧縮でどれだけ転送データ量が減るかをコンソールで知ることが出来る。

- **指定されたファイルが見つからないときは、404(Not Found)応答を返さないで、HTML で返す。**これはそのファイルが存在していないことを正しくユーザに伝える為である。404 ステータス応答に対するブラウザの解釈と対応が異なっていることもあるし、Tomcat からのエラー・ページや「Web ページがみつかりません」というブラウザの表現は、ユーザを混乱させる。

圧縮転送は実は後で解説するサーブレット・フィルタの典型的なアプリケーションである。ここに説明したサーブレットを使わなくても、圧縮用のフィルタがいろいろ存在する。例えば Tomcat 7 に既に同梱されている examples というアプリケーションの中には compressionFilters というパッケージが存在する。このフィルタを使うと、応答のデータ量が閾値を超すとそれを圧縮してくれる。従って静的ドキュメントだけでなく、サーブレットが作る動的ドキュメントも圧縮する。

8.8節　この章のまとめ

この章のまとめとして、サーブレット開発における応答関係のポイントは次のようである（前項のポイントも参考のこと）：

1. Tera Term やプロキシが、デバッグには良いツールとなる。
2. 必要なヘッダ行をセットした後で、ボディ部をセットしないと、チャンク転送ではエラーが起きる。ボディ部のセットはサーブレットの処理の最後の部分にまとめる。
3. HTTP 要求スレッドがサーブレットを抜けるときは、使ったバッファ類はフラッシュとクローズがなされるが、明示的に書いておくことは良い習慣である。
4. PC 用のブラウザの総てが GZIP 圧縮を受け付けるので、テキスト・ベースの大きなデータを送信するときは、圧縮を行うことが好ましい。
5. 400 及び 500 番台のエラー・コードは sendError を、200 及び 300 番台のエラー・コードは setStatus を使う。

第9章　セッション

セッションとは「クライアント状態の維持」の節で示したように、あるクライアントとの間での結び付いている一連の要求を識別することである。これは要求/応答で完結しているステートレスのHTTP プロトコルに基づくウェブ・アプリケーションでこれを実現する為に、サーブレットにはHttpSession インターフェイスが用意されている。以下このHttpSession とその仕組み、動作、及び使用に関して述べることにする。

セッションに関しては、サーブレット3.0 仕様書の第 7 章に記されているので、これも一読をお勧めする。

9.1節　　セッションとは

セッションという概念はいろんなレベルで定義される。一番低い階層では、TCP、IP、データリンク層などの通信プロトコルでのセッションである。これらは各層のピア間でのデータ転送の開始や終了にかかわる一連の過程を言う。例えばダイヤルアップのデータリンク層の PPP セッションは、ユーザがインターネットに電話回線などを介してプロバイダと接続してから終了するまでの間はそのセッションが維持される。TCP のセッションはクライアントがサーバとの間に所定の受け付けポート番号で TCP 接続を開始して TCP パケットの一連の転送を行い、それが終了したら TCP 接続を開放するが、それまでの接続期間で TCP セッションが維持されたことになる。

反対にビジネスの階層で考えれば、ビジネスのためのオブジェクト群にあるクライアントのオブジェクトが参加してからそれが開放されるまでの期間がそのクライアントとのセッションが張られた期間だと言えよう。一般には顧客や取引先の口座がそれにあたる。具体的には、ある顧客がネットワークを介してオンライン・バンクに登録を始めたときから、そのオンライン・バンクのデータベースからそのクライアントのデータ（データベースの用語では、継続性(Persistence)オブジェクトと呼ぶ）がすべて削除されるまではその顧客とのセッションが維持されていると考えられる。

しからばサーブレットのレベルでのセッションとは何であろうか？一般的にはこれは、例えばあるクライアントがそのオンライン・バンクのシステムに預金残高確認のためにアクセスを開始してからそのアクセスを終了するまでの期間、あるいはあるクライアントがあるショッピングのサイトを訪問して、ショッピング・カートで買い物をし、その取引を終了してそのサイトを出るまでの期間のそのユーザの識別として使われる。クライアントはその間一連の画面を推移していく。しかしながらクライアントは画面を表示させたままで長時間席を外したり、逆にサーバからの応答が到着するまえに更新や元のページに戻ったり、別のサイトに移ってしまったり意図しない行動をしばしばとる。場合によってはネットワークの障害によるセッションの遮断もありえよう。

従ってセッションの終了の判断には、タイムアウト、状態管理やネットワークの障害なども配慮しなければならない。タイムアウトは HttpSession インターフェイスに用意されている。ネットワーク障害は、検出したオブジェクトがスローしたものがセッション管理のために受け取れるようにするのが好ましい。

セッション管理のために用意されているサーブレット API は特定のユーザが継続してこのサイトにアクセスしていることを確認する手段を提供はするが、実際の業務アプリケーションでのユーザのオブジェクトがどの状態にあるかという状態及びその遷移の管理はソフトウエア開発者の責任である。これの情報はセッションの属性(Attribute)オブジェクトにあるいは外部の管理オブジェクトに持たせることになる。

HttpSession インターフェイスを実装したオブジェクトは、これらのような用途だけでなく、継続性オブジェクトとして

機能させることも可能である。セッション・データを永続データにする場合、後述のようにその永続データをデータベースに格納できる。データ量が少なく、データベース機能が不要なときは、独自の Cookie を作成して使用するか、多くの場合、ファイル・システムまたはその他のリモート・オブジェクトを使用することも可能である。Cookie を使う場合は、そのデータがクライアントに蓄積されまた送受信されるので、効率が下がりまた信頼性も下がる。

セッション管理で更に注意すべき点として、このサーブレットのインスタンスが負荷分散のために分散可能(Distributable)として複数のバーチャルホストに分散して配置される場合である。(SingleThreadModel を実装し、このサーブレットのインスタンスが複数生成されている場合は HttpSession オブジェクトはひとつのコンテナ上で管理されるため大丈夫である)。この場合は、クライアントはいつも特定のエンジン上のサーブレットのインスタンスにアクセスする訳ではないので、何らかの対策が必要となる。

クラスタ構成の複数のアプリケーション・サーバ間でセッション状態を維持する為の基本的なアプローチは次の 3 つになる:

- セッション・オブジェクトを共有データ・ベースにストアする
- JMS のような通知のメカニズムを使ってメモリ内セッションたちを同期化する
- ユーザたちを特定のアプリケーション・サーバ・インスタンスに固定させるためにセッション・アフィニティ(Session affinity)を使う

最初の 2 つのアプローチではセッションのコピーを作ることになる。ほとんど全てのサーブレット・コンテナは、HttpSession に保存されたオブジェクトをシリアル化することによって HttpSession の複製が可能である。Tomcat もクラスタリングとセッション複製の機能をサポートしている。従って配布可能なアプリケーションを作る場合には、**セッションの属性として含められたオブジェクトは総てシリアル化可能な(即ち java.io.Serializable インターフェイスを実装している)オブジェクトのみにしておく必要がある**。但し直列化して送るのは、属性として含めたオブジェクトが大きいと時間がかかってしまうので、なるべく 3 番目のアプローチをとることが好ましい。

9.2節　サーブレットにおけるセッション管理の仕組み

サーブレット API の仕様書は、特定のユーザからの一連の要求を互いに関連付けるメカニズムを HttpSession というインターフェイスで提供している。サーブレット・エンジンは各クライアントに対応する HttpSession のオブジェクトを簡単なデータベースで管理する。到来要求のセッションがこのデータベースに存在すればそのユーザはこのセッションに参加していると判断する。

HTTP はステートレスのプロトコルと言われる。要求と応答のセットで閉じてしまっており、おまけに HTTP/1.0 では TCP の接続も切断される。HTTP/1.0 対応ブラウザも存在している限り、TCP 接続はセッション管理には使えない。サーブレット仕様書では、セッション管理のメカニズムとして URL 書き換え(URL Rewriting)、クッキー(Cookies)、それに SSL セッションの使用を示している。もうひとつセッション管理に利用できるものとして、HTML の hidden で指定した文字列であるが、これは静的な HTML ドキュメントでは使えない。

9.2.1　HTTP におけるセッション管理の選択肢

HTTP におけるセッション管理には幾つかの手段がある。サーブレットの API では、そのうち URL 書き換えとクッキーを使ったセッション管理手段を提供している。

9.2.1.1　URL 書き換え(URL Rewriting)

URL 書き換えはセッション管理として最小限エンジンが実装すべきもので、クッキーを受け付けないクライアントでも対応できる。URL 再書き込みは URL パス情報にセッションの ID を付加するもので、例えば、

http://www.myserver.co.jp/catalog/index.html;jsessionid=1234

のような URL となる。ここにセッションのパラメタ名は jsessionid である。

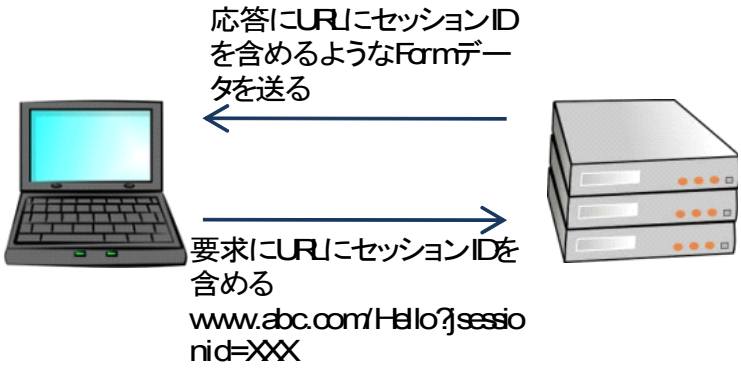


図 9-1：URL 書き換え

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し、HTML ソースを開いて変更できるので、セキュリティが問題なアプリケーションでは使わないほうが良い。ユーザが一旦そのセッションを抜け、その後ブックマークやリンクで戻ってきたときには、そのセッション情報は失われてしまうことにも注意しなければならない。またプレーン（静的）な HTML ファイルはそのアプリケーションでは使えない（フォームがないと、セッション ID が返ってこない）。

サーブレット仕様書では、「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を除いては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」と書いている。

しかしながら特に欧州など、プライバシに対する考え方が厳格な地域においては、ユーザが自分のブラウザをクッキーを無効に設定していることがある。また、携帯電話機のブラウザでは、そのキャリアたちは自分たちのゲートウェイでクッキーを通過させなくしている場合もある（最近のスマートホンではこの制限は外されてきている）。

サーブレットをのコードを書くときは、クッキーを受け付けないクライアントには URL 書き込みで対応するようにしなければならない。

9.2.1.2　クッキー(Cookies)

クッキーによるセッションのメカニズムも全てのサーブレット・エンジンに実装が要求されている。サーブレット・エンジンがクライアントに送ったクッキーとしての ID を、クライアントはその後の要求にクッキーのパラメタとして添付する。パラメタ名は JSESSIONID と今度は大文字である。

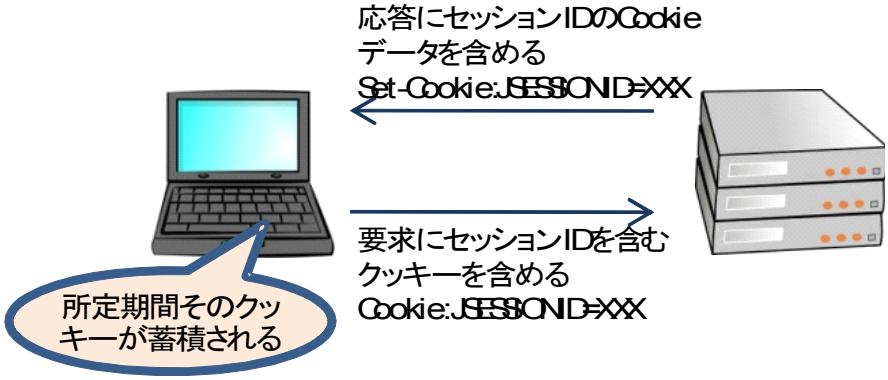


図 9-2:クッキーによる識別

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。サーブレット 3.0 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。現に後で示すように **Tomcat 7 では set-cookie ヘッダに HttpOnly を付している**。HttpOnly の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。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。

この方式の問題点は:

- プライバシに敏感なユーザは自分の PC にクッキーが蓄積されないように設定していることがあるので、URL 書き換えなどでそれに対応できるようにしておかねばならない。
- 最近のスマートホン以外のいわゆる「機能電話機」では、クッキーは使用できないことが多いので、URL 書き換えでそれに対応できるようにしておかねばならない。
- **同じ PC 上で同じブラウザを幾つか開く、あるいは幾つかのタブで開いたときには、それらの画面からは同じクッキーが返されることになる(「ブラウザにおけるクッキーの保持」の項を参照されたい)**。こんなことは想定外かもしれないが、1 台の PC 上で 2 人のユーザが 2 つの IE を使って、あるいは 2 つのタブを開いて、同じ URL をアクセスしたら、サーブレットはそれらのユーザを区別できなくなってしまう。ゲームなどではあり得るかも知れない、もし E-コマースのアプリケーションでそのような使われ方をしたらどうなるだろうか?そのような場合には、**URL 書き換え、後述の SSL セッション、あるいは後述の Hidden とクッキー・セッションの組み合わせ、を使ってユーザを識別**しなければならなくなる。Tomcat では後述のように、セッション管理にクッキーを使わないよう設定することは可能である。

9.2.1.3 SSL セッション

SSL は TCP とサーブレット・エンジンを含むアプリケーション層とのインターフェイスに置かれた暗号化とユーザ識別のメカニズムであり、これをサーブレット・エンジンはセッション管理に使うことができる。

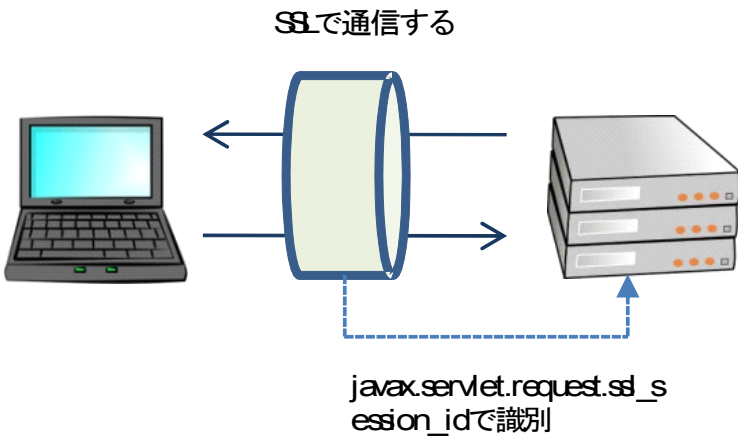


図 9-3 : SSL セッションによる識別

クライアントからの要求が例えば 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 メソッドによって知ることが出来る。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提供している:

表 9-1 : 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、サーブレット・コンテナはサーブレットのプログラマに対しそれを javax.servlet.request.X509Certificate 型のオブジェクトの配列で示し、javax.servlet.request.X509Certificate の ServletRequest 属性を介してアクセスできるようにする。

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。

SSL セッションに関しては、「セキュリティ」の章の「SSL 接続上のサーブレット」の項で詳しく説明してある。

9.2.1.4 Hidden による識別

これは HTML の中に表示されない隠しデータとしてセッション ID を含めるものである。ユーザが Submit を押すことで、このデータが HTTP 要求パラメタのひとつとして送信される。

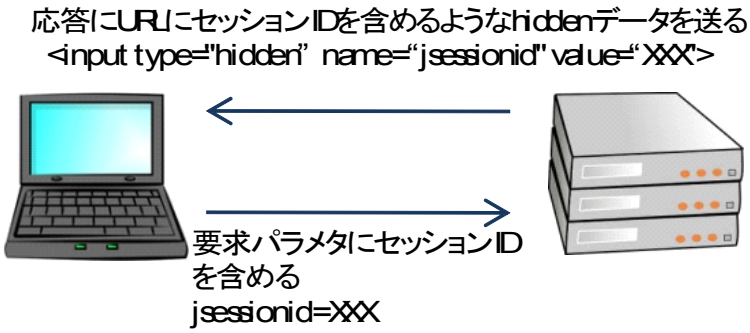


図 9-4：Hidden による識別

この方式に対してはサーブレット仕様書は言及していない。この方式は URL 書き換えと同じく、ユーザが HTML ソースを変更出来ることから、セキュリティが問題なアプリケーションには使えなし、各ページが動的に生成されていないといけないという制限が付く。

9.2.2　セッション管理のメカニズム

セッション管理で使われているメカニズムと特性を把握しておくことは重要である。使われているメカニズムは HttpSession、及び HttpSessionBindingListener を含むの 4 つリスナのインターフェイスと HttpSessionBindingEvent を含む 2 つのイベントのクラスで、プログラマからは一応遮蔽されているとはいえ、**URL 書き換えはプログラマが関与しなければならない**。即ちクライアントが HTML Form でサーバに返す URL にセッション ID を含ませるように、プログラマはクライアントに送る HTML テキストにセッション ID つきの URL を含めねばならない。

以下にサーブレットにおけるセッションのメカニズムのイメージ（セキュリティは除く）を示す。我々はのちほど SimpleSessionTest というサーブレットを使ってこのイメージを確認していくことにする。

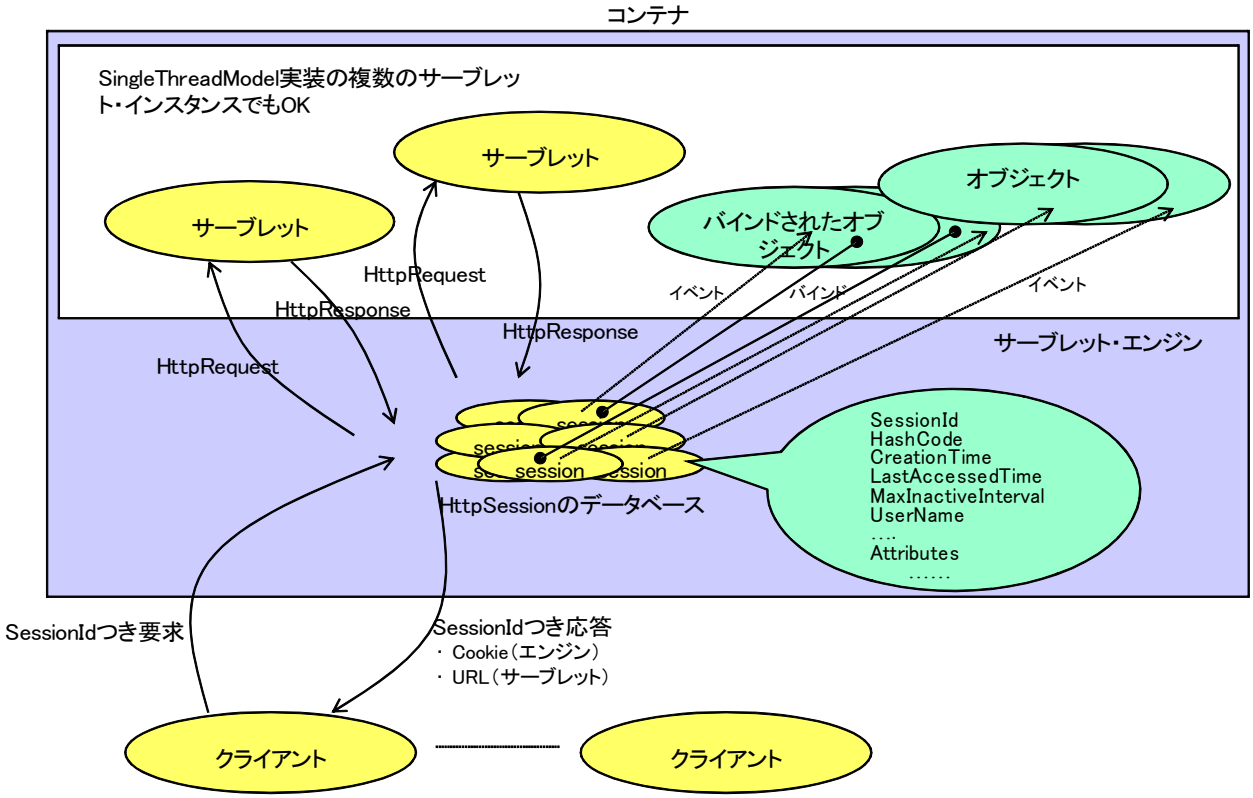


図 9-5：セッション管理の為のメカニズム

サーブレット・エンジンは HttpSession のオブジェクトを管理する。サーブレットは getSession や invalidate のメソッドでこのオブジェクトの生成と削除をエンジンに依頼する。クライアントとのセッションの維持は SessionId でなされる。この ID はサーブレットがこのコンテナ上で重複しないようエンジンが生成する。SessionId はクッキーや URL 再書き込みを使ってクライアントとの間で交わされる。HttpSession のオブジェクトは緑色で示すように幾つかのフィールドを持つ。サーブレットは Attribute というフィールドで複数のオブジェクトをバインドできる。オブジェクトが session にバインドされときあるいは、バインドから外されたときは、イベントでそのオブジェクトに、あるいは属性に変化があったときは別のオブジェクトに通知できる。

イベントによる通知は非常に有用であり、タイムアウトによりサーブレット・エンジンがそのセッションが終了とみなして削除したことによる処理の開始や、各サーブレットのインスタンスのクライアント対応を統括するオブジェクトが各セッションの状況を把握したりするのに使うことができる。

セッションのオブジェクトたちはサーブレット・コンテナ上の簡単なデータベースに登録されるので、これが多くなるとサーブレット・エンジンのオーバヘッドが増えることになる。したがって**クライアントがログアウトした時点で削除(invalidate)することが好ましい。また有効期間も支障が出ない範囲でなるべく短く設定することが好ましい。**

9.2.3 HttpSession オブジェクト

上図で示す HttpSession オブジェクトとはどのようなものであろうか？これはあるクライアントとのセッションを表現しており、以下のような属性を持つ：

1. そのセッション識別のためのセッション ID
2. そのセッションが生成された時刻
3. 前回アクセスされた時刻
4. そのセッションの有効期間（あるいはタイムアウト時間）
5. 属性としてバインドされたオブジェクトたち

5 番目のオブジェクトのバインド、即ち貼り付けは、情報の共有の節で述べたとおり、同じコンテキスト内に属し、その要求オブジェクトを使う他の JSP やサーブレットと、そのバインドされたオブジェクトを、そのセッションが有効な限り共有できる。有効範囲を持ったオブジェクトをスコープ・オブジェクトと呼ぶ。この場合はスコープはセッション・オブジェクトである。

オブジェクトによっては、あるオブジェクト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場合がある。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がある：

- 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。

このようなイベント/リスナの仕組みは、サーブレットではコンテキスト、セッション、及び要求で用意されている。

表 9-2：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

9.3節　　セッションの管理の為のメソッドたち

9.3.1　HTTP 要求のセッションの状態を知る、及び新たなセッションの生成のためのメソッドたち

HttpServletRequest のオブジェクトには、その要求に対するセッションの状況、及びセッションが存在しないときに新たなセッションを取得する為のメソッドが用意されている。

表 9-3：セッションに関する HttpServletRequest のメソッドたち

メソッド	内容
getSession	この要求に対応する現在のセッションを返す。この要求がセッションを持っていないときは新しいセッションを作る。
getSession(boolean create)	この要求に対応した現在の HttpSession を返すか、現行のセッションが存在せず、かつ create が true のときは、新しいセッションを返す。create が false で、かつこの要求が有効(varid)な HttpSession を有さないときは、このメソッドは null を返す。セッションが正しく維持されるためにはこのメソッドを応答がコミットされる前に呼ばねばならない true – 必要な場合は新規のセッションをこの要求に生成する。 false – 現在セッションが存在していないときは null を返す。
getRequestedSessionId()	そのクライアントによって指定されているセッション ID を返す。これはその要求に対して現在有効なセッションの ID と異なっている可能性もある。そのクライアントがセッション ID を指定してきていないときは、このメソッドは null を返す。
isRequestedSessionIdValid()	指定されたセッション ID がまだ有効かを返す。
isRequestedSessionIdFromCookie()	要求されたセッション ID がクッキーでなされているかどうかを返す。
isRequestedSessionIdFromURL()	要求されたセッション ID が URI でなされているかどうかを返す。

ここで注意しなければならないのは最初の 2 つの getSession メソッドたちである。これらのメソッドでは、サーブレット・コンテナはその要求に対応する（即ちその要求に含まれているセッション ID が一致する）セッション・オブジェクトを自分の持っている**セッションのデータ・ベースから探しだすだけではなく、もし存在しなければ、新しいセッション・オブジェクトを生成する**。例えば初めての訪問者の場合にログインのページに回すためにセッション

があるかどうかを調べたとき、あるいはもしセッションがあればそれをログしたいときなど、新しいセッション・オブジェクトを生成させたくないときは、getSession(false)を使用しなければならない。

残りの 4 つのメソッドは、もし有れば、要求オブジェクトに含まれているセッション ID 及びそれに対応したセッションに関する情報を提供する。

Tomcat におけるセッション ID は 128 ビットの乱数で、これを 32 桁の 16 進数の String として使っている。

9.3.2 セッションの情報取得と設定の為のメソッドたち

「セッション管理の為のメカニズム」の項で示したように、セッション・オブジェクトたちは幾つかのパラメタを持っていて、サーブレット・コンテナが管理している。これらのパラメタはプログラム的に取得と設定が出来る。以下はそのような操作の為のメソッドたちである:

表 9-4:セッションの情報の取得と設定の為のメソッドたち

メソッド	内容
isNew()	クライアントがまだこのセッションを知らないとき、またはクライアントがセッションに参加しない選択をしているときに true を返す。
invalidate()	本セッションを無効とし、これにバインドされている全てのオブジェクトをバインドから外す
getServletContext()	そのセッションが属している ServletContext を返す。
getId()	本セッションに付与された唯一無二の識別子を含む文字列を返す。識別子はサーブレット・コンテナが付与し、インプリメントに依存する。
getCreationTime()	本セッションが生成された時刻を January 1, 1970 の真夜中からのミリ秒の long 型で返す。
getLastAccessedTime()	このセッションでクライアントが最後に要求を送信した時刻を January 1, 1970 の真夜中からのミリ秒の long 型で返す。本セッションに関わる値の取得や設定などのアプリケーションが行うアクションに対してはこの時間は影響を受けない。
setMaxInactiveInterval(int interval)	サーブレット・コンテナがこのセッションを無効としないクライアントの要求間のインターバルを int 値の秒で設定する。マイナスの値を設定すると永久にタイムアウトを生じない。
getMaxInactiveInterval()	サーブレット・コンテナが本セッションを維持するクライアントのアクセスとアクセスの間の最大時間を int 値の秒で返す。このインターバルを超えるとサーブレット・コンテナはこのセッションを無効とする。

9.3.2.1 isNew メソッドに対する注意

注意が必要なメソッドとして isNew がある。**このメソッドは、そのクライアントが始めての訪問であることを知る為のもので、最初の画面(あるいはページ)の訪問を識別するものではない**。このメソッドは、クライアントがまだこのセッションを知らないとき、またはクライアントがセッションに参加しない選択をしている(例えばクッキーを受け付けなくしている)ときに true を返す。例えば、サーバがクッキー・ベースのセッションだけを使っていて、かつクライアントがクッキーの使用を不可としているときは、セッションは各要求ごとに new の状態となる。

このメソッドは、そのユーザがセッション情報を持っていないことを知って、クライアント認証を受けさせる、あるい

はウェルカム画面に転送(ディスパッチ)する、などのために使うことが出来る。したがってそのような目的のフィルタに使うことが出来る。逆にいえば、フォーム・ベースの認証では、認証された結果既にセッションが確立されてきていることに注意すべきである。

ある到来要求にたいし、それを初期画面生成処理にまわすか、あるいは別の画面生成処理にまわすかどうかの判断には、このメソッドは使えない。確かに isNew==true であれば、初期画面を表示することになろうが、セッションが存在していても初期画面を表示する必要性が出てくる。例えばそのクライアントがそのアプリケーションを一度アクセスし、そのアクセスを終了したあとで、再度そのアプリケーションにアクセスした場合は、セッション・オブジェクトは未だ存在している。あるいはあるクライアントがそのアプリケーションの中で途中で遷移を放棄し、そのアプリケーションを最初からはじめたいと思って、意図的に初期画面を呼ぶこともあり得る。またフォーム・ベースの認証の結果セッションが存在していることもある。そのような画面遷移上の注意事項は別途説明する。

9.3.3 セッションの属性のセットと取得の為のメソッドたち

セッション管理の為のメカニズムの図で示したように、セッション・オブジェクトにはあるオブジェクトを属性(Attribute)として名前をつけてバインド(貼り付け)できる。バインドされたオブジェクトはそのセッションのスコープ内で他のオブジェクトから参照、即ち共有できる。

表 9-5:セッションの属性のセットと取得の為のメソッドたち

メソッド	内容
getAttribute(String name)	本セッションにバインドされている指定した名前のオブジェクトを返す。その名前のオブジェクトがバインドされていないときは null を返す。name はそのオブジェクトの名前を指定する文字列である。
getAttributeNames()	本セッションにバインドされている全てのオブジェクトの名前を含む String オブジェクトの列挙型を返す。
setAttribute(String name, Object value)	指定した名前であるオブジェクトを本セッションにバインドする。このセッションに同じ名前のオブジェクトが既にバインドされているときはオブジェクトは置き換えられる。
removeAttribute(String name)	指定した名前でバインドされているオブジェクトを削除する。指定した名前のオブジェクトがバインドされていないときは本メソッドは何もしない。

9.3.4 セッション ID を応答メッセージに含める為のメソッドたち

クライアントとのセッション維持の為には、そのクライアントに対しセッション ID を教え、それ以降の要求に対しそのセッション ID をクッキーあるいは URL の一部として含めるようにさせねばならない。

サーブレット・エンジンは、その Set-Cookie ヘッダをその応答に含める条件は:

- その要求に対して新たなセッション・オブジェクトが生成されたときに、新たなセッション ID が生成され、その要求に対する応答にその ID を含む Set-Cookie ヘッダを付加する。その後のそのセッションに対応する要求への応答には Set-Cookie ヘッダは付加されない。

クライアントがクッキーを記録しないよう設定されている場合には、サーブレットは生成する HTML テキストの FORM の ACTION に付加する URL に/tutorial/SimpleSessionTest;jsessionid=xxxxx.....などのようにセッション ID を付加しなければならない。以下に示す encodeURL というメソッドはその為にある。例えば encodeURL("/tutorial/SimpleSessionTest")とこのメソッドを呼んだ場合は、サーブレット・エンジンはその要求が isNew であって、Cookie ヘッダを含んでいないときに、指定された URL に対し;jsessionid=xxxxx.....という具合

に、セッション ID を含む文字列をその URL に追加する。**encodeURL と URLEncoder#encode と混同しないように**注意されたい。

表 9-6：セッション ID を応答に含める為のメソッドたち

メソッド	内容
encodeURL(String url)	指定の URL に session ID を含めた形にエンコードする。もしエンコードの必要がなければ入力した URL がそのまま返される。エンジンは session ID を URL に含めるべきか否かのロジックを持つ。 public java.lang.String
encodeRedirectURL(String url)	SendRedirect メソッドに使うために指定した URL をエンコードする。もしエンコードの必要がなければ入力した URL がそのまま返される。エンジンは session ID を URL に含めるべきか否かのロジックを持つ。

9.3.5　イベントとリスナの為のメソッドたち

HTTPSession には、あるイベントが発生したらそれをあるオブジェクトに通知し、またそのオブジェクトがその通知をイベント・ドリブンで受け取る為のリスナ・インターフェイスが用意されている。これらは既に「HTTP セッション・イベント」として表で示されているが、これらのセッションに関わるイベントのリスナとしては以下に示すものがある。HttpSessionEvent イベントには、変化があったセッションを返す HttpSession#getSession()というメソッドがあり、リスナを実装したクラスはそのセッションをもとに、イベント・ドリブンで処理を行う。

9.3.5.1　SessionBindingListner

一部のオブジェクトたちは**自分があるセッションにバインドされた、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。セッションは HttpSession.putValue を呼ぶことでそのオブジェクトをバインドし、HttpSession.removeValue を呼ぶことでそのバインドを外す。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
  このオブジェクトにこれがあるセッションにバインドされたことを通知し、そのセッションを知る。
- ValueUnbound
  このオブジェクトにこれがあるセッションからのバインドが外されたことを通知し、そのセッションを知る。

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。

9.3.5.2　HttpSessionAttributeListner

これは**あるオブジェクトが HttpSession 属性の変更に関するイベント通知を受ける**ためのインターフェイスである。このイベント通知を受けるためには、実装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中で宣言されているか、WebListener でアノテートされているか、あるいは ServletContext の addListener のメソッドたちのひとつで登録されているかしなければならない。

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、置き換えられた、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：

- attributeAdded
  あるセッションにある属性が追加されたことの通知を受ける。**同じ名前で同じオブジェクトを繰り返しセットした場合は、最初のセットででしかイベント生起されない**ので注意を要する。例えばあるオブジェクトを変更させて同じ名前で再度属性としてセットしたときにイベントとして通知させたいときは、その前に一旦その属性を削除する必要がある。
- AttributeRemoved
  あるセッションからある属性が除去されたことの通知を受ける。返される属性は**置き換えられた属性であって、置き換えた属性ではない**ことに注意しなければならない。
- AttributeReplaced
  あるセッションの中である属性が置き換えられたことの通知を受ける。

9.3.5.3 HttpSessionActivationListner

あるセッションに**バインドされたオブジェクトたちは、セッションが活性化(サーブレット・コンテナにセッション・オブジェクトが存在して利用可能な状態に)されるあるいは不活性化(サーブレット・コンテナにあるオブジェクトが外部記憶等に退避され、利用できない状態に)されることを通知**する**コンテナ・イベント**を聞くことが出来る。セッションを VM たち間で移行させる、あるいはセッションを永続化させるコンテナでは、HttpSessionActivationListener を実装したセッションにバインドされた総ての属性を通知する必要がある。

- sessionWillPassivate
  そのセッションが非活性化されようとしていることの通知
- sessionDidActivate
  そのセッションが活性化されたばかりだということに通知

これはサーブレット・コンテナの内部動作に関わるものなので、このリスナの使用頻度はあまり多くないと考えられる。サーブレット・コンテナがあるセッションを他の VM に移そうと判断したときに、HttpSessionActivationListener に通知される。しかし直列化されているときに特にそのオブジェクトに対し対処しなければならないときには、これらのメソッドが利用できる。

9.3.5.4 HttpSessionListner

HttpSession のライフサイクルの変化に関する通知イベントを受けるためのインターフェイス。このイベント通知を受けるためには、実装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中で宣言されているか、WebListener でアノテートされているか、あるいは ServletContext の addListener のメソッドたちのひとつで登録されているかしなければならない。

- sessionCreated
  あるセッションが生成されたことの通知を受ける
- sessionDestroyed
  あるセッションが無効になろうとしていることの通知を受ける。

9.3.6 サーブレット 3.0 でのセッション追跡関係の強化

サーブレット 3.0 では、ServletContext インターフェイスに以下のようなメソッドが新たに追加されている:

表 9-7:サーブレット 3.0 での追加メソッドたち

メソッド	内容
getSessionCookieConfig()	サーブレット・コンテナが管理するセッション・クッキーの設定を取得、これによりプログラムで設定の変更が出来る

メソッド	内容
setSessionTrackingModes(java.util.Set<SessionTrackingMode>)	プログラムでセッション追跡モードを設定する
getDefaultSessionTrackingModes()	デフォルトのセッション追跡モードたちのセットを取得、これによりプログラムによるセッション追跡モードの変更が出来る
getEffectiveSessionTrackingModes()	実効的なセッション追跡モードたちのセットを取得

SessionCookieConfig は、サーブレット・コンテナがセッション管理の為に HTTP 応答メッセージに付加する Set-Cookie:ヘッダ行の要素をプログラムで設定できるようにしている。これらの要素は、コメント、名前、ドメイン、パス、HttpOnly かどうか、secure かどうか、及び有効期限である。

セッション追跡モードの取得と変更に関しては、「クッキーをセッションに使わないよう Tomcat を設定する」の項を参照されたい。

9.3.7 セッション管理以外の目的でクッキーを使う

名前と値で構成されるクッキーはセッション管理以外の目的でブラウザにセットできる。その有効期間は整数で秒で指定できる。マイナスを指定すると、有効期間はそのブラウザが少なくともひとつインスタンスが生成されている間となる。即ち PC 上からそのブラウザを終了させるとセットしたクッキーは消去され、その後は指定したパスへの HTTP 要求メッセージにはそのクッキーのヘッダ行は付加されない。ゼロを指定するとその名前のクッキーを削除させる。**クッキーの名前と値は URL エンコード**されていなければならないことに注意されたい。ブラウザ側で JavaScript を使ってその情報を処理する場合は、特に注意が必要である。その詳細は当社の技術情報の「JavaScript における URL エンコードの処理」という記事を見て頂きたい。クッキー設定ヘッダ行は何個でも設定できるが、サーブレット・コンテナは以下のブラウザを想定している:

- ウェブ・サーバあたり 20 クッキーまで受け付ける
- 総計 300 クッキーまで受け付ける
- 各クッキーのサイズは 4kB まで

```
if (cookieName != null && cookieValue != null) {
  Cookie cookie = new Cookie(cookieName, cookieValue);
  res.addCookie(cookie);
}
```

サーブレットでこのクッキーを取り出す方法は、HttpRequestDump.java のコードを参考にされたい。

9.3.7.1 Cookie のメソッドたち

以下は Cookie のメソッドたちである:

表 9-8:Cookie のメソッド一覧

コンストラクタ	内容
Cookie(java.lang.String name, java.lang.String value)	指定した名前と値を持った Cookie オブジェクトを生成
メソッド	**内容**
clone()	この Cookie のクローンを返す
getComment()	このクッキーのコメントを取得
getDomain()	このクッキーのドメイン名を返す

コンストラクタ	内容
getMaxAge()	このクッキーの最大有効期間を秒で返す
getName()	このクッキーの名前を返す
getPath()	ブラウザがこのクッキーを返す為のこのサーバへのパスを返す
getSecure()	ブラウザがセキュアなプロトコル経由でクッキーを返しているかを知る
getValue()	このクッキーの値を返す
getVersion()	このクッキーが対応しているプロトコルのバージョンを返す
isHttpOnly()	このクッキーが HttpOnly としてマークされているかどうかを知る
setComment(java.lang.String purpose)	このクッキーの目的を記述するコメントを指定
setDomain(java.lang.String domain)	このクッキーをブラウザが返すべきドメインを指定
setHttpOnly(boolean isHttpOnly)	このクッキーを HttpOnly としてマークする、またはマークを外す
setMaxAge(int expiry)	このクッキーの最大有効期間を秒で指定する
setPath(java.lang.String uri)	クライアントがこのクッキーを返すべきパスを指定する
setSecure(boolean flag)	ブラウザに対しこのクッキーを HTTPS や SSL のようなセキュアなプロトコル経由でおくるべきかどうかを指定
setValue(java.lang.String newValue)	このクッキーに新しい値をセットする
setVersion(int v)	このクッキーが対応するクッキー・プロトコルのバージョンを指定

クッキーも HTTP プロトコルでは他のヘッダ行と同じではあるが、サーブレット API では取り扱いに相違がある。これはクッキーには複数の属性があるからである:

1. 応答にヘッダを付加するときは次のように String の名前と値のペアを渡す
   response.addHeader("attributeName", "attributeValue");
2. しかしながらクッキーを応答に付加するときは、Cookie オブジェクトを渡し、名前と値のペアは Cookie のコンストラクタに渡す
   Cookie cookie = new Cookie("cookieName", "cookieValue");
   response.addCookie(cookie);
3. また、他のヘッダの場合は setHeader()と addHeader()の双方があるが、クッキーには setCookie()メソッドのみである。

9.3.7.2 CookieTest サーブレットによる確認

これらのメソッドで、ブラウザにはどのようなクッキー情報が渡され、またブラウザはどのようなクッキーを返すのかを知る為の簡単なサーブレット(CookieTest.java)を示す:

```
package session;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
```

```
/**
 * CookieTest は、Cookie のメソッドをテストするサーブレット
 * December 2010, CRESC Corps.
 */
@WebServlet("/CookieTest")
public class CookieTest extends HttpServlet {
   private static final long serialVersionUID = 1L;

   /**
    * 処理
    */
   public void performTask(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {

      Cookie cookie = new Cookie("MyCookieName", "MyCookieValue");
      cookie.setComment("Test Cookie");         // コメントセット
      cookie.setPath("/tutorial");          // パスを設定
      cookie.setHttpOnly(true);
      cookie.setMaxAge(60 * 3);             // 3 分間のテスト
      HttpSession session = req.getSession();      // セッション・クッキー設定
      session.setAttribute("MyCookie", (Object)cookie); // セットしたクッキーはセッションに保管
      session.setMaxInactiveInterval(60 * 5);      // 5 分間のテスト
      Cookie[] cookies = req.getCookies();      // クライアントがこのクッキーを返さないとき
      boolean hasMyCookie = false;          // 応答に set-cookie ヘッダ行を追加する
      if ((cookies != null) && (cookies.length > 0)) {
         for (int i = 0; i < cookies.length; i++) {
            Cookie ck = cookies[i];
            if (ck.getName().equals("MyCookieName")) hasMyCookie = true;
         }
      }
      if(!hasMyCookie) res.addCookie(cookie);
      res.setContentType("text/html; charset=windows-31j");// 応答ヘッダ Content-Type 追加
      PrintWriter out = res.getWriter();        // 出力バッファ取得
      dumpCookie(req, res, out);            // ダンプ処理のメソッドを呼ぶ
      out.println("</PRE></BODY></HTML>");      // HTML フッタの出力
      out.flush();                          // バッファの明示的吐き出し
      out.close();                          // バッファの明示的クローズ
   }

   /**
    * ダンプ処理の実体
    * @throws IOException
    */
   public void dumpCookie(HttpServletRequest req, HttpServletResponse res,
         PrintWriter out) throws IOException {
      // HTML ヘッダの出力
      out.println("<HTML><HEAD><TITLE>");
      out.println("クッキーの情報");
      out.println("</TITLE></HEAD>");
      // HTML ボディの出力
      out.println("<BODY><H1>要求のクッキー情報</H1><PRE>");
      Cookie[] cookies = req.getCookies();
      if ((cookies != null) && (cookies.length > 0)) {
         for (int i = 0; i < cookies.length; i++) {
            Cookie cookie = cookies[i];
            out.println(i + ":");
            out.println("  getName() : " + cookie.getName());
            out.println("  getValue() : " + cookie.getValue());
            if(cookie.getName().equals("MyCookieName")){
               HttpSession session = req.getSession(false);
               if ((session != null) && (session.getAttribute("MyCookie") != null) ){
                  cookie = (Cookie)session.getAttribute("MyCookie");
                  out.println("  セッション属性の cookie");
                  out.println("    isHttpOnly() : " + cookie.isHttpOnly());
                  out.println("    getDomain() : " + cookie.getDomain());
                  out.println("    getPath() : " + cookie.getPath());
                  out.println("    getMaxAge() : " + cookie.getMaxAge());
                  out.println("    getSecure() : " + cookie.getSecure());
                  out.println("    getVersion() : " + cookie.getVersion());
                  out.println();
```

```
                }
            }
        }
    }
}



/**
 * 到来HTTP GET要求の処理
 */
@Override
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws javax.servlet.ServletException, java.io.IOException {
   performTask(req, res);
}

/**
 * 到来HTTP POST要求の処理
 */
@Override
public void doPost(HttpServletRequest req, HttpServletResponse res)
throws javax.servlet.ServletException, java.io.IOException {
   performTask(req, res);
}

/**
 * 本サーブレット情報の文字列を返す
 */
@Override
public String getServletInfo() {
   return "HttpRequestDump, Version 2.0 by Cresc";
}
}
```

このサーブレットは Cookie cookie = new Cookie("MyCookieName", "MyCookieValue");という行で先ず Cookie のオブジェクトを用意し、これにコメント、パス情報、HttpOnly、及び 3 分間の有効期間をセットし、セッションに属性としてセットしている。クライアントからの要求に MyCookieName という名前のクッキーが存在していなかったら、このオブジェクトを要求オブジェクトにセットする。これは同じ名前のクッキーをセットすると、ブラウザは例え名前と値が同じであっても、別のものとして蓄積してしまい、ブラウザにクッキーが積み上がるのを防ぐ為である。セッション管理の為の具体的なコードは次の節で詳しく説明する。なお **Cookie のコンストラクタで使われる名前と値は、日本語を含む 2 バイト文字や危険な文字を含まないように注意**すること。そのような文字を使う場合は URL エンコードが必要になる。

プロキシを稼働させた状態でこのプログラムを http://localhost:12345/tutorial/CookieTest という具合にアクセスすると、次のような応答がブラウザに返されていることが、プロキシのデータから判る。

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=87BA3D1C33215E8164C7298BB3552981; Path=/tutorial; HttpOnly
Set-Cookie: MyCookieName=MyCookieValue; Version=1; Comment="Test Cookie"; Max-Age=180; Expires=Wed,
29-Dec-2010 08:04:43 GMT; Path=/tutorial; HttpOnly
Content-Type: text/html;charset=windows-31j
Transfer-Encoding: chunked
Date: Wed, 29 Dec 2010 08:01:43 GMT
```

この応答メッセージには 2 つの Set-Cookie ヘッダ行が含まれており、ひとつはセッション ID をセットする為に Tomcat が付けたものである。もうひとつはこのサーブレットがセットしたものである。これには名前と値以外にコメント("Test Cookie")、有効期間(Max-Age=180; Expires=Wed, 29-Dec-2010 08:04:43 GMT)、パス(Path=/tutorial)、及び HttpOnly が set-Cookie 行に追加されている。更に Version=1 もセットされている。

それでは、この Set-Cookie 行に対して、次にブラウザがこのサーブレットに返す要求メッセージのなかの Cookie ヘッダ行はどうなっているかというと、次のようにそっけないものになってしまっている:

```
Cookie: JSESSIONID=647D47DD57CBC1A1CA190B3B949A33C6; MyCookieName=MyCookieValue
```

つまり**ブラウザは名前と値のペアしか返さないので、サーブレット側ではブラウザが正しく自分が送った Set-Cookie ヘッダを受け取ったかを知る手段がない**。このサーブレットでは、自分が用意した Cookie オブジェクトをセッションの属性として保持し、それを HTML でクッキー情報として表示させている。下図はそのブラウザ画面である。この画面は 2 回目のアクセスで表示されるものである。初回ではセッションが確立されず、また MyCookieName という名前のクッキーもブラウザが持っていないので、表示するクッキー情報は存在しない。

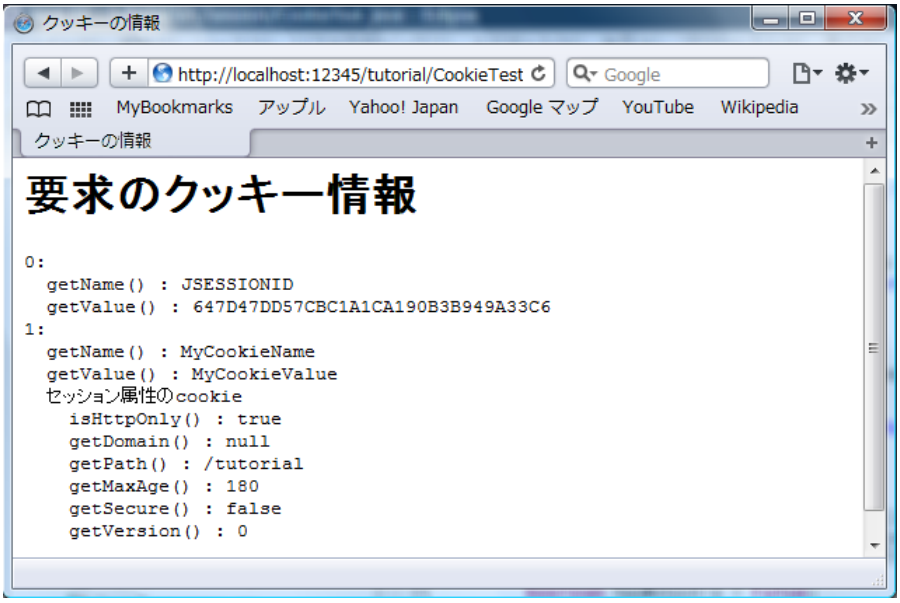


図 9-6:プロキシ経由での CookieTest の 2 回目のアクセスの結果

9.4節　簡単なサーブレットでセッション管理のメカニズムを確認する

*この節では、SimpleSessionTest というサーブレットで、実際にサーブレットがどのように管理されているかを確認する。このサーブレットは、そのクライアントとのセッションを確立し、その訪問回数をセッションの属性として保持する。訪問回数とセッションに関する情報を HTML テキストでそのクライアントに返す。**このサーブレットは、セッション管理のコードとしては基本的、また雛型的なもの**であるので、良く理解することが必要である。*

9.4.1　SimpleSessionTest.java のコード

以下にこのサーブレットのコードを示す。

```
package session;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import javax.servlet.http.HttpSession;

@WebServlet("/SimpleSessionTest")
public class SimpleSessionTest extends HttpServlet {
   /**
    * SimpleSessionTest は、セッションに関する基礎的な動作を学習するためのサーブレット
    * @author Cresc Corps.
    * @December 2010, CRESC Corps.
    */
      private static final long serialVersionUID = 1L;

      /**
       * 処理本体
       */

      public void performTask(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
         //到来要求に関するセッション情報をテキスト化
         StringBuffer sb  = new StringBuffer();
         ServletContext context = getServletContext();
         Utilities.dumpSession(req, sb, context);
         int visitTimes = 0;
         //セッションの取得または生成
         HttpSession session = req.getSession(true);
         //今回はそのセッションのタイムアウトは 1 分間とする。デフォルトは 30 分間
         session.setMaxInactiveInterval(60);
         if (session.getAttribute("visitTimes") != null){
            visitTimes = (Integer)session.getAttribute("visitTimes") + 1;
         }
         //セッションに属性をセット、Integer は Serializable インターフェイスを実装していることに注意
         session.setAttribute("visitTimes", (Integer)visitTimes);
         //リセット・ボタン時はそのセッションを無効にする
         if (req.getParameter("reset") != null){
            session.invalidate();
            visitTimes = 0;
         }
/*       //セッションが存在してもリンク等で訪問した場合にどうするかのオプション
         if ((req.getParameter("reset") == null) && (req.getParameter("proceed") == null)){
            session.invalidate();   //そのセッションを無効にする
            visitTimes = 0;         //訪問回数をリセットする
         }
*/
         //クッキーが使えないときに URL 書き換えを行う
         String url = null;
         String urlBase = req.getContextPath() + req.getServletPath();
         url = res.encodeURL(urlBase);

         //強制的に URL 書き換えを行う実験
//       url = urlBase + ";jsessionid=" + session.getId();

         //応答メッセージの作成
         res.setContentType("text/html; charset=windows-31j");//応答ヘッダ Content-Type 追加
         res.setHeader("Cache-Control", "no-cache"); //キャッシュを殺しておかないと分かりにくくなる
         PrintWriter out = res.getWriter();                //出力バッファ取得
         out.println("<HTML><HEAD><TITLE>");
         out.println("簡単なセッションのテスト</TITLE>");
         out.println(
         "<META HTTP-EQUIV=\"cache-control\" CONTENT=\"no-cache\">" +
         "<META HTTP-EQUIV=\"content-type\" CONTENT=\"text/html; charset=Windows-31J\"></HEAD>");
         out.println("<BODY><H1>簡単なセッションのテスト</H1><BR>");
         out.println(sb.toString());                       //セッション情報の出力
         out.println("<BR><BR>そのセッションでの現在の訪問回数：" + visitTimes);
         out.println("<BR><form method=\"get\" action=\"" + url + "\">" +
               "<input type=\"submit\" name=\"proceed\" value=\"次へ\">" +
               "<input type=\"submit\" name=\"reset\" value=\"セッションのリセット\"></form>");
         out.println("</BODY></HTML>");

         //SingleThreadModel による本サーブレットの複数のインスタンスにも対応可能なことの実験
//       Thread.sleep(10000L);
```

```
        out.flush();
    }

    /**
     * 到来HTTP GET要求の処理
     */
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws javax.servlet.ServletException, java.io.IOException {
        performTask(req, res);
    }

    /**
     * 到来HTTP POST要求の処理
     */
    @Override
    public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws javax.servlet.ServletException, java.io.IOException {
        performTask(req, res);
    }

    /**
     * 本サーブレット情報の文字列を返す
     */
    @Override
    public String getServletInfo() {
        return "SimpleSessionTest, Version 1.0 by Cresc";
    }
}
```

このコードの幾つかのポイントを示すと:

- このサーブレットは別途 Utilities.dumpSession(req, sb, context)というメソッドを呼んでいるが、これは到来要求に対するセッション情報を HTML テキストとして StringBuffer に返すものである。このメソッドではセッション生成を行わないよう getSession(false)でセッションの有無を調べている。なおこのメソッドは、サーブレット・パス等を知るために Context を渡してもらう必要がある。

- セッションの生成(初めての場合)はその後に行っている。セッションの生成や応答ヘッダの書き込みはボディ部に HTML テキストを書きだす前に行われねばならない。getSession メソッドは、それをもとに Set-Cookie ヘッダを応答ヘッダに付すかどうかを判断する。

- セッションの最大有効時間(セッションに対する何らかの操作が行われていない最大時間)を 1 分間として、その動作を確認し易くしている。**デフォルトではこのタイムアウト値は Tomcat では 30 分**である。セッションのオブジェクトが増えてサーブレット・エンジンのオーバヘッドが増えるのを防ぐ為にも、支障が無い範囲でこの時間を短くすることが好ましい。この時間とブラウザが保持したクッキーの有効期間とは関係がないことに注意されたい。**Tomcat は Set-Cookie ヘッダに Expires 属性で有効期限を指定する訳ではない**。但し**殆どのブラウザでは、後述のように、Expires 属性の無いクッキーはセッション用だと判断して、そのブラウザを閉じるときにそれらのクッキーを削除している**。従ってブラウザはサーブレットが無効にしたセッション ID のクッキーを要求メッセージに付けてくる可能性がある。なお、同じ名前のクッキーを Set-Cookie でブラウザに送ると、その値は新しいものに更新される。
  また**サーブレット・エンジンによっては Expires 属性をセットするものもあるので、注意が必要である。**
  IBM の WebSphere では、HTTP 応答に Set-Cookie ヘッダーが付与されるだけでなく、この時、特定のバージョン/メンテナンスレベルでは、Expires ヘッダーと Cache-Control: no-cache="set-cookie, set-cookie2"ヘッダーが付与される。10 年前にアップロードしたサーブレット・チュートリアルの「セッション管理のメカニズムを理解する為の実験」のところで示したように、実は IBM の WebSphere サーバ(WAS)では、次のようなヘッダ行を付加していた。特に Expires 属性の日付は明らかなバグであったが、最近これが問題を起こした。

```
Set-Cookie: sesessionid=LV150HYAAAABZQFITEHUD1Y;Path=/
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

- 生成された HttpSession オブジェクトに対し、session.setAttribute("visitTimes", (Integer)visitTimes);でそのセッションでの訪問回数を Integer 型のオブジェクトとして visitTimes という名前でバインドしている。「セッションとは」の節で述べたように、配布可能なアプリケーションを作る場合には、セッションの属性として含められたオブジェクトは総てシリアル化可能な(即ち java.io.Serializable インターフェイスを実装している)オブジェクトのみにしておく必要がある。Integer はこのインターフェイスを実装しているが、何らかの新しいバインドするクラスを作成するときは、**必ず java.io.Serializable インターフェイスを実装するのは良い習慣である**。

- ブラウザ側でクッキーを受け付けないように設定されている場合に備え、String url = res.encodeURL(urlBase);という具合に **encodeURL メソッドを通す**。これにより、サーブレット・エンジンはセッション ID が:
  - セッションが確立されていない(isNew()==true)状態で、何らかのセッション ID がクッキー経由で送られていない
  - 渡したセッション ID がクッキーではなくて URL 経由で返されている

  場合は、指定された文字列にセッション ID を付加する。その文字列をどのようにクライアントに渡すかは、サーブレットの責任である。クッキーが使えるかどうかを判断する為には:
  - 何らかのセッション ID がクッキー経由で送られている場合はクッキーが使えると判断する。そうでないときは:
  - 最初にクッキーと URL 書き換えの双方の手段をクライアントに渡し、どちらでセッション ID が返ってくるかでその後どちらを使うか判断する。この場合はセッション ID はブラウザのアドレスバーに表示されることになる。
  - セキュリティが問題な場合は、最初にクッキーを送ってみて、クッキーが返ってこなかったらクッキーが使えないことを知る。通常このような場合はクライアントに対し、クッキーをオンにするように画面で指示するか、サービスをしないことが多い。
  - 欧州ではセキュリティに対する考え方が厳格であり、クッキーを受付けないようにブラウザを設定しているユーザが多いので、欧州向けのアプリケーションは注意が必要である。

- res.setHeader("Cache-Control", "no-cache");でブラウザ及びプロキシでのキャッシングをさせないようにする。これは**ブラウザがキャッシュされたページを表示することでセッションの推移が判らなくなることを避ける**ためである。**セッションを使うアプリケーションでは、キャッシュをさせない**のが一般的である。**サーブレット・エンジンによっては Set-Cookie ヘッダとともにこの Cash-Control ヘッダを自動的に付加する**ので、注意されたい。**IBM の WebShere では、特定のバージョン/メンテナンスレベルでは、Expires ヘッダーと Cache-Control: no-cache="set-cookie, set-cookie2" ヘッダーが付与される。**

- res.getWriter();で取得した **PrintWriter にテキストを書き込むのは、ヘッダまわりの処理が終わったあと**にまとめて行うべきである。

- Thread.sleep(10000L);と 10 秒間のスリープをいれるコメントアウトされた行があるが、これは後ほど複数のスレッドにも対応できることを試すために挿入されている。

なお、Utilities.dumpSession(req, sb, context)のコードは次のようになっている:

```
   // Generates an HTML text describing Session information fore the request.
   // This method does not create an new session object.
   public static void dumpSession(HttpServletRequest req, StringBuffer out, ServletContext context)
throws IOException {

      out.append("<BR>要求オブジェクトのなかのセッション関連情報");
      out.append("<BR>getAuthType: " + req.getAuthType());
      out.append("<BR>getMethod: " + req.getMethod());
      out.append("<BR>getPathInfo: " + req.getPathInfo());
      out.append("<BR>getPathTranslated: " + req.getPathTranslated());
      out.append("<BR>getProtocol: " + req.getProtocol());
```

```
      out.append("<BR>getQueryString: " + req.getQueryString());
      out.append("<BR>getEffectiveSessionTrackingModes:");
      Set<SessionTrackingMode> stms = context.getEffectiveSessionTrackingModes();
      for (SessionTrackingMode c : stms)
          out.append("  " + c);
      out.append("<BR>getRequestedSessionId: " + req.getRequestedSessionId());
      out.append("<BR>isRequestedSessionIdValid: " + (new
Boolean(req.isRequestedSessionIdValid())).toString());
      out.append("<BR>isRequestedSessionIdFromCookie: " + (new
Boolean(req.isRequestedSessionIdFromCookie())).toString());
      out.append("<BR>isRequestedSessionIdFromURL: " + (new
Boolean(req.isRequestedSessionIdFromURL())).toString());
      out.append("<BR>isSecure: " + (new Boolean(req.isSecure())).toString());
      out.append("<BR>getServletPath: " + req.getServletPath());
      out.append("<BR>getContextPath: " + req.getContextPath());
      out.append("<BR>");

      out.append("<BR>クッキー(Cookies):");
      Cookie[] cookies = req.getCookies();
      if ((cookies != null) && (cookies.length > 0)) {
         for (int i = 0; i < cookies.length; i++) {
            String name = cookies[i].getName();
            String value = cookies[i].getValue();
            out.append("<BR>  " + name + " : " + value);
         }
      }

      HttpSession session = req.getSession(false);
      if (session != null) {
         out.append("<BR><BR>その要求に対応するセッションの情報:");
         out.append("<BR>isNew: " + session.isNew());
         out.append("<BR>getId: " + session.getId());
         out.append("<BR>getCreationTime: " + new Date(session.getCreationTime()));
         out.append("<BR>getLastAccessedTime: " + new Date(session.getLastAccessedTime()));
         out.append("<BR>getMaxInactiveInterval in sec.: " + session.getMaxInactiveInterval());
         out.append("<BR>getServletContext: " +
session.getServletContext().getServletContextName());
         out.append("<BR>属性:");
         Enumeration<?> attributeNames = session.getAttributeNames();
         while (attributeNames.hasMoreElements()) {
            String name = (String) attributeNames.nextElement();
            out.append("<BR>  " + name + " : " +session.getAttribute(name) .toString());
         }
         out.append("<BR>");
      }
      out.append("<BR>***  以上  ***<BR>");
   }
```

このメソッドは、HttpSession session = req.getSession(false);という具合に、新たなセッション・オブジェクトを生成させないようにしている。

9.4.2　クッキーを受け付けなくしたブラウザでのセッション確立の確認

9.4.2.1　ブラウザの設定を変更してクッキーをブロックする

この実験を始める前に、自分のブラウザの Cookie に関する設定を変更し、Cookie セットをブロックする。各ブラウザの設定変更は次のようである。

Internet Explorer では、「ツール」→「インターネット・オプション」→「プライバシー」のペインで図のようにプライバシーの保護を最大にする。

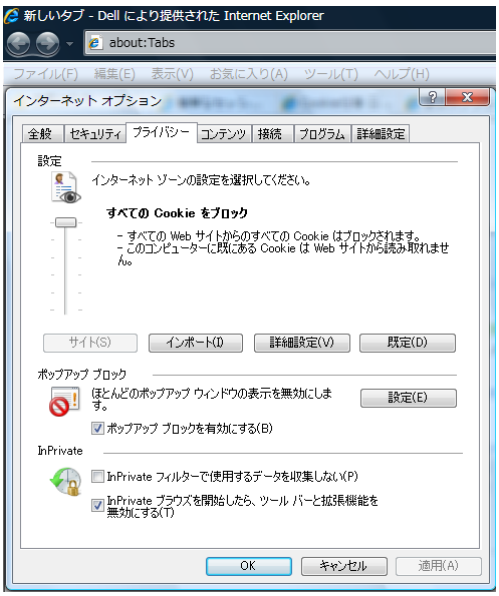


図 9-7：Internet Explorer での Cookie のブロック

Mozilla Firefox では、「ツール」→「オプション」→「プライバシー」のペインで、「経歴」のブロックの「Firefox に」の選択ボックスの「記憶させる経歴を詳細設定する」を選択、「サイトから送られてきた Cookie を保存する」のチェックを外す。

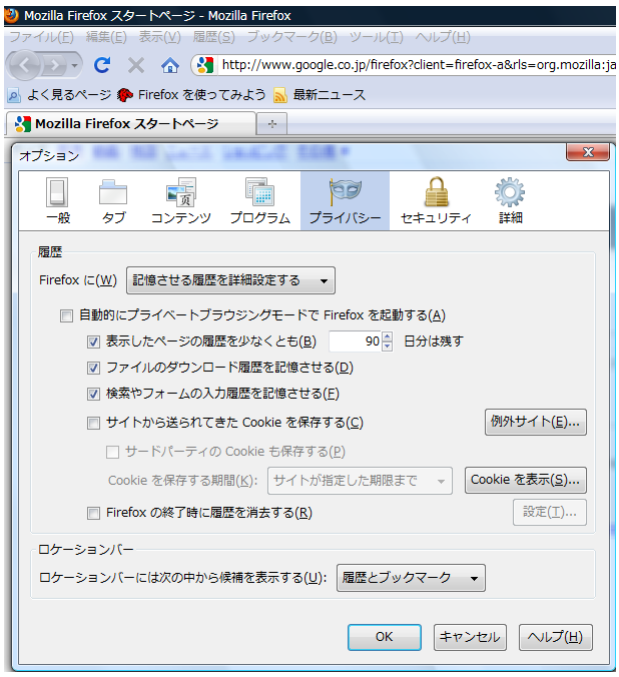


図 9-8：Mozilla Firefox でのクッキーのブロック

Google Chrome では、「Google Chrome の設定」→「オプション」→「高度な設定」→「コンテンツの設定」のダイヤログで、「サイトからのデータ設定を総てブロックする」を選択する。

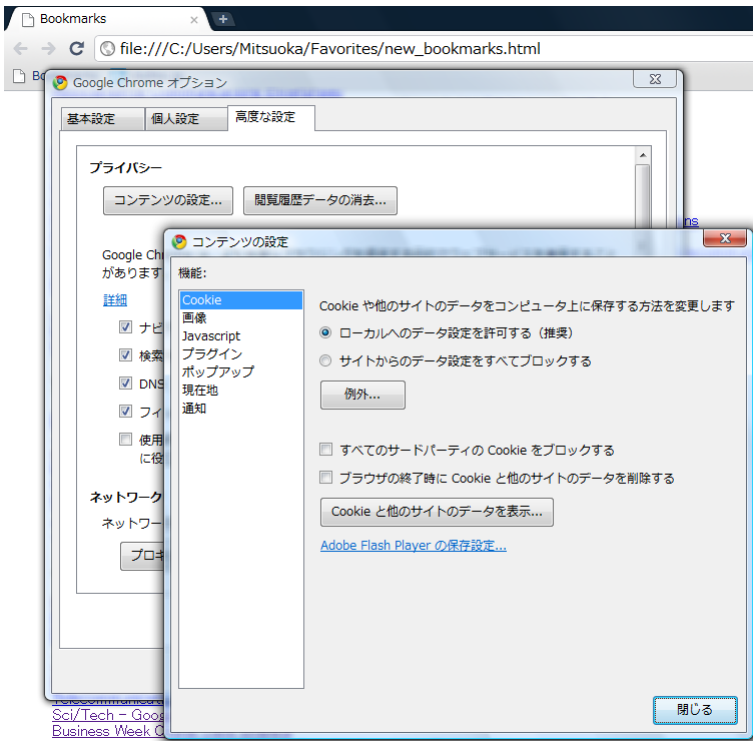


図 9-9：Google Chrome でのクッキーのブロック

Apple Safari では「一般設定」→「設定」→「セキュリティ」→「Cookie の受け入れ」で、「しない」を選択する。

図 9-10：Apple Safari でのクッキーのブロック

9.4.2.2　MINA Proxy と Tomcat 7 を Eclipse 上で開始させる

MINA Proxy の開始は添付資料「MINA プロキシのダウンロードと実行」の節の「プロキシの開始」の項を見て頂きたい。Tomcat 7 の開始は説明の要がないが、「Tomcat 7 の開始と停止」の項に説明してある。プロキシがたち上がったことは「ポート 12345 で待機中」とコンソールに表示されることでわかる。Tomcat 7 が立ち上がっていることは「情報: Server startup in xxx ms」とコンソールに表示される、あるいはサーバのタブで「Started, Synchronized」と表示されることでわかる。

9.4.2.3　SimpleSessionTest へのアクセス

この状態で、自分のブラウザのアドレス・バーに http://localhost:12345/tutorial/SimpleSessionTest を入力して、プロキシ経由で SimpleSessionTest サーブレットをアクセスする。

9.4.2.4　Mozilla Firefox での交信例

どのブラウザでも同じ結果が得られるが、ここでは Mozilla Firefox でプロキシ経由で SimpleSessionTest にアクセスした場合を示すと、次のような初期画面になる:

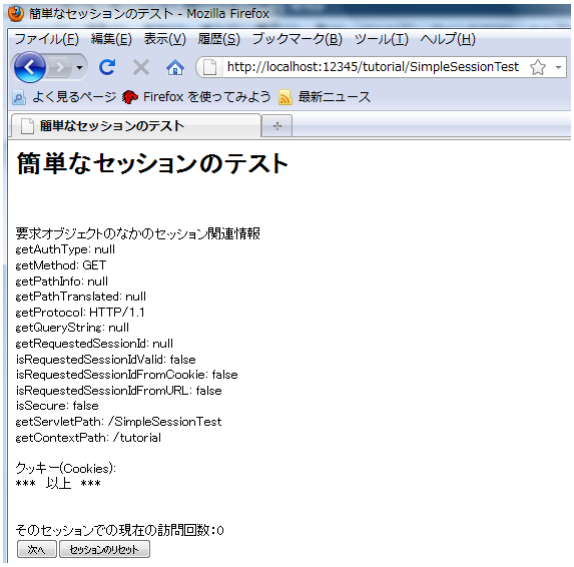


図 9-11 : SimpleSessionTest の初期画面

「次へ」をクリックするとそれ以降は次のような画面になる:

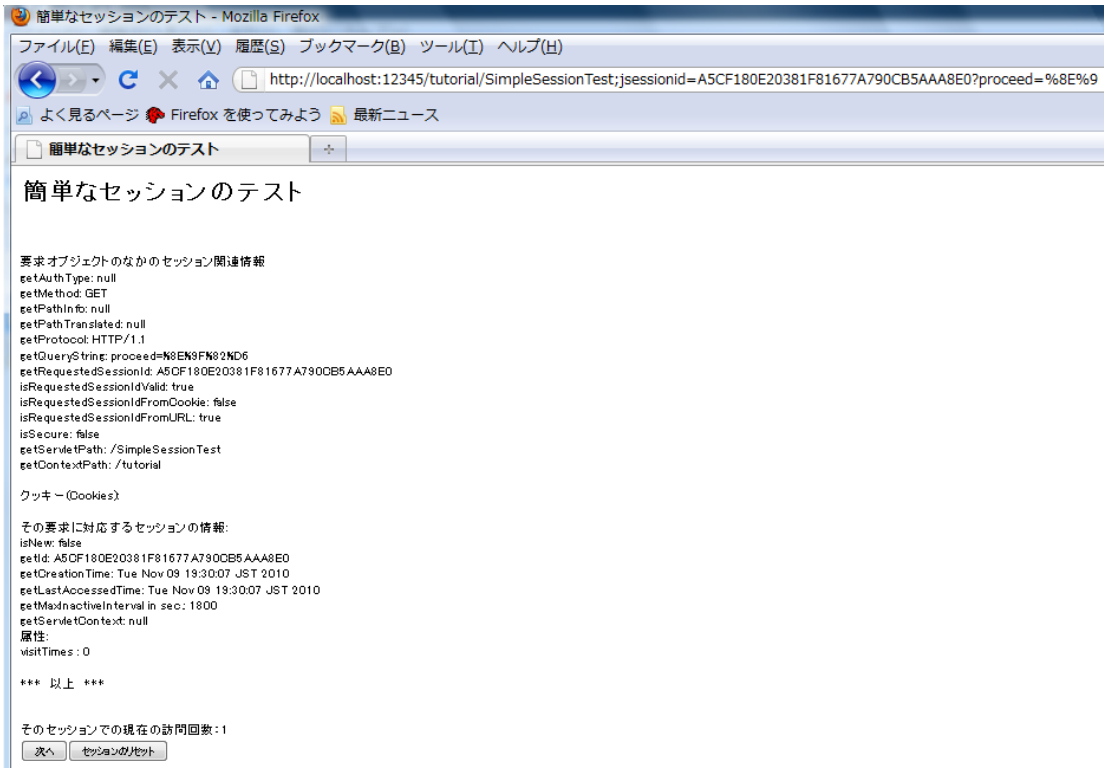


図 9-12 : 2 回目以降の画面

ここで注意することは、**アドレス・バーに表示される URL に、;jsessionid=A5CF180E20381F81677A790CB5AAA8E0 と、セッション ID が付加されている**ことである。これらの経過は、プロキシの記録で正確に把握できる。

初回の要求と応答:

HTTP 要求メッセージ

```
GET /tutorial/SimpleSessionTest HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3 (
.NET CLR 3.5.30729; .NET4.0C)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
```

HTTP 応答メッセージ

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=A5CF180E20381F81677A790CB5AAA8E0; Path=/tutorial; HttpOnly
Cache-Control: no-cache
Content-Type: text/html;charset=windows-31j
Transfer-Encoding: chunked
Date: Tue, 09 Nov 2010 10:30:07 GMT

3d8
<HTML><HEAD><TITLE>
簡単なセッションのテスト</TITLE>
<META HTTP-EQUIV="cache-control" CONTENT="no-cache"><META HTTP-EQUIV="content-type"
CONTENT="text/html; charset=Windows-31J"></HEAD>
<BODY><H1>簡単なセッションのテスト</H1><BR>
<BR>要求オブジェクトのなかのセッション関連情報<BR>getAuthType: null<BR>getMethod: GET<BR>getPathInfo:
null<BR>getPathTranslated: null<BR>getProtocol: HTTP/1.1<BR>getQueryString:
null<BR>getRequestedSessionId: null<BR>isRequestedSessionIdValid:
false<BR>isRequestedSessionIdFromCookie: false<BR>isRequestedSessionIdFromURL: false<BR>isSecure:
false<BR>getServletPath: /SimpleSessionTest<BR>getContextPath: /tutorial<BR><BR>クッキー
(Cookies):<BR>***  以上  ***<BR>
<BR><BR>そのセッションでの現在の訪問回数：0
<BR><form method="get"
action="/tutorial/SimpleSessionTest;jsessionid=A5CF180E20381F81677A790CB5AAA8E0"><input
type="submit" name="proceed" value="次へ"><input type="submit" name="reset" value="セッションのリセット
"></form>
</BODY></HTML>

0
```

最初の応答では、サーブレットはその要求は isNew であることを知り、A5CF180E20381F81677A790CB5AAA8E0 という ID を持った HttpSession のオブジェクトを生成する。その際サーブレット・エンジンは Set-Cookie: JSESSIONID=A5CF180E20381F81677A790CB5AAA8E0; Path=/tutorial; HttpOnly というヘッダを応答メッセージの一部としてセットする。

ここで注意することは、HttpOnly というプロパティであって、これを使用すると、Cookie の盗難につながるサイト間スクリプトの脅威を軽減できる。この **HttpOnly プロパティに対応したブラウザでは、クライアント側のスクリプトからのこのクッキーへのアクセスが禁止されるので、セキュリティが向上する。**

新たにセッションが作られた場合は、サーブレット・エンジンはそのクライアントがクッキーに対応しているかどうか判らないので、encodeURL メソッドでは URL に;jsessionid=A5CF180E20381F81677A790CB5AAA8E0 という文字列を追加する。この場合は jsessionid と小文字になっている。

2 回目以降の要求と応答のメッセージ:

HTTP 要求メッセージ

```
GET /tutorial/SimpleSessionTest;jsessionid=A5CF180E20381F81677A790CB5AAA8E0?proceed=%8E%9F%82%D6
HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3 (
.NET CLR 3.5.30729; .NET4.0C)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://localhost:12345/tutorial/SimpleSessionTest
```

HTTP 応答メッセージ

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Cache-Control: no-cache
Content-Type: text/html;charset=windows-31j
Transfer-Encoding: chunked
Date: Tue, 09 Nov 2010 10:30:11 GMT

532
<HTML><HEAD><TITLE>
簡単なセッションのテスト</TITLE>
<META HTTP-EQUIV="cache-control" CONTENT="no-cache"><META HTTP-EQUIV="content-type"
CONTENT="text/html; charset=Windows-31J"></HEAD>
<BODY><H1>簡単なセッションのテスト</H1><BR>
<BR>要求オブジェクトのなかのセッション関連情報<BR>getAuthType: null<BR>getMethod: GET<BR>getPathInfo:
null<BR>getPathTranslated: null<BR>getProtocol: HTTP/1.1<BR>getQueryString: proceed=%8E%9F
%82%D6<BR>getRequestedSessionId: A5CF180E20381F81677A790CB5AAA8E0<BR>isRequestedSessionIdValid:
true<BR>isRequestedSessionIdFromCookie: false<BR>isRequestedSessionIdFromURL: true<BR>isSecure:
false<BR>getServletPath: /SimpleSessionTest<BR>getContextPath: /tutorial<BR><BR>クッキー
(Cookies):<BR><BR>その要求に対応するセッションの情報:<BR>isNew: false<BR>getId:
A5CF180E20381F81677A790CB5AAA8E0<BR>getCreationTime:
711221 [NioProcessor-7] INFO MINA_proxy.AbstractProxyIoHandler -  Tue Nov 09 19:30:07 JST
2010<BR>getLastAccessedTime: Tue Nov 09 19:30:07 JST 2010<BR>getMaxInactiveInterval in sec.:
1800<BR>getServletContext: null<BR>属性:<BR>  visitTimes : 0<BR><BR>***  以上  ***<BR>
<BR><BR>そのセッションでの現在の訪問回数：1
<BR><form method="get"
action="/tutorial/SimpleSessionTest;jsessionid=A5CF180E20381F81677A790CB5AAA8E0"><input
type="submit" name="proceed" value="次へ"><input type="submit" name="reset" value="セッションのリセット
"></form>
</BODY></HTML>

0
```

2 回目以降の応答メッセージには、もはやサーブレット・エンジンは Set-Cookie ヘッダをセットしない。セッション ID は URL から来ていることが判ったので、encodeURL メソッドは引き続き URL に;jsessionid=A5CF180E20381F81677A790CB5AAA8E0 という文字列を追加する。

このようにして URL 経由でのそのクライアントとのセッション維持がなされる。セッションが解放されるのは:

- 要求/応答がタイムアウトを超えるまで起きなかった
- そのクライアントが「セッションのリセット」をクリックした

場合である。クライアントが「セッションのリセット」をクリックした場合は、その時点で該セッションが廃棄されるので、次回アクセスしたときは isNew の状態になり、初期画面が表示されることになる。

9.4.2.5　クッキーを受け付けないブラウザでのシーケンス図

以上の結果をまとめると以下のようなシーケンス図になる:

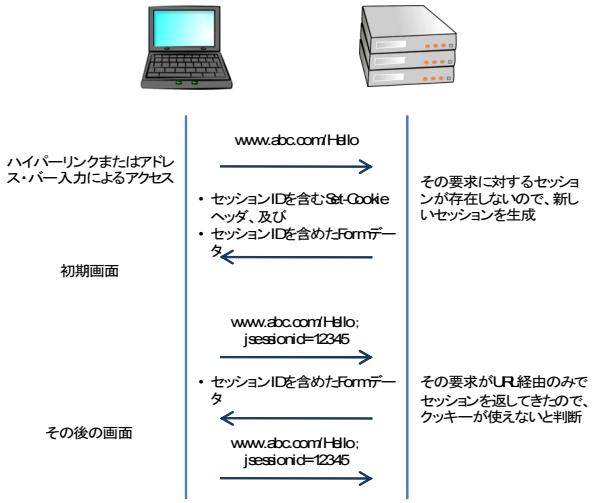


図 9-13:クッキーを受け付けないブラウザでのシーケンス図

9.4.3　ブラウザがクッキーを受け付ける状態でのセッションの確立と維持の確認

ブラウザの設定をクッキーを受け付けるようにして、同じような実験を行ってみよう。

9.4.3.1　交信結果

以下は同じく Mozilla Firefox で**セッションを受け付ける**よう設定を戻した時の交信結果である。

初回の要求と応答:

HTTP 要求メッセージ

```
GET /tutorial/SimpleSessionTest HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3 (
.NET CLR 3.5.30729; .NET4.0C)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
```

HTTP 応答メッセージ

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=6C89332C5B53280AACD3264627B0F1E0; Path=/tutorial; HttpOnly
Cache-Control: no-cache
Content-Type: text/html;charset=windows-31j
Transfer-Encoding: chunked
Date: Tue, 09 Nov 2010 11:58:09 GMT

3d8
<HTML><HEAD><TITLE>
簡単なセッションのテスト</TITLE>
<META HTTP-EQUIV="cache-control" CONTENT="no-cache"><META HTTP-EQUIV="content-type"
CONTENT="text/html; charset=Windows-31J"></HEAD>
<BODY><H1>簡単なセッションのテスト</H1><BR>
<BR>要求オブジェクトのなかのセッション関連情報<BR>getAuthType: null<BR>getMethod: GET<BR>getPathInfo:
```

```
null<BR>getPathTranslated: null<BR>getProtocol: HTTP/1.1<BR>getQueryString:
null<BR>getRequestedSessionId: null<BR>isRequestedSessionIdValid:
false<BR>isRequestedSessionIdFromCookie: false<BR>isRequestedSessionIdFromURL: false<BR>isSecure:
false<BR>getServletPath: /SimpleSessionTest<BR>getContextPath: /tutorial<BR><BR>クッキー
(Cookies):<BR>***  以上  ***<BR>
<BR><BR>そのセッションでの現在の訪問回数：0
<BR><form method="get"
action="/tutorial/SimpleSessionTest;jsessionid=6C89332C5B53280AACD3264627B0F1E0"><input
type="submit" name="proceed" value="次へ"><input type="submit" name="reset" value="セッションのリセット
"></form>
</BODY></HTML>

0
```

これはクッキーを受け付けなくしたブラウザの場合とまったく同じである。

サーブレットはその要求は isNew であることを知り、6C89332C5B53280AACD3264627B0F1E0 という ID を持った HttpSession のオブジェクトを生成する。その際サーブレット・エンジンは Set-Cookie: JSESSIONID=6C89332C5B53280AACD3264627B0F1E0; Path=/tutorial; HttpOnly というヘッダを応答メッセージの一部としてセットする。

新たにセッションが作られた場合は、サーブレット・エンジンはそのクライアントがクッキーに対応しているかどうか判らないので、encodeURL メソッドでは URL に;jsessionid=6C89332C5B53280AACD3264627B0F1E0 という文字列を追加する。この場合は jsessionid と小文字になっている。

2 回目の要求と応答:

HTTP 要求メッセージ

```
GET /tutorial/SimpleSessionTest;jsessionid=6C89332C5B53280AACD3264627B0F1E0?proceed=%8E%9F%82%D6
HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3 (
.NET CLR 3.5.30729; .NET4.0C)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://localhost:12345/tutorial/SimpleSessionTest
Cookie: JSESSIONID=6C89332C5B53280AACD3264627B0F1E0
```

HTTP 応答メッセージ

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Cache-Control: no-cache
Content-Type: text/html;charset=windows-31j
Transfer-Encoding: chunked
Date: Tue, 09 Nov 2010 11:58:19 GMT

539
<HTML><HEAD><TITLE>
簡単なセッションのテスト</TITLE>
<META HTTP-EQUIV="cache-control" CONTENT="no-cache"><META HTTP-EQUIV="content-type"
CONTENT="text/html; charset=Windows-31J"></HEAD>
<BODY><H1>簡単なセッションのテスト</H1><BR>
<BR>要求オブジェクトのなかのセッション関連情報<BR>getAuthType: null<BR>getMethod: GET<BR>getPathInfo:
null<BR>getPathTranslated: null<BR>getProtocol: HTTP/1.1<BR>getQueryString: proceed=%8E%9F
%82%D6<BR>getRequestedSessionId: 6C89332C5B53280AACD3264627B0F1E0<BR>isRequestedSessionIdValid:
true<BR>isRequestedSessionIdFromCookie: true<BR>isRequestedSessionIdFromURL: false<BR>isSecure:
false<BR>getServletPath: /SimpleSessionTest<BR>getContextPath: /tutorial<BR><BR>クッキー
(Cookies):<BR>  JSESSIONID : 6C89332C5B53280AACD3264627B0F1E0<BR><BR>その要求に対応するセッションの情
報:<BR>isNew: false<BR>getId: 6C89332C5B53280AACD3264627B0F1E0<BR>getCreationTime: Tue Nov 09
```

```
20:58:09 JST 2010<BR>getLastAccessedTime: Tue Nov 09 20:58:09 JST 2010<BR>getMaxInactiveInterval in
sec.: 1800<BR>getServletContext: null<BR>属性:<BR>  visitTimes : 0<BR><BR>***  以上  ***<BR>
<BR><BR>そのセッションでの現在の訪問回数：1
<BR><form method="get" action="/tutorial/SimpleSessionTest"><input type="submit" name="proceed"
value="次へ"><input type="submit" name="reset" value="セッションのリセット"></form>
</BODY></HTML>

0
```

Tomcat は URL とクッキーと双方でセッション ID が送られてきた場合は、クッキーを優先する。従って isRequestedSessionIdFromURL: false となり、encodeURL メソッドは URL にセッション ID を付加しない。またセッション ID がクッキーとしてクライアントに蓄積されたことが判ったので、**応答メッセージにはもはや Set-Cookie ヘッダは付けられない。**

3 回目以降の要求メッセージ

```
GET /tutorial/SimpleSessionTest?proceed=%8E%9F%82%D6 HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3 (
.NET CLR 3.5.30729; .NET4.0C)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer:
http://localhost:12345/tutorial/SimpleSessionTest;jsessionid=6C89332C5B53280AACD3264627B0F1E0?
proceed=%8E%9F%82%D6
Cookie: JSESSIONID=6C89332C5B53280AACD3264627B0F1E0
```

3 回目からは要求メッセージの URL にはセッション ID が付加されない。**ブラウザからはクッキーのヘッダのみが送られる。**

9.4.3.2　クッキーを受け付けるブラウザでのシーケンス図

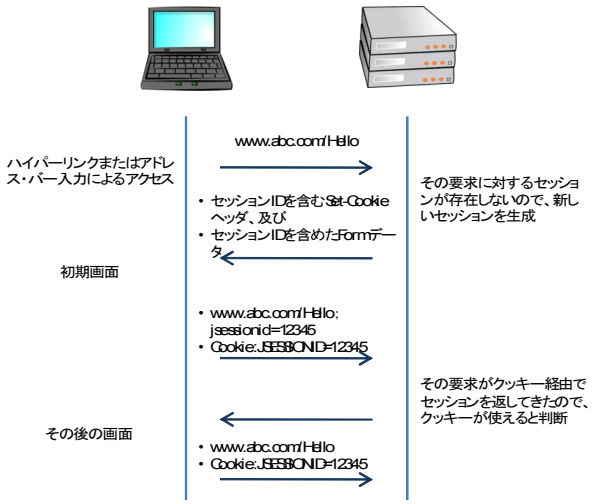


図 9-14：クッキーを受け付けるブラウザでのシーケンス図

9.4.4　ブラウザにおけるクッキーの保持

一般的には Expires 属性による期限指定がないクッキーに対しては、最近のブラウザたちはそれをセッション・

クッキーだとみなして、そのブラウザのセッションの終了時点（そのブラウザのインスタンスが閉じる時点）でそれらのクッキーを削除している。Internet Explorer、 Opera、Safari、及び Chrome の総てがそのような動作になっている。

セッション情報を長期間ブラウザに残したいアプリケーションでは、別の情報の形で Set-Cookie:ヘッダを付加してやる必要がある（クッキーでクライアントに情報を蓄積させる場合は、渡す名前と値に英数文字以外を使う時は必ず URL エンコードする）。setMaxInactiveInterval(int interval)メソッドは Set-Cookie:ヘッダに影響を与えない。

しかしながら、**Firefox (3.0.9 時点) ではそのような規則に準じておらず、そのブラウザを閉じたときだけでなく OS を再スタートしたときでもそのクッキーは存続する**という報告がある。これは Firefox の設計によるもので、Firefox を閉じるときにタブを保存するかどうかを聞いてくるが、「保存して終了」を選択すると再立ち上げしたときにこれまでの状態に復旧する。これを「セッション復旧(session restore)」と呼んでいる。もしそれを望まないときは、タブの総てを閉じてからブラウザを閉じれば良い。

実際そうなっているかを SimpleSessionTest のコードのタイムアウト時間設定をデフォルトの 30 分間の状態にして試してみよう。

1. Internet Explorer の場合：
    1. PC 上でひとつのブラウザを開いた場合
        1. ブラウザを閉じるとセッション ID のクッキーは削除される
        2. 2 つのタブで同じ SimpleSessionTest をアクセスした場合は、セッション ID のクッキーはタブ間で共有される。一方のタブを閉じてもセッション ID のクッキーは保持される。
    2. 同じ PC 上で 2 つのブラウザを開いた場合 (Ctrl＋N キー)
        1. 片方のブラウザで蓄積した ID のクッキーが、他のブラウザでも共有される
        2. その状態で片方のブラウザを閉じてもセッション ID のクッキーは保持される。

2. Mozilla Firefox の場合：
    1. PC 上でひとつのブラウザを開いた場合
        1. **「保存して終了」でこのブラウザを閉じると、セッション ID のクッキーは維持される。**
        2. 保存しないで「終了」でブラウザを閉じるとセッション ID のクッキーは削除される
        3. 2 つのタブで同じ SimpleSessionTest をアクセスした場合は、セッション ID のクッキーはタブ間で共有される。一方のタブを閉じてもセッション ID のクッキーは保持される。
    2. 同じ PC 上で 2 つのブラウザを開いた場合
        1. 片方のブラウザで蓄積した ID のクッキーが、他のブラウザでも共有される
        2. その状態で片方のブラウザを閉じてもセッション ID のクッキーは保持される。

3. Google Chrome の場合：
    1. PC 上でひとつのブラウザを開いた場合
        1. ブラウザを閉じるとセッション ID のクッキーは削除される
        2. 2 つのタブで同じ SimpleSessionTest をアクセスした場合は、セッション ID のクッキーはタブ間で共有される。一方のタブを閉じてもセッション ID のクッキーは保持される。
    2. 同じ PC 上で 2 つのブラウザを開いた場合
        1. 片方のブラウザで蓄積した ID のクッキーが、他のブラウザでも共有される
        2. その状態で片方のブラウザを閉じてもセッション ID のクッキーは保持される。

4. Apple Safari の場合：
    1. PC 上でひとつのブラウザを開いた場合
        1. ブラウザを閉じるとセッション ID のクッキーは削除される
        2. 2 つのタブで同じ SimpleSessionTest をアクセスした場合は、セッション ID のクッキーはタブ間

で共有される。一方のタブを閉じてもセッション ID のクッキーは保持される。

2. 同じ PC 上で 2 つのブラウザを開いた場合
   1. 片方のブラウザで蓄積した ID のクッキーが、他のブラウザでも共有される
   2. その状態で片方のブラウザを閉じてもセッション ID のクッキーは保持される。

このように、確かに Mozilla Firefox の「セッション復旧(session restore)」により、セッション ID のクッキーが蓄積されたままになることが判る。従って、サーブレット側はこれに対処する必要がある。即ち:

- セッションのタイムアウト時間を適正に選択する
- そのユーザとのセッションが終わったら直ちに invalidate メソッドでそのセッションを無効にする
- セッション ID のクッキーのヘッダ付きで初期画面がアクセスされた場合は、その前にそのアプリケーションをアクセスしていたクライアントであるので、その情報は活用できる場合もあろう。

もし**同じ PC 上で複数のブラウザを立ち上げたり、あるいは複数のタブを使って、同じアプリケーションを独立してアクセスしたい場合には、クッキーを使ったセッション管理は使えない**。そのようなことはテスト時以外はあまりないだろうが、URL 書き換えを使うことになる。

9.4.5 クッキーをセッションに使わないように Tomcat を設定する

クッキーを使ったセッション管理では、前項あるいは「クッキー(Cookies)」の項で説明したように、問題がある。従って、ブラウザ側ではクッキーを受け付ける設定になっていても、サーブレット・エンジンでそのクッキーをセッション管理に使わないよう設定することが出来る。

9.4.5.1 context.xml による方法

その為には、Eclipse 上で tutorial\WebContent\META-INF のディレクトリに、次のような内容の context.xml という名前の設定ファイルを置けばよい:

```
<?xml version='1.0' encoding='UTF-8'?>
<Context path='/tutorial' cookies='false'>
  <!-- other settings -->
</Context>
```

このように設定した場合の SimpleSessionTest サーブレットへのアクセス例は次のようになる:

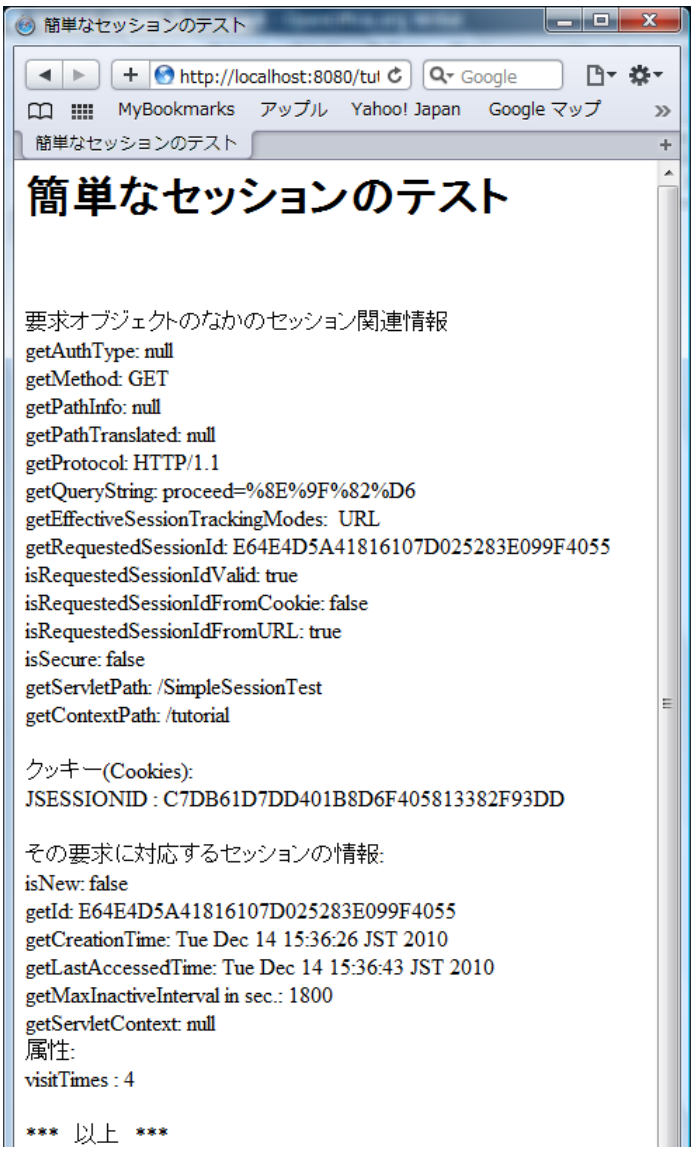


図 9-15：Tomcat をクッキーを使わないよう設定した場合の例

この図で判るように、

- getEffectiveSessionTrackingModes: URL

と、COOKIE が消えており、またブラウザからは JSESSIONID : C7DB61D7DD401B8D6F405813382F93DD というクッキーが送られているものの、同時に URL からは jsessionid=E64E4D5A41816107D025283E099F4055 という別の値の ID が送られており、Tomcat は

- isRequestedSessionIdFromCookie: false 及び
- isRequestedSessionIdFromURL: true

として URL のほうを採用している。Tomcat からはセッション ID を含むクッキーのヘッダ行(Set-Cookie)は送信されないことを読者はプロキシで確認されたい。

9.4.5.2 setSessionTrackingModes による方法

サーブレット 3.0 ではフレームワークとの親和性を考えて、プログラム的にセッション追跡モードの設定が可能になっている。設定はこの ServletContext を初期化するときに行われねばならない。ServletContext が走ってしまっているときは変更はできない。従って設定は：

1. ServletContextListner の contextInitialized メソッド
2. ServletContextInitializer の onStartup メソッド

でそのアプリケーションを初期化するときにのみ可能である。

以下はその為の ServletContextInitializer というリスナである。このクラスはそのアプリケーションのサーブレット・コンテキストの変化の通知を受けるもので、ここでは初期化中である通知(contextInitialized)をオーバライドしている。

```
import java.util.HashSet;
import java.util.Set;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.SessionTrackingMode;
import javax.servlet.annotation.WebListener;

/**
 * ServletContextInitializeは、セッション、及びサーブレット・コンテキストに関する
 * プログラムによる設定法を学習するためのクラス
 * @author Cresc Corps.
 * @December 2010, CRESC Corps.
 */

@WebListener
public class ServletContextInitializer implements ServletContextListener{

   @Override
   public void contextDestroyed(ServletContextEvent arg0) {
   }

   @Override
   public void contextInitialized(ServletContextEvent servletContextEvent) {
      ServletContext cxt = servletContextEvent.getServletContext();
      Set<SessionTrackingMode> stms = cxt.getDefaultSessionTrackingModes();
      System.out.print("DefaultSessionTrackingMode : ");
      for (SessionTrackingMode c : stms)
         System.out.print(c + ", ");
      System.out.println();
      stms = new HashSet<SessionTrackingMode>();
      stms.add(SessionTrackingMode.URL);
//    stms.add(SessionTrackingMode.COOKIE);
      cxt.setSessionTrackingModes(stms);
      System.out.print("EffectiveSessionTrackingMode : ");
      for (SessionTrackingMode c : stms)
         System.out.print(c + ", ");
      System.out.println();
   }

}
```

Eclipse 上でこのクラスを下図のように tutorial/Java Resouces: SRC/Default Package に置く。

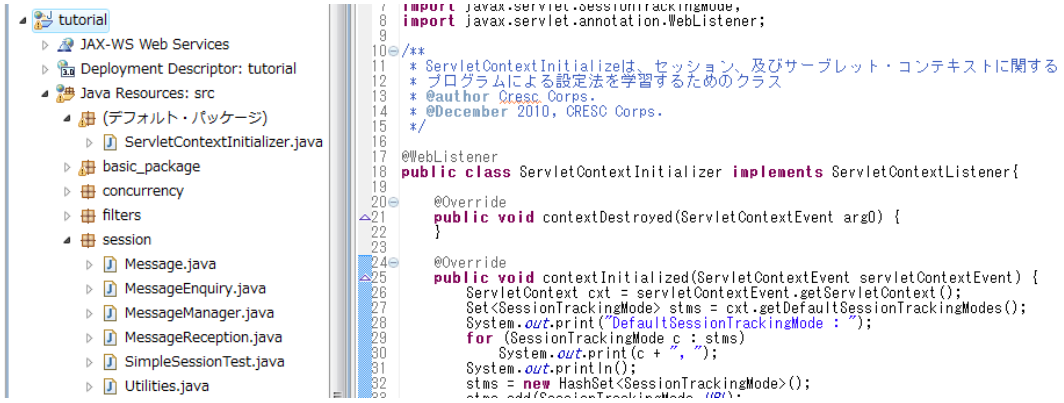


図 9-16：ServletContextInitializer.java の配置

この状態で Tomcat 7 をたち上げると以下のようなメッセージがコンソールに表示されるはずである:

```
情報: Starting Servlet Engine: Apache Tomcat/7.0.2
DefaultSessionTrackingMode : COOKIE, URL,
EffectiveSessionTrackingMode : URL,
2010/12/17 15:52:56 org.apache.coyote.http11.Http11Protocol start
情報: Coyote HTTP/1.1 を http-8080 で起動します
2010/12/17 15:52:56 org.apache.coyote.ajp.AjpProtocol start
情報: Starting Coyote AJP/1.3 on ajp-8009
2010/12/17 15:52:56 org.apache.catalina.startup.Catalina start
情報: Server startup in 744 ms
```

これで context.xml で試したと同じ結果が得られることを各自確認されたい。

このクラスは、セッション追跡モードだけでなく、各種のコンテキストの設定が可能であり、有用なものである。

9.5節　セッション管理のスレッド対応

サーブレットには同時に複数の要求スレッドが通過することになる。サーブレット・エンジンのセッション管理のメカニズムはこれに十分耐えるものである。各スレッドが各々別の HttpSession オブジェクトにアクセスし、またそのオブジェクトにバインドされるオブジェクトもスレッド間で共有されない場合は、スレッドに関して気を使う必要はない。

ところが同じ HttpSession オブジェクトに複数の要求スレッドがアクセスする可能性はないことは無い。**要求処理に時間がかかる場合は、ユーザは続けて Submit ボタンを押す可能性がある。また同じユーザが同じ PC 上で複数のタブあるいはブラウザ・インスタンスから同じサーブレットをアクセスすることもある。**そのような場合にはブラウザによっては同じ HttpSession オブジェクトを複数のスレッドが同時にアクセスする危険性が存在する。あるいは同じことが起きるもうひとつの可能性は、そのユーザがそのページで関わり合っている最中に、フレームあるいは Ajax を使ってそのサーバからデータを取り出すようなウェブ・アプリケーションである。

ServletRequest、HttpSession、及び ServletContext は、ともに「情報の共有の節」で説明したようにスコープ・オブジェクト、あるいはスコープ・コンテナと呼ばれ、これらのオブジェクトの有効な範囲に置いて、オブジェクト間で情報を共有することが出来る。有効な範囲とは ServletRequest ではその要求オブジェクトの処理が終わるまでであり、HttpSession ではそのセッションが無くなるまで(プログラム的な invalidate による無効化、タイムアウト、及びクライアント側でのブラウザを閉じるなどでのクッキーの消去後の再訪問まで)であり、ServletContext はそのアプリケーションが無効化されるまでである。**複数のオブジェクトで共有(あるいは参照)され得るオブジェクトは、常にスレッド安全性が配慮されねばならない。**

HttpSession においても、複数の要求スレッドが同時に同じ HttpSession を獲得し、操作する可能性があることは前述のとおりである。ただし、「複数スレッドの同時通過の確認」の節で示すように、**Google の Chrome と Mozilla の Firefox では同じアドレスを同時にアクセスするとそれらの要求はブラウザ側で直列化される**ので、その問題にある程度配慮がされている。

この問題に関しては、Oracle の上級スタッフ技術者の Brian Goetz 氏が詳しく説明しているので、そちらの邦訳技術資料を見て頂きたい。またサーブレットのスレッド安全の問題は次の章で取り上げることとする。

またサーブレット 3.0 仕様書の 7.7.2 項の「分散環境(Distributed Environments)」では、セッションに対するスレッ

ド問題が詳細に書かれている。

9.6節　　セッションのイベント処理

「リスナ・クラスの定義」の項、及び「イベントとリスナのためのメソッドたち」の項で説明したように、属性の追加等の HttpSession オブジェクトのイベントを、他のリスナ・インターフェイスを実装したオブジェクトにそのリスナを介して通知することが出来る。

そのようなイベント通知を受けるオブジェクトは、一般にはこれらのセッションを統括、管理、あるいは監視するような役割を持つ。例えば、オンライン・ショッピングのショッピング・カートは HttpSession の属性としてバインドされ、受注を統括する在庫確認や受注集計などのオブジェクトは、そのショッピング・カートの変化を受けてイベント・ドリブンで処理を行う。イベント通知のその他の用途としてはロギングなどが考えられる。

ここではホテルのメッセージ・サービスの簡単なアプリケーションで、その使い方を学習する。その昔アメリカに出張して一寸したホテルに泊まると、メッセージングのサービスがあった。当時は携帯電話機やメールなども無く、同じホテルに宿泊する予定でまだチェックインしていない人に、メッセージをフロントやコンシェルジェに言付けることができた。またそのホテルに電話した相手が不在のときは交換手にメッセージを言付けた。そのような場合は、その顧客は自分の部屋にある電話機の赤ランプが点滅していることで、メッセージがあることを知ることが出来る。

ここでは、メッセージの受け付け担当（フロントやコンシェルジェや交換手）のサーブレットである MessageReception と、当該顧客に自分宛のメッセージを伝える担当（フロントや交換手）のサーブレットである MessageEnquiry を考えてみよう。

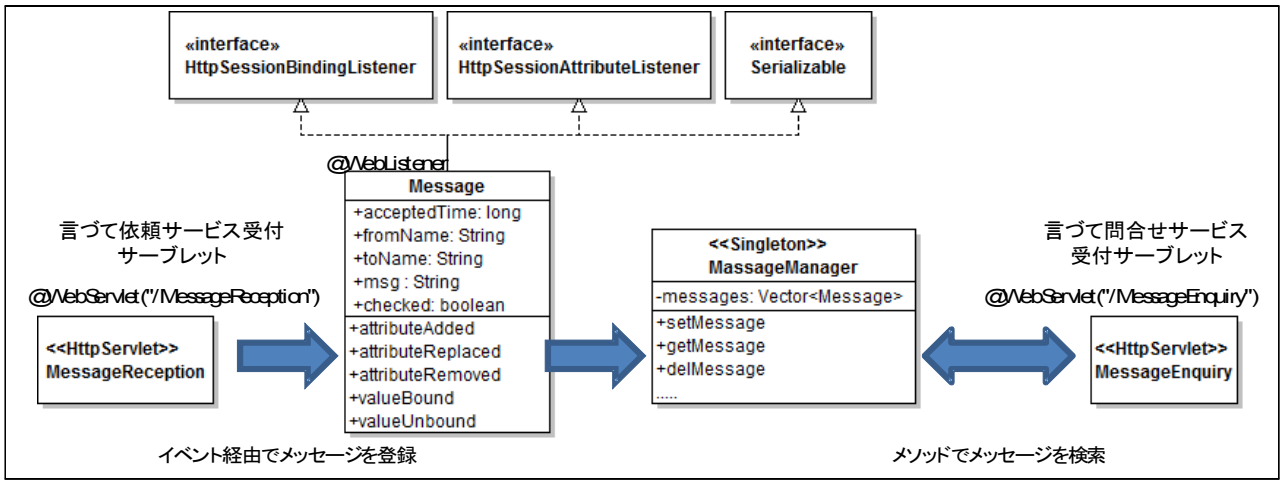


図 9-17：メッセージ・サービスのクラス構成

このサービスは上図のように 4 つのクラスで構成されており、両サイドは依頼と問合せの受付サーブレットである。これらは@WebSerblet アノテーションでサーブレット・コンテナはこれらが指定されたパス・マッピングをもった HttpServlet であることを知る。

Message というクラスは、顧客がそのホテルに依頼したメッセージを表現したものであり、受付時間、発信者名、言づて先の顧客名、メッセージそのもの、及び顧客がそのメッセージを受けたことを示すチェックで構成されている。実際のこの種のアプリケーションでは、顧客の名前に加えてルーム番号、あるいはパスワードなどの識別が必要になろう。またこの場合は受け付け時刻として System.currentTimeMillis()で得られるミリ秒単位の long の変数が使われていて、且つそれを Message オブジェクトの識別にも使われている。従って、1 ミリ秒という極めて短

時間ではあるが、同じ時刻を持った Message オブジェクトが生成されないような配慮が必要になっている。このクラスは HttpSessionBindingListner と HttpSessionAttributeListner の双方のインターフェイスを実装しているが、これはこれら 2 つのリスナを実験し易いようにしたことによる。実際のアプリケーションではどちらかのインターフェイスを実装するだけで良いだろう。またこのクラスはセッションにバインドされる為、分散環境でも使えるよう Serializable インターフェイスも実装している。このクラスは@WebListner というアノテーションで、サーブレット・コンテナに対しリスナ・クラスであることを知らせている。このアノテーションは@WebSerblet と同様、サーブレット 3.0 版から導入されたもので、面倒な配備記述子(web.xml)記述の必要性が無くなり、アプリケーション開発がより容易なものとなっている。このクラスのメソッドたちは、これらのリスナに対応したイベント受け付けのメソッドのみで構成されている。

MessageManager はそのホテルが預かったメッセージの台帳を表現している。台帳であるのでただひとつのインスタンスしか生成されない。そのような構造のクラスは「シングルトン(Singleton)」と呼ばれる。預かっているメッセージは messages というプライベートなフィールドである。messages は Message のオブジェクトの Vector である。本来はこの部分はデータベースであるべきだが、ここでは簡単化する為に Vector が使われている。messages へのアクセスはこの MessageManager クラスが用意しているメソッドたちを介して行われる。

MessageReception はセッションにたいするイベントのみで顧客が入力したメッセージを messages に登録する。MessageReception はそのメッセージをどのオブジェクトが受け取るかを一切知ることなく、そのメッセージをセッションにバインドするだけでその処理を依頼できる。またその過程でイベント・ハンドラにログをとる、あるいはそのメッセージを翻訳等加工するなどの処理を簡単に付加することが出来る。

一方 MessageEnquiry サーブレットのほうは MessageManager と直接やりとりすることで、顧客からの問い合わせに対応している。

それでは、各クラスに関して、ソース・コードをもとにより詳細に説明することにする。

9.6.1 メッセージ受付サーブレット

MessageReception サーブレットは、下図のような 3 つの画面で構成されている。

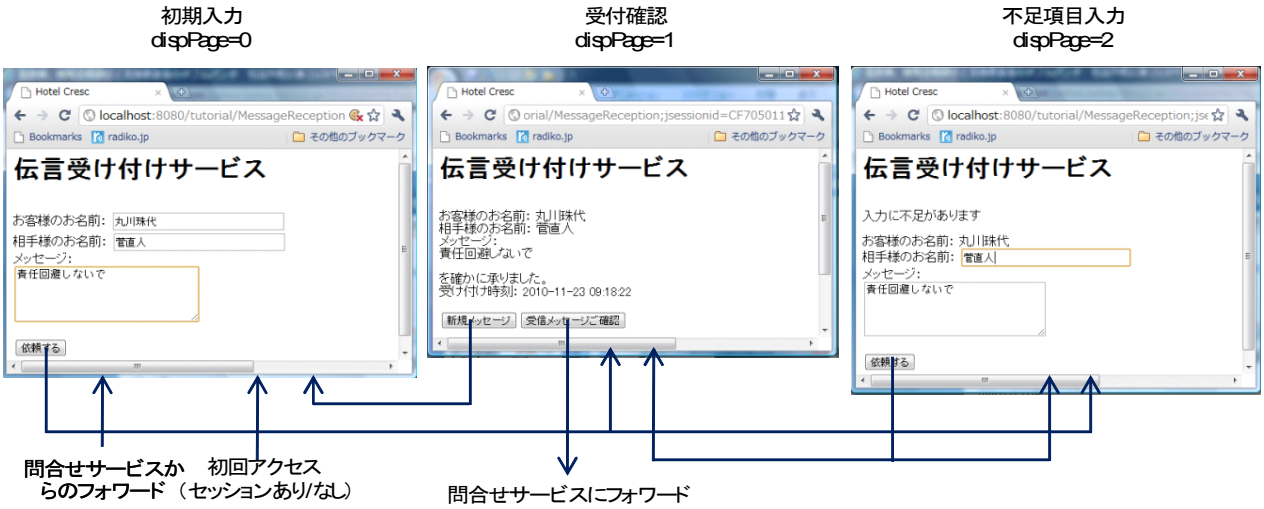


図 9-18：メッセージ受け付けの画面

最初にアクセスすると初期画面が表示される。名前に関しては既にセッションが存在する場合は既知であるので、入力する必要はない。名前（必要な場合）、伝言相手の名前、及びメッセージを記入したら「依頼する」のボタンをクリックする。入力に漏れがあったときは不足項目入力の画面に推移する。正しく入力された場合は受付確認

の画面が表示される。この画面では、更にメッセージを依頼する為の「新規メッセージ」のボタンと、自分宛のメッセージが無いかを問い合わせるサービスを呼ぶための「受信メッセージ確認」のボタンが存在する。

どの画面を表示するかの判断は、

- どのサブミット・ボタンが押されたか
- セッションが存在するか
- 要求パラメタに Message の要素が存在するか
- フォワードされてきた場合はセッションに名前が属性として存在するか

に対する判断テーブル（即ち入力条件対表示ページのマトリクス）を作成し、コーディングを行う。より複雑なアプリケーションでは、画面推移の為の条件テーブルをそのまま組み込むこともある。

画面推移の判断にこれらの条件では不十分なときは、例えば HTML Form の Hidden 型（即ち<input type = “hidden” name = “xxx” value = “yyyy”>）を使って必要な情報を渡すこともある。

まずそのコードを示す：

```
package session;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Calendar;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;


@WebServlet("/MessageReception")
public class MessageReception extends HttpServlet{

   /**
    * MessageReceptionは、セッションのイベント学習用の基礎的なサーブレット
    * @author Cresc Corps.
    * @December 2010, CRESC Corps.
    */

   private static final long serialVersionUID = 1L;

   /**
    * 処理本体    *
    */

   public void performTask(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {

      // 入力されたデータの取り込みと文字変換
      HttpSession session = req.getSession();
      String command = req.getParameter("command");      // Submitボタンの値
      if (command == null) command = "";
      else command = new String (command.getBytes("8859_1"), "Windows-31J");
      String fromName = req.getParameter("fromName");    // 発信者名
      String toName = req.getParameter("toName");        // 着信者名
      String msg = req.getParameter("msg");           // メッセージ
      if (fromName != null){
         fromName = new String (fromName.getBytes("8859_1"), "Windows-31J");
         session.setAttribute("fromName", fromName);
      }
      else {
         fromName = (String)session.getAttribute("fromName");
         if (fromName == null) fromName = "";
      }
      if (toName == null) toName = "";
      else toName = new String (toName.getBytes("8859_1"), "Windows-31J");
      if (msg == null) msg = "";
```

```
      else msg = new String (msg.getBytes("8859_1"), "Windows-31J");

      // フォワードされてきた場合
      if (req.getAttribute("command") != null) command = (String)req.getAttribute("command");

      // アドレスバー入力等でセッションが存在しなければ顧客に名前を要求
      boolean nameRequired = true;
      if ((command == null) || (fromName.equals("")) nameRequired = true;   // 初回訪問や名前未記入
      if (!fromName.equals("")) nameRequired =false;
      else {
         String name = (String)session.getAttribute("fromName");
         if (name != null) {  // 初回でもセッションがあれば属性をチェック
            fromName = name; nameRequired = false;
         }
      }

      // 表示ページモードの判断
      int dispPage = 0;
      if (command.equals("メッセージお言付けへ") || command.equals("") || command.equals("新規メッセージ
")) dispPage = 0;
      // 入力に不足が無いかで振り分け
      else if((toName.equals("") || msg.equals("") || fromName.equals(""))
            && (command.equals("依頼する"))) dispPage = 2;
      if(((!toName.equals("")) && (!msg.equals("")) && (!fromName.equals("")))
            && (command.equals("依頼する") )) dispPage = 1;
      if (command.equals("受信メッセージご確認")){              // 問合せサービスへ転送
         req.setAttribute("command", "受信メッセージご確認");  // コマンドを要求に貼り付け
         String url = "/MessageEnquiry";
         getServletContext().getRequestDispatcher(url).forward(req, res);
      }

      // 入力がOKならMessageを組み立ててセッションに属性として追加
      long acceptedTime = 0;
      if (dispPage == 1){
         synchronized(this) {
            Message message = MessageManager.getSingletonObject().getMessage();
            acceptedTime = System.currentTimeMillis();
            message.acceptedTime = acceptedTime;
            message.fromName = fromName;
            message.toName = toName;
            message.msg = msg;
            try { // 2つのメッセージが同じ送信時刻を持つのを避ける
               Thread.sleep(2);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            session.setAttribute("message", message);
         }
      }

      //クッキーが使えないときの為にURL書き換えを行う
      String urlBase = req.getContextPath() + req.getServletPath();
      String url = res.encodeURL(urlBase);

      // 応答メッセージの作成開始
      res.setContentType("text/html; charset=windows-31j");//応答ヘッダContent-Type追加
      res.setHeader("Cache-Control", "no-cache");           //キャッシュを殺しておく
      PrintWriter out = res.getWriter();                 //出力バッファ取得
      out.println("<html><head><title>Hotel Cresc</title>");
      out.println(         // 同じことをHTMLテキストでも指定
            "<meta http-equiv=\"cache-control\" content=\"no-cache\">" +
      "<meta http-equiv=\"content-type\" content=\"text/html; charset=Windows-31J\"></head>");
      // タイトル表示
      out.println("<body style=\"width:20cm\"><h1>伝言受け付けサービス</h1>");
      if (dispPage == 2) out.println("<br>入力に不足があります");
      // 名前の入力または表示
      out.println("<form method=\"post\" action=\"" + url + "\">");
      out.println("<br>お客様のお名前： ");
      if (nameRequired){
         out.print("<input type=\"text\" name=\"fromName\" size=\"30\"");
```

```
         if (!fromName.equals("")) out.print("value=\"" + fromName + "\"");
         out.println(">");
      }else out.println(fromName);
      // 相手先の入力または表示
      out.println("<br>相手様のお名前： ");
      if ((dispPage == 2) || (dispPage == 0)){
         out.print("<input type=\"text\" name=\"toName\" size=\"30\"");
         if (!toName.equals("")) out.print("value=\"" + toName + "\"");
         out.println(">");
      }else out.println(toName);
      // メッセージの入力または表示
      out.println("<br>メッセージ：<br>");
      if ((dispPage == 2) || (dispPage == 0)){
         out.println("<textarea name=\"msg\" cols=\"30\" rows=\"5\">");
         if (!msg.equals("")) out.print(msg);
         out.println("</textarea>");
      }else {
         out.print(msg + "<br><br>を確かに承りました。<br>受け付け時刻： ");
         Calendar c = Calendar.getInstance(); c.setTimeInMillis(acceptedTime);
         out.println(String.format("%tF %tT", c, c));
      }
      // 送信ボタンを設定
      if ((dispPage == 0) || (dispPage == 2)){
         out.print("<br><br><input type=\"submit\" name = \"command\" value=\"依頼する\">" );
      }
      if (dispPage == 1){
         out.print("<br><br><input type=\"submit\" name = \"command\" value=\"新規メッセージ\">" );
         out.print("<input type=\"submit\" name = \"command\" value=\"受信メッセージご確認\">" );
      }

      // デバッグ用：蓄積されたメッセージを表示
      out.println("<pre>" + MessageManager.getSingletonObject().dumpMessages().toString() +
"</pre>");

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


   /**
    * 到来HTTP GET要求の処理
    */
   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res);
   }

   /**
    * 到来HTTP POST要求の処理
    */
   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res);
   }

   /**
    * 本サーブレット情報の文字列を返す
    */
   @Override
   public String getServletInfo() {
      return "Message Reception Servlet, Version 1.0 by Cresc";
   }
}
```

このコードでの幾つかのポイントを挙げる:

1. 要求の転送(フォワード)
   あるサーブレットから別のサーブレットに処理を移管する場合は、必要な情報を共有オブジェクトである要求、セッション、あるいはコンテキストに添付して渡すことになる。ここではセッションには既に Message

のオブジェクトが付加されている。残りは「受信メッセージご確認」ボタンが押されたという情報を "command"という名前でセットし、セッションに発信者名を属性としてセットし、またクッキーが使えない場合の為に転送先にセッション ID が付加されたアドレスを要求ディスパッチャに渡している。セッションには Message がセットされており、そこから発信者名を取得できるが、受信メッセージ確認のサービスでは Message がセットされていない。従って共通してセッションに発信者名をセットしている。

```
req.setAttribute("command", "受信メッセージご確認");  // コマンドを要求に貼り付け
String url ="/MessageEnquiry";
getServletContext().getRequestDispatcher(res.encodeURL(url)).forward(req, res);
```

2. Message のオブジェクトのセッションへのバインド

顧客からの伝言が正しく入力されているときは、それを Message のオブジェクトとしてまとめ、セッションにセットする。これによりイベントがそのオブジェクトに渡され、そのイベント処理によりそのオブジェクトがそのホテルの台帳に MessageManager 経由で登録される。Message のインスタンスである message は MessageManager の getMessage メソッドで取得されているが、直接 Massage のコンストラクタを呼んでも構わない。MessageManager.getSingletonObject()は MessageManager のオブジェクトを返すメソッドであり、そのインスタンスが存在しな場合はただひとつのインスタンスを生成する。注意点として、message の組み立てとセッションへの属性としてのセットが同期ブロック化されていることである。同期ブロックは次の章で解説するが、1)前節でのべたように、ブラウザの使い方によっては同じセッションを持った複数のスレッドが同時にサーブレットを通過する可能性があること、2)またタイムスタンプである acceptedTime をメッセージの識別用に使っているので 1mS という短時間ではあるが同じタイムスタンプを持ったメッセージが作られる可能性を排除する為にスレッド間に少なくとも 2mS の間隔を設けるため、にこのブロック間ではスレッドを直列化している。同じセッションを持ったスレッドが同時に到来する場合の危険性はショッピング・カートのように HttpSession に保管されたオブジェクトを読み出してそれを加工してまた保管する場合に発生する。今回のアプリケーションでは、セッションにバインドされたオブジェクトを読み出す操作をしていないので、その問題は無い。

```
synchronized(this) {
   Message message = MessageManager.getSingletonObject().getMessage();
   acceptedTime = System.currentTimeMillis();
   message.acceptedTime = acceptedTime;
   message.fromName = fromName;
   message.toName = toName;
   message.msg = msg;
   try { // 2 つのメッセージが同じ送信時刻を持つのを避ける
      Thread.sleep(2);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
   session.setAttribute("message", message);
}
```

3. デバッグ用に蓄積されているメッセージを出力する

デバッグに便利なように、MessageManager には dumpMessages()というメソッドが用意されている。これは台帳に登録されている総てのメッセージを StringBuffer にセットするもので、以下のような 1 行を追加するだけで台帳の内容の変化をチェックできる。実際の運用時はこの行をコメント・アウトすればよい。

```
    // デバッグ用：蓄積されたメッセージを表示
    out.println("<pre>" + MessageManager.getSingletonObject().dumpMessages().toString() +
"</pre>");
```

4. isNew()に関して

前にも指摘したが、**isNew()はその要求に対するセッションが存在しないという情報であり、初期画面を表示する為の条件のひとつでしかない**。別の画面上のサブミット・ボタン操作で初期画面に戻る場合もある。ユーザによっては（特にクッキーを受け付けなくしたブラウザで）別の画面上でアドレスバーに初期画面のアドレスをセットしてこのサーブレットをアクセスする場合もあろう。表示画面決定要素としての isNew()の役割は大きくはない。また**実際の商用アプリケーションでは、このメソッドでユーザが始めて訪問したと判断してはいけない**。別のブラウザでそのユーザが既にそのアプリケーションにログインしてい

ることもあり得る。

9.6.2 Message クラス

Message はこの節のはじめに述べたように、3 つのインターフェイスを実装した簡単なクラスである:

```
package session;

import java.io.Serializable;

import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionBindingListener;

import session.MessageManager;

@WebListener   // このアノテーションはサーブレット 3.0 仕様書の 8.1.4 節を参照のこと
public class Message implements Serializable, HttpSessionAttributeListener,
HttpSessionBindingListener{
   private static final long serialVersionUID = 1L;

   /**
    * Message は、セッションのイベント学習用の基礎的なリスナ
    * @author Cresc Corps.
    * @December 2010, CRESC Corps.
    */

   // メッセージの要素
   public long acceptedTime = 0;
   public String fromName = null;
   public String toName = null;
   public String msg = null;
   public boolean checked = false;

   // セッションに Message が付加されたときにこれを台帳に追加
   @Override
   public void attributeAdded(HttpSessionBindingEvent e){
      // 属性として同じ名前でオブジェクトがセットされたときはイベントとされない
   }
   @Override
   public void attributeReplaced(HttpSessionBindingEvent e){
      // 属性として同じ名前でセットされたときに、置き換えられたオブジェクトを返す
   }
   @Override
   public void attributeRemoved(HttpSessionBindingEvent e) {
      // ある名前でセットされた属性が削除されたとき
   }
   @Override
   public void valueBound(HttpSessionBindingEvent e) {
      // 属性として自分がセットされたとき
      MessageManager.getSingletonObject().setMessage((Message)e.getValue());
   }
   @Override
   public void valueUnbound(HttpSessionBindingEvent e) {
      // 自分が属性から削除されたとき
   }
}
```

1. @WebListener アノテーション
   そのウェブ・アプリケーションの配備記述子の web-app 要素上に“metadata-complete”属性が"true”にセットされていない限り、またこのクラスが WEB-INF/classes ディレクトリに置かれているとき、このアノテーションが有効となる。@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

2. 実装インターフェイス
   ここでは実験がしやすいように終わりの 2 つのインターフェイスが実装されているが、実際のアプリケーションではどれかひとつを実装すれば良い。このクラスはそのオブジェクトがセッションにバインドされる構成になっていて HttpSessionAttributeListener が利用できる。またその為このクラスは分散環境用にも使えるように Serializable インターフェイスを実装している必要がある。

3. どのイベントを使うか
   このクラスでは 2 つのインターフェイスが用意している 5 つのメソッドが利用できるようになっている。注意しなければならないのは **attributeAdded メソッドで、これは新たな名前で属性がセットされたときに呼び出される**。今回のように同じ名前で属性のセットが繰り返される場合には、valueBound を使うか、あるいは一旦その属性を削除する必要がある。もうひとつ注意しなければならないのは attributeReplaced メソッドで、これは同じ名前の置き換えられるオブジェクトを返すことで、置き換えるオブジェクトと混同してはならない。HttpSessionAttributeListener のイベントには顧客がサービスを切り替えた際に発信者名がセットされた場合も含まれるので、これの識別が必要になる。また HttpSessionAttributeListener を使う場合は、例えば次のようにイベント名を確認する必要がある。

```
public void attributeReplaced(HttpSessionBindingEvent e){
   // 属性として同じ名前でセットされたときに、置き換えられたオブジェクトを返す
   if (e.getName() == "message"){
      ;  // AttributeListner では属性名の確認が必要
   }
```

9.6.3 メッセージ問合せサーブレット

このサーブレットは 2 つの画面で構成されている。初期入力画面はユーザの名前(fromName)が判らないときの名前を入力するための画面である。受付確認の画面はそのユーザあてに預かっているメッセージを出力する。チェック・ボックスはその顧客が了解したことを示す。「チェック・メッセージ削除」ボタンをクリックすれば、そのメッセージは台帳から削除される。「メッセージお言付けへ」というボタンは受付サービスへのフォワード用である。

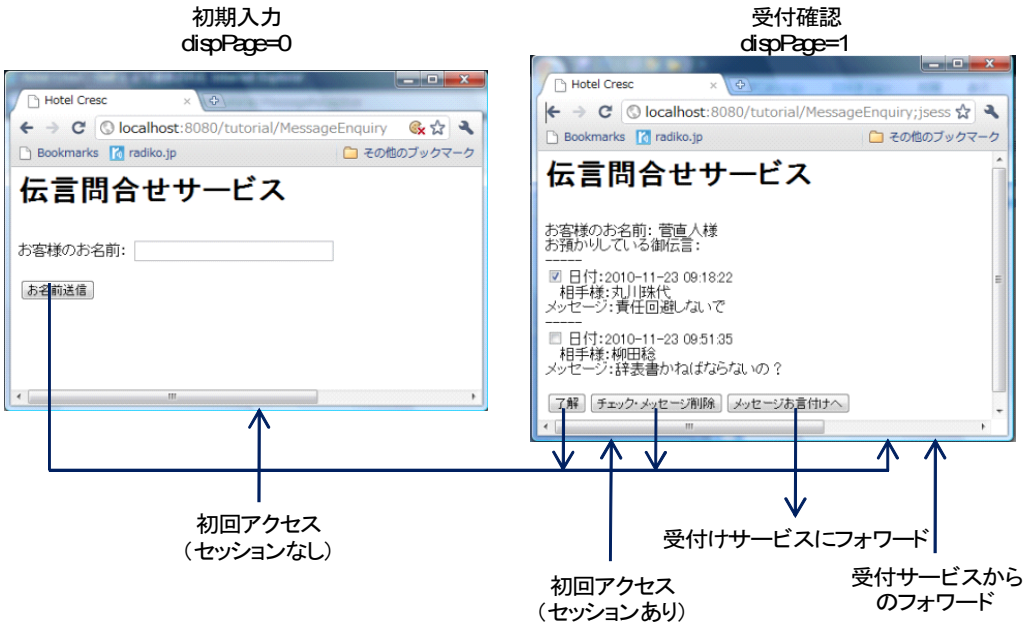


図 9-19:伝言問合せサービスの画面

このサーブレットのコードは以下のようである:

```
package session;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Calendar;
import java.util.Iterator;
import java.util.Vector;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@WebServlet("/MessageEnquiry")
public class MessageEnquiry extends HttpServlet{

   /**
    * MessageEnquiryは、セッションのイベント学習用のMessageReceptionの補完的サーブレット
    * @author Cresc Corps.
    * @December 2010, CRESC Corps.
    */

   private static final long serialVersionUID = 1L;

   /**
    * 処理本体    *
    */

   public void performTask(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {

      HttpSession session = req.getSession();
      ServletContext ctxt = getServletContext();
      MessageManager mmngr = MessageManager.getSingletonObject();
      String command = req.getParameter("command");      // Submitボタンの値
      if (command == null) command = "";
      else command = new String (command.getBytes("8859_1"), "Windows-31J");
      if (req.getAttribute("command") != null) command = (String)req.getAttribute("command");
      String fromName = req.getParameter("fromName");    // 発信者名
      if (fromName != null){
```

```
        fromName = new String (fromName.getBytes("8859_1"), "Windows-31J");
        session.setAttribute("fromName", fromName);
     }
     else {
        fromName = (String)session.getAttribute("fromName");
        if (fromName == null) fromName = "";
     }

     // 表示ページの選択
     int dispPage = 1;
     String name = (String) session.getAttribute("fromName");
     if (name != null) {
        fromName = name; dispPage = 1;
     }
     if (session.isNew()) dispPage = 0;
     if (fromName.equals("")) dispPage = 0;
     if (command.equals("お名前送信") && fromName.equals("")) dispPage = 0;
     if (command.equals("受信メッセージご確認")) dispPage = 1;

     if (dispPage == 1){

        // 「言付サービス」指定のときは受け付けサービスにディスパッチ
        if (command.equals("メッセージお言付けへ")){
           req.setAttribute("command","メッセージお言付けへ");   // コマンドを要求に貼り付け
           String url = "/MessageReception";
           ctxt.getRequestDispatcher(url).forward(req, res);
        }

        // チェック項目の変更処理
        if (command.equals("了解") || command.equals("チェック・メッセージ削除")){
           String[] cks = req.getParameterValues("checked");
           if (cks == null) cks = new String[0];
           long[] checks = new long[cks.length];
           for (int i = 0; i< cks.length; i++){
              checks[i] = new Long(cks[i]);
           }
           mmngr.setChecks(fromName, checks);
        }

        // チェック・メッセージの削除
        if (command.equals("チェック・メッセージ削除")){
           mmngr.delCheckedMessages(fromName);    // チェックされたメッセージを消去
        }
     }

     // ここから画面作成

     //クッキーが使えないときの為にURL書き換えを行う
     String urlBase = req.getContextPath() + req.getServletPath();
     String url= res.encodeURL(urlBase);

     // 応答メッセージの作成開始
     res.setContentType("text/html; charset=windows-31j");//応答ヘッダContent-Type追加
     res.setHeader("Cache-Control", "no-cache");           //キャッシュを殺しておく
     PrintWriter out = res.getWriter();                 //出力バッファ取得
     out.println("<html><head><title>Hotel Cresc</title>");
     out.println(         // 同じことをHTMLテキストでも指定
           "<meta http-equiv=\"cache-control\" content=\"no-cache\">" +
     "<meta http-equiv=\"content-type\" content=\"text/html; charset=Windows-31J\"></head>");

     // タイトル表示
     out.println("<body style=\"width:20cm\"><h1>伝言問合せサービス</h1>");

     // 初期画面
     out.println("<form method=\"post\" action=\"" + url + "\">");
     out.println("<br>お客様のお名前： ");
     if (dispPage == 0){
        out.print("<input type=\"text\" name=\"fromName\" size=\"30\">");
        out.print("<br><br><input type=\"submit\" name=\"command\" value=\"お名前送信\"" );
        out.println("</form></body></html>");
```

```
      }

      // サービス画面
      if (dispPage == 1){
         out.println(fromName + "様<br>お預かりしている御伝言：");
         Vector<Message> msgs = mmngr.getToMessages(fromName);
         if (msgs.isEmpty()) out.println("ございません。<br>");
         else {
            Iterator<Message> im = msgs.iterator();
            while (im.hasNext()) {
               out.println("<br> -----");
               Message msg = im.next();
               out.print("<br> <input type = \"checkbox\" name = \"checked\" value = \"");
               out.print(msg.acceptedTime + "\" ");
               if (msg.checked) out.print("checked");
               out.println(">");
               Calendar c = Calendar.getInstance(); c.setTimeInMillis(msg.acceptedTime);
               out.println("  日付：" + String.format("%tF %tT", c, c));
               out.println("<br>  相手様：" + msg.fromName);
               out.println("<br>メッセージ：" + msg.msg);
            }
         }
         out.println("<input type=\"hidden\" name=\"fromName\" value=\"" + fromName +"\">");
         out.print("<br><br><input type=\"submit\" name=\"command\" value=\"了解\">");
         out.print("<input type=\"submit\" name=\"command\" value=\"チェック・メッセージ削除\">");
         out.print("<input type=\"submit\" name=\"command\" value=\"メッセージお言付けへ\">");
         out.println("</form></body></html>");
      }
   }

   /**
    * 到来HTTP GET要求の処理
    */
   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res);
   }

   /**
    * 到来HTTP POST要求の処理
    */
   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res);
   }

   /**
    * 本サーブレット情報の文字列を返す
    */
   @Override
   public String getServletInfo() {
      return "Message Enquiry Servlet, Version 1.0 by Cresc";
   }
}
```

このサーブレットに関しては特に注意すべきことは無い。このサーブレットは次の項で示すMessageManagerと交信してサービスを実行する。最初の

```
      ServletContext ctxt = getServletContext();
      MessageManager mmngr = MessageManager.getSingletonObject();
```

で、コンテキストctxtとMessageManagerのインスタンスmmngrを取得している。

9.6.4 MessageManagerクラス

この節の最初に説明したように、このクラスはメッセージの台帳であるVector<Message>のmessagesを保持したシングルトンである。シングルトンの設計パタンは自分のコンストラクタをprivateとし、自分自身でしかインスタン

ス化できなくし、外部からは getSingletonObject メソッドでそのオブジェクトを渡すようにしている。このメソッドでは既に自分自身のインスタンスが存在しなければコンストラクタを呼ぶことで、ただひとつのインスタンスしか出来ないようにしている。

問題はこのようなクラスは static な要素を持っている為、「自動再ロード」の項や「自動再ロード機能の確認」の項でも指摘したように、自動再ロードが出来るように設定されていると、テストでのコードの変更などでコンテキストの再ロードが行われると、これらの要素がリセットされることである。この件は別途説明することにする。

messages のような要素へのアクセスは、このクラスで用意されているセッタ・ゲッタその他のメソッドを介することになり、通常それらのメソッドは synchronized キーワードにより同期化される。このような構成をオブジェクト・ロック構成という。そのオブジェクトの同期化されたメソッドにひとつにあるスレッドがアクセスする場合は、他のスレッドは同期化されたメソッド総てに対してロックされる。

このクラスでのスレッドに関するもうひとつの注意点は、Iterator を使った Vector (Collection 総てに言える) 要素へのアクセスで、その要素に変更を加えるときには Iterator が用意している remove などのメソッド以外を使うとスレッド安全でないというエラー (ConcurrentModificationException) がスローされることである。

以下はこのクラスのコードである:

```
package session;

import java.util.Calendar;
import java.util.Iterator;
import java.util.Vector;

public class MessageManager{

   /**
    * MessageManager は、セッションのイベント学習用の基礎的な
    * メッセージ保管のシングルトン・クラス
    * @author Cresc Corps.
    * @December 2010, CRESC Corps.
    */

   // 預かったメッセージの台帳
   private static Vector<Message> messages = new Vector<Message>();

   private static MessageManager singletonObject;
   // プライベートなコンストラクタにして他のクラスがインスタンス化するのを防止
   private MessageManager() {
   }
   // スタティックなアクセスのメソッドで、このオブジェクトをかえす
   // 同期化することで複数のスレッドのアクセスを防止
   public static synchronized MessageManager getSingletonObject() {
      if (singletonObject == null) {
         singletonObject = new MessageManager();
      }
      return singletonObject;
   }

   // このコードはクローンを使って複数のインスタンスが作られるのを防止する
   @Override
   public Object clone() throws CloneNotSupportedException {
      throw new CloneNotSupportedException();
   }

   // Message の初期オブジェクトを返す
   public Message getMessage(){
      return new Message();
   }

   // あるメッセージを追加記録する
   public synchronized void setMessage(Message msg){
      messages.add(msg);
   }
```

```
// 与えられたタイムスタンプのメッセージを消す
public synchronized void delMessage(long ts){
   Iterator<Message> im = messages.iterator();
   while (im.hasNext()) {
      Message msg = im.next();
      if (msg.acceptedTime == ts) im.remove();
   }
}

// 与えられたタイムスタンプのメッセージをチェックする
public synchronized void checkMessage(long ts){
   for (int i = 0; i<messages.size(); i++){
      Message msg = messages.get(i);
      if (msg.acceptedTime == ts){
         msg.checked = true;
         messages.set(i, msg);
      }
   }
}

// 与えられたタイムスタンプのメッセージのチェックを外す
public synchronized void unCheckMessage(long ts){
   for (int i = 0; i<messages.size(); i++){
      Message msg = messages.get(i);
      if (msg.acceptedTime == ts){
         msg.checked = false;
         messages.set(i, msg);
      }
   }
}

// ある宛先名のチェックされたメッセージたちを削除する
public synchronized void delCheckedMessages(String toName){
   Iterator<Message> im = messages.iterator();
   while (im.hasNext()) {
      Message msg = im.next();
      if (msg.checked && msg.toName.equals(toName)) im.remove();
   }
}

// ある宛先のメッセージで指定されたIDをチェックし、それ以外のチェックを外す
public synchronized void setChecks(String toName, long[] checkedIds){
   if (checkedIds.length == 0){
      for (int i = 0; i<messages.size(); i++){
         Message msg = messages.get(i);
         if (msg.toName.equals(toName)){
            msg.checked = false;
            messages.set(i, msg);
         }
      }
   }
   else{
      for (int i = 0; i<messages.size(); i++){
         Message msg = messages.get(i);
         if (msg.toName.equals(toName)){
            msg.checked = false;
            for (int j = 0; j<checkedIds.length; j++){
               if (msg.acceptedTime == checkedIds[j]){
                  msg.checked = true;
               }
            }
            messages.set(i, msg);
         }
      }
   }
}


// 与えられたtoNameのメッセージたちを返す
public synchronized Vector<Message> getToMessages(String toName){
   Vector<Message> msgs = new Vector<Message>();
   Iterator<Message> im = messages.iterator();
   while (im.hasNext()) {
      Message msg = im.next();
```

```
         if (msg.toName.equals(toName)) msgs.add(msg);
      }
      return msgs;
   }

   // 与えられた fromName のメッセージたちを返す
   public synchronized Vector<Message> getFromMessages(String fromName){
      Vector<Message> msgs = new Vector<Message>();
      Iterator<Message> im = messages.iterator();
      while (im.hasNext()) {
         Message msg = im.next();
         if (msg.fromName.equals(fromName)) msgs.add(msg);
      }
      return msgs;
   }

   // テスト用、蓄積されたものの出力
   public synchronized StringBuffer dumpMessages() {
      StringBuffer sb = new StringBuffer();
      Iterator<Message> im = messages.iterator();
      while (im.hasNext()) {
         Message msg = im.next();
         Calendar c = Calendar.getInstance(); c.setTimeInMillis(msg.acceptedTime);
         sb.append("\nTime stamp : " + String.format("%tF %tT", c, c));
         sb.append("\nfrom : " + msg.fromName);
         sb.append("\nto   : " + msg.toName);
         sb.append("\nmessage : " + msg.msg);
         sb.append("\nchecked : " + msg.checked);
         sb.append("\n -----");
      }
      return sb;
   }

}
```

9.6.5　言伝サービスの実験

このサービスの実験に際しては、同じブラウザでタブごとに別のセッションを持つことが出来るので、ブラウザをクッキー受付不可に設定すると良い。読者は次のような項目を試して見られたい：

1. クッキーを受け付けなくしたブラウザでタブごとに別のユーザとしてアクセスしてその動作を確認する。
2. クッキーを受け付けるように設定したブラウザで別のタブで同じサーブレットをアクセスしたらどうなるか。
3. Message クラスで attributeAdded を使うようにするにはどうしたら良いか。これはバインドされるオブジェクトのクラス以外でこのようなイベントを処理するのに必要になる。
4. Message のクラスの各メソッドで、そのイベントの詳細ログを SLF4J を使って作成させる。
5. コンシェルジュもメッセージ受付ができる。MessageReception を継承した MessageConcierge.java をつくり、共に同じサービスが出来るようにする。

9.7節　　この章のまとめ

この章で説明したことから、セッション管理におけるポイントは次のようになる：

1. クッキーを使えなくしたブラウザにも確実に対応できるようにしておく。これは特に携帯電話機（どの機種でも使えるようにするには）、欧州などのプライバシに厳格な地域、あるいはプライバシに敏感なユーザにたいしては不可欠である。自分の開発したサーブレットがきちんと対応していることの確認は以下の手段を使う：

    - 各ブラウザをクッキーを受け付けないように設定する
    - Tomcat でクッキーを使わないように context.xml ファイル、あるいは setSessionTrackingMode メソッドで設定する
2. ブラウザの使い方によっては、同じセッションを有するスレッドがサーブレットで並行処理される危険性がある。ブラウザによって異なるので、注意が必要である。
3. ブラウザによってはセッション・クッキーの保持に相違があるので、注意を要する。
4. セッションにバインドするオブジェクトは Serializable インターフェイスを実装する。
5. セッションのイベント通知を使う場合は、どのイベントを使うかを良く検討する。

第10章　　スレッド安全

大規模なウェブ・アプリケーションでは大量の要求スレッドがサーブレットに並行アクセスする。その為、サーブレットはスレッド安全(スレッド・セーフともいう)なコードとなっていなければならない。10 年前にこのチュートリアルの初版を書いたときには、スレッド安全に対するプログラマたちの認識は高くは無かったが、現在はレベルが上がっており、スレッドに対するかなりの知識をプログラマたちは持っている。また現在はスレッド安全に関する多くの資料や記事が存在する。したがって、この章ではサーブレット開発の際のチェック・シートに使えそうな幾つかのポイントを記述することにする。

スレッド安全の問題は、「情報の共有」の節で述べたように、スレッドたちが共有するリソース(変数、オブジェクト、ドキュメント…)へのアクセスの制御になる。サーブレットではインスタンス変数やクラス変数、セッションあるいはコンテキストにバインドされたオブジェクト(注:要求オブジェクトはスレッド安全である)、あるいは外部リソースなどへのアクセスに特に注意が必要である。これを整理すると次のようになる:

1. **コンテキストはスレッド安全ではない**。コンテキストはサーブレット単体のものではなく、複数のサーブレットを走る複数のスレッドから可視である。従って後述のオブジェクト・ロックのメカニズムをつかってコンテキスト・オブジェクトに対する排他制御をおこなうことになる。
2. **セッション・オブジェクトもスレッド安全ではない**。各要求が単一のクライアント(ブラウザ)から出されているので、セッションは一見スレッド安全であるかに見える。しかしながら「セッション管理のスレッド対応」の節で述べたように、同じ PC 上の複数のブラウザのインスタンスが同じセッションを共有したらどうなるであろうか(「複数のタブまたは画面から SimpleThreadTest をアクセスした場合のブラウザの動作」の表参照)？あるいはそのユーザがそのページで関わり合っている最中に、Ajax を使ってそのサーバからデータを取り出すようなウェブ・アプリケーションの場合はどうだろうか？従ってこれも後述のオブジェクト・ロックのメカニズムをつかってセッション・オブジェクトに対する排他制御をおこなうことになる。
3. **要求オブジェクトにバインドされたオブジェクトはスレッド安全である。**要求オブジェクトは要求スレッドが連れてきたものでありその要求スレッドのみが使用する。
4. **ローカル変数だけはスレッド安全である。但しクラス変数とインスタンス変数はスレッド安全ではない。**ローカル変数はリエントラントな構造になっていてスレッド毎にそのメモリ領域が割り当てられる。
5. init メソッドはコンテナがそのサーブレットをインスタンス化するときに 1 回だけ単一のスレッドが呼ぶので、このメソッドの中ではスレッド安全の心配はいらない。しかし **destroy メソッドは注意が必要**である。destroy メソッドを呼んでいるときに他のスレッドがそのサーブレットを実行している可能性がある。これに関しては「サーブレットの終了」の節で述べてある。
6. 殆どの API はスレッド安全ではあるが、例外的にスレッド安全でないものが存在するので、注意が必要である。

Wikipedia ではスレッド安全を実現する方法として以下のようなものがあるとしている:

1. リエントラント(Re-entrancy)
   これは各スレッドごとにそのリエントラントな関数のローカル変数をスタックのようなものに蓄積してリエントラント化することでスレッドセーフを実現できるが、広域変数などを使った状態情報のセーブができない。プログラマたちが作成したメソッドやブロック内のローカル変数はそうなっている。Java の標準の API は殆どがそのような構造になっているが、例外的にスレッド安全でないクラスが存在する。その件に関してはあとで説明する。

2. 相互排他(Mutual exclusion)
   これは良くミューテックス(Mutex)とも呼ばれ、Java における synchronized による排他制御でこれが実現

できるし、java 1.5 版からは java.util.concurrent.Mutex ほかのクラスが用意されている。共有データへのアクセスを「逐次化あるいは直列化(serialize)」、即ち待ち行列化することでスレッドセーフを実現する。ただし、複数の共有データにアクセスする際には十分に注意しなければならない。良く遭遇する問題は、競合条件(race conditions)、デッドロック(deadlocks)、ライブロック(livelocks)、枯渇(starvation)などがある。

3. スレッド・ローカル・データ(Thread-local storage)
例えばスレッドの識別子(番号)をキーとして広域変数のコピーをスレッド毎に持たせることでメソッドを超えた範囲で変数を保持できるようにする。各変数にアクセスするメソッド自体はリエントラントではないが、特定のスレッドだけが特定の広域変数にアクセスすることが保証できれば、スレッド安全となる。java では 1.2 版から java.lang.ThreadLocal<T>というクラスが用意されている。

4. アトミック操作(Atomic operations)
共有データを何らかのアトミック(それ以上分割できない、即ちその間は他のスレッドからの同時アクセスを許さない)な操作でアクセスすることで他のスレッドから同時アクセスされないことを保証する。例えば i++;といったひとつのステートメントであっても、これはアトミックな操作ではない。i を読み出し、インクリメントし、また i に書き込む途中で他のスレッドが i を変更できる。従ってこのような場合のアトミック操作には一般に特別なハードウエア命令を必要とするが、ライブラリがそのような機能をサポートしている場合がある。Java には 1.5 版から java.util.concurrent.atomic パッケージが用意されており、機械語レベルのプリミティブを使うことで、オーバヘッドが少ない操作が可能になっている。

本章では、ほぼこの順に幾つかのポイントを記述することにする。

10.1節　Java における排他制御

10.1.1　同期化(Synchronization)

Java では、各オブジェクトはモニタ・ロック(Monitor Lock)という(あるいは固有ロック(Intrinsic Lock)ともいう)排他機能を有している。このロック機能は同期化の為の 2 つの点で重要な役割を持っている:

- アトミック性を維持する為に必要な、あるオブジェクトの状態への排他的アクセスを行う
- 可視性に対し不可欠な、事前発生(Happens-before)関係を確立する

アトミック性とは、それ以上分割できないという意味で使われている用語で、そのルーチンがアトミックだということは、そのルーチンの実行中に割りこまれることが無いことを意味する。これはハードウエアによって実現されることもあればソフトウエアでシミュレートされることもある。一般にアトミックな命令はハードウエアで提供され、それはソフトウエア内でアトミックなルーチンをシミュレートする為に使われる。ここでは、例えばあるスレッドが複数のデータ項目を更新しているときに、データ項目たちの整合が取れない状態で別のスレッドがこのデータを読み出すことを防止することをいう。また「中間状態を確認できないルーチン」と、より幅広い意味でアトミックという言葉を使うこともある。例えば銀行業務の例では、「口座を確認」し「口座の状態を変更」するという操作がアトミックだとすると(その操作全体が分割できない)、最初のスレッドが口座の状態を変更し終了するまでは他のスレッドが同じ口座をアクセスできないことになる。

可視性とは書き込みスレッドがあるオブジェクトのデータ項目を更新しているにも関わらず他の読み出しスレッドからは以前のあるいは一貫性の無い状態のデータが見えないようにすることを言う。この問題の解決には**「事前発生(Happens-before)」の関係**を維持することになる。即ち
int counter = 0;
と初期化された counter がスレッド A とスレッド B によって共有されており、スレッド A がステートメント 1 でこれを

インクリメントし、

- ステートメント 1： counter++;

その直後にスレッド B がそれをステートメント 2 でコンソールに出力しようとする際は、

- ステートメント 2： System.out.println(counter);

この counter に対する**スレッド A でなされた変更(happens)がスレッド B からあらかじめ(before)可視**でないと、"0”と出力してしまう可能性が出る。それを防止するにはこれらの 2 つのステートメント間で事前発生関係を確立させねばならない。

ひとつ以上の synchronized メソッドを持つオブジェクトをモニタと称している。Wikipedia に書かれているように、Java ではモニタとはスレッドを待機させ準次実行させることができるオブジェクトである。synchronized メソッドを実行中のスレッドは、このモニタのロックのオーナである。このことから、モニタはオブジェクトの機能であることが理解されよう。現に総てのクラスはスーパークラスとして Object というクラスを持ち、そこに wait()や notify()などが定義されている。

あるオブジェクトのフィールドへの排他的で一貫したアクセスを必要とするスレッドは、それらのフィールドにアクセスす前にそのオブジェクトのモニタ・ロックを取得し、またそのアクセスが終了したらそのモニタ・ロックを開放しなければならない。

スレッド同士が同期化されたメソッドを呼ぶときには、次のようにオブジェクトがベースとなっている。

オブジェクト・ロックのルール：

- 同期化されたメソッドには複数のスレッドのアクセスが直列化されるが、その中で wait によりスレッドが待機状態になったときは、次のスレッドはそこまでの進行が許される。(Thread.sleep の場合は後続のスレッドはそのメソッドの入り口で同期化される)
- 同じオブジェクトの二つの同期化されたメソッド各々を、各々別のスレッドがアクセスしようとすると、二つのスレッド同士は同期化される。
- 同じオブジェクトのひとつの同期化されたメソッドと同期化されていないメソッドを各々別のスレッドがアクセスするときは、二つのスレッドは平行して進行する。
- 二つのスレッドが各々別のオブジェクトの同期化されたメソッドを呼ぶときは、二つのスレッドは平行して進行する。
- ただしスタテックなメソッド（クラス・メソッド）はオブジェクトではなくクラスとして上記ルールが適用される。つまり上記ルールのオブジェクトをクラスと置き換える。そのクラスのどのインスタンスたちのロックとは区別される。
- あるスレッドがモニタ・ロックを開放するときは、そのアクション（メソッドまたはブロック）とそれにつづく同じロックの取得間には事前発生関係が確立される。

Java ではとりわけ容易にこのオブジェクトのロックのメカニズムを利用できるようになっている。メソッドの定義に”synchronized”キーワードを付加すれば良い。あるスレッドがあるオブジェクトの synchronized メソッドを呼んだときは、該スレッドは該オブジェクトのロックを自動的に取得する。他のスレッドが該オブジェクトの同じメソッド、あるいは該オブジェクトのほかの synchronized メソッドを呼んでも最初のスレッドがロックを保持している限り待たされ待ち行列に入れられる。最初のスレッドが該 synchronized メソッドを抜ければロックは解除される。これがさっきのルールの元の仕組みである。

上記の同期メソッド(Synchronized Methods)にたいし、同期ブロック(Synchronized Blocks)（あるいは同期ステートメント(Synchronized Statements)ともいう）を使うことも出来る。同期ブロックを使う場合は、モニタ・ロックを取得するオブジェクトを指定しなければならない。

次のコードを見てみよう：

```
public void addName(String name) {
   synchronized(this) {
      lastName = name;
```

```
        nameCount++;
    }
    nameList.add(name);
}
```

この例では、addName メソッドは lastName と nameCount への変更に際しては同期化が必要だが、他のオブジェクトのメソッドの呼びだしを同期化するのを回避する必要がある。**同期化されたコードから他のオブジェクトのメソドを呼び出すのは、ライブロック問題を起こす可能性がある**。addName メソッドを同期化しようとすると、nameList.add を呼び出すためだけの非同期化メソッドを別途作らないといけなくなる。

コンテキストの属性を扱うには、一般的に次のような書き方になろう(要求オブジェクトは要求スレッドが連れてきたものなので、スレッド安全であることに注意のこと):

```
      synchronized(context)
      {

          context.setAttribute("key","value");

       }
```

あるいはセッション属性も、場合によっては同期化が必要になることがある(「セッション管理のスレッド対応」の節を参照のこと)。

```
       synchronized(session)

       {

          sesion.setAttribute("key","value");

       }
```

10.1.2 ThreadLocal

Java 1.2 から導入された java.lang.ThreadLocal<T>は、スレッド毎の値を保持する為のクラスで、あるクラス(のインスタンス)がマルチスレッドで呼ばれる際に、**スレッド毎に異なる値(特にスレッド安全でないクラスのインスタンス)を使いたい場合に使用する**。各スレッドはあるインスタンスの自分用のコピーを持つことが出来るので、このクラスを使えばコスト(メモリ容量とスループットのオーバヘッドの双方で)がかかる同期化を使わないで、スレッド安全でないクラスをマルチ・スレッド環境で利用できるようになる。これはこの章の最初で示した Wikipedia の 4 つのスレッド安全対策の 3 番目に相当する。

サーブレットに於いては、スレッドは要求スレッドであり(ワーカ・スレッドを持つ場合は別として)、スレッド毎にあるオブジェクトを持つよりは HttpRequest のオブジェクトの属性として持たせるほうが好ましい。従って ThreadLocal はスレッド安全でないクラスのオブジェクトを保持させるのに使われる。このクラスの用途に関しては、IBM の資料が参考になる。

まず ThreadLocal を継承したサブクラスを作り、その initialValue()をオーバーライドして実装しておく。このメソッドは、スレッド内で値を取得する際に"そのスレッド専用の値"を生成する為に呼ばれる。(そのスレッド用の値をクリアしない限り、1 スレッドにつき 1 回だけ呼ばれる)

API 参照は次のように書いている:

> このクラスはスレッドローカル変数を提供します。これらの変数は、get メソッドまたは set メソッドを使ってアクセスするスレッドがそれぞれ独自に、変数の初期化されたコピーを持つという点で、通常の変数と異なります。通常、ThreadLocal インスタンスは、状態をスレッドに関連付けようとするクラスでの private static フィールドです(ユーザ ID、トランザクション ID など)。

たとえば、以下のクラスでは、private static ThreadLocal インスタンス(serialNum)は、クラスの static SerialNum.get()メソッドを呼び出すスレッドごとに「シリアル番号」を管理します。スレッドのシリアル番号は、SerialNum.get()の最初の呼び出し時に割り当てられ、その後の呼び出しで変更されることはありません。

```
public class SerialNum {
    // 割り当てられる次のシリアル番号（初期値はゼロ）
    private static int nextSerialNum = 0;
    // そのシリアル番号を返してインクリメントする Integer を返す initialValue メソッドをもつ
    // serialNum という private で static な ThreadLocal 型のオブジェクト
    private static ThreadLocal serialNum = new ThreadLocal() {
        protected synchronized Object initialValue() {
            return new Integer(nextSerialNum++);
        }
    };
    // get というメソッドはシリアル番号を返す
    public static int get() {
        return ((Integer) (serialNum.get())).intValue();
    }
}
```

各スレッドは、スレッドが生存していて ThreadLocal インスタンスがアクセス可能である間は、スレッドローカル変数のコピーへの暗黙的な参照を保持します。スレッドが終了すると、スレッドローカルインスタンスのコピーは、すべてガベージコレクトされます(これらのコピーへの参照がほかに存在する場合を除く)。

このクラスでは 4 つのメソッドが定義されている:

1. protected T **initialValue**()
   このスレッドローカル変数に対する現在のスレッドの初期値を返す。このメソッドは、スレッドローカルごとにアクセスしているスレッド当たり 1 回だけ呼び出される。最初に、スレッドは get()メソッドで変数にアクセスする。スレッドが get メソッドを呼び出す前に set(T)メソッドを呼び出す場合、initialValue メソッドはスレッドで呼び出されない。
   この実装は、単に null を返すだけなので、プログラマがスレッド・ローカル変数を null 以外の値で初期化する場合、ThreadLocal をサブクラス化して、このメソッドをオーバーライドする必要がある。通常、匿名の内部クラスが使用される。initialValue の典型的な実装は、適切なコンストラクタを呼び出して、新たに構築されたオブジェクトを返すものになる。
2. public T **get**()
   このスレッドローカル変数の現行スレッドのコピー内の値を返す。スレッドで初めてこのメソッドが呼び出されたときは、コピーが作成されて初期化される。
3. public void **set**(T value)
   このスレッドローカル変数の現在のスレッドのコピーを指定された値に設定する。スレッド・ローカルの値を設定するのに initialValue()メソッドに大きく依存している多くのアプリケーションではこの機能は必要ない。
4. public void **remove**()
   この ThreadLocal の値を削除する。削除すると、ThreadLocal の格納要件が少なくなる。この ThreadLocal がもう一度アクセスされると、ThreadLocal はデフォルトの initialValue に設定される。

このクラスの実際の使い方は、この章の「スレッド安全でない API の対策」の節を参照されたい。

10.1.3 Java.util.concurrent パッケージ

Java 1.5 から Java.util.concurrent という並行処理のためのパッケージが提供されている。これはこれまでのスレッド用の API を強化したものである。これにより複雑なマルチスレッド処理の開発がより容易になった。Java が持っている同期化（メソッドとブロック）はロック・ベースのアプリケーションには有効なものではあるが、以下のような限

界があった：

- 既に所有されているロックを取得しようとすることから撤回する、一定の時間を待ったあとで取得するのをあきらめる、あるいは割り込み後にロック取得をキャンセルする手段がない
- 例えばリエントラント性、書き込み対読み出し保護、あるいは公正性に関して、ロックに対する意味合いを変更できない
- 同期化の制御へのアクセスが出来ない。どのアクセス可能なオブジェクトに対しても、どのメソッドもsynchronized (obj)を実施できる
- 同期化はメソッドとブロック内でなされるので、厳格なブロック構造のロックに使用が制限される。言い換えると、あるメソッド内でロックを取得して他のメソッド内でそのロックを開放したりできない。実際のアプリケーションではより広いスコープ（適用域）でのロックが必要になることがある

そのような問題は排他ロックのユーティリティ・クラス（例えばセマフォ）を作って克服できようが、通常この場合は同期メソッドや同期ブロックのようにネスト（入れ子構造）はできない。Java.util.concurrent を利用すれば標準化されたクラスを使ってこれらを解決できる。このパッケージは NY 州立大学教授で JSR 166 委員長だった Doug Lea が作り、1990 年代終わりのころから使われていた concurrent パッケージが基になっている。これにより、ソフトウエア開発が容易になる、高い性能を活用できる、信頼性が向上する、及び維持が容易になる、などの利点が得られる。

特に 4 番目の問題点に関して言えば、同期ロックのメカニズムより広いスコープ（適用域）のロックのメカニズムが必要になることもある。その為にはスレッド間にあるオブジェクトのアクセス可否を通知しあう仕掛けが必要である。これがビジー・フラグとかセマフォとかよばれるものである。そのような事例は、以前のチュートリアルで紹介したやや古い書物ではあるが、Scott Orks と Henry Wong の”JAVA Threads”という著書（オライリー・ジャパンから邦訳版が出版され、オーム社から発売されている(ISBN4-900900-54-0)）の第 3 章に示されている。シナリオは銀行の ATM で、「Bob は自分の口座の残高が$500 あることを確認して$450 引きだそうとしたが出来なかった。原因は全く同じ時刻に妻の Jane が$100 引き出したためである」。この事例はシステムとしては間違ってはいないが、顧客に対してはきちんとしたサービスにはなっていない。このとき顧客は不審に思って銀行に苦情を申し入れるであろう。従ってひとつの口座に対して同時にはひとつの ATM にしかサービスできないとすべきであろう。銀行の窓口が係りをひとり置き（これはシングルトンのサーブレットが一括共有資源を管理する方法である）、これが ATM と口座の対応をとり、複数の ATM が同じ口座へのサービスを要求してきたらそれをひとつずつ対応すればよい。しかしながらそうしない場合は、モニタのメカニズムでこれを解決しようとしても無理である。ある顧客のセッションのあいだ（これをクリチカル区間(critical section)ともいう）その顧客の口座のアクセスを他のスレッドにロックするのは、モニタのメカニズムの領域を越えている

java.util.concurrent パッケージには以下のような同期化のために有用なユーティリティ・クラスがある：

1. Semaphoe：
   これは古典的な同期化のツールのセマフォである。Scott Orks と Henry Wong の著書の 4.2 節に BusyFlag というクラスが示されていたが、java.util.concurrent.Semaphoe は同じ機能を持っているロックであるが、カウンタが付いたロックを持っている。このクラスは許可のセットを管理する。acquire()はひとつの許可が使えるようになってそれを得るまでそのスレッドをブロックする。各 release()ごとに許可が追加される。許可というオブジェクトが使われている訳ではないことに注意されたい。Semaphoe は利用可能な許可のカウントを保持している。Semaphoe はスレッドだけでなくプロセス間の排他制御にも使える。

2. Barrier
   これはスレッドたちのセットが総てある共通のバリア点に達するまで互いに待つことが出来る同期化のひとつのツールである。このバリアとのインターフェイスは CyclicBarrier クラスで、待っていたスレッドたちがリリースされたら再使用できるのでサイクリックという名前が付いている。これは並行処理には有用である。

3. CountDown Latch（カウントダウン・ラッチ）
   ラッチというのは、最初は false だが一旦 true にセットされると永久に true になったままになるものである。

java.util.concurrent.CountDownLatch クラスは、他のスレッドたちで行われている操作たちのセットが完了するまでひとつあるいはそれ以上のスレッドを待たすことが出来るようにする同期化のひとつのツールである。これはある条件が満たされるまでスレッドたちを待たすという意味では前記のバリアと同じではあるが、解放の条件が待機しているスレッドの数ではないことである。その代わり指定したカウントがゼロになったらスレッドたちが解放される。初期カウント数はコンストラクタで指定する。このラッチはリセットすることは無いし、その値を後で下げることもできない。

4. Exchanger
これは 2 つのスレッドが待ちあわせ（ランデブ）点でオブジェクトを交換できるようにするもので、パイプラインの設計には有用なものである。各スレッドは exchange()メソッドの入り口で何らかのオブジェクトを提示し、その代りに他のスレッドが提示したオブジェクトを受け取る。これは Producer-Consumer（生産者-消費者）パタンのアプリケーション（2 つのスレッド間でデータを共有し、producer がデータを作り、consumer がそれで何かをするアプリケーション）に有用である。

10.1.4 Java.util.concurrent.locks パッケージ

このパッケージでは従来の同期化とモニタとは異なるロックと条件待ちのクラスたちが用意されている。このパッケージはより柔軟なロックと条件の使い方ができるが、文法はやや面倒くさくなる：

1. ReentrantLock
この ReentrantLock は再入可能（同じスレッドが同じロックを繰り返し取得できる）な排他ロックである。このロックは最後に成功裏にロックしたがまだロック解除していないスレッドが所有する。ロックが別のスレッドに所有されていない場合、lock を呼び出したスレッドは復帰してロックの取得に成功する。現在のスレッドがロックをすでに所有している場合、メソッドはただちに復帰する。これは、isHeldByCurrentThread()および getHoldCount()メソッドを使用してチェックできる。

このクラスのコンストラクタは、オプションとして「公平性」パラメータを受けつける。これが true に設定されると、競合が存在する場合、ロックはもっとも長く待機しているスレッドへのアクセスを許可するように応答する。そうでない場合、このロックが特定のアクセス順を保証しない。多数のスレッドによりアクセスされる公平ロックを使用するプログラムは、デフォルト設定を使用するプログラムよりも低い(より低速な、多くの場合非常に低速な)全体スループットをもたらす場合があるが、ロックを取得する際の変動はより小さくなり、枯渇しないことが保証される。ただし、ロックの公平性により、スレッドスケジューリングの公平性が保証されるわけではない。このため、公平ロックを使用する多数のスレッドの 1 つが複数回連続して取得し、アクティブなほかのスレッドの進捗が見られず、ロックを保持していない状態になることもある。また、時間指定のない tryLock メソッドは公平性設定を尊重せず、受け入れない。ほかのスレッドが待機中でもロックが有効であればこのメソッドは成功する。

2. Reader / Writer Locks
あるオブジェクトからあるスレッドがデータを読み出すときには他のスレッドも同時に同じデータを読むのを待たせる必要は無い。スレッドたちがデータを読み出していて、そのデータが変更されない限りは並行読み出しをしても構わないはずである。このパッケージはこの種のロックを実装できるクラスたちを含んでいる。ReadWriteLock インターフェイスはそれに関わる 2 つのロックを保持し、ひとつはリード・オンリの為、もうひとつは書き込みの為である。readLock()はライタが存在しな限り複数のスレッドが同時に保持できる。writeLock()のほうは排他的なロックである。

理論的には、同時性を高めるために Reader / Writer ロックを使うことで、Mutex ロックを使うよりは性能改善が図れることになるが、この性能改善はマルチ・プロセサ環境、データの変更に比べて読み出しの頻度が大きい、また読み出し操作と書き込み操作の時間の比が大きいときにその効果が大きい。

10.1.5 Java.util.concurrent.atomic パッケージ

このパッケージは単一の変数でのロック不要でスレッド安全なプログラミングをサポートするクラスたちが含まれている。これらのクラスは volatile（揮発性な）値、フィールド、及び配列要素の概念を、アトミックな条件付き更新操作も提供するクラスにまで拡張する。例えば既に述べたように、揮発的な整数では++演算子は使えない。何故ならこの演算子は複数の命令を含んでいるからである。AtomicInteger クラスは、それが持っている整数値を同期化を使うことなくアトミックにインクリメントするメソッドを持っている。しかしながらアトミックなクラスは同期化を必要としないコードを作るといったより複雑なタスクにも使える。

次のサンプル・コードは AtomicLong クラスを使ったシーケンス番号発生器である：

```
import java.util.concurrent.atomic.*;

public class SequenceGenerator {
   private AtomicLong number = new AtomicLong(0);
   public long next() {
     return number.getAndIncrement();
   }
}
```

AtomicBoolean、AtomicInteger、AtomicLong、及び AtomicReference のクラスのインスタンスの各々が、各々の型の単一の変数へのアクセスと更新ができる。加えて各クラスは、アトミックなインクリメントといったタイプのユーティリティのメソッドを持っている。これらのクラスの getAndSet()メソッドは、同期ロックなしでその変数に新しい値をセットし、これまでの値を返す。compareAndSet()と weakCompareAndSet()メソッドは条件付き変更メソッドである：これらのメソッドは 2 つの引数を持っており、値はこのメソッドが開始するときに持っていることが期待される値で、新しい値はそれにセットする値である。言い換えると、これらのメソッドは現在の値が期待される値と同じであるときにのみある値を自動的にセットできる。

これらのメソッド仕様に基づく JVM 実装により、最新のプロセッサで使用可能な高効率のマシンレベルのアトミック命令を使用することが可能になる。ただし、一部のプラットフォームでは、これをサポートすることで、なんらかの内部ロックが伴う可能性がある。このため、メソッドで非ブロッキングが厳密に保証されるわけではなく、スレッドは、操作を実行する前に一時的にブロックを実行することがある。

10.2節　複数スレッドの同時通過の確認

サーブレット・エンジンはクライアントからの HTTP 要求をひとつのスレッドとして、要求と応答のオブジェクトを連れてそのサーブレットの service メソッドに送りこむ。クライアントからの要求ごとにスレッドは生成される。スレッド間のタイミングはなく、service（さらにそれが呼ぶ doPost や doGet など）メソッドには複数のスレッドが走ることになる。

それを確認する為に SimpleThreadTest という簡単なサーブレットを紹介しよう。このサーブレットは 10 秒後にクライアントに画面なしの応答を返すだけである。この 10 秒間は、ほかのスレッドもこのサーブレットに入るのに十分な時間である。このサーブレットは入ってきた、及び出て行ったスレッドと、このサーブレットに存在しているスレッドの数をログに出力する。

以下はそのコードである：

```
package concurrency;
```

```
import java.io.IOException;
import java.util.Calendar;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * SimpleThreadTest は、複数のスレッドがサーブレットを
 * 同時通過することを確認する為のサーブレット
 * December 2010, CRESC Corps.
 */
@WebServlet("/SimpleThreadTest")
public class SimpleThreadTest extends HttpServlet {
   private static final long serialVersionUID = 1L;
   private final static Logger LOGGER = LoggerFactory.getLogger(SimpleThreadTest.class);
   // serviceCounter は現在通過中のスレッドの数
   private int serviceCounter = 0;
   // serviceCounter にアクセスするメソッドたち
   protected synchronized void enteringServiceMethod() {
      serviceCounter++;
      LOGGER.info(String.format("スレッド参加: %tc: 参加スレッド数：%d "
            , Calendar.getInstance(), numServices()));
   }
   protected synchronized void leavingServiceMethod() {
      serviceCounter--;
      LOGGER.info(String.format("スレッド離脱: %tc: 参加スレッド数：%d "
            , Calendar.getInstance(), numServices()));
   }
   protected synchronized int numServices() {
      return serviceCounter;
   }
   // スレッドが service メソッドを通過する前後で counter を増減
   protected void service(HttpServletRequest req,
         HttpServletResponse res)
   throws ServletException,IOException {
      enteringServiceMethod();
      try {
         super.service(req, res);
      }
      finally {
         leavingServiceMethod();
      }
   }
   public void doGet(HttpServletRequest req,HttpServletResponse res){
      performTask(req, res);
   }
   public void doPost(HttpServletRequest req,HttpServletResponse res){
      performTask(req, res);
   }
   // このサーブレットは単に 10 秒間待つだけ
   private void performTask(HttpServletRequest req,HttpServletResponse res){
      try {
         Thread.sleep(10 * 1000);
      } catch (InterruptedException e) {
      }
   }
}
```

このコードの多くは「サーブレットの終了」の節で示されているコードをもとにしている。

ブラウザ（例えば Internet Explorer）を 2 つたち上げて、http://localhost:8080/tutorial/SimpleThreadTest を 10 秒以内にこれらのブラウザからアクセスすると、次のようなログが出力される：

```
183506 [http-8080-exec-5] INFO concurrency.SimpleThreadTest - スレッド参加: 土 12 11 11:12:54 JST 2010: 参加スレッド数：1
184443 [http-8080-exec-4] INFO concurrency.SimpleThreadTest - スレッド参加: 土 12 11 11:12:55 JST 2010: 参加スレッド数：2
```

```
193507 [http-8080-exec-5] INFO concurrency.SimpleThreadTest - スレッド離脱: 土 12 11 11:13:04 JST 2010: 参加スレッド数：1
194444 [http-8080-exec-4] INFO concurrency.SimpleThreadTest - スレッド離脱: 土 12 11 11:13:05 JST 2010: 参加スレッド数：0
```

この例では、このサーブレットに 1 秒間の間隔をおいて 2 つのスレッドが入ってきており、また同じ間隔をおいて 10 秒後に抜けていることがわかる。

それでは Firefox Mozzila の場合はどうだろうか？

```
480730 [http-8080-exec-8] INFO concurrency.SimpleThreadTest - スレッド参加: 土 12 11 11:17:52 JST 2010: 参加スレッド数：1
490731 [http-8080-exec-8] INFO concurrency.SimpleThreadTest - スレッド離脱: 土 12 11 11:18:02 JST 2010: 参加スレッド数：0
490732 [http-8080-exec-8] INFO concurrency.SimpleThreadTest - スレッド参加: 土 12 11 11:18:02 JST 2010: 参加スレッド数：1
500747 [http-8080-exec-8] INFO concurrency.SimpleThreadTest - スレッド離脱: 土 12 11 11:18:12 JST 2010: 参加スレッド数：0
```

このブラウザでは、10 秒以内に 2 つのブラウザからこのサーブレットをアクセスしたにも関わらず、**片方のブラウザからの HTTP 要求は、もうひとつのブラウザから先に出された HTTP 要求が戻ってくるのを待っている**（即ち同じアドレスに対しては HTTP 要求を直列化している）ことが判る。これはセッションの章で説明したように、同じセッションを持つ HTTP 要求が同じサーブレットを並行通過するのを防止するという意味では好ましい。しかしながら、**再読み込みボタンをクリックした場合はそうはならず**、この場合はやはり同じセッションを持ったスレッドが複数通過できることは、簡単に確認できる。これは Google の Chrome でもいえる。

表 10-1：複数のタブまたは画面から SimpleThreadTest をアクセスした場合のブラウザの動作

	Internet Explorer	Apple Safari	Firefox Mozzila	Google Chrome
2 つの画面またはタブからほぼ同時アクセス	2 つの HTTP 要求が続けて送信される	2 つの HTTP 要求が続けて送信される	2 つの HTTP 要求は直列化されて送信される（最初の要求に対する応答が戻ってくるまでは次の要求を送信しない）	2 つの HTTP 要求は直列化されて送信される（最初の要求に対する応答が戻ってくるまでは次の要求を送信しない）
更新ボタンを続けて押す	その数の HTTP 要求が送信される	その数の HTTP 要求が送信される	その数の HTTP 要求が送信される	その数の HTTP 要求が送信される
取り消しと更新を続けて押す	更新ボタンを押した数の HTTP 要求が送信される	更新ボタンを押した数の HTTP 要求が送信される	更新ボタンを押した数の HTTP 要求が送信される。更新と取り消しのボタンはフリップ・フロップ	更新ボタンを押した数の HTTP 要求が送信される。更新と取り消しのボタンはフリップ・フロップ

同じセッションを持ったスレッドが同じアプリケーション内を複数並行して通過する可能性に関しては、「セッション管理のスレッド対応」の節で指摘したとおりである。

10.3節　SingleThreadModel

クライアントからの要求ごとにスレッドは生成される。スレッド間のタイミングはなく、service（さらにそれが呼ぶ doPost や doGet など）メソッドには複数のスレッドが走ることになる。そのような状態においても問題が無い（スレッド・セーフあるいはスレッド安全という）サーブレットが開発されねばならない。

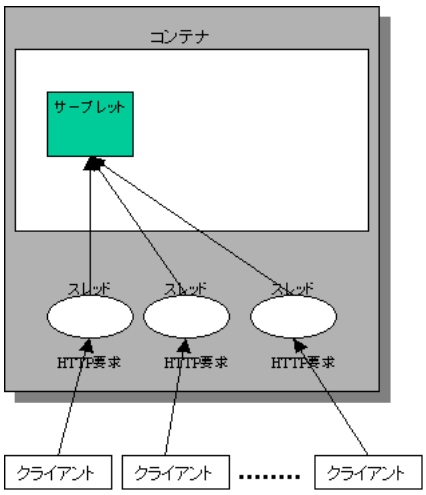


図 10-1：通常の HTTP 要求スレッド

サーブレットが、パラメタとして持ち込まれた HttpServletRequest や HttpServleResponse のオブジェクトや service メソッド内でローカルに宣言した変数やパラメタを使っている限りスレッド・セーフであり、心配する必要が無い。HttpServletRequest や HttpServleResponse のオブジェクトは参照（云ってみればポインタ）としてスレッドが持ってきたものであり、またメソッド内のローカル変数は各スレッド毎に割り当てられる（リエントラントな構造という）からである。**しかしながらそのサーブレットで宣言したインスタンス変数やクラス変数は各スレッドに共有される。**「共有資源への同時アクセス」の項で示したように、そのサーブレット外のセッション、コンテキストなどのスコープ・オブジェクト、データベース、ヘルパー、ネットワーク接続などの資源もスレッドが共有する。その場合はスレッド間の干渉には十分な注意が必要である。

初期のサーブレット API には SingleThreadModel というインターフェイスが用意されていた。**現在はこのインターフェイスは廃止対象(Deprecated)**になっている。このインターフェイスを実装(implement)すると、下図のようにエンジンはコンテナ上にそのサーブレットの複数のインスタンスのプールを用意し、ひとつのサーブレットのオブジェクトにはひとつのスレッドしかアクセスしないよう制御する。従ってインスタンス変数の共有問題は解決されるが、クラス変数は各スレッドによって共有されるので、問題は残る。

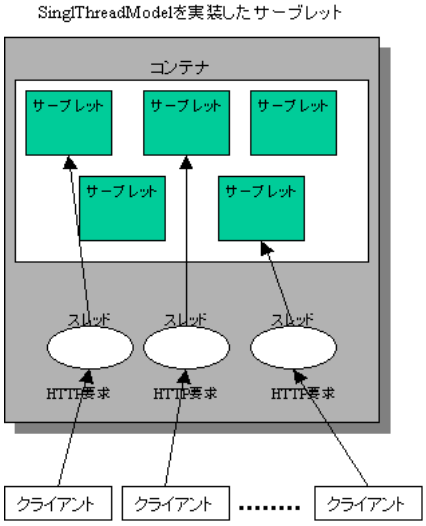


図 10-2：SingleThreadModel での HTTP 要求スレッド

何個のインスタンスを生成するか（IBM の WebSpere では 4 個以上であった）、それ以上の HTTP 要求がきたらどうするか（多分待ち行列に入れる）はサーブレット・エンジンの実装の問題である。

またサーブレット仕様書は **service（さらにそれが呼ぶ doPost や doGet など）メソッドに synchronize キーワードを**

付して同期化しないよう強く勧告している。確かに入り口のメソッドを同期化すればそこでスレッドは直列化されスレッド競合問題は生じなくはなるが、その為スループットが大きく落ちてしまうからである。

SingleThreadModel インターフェイスを実装した場合に、サーブレット・エンジンが何個までのインスタンスを生成したかを知る簡単なサーブレットを紹介しよう。プログラム・リストを見ていただければ判ると思うが、static Hashtable instances なる変数は現在生成されてコンテナ上に配備されているインスタンスの数を蓄積するテーブルである。各インスタンスは自分がインスタンス化されたときに init メソッドにて自分自身をこのテーブルに登録する。反対に削除されるときは destroy メソッドにて自分自身の登録を削除する。またサーブレットにスレッドが通過するごとに自分のオブジェクトをこのテーブルに追加する。同じキーで追加した場合はテーブルのサイズは変わらない。従ってこのテーブルに何個のオブジェクトが登録されているかでコンテナ内のこのサーブレットのインスタンス数を知ることができる。

@SuppressWarnings("deprecation")というアノテーションはこのインターフェイスは廃止対象であるという警告を抑制する。また画面をクライアントに送った後 10 秒間のポーズをおいてスレッドがこのサーブレットから戻るので、この間に別の HTTP 要求をクライアントから送信して、新たなインスタンスのコピーがそのスレッドに渡されるのを確認できる。

```
package concurrency;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Hashtable;

import javax.servlet.ServletException;
import javax.servlet.SingleThreadModel;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * このサーブレットは呼び出されるたびにその呼出回数とインスタンス数を返す
 * 作成日 : (01/07/05 13:26:15)(10/11/29 Revised for Tomcat 7.0)
 * @author: Cresc
 */
@WebServlet("/HowManyInstances")
@SuppressWarnings("deprecation")
public class HowManyInstances extends HttpServlet implements SingleThreadModel {
// public class HowManyInstances extends HttpServlet{

   private static final long serialVersionUID = 1L;
   static int totalCount = 0;  // 全てのインスタンスが共有するアクセス回数カウンタ
   int count = 0;              // 各インスタンス毎の呼び出された回数カウンタ
   static Hashtable<String, HowManyInstances> instances = new Hashtable<String, HowManyInstances>();
// 全てのインスタンスが共有するインスタンス計数用Hashtable

   /**
    * サーブレット・エンジンはインスタンスの生成にあたってこのメソッドを呼ぶ
    * 自分自身をHashtableに登録してコンソールに表示
    */
   public synchronized void init() {
      // 自分の文字列表現をキーとして自分を登録するとこのクラスのインスタンスの総数を示す
      try {
         super.init();
      } catch (ServletException e) {
         e.printStackTrace();
      }
      instances.put(this.toString(), this);
      System.out.println("インスタンス生成: " + this.toString());
      System.out.println("インスタンスの数: " + instances.size());
   }

   /**
    * サーブレット・エンジンはインスタンスの削除にあたってこのメソッドを呼ぶ。
```

```
    * 自分自身をHashtableから削除してコンソールに表示
    */
   public void destroy() {
      instances.remove(this.toString());
      System.out.println("インスタンス削除: " + this.toString());
      System.out.println("インスタンスの数: " + instances.size());
      super.destroy();
   }

   /**
    * クライアントにアクセス回数と現在のインスタンスの数を報告する
    */
   public void performTask(HttpServletRequest request, HttpServletResponse response) {
      try
      {
         response.setContentType("text/html; charset=Windows-31J");
         PrintWriter out = response.getWriter();
         out.println("<html><head><title>HowManyInstances</title>");
         out.println(         // 同じことをHTMLテキストでも指定
               "<meta http-equiv=\"cache-control\" content=\"no-cache\">" +
               "<meta http-equiv=\"content-type\" content=\"text/html; charset=Windows-
31J\"></head>");
         out.println("<body><h1>HowManyInstances サーブレット</h1><pre>");
         out.println("このサーブレットのインスタンス: " + this.toString());
         count++;
         out.println("インスタンス化されてからの呼び出し回数: " + count);
         instances.put(this.toString(), this);
         out.println("コンテナ内のインスタンスの数: " + instances.size());
         totalCount++;
         out.println("インスタンス全部がアクセスされた回数: " + totalCount);
         out.println("</pre></body></html>");
         out.flush();
         out.close();
         response.flushBuffer();
         //10秒間このスレッドを待たせ、このインスタンスをビジーにする
         //この間別のブラウザやタブからこのサーブレットをアクセスして実験する
         Thread.sleep(10000L);
      }
      catch(Throwable theException)
      {
      }
   }

   public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
      performTask(request, response);
   }

   public void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
      performTask(request, response);
   }

   public String getServletInfo() {
      return "HowManyInstances, Version 1.1 by Cresc";
   }

}
```

下図はこのサーブレットを4つのブラウザ(このテストではIEまたはSafariを使う必要があることは、前節で示した)からアクセスした例である。最初にひとつのブラウザからhttp://localhost:8080/tutorial/HowManyInstancesをアクセスし、これが戻ってくる10秒間以内に残ったブラウザに同じアドレスを順番にアクセスすると、サーブレット・コンテナは最初にサーブレットのインスタンスとは別のインスタンスを新たな要求スレッドに割り当ててくれる。Tomcatの場合はIBMのWebShereとは実装が異なっており、最初からある数のインスタンスを用意することはしていない。最初のHTTP要求が到来すると、Tomcatはこのサーブレットのclassをロードしてインスタンスをただひとつ生成する。そしてその要求スレッドに対し**そのインスタンスのコピー**をひとつ割り当てる。従って、Tomcatの場合はこのサーブレットのinitメソッドは1回しか呼ばれない。しかしながら総てのインスタンスでdestroyメソッドはコンテキストの終了時に呼ばれる。そのスレッドがserviceを抜ける前に新しい要求が発生したらまた新たな

コピーを作ってそのスレッドに割当てる。下図の例では、コンテナ内のインスタンスの数は 8 まで増加したが、そのうちひとつはオリジナルのインスタンスで、要求スレッドには割り当てられない。

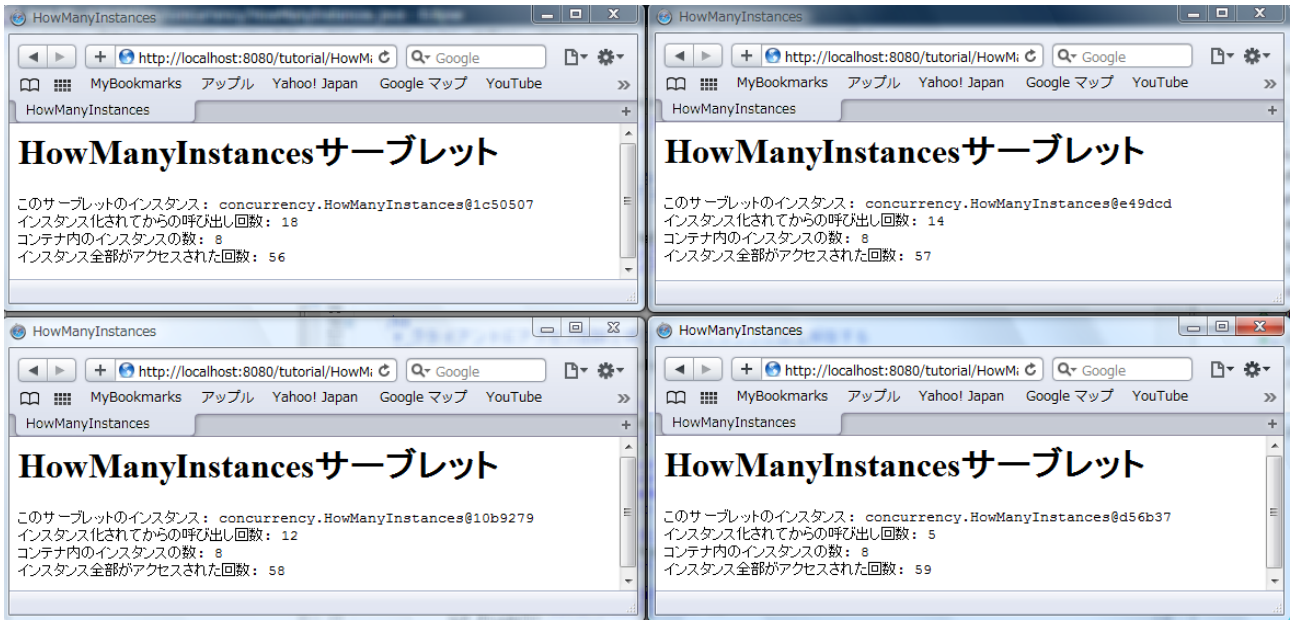


図 10-3：HowManyInstances のアクセス例

10.4節　　サーブレット内での同期ブロック

サーブレットでは、インスタンス変数、ServletContxt 及び HttpSession のオブジェクトなどはスレッド安全でないことは既に述べたとおりである。そのようなオブジェクトをアクセスするには：

- Beans のようにセッタ/ゲッタをもったクラスでラップし、そのクラスのセッタ/ゲッタのメソッドたちを同期化する。これは「セッション」の章の教材の MessageManager クラスが良い例である。
- service (doGet、doPost....) メソッドの中から ServletContxt 及び HttpSession のオブジェクトをアクセスする個所を同期ブロック化する。
- より大きなスコープを持ったオブジェクトの場合は、Java.util.concurrent が用意しているロックを使用する

などが考えられよう。

同期ブロックはそのなかでも良く使われる手段であろう。なお、「同期化」の項で述べたように、同期化されたコードから他のオブジェクトのメソドを呼び出すのは、ライブロック問題を起こす可能性があるので注意しなければならない。

以下は HttpSession オブジェクトをアクセスする個所を同期ブロックにしたものである。

```
public void doGet(…) {
…
      HttpSession session = request.getSession();
      synchronized(session) {
            session.setAttribute("attributeName", "attributeValue");
            out.println(session.getAttribute("attributeName"));
      }
      …
```

```
}
```

取得するロックはセッション・オブジェクトのロックであることに注意されたい。synchronized(this)を使ってはいけない。

同期ブロックは何個存在してもかまわない。session というオブジェクトのロックはひとつしか存在せず、ブロック間で排他制御がされる。処理能力を下げない為に、同期ブロックはなるべく小さくし、かつブロック数を多くしないことが好ましい。この方法はうっかり session を使うコードをブロック外に置いてしまう可能性があるので、注意が必要である。コンパイラはそれを間違いだと指摘してくれない。

10.5節　スレッド安全でない API への対策

Java の標準の API のほぼ総てがスレッド安全であるが、例外的にそうでないものが存在する。その典型的なものが java.text.**SimpleDateFormat** というクラスで、これは非常に有用な API で良く使われているが、スレッド安全でない例外的な API のひとつであるため、良く問題を起こすことでも有名である。サーブレット・コンテナ上で使用するときも、きちんとスレッド安全な使い方をしないといけない。スレッド安全でない他のクラスには:

- 数字用の NumberFormat、DecimalFormat、および DecimalFormatSymbols
- 日付や時間用の DateFormat、SimpleDateFormat、および DateFormatSymbols
- テキスト・メッセージ用の MessageFormat、および ChoiceFormat

がある。

更に**同じ問題を持っているのが 1.5 版から導入された Formatter（フォーマッタ）というクラスと Formattable というインターフェイスである**。Formatter は C を知っているに人にはわかりやすい printf に似た書式付の文字列出力用のインタープリタである。書式を指定して、ある値（数値、日付、時刻など）を文字列に変換する。

しかしながら同じく 1.5 版で導入された

- **PrintStream#format / printf（及びこれを使っている System.out.printf や System.err.printf など）**
- **及び String#format**

は、Formattable を暗黙的に実装しているにもかかわらず、スレッド安全であるので、極力これらを使用することをお勧めする。API ドキュメントには次のように書かれている:「マルチスレッドアクセスを実行する場合、Formattable は必ずしも安全ではありません。スレッドの安全性は、このインタフェースを拡張および実装するクラスによってオプションで保証されます。」

10.5.1　同期メソッドによる対策

もう既に閉鎖されているが、ドイツのソフトウエアのコンサルタントの Steve McLeod 氏がインターネット上に掲載した対策には、以下のようなコードがあった:

```
//safe - uses synchronization
import java.util.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

private static DateFormat df=SimpleDateFormat.getInstance();

public synchronized static String format(Date date){
   return df.format(date);
}
```

この方法は df というスタティックでプライベートなインスタンスを用意し、それに対し同期化されたスタティックなメソッドを用意するものである。ユーザはこの format というメソッドをアクセスする。Format 以外のメソッドを使う時はそれに応じた同期化されたメソッドを幾つか用意すれば良い。同期化されたメソッド間では排他制御がされる。この例で使われている getInstance()というメソッドはデフォルトの日付と時間共に SHORT のスタイルを使うフォーマッタのインスタンスを取得するものである。そうでなくてパタンとロケールを指定したいときは new キーワードを使って SimpleDateFormat のコンストラクタを呼べばよい。

10.5.2 同期ブロックによる対策

これは synchronized ステートメントでそのブロックへのスレッドたちを同期化(待ち行列をつくる)させるものである。この場合はモニタ・ロックを使うオブジェクトを指定することになる:

```
//safe - uses synchronized block
import java.text.SimpleDateFormat;

public static SimpleDateFormat df=new SimpleDateFormat(); //designate pattern and locale here if
necessary

// ……..

synchronized (df){
   // use df only within this synchronized block
}
```

これは **df というオブジェクトを使うコードを必ず同期ブロックに含める**ものである。同期ブロックは何個存在してもかまわない。df というオブジェクトのロックはひとつしか存在せず、ブロック間で排他制御がされる。処理能力を下げない為に、同期ブロックはなるべく小さくし、かつブロック数を多くしないことが好ましい。この方法はうっかり df を使うコードをブロック外に置いてしまう可能性があるので、注意が必要である。コンパイラはそれを間違いだと指摘してくれない。

そのようなミスが心配な人は、多少処理能力が落ちても df オブジェクトをブロック内で毎回生成することをお薦めする:

```
//safe - always creates new instances of SimpleDateFormat within a synchronized block
import java.text.SimpleDateFormat;


// ……..

synchronized (this){
   SimpleDateFormat df=new SimpleDateFormat(); //designate pattern and locale here if necessary
   // use df only within this synchronized block
}
```

この場合は同期ブロックはロックを取得するオブジェクトは特にないので this としておく。

10.5.3 ThreadLocal を使ってスレッド毎に DateFormat のオブジェクトを持つ方法

これは ThreadLocal の項で説明したスレッド毎にスレッド安全でないクラスのコピーをもつ方法である。これは The Java Specialists' Newsletter の 172 号に Dr. Heinz M. Kabutz が寄稿したものが参考になる。

SoftReference を含まない ThreadLocal<DateFormat>のオブジェクト tl で先ず考えてみると、スレッドは DateConverter をインスタンス化することで DateFormat 型のデータを持つ tl を持ち、testConvert というメソッドを呼ぶと、getDateFormat でその自分の DateFormat オブジェクトを呼び出し、それの parse メソッドで入力文字列を解析し、format メソッドで目的の文字列に変換する。そのスレッドが存在している限り、たとえそのスレッドが再度 DateFormat を使うことが無いにしても、この tl オブジェクトは存在し続ける。SoftReference を使用したのは、これを配慮したためである。SoftReference はメモリー要求に応じてガベージ・コレクタの判断でクリアされるソフト参照

オブジェクトである。もしガベージ・コレクタによってメモリ上から削除されていたときには、getDateFormat メソッドは新しく SoftReference を作成している。

```
import java.lang.ref.SoftReference;
import java.text.*;
import java.util.Date;

public class DateConverter {
  private static final ThreadLocal<SoftReference<DateFormat>> tl
      = new ThreadLocal<SoftReference<DateFormat>>();

  private static DateFormat getDateFormat() {
    SoftReference<DateFormat> ref = tl.get();
    if (ref != null) {
      DateFormat result = ref.get();
      if (result != null) {
        return result;
      }
    }
    DateFormat result = new SimpleDateFormat("yyyy/MM/dd");
    ref = new SoftReference<DateFormat>(result);
    tl.set(ref);
    return result;
  }

  public void testConvert(String date) {
    try {
      DateFormat formatter = getDateFormat();
      Date d = formatter.parse(date);
      String newDate = formatter.format(d);
      if (!date.equals(newDate)) {
        System.out.println(date + " converted to " + newDate);
      }
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

testConver というメソッドの代わりにプログラマたちは必要なコードを自分のアプリケーションにあわせて書けばよい。これにより format あるいは parse のメソッドでのスレッド競合問題を気にしなくて済む。

10.6節　デッドロック、枯渇、及びライブロック

今まで学んできた排他制御だけでは、スレッド安全性は確保できない。この章の最初に示した Wikipedia の対策の 2 番目の相互排他のところで示したように、いわゆるスレッド同士が待ちあうデッドロックなどの問題は相互排他では避けられず、また発生するとサービス停止など深刻な結果をもたらす。これはプログラマの責任であるが、万が一にそなえてデッドロックを検出するウオッチドックなどが高信頼のアプリケーションに必要になることがある。

デッドロック(Deadlock)は 2 つ以上のスレッドが互いに待ちあって膠着状態になることを言う。これは複数のスレッドが複数のリソースを共有していて同じロックたちを取得する必要があるがその順番が異なるときに起き得る。
デッドロックは他の 2 つに比べて発生頻度が高いトラブルである。

枯渇状態(Starvation)はあるスレッドが共有のリソースたちへの通常のアクセスが得られなくなり、進められなくなる状況を言う。これは貪欲(greedy)なスレッドたちによって長期間共有リソースが使えなくなるときに生じる。例えば、戻ってくるのにたまたま長期間かかる同期化されたメソッドを持つあるオブジェクトを考えると、ひとつのスレッドがこのメソッドを頻繁に呼び出していると、同じオブジェクトに頻繁に同期されたアクセスをする必要がある他のスレッドたちがしばしばブロックされてしまうことになる。

ライブロック(Livelock)：スレッドはしばしば別のスレッドのアクションに応じてアクションをすることがある。もしその別のスレッドもまた別のスレッドのアクションに応じてアクションしていると、ライブロックが生じ得る。デッドロックと同様にライブロックも先に進むことが出来ない。しかしながらスレッドたちはブロックされている訳ではない：それらは作業を再開する為に互いに対し対応するのに忙しすぎるだけである。これは通路をたまたま2人が並行して歩いていて、AはBを抜こうとBの右を急ぎ、BはまたAを抜こうとしてAの左を急ぐ状態になったことを想像すれば良い。AとBはともに歩いているが、AとBは互いにブロックしている。

10.6.1 デッドロック

デッドロックは昔から知られていた障害である。「哲学者たちの食事(Dining philosophers)」という有名なシナリオある。これにはバリエーションが幾つもあってどれが最初かはっきりしないが、多分次の論文であろう。
Dijkstra, E. W. “Co-operating Sequential Processes”, Programming Languages, Genuys, F.(ed.), Academic Press, 1965

そのバリエーションのひとつを紹介すると、「専ら考えることを生業とする5人の哲学者たちがひとつの丸い食卓に座っている。中央には大きな皿にスパゲッティが盛ってある。5つのフォークが哲学者たちの間に置いてある。また各哲学者の前には取り皿がおいてある。思考中腹がへったことに気がついた哲学者は、やおら自分の左側に置いてあるフォークを左手にもち、更に右手で右側にあったフォークをとり、二つのフォークを使って自分の皿にスパゲッティをとり、食べる。終わったら二つのフォークを元に戻し再び思考を開始する。さてここで、たまたま5人の哲学者たちが同時にスパゲッティを取ろうと自分の左側のフォークをとったらどうなるであろうか？ 5人とも右側のフォークが空くまで永久に待ちつづけてしまい、飢餓状態に陥る」

これは哲学者たちという5つのスレッドが、5つの共有オブジェクトをアクセスしようとしたために発生している。ともに左のオブジェクトのロックを確保してから右のオブジェクトのロックを確保しようとしたが、右のオブジェクトのロックはすでに確保されており、その状態が複数のスレッドの膠着を招いている。

一番単純なのは、下図のように、二つのオブジェクトを二つのスレッドがアクセスしようとした場合である。二つのスレッドがともに相手がかけたロックが解除されるのを待ちつづけてしまう。

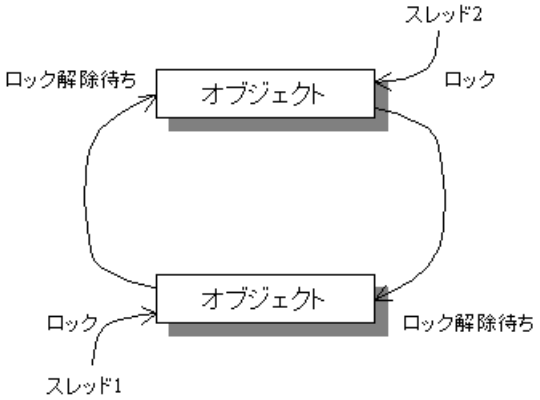


図 10-4：単純なデッドロック

ロックする複数のオブジェクトはお互いに独立していたりループ配置されていたり、入れ子になっていたりするこ

とがあるので、ついこの危険性を見落とすことがあるので、注意が必要である。

この問題は今までの手段では解決できず、また発生するとスレッドが永久に存続し、またオブジェクトがロックされたままとなるので深刻な事態を引き起こす。Java コンパイラや Java 仮想マシンはデッドロックを検出できない。デッドロックの生じないプログラムの作成はプログラマの責任である(検出ツールは存在する)。更には次のような手段も検討されよう。

- 総てのスレッドがその複数のロックを必ず定められた順(例えばロック A→ロック B→ロック C...)で取得するようにする。しかしこれは総てのアプリケーションで適用できないであろう。
- 複数のロック対象オブジェクトをなるべく作らない。例えばサーバにまかせる。「哲学者たちの食事」の場合のサーバとはまさしく給仕である。給仕は各哲学者から食事をとりたいという要求があれば、ふたつのフォークを使ってスパゲッティを取り皿に盛って哲学者に渡す。こうすればデッドロックは発生しない。負荷分散の問題も、給仕がラウンドロビンで各哲学者に聞いて回るなどで解決できる。必要とあらば、仲の良い哲学者に優先度を持たせて、サービスに差をつけることも出来よう。
- 問題が起きてもそれから脱出できるように、危険と思われる個所にはタイムアウトを設定する。絶対に大丈夫と自信をもって開発したプログラムも、稼動させるとしばしばバグが顕在化する。例えば昔筆者が担当していた通信制御のソフトウエアは、殆ど全ての状態にタイムアウトが設定されており、如何なる想定外の事態が生じてもシステムの停止を防止していた。

10.6.2 タイムアウトによる対策

スレッドに関するシステム障害はデッドロックのようにしばしばストールを起こす。そのような場合は時間監視が有効である。

O'Reilly 社から出版されていた Scott Oaks と Henry Wong 共著の「Java スレッド プログラミング」という書物の初期の版には、BusyFlag というロックが使われていた。9 年前にアップロードしたこのチュートリアルの以前の版では、筆者はこの BusyFlag を加工して、監視タイマをこれに組み込んでいた。しかしながら Java 1.5 では新しいロックが導入されており、特に Java.util.concurrent.ReentrantLock は BusyFlag を更に強化したものとなっている。しかもこのクラスには **tryLock(long timeout, TimeUnit unit)というタイマ付きのロック取得のメソッドが用意されている**。指定した時間内にロックが取得できないときは false を返す。従って簡単にタイムアウト監視が可能になっている。

1. ReentrantLock は「Java.util.concurrent.locks パッケージ」の項で示したように、BusyFlag と同様にネスト可能な排他ロックであり、所有者であるスレッドは複数回そのロックを保持できる。これにより再入ロックアウト問題(Reentrance lockout)が解決される。また、tryLock(timeout, unit)メソッドは待機時間指定でこのロックを取得しようとする。この時間内にロックが取得できないときは false が返されるので、これで何らかの異常を検出できる。

タイムアウトはデッドロックを含む障害を検出するだけでなく、デッドロック防止にも使える。例えば次のようなシナリオを考えてみよう:

1. スレッド 1:A をロック
2. スレッド 2:B をロック
   (双方とも何らかの処理を行う)
3. スレッド 1:B をロックしようとするがブロックされる
4. スレッド 2:A をロックしようとするがブロックされる
   (タイムアウトがなければここでデッドロック状態になる)
5. スレッド 1:B のロック取得がタイムアウトになり、B のロック取得作業をやめ、また A のロックも返す
6. スレッド 1:ランダムな時間(例えば 100ms)を置いて再試行する

7. スレッド 2：A のロック取得がタイムアウトになり、A のロック取得作業をやめ、また B のロックも返す
8. スレッド 2：ランダムな時間（例えば 10ms）を置いて再試行する

この例では、スレッド 2 は 90ms 早くロック取得を再開する。したがってスレッド 2 が終わった後でスレッド 1 はその後 2 つのロックの取得ができる。ただしタイムアウトが起きたということは必ずしもデッドロックが起きたということではないことに注意しなければならない。またランダムな待ち時間をどうするかは、アプリケーション（各プロセスの処理時間）と予定スレッド数に依存する。

10.6.3 ウォッチドッグ・タイマ

ロックとは別に、**ウォッチドッグ・タイマ**を用意することもある。これは通信制御やマイクロコントローラなどで良く使われる手法である。特に高い障害耐性が求められるシステムでは、ハードウエアのウォッチドッグ・タイマが使われることもある。アプリケーション側で定期的にこのタイマに信号を送らせ、その信号が所定時間を過ぎても到来しないときは障害が発生したと判断する。

ウォッチドッグ・タイマは低レベルのシステムだけに使われるものではなく、例えばあるファイルをダウンロードする為のあるスレッドが、ネットワーク・ライブラリで障害が起きたときには、プログラマのコントロール外の理由でハングアップしてしまうこともある。これはそのような状態のときに高い信頼性を持ってかつ上品に復旧させるひとつの手段である。

そのような用途のためのウォッチドッグ・タイマ(Watchdog.java)を紹介する。このプログラムは、自分が開発したウェブ・アプリケーションが万一運用中にハングアップが起きたときの対策に組み込むことを想定している。しかしこれはデバッグ用にも使うことが出来るし、コードを少し変更することで、あるプロセスの処理時間を計測するようにもできる。このクラスはひとつのコンストラクタと 3 つのメソッドが用意されている：

表 10-2：Watchdog クラスのメソッド

メソッド	内容
Watchdog(HttpServletRequest req, Thread watchedThread, int timeSec)	HTTP 要求オブジェクト、自分のスレッド、及び秒で指定したタイムアウト時間をもとにこのクラスのインスタンスを生成する。
startWatchdog(String stage)	どこを監視しているかを識別する為の文字列を指定して、この監視タイマのスレッドを開始させる。このメソッドは呼ばれるごとにタイマをリセットして監視を開始する。タイムアウトが発生したら、ロガーにその内容を出力する。
cancelWatchdog()	現在監視中のタイマを停止する。監視スレッドはアイドル状態で留まるので、その後 startWatchdog メソッドを呼んで新たに監視を行うことが出来る。
terminateWatchdog()	監視スレッドを終了させる。現在の監視作業は放棄される。

Watchdog スレッドは下図のような状態遷移を行う：

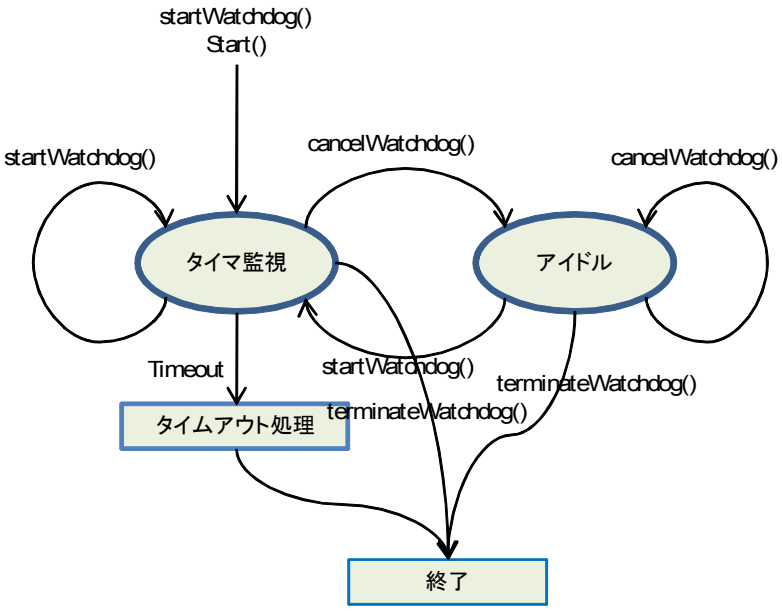


図 10-5：Watchdog スレッドの状態遷移

このインスタンスが生成されたあとで startWatchdog メソッドが呼ばれるとタイマ監視モードになる。このモードでタイムアウトが発生すればタイムアウト処理がなされ、このスレッドは終了する。このモードで terminateWatchdog メソッドが呼ばれると、このスレッドは終了する。このモード中に再度 startWatchdog メソッドが呼ばれると、タイマ監視をやり直す。またこのモード中に cancelWatchdog メソッドが呼ばれると、アイドル（待機）モードとなる。アイドル・モードで startWatchdog メソッドが呼ばれるとタイマ監視モードになる。アイドル・モードで再度 cancelWatchdog メソッドが呼ばれると、アイドル・モードをやり直す。またこのモード中に terminateWatchdog メソッドが呼ばれると、このスレッドは終了する。

以下はこのクラスのコードである：

```
package concurrency;

import java.util.Calendar;

import javax.servlet.http.HttpServletRequest;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Watchdogは、ウェブ・アプリケーションでのハングアップ検出に使うための
 * ソフトウエアによるウォッチドッグ・タイマのスレッドである
 * December 2010, CRESC Corps.
 */

public class Watchdog extends Thread{
   HttpServletRequest req;
   int timeSec;
   Thread watched;
   boolean restartRequested;
   boolean cancelRequested;
   boolean endRequested;
   boolean interruptReady;
   String stage = "";
   private final static Logger LOGGER = LoggerFactory.getLogger(Watchdog.class);
   public Watchdog(HttpServletRequest req, Thread watchedThread, int timeSec){
      this.req = req;
      this.watched = watchedThread;
      this.timeSec = timeSec;
   }

   public void run() {
```

```
      do{
         LOGGER.info(String.format("タイマー開始:    %tc: スレッド: "
               , Calendar.getInstance()) + watched.toString());
         restartRequested =  false;
         cancelRequested = false;
         endRequested = false;
         interruptReady = true;
         try {
            sleep(timeSec * 1000);
         } catch (InterruptedException e) {
            interrupted(); // 割り込みフラグのリセット
            if (cancelRequested) doIdle();   // キャンセルで待機モードへ
         }
      } while (restartRequested || cancelRequested);
      if (!endRequested){     // タイムアウト処理
         LOGGER.info(String.format("タイムアウト発生: %tc: スレッド: "
               , Calendar.getInstance()) + watched.toString());
         LOGGER.info("  ステージ: " + stage);
         LOGGER.info("  getRemoteAddr: " + req.getRemoteAddr());
         //    watched.stop(); // これは勧められない
      }
//    System.out.println("Timer thread terminated!");
   }

   private void doIdle(){
      do{
         LOGGER.info(String.format("待機開始:       %tc: スレッド: "
               , Calendar.getInstance()) + watched.toString());
         restartRequested =  false;
         cancelRequested = false;
         interruptReady = true;
         try {
            sleep(Long.MAX_VALUE);
         } catch (InterruptedException e) {
            interrupted();
         }
      }while (cancelRequested);
   }

   public synchronized void startWatchdog(String stage){
      this.stage = stage;
      if (!isAlive()){
         start();
//       System.out.println("Timer thread started");
      }
      else {
         do {} while (!interruptReady);
         interruptReady = false;
         restartRequested = true;
         interrupt();
      }
   }

   public synchronized void cancelWatchdog(){
      do {} while (!interruptReady);
      interruptReady = false;
      cancelRequested = true;
      interrupt();
   }

   public synchronized void terminateWatchdog(){
      do {} while (!interruptReady);
      interruptReady = false;
      endRequested = true;
      interrupt();
   }
}
```

このコードは、状態遷移図を見れば理解が早いだろう。注意することは、このスレッドを使うサーブレットの HTTP 要求スレッドとこのスレッド間の受け渡しであろう。基本的には HTTP 要求スレッドはウォッチドッグのスレッドに割り込みをかけることでその状態を遷移させている。ウォッチドッグのスレッドが HTTP 要求スレッドからの割り込みを受け付ける状態にあることを interruptReady というフラグで通知して、適正な割り込みをかけるようにしている。

タイムアウト処理は、ここではログ出力しているだけである。タイムアウトが発生したときに、HTTP 要求スレッドを止めたり、HTTP 応答を出すことはできない。HTTP 要求スレッドはサーブレット・コンテナが管理しているものであり、これを止めたりは出来ない。また何らかの応答をクライアントに出したいときは、後の章で説明する非同期操作を使うことになろう。非同期操作では、HTTP 応答を作るのはサーブレット側が用意するスレッドだからである。

このウォッチドッグをサーブレットでどのように使うかを HelloWorld を例に説明する:

```
package concurrency;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/WatchdoggedHelloWorld")
public class WatchdoggedHelloWorld extends HttpServlet
{
   private static final long serialVersionUID = 1L;
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
   {
      Watchdog wd = new Watchdog(req, Thread.currentThread(), 5);
      wd.startWatchdog("Stage 1");
      try {
         Thread.sleep(1 * 1000);
      } catch (InterruptedException exc) {}
//    wd.startWatchdog("Stage 2");
      res.setContentType("text/html; charset=Windows-31J");
      wd.cancelWatchdog();
      PrintWriter out = res.getWriter();
      wd.cancelWatchdog();
      out.println("<HTML>");
      wd.startWatchdog("Stage 3");
      out.println("<HEAD><TITLE>Hello World</TITLE></HEAD>");
      out.println("<BODY>");
      out.println("<BIG>Hello World from 自分の名前</BIG>");
      out.println("</BODY></HTML>");
      try {
         Thread.sleep(8 * 1000);
      } catch (InterruptedException exc) {}
      wd.terminateWatchdog();
   }
}
```

赤の太字で示した部分がウォッチドッグ・タイマを実験する為に幾つかの行を挿入したコードである。使い方は:

1. Watchdog wd = new Watchdog(req, Thread.currentThread(), 5);のように、先ずこのウォッチドッグ・タイマをインスタンス化する。引数は要求オブジェクト、自分のスレッド、及びタイムアウト時間(秒)で、5 というのはタイムアウト時間として 5 秒間を設定したことを示す。
2. 監視したい区間を wd.startWatchdog("Stage 1");及び wd.cancelWatchdog();で指定する。StartWatchdog メソッドの引数としてはその区間を識別する為の文字列をセットする。
3. 最後にこのサーブレットを抜けるときに必ず wd.terminateWatchdog();を実行する。サーブレット・エンジンからの HTTP 要求スレッドは、サーブレットを抜けるときにはそのサーブレットで自分が作ったオブジェクトは削除するが、タイマのスレッドはそのまま待機で残ってしまう可能性があるので、このメソッドを呼ぶ必要がある。その点からも、現在は実質無限大になっているアイドル(待機)モードのタイムアウト時間をもっと短くしておいたほうが無難である。割り込みが無い状態でアイドル・タイマがタイムアウトになると、このスレッドはタイムアウトのログを出力したあとで終了する。
4. wd.terminateWatchdog を実行した後で、startWatchdog あるいは cancelWatchdog メソッドを呼んではいけない。

このサーブレットをブラウザからアクセスすると、次のようなログが出力される:

```
49584 [Thread-3] INFO concurrency.Watchdog - タイマー開始:     金 12 10 16:24:37 JST 2010: スレッド: Thread[http-8080-exec-10,5,main]
50570 [Thread-3] INFO concurrency.Watchdog - 待機開始:         金 12 10 16:24:38 JST 2010: スレッド: Thread[http-8080-exec-10,5,main]
50570 [Thread-3] INFO concurrency.Watchdog - 待機開始:         金 12 10 16:24:38 JST 2010: スレッド: Thread[http-8080-exec-10,5,main]
50570 [Thread-3] INFO concurrency.Watchdog - タイマー開始:     金 12 10 16:24:38 JST 2010: スレッド: Thread[http-8080-exec-10,5,main]
55571 [Thread-3] INFO concurrency.Watchdog - タイムアウト発生: 金 12 10 16:24:43 JST 2010: スレッド: Thread[http-8080-exec-10,5,main]
55571 [Thread-3] INFO concurrency.Watchdog -   ステージ: Stage 3
55571 [Thread-3] INFO concurrency.Watchdog -   getRemoteAddr: 0:0:0:0:0:0:0:1
```

これはタイマが開始したあと1秒後にキャンセルが2回行われ、その後タイマ監視が新たに開始され、その5秒後にタイムアウトが発生し、その区間はステージ3であることを示している。タイムアウトが発生したのはHTTP要求スレッドが8秒間スリープしたからである。これを4秒間にすると、タイムアウトが発生しなことなど、各自試してみられたい。

同じPC上で2つのブラウザをたち上げてともにこのサーブレットをアクセスすると次のようになる:

```
399585 [Thread-11] INFO concurrency.Watchdog - タイマー開始:     金 12 10 16:46:48 JST 2010: スレッド: Thread[http-8080-exec-2,5,main]
400624 [Thread-11] INFO concurrency.Watchdog - 待機開始:         金 12 10 16:46:49 JST 2010: スレッド: Thread[http-8080-exec-2,5,main]
400624 [Thread-11] INFO concurrency.Watchdog - 待機開始:         金 12 10 16:46:49 JST 2010: スレッド: Thread[http-8080-exec-2,5,main]
400624 [Thread-11] INFO concurrency.Watchdog - タイマー開始:     金 12 10 16:46:49 JST 2010: スレッド: Thread[http-8080-exec-2,5,main]
400688 [Thread-12] INFO concurrency.Watchdog - タイマー開始:     金 12 10 16:46:49 JST 2010: スレッド: Thread[http-8080-exec-8,5,main]
401702 [Thread-12] INFO concurrency.Watchdog - 待機開始:         金 12 10 16:46:50 JST 2010: スレッド: Thread[http-8080-exec-8,5,main]
401702 [Thread-12] INFO concurrency.Watchdog - 待機開始:         金 12 10 16:46:50 JST 2010: スレッド: Thread[http-8080-exec-8,5,main]
401702 [Thread-12] INFO concurrency.Watchdog - タイマー開始:     金 12 10 16:46:50 JST 2010: スレッド: Thread[http-8080-exec-8,5,main]
405634 [Thread-11] INFO concurrency.Watchdog - タイムアウト発生: 金 12 10 16:46:54 JST 2010: スレッド: Thread[http-8080-exec-2,5,main]
405634 [Thread-11] INFO concurrency.Watchdog -   ステージ: Stage 3
405634 [Thread-11] INFO concurrency.Watchdog -   getRemoteAddr: 0:0:0:0:0:0:0:1
406711 [Thread-12] INFO concurrency.Watchdog - タイムアウト発生: 金 12 10 16:46:55 JST 2010: スレッド: Thread[http-8080-exec-8,5,main]
406711 [Thread-12] INFO concurrency.Watchdog -   ステージ: Stage 3
406711 [Thread-12] INFO concurrency.Watchdog -   getRemoteAddr: 0:0:0:0:0:0:0:1
```

このように、この方式は、このサーブレット内を走っているHTTP要求スレッドと同じ数のウォッチドッグのスレッドが存在することになり、処理性能に影響を与える可能性があることに注意する必要である。

これはIEやSafariを使った場合だが、**Googleの**Chromeと**Mozillaの**Firefoxでは同じアドレスを同時にアクセスするとそれらの要求は直列化されるので、このような結果が得られないので、注意が必要である。

10.7節　データベースとスレッド安全

読者たちが開発するサーブレットの殆どがデータベースをアクセスする。その場合、スレッド安全性はどうなっているのだろうか。本資料はデータベースのチュートリアルではないので、隔離レベルについて簡単に紹介するに留める。

データベースの世界では、並行処理はトランザクションの並行処理として扱われる。共有資源のアクセスの問題は、トランザクション処理の隔離性(Isolation)として捉えられ、きちんとしたメカニズムが用意されている。これは40年以上前のメインフレームの時代からの歴史がありまた枯れた技術でもある。

トランザクションの隔離性確保のためには、この章で説明してきたスレッド安全性確保の方法に似て、3つのアプローチが考えられる:

1. 直列化(Serialize)
   これは総てのトランザクションをひとつずつ処理するもので、小規模のアプリケーションではタンタンで確実であるが、全体の処理能力と性能が低下する。
2. ロックによる排他制御
   これは対象データをロックして、他のスレッドがアクセスするのを待たせる方式である。これは後で述べる

隔離レベルを設定する、あるいは SQL 文の SELECT ～ FOR UPDATE などを使うことで行われるが、排他制御で説明したように、処理のさせかたによってはデッドロックを起こす可能性がある。また少数のデータを多数のユーザが頻繁に更新する、あるいはロックのスコープが大きいようなアプリケーションではスループットが低下する。

3. バージョニングを使う
   これは、データが更新される度にバージョン番号を割り振ってゆくことで、トランザクション間の干渉を検出する方法である。このような競合が検出されたら処理をやり直すというアプローチは、ロックが不要な為スループットが向上するが、エラー検出によるリトライが必要になるという欠点を持つ。

これらのアプローチをもとに実現される隔離レベル(Isolation Level)は、自分自身が持つ共有資源とのインタラクションについてするとともに、同じ共有資源をもつ他のスレッドとのインタラクションについてネゴシエーションするのに使われる。TRANSACTION_SERIALIZABLE はデータの信頼性のための最高のレベルである。

- TRANSACTION_SERIALIZABLE: このレベルは最高のデータ保全性をもつ。serializable なトランザクションが終了するまでは他のトランザクション・スレッドはこのデータへの読み書きは出来ない。Serializable というのは、直列な操作としてのプロセスをいい、Java でいるオブジェクトの直列化と混同してはいけない。全体の処理能力は一番低くなるので、その点で問題がある場合は別のレベルに設定する。

- TRANSACTION_REPEATABLE_READ: このレベルでは、他のトランザクションは読み出しはできるが変更は出来ない。最初のトランザクションが変更し、その変更した値を書き戻さない限り、読み出したデータは再度読み出しても同じであることが保証される。

- TRANSACTION_READ_COMMITTED: このレベルでは、最初のトランザクションがコミット(commit)またはロールバック(rolls back)するまでは、他のトランザクションはそのデータの読み出しができない。これは一般的な JDBC のデフォルトの隔離レベルである。

- TRANSACTION_READ_UNCOMMITTED: このレベルでは、最初のトランザクションが終了またはロールバックする前であっても他のスレッドがそのデータを読み出すことができる。

金額に絡むトランザクションは TRANSACTION_SERIALIZABLE のレベルを指定すべきだが、統計やマーケッティングなど細かな整合性を問わないアプリケーションではその他のレベルを採用して性能をあげる。

データベースは指定された隔離レベルに基づき、複数のトランザクションを受け付ける。従ってサーブレット開発者は、データベースアクセスのスレッド安全性に関しては、隔離レベルをきちんと把握するだけでよかろう。データベースに関しては接続プールのテーマもあるが、ここでは省略する。

10.8節　この章のまとめ

以下は、この章の内容をもとにした、サーブレットにおけるスレッド安全の幾つかのポイントである：

1. インスタンス変数やクラス変数は使ってはいけない。
2. サーブレットそのものを synchronized を使って同期化してはいけない。
3. コンテキスト属性へのアクセスには同期化が必要である。セッションの属性へのアクセスも場合によっては同期化が必要になる。要求の属性へのアクセスは同期化の必要は無い。
4. API によってはスレッド安全でないクラスが存在するので、そのようなクラスは同期メソッド、同期ブロック、

あるいは ThreadLocal を使って対応する。

5. SingleThreadModel はインスタンス変数の問題しか対処できないので、よほど小規模なもの以外は使ってはいけない。
6. ブラウザやその使い方によっては同じセッションを持ったスレッドがそのサーブレット内で並行する可能性があるので注意しなければならない。
7. どのリソースが共有されるのかを先ず特定し、それに対するスレッド安全化には、そのスコープに応じて同期メソッド、同期ブロック、あるいは java.util.concurrent が持っている各種ロックを使うかを判断する。
8. 共有されるリソースは、セッションの章の「MessageManager クラス」の項で示した MessageManager.java のようなシングルトンのマネージャでラップする。Enterprise JavaBeans (EJB)は、DB とのやりとりを含む共通化されたビジネス・ロジックのコンポネントで、スレッド安全に関してはきちんとした対応がとられているものが多い。
9. 複数のロックが存在する場合は、ウォッチドッグ・タイマによる検出を検討する。

第11章　　サーブレット・コンテキスト

サーブレット・コンテキストとはひとつのアプリケーションに1対1で対応したものである。ServletContextはウェブ・サーバ内の既知の要求パスとしてルートづけられる。例えば、あるサーブレット・コンテキストはhttp://www.mycorp.com/catalogに位置しているとする。/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。

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。ServletContextオブジェクトは、サーブレットが自分のサーブレット・コンテナと通信するために使うメソッドたちのセットが用意されている。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、あるファイルのMIMEタイプを取得、要求のディスパッチ、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、などができるようになる。

JVMあたり「ウェブ・アプリケーション」あたりひとつのコンテキストが存在する。(「ウェブ・アプリケーション」とは、/catalogのようなそのサーバのURL名前空間の特定のサブセット、のもとでインストールされるサーブレットたちとコンテントの集まりであって、.warファイルによってインストールされ得る。)

その配備記述子で"distributed"とマークされたウェブ・アプリケーションの場合は、各VMごとにひとつのコンテキストのインスタンスが存在することになる。この状況のときは、そのコンテキストはグローバルな情報を共有する場所としては使えない。(なぜなら、その情報は真にグローバルな物とならないからである。)この場合は代わりにデータベースのような外部リソースを使用する。

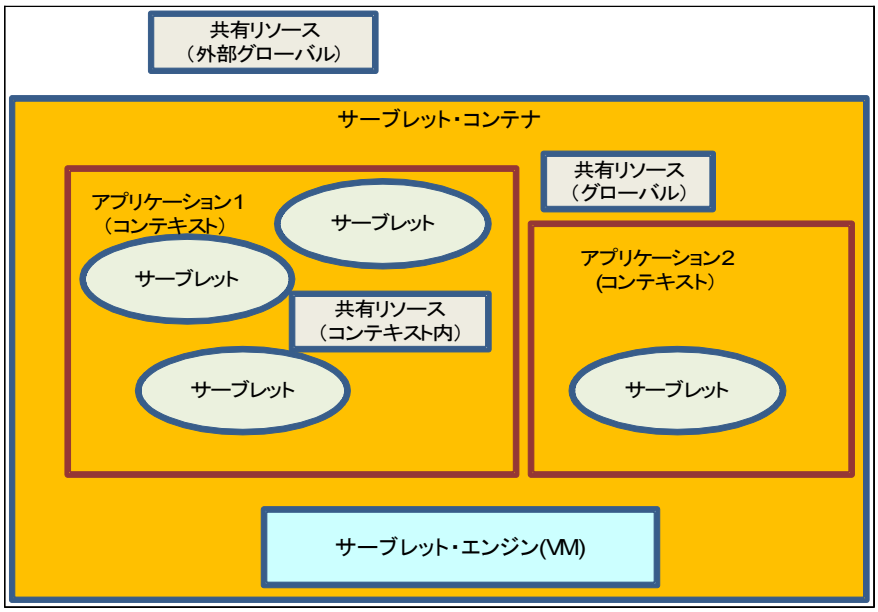


図 11-1:コンテキストのイメージ

上図はサーブレット・コンテナ内のアプリケーションのイメージを示したものである。ウェブ・アプリケーションは以下の要素で構成される。

- サーブレットたち
- フィルタたち
- リスナたち
- JSP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、class たち
- それらを結び付ける記述的メタ情報

共有リソースにはやはりサーブレット、JSP ページ、静的ドキュメント、データベース、ユーティリティ・クラスなどが含まれ、アプリケーション内で共有される場合もあれば、複数のアプリケーションで共有されることもあろう。

なお、ServletContext オブジェクトは ServletConfig オブジェクトの中に含まれており、この ServletConfig オブジェクトはそのサーブレットが初期化されるときにウェブ・サーバによってそのサーブレットに提供される。このコンテキストを取得するには、getServletContext メソッドを使用する。このコンテキストにより、以下のものたちへのアクセスが可能となる:

- 初期化パラメタ
- そのウェブ・コンテキストに関わるリソース
- オブジェクトとして貼り付けられている属性
- ロギング機能

ServletContext インターフェイスは、サーブレット 3.0 仕様書の 4.4 章ほかで記されているように、サーブレット 3.0 で以下の事項が追加されている:

- 動的(ダイナミック)なサーブレットの追加と設定
- 動的なフィルタの追加と設定
- 動的なリスナの追加と設定
- セキュリティ・ロールの動的な設定
- セッション追跡モードの動的な設定
- クラス・ローダの取得

なおサーブレット 3.0 からは、非同期処理のための非同期コンテキスト(AsyncContext)が導入されている。非同期処理と非同期コンテキストに関しては、別の章で説明する。

11.1節　ServletConfig、ServletContext、及び Servlet 間の関係

最初にサーブレット API における ServletConfig、ServletContext、及び Servlet の関係をクラス図をもとに説明する。

Servlet インターフェイスは、各サーブレットが実装しなければならないメインのインターフェイスである。ここではサーブレット・コンテナがそのサーブレットをコントロールする為に呼び出す主要メソッドが定義されている。「サーブレットのライフサイクル」の節で説明したように、サーブレット・コンテナはそのサーブレットをサービス状態に置くのに先ずそのサーブレットをインスタンス化し、その init()メソッドを呼ぶ。サーブレットは SingleThreadModel を実装していない限り、そのライフサイクル中に一度だけインスタンス化される。一旦そのサーブレットがインスタンス化されロードされたら、サーブレット・コンテナはそのサーブレットに対応した HTTP 要求を要求と応答のオブジェクトのペアを引数としてそのサーブレットの service()メソッドを呼ぶ。これは最終的に HttpServlet の de*HttpRequestName*()メソッドに渡される。各要求は各々がスレッドとして処理される。サービスが不要になったら、サーブレット・コンテナは先ずそのサーブレットの destroy()メソッドを呼び、その後このオブジェクトを削除する。

GenericServlet は抽象クラスであって、サーブレット実装の開始点として使われる。特にこのクラスは Servlet イン

ターフェイスの総てのメソッドたちを実装している。この抽象クラスはまた ServletConfig インターフェイスを実装しており、これによりサーブレット・コンテナがサーブレットに情報を渡すことが出来る。GenericServlet では、super.init()を呼ばなくても良い init()、ログ出力の為の log()が定義されている。

HttpServlet はこれを継承したもので、service()をさらに HTTP のメソッドにあわせて展開したメソッドたちが用意されている。開発者はこれらのメソッド、特に doGet()や doPost()をオーバライドすることになる。

ServletConfig インターフェイスはサーブレット・コンテナが初期化中にサーブレットに情報を渡すのに使うサーブレットの設定オブジェクトである。これによりサーブレット・コンテナがサーブレットに情報を渡すことが出来る。

ServletContext インターフェイス前述のように、サーブレットが自分のサーブレット・コンテナと通信するために使うメソッドたちのセットが用意されている。ServletContext は ServletConfig に含まれており、このオブジェクトは ServletConfig.getServletContext()メソッドを使って取得することになる。実際は HttpServlet クラスは ServletConfig インターフェイスを実装しているので、サーブレットの中から単に getServletContext()を呼べばよい。

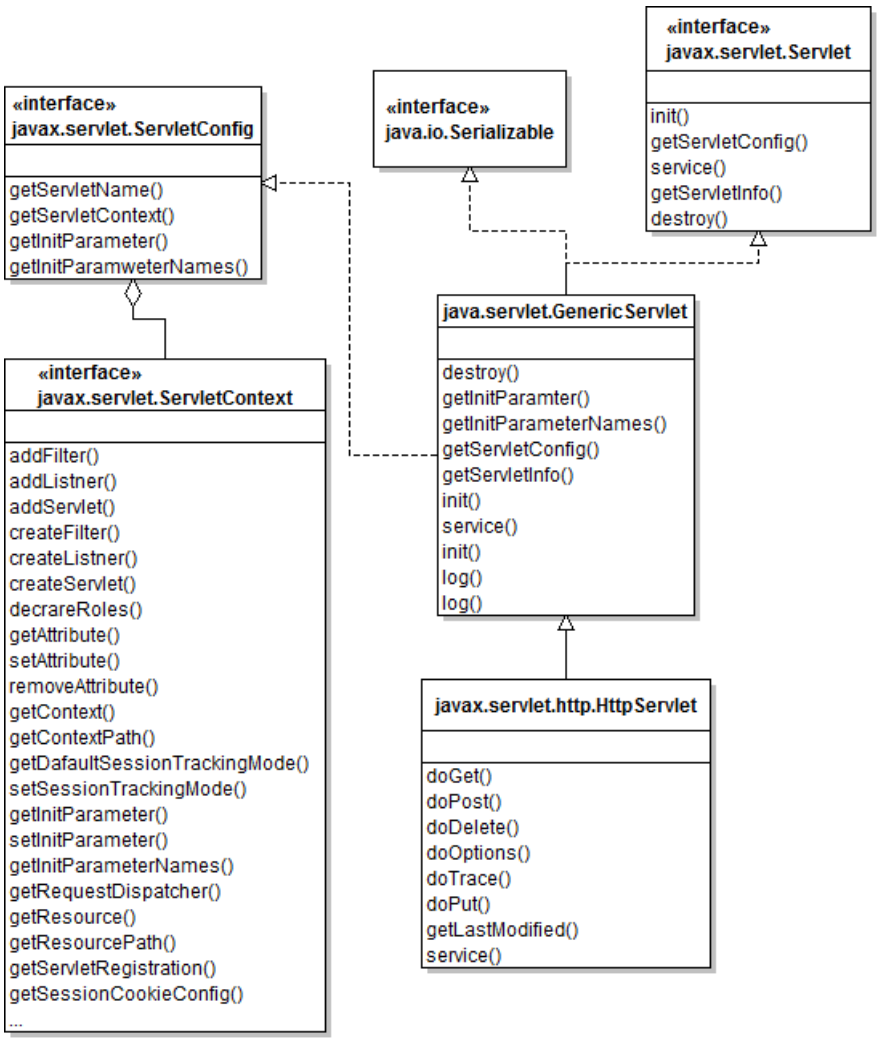


図 11-2：ServletConfig、ServletContext、及び Servlet の関係

11.2節　ServletContext のメソッドたち

ServletContext インターフェイスの API ドキュメントでは、あるファイルの MIME タイプを取得する、要求をディスパッチする、あるいはログ・ファイルに書き込むといった、サーブレットが自分のサーブレット・コンテナと通信するために使うメソッドたちのセットを定めている。この節ではこれらのメソッドを用途別に区分して紹介している。その詳細は添付の「サーブレット3.0 の主要クラスのメソッド一覧」の節を見ていただきたい。

JVM あたり「ウェブ・アプリケーション」あたりひとつのコンテキストが存在する。(「ウェブ・アプリケーション」とは、/catalog のようなそのサーバの URL 名前空間の特定のサブセット、のもとでインストールされるサーブレットたちとコンテントの集まりであって、.war ファイルによってインストールされ得る。)

その配備記述子で"distributed"とマークされたウェブ・アプリケーションの場合は、各 VM ごとにひとつのコンテキストのインスタンスが存在することになる。この状況のときは、そのコンテキストはグローバルな情報を共有する場所としては使えない。(なぜなら、その情報は真にグローバルな物とならないからである。)この場合は代わりにデータベースのような外部リソースを使用する。

ServletContext オブジェクトは ServletConfig オブジェクトの中に含まれており、この ServletConfig オブジェクトはそのサーブレットが初期化されるときにウェブ・サーバによってそのサーブレットに提供される。

11.2.1　初期化パラメタに関するメソッドたち

初期化パラメタはサーブレットをサービス稼動状態に置く際に設定されるそのサービスに必要なパラメタたちである。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。これらのパラメタは通常はアノテーション、配備記述子(web.xml あるいは web-fragment.xml)、あるいは後述のプログラム的な設定で設定される。初期化パラメタは String 型で(名前、値)のペアで構成される。通常はサーブレットの初期化は init()メソッドをオーバライドして、そのメソッドの中で行う。init メソッドに関しては「サーブレットの初期化」の節を参照のこと。

表 11-1:初期化パラメタ関係のメソッドたち

メソッド	内容
getInitParameter	名前がつけられたコンテキストにわたる初期化パラメタの値を含む String を返す
getInitParameterNames	このコンテキストの初期化パラメタたちの名前を String オブジェクトたちの Enumeration として返す
setInitParameter	この ServletContext 上で指定した値と名前を持った初期化パラメタをセットする。このメソッドに関しては、後述の「プログラム的な設定」を参照のこと
getJspConfigDescriptor	この ServletContext によって代表されるウェブ・アプリケーションの配備記述子から jsp-config 要素の設定を取得する

11.2.2　データ共有の為のメソッドたち

これは「情報の共有」の節を参照されたい。ServletContext もスコープ・コンテナであって、そのアプリケーションに渡って共有されるデータを属性として保持させることが出来る。使い方は HttpRequest、あるいは HttpSession で学習したと同じである。

表 11-2:データ共有の為のメソッドたち

メソッド	内容
getAttribute	この ServletContext の与えられた名前の属性の値(オブジェクト)を返す
getAttributeNames	この ServletContext の属性の名前の総てを Enumeration として返す
setAttribute	この ServletContext に与えられた名前の与えられたオブジェクトを属性としてセットする。
removeAttribute	この ServletContext から与えられた名前の属性を除去する

これらの属性はこのアプリケーション内のサーブレットやオブジェクトたちで共有される。その変化は ServletContextAttributeListener を実装したオブジェクトに通知される。

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、コンテキスト属性は使えず**、それらの情報はセッションに置かれる、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。

11.2.3 プログラム的なコンテキストへの追加と設定

Servlet 3.0 からは Struts などのフレームワークとの親和性を高めるために、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、それらをマップするための URL パタン、またセッション追跡モードなどをプログラム的(動的)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。**例えば「setSessionTrackingModes による方法」の項で示した ServletContextInitializer というリスナがその良い事例である。もう一度このクラスを示すと:

```
@WebListener
public class ServletContextInitializer implements ServletContextListener{

   @Override
   public void contextDestroyed(ServletContextEvent arg0) {
   }

   @Override
   public void contextInitialized(ServletContextEvent servletContextEvent) {
      ServletContext cxt = servletContextEvent.getServletContext();
      Set<SessionTrackingMode> stms = cxt.getDefaultSessionTrackingModes();
      System.out.print("DefaultSessionTrackingMode : ");
      for (SessionTrackingMode c : stms)
         System.out.print(c + ", ");
      System.out.println();
      stms = new HashSet<SessionTrackingMode>();
      stms.add(SessionTrackingMode.URL);
      stms.add(SessionTrackingMode.COOKIE);
      cxt.setSessionTrackingModes(stms);
      System.out.print("EffectiveSessionTrackingMode : ");
      for (SessionTrackingMode c : stms)
         System.out.print(c + ", ");
      System.out.println();
   }
}
```

このクラスは contextInitialized メソッドをオーバライドして、セッション追跡モードを設定している。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。詳細はサーブレット 3.0 仕様書の 4.4 節を参照されたい。

11.2.3.1 サーブレットのプログラム的な追加と設定

これはフレームワーク親和性のために用意されている。例えば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。

表 11-3：サーブレットのプログラム的な追加と設定

メソッド	内容
AddServlet (3 つ存在する)	これらのメソッドにより、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。これら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。
createServlet	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。
getServletRegistration	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
getServletRegistrations	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

11.2.3.2 フィルタのプログラム的な追加と設定

表 11-4：フィルタのプログラム的な追加と設定

メソッド	内容
AddFilter (3 つ存在する)	これらのメソッドにより、そのアプリケーションはプログラム的にあるフィルタを宣言できる。これら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フィルタをそのサーブレット・コンテキストに追加する。
createFilter	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。
getFilterRegistration	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、またはその名前の FilterRegistration が無い場合は null を返す
getFilterRegistrations	その ServletContext で登録された総てのフィルタに対応した名前をキーとした FilterRegistration のオブジェクトのマップを返す

11.2.3.3 リスナのプログラム的な追加と設定

表 11-5：リスナのプログラム的な追加と設定

メソッド	内容
AddListner (3 つ存在する)	これらのメソッドにより、そのアプリケーションはプログラム的にあるリスナを宣言できる。これら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リスナをそのサーブレット・コンテキストに追加する。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。

メソッド	内容
	• javax.servlet.ServletContextAttributeListener • javax.servlet.ServletRequestListener • javax.servlet.ServletRequestAttributeListener • javax.servlet.http.HttpSessionListener • javax.servlet.http.HttpSessionAttributeListener • javax.servlet.http.HttpSessionAttributeListener
createListner	このメソッドは与えられた Listner クラスをインスタンス化する。

11.2.3.4 セッション追跡モードの追加と設定

これらの使い方は「setSessionTrackingModes による方法」の項で示した ServletContextInitializer を参照されたい。

表 11-6：セッション追跡モードの設定

メソッド	内容
getDefaultSessionTrackingModes	この ServletContext がデフォルトで対応しているセッション追跡モードたちを返す
getEffectiveSessionTrackingModes	この ServletContext が現在実効的に対応しているセッション追跡モードたちを返す
getSessionCookieConfig	この ServletContext がセッション用に設定する set-cookie 応答ヘッダの属性を取得する
setSessionTrackingModes	この ServletContext で有効になるセッション追跡モードたちをセットする

11.2.3.5 ロールのプログラムによる宣言

表 11-7：ロールの宣言

メソッド	内容
declareRoles	isUserInRole でテストされるロール名たちを宣言する。ServletRegistration インターフェイスの setServletSecurity あるいは setRunAsRole メソッドでそれらを使った結果として明示的に宣言されるロールたちは、宣言される必要が無くなる。

11.2.4 コンテキスト・パス取得

「HTTP URL の例」の図では、HTTP の URL の各要素を示したが、これとサーブレットのコンテキストとの関係を以下に示す。(「要求からの情報の取得」の項の URL の構成の記述も参考のこと)

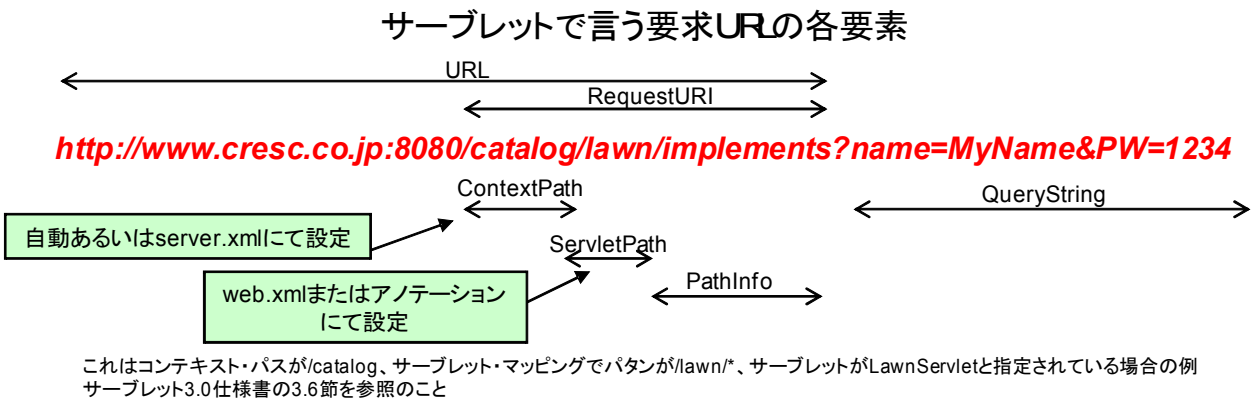


図 11-3：要求 URI の各要素

- 要求 URI：HTTP 要求行のプロトコル名とクエリ文字列（あるいはプロトコル・バージョン）の間をいう。例えば：
  - POST /some/path.html HTTP/1.1 >> /some/path.html
  - GET http://foo.bar/a.html HTTP/1.0 >> /a.html
  - HEAD /xyz?a=b HTTP/1.1 >> /xyz

  要求 URI は HttpRequest.getRequestURI メソッドで取得できる。
- コンテキスト・パス：コンテキスト・パスは要求 URI の一部であってその要求のコンテキストを選択するのに使われている。このコンテキスト・パスは常に要求 URI のなかの最初に位置する。パスは'/'文字で始まるが'/'文字では終わらない

  コンテキスト・パスは ServletContext.getContextPath メソッドを使って取得することになる。

表 11-8：コンテキスト・パス取得メソッド

メソッド	内容
getContextPath	そのウェブ・アプリケーションのコンテキスト・パスを返す。

11.2.5 他のアプリケーションやリソースのアクセス関連メソッドたち

「他のウェブ・リソースの呼び出し」の節を参照されたい。

表 11-9：リソース・アクセス関連メソッドたち

メソッド	内容
getClassLoader	この ServletContext で代表されるウェブ・アプリケーションのクラス・ローダを取得する
getContext	指定した URL に対応した ServletContext オブジェクトを返す。このメソッドにより、サーブレットたちはそのサーバのいろんな要素の為のコンテキストにアクセスできるようになり、必要ならそのコンテキストから RequestDispatcher オブジェクトを取得できる。与えられるパスは'/'で始まり、それはそのサーバのドキュメント・ルートに対し相対だと解釈され、このコンテナ上でホストされている他のウェブ・アプリケーションのコンテキスト・ルートに対し合致しているものである。
getMimeType	指定したファイルの MIME タイプを返す。これは配備記述子で登録されているものからになる。
getRealPath	指定したパスのサーバ上の実際のパスを返す

メソッド	内容
getResource	指定したパスにマップされているリソースに対する URL を返す。指定したパスは現在のコンテキスト・パスにたいし相対的なもの、あるいはそのウェブ・アプリケーションの WEB-INF/lib ディレクトリの中にある JAR ファイルの/META-INF/resources ディレクトリに対し相対的、とする。
getResourceAsStream	指定したパスにマップされているリソースに対する
getResourcePaths	そのウェブ・アプリケーション内のリソースに対する総てのパスをディレクトリ的なリストとして返す

11.2.6 要求ディスパッチャ関連メソッドたち

要求ディスパッチャを使った HTTP 要求のディスパッチの使い方は、既に「セッションのイベント処理」の項で示したメッセージ受付サービスとメッセージ問合せサービスのサーブレット間で示されている。お互いに HTTP 要求を相手に依頼(ディスパッチ)するときは、先ずディスパッチャをコンテキストから取得して。そのディスパッチャの forward メソッドを呼び出していた。必要な情報を渡すときには、HttpRequest オブジェクトにその情報を属性として付加していた。

次の節でより詳細を説明する。

表 11-10:要求ディスパッチャ取得メソッドたち

メソッド	内容
getRequestDispatcher	与えられたパスに位置するリソースの為のラッパとして機能する RequestDispatcher を返す。RequestDispatcher オブジェクトは、その要求をそのリソースにフォワードする、あるいはある応答にそのリソースを含める為に使える。そのリソースは静的なものあるいは動的なものであり得る。pathname は'/'で始まらねばならず、現在のコンテキスト・ルートに対し相対的なものであると解釈される。他のコンテキスト内のリソースに対する RequestDispatcher を取得するには、getContext を使用のこと。
getNamedDispatcher	名前が指定されたサーブレットのラッパーとして機能する RequestDispatcher オブジェクトを返す。サーブレットたち(JSP ページたちも含まれる)はサーバの管理を介して、あるいはウェブ・アプリケーション配備記述子を介して名前が付られ得る。サーブレット・インスタンスは ServletConfig.getServletName()を使ってその名前を判断できる。

注意しなければならないのは **ServletRequest(あるいはそのサブクラスの) HttpServletRequest にも getRequestDispatcher と同じ名前のメソッドが存在する**ことである。その相違は、ServletRequest#getRequestDispatcher のほうはより強化されていて、相対パスと絶対パスが使えることである。相対的(即ち'/'で始まる)な使い方では引数として渡すパス名は現在のサーブレット・コンテキストの外部には拡張できない。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")呼び出しとまったく同じふるまいをする。

11.2.7 その他のメソッドたち

表 11-11:その他のメソッドたち

メソッド	内容
getMajorVersion	このサーブレット・コンテナが対応している API のメジャー・バージョンを返す
getEffectiveMajorVersion	このアプリケーションがベースにしている API のメジャー・バージョンを返す
getMinorVersion	このサーブレット・コンテナが対応している API のマイナ・バージョンを返す
getEffectiveMinorVersion	このアプリケーションがベースにしている API のマイナ・バージョンを返す
getServerInfo	このサーブレットが走っているサーブレット・コンテナの名前とバージョンを返す
getServletContextName	配備記述子の display-name 要素で設定されたこのコンテキストに対応したウェブ・アプリケーション名を返す
log（2 つあり）	指定したメッセージをサーブレットのログ・ファイルに書き込む

11.3節　要求のフォワードとインクルード

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（フォワード:forward）、あるいは別のサーブレットの出力を応答に含める（インクルード:include）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、非同期コンテキスト(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。非同期処理と非同期コンテキストに関しては、別の章で説明する。

要求ディスパッチャについては、「要求ディスパッチャ関連メソッドたち」の項でも既に説明されている。getRequestDispatcher メソッドは、指定されたパスに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（JSP も含む）が見つけられないときは、その RequestDispatcher はそのパス上の静的なコンテントを返すものになる。

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たち、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、のどちらかになる。後者の場合は、ラッパ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。

11.3.1　include メソッド

このメソッドは、簡単にいえば HTTP 応答メッセージのボディ部の為に ServletOutputStream または Writer にデータを書き込む（あるいはそれをフラッシュするまで）作業をターゲットのサーブレットの依頼する。従って、ターゲットのサーブレットは：

- HTTP 応答メッセージのヘッダ（クッキーも含まれる）をセットしてはならない
- 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 をスローされる

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。

このメソッドでは、以下の要求属性たちがセットされる:

- 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。

以下にその例を示そう:
例えば/tastapp というアプリケーションの/testapp/index.jsp に次のようにフォワード文が書かれていたとする:

```
/testapp/index.jsp  <jsp:include page="target.jsp"/>
```

これは

```
ServletContext.getRequestDispatcher("target.jsp").include();
```

と等価であるが、これに対して呼ばれた側でのパス関連情報は次のようになる:

表 11-12:インクルード後のパス関連情報

インクルード後の target.jsp 内でのパス関連メソッド	
request.getRequestURI()	/testapp/index.jsp
request.getContextPath()	/testapp
request.getServletPath()	/index.jsp
request.getPathInfo()	null
request.getQueryString()	null
target.jsp で得られる要求の属性	
request.getAttribute("javax.servlet.include.request_uri")	/testapp/target.jsp
request.getAttribute("javax.servlet.include.context_path")	/testapp
request.getAttribute("javax.servlet.include.servlet_path")	/target.jsp
request.getAttribute("javax.servlet.include.path_info")	null
request.getAttribute("javax.servlet.include.query_string")	null

11.3.2 forward メソッド

このメソッドでは、以下の要求属性たちがセットされる:

- 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

例えば/testapp というアプリケーションの index.jsp に対する/testapp/index.jsp?foo=bar という HTTP 要求に対し、
<jsp:forward page="target.jsp"/>
次のようにフォワード文が書かれていたとする:

```
<jsp:forward page="target.jsp"/>
```

これは

```
ServletContext.getRequestDispatcher("target.jsp").forward();
```

と等価であるが、これに対して呼ばれた側でのパス関連情報は次のようになる:

表 11-13:フォワード後のパス関連情報

フォワード後の target.jsp 内でのパス関連メソッド	
request.getRequestURI()	/testapp/index.jsp
request.getContextPath()	/testapp
request.getServletPath()	/target.jsp
request.getPathInfo()	null
request.getQueryString()	null
target.jsp で得られる要求の属性	
request.getAttribute("javax.servlet.forward.request_uri")	/testapp/index.jsp
request.getAttribute("javax.servlet.forward.context_path")	/testapp
request.getAttribute("javax.servlet.forward.servlet_path")	/index.jsp
request.getAttribute("javax.servlet.forward.path_info")	null
request.getAttribute("javax.servlet.forward.query_string")	null

11.3.3 クエリ文字列の追加

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、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);
```

あるいは JSP では次のようなコードになる:

```
<jsp:include page="target.jsp?foo=override"/>
```

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。

11.3.4 サーブレット間通信

このように forward と include のメソッドはサーブレット間のデータ交換の手段でもある。forward あるいは include を呼ぶ側のサーブレットは:

- セッションの属性としてオブジェクトをバインドする
- 要求の属性としてオブジェクトをバインドする
- クエリ文字列としてデータを添付する

ことで、ターゲットのサーブレットはこのサーブレットの結果を使用できる。

11.3.5 sendRedirect との相違

HTTP 要求を別のサーブレットに転送するだけなら、「リダイレクション」の節で述べたとおり、HttpServletResponse#sendRedirect メソッドも使用可能である。その相違を以下に説明する。

sendRedirect(String path)メソッドは HTTP 応答ヘッダを変更し、また新しい場所を指定する location ヘッダ行を追加して、ブラウザにそちらのサーブレットをアクセスするように告げる。それによりその要求はターゲットのサーブレットに転送される。しかしながらこの場合は:

- HTTP 応答と要求の転送時間が追加される
- 要求パラメタが総て失われる(クエリ文字列は追加できる)
- クエリ文字列以外は、属性といかたちでサーブレット間で情報を渡すことが出来ない

などの問題があるが、ターゲットはそのサーバ上でなくてもどこにあっても良いという利点もある。

sendRedirect は既に HTTP 応答がコミットされている、即ち既にボディ部にデータが書き込まれているときは使えないことに注意しなければならない。

11.4節　オブジェクト共有

*ServletContext もスコープ・コンテナであり、そのコンテキストにある総てのリソースが ServletContext のオブジェクトにバインドされたオブジェクトを共有できる。これらの共有オブジェクトに対するマルチ・スレッド対策は、既に「スレッド安全」の章で説明してある。なお、**コンテキスト属性は、分散環境では利用できない**ことに注意しなければならない。*

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

これらのオブジェクト属性の有効域(スコープ)はそのアプリケーション内である。

ServletRequest の属性や HttpSession の属性では、属性としてストアされるオブジェクトは、serializable インターフェイスを実装して、負荷バランスサーバ環境のなかで異なったマシン上の JVM たちのもとで走るサーブレットたちが共有できるようにしなければならない。一方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、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。どうしてこうなっているかは、次の節で示す ServletContext 属性のリストを見れば理解されよう。

11.5節　簡単なサーブレットによる確認

ここでは ForwardTest 及び ForwardTarget というサーブレット、及び ForwardTestForm.html という HTML テキスト・ファイルを使って、ServletContext 及び RequestDispatcher の使い方を確認することにする。

下図はそれらの実験材料の関係を示したものである:

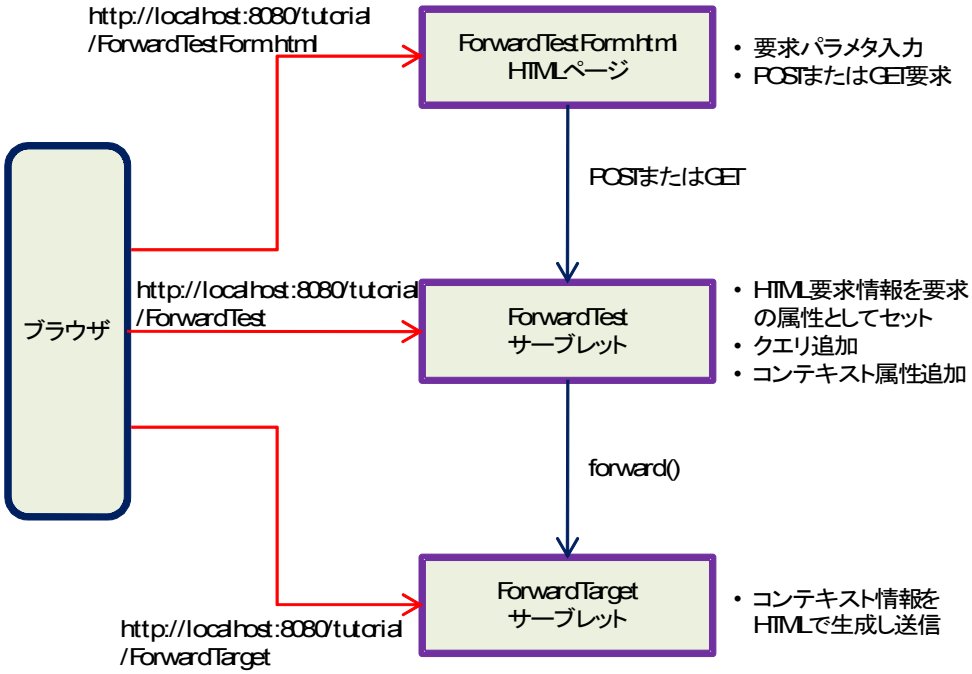


図 11-4:コンテキストとディスパッチャの実験

ForwardTarget サーブレットは、他のサーブレットからフォワードされた HTTP 要求を処理し、コンテキストとディスパッチャ関係の情報をブラウザに返す。このサーブレットを直接ブラウザからアクセスすることもできる。

ForwardTest サーブレットは、ブラウザからの HTTP 要求を ForwardTarget サーブレットにディスパッチする役割を持つ。このサーブレットもブラウザから直接アクセスすることもできる。このサーブレットは HTTP 要求をディスパッチする前に、自分が得た HTTP 要求関係の情報を HTML テキストにして、それを要求オブジェクトの属性として付加している。従って、ForwardTarget サーブレットはその属性を自分が作る HTTP 応答に含めることが出来る。またこのサーブレットは、要求ディスパッチャ取得のひとつの特徴であるクエリ文字列の付加のテストが出来るし、コンテキスト属性も付加して、それがどのようにターゲットのサーブレットで見えるかを確認できるようにしている。

ForwardTestForm.html は、ForwardTest サーブレットを呼び出すための静的なファイルで、以前要求オブジェクトで使った RequestTestForm1.html を一部変更しただけのものであり、GET と POST の双方のメソッドで ForwardTest サーブレットを呼ぶことが出来る。これを使うと、HTML の Form データと、ForwardTest サーブレットで付加したクエリ文字列のデータが、どのようにターゲットのサーブレットで見えるのかを調べることが出来る。

11.5.1 ForwardTest サーブレット

以下はこのサーブレットのコードである:

```
package context;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;
```

```
import java.util.Enumeration;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


@WebServlet("/ForwardTest")
public class ForwardTest extends HttpServlet {

   /**
    * ForwardTest は、RequestDispatcher の学習用の基礎的なサーブレット
    * @author Cresc Corps.
    * @December 2010, CRESC Corps.
    */

   private static final long serialVersionUID = 1L;

   /**
    * 処理本体    *
    */

   public void performTask(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {
      StringBuffer sb = new StringBuffer();
      ServletContext ctxt = getServletContext();
      sb.append("<h1>ディスパッチするサーブレットからの報告</h1>");
      sb.append("<pre><br>パス関係情報(Path Related Methods) : ");
      sb.append("<br>getRequestURI: " + req.getRequestURI());
      sb.append("<br>getContextPath: " + req.getContextPath());
      sb.append("<br>getServletPath: " + req.getServletPath());
      sb.append("<br>getPathTranslated: " + req.getPathTranslated());
      sb.append("<br>getRealPath: " + ctxt.getRealPath(req.getRequestURI()));
      sb.append("<br>getPathInfo: " + req.getPathInfo());
      sb.append("<br>getQueryString: " + req.getQueryString());
      sb.append("<br><br>要求パラメタ(Request Parameters):");
      Enumeration<?> paramNames = req.getParameterNames();
      while (paramNames.hasMoreElements()) {
         String name = (String) paramNames.nextElement();
         String[] values = req.getParameterValues(name);
         try{
            sb.append("<br>  "+new String(name.getBytes("8859_1"), "Windows-31J")+":");
         }catch(UnsupportedEncodingException theException){
            System.out.println("UnsupportedException の例外が発生");
         }
         for (int i = 0; i < values.length; i++) {
            try{
               sb.append("<br>  "+i+") "+new String(values[i].getBytes("8859_1"), "Windows-31J"));
            }catch(UnsupportedEncodingException theException){
               System.out.println("UnsupportedException の例外が発生");
            }
         }
      }
      sb.append("<br></pre>");
      req.setAttribute("sourceSrevletData", sb);
      // コンテキスト属性追加実験
      ctxt.setAttribute("コンテキスト属性名", "コンテキスト属性値");
      // クエリ追加実験
      String query = "?" + URLEncoder.encode("内閣総理大臣", "Windows-31J")
      + "=" + URLEncoder.encode("菅直人", "Windows-31J");
      ctxt.getRequestDispatcher("/ForwardTarget" + query).forward(req, res);
   }

   /**
    * 到来 HTTP GET 要求の処理
    */
   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res);
```

```
    }

    /**
     * 到来HTTP POST要求の処理
     */
    @Override
    public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws javax.servlet.ServletException, java.io.IOException {
       performTask(req, res);
    }

    /**
     * 本サーブレット情報の文字列を返す
     */
    @Override
    public String getServletInfo() {
       return "Forward Test Source Servlet, Version 1.0 by Cresc";
    }
}
```

このコードから判るように、このサーブレットは、StringBuffer を使ってブラウザに返す HTML テキストの一部を作成していて、応答オブジェクトから得られる PrintWriter は使っていない。これにより、ターゲットのサーブレットは HTTP 応答メッセージのヘッダ部から設定できる自由度を持つ。

このサーブレットが作るデータは、すでに学習した HttpRequestDump サーブレットが出している HTML データのうちパス関連のメソッドの結果と、要求パラメタの表示の部分である。

このサーブレットでは、

```
ctxt.setAttribute("コンテキスト属性名", "コンテキスト属性値");
```

と、コンテキスト属性として、名前が"コンテキスト属性名"で、値が"コンテキスト属性値"という String オブジェクトを付加している。この属性がターゲットのサーブレットできちんと読めることを確認することにする。

コンテキスト取得では、

```
ctxt.getRequestDispatcher("/ForwardTarget" + query)
```

という具合に、パスを示す文字列にクエリ文字列を付加することが出来る。クエリ文字列を付加することで、ターゲットのサーブレットに新たな要求パラメタを渡すことが出来る。これは「サーブレット間通信」の項で説明してある。ここでは、

```
      String query = "?" + URLEncoder.encode("内閣総理大臣", "Windows-31J")
      + "=" + URLEncoder.encode("菅直人", "Windows-31J");
```

という具合に、名前が"内閣総理大臣"、値が"菅直人"という要求パラメタを新たにこのサーブレットが付加している。問題は**クエリ文字列の要素は URL エンコードしなければならない**ことである。何故なら、このパス名とクエリ文字列の連結はそのままターゲットのサーブレット呼び出しに使われているからである。日本語だけでなくてデフォルトの ISO-8859-1 であっても、URL として危険な文字は URL エンコードしなければならない。

11.5.2 ForwardTarget サーブレット

以下はこのサーブレットのコードである。

このサーブレットは、応答する HTML テキストに、

```
      // フォワード元が作った HTML テキストを追加
      StringBuffer ssd = (StringBuffer)req.getAttribute("sourceSrevletData");
      if (ssd != null) out.println(ssd);
```

と、フォワード元が生成した HTML テキストが存在すればそれを付加している。

そのあとはコンテキストと要求ディスパッチャ関連の一連のメソッドたちの戻り値を HTML テキストとしているが、それは次の項の実際の実験結果を見てもらったほうが判りやすい。

サーブレット 3.0 からは javax.servlet.ServletRegistration 及び javax.servlet.FilterRegistration というインターフェイスが導入されている。これらはサーブレットとフィルタを更にプログラム的に設定する為のインターフェイスである。このプログラムでは設定はしていないので、得られる Registration の Map からリスト出力しているだけである。

また、RequestDispatcher には「要求のフォワードとインクルード」の節で説明したように幾つかのフィールドが定義されており、それを呼び出した結果も HTML テキスト出力している。

```
package context;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

import javax.servlet.FilterRegistration;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import javax.servlet.SessionTrackingMode;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/ForwardTarget")
public class ForwadTarget extends HttpServlet {

   /**
    * ForwardTarget は、RequestDispatcher の学習用の基礎的なサーブレット
    * @author Cresc Corps.
    * @December 2010, CRESC Corps.
    */

   private static final long serialVersionUID = 1L;

   /**
    * 処理本体    *
    */

   public void performTask(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {
      res.setContentType("text/html; charset=windows-31j");//応答ヘッダ Content-Type 追加
      res.setHeader("Cache-Control", "no-cache");           //キャッシュを殺しておく
      PrintWriter out = res.getWriter();                 //出力バッファ取得
      out.println("<html><head><title>Forward Target</title>");
      out.println(         // 同じことを HTML テキストでも指定
            "<meta http-equiv=\"cache-control\" content=\"no-cache\">" +
      "<meta http-equiv=\"content-type\" content=\"text/html; charset=Windows-31J\"></head>");
      out.println("<body>");

      // フォワード元が作った HTML テキストを追加
      StringBuffer ssd = (StringBuffer)req.getAttribute("sourceSrevletData");
      if (ssd != null) out.println(ssd);

      // タイトル表示
      out.println("<h1>ディスパッチされたサーブレットからの報告</h1>");

      // このサーブレットのデータを付加
      out.println("<pre>");
      out.println("コンテキスト情報(Context Informations) : ");
      ServletContext ctxt = getServletContext();
      out.println("getServerInfo: " + ctxt.getServerInfo());
      out.println("getEffectiveMajorVersion: " + ctxt.getEffectiveMajorVersion());
      out.println("getEffectiveMinorVersion: " + ctxt.getEffectiveMinorVersion());
      out.println("getClassLoader: " + ctxt.getClassLoader().toString());
      out.println("getServletContextName: " + ctxt.getServletContextName());
      out.println("getServletRegistrations: ");
```

```
      Map<String, ? extends ServletRegistration> srs = ctxt.getServletRegistrations();
      Iterator<String> it = srs.keySet().iterator();
      while (it.hasNext()) {
         out.println("  " + it.next());
      }
      out.println("getFilterRegistrations: ");
      Map<String, ? extends FilterRegistration> frs = ctxt.getFilterRegistrations();
      it = frs.keySet().iterator();
      while (it.hasNext()) {
         out.println("  " + it.next());
      }
      Set<SessionTrackingMode> stms = ctxt.getDefaultSessionTrackingModes();
      out.print("DefaultSessionTrackingMode : ");
      for (SessionTrackingMode c : stms)
         out.print(c + ", ");
      out.println();
      out.println();
      out.println("パス関係情報(Path Related Methods)：");
      out.println("getRequestURI: " + req.getRequestURI());
      out.println("getContextPath: " + req.getContextPath());
      out.println("getServletPath: " + req.getServletPath());
      out.println("getPathTranslated: " + req.getPathTranslated());
      out.println("getRealPath: " + ctxt.getRealPath(req.getRequestURI()));
      out.println("getPathInfo: " + req.getPathInfo());
      out.println("getQueryString: " + req.getQueryString());
      out.println("javax.servlet.forward.request_uri: " +
req.getAttribute("javax.servlet.forward.request_uri"));
      out.println("javax.servlet.forward.context_path: " +
req.getAttribute("javax.servlet.forward.context_path"));
      out.println("javax.servlet.forward.servlet_path: " +
req.getAttribute("javax.servlet.forward.servlet_path"));
      out.println("javax.servlet.forward.path_info: " +
req.getAttribute("javax.servlet.forward.path_info"));
      out.println("javax.servlet.forward.query_string: " +
req.getAttribute("javax.servlet.forward.query_string"));
      out.println();
      out.println("要求パラメタ(Request Parameters):");
      //注意：setCharacterEncodingメソッドはGET要求には効かないので使わないことを奨励
      Enumeration<?> paramNames = req.getParameterNames();
      while (paramNames.hasMoreElements()) {
         String name = (String) paramNames.nextElement();
         String[] values = req.getParameterValues(name);
         try{
            out.println("  "+new String(name.getBytes("8859_1"), "Windows-31J")+":");
         }catch(UnsupportedEncodingException theException){
            out.println("UnsupportedExceptionの例外が発生");
         }
         for (int i = 0; i < values.length; i++) {
            try{
               out.println("  "+i+")  "+new String(values[i].getBytes("8859_1"), "Windows-31J"));
            }catch(UnsupportedEncodingException theException){
               out.println("UnsupportedExceptionの例外が発生");
            }
         }
      }
      out.println();
      out.println("コンテキスト属性(Context Attributes):");
      Enumeration<?> attributeNames = ctxt.getAttributeNames();
      while (attributeNames.hasMoreElements()) {
         String name = (String) attributeNames.nextElement();
         Object value = ctxt.getAttribute(name);
         out.println("  " + name + " : " + value.toString());
      }
      out.println("</pre></body></html>");              //HTMLフッタの出力
      out.flush();                                  //バッファの明示的吐き出し
      out.close();                                  //バッファの明示的クローズ
   }


   /**
    * 到来HTTP GET要求の処理
    */
   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
```

```
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res);
   }

   /**
    * 到来HTTP POST要求の処理
    */
   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws javax.servlet.ServletException, java.io.IOException {
      performTask(req, res);
   }

   /**
    * 本サーブレット情報の文字列を返す
    */
   @Override
   public String getServletInfo() {
      return "Forward Test Taget Servlet, Version 1.0 by Cresc";
   }
}
```

11.5.3 ForwardTestForm.html

このHTMLファイルは前述のようにForwardTestForm.htmlは、ForwardTestサーブレットを呼び出すための静的なファイルで、以前要求オブジェクトで使ったRequestTestForm1.htmlを一部変更しただけのものであり、Eclipse上ではtutorial\WebContentのフォルダに置かれる。このページはブラウザからはhttp://localhost:8080/tutorial/ForwardTestForm.htmlを指定してアクセスできる。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-31j">
<title>Insert title here</title>
</head>
<body>
<H1>ForwardTestサーブレットに要求パラメタとしてテキスト・ボックスの内容をわたす</H1>
<form method="post" action="http://localhost:8080/tutorial/ForwardTest">

<textarea rows="4" cols="40" name="サブミット・データ"></textarea><br>
<input type="submit" value="Submit using POST">
</form>
<br>
<form method="get" action="http://localhost:8080/tutorial/ForwardTest">
<textarea rows="4" cols="40" name="ゲット・データ"></textarea><br>
<input type="submit" value="Submit using GET">
</form>
</body>
</html>
```

11.5.4 テスト結果例

それでは実際にForwarTestForm.html経由で、ForwardTest(そしてForwardTarget)をアクセスした例を示そう。

この例はGETで名前が"ゲット・データ”、値が"本日は晴天なり”というテキストをHTTP要求パラメタとしてForwardTestサーブレットを呼び出したものである。

以下はそのブラウザ画面の一部である。これらの結果を見れば、「ServletContextのメソッドたち」の節で示したAPIの動作と使い方の理解がより早くなろう。

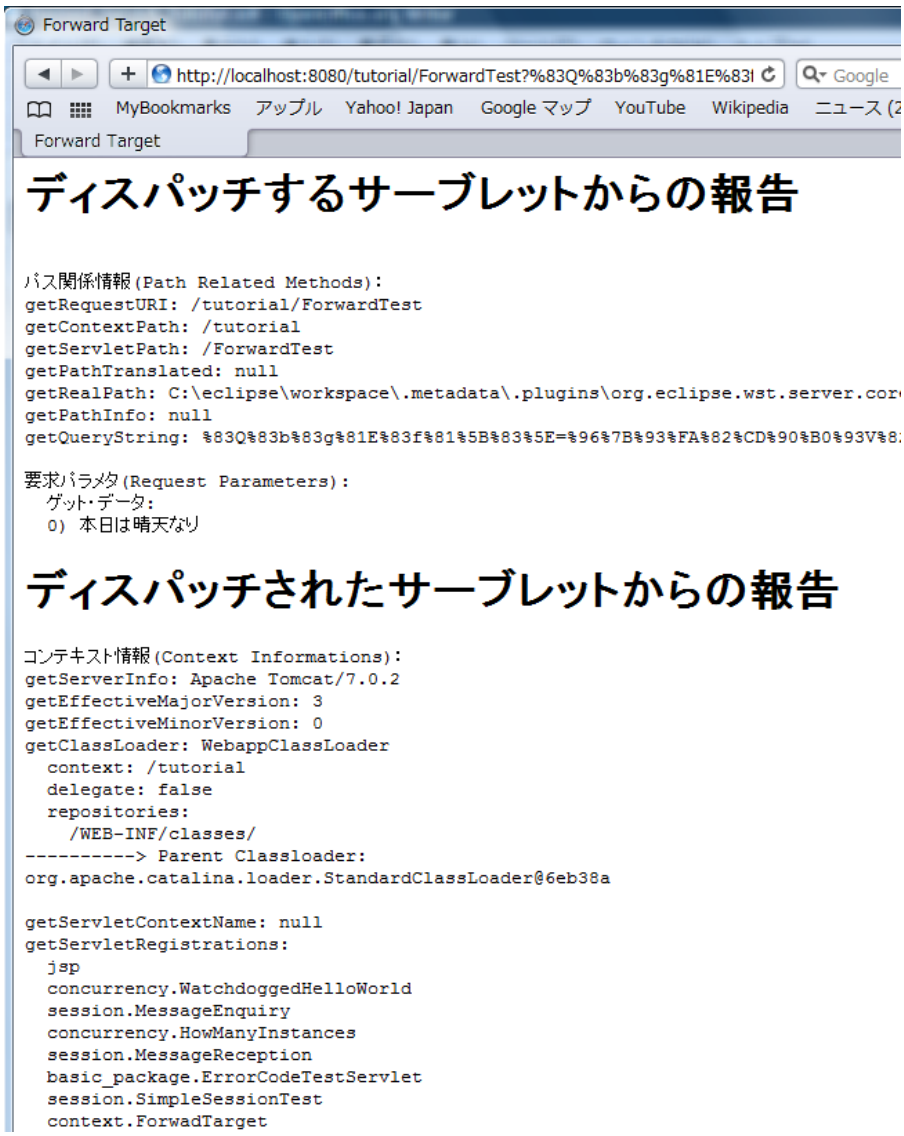


図 11-5:ForwardTest サーブレットのアクセス例

この出力は長いので、行番号を付して説明することにする。

- 004-015 行は、ForwardTest サーブレットが作成したパス関連情報である。これは既に応答オブジェクトの節で使った HttpRequestDump の出力で馴染みのデータである。GET 要求パラメタとして日本語のテキストが使われているので、getQueryString メソッドの戻り値は URL エンコードもあって長くなっている。

- 016-050 行は、ForwardTarget サーブレットが作ったコンテキスト情報である。これらの結果は、ServletContext インターフェイスのメソッドたちの使い方の例として参考になろう。
  - 022 行の getClassLoader では、このコンテキストの為に使われているクラス・ローダが返される。これは「Tomcat のクラス・ローダ」の節を参照されたい。そこで説明されているように、WebappClassLoader という名前のクラス・ローダが使われていて、その属性としては context が/tutorial、delegate が false(即ち委譲モデルを使わない)ことなどが判る。
  - 031 行の getServletRegistrations では、そのコンテキストで登録されているサーブレットの ServletRegistration オブジェクトのマップが出力される。ここでは我々が使ったサーブレット以外に jsp 及び default が登録されている。Default servlet は静的ドキュメントを呼び出すために使われている。
  - 048-049 行の getFilterRegistrations: には filters.SessionTimeoutFilter という「フィルタ」の章で使うテ

スト用のフィルタが登録されていることが判る。

- 050 行の DefaultSessionTrackingMode メソッドは、「クッキーをセッションに使わないよう設定する」の項と「セッショ追跡モードのプログラム的な追加と設定」の項で既に説明している。ここではデフォルトではクッキーと URL 書き換えの双方が使えることが判る。

- 052-064 行はパス関係のメソッドである。これと ForwardTest サーブレットが作ったパス関係のメソッド出力を比較されたい。ここでは ForwardTarget のパス情報が出力されていることが判る。それではフォワード元のパス情報はどうやって知るかといえば、それは 060-064 行のように、RequestDispatcher のフィールドを使うことになる。

- 059 行のように、フォワード先に渡されるクエリ文字列は、”0%93%E0%8A%74%91%8D%97%9D%91%E5%90%62=%90%9B%92%BC%90%6C”と、フォワード元が作ったクエリ文字列であり、フォワード元が受けたクエリ文字列は 064 行のように"%83Q%83b%83g%81E%83f%81%5B%83%5E=%96%7B%93%FA%82%CD%90%B0%93V%82%C8%82%E8”という文字列である。

- 066-070 行は要求パラメタであり、014-015 行のフォワードもとでの要求パラメタ（ゲット・データ、本日は晴天なり）に加えて、フォワード元がクエリ文字列で追加した要求パラメタ（内閣総理大臣、菅直人）が追加されていることに注意されたい。

- 072-960 行はコンテキスト属性である。これで見て判るように、ServletContext 属性には、setAttribute でサーブレットがセットしたもの以外に、多くの属性が含まれていることが判る。
  - 073　org.apache.tomcat.util.scan.MergedWebXml：はアノテーション、配備記述子などを合わせた実効的な配備記述子ともいえ、これは非常に長いものである。その理由はサーブレットのマッピングと MIME タイプが含まれているからである。それ以外は 954-960 行の 7 個である。
  - ForwardTest サーブレットが付加した属性は、959 行目のコンテキスト属性名：コンテキスト属性値である。

  この属性たちを見れば、これはマシンに対応したものが多く含まれていて、分散環境では共有出来ないものであることが判る。**従ってコンテキスト属性は、分散環境では利用できない**ことが理解できよう。

以下はブラウザ画面（行番号付き）の例である：

```
001 ディスパッチするサーブレットからの報告
002
003
004 パス関係情報(Path Related Methods)：
005 getRequestURI: /tutorial/ForwardTest
006 getContextPath: /tutorial
007 getServletPath: /ForwardTest
008 getPathTranslated: null
009 getRealPath:
C:\eclipse\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\tutorial\tutori
al\ForwardTest
010 getPathInfo: null
011 getQueryString: %83Q%83b%83g%81E%83f%81%5B%83%5E=%96%7B%93%FA%82%CD%90%B0%93V%82%C8%82%E8
012
013 要求パラメタ(Request Parameters):
014   ゲット・データ：
015   0） 本日は晴天なり

016 ディスパッチされたサーブレットからの報告
017
018 コンテキスト情報(Context Informations)：
019 getServerInfo: Apache Tomcat/7.0.2
020 getEffectiveMajorVersion: 3
021 getEffectiveMinorVersion: 0
022 getClassLoader: WebappClassLoader
023   context: /tutorial
```

```
024   delegate: false
025   repositories:
026     /WEB-INF/classes/
027 ----------> Parent Classloader:
028 org.apache.catalina.loader.StandardClassLoader@6eb38a
029
030 getServletContextName: null
031 getServletRegistrations:
032   jsp
033   concurrency.WatchdoggedHelloWorld
034   session.MessageEnquiry
035   concurrency.HowManyInstances
036   session.MessageReception
037   basic_package.ErrorCodeTestServlet
038   session.SimpleSessionTest
039   context.ForwadTarget
040   default
041   basic_package.HelloWorld
042   basic_package.HttpRequestDump
043   concurrency.SimpleThreadTest
044   basic_package.SendJpegBinary
045   basic_package.RequestBodyOutViaStream
046   basic_package.GetStaticDocument
047   context.ForwardTest
048 getFilterRegistrations:
049   filters.SessionTimeoutFilter
050 DefaultSessionTrackingMode : COOKIE, URL,
051
052 パス関係情報(Path Related Methods):
053 getRequestURI: /tutorial/ForwardTarget
054 getContextPath: /tutorial
055 getServletPath: /ForwardTarget
056 getPathTranslated: null
057 getRealPath:
C:\eclipse\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\tutorial\tutori
al\ForwardTarget
058 getPathInfo: null
059 getQueryString: %93%E0%8A%74%91%8D%97%9D%91%E5%90%62=%90%9B%92%BC%90%6C
060 javax.servlet.forward.request_uri: /tutorial/ForwardTest
061 javax.servlet.forward.context_path: /tutorial
062 javax.servlet.forward.servlet_path: /ForwardTest
063 javax.servlet.forward.path_info: null
064 javax.servlet.forward.query_string: %83Q%83b%83g%81E%83f%81%5B%83%5E=%96%7B%93%FA%82%CD
%90%B0%93V%82%C8%82%E8
065
066 要求パラメタ(Request Parameters):
067   ゲット・データ:
068   0)  本日は晴天なり
069   内閣総理大臣:
070   0)  菅直人
071
072 コンテキスト属性(Context Attributes):
073   org.apache.tomcat.util.scan.MergedWebXml :
074
075 ＊＊＊＊以下は配備記述なので省略　＊＊＊＊
076
245 ＊＊＊＊以下はMIMEタイプなので省略　＊＊＊＊

942
954   org.apache.catalina.WELCOME_FILES : [Ljava.lang.String;@1b6101e
955   javax.servlet.context.tempdir :
C:\eclipse\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\work\Catalina\localhost\tu
torial
956   org.apache.catalina.jsp_classpath :
/C:/eclipse/workspace/.metadata/.plugins/org.eclipse.wst.server.core/tmp0/wtpwebapps/tutorial/WEB-
INF/classes/;/C:/eclipse/workspace/.metadata/.plugins/org.eclipse.wst.server.core/tmp0/wtpwebapps/t
utorial/WEB-INF/lib/slf4j-api-
1.6.1.jar;/C:/eclipse/workspace/.metadata/.plugins/org.eclipse.wst.server.core/tmp0/wtpwebapps/tuto
rial/WEB-INF/lib/slf4j-simple-1.6.1.jar;/C:/tomcat7/lib/;/C:/tomcat7/lib/annotations-
api.jar;/C:/tomcat7/lib/catalina-ant.jar;/C:/tomcat7/lib/catalina-ha.jar;/C:/tomcat7/lib/catalina-
tribes.jar;/C:/tomcat7/lib/catalina.jar;/C:/tomcat7/lib/ecj-3.6.jar;/C:/tomcat7/lib/el-
api.jar;/C:/tomcat7/lib/jasper-el.jar;/C:/tomcat7/lib/jasper.jar;/C:/tomcat7/lib/jsp-
api.jar;/C:/tomcat7/lib/servlet-api.jar;/C:/tomcat7/lib/tomcat-api.jar;/C:/tomcat7/lib/tomcat-
coyote.jar;/C:/tomcat7/lib/tomcat-dbcp.jar;/C:/tomcat7/lib/tomcat-i18n-
```

```
es.jar;/C:/tomcat7/lib/tomcat-i18n-fr.jar;/C:/tomcat7/lib/tomcat-i18n-
ja.jar;/C:/tomcat7/lib/tomcat-util.jar;/C:/tomcat7/bin/bootstrap.jar;/C:/tomcat7/bin/tomcat-
juli.jar;/C:/Program%20Files/Java/jdk1.6.0_20/lib/tools.jar;/C:/Program
%20Files/Java/jdk1.6.0_20/jre/lib/ext/dnsns.jar;/C:/Program
%20Files/Java/jdk1.6.0_20/jre/lib/ext/localedata.jar;/C:/Program
%20Files/Java/jdk1.6.0_20/jre/lib/ext/sunjce_provider.jar;/C:/Program
%20Files/Java/jdk1.6.0_20/jre/lib/ext/sunmscapi.jar;/C:/Program
%20Files/Java/jdk1.6.0_20/jre/lib/ext/sunpkcs11.jar
957   org.apache.catalina.resources : org.apache.naming.resources.ProxyDirContext@1dc423f
958   org.apache.tomcat.JarScanner : org.apache.tomcat.util.scan.StandardJarScanner@1815bfb
959   コンテキスト属性名 ： コンテキスト属性値
960   org.apache.tomcat.InstanceManager : org.apache.catalina.core.DefaultInstanceManager@1bdc9d8
```

11.6節　自動再ロードの問題

セッションの教材として使ったMessageManager クラスなどのシングルトンなどでも判るように、static キーワードはデータベース接続プールのような共有資源では不可欠と言える。しかしながらstatic キーワードを付したできたインスタンスを、はたしてJVM は唯一のものとして扱ってくれるのだろうか？実際はそんなに簡単なものではなく、static なクラスやインスタンスの範囲はあるクラス・ローダのなかでのみ有効なのである。このことは特に複数のクラス・ローダが存在するTomcat のようなサーブレット・コンテナやEJB サーバでは要注意である。この問題に関しては、例えばTed Neward 氏が書いたJavageeks.com のホワイト・ペーパなど幾つかの資料が詳しく取り扱っている。また当社のサイトの技術資料でもこれを説明しているが、もう一度ここでも取り上げることにする。

11.6.1　自動再ロードで static な要素も再ロードされる

読者は「セッションのイベント処理」の節で示したメッセージ・サービスを実験した際、別のクラスを編集したにも関わらず自動再ロードで MessageManager が蓄積したメッセージが消えてしまったことに気がつかれたかもしれない。

例えば下図のように、Tomcat のコンテナ上でメッセージ・サービスに関するクラスたちと SimpleSessionTest.java を呼び出してインスタンス化してみよう。

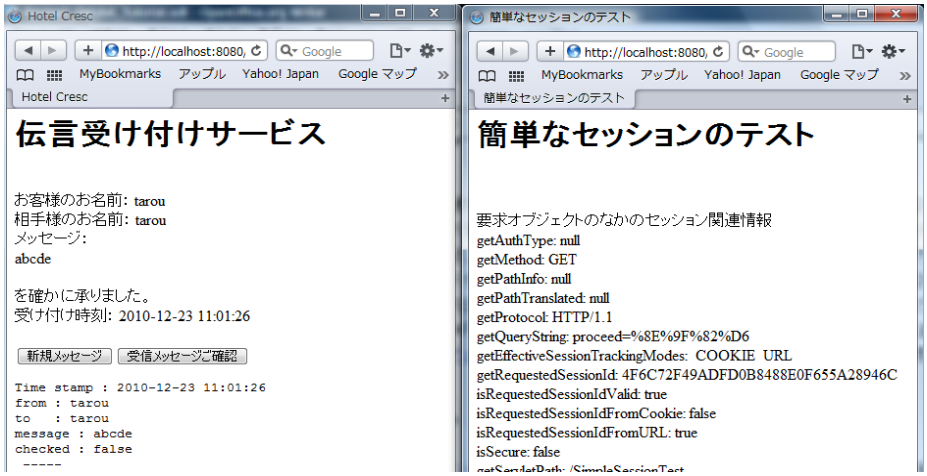


図 11-6：自動再ロードの実験

この状態で Eclipse のエディタ上で SimpleSessionTest.java のソースコードを編集（例えばどこかに空白行を追加

するとか)して、それを保管しよう。そうするとコンソールに次のようなメッセージが出力される。

```
2010/12/23 11:01:05 org.apache.catalina.core.StandardContext reload
情報: このコンテキストの再ロードを開始しました
```

この状態でメッセージ・サービスを継続すると、これまでに蓄積されていたメッセージが消えてしまうのが確認されよう。この問題は、実際の商用配備で起きると深刻である。あるサービスが稼働中に一部の簡単な変更が必要になったとき、Tomcat を稼働させたままでそれを行うと、そのコンテナ内でインスタンス化されている static な要素が総て再ロードされ、初期化されてしまう。

11.6.2 何が起きるのか

クラス変数(MessageManager の場合は private static Vector<Message> messages = new Vector<Message>();として定義されていた messages がそうである)は、そのクラスが何個インスタンス化されてもその実体はただひとつである。クラス・ローダがそのクラスを再ロードしたとしても、既にそのクラス変数が存在しているので、それを更新することはない。

しかしながらそのような static 要素の取り扱いは、クラス・ローダのインスタンスあたりであって、総てのクラス・ローダが上記のルールに従っている訳ではない。これは同じ JVM 上で別のクラス・ローダによってロードされる限り、同じクラスの別のバージョンのインスタンスがロードできるようにした為である。あるサーバ上で古いバージョンのクラスのインスタンスがそのまま存続していても、新しいバージョンのクラスのインスタンスが生成できれば、たとえサービス中であってもそのサービスを止めることなくバージョンアップが可能となる。但しそれは「static 要素が問題にならないようになっていれば」という条件が付く。

クラス・ローダは、トリー構成になっている。Java APIドキュメントには ClassLoader に関して次のように書いている:

> ClassLoader クラスは、委譲モデルを使ってクラスとリソースを探します。ClassLoader の各インスタンスは、関連する親クラス・ローダーを持ちます。クラスまたはリソースを見つけるために呼び出されると、ClassLoader インスタンスはそれ自体でクラスまたはリソースの検索を試みる前に、その検索を親クラスに委譲します。「ブートストラップ・クラス・ローダー」と呼ばれる仮想マシンの組み込みクラス・ローダーはそれ自体では親を持たず、ClassLoader インスタンスの親として動作します。

従って別のクラス・ローダがそのクラスをインスタンス化するというコードでは、そのクラス・ローダは親のクラス・ローダにその作業を委譲(delegate)する。親のクラス・ローダはその子供のひとつがそれを既にインスタンス化していることを調べることなく、そのクラスを自分のクラスパス領域で見つけられないと、見つからないとそのクラス・ローダに返してくるので、そのクラス・ローダは兄弟のクラス・ローダを調べることなく、新しいインスタンスを生成してしまう。そのクラス・ローダにとっては、そのクラスを初めてインスタンス化するのであるから、static な要素も新しく生成してしまう。

再ロード機能を持ったサーブレット・エンジンも、サービスを停止することなく(古いバージョンのインスタンスがあるスレッドによって使われているとしても)新しいサーブレットのバージョンに更新する為に、別のクラス・ローダによって新しいバージョンのクラスをロードする方法をとっている。Servlet 仕様書のバージョン 2.3 以降の webappX のクラス・ローダは、Java 2 の仕様と異なり、親のクラス・ローダに委譲する前に先ず自分のレポジトリを調べるようになってはいるが、仕様により別のクラス・ローダが既にインスタンス化しているかどうかは調べないので、結局クラス変数を含めて新しいインスタンスが作られることになる。こうすることで、たとえ古いバージョンのインスタンスをある要求スレッドが使っていても、その間起きた再ロードの影響を受けないことは、当社の技術資料の実験でも理解されよう。

しかしながら古いバージョンのインスタンスを残すことによるメリットに比べて、static 要素がリセットされるというデメ

リットはかなり大きいといわねばならない。

11.6.3 対策

Tomcat では「Tomcat 7 のクラス・ローダ階層」の図で示したようなトリー構成になっている。従って各アプリケーション用の webappX のクラス・ローダの親は Common というクラス・ローダになる。common という名前は、サーバと総てのウェブ・アプリケーションで共有されるクラスが置かれるということから名づけられている。Common のクラス・ローダのクラスパスは$CATALINA_HOME/lib にある総てのパックされていないクラスとリソース、及び JAR ファイルの総てのクラスとリソースが可視となっている。従って**そのような共有クラスは$CATALINA_HOME/lib に置く**のがこの問題を回避する手段のひとつである。ただしこのディレクトリに置かれたクラスは、高い信頼性があることが求められる。

メッセージ・サービスの教材では Vector<Message>をメッセージのリポジトリとして使っていたが、殆どのウェブ・アプリケーションではデータベースが使われる。その場合は複数のスレッドあるいはサーブレットからのデータベース接続を効率化する為に、シングルトンのデータベース接続マネージャを使用する。シングルトンはこれまで説明してきたように、static なフィールドを使用している。従ってデータベース接続マネージャも自動再ロード問題の対象になる。その為、Tomcat の$CATALINA_HOME/lib に tomcat-dbcp.jar という接続プールのクラスのパッケージが収容されている。たとえば、Tomcat の「JNDI データ・ソースの手引き」には次のように記されている:

> 注意:これらの jar ファイルを/WEB-INF/lib や$JAVA_HOME/jre/lib/ext、その他の場所にインストールしないでください。$CATALINA_HOME/common/lib 以外の場所にインストールすると問題が発生するでしょう。

Eclipse で開発したアプリケーション、例えば教材のメッセージ・サービスを自動再ロードをオンにした実際の Tomcat に配備した後で、その一部例えば session\MessageManager.class を$CATALINA_HOME/lib に移すだけでは、通常はそのアプリケーションは動作しない。Message.class も移さないと、Common のクラス・ローダは見つからないとの例外をスローする。以下はそのようにした場合に、再ロードを行っても MesageManager が保持しているデータが消えたりしなことを確認する手順である。

1. $CATALINA_HOME/lib/session に、$CATALINA_HOME/webapps/tutorial/web-inf/classes/session/のなかの MessageManager.class 及び Message.class を移す。
2. この状態で DOS レベルで Tomcat 7 をスタートさせ、ブラウザで幾つかのメッセージを MessageReception サーブレットに入力する。
3. この状態で、Eclipse 上で MessageEnquiry.java をスペースを追加するとかで再保管し、Tomcat を開始/停止を行う。
4. そうすると、
   C:\eclipse\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\tutorial\WEB-INF\classes\session
   には、新しいバージョンの MessageEnquiry.class ができるので、これをエクスプローラを使って $CATALINA_HOME/webapps/tutorial/web-inf/classes/session/に上書きコピーする。
5. Tomcat のウィンドウには「コンテキストの再ロードを開始しました」及び「コンテキストの再ロードを終了しました」のメッセージが出力される。
6. この状態で引き続きブラウザからの MessegeReception サーブレットへの入力を行うと、蓄積されていたメッセージがすべて維持されていることが確認できる。

しかしなら、一番良い対策は商用配備時には自動再ロードをしないよう設定することかもしれない。この機能では、常時クラス・ファイルに変更が出たかを監視するので、オーバヘッドになるからである。

11.7節　　この章のまとめ

以下は、この章の内容をもとにした、サーブレットにおけるスレッド安全の幾つかのポイントである:

1. サーブレット 3.0 では ServletContext クラスには多くのメソッドが追加されている。これらを理解すると、より効率的なアプリケーション開発が可能になる。
2. ServletContext 以外に ServletRequest (あるいはそのサブクラスの) HttpServletRequest にも getRequestDispatcher と同じ名前のメソッドが存在する。その相違は、ServletRequest#getRequestDispatcher のほうはより強化されていて、相対パスと絶対パスが使えることである。
3. 要求ディスパッチャ取得にクエリ文字列を指定するときは、名前と値には URL エンコードが必要である。
4. サーブレット・コンテキストに付加する属性は、分散環境では使用できないので、データベースなどの使用を考える。
5. 自動再ロードをオンにするときは、static な要素をもったクラス・ファイルを、WebappX ではなくて Common のクラス・ローダの領域におかねばならない。

第12章　　サーブレット・フィルタ

サーブレット・フィルタはサーブレットの前に挿入され、要求あるいは/及び応答オブジェクトに必要な前後処理を行う。フィルタに関しては、「要求と応答のフィルタリング」の節でその概要を示してある。フィルタはサーブレットの 2.3 版から導入されている。

可能な処理としては:

- 要求を調べそれに従ってアクションをとる
- 要求と応答の対がこれ以上通過するのを阻止する
- HTTP 要求メッセージのヘッダとデータに手を加える。これはカスタム化された要求のバージョンをわたすことでなされる
- HTTP 応答のメッセージヘッダとデータに手を加える。これはカスタム化された応答のバージョンを渡すことでなされる
- 外部リソースと関わり合う

サーブレット・フィルタの応用としては:

- 認証フィルタ
- ログと監査の為のフィルタ
- イメージの変換のフィルタ
- データ圧縮フィルタ
- 暗号化のフィルタ
- トークン化(Tokenizing)フィルタ
- リソースへのアクセスのイベントを発生させるフィルタ
- XSL/T フィルタ
- MIME タイプのチェインのフィルタ

等々がある。

12.1節　　サーブレット・フィルタの概念

この節では、フィルタの基本的な概念を簡単なフィルタを使って説明する。

サーブレットにおけるフィルタは、以下のような概念でとらえることが出来よう:

1. フィルタは配備記述子及び@WebFilter アノテーションの宣言に基づきサーブレット・コンテナが呼び出す。
2. フィルタはサーブレット・エンジンとターゲットとなるサーブレットの間に介在する。設計上はインターセプタ・フィルタ・パタン(Intercepting Filter pattern)が使われていて、ターゲットとなるサーブレット(あるいは JSP ページ)が感知することなく要求と応答に関与する。
3. フィルタは複数チェインとして配置できる。
4. 各フィルタはサーブレットの service()に相当する呼び出しの為の doFilter()を持つ。
5. 各フィルタは次のフィルタ(フィルタ・チェインの最後のフィルタの場合はターゲットとなるサーブレットまたは JSP)を呼ぶ場合は、FilterChain のオブジェクトの doFilter()を呼ぶことで、これを行う。

6. フィルタから次のフィルタまたはサーブレットに渡す要求と応答のオブジェクトのペアは、通常は自分が受けた要求及び/あるいは応答のオブジェクトをラップしたものである。何故なら、フィルタは通常、要求及び/あるいは応答のオブジェクトに何らかの手を加える必要があり、サーブレット・エンジンが渡したオリジナルの要求と応答のオブジェクトのペアをそのまま渡すと、その目的が果たせなくなるからである。

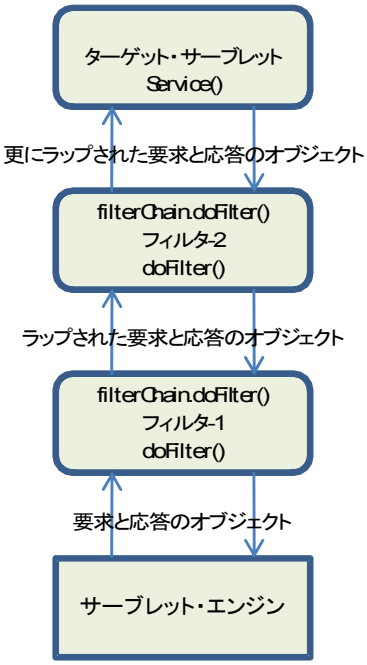


図 12-1 : フィルタのコンセプト

サーブレット・フィルタの概念は、実際のコードを見ればより理解が早まろう。以下は簡単なアクセス・ログの為のフィルタである。このフィルタは、クライアントが HelloWorld 及び HttpRequestDump のサーブレットをアクセスしたときに、クライアント IP アドレスと、アクセス時刻をロガーに出力するものである。

```
package filters;

import java.io.IOException;
import java.util.Date;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@WebFilter(filterName = "AccessLogFilter", urlPatterns = {"/HelloWorld", "/HttpRequestDump"})
public class AccessLogFilter implements Filter {

   private final static Logger LOGGER = LoggerFactory.getLogger(AccessLogFilter.class);

   @Override
   public void doFilter(ServletRequest request, ServletResponse response,
         FilterChain filterChain) throws IOException, ServletException {

      HttpServletRequest req = (HttpServletRequest) request;

      // URL パタンとクライアントの IP アドレスを取得
      String ipAddress = req.getRemoteAddr();
      String urlPattern = req.getServletPath();
```

```
        // URL パタン、IP アドレス、及びタイムスタンプをログ
        LOGGER.info("Servlet " + urlPattern + ", IP "+ipAddress + ", Time "
              + new Date().toString());

        // フィルタ・チェインの次のフィルタまたはサーブレットを呼ぶ
        filterChain.doFilter(req, response);
    }

    @Override
    public void init(FilterConfig arg0) throws ServletException {
        // 初期化パラメタの取得等を必要に応じここに書く
    }

    @Override
    public void destroy() {
        // サービス終了時に必要なリソースの開放をここで行う
    }
}
```

このフィルタは、次のようなログを出力する:

```
8092 [http-8080-exec-7] INFO filters.AccessLogFilter - Servlet /HttpRequestDump, IP
0:0:0:0:0:0:0:1, Time Sat Jan 01 21:27:56 JST 2011
```

このコードは一目瞭然なので、ポイントだけを記すと:

- @WebFilter アノテーションで、このクラスはフィルタであることをサーブレット・コンテナに知らせる。フィルタでは、URL パタンを知らせる必要がある
- フィルタは、javax.servlet.Filter インターフェイスを実装する
- doFilter メソッドは、ServletRequest、ServletResponse、FilterChain の 3 つのオブジェクトが引数となる。サーブレットの引数である HttpServletRequest 及び HttpServletResponse にするには、キャストしなければならない
- 次のフィルタまたはサーブレットの呼び出しは filterChain.doFilter(request, response)のような形式になる
- init メソッドはこのフィルタが使用するコードの初期化に使用する。引数としてもたらされる FilterConfig には、配備記述子 Web.xml やアノテーションからの初期化パラメタたちや、その他のフィルタ・レベルの情報が含まれている
- destroy メソッドはそのフィルタを削除する際にサーブレット・コンテナが呼び出すもので、このフィルタに必要なリソースの開放をここで行う

12.2節　サーブレット・フィルタの為のメソッドたち

フィルタに絡む API の詳細は「参考資料」の章の「フィルタ関係」の項を見て頂きたい。ここでは必要なインターフェイスのメソッドを略説する。なおこれ以外に要求と応答のラッパである HttpRequestWrapper と HttpResponseWrapper、及びサーブレット 3.0 で新しく追加されている動的(ダイナミック、あるいはプログラム的ともいう)なフィルタの登録・マッピングの為のインターフェイス等は別途説明する。

12.2.1　Filter インターフェイス

フィルタとはあるリソース(サーブレットあるいは静的コンテント)への要求、あるいはあるリソースからの応答、あるいはその双方上で、フィルタリングのタスクを実行するオブジェクトであり、前節で述べたように、init、doFilter、及び destroy の 3 つのメソッドがある。

Filter にはコンストラクタは存在せず、その Filter のインスタンスは配備記述子あるいは/及びアノテーションで指定されたマッピングに基づき、サーブレット・コンテナが生成する。

このインタフェースの主要なメソッドである doFilter()は、要求と応答のオブジェクトのペアに加えて、フィルタ・チェーン全体を表すためにサーブレット・コンテナが作った javax.servlet.FilterChain インスタンスを引数としている。

表 12-1：Filter インターフェイスのメソッドたち

メソッド	内容
void init(FilterConfig filterConfig)	サーブレット・コンテナがこのフィルタに対しそのフィルタをサービス開始にしようとしていることを知らせる為に呼び出す。サーブレット・コンテナはこのフィルタをインスタンス化した後で 1 回のみこの init メソッドを呼ぶ
void doFilter(ServletRequest, ServletResponse, FilterChain)	そのチェインの末端にあるリソースの為のクライアント要求に基づいたある要求/応答のペアがこのチェインを通過する度に、コンテナによって呼び出される。このメソッドが渡す FilterChain オブジェクトを使って、Filter はこのチェインの次にあるエンティティ（フィルタ、サーブレット、JSP ページ）に要求と応答を渡すことが出来る
void destroy()	サーブレット・コンテナがこのフィルタに対しそのフィルタをサービス状態から外そうとしていることを知らせる為に呼び出す。このメソッドは、このフィルタの doFilter メソッド内の総てのスレッドがこのメソッドを出た後、あるいはタイムアウト期間を経過した後にのみ呼び出される

12.2.2 FilterChain インターフェイス

FilterChain はソフトウエア開発者たちの為に、あるリソースの為のフィルタリングされた要求を渡すことが可能なように、サーブレット・コンテナが用意するオブジェクトである。フィルタたちは doFilter メソッドで渡されるこの FilterChain を使ってそのチェインの次のフィルタを呼び出す、あるいはそのフィルタがそのチェインの最後のフィルタであるときはそのチェインの最後にあるリソースを呼び出す。

表 12-2：FilterChain インターフェイスのメソッド

メソッド	内容
void doFilter(ServletRequest, ServletResponse)	そのチェインの次のフィルタを呼び出す、あるいは呼び出しもとのフィルタがそのチェインの最後のフィルタであるときは、そのチェインの最後にあるリソースが呼び出される。引数はそのチェインに沿って渡す要求と応答のオブジェクトであり、そのフィルタが要求あるいは/及び応答に何らかの加工をする場合は、ラップされたものを渡すことになる。

12.2.3 FilterConfig インターフェイス

これはフィルタの初期化中にそのフィルタに情報を渡す為にサーブレット・コンテナが使うフィルタ設定オブジェクトである。

表 12-3：FilterConfig のメソッドたち

メソッド	内容
String getFilterName()	このサーブレット・インスタンスの名前を返す。この名前はサーバの管理者が用意し、アプリケーション配備記述子の中で指定されるか、未登録（したがって無名）サーブレットの場合はそのサーブレットのクラス名が返される

メソッド	内容
ServletContext getServletContext()	呼び出し側が実行している SevletContext への参照を返す
String getInitParameter(String)	与えられた名前の初期化パラメタの値を返す。戻り値はその初期化パラメタの値を含む String、あるいは存在しないときは null
Enumeration<String> getInitParameterNames()	そのサーブレットの初期化パラメタたちの名前を String オブジェクトの Enumeration として返す、あるいはそのサーブレットが初期化パラメタを持っていないときは空の Enumeration を返す

12.3節　要求と応答のラップ

「カスタム化された要求と応答のプログラミング」の項で記されているように、クライアントからの要求を受けてサーブレット・エンジンが用意した要求及び/あるいは応答のオブジェクトにたいし、フィルタが何らかの加工を施して次のフィルタあるいはサーブレットに渡すときは、自分が用意したあるいは加工した要求あるいは応答をラップして渡さねばならない。

応答に加工する(応答メッセージのボディ部の圧縮、追加あるいは変更、あるいは差し替えなど)フィルタの場合は、通常その応答をクライアントに返す前にその応答を捕捉しなければならない。その為には、その応答を発生させるサーブレットに対し代役のストリームを渡さねばならない。その代役のストリームにより、そのサーブレットがそれを終わらせたときにオリジナルの応答のストリームを閉じてしまうことを防止し、そのフィルタがサーブレットの応答に手を加えることを可能とする。

その為に代役の入出力ストリームを渡す為に HttpServletRequestWrapper 及び HttpServletResposeWrapper というクラスが用意されている。これらのクラスは ServletRequest インターフェイスを実装した ServletRequestWrapper、及び ServlerResponse インターフェイスを実装した ServletResponseWrapper を継承したものであるので、HttpServletRequest 及び HttpServletResponse が持っているメソッドがそのまま使える。このラッパがフィルタ・チェインの doFilter メソッドに渡される。ラッパのメソッドたちはデフォルトではラップされた要求または応答オブジェクトを呼ぶようになっている。

これらの Wrapper のオブジェクトを生成するには、フィルタが用意あるいは加工した要求及び応答のオブジェクトを引数にしたコンストラクタを使用する。但し通常のフィルタでは、この Wrapper のクラスを継承したラッパを用意し、ターゲットのサーブレットではそのフィルタの存在を認識しなくても良いようにする。**フィルタはターゲットのリソースが要求あるいは応答の為どのメソッドを使用しているかに依存してはならない**。

12.3.1　汎用応答ラッパ

暗号化フィルタや、訪問回数、会社名、著作権その他の共通情報を応答するページに付加するフィルタなどでは、応答を操作することになる。その為にはここでは Oracle (Sun を買収した) の開発者向けガイドにある汎用応答ラッパ(ターゲットが JSP であることを想定している)が雛型として有用なので、まずこれを紹介する。

この例では、ServletOutputStream をカスタム化したものを用意し、そのオブジェクトにサーブレットあるいは JSP が getWriter または getOutputStream メソッドを使ってフィルタが用意するバイト配列に書き込ませるようにしている。その為に、getWriter メソッドは getOutputStream で取得した出力ストリームをもとに PrintWriter オブジェクトを

生成するようにさせている。ターゲットが書き込んだ HTTP 応答メッセージのボディ部コンテントはバイト配列出力ストリームに(ByteArrayOutputStream)書き込まれ、ネットワークには送出されない。フィルタはあとでそのコンテントを加工するときは、このバイト配列出力ストリームにアクセスし、終わったらそれをサーブレット・エンジンが用意している出力バッファに送りこむ(サーブレット・エンジンが用意する出力バッファに関しては、「ボディ部への書き込み」の節を参照のこと)。

この例での 3 つのクラスが用意されている:

- GenericResponseWrapper:これは応答ラッパ・インターフェイスの汎用実装物である。
- FilterServletOutputStream:これは応答ラッパの為に ServletOutputStream を継承したもので、ターゲットが呼ぶメソッドで必要なものをオーバライドしている。
- PrePostFilter:これが Filter インターフェイスを実装したサンプルのフィルタのコードである。

この例では、ターゲットに渡す前に応答をラップする為の HttpServletResponseWrapper クラスを使っている。このクラスは ServletResponse オブジェクトのラッパとして機能するオブジェクトで、デコレータ・デザイン・パタンを使っている。これによりターゲットが応答を返した後でそれを加工することが可能になる。

この例で使われている HTTP サーブレット応答ラッパはカスタムのサーブレット出力ストリーム(FilterServletOutputStream)を使っており、このラッパがサーブレット(あるいは JSP)側で書き込み終了後にフィルタ側で操作することを可能にしている。こうしないと、通常はサーブレットの service メソッドを抜けた時点で書き込まれたデータはネットワークに送出されてしまい、フィルタは手を加えることが出来ない。

下図はこの汎用応答ラッパの構成である:

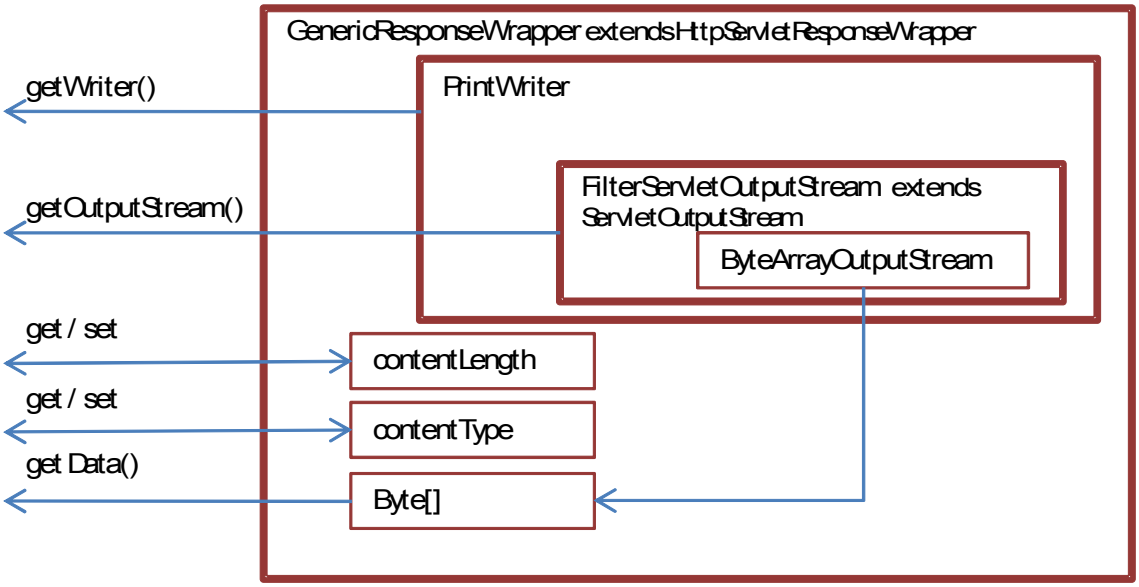


図 12-2:汎用応答ラッパの構成

ターゲットのサーブレットや JSP は、従来どおり getWriter メソッドで PrintWriter を、または getOutputStream メソッドを使って OutputStream のオブジェクトを取得して応答データを書き込むが、実際は ByteArrayOutputStream に書き込むことになる。

注意しなければならないのは、contentLength と contentType で、これはフィルタ側でもアクセスできるようにゲッタとセッタが用意してある。また、フィルタ側でこのラッパに書き込まれたデータをバイト配列として取り出す為の getData というメソッドも用意されている。

12.3.2 汎用応答ラッパとそれを使ったフィルタ例のプログラム・コード

まず、汎用応答ラッパである GenericResponseWrapper クラスのコードを示す:

GenericResponseWrapper

```
package filters;

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

public class GenericResponseWrapper extends HttpServletResponseWrapper {
   private ByteArrayOutputStream output;
   private int contentLength;
   private String contentType;

   public GenericResponseWrapper(HttpServletResponse response) {
      super(response);
      output=new ByteArrayOutputStream();
   }

   public byte[] getData() {
      return output.toByteArray();
   }

   public ServletOutputStream getOutputStream() {
      return new FilterServletOutputStream(output);
   }

   public PrintWriter getWriter() {
      return new PrintWriter(getOutputStream(),true);
   }

   public void setContentLength(int length) {
      this.contentLength = length;
      super.setContentLength(length);
   }

   public int getContentLength() {
      return contentLength;
   }

   public void setContentType(String type) {
      this.contentType = type;
      super.setContentType(type);
   }

   public String getContentType() {
      return contentType;
   }
}
```

汎用応答ラッパの構成図をみれば、このコードは直ちに理解されよう。

- このラッパのコンストラクタでは、指定した応答オブジェクトを引数にしてスーパー・クラスである HttpServletResponseWrapper のコンストラクタを呼んで応答アダプタを用意し、出力の為の ByteArrayOutputStream を用意している。また、contentLength と contentType も定義している。
- getData メソッドは、フィルタ・チェインの次のエンティティの為に用意した ByteArrayOutputStream のオブジェクトの内容をバイト配列として取得する。
- getOutputStream メソッドは、フィルタ・チェインの次のエンティティがこのメソッドを使って出力ストリームを取得してデータを書き込む場合の為に、サーブレット・コンテナが用意した OutputStream ではなくて、このラッパのバイト配列出力ストリームに書き込ませる為に用意されている。FilterServletOutputStream は、ServletOutputStream を継承したもので、後述のように write メソッドたちをバイト配列出力ストリーム用にオーバライドさせている。
- getWriter メソッドは、同じくフィルタ・チェインの次のエンティティがこのメソッドを使って PrintWriter を取得してデータを書き込む場合の為に、サーブレット・コンテナが用意した PrintWriter ではなくて、このラッパのバイト配列出力ストリームに書き込ませる為に用意されている。これは先ほどの

getOutputStream で取得した出力ストリームをもとに PrintWriter を生成したものを返している。PrintWriter コンストラクタの引数の true は自動フラッシュをオンにするものである。つまり println、printf、または format メソッドでフラッシュがかかりそのデータは出力ストリームに移される。

- 残りのメソッドは contentType と contentLength へのゲッタとセッタたちで、フィルタによってこれが変更となるので、その為に用意されている。

FilterServletOutputStream のコードは次のようになっている:

FilterServletOutputStream のコード

```
package filters;

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

public class FilterServletOutputStream extends ServletOutputStream {

   private DataOutputStream stream;

   public FilterServletOutputStream(OutputStream output) {
      stream = new DataOutputStream(output);
   }

   public void write(int b) throws IOException  {
      stream.write(b);
   }

   public void write(byte[] b) throws IOException  {
      stream.write(b);
   }

   public void write(byte[] b, int off, int len) throws IOException  {
      stream.write(b,off,len);
   }

   public void close(){
      // Do nothing, next entity should not close the stream
   }
}
```

このコードでは、与えられたストリームをもとに DataOutputStream を用意し、それに対するこのフィルタ及びフィルタ・チェインの次のエンティティが書き込みに使う為の幾つかのメソッドを用意している。このガイドの例では JSP をターゲットにすることを想定しているので(**実装上 JspWriter は getOutputStream()を使っている**ことに注意)、サーブレットの場合は必要ならそれ以外のメソッドを追加すれば良い。

注意して頂きたいのは、Oracle のオリジナルのコードにたいし、close メソッドを筆者が追加したことである。このメソッドは close メソッドを呼ばれてもこのストリームをクローズしないよう、何もしないメソッドとしてオーバライドされている。**これを追加した理由は、サーブレット側で明示的な出力ストリームのクローズが書かれていた場合に対処する為**である。サーブレット側では、service メソッドを抜けた時点でそれまで使われていた出力のライタやストリームは自動的にクローズされるが、プログラマは service メソッド(あるいはそれで展開された doGet 等のメソッド)の最後にフラッシュとクローズを明示的に書くのは、それを常に認識しておく為には良い習慣であり、多くのコーディング・スタイルで見ることが出来る。

Oracle のガイドの中で示されている例としての PrePostFilter のコードは次のようになっている。但しサーブレット 3.0 で追加されたアノテーションを使って、配備記述子を書かなくても良いようにしてある。このフィルタは、ターゲット(URL パタンが"/FilteredHelloWorld")が作る HTML テキストの前後にラインで挟んだテキスト(ここでは PRE と POST)を追加するものである。

PrePostFilter のコード

```
package filters;
```

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


@WebFilter(filterName = "PrePostFilter", urlPatterns = {"/FilteredHelloWorld"})
public class PrePostFilter extends MyGenericFilter {

   public void doFilter(final ServletRequest request,
         final ServletResponse response,
         FilterChain chain)
   throws IOException, ServletException {
      OutputStream out = response.getOutputStream();
      out.write(new String("<HR>PRE<HR>").getBytes());
      GenericResponseWrapper wrapper =
         new GenericResponseWrapper((HttpServletResponse) response);
      chain.doFilter(request,wrapper);
      out.write(wrapper.getData());
      out.write(new String("<HR>POST<HR>").getBytes());
      out.close();
   }
}
```

このコードは特に説明するまでもないが、MyGenericFilter (Filter を実装したフィルタの雛型) を継承したもので、doFilter メソッドの中で、サーブレット・エンジンが用意した出力ストリームに対し、以下のものを書き込む:

1. "<HR>PRE<HR>"というテキストをバイト配列にしたもの
2. フィルタ・チェインの次のエンティティが書き込んだバイト配列データを応答ラッパからとり出したもの
3. "<HR>POST<HR>"というテキストをバイト配列にしたもの

フィルタ・チェインの次のエンティティを呼び出す部分は、雛型的なものである:

```
GenericResponseWrapper wrapper = new
GenericResponseWrapper((HttpServletResponse) response);
chain.doFilter(request,wrapper);
```

なおフィルタの雛型である MyGenericFilter は次のようになっている。これを使うかどうかはプログラマの判断であろう:

MyGenericFilter のコード

```
package filters;

import javax.servlet.*;

public class MyGenericFilter implements javax.servlet.Filter {
   public FilterConfig filterConfig;

   public void doFilter(final ServletRequest request,
         final ServletResponse response,
         FilterChain chain)
   throws java.io.IOException, javax.servlet.ServletException {
      chain.doFilter(request,response);
   }

   public void init(final FilterConfig filterConfig) {
      this.filterConfig = filterConfig;
   }

   public void destroy() {
   }
}
```

12.3.3 PrePost フィルタの実行例

このフィルタのターゲットである FilteredHelloWorld.java のコードは次のようである:

FilteredHelloWorld サーブレットのコード

```
package filters;

import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/FilteredHelloWorld")
public class FilteredHelloWorld extends HttpServlet
{
   private static final long serialVersionUID = 1L;
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
   {
      res.setContentType("text/html; charset=Windows-31J");

      // PrintWriter 使用時の実験
      /**
   PrintWriter out = res.getWriter();
   out.println("<HTML>");
   out.println("<HEAD><TITLE>Filtered Hello World</TITLE></HEAD>");
   out.println("<BODY>");
   out.println("<BIG>フィルタの学習用ターゲット・サーブレット (PrintWriter で書き込み) </BIG>");
   out.println("</BODY></HTML>");
   out.flush();   // 明示的 flush に耐えられるか
   out.close();   // 明示的 close に耐えられるか
       **/
      // OutputStreamWriter 使用時の実験
      OutputStream os = res.getOutputStream();
      OutputStreamWriter osw = new OutputStreamWriter(os, "Windows-31J");
      osw.write("<HTML>");
      osw.write("<HEAD><TITLE>Filtered Hello World</TITLE></HEAD>");
      osw.write("<BODY>");
      osw.write("<BIG>フィルタの学習用ターゲット・サーブレット(OutputStreamWriter で書き込み)</BIG>");
      osw.write("</BODY></HTML>");
      osw.flush();   // 明示的 flush に耐えられるか
      osw.close();   // 明示的 close に耐えられるか
   }
}
```

このコードでは、PrintWriter で HTML テキストを書きこむブロックと、OutputStream で書き込むブロックを、コメント・アウトを使って選択できる。また双方のブロックとも明示的な flush と close が書かれている。OutputStream を使ったときは、日本語が扱えるように OutputStreamWriter でラップしている。

このようなサーブレットに対しても、PrePostFilter が正しく機能することを確認してみよう。下図はプロキシと PrePostFilter 経由でこのサーブレットをアクセスしたときの、ブラウザ画面である。PrintWriter を使ったときも、OutputStream を使ったときも同じ結果が得られることが確認できよう。

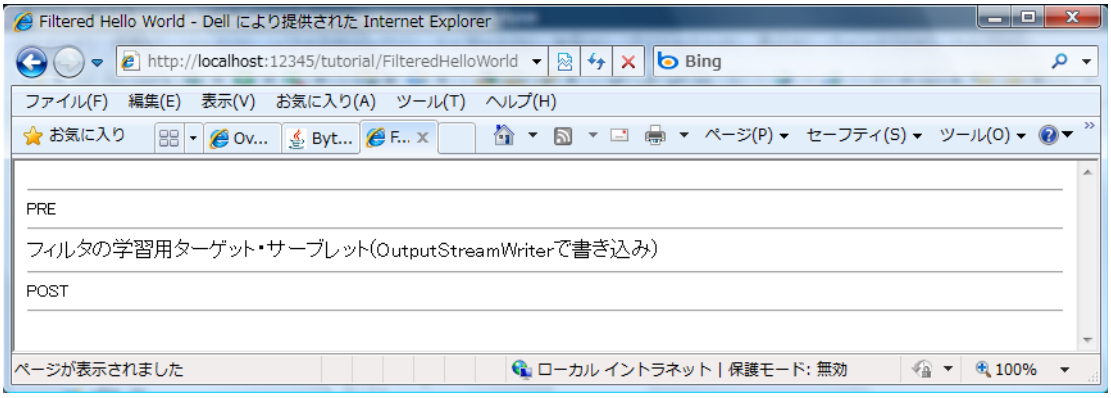


図 12-3：プロキシ経由、PrePostFilter 経由での FilteredHelloWorld のアクセス結果

以下はプロキシが出力した Tomcat からの HTTP 応答メッセージである。FilteredHelloWorld が出力した HTML テキストの前後に、PrePostFilter が付加した<HR>PRE<HR>及び<HR>POST<HR>が付けられて、正しい Content-Length ヘッダ行つきでブラウザに返されていることが判る。

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=Windows-31J
Content-Length: 178
Date: Thu, 06 Jan 2011 23:53:32 GMT

<HR>PRE<HR><HTML><HEAD><TITLE>Filtered Hello World</TITLE></HEAD><BODY><BIG>フィルタの学習用ターゲット・
サーブレット(OutputStreamWriter で書き込み)</BIG></BODY></HTML><HR>POST<HR>
```

この PrePostFilter に手を加えることで、一貫した形式でのこのページのアクセス・カウンタ、このサイトのトピックス、会社のロゴや事業所の情報、あるいは著作権などを表示するようにできる。ターゲットが作った HTML テキストに対し、より細かな加工が必要な場合は、ByteArrayOutputStream よりは、更に高度な ByteBuffer の使用も考えられよう。

12.4節　フィルタ・チェイン

サーブレットの APIドキュメントでは、**末端にあるサーブレットや JSP 等を含めてフィルタ・チェインと定義**している。従ってフィルタ・チェインの最後に位置するのはターゲットのリソースであるサーブレットや JSP である。そのようなチェインはどのように構成されるのであろうか。

どの要求 URL パタンに対しどのフィルタをどの順にマッピングするかを決めるのは、配備記述子と@WebFilter アノテーションである。その作業は通常は配備担当者であって、プログラマの仕事ではない。フィルタ・マッピングに関しては、「フィルタ・マッピングの指定」の項で既に略説してある。

サーブレット・コンテナがフィルタ・チェインを処理するかは仕様書に書かれている訳ではないが、Bryan Basham 他の"Head First Servlets & JSP"によれば、フィルタのチェイン化を次のようなスタックのプッシュ・ポップ操作で表現したほうが理解しやすいという:

表 12-4:フィルタ・チェインの実行

		サーブレット A		
	フィルタ#7	フィルタ#7	フィルタ#7	
フィルタ#3	フィルタ#3	フィルタ#3	フィルタ#3	フィルタ#3
スタック	スタック	スタック	スタック	スタック
1	2	3	4	5
コンテナはフィルタ#3 の doFilter メソッドを呼び、フィルタ#3 はその chain.doFilter メソッドに遭うまで実行する	コンテナはスタック上にフィルタ#7 をプッシュし、フィルタ#7 はその chain.doFilter メソッドに遭うまで実行する	コンテナはスタック上にサーブレット A の service メソッドをプッシュし、サーブレットはそれを終了するまで実行し、次にコンテナはそのスタックをポップ・オフする	コンテナはコントロールをフィルタ#7 に戻し、フィルタ#7 はその doFilter メソッド(doFilter 呼び出しの後の部分)を終了させ、次にコンテナはフィルタ#7 をポップ・オフする	コンテナはコントロールをフィルタ#3 に戻し、フィルタ#3 はその doFilter メソッド(doFilter 呼び出しの後の部分)を終了させ、次にコンテナはフィルタ#3 をポップ・オフし、それでコンテナはその応答を完了させ

				る

12.4.1 アプリケーションの中でのフィルタの設定

フィルタは配備記述子あるいは@WebFilter アノテーションで定義される。アノテーションによる定義は、配備記述子で metadata-complete が true に指定されている場合は、その設定でサポートされている総てのアノテーションがスキャンされなくなる(即ち無視される)ので、注意が必要である。

12.4.1.1 アノテーションによる設定

javax.servlet.Filter インターフェイスを実装したクラスに対し@WebFilter アノテーションが使える。サーブレットの@WebServlet アノテーションと同様に、このアノテーションで URL パタンを指定しなければならない。一番簡単なアノテーションは:

```
@WebFilter("/foo")
```

のような形式で、これは/foo というパスのサーブレットにこのフィルタが適用されることを意味する。

より細かな指定形式は:

```
@WebFilter(filterName = "AuthenticateFilter", urlPatterns = {"/stock.jsp",
"/getquote"})
```

のような形式で、ここではフィルタ名と、JSP を含む複数の URL パタンが指定されている。

@WebInitParam アノテーションは Servlet または Filter に渡せばならない 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ので、次のように含めることが出来る:

```
@WebFilter(filterName = "AuthenticateFilter", urlPatterns = {"/stock.jsp",
"/getquote"}, initParams={@WebInitParam(name="default_market",
value="NASDAQ")})
```

これは株価問合せサービスの認証フィルタに対し、デフォルト市場は NASDAQ であることを設定する例である。

@WebFilter アノテーションには以下の属性がある:

- filterName(そのフィルタの名前)
- description(そのフィルタの記述)
- displayName(そのフィルタの表示名)
- initParams(そのフィルタの初期化パラメタたち)
- servletNames(ターゲットとなるサーブレットたちの名)
- value(そのフィルタが適用される URL パタン)
- urlPatterns(そのフィルタが適用される URL パタンたち)
- dispatcherTypes(そのフィルタが適用されるディスパッチャのタイプたち)
- asyncSupported(そのフィルタが非同期動作対応かどうか)
- smallIcon(そのフィルタの為の小さなアイコン)
- largeIcon(そのフィルタの為の大さなアイコン)

URL パタン以外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。

12.4.1.2 配備記述子による設定

これはサーブレットのマッピングと似ているので、サーブレットの配備記述子を使った人には馴染みやすいもの

である。サーブレット仕様書によれば、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ので、注意が必要である。

最初のそのフィルタ・クラス(完全修飾名)とフィルタ名との対応の宣言(フィルタ名宣言)を行う:

```
<filter>
      <filter-name>Logging Filter</filter-name>
      <filter-class>filter.LoggingFilter</filter-class>
</filter>
```

次にフィルタ名とターゲットとのマッピング(フィルタ・マッピング宣言)を行うが、これにはフィルタ名とサーブレット名との対応付けを指定する方法と、フィルタ名と URL パタンとの対応付けで指定する方法がある。前者の場合は:

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

となり、後者の場合は:

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

のような書き方になる。この例では、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 が適用されることになる。

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

それでは、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どうなるだろうか？その場合は、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る。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>
```

以下のように展開し、前記のルールが適用される:

```
<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>
```

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:

1. サーブレット仕様書 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。
2. 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。
3. 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>がサーブレット仕様書の第 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。
4. 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。

確かにこれらのアノテーションの導入でウェブ部品の設定に際し配備記述子(web.xml)はオプショナルなものになっている。しかしながら、その設定に対し変更や更新が必要な場合には配備記述子を使ってアノテーションをオーバライドすることになろう。**配備記述子、ウェブ・フラグメント、及びアノテーション間で齟齬が起きたときは配備記述子が最高の優先度を持つ**と規定されている。また、サーブレット・コンテナは配備記述子の中で metadata-complete 要素の値によってアノテーションを使うかどうかを判断する。この値が true のときは、コンテナはアノテーションとウェブ・フラグメントを処理せず、配備記述子のみがメタデータ情報源となる。コンテナは配備記述子の metadata-complete 要素が無いかその値が true でないときにのみアノテーションとウェブ・フラグメントを処理する。

配備記述子で metadata-complete 要素が true に指定されている場合は、その設定でサポートされている総てのアノテーションがスキャンされなくなるので問題がないが、そうでなくて双方で指定されているときは、サーブレット 3.0 仕様書では 8.2 節で配備記述子がアノテーションを凌駕するとしている。しかしながら 8.2 節の内容はかなり複雑であり、その内容が理解できない場合は、**配備記述子で metadata-complete 要素を true にすることが問題を少なくする**。サーブレット 3.0 では、@WebServlet と@WebFilter アノテーションの、Web-Fragment、及びプログラム的な(動的な)フィルタの付加と設定、が導入されたことで、Web-xml との整合や順序付け(相対および絶対)が非常に複雑になり、仕様書も判りにくいものになっている。

12.5節　フォワードとインクルードの為のフィルタ

直接要求ターゲットに対するフィルタリングに加えて(またはそのかわりに)フォワード・ターゲットまたはインクルード・ターゲットに対してフィルタリングする、つまり要求ディスパッチャにも対応するようにフィルタを構成することも可能である。これはサーブレット 2.4 版から導入されている。

要求ディスパッチされたリソースの為のフィルタ・マッピング宣言は次のようになる:

```
<web-app…>
      <filter-mapping>
            <filter-name>ExampleFilter</filter-name>
            <url-pattern>*.jsp</url-pattern>
            <dispatcher>REQUEST</dispatcher>
            (and/or)
            <dispatcher>INCLUDE</dispatcher>
            (and/or)
            <dispatcher>FORWARD</dispatcher>
            (and/or)
            <dispatcher>ERROR</dispatcher>
      </filter-mapping>
</web-app>
```

0 から 4 つの<dispatcher>要素が持てることに注意されたい:

- REQUEST:クライアントからの要求に対しこのフィルタを呼ぶ。あるいは、指定された要求のターゲットに適用されるフィルタに対し、INCLUDE または FORWARD 設定に加えて使用する。<dispatcher>要素が含まれていないときは REQUEST がデフォルトとなっている。
- INCLUDE:include 呼び出しによる要求ディスパッチの為のフィルタとしてこのフィルタを呼ぶ。
- FORWARD:forward 呼び出しによる要求ディスパッチの為のフィルタとしてこのフィルタを呼ぶ。
- ERROR:エラー・ハンドラが呼び出すリソースの為のフィルタとしてこのフィルタを呼び出す。
- ASYNC:サーブレット 3.0 で導入された非同期コンテキスト・ディスパッチのメカニズムで dispatch 呼び出しで呼ばれるターゲットに対するフィルタとしてこのフィルタを呼び出す。

例えば以下の設定の場合:

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

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。

また、次の設定の場合は:

```
<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、及び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。

第13章　　非同期処理(Asynchronous Processing)

非同期処理(Asynchronous Processing)はサーブレット3.0 で新たに付加された機能である。多くの場合サーブレットはデータ処理の為に外部リソースと関わり合うことになる。そのようなリソースとしては、データベース、ウェブ・サービス、メッセージ・サービス、及びチャット・アプリケーションなどがある。サーブレットを走っている要求スレッドは、応答を実際に生成する前に、これらのリソースと関わり合っている間、それらのリソースからの応答を待つあいだブロック状態に置かれ、従って効率が悪くなてしまう。サーブレット3.0 では非同期処理を導入することでこの問題に対処している。非同期処理により、スレッドはブロックされることなくそのリソースを呼び出し、サーブレット・コンテナに戻ることが出来る。そのスレッドは他のタスクを実行でき、サーブレットの性能の効率化が図れる。リソースからの応答を管理する為の非同期コンテキスト(AsyncContext)が、そのリソースからの応答を同じスレッドに処理させるかそれともそのコンテナ内の新しいリソースにディスパッチさせるかを判断する。AsyncContext には非同期処理を実行する為のstart、dispatch、及びcomplete などのメソッドが用意されている。非同期処理はサーブレット3.0 仕様書の2.3.3.3 項に記されているので、そちらも参照されたい。

13.1節　　基本的なコンセプト

13.1.1　　問題

以下はサーブレット3.0 の仕様作成のリーダである Sun Microsystems（現在は Oracle によって買収されている）の上級スタッフ技術者の Rajiv Mordani 氏（名前でわかるようにインド人で、インド人には優れたソフトウエア技術者が多い）ほかのプレゼンテーションにあった非同期処理の必要性に関する箇所をまとめたものである：

サーブレットが待ちの状態（ブロックされた状態だともいう）になるのは以下のようなものである：

- リソースが取得可能になるのを待つ（例えば DB 接続プールから接続を取得する）
- イベントを待つ（例えばチャットのアプリケーションであるユーザとの接続を維持したまま、そのユーザがチャットしてくるのを待つ）
- 応答を待つ（例えばウェブ・サービスの応答を待つ、あるいは QoS 制御でトラフィック制限を受けた）

そのような待ちがリソースの消費をもたらしてしまう。例えばリモートのウェブ・サービスを使っているアプリケーションを考えてみよう。そのサーバが毎秒 1,000 個の要求を処理しているとし、その要求の 50%がリモートのウェブ・サービスを呼んでいて、コンテナのスレッド・プールには 500 個のスレッドが保持されているとする。しかしながら、もしリモートのウェブ・サービスが遅くて 1 秒かかるとすると、1 秒間でスレッド枯渇が起きてしまう。何故なら到来要求の 50%がその 500 個のスレッドの全部を使ってしまうからである。

スレッド枯渇が起きれば、クライアントへの応答が遅れ、サービス品質(QoS)を落とす。スレッド枯渇に対処する為にプールを大きくすればその分メモリ・リソースを消費し、また CPU 負荷が増大し、それもサービス品質(QoS)低下をもたらす。

Mondani が指摘しているリソースからの待ちによるブロック状態に加え、ウェブ・アプリケーションに対し近年更に負荷をかけているのが Ajax の類であろう。よりリッチなユーザ体験を提供しようとして Ajax を使用するウェブ・アプリケーションが増えてきている。Ajax アプリケーションのユーザはページ毎のモデル(page-by-page model)より

も遥かに頻繁にウェブ・サーバに関わり合う。通常のクライアントの要求と違って、Ajax の要求はひとつのクライアントから頻繁に送出される。加えて、そのクライアント・サイドで走っているスクリプトとクライアントの双方が更新の為に定常的にウェブ・サーバにポーリングをかける。それが同時要求数を増やし、スレッドの消費が増大する。

13.1.2 非同期処理

下図も同じプレゼンテーションに合った比較図であるが、この例では 2 つのウェブ・サービスを呼ぶアプリケーションがあったとしている。、サーバ側はクライアントから HTTP 要求が到来し、2 つのウェブ・サービスを待ち、HTTP 応答を返すまでずっとそのスレッドが占有され、その間ブロックされている。これに対し非同期処理では、ウェブ・サービス呼び出しは並行して行われ、サーバ側のスレッドは 2 つのウェブ・サービスを呼び出した時点でブロックされずに戻ってしまう。2 つのウェブ・サービスからの応答はそれは別のスレッドが処理し、HTTP 応答をクライアントに返している。これにより、前項で示されているスレッド枯渇の問題に対処することが出来る。またより早くクライアントに HTTP 応答を返すことが出来る。即ち全体としてのスループットが大きく改善される。

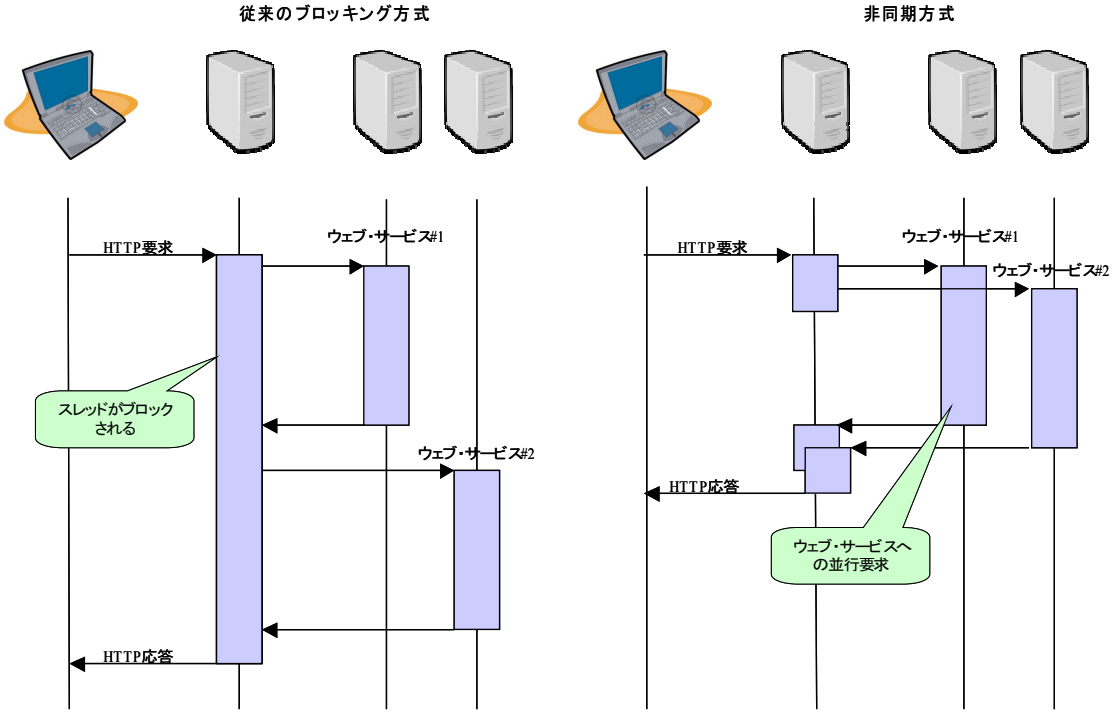


図 13-1：ブロッキング方式と非同期方式の比較

注意：
非同期処理は実は下位層（I/O レベル）で既に導入されている。その為に用意されているのが java.nio パッケージである。ネットワーク層でも TCP 接続毎にスレッドを用意すると、大きなシステムでは同じようにスレッド数が増大してスレッド枯渇、メモリ・リソース圧迫、及びそれによるサービス品質劣化の問題が発生する。下位層での非同期処理に関しては、当社のサイトの「Java による TCP プログラミング」というチュートリアルを見て頂きたい。現在殆どのサーブレット・コンテナ（Tomcat、Jetty、GlassFish (Grizzly)、WebLogic、及び WebSphere など）はこれを採用している。

サーブレット第 3 版ではこのように要求の非同期処理が導入されており、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。非同期処理はサーブレット、及びサーブレット・フィルタに適用できる。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して処理を続ける。

Rajiv Mordani が示している非同期処理のサーブレットのコードを見ると、そのコンセプトの理解が早い:

```
  @WebServlet("/foo" asyncSupported=true)
  public class MyServlet extends HttpServlet {
      public void doGet(HttpServletRequest req, HttpServletResponse res) {

         AsyncContext aCtx = request.startAsync(req, res);
         ScheduledThreadPoolExecutor executor = new ThreadPoolExecutor(10);
         executor.execute(new AsyncWebService(aCtx));
      }
  }

  public class AsyncWebService implements Runnable {
      AsyncContext ctx;
      public AsyncWebService(AsyncContext ctx) {
         this.ctx = ctx;

      public void run() {
         // Invoke web service and save result in request attribute
         // Dispatch the request to render the result to a JSP.
         ctx.dispatch("/render.jsp");
  }
}
```

このアプリケーションでは、このサーブレットは 2 つのウェブ・サービスに対し非同期要求を行い、それの呼び出しが戻ってくるのを待ち、次にその要求をコンテナに戻し、JSP にその結果を結果を処理させている。

- @WebServlet("/foo" asyncSupported=true)アノテーションで、このサーブレットが非同期処理対応であることを宣言している
- 次に AsyncContext aCtx = request.startAsync(req, res);として、要求オブジェクトの非同期開始メソッドである startAsync メソッドを呼ぶことで非同期コンテキストを取得している
- ScheduledThreadPoolExecutor executor = new ThreadPoolExecutor(10);で非同期処理のスレッド executor を用意する
- そのスレッドで、executor.execute(new AsyncWebService(aCtx));と AsyncWebService のオブジェクトを実行する
- doGet メソッドはそこで終了するので、要求スレッドはこの時点でコンテナのメインのスレッド・プールに戻ることになる
- AsyncWebService は非同期ハンドラで、スレッド・クラスであり、そのコンストラクタで非同期コンテキストを取得している
- 非同期処理は run メソッドの中で行われ、まずウェブ・サービスを呼び出し、その結果を要求の属性としてストアし、次に dispatch メソッドでその要求をコンテナが用意するスレッドで"/render.jsp"という JSP にディスパッチさせて応答への書き出しをさせている
- クライアントに返す応答を生成しているのがこの JSP になる。この dispatch された JSP を実行したスレッドは、コンテナに戻る

このコードにもあるように、@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。

13.1.3 より具体的なコード例

同じ Oracle の技術者である Arun Gupta (これもインド人) が示しているサンプル・コードを少し加工したものを紹介する。変更したのは System.out の代わりにロガーを使ったことだけである。ロガーを使うとタイムスタンプが付くこと、及びどのスレッドの出力かがはっきりするからである。このサーブレットは Rajiv Mordan のコードと似ているので、理解し易い。このコードでは非同期リスナ(AsyncListener)が付加されているので、それの使い方を知るのに都合が良い。非同期リスナの onComplete"メソッドは、非同期処理中に生成したリソースのクリーンアップに使用する。

```
package async_processing;

import java.io.IOException;
import java.util.concurrent.ScheduledThreadPoolExecutor;

import javax.servlet.AsyncContext;
import javax.servlet.AsyncEvent;
import javax.servlet.AsyncListener;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


@WebServlet(name="AsyncServlet", urlPatterns={"/AsyncServlet"}, asyncSupported=true)
public class AsyncServlet extends HttpServlet {

   private static final long serialVersionUID = 1L;
   private final static Logger LOGGER = LoggerFactory.getLogger(AsyncServlet.class);
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
      response.setContentType("text/html;charset=Windows-31J");
      try {
         LOGGER.info("doGet を開始");
         AsyncContext ac = request.startAsync();
         ac.addListener(new AsyncListener() {

            @Override
            public void onComplete(AsyncEvent event) throws IOException {
               LOGGER.info("非同期リスナの onComplete を開始");
            }

            @Override
            public void onTimeout(AsyncEvent event) throws IOException {
               LOGGER.info("非同期リスナの onTimeout を開始");
            }

            @Override
            public void onError(AsyncEvent event) throws IOException {
               LOGGER.info("非同期リスナの onError を開始");
            }

            @Override
            public void onStartAsync(AsyncEvent event) throws IOException {
```

```
                LOGGER.info("非同期リスナの onStartAsync を開始");
            }
        });

        LOGGER.info("doGet 内で何かをここで行う ...");


        // Start another service
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(10);
        executor.execute(new MyAsyncService(ac));

        LOGGER.info("doGet 内で非同期ハンドラを呼んだあと何かをここで行う ...");
     } finally {
     }
  }

  class MyAsyncService implements Runnable {
     AsyncContext ac;

     public MyAsyncService(AsyncContext ac) {
        this.ac = ac;
        LOGGER.info("非同期ハンドラの\"MyAsyncService\"がインスタンス化された");
     }

     @Override
     public void run() {
        LOGGER.info("\"MyAsyncService\"内の run を実行");
        ac.complete();
     }
  }
}
```

このサーブレットを http://localhost:8080/tutorial/AsyncServlet で呼び出すと、次のような 3 つのスレッド（オリジナルの要求スレッド、このサーブレットが作った非同期ハンドラの為のスレッド、及び非同期リスナの為にコンテナが用意したスレッド）からのログ出力が表示される:

```
64836 [http-8080-exec-5] INFO async_processing.AsyncServlet - doGet を開始
64836 [http-8080-exec-5] INFO async_processing.AsyncServlet - doGet 内で何かをここで行う ...
64836 [http-8080-exec-5] INFO async_processing.AsyncServlet - 非同期ハンドラの"MyAsyncService"がインスタンス化された
64836 [http-8080-exec-5] INFO async_processing.AsyncServlet - doGet 内で非同期ハンドラを呼んだあと何かをここで行う ...
64836 [pool-2-thread-1] INFO async_processing.AsyncServlet - "MyAsyncService"内の run を実行
64836 [http-8080-exec-6] INFO async_processing.AsyncServlet - 非同期リスナの onComplete を開始
```

ログ出力順にこれを説明すると:

1. doGet メソッドの実行を開始した
2. 非同期モードを開始した後 doGet 内で処理を継続
3. 非同期ハンドラの MyAsyncService のコンストラクタが呼ばれた
4. 非同期ハンドラの MyAsyncService を自分が用意したスレッド・プールからのスレッドで開始させた後で doGet 内での処理を行う
5. 非同期ハンドラの MyAsyncService が開始し、非同期要求が処理されている
6. 非同期ハンドラの処理が終了した

つぎにこの非同期サーブレットに JSP へのディスパッチをさせてみよう。run()メソッドの中の ac.complete();のステートメントの代わりに、

```
ac.dispatch("/DispatchedJsp.jsp");
```

を挿入する。ここに DispatchedJsp.jsp のコードは次のようになる。このファイルは tutorial \ WebContent のディレクトリに置く:

```
<%@ page language="java" contentType="text/html; charset=windows-31j"
   pageEncoding="windows-31j"%>
<%@page import="org.slf4j.Logger" %>
<%@page import="org.slf4j.LoggerFactory" %>
<%!private final static Logger LOGGER = LoggerFactory.getLogger("DispatchedJsp"); %>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-31j">
<title>ディスパッチされた JSP</title>
</head>
<body>
<H1>ディスパッチ・ターゲットの JSP</H1>
</body>
</html>
<% LOGGER.info("JSP の出力終了"); %>
```

同じようにこのサーブレットを http://localhost:8080/tutorial/AsyncServlet で呼び出すと、以下のようなログ出力となる(これは DOS 上での Tomcat 7.0.6 の実行例) :

```
70213 ["http-apr-8080"-exec-7] INFO async_processing.AsyncServlet - doGet を開始
70219 ["http-apr-8080"-exec-7] INFO async_processing.AsyncServlet - doGet 内で何かをここで行う ･･･
70223 ["http-apr-8080"-exec-7] INFO async_processing.AsyncServlet - 非同期ハンドラの"MyAsyncService"が
インスタンス化された
70226 ["http-apr-8080"-exec-7] INFO async_processing.AsyncServlet - doGet 内で非同期ハンドラを呼んだあと
何かをここで行う ･･･
70226 [pool-1-thread-1] INFO async_processing.AsyncServlet - "MyAsyncService"内の run を実行
70238 ["http-apr-8080"-exec-8] INFO DispatchedJsp - JSP の出力終了
```

JSP を実行するのはコンテナが用意したスレッドであることが理解されよう。コンテナは自分が非同期ディスパッチしたスレッドが戻ってくるのでその終了が判り、**明示的に complete を呼ぶ必要は無い**(暗示的 complete)。但し**この場合は onComplete イベントは通知されない**。

13.1.4 一般的な非同期処理のシーケンス

以下に一般的な処理のシーケンスを図示する:

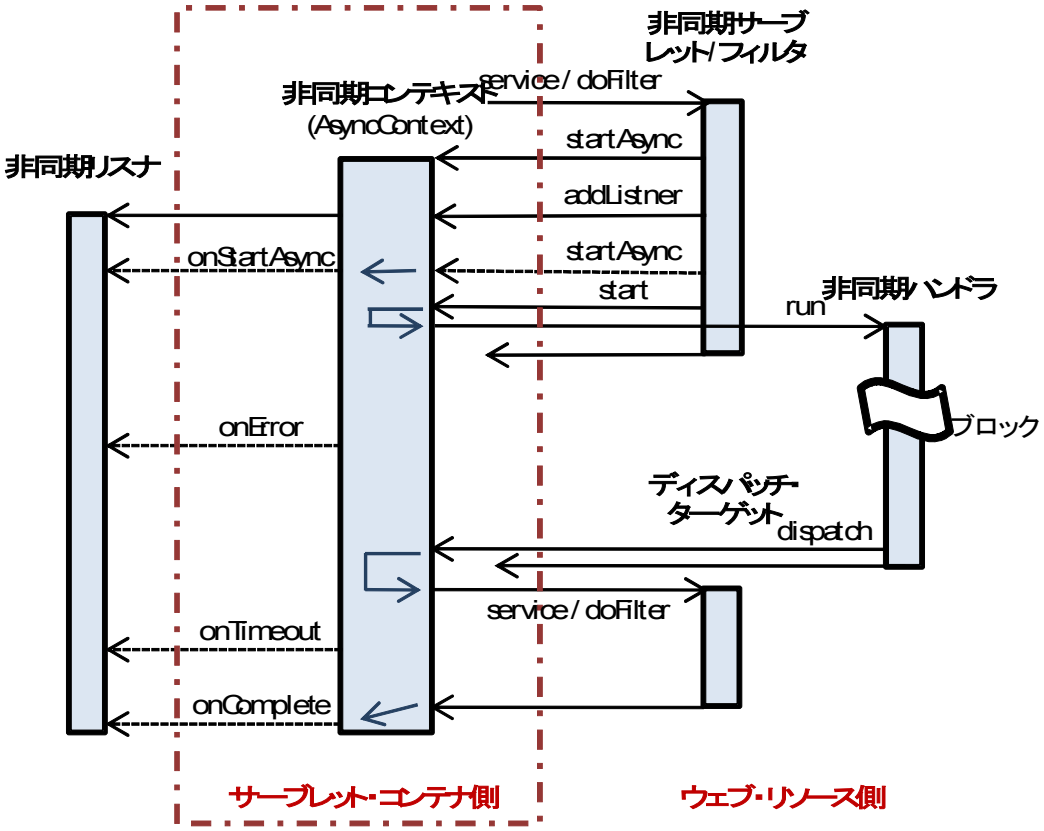


図 13-2：一般的な非同期動作のシーケンス

1. 非同期サーブレット（あるいは非同期サーブレット・フィルタ）は、startAsync メソッドを使って自分の非同期コンテキスト(AsyncContext)を初期化し、要求と応答のオブジェクトを渡す。
2. 必要ならこの非同期コンテキストに addListner メソッドで非同期リスナを付加する。
3. startAsync メソッドで AsyncContext が更に再初期化されることは認められている。
4. 非同期サーブレットは start メソッドを呼ぶことで Runnnable インターフェイスを実装した非同期ハンドラを開始させる。そのときはコンテナは非同期ハンドラを実行させるためのスレッドを用意するが、非同期サーブレット側でスレッド・プールを用意して直接非同期ハンドラを実行させても良い。
5. 非同期ハンドラは通常は他のリソースからのイベント待ちでブロックされる。このハンドラが必要な情報を取得したら、その情報を要求の属性としてストアして、例えば JSP 経由でクライアントに渡す為に、dispatch を呼ぶ。
6. 非同期ハンドラでクライアント向けの HTML データを作ってしまう（要求と応答のオブジェクトは非同期コンテキストから取得できる）場合は、ディスパッチの必要がないので、その場合は最後に complete メソッドでこれをコンテナに知らせる。
7. complete が呼ばれるとコンテナは非同期処理を完了させ、応答のコミットとクローズする。
8. ディスパッチさせるときは、コンテナは自分が用意したスレッドでディスパッチ・ターゲットを実行する。ディスパッチ・ターゲットの実行が終了したら、そのスレッドが戻ってくるので、complete がよばれたと同じ処理を行う。
9. 非同期リスナには、非同期コンテキストの初期化、タイムアウトの発生、エラーの発生、及び非同期動作の完了のイベントが通知される。

13.2節　非同期動作の為のクラスとメソッドたち

非同期処理の為のAPIの翻訳は、参考資料の「非同期処理関係」の項にあるので、詳細はそちらを参照されたい。

13.2.1　AsyncContextクラス

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(この要求に対する非同期処理実効環境)を表現している。AsyncContextオブジェクトは要求オブジェクト毎に対して生成されることになるから、上記のようにServletRequest#startAsyncを要求と応答のオブジェクトを引数にして呼ぶことで初期化されるようになっている。

AsyncContextのインスタンスは、startAsyncメソッドで指定された要求と応答のオブジェクトを保持する。これをディスパッチ・ターゲットが使うことになる。応答はcompleteが呼ばれるまではクローズされなくなる。

このクラスには、それを使って保持している要求と応答のオブジェクトにアクセスできる多くのメソッドが用意されている。例えば、AsyncContext#dispatch()、AsyncContext#dispatch(path)、あるいはAsyncContext#dispatch(servletContext, path)メソッドを使ってその要求をコンテナにディスパッチできる。これらのディスパッチ・メソッドのどれかを使うことで、その要求で開始した非同期動作は、それが完了した後(例えば、あるウェブ・サービスへの呼び出しの戻りを待った後)でその処理を該コンテナに戻すことが出来る。これらのディスパッチ・メソッドたちはその要求をディスパッチしてコンテナに戻すので、コンテナは自分が用意するスレッドを使ってJSPのようなフレームワークで応答を生成させることが出来る。このディスパッチという概念はわかり難いかもしれない。これはもとの原案ではjavax.servlet.AsyncContext#forwardだったが、非同期処理の汎用性と柔軟性を高めるため改名と拡張がされている。

- 引数を持たない**dispatch()**メソッドでは、その要求をオリジナルのもとのURLにフォワードする。また、非同期コンテキストが初期化された後でAsyncContext#dispatchまたはRequestDispatcher#forward呼び出しが生じたときは、このdispatch()メソッドはこの要求をこのAsyncContextのパス、あるいはRequestDispatcher関連要求むけのパスにフォワードする。
- **dispatch(path)**メソッドは、その要求のコンテキストに相対的なパスにこの要求をフォワードする。
- **dispatch(ServletContext, path)**メソッドは指定したコンテキストに相対的なパスにこの要求をフォワードする。

  例えば、ctx.dispatch("/render.jsp");の場合は、コンテナにその要求をフォワードして戻している呼び出しで、JSPが応答を生成できる。

- もうひとつのAsycContextのメソッドである**complete()**はこのAsyncContextオブジェクトを初期化するのに使われた要求で開始した非同期動作を完了させる。そのアプリケーションがその後でその要求でstartAsyncを呼ぶことなしでフォワード・メソッドを使ってコンテナにその要求をディスパッチして戻す場合は、暗示的にこのメソッドを呼ぶことが出来る。

13.2.2　アノテーション属性

サーブレット3.0では、ウェブ・アプリケーション内のサーブレット関連設定のための配備記述子の代替物としてのアノテーション使用を導入している。これにはサーブレット指定用の@WebServletとサーブレット指定用の@WebFilterがある。これらのアノテーション双方に**asyncSupported**という属性が存在する。このasyncSupported属性をtrueにセットすると、このサーブレットまたはサーブレット・フィルタが非同期処理に対応し

ていることを宣言することになる。例えば、次のアノテーションではあるウェブ・アプリケーションの中のサーブレットを定義し、そのサーブレットは非同期処理対応であることを宣言している:

```
@WebServlet(url="/foo" asyncSupported=true)
```

この asyncSupported 属性はある非同期コンテキスト内で使う為に書かれたものと同期処理の為に書かれたコードとを区別する為に必要とされる。あるアプリケーションが非同期機能を使うには、要求処理チェイン全体がアノテーションまたは配備記述子でこの属性をセットしなければならない。あるアプリケーションが非同期動作を開始させようとし、その要求処理チェインの中に非同期処理に対応しないサーブレットまたはサーブレット・フィルタが存在する場合は、IllegalStateException がスローされる。

13.2.3 ServletRequest のなかのメソッドたち

非同期処理をサポートする為に、**startAsync(servletRequest, servletResponse)**、**startAsync()**、及び**getAsyncContext()**といった ServletRequest のメソッドたちが新しく追加されている。非同期処理を対応する為に要求処理チェインの中に asyncSupported 属性をセットしたら、非同期要求をする為に startAsync(servletRequest, servletResponse)あるいは startAsync()メソッドを呼ぶことになる。

例えば非同期要求をする為に次のような呼び出しを行う:

```
AsyncContext aCtx = req.startAsync(req, res);
```

2 つの startAsync メソッドの相違は、startAsync()のほうはオリジナルの要求と応答を暗示的に使うが、startAsync(servletRequest, servletResponse)メソッドのほうはこの呼び出しで渡された要求と応答のオブジェクトを使用する。この呼び出しで渡された要求と応答は、この要求処理チェイン内のフィルタあるいは他のサーブレットたちによってラップされているかともあり得る。この startAsync メソッドは AsyncContext オブジェクトを返すことに注意されたい。この AsyncContext オブジェクトは使われているメソッドによって適切な要求と応答のオブジェクトによって初期化される。**応答をラップして引数なしの startAsync()メソッドを呼ばないよう注意が必要**である。そのときはラップされた応答に書かれたいたデータがあったらそれは失われてしまい、応答ストリームにフラッシュされないことになる。

ServletRequest には非同期処理の為のメソッドとして更に幾つかのメソッドがある。**AsyncSupported()**及び**isAsyncStarted()**は、アプリケーションの中で非同期処理対応になっているか、あるいはある要求でスタートしているかを判断するのに使える。

13.2.4 非同期リスナ

サーブレット 3.0 では **AsyncListner** という新しいリスナ・クラスが付加されている。このクラスを使ってあるアプリケーションが非同期処理が完了したとき、タイムアウトが発生したとき、エラーが発生したとき、あるいは startAsync 呼び出しがそのあとで発生したときに通知を受けるようにできる。以下のコードは新しい AsyncListner オブジェクトを生成し、非同期処理が完了したときに通知を受ける例である:

```
AsyncContext ac = req.startAsync();
req.addAsyncListener(new AsyncListener() {
  public void onComplete(AsyncEvent event) throws IOException {
    ...
  }
...
```

```
    }
```

onComplete メソッドで渡されるパラメタは **AsyncEvent** オブジェクトであることに注意のこと。この AsyncEvent クラスも非同期処理の為にサーブレット 3.0 で新たに追加されたものである。これは非同期処理中に ServletRequest が完了した、タイムアウトした、あるいはエラーが発生した際に生起されるイベントを代表している。

13.3節　非同期サーブレットのサンプル

非同期サーブレットをより理解するには、既にネット上で公開されているサンプルを参考にすると良いだろう。一般的な用途は Ajax のロング・ポールやストリーミング対応（即ちプッシュ方式）のアプリケーションと云うことになろう。ここでは、ロング・ポールとストリーミングの為のサンプルをひとつずつ紹介する。

下図は Oracle の Jeanfrancois Arcand と Ted Goddard, Ph.D.のプレゼンテーションにあった Ajax でのポーリング方式とプッシュ方式の比較図である：

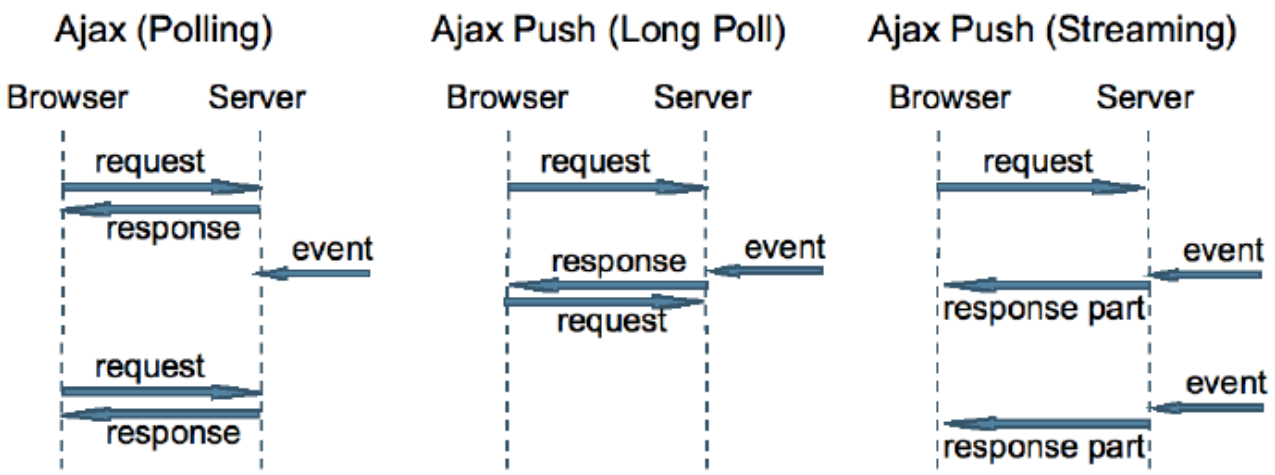


図 13-3：Ajax でのポーリングとプッシュ

- ポーリング方式：
  - 定期的にサーバに要求を送出する
  - 更新するものがなければ空の応答が返される
- ロング・ポール方式：
  - サーバに要求を送出し、サーバはイベント発生を待ち、その後応答を返す
  - 応答が空になることは無い
  - HTTP 仕様が満足されている：「遅い」サーバとの区別が出来ない
- HTTP ストリーミング方式：
  - サーバに要求を送出し、サーバはイベント発生を待ち、multi-part/chunked の応答をストリームし、次にイベント発生を待つ
  - 応答は次々と付加されてゆく

プッシュ方式では TCP 接続は維持されたままになる。更にストリーミングでは応答メッセージもクローズされないで、チャンク方式で次々とボディ部が送信される。

13.3.1 ロング・ポール：株価問合せのサーブレット

金融機関に勤めているポーランド人のソフトウエア技術者の Artur Karaźniewicz が自分のブログに出した株価表示の簡単なサンプルを紹介しよう。これはロング・ポール方式であり、これを参考にすれば、株価、為替、あるいは天気予報などのウェブ・サービスを自分のアプリケーションに取り込み、ダイナミックにそれをクライアントの画面に表示させることなどのアプリケーションの開発が可能になろう。

http://localhost:8080/tutorial/StockQuote.html をアクセスすると、このウィジェットが表示される。ここでは会社シンボル（ここでは GOOG）とその株価が表示され、株価は 5 秒毎に更新させるというものである。

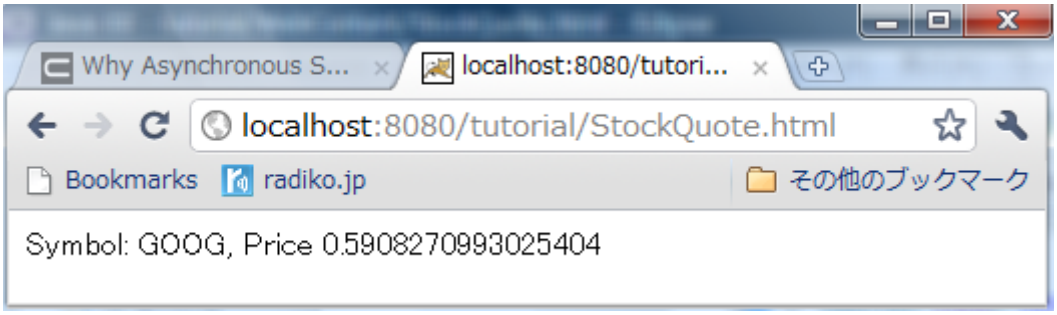


図 13-3：株価表示ウィジェット

この HTML ファイルは次のようになっている：

StockQuote.html

```
001 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
002 <html>
003 <head>
004 <script type="text/javascript" src="http://www.google.com/jsapi"></script>
005 <script type="text/javascript">google.load("jquery", "1.3.2");</script>
006
007 <script type="text/javascript">
008 $(document).ready(longPoll);
009 function longPoll() {
010 $.get("/tutorial/longPollingStock", {}, function ( data, status ) {
011 $("#quoteWidget").html(data);
012 longPoll();
013 } );
014 }
015 </script>
016
017 </head>
018 <body>
019 <div id="quoteWidget"></div>
020 </body>
021 </html>
```

jQuery を使ったこの HTML テキストの 9-14 行目のコードは判りにくいかもしれない。longPoll 関数は/longPollingStock のサーブレットに GET 要求を送信し、その応答（新しい株価）を待つ。その応答が到来したら、このウィジェット(quoteWidget div)を更新して、次のポールを開始する。これをこのページが表示されている間ずっと繰り返す。Long Polling とはこのように、サーバがイベントが発生するまで応答を返すのを保留して、クライアント側は応答が返ると同時に再度要求を送信する方式のことを言う。10 行目の jQuery.get(url, data, callback) のメソッドは、url に data をクエリとして GET 要求をし、その結果を callback 関数で処理するものである。11 行目は HTML の data で div 要素の中身を更新する。

このアプリケーションは次の 4 つのクラスで構成されている：

- MockStockQuoteService

株価問合せサービスのモックアップ

- StockQuote インターフェイス
  株価問合せの為のメソッドを定義
- QuoteObserver
  java.util.Observer を実装、イベントを受けクライアント向けの応答を作成したら非同期コンテキストの complete を呼ぶ
- LongPollingStockServlet サーブレット
  クライアントからの GET 要求を受けたら、非同期処理で 5 秒後に MockStockQuoteService に QuoteObserver を付加する

13.3.1.1 MockStockQuoteService と StockQuote

これはモックアップであって、実際はウェブ・サービスから株価情報を受けるコードを書かねばならない。ここでは "GOOG"という会社シンボルと乱数を返すだけの簡単なものである。これは Runnable を実装し、また Observable を継承している。run のなかでオブザーバにデータ取得が可能であることを通知している。

```
001 package stockquote;
002
003 import java.util.Observable;
004 import java.util.Random;
005
006 public class MockStockQuoteService extends Observable implements StockQuote, Runnable {
007
008   public void run() {
009      // System.out.println("MockStockQuoteService: start");
010      setChanged();
011      notifyObservers();
012   }
013   public String getSymbol() {
014      return "GOOG";
015   }
016
017   public Double getValue() {
018      return Math.abs(new Random().nextDouble());
019   }
020 }
```

StockQuote はその為のメソッドを定義しているインターフェイスである:

```
001 package stockquote;
002
003 public interface StockQuote {
004   public String getSymbol();
005   public Double getValue();
006 }
```

13.3.1.2 QuoteObserver

これは Observer インターフェイスを実装した非同期ハンドラのクラスである。

- コンストラクタのなかで非同期コンテキストをセーブする
- 通知を受けて呼び出される update メソッドが実際の応答オブジェクトへの株価データ書き込みになる
- 書き込みが終わったらオブザーバを削除し、非同期コンテキストの complete を呼び出す

```
001 package stockquote;
002
003 import java.io.IOException;
004 import java.io.Writer;
005 import java.util.Observable;
006 import java.util.Observer;
```

```
007
008 import javax.servlet.AsyncContext;
009
010 public class QuoteObserver implements Observer {
011   private AsyncContext ctx;
012
013   public QuoteObserver(AsyncContext ctx) {
014      this.ctx = ctx;
015   }
016   public void update(Observable observable, Object arg) {
017      // System.out.println("QuoteObserver: update");
018      StockQuote quote = (StockQuote) observable;
019      try {
020         ctx.getResponse().setContentType("text/html");
021         Writer writer = ctx.getResponse().getWriter();
022         writer.write("<span>Symbol: " + quote.getSymbol() + ", Price " + quote.getValue() +
"</span>");
023         observable.deleteObserver(this);
024         ctx.complete();
025      } catch (IOException e) { /* ignore */ }
026   }
027 }
028
```

13.3.1.3 LongPollingStockServlet

このサーブレットが jQuery からの Long Polling による GET 要求を受け付ける。この要求方式では、サーバがイベントが発生するまで応答を返すのを保留して、クライアント側は応答が返ると同時に再度要求を送信する。サーバ側はイベントが発生するまでの間にスレッドをブロックすることになり、この非同期処理がスレッド増大対策として有効な手段となる。

- 15 行目：このサーブレットの URL パタンは/longPollingStock で、非同期処理対応であることを宣言
- 18 行目：株価問合せサービスの stockQuoteServise のインスタンスを取得
- 19 行目：ScheduledThreadPoolExecutor 型のワーカを用意
- 20 行目：このワーカを 5 秒後に開始させるスケジュールを行う（実行は stockQuoteService、初期ディレイは 0、ロング・ディレイは 5 秒）
- 27 行目：startAsync メソッドで非同期コンテキストを取得
- 28 行目：stockQuoteServise にオブザーバを付加
- 29 行目：要求スレッドはここでコンテナに戻る

```
001 package stockquote;
002
003 import java.io.IOException;
004 import java.util.concurrent.Executors;
005 import java.util.concurrent.ScheduledThreadPoolExecutor;
006 import java.util.concurrent.TimeUnit;
007
008 import javax.servlet.AsyncContext;
009 import javax.servlet.ServletException;
010 import javax.servlet.annotation.WebServlet;
011 import javax.servlet.http.HttpServlet;
012 import javax.servlet.http.HttpServletRequest;
013 import javax.servlet.http.HttpServletResponse;
014
015 @WebServlet(urlPatterns = "/longPollingStock", asyncSupported = true)
016 public class LongPollingStockServlet extends HttpServlet {
017   private static final long serialVersionUID = 1L;
018   static MockStockQuoteService stockQuoteService = new MockStockQuoteService();
019   static ScheduledThreadPoolExecutor worker = (ScheduledThreadPoolExecutor)
Executors.newScheduledThreadPool(1);
020   static {
021      worker.scheduleAtFixedRate(stockQuoteService, 0, 5, TimeUnit.SECONDS);
022   }
023
024   @Override
```

```
025   protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
026      // System.out.println("LongPollingStockServlet: doGet");
027      AsyncContext ctx = req.startAsync(req, resp);
028      stockQuoteService.addObserver(new QuoteObserver(ctx));
029   }
030 }
```

13.3.2 ストリーミング：Java EE 6 SDK のチャットのサンプル・アプリケーション

Rajiv Mordani のメモにはサンプルとしてチャットのアプリケーションが示されている。これはストリーミング方式のプッシュを採用している例である。このアプリケーションは Glassfish 用に書かれているが、Tomcat でも async_request というプロジェクトとしてインポートし、application.js という JavaScript の一行だけを下記のように変更すれば走らせることが出来る。即ち URL を自分が設定したコンテキスト名に変更する。(Eclipse 用の WAR ファイルはここからダウンロード出来る)

```
var app = {
   url: '/async_request/chat',
   initialize: function() {
      $('login-name').focus();
      app.listen();
   },
```

Eclipse 上ではこのアプリケーションは次のようになる：

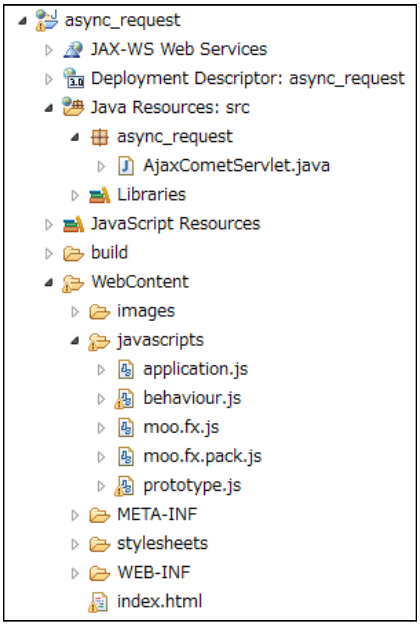


図 13-5：Eclipse 上での async_request の構成

13.3.2.1 実行例

下図はこのアプリケーションを Eclipse 上で走らせた例である：

ふたつのブラウザ（ブラウザ A とブラウザ B とする）から http://localhost:8080/async_request/と指定してこのアプリケーションを開く（index.html が呼ばれる）。

- チャットのエントリ表示の為のテキストエリアと、ユーザ ID 入力の為のテキスト・フィールド、そしてログイ

ンのボタンが表示される

- ブラウザ A でユーザ名（ここでは Obama）を入れ、ログイン・ボタンをクリックすると、以下のようなメッセージがテキスト・エリアに表示される

```
System Message:
Obama has joined.
```

- 同様にブラウザ B でもそのユーザがユーザ名（ここでは Hu Jintao）を入れ、ログイン・ボタンをクリックすると、以下のようなメッセージがテキスト・エリアに表示される

```
System Message:
Hu Jintao has joined.
```

- 双方の画面には、メッセージのポスト用の新しいテキスト・エリアと Post Message というラベル付きボタンが表示される
- これでブラウザ A とブラウザ B の双方のユーザがメッセージのポスト用の新しいテキスト・エリアにメッセージを入力できる
- 入力されたメッセージは、入力ユーザ ID とともに、エントリ表示の為のテキスト・エリアに表示される

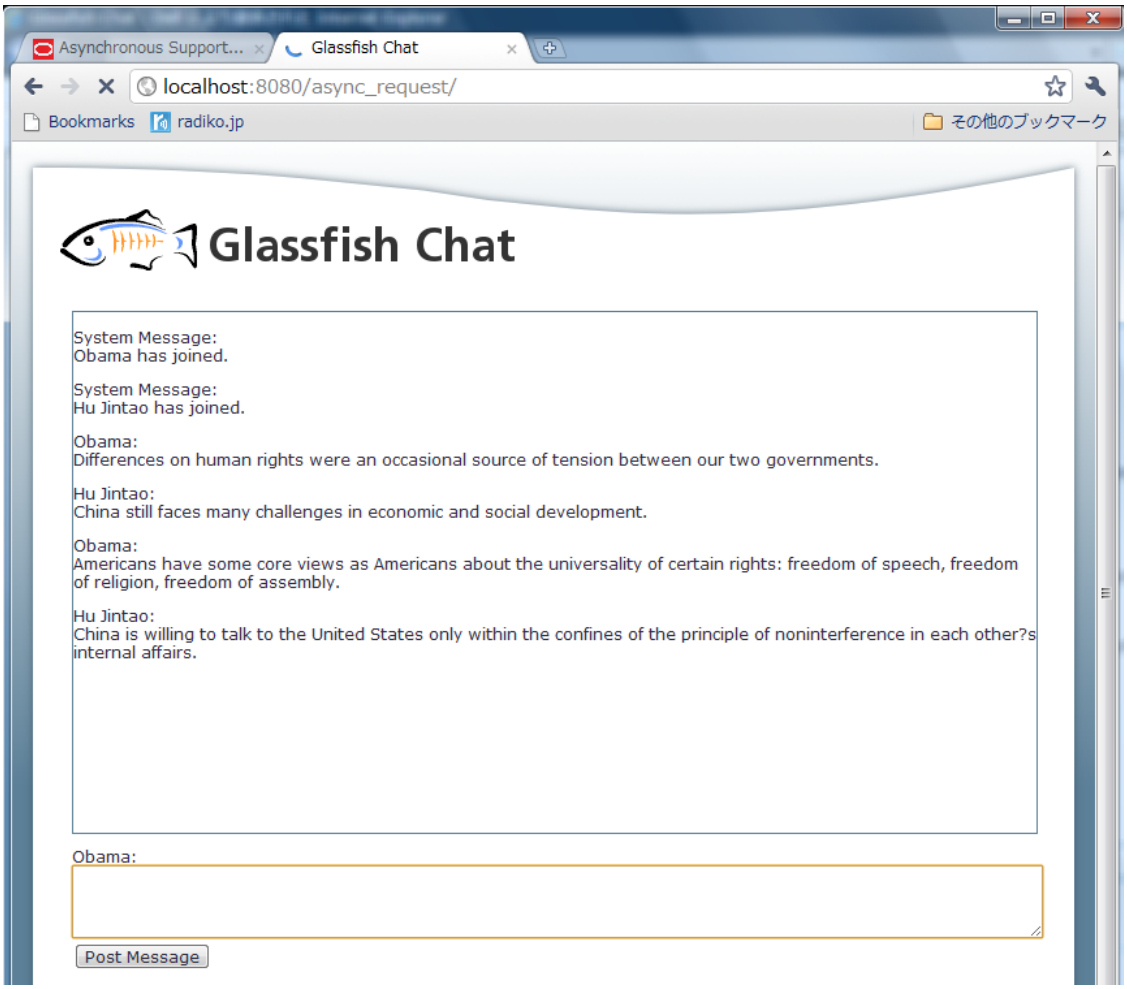


図 13-4：チャットの実行例

13.3.2.2 AjaxCometServlet サーブレット

このサーブレットは Grizzly フレームワーク上の Comet（サーバ・プッシュ型手法）のサンプルをサーブレット 3.0 の API を使うように変更したものである。従って Comet のコンセプトをまず理解する必要がある。Meebo や Gmail チャットなどが Ajax Comet を採用している。

- 80-107 行：init メソッドの中で notifierThread というスレッドを走らせている。このスレッドは、ユーザのエントリ表示用のエリアに誰かがチャットをエントリするごとにログインしているユーザたち、つまり非同期コンテキストの待ち行列にある各非同期コンテキストの応答に出力する。即ちこのスレッドは待ち行列に入っているメッセージをとりだし、また非同期コンテキストの待ち行列からの非同期コンテキストで応答及び PrintWriter を取得して、それにそのメッセージを書き込み flush し、ユーザたちに送信することを繰り返す。
- 110-139 行の doGet メソッドは、ユーザが最初にこのアプリケーションをアクセスしたときに index.html 経由で呼び出される。このメソッドではそのユーザの為の非同期コンテキストを用意し、これを非同期コンテキストの待ち行列に入れる。
- 142-165 行：doPost メソッドは、ユーザが入力用のテキスト・エリアに入力し、submit ボタンをクリックしたときに呼び出される。メッセージ入力のときは 157-159 行にあるように、名前とメッセージを組にして、173 行目からの notify メソッドでメッセージの待ち行列に置いている。つまり処理を notifierThread に任せている。

AjaxCometServlet.java

```
001 package async_request;
002
003 /*
004  * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
005  *
006  * Copyright 1997-2009 Sun Microsystems, Inc. All rights reserved.
007  *
008  * The contents of this file are subject to the terms of either the GNU
009  * General Public License Version 2 only ("GPL") or the Common Development
010  * and Distribution License("CDDL") (collectively, the "License").  You
011  * may not use this file except in compliance with the License. You can obtain
012  * a copy of the License at https://glassfish.dev.java.net/public/CDDL+GPL.html
013  * or glassfish/bootstrap/legal/LICENSE.txt.  See the License for the specific
014  * language governing permissions and limitations under the License.
015  *
016  * When distributing the software, include this License Header Notice in each
017  * file and include the License file at glassfish/bootstrap/legal/LICENSE.txt.
018  * Sun designates this particular file as subject to the "Classpath" exception
019  * as provided by Sun in the GPL Version 2 section of the License file that
020  * accompanied this code.  If applicable, add the following below the License
021  * Header, with the fields enclosed by brackets [] replaced by your own
022  * identifying information: "Portions Copyrighted [year]
023  * [name of copyright owner]"
024  *
025  * Contributor(s):
026  *
027  * If you wish your version of this file to be governed by only the CDDL or
028  * only the GPL Version 2, indicate your decision by adding "[Contributor]
029  * elects to include this software in this distribution under the [CDDL or GPL
030  * Version 2] license."  If you don't indicate a single choice of license, a
031  * recipient has the option to distribute your version of this file under
032  * either the CDDL, the GPL Version 2 or to extend the choice of license to
033  * its licensees as provided above.  However, if you add GPL Version 2 code
034  * and therefore, elected the GPL Version 2 license, then the option applies
035  * only if the new code is made subject to such option by the copyright
036  * holder.
037  */
038
039
040 import java.io.IOException;
041 import java.io.PrintWriter;
042 import java.util.Queue;
043 import java.util.concurrent.ConcurrentLinkedQueue;
044 import java.util.concurrent.BlockingQueue;
045 import java.util.concurrent.LinkedBlockingQueue;
046
047 import javax.servlet.AsyncContext;
048 import javax.servlet.AsyncEvent;
049 import javax.servlet.AsyncListener;
050 import javax.servlet.ServletConfig;
051 import javax.servlet.ServletException;
052 import javax.servlet.annotation.WebServlet;
053 import javax.servlet.http.HttpServlet;
054 import javax.servlet.http.HttpServletRequest;
```

```
055 import javax.servlet.http.HttpServletResponse;
056
057 /**
058  * This class illustrates the usage of Servlet 3.0 asynchronization APIs.
059  * It is ported from Grizzly Comet sample and use Servlet 3.0 API here.
060  *
061  * @author Shing Wai Chan
062  * @author JeanFrancois Arcand
063  */
064 @WebServlet(urlPatterns = {"/chat"}, asyncSupported = true)
065 public class AjaxCometServlet extends HttpServlet {
066
067     private static final Queue<AsyncContext> queue = new ConcurrentLinkedQueue<AsyncContext>();
068
069     private static final BlockingQueue<String> messageQueue = new
LinkedBlockingQueue<String>();
070
071     private static final String BEGIN_SCRIPT_TAG = "<script type='text/javascript'>\n";
072
073     private static final String END_SCRIPT_TAG = "</script>\n";
074
075     private static final long serialVersionUID = -2919167206889576860L;
076
077     private Thread notifierThread = null;
078
079     @Override
080     public void init(ServletConfig config) throws ServletException {
081         Runnable notifierRunnable = new Runnable() {
082             public void run() {
083                 boolean done = false;
084                 while (!done) {
085                     String cMessage = null;
086                     try {
087                         cMessage = messageQueue.take();
088                         for (AsyncContext ac : queue) {
089                             try {
090                                 PrintWriter acWriter = ac.getResponse().getWriter();
091                                 acWriter.println(cMessage);
092                                 acWriter.flush();
093                             } catch(IOException ex) {
094                                 System.out.println(ex);
095                                 queue.remove(ac);
096                             }
097                         }
098                     } catch(InterruptedException iex) {
099                         done = true;
100                         System.out.println(iex);
101                     }
102                 }
103             }
104         };
105         notifierThread = new Thread(notifierRunnable);
106         notifierThread.start();
107     }
108
109     @Override
110     protected void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
111         res.setContentType("text/html");
112         res.setHeader("Cache-Control", "private");
113         res.setHeader("Pragma", "no-cache");
114
115         PrintWriter writer = res.getWriter();
116         // for IE
117         writer.println("<!-- Comet is a programming technique that enables web servers to send
data to the client without having any need for the client to request it. -->\n");
118         writer.flush();
119
120         final AsyncContext ac = req.startAsync();
121         ac.setTimeout(10  * 60 * 1000);
122         ac.addListener(new AsyncListener() {
123             public void onComplete(AsyncEvent event) throws IOException {
124                 queue.remove(ac);
125             }
126
127             public void onTimeout(AsyncEvent event) throws IOException {
128                 queue.remove(ac);
```

```
129             }
130
131             public void onError(AsyncEvent event) throws IOException {
132                 queue.remove(ac);
133             }
134
135             public void onStartAsync(AsyncEvent event) throws IOException {
136             }
137         });
138         queue.add(ac);
139     }
140
141     @Override
142     protected void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
143         res.setContentType("text/plain");
144         res.setHeader("Cache-Control", "private");
145         res.setHeader("Pragma", "no-cache");
146
147         req.setCharacterEncoding("UTF-8");
148         String action = req.getParameter("action");
149         String name = req.getParameter("name");
150
151         if ("login".equals(action)) {
152             String cMessage = BEGIN_SCRIPT_TAG + toJsonp("System Message", name + " has
joined.") + END_SCRIPT_TAG;
153             notify(cMessage);
154
155             res.getWriter().println("success");
156         } else if ("post".equals(action)) {
157             String message = req.getParameter("message");
158             String cMessage = BEGIN_SCRIPT_TAG + toJsonp(name, message) + END_SCRIPT_TAG;
159             notify(cMessage);
160
161             res.getWriter().println("success");
162         } else {
163             res.sendError(422, "Unprocessable Entity");
164         }
165     }
166
167     @Override
168     public void destroy() {
169         queue.clear();
170         notifierThread.interrupt();
171     }
172
173     private void notify(String cMessage) throws IOException {
174         try {
175             messageQueue.put(cMessage);
176         } catch(Exception ex) {
177             IOException t = new IOException();
178             t.initCause(ex);
179             throw t;
180         }
181     }
182
183     private String escape(String orig) {
184         StringBuffer buffer = new StringBuffer(orig.length());
185
186         for (int i = 0; i < orig.length(); i++) {
187             char c = orig.charAt(i);
188             switch (c) {
189             case '\b':
190                 buffer.append("\\b");
191                 break;
192             case '\f':
193                 buffer.append("\\f");
194                 break;
195             case '\n':
196                 buffer.append("<br />");
197                 break;
198             case '\r':
199                 // ignore
200                 break;
201             case '\t':
202                 buffer.append("\\t");
203                 break;
```

```
204             case '\'':
205                 buffer.append("\\'");
206                 break;
207             case '\"':
208                 buffer.append("\\\"");
209                 break;
210             case '\\':
211                 buffer.append("\\\\");
212                 break;
213             case '<':
214                 buffer.append("&lt;");
215                 break;
216             case '>':
217                 buffer.append("&gt;");
218                 break;
219             case '&':
220                 buffer.append("&amp;");
221                 break;
222             default:
223                 buffer.append(c);
224             }
225         }
226
227         return buffer.toString();
228     }
229
230     private String toJsonp(String name, String message) {
231         return "window.parent.app.update({ name: \"" + escape(name) + "\", message: \"" +
escape(message) + "\" });\n";
232     }
233 }
234
```

第14章　　ファイル・アップロード

写真、ビデオなどのファイルをサーバにアップロードして、友人や親しい人たちとそれを共有するといったインターネットの使い方が普及しているなかで、ウェブ・アプリケーションにそのような機能を持たすことの要求が一般化してきている。これまではファイル・アップロードの機能をアプリケーションに持たせるためには、外部のライブラリ(例えばApache のorg.apache.commons.fileupload パッケージ)を使うか、自分で面倒な入力処理を書くかしかなかった。この問題に対処する為に、サーブレット3.0 ではサーブレット・コンテナ自身がmultipart/form-data つきのHTTP 要求メッセージを構文解析しHttpServletRequest 経由で利用できるようになっている。サーブレット3.0 では汎用的で可搬性(ポータビリティ)がある手段としてのアノテーションとPart というインターフェイスが新たに用意されている。これらの新しいAPI により、フォーム・データ送信(submission)が容易に処理できるようになった。

Multipart/form-data というのはファイル、非 ASCII テキスト、及びバイナリ・データをサーバに送信する為の HTTP プロトコルのコンテント・タイプである。HTML の FORM でこれを送信するには RFC 1867「HTML におけるフォーム・ベースのアップロード」仕様書に規定されている記述を使用する。

- @MultipartConfig
  サーブレット 3.0 仕様にはファイルのアップロード機能が組み込まれている。あるサーブレットにこのアノテーションが指定されたときには、そのサーブレットへの要求は mime/multipart のタイプのものであることを示す。そのサーブレットへの HttpServletRequest オブジェクトは、getParts 及び getPart メソッドでいろんな MIME 添付コンテンツを取得できるようになる。このアノテーションはまたファイルたちをストアする場所、アップロードされるファイルの最大サイズ、最大要求サイズ、及びディスクに書き込む為のサイズの閾値を設定できる。このアノテーションに相当する<multipart-config>要素が配備記述子(web.xml)定義されており、これを使用することも可能である。

- Part
  これは multipart/form-data で送信される HTTP 要求メッセージのボディ部が複数のパート(Part)で構成されている為、それらの Part を代表するインターフェイスである。サーブレットは getPart または getParts でこの Part のオブジェクト(あるいはその Collection)を取得できる。

14.1節　　ブラウザが multipart 送信するときの HTTP 要求メッセージ

HTML から multipart/form-data の送信をする為の一般的な記述は次のようなものである:

FileUploadTest.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-31j">
<title>ファイル・アップロード・テスト</title>
</head>
<body>
<form action="http://localhost:12345/tutorial/HttpRequestDump"
   enctype="multipart/form-data" method="POST"><br>
   What is your name? <input type="text" name="submitter"> <br>
   What files are you sending?<input type="file" name="content"> <br>
   <input type="submit" value="Send File"">
```

```
</form>
</body>
</html>
```

ここで重要なのは
<form action="http://localhost:12345/tutorial/HttpRequestDump" enctype="multipart/form-data" method="POST">というタグ記述である。action では要求の送信先を指定しており、ここではプロキシ経由で HttpRequestDump サーブレットをアクセスするようにしている。また enctype の"multipart/form-data"と method の"POST"は必須である。

Tomcat 7 及びプロキシを開始させてこの HTML を呼ぶと次のような画面が得られる:

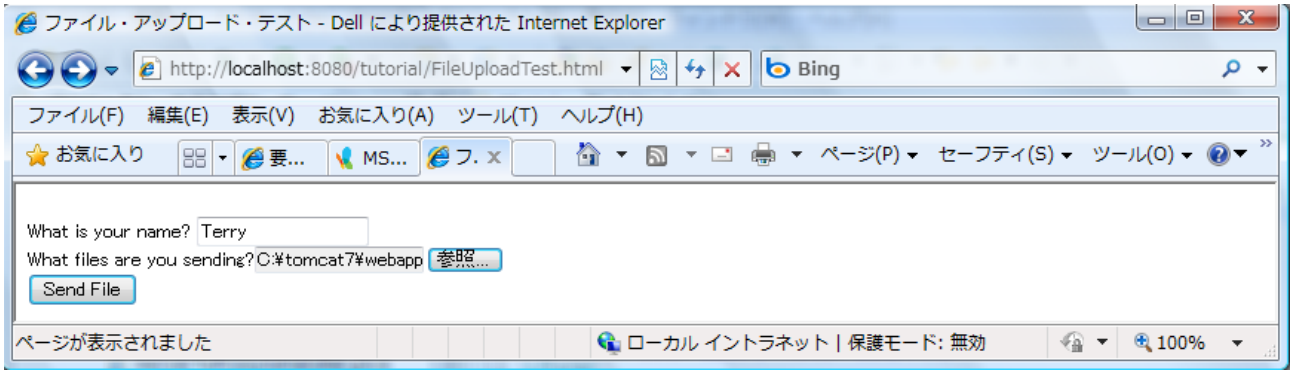


図 14-1:ファイル・アップロードのテスト画面

この状態で Send File ボタンをクリックしたときのブラウザ出力は、このサーブレットの応答とプロキシのログで確認できる。HttpRequestDump 出力の一部を示すと:

```
要求パラメタ(Request Parameters):

要求ヘッダ(Headers):
  accept : image/gif, image/jpeg, image/pjpeg, application/x-ms-application, application/vnd.ms-
xpsdocument, application/xaml+xml, application/x-ms-xbap, application/x-shockwave-flash,
application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, */*
  referer : http://localhost:8080/tutorial/FileUploadTest.html
  accept-language : ja
  user-agent : Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; SLCC1; .NET CLR
2.0.50727; Media Center PC 5.0; MDDC; .NET CLR 3.5.30729; .NET CLR 3.0.30729;
OfficeLiveConnector.1.5; OfficeLivePatch.1.3; .NET4.0C)
  content-type : multipart/form-data; boundary=----------------------------7db3475405c0
  accept-encoding : gzip, deflate
  host : localhost:12345
  content-length : 747
  connection : Keep-Alive
  cache-control : no-cache
```

となっていて、これから次のようなことが判る:

- この要求には要求パラメタは存在していない
- content-type :ヘッダ行では、multipart/form-data 形式であること、バウンダリは”----------7db3475405c0”であることを示している

それではボディ部はどうなっているのだろうか？HttpRequestDump のヘキサ・ダンプを見ても良いが、プロキシのログのほうが見やすいだろう:

Internet Explorer の要求メッセージの内容

```
POST /tutorial/HttpRequestDump HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, application/x-ms-application, application/vnd.ms-
```

```
xpsdocument, application/xaml+xml, application/x-ms-xbap, application/x-shockwave-flash,
application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, */*
Referer: http://localhost:8080/tutorial/FileUploadTest.html
Accept-Language: ja
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; SLCC1; .NET CLR
2.0.50727; Media Center PC 5.0; MDDC; .NET CLR 3.5.30729; .NET CLR 3.0.30729;
OfficeLiveConnector.1.5; OfficeLivePatch.1.3; .NET4.0C)
Content-Type: multipart/form-data; boundary=---------------------------7db3475405c0
Accept-Encoding: gzip, deflate
Host: localhost:12345
Content-Length: 747
Connection: Keep-Alive
Cache-Control: no-cache

-----------------------------7db3475405c0
Content-Disposition: form-data; name="submitter"

Terry
-----------------------------7db3475405c0
Content-Disposition: form-data; name="content"; filename="C:\tomcat7\webapps\tutorial\WEB-
INF\upload_test_file.txt"
Content-Type: text/plain

首相は演説の中で「与野党」を４度も繰り返し、野党との協議を呼びかけた。
特に、社会保障と税の一体改革では「一政治家、そして一政党の代表として、この問題を与野党で協議することを提案します」と２
度にわたって主張。
野党を協議のテーブルに引き込もうという思惑が透ける。
民主党が政権交代前の１９年、福田康夫首相（当時）が呼びかけた「社会保障国民会議」への参加を拒否したことへの反省もなく、
いかにも「ご都合主義」だ。

-----------------------------7db3475405c0--
```

これは C:\tomcat7\webapps\tutorial\WEB-INF\upload_test_file.txt というテキスト・ファイルをサーバに送信したものである。

- 3 つのバウンダリで 2 つのパートが挟まれている。ひとつは送信者名であり、もうひとつは送信ファイルのデータである。これらのパートは input の name によって識別される
- バウンダリの文字列にはその前に"--"が、また最後のバウンダリには前後に"--"が付けられている
- 送信ファイルのデータは、Content-Disposition:、Content-Type:のヘッダ行と、ファイルの中身で構成されている
- Content-Disposition:ヘッダは RFC 2183 で規定されている。Content-Disposition は本来、メールの MIME 仕様に従うデータの提示的情報(presentational information)を伝えるに作られたヘッダであるが、HTTP メッセージの形式が MIME に似ている為、HTML フォームからマルチパート・データをアップロードする際に使用されている。filename 属性を使うと、マルチパート・データとして転送されたデータのファイル名を提示することになる。サーバ側はこの提示されたファイル名を受け入れる必要は無く、自由にファイル名を決定する事ができるが、誤ってデータを上書きしてしまうことの無いよう注意が必要である。
- Content-Type:ヘッダは、このファイルはテキスト・ファイルであるので、text/plain となっている。例えば DLL ファイルだと Content-Type: application/octet-stream となる。MIME タイプの詳細は W3C のリストなどを参照されたい。

14.1.1 ブラウザによる相違

それではブラウザによって送信される HTTP 要求メッセージはどのようになるのだろうか？Internet Explorer ではファイル名がディレクトリも含めて提示されているのに対し、**Apple Safari、Google Chrome、及び Mozilla Firefox の場合は FileUploadTest.html とファイル拡張子付きファイル名だけとなっている**。それ以外の相違はむしろブラウザに表示される入力画面であろう。

Mozilla Firefox の場合

```
POST /tutorial/HttpRequestDump HTTP/1.1
```

```
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja; rv:1.9.2.13) Gecko/20101203 Firefox/3.6.13
( .NET CLR 3.5.30729; .NET4.0C)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://localhost:8080/tutorial/FileUploadTest.html
Content-Type: multipart/form-data; boundary=---------------------------41184676334
Content-Length: 708

-----------------------------41184676334
Content-Disposition: form-data; name="submitter"

Terry
-----------------------------41184676334
Content-Disposition: form-data; name="content"; filename="upload_test_file.txt"
Content-Type: text/plain

首相は演説の中で「与野党」を４度も繰り返し、野党との協議を呼びかけた。
特に、社会保障と税の一体改革では「一政治家、そして一政党の代表として、この問題を与野党で協議することを提案します」と２
度にわたって主張。
野党を協議のテーブルに引き込もうという思惑が透ける。
民主党が政権交代前の１９年、福田康夫首相（当時）が呼びかけた「社会保障国民会議」への参加を拒否したことへの反省もなく、
いかにも「ご都合主義」だ。

-----------------------------41184676334--
```

14.1.2 日本語のパラメタ名とファイル名の扱い

これは一見 URL エンコードが必要になるのではと心配させるかも知れないが、これはボディ部のデータであり、どのブラウザも正しく指定した文字エンコーディング（ここでは Windows-31J）で送信している。従ってメールの場合と違って、HTTP ではこの部分での **URL エンコードの問題は存在しない**。

但しサーブレット側ではデータ（正確には Part のデータ部）をバイトのストリームとして渡すので、これを例えば Windows-31J として文字列に変換する必要がある。

14.2節 新しい API

前述のようにサーブレット3.0 では、ファイル・アップロードの為の API として、@MultipartConfig というアノテーションと、Part というインターフェイスが新しく用意されている。

14.2.1 @MultipartConfig アノテーション

あるサーブレット上でこれが指定されているときは、そのサーブレットが期待している要求が type mime/multipart であることを示す。そのサーブレットは getParts 及び getPart メソッドを使って MIME 添付物を利用できるようになる。@MultipartConfig アノテーションはそれらのファイルがストアできる場所、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、及びディスクへの書き込みを開始させるサイズ閾値を指定できる。

また、これと等価な配備記述子(web.xml)上の<multipart-config>も利用できる。配備記述子とアノテーションとと

もに同じ要素で異なる値が指定されている場合は、配備記述子が優先される。また、配備記述子で metadata-complete 指定されているときは、アノテーションは無効となる(無視される)。

public @interface MultipartConfig に関する詳細は、参考資料の「サーブレット 3.0 の主要クラスのメソッド一覧」の「ファイル・アップロード関係」の項を見て頂きたい。

14.2.2 新しいメソッドとクラス

追加されたメソッドとクラスを下表に示す。詳細は参考資料の「サーブレット 3.0 の主要クラスのメソッド一覧」の「ファイル・アップロード関係」の項を見て頂きたい:

表 14-1:ファイル・アップロードののために追加されたメソッド

メソッド	内容
HttpServletRequest#getParts	Part 型の Collection で、multipart/form-data エンコーディングで送られた Part 要素の総てを取得
HttpServletRequest#getPart	与えられた名前の Part を取得
Part#getInputStream	このパートのコンテントを InputStream として取得
Part#getContentType	このパートのコンテント・タイプを取得
Part#getName	このパートの名前を取得
Part#getSize	このファイルのサイズを返す
Part#write	このアップロードされたアイテムをディスクに書き込むための利便性のためのメソッド
Part#delete	ファイル・アイテムのための蓄積を、それに関わる一時的ディスク・ファイルを含めて削除する
Part#getHeader	指定された MIME ヘッダの値を String として返す
Part#getHeaders	与えられた名前の Part ヘッダの値たちを返す
Part#getHeaderNames	この Part のヘッダ名たちを返す

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“@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要素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。**仕様書の 3.2 節では「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、その要素の文字列値が 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。」と書かれているが、Tomcat 7 では現時点でそうなっていないことに注意されたい。即ち、content-disposition ヘッダを解析して、それらの要素も入力ストリーム経由で読み出さねばならない。**

getParameter / getParameterValues メソッドを使って Content-Disposition ヘッダのなかのパラメタを取得したいときには、BalusC というオランダ人の技術者が書いたメモを参考にされたい。

14.2.3 ファイル・アップロードの為の基本的なコード

これらのアノテーションとメソッドたちを使ったサーブレットの最もシンプルなもの（一部省略されている）は以下のようようなものになる：

```
001 @WebServlet(name = "FileUploadServlet", urlPatterns = {"/FileUploadServlet"})
002 @MultipartConfig(location = "c:/tmp/uploaded/")/
003 public class UploadServlet extends HttpServlet {
004
005     protected void performTask(HttpServletRequest request, HttpServletResponse response)
006             throws ServletException, IOException {
007         response.setContentType("text/html;charset=Windows-31J");
008         PrintWriter out = response.getWriter();
009         try {
010             request.getPart("content").write("hello.jpg");
011             out.println("ファイルをサーバで保管しました。");
012         } finally {
013             out.close();
014         }
015     }
```

- 002 行目の@MultipartConfig で、アップロードされたファイルをストアする為のディレクトリを指定する
- 010 行目で"content”という名前の Part を取得して、その write メソッドで"hello.jpg"というファイル名でアップロードされたファイルを指定したディレクトリに保管している

Part#write というメソッドはこのように入力ストリームを使わなくても良く、非常にシンプルな記述でファイルに保管できるのが特徴である。このメソッドを使わないとしたら、この行に等価なコードは次のようなものになってしまう。

```
001       InputStream is = request.getPart("content").getInputStream();
002       int i = is.available();
003       byte[] b  = new byte[i];
004       is.read(b);
005       FileOutputStream os = new FileOutputStream("c:/temp/uploaded/hello.jpg");
006       os.write(b);
```

ただしこのコードでは、単一のファイルしか受け付けず、またファイル名も固定されているので一般的ではない。従って、実用上は次の節で示すようなアプリケーションが参考になろう。

14.3節 サンプル・アプリケーション

ここでは簡単なアップロードのアプリケーションで、これらの API のより具体的な使い方について説明する。

このアプリケーションは次のように使用する：

1. エクスプローラを使って c:\temp\uploaded というフォルダを用意する
2. Eclipse 上で FileUploadServlet.java を tutorial\Java Resouces src\file_upload のパッケージに含める
3. FileUploadTest.jsp を tutorial\WebContent に含める
4. Tomcat 7 を開始させる
5. 自分のブラウザ（ここでは Google Chrome）から http://localhost:8080/tutorial/FileUploadServlet でこのサーブレットをアクセスする
6. 直ちに FileUploadTest.jsp が呼び出され、次のような画面が表示されるので、サブミッタ名（ここでは岡田

克也)とファイル(ここでは c:\tomcat7\webapps\tutorial\WEB-INF\アップロード・テスト.txt というテキスト・ファイル)を指定する。

図 14-2:FileUploadServlet の初期画面

7. Send File ボタンをクリックすると、指定したファイルが FileUploadServlet 経由で c:\temp\uploaded のフォルダに送られ、下図のような画面が FileUploadTest.jsp 経由で返される。

図 14-3:FileUploadServlet の応答画面

この画面では、サブミッタ名とサーバが受理したファイルがまず表示され、また名前の入力ボックスにもこの名前が表示される。

8. このときに c:\temp\uploaded のフォルダには下図のように送信されたファイルと一時ファイル 2 つが置かれる(その理由は不明)。これらの TMP ファイルは一定時間が経過すれば、Tomcat がこれらを削除する。

名前	更新日時	種類	サイズ
upload_34ebae97_12dc51d57f5__8000_00000000.tmp	2011/01/27 10:36	TMP ファイル	1 KB
upload_34ebae97_12dc51d57f5__8000_00000001.tmp	2011/01/27 10:36	TMP ファイル	1 KB
アップロード・テスト.txt	2011/01/27 10:36	ﾃｷｽﾄ文書	1 KB

図 14-4:サーバにストアされたアップロード・ファイル

9. また、Tomcat のコンソールには次のようなログが表示される:

```
228796 ["http-bio-8080"-exec-7] INFO file_upload.FileUploadServlet - Submitter name: 岡田克也
228796 ["http-bio-8080"-exec-7] INFO file_upload.FileUploadServlet - Source file name:
C:\tomcat7\webapps\tutorial\WEB-INF\アップロード・テスト.txt
228796 ["http-bio-8080"-exec-7] INFO file_upload.FileUploadServlet - Content type: text/plain
228796 ["http-bio-8080"-exec-7] INFO file_upload.FileUploadServlet - Destination file name: アップ
```

これは Internet Explorer の場合であるが、ブラウザからのファイル名からフィル名とファイル拡張子のみを抜き取ってストアするファイル名としていることに注意されたい。

10. Internet Explorer やその他のブラウザでも同じ結果が得られることを各自確認されたい。

14.3.1 FileUploadServlet.java

このサーブレットは GET 要求が来たら直ちにそれを FileUploadTest.jsp にリダイレクトし、POST 要求が来たときは送られてきたファイルをストアし、その結果を FileUploadTest.jsp 経由でクライアントに知らせる。以下はそのコードである:

FileUploadServlet.java

```
001 package file_upload;
002
003 import java.io.IOException;
004 import java.io.InputStream;
005 import java.util.ArrayList;
006
007 import javax.servlet.RequestDispatcher;
008 import javax.servlet.ServletException;
009 import javax.servlet.annotation.MultipartConfig;
010 import javax.servlet.annotation.WebServlet;
011 import javax.servlet.http.HttpServlet;
012 import javax.servlet.http.HttpServletRequest;
013 import javax.servlet.http.HttpServletResponse;
014 import javax.servlet.http.Part;
015
016 import org.slf4j.Logger;
017 import org.slf4j.LoggerFactory;
018
019 @WebServlet(urlPatterns = "/FileUploadServlet")
020 @MultipartConfig(location = "c:/temp/uploaded/")
021 public class FileUploadServlet extends HttpServlet {
022   private static final long serialVersionUID = 1L;
023   private final static Logger LOGGER = LoggerFactory.getLogger(FileUploadServlet.class);
024
025   @Override
026   protected void doPost(HttpServletRequest request,
027         HttpServletResponse response) throws ServletException, IOException {
028      String submitter = null;
029      ArrayList<String> acceptedFiles = new ArrayList<String>();
030      for (Part part : request.getParts()) {
031         String sub = getSubmitter(part);
032         if (sub != null){
033            submitter = sub;
034            LOGGER.info("Submitter name: " + submitter);
035         }
036         else {
037            String fileName = getFileName(part);
038            if (getFileName(part) != null){
039               LOGGER.info("Source file name: " + fileName);
040               LOGGER.info("Content type: " + part.getContentType());
041               String[] dr = fileName.split("\\\\");
042               fileName = dr[dr.length-1];
043               LOGGER.info("Destination file name: " + fileName);
044               part.write(fileName);
045               acceptedFiles.add(fileName);
046            }
047         }
048      }
049      request.setAttribute("submitter", submitter);
050      request.setAttribute("acceptedFiles", acceptedFiles);
051      RequestDispatcher rd = request.getRequestDispatcher("/FileUploadTest.jsp");
052      rd.forward(request, response);
053   }
054
055   private String getSubmitter(Part part) throws IOException{
056      String partHeader = part.getHeader("content-disposition");
057      if (! partHeader.contains("submitter")) return null;
058      else{
059         InputStream is = part.getInputStream();
```

```
060         byte[] b  = new byte[is.available()];
061         is.read(b);
062         return new String(b, "Windows-31J");
063      }
064   }
065
066   private String getFileName(Part part) {
067      String partHeader = part.getHeader("content-disposition");
068      if (partHeader == null)return null;
069      else{
070         for (String cd : partHeader.split(";")) {
071            if (cd.trim().startsWith("filename")) {
072               return cd.substring(cd.indexOf('=') + 1).trim().replace("\"", "");
073            }
074         }
075         return null;
076      }
077   }
078
079   @Override
080   protected void doGet(HttpServletRequest request,
081         HttpServletResponse response) throws ServletException, IOException {
082      response.sendRedirect("/tutorial/FileUploadTest.jsp");
083   }
084
085 }
```

- 020 行目のアノテーションで、このサーブレットはマルチパートの要求を処理すること、及びアップロードされたファイルの置き場は c:/temp/uploaded/であることをサーブレット・コンテナに知らせている。
- 026 行目からの POST 要求処理では、030 行目のループで各パートを調べている。
- 031-035 行目で、そのパートにはサブミッタの名前が含まれているかを調べている。これには 055-064 行目のパートからサブミッタを取り出すメソッドが使われている。
- 次いで 037-047 行はそのパートにファイル名があるかどうかを調べている。これには 066-077 行目の getFileName というメソッドが使われている。もしファイル名が存在していたら、Internet Explorer ではフルパス付きのファイル名となっているので、最後のファイル名とファイル拡張子のみを 041-042 行で取り出している。
- 044 行では、この要求で幾つのファイルを受け付けたかを記録する ArrayList<String>にそのファイル名を追加する。
- 049-050 行目ではこの ArrayList とサブミッタ名を要求の属性としてセットしている。
- 051-052 行ではこれらの情報をセットした要求、及び応答のオブジェクトを FileUploadTest.jsp にフォワードしている。
- 055 行目からの getSubmitter メソッドは、与えられた Part オブジェクトから"submitter”という名前のデータが存在しているかを調べ、あればそれを、なければ null を返している。前述のように、この名前と値のペアは getParameter や getParameterValue といったメソッドで取り出すことは出来ない。従ってバイト入力ストリームから読みだして文字エンコードいることに注意されたい。
- 066 行目からの getFileName メソッドも Content-Disposition:ヘッダの中身を構文解析することになる。この部分は文字エンコードの必要は無い。例えば Internet Explorer では次のようなヘッダを返しているので、
  Content-Disposition: form-data; name="content"; filename="C:\tomcat7\webapps\tutorial\WEB-INF\アップロード・テスト.txt"
  ここから C:\tomcat7\webapps\tutorial\WEB-INF\アップロード・テスト.txt という文字列を 070-072 行目のコードで取り出している。
- このサーブレットはログとして要求に含まれているサブミッタの名前、送信されてきたパート毎のファイル名、送信されてきたパート毎のコンテント・タイプ、及び実際に記録したファイル名を出力している。不要の場合はこれをコメント・アウトすればよい。

14.3.2 FileUploadTest.jsp

この JSP は見づらいものになってしまっているので恐縮だが、基本的にはこの章の最初に示した FileUploadTest.html がベースになっている。

FileUploadTest.jsp

```
001 <%@ page language="java" contentType="text/html; charset=windows-31j"
002   pageEncoding="windows-31j"%>
003 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
004 <%@ page import="java.util.ArrayList"%>
005 <html>
006 <head>
007 <meta http-equiv="Content-Type" content="text/html; charset=windows-31j">
008 <title>ファイル・アップロード・テストの JSP</title>
009 </head>
010 <body>
011 <%
012   String submitter = (String)request.getAttribute("submitter");
013   if (submitter != null){
014      %><br>
015 Submitter :
016 <%= submitter %>
017 <%
018   }
019   ArrayList<String> acceptedFiles = null;
020   if (request.getAttribute("acceptedFiles") != null){
021      acceptedFiles = (ArrayList<String>)request.getAttribute("acceptedFiles");
022      for (int i=0; i<acceptedFiles.size(); i++){
023         %><br>
024 Accepted file :
025 <%= acceptedFiles.get(i).toString() %>
026 <%
027      }
028   }
029 %>
030 <form action="http://localhost:8080/tutorial/FileUploadServlet"
031   enctype="multipart/form-data" method="POST"><br>
032 What is your name? <input type="text" name="submitter"
033   <% if (submitter != null){
034      %> value="<%= submitter%>" <%
035   }
036      %>>
037 <br>
038 What files are you sending?<input type="file" name="content"> <br>
039 <input type="submit" value="Send File""></form>
040 </body>
041 </html>
```

- 012-016 行目で、FileUploadServlet.java で要求オブジェクトの属性として付加されたサブミッタ名を取り出して画面に表示している。
- 021 行目で要求に付加されている ArrayList<String>を取り出している。
- 022-025 行目でサーバが受け取ったファイルを取り出して画面に表示している。
- 032-036 行目の FORM の中では名前を入力するテキスト・ボックスに、サーブレットが渡したサブミッタ名を表示するようにしている。

第15章　　セキュリティ

*この章は、サーブレット3.0 仕様書の第13 章「セキュリティ」がベースになっている。従ってユーザ認証とアクセス認可制御が中心となっているが、後半ではSSL 上のHTTPS などの秘匿・完全性の為のプロトコルに関して取り上げることにしたい。なお、**この章で使う教材のアプリケーションはsecurity.war であり、これまでのtutorial.war ではないことに注意**されたい。このアプリケーションのインストールに関しては、「Eclipse 上でのsecurity アプリケーションのインストール法」の項を読んで頂きたい。*

サーブレット3.0 では以下の事項がセキュリティ関係で強化されている（詳細は別途説明する）：

- アノテーションによるセキュリティ制約設定
  - @ServletSecurity アノテーション
    - RolesAllowed
    - emptyRoleSemantic
    - TransportGuarantee
  - HTTP 全体（@HttpConstraint）だけでなく、HTTP のメソッド毎（@HttpMethodConstraint）に適用できる
  - 配備記述子(web.xml)のセキュリティ設定がアノテーションを凌越する。また metdata-complete はアノテーションを無効にする
  - Java EE 技術対応：@DeclareRoles アノテーション
- プログラム的なコンテナによる認証とログアウトが HttpServletRequest に付加された
  - HttpServletRequest#login(String username, String password)
    - フォーム・ベースのログインの代替
    - アプリケーションがクレデンシャル（認証資格情報）収集を管理
  - HttpServletRequest#authenticate(HttpServletResponse)
    - 認証制約でカバーされていないリソースからアプリケーションがコンテナ仲介認証を開始させる
    - アプリケーションが認証を何時させるかを判断する
  - HttpServletRequest#logout
- クロス・サイト・スクリプティング等に対処する為に、セッション・クッキーに対しては http-only が使われていることは「セッション」の章で説明した。
- SSL セッション ID によるセッション追跡対応

なおウェブ・アプリケーションにおけるセキュリティには、ウェブ・アプリケーションとクライアント（即ちブラウザ）との間のセキュリティだけでなく、後で説明するレルムを保持するデータ・ベースとの間のセキュリティも存在する。また、Tomcat の場合は Apache のようなウェブ・サーバにピギーバックする形態が一般的であり、その場合にはウェブ・サーバがセキュリティを部分的に受け持つ場合もある。従って、このチュートリアルはサーブレット・コンテナを対象にしている為、スタンドアロンのサーブレットコンテナの場合に限定し、またレルムは最も初歩的なメモリ・レルムに限定して解説することとする。より詳細なセキュリティの解説は、Oracle 社の Java EE6 のチュートリアルを参照されたい。

15.1節　　サーブレットにおけるセキュリティ

セキュリティとは、インターネットの如きオープンなネットワークを利用するアプリケーションにおいて、以下の行為

に対する安全性を図るものである:

- *なりすまし*
- *改竄*
- *傍受*

それらの不正安全性を図る為には、以下の機能が必要とされる。

- *認証(Authentication)： なりすまし対策として使われ、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って動作していることを証明するためのメカニズム*
- *認可(Autholization)： 改竄対策として使われる。アクセス制御とも呼ばれ、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、リソースのへの関与を制限するためのメカニズム*
- *データ保全性(Data Integrity)： その情報が通過する間に、第3者によって傍受変更されていないことのメカニズム*
- *機密性(Confidentiality)またはデータのプライバシー： これも傍受対策であり、その情報をアクセスするために、認可されたユーザに対してのみ取得可能であり、伝送の際不信をまねくものとなっていないことを確たるものにするメカニズム。即ち、その情報がそれをアクセスする為に認可されたユーザに対してのみ取得可能とするメカニズム*

サーブレットは以下に示すように、そのなかでの認証と認可のメカニズムを提供している。

15.1.1 認証と認可

Oracle 社の Java EE6 チュートリアルでは、認証と認可は一般的流れを次のように説明している(Oracle の場合はウェブ・サーバとして説明しているが、ここではサーブレット・コンテナに置き換えて記述する)：

1. 初期要求：
   最初にクライアントがメインのアプリケーションの URL を要求する。そのユーザはそのアプリケーション環境に対し認証をとっていないので、そのアプリケーションをサービスしているサーブレット・コンテナはそのことを検出し、このリソースの為のしかるべき認証メカニズムを呼び出す。

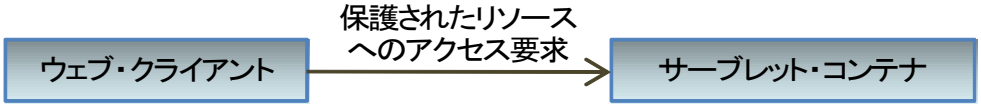


図 15-1：初期要求

2. 初期認証：
   クライアント側がそのユーザから認証の為のデータ(例えばユーザ名とパスワード)を収集するのに使うフォームをサーブレット・コンテナが返す(フォーム・ベースの認証の場合、HTTP ベーシックや HTTP ダイジェスト認証の場合はサーブレット・コンテナはその認証に必要な情報を返す)。そのウェブ・クライアントはその認証データをサーブレット・コンテナに返し、そのサーブレット・コンテナはその認証データを検証する。検証のメカニズムはそのサーブレット・コンテナのにとってローカルなものである場合もあるし、ウェブ・サーバのセキュリティ・サービスを使うこともあろう。この検証がパスすれば、サーブレット・コンテナはクレデンシャル(ユーザ名とパスワードを含む認証資格情報)をそのユーザにセットする。クレデンシャルは通常は最終的にセッションやコンテキストのオブジェクトの属性としてセットされよう。

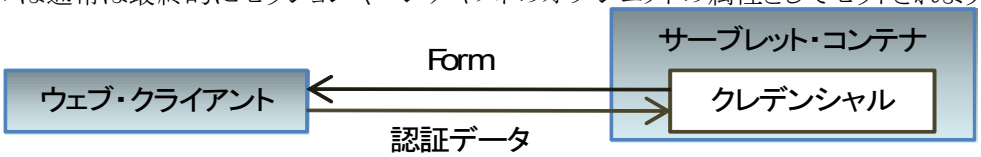


図 15-2：初期認証

3. URL 認可:
このクレデンシャルはそのユーザが今後その制限されたリソースへのアクセスへの認可(権限)を持っているかどうかの判断に使われる。サーブレット・コンテナはそのウェブ・リソースに対するセキュリティのポリシー(配備記述子、またはアノテーションから得られる)を調べ、そのリソースへのアクセスが許可されているセキュリティ・ロールたちを調べる。サーブレット・コンテナは次に各ロールに対しそのユーザのクレデンシャルを調べ、そのユーザがそのロールに含まれているかどうかを判断する。サーブレット・コンテナがそのユーザとあるロールとの対応がとれたら、「認可されている」という結果でそのテストを終了する。許可されているどのロールにもそのユーザが対応していないときには、「認可されていない」という結果に終わる。その場合は 401 エラー・ページが応答として返される。

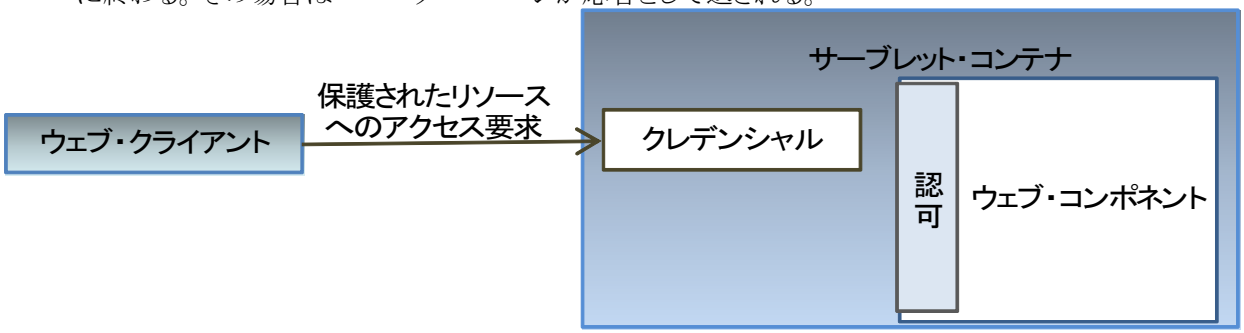


図 15-3:URL 認可

4. オリジナルの要求に応じたリソースを返す:
そのユーザが認可されたときには、サーブレット・コンテナはオリジナルの URL 要求の結果をクライアントに返す。図の例では、JSP ページの応答 URL が返されており、そのアプリケーションのビジネス・ロジック部分で処理する為に必要なフォーム・データをそのユーザが POST 出来るようになる。

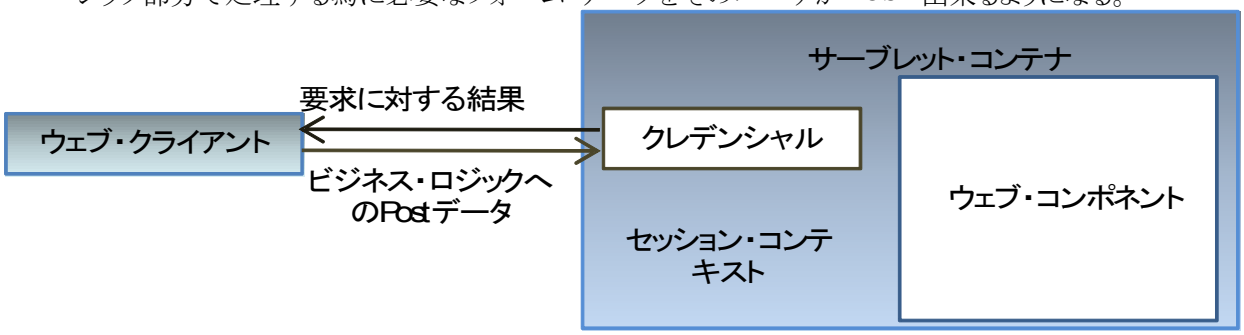


図 15-4:オリジナルの要求に対する応答

5. 法人アプリケーションの場合は、この JSP ページはそのユーザのクレデンシャルを使って EJB(Enterprise JavaBeans)へのリモート・メソッド呼び出しを行うことになろう。これにより JSP ページと EJB 間の安全な関係づけが確立される。

15.1.2 レルム、ロール、プリンシパル、クレデンシャル

ここでは、セキュリティの用語であるレルム、ロール、及びプリンシパル等に就いて説明する。これらの用語は仕様書やメーカによって異なった意味に使われていることがあるので、注意が必要である。ここでは、Java EE6 のチュートリアルをベースにし、一部は Tomcat で使われている定義も採用する。

15.1.2.1 レルム(realm)

セキュリティ・サービスの管理者が実施する適用範囲であり、セキュリティ・ポリシー・ドメイン(security policy domain)とも呼ばれる。レルムは例えば"Secure Area"(保護領域)などといったある保護空間を定義する文字列

でもあり、「HTTP ベーシック認証」の項で示すように、HTTP ベーシック(Basic)認証や HTTP ダイジェスト(Digest)認証の際サーバから認証データの一部としてクライアントに渡され、ユーザに表示される。

一方サーバ上のセキュリティ保護されたリソースたちは幾つかの保護空間に区分され、各々の区分はそれぞれ認証のスキームあるいは／または認可のデータベースを持つ。このデータベースはグループ化されたあるいはされていないユーザたちの集合で構成される。

またサーバの世界ではユーザのクレデンシャルを集めたものをレルムということも多い。レルムはファイルやデータベースとして安全にストアされねばならない。従って、その為の暗号化や認証も必要となる。

Java EE サーバの認証サービスでは、複数のレルム内（ファイル・レルム、管理者レルム、及び証明レルム）でユーザたちを統括できる。

- ファイル・レルム：サーバは keyfile という名前のファイルにユーザのクレデンシャルたちをローカルにストアする。管理者は Admin Console という管理者用のコンソールを使ってファイル・レルム内のユーザたちを管理できる。ファイル・レルムを使うときは、サーバはこのファイル・レルムをチェックしてユーザ ID を検検証する。このレルムは HTTP プロトコルと証明を使うウェブ・ブラウザ・クライアント以外の総てのクライアントの認証のために使われる。
- 証明レルム：証明レルムではサーバはユーザのクレデンシャルたちを証明データベースにストアする。証明レルムを使う場合は、サーバはウェブのクライアントたちを認証するのに HTTPS プロトコルでの証明を使用する。証明レルム内でのユーザ ID 検証には、認証サービスは X.509 証明を使用する。X.509 証明の共通名フィールドがプリンシパル名として使用される。
- 管理者レルム：これはファイル・レルムでもあるが、admin-keyfile という名前のファイルに管理者クレデンシャルをローカルにストアする。管理者は Admin Console という管理者用のコンソールを使ってファイル・レルム内のユーザたちを管理するのと同じやり方で管理者レルム内の管理者を管理できる。

Tomcat の場合は、レルムを「ユーザ名とパスワードの組み合わせのようなあるウェブ・アプリケーション（複数のアプリケーションの集合の場合もあり）のユーザを一意的に定める為のデータベースと、認証された各ユーザに付与されているロールの一覧を列挙したものの双方を合わせたもの」と定義している。サーブレット仕様書の 13.6.5 項ではアプリケーションたちがセキュリティ要求を（配備記述子で）宣言する為の可搬性あるメカニズムが記されているが、サーブレット・コンテナとそれに結び付けられたユーザとロールの情報間のインターフェイスを定めた可搬性ある API というものは存在していない。しかしながら多くの場合は、商用環境で既に存在している既存の認証データベースあるいはメカニズムに、サーブレット・コンテナを「接続」出来ることが好ましい。したがって Tomcat では、この接続を確立する為に「プラグ・イン」部品が実装できる Java インターフェイス(org.apache.catalina.Realm)が定義されている。これには各種認証情報源への接続をサポートする標準の 6 つのプラグ・インが用意されている：

- JDBCRealm：リレーショナル・データベースにストアされている認証情報に、JDBC ドライバを介して、アクセスする。
- DataSourceRealm：リレーショナル・データベースにストアされている認証情報に、指定した JNDI JDBC DataSource を介して、アクセスする。
- JNDIRealm：LDAP ベースのディレクトリ・サーバ内にストアされている認証情報に、JNDI のプロバイダを介して、アクセスする。
- UserDatabaseRealm：ある UserDatabase JNDI リソース内にストアされている認証情報に、アクセスする。この UserDatabase JNDI リソースは一般的には XML ドキュメント(conf/tomcat-users.xml)で指定される。**Tomcat に同梱されている server.xml では、これがデフォルトとして記述されている。**
- MemoryRealm：XML ドキュメント(conf/tomcat-users.xml)で初期化されるメモリ内オブジェクトのコレクションにストアされている認証情報にアクセスする。これは簡単なデモ用を対象にしている。
- JAASRealm：Java 認証認可サービス(JAAS)を介して認証情報にアクセスする。

15.1.2.2 ロール(Role)

アプリケーションの配備者が、あるリソースたちのセットへのアクセスを許可する為に複数のユーザを対象として定義する抽象的な論理グループのことをいう。ロール（役割）という名前でも理解されるように、具体的には情報管理職、一般社員、外注事業者、得意先、一般訪問者、などをイメージすれば良い。定義されたロールは、アプリケーションの配備時に運用環境でプリンシパルやグループなどのセキュリティ ID にマップされる。Java EE6 チュートリアルでは、「ロールはロックを開く為の鍵に例えれば良い。多くの人がその鍵の合鍵を持っている可能性がある。ロックはその人が誰かということは気にしないで、正しい鍵を持っているかどうかを見極めるだけである」と書いている。

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。

サーブレットは、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、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：

- 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。
- 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。

15.1.2.3 プリンシパル(Principal)

セキュリティ・サーバ内の認証プロトコルで認証され得る実体のことを抽象化したもの。プリンシパルはプリンシパル名を使って識別され、認証データを使って認証される。具体的には、ユーザや、ユーザが所属するグループ、あるいはコンピュータなどを指す。

15.1.2.4 クレデンシャル(Credential)

あるプリンシパルを認証するのに使われる情報（セキュリティ属性、例えばユーザ名とパスワードなど）、またはその情報への参照である。プリンシパルは認証によりクレデンシャルを取得する。

15.1.3 宣言的(Declarative)セキュリティとプログラム(Programmatic)的セキュリティ

ウェブ・アプリケーションは通常、コンテナに配備できる複数のリソース（構成要素）で構成されている。これらのリソースに対するセキュリティはコンテナが提供する。それには宣言的セキュリティとプログラム的セキュリティが存在する。

15.1.3.1 宣言的セキュリティ

宣言的セキュリティでは、そのアプリケーションのリソースにたいする、ロール、アクセス制御、認証の要求事項などを含むセキュリティ要求を配備記述子を使って表現する。配備記述子はアプリケーションにとって外部にあり、セキュリティのロールたちとアクセス要求が、環境固有のセキュリティ・ロール、ユーザ、及びポリシに、如何にマップされるかを指定する情報で構成される。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、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たいし、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**。

アノテーション（メタデータとも呼ばれる）はクラス・ファイル内でセキュリティに関する情報を指定するのに使われる。アプリケーションが配備されたときに、この情報が使用されるか、あるいはそのアプリケーションの配備記述子によって凌駕されるかする。アノテーションを使えば XML 記述子内で宣言的情報を書く必要が無くなる。コード上でアノテーションを追加するだけで、要求情報が生成される。@ServletSecurity アノテーションはこのように、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 メソッドを介して宣言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。その詳細は別途説明する。

15.1.3.2 プログラム的セキュリティ

プログラム的なセキュリティはアプリケーションの中に組み込まれ、セキュリティの決定をするのに使われる。プログラム的なセキュリティは、宣言的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・モデルを表現するのに不十分な場合に有用である。

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：

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

これらのメソッドの詳細は、別途詳細に説明するが、そのなかのポイントとなるメソッドたちを簡単に説明する：

- HttpServletRequest#login(String username, String password)
  これはフォーム・ベースのログインの置き換えになる。アプリケーションはクレデンシャルの収集を管理できるようになる。
- HttpServletRequest#authenticate(HttpServletResponse)
  認証制約でカバーされていないリソースから、アプリケーションはコンテナ仲介の認証を開始させることが出来る。また、アプリケーションは何時認証を必要とさせるかを決めることが出来るようになる
- HttpServletRequest#logout

これらに就いては「プログラム的なセキュリティ」の節で説明する。

15.2節 認証メカニズム

アプリケーションの認証とその設定は、配備記述子あるいは HttpServletRequest#authenticate メソッドを使って比較的簡単にできるが、そのメカニズムをきちんと理解することは重要であろう。ここではこれらのメカニズムを実験を含めてやや詳しく説明することとする。

サーブレット 3.0 仕様書の 13.6 節に記されているように、サーブレット・コンテナは以下の 4 つの認証のメカニズムに対応している。

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

これらのコンテナが関与する認証では、一旦そのユーザが認証されたら、そのユーザのプリンシパルが request.getUserPrincipal()で利用できるようになる。ユーザ名は request.getUserPrincipal().getName()で取得できる。そのユーザが指定したロールに属するかは request.isUserInRole(roleToCheck)でチェックできる。

15.2.1 HTTP ベーシック認証

このベーシック認証は HTTP/1.0 (RFC 1945)から導入されている。その後この認証方式は HTTP/1.1 (RFC 2616)で引き継がれ、更にで RFC 2617 はこの認証方式と次に示す HTTP ダイジェスト認証が規定されている。

この認証方式は、ブラウザあるいは他のクライアントのプログラムが要求をサーバに送るときにユーザ名とパスワードの形でクレデンシャルを送信できるようにした方式である。ユーザ名とパスワードはコロン(:)で接続され、Base64 アルゴリズムでエンコードされ送信される。Base64 エンコーディングは 8 ビット ASCII の 3 文字列を 6 ビット(即ち 64 文字のセット)の 4 文字に変換する MIME 方式のひとつである。これは当初は 7 ビット(パリティ 1 ビットがこれに付加される)の伝送路でも送信できるよう、また多バイト文字や制御文字も正しく伝送できるよう考案されたエンコーディングであるが、これによりユーザ名とパスワードを読みとり難くする、即ちセキュリティが上がるという特徴を持つ。

15.2.1.1 HTTP ベーシック認証のシーケンス

具体的な実験でそのシーケンスを説明する。なおこの実験を繰り返す場合は、その都度ブラウザを閉じ、また開く必要がある。そうしないとその URL を再度要求すると、ブラウザは認証ヘッダ付きのメッセージをサーバに送ってしまうからである。:

1. クライアントが保護されたリソースをサーバに要求する。
   これは Apple の Safari で http://localhost:12345/security/BasicAuthentification をアクセスした例である。
   HTTP メッセージを調べる為に、ポート番号を 12345 としてプロキシを介在させている。

```
GET /security/BasicAuthentication HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja-JP) AppleWebKit/533.19.4 (KHTML, like
Gecko) Version/5.0.3 Safari/533.19.4
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: ja-JP
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

2. サーバは保護されたリソースへの要求で且つ初めてのユーザである為、次のような 401 応答を返す:

```
HTTP/1.1 401 Unauthorized
Server: Apache-Coyote/1.1
WWW-Authenticate: Basic realm="Secure Area"
Content-Type: text/html;charset=Windows-31J
Transfer-Encoding: chunked
Date: Fri, 25 Feb 2011 01:38:14 GMT

a6
<HTML>
<HEAD><TITLE>Error</TITLE>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=Windows-31J"></HEAD>
<BODY><H1>401 Unauthorized.</H1></BODY></HTML>

0
```

ここで注意することは、ステータス行で"401 Unauthorized"と認証が必要であることをクライアントに知らせていること、及び WWW-Authenticate ヘッダ行で認証手順が Basic で、レルムが"Secure Area"であることを通知していることである。

3. ブラウザはこの応答を基に、次のような画面を表示して、ユーザにユーザ名とパスワードの入力を求める。渡されたレルムがこの画面で「localhost:8080 サーバの Secure Area では、ユーザー名とパスワードが必要です」などと表示される。また、「パスワードは暗号化されずに送信されます」などと表示される:

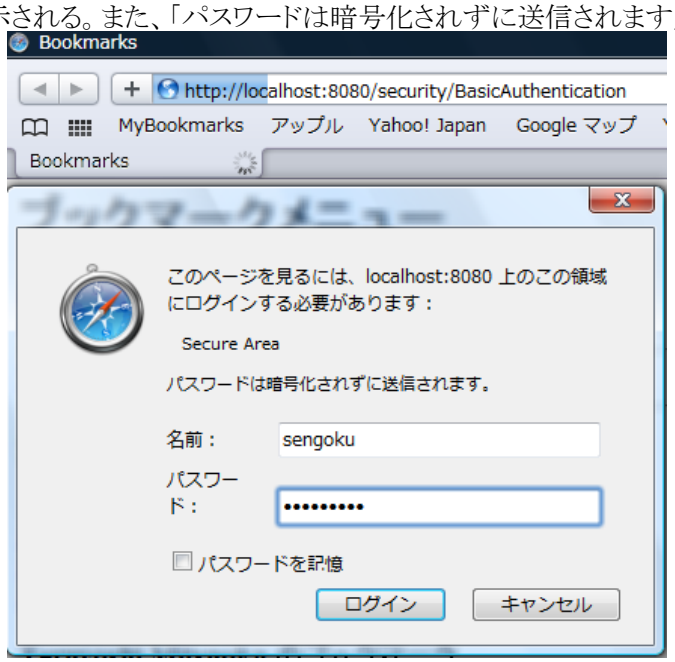


図 15-5：ベーシック認証でのログイン画面

4. ユーザがユーザ名とパスワードを入力して「ログイン」ボタンをクリックすると、次のようなメッセージがサーバに送信される:

```
GET /security/BasicAuthentication HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja-JP) AppleWebKit/533.19.4 (KHTML, like
Gecko) Version/5.0.3 Safari/533.19.4
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: ja-JP
Accept-Encoding: gzip, deflate
Authorization: Basic c2VuZ29rdTpzZW5nb2t1Mzg=
Connection: keep-alive
```

ここで注意することは、Authlization ヘッダ行が付加され、その値は"Basic c2VuZ29rdTpzZW5nb2t1Mzg="となっていることである。これはユーザ名(sengoku)とパスワード(sengoku38)をコロンで接続した文字列を Base64 エンコーディングしたものである。従って、ユーザ名とパスワードは直接は読み出し難くなっている。なお**ユーザ名とパスワードには半角英数文字に制限すべきで、日本語を使うことは好ましくない**。ブラウザによって期待した文字セットでエンコードされない可能性があるし、バイト変換したときにコロンのような文字になってはいけないからである。

5. サーバはこのメッセージを受けると、認証を確認したのち次のような応答を返している。

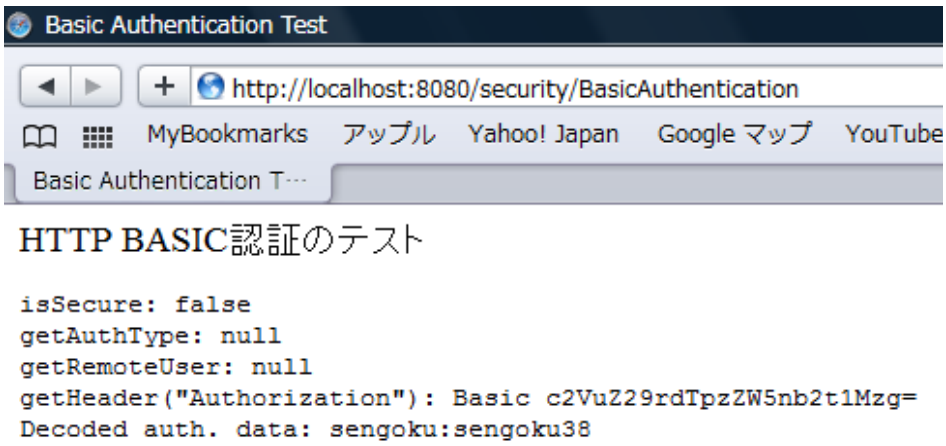


図 15-6：ベーシック認証テスト・サーブレットの応答

15.2.1.2 HTTP ベーシック認証テスト用のサーブレット

以下はこの HTTP ベーシック認証テスト用サーブレットのコードである：

BasicAuthenticationTest サーブレット

```
001 package security;
002
003 import java.io.IOException;
004 import java.io.PrintWriter;
005
006 import javax.servlet.ServletException;
007 import javax.servlet.annotation.WebServlet;
008 import javax.servlet.http.HttpServlet;
009 import javax.servlet.http.HttpServletRequest;
010 import javax.servlet.http.HttpServletResponse;
011
012 import org.apache.commons.codec.binary.Base64;
013
014 @WebServlet("/BasicAuthentication")
015 public class BasicAuthenticationTest extends HttpServlet{
016   private static final long serialVersionUID = 1L;
017   public static Base64 codec = new Base64(true);
018
019   public void doGet(HttpServletRequest req, HttpServletResponse res)
020   throws ServletException, IOException
021   {
022      if(req.getHeader("Authorization")==null){
023         res.addHeader("WWW-Authenticate","Basic realm=\"Secure Area\"");
024         res.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
025         res.setContentType("text/html; charset=Windows-31J");
026         PrintWriter out = res.getWriter();
027         out.println("<HTML>");
028         out.println("<HEAD><TITLE>Error</TITLE>");
029         out.println("<META HTTP-EQUIV=\"Content-Type\" CONTENT=\"text/html; charset=Windows-
31J\"></HEAD>");
030         out.println("<BODY><H1>401 Unauthorized.</H1></BODY></HTML>");
031      }else{
032         res.setContentType("text/html; charset=Windows-31J");
033         PrintWriter out = res.getWriter();
034         out.println("<HTML>");
035         out.println("<HEAD><TITLE>Basic Authentication Test</TITLE></HEAD>");
036         out.println("<BODY>");
037         out.println("<BIG>HTTP BASIC認証のテスト</BIG><PRE>");
038         out.println("isSecure: " + (new Boolean(req.isSecure())).toString());
039         out.println("getRemoteUser: " + req.getRemoteUser());
040         String auth = req.getHeader("Authorization");
041         out.println("getHeader(\"Authorization\"): "+ auth);
042         if (auth != null){
043            String decodedAuth = auth.substring(auth.lastIndexOf(" ")+1);
044 //         System.out.println("Decoded auth. data: "+ decodedAuth);
045            decodedAuth = new String(codec.decode(decodedAuth));
046            out.println("Decoded auth. data: "+ decodedAuth);
047         }
```

```
048         out.println("</PRE></BODY></HTML>");
049      }
050   }
051
052   static void main(){
053      System.out.println(codec.encodeToString((new String("sengoku:sengoku38")).getBytes()));
054   }
055 }
```

このコードはテスト用のものであり、Base64 のコーデック(エンコーダ/デコーダ)が使われている。Java 標準の API には Base64 用のクラスは存在していない。ここでは Apache Commons が用意しているライブラリ(org.apache.commons.codec)を採用している。このライブラリはダウンロードのページからダウンロードできるが、本チュートリアルの WAR ファイルにはすでにインポートされている。このクラスの Javadoc の翻訳は「参考資料」の章にあるので、そちらを見て頂きたい。

クライアントからの認証クレデンシャルの取り出し方は 040 行から 046 行を参考にすれば良い。Base64 にはスペース文字が含まれないので、043 行に示すように最後のスペース文字以降が Base64 エンコードされた部分だと判断し、045 行に示すようにこれをデコードしている。デコードした結果はバイト配列となるのでそれを基に String オブジェクトを生成している。

15.2.2 HTTP ダイジェスト認証

HTTP ダイジェスト認証は、HTTP ベーシック認証とともに RFC 2617 で定められている(日本語訳はここ)。参考となる日本語の資料はこちらを参考にされたい。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ユーザの認証を行う。しかしながら 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(及び追加的データ)の MD5 反方向暗号化ハッシュ(ダイジェスト)を送信する。これにより盗聴や改竄に対する耐性が向上する。ただし HTTP ダイジェスト認証は、現在はさほど広くは普及していない。

日本語の Wikipedia では、その手順を次のように説明している:

1. クライアントは認証が必要なページを要求する。しかし、通常ここではユーザ名とパスワードを送っていない。なぜならばクライアントはそのページが認証を必要とするか否かを知らないためである。

2. サーバは 401 応答コードを返し、レルム(認証領域)や認証方式(Digest)に関する情報をクライアントに返す。このとき、ランダムな文字列(nonce)も返される。RFC には例として次のような応答メッセージが示されている:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest
       realm="testrealm@host.com",
       qop="auth,auth-int",
       nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
       opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

3. それを受けたクライアントは、認証領域(通常は、アクセスしているサーバやシステムなどの簡単な説明)をユーザに提示して、ユーザ名とパスワードの入力を求める。ユーザはここでキャンセルすることもできる。
4. ユーザによりユーザ名とパスワードが入力されると、クライアントは nonce とは別のランダムな文字列(cnonce)を生成する。そして、ユーザ名とパスワードとこれら 2 つのランダムな文字列などを使ってハッシュ文字列(response:128 ビットからなる)を生成する。

5. クライアントはサーバから送られた認証に関する情報とともに、ユーザ名と応答をサーバに送信する。RFC には例として次のような応答メッセージのなかの Authorization ヘッダが示されている:

```
Authorization: Digest username="Mufasa",
```

```
            realm="testrealm@host.com",
            nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
            uri="/dir/index.html",
            qop=auth,
            nc=00000001,
            cnonce="0a4f113b",
            response="6629fae49393a05397450978507c4ef1",
            opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

6. サーバ側では、クライアントから送られてきたランダムな文字列(nonce、cnonce)などとサーバに格納されているハッシュ化されたパスワードから、正解のハッシュ(ダイジェスト)を計算する。

7. この計算値とクライアントから送られてきた response とが一致する場合は、認証が成功し、サーバはコンテンツを返す。不一致の場合は再び 401 応答コードが返され、それによりクライアントは再びユーザにユーザ名とパスワードの入力を求める。

15.2.2.1 認証要求ヘッダと認証応答ヘッダ

サーバからクライアントに送信する HTTP メッセージの中の認証応答ヘッダの指示子は以下のようである:

- realm
  これはユーザがどのユーザ名とパスワードを使うかを判断する為にユーザに表示する文字列である。これには少なくともその認証を行うホストの名前が含まれねばならず、また付加的にアクセスを持っているユーザたちの集まりを示すものである。例えば"registered_users@gotham.news.com"という文字列になる。

- Domain
  保護空間を定義する、ダブル・クォーテーションにて括られ、更にスペースでくぎられた URI のリストで、保護空間を規定するもの。ある URI が絶対パス(abs_path)である場合は、それはアクセスされているサーバの正規ルート URI に相対的なものとなる。このリストのなかの absoluteURI(絶対 URI)は、アクセスされているもの以外のサーバを参照していることもある。クライアントはこのリストを使って、同じ認証情報を送信できる URL たちのセットを判断できる:即ちプレフィックスとしてこのリストの URI を持っているどの URI も同じ保護空間にあるとみなして良い。この指示子が省略されるか、あるいは空の場合は、クライアントは、応答するサーバ上の全ての URI が保護空間だとみなさねばならない。

- nounce(「ナーンス」と発音、その場限りのという意味の言葉)
  401 応答が返されるごとに一意的に生成されねばならないサーバが指定するデータ文字列。この文字列は base64 あるいは 16 進データである事が推奨される。特に、この文字列はクォート文字列としてヘッダ・ライン中にて渡されるので、ダブルクォート文字の使用は許されない。nonce の内容は実装依存である。その実装の品質は良い選択がされているかどうかに依存する。例えば nonce は以下の base 64 エンコーディングから成るものとする事ができる:

```
time-stamp H(time-stamp ":" ETag ":" private-key)
```

  ここに time-stamp はサーバが生成する時刻あるいは非繰り返しの値、ETag はこの要求エンティティに結び付けられた HTTP Etag ヘッダ値、private-key はこのサーバのみが知っているデータである。この様式の nounce を使ってサーバは、クライアントからの認証ヘッダを受けたらハッシュ部分を計算し、そのヘッダからの nounce と一致しない、あるいは time-stamp 値が十分最新のものでないときは、その要求を拒否する。このようにしてサーバはこの nounce の有効期限を持たすことが出来る。ETag を含めることで、アップデートされたリソースのバージョンに対し再要求されることを防止できる。
  nounce はクライアントにとっては意味不明(opaque)である。

- opaque(「オペイク」と発音、不透明あるいは意味不明という意味の単語)
  サーバが指定した文字列のデータで、これに続く同じ保護空間のなかの URI たちへの要求の

Authorization ヘッダのなかで、変更なしでクライアントがサーバに戻さねばならない。この文字列は Base64 または 16 進表記データであることが推奨される。

- stale (「ステイル」と発音、古くなったという意味の単語)
  返された nounce 値は古くなったものなのでそのクライアントからのその前の要求は拒否されたことを意味するフラグ。この stale が TRUE (大文字と小文字は区別しない) のときは、クライアントはユーザに再度ユーザ名とパスワードを要求することなく、新しく暗号化した応答でその要求を再試行できる。サーバはその nounce に対するダイジェストは有効ではあるがその nounce が無効な場合に限り stale を TRUE にセットできる。stale が FALSE、TRUE 以外の文字列、あるいは stale 指示子が存在しない場合は、そのユーザ名とパスワードは無効であり、新しい値を取得しなければならない。

- algorithm (アルゴリズム)
  ダイジェストとチェックサムを生成するのに使われるアルゴリズムのペアを示す文字列である。これが指定されていないときは、"MD5"だとみなされる。このアルゴリズムが不明なときはこのチャレンジは無視される。

クライアントからサーバに送る HTTP メッセージの中の認証要求ヘッダの指示子は以下のようである:

- opaque と algorithm のフィールドの値は、要求されているエンティティへの WWW-Authenticate ヘッダのなかで供給されているものと同じでなければならない。

- Response
  以下に記すように計算された 32 桁の 16 進数文字列で、そのユーザがパスワードを知っていることを証明する為のものである。

- Username
  指定されたレルム内のそのユーザの名前

- digest-uri
  Request-Line (HTTP メッセージのトップにある要求行) の Request-URI からの URI で、重複しているがこれによりプロキシたちが中継中に Request-Line を変更できるようにする為である。

- qop
  そのメッセージに対しどんな「保護品質 (quality of protection) 」をそのクライアントが適用しているのかを示す。これが存在する場合は、その値はサーバが WWW-Authenticate ヘッダで対応していることを示す代替物のひとつで無ければならない。この値は要求ダイジェストの計算に影響を与える。

- cnonce
  qop 指示子が送信されているときはこれが指定されていなければならないしまた、サーバが WWW-Authenticate ヘッダ・フィールドで qop 指示子を送信していないときはこれは指定されてはいけない。cnonce-value はクライアントが指定する意味不明の引用符ではさまれた文字列の値であり、サーバとクライアント双方にとって選択平文攻撃を回避し、相互認証を提供し、ある程度のメッセージ耐性保護をもたらす為に使われる。

- Nonce-count
  qop 指示子が送信されているときはこれが指定されていなければならないしまた、サーバが WWW-Authenticate ヘッダ・フィールドで qop 指示子を送信していないときはこれは指定されてはいけない。この nc-value はこの要求でこの nonce 値でクライアントが送信した要求の数 (現在の要求を含める) を 16 進表示したものである。例えば、もし与えられた nonce 値にたいする応答で送信された最初の要求の中では、そのクライアントは"nc=00000001"を送信する。この指示子の意図はサーバがこのカウントのコピーを保持することで再要求を検出できるようにすることである–もし同じ nc-value が 2 回送られてきたときは、その要求は再要求である。

- Auth-param
  これは将来の拡張の為の指示子で、認識できないどの指示子も無視されねばならない。

更なる詳細は RFC 2617 を見て頂きたい。

15.2.2.2 org.apache.catalina.authenticator.DigestAuthenticator 及び RealmBase

Tomcat の場合は、配備記述子、アノテーション、HttpServletRequest#authenticate メソッドを使って組み込みのダイジェスト認証機能を動作させることが出来る。

しかしながら、ここでは Tomcat 7 の内部 API のコードを使用して、より詳細に HTTP ダイジェスト認証のシーケンスを理解することにする。これらの内部 API は殆どのメソッドが protect キーワードでアクセスが制限されているうえに、**内部の Request や Response のクラスは RequestFacade と ResponseFacade でラップされていて、サーブレット側から ServletRequest や ServletResponse instances のインスタンスを Request や Response にダウンキャストしてアクセス出来ないようにして、セキュリティを高めている。**従って、ダイジェスト・パスワードを動的に計算する必要のあるアプリケーション、あるいはパスワードをダイジェスト化してデータベース等に蓄積するアプリケーションの為に使うことが出来る Tomcat の内部 API は限られている。但しソース・コードは公開されているので、ソース・コードを流用することは可能である。

セキュリティ関係では、org.apache.catalina.authenticator.DigestAuthenticator 及び RealmBase などといったクラスが用意されている。ここではこれらのクラスをまず説明する。

最初に DigestAuthenticator クラスであるが、これは org.apache.catalina.authenticator.AuthenticatorBase のダイジェスト認証の為の継承物である。これは配備記述子の<security-constraint>要素を行使する Valve（要求の処理部品）インターフェイスの実装物でもある。従って配備記述子（及びアノテーション）で<security-constraint>が設定されていることが前提である。

このクラスの使用には以下の制約がある：
- このクラスを使うときには、これを付加させる Context は、ユーザたちを認証をし、彼らが割り当てられているロールたちを列挙する為に使える、結び付けられた Realm を持っていなければならない
- この Valve は HTTP 要求にのみ使える

下表はその中の主要なメソッドたちである（詳細は「参考資料」の章の表を見られたい。また、ソース・コードはこのリンクなどを見て頂きたい）：

表 15-1：DigestAuthenticator の主たるメソッドたち

メソッド	内容
public boolean **authenticate**(Request request, HttpResponse response, LoginConfig config) throws java.io.IOException	指定された login 設定に基づいて、この要求をしているユーザを認証する。指定された制約を満たされてときは true を、既に応答チャレンジを作ってしまっているときは false を返す。 規定： AuthenticatorBase の authenticate 引数： request - 処理中の要求オブジェクト response - 作成中の応答オブジェクト config – どのように認証を行うべきかを記述したログイン設定 例外： java.io.IOException - 入出力のエラーが発生したとき
protected java.lang.String **parseUsername**(java.lang.String authorization)	指定された認証文字列から username を構文解析して取り出す。特定できないときは null を返す 引数： authorization - 構文解析する認証文字列

メソッド	内容
protected java.lang.String **generateNOnce**(javax.servlet.http.HttpServletRequest request)	一意的トークンを生成する。このトークンは次のようなパタンに従って生成される: `NOnceToken = Base64 ( MD5 ( client-IP ":" time-stamp ":" private-key ) )` 引数: request - HTTP サーブレット要求

protected void
setAuthenticateHeader(javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response,
LoginConfig config, java.lang.String nOnce)
	WWW-Authenticate ヘッダを生成する。このヘッダは以下のテンプレートに従わねばならない:
	`WWW-Authenticate = "WWW-Authenticate" ":" "Digest" digest-challenge`
	
	`digest-challenge = 1#( realm | [ domain ] | nOnce | [ digest-opaque ] |[ stale ] | [ algorithm ] )`
	
	`realm = "realm" "=" realm-value`
	`realm-value = quoted-string`
	`domain = "domain" "=" <"> 1#URI <">`
	`nonce = "nonce" "=" nonce-value`
	`nonce-value = quoted-string`
	`opaque = "opaque" "=" quoted-string`
	`stale = "stale" "=" ( "true" | "false" )`
	`algorithm = "algorithm" "=" ( "MD5" | token )`
	引数:
	request - HTTP サーブレット要求
	response - HTTP サーブレット応答
	config - 以下に認証を行うかを記述したログイン設定
	nOnce - nonce トークン

次の org.apache.catalina.realm.RealmBase クラスは、先に説明した Tomcat が用意しているデータベースの認証の為の 6 つのレルム (CombinedRealm、DataSourceRealm、JAASRealm、JDBCRealm、JNDIRealm、MemoryRealm、及び UserDatabaseRealm) のベースとなっているものである。あとで説明するように、ダイジェスト・パスワードを動的に計算する場合には org.apache.catalina.realm.RealmBase クラスの static メソッドである Digest()に平文のパスワードとダイジェスト・アルゴリズム名を引数として渡して呼び出せば良い。このメソッドはダイジェスト化されたパスワードを返す。このクラスの Javadoc はここにあるので参考にされたい。なお java.security.MessageDigest も利用可能である。

以下はこの項でそのソース・コードが使われているこのクラスのメソッドたちである(詳細は「添付資料」にあるので見て頂きたい)。このなかで public として外部からアクセスできるメソッドは Digest のみである。

表 15-2:RealmBase の主たるメソッド

メソッド	内容
public static final java.lang.String **Digest**(java.lang.String credentials, java.lang.String algorithm, java.lang.String encoding)	指定されたアルゴリズムを使ってパスワードをダイジェスト化し、その結果を対応する 16 進数の文字列に変換する。例外が起きたときは、平文のクレデンシャルが返される。 引数: credentials - そのユーザ(username)名を認証するのに使われるパスワードあるいは他のクレデンシャル algorithm - このダイジェストに使われたアルゴリズム encoding - ダイジェスト化する為の文字列の文字エンコーディング
protected static java.security.Principal **findPrincipal**(javax.servlet.http.HttpServletRequest request, java.lang.String authorization, Realm realm)	指定した認証クレデンシャルを構文解析し、指定されたレルムからこれらのクレデンシャルを認証する Principal を返す。そのような Principal が存在しないときは null を返す。 引数: request - HTTP サーブレット要求 authorization - この要求からの認証クレデンシャル realm - Principal たちを認証するのに使われる Realm
protected java.lang.String **parseUsername**(java.lang.String authorization)	指定された認証文字列から username を構文解析して取り出す。特定できないときは null を返す 引数: authorization - 構文解析する認証文字列

メソッド	内容
protected static java.lang.String **removeQuotes**(java.lang.String quotedString, boolean quotesRequired)	文字列上の引用符を外す。RFC2617 ではレルムを除く総てのパラメタで引用符はオプショナルだと書かれている。
protected static java.lang.String **removeQuotes**(java.lang.String quotedString)	文字列上の引用符を外す。
protected java.lang.String **generateNOnce**(javax.servlet.http.HttpServletRequest request)	一意的トークンを生成する。このトークンは次のようなパタンに従って生成される: `NOnceToken = Base64 ( MD5 ( client-IP ":" time-stamp ":" private-key ) )` 引数: request - HTTP サーブレット要求

protected void **setAuthenticateHeader**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response, LoginConfig config, java.lang.String nOnce)	WWW-Authenticate ヘッダを生成する。このヘッダは以下のテンプレートに従わねばならない:
	`WWW-Authenticate = "WWW-Authenticate" ":" "Digest" digest-challenge`
	
	`digest-challenge = 1#( realm | [ domain ] | nOnce | [ digest-opaque ] |[ stale ] | [ algorithm ] )`
	
	`realm = "realm" "=" realm-value`
	`realm-value = quoted-string`
	`domain = "domain" "=" <"> 1#URI <">`
	`nonce = "nonce" "=" nonce-value`
	`nonce-value = quoted-string`
	`opaque = "opaque" "=" quoted-string`
	`stale = "stale" "=" ( "true" | "false" )`
	`algorithm = "algorithm" "=" ( "MD5" | token )`
	引数:
	request - HTTP サーブレット要求
	response - HTTP サーブレット応答
	config - 以下に認証を行うかを記述したログイン設定
	nOnce - nonce トークン

15.2.2.3 HTTP ダイジェスト認証のシーケンス

それでは、これらの API のソース・コードを使った DigestAuthenticationTest サーブレットを用いて、HTTP ダイジェスト認証のシーケンスを確認することにする。なおこの実験を繰り返す場合は、その都度ブラウザを閉じ、また開く必要がある。そうしないとその URL を要求すると、ブラウザは認証ヘッダ付きのメッセージをサーバに送ってしまうからである。

1. クライアントが保護されたリソースをサーバに要求する。
   これは Apple の Safari で http://localhost:12345/security/DigestAuthentication をアクセスした例である。
   HTTP メッセージを調べる為に、ポート番号を 12345 としてプロキシを介在させている。

```
GET /security/DigestAuthentication HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja-JP) AppleWebKit/533.19.4 (KHTML, like
Gecko) Version/5.0.3 Safari/533.19.4
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: ja-JP
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

2. サーバは保護されたリソースへの要求で且つ初めてのユーザである為、次のような 401 応答を返す:

```
HTTP/1.1 401 Unauthorized
Server: Apache-Coyote/1.1
WWW-Authenticate: Digest realm="protected area", qop="auth",
nonce="a4b3b52b9552f9632caa7b9792c18a74", opaque="bba9fe3e26aadb85a49e4496a403d4f3"
Content-Type: text/html;charset=Windows-31J
Transfer-Encoding: chunked
Date: Fri, 25 Feb 2011 02:18:24 GMT

a6
```

```
<HTML>
<HEAD><TITLE>Error</TITLE>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=Windows-31J"></HEAD>
<BODY><H1>401 Unauthorized.</H1></BODY></HTML>

0
```

ここで注意すべきことは、WWW-Authenticate 認証要求ヘッダ行の中で、Digest 認証を指定するともに、RFC 2617 に準拠して realm、qop、nonce、及び opaque のデータがクライアントに渡されていることである。

3. このメッセージを受けたクライアントは、以下のような認証画面を表示して、ユーザにユーザ名とパスワードの入力を促す。渡されたレルムがこの画面で「このページを見るには、localhost:8080 上のこの領域にログインする必要があります」などとレルムとともに表示される(Safari のこの日本語は翻訳そのもので、一般のユーザには判り難いかもしれない)。**また、「ログイン情報はセキュリティ保護されて送信されます」などと表示される。ユーザはこれにより、このログインはダイジェスト認証であることを知ることが出来る**:

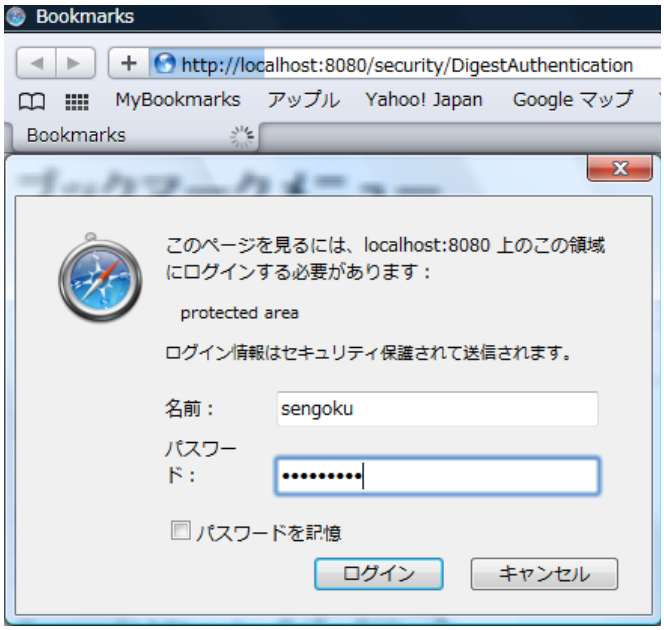


図 15-7:ダイジェスト認証でのログイン画面

4. ユーザが**ユーザ名("sengoku")とパスワード("sengoku38")**を入力し、「ログイン」ボタンをクリックすると、次のような HTTP 要求メッセージがサーバに送信される(プロキシ経由の場合):

```
GET /security/DigestAuthentication HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja-JP) AppleWebKit/533.19.4 (KHTML, like
Gecko) Version/5.0.3 Safari/533.19.4
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: ja-JP
Accept-Encoding: gzip, deflate
Authorization: Digest username="sengoku", realm="protected area",
nonce="a4b3b52b9552f9632caa7b9792c18a74", uri="/security/DigestAuthentication",
response="f9dd0cb41adef38752cee5cda27f684d", opaque="bba9fe3e26aadb85a49e4496a403d4f3",
cnonce="6403a291408e405c2e2b792d53843fda", nc=00000001, qop="auth"
Connection: keep-alive
```

ここで注意すべき点は、Autholization ヘッダ行で、RFC 2617 に準拠して Digest 応答であることとともに、username、realm、nonce、uri、response、opaque、cnonce、nc、qop 情報が通知されていることである。

5. サーバはこの認証メッセージを受けて、次のような応答を返している。これらの情報の詳細は、このサーブレットのコードの説明のところで詳しく説明するが、クライアントからのダイジェスト(ClientDigest)と、

サーバが生成したダイジェスト(ServerDigest)が一致しており、ここではクライアントからのユーザ名("sengoku")とパスワード("sengoku38")がサーバに登録されているものと一致したことを示している:

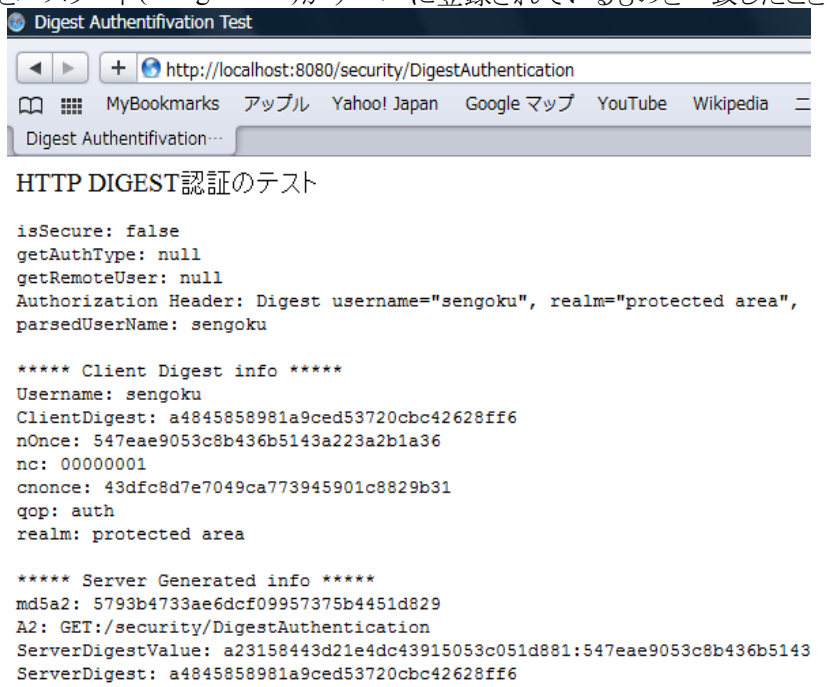


図 15-8:ダイジェスト認証テスト・サーブレットの出力

15.2.2.4 HTTP ダイジェスト認証テスト用のサーブレット

このサーブレットのコードは、Tomcat の内部コードを含めているので、やや長くなってしまっているが、Tomcat が実際に HTTP ダイジェスト認証で何をやっているのかを理解するのに寄与しよう。

```
001 package tutorial_security;
002
003 import java.io.IOException;
004 import java.io.PrintWriter;
005 import java.security.MessageDigest;
006 import java.security.NoSuchAlgorithmException;
007 import java.util.StringTokenizer;
008
009 import javax.servlet.ServletException;
010 import javax.servlet.annotation.WebServlet;
011 import javax.servlet.http.HttpServlet;
012 import javax.servlet.http.HttpServletRequest;
013 import javax.servlet.http.HttpServletResponse;
014
015 import org.apache.catalina.deploy.LoginConfig;
016 import org.apache.catalina.util.MD5Encoder;
017
018 @WebServlet("/DigestAuthentication")
019 public class DigestAuthenticationTest extends HttpServlet{
020   private static final long serialVersionUID = 1L;
021
022   protected static final MD5Encoder md5Encoder = new MD5Encoder();
023   protected static MessageDigest md5Helper;
024   protected String key = "Catalina";
025
026   public void init(){
027      try{
028         md5Helper = MessageDigest.getInstance("MD5");
029      } catch (NoSuchAlgorithmException e) {
030         e.printStackTrace();
031         throw new IllegalStateException();
032      }
033   }
034
```

```
035   public void doGet(HttpServletRequest req, HttpServletResponse res)
036   throws ServletException, IOException
037   {
038      org.apache.catalina.deploy.LoginConfig config = new LoginConfig();
039      config.setRealmName("protected area");
040
041      if(req.getHeader("Authorization")==null){
042         String nounce = generateNOnce(req);
043         setAuthenticateHeader(req, res, config, nounce);
044         res.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
045         res.setContentType("text/html; charset=Windows-31J");
046         PrintWriter out = res.getWriter();
047         out.println("<HTML>");
048         out.println("<HEAD><TITLE>Error</TITLE>");
049         out.println("<META HTTP-EQUIV=\"Content-Type\" CONTENT=\"text/html; charset=Windows-
31J\"></HEAD>");
050         out.println("<BODY><H1>401 Unauthorized.</H1></BODY></HTML>");
051
052      }else{
053         res.setContentType("text/html; charset=Windows-31J");
054         String authorization = req.getHeader("Authorization");
055         String username = parseUsername(authorization);
056         PrintWriter out = res.getWriter();
057         out.println("<HTML>");
058         out.println("<HEAD><TITLE>Digest Authentication Test</TITLE></HEAD>");
059         out.println("<BODY>");
060         out.println("<BIG>HTTP DIGEST認証のテスト</BIG><PRE>");
061         out.println("isSecure: " + (new Boolean(req.isSecure())).toString());
062         out.println("getAuthType: " + req.getAuthType());
063         out.println("getRemoteUser: " + req.getRemoteUser());
064         out.println("Authorization Header: " + authorization);
065         out.println("parsedUserName: " + username);
066         findPrincipal(req, authorization, config.getRealmName(), out);
067         out.println("</PRE></BODY></HTML>");
068      }
069   }
070
071
072   protected String parseUsername(String authorization) {
073      //System.out.println("Authorization token : " + authorization);
074      // Validate the authorization credentials format
075      if (authorization == null)
076         return (null);
077      if (!authorization.startsWith("Digest "))
078         return (null);
079      authorization = authorization.substring(7).trim();
080      StringTokenizer commaTokenizer = new StringTokenizer(authorization, ",");
081      while (commaTokenizer.hasMoreTokens()) {
082         String currentToken = commaTokenizer.nextToken();
083         int equalSign = currentToken.indexOf('=');
084         if (equalSign < 0)   return null;
085         String currentTokenName = currentToken.substring(0, equalSign).trim();
086         String currentTokenValue = currentToken.substring(equalSign + 1).trim();
087         if ("username".equals(currentTokenName))
088            return (removeQuotes(currentTokenValue));
089      }
090      return (null);
091   }
092
093   protected static String removeQuotes(String quotedString, boolean quotesRequired) {
094      //support both quoted and non-quoted
095      if (quotedString.length() > 0 && quotedString.charAt(0) != '"' && !quotesRequired) {
096         return quotedString;
097      }
098      else if (quotedString.length() > 2) {
099         return quotedString.substring(1, quotedString.length() - 1);
100      } else {
101         return new String();
102      }
103   }
104
105   protected static String removeQuotes(String quotedString) {
106      return removeQuotes(quotedString, false);
107   }
108
109   protected String generateNOnce(HttpServletRequest request) {
```

```
110      long currentTime = System.currentTimeMillis();
111      String nOnceValue = request.getRemoteAddr() + ":" + currentTime + ":" + key;
112      byte[] buffer = null;
113      synchronized (md5Helper) {
114         buffer = md5Helper.digest(nOnceValue.getBytes());
115      }
116      nOnceValue = md5Encoder.encode(buffer);
117      return nOnceValue;
118   }
119
120   protected void setAuthenticateHeader(HttpServletRequest request, HttpServletResponse response,
121         org.apache.catalina.deploy.LoginConfig config, String nOnce) {
122      // Get the realm name
123      String realmName = config.getRealmName();
124      if (realmName == null)
125         realmName = request.getServerName() + ":" + request.getServerPort();
126      byte[] buffer = null;
127      synchronized (md5Helper) {
128         buffer = md5Helper.digest(nOnce.getBytes());
129      }
130      String authenticateHeader = "Digest realm=\"" + realmName + "\", "
131      +  "qop=\"auth\", nonce=\"" + nOnce + "\", " + "opaque=\""
132      + md5Encoder.encode(buffer) + "\"";
133      response.setHeader("WWW-Authenticate", authenticateHeader);
134   }
135
136   protected static boolean findPrincipal(HttpServletRequest request,
137         String authorization, String realm, PrintWriter out ) {
138      //System.out.println("Authorization token : " + authorization);
139      // Validate the authorization credentials format
140      if (authorization == null)
141         return (false);
142      if (!authorization.startsWith("Digest "))
143         return (false);
144      authorization = authorization.substring(7).trim();
145      String[] tokens = authorization.split(",(?=(?:[^\"]*\"[^\"]*\")+$)");
146      String userName = null;
147      String realmName = null;
148      String nOnce = null;
149      String nc = null;
150      String cnonce = null;
151      String qop = null;
152      String uri = null;
153      String response = null;
154      String method = request.getMethod();
155
156      for (int i = 0; i < tokens.length; i++) {
157         String currentToken = tokens[i];
158         if (currentToken.length() == 0)
159            continue;
160         int equalSign = currentToken.indexOf('=');
161         if (equalSign < 0)
162            return false;
163         String currentTokenName = currentToken.substring(0, equalSign).trim();
164         String currentTokenValue = currentToken.substring(equalSign + 1).trim();
165         if ("username".equals(currentTokenName))
166            userName = removeQuotes(currentTokenValue);
167         if ("realm".equals(currentTokenName))
168            realmName = removeQuotes(currentTokenValue, true);
169         if ("nonce".equals(currentTokenName))
170            nOnce = removeQuotes(currentTokenValue);
171         if ("nc".equals(currentTokenName))
172            nc = removeQuotes(currentTokenValue);
173         if ("cnonce".equals(currentTokenName))
174            cnonce = removeQuotes(currentTokenValue);
175         if ("qop".equals(currentTokenName))
176            qop = removeQuotes(currentTokenValue);
177         if ("uri".equals(currentTokenName))
178            uri = removeQuotes(currentTokenValue);
179         if ("response".equals(currentTokenName))
180            response = removeQuotes(currentTokenValue);
181      }
182      if ( (userName == null) || (realmName == null) || (nOnce == null)
183            || (uri == null) || (response == null) )
184         return false;
185      // Second MD5 digest used to calculate the digest :
186      // MD5(Method + ":" + uri)
```

```
187      String a2 = method + ":" + uri;
188
189      byte[] buffer = null;
190      synchronized (md5Helper) {
191         buffer = md5Helper.digest(a2.getBytes());
192      }
193      String md5a2 = md5Encoder.encode(buffer);
194
195      out.println();
196      out.println("***** Client Digest info *****");
197      out.println("Username: " + userName);
198      out.println("ClientDigest: " + response);
199      out.println("nOnce: " + nOnce);
200      out.println("nc: " + nc);
201      out.println("cnonce: " + cnonce);
202      out.println("qop: " + qop);
203      out.println("realm: " + realmName);
204      out.println();
205      out.println("***** Server Generated info *****");
206      out.println("md5a2: " + md5a2);
207      out.println("A2: " + a2);
208
209      String md5a1 = getDigest(userName, realmName);
210      String serverDigestValue = md5a1 + ":" + nOnce + ":" + nc + ":"
211      + cnonce + ":" + qop + ":" + md5a2;
212      String serverDigest = md5Encoder.encode(md5Helper.digest(serverDigestValue.getBytes()));
213      out.println("ServerDigestValue: " + serverDigestValue);
214      out.println("ServerDigest: " + serverDigest);
215
216      return true;
217   }
218
219
220   protected static String getDigest(String userName, String realmName) {
221      if (md5Helper == null) {
222         try {
223            md5Helper = MessageDigest.getInstance("MD5");
224         } catch (NoSuchAlgorithmException e) {
225            e.printStackTrace();
226            throw new IllegalStateException();
227         }
228      }
229      String digestValue = userName + ":" + realmName + ":" + getPassword(userName);
230      byte[] digest = md5Helper.digest(digestValue.getBytes());
231      return md5Encoder.encode(digest);
232   }
233
234   protected static String getPassword(String userName){
235      return "sengoku38";
236   }
237
238
239 }
```

このサーブレットを説明すると下表のようになる:

表 15-3 : DigestAuthenticationTest サーブレットの説明

行	説明
018	このサーブレットのパスは/security/DigestAuthentication である
022-033	このクラスは以下のインスタンス変数を持つ: • md5Encoder : 128 ビットの MD5 ハッシュを 32 バイトの 16 進表示に変換する • md5Helper : これは java.security.MessageDigest のインスタンスで、"MD5"アルゴリズムで初期化されている • key : これは nounce 生成の為のプライベート・キーで、Tomcat では"Catalina"が使用されている
035-069	doGet メソッドで、これは初回時の処理 041-050 と認証データ処理 053-068 で構成される

038-039	ここで LoginConfig のオブジェクトを用意している。実際は Catalina が配備記述子とアノテーションからこのコンテキストの為に用意する
041-050	このクライアントが初めてこのサーブレットをアクセスしたことを知り、認証データ送信をクライアントに送信する
042、 109-118	nounce を用意する。generateNOnce メソッドは、リモート・アドレス、現在時刻、及びプライベート・キーをコロンで結んだ文字列を MD5 ダイジェスト化し、それを 32 桁の文字列にしている
043、 120-134	クライアントに送信する Authenticate ヘッダ行を応答ヘッダに付加する。 • LoginConig でレルムが指定されていないときは、サーバ名とサーバ・ポート番号をセミコロンでつないだものをレルムとする • gop は"auth" • nonce は 042 行で用意した nounce • opaque は nounce を更に MD5 ダイジェスト化し 32 桁の文字列にしている
045-050	ボディ部に HTML テキストを付加している。これは一般的にはブラウザ側は無視している
053-068	クライアント側からの HTTP 認証メッセージを処理する。即ちクライアントからの認証データの分析結果と、受信したユーザ名を基にサーバ側で作った認証データを表示する HTML テキストを生成して、クライアントに送り返す
054	クライアント側からの HTTP 認証メッセージの Authorization ヘッダの値を取得する
055、 072-103	Authorization ヘッダの値からユーザ名を抽出する。ParseUserName というメソッドも org.apache.catalina.authenticator.DigestAuthenticator に含まれているメソッドである
066、 136-217	findPrincipal は org.apache.catalina.authenticator.DigestAuthenticator の中心になっているメソッドのひとつで、要求、Authorization ヘッダ値、及び自分が持っているレルムをもとにプリンシパルを求める。ここではそれを少し加工して流用している。即ち応答用の PrintWriter オブジェクトを渡して、その結果をクライアントに返すようにしている
140-184	クライアントからの Authorization ヘッダを分析して username、realm、nonce、nc、cnonce、qop、uri、response の値を取り出している
185-193	ダイジェスト計算に必要な第 2 の MD5 ダイジェストを生成する。即ちメソッドと uri をセミコロンでつないだ文字列をダイジェスト化し 32 桁の文字列にし、md5a2 としている
196-207、213、214	クライアントからの認証データと、サーバで生成した認証データを表示する
209、 220-232	クライアントからのユーザ名を基に、自分が蓄積しているそのユーザ名に対応するパスワードをレルムから取り出し、ユーザ名、レルム名、及びそのパスワードをセミコロンでつないだ文字列をダイジェスト化し 32 桁の文字列として、md5a1 としている
210-212	この md5a1、nonce、nc、cnonce、qop、及び md5a2 をコロンでつないだ文字列が serverDigestValue である。これを MD5 ダイジェスト化し 32 桁の文字列としたものがサーバ生成のダイジェストになる。これがクライアントから送信してきたダイジェスト(response)と一致すれば、そのユーザが認証されたことになる
234-235	getPassword メソッドは、org.apache.catalina.realm.RealmBase で定義されていて、Tomcat が用意している各種認証情報源への接続をサポートする標準の 6 つのプラグ・インで実装されている。ここでは実験なので、単にパスワードとして"sengoku38"という文字列を返すだけである

15.2.3 フォーム・ベース認証

フォーム・ベース認証(Form Based Authentication)というのは、その名の通りHTMLのFormを使った認証方式である。これまでのクライアント(即ちブラウザ)が用意するログイン画面(ポップアップのダイアログ・ボックス)と違って、アプリケーションに適した見た目と感じ(look and feel)をユーザに与えることが出来る。しかしながらFormの出力は暗号化されてはいない(URLエンコードは行われる)ので、電子商取引のようなアプリケーションではSSLなどのより安全なトランスポートを介して送信されるのが一般的である。

フォーム・ベース認証に関しては、サーブレット3.0仕様書の13.6.3項に次のように記されている:

> 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Post要求で返す。
> 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、302応答により、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。

従って、アプリケーションは次のものを用意しなければならない:

- 配備記述子:通常のセキュリティ設定に加えてログイン・ページとエラー・ページが宣言される
- ログイン・ページ:これはログインの為のFormが含まれるHTMLページ、JSPページ、またはサーブレットである
- エラー・ページ:これはユーザからのログイン情報がそのアプリケーションのセキュリティ制約を満たさないときにクライアントに送信されるHTMLページ、JSPページ、あるいはサーブレットである

下図はOracleのJava EEチュートリアルに示されていたフォーム・ベースの認証の流れである:

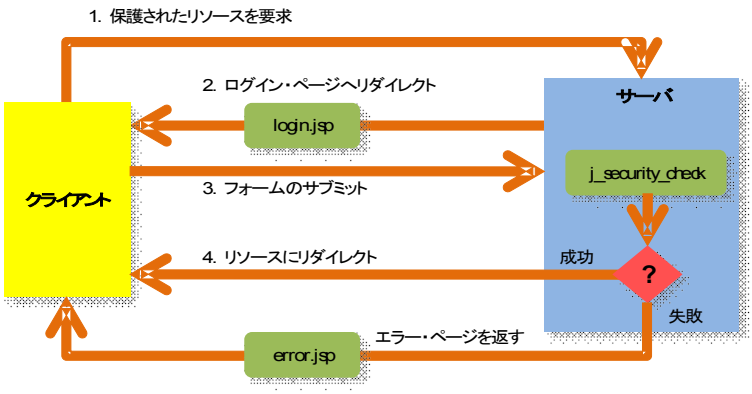


図 15-9:フォーム・ベース認証処理の流れ

以下の説明で判るように、この認証メカニズムを使う場合には次の事項に注意しなければならない:

1. **認証が通過したときにはセッションが確立されている**ので、サーブレットはそれを前提としなければならない。IsNew メソッドは false を返す
2. サーブレット側でセッションを無効あるいはタイムアウトさせると、そのユーザがこのサーブレットを再度呼び出すときは、再度認証手順をとらねばならなくなる
3. **Tomcat では、クッキーを受け付けなくしたブラウザによってはログインを 2 回入力しなければならなくなる**。クッキーを受け付けないブラウザに対しては、セッションは URL 書き換えで維持されて、サーブレットの service メソッドが呼ばれる。またログイン・ページは URL 書き換えに対応しなければならない

15.2.3.1 ログイン・ページとエラー・ページ

Tomcat のサンプルには次のような login.jsp が示されている(これはまた添付の security.war を Eclipse にインポートしたときに/security/WebContent/javascripts/security/protected/login.jsp に置かれている):

login.jsp

```
001 <!--
002  Licensed to the Apache Software Foundation (ASF) under one or more
003   contributor license agreements.  See the NOTICE file distributed with
004   this work for additional information regarding copyright ownership.
005   The ASF licenses this file to You under the Apache License, Version 2.0
006   (the "License"); you may not use this file except in compliance with
007   the License.  You may obtain a copy of the License at
008
009       http://www.apache.org/licenses/LICENSE-2.0
010
011   Unless required by applicable law or agreed to in writing, software
012   distributed under the License is distributed on an "AS IS" BASIS,
013   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
014   See the License for the specific language governing permissions and
015   limitations under the License.
016 -->
017 <html>
018 <head>
019 <title>Login Page for Examples</title>
020 <body bgcolor="white">
021 <form method="POST" action='<%= response.encodeURL("j_security_check") %>' >
022   <table border="0" cellspacing="5">
023     <tr>
024       <th align="right">Username:</th>
025       <td align="left"><input type="text" name="j_username"></td>
026     </tr>
027     <tr>
028       <th align="right">Password:</th>
029       <td align="left"><input type="password" name="j_password"></td>
030     </tr>
031     <tr>
032       <td align="right"><input type="submit" value="Log In"></td>
033       <td align="left"><input type="reset"></td>
034     </tr>
```

```
035   </table>
036 </form>
037 </body>
038 </html>
```

ログイン・ページで注意する点は次のようである:

1. 021 行目にあるように、この Form のアクションは **POST 要求**であり、エンコードはデフォルトの URL エンコーディングである。
2. 送信先(アクション)は**"j_security_check"**と特別なアドレスとなっている。仕様書では「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」と書いており、このログイン・ページが汎用的に使え、またサーバ側もその都度アクションを指定しなくても良いようにしている。オリジナルの要求パラメタはコンテナが保存しているので、コンテナはこのアドレスを受けたら、認証が OK の場合は、オリジナルの要求 URI にその要求をリダイレクトできる。
3. response.encodeURL("j_security_check")と、URL 書き換えによるセッションに対応させている。サーブレット仕様書は次のように書いている:「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。」
4. 025 と 029 行目で判るように、ユーザ名とパスワードの名前は**"j_username"**と**"j_password"**である。

エラー・ページは次のようなものとなる(これも Tomcat のサンプルを基にしている):

error.jsp

```
<!--
 Licensed to the Apache Software Foundation (ASF) under one or more
  contributor license agreements.  See the NOTICE file distributed with
  this work for additional information regarding copyright ownership.
  The ASF licenses this file to You under the Apache License, Version 2.0
  (the "License"); you may not use this file except in compliance with
  the License.  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
-->
<html>
<head>
<title>Error Page For Examples</title>
</head>
<body bgcolor="white">
Invalid username and/or password, please try
<a href='<%= response.encodeURL("/security/FormAuthentication") %>'>again</a>
.
</body>
</html>
```

これは特に説明する必要もなかろう。

15.2.3.2 j_security_check のメカニズムと注意点

j_security_check の処理の実装は、コンテナのメーカ次第である(仕様書ではその実装法は指定していない)。Tomcat などではサーブレット・コンテナは新しい HttpSession オブジェクト(及びその ID)を用意し、それをログイン・ページとともにクライアントに渡す。そのチェックが成功しそのユーザの認証がなされたら、サーブレット・コンテナはそのユーザに関する認証オブジェクトを属性として HttpSession オブジェクトに貼り付ける。この認証オブジェクトには名前などのユーザ固有の情報、もし使われていれば SSL 認証、及びタイム・スタンプが含まれる。クライアントにはセッション ID つきのクッキー(もしブラウザがクッキー対応していれば)が渡されているので、クライ

アントはその後のそのパスへの要求にはそのクッキーを付加する。サーブレット・コンテナはそのクッキーが存在すればそのユーザが認証済みであることが判る。したがって、**Tomcat の場合は response.encodeURL("j_security_check")というアクションが必須**となる。これがないと、クッキーを受け付けなくしたブラウザの場合はこの認証が使えなくなる。またクッキーを受け付けなくしたブラウザでは、そのリソースを要求するたびに認証が必要になる。一**旦成功した認証の有効期限はセッションの有効期限に従う**ので注意が必要である。Tomcat では次のようなヘッダ行がクライアントに渡されるログイン・ページの HTTP メッセージに付加されている:

```
Set-Cookie: JSESSIONID=D0144F47B748AA2B0A4B234C68C6DFEA; Path=/security; HttpOnly
```

したがってこのクッキーはそのブラウザのインスタンスが存在する間がそのセッションの有効期間となる。もしターゲットのサーブレットで**セッションをタイムアウトや無効(invalidate)とすると、認証も無効になる**のでこれも注意しなければならない。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。

15.2.3.3 フォーム・ベース認証テスト用のサーブレット

はじめにフォーム・ベース認証テスト用のサーブレットを示す。このサーブレットは認証されたユーザからの HTTP 要求の要求オブジェクトの詳細を出力するだけであり、「要求オブジェクト」の章の HttpRequestDump サーブレットの一部を流用したものである。但し 020 行目にあるように、@ServletSecurity(value = @HttpConstraint(rolesAllowed = "manager"))と、このサーブレットにアクセスできるロールを manager に限定している。従って、このサーブレットは配備記述子を変更するだけで他の認証方式のテストにも使用できる。

また、セキュリティ関係の出力データは:

- 064 行目: req.getAuthType()...認証タイプ
- 081 行目: req.getRemoteUser()...クライアントのユーザ名.
- 089 行目: req.getUserPrincipal().getName()...プリンシパルのなかの名前の属性
- 090 行目: req.isUserInRole("manager")...そのユーザが指定されたロール(ここでは"manager")に属するか
- 094 行目: req.getScheme()...HTTP、HTTPS などのその要求に使われているスキーム
- 095 行目: req.isSecure()...HTTPS のようなセキュアなチャンネル経由での要求かどうか

なお、039 行目のコメントを外すと、プログラム的なログアウトの実験が出来る。

FormBasedAuthenticationTest.java

```
001 package tutorial_security;
002
003 import java.io.IOException;
004 import java.io.PrintWriter;
005 import java.io.UnsupportedEncodingException;
006 import java.util.Enumeration;
007
008 import javax.servlet.ServletContext;
009 import javax.servlet.ServletException;
010 import javax.servlet.annotation.ServletSecurity;
011 import javax.servlet.annotation.HttpConstraint;
012 import javax.servlet.annotation.WebServlet;
013 import javax.servlet.http.Cookie;
014 import javax.servlet.http.HttpServlet;
015 import javax.servlet.http.HttpServletRequest;
016 import javax.servlet.http.HttpServletResponse;
017 import javax.servlet.http.HttpSession;
018
019 @WebServlet("/FormAuthentication")
020 @ServletSecurity(value = @HttpConstraint(rolesAllowed = "manager"))
021 public class FormBasedAuthenticationTest extends HttpServlet {
022   private static final long serialVersionUID = 1L;
023
024   /**
```

```
025    * 到来HTTP要求処理
026    */
027   public void performTask(HttpServletRequest req, HttpServletResponse res)
028   throws ServletException, IOException
029   {
030      res.setContentType("text/html; charset=Windows-31J");
031      PrintWriter out = res.getWriter();
032      out.println("<HTML>");
033      out.println("<HEAD><TITLE>HTTP Form Based Authentication Test</TITLE></HEAD>");
034      out.println("<BODY>");
035      out.println("<BIG>フォーム・ベース認証テスト・サーブレット</BIG>");
036      dumpRequest(req, res, out);
037      out.println("</BODY></HTML>");
038      //ログアウトをテストするときは次の行を活かす
039      //req.logout();
040   }
041
042   /**
043    * 到来HTTP GET / POST要求の処理
044    */
045   @Override
046   public void doGet(HttpServletRequest req, HttpServletResponse res)
047   throws javax.servlet.ServletException, java.io.IOException {
048      performTask(req, res);
049   }
050   @Override
051   public void doPost(HttpServletRequest req, HttpServletResponse res)
052   throws javax.servlet.ServletException, java.io.IOException {
053      performTask(req, res);
054   }
055
056   /**
057    * ダンプ処理の実体
058    * @throws IOException
059    */
060   public void dumpRequest(HttpServletRequest req, HttpServletResponse res,
061         PrintWriter out) throws IOException {
062
063      out.println("<BODY><BR><BR>*** 要求オブジェクトの情報 ***<PRE>");
064      out.println("getAuthType: " + req.getAuthType());
065      out.println("getCharacterEncoding: " + req.getCharacterEncoding());
066      out.println("getContentLength: " + req.getContentLength());
067      out.println("getContentType: " + req.getContentType());
068      out.println("getContextPath: " + req.getContextPath());
069      out.println("getLocalAddr: " + req.getLocalAddr());
070      out.println("getLocale: " + req.getLocale().toString());
071      out.println("getLocalName: " + req.getLocalName());
072      out.println("getLocalPort: " + req.getLocalPort());
073      out.println("getMethod: " + req.getMethod());
074      out.println("getPathInfo: " + req.getPathInfo());
075      out.println("getPathTranslated: " + req.getPathTranslated());
076      out.println("getProtocol: " + req.getProtocol());
077      out.println("getQueryString: " + req.getQueryString());
078      out.println("getRemoteAddr: " + req.getRemoteAddr());
079      out.println("getRemoteHost: " + req.getRemoteHost());
080      out.println("getRemotePort: " + req.getRemotePort());
081      out.println("getRemoteUser: " + req.getRemoteUser());
082      out.println("getRequestURI: " + req.getRequestURI());
083      out.println("getRequestedSessionId: " + req.getRequestedSessionId());
084      HttpSession session = req.getSession(false);
085      out.println("getSession(false): " + session);
086      if (session != null){
087         out.println("session.getId: " + session.getId());
088      }
089      out.println("getUserPrincipal.getName: " + req.getUserPrincipal().getName());
090      out.println("isUserInRole(\"manager\"): " + req.isUserInRole("manager"));
091      out.println("isRequestedSessionIdValid: " + (new
Boolean(req.isRequestedSessionIdValid())).toString());
092      out.println("isRequestedSessionIdFromCookie: " + (new
Boolean(req.isRequestedSessionIdFromCookie())).toString());
093      out.println("isRequestedSessionIdFromURL: " + (new
Boolean(req.isRequestedSessionIdFromURL())).toString());
094      out.println("getScheme: " + req.getScheme());
095      out.println("isSecure: " + (new Boolean()).toString());
096      out.println("getServerName: " + req.getServerName());
```

```
097      out.println("getServerPort: " + req.getServerPort());
098      ServletContext context = getServletContext();
099      out.println("getRealPath: " + context.getRealPath(req.getRequestURI()));
100      out.println("getServletPath: " + req.getServletPath());
101      out.println("getContextPath: " + req.getContextPath());
102      out.println("isAsyncSupported: " + (new Boolean(req.isAsyncSupported())).toString());
103      out.println("isAsyncStarted: " + (new Boolean(req.isAsyncStarted())).toString());
104
105      out.println();
106      out.println("要求パラメタ(Request Parameters):");
107      //注意：setCharacterEncodingメソッドはGET要求には効かないので使わないことを奨励
108      Enumeration<?> paramNames = req.getParameterNames();
109      while (paramNames.hasMoreElements()) {
110         String name = (String) paramNames.nextElement();
111         String[] values = req.getParameterValues(name);
112         try{
113            out.println("  "+new String(name.getBytes("8859_1"), "Windows-31J")+":");
114         }catch(UnsupportedEncodingException theException){
115            out.println("UnsupportedExceptionの例外が発生");
116         }
117         for (int i = 0; i < values.length; i++) {
118            try{
119               out.println("  "+i+")  "+new String(values[i].getBytes("8859_1"), "Windows-31J"));
120            }catch(UnsupportedEncodingException theException){
121               out.println("UnsupportedExceptionの例外が発生");
122            }
123         }
124      }
125
126      out.println();
127      out.println("要求ヘッダ(Headers):");
128      Enumeration<?> headerNames = req.getHeaderNames();
129      while (headerNames.hasMoreElements()) {
130         String name = (String) headerNames.nextElement();
131         String value = req.getHeader(name);
132         try{
133            String correctName = new String(name.getBytes("8859_1"), "Windows-31J");
134            String correctValue = new String(value.getBytes("8859_1"), "Windows-31J");
135            out.println("  " + correctName + " : " + correctValue);
136         }catch(UnsupportedEncodingException theException){
137            out.println("UnsupportedExceptionの例外が発生");
138         }
139      }
140
141      out.println();
142      out.println("属性(Attributes):");
143      Enumeration<?> attributeNames = req.getAttributeNames();
144      while (attributeNames.hasMoreElements()) {
145         String name = (String) attributeNames.nextElement();
146         Object value = req.getAttribute(name);
147         out.println("  " + name + " : " + value.toString());
148      }
149
150      out.println();
151      out.println("クッキー(Cookies):");
152      Cookie[] cookies = req.getCookies();
153      if ((cookies != null) && (cookies.length > 0)) {
154         for (int i = 0; i < cookies.length; i++) {
155            String name = cookies[i].getName();
156            String value = cookies[i].getValue();
157            out.println("  " + name + " : " + value);
158         }
159      }
160   }
161 }
162
```

15.2.3.4 フォーム・ベース認証のシーケンスの確認

FormBasedAuthenticationTest サーブレットを使ってフォーム・ベース認証の実験を行う。この実験を繰り返し行うときは、その都度ブラウザを再スタートする必要がある。何故ならセッションが確立されたクッキーが要求時に送

信されてしまうからである。

実験に際しては、login.jsp、error.jsp、web.xml、及び tomcat_users.xml が設定されていなければならない。この資料には security.war ファイルが添付されているので、それをインポートすれば次のようにそれらのファイルが置かれる。その詳細は「配備記述子による設定」の項で説明するが、

- web.xml では、<url-pattern>/FormAuthentication</url-pattern>とこのサーブレットが保護されたリソースであることを宣言し、また<auth-method>FORM</auth-method>で認証方式が FORM であることを宣言している

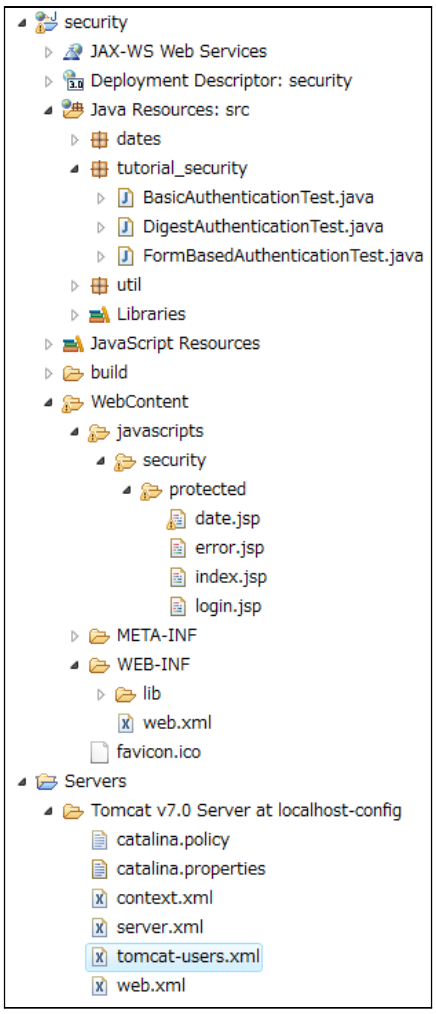


図 15-10：security.war の展開

tomcat_users.xml は赤字で示したように各自が設定しなければならない。

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Licensed to the Apache Software Foundation (ASF) under one or more
  contributor license agreements.  See the NOTICE file distributed with
  this work for additional information regarding copyright ownership.
  The ASF licenses this file to You under the Apache License, Version 2.0
  (the "License"); you may not use this file except in compliance with
  the License.  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
  See the License for the specific language governing permissions and
  limitations under the License.
--><tomcat-users>
<!--
  NOTE:  By default, no user is included in the "manager-gui" role required
  to operate the "/manager/html" web application.  If you wish to use this app,
  you must define such a user - the username and password are arbitrary.
-->
<!--
  NOTE:  The sample user and role entries below are wrapped in a comment
  and thus are ignored when reading this file. Do not forget to remove
  <!.. ..> that surrounds them.
-->

<user name="sengoku" password="sengoku38" roles="administrator,manager"/>
  <role rolename="administrator"/>
  <role rolename="manager"/>

  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <user password="tomcat" roles="tomcat" username="tomcat"/>
  <user password="tomcat" roles="tomcat,role1" username="both"/>
  <user password="tomcat" roles="role1" username="role1"/>

</tomcat-users>
```

ここでは、<user name="sengoku" password="sengoku38" roles="administrator, manager"/> <role rolename="administrator"/> <role rolename="manager"/>と、ユーザ名、パスワード、及びそのロールを宣言している。

1. ブラウザから http://localhost:8080/security/FormAuthentication をアクセスすると、サーブレット・コンテナは認証の為に次のようなユーザ名とパスワード入力の為のフォームを含んだ HTTP 応答メッセージを返す:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Thu, 01 Jan 1970 09:00:00 JST
Set-Cookie: JSESSIONID=C64F7F42333E426B4BD52C42F4EBC817; Path=/security; HttpOnly
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 1460
Date: Tue, 01 Mar 2011 12:18:46 GMT

＊＊＊　途中省略　＊＊＊

<html>
<head>
<title>Login Page for Tutorial Examples</title>
<body bgcolor="white">
<form method="POST" action='j_security_check;jsessionid=C64F7F42333E426B4BD52C42F4EBC817' >
  <table border="0" cellspacing="5">
    <tr>
      <th align="right">Username:</th>
      <td align="left"><input type="text" name="j_username"></td>
    </tr>
    <tr>
      <th align="right">Password:</th>
      <td align="left"><input type="password" name="j_password"></td>
    </tr>
    <tr>
      <td align="right"><input type="submit" value="Log In"></td>
      <td align="left"><input type="reset"></td>
    </tr>
  </table>
</form>
```

```
</body>
</html>
```

ここでの注意点は、3 つのヘッダ行でキャッシュをさせないようにしていること、及びクッキー・ヘッダ行が付いていることである。このクッキーはサーブレット・コンテナが付加したもので、そのユーザの認証が成功したら、その ID のセッション・オブジェクトにその情報を属性として付加することで、サーブレット・コンテナは要求にそのセッション ID のクッキーが到来したら、そのユーザは認証済みであることを判断できるようになる。またボディ部は login.jsp の出力がインクルードされている。そして Form のアクションにはセッション ID が付加されている。

2. この応答を受け、ブラウザは下図のようにログイン画面を表示する。

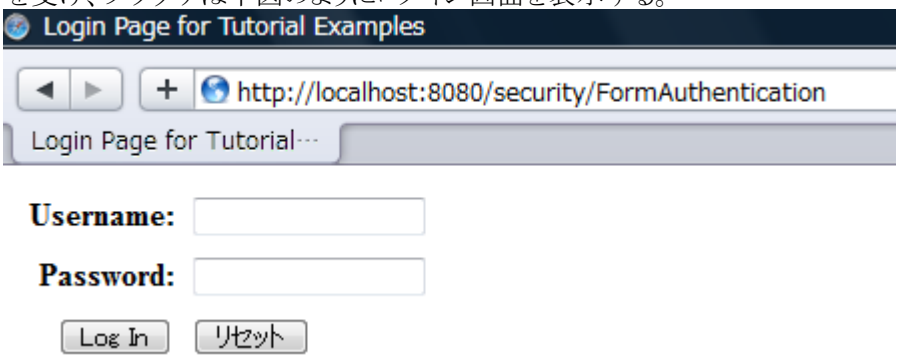


図 15-11：フォーム・ベースのログイン画面

3. Username に"sengoku"、Password に"sengoku38"を入力し、Log In ボタンをクリックすると、ブラウザは次のような HTTP 要求メッセージを送信する（注意：これはプロキシのログ）：

```
POST /security/j_security_check;jsessionid=C64F7F42333E426B4BD52C42F4EBC817 HTTP/1.1
Host: localhost:12345
Referer: http://localhost:12345/security/FormAuthentication
Accept:
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: ja-JP
Origin: http://localhost:12345
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja-JP) AppleWebKit/533.19.4 (KHTML,
like Gecko) Version/5.0.3 Safari/533.19.4
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Cookie: JSESSIONID=C64F7F42333E426B4BD52C42F4EBC817
Content-Length: 39
Connection: keep-alive

j_username=sengoku&j_password=sengoku38
```

ここでの注意点は、要求 URI が、/security/j_security_check;jsessionid=C64F7F42333E426B4BD52C42F4EBC817 と、なっていることで、サーブレット・コンテナはこの URL に含まれるセッション ID あるいはクッキー行に含まれるセッション ID から、この認証の為のデータを送信しているクライアントを特定できる。
j_username=sengoku と j_password=sengoku38 というログイン情報は POST としてボディ部に平文（URL エンコードされているが）のまま含められる。

4. サーブレット・コンテナがその認証データが正しいことを確認したら、クライアントに対しそのリソースへのアクセスを与える為に、次のようなリダイレクトの為の 302 応答をクライアントに返す（注意：これはプロキシのログ）。これによりクライアントに対し、このサーブレットの URL に新しい要求を送信させる：

```
HTTP/1.1 302 Moved Temporarily
Server: Apache-Coyote/1.1
Location: http://localhost:12345/security/FormAuthentication
Transfer-Encoding: chunked
Date: Tue, 01 Mar 2011 12:18:57 GMT

0
```

5. これを受けてクライアントのブラウザは次のように要求を送信する(注意:これはプロキシのログ):

```
GET /security/FormAuthentication HTTP/1.1
Host: localhost:12345
Origin: http://localhost:12345
Accept-Encoding: gzip, deflate
Accept-Language: ja-JP
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ja-JP) AppleWebKit/533.19.4 (KHTML,
like Gecko) Version/5.0.3 Safari/533.19.4
Accept:
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Referer: http://localhost:12345/security/FormAuthentication
Cookie: JSESSIONID=C64F7F42333E426B4BD52C42F4EBC817
Connection: keep-alive
```

6. サーブレットコンテナはその要求を受けたら、**オリジナルの要求オブジェクト**(認証の為の情報が付加された)を応答オブジェクトとともに引数として FormBasedAuthenticationTest サーブレットの service メソッドを呼ぶ。下図はこのサーブレットからの応答である:

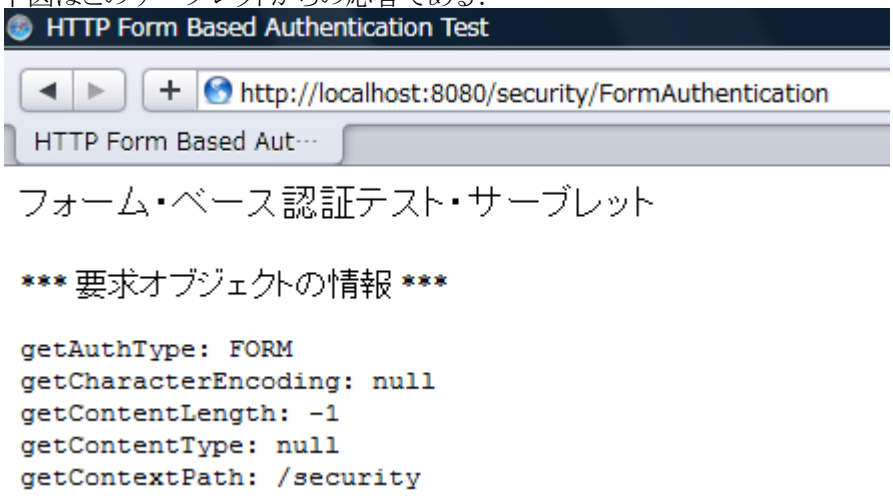


図 15-12:FormBasedAuthenticationTest サーブレットの応答

7. ユーザ名あるいは/及びパスワードが正しくないときは、以下のようにエラー・ページから再試行をユーザに促す:

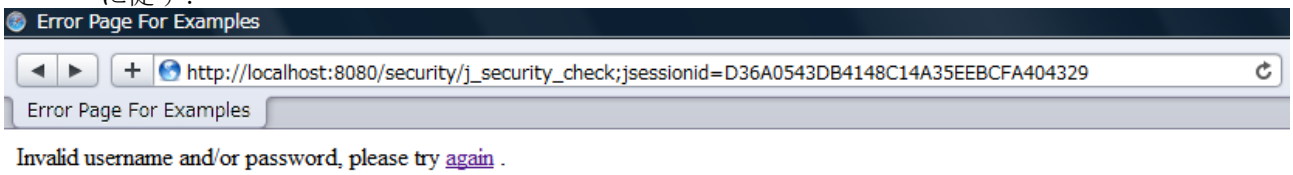


図 15-13:エラー画面

15.2.3.5 FormBasedAuthenticationTest サーブレットの出力

それではこのサーブレットの出力を更に詳しく見てみよう:

```
001 フォーム・ベース認証テスト・サーブレット
002
003 *** 要求オブジェクトの情報 ***
004 getAuthType: FORM
005 getCharacterEncoding: null
006 getContentLength: -1
007 getContentType: null
008 getContextPath: /security
009 getLocalAddr: 0.0.0.0
010 getLocale: ja_JP
011 getLocalName: 0.0.0.0
```

```
012 getLocalPort: 8080
013 getMethod: GET
014 getPathInfo: null
015 getPathTranslated: null
016 getProtocol: HTTP/1.1
017 getQueryString: null
018 getRemoteAddr: 0:0:0:0:0:0:0:1
019 getRemoteHost: 0:0:0:0:0:0:0:1
020 getRemotePort: 50274
021 getRemoteUser: sengoku
022 getRequestURI: /security/FormAuthentication
023 getRequestedSessionId: 2E9D3313089016CC50B6278B8C38DEF8
024 getSession(false): org.apache.catalina.session.StandardSessionFacade@eb67e8
025 session.getId: 2E9D3313089016CC50B6278B8C38DEF8
026 getUserPrincipal.getName: sengoku
027 isUserInRole("manager"): true
028 isRequestedSessionIdValid: true
029 isRequestedSessionIdFromCookie: true
030 isRequestedSessionIdFromURL: false
031 getScheme: http
032 isSecure: false
033 getServerName: localhost
034 getServerPort: 8080
035 getRealPath:
C:\eclipse\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\security\securi
ty\FormAuthentication
036 getServletPath: /FormAuthentication
037 getContextPath: /security
038 isAsyncSupported: false
039 isAsyncStarted: false
040
041 要求パラメタ(Request Parameters):
042
043 要求ヘッダ(Headers):
044   connection : keep-alive
045   accept-language : ja-JP
046   host : localhost:8080
047   accept :
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
048   user-agent : Mozilla/5.0 (Windows; U; Windows NT 6.0; ja-JP) AppleWebKit/533.19.4 (KHTML,
like Gecko) Version/5.0.3 Safari/533.19.4
049   accept-encoding : gzip, deflate
050
051 属性(Attributes):
052   org.apache.catalina.valves.AccessLogValve.t1 : 1298962027463
053   org.apache.catalina.ASYNC_SUPPORTED : false
054
055 クッキー(Cookies):
```

- 004 行目：getAuthType: FORM は、この要求が FORM 認証を通過したものであることを示している
- 021 行目：getRemoteUser: sengoku は、認証の結果判ったユーザ名を示す
- 023 行目：getRequestedSessionId: 2E9D3313089016CC50B6278B8C38DEF8 は、フォーム認証要求にはこのセッション ID が含まれていることを示す
- 024 行目：getSession(false): org.apache.catalina.session.StandardSessionFacade@eb67e8 は、このフォーム要求は、既にセッションが存在していることを示す
- 025 行目：session.getId: 2E9D3313089016CC50B6278B8C38DEF8 は、そのセッションの ID を示す
- 026 行目：getUserPrincipal.getName: sengoku は、認証の結果得たプリンシパルの名前の属性が sengoku であることを示す
- 027 行目：isUserInRole("manager"): true は、このユーザは manager というロールに属することを示す
- 028 行目：isRequestedSessionIdValid: true は、このセッション ID は有効であることを示す
- 029 行目：isRequestedSessionIdFromCookie: true は、このセッションがクッキーで維持されていることを示す、次の項で説明するように、クッキーを受け付けなくしたブラウザでは、セッションは URL ベースとなる
- 031 行目：getScheme: http は、この要求のスキームが HTTP であることを示す
- 032 行目：isSecure: false は、この要求は安全化されたチャンネルを介して送られていないことを示す

- 036 行目：getServletPath: /FormAuthentication は、このサーブレットのパスを示す
- 055 行目：クッキー(Cookies):はこの要求にはクッキーが無いことを示す。029 行目でセッションがクッキー経由であるのに、要求には cookie:ヘッダが無いのは奇異に感じるかも知れないが、**サーブレットに渡されるヘッダ情報は、実はサーブレット・コンテナが保管していたオリジナルの要求のヘッダの情報**だからである。

15.2.3.6 クッキーを無効にしたブラウザでの問題

以上でフォーム・ベースの認証のメカニズムは理解されたであろう。最後に、クッキーを受け付けなくしたブラウザでこのメカニズムの動作を調べることとする。ブラウザの設定は「クッキーを受け付けなくしたブラウザでのセッション確立の確認」の項で説明したので、それを見て頂きたい。

実験して判るように、この場合は Internet Explorer では正常に動作するが、**Mozilla Firefox、Google Chrome と Apple Safari では、2 回ユーザ名とパスワードを送信しなければならなくなる。**なぜだろうか？

これは、Tomcat が URL 書き換えでセッションが確立するの確認するために、302 応答を使って URL にセッション ID が付いたオリジナル要求を再度させている為だと考えられる。その際 Internet Explorer では、プライバシ設定を最大（「総てのクッキーをブロック」）にしても、あるいは「詳細設定」の「自動クッキー処理をオーバライドする」をともに「ブロックする」としても、cookie:ヘッダ行を付けて返されている。Internet Explorer で完全にクッキーをブロックするにはプライバシ設定ファイルをインポートしなければならない。一方 Mozilla Firefox、Google Chrome 及び Apple Safari では cookie:ヘッダ行が付加されない（そのほうが自然なのだが）ので、URL 書き換えによるセッション維持のためのステップが必要になっていると考えられる。

2 回のログインが成功したあとでは、セッションは URL ベースで維持されることは、このサーブレットの出力で確認される。

15.2.4 TLS (SSL)と CLIENT-CERT 認証

TLS (Transport Layer Security)は、その名のとおりトランスポート層を安全なものとする為の仕掛けであり、ネットワーキングのプロトコル下位層の TCP 層の上に介在させるプロトコルである。基になった技術が SSL (Secure Sockets Layer)であるため、むしろ SSL という言葉が良く使われる。詳細は別途説明する予定だが、とりあえずは Wikipedia などを参照されたい。

下図はそのプロトコルの位置づけである：

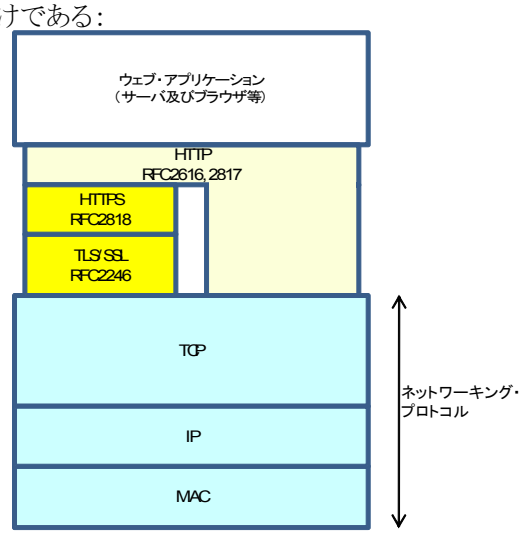


図 15-14：プロトコル構成

TLS は TCP の上に置かれ、また HTTP プロトコルとの仲介に HTTP プロトコルが存在し、また HTTP そのものも TLS 対応の為に RFC 2817 でアップグレードされている。TLS は接続型のプロトコルで、クライアントとサーバ間で安全(セキュア)な接続を確立し、その接続上でデータ(HTTP メッセージ)を交換する。TLS 対応のブラウザは TLS クライアントになり、サーバのしかるべき TCP ポート(デフォルトは 443、Tomcat は 8443) 上への接続を開始し、TLS ハンドシェイクを始めるために、TLS ClientHello メッセージを送る。TLS ハンドシェイクが完了すれば、接続が確立されたことになり、その接続上でクライアントは最初の HTTP 要求をサーバに送信できるようになる。

クライアントとサーバ間で安全(セキュア)な接続を確立ための TLS の主要な要素は、暗号化と認証である。認証に際してはサーバはサーバ証明書(Certificate)、あるいは加えてサーバ側が信頼する認証局のリストをブラウザに送信してそのサーバが合法であることを提示する。一般の TLS 対応のアプリケーションでは、お互いが正当かどうかに関しては、サーバは認証局(CA)が発行する認証局の認証済みサーバ証明書で確認されるが、クライアントは認証局が出す公開鍵を使うだけである。このままではサーバから見たクライアントは、信頼できる相手と確信できない。アプリケーションのレベルでこれまでの認証と同じようにパスワード認証を使うことが出来る。しかし、TLS ではその為のクライアント認証が用意されており、Tomcat ではこれを使用することもできる。

「クライアント認証」(CLIENT-CERT 認証)では、更に安全性を高める為に、サーバがクライアントに対し自分が誰であるかを証明させる。サーバに SSL で接続する際、クライアントに証明書を提示させ、接続元を認証する。通常、サーバーが信用する認証局が署名した電子証明書をクライアントが提示してはじめて SSL の接続が成立する。TLS 対応のサーバの殆どはこのクライアント認証をクライアントに要求していない。クライアント認証では、クライアントとサーバが相互に認証し合う必要がある。クライアント認証に関しては日本ベリサイン社の担当者が投稿した記事が判り易く説明されており、参考になろう。

クライアント認証は通常電子商取引や企業内網で使用されている。

クライアント認証を使わない TLS 接続を使ったときは、前述の 3 つのログイン(ベーシック、ダイジェスト、及びフォーム)のメカニズムのどれかを併用するのが一般的である。

15.2.4.1 TLS の基本シーケンス

TLS のシーケンスに関しては、日経 NETWORK の半沢智氏の記事が判り易く書かれているので、それを読んでいただいたほうが良かろう。

TLS の基本シーケンスは次のようである(出典:http://www.ehow.com/how_7664027_tls-ssl-tutorial.html):

1. ネットワーク上のノードのサーバとクライアントが証明書を交換することで互いを特定し確認する。証明書は VeriSign のような認証局によって発行される。サーバの ID、サーバの公開鍵、及び認証局の署名などを含んだこの証明書は、認証局が持っている秘密鍵によって暗号化されている。そうするとクライアントは認証局の公開鍵を使ってこの証明書を復号し、サーバの公開鍵を取得できる。公開鍵暗号方式では，一方の鍵で暗号化したデータは，ペアとなっているもう一方の鍵でしか復号できない(これを非対象アルゴリズム暗号方式という)。公開鍵で暗号データを正しく復号できた場合，その暗号データはペアとなっている秘密鍵を使って暗号化されたことが保証される。秘密鍵は，所有者だけが持っているはずなので，これによってその証明書は認証局が発行した証明書であることがわかる。これで、クライアントは証明書から取得したサーバの公開鍵が信用できるものであることが判るので、これを使ってサーバのメッセージを復号化出来る。同じようにサーバはクライアントの公開鍵を取得できる。ただウェブ・アプリケーションの場合は、クライアント(ブラウザ)がサーバを信頼できるものであるかどうかを判断するだけの方法が一般的である。従って、以下そのような手順を説明することとする。

2. クライアントはサーバに対して取得したサーバの公開鍵で暗号化されたランダムなデータを送る。クライアントは自分が生成したランダムなデータをもとに共通鍵(サーバ鍵とクライアント鍵)を生成する。サーバはそのデータを自分が取り出したサーバの公開鍵を使ってそのデータを暗号化する。サーバは受け

た暗号化されたデータを自分の秘密鍵を使って復号する。サーバは復号化されたランダム・データをもとにクライアントと同じ共通鍵を生成する。これでクライアントとサーバがともに同じ鍵（共通鍵）を持ったことになる。

3. クライアントとサーバは電子的にメッセージ交換時の暗号化と復号に使うアルゴリズムの折衝を行う。折衝されたアルゴリズムはクライアントとサーバの双方が受け付けられるものでなければならない。

4. 共通鍵を使って双方がデータの交換を行う。即ち、サーバはサーバ鍵を使って暗号化したデータをクライアントに送信し、クライアントはそれをサーバ鍵を使って復号する。クライアントはクライアント鍵を使って自分が送りたいデータを暗号化しそれをサーバに送り、サーバをそれを受けたらクライアント鍵を使って復号化する。

実際のネットワーク上の TCP レベルのシーケンスは RFC 2246 によれば以下のようになっている：

表 15-4：TLS のシーケンス

<table>
<tr><td>SSL クライアント
（ブラウザ）</td><td>SSL サーバ</td><td></td></tr>
<tr><td colspan="3">ユーザが https://で始まるアドレスをクリック</td></tr>
<tr><td colspan="2">SYN
⟶
TCP_Port = 443</td><td rowspan="3">このセッションには安全化された（secure)接続が必要である。クライアントは HTTPS TCP ポート番号 443 で TCP 接続を確立する</td></tr>
<tr><td colspan="2">SYN+ACK
⟵</td></tr>
<tr><td colspan="2">ACK
⟶</td></tr>
<tr><td colspan="3">新しい TCP 接続上での SSL ハンドシェイク</td></tr>
<tr><td colspan="2">HELLO_REQUEST
⟵</td><td>サーバは何時でもこのメッセージを送信し、ハンドシェイクの手順を最初からやり直すことをクライアントに要求できる</td></tr>
<tr><td colspan="2">CLIENT_HELLO
⟶
Highest SSL Version,
Ciphers Supported,
Data Compression Methods,
SessionId = 0,
Random Data</td><td>クライアントはハンドシェイクの最初にまず CLIENT_HELLO メッセージを送信するが、これには以下のものが含まれる：
• このクライアントが対応している最も新しい SSL と TLS のバージョン
• このクライアントが対応している暗号化アルゴリズムたち。これは好ましいもの順でリスト化されている
• このクライアントが対応しているデータ圧縮法たち
• このセッションの ID。もしそのクライアントが新しいセッションを開始するときはこのセッション ID は 0
• ランダム・データはクライアントが生成し、これはキー生成プロセスで使用される</td></tr>
<tr><td colspan="2">SERVER_HELLO
⟵
Selected SSL Version,
Selected Cipher,
Selected Data Compression Method,
Assigned Session Id,
Random Data</td><td>サーバは CLIENT_HELLO メッセージを受けてサーバはこの SSL セッションで使用する暗号方式などを決定し、SERVER_HELLO メッセージをクライアントに送信するが、これには以下のものが含まれる：
• この SSL セッションの為に使われる SSL または TLS のバージョン
• この SSL セッションで使う暗号化アルゴリズム
• この SSL セッションで使用するデータ圧縮
• この SSL セッションの為のセッション ID</td></tr>
</table>

	• サーバが生成したランダム・データで、キー生成プロセスで使われる
SERVER_CERTIFICATE ← Public Key, Authentication Signature	サーバはサーバ認証の為の SERVER_CERTIFICATE メッセージを送信する。このコマンドは次のものを含む: • このサーバの証明書 • オプションとしてこのサーバの証明書を割り当てた認証局の証明書で始まる証明書たちのチェイン
CLIENT_CERTIF_REQUEST ←	サーバは CLIENT_CERTIF_REQUEST メッセージを送信し、クライアント認証を求めることができる。
SERVER_HELLO_DONE ←	サーバは SERVER_HELLO_DONE メッセージを送信する。このコマンドはこのサーバがこの SSL ハンドシェイクのフェイズを完了したことを示す。
CLIENT_CERTIFICATE →	SERVER_HELLO_DONE メッセージを受けたらクライアントは最初にこのメッセージを送信できる。これはサーバがクライアント認証を求めたときにのみ送信される。このメッセージで送信されるものは SERVER_CERTIFICATE と似て以下のものとなる: • このクライアントの証明書 • オプションとしてこのクライアントの証明書を割り当てた認証局の証明書で始まる証明書たちのチェイン
サーバの証明書を検証する	
クライアントの証明書を検証する	
CERTIFICATE_VERIFY ←	サーバはクライアントに対しそのクライアントの証明書を検証したことを知らせる。このメッセージはクライアント認証が求められているときに送信される
CHANGE_CIPHER_SPEC →	クライアントは CHANGE_CIPHER_SPEC メッセージを送信する。このメッセージはこのセッション中にこのクライアントが送信するこれからの SSL データ・レコードの中身が暗号化されることを通知するものである。5 バイトからなる SSL レコード・ヘッダは暗号化されない
FINISHED →	クライアントは FINISHED メッセージを送信する。このメッセージはこの時点までのクライアントとサーバ間で交わされた総ての SSL ハンドシェイク・コマンドたちのダイジェストを含む。このメッセージにより、クライアントとサーバ間で暗号化されないでこれまで交わされたメッセージたちのどれもが途中で誰かによって変えられていないことを確認する為に送信される。
CHANGE_CIPHER_SPEC ←	サーバは CHANGE_CIPHER_SPEC メッセージを送信する。このメッセージはサーバが送信するこれからの SSL データ・レコードの中身が暗号化されることを通知するものである。
FINISHED ←	サーバは FINISHED コメッセージを送信する。このメッセージはこの時点までのクライアントとサーバ間で交わされた総ての SSL ハンドシェイクメッセージたちのダイジェストを含む
この時点で、クライアントはサーバの SSL 証明書のなかで受け取った公開鍵で暗号化した対象秘密鍵を送信できる。この暗号化された秘密鍵はサーバの秘密鍵でのみ復号化出来る。従ってサーバのみがこのメッセージを復号化出来る	

この表で赤で示したメッセージが CLIENT-CERT 認証に関わるものである。

15.2.4.2 Tomcat における CLIENT-CERT 認証の動作

Tomcat 7.0 の SSL 設定ガイドは筆者が翻訳したものをこの資料の「参考資料」の章に含めてあるので、そちらを参照されたい。また、Tomcat の旧バージョンの SSL の設定ガイドは、日本語化されているので、そちらを見て頂きたい。ただし Tomcat 7 の英文の設定ガイドでは、幾つかの追加と変更がされている。

Tomcat における CLIENT-CERT 認証は以下のように行われる:

- server.xml のなかで tomcatAuthentication="false"がセットされていると、Tomcat は単にその AJP(Apache JServ Protocol)コネクタからの要求から username を取り出し、また総ての認証が既になされていると見做す。
- もし tomcatAuthentication="true"がセットされていると、CLIENT-CERT によりそのアプリケーションの Context に org.apache.catalina.authenticator.SSLAuthenticator バルブが自動的に挿入される。要求が到来したときには、このバルブは以下のことを行う:
  - このバルブはこの要求に結び付けられたプリンシパルが既に存在するかどうかを調べ、もしそうなら、これは既に認証済みだと見做す。そうでないときは、認証レルムを呼び出す。
  - server.xml のなかでこの認証レルムが validate="true"と指定されているときは、この Realm はこの証明書を検証する。もし validate="false"がセットされていたときは、このバルブは証明書の検証チェックをスキップする。
  - 検証のステップが発生したあとで、このバルブはチェインのなかの最初の証明書から取り出した username 情報で、getPrincipal(username)を呼び出す。もしその username が DB のなかに存在するときは、この認証プロセスは通過することになる。パスワードのチェックは決してされない。

出典:http://www.tomcatexpert.com/knowledge-base/client-certificate-authentication-tomcat

これらの実際の使い方は後で説明することとする。また次の記事も参考になろう:
http://www.techscore.com/tech/J2EE/Servlet/11-5.html

15.3節 配備記述子による設定

セキュリティ設定は、Tomcat のようなサーブレット・コンテナを Apache のようなサーバに乗せる(ピギーバックする)場合と、スタンドアロンとして使う場合とで異なる。このチュートリアルはサーブレットを対象としているので、スタンドアロンでの設定を説明することとする。他のウェブ・サーバに乗せる場合は、Tomcat のコネクタ設定マニュアルを見て頂きたい。

15.3.1 レルム設定

セキュリティ設定では、まずレルム設定を行う必要がある。これは「レルム」の項で説明したように、Tomcat の場合は CATALINA_HOME/conf/server.xml ファイルの中で行う。但しアプリケーション毎にレルムを設定する場合は、web.xml に記述される。Eclipse 上では、このファイルはプロジェクト・エクスプローラ上の/Servers/Tomcat v7.0 Server at localhost-config/server.xml である。設定法は Tomcat のマニュアルを見て頂きたい。

設定は次のような構成となる:

```
<Realm className="... このレルム実装物のクラス名"
      ... この実装物の他の属性たち...>
```

この<Realm>要素は以下のコンテナ要素たちのどれかの中に配置できる。どのコンテナ要素(この場合はルート要素と呼ぶことがある)に含めるかでそのレルムの適用範囲(scope:スコープ、即ちどのウェブ・アプリケーションが同じ認証情報を共有するか)が決まる。

- <Engine>要素内:下位の<Host>あるいは<Context>要素内にレルムが組み入れられることで凌越されない限り、このレルムは総ての仮想ホスト上の総てのアプリケーションにわたって共有される。
- <Host>要素内:下位の<Context>要素内にレルムが組み入れられることで凌越されない限り、この仮想ホスト上の総てのウェブ・アプリケーションにわたってこのレルムは共有される。
- <Context>要素内:このレルムはこのウェブ・アプリケーションでのみ使用される。

この資料はサーブレット・エンジンのチュートリアルであるので、Tomcat に同梱されている server.xml のままで使われている UserDatabase のみを説明する。UserDatabase レルムはユーザー情報をファイルで管理する方法で、テスト用など一時的に利用する場合に便利な方式でであるが、商用には適さない。それ以外のレルムを使用するときは、Tomcat のマニュアル等を参照されたい。

Tomcat 7 に同梱されている server.xml には以下のような UserDatabase を定義する設定がある:

```
036   <GlobalNamingResources>
037     <!-- Editable user database that can also be used by
038          UserDatabaseRealm to authenticate users
039     -->
040     <Resource name="UserDatabase" auth="Container"
041               type="org.apache.catalina.UserDatabase"
042               description="User database that can be updated and saved"
043               factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
044               pathname="conf/tomcat-users.xml" />
045   </GlobalNamingResources>
```

また終りのほうに以下のような<Realm>要素が<host>要素内に挿入されている:

```
110       <!-- Use the LockOutRealm to prevent attempts to guess user passwords
111            via a brute-force attack -->
112       <Realm className="org.apache.catalina.realm.LockOutRealm">
113         <!-- This Realm uses the UserDatabase configured in the global JNDI
114              resources under the key "UserDatabase".  Any edits
115              that are performed against this UserDatabase are immediately
116              available for use by the Realm.  -->
117         <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
118                resourceName="UserDatabase"/>
119       </Realm>
```

従って、server.xml に何も手を加えなければ、この仮想ホスト上ではこの UserDatabase レルムが使用されそのパスは CATALINA_HOME/conf/tomcat-users.xml である。

tomcat-users.xml は以下のような構成となる:

- 各認可されたユーザは、ルート要素内に組み入れられた単一の<user>要素によって記述されねばならない
- 各<user>要素は、以下のような属性を持っていなければならない:
  - name:このユーザの username(このファイル内では単一でなければならない)
  - password:このユーザのパスワード(平文)
  - roles:このユーザに割り当てられているロール名たちのカンマで区切られたリスト

教材として使われている tomcat-users.xml は「フォーム・ベース認証のシーケンスの確認」の項で説明したように、以下のように<user>要素他が設定されているはずである:

```
<user name="sengoku" password="sengoku38" roles="administrator,manager"/>
  <role rolename="administrator"/>
  <role rolename="manager"/>
```

"sengoku"というユーザが"sengoku38"というパスワードを持ち、そのロールは"administrator"と"manager"である。

UserDatabase レルムは以下の規則に従って動作する:

- Tomcat が最初に起動するときに、このユーザたちのファイルから総ての定義されたユーザたちとそれに

結び付けられた情報をロードする。**このファイルになされた何らかの変更は Tomcat が再スタートするまでは反映されない**。変更は UserDatabase リソースからなされる。Tomcat にはその為に JMX を介してアクセスできる MBeans が用意されている。

- あるユーザが初めて保護されたあるリソースにアクセスしようとしたときは、Tomcat はこのレルムの authenticate()メソッドを呼ぶ。
- あるユーザが一旦認証されたら、そのユーザ(及びそのユーザに結び付けられたロールたち)は、そのユーザがログインしている期間は Tomcat 内にキャッシュされる。(**FORM ベースの認証の場合は、このことはそのセッションがタイムアウトあるいは無効化する前であることを意味する;BASIC 認証の場合は、このことはユーザが自分のブラウザを閉じるまでの期間であることを意味する**)。キャッシュされたユーザはセッションの直列化(serialization)にわたっては保管と読みだしはなされない。

15.3.2 セキュリティ制約の設定

この項はサーブレット 3.0 仕様書の 13.8 節「セキュリティ制約の指定」、および第 14 章「配備記述子」の 14.4 節の「配備記述子図」の 17-19 項を参照されたい。

セキュリティ制約は<security-constraint>要素で指定する。この要素はセキュリティ制約を課すリソースの集まりである<web-resource-collection>毎に認可制約<auth-constraint>及びユーザ・データ制約<user-data-constraint>とともにまとめられる。具体的には次のような構成となる:

表 15-5:セキュリティ制約の要素たち

セキュリティ制約 <security-constraint>		
ウェブ・リソース・コレクション <web-resource-collection>	URL パタン <url-pattern>	
	HTTP メソッド <http-method>または <http-method-omission>	http-method-omission がある URL パタンに対し設定されているときは、そこにリストされていないどのメソッドもセキュリティ制約がかかる。
認可の制約 <auth-constraint>	ロール名 <role-name>	"*"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
ユーザ・データ制約 <user-data-constraint>	トランスポート保障 <transport-guarantee>	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。**ほとんどの場合、INTEGRAL フラグまたは CONFIDENTIAL フラグは、SSL を使用する必要があることを示す**。

例えば acme というアプリケーションにおいて、次のようなセキュリティ制約が指定されていたとしよう:

```
001 <security-constraint>
002   <web-resource-collection>
003      <web-resource-name>precluded methods</web-resource-name>
```

```
004      <url-pattern>/*</url-pattern>
005      <url-pattern>/acme/wholesale/*</url-pattern>
006      <url-pattern>/acme/retail/*</url-pattern>
007      <http-method-exception>GET</http-method-exception>
008      <http-method-exception>POST</http-method-exception>
009   </web-resource-collection>
010   <auth-constraint/>
011 </security-constraint>
012
013 <security-constraint>
014   <web-resource-collection>
015      <web-resource-name>wholesale</web-resource-name>
016      <url-pattern>/acme/wholesale/*</url-pattern>
017      <http-method>GET</http-method>
018      <http-method>PUT</http-method>
019   </web-resource-collection>
020   <auth-constraint>
021      <role-name>SALESCLERK</role-name>
022   </auth-constraint>
023 </security-constraint>
024
025 <security-constraint>
026   <web-resource-collection>
027      <web-resource-name>wholesale 2</web-resource-name>
028      <url-pattern>/acme/wholesale/*</url-pattern>
029      <http-method>GET</http-method>
030      <http-method>POST</http-method>
031   </web-resource-collection>
032   <auth-constraint>
033      <role-name>CONTRACTOR</role-name>
034   </auth-constraint>
035   <user-data-constraint>
036      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
037   </user-data-constraint>
038 </security-constraint>
039
040 <security-constraint>
041   <web-resource-collection>
042      <web-resource-name>retail</web-resource-name>
043      <url-pattern>/acme/retail/*</url-pattern>
044      <http-method>GET</http-method>
045      <http-method>POST</http-method>
046   </web-resource-collection>
047   <auth-constraint>
048      <role-name>CONTRACTOR</role-name>
049      <role-name>HOMEOWNER</role-name>
050   </auth-constraint>
051 </security-constraint>
```

この場合は以下のような制約が課されることになる:

表 15-6: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

サーブレット・コンテナはある要求を受理したら、その要求 URL パタンから接続のタイプを調べそれが満たされていないときは HTTPS ポートにリダイレクトする。また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を満足しているかどうかを調べる。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)が返される。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、認証が引き起こされる。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る。

15.3.3 ログイン設定

これは既に「フォーム・ベース認証」の項で示した web.xml で出てきたものである:

```
<!-- Default login configuration uses form-based authentication -->
<login-config>
   <auth-method>FORM</auth-method>
   <realm-name>Protected Area</realm-name>
   <form-login-config>
      <form-login-page>/javascripts/security/protected/login.jsp</form-login-page>
      <form-error-page>/javascripts/security/protected/error.jsp</form-error-page>
   </form-login-config>
</login-config>
```

ログイン設定要素は次のような構成となっている:

表 15-7:ログイン設定の要素たち

ログイン設定 <login-conig>		
認証手段 <auth-method>		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配備設定する。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。
レルム名 <realm-name>		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。
FORM ログイン設定 <form-login-config>	FORM ログイン・ページ <form-login-page>	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。
	FORM エラー・ページ <form-error-page>	

15.4節 アノテーションによる指定

この節は、サーブレット 3.0 仕様書の 13.4 節の「プログラム的なアクセス制御アノテーション」をベースとしているので、不明な点はこの仕様書のこの節を読んでいただきたい。

@ServletSecurity アノテーションは、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だといえる。

アノテーションによるセキュリティは、このチュートリアルでは既に FormBasedAuthenticationTest.java クラスで以下のように使用していた。

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "manager"))
```

@ServletSecurity アノテーションは、以下の 2 つの要素で構成されている:

表 15-8 : 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	{}

デフォルトとなっている@HttpConstraint は@ServletSecurity アノテーションの中で使われ、、
httpMethodConstraints で返される HTTP メソッド以外のメソッドに対する制約であり、以下のような構成となっている:

表 15-9 : 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

もうひとつの要素である@httpMethodConstraint 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、以下のような構成となっている:

表 15-10 : 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

これらのアノテーションを使えば、セキュリティ制約がサーブレットごとに設定でき、また煩わしい配備記述子を編

集する必要もなくなる。また配備記述子の中で 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。即ち、配備記述子がアノテーションを凌駕し、metadata-complete=true では総てのアノテーションは無視される。

具体的な例はサーブレット 3.0 仕様書の 13.4.1.1 に示されているので、それを参考に@ServletSecurity アノテーションを記述すれば良い。

15.5節　プログラム的なセキュリティ

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、セキュリティ利用アプリケーションが使うことが出来る。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

これらのメソッドたちの詳細（API ドキュメントの翻訳）は、「添付資料」の「HttpServletRequest のなかのセキュリティ関係メソッドたち」の表にあるので、それらを見て頂きたい。

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。

15.5.1 プログラム的なログインのテスト

簡単なサーブレットで、プログラム的なログイン/ログアウト及びそのユーザ情報の読み出しを試してみよう。以下はそのコードである:

```
001 package tutorial_security;
002
003 import java.io.IOException;
004 import java.io.PrintWriter;
005
006 import javax.servlet.ServletException;
007 import javax.servlet.annotation.WebServlet;
008 import javax.servlet.http.HttpServlet;
009 import javax.servlet.http.HttpServletRequest;
010 import javax.servlet.http.HttpServletResponse;
011
012 @WebServlet(name="ProgrammaticLogin", urlPatterns={"/ProgrammaticLogin"})
013 public class ProgrammaticLogin  extends HttpServlet {
014   private static final long serialVersionUID = 1L;
015
016   /**
017    * 到来HTTP要求処理
018    */
019   public void performTask(HttpServletRequest req, HttpServletResponse res)
020   throws ServletException, IOException
021   {
022
023      res.setContentType("text/html; charset=Windows-31J");
024      PrintWriter out = res.getWriter();
025      out.println("<html>");
026      out.println("<head><title>Programmatic Login Test</title></head>");
027      out.println("<body>");
028      out.println("<big>プログラム的ログインのテスト</big><br><br>");
029
030      out.println("ログイン前" + "<br><br>");
031      out.println("IsUserInRole: "
032            + req.isUserInRole("manager")+"<br>");
033      out.println("getRemoteUser: " + req.getRemoteUser()+"<br>");
034      out.println("getUserPrincipal: "
035            + req.getUserPrincipal()+"<br>");
036      out.println("getAuthType: " + req.getAuthType()+"<br><br>");
037
038      // ログイン
039      try {
040         req.login("sengoku", "sengoku38");
041      } catch(ServletException ex) {
```

```
042         out.println("Login Failed with a ServletException: "
043               + ex.getMessage());
044         return;
045      }
046      out.println("ログイン後" + "<br><br>");
047      out.println("IsUserInRole: "
048            + req.isUserInRole("manager")+"<br>");
049      out.println("getRemoteUser: " + req.getRemoteUser()+"<br>");
050      out.println("getUserPrincipal: "
051            + req.getUserPrincipal().getName()+"<br>");
052      out.println("getAuthType: " + req.getAuthType()+"<br><br>");
053
054      // ログアウト
055      req.logout();
056      out.println("ログアウト後" + "<br><br>");
057      out.println("IsUserInRole: "
058            + req.isUserInRole("manager")+"<br>");
059      out.println("getRemoteUser: " + req.getRemoteUser()+"<br>");
060      out.println("getUserPrincipal: "
061            + req.getUserPrincipal()+"<br>");
062      out.println("getAuthType: " + req.getAuthType()+"<br><br>");
063
064      out.println("</body></html>");
065      out.close();
066   }
067
068   /**
069    * 到来HTTP GET / POST要求の処理
070    */
071   @Override
072   public void doGet(HttpServletRequest req, HttpServletResponse res)
073   throws javax.servlet.ServletException, java.io.IOException {
074      performTask(req, res);
075   }
076   @Override
077   public void doPost(HttpServletRequest req, HttpServletResponse res)
078   throws javax.servlet.ServletException, java.io.IOException {
079      performTask(req, res);
080   }
081
082 }
083
```

このサーブレットは 040 行目でプログラム的なログインを行っている。また 055 行目でプログラム的なログアウトを行っている。ログイン時の("sengoku", "sengoku38")という名前とパスワードのペアが正しければ、ログインは成功する。ログイン前、ログイン後、及びログオフ後の以下の情報がクライアントに返される:

- req.isUserInRole("manager")
- req.getRemoteUser()
- req.getUserPrincipal()または req.getUserPrincipal().getName()
- req.getAuthType()

このサーブレットを http://localhost:8080/security/ProgrammaticLogin でブラウザからアクセスすると、次のような応答が返される。

web.xml は FORM ベースの認証のテストの設定のままなので、ログインに成功したときの req.getAuthType()は"FORM"を返している。ログイン前とログオフ後では、req.getUserPrincipal()は null を返している。ログイン後では req.getUserPrincipal().getName()は"sengoku"を返している。

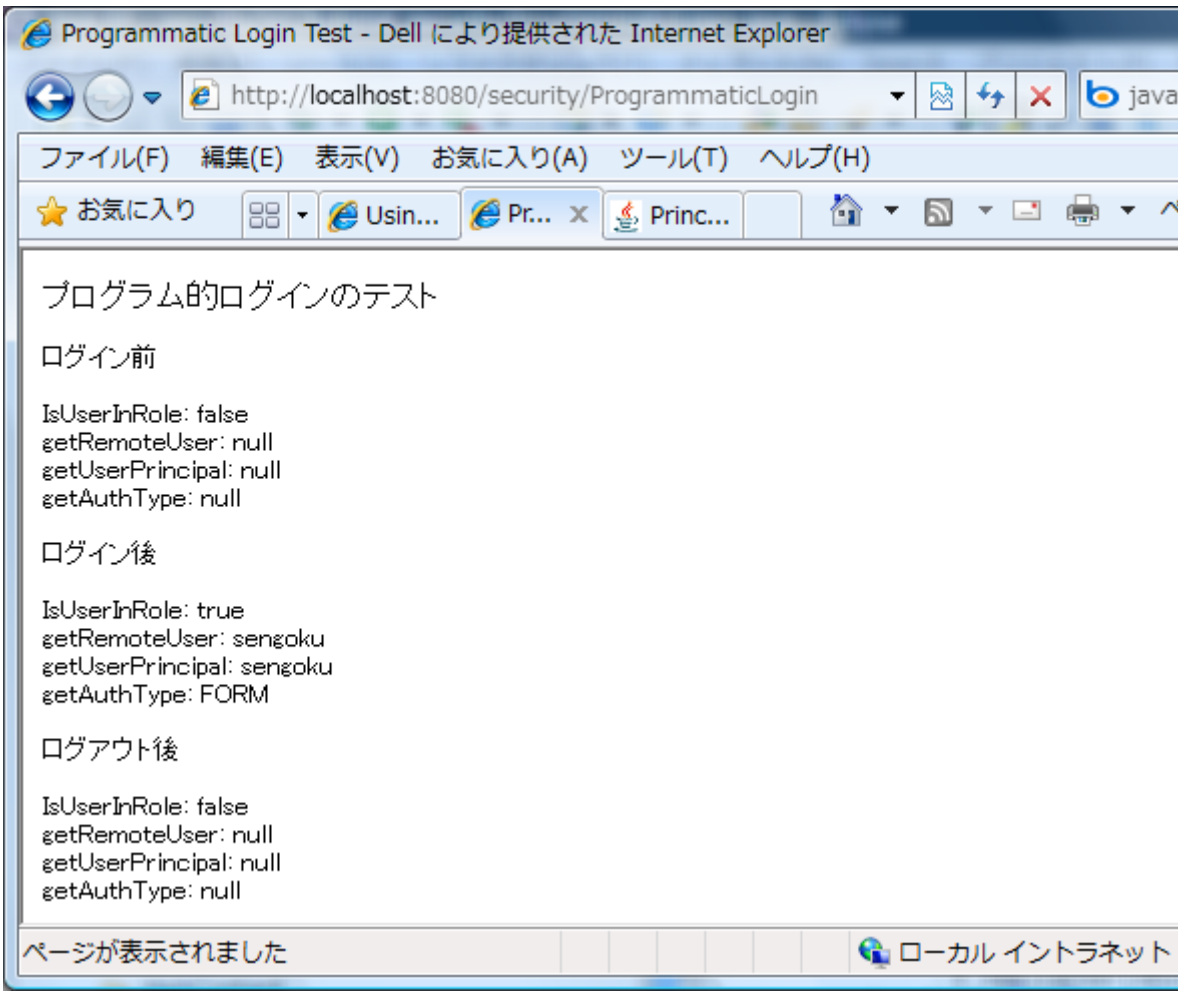


図 15-15：プログラム的なログイン・テストの結果（成功したとき）

それでは間違ったパスワードを指定すると、このサーブレットはどのような応答を返すだろうか？下図はその結果である。この場合は login メソッドは ServletException をスローする。これを受けてサーブレット・コンテナは下図のような応答メッセージを返している。

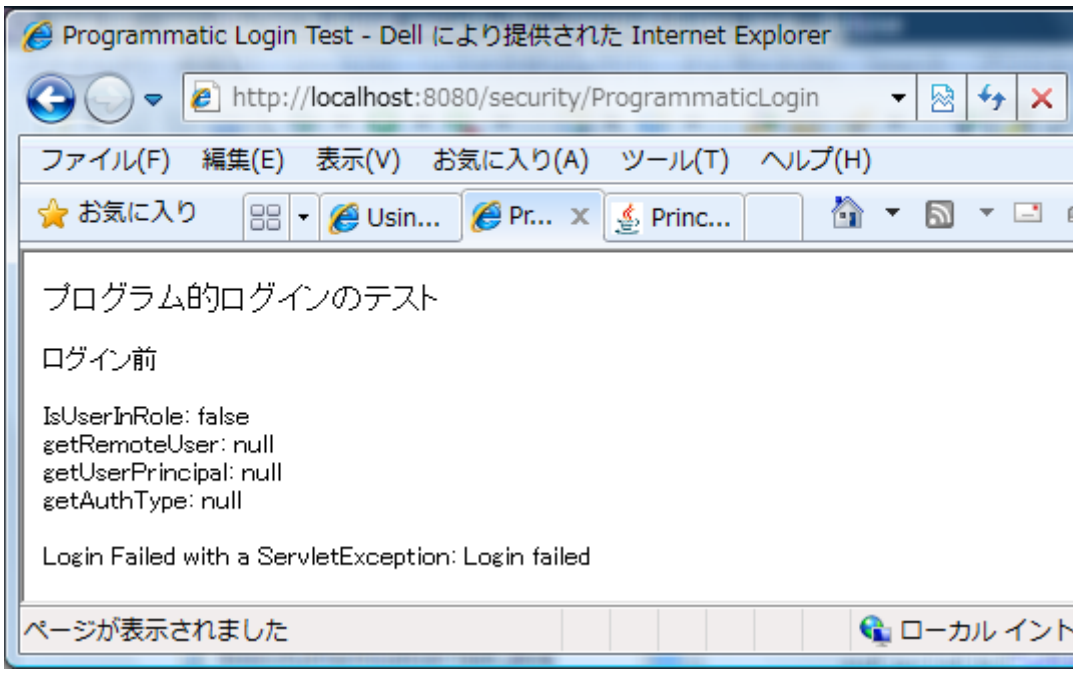


図 15-15：プログラム的なログイン・テストの結果（失敗したとき）

15.5.2 プログラム的な認証のテスト

プログラム的な認証もプログラム的なログインのテストのサーブレットと似たものとなる。異なっているのは 038-045 行の部分だけである。

```
001 package tutorial_security;
002
003 import java.io.IOException;
004 import java.io.PrintWriter;
005
006 import javax.servlet.ServletException;
007 import javax.servlet.annotation.WebServlet;
008 import javax.servlet.http.HttpServlet;
009 import javax.servlet.http.HttpServletRequest;
010 import javax.servlet.http.HttpServletResponse;
011
012 @WebServlet(name="ProgrammaticAuthentication", urlPatterns={"/ProgrammaticAuthentication"})
013 public class ProgrammaticAuthenticatipon extends HttpServlet {
014   private static final long serialVersionUID = 1L;
015
016   /**
017    * 到来HTTP要求処理
018    */
019   public void performTask(HttpServletRequest req, HttpServletResponse res)
020   throws ServletException, IOException
021   {
022
023      res.setContentType("text/html; charset=Windows-31J");
024      PrintWriter out = res.getWriter();
025      out.println("<html>");
026      out.println("<head><title>Programmatic Login Test</title></head>");
027      out.println("<body>");
028      out.println("<big>プログラム的認証のテスト</big><br><br>");
029
030      out.println("ログイン前" + "<br><br>");
031      out.println("IsUserInRole: "
032            + req.isUserInRole("manager")+"<br>");
033      out.println("getRemoteUser: " + req.getRemoteUser()+"<br>");
034      out.println("getUserPrincipal: "
035            + req.getUserPrincipal()+"<br>");
036      out.println("getAuthType: " + req.getAuthType()+"<br><br>");
```

```
037
038         // 認証
039         try {
040            req.authenticate(res);
041         } catch(ServletException ex) {
042            out.println("Login Failed with a ServletException: "
043                  + ex.getMessage());
044            return;
045         }
046         out.println("認証後" + "<br><br>");
047         out.println("IsUserInRole: "
048               + req.isUserInRole("manager")+"<br>");
049         out.println("getRemoteUser: " + req.getRemoteUser()+"<br>");
050         out.println("getUserPrincipal: "
051               + req.getUserPrincipal()+"<br>");
052         out.println("getAuthType: " + req.getAuthType()+"<br><br>");
053
054         // ログアウト
055         req.logout();
056         out.println("ログアウト後" + "<br><br>");
057         out.println("IsUserInRole: "
058               + req.isUserInRole("manager")+"<br>");
059         out.println("getRemoteUser: " + req.getRemoteUser()+"<br>");
060         out.println("getUserPrincipal: "
061               + req.getUserPrincipal()+"<br>");
062         out.println("getAuthType: " + req.getAuthType()+"<br><br>");
063
064         out.println("</body></html>");
065         out.close();
066      }
067
068      /**
069       * 到来HTTP GET / POST要求の処理
070       */
071      @Override
072      public void doGet(HttpServletRequest req, HttpServletResponse res)
073      throws javax.servlet.ServletException, java.io.IOException {
074         performTask(req, res);
075      }
076      @Override
077      public void doPost(HttpServletRequest req, HttpServletResponse res)
078      throws javax.servlet.ServletException, java.io.IOException {
079         performTask(req, res);
080      }
081
082   }
```

このサーブレットをブラウザから http://localhost:8080/security/ProgrammaticAuthentication でアクセスすると、まず FORM 認証の画面が下図のように呼び出される:

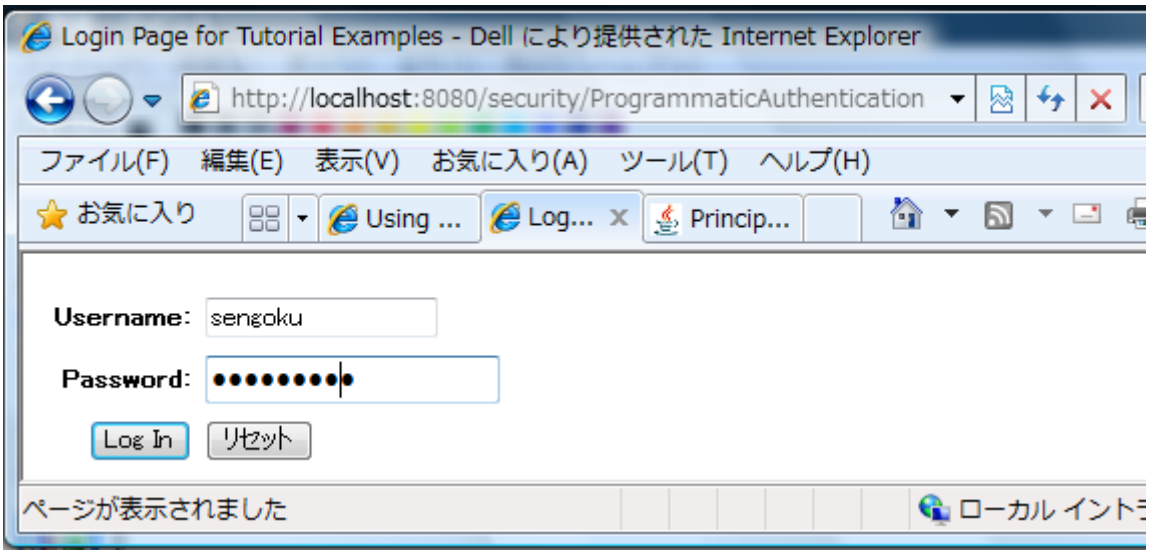


図 15-16：プログラム的な認証

ユーザ名とパスワードが正しければ、下図のような出力となる。正しくない場合は http://localhost:8080/security/j_security_check が呼び出される（「フォーム・ベース認証」の項を参照のこと）。なお再度ログインする為には error.jsp のなかの戻りのリンクを変更する必要がある。

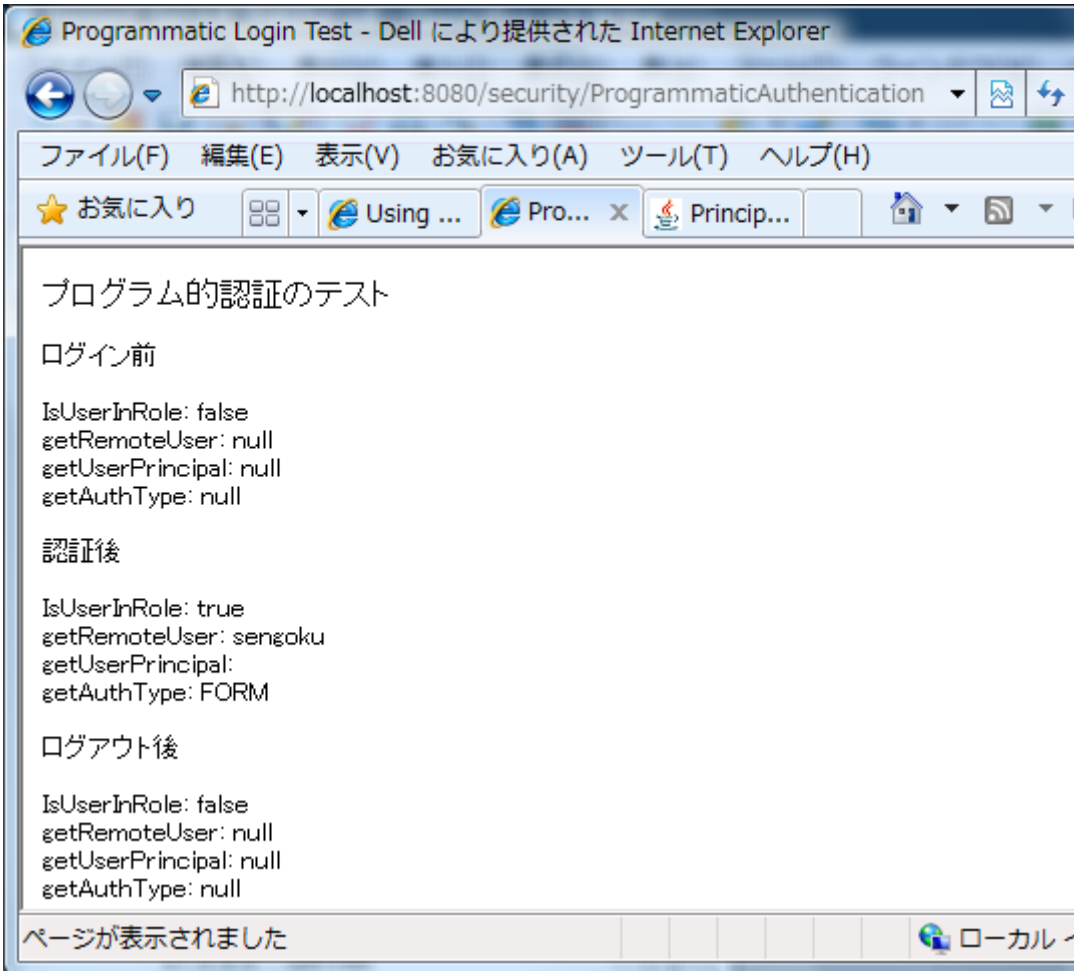


図 15-18：プログラム的な認証テスト・サーブレットの出力

15.6節　TLS (SSL) 設定

「TLS (SSL)とCLIENT-CERT 認証」の項で示したように、ウェブ・アプリケーションにTLS 接続を導入するには、サーブレット・コンテナがTLS 接続を扱う場合と、Tomcat の場合のようにウェブ・サーバ側で扱う場合があり得る。ここでは、サーブレット・コンテナが扱う場合の設定を解説する。

Tomcat 7.0 の SSL 設定ガイドは筆者が翻訳したものをこの資料の「参考資料」の章に含めてあるので、そちらを参照されたい。また、Tomcat の旧バージョンの SSL の設定ガイドは、日本語化されているので、そちらを見て頂きたい。ただし Tomcat 7 の英文の設定ガイドでは、幾つかの追加と変更がされている。

TLS の設定は、Eclipse 上ではなくて、まず DOS 上の Tomcat で行うことをお勧めする。その為に「Tomcat 7 のイ

ンストールと設定」の章で示したインストールと設定がなされている必要がある。

15.6.1 配備記述子による TLS 設定

「配備記述子による設定」の項で示したように、クライアントとの交信に SSL を使用するときは、ユーザ・データ制約<user-data-constraint>を CONFIDENTIAL とすることになる。一般的な配備記述子の記述は次のようなものとなろう:

```
<security-role>
  <role-name>administrator</role-name>
  <role-name>manager</role-name>
</security-role>

<!-- Define a constraint to restrict access to /security/ssl/* -->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>SSL Area</web-resource-name>
    <url-pattern>/security/ssl/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>

  <!-- Only administrator and manager can access this urls -->
    <role-name>administrator</role-name>
    <role-name>manager</role-name>
  </auth-constraint>
  <user-data-constraint>
    <!-- All access to this area will be SSL protected -->
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>

<!-- This application uses FORM authentication -->
<login-config>
     <auth-method>FORM</auth-method>
     <realm-name>Administrator or Manager Login</realm-name>
</login-config>
```

15.6.2 証明書キーストアを用意する

KeyStore というのは暗号化の鍵と証明書の格納場所である。単一の自己署名の証明書を含んだキーストアを新たに生成するには、Tomcat の SSL 設定マニュアルにあるように、keytool コマンドを利用する(「環境変数の設定」の節で示したように、自分のコンピュータの JavaJDK\bin にシステム環境変数の path が含まれていることを確認のこと)。

1. まず証明書をストアする為のディレクトリを作る
   例えば c:\ssl

2. DOS プロンプト(コマンド・ウィンドウ)を開いてそのディレクトリに移る
   cd c:\ssl

3. 以下のコマンドを入力する
   keytool -genkey -alias tomcat -keyalg RSA -keystore shop_cresc.jks
   ファイル名の"shop_cresc"は自分が保護したいドメインの名前にする。

4. 以下のようにプロンプトに答える:

```
C:\Users\****>cd c:\ssl

c:\ssl>keytool -genkey -alias tomcat -keyalg RSA -keystore shop_cresc.jks
キーストアのパスワードを入力してください:changeit
新規パスワードを再入力してください:changeit
姓名を入力してください。
  [Unknown]:  localhost
組織単位名を入力してください。
  [Unknown]:  Tech
組織名を入力してください。
  [Unknown]:  Cresc
都市名または地域名を入力してください。
  [Unknown]:  Mita
州名または地方名を入力してください。
  [Unknown]:  Tokyo
この単位に該当する 2 文字の国番号を入力してください。
  [Unknown]:  JP
CN=localhost, OU=Tech, O=Cresc, L=Mita, ST=Tokyo, C=JP でよろしいですか?
  [no]:  yes

<tomcat> の鍵パスワードを入力してください。
        (キーストアのパスワードと同じ場合は RETURN を押してください):

c:\ssl>
```

姓名を入力してくださいというプロンプトに対しては、自分の生命を入力してはならない。代わりにそれに対し証明書を受ける名前を入力する。具体的には www.yourdomain.com, mail.yourdomain.com, *.yourdomain.com などを入力する。ここではこのサーバは localhost なのでそれを入力する。

以上の操作で、現在のディレクトリにキーストア(keystore.jks)が生成されるはずである。

作成したキーストアの内容は以下に示すように、keytool の-list コマンドを使って確認できる:

```
cd c:\ssl

c:\ssl>keytool -list -v -keystore shop_cresc.jks
キーストアのパスワードを入力してください:

キーストアのタイプ: JKS
キーストアのプロバイダ: SUN

キーストアには 1 エントリが含まれます。

別名: tomcat
作成日: 2011/04/08
エントリタイプ: PrivateKeyEntry
証明連鎖の長さ: 1
証明書[1]:
所有者: CN=localhost, OU=Tech, O=Cresc, L=Mita, ST=Tokyo, C=JP
発行者: CN=localhost, OU=Tech, O=Cresc, L=Mita, ST=Tokyo, C=JP
シリアル番号: 4d9e9b7d
有効期間の開始日: Fri Apr 08 14:22:05 JST 2011 終了日: Thu Jul 07 14:22:05 JST 2011
証明書のフィンガープリント:
        MD5:  FC:38:4A:44:61:AE:D3:0B:5F:85:DF:53:90:AF:86:A3
        SHA1: 48:CD:3D:5B:1E:0E:09:3C:D9:18:EE:0B:D7:D8:C8:B2:DD:89:E0:77
        署名アルゴリズム名: SHA1withRSA
        バージョン: 3


*******************************************
*******************************************
```

```
c:\ssl>
```

15.6.3 server.xml の SSL コネクタの設定

Tomcat では server.xml には次のような記述があるはずである:

```
<!-- Define a SSL HTTP/1.1 Connector on port 8443
    This connector uses the JSSE configuration, when using APR, the
    connector should be using the OpenSSL style configuration
    described in the APR documentation -->
<!--
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
         maxThreads="150" scheme="https" secure="true"
         clientAuth="false" sslProtocol="TLS" />
-->
```

エディタを使ってこの部分のコメント・アウトを外して、以下のような変更をする必要がある:

```
<!-- Define a SSL HTTP/1.1 Connector on port 8443
    This connector uses the JSSE configuration, when using APR, the
    connector should be using the OpenSSL style configuration
    described in the APR documentation -->

<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
         port="8443" SSLEnabled="true"
         maxThreads="150" scheme="https" secure="true"
         clientAuth="false" sslProtocol="TLS"
         keystoreFile="c:\ssl\shop_cresc.jks" keystorePass="changeit" />
```

このチュートリアルではスタンドアロンでの動作を使うので、Java (JSSE)コネクタのみを使用するので、その為にブロッキング・モードの NIO コネクタを指定している。またキーストアの場所とパスワードを指定していることに注意されたい。詳細は「Tomcat 7 における SSL 設定」の章を見て頂きたい。

15.6.4 DOS 上での Tomcat 7 への SSL アクセス

これまでの作業で、スタンドアロンでポート番号 8443 で SSL アクセスが出来るようになっているはずである。

しかしながら、DOS プロンプト画面を使って Tomcat を起動し、ブラウザから https://localhost:8443/をアクセスすると、ブラウザは次のような警告を出す:

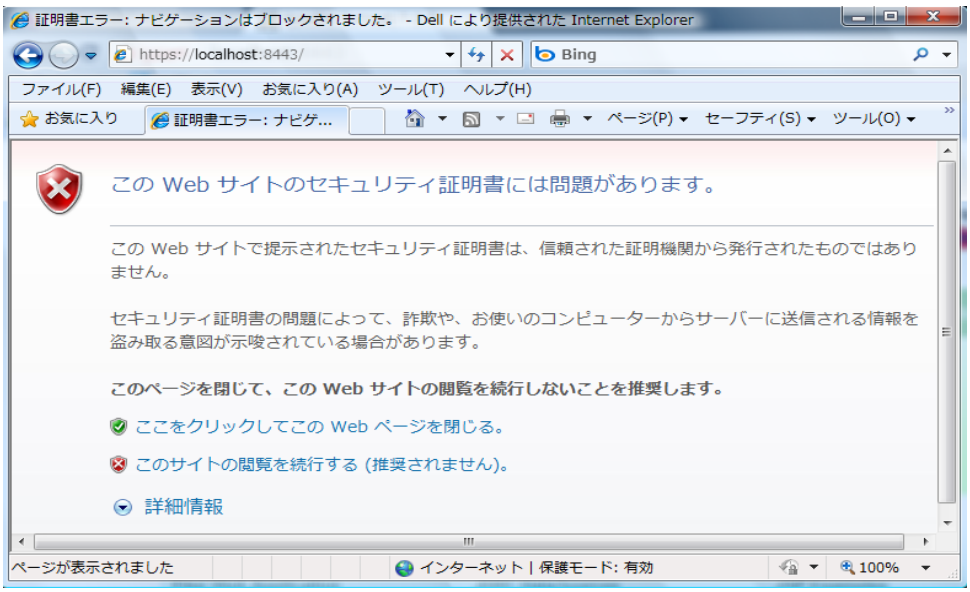


図 15-19：Tomcat への SSL アクセス(1)

この状態で続行を選択すると、下図のようにアドレス・バーに安全でないことをが表示された状態で、Tomcat のデフォルト・ページが表示される。

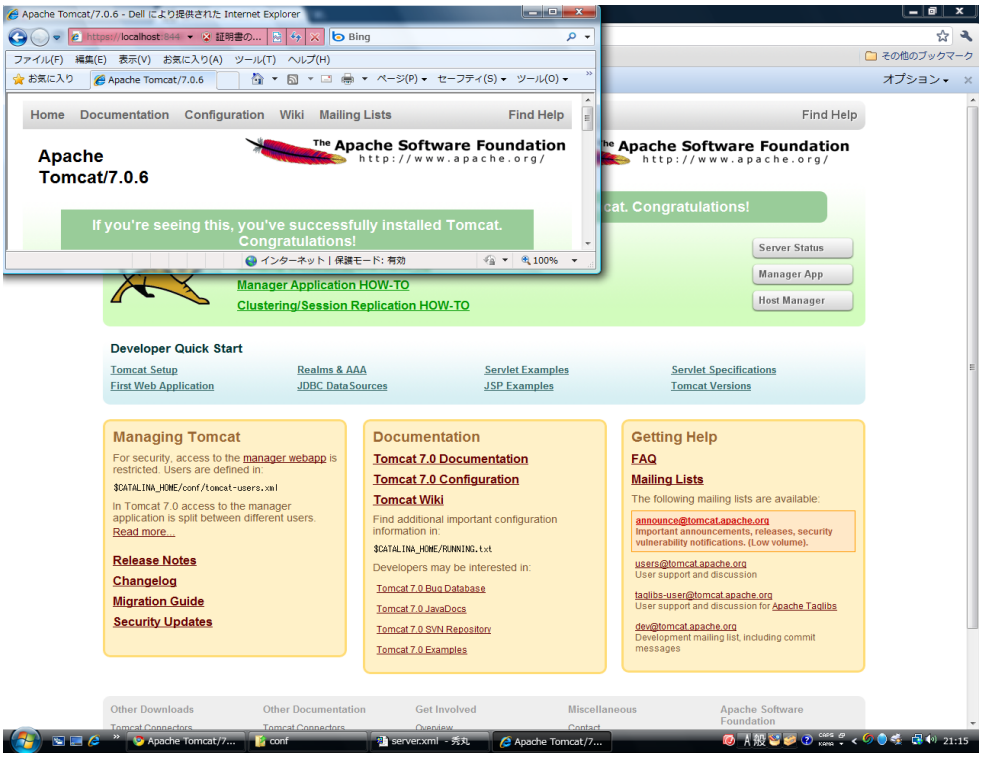


図 15-20：Tomcat への SSL アクセス(2)

これは作成したキーストアの内容が正式な認証局(CA)の証明がされていない為で、ブラウザがこれを検知したものである。このままでも SSL 接続の実験は進められるが、以下の項で説明するように、認証局からテスト用の証明書を取得するか、あるいは自己ルート証明書を用意するかしたほうが好ましい。

15.6.5 認証局からの証明書の取得

この状態で認証局からテスト用の証明書を受理することが可能である。その為には、自分のキーストアから認証局に対する証明書署名要求(CSR: Certificate Signing Request)を生成しなければならない。

1. c:\ssl のディレクトリから、次の keytool コマンドを実行する:
   keytool -certreq -alias tomcat -keyalg RSA -file shop_cresc.csr -keystore shop_cresc.jks

2. キーストアのパスワードを入力する:
   changeit

3. これで shop_cresc.csr ファイルが生成されている筈である。このファイルをテキスト・エディタで開くと、次のような内容になっている:


```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBnzCCAQgCAQAwXzELMAkGA1UEBhMCSlAxDjAMBgNVBAgTBVRva3lvMQ0wCwYDVQQHEwRNaXRh
MQ4wDAYDVQQKEwVDcmVzYzENMAsGA1UECxMEVGVjaDESMBAGA1UEAxMJbG9jYWxob3N0MIGfMA0G
CSqGSIb3DQEBAQUAA4GNADCBiQKBgQC9Ed07IjKC057ePypWqYOtBSl1INdqp8GDtU3RYcWKALru
2aSU9/E966kQHhHu+LFOG+go6dLI0bcwGeP4WL3IicOeACQp23g9YWDtQvcqqeKCaXFpFqH+/bHP
CRE1PS8Is8uFv3kdO03dDaTaUJzZlUD6k/lJUwAacS5OvWA2CQIDAQABoAAwDQYJKoZIhvcNAQEF
BQADgYEAdDDClwAOtjlX+f9dorh1VF8sKJdRU47YBJa2qGpUQeHeM/C76w8xuIZfy3jabMK5IMqF
rI3otz1ChZgOP9K++EN473TeyEecuhyxY8+EzSG0PZQUvUqUDGM6NMCBHae8g+GhRmgU2yuLz3a0
stic/vWc94H4BG9qb9eZz+4t+00=
-----END NEW CERTIFICATE REQUEST-----
```


この CSR ファイルの内容を認証局に送信して、CA 署名付きの正規の証明書を受領しなければならない。認証サービスは各社が行っているが、日本ベリサインでは無料でテスト用のセキュア・サーバ ID を提供しているので、これを使用する。詳細は同社の技術情報を見て頂きたい。これは 14 日間有効なものである。また、テスト用セキュア・サーバ ID の利用は、動作確認テストの用途に限られ、実稼動の商用サイトなどでは利用できないことに注意のこと。また、テスト用セキュア・サーバ ID で接続テストをする場合、専用のルート CA 証明書を、テストで用いるブラウザ全てにインストールする必要がある。

1. ベリサインのテスト用サーバ ID のサイトをアクセスする。

2. テスト用セキュア・サーバ ID 取得と手順に従って、必要な事項を入力する:ステップ 2 の CSR の提出では、テキスト・エディタで開いた shop_cresc.csr ファイルの内容をコピーし、CSR の貼り付けの個所に貼り付ける。

3. セキュア・サーバ ID の選択では、1024 ビット長を選択する。

4. ステップ 4 では、ブラウザにテスト用ルート証明書をインストールする。エディタを使ってテスト用ルート証明書を root_CA.cer といったファイルに保存する。次にブラウザでこのファイルを開く。VeriSign の説明に従ってインストールすれば良い。

5. テスト用セキュア・サーバ ID 用（公開鍵長　1,024bit) 中間認証局証明書を同じように intermediate_CA.cer といったファイルに保存する

6. 電子メールで Test Server ID (certificate)が送られてくるので、これもエディタを使って例えば localhost_CA.cer といった名前で保管する。

下図はこれらの証明書を c:\ssl にストアした例（エクスプローラ上の表示）である:

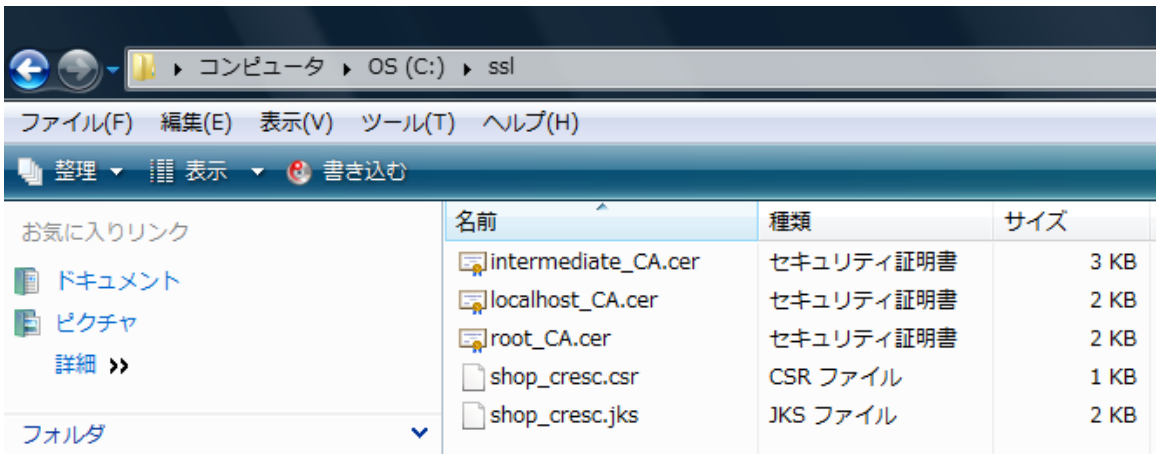


図 15-21：CA から取得した証明書たち

次にこの認証局から送られてきた CA 署名済みの中間証明書及びプライマリ証明書をキーストアにインストールする。

4. c:\ssl のディレクトリから、次の keytool コマンドを実行しルート証明書をインポートする：
   keytool -import -trustcacerts -alias root -file root_CA.cer -keystore shop_cresc.jks
   次のような結果が得られる：

```
c:\ssl>keytool -import -trustcacerts -alias root -file root_CA.cer -keystore shop_cresc.jks
キーストアのパスワードを入力してください:changeit
所有者: OU=For Test Purposes Only, OU=VeriSign Trust Network, OU=Terms of use at
https://www.verisign.com/cps/testca/ (c)02, OU=Class 3 TEST Public Primary Certification Authority
- G2, O="VeriSign, Inc.", C=US
発行者: OU=For Test Purposes Only, OU=VeriSign Trust Network, OU=Terms of use at
https://www.verisign.com/cps/testca/ (c)02, OU=Class 3 TEST Public Primary Certification Authority
- G2, O="VeriSign, Inc.", C=US
シリアル番号: 4ec0032f22e765316d20c19d4d723ce6
有効期間の開始日: Sat Apr 20 09:00:00 JST 2002 終了日: Wed Aug 02 08:59:59 JST 2028
証明書のフィンガープリント:
        MD5:  22:CC:83:73:97:80:18:55:6E:C5:B3:71:8C:65:A2:20
        SHA1: 64:0D:8F:12:34:B0:58:AA:DB:F5:CB:ED:C1:E5:31:AA:82:66:4B:EA
        署名アルゴリズム名: SHA1withRSA
        バージョン: 1
この証明書を信頼しますか? [no]:  yes
証明書がキーストアに追加されました。

c:\ssl>
```

5. c:\ssl のディレクトリから、次の keytool コマンドを実行し、中間証明書をインポートする：
   keytool -import -trustcacerts -alias intermediate -file intermediate_CA.cer -keystore shop_cresc.jks

```
c:\ssl>keytool -import -trustcacerts -alias intermediate -file intermediate_CA.cer -keystore
shop_cresc.jks
キーストアのパスワードを入力してください:changeit
証明書がキーストアに追加されました。

c:\ssl>
```

6. 最後に CA 署名済みサーバ証明書をインポートする：
   keytool -import -alias tomcat -file localhost_CA.cer -keystore shop_cresc.jks

```
c:\ssl>keytool -import -alias tomcat -file localhost_CA.cer -keystore shop_cresc
.jks
キーストアのパスワードを入力してください:
証明書応答がキーストアにインストールされました。

c:\ssl>
```

ブラウザにルート証明書のインポートがうまくいかない人は、下図のように Internet Explorer 上のヘルプから証明書マネージャを呼び出し、信頼されたルート証明機関の証明書を選択し、操作のインポートを選択する:

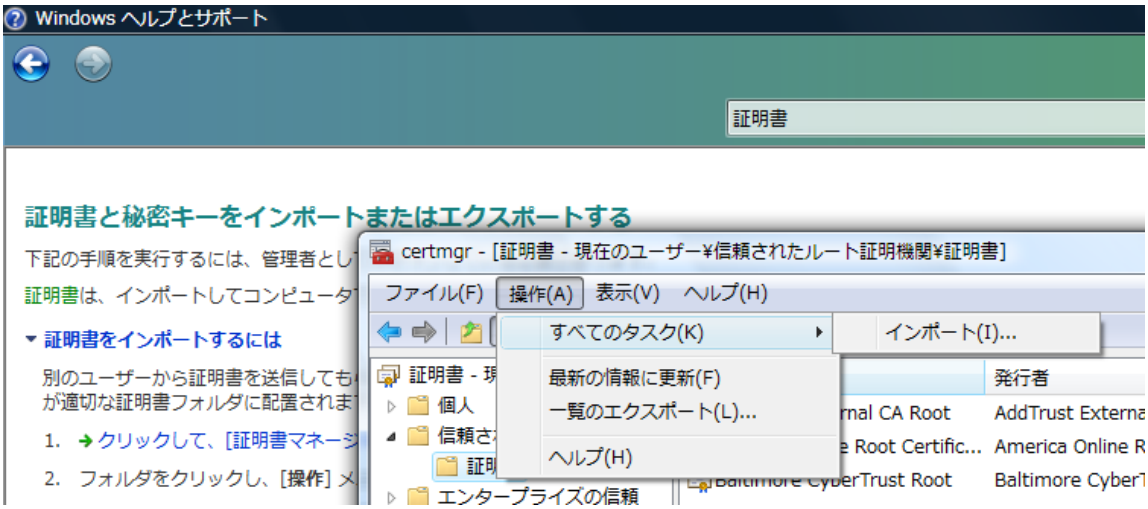


図 15-22:IE におけるルート証明書のインポート

Mozilla Firefox の場合は:

1. ツール→オプション→詳細→証明書を表示で証明書マネージャを開く
2. 認証局証明書のタブを開いてインポート・ボタンをクリック
3. 参照で c:\ssl\root_CA.cer ファイルを選択
4. この認証局による Web サイトの識別を信頼するを選択

Apple Safari の場合は:

1. https://localhost:8443/をアクセスすると識別情報を検証できないという警告のウインドウが開く
2. 証明書を表示のボタンをクリックすると証明書のウィンドウが開く
3. 証明書のインストールのボタンをクリックする
4. 証明書ストアとしては信頼されたルート証明機関を選択する
5. この証明書をインストールするかを聞いてくるので、はいを選択する

Google Chrome ブラウザの場合は:

1. オプションを選択
2. 高度な設定→セキュリティ→証明書の管理ボタンを押す
3. 信頼されたルート証明機関のタブを開く
4. インポートをクリックして証明書インポート・ウィザードの開始が表示される
5. 参照で c:\ssl\root_CA.cer ファイルを選択
6. 完了が表示されるまで次へをクリック
7. 同じことを中間証明機関でも c:\ssl\intermediate_CA.cer 行う

これで認証局からの証明書の取得と、ブラウザへのルート証明書のインポートが終了した。従ってこの状態で DOS 上で Tomcat 7 を起動し、インポートしたブラウザから https://localhost:8443/をアクセスすると、ブラウザは警告を出すことなく下図のように Tomcat 7 のデフォルト・ページを表示する。アドレス・バーの右の鍵のアイコンをクリックするとこのサイトの認証状況が表示される。

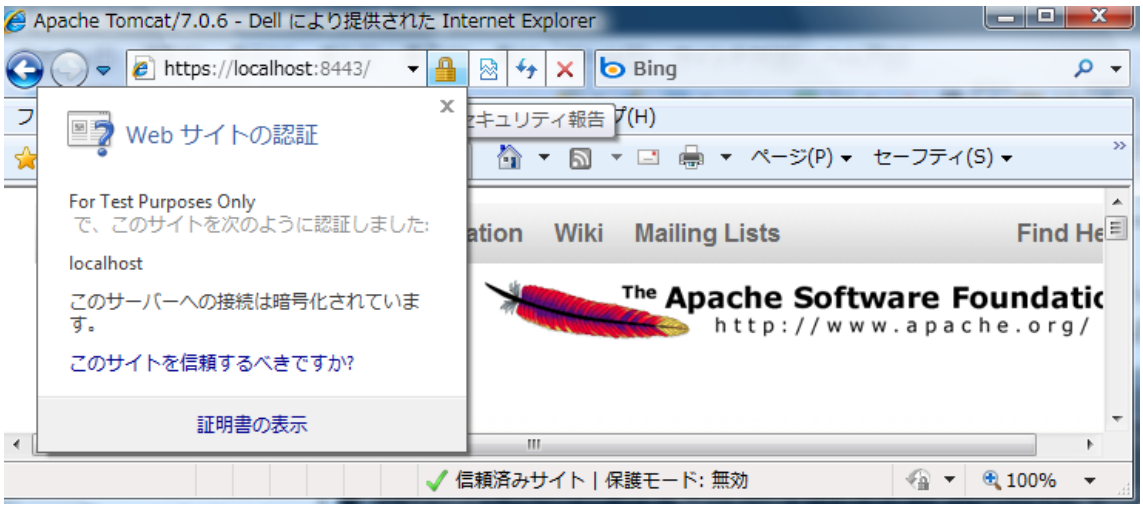


図 15-23 : VeriSign のテスト用証明書を使った SSL アクセス例

15.6.6 自己証明書の作成とインポート

単にテストの為だけの目的の場合は、ベリサインのような公的な認証機関からの証明書ではなくて、自己証明書を使うことも出来る。この場合はルート証明書をブラウザにインストールする必要がある。

1. 以下のように DOS プロンプトを使って自己証明書を作成する:

```
C:\ssl>keytool -selfcert -alias tomcat -validity 90 -keystore
shop_cresc.jks
キーストアのパスワードを入力してください: changeit

C:\ssl>
```

ここに、-validity は有効期間を日で指定する。またパスワードは changeit を入力する
これにより alias tomcat は新しい証明連鎖によって置き換えられる。

2. 以下のようにルート証明書をエクスポートする:

```
C:\ssl>keytool -export -alias tomcat -file c:\ssl\root_self.cer
-keystore shop_cresc.jks
キーストアのパスワードを入力してください: changeit
証明書がファイル <c:\ssl\root_self.cer> に保存されました。

C:\ssl>
```

3. この証明書を前回と同じようにブラウザにインポートする。下図は Google Chrome にインポートしたときの例で、localhost という発行者で登録されている:

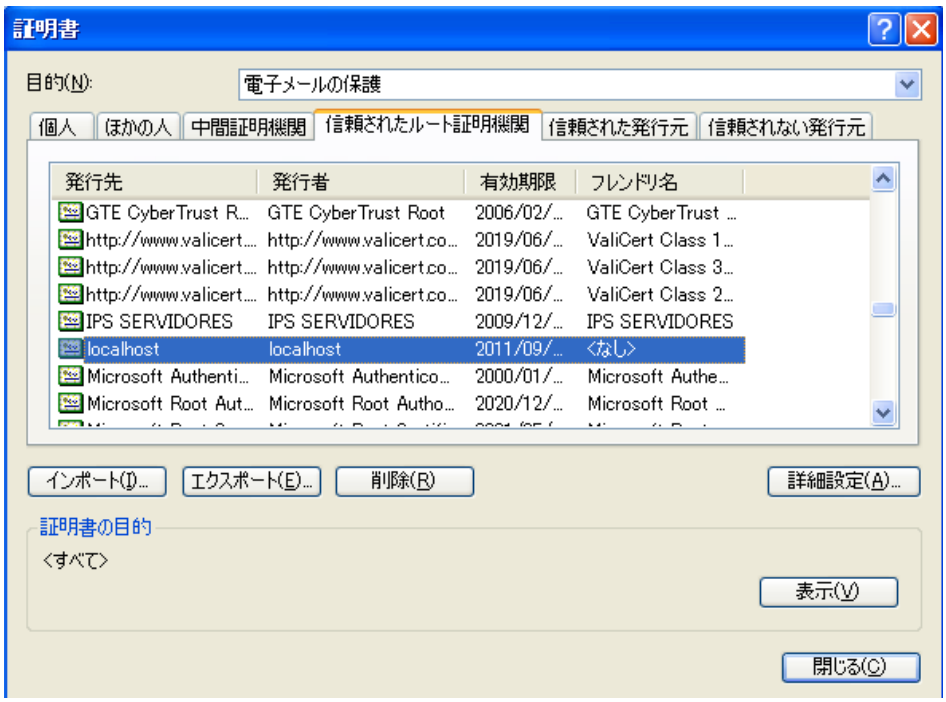


図 15-24：自己ルート証明書のインポート(Chrome の場合)

15.6.7 Eclipse 上での動作確認

Eclipse 上の server.xml は DOS 上の Catalina_home\conf\server.xml とは別のものなので、Eclipse のパッケージ・エクスプローラ上の servers\Tomcat v7.0 Server at localhost\server.xml を開き、その SSL コネクタ設定個所を DOS のそれに合わせて以下のように変更する：

```
  <!-- Define a SSL HTTP/1.1 Connector on port 8443
       This connector uses the JSSE configuration, when using APR, the
       connector should be using the OpenSSL style configuration
       described in the APR documentation -->

 <Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
           port="8443" SSLEnabled="true"
           maxThreads="150" scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLS"
           keystoreFile="c:\ssl\shop_cresc.jks" keystorePass="changeit" />
```

この状態で Tomcat を開始させ、テスト用のルート証明書をインポートしたブラウザから **https://localhost:8443/tutorial/HelloWorld** とアクセスし、正常に HelloWorld サーブレットが HTTPS 接続でアクセスできることを確認する。この際 Tomcat 7 のコンソールには以下のように NIO スレッドからのアクセス・ログが出力される。

```
75920 ["http-nio-8443"-exec-1] INFO filters.AccessLogFilter - Servlet /HelloWorld, IP 127.0.0.1,
Time....
```

15.7節 SSL 接続上のサーブレット

これまでの手順を終了したら、より具体的な SSL 接続上のサーブレットを学習できるようになる。

15.7.1 SSL セッション追跡モードのセット

サーブレット 3.0 ではセッション追跡に SSL セッション ID が使えるようになっている。その為には「Tomcat 7 における SSL 設定」の節で示されているように、幾つかの制限が存在する。

即ち:

- Tomcat は isSecure 属性が true にセットされたコネクタを持っていなければならない
- もし SSL 接続がプロキシあるいはハードウエアのアクセラレータで管理されているときは、SSL セッション ID が Tomcat にとって可視である為には、これらの SSL 接続は SSL 要求ヘッダたちを取り込んでいなければならない
- SSL セッション ID は各ノードによって異なる為、Tomcat がその SSL 接続を終了したときは、セッション・レプリケーションが使えなくなる

Tomcat 7 では、SSL セッション追跡は現在 BIO 及び NIO コネクタ用に実装されている。APR コネクタ用は未だ実装されていないことに注意が必要である。本チュートリアルでは NIO コネクタを使用している。

SSL セッション追跡が使えるようにするには、コンテキスト・リスナを使い、**そのコンテキストの為の追跡モードを SSL だけにセットする必要がある**(もし他の追跡モードが可能となっていると、そちらが優先して使われる)。これは以下のようなコードとなろう:

ServletContextInitializer.java

```
001 import java.util.HashSet;
002 import java.util.Set;
003
004 import javax.servlet.ServletContext;
005 import javax.servlet.ServletContextEvent;
006 import javax.servlet.ServletContextListener;
007 import javax.servlet.SessionTrackingMode;
008 import javax.servlet.annotation.WebListener;
009
010 /**
011  * ServletContextInitializerは、セッション、及びサーブレット・コンテキストに関する
012  * プログラムによる設定法を学習するためのクラス
013  * @author Cresc Corps.
014  * @December 2010, CRESC Corps.
015  */
016
017 @WebListener
018 public class ServletContextInitializer implements ServletContextListener{
019
020   @Override
021   public void contextDestroyed(ServletContextEvent arg0) {
022   }
023
024   @Override
025   public void contextInitialized(ServletContextEvent servletContextEvent) {
026      ServletContext cxt = servletContextEvent.getServletContext();
027      Set<SessionTrackingMode> stms = cxt.getDefaultSessionTrackingModes();
028      System.out.print("DefaultSessionTrackingMode for the security application: ");
029      for (SessionTrackingMode c : stms)
030         System.out.print(c + ", ");
031      System.out.println();
032      stms = new HashSet<SessionTrackingMode>();
033      stms.add(SessionTrackingMode.SSL);
034      cxt.setSessionTrackingModes(stms);
035      System.out.print("EffectiveSessionTrackingMode for the security application: ");
036      for (SessionTrackingMode c : stms)
037         System.out.print(c + ", ");
038      System.out.println();
039   }
040 }
```

このコードは「セッション」の章の「setSessionTrackingModes による方法」の項で示した tutorial のアプリケーションの同じ名前のクラスと殆ど同じで、033 行でセッション追跡モードを SSL のみにしているところが異なるだけである。

このクラスを security アプリケーションのデフォルト・パッケージに置いて、Tomcat 7 を起動させると、コンソールには次のメッセージが出力される:

```
DefaultSessionTrackingMode for the security application: COOKIE, URL,
EffectiveSessionTrackingMode for the security application: SSL,
```

15.7.2 SslSessionTest サーブレット

SSL 接続上のセッションの動作を確認する為の簡単なサーブレットを以下に示す:

SslSessionTest.java

```
001 package tutorial_security;
002
003 import java.io.IOException;
004 import java.io.PrintWriter;
005 import java.util.Date;
006 import java.util.Enumeration;
007 import java.util.Set;
008
009 import javax.servlet.ServletContext;
010 import javax.servlet.ServletException;
011 import javax.servlet.SessionTrackingMode;
012 import javax.servlet.annotation.ServletSecurity;
013 import javax.servlet.annotation.HttpConstraint;
014 import javax.servlet.annotation.ServletSecurity.TransportGuarantee;
015 import javax.servlet.annotation.WebServlet;
016 import javax.servlet.http.Cookie;
017 import javax.servlet.http.HttpServlet;
018 import javax.servlet.http.HttpServletRequest;
019 import javax.servlet.http.HttpServletResponse;
020 import javax.servlet.http.HttpSession;
021
022 /**
023  * Tomcat での SSL セッションのテスト用サーブレット
024  *
025  * このサーブレットをブラウザから以下のように呼び出す：
026  *   https://localhost:8443/security/sslsession/SslSessionTest
027  *
028  * このアノテーションと等価な配備記述子のセキュリティ制約：
029  * <security-constraint>
030  *   <web-resource-collection>
031  *   <web-resource-name>SSL Session Test Area</web-resource-name>
032  *   <url-pattern>/sslsession/*</url-pattern>
033  *   </web-resource-collection>
034  *   <user-data-constraint>
035  *   <!-- All access to this area will be SSL protected -->
036  *   <transport-guarantee>CONFIDENTIAL</transport-guarantee>
037  *   </user-data-constraint>
038  * </security-constraint>
039  *
040  * @author Cresc Co., Ltd.
041  */
042
043 @ServletSecurity(@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL))
044 @WebServlet(name="SslSessionTest", urlPatterns={"/sslsession/SslSessionTest"})
045 public class SslSessionTest extends HttpServlet {
046   private static final long serialVersionUID = 1L;
047
048
049   /**
050    * 到来 HTTP 要求処理
051    */
052   public void performTask(HttpServletRequest req, HttpServletResponse res)
053   throws ServletException, IOException
054   {
```

```
055      ServletContext context = getServletContext();
056      int visitTimes = 1;     // 訪問回数
057
058      // セッションの取得または生成と訪問回数のセット
059      HttpSession session = req.getSession();
060      if (session.getAttribute("visitTimes") != null){
061         visitTimes = (Integer)session.getAttribute("visitTimes") + 1;
062      }
063      session.setAttribute("visitTimes", (Integer)visitTimes);
064
065      // クッキーが使えないときにURL書き換えを行う
066      String urlBase = req.getContextPath() + req.getServletPath();
067      String url = res.encodeURL(urlBase);
068
069      // セッション無効のコマンドの受付（セッションのみを無効化）
070      if ((req.getParameter("reset") != null) ){
071         session.invalidate();
072      }
073
074      // SSLセッションも終了させる場合（TomcatのBIOとNIOのコネクタでのみ有効）
075      if ((req.getParameter("resetssl") != null) ){
076         session.invalidate();   // まずSessionを無効化
077         org.apache.tomcat.util.net.SSLSessionManager mgr =
078            (org.apache.tomcat.util.net.SSLSessionManager)
079            req.getAttribute("javax.servlet.request.ssl_session_mgr");
080         mgr.invalidateSession();   // SSLセッション・マネージャによるSSLセッションの無効化
081         res.setHeader("Connection", "close"); // TCP接続も切断させる
082      }
083
084
085      // 応答メッセージの作成
086      res.setContentType("text/html; charset=windows-31j");// 応答ヘッダContent-Type追加
087      res.setHeader("Cache-Control", "no-cache"); // キャッシュを殺しておかないと分かりにくくなる
088      PrintWriter out = res.getWriter();                  // 出力バッファ取得
089      out.println("<HTML><HEAD><TITLE>");
090      out.println("簡単なセッションのテスト</TITLE>");
091      out.println("<META HTTP-EQUIV=\"cache-control\" CONTENT=\"no-cache\">" +
092      "<META HTTP-EQUIV=\"content-type\" CONTENT=\"text/html; charset=Windows-31J\"></HEAD>");
093      out.println("<BODY><H1>SSL接続上のセッションのテスト</H1><BR>");
094      // セッション情報の出力
095      dumpSession(req, out, context);
096      out.println("<BR><BR>このセッションでの現在の訪問回数：" + visitTimes);
097      out.println("<BR><form method=\"get\" action=\"" + url + "\">" +
098            "<input type=\"submit\" name=\"proceed\" value=\"次へ\">" +
099      "<input type=\"submit\" name=\"reset\" value=\"セッションのリセット\">" +
100      "<input type=\"submit\" name=\"resetssl\" value=\"SSLセッションのリセット\">" +
101      "</form>");
102      out.println("</BODY></HTML>");
103      out.flush();
104   }
105
106   /**
107    * 到来HTTP GET / POST要求の処理
108    */
109   @Override
110   public void doGet(HttpServletRequest req, HttpServletResponse res)
111   throws javax.servlet.ServletException, java.io.IOException {
112      performTask(req, res);
113   }
114   @Override
115   public void doPost(HttpServletRequest req, HttpServletResponse res)
116   throws javax.servlet.ServletException, java.io.IOException {
117      performTask(req, res);
118   }
119
120   /**
121    * セッション情報の出力
122    */
123   public void dumpSession(HttpServletRequest req, PrintWriter out, ServletContext context)
throws IOException {
124
```

```
125      out.append("<br>＊要求オブジェクトのなかのセッション関連情報");
126      out.append("<br>getMethod: " + req.getMethod());
127      out.append("<br>getPathInfo: " + req.getPathInfo());
128      out.append("<br>getPathTranslated: " + req.getPathTranslated());
129      out.append("<br>getProtocol: " + req.getProtocol());
130      out.append("<br>getQueryString: " + req.getQueryString());
131      out.append("<br>getEffectiveSessionTrackingModes:");
132      Set<SessionTrackingMode> stms = context.getEffectiveSessionTrackingModes();
133      for (SessionTrackingMode c : stms) out.append("  " + c);
134      out.append("<br>getRequestedSessionId: " + req.getRequestedSessionId());
135      out.append("<br>sslID: " + (String)req.getAttribute("javax.servlet.request.ssl_session"));
136      out.append("<br>isRequestedSessionIdValid: " + (new
Boolean(req.isRequestedSessionIdValid())).toString());
137      out.append("<br>isRequestedSessionIdFromCookie: " + (new
Boolean(req.isRequestedSessionIdFromCookie())).toString());
138      out.append("<br>isRequestedSessionIdFromURL: " + (new
Boolean(req.isRequestedSessionIdFromURL())).toString());
139      out.append("<br>isSecure: " + (new Boolean(req.isSecure())).toString());
140      out.append("<br>getServletPath: " + req.getServletPath());
141      out.append("<br>getContextPath: " + req.getContextPath());
142      out.append("<br>");
143
144      out.append("<br>＊クッキー(Cookies):");
145      Cookie[] cookies = req.getCookies();
146      if ((cookies != null) && (cookies.length > 0)) {
147         for (int i = 0; i < cookies.length; i++) {
148            String name = cookies[i].getName();
149            String value = cookies[i].getValue();
150            out.append("<br>  " + name + " : " + value);
151         }
152      }
153
154      HttpSession session = req.getSession(false);
155      if (session != null) {
156         out.append("<br><br>＊その要求に対応するセッションの情報:");
157         out.append("<br>isNew: " + session.isNew());
158         out.append("<br>getId: " + session.getId());
159         out.append("<br>getCreationTime: " + new Date(session.getCreationTime()));
160         out.append("<br>getLastAccessedTime: " + new Date(session.getLastAccessedTime()));
161         out.append("<br>getMaxInactiveInterval in sec.: " + session.getMaxInactiveInterval());
162         out.append("<br>getServletContext: " +
session.getServletContext().getServletContextName());
163         out.append("<br>属性:");
164         Enumeration<?> attributeNames = session.getAttributeNames();
165         while (attributeNames.hasMoreElements()) {
166            String name = (String) attributeNames.nextElement();
167            out.append("<br>  " + name + " : " +session.getAttribute(name) .toString());
168         }
169         out.append("<br>");
170      }
171   }
172   /**
173    * 本サーブレット情報の文字列を返す
174    */
175   @Override
176   public String getServletInfo() {
177      return "SslSessionTest, Version 1.0 by Cresc";
178   }
179 }
180
```

このサーブレットは tutorial アプリケーションの tutorial.session.SimpleSessionTest サーブレットと類似しているが、SSL セッションに関する情報の出力が付加されている。

- 043 行目 : @ServletSecurity(@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL))と、アノテーションでこのサーブレットの接続は SSL であることを宣言している。
- 044 行目 : @WebServlet(name="SslSessionTest", urlPatterns={"/sslsession/SslSessionTest"})というアノテーションで、このサーブレットの要求 URL パタンが/sslsession/SslSessionTest であることを宣言してい

る。

- @ServletSecurity アノテーションは、配備記述子で以下のように記述するのと等価である:

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>SSL Session Test Area</web-resource-name>
      <url-pattern>/sslsession/*</url-pattern>
   </web-resource-collection>
   <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

- 056 行目 : visitTimes はこのセッションでこのサーブレットへの訪問回数で、初期値は 1 である。この変数は Integer オブジェクトとしてセッションにバインドされ、訪問毎にインクリメントされる。

- 069-072 行目 : クライアントからセッションのリセット・ボタンがクリックされたときは、従来のセッションのバインド同じように HttpSession オブジェクトの invalidate メソッドを呼ぶ。

- 076-081 行目 : クライアントから SSL セッションのリセット・ボタンがクリックされたときは、更に SSL セッション・マネージャの invalidate メソッドを呼ぶとともに TCP 接続も切断する。これは、添付資料の「自分のアプリケーション内でセッション追跡に SSL を使用する」の項を参考にされたい。この部分は Tomcat 7 独自であるので、他のサーブレット・コンテナにはそのまま移植出来ないので注意する必要がある。

- 134 行目 : out.append("
getRequestedSessionId: " + req.getRequestedSessionId());で要求オブジェクトから取り出したセッション ID を出力している。

- 135 行目 : 一方この行では、out.append("
sslID: " + (String)req.getAttribute("javax.servlet.request.ssl_session"));で、要求の属性として貼り付けられている SSL のセッション ID を出力している。134 行目と 135 行目は実質等価であることがこのサーブレットの実行結果で理解されよう。

下図はこのサーブレットを Safari ブラウザからアクセスした例である:

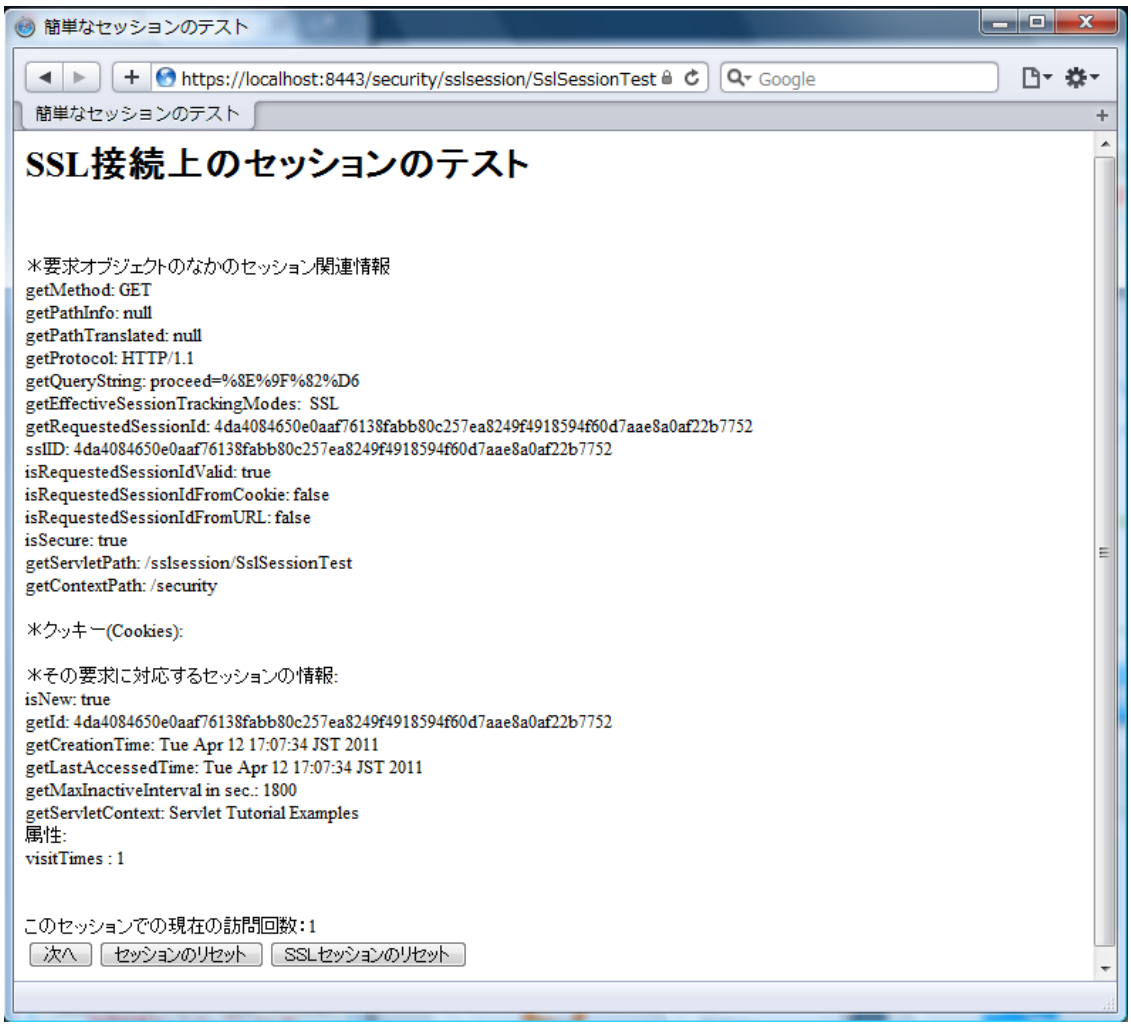


図 15-25：SslSessionTest サーブレットへのアクセス例

このサーブレットの出力を行番号つきで示すと次のようになる：

```
001 ＊要求オブジェクトのなかのセッション関連情報
002 getMethod: GET
003 getPathInfo: null
004 getPathTranslated: null
005 getProtocol: HTTP/1.1
006 getQueryString: null
007 getEffectiveSessionTrackingModes:  SSL
008 getRequestedSessionId: 4da44da289f7c73420a66ffdd1f7509cfe2e97409baa22eb8d7ddc6e717efce1
009 sslID: 4da44da289f7c73420a66ffdd1f7509cfe2e97409baa22eb8d7ddc6e717efce1
010 isRequestedSessionIdValid: true
011 isRequestedSessionIdFromCookie: false
012 isRequestedSessionIdFromURL: false
013 isSecure: true
014 getServletPath: /sslsession/SslSessionTest
015 getContextPath: /security
016
017 ＊クッキー(Cookies):
018
019 ＊その要求に対応するセッションの情報:
020 isNew: true
021 getId: 4da44da289f7c73420a66ffdd1f7509cfe2e97409baa22eb8d7ddc6e717efce1
022 getCreationTime: Tue Apr 12 22:03:30 JST 2011
023 getLastAccessedTime: Tue Apr 12 22:03:30 JST 2011
024 getMaxInactiveInterval in sec.: 1800
025 getServletContext: Servlet Tutorial Examples
026 属性:
027 visitTimes : 1
028
029
```

```
030 このセッションでの現在の訪問回数：1
```

以下のことを各自確認されたい：

- 007-015 行で判るように、セッション追跡モードが SSL になっており、getRequestedSessionId と sslID とは同じになっている。また isSecure も true になっている。
- 「セッションのリセット」ボタンをクリックしても visitTimes と isNew がリセットされるだけで、セッション ID は変更されない。
- 「SSL セッションのリセット」ボタンをクリックすると、セッション ID と sslID が更新される。

15.7.3 SSL 接続上の認証

SSL 接続上での認証方式は、多分 BASIC 及び FORM が一般的であろう。これれの認証方式はユーザ名とパスワードが安全でない形でサーバに送信されるからである。一方 DIGEST 認証は安全な方式であり、更にこれを SSL 接続に乗せる必要はなかろう。

以下は SSL 上での認証のテストの為のサーブレットである。配備記述子である web.xml では、security/ssl/*及び security/ssllogin/*という要求パスには SSL を適用することは既に指定されている。またこのアプリケーションのデフォルトの認証方式は FORM であると指定されている。アプリケーションあたり唯一つしか認証方式は指定できないことに注意されたい。配備記述子の auth-method 要素の値を FORM、BASIC あるいは DIGEST に変更してこのサーブレットを呼び出す。FORM 以外の認証方式を選択したときは form-login-config 要素は無視される。

```
<!-- Default login configuration uses form-based authentication -->
<login-config>
   <auth-method>FORM</auth-method>
   <realm-name>Protected Area</realm-name>
   <form-login-config>
      <form-login-page>/javascripts/security/protected/login.jsp</form-login-page>
      <form-error-page>/javascripts/security/protected/error.jsp</form-error-page>
   </form-login-config>
</login-config>
```

15.7.3.1 SSL 接続上の BASIC 認証

最初に BASIC 認証でこのサーブレットを https://localhost:8443/security/ssllogin/SslLoginTest という URL でアクセスすると、先ず次のようなログイン・ページが表示される。

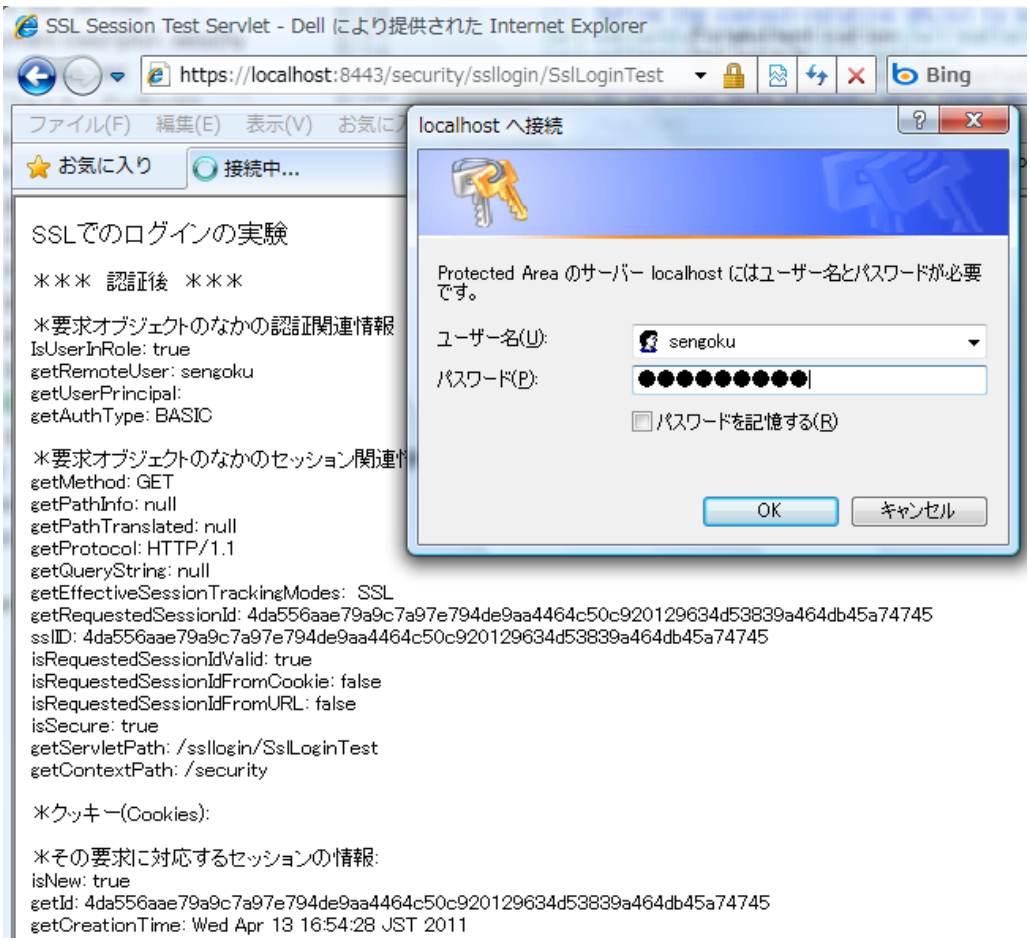


図 15-26：SslLoginTest サーブレットのアクセス例

正しいユーザ名とパスワードを入力して Log In ボタンをクリックすると、このサーブレットは上図のような応答をブラウザに返す。このサーブレットは認証後、ログオフ後、及び SSL セッション解放後の 3 つの状態での認証及びセッションにかかわる情報を出力しているので、実際のアプリケーションのコーディングに有用なものである。

15.7.3.2 SSL 接続上の FORM 認証

FORM 認証でこのサーブレットをアクセスすると、以下のような応答が返される：

```
SSL でのログインの実験

＊＊＊　認証後　＊＊＊

＊要求オブジェクトのなかの認証関連情報
IsUserInRole: true
getRemoteUser: sengoku
getUserPrincipal:
getAuthType: FORM

＊要求オブジェクトのなかのセッション関連情報
getMethod: GET
getPathInfo: null
getPathTranslated: null
getProtocol: HTTP/1.1
getQueryString: null
getEffectiveSessionTrackingModes:  SSL
getRequestedSessionId: 01C3F83FC492DF9C1D82C252307D8CE3
sslID: 4da55cb85b746394cecc1367917110ea3fdb316e579695e5c71eb7ff0884cc86
isRequestedSessionIdValid: true
isRequestedSessionIdFromCookie: false
isRequestedSessionIdFromURL: false
isSecure: true
```

```
getServletPath: /ssllogin/SslLoginTest
getContextPath: /security
（以下省略）
```

注意しなければならないことは、**FORM 認証ではセッション ID と SSL セッション ID とが異なっている**ことである。これは、Javadoc の getRequestedSessionId メソッドの説明に、「そのクライアントによって指定されているセッション ID を返す。これはその要求に対して現在有効なセッションの ID と異なっている可能性もある。そのクライアントがセッション ID を指定してきていないときは、このメソッドは null を返す」と記されている。しかしながら req.getSession(false).getId()の値も SSL セッション ID と異なっており、注意が必要である。

15.7.3.3 SslLoginTest サーブレット

このサーブレットのコードは次のようである。基本的には「プログラム的なセキュリティ」の節で示した ProgrammaticLogin がベースになっている：

SslLoginTest.java

```
001 package tutorial_security;
002
003 import java.io.IOException;
004 import java.io.PrintWriter;
005 import java.util.Date;
006 import java.util.Enumeration;
007 import java.util.Set;
008
009 import javax.servlet.ServletContext;
010 import javax.servlet.ServletException;
011 import javax.servlet.SessionTrackingMode;
012 import javax.servlet.annotation.HttpConstraint;
013 import javax.servlet.annotation.ServletSecurity;
014 import javax.servlet.annotation.WebServlet;
015 import javax.servlet.annotation.ServletSecurity.TransportGuarantee;
016 import javax.servlet.http.Cookie;
017 import javax.servlet.http.HttpServlet;
018 import javax.servlet.http.HttpServletRequest;
019 import javax.servlet.http.HttpServletResponse;
020 import javax.servlet.http.HttpSession;
021
022 /**
023  * Tomcat での SSL セッション上でのログインのテスト用サーブレット
024  *
025  * 配備記述子の<login-config>要素の<auth-method>要素を BASIC あるいは FORM として、
026  * このサーブレットをブラウザから以下のように呼び出す：
027  * https://localhost:8443/security/ssllogin/SslLoginTest
028  * @author Cresc Corp.
029  *
030  */
031 @ServletSecurity(@HttpConstraint
032      (rolesAllowed = "manager", transportGuarantee = TransportGuarantee.CONFIDENTIAL))
033 @WebServlet(name="SslLoginTest", urlPatterns={"/ssllogin/SslLoginTest"})
034 public class SslLoginTest extends HttpServlet {
035  private static final long serialVersionUID = 1L;
036
037  /**
038   * 到来 HTTP 要求処理
039   */
040  public void performTask(HttpServletRequest req, HttpServletResponse res)
041  throws ServletException, IOException
042  {
043     HttpSession session  = req.getSession(true); // FORM の場合は既にセッションが作られている
044     res.setContentType("text/html; charset=Windows-31J");
045     PrintWriter out = res.getWriter();
046     out.println("<html>");
047     out.println("<head><title>SSL Session Test Servlet</title></head>");
048     out.println("<body>");
049     out.println("<big>SSL でのログインの実験</big><br><br>");
050
051     out.println("＊＊＊ 認証後 ＊＊＊" + "<br>");
052     ServletContext ctx = getServletContext();
```

```
053     dumpSession(req, out, ctx);
054
055     // ログアウト
056     req.logout();
057     out.println("<br><br>＊＊＊ ログアウト後 ＊＊＊<br>");
058     dumpSession(req, out, ctx);
059
060     // SSLセッションも終了させる場合 (TomcatのBIOとNIOのコネクタでのみ有効)
061     session.invalidate();   // まずSessionを無効化
062     org.apache.tomcat.util.net.SSLSessionManager mgr =
063        (org.apache.tomcat.util.net.SSLSessionManager)
064        req.getAttribute("javax.servlet.request.ssl_session_mgr");
065     mgr.invalidateSession();   // SSLセッション・マネージャによるSSLセッションの無効化
066     res.setHeader("Connection", "close"); // TCP接続も切断させる
067     out.println("<br><br>＊＊＊ SSLセッション終了後 ＊＊＊<br>");
068     dumpSession(req, out, ctx);
069
070     out.println("</body></html>");
071     out.close();
072  }
073
074  /**
075   * 到来HTTP GET / POST要求の処理
076   */
077  @Override
078  public void doGet(HttpServletRequest req, HttpServletResponse res)
079  throws javax.servlet.ServletException, java.io.IOException {
080     performTask(req, res);
081  }
082  @Override
083  public void doPost(HttpServletRequest req, HttpServletResponse res)
084  throws javax.servlet.ServletException, java.io.IOException {
085     performTask(req, res);
086  }
087
088  /**
089   * 情報のダンプ出力
090   */
091  public void dumpSession(HttpServletRequest req, PrintWriter out, ServletContext context)
throws IOException {
092
093     out.append("<br>＊要求オブジェクトのなかの認証関連情報");
094     out.println("<br>IsUserInRole: " + req.isUserInRole("manager"));
095     out.println("<br>getRemoteUser: "+ req.getRemoteUser());
096     out.println("<br>getUserPrincipal: "+ req.getUserPrincipal());
097     out.println("<br>getAuthType: " + req.getAuthType()+"<br>");
098
099     out.append("<br>＊要求オブジェクトのなかのセッション関連情報");
100     out.append("<br>getMethod: " + req.getMethod());
101     out.append("<br>getPathInfo: " + req.getPathInfo());
102     out.append("<br>getPathTranslated: " + req.getPathTranslated());
103     out.append("<br>getProtocol: " + req.getProtocol());
104     out.append("<br>getQueryString: " + req.getQueryString());
105     out.append("<br>getEffectiveSessionTrackingModes:");
106     Set<SessionTrackingMode> stms = context.getEffectiveSessionTrackingModes();
107     for (SessionTrackingMode c : stms)
108        out.append("  " + c);
109     out.append("<br>getRequestedSessionId: " + req.getRequestedSessionId());
110     out.append("<br>sslID: " + (String)req.getAttribute("javax.servlet.request.ssl_session"));
111     out.append("<br>isRequestedSessionIdValid: " + (new
Boolean(req.isRequestedSessionIdValid())).toString());
112     out.append("<br>isRequestedSessionIdFromCookie: " + (new
Boolean(req.isRequestedSessionIdFromCookie())).toString());
113     out.append("<br>isRequestedSessionIdFromURL: " + (new
Boolean(req.isRequestedSessionIdFromURL())).toString());
114     out.append("<br>isSecure: " + (new Boolean(req.isSecure())).toString());
115     out.append("<br>getServletPath: " + req.getServletPath());
116     out.append("<br>getContextPath: " + req.getContextPath());
117     out.append("<br>");
118
119     out.append("<br>＊クッキー(Cookies):");
120     Cookie[] cookies = req.getCookies();
121     if ((cookies != null) && (cookies.length > 0)) {
```

```
122         for (int i = 0; i < cookies.length; i++) {
123            String name = cookies[i].getName();
124            String value = cookies[i].getValue();
125            out.append("<br>  " + name + " : " + value);
126         }
127      }
128
129      HttpSession session = req.getSession(false);
130      if (session != null) {
131         out.append("<br><br>＊その要求に対応するセッションの情報:");
132         out.append("<br>isNew: " + session.isNew());
133         out.append("<br>getId: " + session.getId());
134         out.append("<br>getCreationTime: " + new Date(session.getCreationTime()));
135         out.append("<br>getLastAccessedTime: " + new Date(session.getLastAccessedTime()));
136         out.append("<br>getMaxInactiveInterval in sec.: " + session.getMaxInactiveInterval());
137         out.append("<br>getServletContext: " +
session.getServletContext().getServletContextName());
138         out.append("<br>属性:");
139         Enumeration<?> attributeNames = session.getAttributeNames();
140         while (attributeNames.hasMoreElements()) {
141            String name = (String) attributeNames.nextElement();
142            out.append("<br>  " + name + " : " +session.getAttribute(name) .toString());
143         }
144         out.append("<br>");
145      }
146   }
147
148 }
149
```

第16章　　参考資料

16.1節　　サーブレット3.0の主要クラスのメソッド一覧

*本節では、サーブレット3.0のAPIのJavadocのなかから、主要なクラスとインターフェイスのメソッドを抄訳している。**翻訳の精度は保証しない**ので、正確な内容は原文を見て頂きたい。*

16.1.1　　サーブレットAPIの主要クラスとインターフェイス

はじめにサーブレットAPIの主要構成要素を説明する。これらの要素は下図のように、javax.servlet及びjavax.servlet.http パッケージに含まれている。

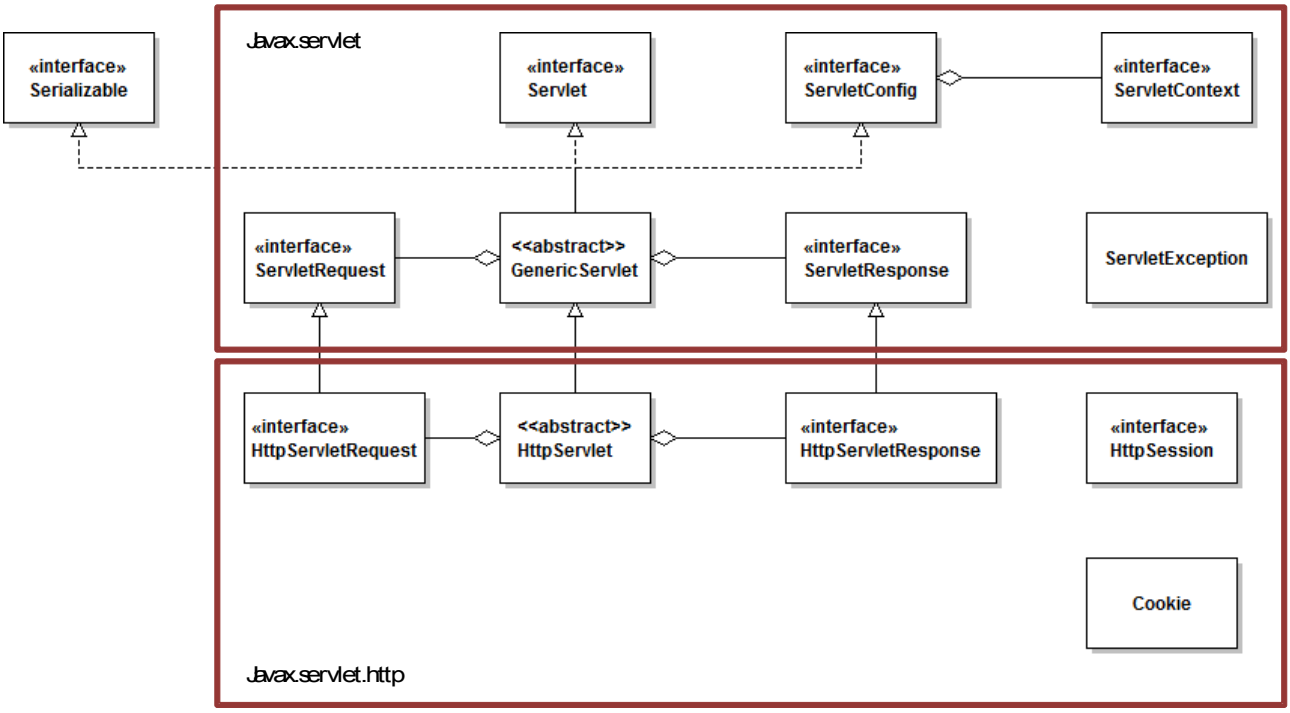


図 AA-1：サーブレットAPIの主要構成要素

javax.servlet パッケージでは、サーブレットにおける要求と応答を表現するJavaのオブジェクトたち、及びその設定パラメタたちと実行環境を反映させるオブジェクトたちを定義している。javax.servlet.http パッケージでは、ジェネリックなservlet要素たちに対するHTTP固有のサブクラスたちを定義しており、これにはウェブ・サーバとあるクライアント間の複数の要求と応答を追跡する為のセッション管理オブジェクトたちが含まれる。2つのパッケージの各要素は次のような機能を持っている。

javax.servlet パッケージ

クラス/インターフェイス	記述

interface Servlet	各サーブレットが実装しなければならないメインのインターフェイス。ここではサーブレット・コンテナがそのサーブレットをコントロールする為に呼び出す主要メソッドが定義されている。
abstract class GenericServlet	この抽象クラスはサーブレット実装の開始点として使われる。特にこのクラスはservice()メソッドを除いて Servlet インターフェイスの総てのメソッドたちを実装している。この抽象クラスはまた ServletConfig インターフェイスを実装しており、これによりサーブレット・コンテナがサーブレットに情報を渡すことが出来る。
interface ServletRequest	このインターフェイスはクライアントからの要求から情報を取り出す為のメソッドたちを用意している。
interface ServletResponse	このインターフェイスはクライアントからの要求に対するしかるべき応答を生成し送信する為のメソッドたちを用意している。
interface ServletConfig	このインターフェイスによりサーブレット・コンテナがサーブレットに情報を渡すことが出来る。
interface ServletContext	このインターフェイスにより、サーブレットはそのコンテナと通信できる。
class ServletException	サーブレットのランタイムのエラーをシグナルする為の汎用例外。

javax.servlet.http パッケージ

クラス/インターフェイス	記述
abstract class HttpServlet	この抽象クラスは GenericServlet クラスを継承しており、HTTP サーブレット（即ち要求と応答に HTTP を使うサーブレット）の実装に使われる。特にこのクラスは要求（GET、POST、HEAD、等）に使われる HTTP メソッドに対応した doHttpRequestMethodName()の為のスタブを用意している。具体的なサーブレットは異なった HTTP 要求メソッドを処理する為にしかるべきメソッドをオーバライドする。
interface HttpServletRequest	このインターフェイスは HTTP 要求を処理する為に ServletRequest を継承している。
interface HttpServletResponse	このインターフェイスは HTTP 要求に対するしかるべき HTTP 応答を生成し送信する為に ServletResponse を継承している。
interface HttpSession	このインターフェイスは複数ページにわたってユーザを特定する、あるウェブ・サイトを訪問する、またそのユーザに対する情報をストアする為の手段を提供している。
class Cookie	このクラスは要求と応答に使われるクッキーをサポートする。

16.1.2 要求関係

16.1.2.1 ServletRequest

public interface javax.servlet.**ServletRequest** クライアント要求情報をサーブレットに渡すためのオブジェクト。サーブレット・コンテナは ServletRequest オブジェクトを生成し、それをそのサーブレットの service メソッドに引数として渡す。ServletRequest オブジェクトはパラメタ名と値、属性、及び入力ストリームを含むデータを提供している。ServletRequest を継承したインターフェイスは、更にプロトコル固有のデータを提供出来る。（例えば、HTTP データは HttpServletRequest.が提供する。）	
java.lang.Object **getAttribute**(java.lang.String name)	指定された名前を持った属性の値を Object 型で返す、あるいはその名前の属性がなければ null を返す。属性は以下の二つの方法でセットできる。サーブレットコンテナが認証などでその要求の顧客情報を取得可能にするためにセットする。例えば、HTTPS を使った要求では、そのクライアントの認証に関する情報を取り出すのに、javax.servlet.request.X509Certificate という属性が使える。 もうひとつはプログラム的に setAttribute(java.lang.String, java.lang.Object メソッドを使ってセットする方法である。こうすると RequestDispatcher 呼び出し前に要求に情報を埋め込むことができる。属性の名前はパッケージ名の規約とおなじ規約に従わねばならない。
java.util.Enumeration<java.lang.String> **getAttributeNames**()	列挙型でこの要求で取得可能な属性たちの名前を返す。この要求に取得できる属性が存在しなければ空の列挙型を返す。
java.lang.String **getCharacterEncoding**()	HTTP 要求パケットのボディ部分に使われている文字エンコーディングを返す。この要求に文字エンコーディングが指定されていないときには null を返す。
Void **setCharacterEncoding**(java.lang.String env) throws java.io.UnsupportedEncodingException	この要求のボディ部に使われている文字エンコーディングの名前をオーバライドする。このメソッドは要求パラメタたちを呼び出す、あるいは getReader()を使って入力を読み出す前に呼ばれねばならない。でないとこのメソッドは効果をもたらさない。
int **getContentLength**()	入力ストリームで取得可能である要求メッセージのボディ部分の長さをバイト数で返す、あるいはこの長さが不明の場合は-1 を返す。HTTP サーブレットの場合は、これは CGI 変数の CONTENT_LENGTH の値と同じである。
java.lang.String **getContentType**()	HTTP 要求パケットのボディ部分の MIME タイプを返す。MIME タイプが不明の場合は null を返す。HTTP サーブレットの場合は、これは CGI 変数の CONTENT_TYPE の値と同じである。
ServletInputStream **getInputStream**() throws java.io.IOException	HTTP 要求パケットのボディ部分のデータを ServletInputStream を用いて読み出すには、このメソッドまたは getReader()のいずれかを使う。この要求のボディ部を含む ServletInputStream オブジェクトを返す。 java.lang.IllegalStateException −該要求に getReader()メソッドが既に呼ばれてしまっている場合。 java.io.IOException −入出力の例外が発生した場合。
java.lang.String **getParameter**(java.lang.String name)	ある要求パラメタの値を String 型で返す。その名前のパラメタが存在しないときには null を返す。要求パラメタはその要求とともに送られる追加情報である。HTTP サーブレットの場合は、パラメタたちはクエリ文字列かフォームデータの POST データにより送られる。このメソッドを使うときにはその名前のパラメタがただひとつの値しか持っていないことをあらかじめ確認しておく必要がある。もし複数の値を持っている可能性があるときには getParameterValues(java.lang.String)を使うこと。複数の値が存在する場合は、このメソッドは最初の値を返す。HTTP POST メソッドなどで起きるように要求が要求ボディで送られている場合は、getInputStream() または getReader()メソッドを使って要求ボディを直接読み出すことは、このメソッドの実行に干渉を与える可能性がある。
java.util.Enumeration<java.lang.String> **getParameterNames**()	この要求に含まれているパラメタの名前の全てを String オブジェクトの列挙型で返す。この要求にパラメタが含まれていなければ空の列挙型のオブジェクトを返す。
java.lang.String[] **getParameterValues**(java.lang.String name)	この要求に存在する指定された名前の要求パラメタの全ての値を String オブジェクトの配列で返す。そのパラメタが存在しなければ null を返す。そのパラメタが単一の値しか待たない場合は、配列の長さは 1 となる。
java.util.Map<java.lang.String,java.lang.String[]> **getParameterMap**()	この要求のパラメタたちの java.util.Map を返す。要求パラメタはその要求とともに送られる追加情報である。HTTP サーブレットの場合は、パラメタたちはクエリ文字列かフォームデータの POST データにより送られる。返されるものはパラメタ名たちをキーとし、パラメタ値たちをマップ値として含むイミュータブルな java.util.Map である。このパラメタ・マップのキーは String 型である。値は String 型の配列である。.
java.lang.String **getProtocol**()	この要求が使っているプロトコルを HTTP/1.1 のように protocol/majorVersion.minorVersion の形式で返す。HTTP サーブレットの場合は、返される値は CGI 変数の SERVER_PROTOCOL と同じである。

java.lang.String **getScheme**()	この要求を作るのに使われた http, https, あるいは ftp などのスキームを返す。RFC 1738 で規定されているように、スキームによって URL の組み立てルールが違ってくる。
java.lang.String **getServerName**()	この要求を受信したサーバのホスト名を返す。これは Host ヘッダ行の Host ヘッダ値の":"の前の部分であるか、あるいは解決されたサーバ名、あるいはそのサーバの IP アドレスである。
int **getServerPort**()	この要求を受信した TCP ポート番号を返す。これは Host ヘッダ行の Host ヘッダ値の":"の後の部分であるか、あるいはそのクライアント接続が受け付けたサーバ・ポート番号である。
java.io.BufferedReader **getReader**() throws java.io.IOException	HTTP 要求パケットのボディ部を BufferedReader を使って文字データとして読み出す。このリーダはこのボディ部に使われている文字エンコーディングに基づいて変換を行う。getReader()とこのメソッドともにボディ読み出しに使えるが同時使用はできない。 java.io.UnsupportedEncodingException – この文字セットエンコーディグはサポートしていないのでデコードできない。java.lang.IllegalStateException –該要求に既に getInputStream() が呼ばれてしまっている。 java.io.IOException – 入出力の例外が発生した。
java.lang.String **getRemoteAddr**()	この要求を送信したクライアントあるいはプロキシの IP アドレスを String で返す。HTTP サーブレットの場合は、これは CGI 変数の REMOTE_ADDR の値と同じである。
java.lang.String **getRemoteHost**()	この要求を送信したクライアントの名前を正式名(完全修飾名 : fully qualified name)の String で返す。エンジンが hostname を解決しないことを選択した場合（性能改善の為）、このメソッドはその IP アドレスのドット表記の文字列を返す。HTTP サーブレットの場合は、これは CGI 変数の REMOTE_HOST の値と同じである。
void **setAttribute**(java.lang.String name, java.lang.Object o)	要求に属性をセットする。属性は要求ごとにリセットされる。このメソッドは RequestDispatcher とともに通常最も使用される。属性名はパッケージ名と同じ規約に従わねばならない。java.*、javax.*、及び com.sun.*で始まる名前は Sun Microsystems によって予約されている。渡されたオブジェクトが null の場合は、それは removeAttribute(java.lang.String)を呼んだと同じ結果をもたらす。警告として、RequestDispatcher によって別のウェブ・アプリケーションの中にあるサーブレットからその要求がディスパッチされているときは、このメソッドでセットされたオブジェクトが呼び出したサーブレットの中で正しく取り出せないかもしれない。
void **removeAttribute**(java.lang.String name)	この要求から属性を除去する。通常は必要としない。属性はその要求が処理されている間だけ存続するので、このメソッドは一般的には必要とされない。属性名はパッケージ名と同じ規約に従わねばならない。 java.*、javax.*、及び com.sun.*で始まる名前は Sun Microsystems によって予約されている。
java.util.Locale **getLocale**()	クライアントが Accept-Language ヘッダ行で指定してきたクライアントが対応可能な言語地域指定を Locale 型で返す。クライアントが特に指定していなければ、このサーバのデフォルトのロケールを返す。
java.util.Enumeration<java.util.Locale> **getLocales**()	上記と同じだが、列挙型でロケールを降順で返す。クライアントが特に Accept-Language で指定していなければ、このサーバのデフォルトのロケールひとつを含んだ列挙オブジェクトを返す。
boolean **isSecure**()	該要求が HTTPS などのセキュアなチャンネルを介してなされているかをブール値で返す。
RequestDispatcher **getRequestDispatcher**(java.lang.String path)	与えられたパスに位置するリソースのラッパとして機能する RequestDispatcher オブジェクトを返す。RequestDispatcher は要求をそのリソースに転送したり、そのリソースを応答に含めてしまうのに使うことができる。このリソースはダイナミックでもスタティックでも良い。指定するパス名は相対で指定しても良いが、その場合は現在のサーブレットコンテキストの外部には拡張は出来ない。パスが”/”で始まる場合はこれは現行のコンテキストルートに対して相対であると解釈される。このサーブレットコンテナが RequestDispatcher を返せないときは null を返す。このメソッドと ServletContext.getRequestDispatcher(java.lang.String)との相違は、このメソッドが相対パスをとり得ることである。
java.lang.String **getRealPath**(java.lang.String path)	廃止対象。Java サーブレット API の 2.1 版の時点では、このメソッドの代わりに ServletContext.getRealPath(java.lang.String)を使うこと。
int **getRemotePort**()	その要求を送信したクライアントまたは最後のプロキシの IP ソース・ポート番号を返す。
java.lang.String **getLocalName**()	その要求を受信した IP インターフェイスのホスト名を返す。
java.lang.String **getLocalAddr**()	その要求を受信した IP インターフェイスの IP アドレスを返す。
int **getLocalPort**()	その要求を受信したインターフェイスの IP ポート番号を返す。
ServletContext **getServletContext**()	この ServletRequest が最後にディスパッチされたサーブレット・コンテキストを取得する。 Servlet 3.0 から導入。
AsyncContext **startAsync()** throws java.lang.IllegalStateException	この要求を非同期モードとし、その AsyncContext をオリジナルの（ラップされていない）ServletRequest と ServletResponse のオブジェクトで初期化する。 このメソッドを呼ぶことで、返された AsyncContext 上で AsyncContext.complete()が呼ばれる、あるいはその

	非同期操作がタイムアウトを起こすまで、それに関わる応答のコミットが遅らせられる。 返された AsyncContext 上で AsyncContext.hasOriginalRequestAndResponse()を呼ぶと true が返される。この要求が非同期モードに置かれた後での下り方向で呼び出されたフィルタたちは、それらが登り方向で追加した要求及び/あるいは応答のラッパが、その非同期操作中は留まる必要がなく、したがってそれに関わるリソースがリリースしてよいことを示すものとしてこれを使うことができる。 このメソッドは、その onStartAsync で各 AsyncListner を呼び出した後で startAsync メソッドたちの以前の呼び出すことで返された AsyncContext に登録された AsyncListener インスタンスたち(もし存在していれば)のリストをクリアする。 その後のこのメソッドあるいはそのオーバロードされたメソッドを呼び出すと、同じ AsyncContext が返され、しかるべく初期化される。 Servlet 3.0 から導入。
AsyncContext **startAsync**(ServletRequest servletRequest, ServletResponse servletResponse) throws java.lang.IllegalStateException	この要求を非同期モードとし、与えられた要求と応答オブジェクトでその AsyncContext を初期化する。 引数である ServletRequest 及び ServlerResponse は、このメソッドが呼ばれている有効範囲内において、Servlet の service メソッドに渡された、あるいは Filter のdoFilterメソッドに渡されたと同じインスタンスたち、あるいはそれらをラップした ServletRequestWrapper 及び ServletResponseWrapper のインスタンスでなければならない。 このメソッドを呼ぶことで、返された AsyncContext 上で AsyncContext.complete()が呼ばれる、あるいはその非同期操作がタイムアウトを起こすまで、それに関わる応答のコミットが遅らせられる。 返された AsyncContext 上で AsyncContext.hasOriginalRequestAndResponse()を呼ぶと、渡された ServletRequest と ServletResponse 引数がオリジナルのものであってアプリケーションが用意したラッパを持っていないときに限り、false が返される。この要求が非同期モードに置かれた後での下り方向で呼び出されたフィルタたちは、それらが登り方向で追加した要求及び/あるいは応答のラッパたちの幾つかは、その非同期操作中は留まるってはいる必要があり、したがってそれに関わるリソースがリリースしてはならないことを示すものとしてこれを使うことができる。AsyncContext を初期化するのに使われ、AsyncContext.getRequest()呼び出しで返されることになる与えられた servletRequest が、ServletRequestWrapper を含まない限り、あるフィルタの登り方向の呼び出し中に適用された ServletRequestWrapper は、そのフィルタの下り方向呼び出しでリリースされて良い。同じことは ServletResponseWrapper でもいえる。 このメソッドは、その onStartAsync で各 AsyncListner を呼び出した後で startAsync メソッドたちの以前の呼び出すことで返された AsyncContext に登録された AsyncListener インスタンスたち(もし存在していれば)のリストをクリアする。 その後のこのメソッドあるいはそのオーバロードされたメソッドを呼び出すと、同じ AsyncContext が返され、しかるべく初期化される。もしこのメソッドの呼び出しの後でゼロ引数のメソッドを呼ぶと、指定された(ラップされていることもあり)要求と応答のオブジェクトたちは、返された AsyncContext 上に閉じ込められる。 Servlet 3.0 から導入。
boolean **isAsyncStarted**()	この要求が非同期モードになっているかどうかをチェックする。 ServletRequest はその startAsync()あるいは startAsync(ServletRequest,ServletResponse)を呼ぶことで非同期モードに置かれる。 このメソッドは、この要求が非同期モードになっているが、AsyncContext.dispatch()メソッドたちのどれかを使ってディスパッチされている、あるいは AsyncContext.complete()メソッド呼び出しで非同期モードから解放されているときは、false を返す。 Servlet 3.0 から導入。
boolean **isAsyncSupported**()	この要求が非同期操作に対応しているかどうかをチェックする。もしこの要求が非同期処理に対応可能とアノテートされていないか配備記述子内でフラグが立てられていないフィルタあるいはサーブレットの有効範囲内にある場合は、非同期動作は不能になっている。 Servlet 3.0 から導入。
AsyncContext **getAsyncContext**()	この要求上で最も最近に startAsync()あるいは startAsync(ServletRequest,ServletResponse)呼び出しで生成または初期化された AsyncContext を取得する。 Servlet 3.0 から導入。
DispatcherType **getDispatcherType**()	この要求のディスパッチャのタイプを取得する。 ある要求のディスパッチャのタイプは、その要求に提供されるべきフィルタをコンテナが選択するのに使われる:一致するディスパッチャ・タイプと URL パタンを持ったフィルタたちのみが適用される。 複数のディスパッチャ・タイプに対応するよう設定されているフィルタが、ある要求に対するディスパッチャを問い合わせできるようにすることで、そのフィルタはそのディスパッチャ・タイプに応じてその要求を処理できるようになる。 ある要求の初期ディスパッチャ・タイプは DispatcherType.REQUEST として定義されている。 RequestDispatcher.forward(ServletRequest, ServletResponse)あるいは 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を介して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 として与えられている。最後に、その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・タイプは DispatcherType.ERROR として与え

	られている。 Servlet 3.0 から導入。

16.1.2.2 HttpServletRequest

public interface **HttpServletRequest** extends ServletRequest HTTP サーブレットのために要求情報を提供するために ServletRequest インターフェイスを継承。 サーブレット・コンテナは HttpServletRequest オブジェクトを生成し、そのサーブレットの service メソッドたち(doGet, doPost, etc)に引数としてそれを渡す。	
フィールド	static java.lang.String BASIC_AUTH ：Basic 認証の為の String 識別子 static java.lang.String CLIENT_CERT_AUTH ：クライアント認定のための String 識別子 static java.lang.String DIGEST_AUTH ：Digest 認証の為の String 識別子 static java.lang.String FORM_AUTH ：Form 認証の為の String 識別子
java.lang.String **getAuthType**()	そのサーブレットを保護する為に使われている認証スキームの名前を返す。総てのサーブレット・コンテナはベーシック、フォーム、及びクライアント証明証による認証に対応しており、さらには追加的にダイジェスト認証をに対応してよい。そのサーバが認証に対応していない場合は、null が返される。 このメソッドは CGI 変数の AUTH_TYPE と同じである。
Cookie[] **getCookies**()	その要求でクライアントが送信した Cookie オブジェクトの総てを含む配列を返す。このメソッドはクッキーが遅れれていないときは null を返す。
long **getDateHeader**(java.lang.String name)	指定された要求ヘッダの値を Date オブジェクトで表現された long 値として返す。このメソッドは If-Modified-Since のような日付を含むヘッダで使用する。 この Date は GMT の 1970 年元旦からの mS の数値として返される。ヘッダ名は大文字と小文字を区別しない。 その要求が指定された名前のヘッダを持っていないときは、ものメソッドは-1 を返す。そのヘッダが Date 型に変換できないときは、このメソッドは IllegalArgumentException をスローする。
java.lang.String **getHeader**(java.lang.String name)	この要求が含んでいる指定されたヘッダ名の値を String として返す。もしその要求が指定された名前のヘッダを含んでいないときは、このメソッドは null を返す。同じ名前で複数のヘッダが存在するときは、このメソッドはその要求の中の最初のものを返す。ヘッダ名は大文字と小文字を区別されない。このメソッドはどの要求ヘッダにも使用可能である。
java.util.Enumeration<java.lang.String> **getHeaderNames**()	この要求が含むヘッダ名の総てを列挙型として返す。その要求がヘッダをもていないときはこのメソッドは空の列挙型を返す。 一部のサーブレット・コンテナはこのメソッドでサーブレットがヘッダにアクセスするのを認めていないが、その場合もこのメソッドは null を返す。
int **getIntHeader**(java.lang.String name)	指定された要求ヘッダの値を int として返す。この要求が指定された名前のヘッダを持ていないときは、このメソッドは-1 を返す。もしこのヘッダが整数に変換できなかったときは、このメソッドは NumberFormatException をスローする。 ヘッダ名は大文字と小文字を区別されない。
java.lang.String **getMethod**()	その要求が出した HTTP メソッドの名前、たとえば GET、POST、あるいは PUT などを返す。 これは CGI 変数の REQUEST_METHOD と同じである。
java.lang.String **getPathInfo**()	クライアントがこの要求をしたとき送信した URL に関わる追加的パス情報を返す。この追加的パス情報はサーブレット・パスに続くものだが、クエリ文字列の前に位置し、"/"文字で始まる。 追加的パス情報がないときはこのメソッドは null を返す。 これは CGI 変数の PATH_INFO の値と同じである。
java.lang.String **getPathTranslated**()	サーブレット名のあとでクエリ文字列の前の追加的パス情報を実際のパスに変換して返す。これは CGI 変数の PATH_TRANSLATED の値と同じである。 その URL が追加的パス情報を持っていないときは、このメソッドは null を返すか、あるいはそのサーブレット・コンテナはバーチャルなパスを現実のパスに何らかの理由（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ときなど）で変換できない。ウェブ・コンテナはこの文字列をデコードしない。
java.lang.String **getContextPath**()	その要求 URI のその要求のコンテキストを示す部分を返す。コンテキスト・パスは常にその要求 URI の最初のところかになる。このパスは"/"文字で始まるが"/"文字では終わらない。デフォルト（ルート）コンテキストにあるサーブレットにたいしては、このメソッドは""を返す。コンテナはこの文字列をデコードしない。 あるサーブレット・コンテナでひとつ以上のコンテキスト・パスに一致するものがある可能性がある。そのような場合には、このメソッドはその要求に使われた実際のコンテキスト・パスを返し、それは ServletContext.getContextPath()で返されたパスとは異なるかもしれない。ServletContext.getContextPath()が

	返すコンテキスト・パスはそのアプリケーションにとってもっとも適切あるいは好ましいコンテキスト・パスだと考えるべきである。
java.lang.String **getQueryString**()	要求 URL のなかでパスの後に含まれているクエリ文字列を返す。このメソッドはその URL がクエリ文字列を含んでいないときに null を返す。これは CGI 変数の QUERY_STRING の値と同じである。
java.lang.String **getRemoteUser**()	この要求をしているユーザが認証されていればその名前を返す、あるいは認証されていないときは null を返す。そのユーザ名がその後の各要求で送信されるかどうかは、ブラウザと認証のタイプに依存する。これは CGI 変数の REMOTE_USER の値と同じである。
boolean **isUserInRole**(java.lang.String role)	その認証されたユーザが指定された論理的な「ロール」に含まれているかどうかを示すブール値を返す。 ロールとロールのメンバーシップは配備記述子を使って定義できる。そのユーザが認証されていないときは、このメソッドは false を返す。
java.security.Principal **getUserPrincipal**()	現在認証されているそのユーザの名前を含む java.security.Principal オブジェクトを返す。そのユーザが認証されていないときは、このメソッドは null を返す。
java.lang.String **getRequestedSessionId**()	そのクライアントによって指定されているセッション ID を返す。これはその要求に対して現在有効なセッションの ID と異なっている可能性もある。そのクライアントがセッション ID を指定してきていないときは、このメソッドは null を返す。
java.lang.String **getRequestURI**()	その HTTP 要求の最初の行のプロトコル名からクエリ文字列までに間のこの要求の URL の部分を返す。 ウェブ・コンテナはこの文字列をデコードしない。例えば: HTTP 要求の最初の行　　返される値 POST /some/path.html HTTP/1.1　　/some/path.html GET http://foo.bar/a.html HTTP/1.0　　/a.html HEAD /xyz?a=b HTTP/1.1　　/xyz スキームとホストを持った URL を再組み立てするには、 HttpUtils.getRequestURL(javax.servlet.http.HttpServletRequest)を使われたい。
java.lang.StringBuffer **getRequestURL**()	その要求をするのにそのクライアントが使った URL を再組み立てする。返された URL はプロトコル、サーバ名、ポート番号、及びサーバ・パスを含んでいるが、クエリ文字列パラメタは含んでいない。 この要求が RequestDispatcher.forward(javax.servlet.ServletRequest, javax.servlet.ServletResponse)を使ってフォワードされているときは、この再組み立てされた URL の中のサーバ・パスは、クライアントが指定したサーバ・パスではなくて、RequestDispatcher を取得するのに使われたパスが反映されていなければならない。 このメソッドは文字列ではなくて StringBuffer を返すので、例えばクエリ文字列を変えるなど、この URL の加工が容易になる。 このメソッドはリダイレクト・メッセージを生成したり、エラーを報告したりするのに有用である。
java.lang.String **getServletPath**()	このサーブレットを呼んでいるこの要求の URL の一部を返す。このパスは"/"文字で始まり、サーブレット名あるいはこのサーブレットへのパスのどちらも含まれるが、追加パス情報あるいはクエリ文字列は含まれない。これは CGI 変数の SCRIPT_NAME の値と同じである。 この要求を処理する為にこのサーブレットがサーブレットが"/*"パタンと一致している場合は、このメソッドは空の文字列("")を返す。
HttpSession **getSession**(boolean create)	この要求に結び付けられている現在の HttpSession を返す、あるいは現在セッションがなく引数の create が true のときには新しいセッションを返す。 引数の create が false でこの要求が有効な HttpSession を持っていないときは、このメソッドは null を返す。 確実にこのセッションが維持されるように、応答がコミット（ネットワークに送出）される前にこのメソッドを呼ばねばならない。 そのコンテナがセッションに完全性を維持する為にクッキーを使っていて、その応答がコミットされる際に新しいセッションを作るよう要求されたときは、このメソッドは IllegalStateException をスローする。
HttpSession **getSession**()	この要求に結び付けられている現在のセッションを返す、あるいはその要求がセッションを持っていないときは、新しいセッションを作る。
boolean **isRequestedSessionIdValid**()	要求されたセッション ID がまだ有効かどうかをチェックする。 そのクライアントが何らセッション ID を指定していないときは、このメソッドは false を返す。
boolean **isRequestedSessionIdFromCookie**()	要求されているセッション ID がクッキーとして送られてきているかどうかをチェックする。
boolean **isRequestedSessionIdFromURL**()	要求されているセッション ID が要求 URL の一部として送られてきているかどうかをチェックする。
boolean **isRequestedSessionIdFromUrl**()	廃止対象メソッド。Java サーブレット API 第 2.1 版時点ではこれの代わりに isRequestedSessionIdFromURL() を使用する。
boolean **authenticate**(HttpServletRespons	この要求をしているユーザを認証する為にこの ServletContext に設定されているコンテナのログイン・メカニ

e response) throws java.io.IOException, ServletException	ズムを使う。このメソッドは HttpServletResponse 引数を修正及びコミットしても良い。 Servlet 3.0 から導入。
void **login**(java.lang.String username, java.lang.String password) throws ServletException	この ServletContext のために設定されているウェブ・コンテナのログイン・メカニズムが使っているパスワード検証レルムの中で与えられた username と password を検証する。 この ServletContext のために設定されているログイン・メカニズムが username と password の認証に対応している場合、及び login 呼び出しの時点でこの要求の発信者の ID がまだ確立されていないとき（言い換えると getUserPrincipal、getRemoteUser、及び getAuthType の総てが null を返すとき）、及び与えられた信用状の検証が成功したときは、このメソッドは ServletException を返すことなく戻る。 このメソッドが例外をスローしないで戻るときは、getUserPrincipal、getRemoteUser、及び getAuthType によって返される値としては確立された非 null の値を持っていなければならない。 引数： username – そのユーザのログイン識別子に対応した String 値 password – 識別されたユーザに対応するパスワード String 例外： ServletException - 設定されているログイン・メカニズムがユーザ名認証に対応していないとき、あるいは非 null の発信者 ID が既に確立されている（login 呼び出し前に）、あるいは与えられた username と password で検証できなかったとき。 Servlet 3.0 から導入。
void **logout**() throws ServletException	その要求に対し getUserPrincipal、getRemoteUser、及び getAuthType が呼ばれたときに戻り値としての null を確立する。 例外： ServletException - ログアウトに失敗したとき Servlet 3.0 から導入
java.util.Collection<Part> **getParts**() throws java.io.IOException, ServletException	この要求が multipart/form-data タイプである場合に、この要求の総ての Part 要素を取得する。この要求が multipart/form-data タイプであるが Part 型の要素を持っていない場合は、返された Collection は空になる。返された Collection に対する何らかの変更はこの HttpServletRequest に影響を与えない。 戻り値： この要求の Part 要素の Collection（空もあり） 例外： java.io.IOException – この要求の Part 要素の検索で I/O エラーが発生したとき。 ServletException – この要求が multipart/form-data のタイプでないとき java.lang.IllegalStateException - この要求ボディが maxRequestSize よりも大きい、あるいはこの要求の Part のどれかが maxFileSize よりも大きいとき。 Servlet 3.0 から導入。
Part **getPart**(java.lang.String name) throws java.io.IOException, ServletException	与えられた名前の Part を取得する。 引数： name – 要求する Part の名前 戻り値： 与えられた名前の Part、あるいはこの要求が multipart/form-data タイプであるが要求された Part を有していないときは null。 例外： java.io.IOException – この要求の Part 要素の検索で I/O エラーが発生したとき。 ServletException – この要求が multipart/form-data のタイプでないとき java.lang.IllegalStateException - この要求ボディが maxRequestSize よりも大きい、あるいはこの要求の Part のどれかが maxFileSize よりも大きいとき。 Servlet 3.0 から導入。

16.1.3 応答関係

16.1.3.1 ServletResponse

public interface **ServletResponse**	
サーブレットが応答をクライアントに送信するのを支援する為のオブジェクトを規定している。サーブレット・コンテナは ServletResponse オブジェクトを生成し、それをそのサーブレットの service メソッドの引数として渡す。 MIME ボディ応答でバイナリ・データを送信するには、getOutputStream()で返される ServletOutputStream を使用する。文字データを送信するには、getWriter()で返される PeintWriter を使用する。例えば multipart 応答のようなバイナリとテキストがミックスした応答ボディを作るには、ServletOutputStream を使用し、文字の区間はマニュアルで書き込む。 MIME ボディ応答の為の文字セットは、setCharacterEncoding(java.lang.String)及び setContentType(java.lang.String)を使って明示的に指定する、あるいは setLocale(java.util.Locale)メソッドを使って暗示的に指定できる。文字セットが指定されていないときは、ISO-8859-1 がデフォルトとして使用される。setCharacterEncoding、setContentType、あるいは setLocale メソッドは、その文字エンコーディングが使われるためには、getWriter が呼ばれる前、そしてその応答をコミットする前に呼ばれねばならない。 MIME に関する更なる情報に関しては RFC 2045 のような IETF の RFC たちを見られたい。SMTP と HTTP のようなプロトコルが MIME のプロファイルを規定しており、これらの標準は未だに進展し続けている。	
メソッド	
java.lang.String **getCharacterEncoding**()	この応答に送られてきたボディ部に使われている文字エンコーディング (MIME charset) の名前を返す。文字セットは、setCharacterEncoding(java.lang.String)及び setContentType(java.lang.String)を使って明示的に指定、あるいは setLocale(java.util.Locale)メソッドを使って暗示的に指定されていることもある。明示的指定のほうが暗示的指定よりも優先される。getWriter が呼ばれたあと、そしてその応答をコミットされた後にこのメソッドを呼んでも文字エンコーディングには影響を与えない。文字セットが指定されていないときは、ISO-8859-1 がデフォルトとして使用される。 文字エンコーディングと MIME に関する更なる情報は RFC 2047 (http://www.ietf.org/rfc/rfc2047.txt)見られたい。 戻り値: 例えば UTF-8 のように、その文字エンコーディングの名前を指定している String
java.lang.String **getContentType**()	この応答に送られてきた MIME ボディのために使われているコンテント・タイプを返す。このコンテント・タイプ属性は、この応答がコミットされる前に setContentType(java.lang.String)メソッドを使って指定されていなければならない。コンテント・タイプが指定されていないときは、このメソッドは null を返す。コンテント・タイプが指定されており、文字エンコーディングが getCharacterEncoding()で示したように明示的あるいは暗示的に指定されている、あるいは getWriter が呼ばれてしまっているときは、返される String にはその文字セット・パラメタが含められる。もし文字エンコーディングが指定されていないときは、その charset パラメタはオミットされる。 戻り値: 例えば、text/html; charset=UTF-8 のようなコンテント・タイプを指定した String、あるいは null 適用開始: Servlet 2.4
ServletOutputStream **getOutputStream**() throws java.io.IOException	この応答にバイナリ・データを書き込むのに適した ServletOutputStream を返す。サーブレット・コンテナはバイナリ・データに対してはエンコードしない。 ServletOutputStream の flush()を呼ぶことでその応答はコミットされる。このメソッドまたは getWriter()のどちらかがボディ部書き込みに呼ばれ、双方共は呼ばれない。 戻り値: バイナリ・データ書き込みのための ServletOutputStream 例外: java.lang.IllegalStateException - この応答で既に getWriter が呼ばれてしまっている java.io.IOException - 入力または出力例外が生じた
java.io.PrintWriter **getWriter**() throws java.io.IOException	クライアントに文字テキストを送信できる PrintWriter オブジェクトを返す。この PrintWriter は getCharacterEncoding()で返される文字エンコーディングを使う。この応答の文字エンコーディングが未だ getCharacterEncoding で示したように指定されていないとき(言い換えれば getCharacterEncoding がデフォルトの ISO-8859-1 を単に返す場合)は、getWriter はそれを ISO-8859-1 とする。 PrintWriter の flush()を呼ぶことでその応答はコミットされる。このメソッドまたは getOutputStream()のどちらかがボディ部書き込みに呼ばれ、双方共は呼ばれない。 戻り値: クライアントに文字データを送ることが出来る PrintWriter オブジェクト 例外: UnsupportedEncodingException - getCharacterEncoding で返された文字エンコーディングが使えないとき java.lang.IllegalStateException - getOutputStream メソッドがこの応答オブジェクトのために既に呼ばれてし

	まっている java.io.IOException - 入出力の例外が発生した
void **setCharacterEncoding**(java.lang.String charset)	例えば UTF-8 のような、そのクライアントに送信される応答の文字エンコーディング (MIME charset) をセットする。setContentType(java.lang.String)あるいは setLocale(java.util.Locale)で既に文字エンコーディングがセットされているときには、このメソッドはそれを優越する。setContentType(java.lang.String)を text/html の String で呼び、このメソッドを UTF-8 の String で呼ぶことは、text/html; charset=UTF-8 の String で setContentType を呼ぶことと等価である。 このメソッドは文字エンコーディングを変えるために繰り返し呼ぶことができる。このメソッドは getWriter が呼ばれたあと、またはこの応答がコミットされた後では効果をもたらさない。 引数: charset – IANA 文字セット(http://www.iana.org/assignments/character-sets)で規定されている文字セットのみを指定する String 導入: Servlet 2.4
void **setContentLength**(int len)	この応答のコンテント・ボディ部の長さをセットする。HTTP サーブレットにおいては、このメソッドは HTTP Content-Length ヘッダをセットする。 引数: len - そのクライアントに返すコンテントの長さを指定する整数;Content-Length ヘッダをセットする
void **setContentType**(java.lang.String type)	その応答がまだコミットされていないならば、そのクライアントに送られる応答のコンテント・タイプをセットする。与えられるコンテント・タイプには、例えば text/html;charset=UTF-8 のように文字エンコーディング指定が含まれていて良い。この応答の文字エンコーディングは、getWriter が呼ばれる前にこのメソッドが呼ばれているときにのみこのコンテント・タイプからセットされる。 このメソッドは文字エンコーディングを変えるために繰り返し呼ぶことができる。このメソッドは getWriter が呼ばれたあと、またはこの応答がコミットされた後では効果をもたらさない。この応答がコミットされた後あるいは getWriter が呼ばれた後でこのメソッドが呼ばれているときは、この応答の文字エンコーディングをセットしない。 コンテナは、そのプロトコルがそうする手段を用意しているときは、そのサーブレットの応答の writer に使われているコンテント・タイプと文字エンコーディングをそのクライアントに連絡しなければならない。HTTP の場合では、その為に Content-Type ヘッダが使われている。 引数: type - そのコンテントの MIME タイプを指定する String
void **setBufferSize**(int size)	その応答のボディのための好ましいバッファ・サイズをセットする。サーブレット・コンテナは少なくとも要求された大きさのバッファを使用する。使用されている実際のバッファ・サイズは getBufferSize で知ることが出来る。バッファが大きければそれだけ多くのコンテントが実際に送信される前に書き込むことが出来るので、サーブレットはしかるべきステータス・コードとヘッダをセットする時間的余裕が大きくなる。小さなバッファだとサーバのメモリが少なくて済み、クライアントはより迅速に受信開始を始める。 このメソッドは何らかの応答ボディが書き込まれる前に呼ばれなければならない。コンテントが既に書かれている、あるいはその応答オブジェクトが既にコミットされているときは、このメソッドは IllegalStateException をスローする。 引数: size – 好ましいバッファ・サイズ 例外: java.lang.IllegalStateException - コンテントが既に書き込まれているあとでこのメソッドが呼ばれた
int **getBufferSize**()	この応答のために使われている実際のバッファ・サイズ。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 を返す。
void **flushBuffer**() throws java.io.IOException	このバッファのコンテントをクライアントに書き込ませる。このメソッド呼び出しで応答は自動的にコミットされ、それはステータス・コードとヘッダたちが書き込まれることを意味する。
void **resetBuffer**()	ヘッダまたはステータス・コードをクリアすることなく、その応答内の内部で使われているバッファのコンテントをクリアする。この応答が既にコミットされているときは、このメソッドは IllegalStateException をスローする。 導入: Servlet 2.3
boolean **isCommitted**()	その応答がコミットされているかどうかを示すブール値を返す。コミットされた応答では既にステータス・コードとヘッダたちが書き込まれてしまっている。
void **reset**()	ステータス・コードとヘッダたちを含めてそのバッファ内に存在するデータの総てをクリアする。この応答が既にコミットされているときは、このメソッドは IllegalStateException をスローする。
void **setLocale**(java.util.Locale loc)	その応答が未だコミットされていなければ、その応答のロケールをセットする。このメソッドはまた、setContentType(java.lang.String)あるいは setLocale(java.util.Locale)で明示的に文字エンコーディングがまだセットされていないとき、getWriter が未だ呼ばれていないとき、そしてその応答が未だコミットされていないときには、そのロケールに適切なその応答の文字エンコーディングをセットする。もし配備記述子が locale-encoding-mapping-list 要素を含んでおり、その要素が与えられたロケールのマッピングを含んでいれば、そ

	のマッピングが使われる。そ出ないときは、ロケールから文字エンコーディングへのマッピングはコンテナに依存する。 このメソッドは文字エンコーディングを変えるために繰り返し呼ぶことができる。このメソッドは getWriter が呼ばれたあと、またはこの応答がコミットされた後では効果をもたらさない。この応答がコミットされた後、あるいは getWriter が呼ばれた後、あるいは setCharacterEncoding(java.lang.String)が呼ばれたあとでこのメソッドが呼ばれているときは、このメソッドは効果をもたらさない。 コンテナは、そのプロトコルがそうする手段を用意しているときは、そのサーブレットの応答の writer に使われているロケールと文字エンコーディングをそのクライアントに連絡しなければならない。HTTP の場合では、その為に Content-Language ヘッダが、テキスト・メディア・タイプの為の Content-Type ヘッダの一部としての文字エンコーディングが、使われている。 そのサーブレットがコンテント・タイプを指定していないとき、そのコンテナは HTTP のヘッダを使って文字エンコーディングを伝えることはできないものの、このメソッドはそのサーブレットの writer を介して書かれるテキストのエンコードに使われることに注意されたい。 引数: loc - その応答のための Locale
java.util.Locale **getLocale**()	setLocale(java.util.Locale)メソッドを使ってこの応答のために指定されているロケールを返す。応答がコミットされた後での setLocale は効果をもたらさない。ロケールが指定されていないときは、そのコンテナのデフォルトのロケールが返される。

16.1.3.2 HTTPServletResponse

public interface **HttpServletResponse** extends ServletResponse 応答を送信する際に HTTP 固有の機能を提供する為に ServletResponse インターフェイスを継承。例えば、このインターフェイスは HTTP ヘッダとクッキーにアクセスするメソッドたちを備えている。 サーブレット・コンテナは HttpServletResponse オブジェクトを生成し、これをそのサーブレットの service メソッドたち(doGet, doPost, etc)に引数として渡す。	
フィールド ステータス・コードに関しては第 2.5 節のステータス行の表を参照のこと	
SC_ACCEPTED	ある要求が処理のために受け付けられたがその処理がまだ完了していないことを示すステータス・コード(202)
SC_BAD_GATEWAY	その HTTP サーバがプロキシまたはゲートウェイとして動作しているときにそれが相談したサーバから無効な応答を受信したことを示すステータス・コード(502)
SC_BAD_REQUEST	そのクライアントが送信した要求が文法的に正しくなかったことを示すステータス・コード(400) Status code (400) indicating the request sent by the client was syntactically incorrect.
SC_CONFLICT	そのリソースの現在の状態と矛盾している為その要求を完了できなかったことを示すステータス・コード(409)
SC_CONTINUE	クライアントが継続できることを示すステータス・コード(100)
SC_CREATED	その要求が成功しサーバ上に新しいリソースが作られたことを示すステータス・コード(201)
SC_EXPECTATION_FAILED	そのサーバが Expect 要求ヘッダで与えられた期待を満足できなかったことを示すステータス・コード(417)
SC_FORBIDDEN	そのサーバがその要求を理解したがそれを満足させることを拒否したことを示すステータス・コード(403)
SC_FOUND	そのリソースが別の URI のもとに一時的に移っていることをを示すステータス・コード(302)
SC_GATEWAY_TIMEOUT	そのサーバがゲートウェイまたはプロキシとして機能しているときに上流からタイムリに応答を受けなかったことを示すステータス・コード(504)
SC_GONE	そのリソースがそのサーバではもはや取得できず、転送するアドレスも分からなかったことを示すステータス・コード(410)
SC_HTTP_VERSION_NOT_SUPPORTED	その要求メッセージに使われている HTTP プロトコル・バージョンに対応していない、あるいは対応することを拒否したことを示すステータス・コード(505)
SC_INTERNAL_SERVER_ERROR	HTTP サーバ内のエラーによりその要求を満足できないことを示すステータス・コード(500)
SC_LENGTH_REQUIRED	Content-Length が指定されていないのでその要求を処理できないことを示すステータス・コード(411)
SC_METHOD_NOT_ALLOWED	その要求行で指定されているメソッドはその要求 URI で指定されているリソースでは許されていないことを示すステータス・コード(405)

SC_MOVED_PERMANENTLY	そのリソースは恒久的に新しい場所に移っており今後の参照はその要求では新しい URI を使わねばならないことを示すステータス・コード(301)
SC_MOVED_TEMPORARILY	そのリソースは一時的に新しい場所に移っているが、今後の参照はそのリソースにアクセスするのにオリジナルの URI を使わねばならないことを示すステータス・コード(302)
SC_MULTIPLE_CHOICES	要求されたリソースが、各々がそれぞれの場所を持つ表現たちのセットのどれか一つに対応していることを示すステータス・コード(300)
SC_NO_CONTENT	その要求の処理は成功したが、送り返すべき新しい情報がなかったことを示すステータス・コード(204)
SC_NON_AUTHORITATIVE_INFORMATION	クライアントが提示したメタ情報はそのサーバからの発信されたものではないことを示すステータス・コード(203)
SC_NOT_ACCEPTABLE	その要求によって特定されたリソースが、その要求で送られてきた accept ヘッダに従うと対応できないコンテント特性を持っている応答物しか発生できないことを示すステータス・コード(406)
SC_NOT_FOUND	要求されているリソースが取得できないことを示すステータス・コード(404)
SC_NOT_IMPLEMENTED	その HTTP サーバがその要求を満足させる機能に対応していないことを示すステータス・コード(501)
SC_NOT_MODIFIED	条件付き GET 操作でそのリソースは取得できるが、変更がされていないことを示すステータス・コード(304)
SC_OK	その要求は正常に成功したことを示すステータス・コード(200)
SC_PARTIAL_CONTENT	そのサーバはそのリソースに対する部分 GET 要求を満たしたことを示すステータス・コード(206)
SC_PAYMENT_REQUIRED	将来用途のために予約されたステータス・コード(402)
SC_PRECONDITION_FAILED	ひとつあるいはそれ以上の要求ヘッダ・フィールドで与えられている前提条件は、そのサーバでテストした際に評価できなかったことを示すステータス・コード(412)
SC_PROXY_AUTHENTICATION_REQUIRED	そのクライアントは先ずプロキシの認証を受けねばならないことを示すステータス・コード(407)
SC_REQUEST_ENTITY_TOO_LARGE	そのサーバは要求物が処理したくないまたは処理できないほど大きいので、その要求処理を拒否したことを示すステータス・コード(413)
SC_REQUEST_TIMEOUT	そのクライアントはそのサーバが待っている時間内に要求をしなかったことを示すステータス・コード(408)
SC_REQUEST_URI_TOO_LONG	要求 URI がそのサーバが解釈できるよりも長かったなおで、そのサーバはその要求のサービスを拒否していることを示すステータス・コード(414)
SC_REQUESTED_RANGE_NOT_SATISFIABLE	そのサーバは要求されたバイト幅で対応出来なかったことを示すステータス・コード(416)
SC_RESET_CONTENT	その要求送信を起こすことになったドキュメント・ビューをそのエージェントがリセットしなければならないことを示すステータス・コード(205)
SC_SEE_OTHER	その要求に対する応答は別の URL で見つけられることを示すステータス・コード(303)
SC_SERVICE_UNAVAILABLE	その HTTP サーバが一時的に過負荷になっており、その要求を処理できないことを示すステータス・コード(503)
SC_SWITCHING_PROTOCOLS	そのサーバが Upgrade ヘッダに従ってプロトコルを切り替えていることをことを示すステータス・コード(101)
SC_TEMPORARY_REDIRECT	要求されているリソースは一時的に別の URI にあることを示すステータス・コード(307)
SC_UNAUTHORIZED	その要求は HTTP 認証が必要であることを示すステータス・コード(401)
SC_UNSUPPORTED_MEDIA_TYPE	要求されているものが、その要求メソッドに対し要求されているリソースでは対応されないフォーマットであるので、そのように対するサービスを拒否したことを示すステータス・コード(415)
SC_USE_PROXY	要求されたそのリソースは、Location フィールドで与えられたプロキシ経由でアクセスされねばならないことを示すステータス・コード(305)
メソッド	
void **addCookie**(Cookie cookie)	指定されたクッキーをその応答に付加する。このメソッドをひとつ以上のクッキーをセットする為に複数回呼ぶことが可能である。
boolean **containsHeader**(java.lang.String name)	指定した応答ヘッダが既にセットされているかを示すブール値を返す 引数: name – ヘッダ名 戻り値: その名前の応答ヘッダが既にセットされていれば true、それ以外は false
java.lang.String **encodeURL**(java.lang.String url)	指定された URL に対し、それにセッション ID を含めることでエンコードする、あるいはもしエンコーディングの必要がなければ、変更なしでその URL を返す。このメソッドに実装には、その URL にセッション ID をエン

	コードする必要があるかどうかの判断のロジックが含まれる。例えば、そのブラウザがクッキーに対応しているとき、あるいはセッション追跡がオフになっているときは、URL エンコードは不要である。 しっかりしたセッション追跡のためには、サーブレットが出す総ての URL はこのメソッドを通すべきである。そうしないと、クッキーに対応しないブラウザで URL 書き換えが使えなくなる。 引数: url - エンコードする URL 戻り値: エンコードが必要なときはエンコードされた URL、それ以外は変更しないままの URL
java.lang.String **encodeRedirectURL**(java.lang.String url)	指定された URL を sendRedirect メソッドで使うためにエンコードする、あるいはエンコーディングが必要でない場合は、変更なしの URL を返す。このメソッドに実装には、その URL にセッション ID をエンコードする必要があるかどうかの判断のロジックが含まれる。この判断の為のルールは、ノーマル・リンクをエンコードするかどうかを判断するひとによって異なる可能性があるので、このメソッドは encodeURL メソッドと区別させている。 HttpServletResponse.sendRedirect メソッドに送られる総ての URL はこのメソッドを通すべきである。そうしないと、クッキーに対応しないブラウザで URL 書き換えが使えなくなる。 引数: url - エンコードする URL 戻り値: エンコードが必要なときはエンコードされた URL、それ以外は変更しないままの URL 参照: sendRedirect(java.lang.String), encodeUrl(java.lang.String)
java.lang.String **encodeUrl**(java.lang.String url)	廃止対象:第 2.1 版時点では代わりに encodeURL(String url)を使用のこと
java.lang.String **encodeRedirectUrl**(java.lang.String url)	廃止対象:第 2.1 版時点では代わりに encodeRedirectURL(String url)を使用のこと
void **sendError**(int sc, java.lang.String msg) throws java.io.IOException	指定されたステータスを使ってクライアントにエラー応答を送信し、バッファをクリアする。サーバのデフォルトではコンテント・タイプには"text/html"をセットして、指定されたメッセージを含んだ HTML 書式のエラー・ページのような応答を作る。サーバはクッキーを保持し、有効な応答としてのエラー・ページに必要なヘッダをクリアまたは更新できる。渡されたステータス・コードに対応してそのウェブ・アプリケーションに対し既にエラー・ページ宣言がされているときは、そのメッセージが優先され、msg 引数は無視される。 応答が既にコミットされているときはこのメソッドは IllegalStateException をスローする。このメソッドを使ったあとでは、この応答はコミットされたと考えられ、書き込みはできない。 引数: sc – エラー・ステータス・コード msg – エラー記述メッセージ 例外: java.io.IOException - 入出力例外発生 java.lang.IllegalStateException - 応答がコミットされている
void **sendError**(int sc) throws java.io.IOException	指定されたステータスを使ってクライアントにエラー応答を送信し、バッファをクリアする。サーバのデフォルトではコンテント・タイプには"text/html"をセットして、指定されたメッセージを含んだ HTML 書式のエラー・ページのような応答を作る。サーバはクッキーを保持し、有効な応答としてのエラー・ページに必要なヘッダをクリアまたは更新できる。渡されたステータス・コードに対応してそのウェブ・アプリケーションに対し既にエラー・ページ宣言がされているときは、そのメッセージがエラー・ページとして返される。 応答が既にコミットされているときはこのメソッドは IllegalStateException をスローする。このメソッドを使ったあとでは、この応答はコミットされたと考えられ、書き込みはできない。 引数: sc – エラー・ステータス・コード 例外: java.io.IOException - 入出力例外発生 java.lang.IllegalStateException - 応答がコミットされている
sendRedirect void **sendRedirect**(java.lang.String location) throws java.io.IOException	指定されたリダイレクト先 URL を使ってそのクライアントに一時的リダイレクト応答を送信し、バッファをクリアする。バッファはこのメソッドがセットしたデータで置き換えられる。このメソッドを呼ぶとステータス・コードは SC_FOUND 302 (Found)にセットされる。このメソッドは相対 URL を受け付けることが出来る;サーブレット・コンテナはその応答をクライアントに送信する前にその相対 URL を絶対 URL に変換しなければならない。もし引数の location が'/'で始まっていないときは、コンテナはそれは現在の要求 URI に相対だと解釈する。引数の location が'/'で始まっているときは、コンテナはそれはサーブレット・コンテナのルートに対し相対だと解釈する。応答が既にコミットされているときはこのメソッドは IllegalStateException をスローする。このメソッドを使ったあとでは、この応答はコミットされたと考えられ、書き込みはできない。 引数: location – リダイレクト先 URL 例外:

	java.io.IOException - 入出力例外発生 java.lang.IllegalStateException - 応答がコミットされている
void **setDateHeader**(java.lang.String name, long date)	与えられた名前とdate-valueを持った応答ヘッダをセットする。このdateはエポック時からのmSで規定されている。このヘッダが既にセットされているときには、新しい値でこれまでの値を置き換える。ContainsHeaderメソッドで値をセットする前にあるヘッダが存在しているかどうかのテストに使える。 引数: name - セットするヘッダの名前 date - 指定されたdate値 参照: containsHeader(java.lang.String), addDateHeader(java.lang.String, long)
void **addDateHeader**(java.lang.String name, long date)	与えられた名前とdate-valueを持った応答ヘッダを追加する。このdateはエポック時からのmSで規定されている。このメソッドにより応答ヘッダに複数の値を持たせることが出来る。 引数: name - セットするヘッダの名前 date - 付加するdate値
void **setHeader**(java.lang.String name, java.lang.String value)	与えられた名前と値を持ったヘッダをセットする。そのヘッダが既にセットされているときには、新しい値でこれまでの値を置き換える。ContainsHeaderメソッドで値をセットする前にあるヘッダが存在しているかどうかのテストに使える。 引数: name - セットするヘッダの名前 value - 付加する値。これが8ビット文字列(訳者注:非ASCII文字)を含んでいる場合は、これはRFC 2047 (http://www.ietf.org/rfc/rfc2047.txt)に従ってエンコードされねばならない。
void **addHeader**(java.lang.String name, java.lang.String value)	与えられた名前と値を持ったヘッダを追加する。このメソッドにより複数の値を持った応答ヘッダが可能になる。 引数: name - セットするヘッダの名前 value - 付加する値。これが8ビット文字列(訳者注:非ASCII文字)を含んでいる場合は、これはRFC 2047 (http://www.ietf.org/rfc/rfc2047.txt)に従ってエンコードされねばならない。
void **setIntHeader**(java.lang.String name, int value)	与えられた名前とinteger値を持った応答ヘッダをセットする。このヘッダが既にセットされているときには、新しい値でこれまでの値を置き換える。ContainsHeaderメソッドで値をセットする前にあるヘッダが存在しているかどうかのテストに使える。 引数: name - そのヘッダの名前 value - 指定されたinteger値
void **addIntHeader**(java.lang.String name, int value)	与えられた名前とinteger値を持ったヘッダを追加する。このメソッドにより複数の値を持った応答ヘッダが可能になる。 引数: name - そのヘッダの名前 value - 指定されたinteger値
void **setStatus**(int sc)	その応答にステータス・コードをセットする。 このメソッドはエラーがないときに戻りステータス(例えばSC_OKあるいはSC_MOVED_TEMPORARILYステータス・コード)をセットするのに使われる。 エラー・コードをセットする為にこのメソッドが使われているときには、そのコンテナのエラー・ページのメカニズムは起動しない。エラーが存在していて、発信者がそのウェブ・アプリケーションで決めたエラー・ページを呼び出したいときには、この代わりにsendError(int, java.lang.String)を使わねばならない。 このメソッドはクッキー及び他の応答ヘッダたちを残す。 有効なステータス・コードは2XX, 3XX, 4XX,及び5XXのレンジである。その他のステータス・コードはコンテナに依存する。
void **setStatus**(int sc, java.lang.String sm)	廃止対象。第2.1版時点では、メッセージ・パラメタの意味が曖昧であるため。ステータス・コードをセットするにはsetStatus(int)を、記述付きのエラーを送るにはsendError(int, String)を使用のこと。
java.lang.String **getHeader**(java.lang.String name)	与えられた名前の応答ヘッダの値を取得する。 与えられた名前の応答ヘッダが存在し、複数の値が含まれているときは、最初に付加された値が返される。 このメソッドはsetHeader(java.lang.String, java.lang.String), addHeader(java.lang.String, java.lang.String), setDateHeader(java.lang.String, long), addDateHeader(java.lang.String, long), setIntHeader(java.lang.String, int),あるいはaddIntHeader(java.lang.String, int)でセットまたは付加されたヘッダを対象にしている。 引数: name - 値を返す応答ヘッダの名前 戻り値: 与えられた名前の応答ヘッダの値、あるいはその名前のヘッダがセットされていないときはnull 導入:

	Servlet 3.0
java.util.Collection<java.lang.String> **getHeaders**(java.lang.String name)	与えられた名前の応答ヘッダの値たちを取得する。 このメソッドは setHeader(java.lang.String, java.lang.String), addHeader(java.lang.String, java.lang.String), setDateHeader(java.lang.String, long), addDateHeader(java.lang.String, long), setIntHeader(java.lang.String, int),あるいは addIntHeader(java.lang.String, int)でセットまたは付加されたヘッダを対象にしている。 戻された Collection に対する変更はこの HttpServletResponse には影響を与えない。 引数: name - 値を返す応答ヘッダの名前 戻り値: 与えられた名前の応答ヘッダの値たちの Collection(空もあり) 導入: Servlet 3.0
java.util.Collection<java.lang.String> **getHeaderNames**()	この要求のヘッダたちの名前たちを返す このメソッドは setHeader(java.lang.String, java.lang.String), addHeader(java.lang.String, java.lang.String), setDateHeader(java.lang.String, long), addDateHeader(java.lang.String, long), setIntHeader(java.lang.String, int),あるいは addIntHeader(java.lang.String, int)でセットまたは付加されたヘッダを対象にしている。 戻された Collection に対する変更はこの HttpServletResponse には影響を与えない。 戻り値: 与えられた名前の応答ヘッダの名前たちの Collection(空もあり) 導入: Servlet 3.0

16.1.4 セッション関係のクラス、インターフェイス、及びメソッド

public interface **HttpServletRequest** extends ServletRequest このインターフェイスにこのクライアントへのセッションの取得とその状況を得るメソッドが含まれているので、その部分のみを再掲した。	
public HttpSession **getSession**()	この要求に対応する現在のセッションを返す。この要求がセッションを持たないときは新しいセッションを作る。
public HttpSession **getSession**(boolean create)	この要求に対応した現在の HttpSession を返すか、現行のセッションが存在せず、かつ create が true のときは、新しいセッションを返す。create が false で、かつこの要求が有効(varid)な HttpSession を有さないときは、このメソッドは null を返す。**セッションが正しく維持されるためにはこのメソッドを応答がコミットされる前に呼ばねばならない** true – 必要な場合は新規のセッションをこの要求に生成する。 false – 現在セッションが存在していないときは null を返す。
java.lang.String **getRequestedSessionId**()	そのクライアントによって指定されているセッション ID を返す。これはその要求に対して現在有効なセッションの ID と異なっている可能性もある。そのクライアントがセッション ID を指定してきていないときは、このメソッドは null を返す。
public boolean **isRequestedSessionIdValid**()	指定されたセッション ID がまだ有効かを返す。
public boolean **isRequestedSessionIdFromCookie**()	要求されたセッション ID がクッキーでなされているかどうかを返す。
public boolean **isRequestedSessionIdFromURL**()	要求されたセッション ID が URI でなされているかどうかを返す。
public interface **HttpServletResponse** extends ServletResponse このインターフェイスにこのクライアントへのセッションの取得とその状況を得るメソッドが含まれているので、その部分のみを再掲した。	
public java.lang.String **encodeURL**(java.lang.String url)	指定の URL に session ID を含めた形にエンコードする。もしエンコードの必要がなければ入力した URL がそのまま返される。エンジンは session ID を URL に含めるべきか否かのロジックを持つ。例えば、ブラウザがクッキを受け付けたりセッション管理がオフになったりしていれば URL エンコーディングは不必要である。確実なセッション管理を得るためには、このメソッドを通した URL をブラウザに渡すべきである。でないと、クッキーをサポートしないブラウザに URL 再書き込みをさせられない。 url: エンコードする URL Returns: エンコードされた URL,またはそのまま
public java.lang.String **encodeRedirectURL**(java.lang.String url)	SendRedirect メソッドに使うために指定した URL をエンコードする。もしエンコードの必要がなければ入力した URL がそのまま返される。エンジンは session ID を URL に含めるべきか否かのロジックを持つ。通常のリンクにエンコードするか否かの判断が必要なので encodeURL メソッドとは区別される。 HttpServletResponse.sendRedirect メソッドに送る全ての URL はこのメソッドを通すべきである。でないと、クッキーをサポートしないブラウザに URL 再書き込みをさせられない。 url: エンコードする URL Returns: エンコードされた URL,またはそのまま
public interface **HttpSession** ひとつ以上のページに亘ってあるウェブサイトに訪問あるいは要求したりしているユーザの識別と、そのユーザに関する情報を蓄積する手段を提供する。サーブレット・コンテナは HTTP クライアントと HTTP サーバ間のセッションを生成するのに本インターフェイスを使う。セッションはそのユーザからの一回以上の接続またはページ要求に亘って一定の時間継続する。セッションは通常はこのサイトに何回も訪問する一人のユーザに対応する。サーバはクッキーや URL 再書き込みなどのいろんな手段でセッションを維持する。このインターフェイスは、 • セッション識別子、生成時刻、最後にアクセスした時刻などのセッションに関する情報の読み出しと変更 • オブジェクトをセッションにバインドして、そのユーザ情報が複数のユーザ接続においても継続できるようにする アプリケーションがオブジェクトをあるセッションにストアしたり反対にセッションから削除する際は、そのセッションはそのオブジェクトが HttpSessionBindingListener をインプリメントしているかどうかをチェックする。もしそうであると、そのサーブレットは該オブジェクトに対しすでにそれがバインドされたとかそのセッションからバイドが外されてしまったとかの通知を行う。 サーブレットはクライアントでクッキーが意図的に使用不可にされているなど、クライアントがセッションに参加しないことを選択している場合に対処できなければならない。そのクライアントがセッションに参加する前は isNew は true を返す。クライアントがセッション不参加の選択をしているときは、getSession は各要求ごとに異なったセッションを返し、isNew は常に true を返す。 セッション情報は現在のウェブ・アプリケーション(ServletContext)の範囲でのみ適用され、ひとつのコンテキストにストアされている情報は、他のコンテキストからは直接は不可視である。	
public long **getCreationTime**()	本セッション生成された時刻を January 1, 1970 の真夜中からのミリ秒の long 型で返す。無効(invalid)なセッションでこのメソッドを呼ぶと java.lang.IllegalStateException がスローされる。
public java.lang.String **getId**()	本セッションに付与された唯一無二の識別子を含む文字列を返す。識別子はサーブレット・コンテナが付与し、インプリメントに依存する。
public long **getLastAccessedTime**()	このセッションでクライアントが最後に要求を送信した時刻を January 1, 1970 の真夜中からのミリ秒の long 型で返す。本セッションに関わる値の取得や設定などのアプリケーションが行うアクションに対してはこの時間は影響を受けない。
GetServletContext ServletContext **getServletContext**()	そのセッションが属している ServletContext を返す。

	導入: Servlet 2.3
public void **setMaxInactiveInterval**(int interval)	サーブレット・コンテナがこのセッションを無効としないクライアントの要求間のインターバルを int 値の秒で設定する。マイナスの値を設定すると永久にタイムアウトを生じない。
public int **getMaxInactiveInterval**()	サーブレット・コンテナが本セッションを維持するクライアントのアクセスとアクセスの間の最大時間を int 値の秒で返す。このインターバルを超えるとサーブレット・コンテナはこのセッションを無効とする。
public java.lang.Object **getAttribute**(java.lang.String name)	本セッションにバインドされている指定した名前のオブジェクトを返す。その名前のオブジェクトがバインドされていないときは null を返す。name はそのオブジェクトの名前を指定する文字列である。このメソッドが無効(invalid)となっているセッションで呼ばれたときは java.lang.IllegalStateException をスローする。
public java.util.Enumeration **getAttributeNames**()	本セッションにバインドされている全てのオブジェクトの名前を含む String オブジェクトの列挙型を返す。このメソッドが無効(invalid)となっているセッションで呼ばれたときは java.lang.IllegalStateException をスローする。
public void **setAttribute**(java.lang.String name, java.lang.Object value)	指定した名前であるオブジェクトを本セッションにバインドする。このセッションに同じ名前のオブジェクトが既にバインドされているときはオブジェクトは置き換えられる。このメソッドを実行し、このオブジェクトが HttpSessionBindingListener を実装しているときは、コンテナは HttpSessionBindingListener.valueBound を呼ぶ。 name: それでこのオブジェクトをバインドする名前で、null は不可。 value: バインドするオブジェクトで null は不可。 このメソッドが無効(invalid)となっているセッションで呼ばれたときは java.lang.IllegalStateException をスローする。
public void **removeAttribute**(java.lang.String name)	指定した名前でバインドされているオブジェクトを削除する。指定した名前のオブジェクトがバインドされていないときは本メソッドは何もしない。このメソッドが呼ばれ、またそのオブジェクトが HttpSessionBindingListener を実装しているときは、サーブレット・コンテナは HttpSessionBindingListener.valueUnbound を呼ぶ。 name: それでこのオブジェクトをバインドする名前。 このメソッドが無効(invalid)となっているセッションで呼ばれたときは java.lang.IllegalStateException をスローする。
public void **invalidate**()	本セッションを無効とし、これにバインドされている全てのオブジェクトをバインドから外す。このメソッドが無効(invalid)となっているセッションで呼ばれたときは java.lang.IllegalStateException をスローする。
public boolean **isNew**()	クライアントがまだこのセッションを知らないとき、またはクライアントがセッションに参加しない選択をしているときに true を返す。例えば、サーバがクッキー・ベースのセッションだけを使っていて、かつクライアントがクッキーの使用を不可としているときは、セッションは各要求ごとに new の状態となる。 true: サーバがセッションを生成したがクライアントがまだこのセッションに参加していない。 このメソッドが無効(invalid)となっているセッションで呼ばれたときは java.lang.IllegalStateException をスローする。
public interface **HttpSessionActivationListener** extends java.util.EventListener あるセッションにバインドされたオブジェクトたちは、セッションが活性化されるあるいは不活性化されることを通知するコンテナ・イベントを聞くことが出来る。セッションを VM たち間で移行させる、あるいはセッションを永続化させるコンテナでは、HttpSessionActivationListener を実装したセッションにバインドされた総ての属性を通知する必要がある。 導入:Servlet 2.3	
void **sessionWillPassivate**(HttpSessionEvent se)	そのセッションが非活性化されようとしていることの通知
void **sessionDidActivate**(HttpSessionEvent se)	そのセッションが活性化されたばかりだということに通知
public interface **HttpSessionAttributeListener** extends java.util.EventListener HttpSession 属性の変更に関するイベント通知を受けるためのインターフェイス。 このイベント通知を受けるためには、実装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中で宣言されているか、WebListener でアノテートされているか、あるいは ServletContext の addListener のメソッドたちのひとつで登録されているかしなければならない。 このインターフェイスの実装が呼び出される順番は規定されていない。 実装:Servlet 2.3	
void **attributeAdded**(HttpSessionBindingEvent event)	あるセッションにある属性が追加されたことの通知を受ける。 引数: event - そのセッションと追加された属性の名前と値を含む HttpSessionBindingEvent
void **attributeRemoved**(HttpSessionBindingEvent event)	あるセッションからある属性が除去されたことの通知を受ける。 引数: event - そのセッションと除去された属性の名前と値を含む HttpSessionBindingEvent
void **attributeReplaced**(HttpSessionBindingEvent event)	あるセッションの中である属性が置き換えられたことの通知を受ける。 引数: event - そのセッションと置き換えられた属性の名前と(古い)値を含む HttpSessionBindingEvent
public class **HttpSessionBindingEvent** extends java.util.EventObject HttpSessionBindingListener を実装しているオブジェクトに対してこのオブジェクトがそのセッションにバインドされたり反対にバインドが外されたときに送出する。セッションは HttpSession.putValue を呼ぶことでオブジェクトをバインドし、HttpSession.removeValue を呼ぶことでバインドを外す。	

HttpSessionBindingEvent(HttpSession session, java.lang.String name)	オブジェクトに対してそれがバインドされたかバインドから外されたということを通知するイベントを生成する。このイベントを受信するには、そのオブジェクトは HttpSessionBindingListener を実装していなければならない。 session – オブジェクトをバインドまたはバインドから外すセッション name – それでオブジェクトをバインドまたはバインドから外される名前
public java.lang.String **getName**()	それでオブジェクトがセッションからバインドまたはバインドから外されている名前を文字列で返す。
public HttpSession **getSession**()	そこにオブジェクトがバインドされているかそこからオブジェクトがバインドをとかれたかしているセッションを返す。
public interface **HttpSessionBindingListener** extends java.util.EventListener オブジェクトがセッションにバインドされたりまたバインドから外されたときに通知を受信できるようにする。このオブジェクトへは HttpSessionBindingEvent のオブジェクトで通知される。	
public void **valueBound**(HttpSessionBindingEvent event)	このオブジェクトにこれがあるセッションにバインドされたことを通知し、そのセッションを知る。 event – そのセッションを識別するイベント
public void **valueUnbound**(HttpSessionBindingEvent event)	このオブジェクトにこれがあるセッションからのバインドが外されたことを通知し、そのセッションを知る。 event – そのセッションを識別するイベント
public class **HttpSessionEvent** extends java.util.EventObject これはあるウェブ・アプリケーション内のセッションたちに変更があったことの通知イベントを表現したクラスである。 実装：Servlet 2.3	
HttpSessionEvent(HttpSession source)	与えられた HttpSession からセッション・イベントを生成する
HttpSession **getSession**()	変化したセッションを返す
public interface **HttpSessionListener** extends java.util.EventListener HttpSession のライフサイクルの変化に関する通知イベントを受けるためのインターフェイス。 このイベント通知を受けるためには、実装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中で宣言されているか、WebListener でアノテートされているか、あるいは ServletContext の addListener のメソッドたちのひとつで登録されているかしなければならない。 このインターフェイスの実装たちが sessionCreated(javax.servlet.http.HttpSessionEvent)メソッドで呼び出される順番はそれが宣言された順番に、sessionDestroyed(javax.servlet.http.HttpSessionEvent)では逆順となる。 実装：Servlet 2.3	
void **sessionCreated**(HttpSessionEvent se)	あるセッションが生成されたことの通知を受ける 引数： se - そのセッションを含む HttpSessionEvent
void **sessionDestroyed**(HttpSessionEvent se)	あるセッションが無効になろうとしていることの通知を受ける。 引数： se - そのセッションを含む HttpSessionEvent
public interface **ServletContext** あるファイルの MIME タイプを取得する、要求をディスパッチする、あるいはログ・ファイルに書き込むといった、サーブレットが自分のサーブレット・コンテナと通信するために使うメソッドたちのセットを定める。	
java.util.Set<SessionTrackingMode> **getDefaultSessionTrackingModes**()	この ServletContext の為にデフォルトとしてサポートされているセッション追跡モードを取得する。
java.util.Set<SessionTrackingMode> **getEffectiveSessionTrackingModes**()	この ServletContext で実効的であるセッション追跡モードを取得する。
SessionCookieConfig **getSessionCookieConfig**()	SessionCookieConfig オブジェクトを取得し、これによりこの ServletContext に代わって作られたセッション追跡クッキーのいろんな属性を設定可能にする。
void **setSessionTrackingModes**(java.util.Set<SessionTrackingMode> sessionTrackingModes)	この ServletContext の為に実効的になるセッション追跡モードをセットする。
public enum **SessionTrackingMode** extends java.lang.Enum<SessionTrackingMode> これはセッション追跡モードの列挙クラス 導入： Servlet 3.0	
ENUM 定数	
COOKIE	public static final SessionTrackingMode COOKIE
URL	public static final SessionTrackingMode URL
SSL	public static final SessionTrackingMode SSL
メソッド	

public static SessionTrackingMode[] **values**()	この enum 型の定数たちを宣言された順番で含んだ配列を返す。このメソッドはこれらの定数で iterate を使って以下のように使えよう: for (SessionTrackingMode c : SessionTrackingMode.values()) System.out.println(c); 戻り値: この enum 型の定数たちを宣言された順番で含んだ配列
public static SessionTrackingMode **valueOf**(java.lang.String name)	指定された名前をもったこの型の enum 定数を返す。文字列はこの型の enum 定数として使われている識別子と正確に一致していなければならない。(ホワイトスペースの付加は許されない) 引数: name - 返されるべき enum 定数の名前 戻り値: 指定された名前を持った enum 定数 例外: java.lang.IllegalArgumentException - この enum 型が指定された名前を持った定数を持っていない java.lang.NullPointerException - 引数が null

16.1.5 コンテキスト関係のインターフェイス、クラス、及びメソッドたち

16.1.5.1 Servlet

public interface **Servlet** これは総てのサーブレットが実装しなければならないメソッドたちを定めている。 サーブレットはウェブ・サーバ内で走る小さな Java プログラムである。サーブレットたちは通常は HTTP でウェブ・クライアントたちからの要求を受け、またその要求に応答する。 このインターフェイスの実装に際しては javax.servlet.GenericServlet を継承したジェネリックなサーブレットを書く、あるいは javax.servlet.http.HttpServlet を継承した HTTP サーブレットを書くことになる。 このインターフェイスはサーブレットの初期化、要求に対するサービス、及びそのサーバからサーブレットを外す、為のメソッドたちが定められている。これらはライフ・サイクル・メソッドとして知られており、以下のシーケンスで呼ばれる: 1. サーブレットがコンストラクトされ、次に init メソッドで初期化される 2. クライアントからの service メソッドへの呼び出しが処理される 3. そのサーブレットがサービスから外され、destroy メソッドで破棄され、ガベージ・コレクトされ、終了される これらのライフ・サイクル・メソッドたちに加え、このインターフェイスにはそのサーブレットがスタートアップ情報を取得する為の getServletConfig メソッド、及びそのサーブレットが著者、バージョン、及び著作権などの自分自身の基本情報を返すことが出来る getServletInfo メソッドが用意されている。	
void **init**(ServletConfig config) throws ServletException	サーブレットに対しそのサーブレットがサービス開始状態に置かれていることを示す為にサーブレット・コンテナが呼び出す。サーブレット・コンテナはそのサーブレットをインスタンス化したあとで一度だけ呼び出す。この init メソッドは正常に終了しないとそのサーブレットは要求を受け付けることが出来ない。 サーブレット・コンテナはこの init メソッドで以下の時にはこのサーブレットをサービス可能に出来ない: 1. ServletException がスローされた 2. そのウェブ・サーバが定めた時間内に戻らなかった 引数: config - そのサーブレットの設定と初期化のためのパラメタたちを含んだ ServletConfig オブジェクト 例外: ServletException - そのサーブレットの通常動作を阻害する例外が発生した
ServletConfig **getServletConfig**()	このサーブレットの為の初期化と開始パラメタたちを含んだ ServletConfig オブジェクトを返す。返される ServletConfig オブジェクトは init メソッドに渡されたものである。 このインターフェイスの実装は、この ServletConfig オブジェクトをストアし、このメソッドがそれを返せるようにしなければならない。このインターフェイスを実装している GenericServlet クラスは既にこれを行っている。 戻り値: このサーブレットを初期化する ServletConfig オブジェクト
void **service**(ServletRequest req, ServletResponse res) throws ServletException java.io.IOException	そのサーブレットが要求に応じるようにサーブレット・コンテナがこのメソッドを呼び出す。このメソッドはそのサーブレットの init メソッドが正しく終了したときにのみ呼び出される。 その応答のステータス・コードは常に、エラーをスローするあるいはエラーを送信するサーブレットによってセットされねばならない。 サーブレットたちは通常、複数の要求を並行して処理できるよう、マルチスレッド化されたサーブレット・コンテナの内部で動作する。開発者たちはファイルたち、ネットワーク接続たち、及びそのサーブレットのクラス及びインスタンス変数たちのような共有されるオブジェクトたちへのアクセスを同期化することに注意しなければならない。マルチスレッドのプログラミングに関する更なる情報は、マルチスレッド・プログラミングに関する Java のチュートリアルで得ることが出来る。 引数: req - そのクライアントの要求を含んだ ServletRequest オブジェクト t res - そのサーブレットの応答を含んだ ServletResponse オブジェクト 例外: ServletException - そのサーブレットの正常な動作を阻害する例外が発生した

	java.io.IOException - 入出力の例外が発生した
java.lang.String **getServletInfo**()	著者、バージョン、及び著作権のようなそのサーブレットに関する情報を返す。このメソッドが返す String はプレーンなテキストであって、HTML、XML のようなマークアップであってはならない。
void **destroy**()	サーブレット・コンテナがサーブレットに対し、そのサーブレットがサービスから外されていることを示す為に呼ぶ。このメソッドはそのサーブレットの service メソッド内にあるスレッドたちが全部抜けた後、あるいはある時間が経過した後でのみ一度呼び出される。サーブレット・コンテナがこのメソッドを読んだあとは、サーブレット・コンテナはこのサーブレットの service メソッドを再度呼ぶことはしない。 このメソッドによりそのサーブレットは所有していたリソースたち（例えばメモリ、ファイル・ハンドル、スレッドたち）をクリーンアップし、何らかの永続状態がメモリ内のこのサーブレットの現在の状態と確実に同期できる機会を持つことが出来る。

16.1.5.2 ServletConfig

public interface **ServletConfig** サーブレット・コンテナが初期化中にサーブレットに情報を渡すのに使うサーブレットの設定オブジェクトである。	
java.lang.String **getServletName**()	このサーブレット・インスタンスの名前を返す。この名前はサーバの管理者が用意し、アプリケーション配備記述子の中で指定されるか、未登録（したがって無名）サーブレットの場合はそのサーブレットのクラス名が返される。 戻り値： このサーブレット・インスタンスの名前
ServletContext **getServletContext**()	呼び出し側が実行している SevletContext への参照を返す。 戻り値： 呼び出し側がそのサーブレット・コンテナと関わりあうために使う ServletContext オブジェクト。
java.lang.String **getInitParameter**(java.lang.String name)	与えられた名前の初期化パラメタの値を返す。 引数： name – その値を取得する初期化パラメタの名前 戻り値： その初期化パラメタの値を含む String、あるいは存在しないときは null
java.util.Enumeration<java.lang.String> **getInitParameterNames**()	そのサーブレットの初期化パラメタたちの名前を String オブジェクトの Enumeration として返す、あるいはそのサーブレットが初期化パラメタを持っていないときは空の Enumeration を返す。

16.1.5.3 GenericServlet

public abstract class **GenericServlet** extends java.lang.Object implements Servlet, ServletConfig, java.io.Serializable 汎用でプロトコルに依存しないサーブレットを規定している。ウェブ上で使う為に HTTP サーブレットを書くには、これではなくて HttpServlet を継承する。 GenericServlet は Servlet 及び ServletConfig インターフェイスを実装している。GenericServlet は直接サーブレットが継承しても良いが、HttpServlet のようなプロトコル固有のサブクラスを継承するのがより一般的である。 GenericServlet はサーブレットをより簡単に書けるようにしている。これには ServletConfig インターフェイスの init 及び destroy のライフサイクル・メソッドたちが用意されている。GenericServlet はまた ServletContext インターフェイスで宣言されているlogメソッドを実装している。 汎用サーブレットを書くにはこの抽象 service メソッドをオーバライドするだけで良い。	
コンストラクタ	
GenericServlet()	何もしない。
メソッド	
public void **destroy**()	Servlet.destroy()を参照のこと。

public java.lang.String **getInitParameter**(java.lang.String name)	ServletConfig.getInitParameter(java.lang.String)を参照のこと。
public java.util.Enumeration<java.lang.String> **getInitParameterNames**()	ServletConfig.getInitParameterNames()を参照のこと。
public ServletConfig **getServletConfig**()	Servlet.getServletConfig()を参照のこと。
public ServletContext **getServletContext**()	ServletConfig.getServletContext()を参照のこと。
public java.lang.String **getServletInfo**()	Servlet.getServletInfo()を参照のこと。
public void **init**(ServletConfig config) throws ServletException	Servlet.init(javax.servlet.ServletConfig)を参照のこと。
public void **init**() throws ServletException	オーバライド出来、super.init(config)を呼ぶ必要がない利便性の為のメソッド。init(ServletConfig)をオーバライドする代わりに、単にこのメソッドをオーバライドするだけで良く、これは GenericServlet.init(ServletConfig config)によって呼ばれる。ServletConfig オブジェクトは getServletConfig()を介して取得できる。 例外: ServletException - そのサーブレットの正常な動作を止める例外が発生した。i
public void **log**(java.lang.String msg)	指定したメッセージをそのサーブレットの名前を頭にしてサーブレットのログ・ファイルに書き込む。ServletContext.log(String)を参照のこと。
public void **log**(java.lang.String message, java.lang.Throwable t)	指定したメッセージと、指定した Throwable 例外のスタック・トレースを、そのサーブレットの名前を頭にしてサーブレットのログ・ファイルに書き込む。ServletContext.log(String, Throwable)を参照のこと。
public abstract void **service**(ServletRequest req, ServletResponse res) throws ServletException, java.io.IOException	Servlet.service(javax.servlet.ServletRequest, javax.servlet.ServletResponse)を参照のこと。 このメソッドは abstract 宣言されているので、HttpServlet のようなサブクラスはこれをオーバライドしなければならない。
public java.lang.String **getServletName**()	ServletConfig.getServletName()を参照のこと。

16.1.5.4 ServletContext

public interface **ServletContext**	
あるファイルの MIME タイプを取得する、要求をディスパッチする、あるいはログ・ファイルに書き込むといった、サーブレットが自分のサーブレット・コンテナと通信するために使うメソッドたちのセットを定める。 JVM あたり「ウェブ・アプリケーション」あたりひとつのコンテキストが存在する。(「ウェブ・アプリケーション」とは、/catalog のようなそのサーバの URL 名前空間の特定のサブセット、のもとでインストールされるサーブレットたちとコンテントの集まりであって、.war ファイルによってインストールされ得る。) その配備記述子で"distributed"とマークされたウェブ・アプリケーションの場合は、各 VM ごとにひとつのコンテキストのインスタンスが存在することになる。この状況のときは、そのコンテキストはグローバルな情報を共有する場所としては使えない。(なぜなら、その情報は真にグローバルな物とならないからである。)この場合は代わりにデータベースのような外部リソースを使用する。 ServletContext オブジェクトは ServletConfig オブジェクトの中に含まれており、この ServletConfig オブジェクトはそのサーブレットが初期化されるときにウェブ・サーバによってそのサーブレットに提供される。	
フィールド	
static java.lang.String **ORDERED_LIBS**	ServletContext 属性名で、その値(java.util.List<java.lang.String>型)は、それらのウェブ・フラグメント名で順序付けされた WEB-INF/lib 内の Jar ファイル(<others/>なしで<absolute-ordering>が使われているときは除外される、という可能性あり)たちの名前のリストを含む、あるいは絶対または相対順序付けが指定されていないときは null である。
static java.lang.String **TEMPDIR**	ServletContext 属性名で、その ServletContext に対しサーブレット・コンテナが提供するプライベートな一時ディレクトリ(java.io.File 型)をストアする。
メソッド一覧	

FilterRegistration.Dynamic **addFilter**(java.lang.String filterName, java.lang.Class<? extends Filter> filterClass)	このサーブレット・コンテキストに与えられた名前とクラス型を持ったフィルタを追加する。この登録されたフィルタは戻される FilterRegistration オブジェクトを介して更に設定できる。 この ServletContext が既に与えられた filterName を持ったフィルタの為の一時的 FilterRegistration を含んでいるときは、それは完了され（それに与えられた filterClass の名前を割り当てることで）、戻される。 このメソッドは、与えられた filterClass がある Managed Bean を示しているときは、リソース注入に対応する。Managed Beans とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を見られたい。 引数： filterName - そのフィルタの名前 filterClass – そこからこのフィルタがインスタンス化されるクラス・オブジェクト 戻り値： この登録されたフィルタを更に設定するのに使える FilterRegistration オブジェクト、あるいは与えられた filterName をもったフィルタの為の完了した FilterRegistration をこの ServletContext が含んでいるときには null 例外： java.lang.IllegalStateException - この ServletContext が既に初期化されている java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml で宣言されてもいなければ WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されている 導入： Servlet 3.0
FilterRegistration.Dynamic **addFilter**(java.lang.String filterName, Filter filter)	与えられた filterName のもとでこのサーブレット・コンテキストに与えられたフィルタのインスタンスを追加する。この登録されたフィルタは戻される FilterRegistration オブジェクトを介して更に設定できる。 この ServletContext が既に与えられた filterName を持ったフィルタの為の暫定 FilterRegistration を含んでいるときは、それは完了され（それに与えられた filterClass の名前を割り当てることで）、戻される。 Parameters: filterName - そのフィルタの名前 filter - 登録するフィルタ・インスタンス 戻り値： この登録されたフィルタを更に設定するのに使える FilterRegistration オブジェクト、あるいは同じコンテナ内にこのあるいは別の ServletContext 内に同じフィルタ・インスタンスが既に登録されているときには null 例外： java.lang.IllegalStateException - この ServletContext が既に初期化されている java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml で宣言されてもいなければ WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されている 導入： Servlet 3.0
FilterRegistration.Dynamic **addFilter**(java.lang.String filterName, java.lang.String className)	このサーブレット・コンテキストに与えられた名前とクラス名を持ったフィルタを追加する。この登録されたフィルタは戻される FilterRegistration オブジェクトを介して更に設定できる。 指定されたこの className はこの ServletContext を代表するアプリケーションに結び付けられたクラス・ローダを使ってロードされる。 この ServletContext が既に与えられた filterName を持ったフィルタの為の暫定 FilterRegistration を含んでいるときは、それは完了され（それに与えられた filterClass の名前を割り当てることで）、戻される。 このメソッドは、与えられた filterClass がある Managed Bean を示しているときは、リソース注入に対応する。Managed Beans とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を見られたい。 Parameters: filterName - そのフィルタの名前 className - そのフィルタの完全修飾クラス名 戻り値： この登録されたフィルタを更に設定するのに使える FilterRegistration オブジェクト、あるいはこの ServletContext が与えられた filterName を持ったフィルタの為の完了した FiltrResistration を既に含んでいるときには null

	例外： java.lang.IllegalStateException - この ServletContext が既に初期化されている java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml で宣言されてもいなければ WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されている 導入： Servlet 3.0
void **addListener**(java.lang.Class<? extends java.util.EventListener> listenerClass)	与えられたクラス型のリスナをこの ServletContext に付加する。与えられた listenerClass は以下のインターフェイスたちのひとつあるいはそれ以上を実装していなければならない： ·ServletContextAttributeListener ·ServletRequestListener ·ServletRequestAttributeListener ·HttpSessionListener ·HttpSessionAttributeListener もしこの ServletContext が ServletContainerInitializer.onStartup(java.util.Set>, javax.servlet.ServletContext)メソッドによって渡されていたときは、この与えられた listenerClass はまた上記のインターフェイスたちに加えて ServletContextListener を実装していても良い。 与えられた listenerClass が、その呼び出し順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場合（言い換えればそれが ServletRequestListener、ServletContextListener、あるいは 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づけられたリストの最後に付加される。 与えられた filterClass がある Managed Bean を示しているときは、このメソッドはリソース注入に対応する。Managed Beans とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を見られたい。 引数： listenerClass – インスタンス化するリスナ・クラス 例外： java.lang.IllegalArgumentException - もし与えられた listenerClass が上記インターフェイスたちのどれをも実装したいないとき、あるいはこの ServletContext が ServletContainerInitializer.onStartup(java.util.Set>, javax.servlet.ServletContext)に渡されていなかったとき。 java.lang.IllegalStateException - この ServletContext が既に初期化されているとき java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されていないとき 導入： Servlet 3.0
void **addListener**(java.lang.String className)	与えられたクラス名のリスナをこの ServletContext に付加する。与えられた listenerClass は以下のインターフェイスたちのひとつあるいはそれ以上を実装していなければならない： ·ServletContextAttributeListener ·ServletRequestListener ·ServletRequestAttributeListener ·HttpSessionListener ·HttpSessionAttributeListener もしこの ServletContext が ServletContainerInitializer.onStartup(java.util.Set>, javax.servlet.ServletContext)メソッドによって渡されていたときは、この与えられた listenerClass はまた上記のインターフェイスたちに加えて ServletContextListener を実装していても良い。 このメソッドの呼び出しの一部として、コンテナはそれが要求されているインターフェイスたちのひとつを確実に実装していることを確保する為に、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。 与えられた名前のそのクラスが、その呼び出し順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場合（言い換えればそれが ServletRequestListener、ServletContextListener、あるいは 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づけられたリストの最後に付加される。 与えられた className をもったクラスが Managed Bean を示しているときは、このメソッドはリソース注入に対応する。Managed Beans とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を見られたい。

	引数: className – そのリスナの完全修飾クラス名 例外: java.lang.IllegalArgumentException - もし与えられた listenerClass が上記インターフェイスたちのどれをも実装したいないとき、あるいはこの ServletContext が ServletContainerInitializer.onStartup(java.util.Set>, javax.servlet.ServletContext)に渡されていなかったとき。 java.lang.IllegalStateException - この ServletContext が既に初期化されているとき java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されていないとき 導入: Servlet 3.0
<T extends java.util.EventListener> void **addListener**(T t)	指定されたリスナをこの ServletContext に付加する。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あるいはそれ以上を実装していなければならない: ·ServletContextAttributeListener ·ServletRequestListener ·ServletRequestAttributeListener ·HttpSessionListener ·HttpSessionAttributeListener もしこの ServletContext が ServletContainerInitializer.onStartup(java.util.Set>, javax.servlet.ServletContext)メソッドによって渡されていたときは、この与えられたリスナはまた上記のインターフェイスたちに加えて ServletContextListener を実装していても良い。 与えられたリスナが、その呼び出し順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場合(言い換えればそれが ServletRequestListener、ServletContextListener、あるいは HttpSessionListener のインスタンスであるとき)は、そのリスナはそのインターフェイスのリスナたちの順序づけられたリストの最後に付加される。 引数: t - t 付加するリスナ 例外: java.lang.IllegalArgumentException - もし与えられた listenerClass が上記インターフェイスたちのどれをも実装したいないとき、あるいはこの ServletContext が ServletContainerInitializer.onStartup(java.util.Set>, javax.servlet.ServletContext)に渡されていなかったとき。 java.lang.IllegalStateException - この ServletContext が既に初期化されているとき java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されていないとき 導入: Servlet 3.0
ServletRegistration.Dynamic **addServlet**(java.lang.String servletName, java.lang.Class<? extends Servlet> servletClass)コンプリートになっている	このサーブレット・コンテキストに与えられた名前とクラス型をもったサーブレットを付加する。登録されたサーブレットは戻される ServletRegistration オブジェクトを介して更に設定できる。 この ServletContext が既に与えられた servletName をもったサーブレットの為の暫定 ServletRegistration を含んでいる場合は、それはコンプリートにされ(それに与えられた servletClass の名前を割り当てることで)、戻される。 このメソッドは与えられた servletClass の為に、ServletSecurity、MultipartConfig、javax.annotation.security.RunAs、及び javax.annotation.security.DeclareRoles のアノテーションたちを内部検査する。加えて、このメソッドはこの servletClass が Managed Bean である場合は、リソース注入に対応している。Managed Beans とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を見られたい。 引数: servletName - そのサーブレットの名前 servletClass - そのサーブレットがインスタンス化されるクラス・オブジェクト 戻り値: 登録されたサーブレットを更に設定する為に使われる ServletRegistration、またはこの ServletContext が既に与えられた servletName の為の ServletRegistration を含んでいるときは null 例外: java.lang.IllegalStateException - この ServletContext が既に初期化されているとき java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどち

	らにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
ServletRegistration.Dynamic **addServlet**(java.lang.String servletName, Servlet servlet)	指定された servletName のもとで与えられたサーブレット・インスタンスをこの ServletContext に登録する。登録されたサーブレットは、返される ServleiRegistration オブジェクトを介して更に設定できる。 この ServletContext が既に与えられた servletName をもったサーブレットの為の暫定 ServletRegistration を含んでいる場合は、それはコンプリートにされ(それに与えられたサーブレット・インスタンスのクラス名を割り当てることで)、戻される。 引数: servletName - そのサーブレットの名前 servlet - 登録するサーブレット・インスタンス 戻り値: 指定されたサーブレットを更に設定する為に使われる ServletRegistration オブジェクト、またはこの ServletContext が既に与えられた servletName の為のコンプリートされた ServletRegistration を含んでいるとき、あるいは同じコンテナ内でこのあるいは別の ServletContext に同じサーブレット・インスタンスが既に登録されているときは null 例外: java.lang.IllegalStateException - この ServletContext が既に初期化されているとき java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
ServletRegistration.Dynamic **addServlet**(java.lang.String servletName, java.lang.String className)	与えられた名前とクラス名を持ったサーブレットをこのサーブレット・コンテキストに付加する。登録されたサーブレットは、返される ServleiRegistration オブジェクトを介して更に設定できる。 指定された className はこの ServletContext で表現されるアプリケーションに結び付けられたクラス・ローダを使ってロードされることになる。 この ServletContext が既に与えられた servletName をもったサーブレットの為の暫定 ServletRegistration を含んでいる場合は、それはコンプリートにされ(それに与えられたサーブレット・インスタンスのクラス名を割り当てることで)、戻される。 このメソッドは与えられた servletClass の為に、ServletSecurity、MultipartConfig、javax.annotation.security.RunAs、及び javax.annotation.security.DeclareRoles のアノテーションたちを内部検査する。加えて、このメソッドはこの servletClass が Managed Bean である場合は、リソース注入に対応している。Managed Beans とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を見られたい。 引数: servletName - このサーブレットの名前 className – このサーブレットの完全修飾クラス名 戻り値: 登録されたサーブレットを更に設定する為に使われる ServletRegistration オブジェクト、またはこの ServletContext が既に与えられた servletName の為のコンプリートされた ServletRegistration を含んでいるとき、は null 例外: java.lang.IllegalStateException - この ServletContext が既に初期化されているとき java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
<T extends Filter> T **createFilter**(java.lang.Class<T> clazz)	与えられた Filter クラスをインスタンス化する。返された Filter インスタンスは addFilter(String,Filter)呼び出しでこの ServletContext に登録される前に更にカスタム化するのに使われる。 与えられる Filter クラスは、それをインスタンス化するのに使われる引数が無いコンストラクタが定義されていなければならない。 与えれらた clazz が Managed Bean を表しているときは、このメソッドはリソース注入に対応する。Managed Beans とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を見られたい。

	引数: clazz – インスタンス化する Filter クラス 戻り値: 新しい Filter インスタンス 例外: ServletException - 与えられた clazz のインスタンス化できない java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
<T extends java.util.EventListener> T **createListener**(java.lang.Class<T> clazz)	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、ServletContextListener、ServletContextAttributeListener、ServletRequestListener、ServletRequestAttributeListener、HttpSessionListener、あるいは HttpSessionAttributeListener インターフェイスの少なくともひとつを実装していなければならない。 返される EventListener インスタンスは、addListener(EventListener)呼び出しでこの ServletContext に登録される前に更にカスタム化出来る。 与えられた EventListner クラスは、それをインスタンス化するのに使われる引数が無いコンストラクタが定義されていなければならない。 与えれらた clazz が Managed Bean を表しているときは、このメソッドはリソース注入に対応する。Managed Beans とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を見られたい。 引数: clazz – インスタンス化する EventListner クラス 戻り値: 新しい EventListner インスタンス 例外: ServletException - 与えられた clazz のインスタンス化できない java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき java.lang.IllegalArgumentException - 指定された EventLister クラスが ServletContextListener、ServletContextAttributeListener、ServletRequestListener、ServletRequestAttributeListener、HttpSessionListener、あるいは HttpSessionAttributeListener インターフェイスのどれをも実装していない。 導入: Servlet 3.0
<T extends Servlet> T **createServlet**(java.lang.Class<T> clazz)	与えられた Servlet クラスをインスタンス化する。返される Servlet インスタンスは、addServlet(String,Servlet)呼び出しでこの ServletContext に登録される前に更にカスタム化できる。 与えられる Servlet クラスは、インスタンス化のために使われる引数なしのコンストラクタが定義されていなければならない。 このメソッドは与えられた clazz の為に、ServletSecurity、MultipartConfig、javax.annotation.security.RunAs、及び javax.annotation.security.DeclareRoles のアノテーションたちを内部検査する。加えて、このメソッドはこの clazz が Managed Bean である場合は、リソース注入に対応している。Managed Beans とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を見られたい。 引数: clazz – インスタンス化する Servlet クラス 戻り値: 新しい Servlet インスタンス 例外: ServletException - 与えられた clazz のインスタンス化できない java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0

void **declareRoles**(java.lang.String. roleNames)	isUserInRole でテストされるロール名たちを宣言する。ServletRegistration インターフェイスの setServletSecurity あるいは setRunAsRole メソッドでそれらを使った結果として明示的に宣言されるロールたちは、宣言される必要が無くなる。 引数: roleNames - 宣言するロール名たち 例外: java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき java.lang.IllegalArgumentException - 引数 roleNames のどれかが null または空の文字列のとき java.lang.IllegalStateException – 既にこの ServletContext が初期化されている 導入: Servlet 3.0
java.lang.Object **getAttribute**(java.lang.String name)	このサーブレット・コンテナの与えられた名前の属性を返す、あるいはその名前の属性が存在しないときは null を返す。属性により、サーブレット・コンテナはそのサーブレットに対し、既にこのインターフェイスで提供されていない付加的情報を与えることが出来る。この属性に関する情報は、サーバのドキュメンテーションを見られたい。対応している属性たちのリストは getAttributeNames をつかって調べることが出来る。 返される属性は java.lang.Object またはそのサブクラスである。 属性名はパッケージ名と同じ規約に従うべきである。Java Servlet API 仕様書は java.*、javax.*、及び sun.* にあった名前を予約している。 引数: name – その属性の名前を指定する String 戻り値: その属性を値を含む Object、あるいは与えられた名前に正確に合致する属性が無い場合は null
java.util.Enumeration<java.lang.String> **getAttributeNames**()	この ServletContext 内で得られる属性の名前たちを含む Enumeration を返す。ある属性の値を所得する為には、属性名を指定した getAttribute(java.lang.String)を使うこと。 戻り値: 属性名たちの Enumeration
java.lang.ClassLoader **getClassLoader**()	この ServletContext で代表されているウェブ・アプリケーションのクラス・ローダを取得する。セキュリティ・マネージャが存在しており、呼び出し側のクラス・ローダが要求されたクラス・ローダと同じでない、または祖先である場合は、その要求されたクラス・ローダへのアクセスが許されるかどうかをチェックする為に、そのセキュリティ・マネージャの checkPermission メソッドが、RuntimePermission("getClassLoader")を使って呼ばれる。 戻り値: この ServletContext で代表されているウェブ・アプリケーションのクラス・ローダ 例外: java.lang.UnsupportedOperationException - この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき java.lang.SecurityException – I セキュリティ・マネージャが要求されたクラス・ローダへのアクセスを拒否している 導入: Servlet 3.0
ServletContext **getContext**(java.lang.String uripath)	そのサーバ上で指定された URL に対応した ServletContext オブジェクトを返す。このメソッドにより、サーブレットたちはそのサーバのいろんな要素の為のコンテキストにアクセスできるようになり、必要ならそのコンテキストから RequestDispatcher オブジェクトを取得できる。与えられるパスは'/'で始まり、それはそのサーバのドキュメント・ルートに対し相対だと解釈され、このコンテナ上でホストされている他のウェブ・アプリケーションのコンテキスト・ルートに対し合致しているものである。 セキュリティが問題である環境では、このサーブレット・コンテナは与えられた URL に対し null を返すことがある。 引数: uripath – そのコンテナ内の他のウェブ・アプリケーションを指定する String 戻り値: 指定された URL に対応した ServletContext オブジェクト、あるいは該当するオブジェクトが無いあるいはそのコンテナがそのアクセスを制限したいときは null
java.lang.String	そのウェブ・アプリケーションのコンテキスト・パスを返す。このコンテキスト・パスは要求 URI の一部であって

getContextPath()	その要求のコンテキストを選択するに使われている。このコンテキスト・パスは常に要求 URI のなかの最初に位置する。パスは'/'文字で始まるが'/'文字では終わらない。デフォルト(ルート)コンテキスト内のサーブレットの場合は、このメソッドは""を返す。 サーブレット・コンテナでひとつ以上のコンテキスト・パスであるコンテキストと一致がとれることがある。そのような場合には、HttpServletRequest.getContextPath()はその要求で使われている実際のコンテキスト・パスを返し、このメソッドが返すパスとは相違しているかもしれない。このメソッドによって返されるコンテキスト・パスはそのアプリケーションにとってプライムあるいは好ましいコンテキスト・パスだと考えるべきである。 戻り値: そのウェブ・アプリケーションのコンテキスト・パス、あるいはデフォルト(ルート)コンテキストの場合は"" 導入: Servlet 2.5
java.util.Set<SessionTrackingMode> **getDefaultSessionTrackingModes()**	この ServletContext がデフォルトで対応しているセッション追跡モードたちを返す。 戻り値: この ServletContext の為のデフォルトで対応しているセッション追跡モードたちのセット 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
int **getEffectiveMajorVersion()**	この ServletContext で代表されるアプリケーションがベースにしているサーブレット仕様書のメジャー・バージョンを取得する。 このメソッドで返される値は、このサーブレット・コンテナが対応しているサーブレット仕様書のメジャー・バージョンを返す getMajorVersion()で返されるものと異なっている可能性がある。 戻り値: この ServletContext で代表されるアプリケーションがベースにしているサーブレット仕様書のメジャー・バージョン 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
int **getEffectiveMinorVersion()**	この ServletContext で代表されるアプリケーションがベースにしているサーブレット仕様書のマイナ・バージョンを取得する。 このメソッドで返される値は、このサーブレット・コンテナが対応しているサーブレット仕様書のマイナ・バージョンを返す getMinorVersion()で返されるものと異なっている可能性がある。 戻り値: この ServletContext で代表されるアプリケーションがベースにしているサーブレット仕様書のマイナ・バージョン 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
java.util.Set<SessionTrackingMode> **getEffectiveSessionTrackingModes()**	この ServletContext で実効的であるセッション追跡モードたちを返す。実効的であるセッション追跡モードは setSessionTrackingModes メソッドで指定したものである。 デフォルトでは、getDefaultSessionTrackingModes が返すセッション追跡モードが実効的になっている。 戻り値: この ServletContext で実効的であるセッション追跡モードたちのセット 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0

FilterRegistration **getFilterRegistration**(java.lang.String filterName)	与えられた名前をもったフィルタに対応する FilterRegistration を取得する。 戻り値: 与えられた filterName を持ったフィルタの(完了または暫定)FilterRegistration、あるいはその名前の FilterRegistration が存在しないときは null 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
java.util.Map<java.lang.String,? extends FilterRegistration> **getFilterRegistrations**()	この ServletContext で登録されている総てのフィルタたちに対応した FilterRegistration オブジェクトたち(フィルタ名がキーになっている)の(空もあり)Map を取得する。返される Map には、総ての宣言された及びアノテートされたフィルタたちに対応した FilterRegistration のオブジェクトたちと、addFilter メソッドたちのひとつで付加された総てのフィルタたちに対応した FilterRegistration のオブジェクトたちが含まれる。 返された Map に対する何らかの変更は ServletContext に影響を与えてはいけない。 戻り値: この ServletContext で登録されている総てのフィルタたちに対応した(完了または暫定の)FilterRegistration オブジェクトたちの Map 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
java.lang.String **getInitParameter**(java.lang.String name)	名前がつけられたコンテキストにわたる初期化パラメタの値を含む String を返す、あるいはそのパラメタが存在しない場合は null を返す。このメソッドにより取得できる設定情報をウェブ・アプリケーション全体にわたって使えるようにする。例えばこれにより、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を提供できる。 引数: name – その値が必要なパラメタの名前を含む String 戻り値: 少なくともサーブレット・コンテナの名前とバージョン番号を含む String
java.util.Enumeration<java.lang.String> **getInitParameterNames**()	このコンテキストの初期化パラメタたちの名前を String オブジェクトたちの Enumeration として返す、あるいはそのコンテキストが初期化パラメタを持っていないときは空の Enumeration を返す。 戻り値: このコンテキストの初期化パラメタたちの名前を String オブジェクトたちの Enumeration
JspConfigDescriptor **getJspConfigDescriptor**()	この ServletContext によって代表されるウェブ・アプリケーションの web.xml と web-fragment.xml 記述ファイルたちから集められている<jsp-config>関連の設定を取得する。 戻り値: この ServletContext によって代表されるウェブ・アプリケーションの web.xml と web-fragment.xml 記述ファイルたちから集められている<jsp-config>関連の設定、あるいはそのような設定が無い場合は null を返す。 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0 参照: JspConfigDescriptor
int **getMajorVersion**()	このサーブレット・コンテナが対応している Servlet API のメジャー・バージョンを返す。Version 3.0 対応の総ての実装物は、このメソッドの戻り値を 3 にしなければならない。 戻り値: 3
java.lang.String	指定したファイルの MIME タイプを返す、あるいは MIME タイプが判らないときは null を返す。この MIME

getMimeType(java.lang.String file)	タイプはこのサーブレット・コンテナの設定で決まり、ウェブ・アプリケーション配備記述子の中で指定される。一般的な MIME タイプには text/html と image/gif がある。 引数: file – ファイルの名前を指定する String 戻り値: そのファイルの MIME タイプを指定した String
int **getMinorVersion**()	このサーブレット・コンテナが対応している Servlet API のマイナー・バージョンを返す。Version 3.0 対応の総ての実装物は、このメソッドの戻り値を 0 にしなければならない。 戻り値: 0
RequestDispatcher **getNamedDispatcher**(java.lang.String name)	名前が指定されたサーブレットのラッパーとして機能する RequestDispatcher オブジェクトを返す。サーブレットたち(JSP ページたちも含まれる)はサーバの管理を介して、あるいはウェブ・アプリケーション配備記述子を介して名前が付られ得る。サーブレット・インスタンスは ServletConfig.getServletName()を使ってその名前を判断できる。 ServletContext が何らかの理由で RequestDispatcher を返せないときはこのメソッドは null を返す。 引数: name – ラップする為のサーブレットの名前を指定する String 戻り値: 名前が指定されたサーブレットのラッパーとして機能する RequestDispatcher オブジェクト、あるいは ServletContext が RequestDispatcher を返せないときは null
java.lang.String **getRealPath**(java.lang.String path)	与えられたバーチャル・パスに対応した実際のパスを取得する。例えば、パスが/index.html に等しかったとすると、このメソッドは http://<host>:<port>/<contextPath>/index.html の形の要求がマップされるこのサーバ上の絶対ファイル・パスを返す。ここに<contextPath>はこのサーブレットのコンテキスト・パスに対応する。 返される実際のパスはこのサーブレット・コンテナが走っているコンピュータと OS に適した形式のものであり、しかるべきパス・セパレータが含まれる。 そのアプリケーションの/WEB-INF/lib ディレクトリ内にバンドルされた JAR ファイルたちの/META-INF/resources ディレクトリの中にあるリソースたちは、そのコンテナが含まれている JAR ファイルからそれらを解凍したときにのみ、検討される。この場合解凍された場所のパスが返されねばならない。 サーブレット・コンテナが与えられたバーチャル・パスを実際のパスに変換できないときは、このメソッドは null を返す。 引数: path – 実際のパスに変換するバーチャル・パス 戻り値: 実際のパス、あるいは変換できないときは null
RequestDispatcher **getRequestDispatcher**(java.lang.String path)	与えられたパスに位置するリソースの為のラッパとして機能する RequestDispatcher を返す。RequestDispatcher オブジェクトは、その要求をそのリソースにフォワードする、あるいはある応答にそのリソースを含める為に使える。そのリソースは静的なものあるいは動的なものであり得る。pathname は'/'で始まらねばならず、現在のコンテキスト・ルートに対し相対的なものであると解釈される。他のコンテキスト内のリソースに対する RequestDispatcher を取得するには、getContext を使用のこと。 ServletContext が RequestDispatcher を返せないときには、このメソッドは null を返す。 引数: path – そのリソースへの pathname を指定する String 戻り値: 指定したパスにあるリソースのためのラッパーとして動作する RequestDispatcher オブジェクト、あるいはその ServletContext が RequestDispatcher を返せないときは null
java.net.URL **getResource**(java.lang.String path)	与えられたパスにマップされているリソースに対する URL を返す。pathname は'/'で始まらねばならず、現在のコンテキスト・ルートに対し相対的なもの、あるいはそのウェブ・アプリケーションの WEB-INF/lib ディレクトリの中にある JAR ファイルの/META-INF/resources ディレクトリに対し相対的、であると解釈される。このメソッドは、/WEB-INF/lib 内の JAR ファイルをサーチする前に、最初に要求されたリソースの為にそのウェブ・アプリケーションのドキュメント・ルートをサーチする。 このメソッドにより、サーブレット・コンテナはサーブレットたちがどのリソースからリソースを取得できるようにする。リソースたちは、ローカルまたはリモートのファイル・システム上、データベース内、あるいは.war ファイル

	内にあり得る。 サーブレット・コンテナはリソースにアクセスするのに必要な URL ハンドラたちと URLConnection オブジェクトたちを実装しなければならない。 その pathname にリソースがマップされていないときは、このメソッドは null を返す。 一部のコンテナにあっては URL クラスたちのメソッドを使ってこのメソッドで返される URL に書き込むことを可能にしていることがある。 そのリソースのコンテントは直接返されるので、.jsp ページを要求すると JSP のソース・コードが返されるので注意のこと。実効の結果を含めるには、これではなくて RequestDispatcher を使う。 このメソッドは、クラス・ローダに基づいてリソースを検索する java.lang.Class.getResource とは違った目的を持っている。このメソッドはクラス・ローダたちを使わない。 引数: path – そのリソースへのパスを指定する String 戻り値: 指定されたパスにあるリソース、あるいはそのパスにリソースが無いときは null 例外: java.net.MalformedURLException - ipathname が正しい形式で与えられていない
java.io.InputStream **getResourceAsStream**(java.lang.String path)	指定されたパスにあるリソースを InputStream オブジェクトとして返す。その InputStream 内のデータはどの形式、どの長さでもあり得る。指定する path は getResource で記されたルールに従って指定しなければならない。指定されたパスにリソースが無いときは、このメソッドは null を返す。 getResource で取得できるコンテント長やコンテント・タイプのようなメタ情報は、このメソッドを使うと失われる。 サーブレット・コンテナはリソースにアクセスするのに必要な URL ハンドラたちと URLConnection オブジェクトたちを実装しなければならない。 このメソッドは、クラス・ローダに基づいてリソースを検索する java.lang.Class.getResourceAsStream とは違った目的を持っている。このメソッドはクラス・ローダたちを使わないで、サーブレット・コンテナがどの場所にあるリソースもサーブレットが取得できるようにしている。 引数: path – そのリソースへのパスを指定する String 戻り値: そのサーブレットに返す InputStream、あるいは指定されたパスにリソースが存在しないときは null
java.util.Set<java.lang.String> **getResourcePaths**(java.lang.String path)	そのウェブ・アプリケーション内のリソースに対する総てのパスをディレクトリ的なリストとして返し、そのリストの最長パスが指定したパス引数に合致する。サブディレクトリ・パスを示す path は'/'で終わる。 返されるパスは総て現在のコンテキスト・ルートに対し相対的なもの、あるいはそのウェブ・アプリケーションの WEB-INF/lib ディレクトリの中にある JAR ファイルの/META-INF/resources ディレクトリに対し相対的、なものであり、'/'が先頭につく。 例えば、以下のものを含むウェブ・アプリケーションの場合は: /welcome.html /catalog/index.html /catalog/products.html /catalog/offers/books.html /catalog/offers/music.html /customer/login.jsp /WEB-INF/web.xml /WEB-INF/classes/com.acme.OrderServlet.class /WEB-INF/lib/catalog.jar!/META-INF/resources/catalog/moreOffers/books.html getResourcePaths("/")は{"/welcome.html", "/catalog/", "/customer/", "/WEB-INF/"}を返し、 getResourcePaths("/catalog/")は{"/catalog/index.html", "/catalog/products.html", "/catalog/offers/", "/catalog/moreOffers/"}を返す。 引数: path - それらのリソースと一致をとるのにつかう部分パスで、'/'で始まらねばならない 戻り値: ディレクトリ・リスティングを含む Set、あるいはそのウェブ・アプリケーションに指定された path で始まるパスにリソースが無いときは null

	導入: Servlet 2.3
java.lang.String **getServerInfo**()	そのサーブレットが走っているサーブレット・コンテナの名前とバージョンを返す。返される String の形式は servername/versionnumber である。例えば、JavaServer Web Development Kit は JavaServer Web Dev Kit/1.0 という文字列を返す。 サーブレット・コンテナは主要文字列の後にカッコで括った他のオプショナルな情報(例えば JavaServer Web Dev Kit/1.0 (JDK 1.1.6; Windows NT 4.0 x86))を返しても良い。 戻り値:少なくともサーブレット・コンテナの名前とバージョン番号を含んだ String
Servlet **getServlet**(java.lang.String name)	Java Servlet API 2.1 時点で廃止対象となっており、直接の代替メソッドは無い。このメソッドは当初は ServletContext からサーブレットを取り出す為に定義された。このバージョンでは、このメソッドは常に null を返し、バイナリ互換性だけの為に存在している。このメソッドは将来の Java Servlet API のバージョンで恒久的に廃止される。 このメソッドの代わりに、サーブレットたちは ServletContext クラスを使って情報を共有でき、一般的な非サーブレットのクラスたちのメソッドたちを呼び出すことで共有されたビジネス・ロジックを実行できる。
java.lang.String **getServletContextName**()	このウェブ・アプリケーション配備記述子で display-name 要素を使って指定された、この ServletContext に対応したウェブ・アプリケーションの名前を返す。 戻り値: このウェブ・アプリケーションの名前、または配備記述子で宣言されていないときは null 導入: Servlet 2.3
java.util.Enumeration<java.lang.String> **getServletNames()**	Java Servlet API 2.1 時点で廃止対象になっており、代替メソッドは無い。このメソッドは当初はこのコンテキストで判っている総てのサーブレット名の Enumeration を返すために定義された。このメソッドは常に null を返し、バイナリ互換性だけの為に存在している。このメソッドは将来の Java Servlet API のバージョンで恒久的に廃止される。
ServletRegistration **getServletRegistration**(java.lang.String servletName)	与えられた servletName をもったサーブレットに対応した ServletRegistration を取得する。 戻り値: 与えられた servletName を持ったサーブレットの(暫定または完了状態の)ServletRegistration、あるいはその名前の ServletRegistration が存在しないときは null 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
java.util.Map<java.lang.String,? extends ServletRegistration> **getServletRegistrations**()	この ServletContext に登録されている総てのサーブレットたちに対応した ServletRegistration(サーブレット名がキーになっている)Map(空の場合もある)を取得する。返された Map 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たちに対応した ServletRegistration オブジェクトたち、及び addServlet のメソッドたちのひとつで追加された総てのサーブレットたちに対応した ServletRegistration オブジェクトたちが含まれる。 返された Map に対する何らかの変更は ServletContext に影響を与えてはならない。 戻り値: この ServletContext で登録されている総てのフィルタたちに対応した(完了または暫定の)ServletRegistration オブジェクトたちの Map 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
java.util.Enumeration<Servlet> **getServlets**()	Java Servlet API 2.1 時点で廃止対象になっており、代替メソッドは無い。このメソッドは当初はこのコンテキストで判っている総てのサーブレットの Enumeration を返すために定義された。このメソッドは常に null を返し、バイナリ互換性だけの為に存在している。このメソッドは将来の Java Servlet API のバージョンで恒久的に廃止される。
SessionCookieConfig **getSessionCookieConfig**()	この ServletContext に代わって作られたセッション追跡クッキーたちのいろんな属性が設定できる SessionCookieConfig オブジェクトを返す。このメソッドを繰り返し呼び出したときは、同じ

	SessionCookieConfig インスタンスが返される。 戻り値: この ServletContext に代わって作られたセッション追跡クッキーたちのいろんな属性が設定できる SessionCookieConfig オブジェクト 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
void **log**(java.lang.String msg)	指定したメッセージ(通常はイベント・ログ)をサーブレットのログ・ファイルに書き込む。サーブレットのログ・ファイルの名前とタイプはサーブレット・コンテナに依存する。 引数: msg - このログ・ファイルに書き込むメッセージを指定する String
void **log**(java.lang.String message, java.lang.Throwable throwable)	与えられた Throwable 例外の説明メッセージとスタック・トレースをサーブレットのログ・ファイルに書き込む。サーブレットのログ・ファイルの名前とタイプはサーブレット・コンテナに依存する。 引数: message - このエラーまたは例外を記述した String throwable - Throwable エラーまたは例外
void **removeAttribute**(java.lang.String name)	この ServletContext から与えられた名前の属性を除去する。除去後は、この属性の値を取得する為の getAttribute(java.lang.String)呼び出しは null を返す。もしリスナたちがこの ServletContext 上で設定されているときは、そのコンテナはそれに従って通知を行う。 引数: name - 除去する属性の名前を指定する String
void **setAttribute**(java.lang.String name, java.lang.Object object)	この ServletContext に属性として与えられた名前でオブジェクトをバインドする。指定された名前が既にある属性用に使われている場合は、このメソッドはその属性を新しい属性で置き換える。もしリスナたちがこの ServletContext 上で設定されているときは、そのコンテナはそれに従って通知を行う。 もし null の値が渡されたときは、その効果は removeAttribute()呼び出しと同じになる。 属性名はパッケージ名と同じ規約に従わねばならない。Java Servlet API 仕様書は java.*、javax.*、及び sun.*に合致する名前を予約している。 引数: name - その属性の名前を指定する String object - バインドされる属性を代表する Object
boolean **setInitParameter**(java.lang.String name, java.lang.String value)	この ServletContext 上で指定した値と名前を持った初期化パラメタをセットする。 引数: name - セットするコンテキスト初期化パラメタの名前 value – セットするコンテキスト初期化パラメタの値 戻り値: 与えられた名前と値を持ったコンテキスト初期化パラメタがこの ServletContext 上にセットされたときは true、この ServletContext が既に同じ名前のコンテキスト初期化パラメタを持っているときは false 例外: java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどちらにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき 導入: Servlet 3.0
void **setSessionTrackingModes**(java.util.Set<SessionTrackingMode> sessionTrackingModes)	この ServletContext で有効となるセッション追跡モードをセットする。与えられた sessionTrackingModes はこの ServletContext 上のこのメソッドのこれまでの呼び出しでセットされているセッション追跡モードに置き換わる。 引数: sessionTrackingModes – この ServletContext で有効となるセッション追跡モードのセット 例外: java.lang.IllegalStateException – この ServletContext が既に初期化されている。 java.lang.UnsupportedOperationException – この ServletContext が web.xml または web-fragment.xml のどち

	らにおいても宣言されていない、あるは WebListner でアノテートされていない ServletContextListener の ServletContextListener.contextInitialized(javax.servlet.ServletContextEvent)メソッドに渡されているとき java.lang.IllegalArgumentException – もしこの指定されたセッション追跡モードが、SessionTrackingMode.SSL 以外のセッション追跡モードと SessionTrackingMode.SSL の組み合わせを sessionTrackingModes が指定している、あるいは sessionTrackingModes がそのサーブレット・コンテナが対応していないセッション追跡モードを指定しているとき 導入: Servlet 3.0

16.1.6 フィルタ関係

16.1.6.1 Filter

public interface **Filter** フィルタとはあるリソース(サーブレットあるいは静的コンテント)への要求、あるいはあるリソースからの応答、あるいはその双方上で、フィルタリングのタスクを実行するオブジェクトである。 フィルタは doFilter メソッドの中でフィルタリング処理を行う。各フィルタはある FilterConfig オブジェクトへのアクセスを持ち、このオブジェクトからそのフィルタは初期化パラメタたち、及び例えばフィルタリングのタスクに必要なリソースをロードする為に使える ServletContext への参照、を取得できる。 フィルタたちはあるウェブ・アプリケーションの配備記述子のなかえ設定が出来る。 フィルタに適しているとされている事例としては: 1. 認証フィルタ 2. ログと監査のフィルタ 3. イメージ変換のフィルタ 4. データ圧縮フィルタ 5. 暗号化フィルタ 6. トークン化フィルタ 7. リソース・アクセス・イベントを生起させるフィルタ 8. XSL/T フィルタ 9. MIME タイプ・チェインのフィルタ 導入: Servlet 2.3	
メソッド	
void **init**(FilterConfig filterConfig) throws ServletException	ウェブ・コンテナがフィルタに対しそのフィルタをサービス開始しようとしていることを知らせる為に呼び出す。 サーブレット・コンテナはそのフィルタをインスタンス化した後で 1 回のみこの init メソッドを呼ぶ。この init メソッドは、そのフィルタが稼働前に成功裏に終了しなければならない。 ウェブ・コンテナは init メソッドが以下のどれかのときはそのフィルタを稼働状態に出来ない: • ServletException をスローした • ウェブ・コンテナが定めた時間内に戻らない
void **doFilter**(ServletRequest request, ServletResponse response, FilterChain chain) throws java.io.IOException, ServletException	この Filter の doFilter メソッドは、そのチェインの末端にあるリソースの為のクライアント要求により生起されたある要求/応答のペアがこのチェインを通過する度に、コンテナによって呼び出される。このメソッドが渡す FilterChain を使って、Filter はこのチェインの次にあるエンティティ(フィルタ、サーブレット、JSP ページ)に要求と応答を渡すことが出来る。 このメソッドの典型的な実装は、次のようなパタンになろう: 1. その要求を調べる 2. 入力フィルタリングの場合は、コンテントあるいはヘッダ行たちをフィルタリングする為にカスタム実装の要求オブジェクトをラップする 3. 出力フィルタリングの場合は、コンテントあるいはヘッダ行たちをフィルタリングする為にカスタム実装の応答オブジェクトをラップする 4. 以下のどれかをする: • FilterChain オブジェクトの chain.doFilter()を使って、このチェインの次のエンティティを呼び出す • その要求処理をブロックする為に、このフィルタ・チェインの次のエンティティにこの要求/応答のペアを渡さない 5. そのフィルタ・チェインの次のエンティティ呼び出し後にその応答のヘッダたちを直接セットする
void **destroy**()	ウェブ・コンテナがフィルタに対しそのフィルタをサービス状態から外そうとしていることを知らせる為に呼び出す。 このメソッドは、このフィルタの doFilter メソッド内の総てのスレッドがこのメソッドを出た後、あるいはタイムアウト期間を経過した後にのみ呼び出される。そのウェブ・コンテナがこのメソッドを呼んだあとは、コンテナはこのフィルタのインスタンス上の doFilter メソッドを呼ぶことをしない。 このメソッドにより、フィルタは保持しているリソースたち(例えばメモリ、ファイル・ハンドラ、スレッド)をクリーンアップし、何らかの継続性状態がメモリ内のこのフィルタの現在の状態と確実に同期しているようにする機会を持つことが出来る。

16.1.6.2 FilterChain

public interface **FilterChain** FilterChain はサーブレット・コンテナがソフトウエア開発者たちの為にあるリソースの為のフィルタリングされた要求を呼び出すことが可能なように用意するオブジェクトである。フィルタたちはこの FilterChain を使ってそのチェインの次のフィルタを呼び出す、あるいはそのフィルタがそのチェインの最後のフィルタであるときはそのチェインの最後にあるリソースを呼び出す。 導入: Servlet 2.3	
メソッド	
void **doFilter**(ServletRequest request, ServletResponse response) throws java.io.IOException, ServletException	そのチェインの次のフィルタを呼び出す、あるいは呼び出しもとのフィルタがそのチェインの最後のフィルタであるときは、そのチェインの最後にあるリソースが呼び出される。 引数: request - そのチェインに沿って渡す要求 response – そのチェインに沿って渡す応答 例外: java.io.IOException ServletException

16.1.6.3 FilterConfig

public interface **FilterConfig** フィルタの初期化中にそのフィルタに情報を渡す為にサーブレット・コンテナが使うフィルタ設定オブジェクト 導入: Servlet 2.3 参照: Filter	
メソッド	
java.lang.String **getFilterName**()	配備記述子で指定されたこのフィルタの filter-name 要素を返す。
ServletContext **getServletContext**()	呼び出し側が実行している ServletContext への参照を返す。 戻り値: そのサーブレット・コンテナと関わり合う為に呼び出し側が使う ServletContext オブジェクト 参照: ServletContext
java.lang.String **getInitParameter**(java.lang.String name)	指定された初期化パラメタの値を含む String、あるいはその初期化パラメタが存在しないときは null を返す。 引数: name – その初期化パラメタの名前を指定する String 戻り値: 指定された初期化パラメタの値を含む String、あるいはその初期化パラメタが存在しないときは null
java.util.Enumeration<java.lang.String> **getInitParameterNames**()	String オブジェクトの列挙型としてこのフィルタの初期化パラメタたちの名前を、あるいはそのフィルタが初期化パラメタを持っていないときは空の列挙型を返す。 戻り値: そのフィルタの初期化パラメタたちの名前を含む String オブジェクトたちの Enumeration

16.1.6.4 HttpServletRequestWrapper

public class **HttpServletRequestWrapper** extends ServletRequestWrapper implements HttpServletRequest その要求をあるサーブレットに合わせたいと思うデベロッパたちがサブクラス化出来る HttpServletRequest インターフェイスの便利な実装物を提供している。 このクラスは Wrapper あるいは Decorator パタンを採用している。メソッドたちはデフォルトでラップされた要求オブジェクトのメソッドたちを呼ぶ。各メソッドの詳細は HttpServletRequest インターフェイスの同じ名前のメソッドを参照されたい。 Servlet 2.3 から導入

コンストラクタ	
public **HttpServletRequestWrapper**(HttpServletRequest request)	与えられた要求をラップする要求オブジェクトを構築する。 例外: java.lang.IllegalArgumentException - その要求が null のとき。
メソッド	
下記以外に javax.servlet.ServletRequestWrapper から引き継いでいるメソッドたちは次のようである: getAsyncContext, getAttribute, getAttributeNames, getCharacterEncoding, getContentLength, getContentType, getDispatcherType, getInputStream, getLocalAddr, getLocale, getLocales, getLocalName, getLocalPort, getParameter, getParameterMap, getParameterNames, getParameterValues, getProtocol, getReader, getRealPath, getRemoteAddr, getRemoteHost, getRemotePort, getRequest, getRequestDispatcher, getScheme, getServerName, getServerPort, getServletContext, isAsyncStarted, isAsyncSupported, isSecure, isWrapperFor, isWrapperFor, removeAttribute, setAttribute, setCharacterEncoding, setRequest, startAsync, startAsync	
public java.lang.String **getAuthType**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getAuthType()を返すことである。 戻り値: static なメンバ変数である BASIC_AUTH, FORM_AUTH, CLIENT_CERT_AUTH, DIGEST_AUTH (== 比較に適す)、あるいはその認証スキームを示すコンテナ固有の文字列、あるいはその要求が認証されていないときは null。
public Cookie[] **getCookies**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getCookies()を返すことである。
public long **getDateHeader**(java.lang.String name)	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getDateHeader(String name)を返すことである。
public java.lang.String **getHeader**(java.lang.String name)	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getHeader(String name)を返すことである。
public java.util.Enumeration<java.lang.String> **getHeaders**(java.lang.String name)	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getHeaders(String name)を返すことである。
public java.util.Enumeration<java.lang.String> **getHeaderNames**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getHeaderNames()を返すことである。
public java.lang.String **getMethod**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getMethod()を返すことである。
public java.lang.String **getPathInfo**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getPathInfo()を返すことである。
public java.lang.String **getPathTranslated**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getPathTranslated()を返すことである。
public java.lang.String **getContextPath**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getContextPath()を返すことである。
public java.lang.String **getQueryString**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getQueryString()を返すことである。
public java.lang.String **getRemoteUser**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getRemoteUser()を返すことである。
public boolean **isUserInRole**(java.lang.String role)	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で isUserInRole(String role)を返すことである。
public java.security.Principal **getUserPrincipal**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getUserPrincipal()を返すことである。
public java.lang.String **getRequestedSessionId**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getRequestedSessionId()を返すことである。
public java.lang.String **getRequestURI**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getRequestURI()を返すことである。
public java.lang.StringBuffer **getRequestURL**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getRequestURL()を返すことである。
public java.lang.String **getServletPath**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getServletPath()を返すことである。

public HttpSession **getSession**(boolean create)	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getSession(boolean create)を返すことである。
public HttpSession **getSession**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getSession()を返すことである。
public boolean **isRequestedSessionIdValid**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で isRequestedSessionIdValid()を返すことである。
public boolean **isRequestedSessionIdFromCookie**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で isRequestedSessionIdFromCookie()を返すことである。
public boolean **isRequestedSessionIdFromURL**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で isRequestedSessionIdFromURL()を返すことである。
public boolean **IsRequestedSessionIdFromUrl**()	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で isRequestedSessionIdFromUrl()を返すことである。
public boolean **authenticate**(HttpServletResponse response) throws java.io.IOException, ServletException	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で authenticate を呼ぶことである。
public void **login**(java.lang.String username, java.lang.String password) throws ServletException	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で login を呼ぶことである。 Servlet 3.0 から導入。
public void **logout**() throws ServletException	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で login を呼ぶことである。 Servlet 3.0 から導入。
public java.util.Collection<Part> **getParts**() throws java.io.IOException, ServletException	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上で getParts を呼ぶことである。 Servlet 3.0 から導入。
public Part **getPart**(java.lang.String name) throws java.io.IOException, ServletException	このメソッドのデフォルトの振る舞いはラップされた要求オブジェクト上 getPart を呼ぶことである。 Servlet 3.0 から導入。

16.1.6.5 HttpServletResponseWrapper

public class **HttpServletResponseWrapper** extends ServletResponseWrapper implements HttpServletResponse サーブレットからの応答を変更したいと思う開発者たちによってサブクラス化出来る便利な HttpServletResponse インターフェイスの実装物を提供する。このクラスは Wrapper あるいは Decorator パタンを使用している。メソッドたちはデフォルトとしてラップされた応答オブジェクトのメソッドを呼び出す。 導入: Servlet 2.3	
コンストラクタ	
public **HttpServletResponseWrapper**(HttpServletResponse response)	与えられた応答をラップする応答アダプタを生成する 例外: java.lang.IllegalArgumentException - 指定された応答が null のとき
メソッド	
下記以外に javax.servlet.ServletResponseWrapper から引き継いでいるメソッドたちは: flushBuffer, getBufferSize, getCharacterEncoding, getContentType, getLocale, getOutputStream, getResponse, getWriter, isCommitted, isWrapperFor, isWrapperFor, reset, resetBuffer, setBufferSize, setCharacterEncoding, setContentLength, setContentType, setLocale, setResponse	
public void **addCookie**(Cookie cookie)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で addCookie(Cookie cookie)を呼ぶことである。
public boolean **containsHeader**(java.lang.String	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で containsHeader(String name)を呼ぶことである。

name)	
public java.lang.String **encodeURL**(java.lang.String url)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で encodeURL(String url)を呼ぶことである。
public java.lang.String **encodeRedirectURL**(java.lang.String url)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で encodeRedirectURL(String url) を呼ぶことである。
public java.lang.String **encodeUrl**(java.lang.String url)	廃止対象。第 2.1 版時点では代わりに encodeURL(String url)を使うこと。
public java.lang.String **encodeRedirectUrl**(java.lang.String url)	廃止対象。第 2.1 版時点では代わりに encodeRedirectURL(String url)を使うこと。
public void **sendError**(int sc, java.lang.String msg) throws java.io.IOException	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で sendError(int sc, String msg)を呼ぶことである。
public void **sendError(**int sc) throws java.io.IOException	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で sendError(int sc)を呼ぶことである。
public void **sendRedirect**(java.lang.String location) throws java.io.IOException	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で sendRedirect(String location)を返すことである。
public void **setDateHeader**(java.lang.String name, long date)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で setDateHeader(String name, long date)を呼ぶことである。
public void **addDateHeader**(java.lang.String name, long date)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で addDateHeader(String name, long date)を呼ぶことである。
public void **setHeader**(java.lang.String name, java.lang.String value)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で setHeader(String name, String value)を返すことである。
public void **addHeader**(java.lang.String name, java.lang.String value)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で addHeader(String name, String value)を返すことである。
public void **setIntHeader**(java.lang.String name, int value)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で setIntHeader(String name, int value)を呼ぶことである。
public void **addIntHeader**(java.lang.String name, int value)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で addIntHeader(String name, int value) を呼ぶことである。
public void **setStatus**(int sc)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で setStatus(int sc)を呼ぶことである。
public void **setStatus**(int sc, java.lang.String sm)	廃止対象。代わりにステータス・コードのセットには setStatus(int)を、記述付きのエラーを送信するには sendError(int, String)を使用のこと。
public int **getStatus**()	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で HttpServletResponse.getStatus()を呼ぶことである。
public java.lang.String **getHeader**(java.lang.String name)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で HttpServletResponse.getHeader(java.lang.String)を呼ぶことである。 導入: Servlet 3.0
public java.util.Collection<java.lang.String> **getHeaders**(java.lang.String name)	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で HttpServletResponse.getHeaders(java.lang.String)を呼ぶことである。返された Collection に対する何らかの変更はこの HttpServletResponseWrapper に影響を与えてはならない。 導入: Servlet 3.0
public java.util.Collection<java.lang.String> **getHeaderNames**()	このメソッドのデフォルトの振る舞いはラップされた応答オブジェクト上で HttpServletResponse.getHeaderNames()を呼ぶことである。返された Collection に対する何らかの変更はこの HttpServletResponseWrapper に影響を与えてはならない。 導入: Servlet 3.0

16.1.6.6 FilterRegistration

public interface **FilterRegistration** extends Registration フィルタを更に設定できるようにするインターフェイス 導入: Servlet 3.0	
ネストしたクラス	
static interface **FilterRegistration.Dynamic**	ServletContext 上の addFilter メソッドたちのひとつを介して登録された Filter を更に設定可能にするインターフェイス
メソッド	
void **addMappingForServletNames**(java.util.EnumSet<DispatcherType> dispatcherTypes, boolean isMatchAfter, java.lang.String servletNames)	与えられたサーブレット名と、この FilterRegistration で表現されている Filter の為のディスパッチャ・タイプを、フィルタ・マッピングに付加する。 フィルタ・マッピングはそれらが付加された順番で一致が調べられる。 isMatchAfter パラメタ次第で、与えられたマッピングはこの FilterRegistration が取得された ServletContext の宣言されたフィルタ・マッピングたちの前に調べられるかそれとも後に調べられるかが決まる。 このメソッドは複数回呼ばれ、各呼び出しでこれまでの効果に積み上げられる。 引数: dispatcherTypes - このフィルタ・マッピングの為のディスパッチャ・タイプ、あるいはデフォルトの DispatcherType.REQUEST が使われるときは null isMatchAfter - そこからこの FilterRegistration が取得された ServletContext の宣言されているフィルタ・マッピングたちの後に与えられたフィルタ・マッピングの一致が調べられるときは true、宣言されたフィルタ・マッピングたちの前に一致を調べるときは false servletNames - このフィルタ・マッピングのサーブレット名 例外: java.lang.IllegalArgumentException - servletNames が null または空のとき java.lang.IllegalStateException – この FilterRegistration が取得された ServletContext が既に初期化されてしまっている
java.util.Collection<java.lang.String> **getServletNameMappings**()	この FilterRegistration で表現されているこのフィルタの現在使えるサーブレット名マッピングを返す。 戻された Collection に対する何らかの変更はこの FilterRegistration に影響を与えてはならない。 戻り値: この FilterRegistration で表現されているこのフィルタの現在使えるサーブレット名マッピングたちの Collection (空の場合もあり)
void **addMappingForUrlPatterns**(java.util.EnumSet<DispatcherType> dispatcherTypes, boolean isMatchAfter, java.lang.String... urlPatterns)	この FilterRegistration で表現されているこのフィルタに対する URL パタンたちとディスパッチャ・タイプたちに、フィルタ・マッピングに付加する。 フィルタ・マッピングたちはそれらが付加された順番で一致検出がとられる。 isMatchAfter パラメタ次第で、与えられたマッピングはこの FilterRegistration が取得された ServletContext の宣言されたフィルタ・マッピングたちの前に調べられるかそれとも後に調べられるかが決まる。 このメソッドは複数回呼ばれ、各呼び出しでこれまでの効果に積み上げられる。 引数: dispatcherTypes - このフィルタ・マッピングの為のディスパッチャ・タイプ、あるいはデフォルトの DispatcherType.REQUEST が使われるときは null isMatchAfter - そこからこの FilterRegistration が取得された ServletContext の宣言されているフィルタ・マッピングたちの後に与えられたフィルタ・マッピングの一致が調べられるときは true、宣言されたフィルタ・マッピングたちの前に一致を調べるときは false urlPatterns - そのフィルタ・マッピングの為の URL パタン 例外: java.lang.IllegalArgumentException - servletNames が null または空のとき java.lang.IllegalStateException – この FilterRegistration が取得された ServletContext が既に初期化されてしまっている
java.util.Collection<java.lang.String> **getUrlPatternMappings**()	この FilterRegistration で表現される Filter に対する現在利用できる URL パタン・マッピングを返す。 戻された Collection に対する何らかの変更はこの FilterRegistration に影響を与えてはならない。 戻り値: この FilterRegistration で表現される Filter に対する現在利用できる URL パタン・マッピングの Collection (空の場合もあり)

16.1.6.7 FilterRegistration.Dynamic

public static interface **FilterRegistration.Dynamic** extends FilterRegistration, Registration.Dynamic ServletContext 上の addFilter メソッドたちのどれかひとつで登録されたフィルタを更に設定する為のインターフェイス。	
ネストしたクラスたち	
javax.servlet.FilterRegistration インターフェイスから継承したネストしたクラス/インターフェイス	FilterRegistration.Dynamic
メソッド	
javax.servlet.FilterRegistration インターフェイスから継承したメソッドたち	addMappingForServletNames, addMappingForUrlPatterns, getServletNameMappings, getUrlPatternMappings
javax.servlet.Registration.Dynamic インターフェイスから継承したメソッドたち	setAsyncSupported
javax.servlet.Registration インターフェイスから継承したメソッドたち	getClassName, getInitParameter, getInitParameters, getName, setInitParameter, setInitParameters

16.1.7 非同期処理関係

16.1.7.1 AsyncContext

public interface **AsyncContext** ある ServletRequest で開始された非同期動作の実行コンテキストを表現しているクラスである。ServletRequest.startAsync()あるいは ServletRequest.startAsync(ServletRequest, ServletResponse)呼び出しで AsyncContext は生成され初期化される。これらのメソッドを繰り返し読んだときは、しかるべく初期化された同じ AsyncContext インスタンスが戻される。 非同期動作中にタイムアウトが発生したときは、コンテナは以下のステップをとらねばならない: 1. これらの onTimeout メソッドで、非同期動作が開始させられた ServletContext で登録された総ての AsyncListener インスタンスたちを呼び出す 2. リスナたちのどれも complete()あるいは 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 に等しいステータス・コード付きでエラーのディスパッチを実施する 3. それに対応したエラー・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 complete()あるいは dispatch() メソッドたちのひとつを呼んでいないときは、complete()を呼ぶ 導入: Servlet 3.0	
フィールド	
static java.lang.String **ASYNC_CONTEXT_PATH**	これによってオリジナルのコンテキスト・パスが dispatch(String)あるいは dispatch(ServletContext,String)のターゲットが利用できる要求の属性の名前
static java.lang.String **ASYNC_PATH_INFO**	これによってオリジナルのパス情報が dispatch(String)あるいは dispatch(ServletContext,String)のターゲットが利用できる要求の属性の名前
static java.lang.String **ASYNC_QUERY_STRING**	これによってオリジナルのクエリ文字列が dispatch(String)あるいは dispatch(ServletContext,String)のターゲットが利用できる要求の属性の名前
static java.lang.String **ASYNC_REQUEST_URI**	これによってオリジナルの要求 URI が dispatch(String)あるいは dispatch(ServletContext,String)のターゲットが利用できる要求の属性の名前
static java.lang.String **ASYNC_SERVLET_PATH**	これによってオリジナルのサーブレット・パスが dispatch(String)あるいは dispatch(ServletContext,String)のターゲットが利用できる要求の属性の名前
メソッド	
ServletRequest **getRequest**()	ServletRequest.startAsync()あるいは ServletRequest.startAsync(ServletRequest, ServletResponse)を呼ぶことでこの AsyncContext を初期化するのに使われた要求オブジェクトを取得する 戻り値: この AsyncContext を初期化するのに使われた要求
ServletResponse **getResponse**()	ServletRequest.startAsync()あるいは ServletRequest.startAsync(ServletRequest, ServletResponse)を呼ぶことでこの AsyncContext を初期化するのに使われた応答オブジェクトを取得する 戻り値: この AsyncContext を初期化するのに使われた応答
boolean **hasOriginalRequestAndResponse**()	この AsyncContext がオリジナル、あるいはアプリケーションがラップした要求と応答のオブジェクトで初期化されているかどうかをチェックする。 この情報は、要求が非同期モードに置かれた後で、下り方向で呼ばれたフィルタたちが、上り方向でそれらが付加した要求及び/あるいは応答ラッパたちが非同期動作中に保持されるべきか、あるいは開放されるべきかを判断するのに使われよう。 戻り値: この AsyncContext が ServletRequest.startAsync()を呼ぶことでオリジナルの要求と応答のオブジェクトたちで初期化されている、あるいは ServletRequest.startAsync(ServletRequest, ServletResponse)を呼ぶことで初期化されていて、ServletRequest 引数も ServletResponse 引数もアプリケーションが用意したラッパを出ないときは true、そうでないときは false
void **dispatch**()	この AsyncContext の要求と応答のオブジェクトたちをサーブレット・コンテナにディスパッチする。もし ServletRequest.startAsync(ServletRequest, ServletResponse)でこの非同期サイクルがスタートされていて、渡された要求が HttpServletRequest のインスタンスである場合は、このディスパッチは HttpServletRequest.getRequestURI()で戻された URI へのものとなる。そうでないときは、このディスパッチは、そのコンテナが最後にディスパッチしたときの要求の URI 向けとなる。 以下のシーケンスが如何にこれが機能するかを示している: `// /url/A への REQUEST ディスパッチ`

```
AsyncContext ac = request.startAsync();
...
ac.dispatch(); // /url/A への ASYNC ディスパッチ

// /url/B への FORWARD ディスパッチ
getRequestDispatcher("/url/B").forward(request,response);
// FORWARD のターゲット内から非同期動作をスタート
// ディスパッチ
ac = request.startAsync();
...
ac.dispatch(); // /url/A への ASYNC ディスパッチ

// /url/B への FORWARD ディスパッチ
getRequestDispatcher("/url/B").forward(request,response);
// FORWARD のターゲット内から非同期動作をスタート
// ディスパッチ
ac = request.startAsync(request,response);
...
ac.dispatch(); // /url/B への ASYNC ディスパッチ
```

このメソッドは、それでディスパッチ操作が実行されることになるコンテナが管理するスレッドに要求と応答を渡した後で直ちに戻る。
この要求のディスパッチャ・タイプは DispatcherType.ASYNC にセットされる。フォワード・ディスパッチと違って、この応答バッファとヘッダはリセットされず、たとえその応答が既にコミットされていたとしてもディスパッチするのは有効である。
要求と応答に対するコントロールはディスパッチ・ターゲットに委譲され、応答は ServletRequest.startAsync() あるいは ServletRequest.startAsync(ServletRequest, ServletResponse)が呼ばれない限り、ディスパッチするターゲットが実行を終了したらクローズされる。
このメソッド実行中に生じたエラーあるいは例外は、以下のようにコンテナによって捕捉され処理されねばならない：

1. onError メソッドで、それに対しこの AsyncContext が生成された ServletRequest に登録されている総ての AsyncListener リスナたちを呼び出し、AsyncEvent.getThrowable()で Throwable が付けるようにする。
2. リスナたちのどれも complete()あるいは 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 に等しいステータス・コード付きでエラーのディスパッチを実施し、上記 Throwable を RequestDispatcher.ERROR_EXCEPTION 要求属性の値として利用できるようにする。
3. それに対応したエラー・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 complete()あるいは dispatch() メソッドたちのひとつを呼んでいないときは、complete()を呼ぶ

ServletRequest.startAsync()メソッドたちのひとつを呼び出すことで開始した非同期サイクルあたりたかだかひとつの非同期ディスパッチ操作が存在し得る。同じ非同期サイクル内で更なる非同期ディスパッチ操作を行おうとすれば IllegalStateException をもたらす。そのディスパッチ要求で続けて startAsync が呼ばれたときは、dispatch あるいは complete()メソッドのどれかが呼ばれる。
例外：
java.lang.IllegalStateException – dispatch のメソッドたちのひとつが呼ばれ、結果としてのディスパッチ中に startAsync メソッドが呼ばれていないとき、あるいは complete()が呼ばれたとき。
参考：
ServletRequest.getDispatcherType()

void **dispatch**(java.lang.String path)

この AsyncContext の要求と応答のオブジェクトたちを指定された path にディスパッチする。
この path パラメタはこの AsyncContext が初期化された ServletContext の適用域で、ServletRequest.getRequestDispatcher(String)と同じやり方で解釈される。
この要求の総てのパス関連クエリ・メソッドたちは、このディスパッチ・ターゲットを反映しなければならない一方で、オリジナルの要求 URI、コンテキスト・パス、サーブレット・パス、及びクエリ文字列は、この要求の ASYNC_REQUEST_URI, ASYNC_CONTEXT_PATH, ASYNC_PATH_INFO, ASYNC_SERVLET_PATH, 及び ASYNC_QUERY_STRING 属性から取り出せる。これらの属性たちはたとえ繰り返しのディスパッチのもとでも常にオリジナルのパス要素たちを反映する。
ServletRequest.startAsync()メソッドたちのひとつを呼び出すことで開始した非同期サイクルあたりたかだかひとつの非同期ディスパッチ操作が存在し得る。同じ非同期サイクル内で更なる非同期ディスパッチ操作を行おうとすれば IllegalStateException をもたらす。そのディスパッチ要求で続けて startAsync が呼ばれたときは、dispatch あるいは complete()メソッドのどれかが呼ばれる。
エラー処理を含め更なる詳細は dispatch()を見られたい。
引数：

	path – ディスパッチ・ターゲットのパスで、この AsyncContext が初期化された ServletContext の適用域を持つ。 例外: java.lang.IllegalStateException – dispatch のメソッドたちのひとつが呼ばれ、結果としてのディスパッチ中に startAsync メソッドが呼ばれていないとき、あるいは complete()が呼ばれたとき。 参考: ServletRequest.getDispatcherType()
void **dispatch**(ServletContext context, java.lang.String path)	この AsyncContext の要求と応答のオブジェクトたちを指定されたコンテキストの適用域の指定された path にディスパッチする。 この path パラメタは、指定されたコンテキストの適用域であることを除いては、この AsyncContext が初期化された ServletContext の適用域で、ServletRequest.getRequestDispatcher(String)と同じやり方で解釈される。 この要求の総てのパス関連クエリ・メソッドたちは、このディスパッチ・ターゲットを反映しなければならない一方で、オリジナルの要求 URI、コンテキスト・パス、サーブレット・パス、及びクエリ文字列は、この要求の ASYNC_REQUEST_URI, ASYNC_CONTEXT_PATH, ASYNC_PATH_INFO, ASYNC_SERVLET_PATH, 及び ASYNC_QUERY_STRING 属性から取り出せる。これらの属性たちはたとえ繰り返しのディスパッチのもとでも常にオリジナルのパス要素たちを反映する。 ServletRequest.startAsync()メソッドたちのひとつを呼び出すことで開始した非同期サイクルあたりたかだかひとつの非同期ディスパッチ操作が存在し得る。同じ非同期サイクル内で更なる非同期ディスパッチ操作を行おうとすれば IllegalStateException をもたらす。そのディスパッチ要求で続けて startAsync が呼ばれたときは、dispatch あるいは complete()メソッドのどれかが呼ばれる。 エラー処理を含め更なる詳細は dispatch()を見られたい。 引数: context - ディスパッチ・ターゲットの ServletContext path – 与えられた ServletContext での適用域をもったディスパッチ・ターゲットのパス 例外: java.lang.IllegalStateException – dispatch のメソッドたちのひとつが呼ばれ、結果としてのディスパッチ中に startAsync メソッドが呼ばれていないとき、あるいは complete()が呼ばれたとき。 参考: ServletRequest.getDispatcherType()
void **complete**()	この AsyncContext を初期化するのに使われたその要求上で開始させられた非同期動作を終了し、この AsyncContext を初期化するのに使われた応答を閉じる。 その為にこの AsyncContext が生成された ServletRequest で登録された AsyncListner 型のリスナたちはそれらの onComplete メソッドで呼び出される。 ServletRequest.startAsync()あるいは ServletRequest.startAsync(ServletRequest, ServletResponse)を呼んだあとで、このクラスの dispatch 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ぶことは合法である。もしこのメソッドが、startAsync を呼んだコンテナが開始させたディスパッチがそのコンテナに戻る前に呼ばれた場合は、この呼び出しはコンテナが開始させたディスパッチがコンテナに戻るまでは、効果を持たない(そして AsyncListener.onComplete(AsyncEvent)の呼び出しは遅れることになる)。
void **start**(java.lang.Runnable run)	指定された Runnable を走らせる為に、おそらくは管理されたスレッド・プールから、コンテナはひとつのスレッドをディスパッチする。コンテナはしかるべきコンテキストの情報を Runnable に伝搬させてよい。 引数: run - 非同期ハンドラ
void **addListener**(AsyncListener listener)	ServletRequest.startAsync()のメソッドたちのひとつを呼ぶことで開始させられたもっとも最近の非同期サイクルに指定された AsyncListner を登録する。 指定された AsyncListner はこの非同期サイクルが成功裏に終了した、タイムアウトになった、あるいはエラーとなったときに AsyncEvent を受ける。 AsyncListener インスタンスたちはそれが付加された順番で通知を受ける。 引数: listener - 登録する AsyncListener 例外: java.lang.IllegalStateException - もしこのメソッドが、ServletRequest.startAsync()メソッドたちのひとつがが呼ばれた間に、コンテナが開始させたディスパッチがコンテナに戻った後で呼ばれたとき。
void **addListener**(AsyncListener listener, ServletRequest servletRequest, ServletResponse servletResponse)	ServletRequest.startAsync()のメソッドたちのひとつを呼ぶことで開始させられたもっとも最近の非同期サイクルに指定された AsyncListner を登録する。 指定された AsyncListner はこの非同期サイクルが成功裏に終了した、タイムアウトになった、あるいはエラーとなったときに AsyncEvent を受ける。 AsyncListener インスタンスたちはそれが付加された順番で通知を受ける。 指定された ServletRequest 及び ServletResponse オブジェクトは、それに提供する AsyncEvent の getSuppliedRequest と getSuppliedResponse メソッドを介して、指定した AsyncListner が取得できるようになる。これらのオブジェクトはこの AsyncEvent が提供された時点で要求の読み出しあるいは応答の書き込みをしてはならない。何故なら指定された AsyncListner が登録されたあとで更なるラッピングが生じているかもしれないからであるが、これらに関わるリソースを解放するのに使えよう。

	引数: listener - 登録する AsyncListener servletRequest - AsyncEvent に含められることになる ServletRequest servletResponse - AsyncEvent に含められることになる ServletResponse 例外: java.lang.IllegalStateException - もしこのメソッドが、ServletRequest.startAsync()メソッドたちのひとつがが呼ばれた間に、コンテナが開始させたディスパッチがコンテナに戻った後で呼ばれたとき。
<T extends AsyncListener> T **createListener(**java.lang.Class<T> clazz) throws ServletException	指定した AsyncListener クラスをインスタンス化する。 戻された AsyncListener インスタンスは、addListener メソッドたちのひとつを呼ぶことでこの AsyncContext に登録する前に、更にカスタム化できる。 指定した AsyncListener クラスには引数なしのコンストラクタが定義されていなければならない。このコンストラクタはインスタンス化の為に使われる。 指定された clazz が Managed Bean である場合は、リソース注入に対応していなければならない。Managed Bean とリソース注入に関する更なる詳細は Java EE プラットホームと JSR 299 の仕様書を参照のこと。 このメソッドは AsyncListener に適用できるアプリケーションたちをサポートしている。 引数: clazz – インスタンス化する AsyncListener クラス 戻り値: 新しい AsyncListener インスタンス 例外: ServletException - 指定した clazz のインスタンス化が出来なかったとき
void **setTimeout**(long timeout)	この AsyncContext に為のタイムアウトをミリ秒でセットする。 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間この AsyncContext に適用される。 タイムアウトは complete()メソッドもディスパッチ・メソッドたちのどれも呼ばれなかったときに発生する。ゼロあるいはそれ以下のタイムアウト値はタイムアウトが発生しないことを意味する。 setTimeout(long)が呼ばれないときは、コンテナが持っているデフォルトのタイムアウトが適用され、それは getTimeout()メソッドで知ることが出来る。 引数: timeout – ミリ行でのタイムアウト時間 例外: java.lang.IllegalStateException - もしこのメソッドが、ServletRequest.startAsync()メソッドたちのひとつがが呼ばれた間に、コンテナが開始させたディスパッチがコンテナに戻った後で呼ばれたとき。
long **getTimeout**()	この AsyncContext の為のタイムアウトをミリ秒で取得する。 このメソッドは非同期動作の為のコンテナのデフォルトのタイムアウト、あるいはもっとも最近の setTimeout(long)呼び出しで渡されたタイムアウト値を返す。 ゼロあるいはそれ以下のタイムアウト値はタイムアウトが発生しないことを意味する。 戻り値: ミリ秒でのタイムアウト時間

16.1.7.2 AsyncEvent

public class **AsyncEvent** extends java.lang.Object ServletRequest 上で(ServletRequest.startAsync()あるいは ServletRequest.startAsync(ServletRequest, ServletResponse)呼び出しを介して)開始した非同期動作が、終了した、タイムアウトになった、あるいはエラーを起こした際に起動するイベントである。 導入: Servlet 3.0	
コンストラクタ	
public **AsyncEvent**(AsyncContext context)	指定した AsyncContext から AsyncEvent を組み立てる 引数: context - この AsyncEvent で提供される AsyncContex
public **AsyncEvent**(AsyncContext context, ServletRequest request, ServletResponse response)	指定した AsyncContext、ServletRequest、及び ServletResponse から AsyncEvent を組み立てる 引数: context - この AsyncEvent で提供される AsyncContex request - この AsyncEvent で提供される ServletRequest response - この AsyncEvent で提供される ServletResponse
public	指定した AsyncContext と Throwable から AsyncEvent を組み立てる

AsyncEvent(AsyncContext context, java.lang.Throwable throwable)	引数: context - この AsyncEvent で提供される AsyncContex throwable - この AsyncEvent で提供される Throwable
public **AsyncEvent**(AsyncContext context, ServletRequest request, ServletResponse response, java.lang.Throwable throwable)	指定した AsyncContext、ServletRequest、ServletResponse、及び Throwable から AsyncEvent を組み立てる 引数: context – この AsyncEvent で提供される AsyncContex request - この AsyncEvent で提供される ServletRequest response - この AsyncEvent で提供される ServletResponse throwable - この AsyncEvent で提供される Throwable
メソッド	
public AsyncContext **getAsyncContext**()	この AsyncEvent から AsyncContext を取得する。 戻り値: この AsyncEvent を初期化するのに使われた AsyncContext
public ServletRequest **getSuppliedRequest**()	この AsyncEvent から ServletRequest を取得する。 この AsyncEvent をもたらした AsyncListener が AsyncContext.addListener(AsyncListener, ServletRequest, ServletResponse)を使って付加されていたものだったら、戻された ServletRequest は上記メソッドで指定したものと同じになる。この AsyncListener が AsyncContext.addListener(AsyncListener)で付加されたものなら、このメソッドは null を返さねばならない。 戻り値: この AsyncEvent を初期化するのに使われた ServletRequest、あるいはこの AsyncEvent が ServletRequest なしで初期化されているときは null
public ServletResponse **getSuppliedResponse()**	この AsyncEvent から ServletResponse を取得する。 この AsyncEvent をもたらした AsyncListener が AsyncContext.addListener(AsyncListener, ServletRequest, ServletResponse)を使って付加されていたものだったら、戻された ServletResponse は上記メソッドで指定したものと同じになる。この AsyncListener が AsyncContext.addListener(AsyncListener)で付加されたものなら、このメソッドは null を返さねばならない。 戻り値: この AsyncEvent を初期化するのに使われた ServletResponse、あるいはこの AsyncEvent が ServletResponse なしで初期化されているときは null
public java.lang.Throwable **getThrowable**()	この AsyncEvent から Throwable を取得する。 戻り値: この AsyncEvent を初期化するのに使われた Throwable、またはこの AsyncEvent が Throwable なしで初期化されているときは null

16.1.7.3 AsyncListner

public interface **AsyncListener** extends java.util.EventListener それに対しこのリスナが付加された ServletRequest 上で開始した非同期動作が完了した、タイムアウトを起こした、あるいはエラーを起こしたというイベントで通知を受けるリスナである。 導入: Servlet 3.0	
メソッド	
void **onComplete**(AsyncEvent event) throws java.io.IOException	この AsyncListner に非同期動作が終了したことを通知する。 完了した非同期処理に対応した AsyncContext は、指定したイベントで getAsyncContext を呼ぶことで取得できる。 加えて、もしこの AsyncListner が AsyncContext.addListener(AsyncListener, ServletRequest, ServletResponse)呼び出しで登録されていたときは、指定した ServletRequest 及び ServletResponse オブジェクトたちは指定したイベント上で getSuppliedRequest 及び getSuppliedResponse 呼び出しで取得できる。 引数: event – 非同期動作が完了したことを示す AsyncEvent 例外: java.io.IOException - 指定した AsyncEvent の処理中に I/O 関連のエラーが発生したとき
void **onTimeout**(AsyncEvent event) throws java.io.IOException	この AsyncListner に非同期動作がタイムアウトを起したことを通知する。 タイムアウトした非同期処理に対応した AsyncContext は、指定したイベントで getAsyncContext を呼ぶことで取得できる。

	加えて、もしこの AsyncListner が AsyncContext.addListener(AsyncListener, ServletRequest, ServletResponse)呼び出しで登録されていたときは、指定した ServletRequest 及び ServletResponse オブジェクトたちは指定したイベント上で getSuppliedRequest 及び getSuppliedResponse 呼び出しで取得できる。 引数: event – 非同期動作がタイムアウトしたことを示す AsyncEvent 例外: java.io.IOException - 指定した AsyncEvent の処理中に I/O 関連のエラーが発生したとき
void **onError**(AsyncEvent event) throws java.io.IOException	この AsyncListner に非同期動作が完了できなかったことを通知する。 完了に失敗した非同期処理に対応した AsyncContext は、指定したイベントで getAsyncContext を呼ぶことで取得できる。 加えて、もしこの AsyncListner が AsyncContext.addListener(AsyncListener, ServletRequest, ServletResponse)呼び出しで登録されていたときは、指定した ServletRequest 及び ServletResponse オブジェクトたちは指定したイベント上で getSuppliedRequest 及び getSuppliedResponse 呼び出しで取得できる。 引数: event – 非同期動作が完了に失敗したことを示す AsyncEvent 例外: java.io.IOException - 指定した AsyncEvent の処理中に I/O 関連のエラーが発生したとき
void **onStartAsync**(AsyncEvent event) throws java.io.IOException	この AsyncListner に、ServletRequest.startAsync()メソッドたちのひとつを呼び出すことで新しい非同期サイクルが初期化されていることを通知する。 この再初期化されている同期処理に対応した AsyncContext は、指定したイベントで getAsyncContext を呼ぶことで取得できる。 加えて、もしこの AsyncListner が AsyncContext.addListener(AsyncListener, ServletRequest, ServletResponse)呼び出しで登録されていたときは、指定した ServletRequest 及び ServletResponse オブジェクトたちは指定したイベント上で getSuppliedRequest 及び getSuppliedResponse 呼び出しで取得できる。 引数: event – 新しい非同期サイクルが初期化中であることを示す AsyncEvent 例外: java.io.IOException - 指定した AsyncEvent の処理中に I/O 関連のエラーが発生したとき

16.1.7.4 ServletRequest のなかの非同期処理関連メソッド

ServletRequest	
メソッド	
AsyncContext **startAsync()** throws java.lang.IllegalStateException	この要求を非同期モードとし、その AsyncContext をオリジナルの(ラップされていない) ServletRequest と ServletResponse のオブジェクトで初期化する。 このメソッドを呼ぶことで、返された AsyncContext 上で AsyncContext.complete()が呼ばれる、あるいはその非同期操作がタイムアウトを起こすまで、それに関わる応答のコミットが遅らせられる。 返された AsyncContext 上で AsyncContext.hasOriginalRequestAndResponse()を呼ぶと true が返される。この要求が非同期モードに置かれた後での下り方向で呼び出されたフィルタたちは、それらが登り方向で追加した要求及び/あるいは応答のラッパが、その非同期操作中は留まる必要がなく、したがってそれに関わるリソースがリリースしてよいことを示すものとしてこれを使うことができる。 このメソッドは、その onStartAsync で各 AsyncListner を呼び出した後で startAsync メソッドたちの以前の呼び出すことで返された AsyncContext に登録された AsyncListener インスタンスたち(もし存在していれば)のリストをクリアする。 その後のこのメソッドあるいはそのオーバロードされたメソッドを呼び出すと、同じ AsyncContext が返され、しかるべく初期化される。 Servlet 3.0 から導入。
AsyncContext **startAsync**(ServletRequest servletRequest, ServletResponse servletResponse) throws java.lang.IllegalStateException	この要求を非同期モードとし、与えられた要求と応答オブジェクトでその AsyncContext を初期化する。 引数である ServletRequest 及び ServlerResponse は、このメソッドが呼ばれている有効範囲内において、Servlet の service メソッドに渡された、あるいは Filter のdoFilterメソッドに渡されたと同じインスタンスたち、あるいはそれらをラップした ServletRequestWrapper 及び ServletResponseWrapper のインスタンスでなければならない。 このメソッドを呼ぶことで、返された AsyncContext 上で AsyncContext.complete()が呼ばれる、あるいはその非同期操作がタイムアウトを起こすまで、それに関わる応答のコミットが遅らせられる。 返された AsyncContext 上で AsyncContext.hasOriginalRequestAndResponse()を呼ぶと、渡された ServletRequest と ServletResponse 引数がオリジナルのものであってアプリケーションが用意したラッパを持っていないときに限り、false が返される。この要求が非同期モードに置かれた後での下り方向で呼び出された

	フィルタたちは、それらが登り方向で追加した要求及び/あるいは応答のラッパたちの幾つかは、その非同期操作中は留まるってはいる必要があり、したがってそれに関わるリソースがリリースしてはならないことを示すものとしてこれを使うことができる。AsyncContext を初期化するのに使われ、AsyncContext.getRequest()呼び出しで返されることになる与えられた servletRequest が、ServletRequestWrapper を含まない限り.、あるフィルタの登り方向の呼び出し中に適用された ServletRequestWrapper は、そのフィルタの下り方向呼び出しでリリースされて良い。同じことは ServletResponseWrapper でもいえる。 このメソッドは、その onStartAsync で各 AsyncListner を呼び出した後で startAsync メソッドたちの以前の呼び出すことで返された AsyncContext に登録された AsyncListener インスタンスたち(もし存在していれば)のリストをクリアする。 その後のこのメソッドあるいはそのオーバロードされたメソッドを呼び出すと、同じ AsyncContext が返され、しかるべく初期化される。もしこのメソッドの呼び出しの後でゼロ引数のメソッドを呼ぶと、指定された(ラップされていることもあり)要求と応答のオブジェクトたちは、返された AsyncContext 上に閉じ込められる。 Servlet 3.0 から導入。
boolean **isAsyncStarted**()	この要求が非同期モードになっているかどうかをチェックする。 ServletRequest はその startAsync()あるいは startAsync(ServletRequest,ServletResponse)を呼ぶことで非同期モードに置かれる。 このメソッドは、この要求が非同期モードになっているが、AsyncContext.dispatch()メソッドたちのどれかを使ってディスパッチされている、あるいは AsyncContext.complete()メソッド呼び出しで非同期モードから解放されているときは、false を返す。 Servlet 3.0 から導入。
boolean **isAsyncSupported**()	この要求が非同期操作に対応しているかどうかをチェックする。もしこの要求が非同期処理に対応可能とアノテートされていないか配備記述子内でフラグが立てられていないフィルタあるいはサーブレットの有効範囲内にある場合は、非同期動作は不能になっている。 Servlet 3.0 から導入。
AsyncContext **getAsyncContext**()	この要求上で最も最近に startAsync()あるいは startAsync(ServletRequest,ServletResponse)呼び出しで生成または初期化された AsyncContext を取得する。 Servlet 3.0 から導入。
DispatcherType **getDispatcherType**()	この要求のディスパッチャのタイプを取得する。 ある要求のディスパッチャのタイプは、その要求に提供されるべきフィルタをコンテナが選択するのに使われる:一致するディスパッチャ・タイプと URL パタンを持ったフィルタたちのみが適用される。 複数のディスパッチャ・タイプに対応するよう設定されているフィルタが、ある要求に対するディスパッチャを問い合わせできるようにすることで、そのフィルタはそのディスパッチャ・タイプに応じてその要求を処理できるようになる。 ある要求の初期ディスパッチャ・タイプは DispatcherType.REQUEST として定義されている。 RequestDispatcher.forward(ServletRequest, ServletResponse)あるいは 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を介して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 として与えられている。最後に、その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・タイプは DispatcherType.ERROR として与えられている。 Servlet 3.0 から導入。

16.1.8 ファイル・アップロード関係

16.1.8.1 HttpServletRequest の中のファイル・アップロード関係メソッド

java.util.Collection<Part> **getParts**() throws java.io.IOException, ServletException	この要求が multipart/form-data タイプである場合に、この要求の総ての Part 要素を取得する。この要求が multipart/form-data タイプであるが Part 型の要素を持っていない場合は、返された Collection は空になる。返された Collection に対する何らかの変更はこの HttpServletRequest に影響を与えない。 戻り値: この要求の Part 要素の Collection（空もあり） 例外: java.io.IOException – この要求の Part 要素の検索で I/O エラーが発生したとき。 ServletException – この要求が multipart/form-data のタイプでないとき java.lang.IllegalStateException - この要求ボディが maxRequestSize よりも大きい、あるいはこの要求の Part のどれかが maxFileSize よりも大きいとき。 Servlet 3.0 から導入。
Part **getPart**(java.lang.String name) throws java.io.IOException, ServletException	与えられた名前の Part を取得する。 引数: name – 要求する Part の名前 戻り値: 与えられた名前の Part、あるいはこの要求が multipart/form-data タイプであるが要求された Part を有していないときは null。 例外: java.io.IOException – この要求の Part 要素の検索で I/O エラーが発生したとき。 ServletException – この要求が multipart/form-data のタイプでないとき java.lang.IllegalStateException - この要求ボディが maxRequestSize よりも大きい、あるいはこの要求の Part のどれかが maxFileSize よりも大きいとき。 Servlet 3.0 から導入。

16.1.8.2 Part

public interface javax.servlet.http.**Part** このクラスは multipart/form-data の POST 要求で受信したパート(part)あるいは Form アイテムを表現している。 Servlet 3.0 から導入。	
java.io.InputStream **getInputStream**() throws java.io.IOException	このパートのコンテントを InputStream として取得する。
java.lang.String **getContentType**()	このパートのコンテント・タイプを取得する。
java.lang.String **getName**()	このパートの名前を取得する。
long **getSize**()	このファイルのサイズを返す。
void **write**(java.lang.String fileName) throws java.io.IOException	このアップロードされたアイテムをディスクに書き込むための利便性のためのメソッド。 このメソッドは同じパートで 1 度以上呼ばれたときはその成功は保証されない。これにより特定の実装、例えばもとになっているデータの総てをコピーするのではなくファイルのリネームが可能になり、大きな性能上の利点が得られる。
void **delete**() throws java.io.IOException	ファイル・アイテムのための蓄積を、それに関わる一時的ディスク・ファイルを含めて削除する。
java.lang.String **getHeader**(java.lang.String name)	指定された MIME ヘッダの値を String として返す。もしその Part が指定された名前のヘッダを含んでいないときは、このメソッドは null を返す。ヘッダ名は大文字と小文字の区別はされない。このメソッドはどの要求ヘッダにも使える。

java.util.Collection<java.lang.String> **getHeaders**(java.lang.String name)	与えられた名前の Part ヘッダの値たちを返す。 返された Collection に対する何らかの変更はこの Part に影響を与えない。 Part ヘッダ名は大文字と小文字の区別はされない。
java.util.Collection<java.lang.String> **getHeaderNames**()	この Part のヘッダ名たちを返す。 一部のサーブレット・コンテナではサーブレットがこのメソッドを使ってヘッダたちにアクセスするのを許していないが、その場合は、このメソッドは null を返す。 返された Collection に対する何らかの変更はこの Part に影響を与えない。

16.1.8.3 MultiPartConfig

public @interface **MultipartConfig** Servlet クラスで指定するアノテーションで、その Servlet のインスタンスが multipart/form-data の MIME タイプに適合した要求を受け付けることを示す。MultipartConfig でアノテートされたサーブレットは getPart あるいは getParts を呼ぶことで与えられた multipart/form-data 要求の Part 要素を取り出すことが出来る。 Servlet 3.0 から導入。	
public abstract java.lang.String **location**	ファイルをスタするディレクトリの場所 デフォルト: ""
public abstract long **maxFileSize**	アップロードされたファイルに許される最大サイズ。 デフォルトら-1L で、これは制限なしを意味する。 デフォルト: -1L
public abstract long **maxRequestSize**	multipart/form-data 要求に許される最大サイズ。 デフォルトら-1L で、これは制限なしを意味する。 デフォルト: -1L
public abstract int **fileSizeThreshold**	それを超えたらディスクに書き込むサイズ閾値 デフォルト: 0

16.1.9 セキュリティ関係

16.1.9.1 HttpServletRequest の中のセキュリティ関係メソッド

HttpServletRequest	
フィールド	static java.lang.String BASIC_AUTH：Basic 認証の為の String 識別子 static java.lang.String CLIENT_CERT_AUTH：クライアント認定のための String 識別子 static java.lang.String DIGEST_AUTH：Digest 認証の為の String 識別子 static java.lang.String FORM_AUTH：Form 認証の為の String 識別子
java.lang.String **getAuthType**()	そのサーブレットを保護する為に使われている認証スキームの名前を返す。総てのサーブレット・コンテナはベーシック、フォーム、及びクライアント証明証による認証に対応しており、さらには追加的にダイジェスト認証に対応してよい。そのサーバが認証に対応していない場合は、null が返される。 このメソッドは CGI 変数の AUTH_TYPE と同じである。
java.lang.String **getRemoteUser**()	この要求をしているユーザが認証されていればその名前を返す、あるいは認証されていないときは null を返す。そのユーザ名がその後の各要求で送信されるかどうかは、ブラウザと認証のタイプに依存する。これは CGI 変数の REMOTE_USER の値と同じである。
boolean **isUserInRole**(java.lang.String role)	その認証されたユーザが指定された論理的な「ロール」に含まれているかどうかを示すブール値を返す。ロールとロールのメンバーシップは配備記述子を使って定義できる。そのユーザが認証されていないときは、このメソッドは false を返す。
java.security.Principal **getUserPrincipal**()	現在認証されているそのユーザの名前を含む java.security.Principal オブジェクトを返す。そのユーザが認証されていないときは、このメソッドは null を返す。
boolean **authenticate**(HttpServletResponse response) throws java.io.IOException, ServletException	この要求をしているユーザを認証する為にこの ServletContext に設定されているコンテナのログイン・メカニズムを使う。このメソッドは HttpServletResponse 引数を修正及びコミットしても良い。 Servlet 3.0 から導入。
void **login**(java.lang.String username, java.lang.String password) throws ServletException	この ServletContext のために設定されているウェブ・コンテナのログイン・メカニズムが使っているパスワード検証レルムの中で与えられた username と password を検証する。 この ServletContext のために設定されているログイン・メカニズムが username と password の認証に対応している場合、及び login 呼び出しの時点でこの要求の発信者の ID がまだ確立されていないとき（言い換えると getUserPrincipal、getRemoteUser、及び getAuthType の総てが null を返すとき）、及び与えられたクレデンシャル（信用状）の検証が成功したときは、このメソッドは ServletException を返すことなく戻る。 このメソッドが例外をスローしないで戻るときは、getUserPrincipal、getRemoteUser、及び getAuthType によって返される値としては確立された非 null の値を持っていなければならない。 引数： username – そのユーザのログイン識別子に対応した String 値 password – 識別されたユーザに対応するパスワード String 例外： ServletException - 設定されているログイン・メカニズムがユーザ名認証に対応していないとき、あるいは非 null の発信者 ID が既に確立されている（login 呼び出し前に）、あるいは与えられた username と password で検証できなかったとき。 Servlet 3.0 から導入。
void **logout**() throws ServletException	その要求に対し getUserPrincipal、getRemoteUser、及び getAuthType が呼ばれたときに戻り値としての null を確立する。 例外： ServletException - ログアウトに失敗したとき Servlet 3.0 から導入

16.1.9.2 org.apache.commons.codec.binary.Base64

英文オリジナル：http://commons.apache.org/codec/apidocs/index.html

public class **Base64** extends Object implements BinaryEncoder, BinaryDecoder

RFC 2045 で規定された Base64 のエンコーディングとデコーディングを提供するクラス。このクラスは RFC 2045 Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies by Freed and Borenstein の、第 6.8.節の Base64 Content-Transfer-Encoding を実装したものである。このクラスは以下のように各種のコンストラクタでパラメタ化出来る:

- URL セーフ・モード:デフォルトではオフ
- 行長:デフォルトでは 76 である。4 の倍数で無い行長は最終的にはエンコードされたデータでは 4 の倍数となる。
- 行区切り:デフォルトでは CRLF ("\r\n")

このクラスは文字ストリーム上ではなくて直接バイト・ストリーム上で動作しするので、下位の 127 の ASCII 文字と互換性ある(ISO-8859-1, Windows-1252, UTF-8, etc)文字エンコーディングのみエンコードとデコードをするように、ハード・コードされている。

導入:
1.0
バージョン:
$Id: Base64.java 801706 2009-08-06 16:27:06Z niallp $
作者:
Apache Software Foundation
参考:
RFC 2045

コンストラクタ	
public **Base64**()	デコーディング(全モード)と URL 非安全モードでエンコーディングを行う為の Base64 コーデックを生成する。エンコードするときは、行長が 76、行区切りは CRLF、そしてそのエンコード表は STANDARD_ENCODE_TABLE となる。 デコーディングに対しては総ての変種に対応する。
public **Base64**(boolean urlSafe)	デコーディング(全モード)と指定された URL 安全モードでエンコーディングを行う為の Base64 コーデックを生成する。エンコードするときは、行長が 76、行区切りは CRLF、そしてそのエンコード表は STANDARD_ENCODE_TABLE となる。 デコーディングに対しては総ての変種に対応する。 引数: urlSafe – true の場合は URL 安全エンコーディングが使用される。殆どの場合はこれは false にセットすべきである。 導入: 1.4
public **Base64**(int lineLength)	デコーディング(全モード)と URL 非安全モードでエンコーディングを行う為の Base64 コーデックを生成する。指定された行長のエンコードをするときは、行区切りは CRLF、そしてそのエンコード表は STANDARD_ENCODE_TABLE となる。 4 の倍数で無い行長のものにたいしては、エンコードされたときは実質的に 4 の倍数の長さになる。 デコーディングに対しては総ての変種に対応する。 引数: lineLength - エンコードされたデータの各行は指定された長さ(4 の倍数に丸められる)までとなる。もし lineLength <= 0 のときは、出力は行に切り分け(チャンク)されない。デコードに際しては無視される。 導入: 1.4
public **Base64**(int lineLength, byte[] lineSeparator)	デコーディング(全モード)と URL 非安全モードでエンコードする為に使う Base64 コーデックを生成する。エンコーディングに際しては行長と行区切りがコンストラクタに与えられ、エンコーディング表は STANDARD_ENCODE_TABLE である。 4 の倍数で無い行長のものにたいしては、エンコードされたときは実質的に 4 の倍数の長さになる。 デコーディングに対しては総ての変種に対応する。 引数: lineLength - エンコードされたデータの各行は指定された長さ(4 の倍数に丸められる)までとなる。もし lineLength <= 0 のときは、出力は行に切り分け(チャンク)されない。デコードに際しては無視される。 lineSeparator – エンコードされたデータの各行はこのバイトたちのシーケンスで終了する。 例外: IllegalArgumentException – 行区切りに Base64 の文字が含まれているときにスローされる。 導入: 1.4
public **Base64**(int lineLength, byte[] lineSeparator, boolean urlSafe)	デコーディング(全モード)と URL 非安全モードでエンコードする為に使う Base64 コーデックを生成する。エンコーディングに際しては行長と行区切りがコンストラクタに与えられ、エンコーディング表は STANDARD_ENCODE_TABLE である。 4 の倍数で無い行長のものにたいしては、エンコードされたときは実質的に 4 の倍数の長さになる。 デコーディングに対しては総ての変種に対応する。 引数: lineLength - エンコードされたデータの各行は指定された長さ(4 の倍数に丸められる)までとなる。もし

	lineLength <= 0 のときは、出力は行に切り分け(チャンク)されない。デコードに際しては無視される。 lineSeparator – エンコードされたデータの各行はこのバイトたちのシーケンスで終了する。 urlSafe – true の場合は+と/文字の代わりに-と__を出力する。urlsafe はエンコーディングにのみ適用される。デコーディングでは双方のモードともにシームレスに処理される。 例外: IllegalArgumentException – 行区切りに Base64 の文字が含まれているときにスローされる。この場合は機能しなくなる! 導入: 1.4
メソッド	
public boolean **isUrlSafe**()	現在のエンコードのモードを返す。URL-SAFE の場合は true、それ以外の場合は false を返す 戻り値: URL-SAFE の場合は true、それ以外の場合は false 導入: 1.4
public static boolean **isBase64**(byte octet)	そのオクテットが Base64 アルファベットに入っているかどうかを返す。 引数: octet – テストするオクテット値 戻り値: そのオクテットが Base64 アルファベットに入っているときは true、それ以外のときは false 導入: 1.4
public static boolean **isArrayByteBase64(**byte[] arrayOctet)	与えられたバイト配列が Base64 アルファベットで有効な文字のみで構成されているかどうかを調べる。現在このメソッドはホワイトスペースは有効として取り扱っている。 引数: arrayOctet – 検査するバイト配列 戻り値: 総てのバイトが Base64 アルファベット文字である場合、あるいはこのバイト配列が空の場合は true、それ以外のときは false
public static byte[] **encodeBase64**(byte[] binaryData)	Base64 アルゴリズムでバイナリ・データをエンコードするが、出力はチャンクしない。 binaryData – エンコードするバイナリ・データ 戻り値: UTF-8 表現をした Base64 文字を含むバイト配列
public static String **encodeBase64String**(byte[] binaryData)	バイナリ・データを Base64 アルゴリズムを使って CRLF で区切られた 76 文字ブロックたちにエンコードする。 引数: binaryData – エンコードするバイナリ・データ 戻り値: Base64 文字を含む String 導入: 1.4
public static byte[] **encodeBase64URLSafe**(byte[] binaryData)	Base64 アルゴリズムの URL 安全バージョンでバイナリ・データをエンコードするが、出力はチャンクしない。URL 安全バージョンは+と/文字の代わりに-と__を出力する。 引数: binaryData – エンコードするバイナリ・データ 戻り値: UTF-8 表現をした Base64 文字を含むバイト配列 導入: 1.4
public static String **encodeBase64URLSafeString** (byte[] binaryData)	Base64 アルゴリズムの URL 安全バージョンを使ってバイナリ・データをエンコードするが、出力をチャンクしない。URL 安全バージョンでは+と/文字の代わりに-と__を出力する。 引数: binaryData – エンコードするバイナリ・データ 戻り値: Base64 文字からなる String 導入: 1.4
public static byte[] **encodeBase64Chunked**(byte[] binaryData)	Base64 アルゴリズムを使ってバイナリ・データをエンコードし、出力を 76 文字ブロックたちとする。 引数: binaryData – エンコードするバイナリ・データ 戻り値: 76 文字ブロックにチャンクされた Base65 文字列

public Object **decode**(Object pObject) throws DecoderException	Object を Base64 アルゴリズムを使ってデコードする。このメソッドは Decoder インターフェイスの要求を満たす為に用意されており、もし指定されたオブジェクトが byte[]あるいは String の型でないときは DecoderException をスローする。 規定: インターフェイス Decoder の decode 引数: pObject – デコードするオブジェクト 戻り値: 与えられた byte[]あるいは String に対応したバイナリ・データを含むオブジェクト(byte[]型の) 例外: DecoderException - 指定した引数が byte[]型で無いとき
public byte[] **decode**(String pArray)	Base64 アルファベットの文字を含んだ String をデコードする 引数: pArray – Base64 文字データを含む String 戻り値: バイナリ・データを含むバイト配列 導入: 1.4
public byte[] **decode(**byte[] pArray)	Base64 アルファベットの文字を含んだ byte[]をデコードする 規定: BinaryDecoder インターフェイスの decode 引数: pArray – Base64 文字データを含むバイト配列 戻り値: バイナリ・データを含むバイト配列
public static byte[] **encodeBase64**(byte[] binaryData, boolean isChunked)	Encodes binary data using the base64 algorithm, optionally chunking the output into 76 character blocks. Parameters: binaryData - Array containing binary data to encode. isChunked - if true this encoder will chunk the base64 output into 76 character blocks Returns: Base64-encoded data. Throws: IllegalArgumentException - Thrown when the input array needs an output array bigger than Integer.MAX_VALUE
public static byte[] **encodeBase64**(byte[] binaryData, boolean isChunked, boolean urlSafe)	バイナリ・データを Base64 アルゴリズムでエンコードして、オプション的に出力を 76 文字ブロックたちにチャンクする。 引数: binaryData - エンコードするバイナリ・データを含んだバイト配列 isChunked - true のときはこのエンコーダは Base64 出力を 76 文字のブロックたちにチャンクする urlSafe – true のときはこのエンコーダは+と/文字の代わりに-と__を出力する。 戻り値: Base64 エンコードされたデータ 例外: IllegalArgumentException – Integer.MAX_VALUE よりも大きな出力配列が必要になるような入力配列のとき 導入: 1.4
public static byte[] **encodeBase64**(byte[] binaryData, boolean isChunked, boolean urlSafe, int maxResultSize)	バイナリ・データを Base64 アルゴリズムでエンコードして、オプション的に出力を 76 文字ブロックたちにチャンクする。 引数: binaryData - エンコードするバイナリ・データを含んだバイト配列 isChunked - true のときはこのエンコーダは Base64 出力を 76 文字のブロックたちにチャンクする urlSafe – true のときはこのエンコーダは+と/文字の代わりに-と__を出力する maxResultSize – 受け付け可能な最大サイズ結果 戻り値: Base64 エンコードされたデータ 例外: IllegalArgumentException – maxResultSize よりも大きな出力配列が必要になるような入力配列のとき 導入: 1.4
public static byte[] **decodeBase64**(String base64String)	Base64 の String をオクテットたちにデコードする 引数: base64String – Base64 データを含む String

	戻り値: デコードされたデータを含む配列 導入: 1.4
public static byte[] **decodeBase64**(byte[] base64Data)	Base64 のデータをオクテットたちにデコードする 引数: base64String – Base64 データを含むバイト配列 戻り値: デコードされたデータを含む配列
public Object **encode**(Object pObject) throws EncoderException	オブジェクトを Base64 アルゴリズムでエンコードする。このメソッドは Encoder インターフェイスの要求を満たす為に用意されており、指定したオブジェクトが byte[]型でないときには EncoderException をスローする。 規定: インターフェイス Encoder の encode 引数: pObject – エンコードするオブジェクト 戻り値: 指定された byte[]に対応した Base64 でエンコードされたデータを含むオブジェクト(byte[]型) 例外: EncoderException - 指定した引数が byte[]型でないとき
public String **encodeToString**(byte[] pArray)	バイナリ・データを含む byte[]を、Base64 アルファベットの文字を含む String にエンコードする。 引数: pArray – バイナリ・データを含むバイト配列 戻り値: Base64 文字データのみを含む String 導入: 1.4
public byte[] **encode**(byte[] pArray)	バイナリ・データを含む byte[]を、Base64 アルファベットの文字を含む byte[]にエンコードする。 規定: インターフェイス BinaryEncoder の encode 引数 pArray – バイナリ・データを含むバイト配列 戻り値: Base64 文字データのみを含むバイト配列
public static BigInteger **decodeInteger**(byte[] pArray)	W3C の XML-Signature のような暗号標準に従って byte64 エンコードされた整数をデコードする。 引数: pArray – Base64 文字データを含むバイト配列 戻り値: BigInteger 導入: 1.4
public static byte[] **encodeInteger**(BigInteger bigInt)	W3C の XML-Signature のような暗号標準に従って byte64 エンコードされた整数をエンコードする。 引数: bigInt – BigInteger 戻り値: Base64 文字データを含むバイト配列 例外: NullPointerException – null が渡されたとき 導入: 1.4

16.1.9.3 org.apache.catalina.authenticator.DigestAuthenticator

これは org.apache.catalina.authenticator.AuthenticatorBase のダイジェスト認証の為の継承物である。これは配備記述子の<security-constraint>要素を行使する Valve(要求の処理部品)インターフェイスの実装物でもある。

このクラスの使用には以下の制約がある:

- このクラスを使うときには、これを付加させる Context は、ユーザたちを認証をし、彼らが割り当てられて

いるロールたちを列挙する為に使える、結び付けられた Realm を持っていなければならない
- この Valve は HTTP 要求にのみ使える

org.apache.catalina.authenticator.DigestAuthenticator inplements Authenticator, Contained, Lifecycle, Valve HTTP DIGEST 認証(see RFC 2069)の為の認証と Valve の実装物 出典: http://tomcat.apache.org/tomcat-5.5-doc/catalina/docs/api/org/apache/catalina/authenticator/DigestAuthenticator.html	
フィールド	
protected static final int **USE_ONCE**	no once トークンが一回だけ使われることを示す 参考: Constant Field Values
protected static final int **USE_NEVER_EXPIRES**	no once トークンが一回だけ使われることを示す 参考: Constant Field Values
protected static final int **TIMEOUT_INFINITE**	no once トークンが一回だけ使われることを示す 参考: Constant Field Values
protected static final MD5Encoder **md5Encoder**	このクラスの為の MD5 ヘルパ・オブジェクト
protected static final java.lang.String **info**	この実装物の記述的情報 参考: Constant Field Values
protected static java.security.MessageDigest **md5Helper**	MD5 メッセージ・ダイジェストのプロバイダ
protected java.util.Hashtable **nOnceTokens**	No once のハッシュテーブル
protected long **nOnceTimeout**	No once の有効期間(ミリ秒)。この値が小さければセキュリティ・レベルが改善されることになる(このトークンがより頻繁に発生されるので)が、その分サーバのオーバヘッドが増える。
protected int **nOnceUses**	指定した数を超えた場合に no once が無効となる。この数が小さければその分オーバヘッドが増える(このトークンをより頻繁に発生させねばならないから)が、安全性は高まる。
protected java.lang.String **key**	プライベート・キー
メソッド	
public java.lang.String **getInfo**()	この Valve 実装物に関する記述的情報を返す。 規定: Valve インターフェイスの getInfo オーバライド: AuthenticatorBase クラスの getInfo
public boolean **authenticate**(HttpRequest request, HttpResponse response, LoginConfig config) throws java.io.IOException	指定された login 設定に基づいて、この要求をしているユーザを認証する。指定された制約を満たされてときは true を、既に応答チャレンジを作ってしまっているときは false を返す。 規定: AuthenticatorBase の authenticate 引数: request - 処理中の要求オブジェクト response - 作成中の応答オブジェクト config – どのように認証を行うべきかを記述したログイン設定 例外: java.io.IOException - 入出力のエラーが発生したとき
protected static java.security.Principal **findPrincipal**(javax.servlet.http.HttpServletRequest request, java.lang.String authorization, Realm realm)	指定した認証クレデンシャルを構文解析し、指定されたレルムからこれらのクレデンシャルを認証する Principal を返す。そのような Principal が存在しないときは null を返す。 引数: request - HTTP サーブレット要求 authorization - この要求からの認証クレデンシャル realm - Principal たちを認証するのに使われる Realm
protected java.lang.String **parseUsername**(java.lang.String	指定された認証文字列から username を構文解析して取り出す。特定できないときは null を返す 引数:

authorization)	authorization - 構文解析する認証文字列
protected static java.lang.String **removeQuotes**(java.lang.String quotedString, boolean quotesRequired)	文字列上の引用符を外す。RFC2617 ではレルムを除く総てのパラメタで引用符はオプショナルだと書かれている。
protected static java.lang.String **removeQuotes**(java.lang.String quotedString)	文字列上の引用符を外す。
protected java.lang.String **generateNOnce**(javax.servlet.http.HttpServletRequest request)	一意的トークンを生成する。このトークンは次のようなパタンに従って生成される: `NOnceToken = Base64 ( MD5 ( client-IP ":" time-stamp ":" private-key ) )` 引数: request - HTTP サーブレット要求

protected void **setAuthenticateHeader**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response, LoginConfig config, java.lang.String nOnce)	WWW-Authenticate ヘッダを生成する。このヘッダは以下のテンプレートに従わねばならない:
	`WWW-Authenticate = "WWW-Authenticate" ":" "Digest" digest-challenge`
	
	`digest-challenge = 1#( realm | [ domain ] | nOnce | [ digest-opaque ] |[ stale ] | [ algorithm ] )`
	
	`realm = "realm" "=" realm-value`
	`realm-value = quoted-string`
	`domain = "domain" "=" <"> 1#URI <">`
	`nonce = "nonce" "=" nonce-value`
	`nonce-value = quoted-string`
	`opaque = "opaque" "=" quoted-string`
	`stale = "stale" "=" ( "true" | "false" )`
	`algorithm = "algorithm" "=" ( "MD5" | token )`
	引数:
	request - HTTP サーブレット要求
	response - HTTP サーブレット応答
	config - 以下に認証を行うかを記述したログイン設定
	nOnce - nonce トークン

16.1.9.4 org.apache.catalina.realm.RealmBase

このクラスは Tomcat が用意している 6 つのレルム (CombinedRealm、DataSourceRealm、JAASRealm、JDBCRealm、JNDIRealm、MemoryRealm、及び UserDatabaseRealm) のベースとなっているものである。あとで説明するように、ダイジェスト・パスワードを動的に計算する場合には org.apache.catalina.realm.RealmBase クラスの static メソッドである Digest()に平文のパスワードとダイジェスト・アルゴリズム名を引数として渡して呼び出せば良い。このメソッドはダイジェスト化されたパスワードを返す。このクラスの Javadoc はここにあるので参考にされたい。なお java.security.MessageDigest も利用可能である。

org.apache.catalina.realm.RealmBase inplements javax.management.MBeanRegistration, Lifecycle, Realm 有効なユーザ、パスワード、及びロールを設定する為に XML ファイルを読みだす Realm インターフェイスのシンプルな実装。ファイル・フォーマット(及びデフォルトの場所)は現在 Tomcat 3.X に対応しているものと同じである。	
フィールド	
protected Container **container**	
protected Log **containerLog**	
protected java.lang.String **digest**	非平文フォーマットでパスワードをストアするのに使われるダイジェスト・アルゴリズム。有効な値は MessageDigest クラスでそのアルゴリズム名が受け入れられるもの、またはダイジェスト化をしない場合は null
protected java.lang.String **digestEncoding**	そのダイジェストの為のエンコーディング文字セット
protected static final java.lang.String **info**	この Realm 実装に関する記述的情報
protected LifecycleSupport	この部品の為の lifecycle イベント対応

lifecycle	
protected java.security.MessageDigest **md**	ユーザ・クレデンシャル（パスワード）のダイジェスト化の為の MessageDigest オブジェクト
protected static final MD5Encoder **md5Encoder**	このクラスの MD5 ヘルパ・オブジェクト
protected static java.security.MessageDigest **md5Helper**	
protected static StringManager **sm**	このパッケージの文字列マネージャ
protected boolean **started**	この部品が開始したかどうか？
protected java.beans.PropertyChangeSupport **support**	この部品の属性変更対応
protected boolean **validate**	クライアント承認チェインが提示されたときにそれらの妥当性を検査すべきか？
protected RealmBase.AllRolesMode **allRolesMode**	全ロール・モード
protected java.lang.String **type**	
protected java.lang.String **domain**	
protected java.lang.String **host**	
protected java.lang.String **path**	
protected java.lang.String **realmPath**	
protected javax.management.ObjectName **oname**	
protected javax.management.ObjectName **controller**	
protected javax.management.MBeanServer **mserver**	
protected boolean **initialized**	
メソッド	
public Container **getContainer**()	この Realm が関連付けられている Container を返す。 規定: Realm インターフェイスの getContainer
public void **setContainer(**Container container)	この Realm が関連付けられている Container をセットする。 規定: Realm インターフェイスの setContainer 引数: container - 関連付けられた Container
public java.lang.String **getAllRolesMode**()	全ロール・モードを返す
public void **setAllRolesMode**(java.lang.String allRolesMode)	全ロール・モードをセットする
public java.lang.String **getDigest**()	クレデンシャルのストアに使われるダイジェスト・アルゴリズムを返す
public void **setDigest**(java.lang.String digest)	クレデンシャルのストアに使われるダイジェスト・アルゴリズムをセットする 引数: digest - 新しいダイジェストのアルゴリズム
public java.lang.String **getDigestEncoding**()	ダイジェスト・エンコーディングの文字セットを返す 戻り値: プラットホームのデフォルトの為の文字セット（null の場合もあり）

public void **setDigestEncoding**(java.lang.String charset)	ダイジェスト・エンコーディングの為の文字セットをセットする 引数: charset - 文字セット(プラットホーム・デフォルトの場合は null)
public java.lang.String **getInfo**()	この Realm 実装に関する記述的情報と対応するバージョン番号を<description>/<version>の書式で返す。 規定: Realm インターフェイスの getInfo
public boolean **getValidate**()	"validate certificate chains"フラグを返す
public void **setValidate**(boolean validate)	"validate certificate chains"フラグをセットする 引数: validate - 新しい validate certificate chains フラグ
public void **addPropertyChangeListener**(java.beans.PropertyChangeListener listener)	この部品に属性変更リスナを付加する 規定: Realm インターフェイスの addPropertyChangeListener 引数: listener - 付加するリスナ
public java.security.Principal **authenticate**(java.lang.String username, java.lang.String credentials)	指定したユーザ名とクレデンシャルに結び付けられている Principal がひとつ存在すればそれを返す、それ以外の場合は null を返す。 規定: Realm インターフェイスの authenticate 引数: username - 検索する Principal のユーザ名 credentials – このユーザ名を認証する為のパスワードあるいは他のクレデンシャル 戻り値: 指定されたユーザ名とクレデンシャルに結び付けられている Principal がひとつ存在すればそれを返す、それ以外の場合は null
public java.security.Principal **authenticate(**java.lang.String username, byte[] credentials)	指定したユーザ名とクレデンシャルに結び付けられている Principal がひとつ存在すればそれを返す、それ以外の場合は null を返す。 規定: Realm インターフェイスの authenticate 引数: username - 検索する Principal のユーザ名 credentials – このユーザ名を認証する為のパスワードあるいは他のクレデンシャル
public java.security.Principal **authenticate**(java.lang.String username, java.lang.String clientDigest, java.lang.String nOnce, java.lang.String nc, java.lang.String cnonce, java.lang.String qop, java.lang.String realm, java.lang.String md5a2)	RFC 2069 で規定された手法を使って与えられたパラメタたちを使って計算したダイジェストと一致する、指定したユーザ名に関連付けられた Principal を返す、それ以外の場合は null を返す。 規定: Realm インターフェイスの authenticate 引数: username – 検索する Principal のユーザ名 clientDigest – クライアントが送信したダイジェスト nOnce - その要求の為に使われているユニークなトークン realm - レルム名 md5a2 - そのダイジェストを計算するのに使われる 2 番目の MD5 ダイジェスト: MD5(Method + ":" + uri)
public java.security.Principal **authenticate**(java.security.cert.X509Certificate[] certs)	X509 クライアント認定の指定されたチェインに関連付けられた Principal を返す。無い場合は null を返す。 規定: Realm インターフェイスの authenticate 引数: certs - クライアント認定の配列で、この配列の最初はそのクライアントそのものの認定である。
public void **backgroundProcess**()	再ロードのような周期的なタスクを実行する。このメソッドはこのコンテナの classloading コンテキストの中で呼び出されることになる。予期されない例外は捕捉され、ログされる。 規定: Realm インターフェイスの backgroundProcess
public SecurityConstraint[] **findSecurityConstraints**(Request request, Context context)	この要求の為の要求 URI を保護する為に設定された SecurityConstraints を返す、あるいはそのような制約がない場合は null を返す。 規定: Realm インターフェイスの findSecurityConstraints 引数: request - 処理している要求 context - その要求がマップされているコンテキスト

public boolean **hasResourcePermission**(Request request, Response response, SecurityConstraint[] constraints, Context context) throws java.io.IOException	指定した認可制約に基づいてアクセス制御をおこなう。この制約が満足されており処理を継続すべきときは true を返し、それ以外のときは false を返す。 規定: Realm インターフェイスの hasResourcePermission 引数: request - 処理している要求 response - 生成中の応答 constraints - 施行中のセキュリティ制約 context – このクラスのクライアントが付加されているコンテキスト 例外: java.io.IOException - 入出力エラーが発生したとき
public boolean **hasRole**(java.security.Principal principal, java.lang.String role)	指定された Principal が、このレルムのコンテキスト内で、指定されたセキュリティ・ロールを持っているときは true を返し、それ以外は false を返す。このメソッドは Realm 実装物によってオーバライドされ得るが、このデフォルトは、この Realm から認証された Principals を代表するのに GenericPrincipal が使われている場合は、適正なものである。 規定:Realm インターフェイスの hasRole 引数: principal - そのロールをチェックする Principal role - チェックされるべきセキュリティ・ロール
public boolean **hasUserDataPermission**(Request request, Response response, SecurityConstraint[] constraints) throws java.io.IOException	要求されたこの URI を保護するセキュリティ制約で要求されているユーザ・データの制約を行使する。この制約が違反しておらず、プロセスを進めるべきときは true を返し、既に応答を作ってしまっているときは false を返す。 規定: Realm インターフェイスの hasUserDataPermission 引数: request - 処理している要求 response - 生成中の応答 constraints - 施行中のセキュリティ制約 context – このクラスのクライアントが付加されているコンテキスト 例外: java.io.IOException - 入出力エラーが発生したとき
public void **removePropertyChangeListener**(java.beans.PropertyChangeListener listener)	この部品から属性変更リスナを除去する 規定: Realm インターフェイスの removePropertyChangeListener 引数: listener - 除去するリスナ
public void **addLifecycleListener**(LifecycleListener listener)	この部品にライフサイクル・イベントのリスナを付加する。 規定: Lifecycle インターフェイスの addLifecycleListener 引数: listener - 付加するリスナ
public LifecycleListener[] **findLifecycleListeners**()	この lifecycle に結び付けられたライフサイクル・リスナを取得する。この Lifecycle にリスナが登録されていない場合は、長さがゼロの配列が返される。 規定: Lifecycle インターフェイスの findLifecycleListeners
public void **removeLifecycleListener**(LifecycleListener listener)	この部品からライフサイクル・イベントのリスナを除去する。 規定: Lifecycle インターフェイスの removeLifecycleListener 引数: listener - 除去するリスナ
public void **start**() throws LifecycleException	この部品のパブリックなメソッドのアクティブな使用を開始に備える。このメソッドはこの部品のパブリックなメソッドたちのどれかを使用する前に呼ばれねばならない。これはまた登録されているリスナたちに START_EVENT 型の LifecycleEvent を通知しなければならない。 規定: Lifecycle インターフェイスの start 例外: LifecycleException - この部品の仕様を阻害する致命的エラーをこの部品が検出したとき
public void **stop()** throws LifecycleException	この部品のパブリックなメソッドのアクティブな使用をきちんと終了させる。このメソッドはこの部品の与えられたインスタンス上で最後に呼ばれるものでなければならない。これはまたこれはまた登録されているリスナたちに STOP_EVENT 型の LifecycleEvent を通知しなければならない。

	規定: Lifecycle インターフェイスの stop 例外: この部品が報告すべき致命的なエラーを検出したとき
public void **destroy()**	
protected java.lang.String **digest**(java.lang.String credentials)	指定されたアルゴリズムを使ってそのパスワードをダイジェスト化し、その結果を対応した 16 進数の文字列に変換する。例外が起きたときは、平文の文字列が返される。 引数: credentials - そのユーザ(username)名を認証するのに使われるパスワードあるいは他のクレデンシャル。
protected boolean **hasMessageDigest**()	
protected java.lang.String **getDigest**(java.lang.String username, java.lang.String realmName)	指定されたユーザ名に結び付けられたダイジェストを返す。
protected abstract java.lang.String **getName**()	ログ・メッセージで使う為に、この Realm 実装物のみじかな名前を返す。
protected abstract java.lang.String **getPassword**(java.lang.String username)	指定したプリンシパルのユーザ名に結び付けられたパスワードを返す。
protected java.security.Principal **getPrincipal**(java.security.cert.X509Certificate usercert)	指定された認定に結び付けられたプリンシパルを返す。
protected abstract java.security.Principal **getPrincipal**(java.lang.String username)	指定されたユーザ名に結び付けられたプリンシパルを返す。
public static final java.lang.String **Digest**(java.lang.String credentials, java.lang.String algorithm, java.lang.String encoding)	指定されたアルゴリズムを使ってパスワードをダイジェスト化し、その結果を対応する 16 進数の文字列に変換する。例外が起きたときは、平文のクレデンシャルが返される。 引数: credentials - そのユーザ(username)名を認証するのに使われるパスワードあるいは他のクレデンシャル algorithm - このダイジェストに使われたアルゴリズム encoding - ダイジェスト化する為の文字列の文字エンコーディング
public static void **main**(java.lang.String[] args)	指定されたアルゴリズムを使ってパスワードをダイジェスト化し、その結果を対応する 16 進数の文字列に変換する。例外が起きたときは、平文のクレデンシャルが返される。
public javax.management.ObjectName **getController**()	
public void **setController**(javax.management.ObjectName controller)	
public javax.management.ObjectName **getObjectName()**	
public java.lang.String **getDomain**()	
public java.lang.String **getType**()	
public java.lang.String **getRealmPath**()	
public void **setRealmPath**(java.lang.String theRealmPath)	
public javax.management.ObjectName **preRegister**(javax.management.MBeanServer server, javax.management.ObjectName name) throws java.lang.Exception	規定: javax.management.MBeanRegistration インターフェイスの preRegister 例外: java.lang.Exception
public void	規定:

postRegister(java.lang.Boolean registrationDone)	javax.management.MBeanRegistration インターフェイスの postRegister
public void **preDeregister**() throws java.lang.Exception	規定: javax.management.MBeanRegistration インターフェイスの preDeregister 例外: java.lang.Exception
public void **postDeregister**()	規定: javax.management.MBeanRegistration インターフェイスの postDeregister
public void **init**()	
protected java.lang.String **getRealmSuffix**()	

16.2節　MINA プロキシのダウンロードと実行

プロキシとはクライアントと別のサーバ間の間に介在する代理サーバであり、おもにセキュリティや負荷配分などに利用されている。Apache MINA は簡単なトンネル・プロキシのサンプルを用意している。これは通過するメッセージの総てをログするだけのものである。しかしこのプロキシは、サーブレットの開発にあたって、TCP レベルでどのようなメッセージがブラウザとサーブレット間で交わされているのかを知るには非常に便利なツールである。TCP レベルでの交わされるデータ確認には Tera Term のような Telnet のソフトウエアも使われるが、この場合は送信データを手入力することになり、長いデータを送信するのは大変である。ブラウザを使ったほうが入力ミスが無くなるし、実際に自分のブラウザがどのような要求を出しているかも把握できる。

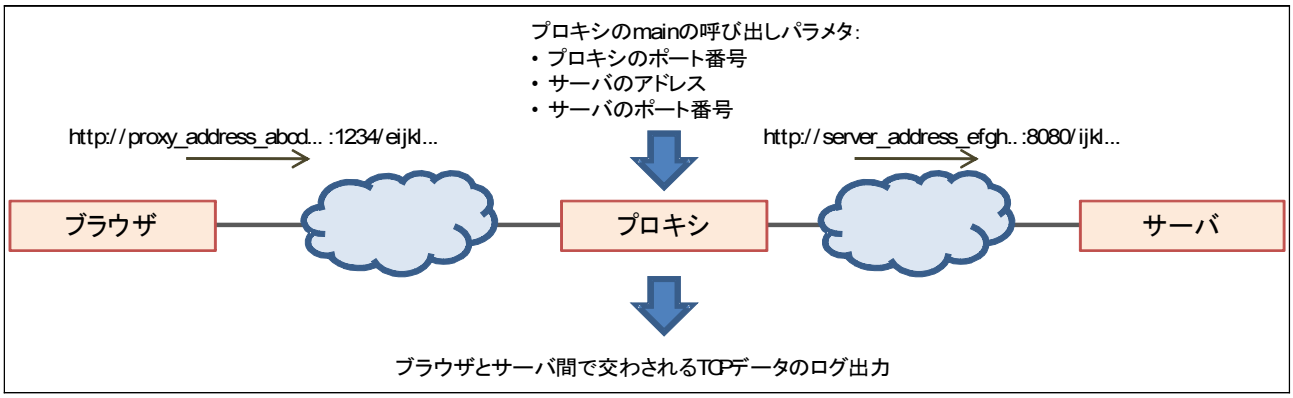


図 AB-1 : プロキシの機能

このプロキシに関しては当社の「Java による TCP プログラミング」というチュートリアルで既に説明済みであるが、サーブレットの学習者たちのために、ダウンロードと Eclipse 上での使い方をわかりやすく説明する。

Apache MINA (「アパッチ・ミーナ」と発音する。Multipurpose Infrastructure for Network Applications : ネットワーク・アプリケーションのための多目的インフラストラクチャの略称) は、オープン・ソースのネットワークアプリケーションのフレームワークである。ネットワークとプロトコル I/O 層の抽象化により、低レベルの NIO よりは高品質なネットワーク・アプリケーションをずっと容易に開発出来るようになっている。

16.2.1　MINA_Proxy のプロジェクトとパッケージの作成

最初に Eclipse 上でこのプロキシのためのプロジェクトとパッケージを用意する。

1. Eclipse のプロジェクト・エクスプローラ上で右クリック→「新規(New)」→「プロジェクト(Project)」→新規プロジェクトのウィザード上で java\java プロジェクトを選択し、「次へ(Next)」をクリック→下図のようにプロジェクト名を入力し、JRE としては JDK を選択し、「完了(Finish)」をクリックする。

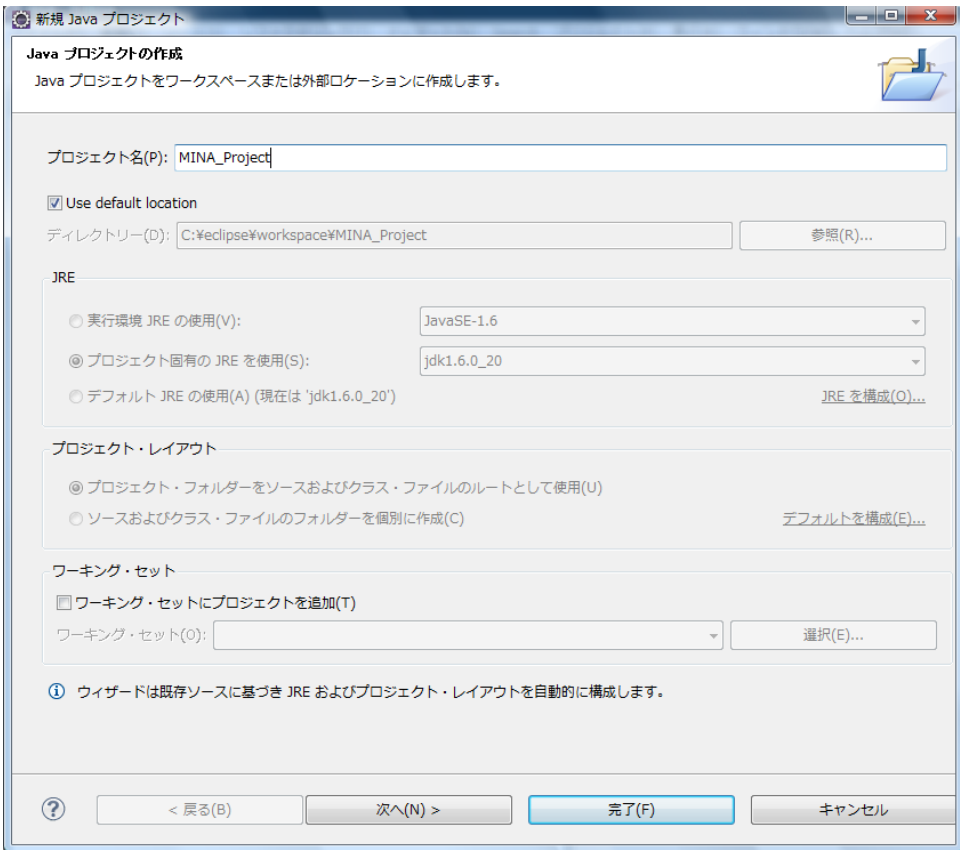


図 AB-2：新規プロジェクトの作成

2. パッケージ・エクスプローラ上で新しく出来た MINA_Project を選択して右クリックし、→「新規(New)」→「パッケージ(Package)」→新規パッケージのウィザード上でパッケージ名"MINA_proxy"を入力して「完了(Finish)」をクリックする。

そうすると下図のようにパッケージ・エクスプローラ上に新しいパッケージが作られる：

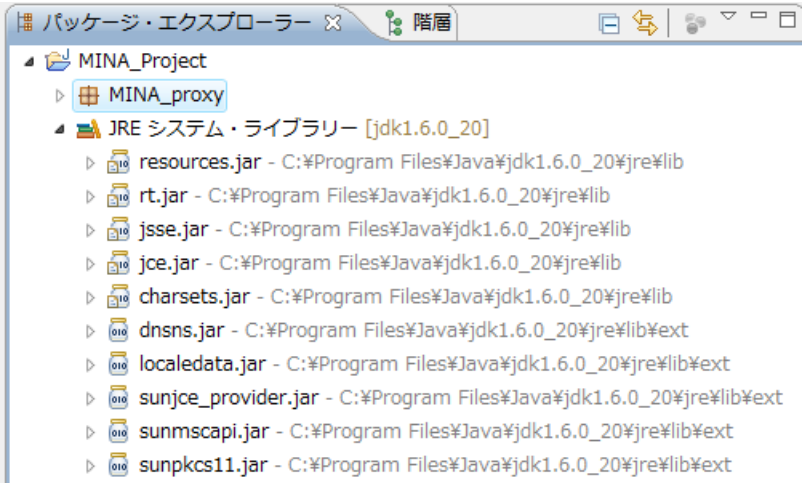


図 AB-3：パッケージ・エクスプローラ上の新しいプロジェクトとパッケージ

16.2.2 MINA ライブラリのダウンロードと登録

次に Apache MINA のライブラリをダウンロードし、これを Eclipse に登録する。

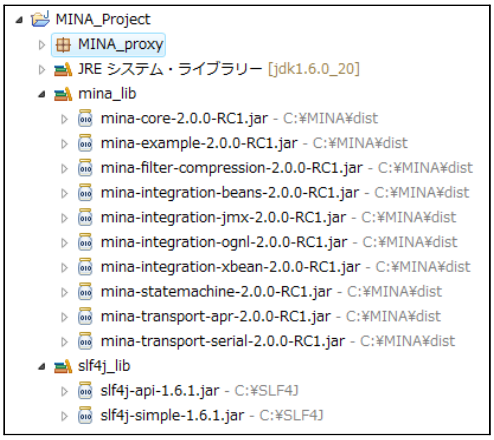
1. まずダウンロードのサイトから適当な圧縮形式(例えばZIP)のファイルを適当なディレクトリにダウンロードする。ここではまだ安定化していないが最新の「Apache MINA 2.0.0-RC1 unstable (Java 5+)」のzipアーカイブ・ファイルをダウンロードする。
2. 次に展開したもののメインのフォルダ(例えばMINA 2.0.0-RC1)をC:MINAとCディレクトリに移す。
3. Eclipseを起動して、「ウィンドウW)」→「設定(P)」→「java」→「ビルド・パス」→「ユーザー・ライブラリ」→「新規(N)」で新しいライブラリ名を入力(例えばmina_libなど)
4. 次にこのライブラリにダウンロードしたjarファイルたち、即ち**c:mina.dist**の中の総てのjarを選択してインポートする
5. 次にパッケージ・エクスプローラ上で自分のパッケージを右クリック→「ビルドパス(B)」→「ライブラリの追加(L)」→「ユーザ・ライブラリ」で自分が名付けたMINAのライブラリをチェックする

16.2.3 SLF4Jのパッケージのダウンロード

Apache MINAでも「ロギング・ファサードのインストール」の項で説明したSLF4Jというロギング・ファザードが使われている。もういちどその方法を示すと:

1. SLF4Jのダウンロードのサイトから適当な圧縮形式(例えばZIPのslf4j-1.6.1.zip)のファイルを適当なディレクトリにダウンロードする。
2. 次に展開したもののメインのフォルダ(例えばSLF4J-1.6.1)をC:SLF4JとCディレクトリに移す。
3. Eclipseを起動して、「ウィンドウ(W)」→「設定(P)」→「java」→「ビルド・パス(B)」→「ユーザー・ライブラリ(U)」→「新規」で新しいライブラリ名を入力(例えばslf4j_libなど)
4. 次にこのライブラリにダウンロードしたjarファイルたち、即ちc:slf4jの中の以下のjarを選択してインポートする:
   1. slf4j-api-1.6.1
   2. slf4j-simple-1.6.1(ここではSystem出力、即ちコンソールを利用する)
5. 次にパッケージ・エクスプローラ上で自分のパッケージを右クリック→「ビルドパス(B)」→「ライブラリの追加(L)」→「ユーザ・ライブラリ(U)」で自分が名付けたSLF4Jのライブラリをチェックする

ここまでの操作により、このプロジェクトには次のようなライブラリが用意される:



図AB-4:追加されたユーザ・ライブラリ

16.2.4 Proxyのプログラムのインポート

1. はじめに適当なフォルダ(例えばダウンロード)に当社のサイトにアップロードされているZIPファイルをダウンロードする。これはMINA_Proxy.zipというアーカイブ・ファイルである。MINAのサイトにあるパッケージでも構わないが、こちらのほうがコメントが日本語化されている。

2. プロジェクト・エクスプローラ上で MINA_proxy という自分が作ったパッケージを右クリックし、「インポート(I)」→インポートのウィザード上で「一般\アーカイブ・ファイル」を選択して「次へ(N)」をクリック。
3. ソース・アーカイブ・ファイルは「参照」で自分がダウンロードした MINA_Proxy.zip を選択する。そうすると下図のように、このファイルに含まれている 4 つの java ファイルがクリックされた状態で表示されるので、「完了(F)」をクリックする。

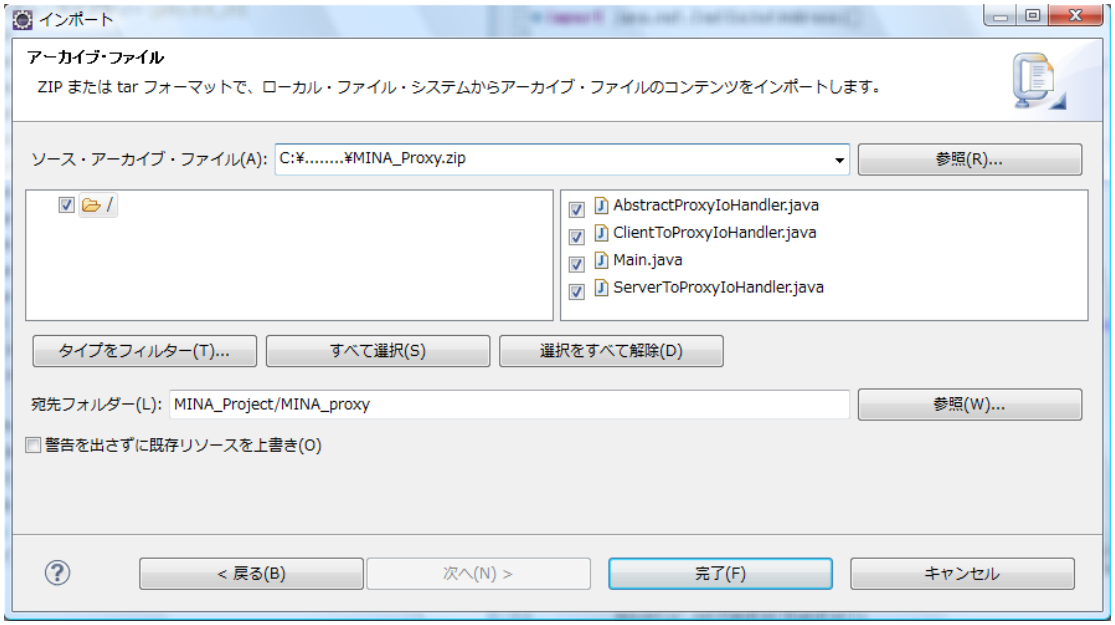


図 AB-5：ソース・ファイルのインポート

16.2.5 プロキシの開始

このプログラムを Eclipse 上で走らせるには、例えば引数を 12345 localhost 8080、としてこの Main クラスを実行させる。引数はプロキシのポート、サーバのホスト名、サーバのポートの順である。

1. プロキシを実行させてサーブレットをテストするには、Eclipse のパースペクティブはデフォルトの Java EE に切り替えておく。「ウィンドウ(P)」→「パースペクティブを開く」→「その他」→Java Ee で「OK」をクリックする。
2. プロジェクト・エクスプローラ上で、MINA_Project\MINA-proxy\Main.java を選択し、右クリック→「実行(R)」→「実行の構成(N)」で、そのウィザード上で下図のように引数をセットする。最初にこれをセットしたら、次回からは「実行(R)」→「Java アプリケーション」だけでその引数は再度入力しなくても良い。

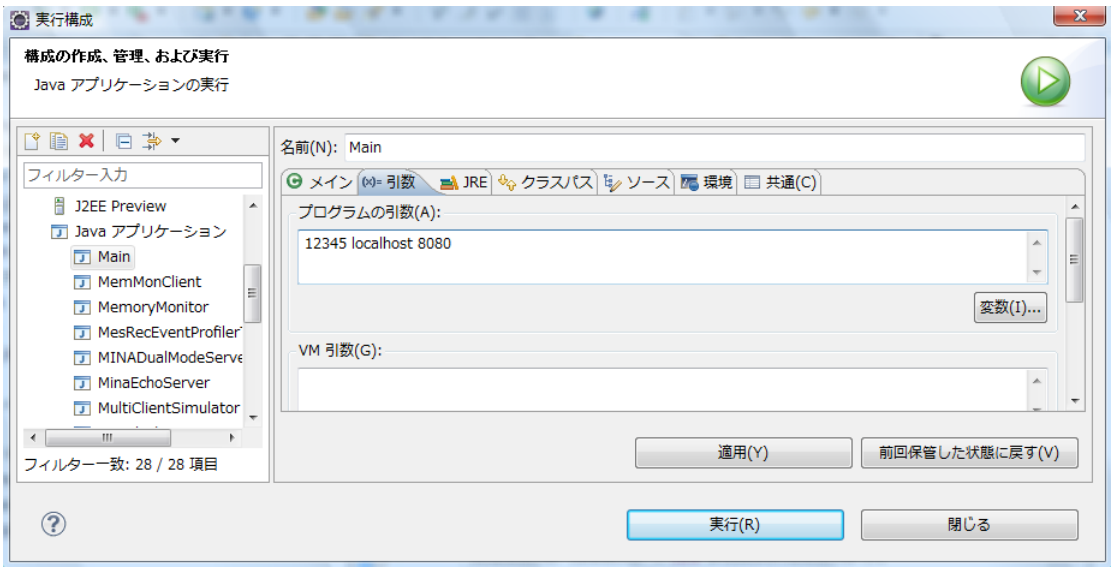


図 AB-6：Main の実行

3. 「実行(R)」ボタンをクリックすれば、このプログラムが開始し、コンソール上に「ポート 12345 で待機中」と表示されるはずである。

16.2.6 サーブレットへのプロキシ経由でのアクセス

1. Tomcat を開始させる
2. ブラウザから次のように HTTP://localhost:12345/tutorial/HelloWorld と HelloWorld をアクセスする。これまでと異なるのは、ポート番号が Tomcat が待っている 8080 ではなくて、プロキシが待っている 12345 番になっていることだけである。
3. そうすると、このアプリケーションのコンソールには下図のようなログが表示される：

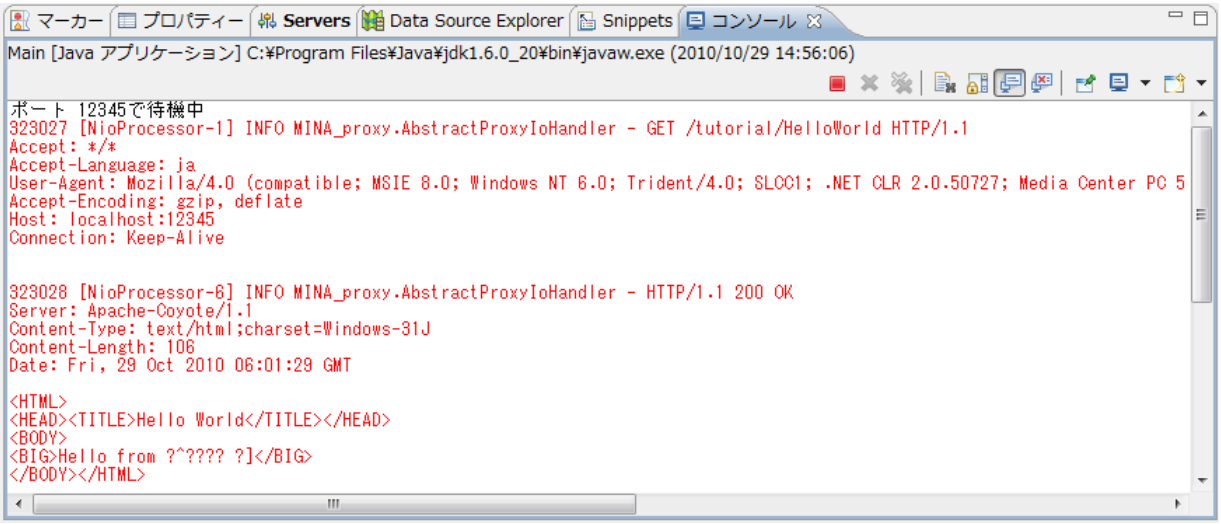


図 AB-7：プロキシからのログ出力

最初のログ（GET 以降）はブラウザ（この場合は Internet Explorer）からの HTTP 要求メッセージの内容をバイト列として表示している。次のログ（HTTP/1.1 以降）は HelloWorld サーブレットからの HTTP 応答メッセージである。ISO8859-1 ではなくて日本語で表示させたいときは、AbstractProxyIoHandler.java の CHARSET というフィールドを"Windows-31J"のように変更しても良い。

このように、サーブレットの開発に際しては、このプロキシが走っていれば、ポート番号を 8080 ではなくて 12345 とするだけで、簡単に実際にネットワーク上で交換されている TCP メッセージを確認することが出来る。

16.3節　　添付 WAR ファイルの Eclipse へのインポート

このチュートリアルには、tutorial.war、async_request.war、及び security.war というアプリケーション・ファイルが添付されている。このファイルの様式は Web Archive を略した WAR 形式と呼ばれているが、基本的には JAR ファイルである。

1. tutorial.war ファイルを当社のサイトのこのチュートリアルが含まれているディレクトリから自分のコンピュータの適当なフォルダにダウンロードする。なお、Internet Explorer ではダウンロードしようとリンクを右クリックすると tutorial.zip と単なる ZIP ファイルとみなしてしまうことがある。その場合はダウンロードしたファイルの拡張子を war に戻さなければならない。面倒な人はブラウザとして Firefox あるいは Chrome を使うことをお勧めする。

2. Eclipse のプロジェクト・エクスプローラを右クリック→「インポート(I)」→「WAR file」を選択すると下図のような画面が表示されるので、ダウンロードしたフォルダからこのファイルをインポートする。

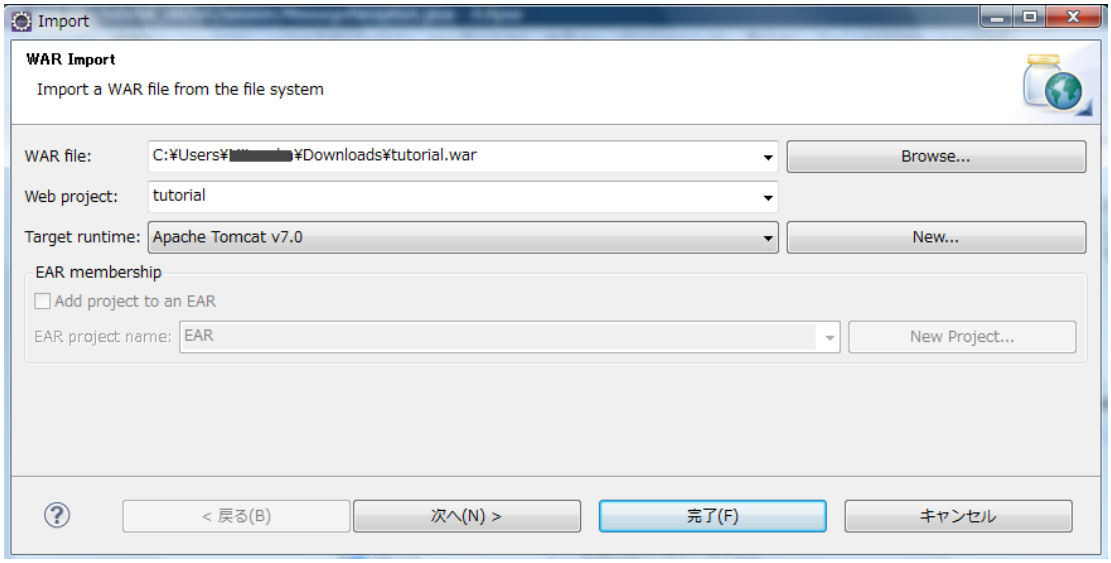


図 AC-1：WAR のインポート元の指定

3. Eclipse 上に既に tutorial というプロジェクトを作ってしまっているときは、そのプロジェクトの名前を変更しておく。
   1. プロジェクト・エクスプローラ上の既存の tutorial を右クリック→「リファクタリング(T)」→「名前の変更(N)」で例えば tutorial_old などと指定してやる。

4. インポートしたら正しくインポートされていることを、プロジェクト・エクスプローラでそのプロジェクトを展開して確認する。

5. Eclipse の Tomcat への登録は「Tomcat 7 の開始と停止」の節を見て頂きたい。実行プロジェクト(ここでは tutorial)を選択するには、Eclipse の右下にあるサーバ・ビューの Servers タブの"Tomcat v7.0 Server at localhost"の個所を右クリックして、下図のように「Add and Remove...」から起動させたいプロジェクト(ここでは tutorial)を Available から Configured に移す。

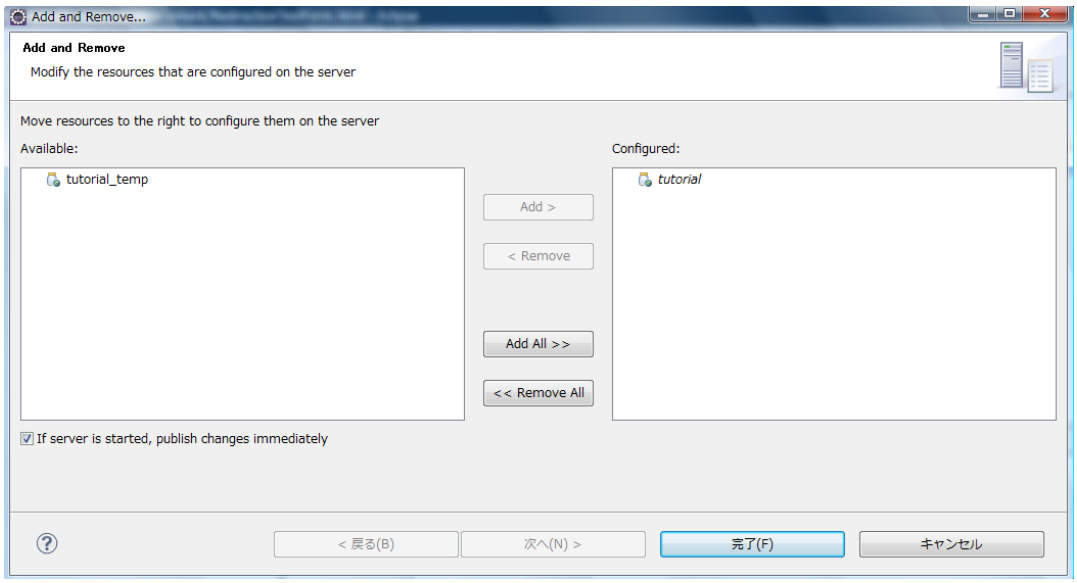


図 AC-2：Tomcat 7 に tutorial をリソースとして設定

16.4節　Eclipse 上の tutorial アプリケーションの Tomcat 7 への配備

このチュートリアルで使用した tutorial というアプリケーションは皆さんが「Tomcat 7 のインストールと設定」の章で示した手順を使ってインストールした Tomcat 7 に配備して、これを実行させることが出来る。なお Tomcat 7 上に既に同じ名前のアプリケーションが存在している場合は、そのフォルダを予め削除しておく必要がある。

16.4.1　WAR 形式のエクスポート

Eclipse のプロジェクト・エクスプローラ上の tutorial というプロジェクトを右クリック→「エクスポート(X)」→「WAR file」を選択すると下図のような画面が表示されるので、下図のように c:\tomcat7\webapps のディレクトリに tutorial.war という宛先を指定する。ソース・ファイルもエクスポートするようチェックする。

完了ボタンをクリックすれば、エクスポートが終了する。

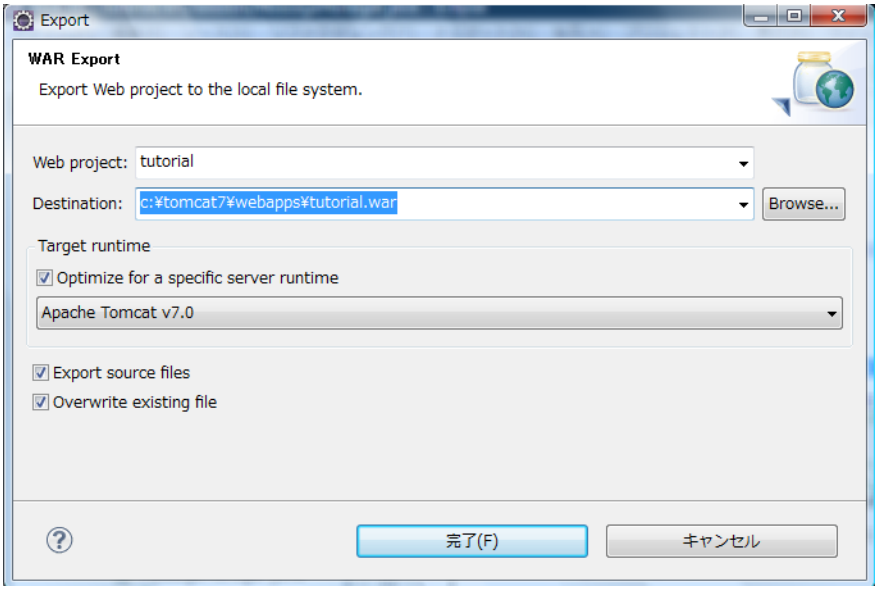


図 AD-1：WAR のエクスポート先の指定

16.4.2　Tomcat 7 の起動

これは「Tomcat のバージョン 7 版（例えば 7.0.2）をインストール」の節のとおりである。

この WAR ファイルが Tomcat によって正しく展開・配備されたことは、Tomcat のウィンドウで次のような表示がされることで判る。

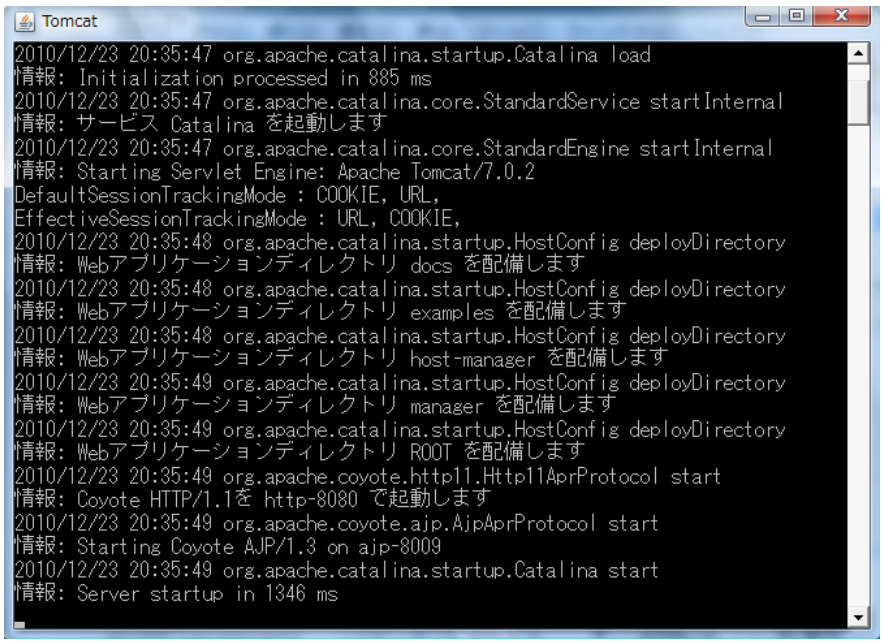


図 AD-2：Tomcat のコンソール上での確認

また、下図のようにエクスプローラでもそれを確認できる。

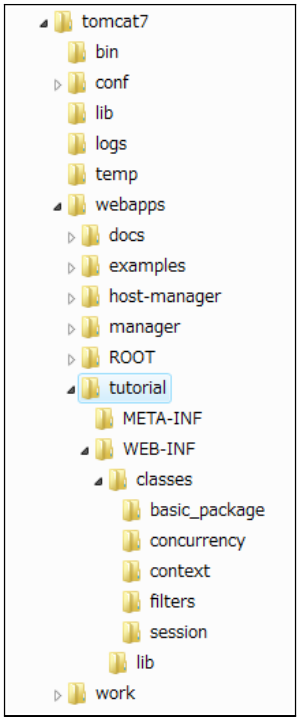


図 AD-3：エクスプローラでの確認

次に自分のブラウザから例えば次のようにアクセスして、正常に動作することを確認する。

```
http://localhost:8080/tutorial/MessageReception
```

16.4.3 DOS 上での WAR ファイルの作成

逆に Tomcat 7 にあるアプリケーションの WAR ファイルの作成する方法をお教えしよう。

1. コマンド・プロンプト画面で、c:\tomcat7\webapps\tutorial のディレクトリに移る。

2. 次のコマンドを実行する:

```
jar cvf tutorial.war .
```

最後のスペースとピリオドを忘れないように注意されたい。
以下はその実行結果の例である。作成された WAR ファイルは tutorial のディレクトリに置かれるので、このファイルをその上の webapps のディレクトリに移しておいたほうが良い。

```
c:\tomcat7\bin>cd c:\tomcat7\webapps\tutorial

c:\tomcat7\webapps\tutorial>jar cvf tutorial.war .
マニフェストが追加されました。
ErrorCodeTest.html を追加中です。(入 = 3666) (出 = 1113)(69% 収縮されました)
ForwardTestForm.html を追加中です。(入 = 760) (出 = 444)(41% 収縮されました)
jpegimage.jpg を追加中です。(入 = 99092) (出 = 74019)(25% 収縮されました)
JspExample1.jsp を追加中です。(入 = 376) (出 = 269)(28% 収縮されました)

<<途中省略>>

WEB-INF/classes/session/SimpleSessionTest.class を追加中です。(入 = 3897) (出 =
1948)(50% 収縮されました)
WEB-INF/classes/session/SimpleSessionTest.java を追加中です。(入 = 3872) (出 = 1
626)(58% 収縮されました)
WEB-INF/classes/session/Utilities.class を追加中です。(入 = 4955) (出 = 2324)(53
% 収縮されました)
WEB-INF/classes/session/Utilities.java を追加中です。(入 = 3269) (出 = 1073)(67%
 収縮されました)
WEB-INF/lib/ を追加中です。(入 = 0) (出 = 0)(0% 格納されました)
WEB-INF/lib/slf4j-api-1.6.1.jar を追加中です。(入 = 25496) (出 = 22179)(13% 収縮
されました)
WEB-INF/lib/slf4j-simple-1.6.1.jar を追加中です。(入 = 7669) (出 = 6294)(17% 収
縮されました)

c:\tomcat7\webapps\tutorial>
```

16.5節　Eclipse 上での security アプリケーションのインストール法

このチュートリアルの「セキュリティ」の章の教材として security.war が用意されている。tutorial のアプリケーションをインポートしたと同様にこの WAR ファイルを Eclipse にインポートすることに加えて、このアプリケーションを動作させるには server.xml と tomcat-users ファイルの設定、及び認証の為の証明書のインストールが必要になる。

16.5.1　server.xml ファイルの設定

security.war をインポートした後では、プロジェクト・エクスプローラ上には下図のような構成が表示されている筈である。Tomcat v7.0 Server at localhost-config のフォルダには、DOS 上の CATALINA-HOME\conf のなかで必要なものを抽出し、Eclipse 用に加工されている。

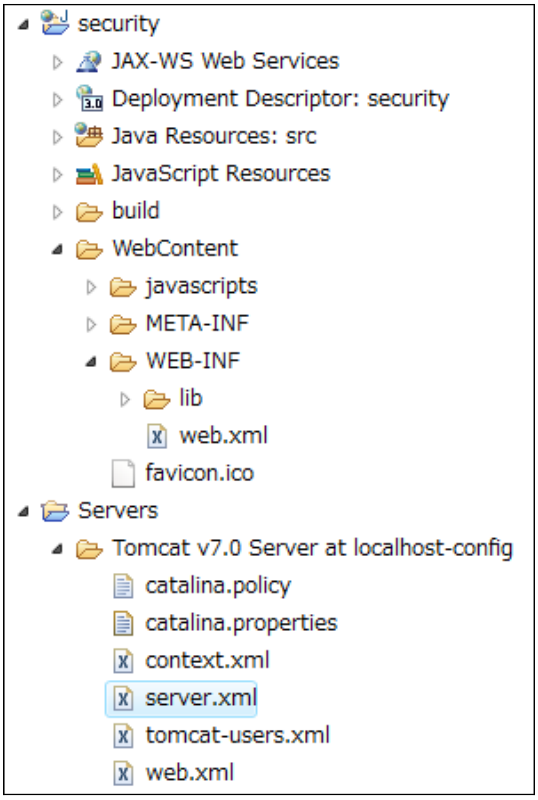


図 AE-1：security アプリケーションと Servers の構成

このアプリケーションでは SSL 接続として Tomcat が用意している非ブロッキングの NIO コネクタを使用している。従って、server.xml ファイルの SSL コネクタの設定の個所を「server.xml の SSL コネクタの設定」の項で示したように、以下のような要素を追加する：

```
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
          port="8443" SSLEnabled="true"
          maxThreads="150" scheme="https" secure="true"
          clientAuth="false" sslProtocol="TLS"
          keystoreFile="c:\ssl\shop_cresc.jks" keystorePass="changeit" />

<!-- Define an AJP 1.3 Connector on port 8009 -->
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443"/>
```

16.5.2 tomcat-users.xml ファイルの設定

本チュートリアルでは、レルムとしてはデフォルトの UserDatabase レルムがそのまま使われており、そのパスは CATALINA_HOME/conf/tomcat-users.xml である。「レルム設定」の項で示したように、tomcat-users.xml ファイルには次のように自分が好きなユーザ名、パスワード、及びロール名を登録する:

```
<user name="sengoku" password="sengoku38" roles="administrator,manager"/>
<role rolename="administrator"/>
<role rolename="manager"/>
```

16.5.3 証明書のインストールとインポート

SSL 認証を実験するには、公式な認証局からの証明書を取得する必要がある。その為には「証明書キーストアを用意する」の項に従ってキーストアを用意し、「認証局からの証明書の取得」の項に従って証明書を取得し、またブラウザへのルート証明書のインポートを行う必要がある。

16.6節　Tomcat 7 における SSL 設定

*この節は Tomcat 7 のユーザ・ガイドの SSL 設定のページの邦訳である。不明な箇所は原文を参照して頂きたい。Tomcat 5 における SSL の設定ガイドは、日本語化されているので、そちらも見て頂きたい。Tomcat 7 の英文の設定ガイドは、Tomcat 5 のそれに対し一部追加と変更がされている。特に **SSL を使ったセッション追跡はサーブレット3.0 の新しい機能**である。また、JSSE 実装と APR 実装の双方のコネクタが用意されている。実際の設定に関しては、「TLS(SSL)設定」の節を参照されたい。*

16.6.1　クイック・スタート

> *殆どの相対パスが解決されるベース・ディレクトリのことを言うのに、ここでは$CATALINA_BASE という変数名を使う。CATALINA_BASE ディレクトリを用意することで複数のインスタンス用に Tomcat を設定することをしていない場合には、$CATALINA_BASE には Tomcat をインストールしたディレクトリである$CATALINA_HOME の値がセットされる。*

Tomcat 上での SSL 対応を行うようにインストールと設定をする為には、以下のようなシンプルなステップをとる必要がある。更なる情報に関しては、この節の残りの個所を読んでいただきたい。

1. 以下のコマンドを実行することで、認証キーストアを生成する:
   Windows:
   ```
   %JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA
   ```
   Unix:
   ```
   $JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
   ```
   そして、"changeit"のパスワード値を指定する。

2. $CATALINA_BASE/conf/server.xml のなかの"SSL HTTP/1.1 Connector"の個所のコメント・アウトを外し、以下の設定の項で示すように修正する。

16.6.2　SSL 概説

SSL、あるいはセキュア・ソケット層(Secure Socket Layer)は、安全化された(セキュアな)接続上でブラウザとサーバが通信できるようにする技術である。このことは、一方の側でデータが暗号化されて送信され、伝送され、他方の側では復号化されたのち、そのデータが処理されることを意味する。これは双方向のプロセスであり、即ちサーバ及びブラウザの双方がデータ送信前に総てのトラフィックを暗号化することを意味する。

SSL プロトコルのもう一方の重要な側面は認証である。このことは、安全化された接続上でブラウザがあるウェブ・サーバとの通信を最初に試みる際に、そのサーバがブラウザに対し証明書(Certificate)のかたちでの証明書のセットを、そのサイトが一体誰であるかの証として提示することを意味する。ある場合には、そのサーバはそのブラウザは一体誰であるかの証として、ウェブ・ブラウザからの証明書を要求することもある。これは「クライアント認証」として知られるものであるが、これは実際的には個人間で使われるのではなくてビジネス間(B2B)の取引でより多く使われるものである。殆どの SSL 対応のウェブ・サーバたちはクライアント認証を要求していない。

16.6.2.1　SSL と Tomcat

安全なソケット(secure sockets)を活用する為に Tomcat を設定することは、通常は Tomcat がスタンド・アロンのウェブ・サーバとして動作しているときにのみ必要になることを指摘しておかねばならない。Tomcat が Apache あるいは Microsoft IIS のような別のウェブ・サーバの背後で主として Servlet/JSP コンテナとして稼働しているときには、ユーザたちからの SSL 接続を処理する為には、通常はプライマリのウェブ・サーバを設定する必要がある。一般的には、このサーバは総ての SSL 関連機能の交渉を行い、次にこれらの要求を暗号の復号化したあとでのみ、Tomcat コンテナ向けの要求を渡す。同様に、Tomcat は平文の応答を返し、それがユーザのブラウザに返される前に暗号化される。このような環境では、Tomcat はプライマリのウェブ・サーバとクライアント間の通信が安全化された接続上でなされてことを知っているが（何故ならそのアプリケーションはこれを求めることが出来る必要があるから）、自分自身が暗号化あるいは復号化に関わり合わない。

16.6.3 証明書

SSL 実装に際しては、ウェブ・サーバは安全化された接続を受け付ける各外部インターフェイス（IP アドレス）ごとに、それに対応した証明書を持たねばならない。この設計の背景は、特に重要な情報を受信する際に、そのサーバの所有者がユーザがそうだと思っている所有者であることの何らかの類の妥当な保障をサーバは提供しなければならないということである。証明書のより一般的な説明はこの資料の範囲外であるが、証明書をあるインターネット・アドレスの為の「運転免許書」だと考えればよい。この免許書は、このサイトの所有者あるいは管理者に関する何らかの基本的なコンタクト（連絡先）情報とともに、このサイトがどの会社に属しているかを記述している。

この「運転免許書」は暗号化されてその所有者が署名しており、したがって他の誰かが偽造するのは極めて難しい。電子商取引、あるいはその他のビジネス取引に関わる身元の認証が重要なサイトでは、証明書は VeriSign や Thawte のようなよく知られた認証局(CA: Certificate Authority)から購入される。そのような証明書は電子的に確認できる－実効的にこの認証局はそれが交付する証明書の認証の正当性を保障するので、ユーザがそれを交付した認証局を信用するなら、そのユーザはその証明書は有効だと信じることが出来る。

しかしながら多くの場合、認証は問題ではない。管理者は単にそのサーバによって送受信されているデータがプライベートな状態にあって、その接続上で傍受しているかもしれない誰かによって詮索されることが無いことが確保されることを望んでいる。幸いにも、Java には keytool と呼ばれる比較的シンプルなコマンド行ツールが用意されており、これは簡単に「自己署名」の証明書を生成できるものである。自己署名の証明書は単にユーザが生成した証明書であって、よく知られている認証局によって公式に登録されたものではなく、したがって信頼できることが保障されたものではまったくない。繰り返すが、これが重要かそうでないかは読者のニーズに依存する。

16.6.4 SSL を稼動させる為の一般的なポイント

あるユーザが皆さんのサイトのある安全化されたページを初めてアクセスするときは、一般的にそのユーザにはその証明書の詳細（会社名と連絡先のような）を含んだダイヤログが提示され、その証明を有効なものだとして受け入れ、その取引を継続するかどうかが聞かれる。ブラウザによっては、ある証明書に対しては永久的に受け付けるオプションが用意されており、このオプションを選択したときには、そのユーザは皆さんのサイトを訪問するたびにプロンプトが出てくる煩わしさが無くなる。他のブラウザではこのオプションが用意されていない。そのユーザによって一旦承認されたら、証明書は少なくともそのブラウザのセッション内では有効と看做されよう。

また、SSL プロトコルは出来るだけ効率的で安全であるよう設計されているが、暗号化と複合化は計算的には性能の点から言えば負荷がかかるものである。ウェブ・アプリケーション全体を SSL 上で走らせることは厳格に必要なものではなく、そのアプリケーションの開発者はどのページが安全化された接続を要するかを選択できる。かなりビジーなサイトでは、重要な情報が交換されうるページたちのみ SSL で動作させるのが通常である。そのようなページにはログイン・ページ、個人情報ページ、クレジット・カードの情報が送信される可能性があるショッピング・カートの清算ページ、などが含まれる。あるアプリケーション内のどのページも、http の代わりに https をアドレスの前につけるだけで安全化されたソケット上での送信を要求できる。絶対的に安全化された接続を必要とする

ページは、そのページ要求に関わるプロトコル・タイプをチェックし、https が指定されていないときはしかるべきアクションをとらねばならない。

最後に、安全化された接続上のユーザ名ベースの仮想ホストは問題を生じ得る。これは SSL プロトコル自身の設計上の制限によるものである。クライアントのブラウザがそのサーバの証明書を受け入れている SSL ハンドシェイクは、その HTTP 要求が受け付けられる前に生じなければならない。その結果、仮想ホスト名を含んだ要求の情報は認証前には判断できず、したがって単一の IP アドレスに対し複数の証明書を割りあてることは不可能である。単一の IP アドレス上の総ての仮想ホストたちが同じ証明書に対し認証する必要がある場合は、複数の仮想ホストの付加によりそのサーバ上の通常の SSL 動作に干渉を与えてはならない。しかしながら、殆どのクライアント・ブラウザはその証明書にリストアップされているドメイン名にたいし、そのサーバのドメイン名を比較している(公式の、CA 署名の証明書を最初に適用して)ことに注意が必要である。一般に、アドレス・ベースの仮想ホストのみが、一般的に商用レベルで SSL とともに使われている。

16.6.5 設定

16.6.5.1 証明書キーストアを用意

Tomcat は現在 JKS、PKCS11、あるいは PKCS12 フォーマットのキーストア上でのみ動作している。JKS フォーマットは Java の標準の"Java KeyStore"フォーマットで、keytool コマンド行ツールで生成されるフォーマットである。このツールは JDK の中に含まれている。PKCS12 フォーマットはインターネット標準であり、(なかでも) OpenSSL 及び Microsoft の Key-Manager により操作できる。

あるキーストア内の各エントリは別名(alias)文字列で識別される。多くのキーストア実装においては別名たちを大文字と小文字を区別していないが、区別した実装も存在する。例えば PKCS11 実装では別名を大文字小文字を区別することを要求している。別名に対する大文字と小文字に関わる問題を回避する為に、大文字と小文字だけで違うような別名たちを使うことは推奨されない。

既存の証明書を JKS キーストアにインポートするには、keytool に関するドキュメンテーション (JDK ドキュメンテーション・パッケージの中にある) を読んで頂きたい。OpenSSL はしばしばそのキーの前に読み取り可能なコメントを付加しているが、keytool はそれに対応しておらず、従って keytool を使ってキーをインポートする前に OpenSSL のコメントがあればそれを削除することに注意のこと。

OpenSSL を使って皆さんの CA によって署名された既存の証明書を PKCS12 キーストアにインポートするには、以下のようなコマンドを実行することになる。

```
openssl pkcs12 -export -in mycert.crt -inkey mykey.key \
                    -out mycert.p12 -name tomcat -CAfile myCA.crt \
                    -caname root -chain
```

より複雑な場合には OpenSSL のドキュメントを参考にされたい。

最初から単一の自己署名の証明書(self-signed Certificate)を含んだ新しいキーストアを生成するには、端末コマンド行から以下のように実行する:

Windows の場合:

```
%JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA
```

Unix の場合:

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
```

(安全なアルゴリズムとしては RSA アルゴリズムが好ましく、これはまた他のサーバたちや部品たちと一般的な互換性を確保させる。)

このコマンドにより、皆さんが走らせているユーザのホーム・ディレクトリ内に、".keystore"という名前の新しいファイルを生成する。別の場所や名前を指定するには、-keystore パラメタとそれに続くそのキーストア・ファイルへの完全パス名を、上記の keytool コマンドに付加する。例えば:

Windows の場合:

```
%JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA \
 -keystore \path\to\my\keystore
```

Unix の場合:

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA \
 -keystore /path/to/my/keystore
```

このコマンドを実行したら、最初にこのキーストアのパスワード入力が促される。Tomcat が使っているデフォルトのパスワードは"changeit"(総て小文字)であるが、好きなカスタムなパスワードを指定することが出来る。また、のちほど示すように、server.xml 設定ファイル内にカスタムのパスワードを指定する必要がある。

次に、会社名、連絡先名などのようなこの証明書に関する全般的な情報の入力が促される。この情報は皆さんのアプリケーション内の安全化されたページにアクセスしようとするユーザに表示され、ここで提供される情報が彼らが期待するものと一致するかどうかを彼らが確認できる。

最後に、パスワードの入力が促されるが、このパスワードは(同じ keystore ファイル内にストアされた他の証明書たちでは無くて)特にこの証明書の為のパスワードである。ここでは keystore パスワード自体に使った**と同じパスワードを使わねばならない**。これは Tomcat 実装での制約である。(現在 keytool のプロンプトは ENTER キーを押すとこれを自動的に行うと告げている。)

総てが成功すれば、皆さんのサーバに使える証明書を持った keystore ファイルを皆さんがこれで持ったことになる。

注意:プライベート・キーのパスワードと keystore のパスワードは同じでなければならない。これが同じでないと、java.io.IOException: Cannot recover key の行つきのエラーが発生する。この問題は Bugzilla issue 38217 でドキュメント化されており、このドキュメントにはこの問題に関する更なる参照が含まれている。

16.6.5.2 Tomcat 設定ファイルの編集

Tomcat は SSL の 2 つの異なった実装を使用している:

- Java ランタイム(1.4 以降)の要素として提供されている JSSE 実装
- APR 実装、これはデフォルトとして OpenSSL エンジンを使用している

設定の詳細はどちらの実装が使われているかに依存する。Tomcat が使っている実装は以下に書かれているようにオーバライドされていない限り自動的に選択されている。そのインストールが APR を使っている(言い換えれば、Tomcat のネーティブなライブラリを皆さんがインストールした)ときは、Tomcat は APR SSL を使用し、そうでないときは Tomcat は Java JSSE 実装を使用する。

自動設定を避ける為には、Connector の protocol 属性のなかの classname を指定することで、どちらの実装を使用するかを指定できる。

APR ライブラリがロードされているかどうかに関わらず Java (JSSE)コネクタを指定するときは:

```
<-- Define a blocking Java SSL Coyote HTTP/1.1 Connector on port 8443 -->
<Connector protocol="org.apache.coyote.http11.Http11Protocol"
         port="8443" .../>

<-- Define a non-blocking Java SSL Coyote HTTP/1.1 Connector on port 8443
-->
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
         port="8443" .../>
```

そうではなくて APR コネクタを指定するときは(APR ライブラリが利用可能になっていなければならない)、次の指定を使う:

```
 <-- Define a APR SSL Coyote HTTP/1.1 Connector on port 8443 -->
<Connector protocol="org.apache.coyote.http11.Http11AprProtocol"
         port="8443" .../>
```

APR を皆さんが使っているときは、OpenSSL の代替エンジンを設定するオプションが使える:

```
 <Listener className="org.apache.catalina.core.AprLifecycleListener"
        SSLEngine="someengine" SSLRandomSeed="somedevice" />
```

デフォルト値は:

```
 <Listener className="org.apache.catalina.core.AprLifecycleListener"
        SSLEngine="on" SSLRandomSeed="builtin" />
```

従って、APR のもとで SSL を使用するには、SSLEngine 属性が off 以外の何かにセットされていることを確認する。デフォルト値は on で、他の値を使うときは、それは有効なエンジン名でなければならない。

Tomcat Native ライブラリ内に皆さんが SSL 対応を組み入れていないときは、この初期化を off に設定できる。

```
 <Listener className="org.apache.catalina.core.AprLifecycleListener"
        SSLEngine="off" />
```

SSLRandomSeed によりエントロピ源を指定できるようにしている。商用システムでは信頼できるエントロピ源が必要になるが、エントロピ収集には多くの時間を要し、従ってテスト・システムでは"/dev/urandom"のようなブロックなしのエントロピ源が使え、これにより Tomcat の立ち上げ時間が短くなる。

最後のステップは$CATALINA_BASE/conf/server.xml ファイル内の Connector を設定することであるが、ここに $CATALINA_BASE はその Tomcat インスタンスのベース・ディレクトリを表現している。SSL 接続の例となる <Connector>要素は Tomcat でインストールされているデフォルトの server.xml ファイルに含まれている。JSSE に場合は、この要素は次のようなものになる:

```
 <-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
<!--
<Connector
         port="8443" maxThreads="200"
         scheme="https" secure="true" SSLEnabled="true"
         keystoreFile="${user.home}/.keystore" keystorePass="changeit"
         clientAuth="false" sslProtocol="TLS"/>
-->
```

上記の例では、Tomcat が APR コネクタを使おうとする為、皆さんのパスに APR と Tomcat Native のライブラリがあると、エラーをスローする。この APR コネクタは SSL キーと証明書の為に異なった属性たちを使っている。
APR 設定の例を以下に示す:

```
 <-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
<!--
<Connector
         port="8443" maxThreads="200"
         scheme="https" secure="true" SSLEnabled="true"
         SSLCertificateFile="/usr/local/ssl/server.crt"
         SSLCertificateKeyFile="/usr/local/ssl/server.pem"
         clientAuth="optional" SSLProtocol="TLSv1"/>
-->
```

Connector 要素自身がデフォルトではコメント・アウトされていることに気がつかれるであろう。従ってそれを囲んでいるコメント・タグを外す必要がある。次に、必要に応じ指定されている属性をカスタマイズ出来る。いろんなオプションについては、サーバ設定参照資料を調べられたい。以下に SSL 通信の設定に際し最も関係がある属性たちのみを示すことにする。

ポート属性(デフォルト値は 8443)は安全化された接続の為に Tomcat が聞く TCP/IP ポート番号である。自分が使いたい任意のポート番号(https 通信でデフォルトとなっているポート番号の 443 のように)にこれを変更できる。しかしながら、多くの OS では、Tomcat を 1024 以下のポート番号で動作させるには特別な設定(それはこの資料の範囲外)が必要になる。

> *ここでポート番号を変更する際は、非 SSL コネクタ上の redirectPort 属性の為に指定されている値も変更しなければならない。これにより Tomcat はサーブレット仕様書に規定されているように SSL が必要と指定されているセキュリティ制約を持ったページをアクセスしようとするユーザを自動的にリダイレクトできるようにしている。*

SSL プロトコルを設定するのに使われる更なるオプションが存在する。皆さんが自分のキーストアをどのように設定したかによって幾つかの属性を付加または変更する必要がある場合がある。Java JSSE ベースの SSL コネクタを使っている場合は、設定オプション関しては Java HTTP コネクタ設定参照資料に記載されている。APR/ネーティブのコネクタを使っているときは、利用可能な設定オプションの詳細に関しては APR コネクタ設定ガイドを見て頂きたい。

これらの設定変更を完了したら、何時も行っているように Tomcat の再スタートを行わねばならない、そうすれば準備完了である。これで SSL を介して Tomcat が対応するウェブ・アプリケーションにアクセスできるようになる。例えば、

```
https://localhost:8443
```

を試してみれば、通常の Tomcat のページが出てくる(ROOT のウェブ・アプリケーションをいじっていない限り)。これがうまくゆかないときの為、以下の項でトラブル・シュートの為の幾つかのポイントを示す。

16.6.6 認証局からの証明書をインストールする

認証局(verisign.com、thawte.com、あるいは trustcenter.de のような)からの証明書を取得しインストールするには、これまでの説明を読んでから以下の指示に従う:

16.6.6.1 ローカルな証明書署名要求(CSR: Certificate Signing Request)を生成する

自分が選択した認証局から証明書を取得するには、いわゆる証明書署名要求(CSR)を生成しなければならない。この CSR は認証局があなたのウェブ・サイトが「安全(secure)」だと確認する証明書を生成するのに使われる。CSR を生成するには以下の手順に従う:

- ローカルな証明書を生成する(これまでの項で説明したように)

```
keytool -genkey -alias tomcat -keyalg RSA \
   -keystore <自分のキーストア・ファイル名>
```

  注意:場合によっては動作する証明書を生成する為に"first- and lastname"のフィールドにあなたのウェブ・サイトのドメイン(例えば www.myside.org)を入力しなければならないことがある。

- 次に以下のコマンドで CSR が生成される:

```
keytool -certreq -keyalg RSA -alias tomcat -file certreq.csr \
   -keystore <自分のキーストア・ファイル名>
```

これで認証局に提出できる certreq.csr と呼ばれるファイルが出来る(提出法に就いては認証局のウェブ・サイトの説明書を見られたい)。返信として証明書が渡される。

16.6.6.2 証明書のインポート

証明書が得られたので、あなたのキーストアにこれをインポートできる。まずいわゆるチェイン証明書あるいはルート証明書を自分のキーストアにインポートしなければならない。そのあとで自分の証明書のインポートに進むことが出来る。

- その証明書を取得した認証局からチェイン証明書をダウンロードする:
  Verisign.com の商用の証明書の場合は http://www.verisign.com/support/install/intermediate.html を、Verisign.com のトライアル証明書の場合は http://www.verisign.com/support/verisign-intermediate-ca/Trial_Secure_Server_Root/index.html を訪問する。
  Trustcenter.de の場合は http://www.trustcenter.de/certservices/cacerts/en/en.htm#server を訪問する。
  Thawte.com の場合は http://www.thawte.com/certs/trustmap.html を訪問する。

- そのチェイン証明書を自分のキーストアにインポートする:

```
keytool -import -alias root -keystore <自分のキーストア・ファイル名> \
   -trustcacerts -file <チェイン証明書のファイル名>
```

- 最後に自分の新しい証明書をインポートする:

```
keytool -import -alias tomcat -keystore <自分のキーストア・ファイル名> \
   -file <自分の証明書のファイル名>
```

16.6.7 トラブル・シューティング

SSL 通信の設定で遭遇するかもしれない共通的な問題と、それらに関してどうすべきかのリストを以下に示す。

- 自分のログ・ファイルに"java.security.NoSuchAlgorithmException"が記入される
  JVM が JSSE JAR ファイルを見つけられない。JSSE のダウンロードとインストールの説明書を調べる。

- Tomcat が起動するときに、"java.io.FileNotFoundException: {some-directory}/{some-file} not found"といったたぐいの例外が起きる
  可能性があるのは Tomcat が探しているキーストア・ファイルが見つけられない場合である。Tomcat は Tomcat が走っているユーザのホーム・ディレクトリ(皆さんの環境が同じでない可能性がある)内に

keystore という名前のキーストア・ファイルが存在していることを想定している。キーストア・ファイルがどこか別の場所にある場合は、Tomcat 設定ファイルの<Factory>要素に keystoreFile 属性を追加する必要がある。

- Tomcat が起動するときに、"java.io.FileNotFoundException: Keystore was tampered with, or password was incorrect" (キーストアが改ざんされている、あるいはパスワードが違っている)といったたぐいの例外が発生する。
  実際に自分のキーストア・ファイルが改ざんされていないとすれば、一番可能性があるのは Tomcat があなたがキーストア・ファイルを生成するときに使ったパスワードと違ったパスワードを使っている場合である。この問題を解決するには、一旦前に戻ってキーストア・ファイルを作り直すか、Tomcat 設定ファイルの<Connector>要素にある keystorePass 属性を追加あるいは更新するかする。パスワードは大文字と小文字が区別されることに注意のこと。

- Tomcat が起動するときに、"java.net.SocketException: SSL handshake errorjavax.net.ssl.SSLException: No available certificate or key corresponds to the SSL cipher suites which are enabled."といった例外が発生する。
  可能性があるのは、Tomcat が指定されたキーストア内にサーバ・キーの別名(alias)が見つけられなかった場合である。Tomcat 設定ファイル内の<Connector>要素にある正しい keystoreFile と keyAlias が指定されているかどうかを確認する。keyAlias の値は大文字と小文字が区別されている可能性があることに注意のこと。

それでも問題が解決できないときは、良い情報源は TOMCAT-USER メーリング・リストである。このリストのこれまでのメッセージたちのアーカイブへのポインタを見つけることが出来る。また加入と退会に関する情報も http://tomcat.apache.org/lists.html にある。

16.6.8 自分のアプリケーション内でセッション追跡に SSL を使用する

これは Servlet 3.0 仕様書の新しい機能である。物理的なクライアント-サーバ間の接続に結び付けられた SSL セッション ID を使用するので、幾つかの制限が存在する。即ち:

- Tomcat は isSecure 属性が true にセットされたコネクタを持っていなければならない
- もし SSL 接続がプロキシあるいはハードウエアのアクセラレータで管理されているときは、SSL セッション ID が Tomcat にとって可視である為には、これらの SSL 接続は SSL 要求ヘッダたちを取り込んでいなければならない (SSLValve を参照のこと)
- SSL セッション ID は各ノードによって異なる為、Tomcat がその SSL 接続を終了したときは、セッション・レプリケーションが使えなくなる

SSL セッション追跡が使えるようにするには、コンテキスト・リスナを使い、そのコンテキストの為の追跡モードを SSL だけにセットする必要がある(もし他の追跡モードが可能となっていると、そちらが優先して使われる)。これは以下のようなコードとなろう:

```
package org.apache.tomcat.example;

import java.util.EnumSet;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.SessionTrackingMode;

public class SessionTrackingModeListener implements ServletContextListener {
```

```
    @Override
    public void contextDestroyed(ServletContextEvent event) {
        // Do nothing
    }

    @Override
    public void contextInitialized(ServletContextEvent event) {
        ServletContext context = event.getServletContext();
        EnumSet<SessionTrackingMode> modes =
            EnumSet.of(SessionTrackingMode.SSL);

        context.setSessionTrackingModes(modes);
    }

}
```

注意：SSL セッション追跡は BIO 及び NIO コネクタ用に実装されている。APR コネクタ用は未だ実装されていない。

16.6.9 その他のポイント

その要求オブジェクトから SSL セッション ID にアクセスするには、次のような行を使用する：

```
String sslID = (String)request.getAttribute("javax.servlet.request.ssl_session");
```

この領域に関する更なる情報は Bugzilla を見て頂きたい。

SSL セッションを終了させるには、以下のようなコードを使用する：

```
// Standard HTTP session invalidation
session.invalidate();

// Invalidate the SSL Session
org.apache.tomcat.util.net.SSLSessionManager mgr =
    (org.apache.tomcat.util.net.SSLSessionManager)
    request.getAttribute("javax.servlet.request.ssl_session_mgr");
mgr.invalidateSession();

// Close the conection since the SSL session will be active until the connection
// is closed
response.setHeader("Connection", "close");
```

このコードは SSLSessionManager クラスを使っているので Tomcat 専用であることに注意されたい。これは現在 BIO と NIO コネクタでのみ利用できるが、APR/ネーティブのコネクタでは使えない。