JRun による
アプリケーションの開発

Windows®、UNIX™、および
Linux™ 用 JRun 3.1

Allaire® Corporation

版権告知

© 2000, 2001 Allaire Corporation. All rights reserved.

本書とその中に記載されているソフトウェアは、ライセンス契約のもとに供給され、このライセンスの条項に従ってのみ使用または複製することができます。本書の内容は、情報の提供のみを目的としており、予告なく変更されることがあります。これについて、Allaire Corporation は一切責任を負いません。Allaire Corporation は、本書の誤りについて一切責任を負いません。

ライセンスによる許可がある場合を除いて、Allaire Corporation の事前の書面による許可なしに、この出版物の一部または全部の複製、検索システムへの保存、あるいは電子的、機械的な記録、または他のいかなる形態や手段による転送を行うことはできません。

ColdFusion は、Allaire Corporation の登録商標です。Allaire、HomeSite、JRun、JRun Studio、<CF_Anywhere>、ColdFusion ロゴ、JRun ロゴ、および Allaire ロゴは、米国および各国における Allaire Corporation の商標です。MacOS は、Apple Computers Inc. の商標です。Microsoft、Windows、Windows NT、Windows 95、Microsoft Access、および FoxPro は、Microsoft Corporation の登録商標です。Java、JavaBeans、JavaServer、JavaServer Pages、JavaScript、JDK、および Solaris は、Sun Microsystems Inc. の商標です。UNIX は、The Open Group の商標です。PostScript は、Adobe Systems Inc. の商標です。その他の製品および製品名は、各所有者に帰属する商標です。

この製品には RSA Data Security からライセンス供与されたコードが含まれています。このソフトウェアの著作権の一部は、Merant, Inc. に帰属します。1991-2001

部品番号 : AA-JJDEV-RK

目次

JRun へようこそ xix

製品の機能 xx

JRun 製品のラインナップ xxi

対象読者 xxii

開発者リソース xxii

JRun 文書の概要 xxiii
印刷およびオンライン文書セット xxiii
オンライン文書へのアクセス xxiii

その他のリソース xxiv

お問い合わせ先 xxv

第 1 部 はじめに. 1

第 1 章 JRun の概要 3

JRun について. 4
サーバー側 Java の利点 4

JRun アーキテクチャ モデル. 5
JRun の 3 階層モデルのサポート 7

JRun の機能. 8
Web サーバーへのプラグイン接続 8
拡張性 8
セキュリティ 9
セッション トラッキング 10
監視ユーティリティ 10
開発ツール 11

後続の章について 13

第 2 章 JRun プログラミング モデル 15

JRun プログラミング環境 16

JRun サーバー. 18
インストール済みの JRun サーバー 19
JRun サーバーと Java Virtual Machine (JVM) の併用 21

Web サーバー 22
JRun Web サーバー. 23

Web アプリケーション 24

Enterprise JavaBeans 26

Java Message Service 26

JRun の設定. 27
JMC の使用. 27
プロパティ ファイルの使用 28

JRun と Java IDE の併用. 29
JRun.main メソッドの指定. 29
クラスパスでの JAR ファイルの指定. 30
コマンドライン引数の指定. 30
EJB エンジンとの統合 31

第 3 章 サーブレットの使用. 33

Java サーブレットの使用 34
サーブレットの呼び出し 35
サーブレットの利点 35
サブレットと CGI 37
サーブレットの作成. 38
JRun によるサーブレットのサポート 38

JRun のサーバー側スクリプト ファイル. 39
サーバー側スクリプトのタイプ 39
サーバー側スクリプトに関する JRun の機能 40

サーブレットと JSP ページ 40
HTTP 要求と応答 41
クライアントへの結果の返送. 42
例外処理 42
ページ コンテキスト情報の維持 42
セッションの処理. 43
アプリケーション コンテキストの追跡 43
設定情報へのアクセス 44
アプリケーションの公開. 44

Java によるサーブレットの作成. 45

JSP ページとしてのサーブレットの作成 46

第 4 章 EJB の概要 47

概要. 48

API 50

サービス 51
- Bean の開発 51
- ライフサイクル 52
- コンテキスト 52
- トランザクション 53
- パーシスタンス 53
- メッセージ サポート. 53
- セキュリティと認証. 54

環境. 55

インストールの必要条件 55

ディレクトリ情報 55

第 5 章 Web アプリケーションの開発. 57

Web アプリケーションの概説. 58
- Web アプリケーションの利点 58
- Web アプリケーションの使用 59
- Web アプリケーションのディレクトリ構造 60
- 公開記述子 (web.xml) 62
- アプリケーション コンポーネント 62
- Web アプリケーション、JRun サーバー、Web サーバー 63
- Web アプリケーション クラスパスの決定. 64
- Web アプリケーション間でのクラスの共有 66
- 分散型 Web アプリケーション 67

既定の Web アプリケーションの使用 67
- 既定の Web アプリケーションに対する要求の処理 68
- default-app Web アプリケーションの使用. 69
- 既定のアプリケーション ディレクトリ構造 69
- 既定の Web アプリケーションのクラスパス. 70

Web アプリケーションの開発. 70
- Web アプリケーションの作成 70
- Web アプリケーション コンポーネントの追加. 71

Web アプリケーションの公開. 76
- 公開用アプリケーションのパッケージ化. 76
- JRun 内での Web アプリケーションの公開. 77

第 6 章 JRun によるサーブレットへの要求のマッピング 79

サーブレット マッピングの基本情報 80

マッピング 80
アプリケーション マッピング 82
サーブレット マッピング 82
invoker サーブレットの使用 83

JRun によるファイルの提供 84
Web サーバーの対話 84
事例 85

第 2 部 サーバー側スクリプトと JSP 89

第 7 章 JSP の作成. 91

JavaServer Pages の作成 92
JSP スクリプトの概要 92
はじめての JSP ページの作成 93
複数の HTML/Java ブロック 94
JSP から Java サーブレットへの変換 95

JSP ファイルの開発 96
JSP の保存 96
変数の宣言 97
JSP への条件ロジックの追加 98
式の使用 98
JSP オブジェクトの使用 99
JSP オブジェクトとパラメータおよび属性の使用 100
include の実行 101
別の JSP の呼び出し 103
JSP 出力のバッファ 105
タグ ライブラリの使用 105
エラーの処理 106
JSP コンパイラの使用 108

JSP の以前のリリースからのアップグレード 109

第 8 章 JSP の構文. 111

JSP 1.1 仕様と JRun の互換性 112

JSP の基本構文 112
JSP テンプレート テキストの挿入 112
空白文字の使用 113
開始タグと終了タグの配置 113
属性値の引用 113
エスケープ文字 113

コメントの挿入 114
JSP における相対 URL の指定 115

ディレクティブ 115
page ディレクティブ 116
include ディレクティブ 120
taglib ディレクティブ 120

スクリプト要素 122
宣言 122
スクリプトレット 123
式 123

アクション 124
jsp:useBean 124
jsp:setProperty 126
jsp:getProperty 128
jsp:include 128
jsp:forward 129
jsp:param 130
jsp:plugin 130

第 9 章 JSP オブジェクト リファレンス 135

JSP オブジェクト 136
JSP オブジェクトへのアクセスの取得 137
JSP オブジェクトの使用 137

application オブジェクト 138

config オブジェクト 139

exception オブジェクト 139

out オブジェクト 140

pageContext オブジェクト 141

request オブジェクト 142

response オブジェクト 143

session オブジェクト 144

第 10 章 JSP のコンパイル 145

JSP コンパイラ 146
JSP コンパイラのプロパティの設定 146
JSP のコンパイルのバイパス 147

JSPC コンパイラ 150
JSPC コンパイラに必要なソフトウェア 150
JSPC コンパイラの起動 151

第 11 章 JSP でのカスタム タグの作成 155

概要 156

カスタム タグの概念 157

構文 159

tag ディレクティブ 159

tagAttribute ディレクティブ 160

tagVariable ディレクティブ 161

taglib ディレクティブ 162

使用方法 163

サンプル 163

単純な例 163

属性との対話 164

本文コンテンツとの対話 165

ループ 166

スクリプト変数の使用法 167

高度な使用方法 168

複数のハンドラの実装 168

ファイル拡張子 .jst の再マッピング 168

第 12 章 JSP の例 169

要求の処理と応答の生成 170

JSP から別の JSP の呼び出し 171

セッションのトラッキング 173

アプリケーション オブジェクトの使用 176

タグ ライブラリの使用 178

第 13 章 JSP のアップグレード 181

旧リリースからのアップグレード 182

バージョン 1.1 PR1 からのアップグレード 182

仕様への変更 182

バージョン 1.1 PD1 からのアップグレード 183

仕様への追加 183

仕様への変更 183

仕様からの削除 183

バージョン 1.0 からのアップグレード 184

仕様への追加 184

仕様への変更 184

バージョン 0.92 からのアップグレード 185
仕様への変更 185
仕様からの削除 185
仕様への追加 185

第 14 章 サーバー側インクルード ファイルの使用 187

サーバー側インクルードの使用 188
Servlet タグ 188
Include タグ 189

第 15 章 プレゼンテーション テンプレート 191

プレゼンテーション テンプレート (THTML ファイル) の使用 192
default.template の使用 192
default.definitions の使用 193
ファイルの位置 195

第 16 章 タグレット 197

タグレットとは 198
SSI タグレット 198
SSI タグレットのロードと使用 199

第 3 部 サーブレットの開発 201

第 17 章 Java サーブレットの処理 203

サーブレットについて 204
Java サーブレット API バージョン 2.2 204
基本サーブレット クラスおよびインターフェイス 205
サーブレットのライフサイクル 206
同期化 208
メソッド シグネチャでの synchronized キーワードの使用 208
同期化されたコードの使用 209
SingleThreadModel インターフェイスの使用 209
オブジェクトスコープ変数にアクセスするメソッドの同期化 209
サーブレットのチェーン化 210
サーブレット チェーンの明示的確立 210
MIME タイプによるサーブレット チェーン化の確立 211
Web アプリケーション 212

第 18 章 サーブレットのチュートリアル 213
第 1 部 214
第 2 部 216
第 3 部 218

第 19 章 サーブレット API の基本情報 221
Java サーブレットのタイプ 222
サーブレット API のパッケージ 222
javax.servlet 222
javax.servlet.http 224
サーブレット API に関するリファレンス情報 225

第 20 章 Java サーブレット API によるプログラミング 227
GenericServlet クラスにおけるメソッドのコーディング 228
service メソッドの書き換え 228
getServletInfo メソッド、init メソッド、および destroy メソッドの書き換え 229
サーブレット情報、要求情報、および アプリケーション情報へのアクセス 231
HttpServlet クラスにおけるメソッドのコーディング 232
service メソッドの書き換え 232
doGet メソッドの書き換え 232
doPost メソッドの書き換え 233
ほかの HTTP メソッドの書き換え 235

第 21 章 サーブレットの例 237
制御の受け渡し 238
ほかのサーブレットに制御を渡す方法 238
JSP に制御を渡す方法 239
セッションのトラッキング 240
データベースへのアクセス 242
JDBC-ODBC ブリッジの使用 242
JDBC ドライバの使用 245
JRun データ ソース サービスの使用 246
クッキーの処理 249
サーブレット コンテキストの使用 250
ほかのファイルからのコンテンツのインクルード 252
include メソッドの使用 252
getResource メソッドの使用 253

第 22 章 カスタム タグとタグ ライブラリの作成 255

カスタム タグとタグ ライブラリについて 256

タグ ライブラリのコーディング 257
- クラスとインターフェイス 257
- JSP 開発者のカスタム タグの使用法 258
- 単純なタグ ハンドラのコーディング 258
- TLD ファイルの作成 260
- 属性との対話 262
- 本文コンテンツとの対話 265
- ネストしたタグ ハンドラのコーディング 269
- タグを使用したスクリプト変数の作成 271

JSP でのタグの使用法 275

タグ ライブラリのパッケージ化 276

第 23 章 サーブレット API の変更点 279

サーブレット API の 2.0 から 2.1 への変更点 280
- API の改良 280
- 機能の拡張 281

サーブレット API の 2.1 から 2.2 への変更点 283
- API の改良 283
- Web アプリケーション 283
- ほかの拡張機能 285

第 4 部 Enterprise JavaBeans の開発 287

第 24 章 EJB のディレクトリ 289

構造 290
- JRun home 291
- servers/ *サーバー名* /deploy 291
- docs 291
- lib 292
- lib/ext 293
- servers/ *サーバー名* /runtime 293
- servers/ *サーバー名* /runtime/classes 294
- samples 294

第 25 章 プロパティ 295

概要 296

サーバー プロパティの設定 296

コンテナ プロパティの設定 297

Bean 情報の指定 297

例. 298
コマンド ラインによる書き換え 299
プロパティ ファイルによる書き換え 300
Bean プロパティによる書き換え. 301
実行時 Bean プロパティによる書き換え 302

第 26 章 リソース管理 303

概要. 304

ローカル ホーム オブジェクト 305

インスタンス マネージャ. 306
コンテキスト /Bean インスタンス プール 306
インスタンスのステートの変化. 307

データベース接続管理 308

ローカル キャッシュ /ストア 308
JRun instance.store 309
プロパティ 309

ロードされたユーザおよびロール 310

第 27 章 Bean の開発. 311

概要. 312

Bean の リモート インターフェイスの作成. 312

Bean のホーム インターフェイスの作成 313

Bean クラス実装の作成 314
エンティティ Bean 314
セッション Bean 316

バージョン管理 317

第 28 章 Bean 管理パーシスタンス. 319

概要. 320

データ ソースのプロパティ. 320

Bean の必須メソッド 321
ejbCreate および ejbPostCreate. 322
ejbLoad 323
ejbStore. 324
ejbRemove 325
finder メソッド 325

ビジネス メソッド 328

第 29 章 コンテナ管理パーシスタンス 329

概要 330

公開記述子の要素 330
複数の SQL ステートメント 334
SQL ステートメントのパラメータ 335

Bean のメソッド 336
ejbCreate および ejbPostCreate 336
ejbLoad 338
ejbStore 339
ejbRemove 340
finder メソッド 340

ストアド プロシージャの呼び出し 343

CMP での開発者の責任 343

第 30 章 Java でのメッセージング 345

概要 346
JRun メッセージング アーキテクチャ 347
JMS サポートの有効化 348

メッセージ コンポーネント 349
メッセージ ヘッダ フィールド 349
メッセージ プロパティ 350
メッセージ本文の種類 351

メッセージ タイプ 351
ポイントツーポイント 351
パブリッシュ / サブスクライブ 363

第 31 章 EJB クライアントのコーディング 377

概要 378

簡単なアクセス 379

Web 認証を通じたアクセス保護 380

カスタム認証によるアクセス保護 381

第 32 章 高度なテクニック 383

概要 384

トランザクション 384
トランザクション属性の設定 384
コンテナ管理トランザクション 386
Bean およびクライアント管理トランザクション 387

その他の高度な EJB トピック 389
デッドロック 389
セキュリティの無効化 389
SSL 389
ローカル Bean 389
クライアント アプリケーション 390
セッション スコープ 390
クライアント接続の定義 390

第 33 章 EJB エンジンの使用 393
概要 394
クラスのロード 394
クラスパス 394
スタンドアロン モードでの EJB エンジンの動作 395
セットアップのトラブルシューティング 396
許可 396
標準拡張 396
サーバー クラスパス (スタンドアロン EJB エンジン) 396
クライアントのセットアップ 396

第 5 部 アプリケーションの公開 397

第 34 章 Web アプリケーションのアセンブルと公開 399
概要 400
Web アプリケーション アセンブルとは 400
Web アプリケーションの公開とは 401
WAR ファイル 401
公開用 Web アプリケーション パッケージの作成 402
JSP ページのコンパイルの無効化 402
WAR ファイルの作成 403
Web アプリケーションの公開 404
JMC の使用 404
コマンド ライン インターフェイスの使用 405
認証のためのユーザとロールの定義 406
ホット デプロイおよびオート デプロイの使用 406

第 35 章 Enterprise JavaBeans の公開 409

概要. 410

公開記述子のコーディング 411
- 基本要素 411
- セキュリティ要素 412
- env-entry 要素 413
- 公開記述子の例 414

JAR ファイルの作成. 418

プロパティ ファイルのコーディング. 418

Deploy ツールの実行 420
- 再公開 421

その他のクラスの取り込み 421

Bean のダイナミック ローディングの使用 422

実行時環境 423
- サーバー環境 423
- クライアント環境 423

第 36 章 J2EE アプリケーションの公開 425

J2EE アプリケーションの公開について 426

EAR ファイル 426

公開用 J2EE アプリケーションのパッケージ化. 427
- application.xml ファイルの作成 428
- EAR ファイルの作成. 429

J2EE アプリケーションの公開 429
- JMC の使用. 429
- コマンド ライン インターフェイスの使用. 430
- セキュリティのためのユーザとロールの定義. 430

第 6 部 JRun での作業. 431

第 37 章 Web サーバー接続の監視 433

Web サーバー接続の監視. 434
- 監視メカニズムの設定 435
- 監視出力形式の設定 436
- 既定の監視形式 438

プロパティの監視 439

第 38 章 ログ 441

概要 442
メッセージのタイプ 442
メッセージの出力先 442
標準のログ ライター 443
既定のログ設定 444
例 446
複数ファイルへのログ メッセージの書き込み 446
ファイルと電子メールへのログ メッセージの書き込み 447
ログ メッセージ定義の変更 449
標準出力と標準エラーへのログ出力 449
電子メールへのログ 450
スクリーン ライターの使用 450
Windows NT/95/98 451
Unix/Linux 452
標準出力 452
ログ メカニズムの定義 453
ログ メッセージの形式 454
ログ情報の形式 454
ログ プロパティ 455
一般プロパティ 455
スレッド ロガー プロパティ 456
ディスパッチ ロガー プロパティ 457
ファイル ライター プロパティ 458
電子メール ライター プロパティ 461
スクリーン ライター プロパティ 462
システム ログ プロパティ 462

第 39 章 Web アプリケーションの認証 463

認証 464
認証の例 465
ユーザ、グループ、およびロール 466
アプリケーション認証とサーバー認証 467
サーバーが認証を実施する時期 467
JRun 認証メカニズムの設定 468
アプリケーション認証の設定 469
Web アプリケーションへの認証ロールの割り当て 469
ユーザ認証メソッドの設定 474
サーバー認証メカニズムの制御 478
既定の JRun 認証メカニズムの使用法 478
JRun によるカスタム認証メカニズムの使用 481
JRun の外部でのアプリケーションの実行 482

認証プロパティ 482
local.properties 内のプロパティ 482
users.properties 内のプロパティ 483

第 40 章 サーブレット メソッド パフォーマンスの監視 485

概要 486

メソッド タイミングの機能 486

例 488
タイミングの有効化と既定のプロパティ値の受け入れ 488
指定されたクラスおよびメソッドのタイミングの有効化 489

メソッド タイミングのプロパティ 489
global.properties ファイルで定義されたプロパティ 490
一般プロパティ 492
クラスおよびメソッド プロパティ 492
ログ プロパティ 494

メッセージの形式 496
現在のメソッド タイミング メッセージの形式 496
呼び出されたメソッド タイミング メッセージの形式 497
タイミング メッセージの書き込み 498
JSP のタイミングの計測 501

第 41 章 デバッグとエラー メッセージング 503

デバッグ 504
JRun によるデバッガの起動 504
スタック トレース 504
コア ダンプの処理 (UNIX システムのみ) 507
メモリ不足エラーの処理 508
クライアント / サーバー間の通信の監視 509
追加デバッグ リンク 514

カスタム エラー メッセージング 515
コネクタによる既定のエラー メッセージの変更 515
web.xml を使用した HTTP エラー ページの設定 516
web.xml を使用した Java 例外メッセージの制御 517

JRun とサードパーティ製 IDE の併用 517

第 42 章 JRun の拡張機能 519

JRun 拡張機能の使用 520

global.jsa ファイルの使用 520
global.jsa ファイルの例 522

サーブレット API の拡張機能 523
allaire.jrun.servlet.JRunResponse クラス 523

第 43 章 JRun と ColdFusion の併用 525

JRun および ColdFusion 526

CFSERVLET の使用 526

CFML テンプレートにおけるサーブレットの呼び出し 526

パラメータと属性が設定されているサーブレットの呼び出し 528

CFOBJECT の使用 532

EJB の公開 532

ColdFusion Administrator における Java 設定の定義 533

CFML ファイルのコーディング 533

索引 537

JRun へようこそ

JRun は完全な Java アプリケーション サーバーで、安全で信頼性のある、拡張可能なサーバー側 J2EE アプリケーションの開発や公開に使用します。JRun では、アプリケーションの開発に使用される最新の業界標準がサポートされています。このアプリケーションは Java サーブレット、JavaServer Pages、Enterprise JavaBeans、および HTML ページなどのスタティック コンテンツ、その他のリソースから構成されます。

JRun でサポートされているプラットフォームには Windows 95/98/NT/2000、UNIX、Solaris、Linux があります。JRun はオープンな設計になっているので、Apache、Microsoft Internet Information Server (IIS)、Microsoft Personal Web Server (PWS)、Netscape Enterprise Server、O'Reilly's WebSite Pro など、さまざまな既存の Web サーバーとともに使用できます。JRun を使用して、ほとんどすべてのプラットフォームに、Web サイトを公開し、動的にコンテンツを生成できます。

この章では、はじめに JRun の主な機能を説明し、次に種々の JRun 製品について、さらに本書が対象とする読者について説明します。また、この章では、JRun アプリケーションの開発に関する関連情報が記載されている種々のリソースについても説明します。

目次

- 製品の機能 xx
- JRun 製品のラインナップ xxi
- 対象読者 xxii
- 開発者リソース xxii
- JRun 文書の概要 xxiii
- その他のリソース xxiv
- お問い合わせ先....................xxv

製品の機能

高度なアプリケーションを構築するために、JRun には次のような機能が用意されています。

- Web 用アプリ ケーショ ンを開発するための J2EE アプリ ケーショ ン標準をサポート します。
- Enterprise JavaBeans (EJB) 1.1 仕様のサポート 。これによってユーザ独自のビジネス ロジックを持つ再利用および拡張可能な Java サーバー コンポーネントを開発し、公開できます。JRun による EJB のサポートには、トランザクション管理、メッセージング、リソース管理、セキュリ ティ、ディストリビューション、ステート管理、およびパーシスタンスが含まれます。
- Java でサーバー側コンポーネント およびカスタム タグ ライブラリ を開発するために、Java サーブレット API 2.2 仕様によって定義されている Java サーブレット をサポートします。
- スクリ プト コード およびユーザ定義のタグ ライブラリ を使用して HTML ページを拡張するために、JavaServer Pages (JSP) 1.1 仕様をサポートします。
- 分散メッセージングをサポートするアプリケーションを公開するために、Java Message Services (JMS) 1.0 仕様をサポートします。
- 複数のアプリケーション コンポーネントを同じトランザクションに含めることができるように、Java Transaction Server (JTA) 1.0 仕様をサポートします。
- Java サーブレット API 2.2 仕様によって定義された Java ベースの Web アプリケーションを開発するための完璧なソリューション。完成した Web アプリ ケーションは、Java サーブレット、JSP ページ、HTML ページのようなスタティック コンテンツ、その他のアプリケーション リソースから構成されることがあります。
- 完全に移植可能な J2EE アプリケーション ソリューション。あるプラットフォームで JRun を使用して作成したアプリ ケーションは、JRun を使用するほかのプラットフォーム上でも使用できます。また、JRun はほかのアプリケーション サーバーのために開発された J2EE 仕様準拠のアプリケーションを実行できます。
- 統合された Java ベースの Web サーバー。これにより、サードパーティ製 Web サーバーを使用しなくても、Web アプリケーションを公開できます。この Web サーバーを RAD 開発ツール、またはシンプルな公開ソリューションとして使用できます。
- JRun アプリ ケーショ ンの開発と 、JRun インスト ールの管理の両方に使用できる開発ツール セット。
- アプリケーションの実行を監視するユーティリティ。これらのユーティリティは、アプリケーションをデバッグする場合や、コード最適化の一部としてアプリケーションの実行におけるボトルネックを識別するために使用できます。

JRun 製品のラインナップ

JRun には、次のエディションがあります。

- JRun Developer 版 アプリケーションの開発とテストを行うことができます。JRun Developer 版では、サーブレット 、JSP、および EJB の 3 つの同時接続がサポートされます。非営利目的で使用する場合には無償ですが、公開するためのライセンスは取得できません。
- JRun Professional 版 サーブレット および JSP を利用する Web アプリ ケーショ ンの開発と 公開を行う こと ができます。JRun Professional 版では、サーブレット および JSP の無制限の同時接続がサポート されます。
- JRun Advanced 版 サーブレットおよび JSP を利用する Web アプリケーションの開発と 公開を行う こと ができます。JRun Professional 版では、サーブレット および JSP の無制限の同時接続がサポート されます。さらに、JRun Advanced 版には Type IV JDBC ドライバ、HTTP ベースのロード バランス、およびサーバーレベルのフェイルオーバ サービスのサポートも用意されています。
- JRun Enterprise 版 エンタープライズ クラスのアプリケーションの開発と公開を行うことができます。JRun Professional 版の全機能に加えて、JRun Enterprise 版では、無制限の EJB 接続がサポートされます。さらに、JRun Enterprise 版には、Type IV JDBCドライバ、HTTP ベースのロード バランス、およびサーバーレ ベルのフェイルオーバ サービスのサポートも用意されています。

さらに、Allaire には、JRun アプリ ケーショ ンを作成するためのビジュアル開発ツール、JRun Studio もあり ます。Studio の直感的に操作できる GUI インターフェイスによって、アプリケーションの構築に必要なツールが利用できます。また、JRun Studio によって、任意の JDBC データ ベースのデータ の選択、挿入、更新、または削除を行う 複雑な SQL ステート メントを作成できます。また、HTTP を通じてリモート サーバー上のデータベースに接続することもできます。これは複雑なネットワーク設定を必要としません。

JRun Studio は JRun Developer 版、JRun Professional 版、JRun Advanced 版、および JRun Enterprise 版とは別売になっています。

JRun および JRun Studio の全バージョンの価格に関する最新情報については、(株)アイ・ティィ・フロンティアにお問い合わせください (株式会社シリウスは、2001 年 4 月に株式会社アイ・ティィ・フロンティアに社名変更いたしました)。Allaire には、www.allaire.com からアクセスできます。

対象読者

本書は、JRun を使用して、Java サーブレット、JavaServer Pages、Enterprise JavaBeans から構成される Web アプリケーションを開発するユーザを対象に記述されています。

開発者リソース

(株) アイ・ティ・フロンティアでは、開発者の教育、テクニカル サポートなどのサービスによりカスタマ サポートを充実させております。以下にご紹介する Web サイトでは、すべてのオンライン リソースにすばやくアクセスできます。

リソース	説明	URL
(株) アイ・ティ・フロンティア JRun のサイト	JRun の詳細な製品情報および関連トピック	http://cfusion.sirius.co.jp/jrun/
JRun サポート フォーラム	Allaire オンライン フォーラムでは豊かな経験を持つ JRun 開発者と連絡をとり、JRun に関連した数多くのトピックについてメッセージを書き込んだり、回答を得ることができます。	http://forums.allaire.com/jrunconf/
開発者コミュニティ	JRun による開発に必要な最先端の情報を提供する、オンライン ディスカッション グループ、知識ベース、技術文書などのあらゆるリソース	www.allaire.com/developer/
JRun 開発者センター	開発のヒント、記事、文書、ホワイト ペーパーに関する情報サイト	www.allaire.com/developer/jrunreferencedesk/

JRun 文書の概要

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

印刷およびオンライン文書セット

JRun の文書セットには、次の文書があり ます。

文書	説明
『JRun セット アップ ガイド 』	JMC を 使用し た JRun のインスト ール、設定、および 管理について説明し ます。
『JRun によるアプリケーシ ョ ンの開発』	Java サーブレ ッ ト、JavaServer Pages、および Enterprise JavaBeans から構成される Web アプリケーションの開発方法について説明します。
『JRun サンプル ガイド 』	サーブレ ッ ト 、JavaServer Pages、Enterprise JavaBeans のコード サンプルおよびサンプル アプリ ケーショ ンを 提供し ます。
『JRun タグ ライブラリ リファレンス』	JRun タグ ライブラリ の JavaServer Pages (JSP) カスタ ム タグについて説明します。
『JRun 拡張設定ガイド 』	ISP、ISV、および OEM カスタ マ用の JRun のインスト ール、使用、設定に関する 情報があり ます。
『JRun JSP クイック リファレンス』	JavaServer Pages (JSP) のディ レクティ ブ、アクショ ン、およびスクリプト要素の簡単な説明と構文が記載されています。
『JRun Version 3.1 機能および移行ガイド 』	JRun バージョン 3.1 の機能と、既存のアプリケーションをバージョン 3.1 に移行する方法について説明します。
『JRun タグ ライブラリ クイック リファレンス』	JRun タグ ライブラリ の JavaServer Pages (JSP) カスタ ム タグの簡単な説明と構文について記載されています。

オンライン文書へのアクセス

すべての JRun 文書は、HTML 形式と Adobe Acrobat ファイル形式でオンラインで利用できます。文書にアクセスするには、JRun を実行している Web サーバーにある URL、*JRun のルート ディレクトリ*/docs/dochome.htm を開きます。

その他のリソース

本書で扱っているトピックの詳細については、次のリソースも参照してください。

書籍

- 『JavaServer Pages Application Development』Scott M. Stirling 他著、Sams 刊、2000 年、ISBN: 067231939X
- 『Java Servlets』Karl Moss 著、McGraw Hill 刊、1999 年、ISBN: 0071351884
- 『Java Servlets: By Example』Alan R. Williamson 著、Manning Publications 刊、1998 年、ISBN: 188477766X
- 『Java Servlet Programming』Jason Hunter、William Crawford 著、O'Reilly & Associates 刊、1998 年、ISBN: 156592391X
- 『Developing Java Servlets』James Goodwill 著、Sams 刊、1999 年、ISBN: 0672316005
- 『Inside Servlets: Server-Side Programming for the Java Platform』Dustin R. Callaway 著、Addison-Wesley Pub.Co. 刊、1999 年、ISBN: 0201379635
- 『Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition』Ed Roman 著、Wiley 刊、ISBN: 0471332291
- 『Enterprise JavaBeans』Richard Monson-Haefel 著、O'Reilly & Associates 刊、ISBN: 1565928695
- 『Enterprise Javabeans: Developing Component-Based Distributed Applications』Thomas C. Valesky 著、Addison Wesley Publishing Company 刊、ISBN: 0201604469

オンライン リソース

- Java servlet API (http://java.sun.com/products/servlet)
- JavaServer Pages (http://java.sun.com/products/jsp)
- Enterprise JavaBeans (http://java.sun.com/products/ejb)
- JSP Resource Index (http://www.jspin.com)
- Servlet Source (http://www.servletsource.com)
- ServerPages.com (http://www.serverpages.com)

お問い合わせ先

販売元

株式会社アイ・ティ・フロンティア
シリウス事業部

電話 : 03-5562-4099
Fax : 03-5562-4070

http://cfusion.sirius.co.jp/jrun/
E-mail : jrunsales@sirius.co.jp

(株式会社シリウスは、2001 年 4 月に株式会社アイ・ティ・フロンティアに社名変更いたしました)

テクニカルサポート

Allaire 社では、電話および Web による幅広いサポート オプションを提供しています。テクニカル サポート サービスについては、http://www.allaire.com/support/ をご覧ください。

JRun サポート フォーラム (http://forums.allaire.com) へは、いつでも投稿できます。

第 1 部

はじめに

ここでは、JRun の概要について説明します。

JRun の概要 3
JRun プログラミング モデル 15
サーブレットの使用 33
EJB の概要 47
Web アプリケーションの開発 57
JRun によるサーブレットへの要求のマッピング 79

第 1 章

JRun の概要

この章では、JRun および JRun アーキテクチャ モデルの概要について説明します。また、アプリケーションの開発および公開時に使用する、JRun のさまざまな機能およびツールについても説明します。

さらに、この章ではさまざまなタイプの JRun ユーザについて説明し、各タイプのユーザが追加情報を見つけられるように JRun の文書内の参照先を記載します。

目次

- JRun について 4
- JRun アーキテクチャ モデル 5
- JRun の機能 8
- 後続の章について 13

JRun について

JRun は Web サーバーを拡張して、Java サーブレット 、JavaServer Pages (JSP)、および Enterprise JavaBeans (EJB) を含む J2EE アプリケーションの開発および公開を可能にします。JRun では、Netscape 用の Netscape Server API (NSAPI)、Microsoft IIS 用の Internet Server API (ISAPI)、および Apache Web サーバー用のプラグイン API など、各 Web サーバーに固有のプラグイン メカニズムを使用して Web サーバーに接続します。

JRun は独自の API ではなく最新の業界標準を使用するように設計されているため、使用する Web サーバーですでに Java サーブレット、JSP、または EJB がサポートされている場合でも JRun は効果を発揮します。

また、JRun によってその他のサーバー側スクリプト テクノロジのサポートも追加されます。JRun は、ECMAScript の完全なサポートを含むサーバー側 JavaScript、Server Side Includes (SSI)、およびプレゼンテーション テンプレートをサポートしています。

サーバー側 Java の利点

Java サーブレット、JSP、EJB は、すべてサーバー側 Java の例です。

サーバー側 Java には、Web サーバー アプリケーションの開発に非常に役立つ次のような重要な機能が数多くあります。

- **Java 機能の一貫性**　サーバー側 Java を使用すると、アプリケーションで必要な Java の機能をサーバーでサポートできます。
- **最新の Java テクノロジ**　Java 規格の進化に伴い、サーバー側アプリケーションはクライアントの準拠レベルにかかわらず、新しい Java 機能をすぐに利用できます。
- **Java Virtual Machine (JVM) の制御**　サーバーはアプリケーションが必要とする JVM を実行します。

サーバー側 Java は、「Write Once, Run Anywhere (一度記述すればどこででも実行可能)」という Java の目的を完全に実現します。制限や拘束を受けることなく、言語のすべての機能を使用できます。Microsoft の J/Direct テクノロジや Sun の RMI テクノロジを使用する場合、これらのアプリケーションを実行するためのクライアントの能力を心配する必要はありません。これらのアプリケーションは Web サーバー上で実行されるため、プラットフォーム、オペレーティング システム、およびアプリケーションを実行する環境のあらゆる機能を制御することもできます。

Web サーバー上の Java は、アプリケーション開発のための真の統合ソリューションです。使用する環境が完全に制御されるため、あるプラットフォームで開発したアプリケーションを別のプラットフォームに公開して、アプリケーションを両方のプラットフォームで正常に実行できます。

Java を使用する利点

J2EE アプリ ケーションの最も 重要な利点は、Java プログラミ ング言語で実装される こ とです。J2EE アプリケーションは Java 固有の移植性を利用して、JRun でサポー ト されているすべての Web サーバーおよびサーバー プラッ ト フォームで実行できます。

Java には、アプリケーション プログラマにとって、次のような多数の利点があります。

- アプリケーションの移植性
- オブジェク ト指向プログラミング
- 標準 API の豊富なセッ ト
- マルチスレッドのサポート
- 自動ガーベッジ コレクション

アプリ ケーショ ンを Java で開発するため、Java プログラミ ング言語の利点も 同時に利用できます。

JRun アーキテクチャ モデル

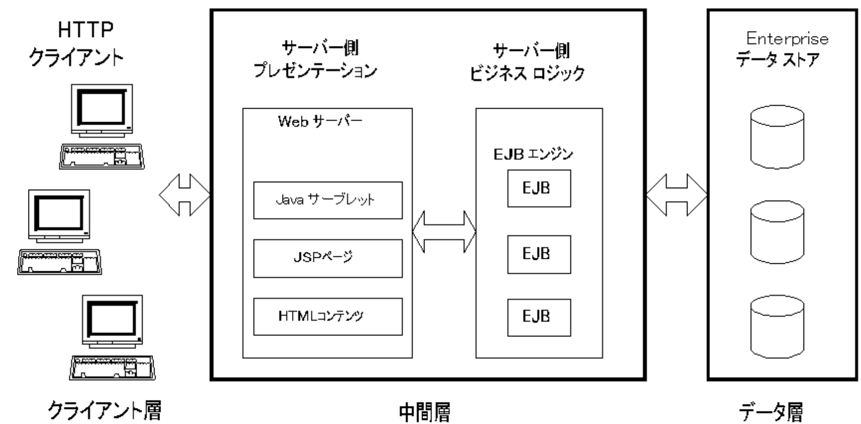
Web でのアプリケーション開発の標準は、J2EE 仕様に基づいています。JRun は、J2EE アプリ ケーショ ン モデルをサポート し 、J2EE アプリ ケーショ ンを実行するための実行時環境を提供し ます。

J2EE アプリ ケーショ ン アーキテク チャは、3 階層アプリ ケーショ ン モデルをサポー ト しています。このモデルはビジネス Web サイ トの機能を、互いに独立して実装される別個のコンポーネントに分割し ます。

3 階層モデルの階層には、次のような特性があります。

- **クライアント層** ブラウザを使用してインターネットなどの HTTP 接続で中間層にアクセスする Web クライアント。この層には、クライアントのコンピュータ上で実行されるアプレッ トが含まれます。
- **中間層** Web サイト のビジネス ロジック。この層には、プレゼンテーショ ン ロ ジックと Web サイトを定義するビジネス ルールの両方が含まれます。JRun を使用して、中間層にアプリケーションを実装し ます。
- **データ層** Web サイトのビジネス データが含まれる Enterprise データ ストア。

次の図は、この 3 階層モデルを示します。



この 3 階層アーキテクチャは、Web サイト開発者にとって次のようなメリットがあります。

- 階層または階層のコンポーネントを複数のハードウェア システムに分散して、システムの拡張性とパフォーマンスを改善できます。
- 中間層は、Enterprise データ ストアへのアクセスの複雑性からクライアントを保護します。
- Java サーブレット API は、Java サーブレット、JSP、HTML ページなどのスタティック コンテンツ、その他のアプリケーション リソースから構成されるように Web アプリケーションを定義します。JRun は、Web サーバーで Web アプリケーションを処理できるようにします。
- Enterprise JavaBeans は、Enterprise データへのアクセスを共有するために複数のアプリケーションが再利用できる安全なコンポーネント ソリューションを提供します。Web アプリケーションは EJB にアクセスできます。または、Web クライアントがこれらに直接アクセスできます。
- コンポーネント アーキテクチャでは、アプリケーション開発を開発グループに分散できます。たとえば、JSP 開発者は通常、ビジネス ルールの実装ではなくプレゼンテーション情報にかかわります。一方、EJB 開発者はデータのアクセスと操作にかかわりますが、プレゼンテーションにはかかわりません。

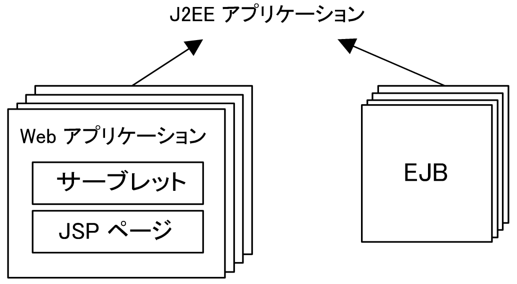
JRun の 3 階層モデルのサポート

JRun を使用して、3 階層モデルの中間層に J2EE アプリ ケーショ ンを実装し ます。JRun は、次のコンポーネントで構成される Web アプリケーションを開発するための最新の業界標準を完全にサポートしています。

- **Java サーブレット** Java サーブレットは、Web サーバーにカスタム機能を追加するのを可能にする Java 言語で書かれたサーバー側コンポーネントです。サーブレットは、HTTP 要求 / 応答モデルをサポートしており、データのアクセスおよび操作に使用される EJB コンポーネントにアクセスできるため、Web ベースのアプリケーションに適合しています。Java サーブレットの作成方法の詳細については、第 3 章を参照してください。
- **JavaServer Pages** JSP によって、HTML とスクリプト コード の組み合わせを含むテキスト ファイルからサーブレットを作成できます。クライアントにより JSP が要求されると、そのページが Java サーブレットに変換されます。JSP のスクリプト 部によって、クライアント にダイナミック コンテンツを返すことができます。また、JSP から Java サーブレット、カスタム タグ ライブラリ、および EJB にアクセスできます。JSP の作成方法の詳細については、第 3 章を参照してください。
- **Enterprise JavaBeans** EJB は、ビジネス ロジックを含む再利用可能な Java コンポーネントの開発および公開を可能にします。EJB 1.1 の仕様は、ソフトウェア コンポーネント モデルを定義します。これにより、EJB 対応のアプリケーション サーバーを使用して、サーバー側アプリ ケーション ロジック (Bean) を公開できます。JRun の EJB サポートには、Bean アクセスを制御するためのセキュリティ サービスとトランザクション、オブジェクトのパーシスタンスが含まれます。EJB の作成方法の詳細については、第 4 章を参照してください。

JRun を使用すると、EJB として実装されたビジネス ロジックを Web アプリ ケーション として実装されたプレゼンテーション ロジックと結合し、単一の J2EE アプリケーションにすることができます。Web アプリケーションは、ダイナミック コンテンツを Web クライアントに配信するために、Java サーブレットおよび JSP を使用して EJB に含まれるビジネス ロジックにアクセスします。

次の図は、J2EE アプリケーションのコンポーネントを示します。



JRun の機能

J2EE アプリケーションを開発および公開する機能のほかに、JRun には次のようなさまざまな機能があります。

- 「Web サーバーへのプラグイン接続」8 ページ
- 「拡張性」8 ページ
- 「セキュリティ」9 ページ
- 「セッション トラッキング」10 ページ
- 「監視ユーティリティ」10 ページ
- 「開発ツール」11 ページ

Web サーバーへのプラグイン接続

J2EE アプリケーションを処理するために、Web サーバーは JRun と通信するクライアントとして機能します。このため、サーバーは JRun との接続を確立させる必要があります。JRun は、この接続を行う ネイティブ サーバー接続モジュールを提供します。

ネイティブ サーバー接続モジュール、すなわちコネクタは、特定の Web サーバー、ハードウェア アーキテクチャ、およびオペレーティング システム用にコンパイルされます。たとえば、JRun は NSAPI を使用して、各ハードウェア アーキテクチャおよび JRun でサポートされているオペレーティング システムに対して、Netscape Application Server 用のコネクタを作成します。

JRun と Web サーバーとの接続の設定については、『JRun セットアップ ガイド』を参照してください。

拡張性

ハードウェアの処理能力を十分に活用するように JRun を設定できます。JRun は、次のようなさまざまな種類の拡張性をサポートしています。

- **単一サーバーの拡張性**　この事例では、サーバーに 1 つまたは複数の CPU が搭載されている場合に、JRun を単一サーバーの処理能力に合わせます。拡張性を制御する唯一の方法は、JRun がクライアント 要求を処理するために割り 当てるスレッド数を制御することです。

  JRun は、Java スレッド メカニズムを使用して、複数の要求を同時に処理します。JRun は各要求に対して新しいスレッドを作成する代わりに、新しい要求のために用意されたハンドラ スレッドのプールを維持します。このスレッドのプールは、Web サーバーに対してさまざまな要求が行われるたびに、拡大したり縮小します。プール用のパラメータは、トラフィックの負荷と Web サーバーの機能のバランスを最適に保ちます。JRun スレッド メカニズムの制御方法の詳細については、『JRun セットアップ ガイド』を参照してください。

- **ネットワークの拡張性**　JRun Enterprise には、Allaire ClusterCATS が含まれています。ClusterCATS は、Web サーバーと JRun サーバーの高度な有効性を支える HTTP ベースのロード バランスおよびフェイルオーバのサービス機能を提供します。ClusterCATS を使用すると、分散型サーバーをパフォーマンスの高い 1 つの Web サーバー リソース環境にまとめることができます。

  ネットワーク上の複数の Web サーバーをクラスタとして構成できます。クラスタに含まれている Web サーバーは、1 つのエンティティとして動作し、その Web サーバー上のリソースへの高速で確実なアクセスが可能です。クラスタは、サーバーのビジー状態や障害によってネットワークの速度が低下する危険性の回避に役立ちます。ClusterCATS を使用すると、帯域幅、応答の遅れ、およびサーバーの輻輳などの問題を回避できます。

  ClusterCATS の詳細については『 Allaire ClusterCATS の使用』を参照してください。

セキュリティ

Web サイトは、不正なアクセスから保護する必要があります。JRun は、さまざまなセキュリティ レベルをサポートしています。

- **JRun 管理セキュリティ**　JRun は、ユーザ名 / パスワード ベースのセキュリティ モデルを提供して、JRun を管理するために使用される JRun 管理コンソール (JMC) へのアクセスを制御します。JMC セキュリティを使用すると、JRun 管理者は組織の開発者ごとにアクセス権を定義できます。この場合、開発者は JRun の特定の領域の管理設定を変更できますが、JRun インストール全体の管理設定は変更できません。

  JMC の使用方法の詳細については、『JRun セットアップ ガイド』を参照してください。

- **Web アプリケーションセキュリティ**　Web アプリケーションに関するセキュリティ問題に対処するため、Java サーブレット API バージョン 2.2 の仕様書では、Web アプリケーション内のリソースへのクライアント アクセスを制御するための認証メカニズムが定義されています。Web アプリケーション セキュリティでは、承認された Web クライアントのみが Web サイト上のリソースにアクセスできます。

  JRun は、Java サーブレット API 2.2 のセキュリティをサポートしています。Web アプリケーション セキュリティの詳細については、第 39 章を参照してください。

- **EJB セキュリティ**　JRun は、EJB セキュリティの実装に関して設定可能なメカニズムを提供します。各 Bean およびそのメソッドのアクセスにより、Access Control List (ACL) を使用してアクセスを許可されたユーザまたはロールを指定できます。公開された各 Bean には、ACL が関連付けられます。

  確認するユーザ情報の種類と内容、ロールを構成する情報、および認証の方法などを指定できます。このような柔軟性により、すでに設置されているセキュリティ方式に対応できます。

  EJB セキュリティの詳細については、第 4 章を参照してください。

セッション トラッキング

HTTP プロトコルはステートレスです。つまり、Web サーバーには複数の要求 / 応答を通してクライアントをトラッキングできません。ただし、JRun はクッキーを使用して、Web アプリケーションへの複数の要求に対してクライアントを追跡するために Web アプリケーションに情報を格納できるようにする、セッショントラッキング メカニズムをサポートしています。クライアントが Web アプリケーション内で別のページに移動しても、情報は破棄されず、ユーザ セッションが継続している間、保持されます。

JRun はまた、ステートフル EJB を使用してセッションのトラッキングをサポートします。ステートフル EJB は、複数の HTTP 要求と複数のメソッド呼び出し全体でクライアントの情報を保持します。

セッション トラッキングには次のような利点があります。

- アクセス制御を実装する Web サイトの場合、複数の要求に対してクライアントを追跡すると、クライアントが要求のたびにログインし直す必要がなくなります。
- クライアントの名前を表示した個人的なあいさつ文を各ページに追加できます。
- クライアントをショッピング カートに関連付けて、クライアントが購入する商品を追跡できます。

JRun セッションのトラッキング メカニズムでは、単一の Web アプリケーションへの複数の要求に対してのみクライアントを追跡できます。複数の Web アプリケーションへの要求に対するクライアント情報を追跡するには、EJB を使用してください。

セッション トラッキングを有効にする方法の詳細については、『JRun セットアップ ガイド』を参照してください。

監視ユーティリティ

JRun には、アプリケーションの実行を監視するためのさまざまなユーティリティが付属しています。これらのユーティリティは、アプリケーションをデバッグする場合や、アプリケーションの最適化の一環として実行時のボトルネックを識別するために使用できます。

JRun 監視ユーティリティには、次のツールが含まれます。

- **ロギング**　アプリケーションを実行している間、JRun から情報、警告、エラー、デバッグ メッセージなどが出力されます。JRun ロギング ユーティリティは、アプリケーションから、ファイル、画面、電子メール メッセージなど、数種類の出力先にこれらのメッセージを転送できます。ロギングの詳細については、第 38 章を参照してください。
- **メソッド タイミング**　メソッド タイミングでは、サーブレットのメソッドとサーブレットのメソッドから呼び出されるメソッドに関する実行時間を記録することができます。これらの実行時間は、アプリケーションのボトルネックを識別するために役に立ちます。メソッド タイミングの詳細については、第 40 章を参照してください。

- **接続ステータス**　JRun では、JRun サーバーと Web サーバーの接続に関するステータス情報が、JRun ログ ファイルに書き込まれます。このステータス情報は JRun サーバーとサードパーティ製 Web サーバーまたは JRun Web サーバー (JWS) のいずれかの接続から取得できます。接続ステータス情報の詳細については、第 37 章を参照してください。

開発ツール

JRun のさまざまな開発ツールには、JRun にバンドルされているものと、アドオン ツールとして別個に購入できるものがあります。

JRun Studio

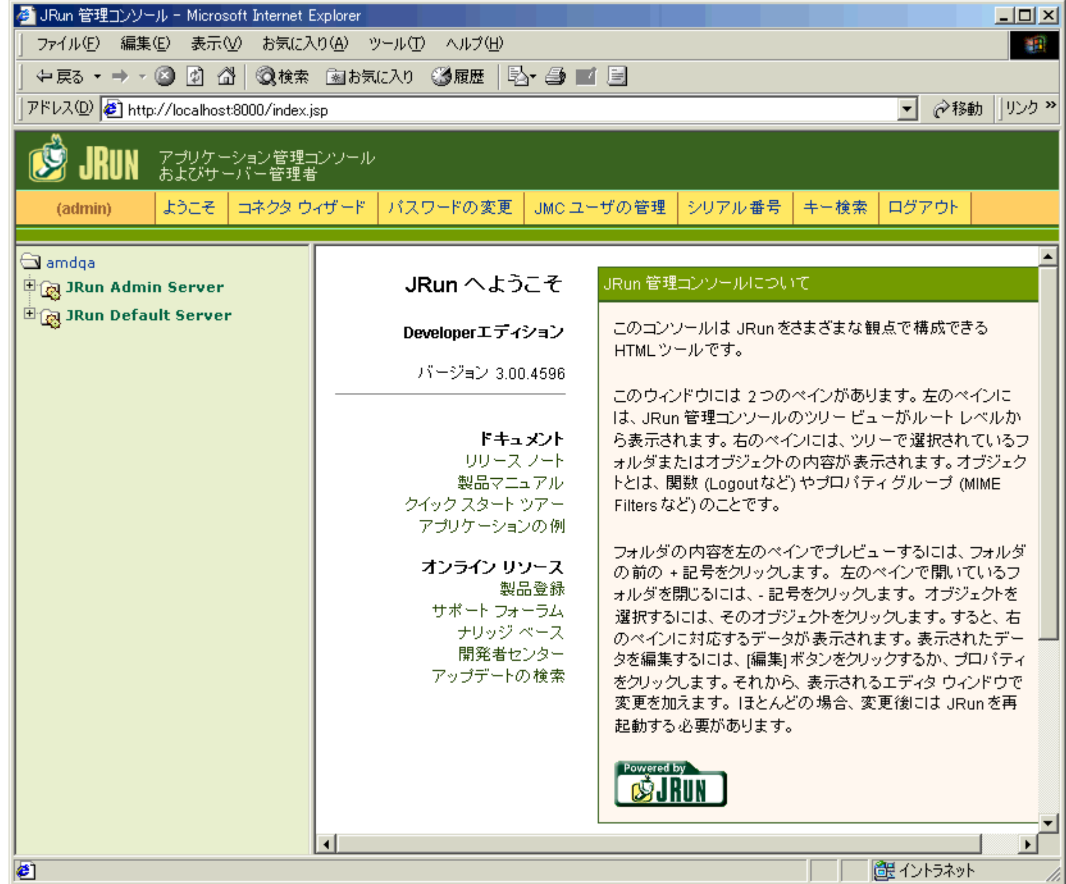
JRun Studio は、JRun アプリケーションを作成するためのビジュアル開発ツールです。この製品は、JRun とは別に販売されています。JRun Studio は、次のような強力なプログラミング ツールによって開発者の生産性を高めます。

- **ビジュアル プログラミング**　ソース編集に透過的に統合される強力な双方向ビジュアル プログラミング ツールにより、開発を促進します。
- **高度なデバッグ ツール**　高度なデバッグ ツールを使用して JRun アプリケーションのデバッグを支援します。
- **HTML デザイン ツール**　数々の賞を受賞した HTML デザイン ツール、Macromedia HomeSite のすべての機能が含まれています。
- **Deploy ツール**　JRun アプリケーションの公開を容易にするユーティリティが含まれています。

Studio の直観的に操作できる GUI インターフェイスによって、アプリケーションの構築に必要なツールが利用できます。また、JRun Studio によって、任意の JDBC データベースのデータの選択、挿入、更新、または削除を行う複雑な SQL ステートメントを作成できます。また、HTTP を通じてリモート サーバー上のデータベースに接続できます。これは複雑なネットワーク設定を必要としません。

JRun 管理コンソール

JRun 管理コンソール (JMC) は、JRun を設定するための Web ベースのツールです。JMC を使用するには、Netscape Navigator または Microsoft Internet Explorer のバージョン 4.0 以降が必要です。次の図は、JMC を示します。



後続の章について

後続の章では、JRunの 一般的な概要について説明しています。これらの章の内容は次のとおりです。

- 第 2 章「JRun プログラミング モデル」15 ページ
- 第 3 章「サーブレットの使用」33 ページ
- 第 4 章「EJB の概要」47 ページ
- 第 5 章「Web アプリケーションの開発」57 ページ
- 第 6 章「JRun によるサーブレットへの要求のマッピング」79 ページ

JRun のユーザはすべて、これらの章の情報をよく理解しておく必要があります。

この後どの章へ進むかは、JRun アプリケーションの開発と公開のための役割によって異なります。

- **システム管理者**　JRun のインストールと管理、JRun の起動と終了、アプリケーションの追加と削除を行います。詳細については、『JRun セットアップ ガイド』を参照してください。
- **JSP 開発者**　クライアントに返されるダイナミック コンテンツを生成する JSP を作成します。これらの JSP は、Java サーブレット、カスタム タグ ライブラリ、および JavaBeans を参照できます。JSP の開発方法については、第 7 章を参照してください。
- **Java サーブレットおよびタグ ライブラリの開発者**　Java でサーブレットを開発し、JSP ページで使用するカスタム タグ ライブラリ を開発します。詳細については、第 17 章を参照してください。
- **EJB 開発者**　Java サーブレット開発者および JSP 開発者が使用する再利用可能なコンポーネントを作成します。EJB 開発の詳細については、第 24 章を参照してください。
- **アプリケーションの公開者**　公開と配布のために JRun アプリケーションのパッケージ化を行います。アプリケーション公開の詳細については、第 34 章を参照してください。
- **すべての JRun 開発者**　アプリケーションの監視、デバッグ、認証を含む一般的な開発タスクをよく理解している必要があります。詳細については、第 37 章を参照してください。

第 2 章

JRun プログラミング モデル

JRun は、J2EE アプリケーションの開発と公開のための完全な Java アプリケーションサーバーです。この章では、アプリ ケーションを開発するための、JRun プログラミングモデルについて説明し、主要な JRun コンポーネントが Web サーバーに統合される方法について記述します。

目次

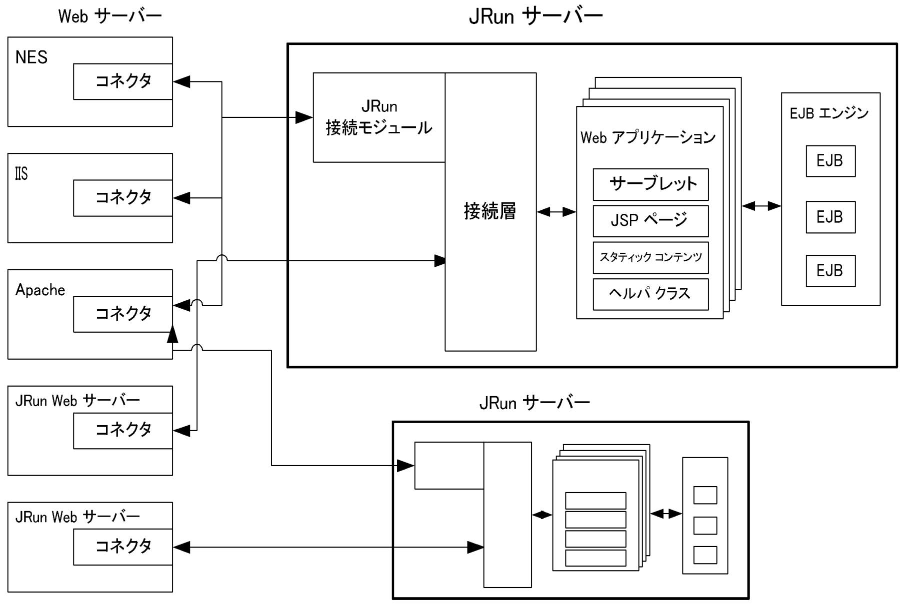
- JRun プログラミング環境 16
- JRun サーバー 18
- Web サーバー 22
- Web アプリケーション 24
- Enterprise JavaBeans 26
- JRun の設定 27
- JRun と Java IDE の併用 29

JRun プログラミング環境

JRun では、Java サーブレット、JavaServer Pages、および Enterprise JavaBeans を組み込んだダイナミック J2EE アプリケーションを開発できます。作成したアプリケーションは、IIS や Apache などのサード パーティの Web サーバー、または JRun に統合された JRun Web サーバー (JWS) からホストできます。

JRun を使用すると、EJB として実装されたビジネス ロジックを Web アプリ ケーション として実装されたプレゼンテーション ロジックに結合できます。Web アプリケーションは、ダイナミック コンテンツを Web クライアント に配信するために、Java サーブレットおよび JSP を使用して EJB に含まれるビジネス ロジックにアクセスします。

次の図は、アプリケーションの開発と公開のために JRun を使用したシステムを示します。



この図は、JRun を実行するシステムの 4 つの主要なコンポーネントを示します。

- **JRun サーバー**　Web サーバーが Java サーブレット、JSP、および EJB を含む J2EE アプリケーションを処理するために必要なサービスを提供します。1 つのシステムで複数の JRun サーバーを作成できます。

  JRun のインストールおよび設定の一部として、サーバー内の JRun 接続モジュールと通信するように Web サーバーのコネクタを設定します。

- **Web サーバー**　クライアントの要求を受信して、Web コンテンツが含まれた応答を配信します。このコンテンツは、スタティック Web ページの場合と、JRun が処理する Java サーブレット や JSP によって生成されるダイナミック Web ページの場合があります。1 つの JRun サーバーに 1 つまたは複数の Web サーバーを接続できます。また、逆に 1 つの Web サーバーに 1 つまたは複数の JRun サーバーを接続できます。

  前の図は、JRun とともに使用できる Web サーバーの例を示します。サポートされているすべての Web サーバーの一覧については、『JRun セットアップ ガイド』を参照してください。

  JRun には、独自の Web サーバーである JRun Web サーバー (JWS) も組み込まれています。JWS は、すべて Java で構成された、高速で軽量な Web サーバーです。これにより、サードパーティ製 Web サーバーをインストールしたり設定することなく、Web アプリケーションの開発およびテストを行うことができます。

- **Web アプリケーション**　Java サーブレット API バージョン 2.2 の仕様書では、Web アプリケーションを Java サーブレット、JSP、HTML ページなどのスタティック コンテンツ、およびほかのアプリ ケーション リソースから生成されたものとして定義しています。JRun サーバーは、異なる URL にマッピングされる複数の Web アプリケーションをサポートします。
- **EJB**　JRun では、EJB コンポーネントのための実行時環境を提供しています。EJB 1.0 仕様では、ソフトウェア コンポーネント モデルを定義しています。これにより、EJB 対応のアプリケーション サーバーを使用して、サーバー側アプリケーション ロジック (Bean) を公開できます。

  JRun EJB エンジンは、コンポーネント のライフサイクル、ネーミング、トランザクション管理、メッセージング、リソース管理、セキュリティ、ディストリビューション、ステート 管理、およびパーシスタンスなどのミドルウェア サービスの自動化を提供します。

次のセクションでは、これらのコンポーネントについて詳しく説明します。

JRun サーバー

JRun サーバーは、Web サーバーが Java サーブレット、JSP、および EJB を含む J2EE アプリケーションを処理するために必要なサービスを提供します。JRun サーバーは、Web サーバー プロセスの外部で独自のプロセスで実行されます。JRun サーバーを別個のプロセスで実行することには、いくつかの利点があります。

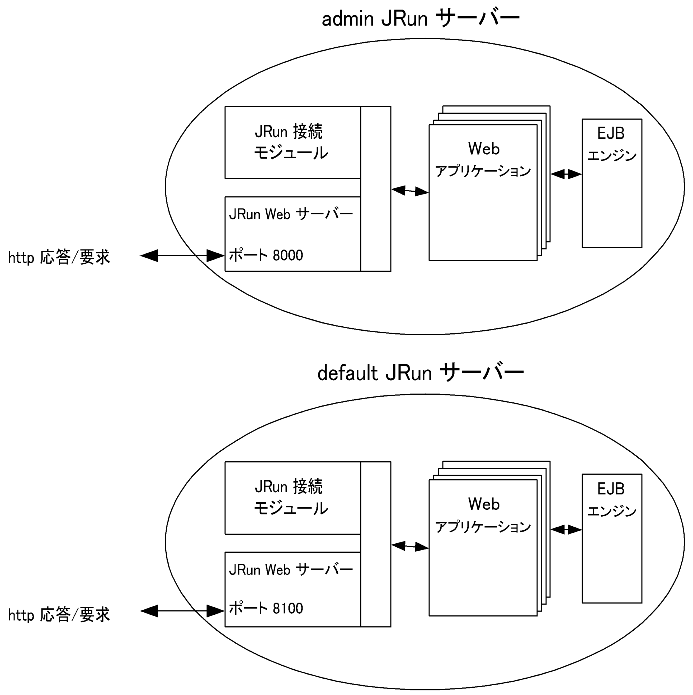
- Web サーバーの安定性の向上
- Web サーバーを JRun と関係なく起動および終了する機能
- Web サーバーを再起動せずにアプリケーションを変更する機能
- 単一の JRun サーバーが複数の Web サーバーと通信する機能

JRun では 1 つのインストールによって複数の JRun サーバーをサポートできます。複数のサーバーを作成する理由の 1 つは、アプリケーションをコンピュータ上の異なるプロセス内に隔離するためです。たとえば、JRun サーバー内のすべてのアプリケーションは 1 つのプロセスで実行します。アプリケーションを複数の JRun サーバーに分離することによって異なるプロセスを使用し、アプリケーションが別のアプリケーションに悪影響を与えないようにすることができます。

アプリケーションをそれぞれ異なる JRun サーバーで実行するもう 1 つの理由は、各 JRun サーバーが独自のユーザ認証メカニズムまたは一連のユーザ認証ルールを実装できるようにするためです。異なる JRun サーバーでアプリケーションを実行することによって、特定のサーバーの認証設定を利用できます。認証の詳細については、第 39 章を参照してください。

インストール済みの JRun サーバー

インストール中に、JRun は admin JRun サーバーと default JRun サーバーという 2 つのサーバーを作成します。また、各 JRun サーバーごとに JWS が作成されます。次の図は、この構成を示します。



この図は、JRun のプロセス ビュー、言い換えると、各 JRun サーバー プロセスの内部で実行されるサービスを示しているため、前の JRun システムの図とは異なります。図のように、各サーバーに関連付けられている JWS は、そのサーバーのプロセス内で実行されます。

admin サーバーは、JRun 管理コンソール (JMC) を含む JRun に付属するすべての管理アプリケーションを実行します。このサーバーは、アプリケーションの開発用に使用しないでください。

Web アプリケーションの場合と同様にアプリケーション リソースへの HTTP 要求を作成して、JMC アプリケーションにアクセスします。したがって、admin サーバーは、関連付けられた JWS を実行してこれらの要求を処理する必要があります。既定では、HTTP 要求に応答する admin に関連した JWS は、8000 より大きいポートを介して受け取ります。このポート番号への要求は次の形式になります。

```
http://localhost:8000/resourceName
```

この例では、*resourceName* は、サーブレット、JSP、HTML ページ、またはその他の Web リソースを表します。たとえば、JRun 管理コンソールに接続するには、次の URL を使用します。

http://localhost:8000/security/login.jsp

default JRun サーバーは、Java サーブレット、JSP 、EJB、Web アプリケーションの開発、テスト、および公開に必要なサービスを提供します。default JRun サーバーは、JRun デモ アプリケーションのホスティングもします。HTTP 要求に応答する default に関連付けられている JWS は、8100 より大きいポートを介して受け取ります。このポート番号への要求は次の形式になります。http://localhost:8100/resourceName

たとえば、JRun デモ アプリケーションに接続するには、次の URL を使用します。http://localhost:8100/demo

既定では、JRun サーバーが起動すると、対応する JWS も起動しますが、JRun サーバーと 通信するよう にサード パーティ製 Web サーバーを設定できます。詳細については、22 ページの「Web サーバー」を参照してください。この場合、JWS を使用して要求を処理する必要はなくなります。このため、あらかじめ JWS を無効にできます。JRun Web サーバーを無効にする方法の詳細については、『JRun セットアップ ガイド』を参照してください。

注意

admin JRun サーバーに関連付けられている JWS のシャットダウンまたは無効化は、ほかの Web サーバーを admin JRun サーバーに接続するように設定してから行ってください。ほかの Web サーバーを設定しないで、JWS のシャット ダウンまたは無効化を行うと、admin JRun サーバーと通信する方法がなくなるため、JRun の管理を実行できなくなります。

JRun サーバーは、Web サーバーが起動された後にのみ要求に応答します。JRun を Windows NT システムにインストールする際に、JRun を Windows NT サービスとして設定できます。Windows NT システムを起動すると、admin サーバーと default サーバーの両方が起動されます。さらに、Windows NT では、[コントロール パネル] の [サービス] から JRun の起動および停止を行うこともできます。Windows 95/98 または UNIX プラットフォームで JRun を実行する場合は、システムを起動した後で各 JRun サーバーを手作業で起動する必要があります。

JRun サーバーの起動および停止については、『 JRun セット アップ ガイド 』を参照してください。

JRun サーバーと Java Virtual Machine (JVM) の併用

Java サーブレット、JSP、および EJB は、すべてサーバー側の Java の例です。JRun サーバーで Java コードを実行するには、JRun サーバーは Java Virtual Machine (JVM) によってロードされる必要があります。JVM は、Java バイト コードを JRun サーバーのホストとなる特定のコンピュータ用のオブジェクト コードに変換します。

各 JRun サーバーには、1 台の JVM が関連付けられます。このため、JRun サーバー内のすべてのサーブレット 、JSP、EJB、および Web アプリ ケーションは、1 台の JVM で実行されます。JVM に関する必要条件は、JVM バージョン 1.1.6 以降の仕様を満たしていることのみです。ただし、EJB を開発する場合はバージョン 1.2 以降が必要です。

メモ

Microsoft Windows バージョンの JRun には、バージョン 1.2 の JVM が付属しているため、JVM を独自に入手する必要はありません。ただし、UNIX バージョンの JRun の場合、JRun と互換性のある JVM を独自に入手する必要があります。互換性のある JVM の詳細については、『JRun セットアップ ガイド』を参照してください。

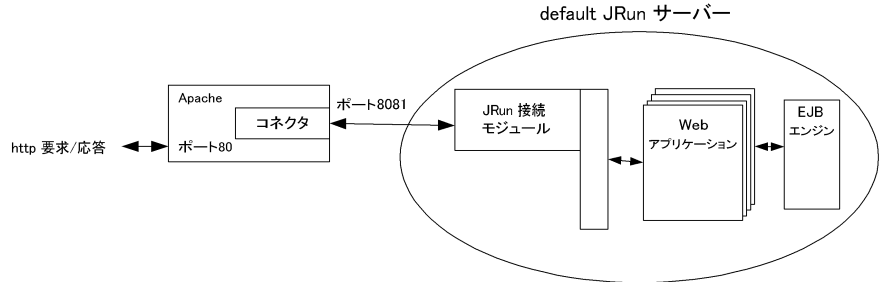
JRun では、使用する JVM のベンダの指定はありません。たとえば、JRun を Microsoft の JVM または Sun の JVM によってロードするように設定できます。JVM の選択の詳細については、『JRun セットアップ ガイド』を参照してください。

Web サーバー

JRun は Web サーバーを拡張して、JRun サーバーがホストとなっている Java サーブレット、JSP、および EJB によって生成されるダイナミック コンテンツを配信する J2EE アプリケーションを処理できるようにします。Web サーバーは JRun と通信するクライアントの役目をするため、サーバーは JRun への接続を確立する必要があります。JRun は、Web サーバーへの接続を作成するネイティブ サーバー接続モジュールを提供します。

ネイティブ サーバー接続モジュール、すなわちコネクタは、特定の Web サーバー、ハードウェア アーキテクチャ、およびオペレーティング システム用にコンパイルされます。たとえば、JRun は NSAPI を使用して、各ハードウェア アーキテクチャおよび JRun でサポートされているオペレーティング システムの組み合わせに対して、Netscape Application Server 用のコネクタを作成します。

各 JRun サーバーに 1 つまたは複数の Web サーバーを接続できます。通常の開発環境では、Web サーバーを default サーバーに接続して、アプリケーションを処理します。次の図は、default サーバーに接続された単一の Web サーバーを示します。



アプリケーション リソースに対して要求が作成されると、Web サーバー上のコネクタにより、JRun サーバー内に常駐する JRun 接続モジュールへのネットワーク接続が開かれます。接続モジュールは、透過コミュニケータの役割を果たし、コネクタから JRun サーバーに要求を変換します。JRun サーバーは要求を処理して、接続モジュール サービスに応答を返します。

各 JRun サーバーは、Web サーバーからの要求に対して異なるネットワーク ポート番号を受信します。上の例では、default JRun サーバーはポート 8081 を受信します。JRun サーバーと Web サーバーの接続を設定するときは、このポート番号を指定する必要があります。

Web サーバーと JRun サーバーの接続を定義するために、2 番目のパラメータであるバインド アドレスを使用することもできます。バインド アドレスは、JRun サーバーが要求を受信するために使用する JRun のホスト コンピュータ上の 各 IP アドレスを指定します。既定では、すべての JRun サーバーのバインド アドレスは * です。これは JRun サーバーがホスト のすべての IP アドレスで要求を受信することを意味します。

メモ
Web サーバーに複数の IP アドレスがあり、JRun サーバーがすべての IP アドレスで要求を受信する場合は、* のバインド アドレスが役に立ちます。

JRun には、Web サーバーと JRun サーバーとの接続を調整するための追加のパラメータがいくつかあります。接続および調整パラメータの詳細については、『JRun セットアップ ガイド』を参照してください。

メモ
JRun には、ユーザの Web サーバーあるいはほかの特殊な場合で使用するための、接続ソース コードが含まれます。このソースコードは、基本的な使用手順とともに *JRun のホーム ディレクトリ*/connectors/src に置かれています。詳細については、『JRun 拡張設定ガイド』を参照してください。これは、Allaire Web サイトから利用できます。

JRun Web サーバー

JRun には、すぐに使用できる Java Web サーバーが用意されています。このため、既存の Web サーバーがない場合でも、Web アプリケーションの開発を開始できます。したがって、開発チームで組み込みの JRun Web サーバー (JWS) を使用して、サーブレットの作成、テスト、およびデバッグを行った後で、互換性のある運用サーバーに公開できます。

ただし、JWS は、セキュリティが組み込まれていない HTTP バージョン 1.0 サーバーです。ほとんどの場合、サード パーティ製 Web サーバーを、現在活動中の Web サイト上にある JRun と組み合わせて使用します。

default JRun サーバーで JWS を開発 Web サーバーとして使用する場合、ドキュメントのルート ディレクトリは次のようになります。

JRun のホーム ディレクトリ/servers/default/default-app

admin サーバーは、アプリケーションの開発用ではなく、主として JRun の管理をサポートするために使用されます。

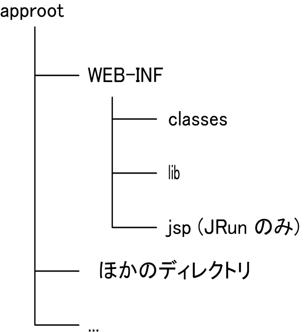
Web アプリケーション

JRun は、完全なスタンドアロン Web アプリケーションの開発、パッケージ化、および公開のためのアプリケーション サーバーです。Java サーブレット API バージョン 2.2 の仕様書で定義されているように、Web アプリケーションは Java サーブレット、JSP、HTML ページのようなスタティック コンテンツ、および Web アプリケーションで必要なその他のリソースの集まりです。

JRun が Web アプリケーションのために、Java サーブレット API 仕様をサポートしているため、JRun を使用した Web アプリケーションの開発と、この仕様をサポートしている Web アプリケーション サーバー上へのそのアプリケーションを公開できます。

Web アプリケーションの設定は、`web.xml` ファイルの内容によって定義されます。このファイルは**公開記述子**とも呼ばれます。このファイルには、アプリケーション サーバーがアプリケーションを実行するために必要なすべての情報が含まれています。`web.xml` の内容は、JRun 固有の情報ではなく、Java サーブレット API バージョン 2.2 の仕様書で定義されています。Java サーブレット API バージョン 2.2 の仕様書で定義された Web アプリケーションをサポートするすべてのプラットフォームは、`web.xml` ファイルの内容を認識し、解釈します。

Web アプリケーションのディレクトリ構造は次のとおりです。



Web アプリケーションの、図の `approot` のようなルート ディレクトリは、Web アプリケーション ファイルを提供するためのドキュメント ルートとして機能します。このディレクトリには、Web アプリケーションの一部として開発された JSP が含まれます。たとえば、`c:/myapp` に配置される Web アプリケーションの場合、既定のトップ ページ ファイルの場所は `c:/myapp/index.html` になります。

実際の Web アプリケーションで使用するのは、上の図に示したディレクトリだけではありません。HTML ファイル、イメージ、その他のアプリケーション リソースなどのために、ほかのディレクトリを追加できます。これらのディレクトリは、クライアントが直接アクセスするリソース用 Web アプリケーションの公開ドキュメント ツリーの一部になります。

approot のサブディレクトリには、次のディレクトリが含まれる場合がありますが、これらに限定されません。

- WEB-INF　アプリケーションのドキュメント ルートに含まれていないアプリケーションに関連するリソースが含まれています。このディレクトリには、アプリケーションの設定情報が格納されたweb.xml ファイルが含まれます。

  このディレクトリは、アプリケーションの公開ドキュメント ツリーの一部ではありません。したがって、このディレクトリまたはそのサブディレクトリに含まれるファイルは、クライアントに直接提供されません。

- WEB-INF/classes　アプリケーションの Java サーブレット用の Java クラスファイルが格納されます。
- WEB-INF/lib　アプリケーション固有のクラスが含まれます。これらのファイルは、Java Archive (JAR) または .zip ファイルに格納されている必要があります。このディレクトリには、タグ ライブラリを格納する JAR ファイルも含まれます。
- WEB-INF/jsp　JSP 変換時に JRun によって生成される .class および .java ファイルが含まれます。このディレクトリは、Web アプリケーション仕様の一部ではなく、JRun によって追加されたものです。

Web アプリケーションを配布する場合は、展開したディレクトリ構造で配布するか、または Web Archive (WAR) ファイルと呼ばれる 1 つの圧縮ファイルで配布することができます。WAR ファイルには、すべてのディレクトリ構造とアプリケーションを定義するすべてのファイルが含まれます。WAR ファイルは、JAR ファイルと同じツールを使用して作成します。

WAR ファイルを使用すると、1 つのファイルで配信が行えるため、アプリケーションの配布が簡単になります。ただし、WAR ファイルに含まれる Web アプリケーションを実行する前に、ファイルを展開する必要があります。WAR ファイルの展開処理は、Web アプリケーション公開の一部になります。JRun にはこの目的に使用する公開ツールがあります。

Web アプリケーションの配布、.war ファイルの作成、および公開ツールの使用の詳細については、第 5 章を参照してください。

Enterprise JavaBeans

JRun は、エンティティとセッションの両方の EJB をサポートしています。このため、ビジネスのニーズを満たす強力なソリューションを開発して公開できます。JRun EJB エンジンは軽量でカスタマイズが可能なため、アプリケーションの特定の要件に合わせて設定できます。JRun EJB の実装により、小さな要領を維持したままで効果的な実行が保証されます。

JRun の EJB サポート は JRun の Web アプリ ケーショ ン サポート とともにインスト ール するか、またはスタンド アロン モード でインスト ールできます。スタンド アロン モード では、Web アプリケーションをサポートするのに必要な追加のオーバヘッド なしで、JRun の EJB 機能が利用できます。

EJB エンジンは、その公開と実行時環境を設定しているプロパティ ファイルによって決まります。プロパティはまた、トランザクション、セキュリティ、パーシスタンス、およびその他のサービスの設定にも使用します。Bean プロパティについて、EJB エンジンは XML ベースの公開記述子によって定義されたプロパティもサポートします。

組み込み Bean を実行するには、Java 2 JRE をインスト ールする必要があります。ただし、Bean を開発する場合や JRun に付属する EJB サンプルを実行する場合は、JDK 1.2 以降のバージョンをインストールする必要があります。

EJB は、JAR ファイルを構築することによって公開されますが、JAR ファイルは、JRun の EJB 公開ツールによって処理されます。EJB の詳細については、第 4 章を参照してください。

Java Message Service

JRun では、Java Message Service (JMS) 1.0 仕様を完全に実装することによって、メッセージング サポートのシームレスな統合を提供します。JRun では、Bean 内のプロデューサ、コンシューマ、および JMS セッションの作成がサポートされています。Bean を非同期メッセージングのリスナとして使用することもできます。メッセージは、EJB エンジンのエンティティ Bean アーキテクチャによって提供されるトランザクション サポートを使用して、完全に処理されます。

JRun は、キュー ベースのポイントツーポイントと、トピック ベースの発行 / 購読の同期メッセージング、および非同期メッセージをサポートしています。メッセージにはパーシスタンスを指定できるので、サーバーのシャットダウン時にメッセージは失われません。

JMS の詳細については、第 30 章を参照してください。

JRun の設定

JRun を設定するには、JMC を使用するか、または JRun プロパティ ファイルを直接編集します。ほとんどの場合は、JMC を使用して設定タスクを実行します。ただし、個々のプロパティ ファイルを編集したり、JMC を配布しないでアプリケーションを配布する場合もあります。このような場合は、プロパティ ファイルを編集して JRun を設定する必要があります。

メモ
プロパティ ファイルはテキスト ファイルであるため、手作業で編集できますが、JMCを使用してシステムを設定してください『 JRun セット アップ ガイド 』の情報の大部分は、JMCによる設定についての説明です。

JMC の使用

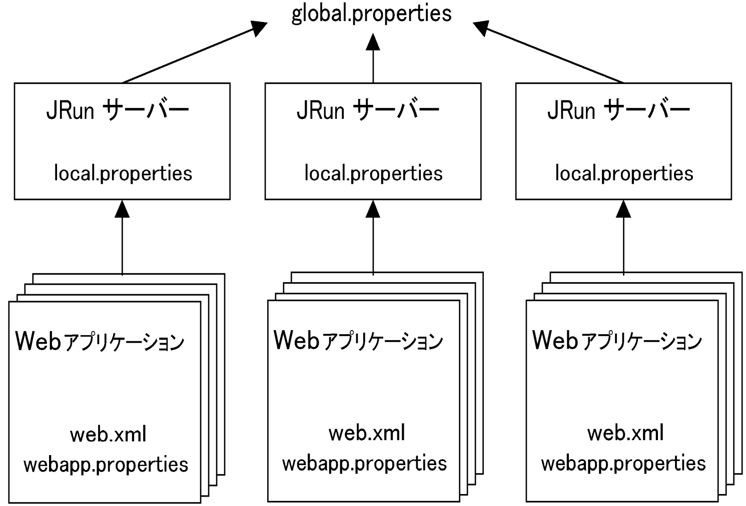
JMC は、JRun を設定するための Web ベースのサーバー管理コンソールです。JMC を使用するには、Netscape Navigator または Internet Explorer のバージョン 4.0 以降が必要です。次の図は、JMC を示します。

このウィンドウには、2 つのペインが表示されます。左ペインには、ルート レベルから始まる JMC ディレクトリ構造のツリー ビューが表示されます。右ペインには、ツリー内で現在選択されているフォルダまたはオブジェクトの内容が表示されます。

JMC の使用方法の詳細については、『JRun セットアップ ガイド』を参照してください。

プロパティ ファイルの使用

JRun プロパティ ファイルは階層で配列され、ルートまたは**グローバル**のプロパティ ファイルによって、すべての JRun サーバーのプロパティが設定されます。次の図は、この階層を示します。



- `global.properties`　JRun インストール全体に関するプロパティ、すべての JRun サーバーに適用されるプロパティ、およびすべての Web アプリケーションに関するプロパティを定義します。このファイルは編集しないでください。このファイルは、*JRun のホーム ディレクトリ*/`lib` にあります。
- `local.properties`　個々の JRun サーバーおよびサーバーで実行されるすべての Web アプリケーションに関するプロパティを定義します。このファイルを直接編集するか、または JMC から編集できます。このファイルの設定は、`global.properties` 内の対応する設定を書き換えます。このファイルは、サーバーのルート ディレクトリに置かれます。たとえば、default JRun サーバーでは、*JRun のホーム ディレクトリ*/`servers/default` です。JRun サーバーを設定する場合は、`local.properties` ファイルをサーバーのルート ディレクトリに含めます。

- `webapp.properties` 特定の Web アプリケーションに関するプロパティをオプションで定義します。通常、このファイルは JMC を使用して編集します。このファイルの設定は、`global.properties` および `local.properties` 内の対応する設定を書き換えます。JRun によりこのファイルが作成されるのは、JMC を使用してアプリケーション特有のプロパティを設定した場合のみです。このファイルは、Web アプリケーションの `WEB-INF` ディレクトリ内にあります。

前の図は、各アプリケーションの `web.xml` ファイル、すなわち公開記述子も示します。Web アプリケーションの開発の一環としてこのファイルの作成と変更をします。JMC によって編集されるものもありますが、ほかの変更についてはユーザが直接ファイルを編集する必要があります。`web.xml` ファイルの情報は、本書内の該当する箇所に記載されています。`web.xml` の構文の詳しい説明については、Java サーブレット API バージョン 2.2 の仕様書を参照してください。

JRun プロパティ ファイルでは、空白文字 (スペースやタブ) が認識されます。これらのファイルに余分な空白文字を入れないようにしてください。たとえば、カンマ区切りリストにあるカンマの後や、行末の改行の前に空白文字が入力されていないことを確認する必要があります。

プロパティ ファイルおよび JRun の設定の詳細については『 JRun セットアップ ガイド 』を参照してください。

JRun と Java IDE の併用

JRun は、Java IDE との統合が可能です。JRun と連動するように IDE を設定する場合は、次の作業を行います。

- JRun サーバーの `main` メソッドを含んでいるクラスを指定します。
- JRun 実行時クラスパスに必要な JAR ファイルを指定します。
- JRun サーバーの `main` メソッドに渡されるコマンドライン引数を指定します。

EJB エンジンをスタンドアロンモードで実行中の場合は、代わりに、EJB エンジンの `main` メソッドを含んでいるクラスを指定し、EJB エンジンに対してコマンドライン `java` 引数を指定します。

各 Java IDE には、特定の統合要件および機能があります。JRun と Java IDE の統合に関する最新情報については、http://allaire.com/Support/KnowledgeBase/SearchForm.cfm にアクセスして、知識ベースの記事番号 14529 をご覧ください。

JRun.main メソッドの指定

IDE では、JRun の `main`メソッドを含んでいる Javaクラスを識別する必要があります。JRun では、`main` メソッドは、*JRun のルート ディレクトリ*`/lib/jrun.jar` に格納されている `JRun` クラスです。パッケージ名はありません。

クラスパスでの JAR ファイルの指定

JRun のクラスパスに指定する JAR ファイルは、J2EE の機能一式を使用しているか、あるいはサーブレット /JSPコンテナのみを使用しているかによって、次のように異なります。

- **J2EE アプリケーション サーバー** クラスパスには、`wddx.jar`、`jsprt.jar`、および `install.jar`を除く *JRun のルート ディレクトリ*`/lib` および *JRun のルート ディレクトリ*`/lib/ext` 内にあるすべてのファイルを含める必要があります。すべての標準 Sun J2EE JARファイルは、*JRun のルート ディレクトリ*`/lib/ext` ディレクトリに格納されています。
- **サーブレット /JSP コンテナ** このコンテナには、次の JAR ファイルが格納されています。
  - `jrun.jar`
  - `jsp.jar`
  - `rhino.jar` (JavaScript を JSP のスクリプト言語として使用する場合のみ)
  - `xt.jar`
  - `ssi.jar` (SSIFilter 機能を使用する場合のみ)
  - `servlet.jar`

どの JAR ファイルを含めたらよいか不明な場合は、*JRun のルート ディレクトリ*`/lib` および *JRun のルート ディレクトリ*`/lib/ext` に格納されているすべての JAR ファイルを含めます。

アプリケーションが使用するすべてのクラス、JAR ファイル、および ZIP ファイルは、クラスパスに追加されます。たとえば、JDBC を使用してアプリケーションをデータベースに接続するには、JDBCドライバの JARファイルをクラスパスに追加する必要があります。アプリケーションが JRun のバージョンとともに分散された Merant データベース ドライバを使用する場合は、必ず *JRun のルート ディレクトリ*`/servers/lib/jrun_drivers.jar` を含めてください。アプリケーションが JRun タグ ライブラリを使用する場合は、必ず *JRun のルート ディレクトリ*`/servers/lib/jruntags.jar` を含めてください。

コマンドライン引数の指定

JRun `main` メソッドに渡されるコマンドライン引数を指定する必要があります。

`-start` *JRun のルート ディレクトリ*`/servers/`*サーバー名* `jrun.rootdir=`*JRun のルート ディレクトリ*

上記の引数は、特定のルート ディレクトリにインストールされている JRun サーバーを起動します。コマンドライン引数の詳細については、『JRun セットアップ ガイド』を参照してください。

メモ

マシンの JRun サーバーを停止してから、IDE 内にある JRun を開始してください。そうしないと、実行中のサーバーとポートが競合するため、JRun は IDE 内で開始されません。

EJB エンジンとの統合

EJB エンジンをスタンドアロン モード で実行中の場合は、EJB エンジンの `main` メソッドを含んでいるクラスを指定し、EJB エンジンに対してコマンドライン `java` 引数を指定します。

メモ
クラスパスに JAR ファイルを指定する場合は、`wddx.jar`、`jsprt.jar` および `install.jar` を除く、*JRun のルート ディレクトリ*`/lib` および *JRun のルート ディレクトリ*`/lib/ext` 内のすべてのファイルを指定します。

スタンドアロン EJB エンジンの場合、main メソッドは、*JRun のルート ディレクトリ*`/lib/ejipt.jar` に格納されている `allaire.ejipt.Ejipt` クラスに含まれています。

スタンドアロン EJB エンジンを使用する場合は、次の `java` コマンドライン引数を指定してください。

```
-Dejipt.classServer.host=127.0.0.1
-Dejipt.classServer.port=2323
-Dejipt.homePort=2333
-Djava.security.policy=JRun のルート ディレクトリ/lib/jrun.policy
-Dejipt.home=JRun のルート ディレクトリ
-Dejipt.ejbDirectory=JRun のルート ディレクトリ/servers/サーバー名
```

上記の引数は、わかりやすくするために個別の行で示されています。これらの引数を IDE に入力するときは、各引数の間にスペースを入れて、すべての引数を同じ行にします。

第 3 章

サーブレットの使用

この章では、Java および JSP を使用したサーブレットの開発について説明します。特に、両方の環境に共通している概念とオブジェクトについて説明します。また、Java サーブレットと JSP のコード例を提示しながら、サーブレット API の違いについても説明します。

目次

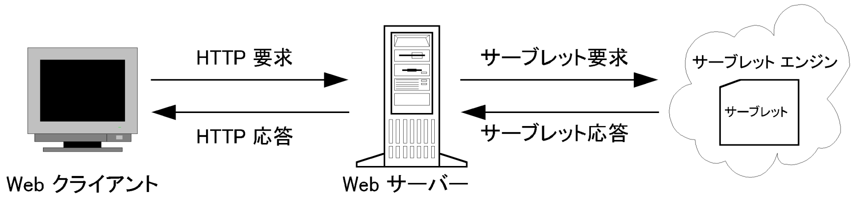
- Java サーブレットの使用 34
- JRun のサーバー側スクリプト ファイル 39
- サーブレットと JSP ページ 40
- Java によるサーブレットの作成 45
- JSP ページとしてのサーブレットの作成 46

Java サーブレットの使用

Java サーブレットは、Web サーバーにカスタム機能を追加できるようにする、Java 言語で書かれたサーバー側コンポーネント です。サーブレット は Web サーバーで動作し、パフォーマンス、データベース接続性、安定性、およびセキュリティの面で高い機能を備えています。

特定の Web サーバーの実装のために C++ や Perl で書かれるケースの多い CGI スクリプトと違い、サーブレットは Java で書かれるため、「Write Once, Run Anywhere (一度記述すればどこででも実行可能)」という Java の利点をはじめ、Java プログラミング言語のすべての利点を活用できます。

サーブレットは HTTP の要求 / 応答プロトコルをサポートしているため、Web ベースのアプリケーションには特に適しています。次の図は、Java サーブレットの基本的な処理モデルを示します。



この図に示すように、HTTP 要求を受け取ると、Web サーバーはその要求がサーブレット を参照しているかどう かを判別し 、適切なサーブレット を呼び出すサーブレット エンジンに転送します。サーブレットは要求を処理し、Web サーバーがクライアントに転送する応答を返します。

サーブレットを使用すると、特定の Web サーバーを実装するための複雑な処理や、プラットフォーム固有の動作にわずらわされることなく、最新のアプリケーションを作成できます。Web アプリケーションの構成要素 (HTML、フォーム データ、要求ヘッダ、クッキーなど) はすべてサーブレット API でサポートされています。

サーブレットの呼び出し

クライアントは URL を Web リソースとして参照し、サーブレットを呼び出します。URL がサーブレットを参照することは、Web クライアントからはわかりません。

通常、クライアントは次のいずれかの方法でサーブレットを呼び出します。

- **JavaServer Pages のアクセス** JSP は、HTML とスクリプト コードの組み合わせで構成されています (通常は Java または JavaScript で書かれています)。クライアントが初めて JSP ファイルにアクセスすると、このファイルは Java ソース コードに変換されます。ソースコード は Java サーブレット にコンパイルされてからロードされ、実行されます。Web サーバーからは、サーブレット のすべての HTML 出力が Web クライアントに戻されます。

  サーブレット の実行可能モジュールがメモリ に常駐しているため、JSP ファイルへのそれ以降のクライアント アクセスは大変効率的に実行されます。これは、JRun がサーブレット イメージをメモリ内で参照できるため、コンパイルとロード手順が省略されることを意味します。

- **サーブレットを呼び出すサーバー側のインクルード ファイル (SHTML) にアクセス** サーバー側のインクルード ファイルは `<servlet>` タグを使用してサーブレット を呼び出すことができます。`<servlet>` タグには、サーブレット に渡されるパラメータがサポートされています。

  JSP ファイルと 同様に、サーブレット はいったんメモリ にロード されると、そのままメモリに常駐します。後続のサーブレットへのアクセスでは、メモリ イメージが参照されます。

- **サーブレットにマッピングされた URL を動的に参照** サーブレットはメモリに常駐し、後続のサーブレットへのアクセスでは、メモリ イメージが参照されます。

これらのメソッドについては、本書の後の章で説明します。

サーブレットの利点

サーブレットには、従来のサーバー側アプリケーション開発技術と比較すると、Web 開発者にとって多数の利点があります。その中には、Java プログラミング言語に関するものもあれば、サーブレット技術に関するものもあります。このセクションでは、サーブレットと Java を使用する場合のいくつかの利点について説明します。

サーブレットを使用した場合の利点

サーブレット技術を使用した場合、Web 開発者にとって次のような利点があります。

- **セキュリティ**　サーブレットは Web サーバーを介して起動するため、ビジネスロジックがクライアントから直接参照されることはありません。さらに、サーブレットは互いに分離されているため、1 つのサーブレットにエラーが起きても、ほかのサーブレットが破損することはありません。
- **パフォーマンス**　CGI などの既存のサーバー側アプリケーションとサーブレットとの最大の違いは、パフォーマンスです。サーブレット は呼び出されたときに一度だけロード されます。サーブレット はメ モリ に常駐し、変更され るまで再びロード されることはありません。サーブレットを変更した場合は、Web サーバーやアプリケーションを再起動せずにロードし直すことができます。

  さらに、サーブレットはマルチスレッド であるため、サーブレット サーバーのプロセス内で実行されます。それぞれのサーブレット要求を処理するのに、プロセス コンテキスト スイッチは必要ありません。
- **移植性**　JRun で実行されるサーブレットは、業界標準のサーブレット仕様に準拠しているため、サーブレット標準をサポートしているか、または JRun を使用している Web サーバーであれば、いずれも 移植できます。移植性はサーブレット ベンダにとって重要です。これは、Web サーバーやサーバー プラットフォームの種類によって、異なるバージョンのサーブレットを管理する必要がないためです。
- **安定性**　サーブレット は Web サーバーのプロ セスの外で実行され ます。したがって、サーブレットでエラーが発生しても、影響を受けるのはサーブレットを実行しているプロ セスだけです。Web サーバーのプロ セスは分離され ているため、影響され ません。
- **ステートの持続性**　静的な情報または持続的な情報は、複数のサーブレット で共有できます。情報は複数のユーザ間、またはセッション内で共有できます。

Java を使用する利点

サーブレットの最も重要な利点は、サーブレットの結果が Java プログラミング言語で実装される点です。サーブレットは、Java 本来の移植性を活用して、すべての Web サーバー、および JRun でサポートされているサーバー プラットフォームで実行できます。

Java には、アプリケーション プログラマにとって、次のような多数の利点があります。

- アプリケーションの移植性
- オブジェクト指向プログラミング
- 簡易化されたプログラミング モデル
- マルチスレッドのサポート
- 自動ガーベッジ コレクション

サーブレットは Java または JavaServer Pages を使用して開発するため、Java プログラミング言語のその他の利点も自動的に付加されます。

サブレットと CGI

Common Gateway Interface (CGI) は、ここ数年、Web サーバーの拡張に使用されるインターフェイスとして主要な位置を占めていました。市場では、すべての Web サーバーに CGI サポートが組み込まれていたため、Web サイトに動的な機能を追加できる開発ツールやアプリケーションに CGI は最も適していました。CGI 言語には、C、C++、および Perl を使用できますが、主流は Perl でした。

そして Java の時代に入ります。まったく新しいネットワーク言語として、Java はインターネットの利用を目的に開発されました。ネットワーク ソケット、データベース接続性、文字列操作など、多数の機能に対するサポート を組み込むことにより 、Java は短期間のうちに最も優れた開発言語として世界中の開発者に受け入れられたのです。それにもかかわらず、Java を CGI 言語として使用することには、まだ問題が残されていました。多く のソリューションは、要求ごとに新しい Java Virtual Machine (JVM) を作成するものであったため、結果としてパフォーマンスの低下を招くことになり ました。Perl インタープリタと同様に、Java も、要求ごとに新しいプロセスをプログラムに作成する必要がありました。

サーブレットは完全に CGI に取って代わります。サーブレットは、開発者にさらに多数の利点をもたらします。これには、開発の簡易化、高速なスループットと応答、サーブレット間通信のほか、Java 特有のすべての機能が含まれています。

また、サーブレットがプラットフォームに依存せず、移植性があるのに対し、CGI プログラムは高い率でプラット フォームに依存します。CGI スクリ プト は通常、特定のハードウェア プラットフォームで実行される特定の Web サーバー用に書かれます。サーブレットの移植性は、複数の Web サーバーやプラットフォーム用のサーブレットを販売するベンダにとって、最も重要な利点といえます。

従来の CGI アプリケーションの主要課題の 1 つは、パフォーマンスです。CGI アプリケーションがクライアントから要求されるたびに、新しいプロセスが作成されます。複数のユーザの要求を処理する人気の高い Web サイト では、この動作がパフォーマンスの問題を引き起こす可能性があります。

その点、サーブレットは、より効率的に要求を処理します。サーブレットは、最初に要求された時点で、Web サーバーのメ モリ 領域にロード されます。したがって、後続のクライアントからの要求は、メモリ内のサーブレット インスタンスを呼び出します。

また、サーブレットはスレッド を使用して複数の要求を同時に処理するのに対し、CGI プログラムには本質的にマルチスレッド機能がありません。

サーブレットの作成

JRun には、サーブレットを作成するための 2 つの方法があります。1 つは Java プログラムを作成する方法で、もう 1 つは JavaServer Pages を作成する方法です。Java プログラムを作成すると、Java のデータ処理機能と利点を十分に活用できます。通常は、Java を使用して、データベースのアクセスなどの複雑なデータ操作を実行するサーブレットを作成します。

また、HTML とスクリプト コードを組み合わせたサーバー側のスクリプトの JSP からもサーブレットを作成できます。JSP ページには、Java プログラミング言語の特性を十分に活用できる機能がありますが、HTML コードに Java コードを組み込むには、簡単なメカニズムを使用します。それらは、クライアントのブラウザに直接返される HTML を生成するサーブレットの実装によく使用されます。

サーブレットを作成するための手段については、どちらも本書で説明しています。

JRun によるサーブレットのサポート

なぜ JRun でサーブレットを実行する必要があるのでしょうか。主な理由の 1 つとして、すべての Web サーバーにサーブレット機能が実装されているわけではないことが挙げられます。JRun により、Web サーバーはサーブレットを処理できるように拡張されます。

Web サーバーにすでにサーブレットの実行機能があったとしても、実装されているサーブレット標準が、そのサーバーやサーバーのホストとなるハードウェア プラットフォームに限定されている場合があります。JRun は、完全に移植性のあるサーブレット ソリューションを提供します。JRun を使用して書かれたサーブレットであれば、JRun を使用しているどの Web サーバーでも、あるいはサーブレット標準を装備しているどの Web サーバーでも使用できます。

JRun には、既存の Web サーバーにアクセスできなくても、サーブレットの開発を始められるように、すぐに使用できる Java Web サーバーが用意されています。この組み込みの JRun Web サーバーを使用して、サーブレットの作成、テスト、デバッグを行ってから、互換性が保証されている実際の運用サーバーに公開できます。

JRun のサーバー側スクリプト ファイル

サーブレット作成をサポートするほかに、JRun はサーバー側のスクリプトを提供します。サーバー側スクリプト ファイルを使用して、スクリプト の出力がクライアント に送信される前に Web サーバーで処理される手順を記述した、Webドキュメント を作成します。このような なサーバー側スクリプト には、アクションを実行したり 、ロジックを Web ドキュメントに追加するためのタブが含まれている場合があります。サーバー側スクリプト は、ブラウザの種類に関係なく 、ダイナミック コンテンツおよびロジックを HTML ベースのページに適用する手段を提供します。

サーバー側ドキュメントは通常、`.html` 以外のファイル拡張子で識別されます。たとえば、`.asp`、`.cfm`、`.shtml`、`.thtml`、`.jsp` などです。この拡張子によって、Web サーバーはどのドキュメントを送信前に処理しなければならないのかを判断します。また、サーバー側ド キュメント の処理を行う コンポーネントも 、拡張子によって識別されます。

JRun にサポートされている JSP 使用すれば、Java コードを書かずにサーブレットを作成できます。JSP から サーブレット を呼び出すことも 、ほかのサーバー側スクリプト から JSP を呼び出すこともできます。

サーバー側スクリプトのタイプ

Web サーバーは、JRun によって、次の数種類のスクリプト技術を処理できるように拡張されます。

- **JavaServer Pages**　Allaire の JavaServer Pages バージョン 1.1 の仕様の実装。JSP によって、HTML とスクリプト コードの組み合わせを含むテキスト ファイルからサーブレットを作成できます。これらのファイルには、拡張子 `.jsp` が付きます。
- **JavaScript** (ECMAScript の完全サポートを含む)　JRun では JSP に対してスクリプト 言語として JavaScript を使用できます。その結果、Java コード を使用しないで JSP を開発できます。
- **サーバー側インクルード拡張機能**　JRun には、サーバー側インクルード ファイルの拡張機能が含まれており、インクルード ファイルからサーブレットを呼び出すことができます。これらのファイルには、拡張子 `.shtml` が付きます。
- **SSI タグレット**　JRun には SSI タグレット がサポートされているため、特定のサーブレット にマッピングされるタグを定義できます。その後、これらのタグレットをスクリプトに使用します。
- **プレゼンテーション テンプレート**　プレゼンテーション テンプレート を使用すれば、一定の外観と使い勝手を HTML アプリケーションに適用できます。テンプレート ファイルに拡張子 `.thtml` を付けます。

本書で、これらのタイプのサーバー側スクリプトについて説明します。

サーバー側スクリプトに関する JRun の機能

JRun のサーバー側スクリプト ファイルには、Java Web サーバーの定義に従ってすべてのページ コンパイル機能と、現在の JavaServer Pages 仕様が完全にサポートされています。JRun には、次のような機能があります。

- JavaServer Pages バージョン 1.1 仕様と 100% の互換性
- JSP の `<jsp:useBean>` タグの完全サポート
- 実際のオブジェクト指向型ページ設計用の拡張 JSP のサポート
- 従属ファイルの再帰的コンパイル機能のサポート
- プレゼンテーション テンプレートのサポート
- すべての Java Virtual Machines (JVM) と Java コンパイラのサポート

さらに、JRun には、サーバー側スクリプト ファイルと Java サーブレットの開発に役立つ例やサンプルが多数用意されています。その他、Allaire の Web サイト http://www.allaire.com/ には、随時最新の例やチュートリアルが掲載されています。

サーブレットと JSP ページ

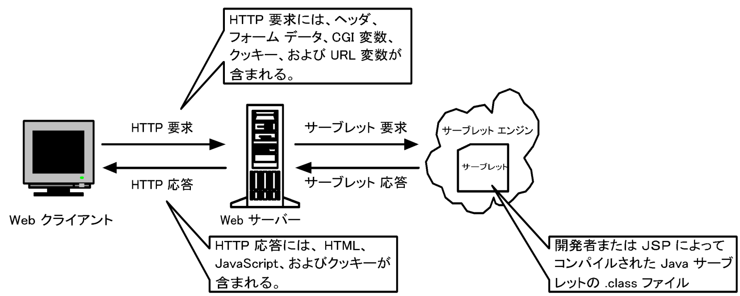
JRun では、Java サーブレットと JSP の両方を使用してサーバー側 Java アプリケーションを作成できます。JSP は JRun によってサーブレットに変換されるため、ネイティブ Java サーブレット に使用できるオブジェクト の多く は、本質的に JSP にも 使用できます。

- HTTP 要求と応答
- 出力
- 例外
- ページ コンテキスト
- セッション
- コンテキスト (アプリケーション)
- 設定情報
- アプリケーションの公開 (例 : WAR ファイル、DTD)

これらのトピックについては、次のページで説明します。上記オブジェクトの使用方法の詳細については、本書の後のセクションで説明します。

HTTP 要求と応答

サーブレットは、HTTP 要求がサーブレットを直接的に Java サーブレットとして参照した場合、または間接的に JSP ファイルとして参照した場合に起動されます。サーブレット内の最も一般的なタスクは、HTTP 要求に格納された情報にアクセスし、その情報を処理してから、結果を HTTP 応答の一部としてクライアントに返すことです。



HTTP 要求には、クライアントからサーブレットに送信された情報が含まれています。たとえば、サーブレット がフォームによって起動された場合は、処理を行う 前に、そのサーブレット は要求に格納されているフォーム データ にアクセスする 必要があり ます。フォーム データ には、ユーザ、データ ベースに書き込まれた登録情報、またはユーザのショッピング カート に追加された製品情報の検証に使われるログイン 情報が含まれています。

HTTP 要求情報には、次のようにアクセスします。

- Java サーブレット では、`javax.servlet.HttpServletRequest` オブジェクト を使用して、HTTP 要求情報にアクセスします。このオブジェクト は、要求内に格納された情報のアクセスに使用できるメソッド を定義します。
- JSP では、暗黙の JSP オブジェクト `request` を使用します。この `request` オブジェクトを使用すると、`javax.servlet.HttpServletRequest` オブジェクトの場合と同じメソッドを使用できます。

サーブレットは、HTTP 応答を構築し、その応答をクライアントに返すことによって、要求に応じます。サーブレット内で、HTTP 応答にアクセスし、クライアントに返された応答内の情報を書き込みます。

HTTP 応答情報には、次のようにアクセスします。

- Java サーブレットでは、`javax.servlet.HttpServletResponse` オブジェクトを使用します。それは、応答内に保存された情報にアクセスするメソッド を定義します。
- JSP では、暗黙の JSP オブジェクト `response` を使用します。この `response` オブジェクト を使用すると、`javax.servlet.HttpServletResponse` オブジェクト の場合と同じメソッドを使用できます。

HTTP 応答には、クライアント への結果の返送に使用する 出力ストリ ームが含まれます。

クライアントへの結果の返送

サーブレット で、要求クライアント にダイナミック コンテンツを返すことができます。出力は、サーブレットが計算する情報またはサーブレットに渡された情報に基づいて生成されます。たとえば、サーブレットは渡されたフォーム属性 (request オブジェクトを介してアクセスされたもの) を使用して、フォーマットされたデータベース データを戻すことができます。一方、アプリ ケーションにユーザ プリ ファレンスを維持するメソッド がある場合は、格納されているプリ ファレンスに基づき、サーブレット でブラウザの表示色を設定できます。

HTTP 応答を使用して、次のように情報を戻します。

- サーブレット は、javax.servlet.HttpServletResponse オブジェクト の PrintWriter、または ServletOutputStream インターフェイスを使用します。これらのインターフェイスには、println メソッドが含まれています。このメソッドは、出力ストリームへの書き込みを行うためのものです。
- JSP には、暗黙の JSP オブジェクト out が使用されます。out オブジェクトにも、println メソッドが含まれています。

例外処理

例外とは、サーブレット内で検出されるエラーのことです。例外は、JRun によって JSP が Java クラス ファイルに変換されるとき、またはサーブレットが実行されるときに発生する可能性があります。

例外は、次のように表現されます。

- Java サーブレットでは、例外はクラス javax.servlet.ServletException のインスタンスで表されます。
- JSP では、exception オブジェクトを使用して例外を表します。

ページ コンテキスト情報の維持

JSP ページの pageContext オブジェクトは、情報を JSP にローカルに格納するメカニズムを提供します。JRun では、ページの要求ごとに新しい pageContext オブジェクトが作成されます。このオブジェクトは、ページが起動すると作成され、ページが終了すると削除されます。pageContext オブジェクト のメソッド を使用すると、JSP の情報にアクセスしたり、ほかのアクションを実行できます。

Java サーブレットには、これに相当するオブジェクトはありません。

セッションの処理

HTTP はステートレス プロトコルです。Web サーバーは、要求を受け取って応答を返すと、クライアントとの接続を終了します。Web サーバーにはクライアントの情報は維持されません。そのため、同じクライアントから別の要求がきても、自動的に判断することはできません。Web サイトからクライアントやナビゲーションを自動追跡できないことが、Web サイトにおける複雑なトランザクションの実行を困難にしています。

ただし、JRun には `session` オブジェクトがサポートされており、これを使用すれば、Web サーバーとの対話全体を通してユーザを追跡できます。`session` オブジェクトで、ショッピングするユーザを追跡し、登録情報や優先情報を送ることができます。サイトに接続するたびに自由に再入力できます。

`session` オブジェクトは、ユーザが Web サイトに接続している間、情報の格納および検索場所を 1 つ提供します。

JRun を使用してセッションを実装する場合は、クライアントのブラウザにクッキーがサポートされ、使用可能になっている必要があります。ブラウザにクッキーがサポートされていない場合、またはクライアントでクッキーが使用不能な場合は、JRun でセッションを追跡することはできません。このような場合は、別のメソッドを使用してユーザを追跡する必要があります。

セッション情報には、次のようにアクセスします。

- Java サーブレットでは、`javax.servlet.http.HttpSession` オブジェクトを使用してセッション情報にアクセスします。
- JSP には、暗黙の JSP オブジェクト `session` が使用されます。

アプリケーション コンテキストの追跡

コンテキスト オブジェクトを使用すると、アプリケーション情報を格納したり、アプリケーションのさまざまなコンポーネント間で情報を共有できます。

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

- 要求に渡される属性
- 初期化パラメータ
- MIME タイプ
- バージョン情報
- パス情報

また、アプリケーション コンテキストには、Web サーバーへのアプリケーションの実装に伴って、アプリケーション情報が格納されます。この情報には、アプリケーション コンポーネントのファイルの位置、サーブレットの初期化パラメータ、アプリケーションのバージョン情報など、アプリケーション固有の情報が含まれています。

アプリケーション情報には、次のようにアクセスします。

- Java サーブレットでは、`javax.servlet.ServletContext` オブジェクトを使用します。
- JSP では、暗黙の JSP オブジェクト `application` を使用します。

設定情報へのアクセス

JRun は、初期化時にサーブレットに設定情報を渡します。設定情報には、初期化パラメータを示す名前 / 値ペア、およびサーブレットが実行されるコンテキストを示す ServletContext オブジェクトが含まれています。

設定情報には、次のようにアクセスします。

- Java サーブレットでは、`javax.servlet.ServletConfig` オブジェクトを使用します。
- JSP では、暗黙の JSP オブジェクト `config` を使用します。

アプリケーションの公開

JRun では、サーブレットを含む Web アプリケーションを公開できます。JRun には、Web ARchive (WAR) ファイルからのアプリケーションの公開がサポートされています。これらのファイルは、アプリケーションを構成するすべてのファイルとそのディレクトリ構造が格納されたアプリケーション配布ファイルの圧縮ファイルです。Web アプリケーションのパッケージ化と公開の詳細については、第 34 章を参照してください。

Java によるサーブレットの作成

Java で作成されたサーブレットの実際を次に示します。次のコード例は、正常に機能するサーブレットの完全なソース コードを示しています。

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

public class SimpleServlet extends HttpServlet {

public void doGet(ServletRequest req, ServletResponse resp)
  throws IOException, ServletException {

  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title> SimpleServlet Output ");
  out.println("</title></head><body>");
  out.println("<h1> SimpleServlet Output </h1>");
  out.println("</body></html>");
}
}
```

サーブレットのテストまたは公開を行うには、先にコンパイルを実行する必要があります。JRun は JSP を自動的にコンパイルします。

要求に応じて、サーブレットは HTML タグを含んでいるテキストをクライアントに返送します。このサーブレット は比較的シンプルですが、Java サーブレット の基本的なフォームと構造を示しています。

Java サーブレット 作成方法に関するチュートリアルと サーブレット の詳細については、第 18 章 を参照してください。

JSP ページとしてのサーブレットの作成

前のセクションでは、Java で作成したサーブレットの例を示しました。JRun には、Java のコーディングにほとんど依存しない、もう 1 つのサーブレットの開発方法がサポートされています。JSPによって、HTMLとスクリプト コードの組み合わせを含むテキスト ファイルからサーブレットを作成できます。

JSP 内のスクリプト コードは、JSP 構文と、通常は JavaScript (ECMAScript のサブセット) または Java の組み合わせになります。JSP 構文、およびスクリプト言語の選択方法の詳細については、第 7 章 および 第 8 章 を参照してください。

JSPファイルは、最初に要求された時点で、JRun によって Javaソース ファイルに変換され、続いて Javaクラス ファイルにコンパイルされます。したがって、Javaコード を 1 行も書かずに、実際の Java サーブレットを作成できます。JSP ファイルの実行時イメージは Java クラス ファイルになるため、Web サーバーは、Java で作成されたファイルと JSP として作成されたファイルの違いを認識できません。

JSPファイルからは、Java で書かれたほかのサーブレット 、または JSPファイルとして実装されたほかのサーブレットを呼び出すこともできます。

次の例は、ブラウザ画面に “Hello World” と 5 回表示する簡単な JSP ページを示しています。

```
<html>
<head>
<title>Greetings</title>
</head>
<body>

<% for(int i=0;i<5;i++) { %>
<h1>Hello World!</h1>
<% } %>

</body>
</html>
```

JSP のファイル名の末尾には、拡張子 .jsp が付きます。JRun は JSP の要求を認識し、JSP を実行可能な Java サーブレットに変換します。JSP の開発方法の詳細については、第 7 章を参照してください。

第 4 章

EJB の概要

この章では、JRun EJB エンジンの機能について簡単に説明します。これらの機能については、第 4 部で詳しく説明します。

目次

- 概要 48
- API 50
- サービス 51
- 環境 55
- インストールの必要条件 55
- ディレクトリ情報 55

概要

EJB エンジンは、コンポーネントのライフサイクル、ネーミング、トランザクション管理、メッセージング、リソース管理、セキュリティ、ディストリビューション、ステート管理、およびパーシスタンスなどのミドルウェア サービスを自動化します。

メモ
EJB の機能は、JRun Professional 版、Advanced 版では使用できません。

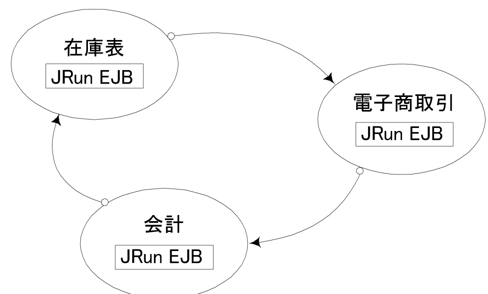
JRun には、EJB コンポーネント アーキテクチャが実装されています。EJB は、ソフトウェア コンポーネント モデルを定義します。これにより、EJB 対応のアプリ ケーション サーバーを使用して、サーバー側アプリケーション ロジック (Bean) を公開できます。

通常、複数の「アプリケーション」から構成される Bean は同じサーバーに公開されます。すべての Bean が内部で開発され、リソースに対する総合的な制御が存在する場合、この事例では何の問題もありません。しかし、同じサーバーやコンテナに公開されたほかの競合する Bean などの影響を受けないソリューションを提供する必要がある場合、この事例をサポートするのは非常に困難です。

従来の EJB 事例では、さまざまなアプリケーションが同じサーバーから提供されるソリューションを定義します。JRun では、EJB サーバーはアプリケーションの内部の目に見えない部分で動作します。

JRun では、EJB パラダイムの利点を十分に活用して、総合的なソリューションを提供できます。

JRun EJB エンジンの組み込み



この設定により、リソース管理とサーバー設定の全体的な制御が可能になります。

JRun は、アプリケーションの内部からサーバーの管理および制御を行うための公開 API を提供します。EJB ベースの API は、セキュリティ、認証、管理、監視など、さまざまな機能のカスタマイズをサポートします。

サーバーに常駐するデータの共有、エンティティ ベースのキャッシュ、およびインプロセスの組み込み動作によって、高速なパフォーマンスを実現できます。拡張性は、エンティティ オブジェクトの自動分散ガーベッジ コレクション、拡張可能な EJB オブジェクト / スタブ アーキテクチャ、およびマルチポート EJB オブジェクト アクセスによって確保されます。

EJB エンジンは軽量でカスタマイズが可能であるため、インストールの特定の要件に合わせて簡単に設定できます。J2EE ベースの実装により、メモリの使用量を少なく保ったままで効果的な実行が保証されます。

使用モード には、スタンド アロンおよびフェイルオーバが含まれます。EJB エンジンのインスタンスまたはサブクラスを直接生成することもできます。標準重視のアーキテクチャと 300 KB 未満のメモリ 使用量に加え、さまざまな使用モード によって、JRun は、専用アプリ ケーション サーバーまたは分散ソリューションの組み込みに理想的な選択肢となっています。

EJB 仕様の完全な実装により、分散型 Java アプリ ケーションの公開および管理に必要な次のようなサポートが提供されています。

- エンティティおよびセッション Bean のサポート
- コンテナ管理パーシスタンス
- 分散型 2 フェーズ コミット トランザクション管理
- 同期および非同期メッセージ
- サーブレットのサポート
- 認証およびセキュリティ
- 効果的なオブジェクトのキャッシュ

EJB エンジンは、J2EE の機能を利用するように設計されており、次のような強力な機能があります。

- リモート起動によるフェイルオーバおよび復旧
- Bean レベルで保護された Secure Sockets Layer (SSL)
- 認証に対する X.509 証明書のサポート
- ACL ベースのカスタマイズ可能なユーザ セキュリティ
- Bean に関する Java セキュリティ ポリシー
- 密封された Bean 実装 JAR
- 参照オブジェクトによる分散型オブジェクト管理
- J2EE コレクションの完全なサポート

これらの機能により、分散型アプリ ケーションの公開のための、安全で安定した環境が提供されます。

API

JRun EJB エンジンは、EJB、JNDI、JTA、JDBC、RMI などの標準 Java API に基づいています。次の表では、これらの API の使用法を簡単に説明しています。

API	使用法
Enterprise JavaBeans (EJB)	セッションおよびエンティティ Bean の完全なサポート。Bean およびコンテナ管理パーシスタンス。高度にカスタマイズできるコンパクトな実装。
Java Naming Directory Interface (JNDI)	JNDI によるサーバー リソースへのアクセス。効果的なローカル コンテキスト検索。LDAP ディレクトリのサポート。
Java Message Service (JMS)	ポイントツーポイントと パブリッシュ /サブスクライブの両方の非同期メッセージング メカニズムを提供します。
Java トランザクション API (JTA)	統合トランザクション管理。暗黙トランザクションまたはクライアント 区分トランザクション。Bean レベルの詳細トランザクション。自動復旧。
Java データベース接続 (JDBC)	標準 SQL データベース アクセス。複数のデータベースのサポート JNDI データ ソースのサポート。効果的な接続プール。プリペアド ステートメント のキャッシュ。
Remote Method Invocation (RMI)	Java オブジェクトの完全なサポート。分散型オブジェクト管理。カスタム ソケット (SSL) のサポート。自動スタブ ダウンロード。

サービス

JRun は、公開した Bean のインスタンスと、それらのインスタンスが使用するリソースを管理するためのサービスを提供します。これらのサービスには、公開、ライフサイクルの管理、コンテキストの実行、分散型 2 フェーズ コミット トランザクションの管理、パーシスタンス、およびセキュリティが含まれます。

Bean には、エンティティ Bean とセッション Bean の 2 種類があります。

- エンティティ Bean はビジネス オブジェクトを表し、複数のクライアントが同時に使用でき、特定のクライアント セッションを越えて存続します。顧客の注文や銀行口座は、エンティティ Bean として表されます。エンティティ Bean は、常に固有の識別子に関連付けられます。この識別子を Bean のプライマリ キーと呼びます。クライアントは、プライマリ キーを使用してエンティティ Bean を見つけます。
- セッション Bean は、特定のクライアント セッション中に限り存続します。セッション Bean に関連するプライマリ キーは存在しないため、通常、複数のクライアントがセッション Bean にアクセスすることはありません。担保計算は、セッション Bean の例です。セッション Bean は、データベース内のデータのアクセスおよび修正に使用できます。

Bean の開発

Bean 開発者は、ビジネス ロジックを含む Bean をコーディングします。これらの Bean は、セッション Bean またはエンティティ Bean のいずれかです。関連する公開プロパティのほかに、Bean には次の環境が必要です。

- ホーム インターフェイス
- リモート インターフェイス
- ビジネス ロジックを含む実装

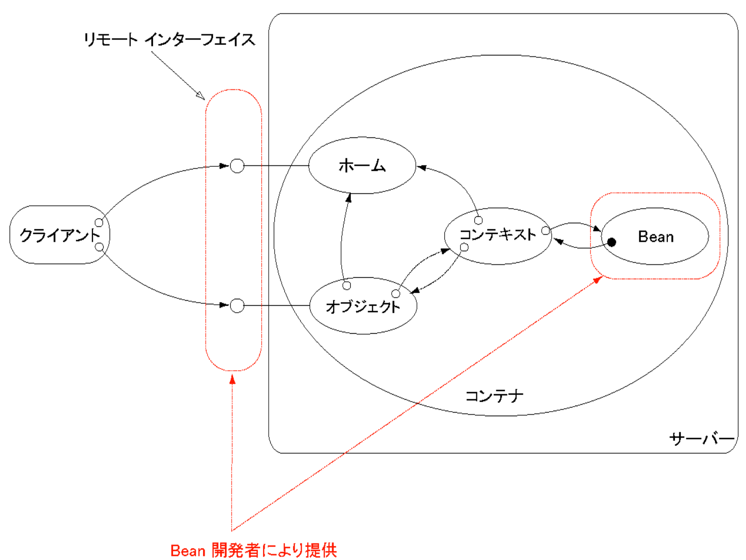
Bean およびそのインターフェイスは、Java ベースの開発環境を使用して開発できます。

Bean 開発者は、Bean に関する公開情報も指定します。公開情報は、標準 Java プロパティ、および Bean プロパティの場合は XML 記述子を使用して指定できます。Bean には、必要なプロパティおよびオプションで設定できるプロパティが数多くあります。JRun JavaDocs ファイルに付属する `EjiptProperties` API マニュアルには、これらのプロパティの完全な一覧が記載されています。

ライフサイクル

EJB エンジンは、公開された Bean のライフサイクルを管理します。Bean に関連付けられたホームおよびリモート インターフェイス (オブジェクト) の実装により、EJB エンジンは Bean のインスタンスの作成、配置、および破棄を行います。

EJB エンジンは、起動時に JNDI を介して Bean を登録し、クライアントが Bean のホーム オブジェクトへの参照を取得できるようにします。ホーム オブジェクトへの参照を取得すると、クライアントは `create` または `find`*`XXX`* メソッドを呼び出して、Bean のオブジェクトへの参照を取得できます。次に、クライアントはこれらのオブジェクトに関するメソッドを呼び出します。オブジェクト実装は、これらのメソッド呼び出しを実際の Bean 実装に転送します。



クライアントは、Bean のホーム オブジェクトとリモート オブジェクトにのみアクセスできます。クライアントは Bean 実装 (インスタンス) に関するメソッドを直接呼び出すことはできません。

コンテキスト

公開された Bean の各インスタンスには、Bean コンテキストが関連付けられます。Bean のコンテキストは、環境にアクセスするための方法を Bean インスタンスに提供します。これには、呼び出し ID、環境プロパティ、Bean のホームおよびリモート オブジェクトへの参照、およびトランザクション情報が含まれます。

インスタンスの有効期間中は、コンテキストは公開された Bean の特定のインスタンスに連結されます。

トランザクション

JRun は、JTA インターフェイスを使用した 2 フェーズ コミット トランザクション管理によって、分散される可能性のある作業単位の整合性を確保します。Bean は、単一のメソッド 呼び出しから 複数の参加者がかかわる 複数のメソッド 呼び出し にいたるまで、すべての作業単位を定義します。

Bean は、自らトランザクションを制御するか、またはプロパティ ファイル内の宣言を通じてトランザクション管理を EJB エンジンに委任できます。EJB に委任する場合、EJB エンジンは Bean に代わって自動的にトランザクションを開始し、コミットします。

分散型 2 フェーズ コミット トランザクション プロトコルの完全な実装が提供されます。ただし、配布の必要がなく、トランザクションの参加者が 1 人だけの場合、トランザクション サポートは自動的に非常に効果的なローカル実行モードに切り替わります。

パーシスタンス

JRun は、Bean 管理パーシスタンス (BMP) と コンテナ管理パーシスタンス (CMP) の両方を提供します。プロパティを使用して、コンテナ管理パーシスタンスのパラメータを定義することで、CMP の正確な設定および制御が可能になります。EJB エンジンは、オブジェクトのパーシスタンスの管理に関して JDBC API に依存します。

データベース リソースは、設定可能なプール内であらかじめ決定されたデータベース接続数で管理されます。接続は、必要に応じてプールから取得されます。トランザクションの完了後、関連する接続は自動的にプールに返され、その後の呼び出しで再利用されます。

データの取得中に結果をキャッシュに保存することで入出力を最小限に抑え、パフォーマンスを最大にします。

ファイル ベースの拡張可能な `instance.store` を、Bean またはコンテナ管理パーシスタンスと組み合わせて使用することもできます。`instance.store` は、データベースを使用できない場合やデータベースが不適切な場合に使用できます。

メッセージ サポート

JRun は、Java Message Service (JMS) 仕様を完全に実装することにより、メッセージ サポートのシームレスな統合を提供します。Bean 内のプロデューサ、コンシューマ、および JMS セッションの作成がサポートされています。Bean を非同期メッセージングのリスナとして使用することもできます。メッセージは、EJB エンジンのエンティティ Bean アーキテクチャによって提供されるトランザクション サポートを使用して、完全に処理されます。

セキュリティと認証

JRun は、セキュリティの実装に関して高度に設定可能なメカニズムを提供します。確認するユーザ情報の種類と内容、ロールを構成する情報、および認証の方法などを指定できます。このような柔軟性により、既存のシステムにすでに設置されているセキュリティ方式に順応できます。

ユーザ認証

ユーザおよびロールの概念は、エンティティ Bean を通じて実装されます。ユーザ Bean では、特定のユーザを電子メール アドレス、電話番号、部署などの詳細情報で表すことができます。また、ユーザ Bean は、業務カテゴリやタイプを表すこともできます。ロール Bean は、業務カテゴリ 、配属、その他の役割など、ユーザのグループを表します。

`java.security.acl.Group` がロールを実装するための基本インターフェイスであるのに対して、`java.security.Principal` は、基礎ユーザを実装するための基本インターフェイスです。ユーザとロールの両方の既定の実装がサーバーに含まれています。これらのクラスを拡張して、特定の認証方式を実装できます。

ユーザおよびロールは、既存のデータベース、フラット ファイル、または `ejipt.properties` ファイルに格納される可能性があります。ユーザとロールが、エンティティ Bean であるため、Bean 管理パーシスタンスとコンテナ管理パーシスタンスのどちらも、データ検索と更新に使用できます。

アクセス制御

Bean およびそのメソッドのアクセス制御は、ユーザまたはロールレベルで簡単に指定できます。各 Bean は、Bean へのアクセスを許可するユーザまたはロールを指定できます。この方法で、特定メソッドへのアクセスも制御できます。

Bean のプロパティ ファイルまたは公開記述子には、EJB エンジンが実行時セキュリティを管理するために使用するアクセス制御エントリが含まれています。これらのエントリは、公開の際にアクセス制御リスト (ACL) を作成するために使用されます。ACL には、ユーザとロールのエントリ、またはどちらかのエントリが含まれます。エントリは、Bean 全体または Bean 内の特定のメソッドに適用できます。公開された各 Bean には、ACL が関連付けられます。

メソッド呼び出しのたびにセキュリティ チェックが実行され、ACL で定義されているアクセス許可に対してユーザおよびロールが確認されます。ロール ツリーは、特定のロールを持つかどうかを確認するために通過されます。

優先 ID

Bean のプロパティ ファイルまたは公開記述子は、特定のメソッドに対して実行 ID を指定できます。実行モードは、使用する ID の種類を指定します。ID の種類は、クライアント、システム管理者、または特定のユーザの ID で指定できます。

特定のユーザの場合は、実際に使用する ID を指定するため、実行 ID プロパティを有効なユーザまたはロール インスタンスに設定する必要があります。

環境

サード パーティ製ツールと JRun との間に、依存関係はありません。開発した Bean は、標準 IDE を使用して公開できます。

公開の際に、公開される Bean のホーム インターフェイスおよびリモート インターフェイスを実装するクラスが、Deploy ツールによって自動的に生成されます。生成されたこれらのオブジェクト クラスをコンパイルするため、Deploy ツールはコンパイラにアクセスする必要があります。サーバーで生成されたオブジェクトは、別のサーバーでロードできます。

インストールの必要条件

JRun は完全に Java で記述されているため、J2EE プラットフォームを使用できるあらゆる環境で実行できます。JRun EJB エンジンは、J2EE の機能を広範囲にわたって使用します。組み込み Bean を実行するには、J2EE JRE をインストールする必要があります。ただし、Bean を開発する場合やサンプルを実行する場合は、JDK 1.2 以降のバージョンをインストールする必要があります。

ディレクトリ情報

本書の全体にわたって JRUN_HOME への参照が記載されていますが、これは、JRun をインストールしたディレクトリへの絶対パスです。すべてのディレクトリ構造の詳細については、第 24 章を参照してください。

JRun を再インストールする際に、既存のディレクトリに保持するファイルが含まれている場合は、インストールする前に既存のディレクトリの名前を変更してください。JRUN_HOME ディレクトリ にインスト ール コンポーネント と同じ名前のファイルが存在する場合、そのファイルはインストーラによって上書きされます。ただし、deploy および runtime ディレクトリ の中身は必ず維持され、削除されたり上書きされることはありません。

第 5 章

Web アプリケーションの開発

完全な Web アプリケーションは、Java サーブレット、JSP、HTML ページのようなスタティック コンテンツ、その他のアプリケーション リソースから構成されます。1 つの JRun サーバーで、異なる URL にマッピングされる複数の Web アプリ ケーションをサポートできます。

この章では、最初に Web アプリケーションの構造について説明し、次に JRun で Web アプリケーションおよびそのリソースを処理する方法について説明します。続いて、既定のアプリケーションと呼ばれる特別な Web アプリケーションについて説明します。最後に、Web アプリケーションの作成と公開の方法について説明します。

目次

- Web アプリケーションの概説 58
- 既定の Web アプリケーションの使用 67
- Web アプリケーションの開発 70
- Web アプリケーションの公開 76

Web アプリケーションの概説

Web アプリケーションは、サーブレット、JSP ドキュメント、HTML ドキュメント、イメージ、およびその他のリソースから構成されています。あらかじめ定義されているディレクトリ構造に従ってこれらのリソースを配置し、どの Web アプリケーション サーバーにも公開できるようにします。

このセクションでは、Web アプリケーションの重要な機能と利点について説明します。

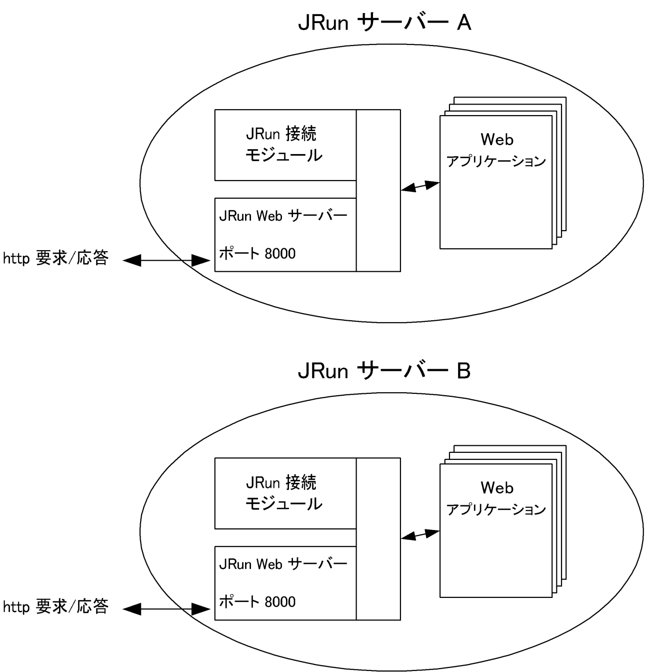
Web アプリケーションの利点

Web アプリケーションについては、Java サーブレット API バージョン 2.2 の仕様書で定義されています。Web アプリケーションによる方法には、次の利点があります。

- アプリケーションのディレクトリ構造や、アプリケーションの定義に必要な情報など、Web アプリケーション表現の標準定義
- Web アプリケーション サーバーへの Web アプリケーションの公開の標準定義。特定の Web アプリケーション サーバー用に記述された Web アプリケーションは、Java サーブレット API バージョン 2.2 の仕様書に準拠するすべての Web アプリケーション サーバーに移植可能であることが保証されます。
- 「Write Once, Run Anywhere (一度記述すればどこででも実行可能)」という Java プログラミング言語の目標に準拠。Java 言語自体は、ほかのプラットフォームに移植可能な言語です。標準化された Web アプリケーション表現を使用すると、異なるアプリケーション サーバーに移植可能なアプリケーション構造を作成できます。
- アプリケーション内のアプリケーション リソースへの相対リンクが使用可能。Web アプリケーションでは絶対参照を使用しないので、アプリケーション サーバ上のアプリケーションの位置は考慮しません。そのため、開発された場所とは異なるディレクトリ、URL、または異なるサーバーに Web アプリケーションを公開できます。

Web アプリケーションの使用

1 つの JRun サーバーで、複数の Web アプリ ケーショ ンをサポート できます。次の図は、複数の Web アプリケーションのホストとして機能している 2 つの JRun サーバーを示します。



Web アプリ ケーショ ンでは、ほかのアプリ ケーショ ン内のリ ソースの参照や、一般的なリソースの共有が可能です。共有することによって、Web アプリケーションは EJB やデータ ベースドライバクラスなど、多く のアプリ ケーショ ンで一般的に使用されているリソースにアクセスできます。

JRun 接続モジュ ール (JCM) は、Web サーバーと JRun サーバー間の接続を管理します。JCM の設定の詳細については、『JRun セットアップ ガイド』を参照してください。

Web アプリケーションでは、データベースや EJB を使用してデータを共有することもできます。たとえば、電子商取引 Web サイトが、複数のアプリケーションから構成されているとします。このタイプのサイト の顧客は、ログイン名とパスワード によって識別できます。そのため、各 Web アプリケーションではログイン名を使用して、ショッピング カート、購入履歴、住所などのデータベース内の顧客の情報にアクセスします。

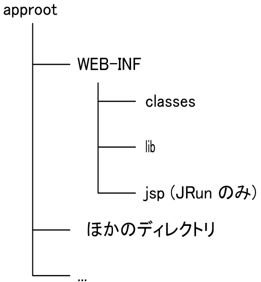
Web アプリケーションを開発する場合に決定しなければならないことは、複数のJRun サーバー上のアプリケーション間の境界線をどこに設定するかという点です。つまり、1 つのJRun サーバーですべてのアプリケーションを稼動できるようにするのか、複数の JRun サーバー間にアプリケーションを分散させる必要があるのかについて決める必要があります。

複数のJRun サーバーを作成する理由の 1 つは、アプリケーションをコンピュータ上の異なるプロセスに分離するためです。たとえば、JRun サーバー内のすべての Java サーブレット、JSP、およびアプリケーションは、1 つのプロセスで実行します。アプリケーションを複数の JRun サーバーに分離することによって異なるプロセスを使用し、アプリケーションが別のアプリケーションに悪影響を与えないようにすることができます。さらに、クラスパス、データ ソース、および EJB をサーバー レベルで定義できます。

Web アプリケーションをそれぞれ異なる JRun サーバーで実行するもう 1 つの理由は、各 JRun サーバーで独自のユーザ認証メカニズムや一連のユーザ認証ルールを実装できることです。異なる JRun サーバーでアプリケーションを実行することによって、特定のサーバーの認証設定を利用できます。認証の詳細については、第 39 章を参照してください。

Web アプリケーションのディレクトリ構造

次の図は、Web アプリケーションのディレクトリ構造を示します。



アプリケーションのルート ディレクトリ (この図では *approot*) は、アプリケーション ファイルを提供するためのドキュメント ルートとして機能します。このディレクトリには、Web アプリケーションの一部として開発された JSP が含まれます。たとえば、Web アプリケーションが `c:/apps/app1` にある場合、既定のトップ ページ ファイルは `c:/apps/app1/index.html` に配置されます。

ディレクトリ ルートのサブディレクトリには、次のディレクトリが含まれる場合がありますが、これらに限定されません。

- `WEB-INF` アプリケーションに関連するリソースが格納されます。このディレクトリには、アプリケーションの設定情報を保存する `web.xml` ファイルが格納されます。

  このディレクトリは、アプリケーションの公開ドキュメント ツリーの一部ではありません。したがって、このディレクトリまたはそのサブディレクトリに含まれるファイルは、クライアントに直接提供されません。たとえば、アプレットが含まれている JAR ファイルはクライアントによるアクセスが可能なディレクトリになければならないため、`WEB-INF` には格納されません。
- `WEB-INF/classes` アプリケーションの Java サーブレットの Java クラス ファイルが格納されます。
- `WEB-INF/lib` アプリケーション固有のクラスが格納されます。これらのファイルは、Java ARchive (JAR) または `.zip` ファイルに保存されている必要があります。このディレクトリには、タグ ライブラリを保存している JAR ファイルも格納されます。
- `WEB-INF/jsp` JSP の変換時に JRun によって作成されるファイル (`.class`、`.java`) が格納されます。このディレクトリは、Web アプリケーション仕様の一部ではなく、JRun によって追加されたものです。

実際の Web アプリケーションで使用するのは、前出の図に示したディレクトリだけではありません。HTML ファイル、イメージ、その他のアプリケーション リソースなどのために、ほかのディレクトリを追加できます。これらのディレクトリは、クライアントが直接アクセスするリソース用 Web アプリケーションの公開ドキュメント ツリーの一部になります。アプリケーションへのディレクトリ追加の詳細については、71 ページの「ディレクトリの追加」を参照してください。

ここに記載されているディレクトリ以外に、JRun サーバーでは、そのサーバーがホストとなる Web アプリケーションごとに一時ディレクトリが用意されます。この一時ディレクトリは JRun 自体では使用されませんが、Java サーブレットや JSP 実行時の一時的な領域スペースを取得できるように用意されています。たとえば、このディレクトリを使用してデータベース クエリの結果をキャッシュできます。

次の命名規約に従って、一時ディレクトリが Web アプリケーションごとに自動的に作成されます。

サーバー名/tmp/*アプリケーション名*

ここで、*サーバー名*は JRun サーバーのディレクトリ名で、*アプリケーション名*はサーバーがホストとなる Web アプリケーションの名前に対応するディレクトリの名前です。実行時は、次のステートメントを使用して、サーブレット内からこのディレクトリへの参照を取得できます。

```
File f = (File) getServletContext().getAttribute
        ("javax.servlet.context.tempdir");
```

公開記述子 (web.xml)

Web アプリケーションは、web.xml ファイルの内容によって定義されます。このファイルは、**公開記述子**とも呼ばれます。このファイルには、アプリケーション サーバー上でアプリケーションを実行するために必要なすべての情報が含まれています。web.xml の内容は JRun 固有のものではなく、Java サーブレット API バージョン 2.2 の仕様書に定義されています。Java サーブレット API バージョン 2.2 の仕様書に定義されているように Web アプリケーションをサポートしているすべてのプラットフォームは、web.xml ファイルの内容を認識して、解釈します。

web.xml ファイルを使用して、Web アプリケーションの次のような設定情報と公開情報を定義します。

- サーブレット 初期化パラメータ
- セッション設定
- サーブレットおよび JSP の定義
- サーブレットおよび JSP URL のマッピング
- MIME タイプのマッピング
- トップ ページ ファイルのリスト
- エラー ページ
- セキュリティ情報

web.xml ファイルには、Web アプリケーションについての情報だけでなく、EJB についての情報も含まれています。この情報には、サーブレットが EJB のホーム インターフェイスを探すときに必要な設定値が含まれています。

web.xml ファイルは標準的なテキスト エディタや XML エディタを使用して編集する XML ファイルです。また、JRun 管理コンソール (JMC) を使用すると、このファイルの多くの属性を変更できます。web.xml ファイルの各設定値については、本書の該当する部分で説明しています。web.xml ファイルのすべてのプロパティの全リストについては、Java サーブレット API バージョン 2.2 の仕様書を参照してください。

アプリケーション コンポーネント

Web アプリケーションは次のコンポーネントから構成されます。

- HTML ページ
- Java サーブレット
- JSP
- カスタム タグ ライブラリ
- リソースのイメージ ディレクトリ、Bean、およびデータベース ドライバなどのリソース用のクラス ファイル

アプリケーションにリソースを追加するには、Web アプリケーションの適切なディレクトリにそのリソースを配置します。アプリケーションへのこれらのリソースの追加の詳細については、71 ページの「Web アプリケーション コンポーネントの追加」を参照してください。

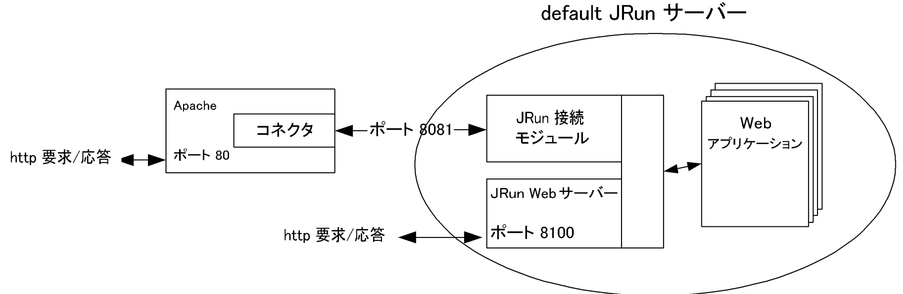
Web アプリケーション、JRun サーバー、Web サーバー

Web アプリケーションは、JRun などのアプリケーション サーバーをホストとして利用します。Web アプリケーション開発時の最初の作業の 1 つとして、アプリケーションを特定の JRun サーバーに関連付けます。多くの場合、Web アプリケーションを default JRun サーバーまたはユーザが作成した JRun サーバーに関連付けます。

メモ

Web アプリケーションを admin JRun サーバーと関連付けないでください。admin JRun サーバーは、主にすべての JRun サーバーを含む JRun インスト ールの管理に使用します。

次の図は、複数のアプリケーションのホストとして機能する default JRun サーバーを示します。



この図には、この JRun サーバーに関連付けられている Web サーバーも示されています。Web サーバーはクライアントからの要求を URL 形式で受け取り、要求が Java サーブレットや JSP などの Web アプリケーション リソースを参照するときに JRun にこの要求を渡します。

JRun サーバーをホストとするさまざまな Web アプリケーションにクライアントの要求を割り 当てるには、各 Web アプリ ケーション をマッピングし て異なる URL パターンに対応します。この方法で、JRun サーバーは適切な Web アプリケーションに要求を転送できます。

アプリケーションのマッピング

アプリケーションのマッピングによって、要求の URL をアプリケーションが含まれている物理ディレクトリに関連付けます。Web アプリケーション公開の一環として、アプリケーションへの URL パスを指定して Web サーバーによって認識されるようにする必要があります。

Web アプリケーションが http://hostname/appURL/resourcename の形式の URL に応答するように、URL マッピングを設定します。このマッピングを設定すると、接頭辞 http://hostname/appURL/ で始まるすべての URL がこのアプリケーションにマッピングされます。

メモ
JRun ではアプリケーションの URL マッピングに制限はありません。アプリケーション名を URL マッピングの一部として使用する必要はありませんが、これを任意の文字列にマッピングできます。

たとえば、次の表は Web アプリケーション、アプリケーション URL マッピング、およびアプリケーション リソースの要求 URL の物理的位置の一覧です。

Web アプリケーションのディレクトリ	アプリケーションの URL マッピング	クライアント要求 URL
`c:/apps/app1`	/app1	http://hostname/app1/resource_name
`c:/apps/app2`	/app2	http://hostname/app2/resource_name

この例では、`c:/apps` という名前のディレクトリを作成し、すべての Web アプリケーションを格納します。次に、要求 URL が app1 または app2 のどちらかにマッピングされるように、JRun でアプリケーション URL マッピングを設定します。

JRun が特定のリソースの要求 URL を解決する方法については、第 6 章を参照してください。

Web アプリケーション クラスパスの決定

Web アプリケーションのクラスパスは、アプリケーションがアクセスできるクラスを定義します。たとえば、Java サーブレット の `.class` ファイルは、サーブレット を処理できるように、JRun 用の Web アプリケーションのクラスパスに含まれているディレクトリ内になければなりません。このクラスパスには、`.class` ファイルが含まれているディレクトリまたは JAR ファイルも含まれます。

Web アプリケーションのクラスパスの定義は、再ロード可能部分と、再ロード不可部分の 2 つに分けられます。再ロード可能なクラスには、JSP や Java サーブレットがあります。実行時は、クラスパスの再ロード可能部分で定義されるクラスがチェックされます。メモリ内のクラスのイメージがディスク上のイメージと異なる場合は、クラスが再ロードされます。

アプリケーションのクラスパスの再ロード不可部分によって参照されるクラスは 1 度だけロードされ、変更についてはチェックされません。通常、Web アプリケーションの開発の一環として変更しない基本的な Java クラス、JRun ファイル、JVM クラスなどが再ロード不可クラスとなります。

Web アプリケーションの既定の再ロード可能なクラスパスには、次のディレクトリが含まれます。

- `approot/WEB-INF/classes`
- `approot/WEB-INF/lib`
- `approot/WEB-INF/jsp`

既定の再ロード不可クラスパスには、次のディレクトリとファイルが含まれます。

- *JRun のホーム ディレクトリ*`/lib/ext` このディレクトリ内のすべての JAR ファイルが含まれます。
- *JRun のホーム ディレクトリ*`/servers/lib` このディレクトリ内のすべての JAR ファイルが含まれます。
- *JRun のホーム ディレクトリ*`/lib/jrun.jar`
- *JRun のホーム ディレクトリ*`/lib/install.jar`
- *JRun サーバーのルート ディレクトリ*`/lib` このディレクトリ 内のすべての JAR ファイルが含まれます。
- このほかの JRun JAR ファイル

Web アプリケーションのクラスパスの変更

再ロード 可能部分と 再ロード 不可部分の両方とも、個別の JRun プロパティ ファイルを使用して Web アプリケーションのクラスパスを変更します。

JRun プロパティ ファイルの階層によって、個々の JRun サーバーによって処理されるすべての Web アプリケーションのクラスパスを設定したり (`local.properties` を使用)、個々の Web アプリケーションのクラスパスを設定できます(`webapp.properties` を使用)。

クラスパスの再ロード 可能部分を変更するには、`webapp.classpath` プロパティの設定を変更します。`webapp.classpath` の既定値は、次のとおりです。

```
webapp.classpath=/WEB-INF/classes;/WEB-INF/lib;/WEB-INF/jsp
```

`webapp.classpath` に指定されるパスは通常、アプリケーションの `/WEB-INF` ディレクトリの下にあります。

再ロード不可部分のアプリケーション クラスパスは、次の 2 つのプロパティから構成されています。

- `jrun.classpath` JRun 自体が必要とする `.class` および JAR ファイル
- `user.classpath` ユーザ指定の `.class` および JAR ファイル。すべてのユーザアプリケーションおよび Java Virtual Machine (JVM) によって使用されます。

これらの 2 つのプロパティによって指定されるディレクトリについては、そのディレクトリに含まれている各 JAR ファイルが、JRun が呼び出される前にクラスパス内のディレクトリの後ろに自動的に追加されます。これらの 2 つのプロパティの既定値は、次のとおりです。

```
jrun.classpath = {jrun.rootdir}/lib/ext;{jrun.rootdir}/lib/
        jrun.jar;{jrun.rootdir}/lib/install.jar
user.classpath={jrun.rootdir}/servers/lib;{jrun.server.rootdir}/lib
```

Web アプリケーション間でのクラスの共有

Web アプリケーションのディレクトリ構造は、Java サーブレット API バージョン 2.2 の仕様書に定義されています。この仕様に定義されているように、Web アプリケーションに関連付けられているすべての `.class` ファイルは、そのアプリケーションのルート ディレクトリに格納する必要があります。

ただし、JRun には、複数の Web アプリケーションで共通ファイルを共有できるように、ディレクトリ構造の外部にある `.class` ファイルにアクセスする Web アプリケーションの機能が追加されました。ファイルを共有すると、複数のアプリケーションに共通するデータベース ドライバ クラスやカスタム タグ ライブラリなどの `.class` ファイルを 1 つの場所に格納しておくことができます。共有リソースの追加は、共有ディレクトリに `.class` や JAR ファイルを移動し、これらのリソースにアクセスするアプリケーションのホストである JRun サーバーを再起動するだけなので簡単です。

メモ
アプリケーションのディレクトリ構造の外部の `.class` ファイルにアクセスするようにアプリケーションを定義する場合、アプリケーションの WAR ファイルを作成する前に、Web アプリケーションのディレクトリ構造にすべての共有リソースをコピーする必要があります。

JRun では、各 Web アプリケーションは JRun サーバーをホストとします。すべての Web アプリケーションが同じ JRun サーバーをホストとするように環境を定義したり、Web アプリケーションを複数の JRun サーバー間に分散できます。Web アプリケーションのホストになる JRun サーバーに関係なく、Web アプリケーション間でクラスファイルを共有できます。

JRun には、共有 `.class`ファイルに使用できる次のライブラリ ディレクトリ があります。

- `jrun/servers/lib`　すべての JRun サーバーからアクセス可能、つまりすべての Web アプリケーションからアクセス可能な JAR ファイルと `.class` ファイルを格納します。JRun のすべての Web アプリケーションは、このディレクトリ内のファイルにアクセスできます。これらのリソースは再ロード可能ではありません。
- *JRun サーバーのルート ディレクトリ*`/lib`　特定の JRun サーバーに関連付けられているすべてのアプリケーションからアクセス可能な JAR ファイルと `.class` ファイルを格納します。JRun サーバーがホストとなっているすべての Web アプリケーションは、このディレクトリ 内のファイルにアクセスできます。これらのリソースは再ロード可能ではありません。

分散型 Web アプリケーション

今回の JRun では、分散型 Web アプリケーションをサポートしていません。

既定の Web アプリケーションの使用

1 つの JRun サーバーで、複数の Web アプリケーションがサポートされます。どのアプリケーションが要求に対応するのか JRun が認識できるように、これらの各アプリケーションは異なる URL にマッピングされます。

Web アプリケーションは、URL にアプリケーション名が含まれている次のような URL にマッピングされることもあります。

http://hostname/app1/*app1_resource*

http://hostname/app2/*app2_resource*

この例では、http://hostname/app1/ で始まる URL は app1 内のリソースを参照し、http://hostname/app2/ で始まる URL は app2 内のリソースを参照します。このアプリケーションのルート ディレクトリは、アプリケーション ファイルを提供するためのドキュメント ルートとして機能します。たとえば、Web アプリケーションが `C:¥apps¥app1` にある場合、既定のトップ ページ ファイルの場所は `C:¥apps¥app1¥index.html` になります。このファイルは、http://hostname/app2/index.html の URL を使用してクライアントに提供できます。

ただし、このアプリケーションのマッピング規則には 1 つの例外があります。既定の Web アプリケーションにはこの規則は当てはまりません。既定の Web アプリケーションは、"/" または "//hostname/" のいずれかにマッピングされます。既定のアプリケーションは、次の形式の URL に応答します。

http://hostname/*resource*

既定の Web アプリケーションには次の特徴があります。

- "/" または "`//hostname/`" にマッピングされます。

  JRun では 最も限定的な参照から最も広範囲の参照の順に URL が解決されます。ある URL にマッピングされるほかのリソースが見つからない場合は、常に既定のアプリケーションを使用して URL が解決されます。

- ドキュメント ルート ディレクトリとして Web サーバーのドキュメント ルート ディレクトリが使用されます。

  既定のアプリケーションでは、Web サーバーのドキュメント ルート ディレクトリの JSP やほかのサーバー側スクリプトを自動的に使用します。たとえば、IIS Web サーバーを使用している場合は通常、`¥inetpub¥wwwroot` がドキュメント ルートになります。既定の Web アプリケーションでは、このディレクトリをドキュメント ルートとして使用するため、JSP やほかのサーバー側スクリプトを `¥inetpub¥wwwroot` に配置できます。

- クラスパスにディレクトリ JRun/servlets が含まれます。

  旧バージョンの JRun では、開発者が、このディレクトリに Java サーブレットの .class ファイルを配置していました。JRun/servlets を既定のアプリケーションのクラスパスに含めることによって、既定のアプリケーションに旧バージョンの JRun との下位互換性を持たせることができます。既定では、このディレクトリがほかの Web アプリケーションで使用されることはありません。

既定の Web アプリケーションを利用する主な理由の 1 つは、旧バージョンの JRun および Java サーブレット API との下位互換性を持たせることです。これによって、既存の JRun アプリケーションからこのバージョンの JRun へのアップグレードが簡単に行えます。旧バージョンの JRun では、開発者が JSP やほかのサーバー側スクリプトを Web サーバーのドキュメント ルート ディレクトリに配置し、Java サーブレットの .class ファイルを {jrun root dir}/servlets ディレクトリに配置していました。

JRun サーバーに接続されている各 Web サーバーには、関連する既定の Web アプリケーションが含まれています。たとえば、default JRun サーバーには、ディレクトリ {jrun root dir}/servers/default/default-app に default-app という名前の Web アプリケーションが含まれています。

既定の Web アプリケーションに対する要求の処理

次に、既定の Web アプリケーションのリソースに対応する URL の例を示します。

http://hostname/*resource*

JRun では 最も限定的な参照から最も広範囲の参照へと URL が解決されます。この場合、URL に一致するアプリケーションは既定のアプリケーションだけなので、既定のアプリケーションがこの要求を処理します。JRun が URL を解決する方法の詳細については、第 6 章を参照してください。

JRun は、リソース名が一致するリソースを、既定のアプリケーションのルート ディレクトリ (通常は、Web サーバーのドキュメント ルート) から検索します。リソースが見つかると、そのリソースが返されます。JSP または Java サーブレット の場合、リソースを返すことは、JSP や Java サーブレットを処理して、その処理結果をクライアントに返すことを意味します。

Web アプリケーションで URL に対応するリソースが見つからない場合、JRun は Web サーバーに処理の要求を返します。たとえば、クライアントが次の要求を発行したとします。

http://localhost/index.htm

"/" は既定のアプリケーションと一致しますが、index.htm に対応するサーブレット のマッピングが既定のアプリケーションにない場合、JRun から Web サーバーに制御が返されます。その後、Web サーバーはドキュメント ルート からクライアント にファイル index.htm が返されます。

default-app Web アプリケーションの使用

JRun には、default JRun サーバーで使用する、`default-app` という既定のアプリケーションがあります。このアプリケーションは、ディレクトリ `JRun/servers/default/default-app` にあります。

JRun をインストールすると、`default-app` には次のような既定のアプリケーションのすべての特徴が加わります。

- "/" にマッピングされます。
- ドキュメント ルート ディレクトリとして Web サーバーのドキュメント ルート ディレクトリが使用されます。
- クラスパスに、ディレクトリ `JRun/servlets` が含まれます。このため、このディレクトリにサーブレット `.class` ファイルを配置できます。

`default-app` によって、最初に Web アプリケーションを作成しなくても、Java サーブレット、JSP、および EJB の開発を開始できます。

Web アプリケーションに対して追加する場合と同じ規則を使用して、コンテンツを `default-app` に追加します。Web アプリケーションへのリソースの追加の詳細については、71 ページの「Web アプリケーション コンポーネントの追加」を参照してください。

既定のアプリケーション ディレクトリ構造

60 ページの「Web アプリケーションのディレクトリ構造」で説明しているように、既定のアプリケーションでは、ほかのアプリケーションと同じディレクトリ構造を使用します。既定のアプリケーションのディレクトリ構造における唯一の違いは、既定のアプリケーションは、独自のドキュメント ルート ディレクトリではなく、アプリケーション ルート ディレクトリとして Web サーバーのドキュメント ルート ディレクトリを使用する点です。

既定のアプリケーションでは、Web サーバーのドキュメント ルート ディレクトリを使用して、Web サーバーのドキュメント ルート の JSP やほかのサーバー側スクリプトを操作できます。この場合、既定のアプリケーションでは、`WEB-INF` の下にある `.class` および JAR ファイル以外のリソースについては、独自のルート ディレクトリを参照しません。

メモ

インストールすると、JRun は default JRun サーバー上で JRun Web サーバー (JWS) を起動するように設定されます。JWS のドキュメント ルート、つまり既定のアプリケーションのドキュメント ルートは、*JRun のホーム ディレクトリ*`/servers/default/default-app` です。

既定の Web アプリケーションのクラスパス

すべての Web アプリケーションのクラスパスについては、64 ページの「Web アプリケーション クラスパスの決定」のセクションで説明しています。このセクションで説明したクラスパスだけでなく、既定の Web アプリケーションのクラスパスには、`{jrun.rootdir}/servlets` というパスも追加されています。これは、既定の Web アプリケーションが、このディレクトリ内のすべての `.class` または JAR ファイルにアクセスできることを意味します。

`{jrun.rootdir}/servlets` ディレクトリは再ロード可能です。これは、JRun はこのディレクトリ内の `.class` および JAR ファイルの依存チェックを行うことを意味します。対応するサーブレットが最後にロードされた後にファイルが変更されていれば、サーブレットが再ロードされます。

Web アプリケーションの開発

JRun は Web アプリケーション サーバーです。したがって、JRun によって動作できるようにするには、すべての Java サーブレットおよび JSP を Web アプリケーションの一部として組み込む必要があります。このセクションでは、最初に新規の JRun アプリケーションの作成手順について説明し、次に Web アプリケーションを構成するリソースおよびコンポーネントの追加方法について説明します。

Web アプリケーションの作成

JRun には、空の新規 Web アプリケーションを作成するユーティリティが用意されています。Web アプリケーションの基本ディレクトリ構造が作成され、適切な情報が格納された `web.xml` ファイルが作成されるとともに、アプリケーションが登録されてアプリケーションの URL マッピングが作成されます。

アプリケーションを作成するには、次の手順を実行します。

1 JMC の左側ペインで、[*マシン名*] > [*サーバー名*] > [**Web アプリケーション**] をクリックします。

2 右側ペインで、[アプリケーションの作成] をクリックします。

3 アプリケーションのサーバー名を選択します。

4 左側ペインのサーバーの下に表示される情報に従ってアプリケーション名を指定します。

5 アプリケーションの Web サーバー ホストを指定します。

6 アプリケーションの URL マッピングを指定します。

7 アプリケーションのルート ディレクトリを指定します。

メモ
Web アプリ ケーショ ンのルート ディレクトリ は、JRun のディレクトリ 構造に入れる必要はありません。システム上の任意の場所に作成できます。

8 [作成] をクリックしてアプリケーションを作成します。

9 Web アプリケーションにコンテンツを追加します。アプリケーション リソースの追加の詳細については、71 ページの「Web アプリケーション コンポーネントの追加」を参照してください。

Web アプリケーション コンポーネントの追加

完全な Web アプリケーションは、Java サーブレット、JSP、HTML ページのようなスタティック コンテンツ、タグ ライブラリ、EJB、その他のアプリケーション リソースから構成されます。Web アプリケーションの開発作業の一部として、アプリケーションのディレクトリ構造にこれらのコンポーネントを追加します。

次のセクションでは、Web アプリケーションにコンポーネントを追加する方法について説明します。

- 「ディレクトリの追加」71 ページ
- 「HTML ページの追加」72 ページ
- 「Java サーブレットの追加」73 ページ
- 「JSP の追加」72 ページ
- 「タグ ライブラリの追加」75 ページ
- 「EJB の追加」75 ページ
- 「リソースの追加」75 ページ

ディレクトリの追加

Web アプリケーションのディレクトリ構造では、`WEB-INF` という名前のサブディレクトリを 1 つ以上定義します。このディレクトリについては、60 ページの「Web アプリケーションのディレクトリ構造」を参照してください。

ただし、多くの Web アプリ ケーションでは、アプリ ケーション ルート ディレクトリ の下に `WEB-INF` 以外のディレクトリも追加されています。そのアプリケーションのクラスパスに含まれていなければならない `.class` または JAR ファイルが新しいディレクトリに含まれていない場合を除いて、アプリケーションのルートにサブディレクトリを追加するときは、ディレクトリ作成以外の特別な作業は必要ありません。

たとえば、通常は、アプリ ケーションのルート ディレクトリ の下にある `images` ディレクトリにイメージ ファイルを配置します。このほかの一般的なディレクトリとして、アプリケーションのルート ディレクトリの下の `include` ディレクトリがあります。このディレクトリには、複数のアプリケーション リソースで共有するファイルが格納されます。この場合は、アプリケーションのルートの下に `include` ディレクトリを作成できます。

HTML ページの追加

このアプリケーションのルート ディレクトリは、アプリケーション ファイルを提供するためのドキュメント ルートとして機能します。アプリケーションの HTML ページをアプリケーション ルートの下か、またはWEB-INF ディレクトリ以外のアプリケーション ルートのサブディレクトリに追加します。たとえば、Web アプリケーションが c:/apps/app1 にある場合、既定のトップ ページ ファイルは c:/apps/app1/index.html に配置されます。

JSP の追加

JSP を使用すると、HTML とスクリプト コードの組み合わせが含まれているテキスト ファイルから、サーブレットを作成できます。JSP (.jsp ファイル) は、クライアントから最初に要求されたときに、Java ソース コード ファイル (.java ファイル) に変換され、続いて Javaクラス ファイル (.class ファイル) にコンパイルされます。JSP 作成方法の詳細については、第 7 章を参照してください。

Web アプリケーションに JSP を追加するには、アプリケーションのルート ディレクトリか、または WEB-INF にあるディレクトリ以外のアプリケーション ルートの下のディレクトリに JSPをコピーします。既定のアプリケーションに JSP を追加する場合は、Web サーバーのドキュメント ルート ディレクトリに JSP をコピーします。このディレクトリが既定のアプリケーションのルートディレクトリとして機能しているためです。

JRun では、JSP の要求に応じて作成される .java および .class ファイルが、アプリケーションの WEB-INF/jsp ディレクトリに書き込まれます。

メモ
WEB-INF/jsp ディレクトリは JRun 固有のものであり、Java サーブレット API バージョン 2.2 の仕様書では定義されていません。ほかのアプリケーション サーバーでは、別の位置に.java および .class ファイルが書き込まれることもあります。

JRun は、実際には jsp という名前のサーブレットを使用して JSP を処理します。このサーブレット は、次のマッピングによって定義されるように、接尾辞 .jspを持つページを要求するすべての URL に応答します。

```
*.jsp = jsp
```

JMC を使用してほかのマッピングを作成することによって、jsp サーブレットの使用を無効にして JSP の要求を処理できます。別のサーブレットを使用して JSP を処理する方法の詳細については、第 10 章を参照してください。

Java サーブレットの追加

Java サーブレットは .class ファイルによって表されます。Web アプリケーションに Java サーブレットを追加する手順は、サーブレットの保存場所によって異なります。通常、サーブレットは次のいずれかの場所に保存されます。

- WEB-INF/classes に .class ファイルとして格納する。
- WEB-INF/lib に JAR ファイル内の .class ファイルとして格納する。
- 複数のアプリケーションによって共有されるクラスのディレクトリに格納する。

ディレクトリ WEB-INF/classes と WEB-INF/lib は、Web アプリケーションのクラスパスに自動的に組み込まれます。また、これらのディレクトリ内のすべての .class および JAR ファイルは再ロード可能です。共有クラスのディレクトリがアプリケーションのクラスパスに含まれていることを確認する必要があります。さらに、共有ディレクトリの中には再ロード可能なものと、そうでないものがあります。共有リソースのディレクトリの詳細については、66 ページの「Web アプリケーション間でのクラスの共有」を参照してください。

JRun で Java サーブレットを実行可能にする場合に最も重要な問題は、アプリケーション サーバーが WEB-INF や WEB-INF のサブディレクトリ内のファイルを直接操作できないことです。このため、サーブレットの .class ファイルを一般に JAR ファイルとして WEB-INF/classes や WEB-INF/lib に格納する場合、これをクライアントで利用可能にする方法が問題になります。

アプリケーションに Java サーブレットを追加する手順は次のとおりです。

1. サーブレットの .class ファイル、またはサーブレットの .class ファイルが含まれている JAR ファイルを、適切なディレクトリ (通常は、WEB-INF/classes または WEB-INF/lib) にコピーします。
2. JRun 管理コンソール (JMC) 内の [*アプリケーション名*] > [**サーブレット URL のマッピング**] プロパティを使用して、サーブレットを適切な Web アプリケーション内に登録します。

   サーブレットを登録すると、要求 URL のサーブレット クラス ファイルへのマッピングが設定されます。この登録情報はアプリケーションの web.xml ファイルに保存されます。

たとえば、ファイル SnoopServlet.class によって表されるサーブレットにアクセスするとします。まず、ファイル SnoopServlet.class を c:/apps/app1/WEB-INF/classes にコピーします。次に、次の表に示すサーブレット マッピングで、JMC を使用してサーブレットを登録します。

仮想パス / 拡張子	呼び出されるサーブレット
/Snoop	SnoopServlet

[仮想パス / 拡張子] フィールドでは、サーブレットに対応する URL を指定します。[呼び出されるサーブレット] フィールドでは、サーブレット名を指定します。サーブレット名は、.class 拡張子を除いたサーブレットのクラス ファイルの名前です。

サーブレットを登録すると、次の URL を使用して要求できます。

```
http://local_host/app1/Snoop
```

JRun ではそのサーブレットを処理し、その結果をクライアントに返します。

次の URL を使用してサーブレットに直接アクセスしようとするとエラーになります。

```
http://localhost/app1/WEB-INF/classes/SnoopServlet.class
```

この URL でエラーになるのは、JRun では WEB-INF の下にあるファイルを操作できないためです。

しかし、app1 のルート ディレクトリに SnoopServlet.class ファイルをコピーして、次の URL を使用してこのファイルにアクセスしようとすると次のようになります。

```
http://localhost/app1/SnoopServlet.class
```

この .class ファイルはアプリケーションのクラスパスに含まれるディレクトリにはないので、JRun では Web サーバーによってそのファイルをクライアントに返します。Web ブラウザでは .class ファイルの内容を表示する方法が不明なので、クライアントにファイルの保存先ディレクトリ名の入力を要求するプロンプトを表示します。しかし、ユーザはクライアントに .class ファイルを戻したいのではなく、JRun がサーブレットをロードして実行し、サーブレットの結果をクライアントに返す必要があると考えています。

invoker サーブレットの使用

JRun には、文字列 /servlet が含まれる URL を JRun invoker サーブレットに関連付ける暗黙的サーブレット マッピングが含まれています。invoker サーブレットを使用すると、サーブレットの .class ファイルを Web アプリケーションのクラスパス内の任意のディレクトリにコピーしたり、サーブレットを最初に登録しなくても参照できます。未登録のサーブレットを参照するには、次の形式の URL を使用します。

```
http://local_host/app1/servlet/<servlet class name>
```

このマッピングは次のような状況で役に立ちます。

- 開発段階やテスト段階で、サーブレットを処理するために登録する必要がありません。invoker サーブレットは、サーブレットのクラス名を使用して一時的なサーブレット登録を自動的に作成します。
- JRun の以前のリリースから移行する場合、このマッピングを使用して、WEB-INF/classes の下にあるサーブレットを、servlets ディレクトリに存在するかのように処理することで、作業の開始をより速やかに行うことができます。

注意

セキュリティとパフォーマンス上の理由から、常にすべてのサーブレットに対して明示的マッピングを定義し、invoker サーブレットには依存しないでください。運用アプリケーションでは、global.properties ファイルからマッピング /servlet=invoker の削除を検討する場合もあります。

タグ ライブラリの追加

JavaServer Pages バージョン 1.1 の仕様書には、タグ ライブラリに関するフレームワークが記述されています。開発者は、タグ ライブラリを使用して関連する機能セットを 1 つの HTML タグ セットにカプセル化できます。JSP では、これらのタグを使用して、ライブラリに組み込まれている機能を利用できます。たとえば、タグ ライブラリを作成してデータベース アクセスを簡略化したり、基本的な電子商取引の操作を実行できます。

タグ ライブラリは、**タグ**または**カスタム タグ**と呼ばれる 1 つ以上の**アクション**から構成され、関連する**タグ ハンドラ** クラスでコーディングされている処理が、それぞれのタグによって実行されます。ユーザは、各カスタム タグを定義し、タグ ハンドラをコーディングするとともに、タグ ライブラリ 記述子 (TLD) ファイルで各カスタム タグの機能 (属性を含む) を定義します。

カスタム タグでスクリプト変数を作成する場合、タグ拡張情報 (TEI) ファイルも作成する必要があります。TEI ファイルは、JSP コードで使用するスクリプト変数とそのスコープを定義する Java クラスです。TEI ファイルを使用して、変換時に属性を検証することもできます。

公開可能なタグ ライブラリでは、タグ ハンドラ、TLD ファイル、TEI クラス、およびその他のサポート クラスを JAR ファイルに組み込む必要があります。この JAR ファイルは、Web アプリケーションの `WEB-INF/lib` ディレクトリに配置する必要があります。

タグ ライブラリ の作成方法と使用方法の詳細については、第 22 章を参照してください。

EJB の追加

EJB は Web アプリケーションと関連付けられていませんが、Web アプリケーションのホストである JRun サーバーと関連付けられています。Web アプリケーションが EJB にアクセスできるようにするには、その EJB を Web アプリケーションのホストである JRun サーバーか、または Web アプリケーションにアクセス可能な JRun サーバーに公開しなければなりません。

JRun における EJB の開発手順と公開手順の詳細については、第 24 章を参照してください。

リソースの追加

追加可能なリソースには、イメージ ディレクトリ や Bean のほか、データ ベース ドライバなどのリソース用のクラス ファイルなどがあります。JRun によって実行される Web アプリケーションにリソースを追加するときに注意する点は、リソースを正しい位置に配置することだけです。Java `.class` ファイルや JAR ファイルによってリソースを提供する場合は、Web アプリケーションのクラスパス (通常は `WEB-INF/classes`) に含まれているディレクトリにファイルを配置する必要があります。クラスパスの定義については、64 ページの「Web アプリケーション クラスパスの決定」を参照してください。

イメージ ディレクトリなどその他のタイプのリソースについては、通常の Web アプリケーションと同じ方法で追加してください。

Web アプリケーションの公開

Web アプリケーションを配布する場合は、展開したディレクトリ構造で配布するか、または Web Archive (WAR) ファイルと呼ばれる 1 つの圧縮ファイルで配布できます。WAR ファイルには、拡張子 `.war` が付きます。

WAR ファイルには、すべてのディレクトリ構造とアプリケーションを定義するすべてのファイルが含まれます。WAR ファイルは、JAR ファイルと同じツールを使用して作成します。

WAR ファイルを使用すると 1 つのファイルで配布が行えるため、アプリケーションの配布が簡単になります。ただし、WAR ファイルに含まれる Web アプリケーションを実行する前に、ファイルを展開する必要があります。このため、JRun には WAR ファイルの展開に使用する Deploy ツールが含まれています。

公開用アプリケーションのパッケージ化

アプリケーションを公開するときは、次のような場合が考えられます。

- **標準的な Web アプリケーションのパッケージ化**　標準的な Web アプリケーションでは共有リソースを使用しません。または、別の Web アプリケーションのサーブレットや JSP を参照します。このタイプの Web アプリケーションは、1 つの WAR ファイルとしてパッケージ化して、公開できます。
- **共有リソースを持つ Web アプリケーションのパッケージ化**　66 ページの「Web アプリケーション間でのクラスの共有」では、Web アプリケーション間でクラスファイルを共有する方法について説明しています。Web アプリケーションで共有ファイルを利用する場合、アプリケーションを WAR ファイルとしてパッケージ化する前に、すべての共有クラスをアプリケーションのルート ディレクトリ のディレクトリ構造にコピーする必要があります。これによって、アプリケーションで必要なすべてのクラス ファイルを WAR ファイルに含めることができます。

  たとえば、ユーザのアプリケーションで、*JRun のルート ディレクトリ*`/servers/lib` にある共有タグ ライブラリを利用しているとします。この場合は、アプリケーションを WAR ファイルとしてパッケージ化する前に、アプリケーションのルート ディレクトリの下の `WEB-INF/lib` ディレクトリに共有ライブラリをコピーしてください。

- **EJB を使用する Web アプリケーションのパッケージ化**　EJB は JAR ファイルを使用して公開します。J2EE エンタープライズ アプリケーションをパッケージ化する場合、EJB の JAR ファイルおよび Web アプリケーションの WAR ファイルは、J2EE エンタープライズ アーカイブ (EAR) ファイルの一部としてパッケージ化されます。

JRun 内での Web アプリケーションの公開

JRun によって Web アプリケーションを開発できますが、ベンダから購入した Web アプリケーションや、ほかの Web アプリケーション サーバーを使用して開発された Web アプリケーションの公開アプリケーション サーバーとしても JRun を使用できます。

公開する Web アプリケーションを受け取る場合、通常は WAR ファイルを受け取ります。JRun で実行するアプリケーションを公開するには、JRun 管理コンソール (JMC) を使用します。

JMC では、アプリケーションを公開するために、次の操作を実行します。

1 WAR ファイルを展開します。

2 必要に応じて、JRun の設定値とプロパティを更新します。

3 アプリケーションのマッピングを設定して、Web アプリケーションで認識される URL にアプリケーションをバインドします。

Web アプリケーションの配布、WAR ファイルの作成、および JMC を使用して Web アプリケーションを公開する方法の詳細については、第 34 章を参照してください。

第 6 章

JRun によるサーブレット への要求のマッピング

この章では、JRun による要求の処理、および Web アプリケーションとサーブレットへの要求のマッピングについて説明します。

目次

- サーブレット マッピングの基本情報 80
- マッピング 80
- JRun によるファイルの提供 84

サーブレット マッピングの基本情報

JRun バージョン 3.1 は、Java サーブレット API バージョン 2.2 の仕様書に記述されている Web アプリケーション アーキテクチャを完全に実装しています。この実装には、仕様に準拠したサーブレットへの要求のマッピングが含まれています。Web アプリケーション アーキテクチャは、前のバージョンのサーブレット API から大きく変更されています。Web アプリケーションの詳細については、第 5 章を参照してください。

JRun は、サーブレット、JSP ファイル、HTML ファイル、およびテンプレートを呼び出すために、次の 2 種類のマッピングを使用します。

- **アプリケーション マッピング**　Web アプリケーションの URL をアプリケーションを含む物理ディレクトリに関連付けます。
- **サーブレット　マッピング**　サーブレットを、`/servlet` などの接頭辞、または `*.jsp` などの接尾辞に関連付けます。

マッピング

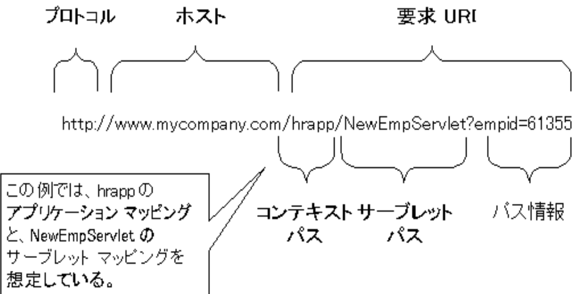
JRun サーバーで実行される各 Web アプリケーションには、1 つのアプリケーション マッピングと複数のサーブレット マッピングが含まれます。Web アプリケーションを最適な方法で使用するには、HTML ファイル、JSP、およびサーブレットに対する要求を処理するために JRun がどのようにアプリケーション マッピングとサーブレット マッピングを使用するかを理解する必要があります。

JRun サーバーには、次の 2 種類の Web アプリケーションが含まれています。

- **既定のアプリケーション**　各 Web サーバーには、既定の Web アプリケーションが含まれます。既定のアプリケーションは、JRun サーバーのほかの Web アプリケーションとは別個に使用されます。JRun は、常にクラスパスを使用してアクセスされる `.class` ファイルおよび明示的にマッピングされたリソースを除いて、Web サーバーのルート ディレクトリに関連する既定のアプリケーションのコンテンツである HTML や JSP ファイルなどを提供します。既定のアプリケーションは、JRun サーバーの `local.properties` ファイルで *`appname`*`.use-webserver-root=true` を設定して指定します。
- **追加の Web アプリケーション**　追加の Web アプリケーションは、JRun サーバーで指定されたアプリケーション URL にマッピングします。JRun は、既定以外のアプリケーションに対して Web アプリケーションのルートに関連する Web アプリケーションのコンテンツを提供します。Web アプリケーションのコンテンツには、サーブレット、JSP、HTML ファイル、イメージ、カスケード スタイル シート、およびその他のリソースが含まれます。

  複数の Web サーバーが 1 つの JRun サーバーに接続する場合、JRun サーバーには、複数の既定のアプリケーションが各 Web サーバーに 1 つずつ含まれます。JRun 接続モジュール (JCM) は、Web サーバーと既定のアプリケーションの関係を管理します。

JRun では、Java サーブレット API バージョン 2.2 の仕様書に準拠して、次の図に示すように URL がプロトコル、ホスト、ポート (オプション)、および要求 URI で構成されるように考慮されています。



アプリケーション マッピングおよびサーブレット マッピングは、要求 URI の異なる部分を使用して、どのサーブレットを呼び出すかを決定します。この要求 URI は次のコンポーネントに分割されます。

- コンテキスト パス `ContextPath` は、Web アプリケーション マッピングに関連付けられたパス接頭辞を指定します。Web サーバーの URL ネームスペースのルート ディレクトリ にある既定のアプリ ケーシ ョンの場合、コンテキスト パスは空の文字列になります。既定以外のアプリケーションの場合、コンテキスト パスは、スラッシュ (/) で始まりますが、スラッシュで終了しません。
- サーブレット パス `ServletPath` は、要求をアクティブにしたりサーブレット マッピングに一致する URL の部分です。このパスはスラッシュ (/) で始まります。
- パス情報 `PathInfo` には、要求パスの残りの部分が含まれます。

JRun がどのように要求 URI を、コンテキスト パス、サーブレット パス、パス情報に分割するかは、アプリケーション マッピングおよびサーブレット マッピングによって決まり ます。たと えば、アプリ ケーショ ン マッピングが `/hrapp`、サーブレット マッピングが `/NewEmpServlet`、要求 URI が /hrapp/NewEmpServlet?empid=61355 の場合、コンテキスト パスは `/hrapp`、サーブレット パスは `/NewAppServlet`、パス情報は `empid=61355` になり ます。また、アプリ ケーショ ン マッピングがなく 、サーブレット マッピングが `/hrapp/NewEmpServlet`、要求 URI が `/hrapp/NewEmpServlet/login` の場合、コンテキスト パスは空白文字、サーブレット パスは `/hrapp/NewEmpServlet`、パス情報は `/login` になります。詳細については、85 ページの「事例」を参照してください。

メモ

要求 URI とパス部分の URL エンコードが異なる場合を除き、次のステートメントは常に true になります。

```
RequestURI = contextpath + servletpath + pathinfo
```

サーブレットに渡される HttpServletRequest オブジェクトには、ContextPath、ServletPath、および PathInfo へのアクセスに使用可能なメソッドが含まれます。詳細については、224 ページの「javax.servlet.http」を参照してください。

アプリケーション マッピング

アプリケーション マッピングは、コンテキスト パスを Web アプリケーションの名前およびディレクトリ パスに関連付けます。これらのマッピングは、JMC を使用して管理します。アプリケーション マッピングは、JRun 内部の local.properties ファイルに記録されます。

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

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

JMC を使用したアプリケーション マッピングの定義の詳細については、『JRun セットアップ ガイド』を参照してください。

サーブレット マッピング

サーブレット マッピングは、サーブレットを URL パターンに関連付けます。URL パターンには、/MyServlet などの接頭辞、または *.jsp などの接尾辞を使用できます。指定された URL パターンに一致するサーブレット パスが要求 URI に含まれる場合、JRun は関連するサーブレットを呼び出します。

Web サーバーがページまたはサーブレットに対する要求を受信すると、JRun は、まず要求 URI のコンテキスト パスを JRun に定義されているアプリケーション URL マッピングに一致させて、Web アプリケーションを検索します。Web アプリケーションが見つかると、JRun はその Web アプリケーションのサーブレット マッピングを使用して、指定されたリソースを検索します。コンテキスト パスに一致するアプリケーション マッピングがない場合、JRun はその Web サーバーの既定のアプリケーションのサーブレット マッピングを使用して、リソースを検索します。

明示的サーブレット マッピングは JMC を介して定義および管理され、web.xml ファイルに記録されます。セキュリティを最大にするには、運用 Web アプリケーションの各サーブレットごとに明示的サーブレット マッピングを定義する必要があります。JMC を使用したサーブレット マッピングの定義の詳細については、『JRun セットアップ ガイド』を参照してください。

JRun は、次のような一連の暗黙的サーブレット マッピングも保持します。

- /servlet = invoker
- *.jrun = invoker
- *.jsp = jsp
- / = file (既定以外のアプリケーションのみ)
- *.shtml = ssifilter
- *.thtml = template

暗黙的サーブレット マッピングは、JMC で新しいマッピングを定義することにより、書き換えられます。たとえば、/servlet を LoginServlet または 404Servlet に関連付けることで、invoker サーブレットのサーブレット マッピングを変更できます。

メモ
暗黙的サーブレット マッピングは、JRun サーバーのすべてのアプリケーションによって共有され、local.properties ファイルに記録されます。

---

invoker サーブレットの使用

JRun には、/servlet を invoker サーブレットに関連付けるサーブレット マッピングが含まれています。このサーブレット マッピングにより、サーブレット パス /servlet を含む要求 URI は、invoker サーブレットによって処理されるようになります。invoker サーブレットは、JRun に対して定義されていないサーブレット用の汎用呼び出しメカニズムを提供します。このマッピングは次のような状況で役立ちます。

- このマッピングにより、開発およびテスト段階でサーブレット マッピングを明示的に定義する手間が省かれます。invoker サーブレットは、クラス名を使用して一時的なサーブレット登録を自動的に作成するからです。
- JRun の以前のリリースから移動する場合、このマッピングを使用して、WEB-INF/classes 以下のサーブレットを、servlets ディレクトリに存在するかのように処理することで、作業の開始をより速やかに行うことができます。

セキュリティとパフォーマンス上の理由から、すべてのサーブレットに対して明示的マッピングを定義して、運用システムの /servlet = invoker マッピングを書き換える必要があります。運用アプリケーションでは、場合によっては、常にエラーを返すカスタマイズしたサーブレットに /servlet のマッピングを関連付けるような方法も考えられます。たとえば、次のように設定します。

```
/servlet = 404servlet
```

JRun は invoker サーブレットを次のように使用します。

1. 要求 URI にアクセスします (例: /app1/servlet/SnoopServlet)。
2. コンテキスト パスを抽出します (例: /app1)。
3. サーブレット パスを抽出します (例: /servlet)。
4. パス情報からサーブレット名を抽出します (例: /SnoopServlet)。
5. ServletContext.getNamedDispatcher(*servletname*) を呼び出して、サーブレットを呼び出します。

メモ

JRun は、まず Web アプリケーションの `web.xml` ファイルを調べて、サーブレットが定義されているかどうかを確認します。一致するサーブレットが見つからない場合、JRun はアプリケーションのクラスパスに記載されたディレクトリ内でサーブレットを検索します。

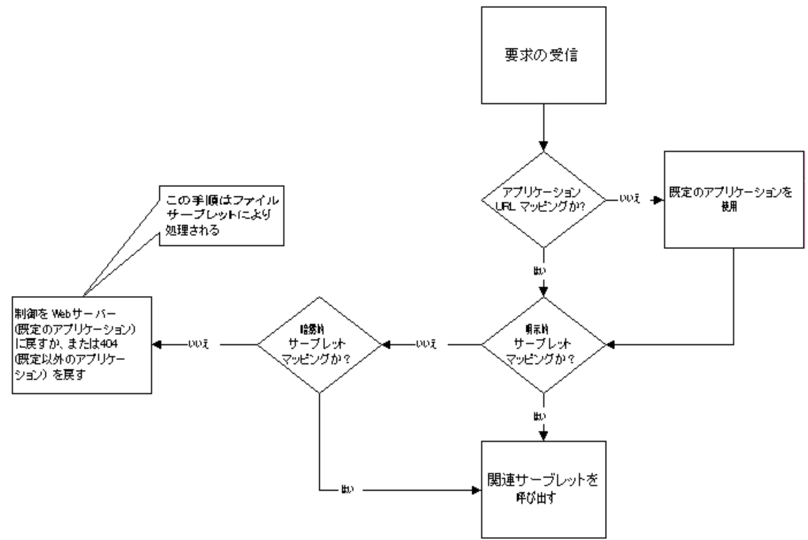
JRun によるファイルの提供

既定のアプリケーションは Web ルート (/) にマッピングするため、JRun は、Web サーバーに送信されるすべての要求を調べます。JRun では、高度なコーディング テクニックを使用して、パフォーマンスを低下させることなく JRun 以外の Web サーバーのファイルを提供できるようにしています。

次のセクションでは、JRun Web サーバーの対話について説明し、2 つのファイル サービス事例を記載します。

Web サーバーの対話

Web サーバーは、着信したすべての要求を JRun に渡します。JRun は、既定および既定以外のアプリケーション用のアプリケーション マッピングと、暗黙的および明示的サーブレット マッピングを使用して、次の図で示すように各要求を処理します。



各 Web サーバーは、既定の Web アプリケーションを個々に所有します。

アプリケーション URL マッピング、サーブレット マッピング、および Web サーバーのファイルの存在に応じて、JRun は次のいずれかの操作を行います。

- 関連するサーブレットまたは JSP を呼び出します。
- Web サーバーがファイルを検索および提供できるように、Web サーバーに制御を返します (既定のアプリケーションのみ)。
- エラーを返します。既定のアプリケーションでリソースが見つからない場合、Web サーバーは 404 を返します。既定以外のアプリケーションでリソースが見つからない場合、JRun は 404 を返します。

JRun は、既定のアプリケーションを除くすべてのクライアントにエラーを直接返します。したがって、Web サーバー用に定義された、404 エラーの処理用ページなどのカスタム エラー ページは使用されません。ただし、JRun アプリケーション用にカスタム エラー ページを定義することで、同じ機能を実装できます。

事例

次のセクションでは、2 つの事例を記載します。それぞれの事例でアクセスの例をいくつか示します。

各事例では、次の内容を説明します。

- 1 つまたは複数のアプリケーション マッピング
- 暗黙的サーブレット マッピング
- 明示的サーブレット マッピング

各例では、次の内容を記載します。

- 要求 URI
- JRun の応答
- 実際に解決されるドキュメント
- 処理順序

単一の既定の Web アプリケーション

この事例では、単一の既定の Web アプリケーションを使用します。既定の Web アプリケーションは、Web サーバーのルート (/) にマッピングします。この事例では、次の暗黙的サーブレット マッピングも使用します。

- `*.jsp = jsp`
- `/servlet = invoker`

次の表は、単一の既定のアプリケーションに対してファイルを提供するアクセスの例を示します。

要求 URI	JRun の応答	実際に解決されるドキュメント	処理順序
/index.html	スラッシュ (/) は既定のアプリケーションに相当しますが、既定のアプリケーションには対応するサーブレット マッピングがありません。JRun は、Web サーバーに制御を返します。	webroot/index.html	1 既定のアプリケーション 2 Web サーバー
/app1/index.html	app1 対するアプリケーション マッピングがないため、JRun は既定のアプリケーションを使用します。既定のアプリケーションには対応するサーブレット マッピングがありません。JRun は、Web サーバーに制御を返します。	webroot/app1/index.html	1 既定のアプリケーション 2 Web サーバー
/app1/foo/bar.jsp	app1 対するアプリケーション マッピングがないため、JRun は既定のアプリケーションを使用します。既定のアプリケーションには、*.jsp に対するサーブレット マッピングがあるため、JRun は jsp サーブレットを呼び出し、jsp サーブレットは、Web サーバーと通信してパスを解決します。	webroot/app1/foo/bar.jsp	1 既定のアプリケーション 2 jsp サーブレット 3 generatedname.class
/app1/snoop	app1 対するアプリケーション マッピングがないため、JRun は既定のアプリケーションを使用します。既定のアプリケーションには、対応するサーブレット マッピングがないため、JRun は要求を Web サーバーに返し、Web サーバーは 404 をクライアントに返します。	File not found (404) (この例を機能させるには、/app1/snoop に対してサーブレット マッピングを定義します。)	1 既定のアプリケーション 2 Web サーバー
/servlet/SnoopServlet	/servlet 対するサーブレット マッピングがないため、JRun は制御を invoker サーブレットに渡します。	サーブレットには適用されません。	1 既定のアプリケーション 2 invoker サーブレット 3 SnoopServlet

JRun サーバーに複数の Web アプリケーションが存在する場合

この事例では、次のアプリケーションを使用します。

- 既定の Web アプリケーションは、Web サーバーのルート (/) にマッピングします。
- App1 Web アプリケーションは、/app1 にマッピングします。この Web アプリケーションには、明示的サーブレット マッピング /snoop = SnoopServlet があります。

両方の Web アプリケーションで、次の暗黙的サーブレット マッピングを使用します。

- *.jsp = jsp
- /servlet = invoker

次の表は、複数のアプリケーションに対するファイルの提供の例を示します。

要求 URI	JRun の応答	実際に解決されるドキュメント	処理順序
/index.html	スラッシュ (/) は既定のアプリケーションに相当しますが、既定のアプリケーションには一致するサーブレット マッピングがありません。JRun は、Web サーバーに制御を返します。	webroot/index.html	1 既定のアプリケーション 2 Web サーバー
/app1 /index.html	JRun は App1 アプリケーションを使用して、Web アプリケーションのルートで見つかった index.html ファイルを提供します。	approot/index.html	1 App1 アプリケーション
/app1/foo /bar.jsp	JRun は App1 アプリケーションを使用します。App1 アプリケーションには、*.jsp に対するサーブレット マッピングがあるため、JRun は jsp サーブレットを呼び出し、jsp サーブレットはアプリケーション マッピングを使用して、パスを解決します。	approot/foo/bar.jsp	1 App1 アプリケーション 2 jsp サーブレット 3 generatedname.class
/app1/snoop	JRun は App1 アプリケーションを使用します。App1 アプリケーションには、/snoop に対するマッピングがあるため、JRun は制御を関連するサーブレット (SnoopServlet) に渡します。	サーブレットには適用されません。	1 App1 アプリケーション 2 SnoopServlet
/app1 /servlet/ SnoopServlet	JRun は App1 アプリケーションを使用します。App1 アプリケーションには、/servlet に対するマッピングがあるため、JRun は制御を invoker サーブレットに渡します。	サーブレットには適用されません。	1 App1 アプリケーション 2 invoker サーブレット 3 SnoopServlet

第 2 部

サーバー側スクリプトと JSP

サーバー側スクリプトは、Web クライアントに情報が返される前に、Web サーバーにより解釈されます。スクリプト解釈作業では、スクリプト内で参照されているサーブレットがすべて Web サーバーにより処理されます。

ここでは、Web サーバーを拡張するために JRun によってサポートされるサーバー側のスクリプトの種類をいくつか説明します。

JSP の作成 91

JSP の構文 111

JSP オブジェクト リファレンス 135

JSP のコンパイル 145

JSP でのカスタム タグの作成 155

JSP の例 169

JSP のアップグレード 181

サーバー側インクルード ファイルの使用 187

プレゼンテーション テンプレート 191

タグレット 197

第 7 章

JSP の作成

JSP によって、HTML とスクリプト コードの組み合わせを含むテキスト ファイルからサーブレットを作成できます。クライアントにより JSP が要求されると、そのページが Java サーブレットに変換されます。JSP のスクリプト部によって、クライアントにダイナミック コンテンツを返すことができます。また、JSP から Java サーブレットおよび EJB コンポーネントにアクセスできます。

この章では、サーブレットを JSP ファイルとして作成するための基礎について説明します。

目次

- JavaServer Pages の作成 92
- JSP ファイルの開発 96
- JSP の以前のリリースからのアップグレード 109

JavaServer Pages の作成

Web アプリケーションのコンポーネントとして JSP を含めることができます。JSP は、HTML と組み込まれたスクリプト コードを含んでいます。JSP は、一般的に、Web クライアントにダイナミック コンテンツを配布するために使用します。

JSP は「処理中に」Java サーブレットに変換されます。JSP にはファイル拡張子 `.jsp` が付けられます。JSP には次の要素の任意の組み合わせを含めることができます。

- テンプレート データ (一般的にはテキストと HTML タグ)
- JSP アクション
- JSP ディレクティブ
- JSP スクリプト コマンド

メモ
JSP アクションと JSP ディレクティブの構文は、JSP 仕様によって決定されます。JSP スクリプト コマンドの構文は、その JSP のために選択したスクリプト言語によって異なります。この章のほとんどの例では Java を使用しますが、JavaScript の例も示しています。

この章では、Allaire 社 の JavaServer Pages を紹介します。JSP の構文および JSP アクションと JSP ディレクティブの詳細については、第 8 章を参照してください。

JSP スクリプトの概要

スクリプトは、Web サーバーが実行する一連のコマンドです。たとえば、スクリプトは次のような処理を実行できます。

- **変数**に値を割り 当てる。変数は、名前が付いている保存場所で、値などのデータを割り当てることができます。
- ブラウ ザに変数の値を 送信するなど 、Web サーバーに対し て何かを 指示する 。ブラウザに値を送信する命令は**出力式**です。
- コマンドを組み合わせて**メソッド**を作成する。メソッドは、1 つの単位として動作する一連のコマンドおよびステート メントに名前を付けたものです。

JSP はスクリプト言語を定義しません。JSP を使用して HTML およびスクリプト コード を含んでいるド キュメント を作成できます。スクリ プト コード の言語としては、Java が多くの場合使用されますが、JavaScript を使用することもできます。

JRun は JSP を Java サーブレットに変換します。ページ変換の利点は、実際の Java サーブレットを作成する必要がないことです。JRun はこの複雑な作業を自動的に処理します。HTML についての一定の知識があれば、JSP を使用してアプリケーションを作成できます。しかし、Java サーブレット の柔軟性が必要な場合は、JSP 内から サーブレット API の全機能を使用できます。

はじめての JSP ページの作成

このセクションでは、初めての JSP を作成します。これは Hello World サーブレットを実装します。

1 新しいテキスト ファイルを作成して、次の行を入力します。

```
<html>
<head>
<title>Greetings</title>
</head>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

2 ファイルを Web サーバーのドキュメント ルートに保存し、greeting.jsp という名前を付けます。

このディレクトリとして、たとえば c:¥Inetpub¥wwwroot や JRun Web サーバーを使用している場合は <JRun home dir>¥servers¥default¥default-app を指定します。ただし、<JRun home dir> は JRun のホーム ディレクトリに対応し、default-app は既定の JRun Web アプリケーションです。

3 ブラウザで適当な URL、たとえば http://<your-host>/greeting.jsp を使用してドキュメントを要求します。

作成されたドキュメントは単純な HTML ドキュメントで、Java コードは含まれていません。このドキュメントの特徴は、ファイル拡張子が .jsp であることです。既定では、JRun は拡張子 .jsp が付いているファイルを認識し、それを Java サーブレット に変換します。この操作は、ページを要求したクライアント からはまったく見えません。

JRun が .jsp 以外の拡張子も JSP として認識できるようにマッピングを設定できます。詳細については、『JRun セットアップ ガイド』を参照してください。

4 greeting.jsp を変更して、組み込まれた Java コード を使用して簡単なループ処理を実行してみましょう。

greeting.jsp ファイルを次のように編集します。

```
<html>
<head>
<title>Greetings</title>
</head>
<body>
<%
  for (int i=0; i < 5; i++) out.println("<h1>Hello World!</h1>");
%>
</body>
</html>
```

ファイルを保存し、もう一度、ブラウザからこのファイルを要求します。JRun は <% %> タグに囲まれたスクリプト コードを認識します。この例では、スクリプト コードは Java です。このループにより、JSP out オブジェクトを使用してブラウザに「Hello World!」というメッセージが 5 回出力されます。

out オブジェクトは、JSP 内で利用できる多くのオブジェクトの 1 つです。このオブジェクトを使用して、出力をクライアントに返します。JSP 内で利用できるすべてのオブジェクトの説明については、第 9 章を参照してください。

JRun では JSP 内で Java 以外のスクリプト言語もサポートします。これらのファイル内で JavaScript を使用できます。一般的には、JavaScript は実行時に解釈されます。しかし、JRunは、JavaScript をコンパイルすることによってサーブレットのパフォーマンスを向上させます。

前の例のコードを JavaScript で表すと、次のようになります。

```
...
<%@  page language = "javascript" %>

<%
  var i;
  for (i=0; i < 5; i++) out.println("<h1>Hello World!</h1>");
%>
...
```

この例の最初の行は、page ディレクティブを使用してスクリプト言語を JavaScript に設定します。JSP の既定の言語は Java なので、このステートメントは必須です。

メモ
1 つの JSP 内で複数のスクリプト言語を使用することはできません。ページのスクリプト言語を定義した後、それを変更することはできません。

複数の HTML/Java ブロック

JSP を作成するとき、Java コードと HTML のブロックを組み合わせることができます。JRun は両方のタイプの入力を処理できます。以下はその例です。

```
<html>
<head><title>Mix me up</title></head>
<body>
<p>
<%
  out.println ("This list is generated by Java code");
%>
<h1> A list generated by Java</h1>
<ul>
<%
  for (int i = 1; i < 10; i++) {
    out.println ("<li>" + i);
  }
```

```
%>
</ul>
</body>
</html>
```

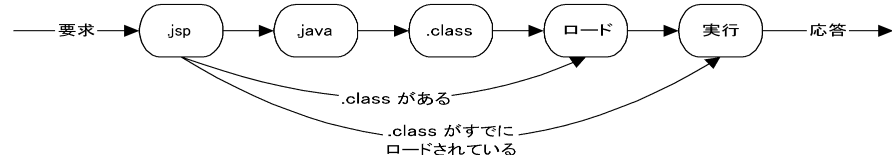
この例の 2 番目のスクリプト コードのブロックは、変数 *i* に保存されているリスト カウントと、それぞれの出力行が HTMLリスト の一部であることを示す HTMLタグを出力します。この例は、JSP の最も重要な機能の 1 つを示しています。それは、スクリプト内で、クライアントに返す HTML テキストを生成することです。

スクリプト 内から HTML テキスト を出力する主な目的の 1 つは、動的な HTML 出力を作成することです。動的な出力は、クライアントの名前や優先度のような、ページに渡された情報によって条件を付けるか、またはクッキーやその他のソースから取得した情報によって制御できます。

JSP から Java サーブレットへの変換

JSP を Java サーブレットに変換するプロセスを**ページ変換**と言います。最初に JSP が要求されると、JRun はファイルを解析して、Java ソース コード ファイルを出力します。Java ソース コード ファイルはサーブレット クラス ファイルにコンパイルされます。次に、そのサーブレット クラス ファイルがロードされ、実行されます。

次の図は、JRun が JSP の要求を受け取ったときに実行する手順を示しています。



次の手順は、JSP が要求されたときに JRun が実行する動作を示しています。

1 JRun は JSP (`.jsp` ファイル) を解析し、Java ソース コード (`.java` ファイル) を作成します。

2 Java ソース コードが Java サーブレット クラス (`.class` ファイル) にコンパイルされます。

3 Java サーブレットの `.class` ファイルが Web サーバー上のメモリにロードされます。

4 サーブレットが実行されます。

   サーブレットからのすべての出力がクライアントに返されます。JSP ファイルの既定の出力には `text/html; charset=ISO-8859-1` タイプがあります。このタイプでは、クライアントに直接送信できるように出力を設定します。

次回に JSP が要求されると、JSP が最後に変換されて以降変更されていない場合は、JRun は手順 4 だけを実行します。なぜなら、最初の要求の後サーブレットがメモリに残っているからです。既定では、JSP ファイルが最後の変換以降に変更された場合は、再変換が行われます。

公開されたアプリケーションで、JSP の自動変換を禁止することもできます。JSP の変換を無効にするには、JRun 管理コンソールを使用します。変換の無効化の詳細については、第 10 章を参照してください。

自動変換を無効にしている場合でも、システム コマンド ラインから JSPC コンパイラを起動することによって JSP を変換できます。この場合、JSPC コンパイラを使用して JSP を `.class` ファイルに変換し、新しい `.class` ファイルをアプリケーションのために適当な場所にコピーし、古い `.class` ファイルを置換します。

JSP ファイルの開発

このセクションでは、JSP で実行できるいくつかの基本的タスクについて説明します。ここで示している例は Java ですが、JSP では Java 以外のスクリプト言語も使用できます。その場合は、それに対応してコード例を変更してください。

このセクションでは、次のタスクについて説明します。

- 「JSP の保存」96 ページ
- 「変数の宣言」97 ページ
- 「JSP への条件ロジックの追加」98 ページ
- 「式の使用」98 ページ
- 「JSP オブジェクトの使用」99 ページ
- 「JSP オブジェクトとパラメータおよび属性の使用」100 ページ
- 「include の実行」101 ページ
- 「別の JSP の呼び出し」103 ページ
- 「JSP 出力のバッファ」105 ページ
- 「タグ ライブラリの使用」105 ページ
- 「エラーの処理」106 ページ
- 「JSP コンパイラの使用」108 ページ

JSP の保存

JSP を Web サーバーのドキュメントのルート ディレクトリに保存します。たとえば、IIS を使用している場合、このディレクトリは `c:¥Inetpub¥wwwroot` です。default JRun サーバーに JRun Web サーバーを使用している場合、このディレクトリは `<JRun home dir>¥servers¥default¥default-app` です。ここで、`<JRun home dir>` は JRun のホーム ディレクトリ 、`default-app` は既定の JRun Web アプリ ケーションを表します。

変数の宣言

JSP は、ほかのスクリプト 作成機能と 同様に、変数宣言を受け付けます。次の例で示すように、変数を定義してから再び割り当てることができます。

```
<html>
<head><title>Using variables</title></head>
<body>
<p>
<% int myVar = 5; %>
<b> <% out.println ("Value of myVar:" + myVar); %> </b>
<p>
<%
  myVar = 2;
  out.println("Value of myVar again:" + myVar);
%>
</body>
</html>
```

myVar 変数は、この変数が宣言されている JSP 内でだけアクセスできます。

ページ内で変数を再度割り当てることはできますが、名前自体は 1 回しか宣言できません。次の例では、変数が正しく使用されていません。この例では、myVar が最初の Java ブロック内で int として宣言され、3 番目の Java ブロックで再び int として宣言されているため、コンパイラ エラーが発生します。

```
<html>
<head><title>Using variables</title></head>
<body>
<p>
<% int myVar = 5; %>
<b> <% out.println ("Value of myVar:" + myVar); %> </b>
<p>
<%
  int myVar = 2;  //ここに間違いがあります
  out.println("Value of myVar again:" + myVar);
%>
</body>
</html>
```

JSP への条件ロジックの追加

次の例では、JSP 内で条件ステートメントを使用しています。

```
<html>
<head><title>Account Balance</title></head>
<body>
<p>
<% double accountBalance = 1.00; %>
Your Current Balance:<% out.println( accountBalance ); %> <br>
<% if(accountBalance <= 1.00) { %>
  <b> Get a Job.</b> <br>
<% } %>
</body>
</html>
```

この例では、変数 accountBalance の値を出力します。accountBalance が 1ドル以下である場合、次のステートメントはユーザに " Get a job" と指示します。ステートメントを変更して、accountBalance を調整できます。

条件ステートメントでは、if ステートメントを閉じるためにブロック <% } %> を使用します。この構文によって、条件が true である場合に HTML" Get a job
" が表示されます。条件を 1 つの Java ブロックだけで表す場合は、次の構文でも同じ結果が得られます。

```
<% if(accountBalance <= 1.00) out.println("<b>Get a job.</b> <br>"); %>
```

この例では、ステートメント " Get a job
" がドキュメント内で HTML テキストとして指定されるのではなく、スクリプト コードに含まれます。どちらの方法でもブラウザは同じようにメッセージを受け取り、表示します。

式の使用

これまでの例では、変数値を表示するために out.println() メソッドを使用しました。しかし、このメソッドを使用すると、値を表示したときにドキュメントが煩雑になる場合があります。式要素を使用すると、out.println() を使用せずに評価されたステートメントを表示でき、JSP が読み取りやすくなります。

次の例は、式要素の使用法を示しています。

```
<html>
<head><title>Account Balance</title></head>
<body>
<p>
<% double accountBalance =1.00; %>
Your Current Balance:<%=accountBalance %> <br>
<% if(accountBalance <= 1.00) { %>
  <b> Get a Job.</b> <br>
<% } %>
</body>
</html>
```

上の例は、前の条件ステートメントの例を変更したものです。この例では、<% out.println(accountBalance); %> の代わりに式 <%= accountBalance %> を使用します。このステートメントは、out.println() を使用せずに、式の値を表示します。

JSP オブジェクトの使用

既定では、JRun は JSP 内で使用するいくつかの JSP オブジェクトを作成します。これらの JSP オブジェクト は、JSP 仕様、Java サーブレット API、またはコア Java ライブラリによって定義されたオブジェクトのインスタンスです。

JSP 内で使用するすべてのスクリプト言語は、JSP オブジェクトにアクセスするのに必要です。これらのオブジェクトへのアクセスとは、オブジェクトのメソッド、すなわち関数を呼び出してオブジェクト内に保存されているデータにアクセスすることです。

これらのオブジェクトを使用して JSP 内から基本的なタスクを実行できます。次のようなオブジェクトがあります。

- `request` オブジェクト　クライアントから JSP に送信された HTTP 要求。HTTP 要求には要求ヘッダ内のすべての名前 / 値ペアが含まれます。
- `response` オブジェクト　JSP によって出力される HTTP 応答。`response` オブジェクトを使用してクッキーなどの情報をクライアントに返すことができます。
- `out` オブジェクト　クライアント に返される出力ストリーム JSP によって出力される HTML テキストは、一般的には `out` オブジェクトに書き込まれ、クライアントによって解釈されます。

たとえば、次の JSP は `out` オブジェクトを使用してクライアントに出力を返します。

```
<html>
<head><title>Using variables</title></head>
<body>
<p>
<% int myVar = 5; %>
<b> <% out.println ("Value of myVar:" + myVar); %> </b>
<p>
</body>
</html>
```

この例では、出力情報にテキストだけが含まれます。しかし、クライアントに返されるすべての出力は、クライアント の HTML ブラウ ザによって解釈されます。したがって、このテキスト出力内に HTML タグを含めることができます。

たとえば、出力行を書き直して、返された出力に「Heading 2 のテキストである」とマークを付けることができます。

```
<b> <% out.println ("<h2>Value of myVar:" + myVar + "</h2>"); %> </b>
```

クライアントから JSP に HTTP 要求の一部として送信されたデータにアクセスするには、JSP `request` オブジェクトを使用します。たとえば、要求には HTML フォームから JSP に渡されたデータを含めることができます。フォーム データは HTTP 要求の名前 / 値ペアとして JSP ページに送られます。この情報にアクセスするには、`request` オブジェクトとそのオブジェクトのメソッドを使用します。

また、パラメータを JSP の要求 URL の一部として、JSP に渡すこともできます。たとえば、JSP を要求するために次の URL を使用し、この要求と合わせて 2 つのパラメータを渡すことができます。

http://localhost/my.jsp?fName=Bob&lName=Smith

request オブジェクトの getParameter メソッドを使用して、フォームまたは要求 URL から JSP に渡されたパラメータを取得できます。

```
<%
  String firstName = request.getParameter("fName");
  String lastName = request.getParameter("lName");
  out.println("Welcome " + firstName + " " + lastName);
%>
```

第 9 章では、JRun によってサポートされているすべてのオブジェクトについて説明しています。

JSP オブジェクトとパラメータおよび属性の使用

多くの JSP オブジェクトは、パラメータ、属性、またはその両方の形式でオブジェクト内に保存されているデータにアクセスするためのメソッドを含んでいます。JSP オブジェクトを使用するとき、どのような場合にパラメータおよび属性を使用するかを知っている必要があります。

パラメータは常に文字列として JSP オブジェクトに保存されます。パラメータの主な用途は、クライアントの要求の中でクライアントからサーバーへデータを渡す、または応答の中でサーバーからクライアントへデータを渡すことです。

たとえば、クライアントがフォームを送信するとき、すべてのフォーム データが request オブジェクト内の名前 / 値ペアとしてサーバーに送られます。名前はパラメータ名に対応します。値はパラメータ値を含む文字列です。JSP 内では、request オブジェクトの getParameter メソッドを使用してパラメータにアクセスします。

次の例では、request オブジェクト を使用して JSP への HTTP 要求に含まれている 2 つのパラメータ *fName* および *lName* の値を取得し、次に out オブジェクトを使用してこれらの値をクライアントに渡します。

```
<%
  String firstName = request.getParameter("fName");
  String lastName = request.getParameter("lName");
  out.println("Welcome " + firstName + " " + lastName);
%>
```

属性は、一般的には JSP や Java サーブレットのようなサーバー側のコンポーネントの間で情報を伝達するために使用するデータ タイプです。たとえば、ある JSP から別の JSP を呼び出すとき、呼び出し側の JSP はアクセス先のページに渡す情報を request または session オブジェクト内の属性として指定できます。

属性は名前 / 値ペアとして保存されます。名前は属性名に対応し、値は Java オブジェクトのインスタンス java.lang.Object として保存されます。これがパラメータと属性の主な違いです。パラメータは常に文字列として保存され、属性は Java オブジェクトとして保存されます。

たとえば、JSP の session オブジェクトに属性 *fName* および *lName* が含まれる場合、次のコードを使用してそれにアクセスできます。

```
<%
  String firstName = (String) session.getAttribute("fName");
  String lastName = (String) session.getAttribute("lName");
  out.println("Welcome " + firstName + " " + lastName);
%>
```

この例では、キャストを使用して getAttribute の戻り値を String に変換します。このキャスト が必要になるのは、getAttribute が常に java.lang.Object のタイプのオブジェクトを返すからです。このキャストが返されたオブジェクトを送信先のフォーマットに変換します。この場合は、文字列として返されます。

属性を使用すると、サーバー側のアプリケーションをより柔軟に開発できます。なぜなら、パラメータを使用する場合と違って、文字列以外のオブジェクトを保存および取得できるからです。属性を使用すると、どのようなタイプのオブジェクトでも保存および取得でき、それらのオブジェクトをアプリケーションのコンポーネントに渡すことができます。

include の実行

include ディレクティブを使用して、解析時にほかのファイルを JSP ファイルにインクルードできます。一般的には include ディレクティブの引数として、インクルードファイルの名前を指定する文字列リテラルを指定します。include タグの構文は次のとおりです。

```
<%@ include file="path" %>
```

path を指定したファイルのパスに設定します。指定したファイルの内容が、JSP ファイルのこのステートメントの位置に挿入されます。

JSP で、include ディレクティブを使用したとき、1 つの JSP が作成されます。これは、1 つの JSP を別の JSP にインクルードした場合でも同様です。JRun は 1 つの JSP とそれに対応する 1 つのサーブレットを作成します。これはもとの 2 つの JSP の内容を含んでいます。

JRun はインクルードされたすべてのファイルについて、実行時に依存チェックを実行します。インクルードしたファイルがメモリにロードされた後でインクルードされたファイルが変更された場合、インクルードしたファイルは次に要求されたときに再変換されます。

次の例では、JSP がヘッダ情報を含む別の JSP をインクルードします。JSP でこの構造を使用して各ページの外観を統一できます。また、この構造を使用するとヘッダ情報の変更が簡単になります。なぜなら、一度変更するだけで済むからです。

```
<html>
<%@ include file="my_header.jsp" %>
<%-- 残りの JSP --%>
...
</body>
</html>
```

ここで my_header.jsp を定義できます。この JSP はヘッダ情報を含んでいます。

```
<head>
<title>Greetings</title>
</head>

<body bgcolor="white" style="font-family:Arial; font-weight:medium;
        font-style:normal">

<center>
<table width=80%>
<tr>
<td><img src="logo.gif" width=200 height=21 alt="Example Include"
        border="0"><P></td>
<td><H1><font color="#336699">Greetings</font></h1></td>
</tr>
</table>
</center>
```

このページは、タイトルを定義し、body タグの既定の色を設定し、ロゴと単語 URL " Greetings" を含むテーブルを作成します。

前の例は、静的なヘッダ ファイルを示しています。したがって、ヘッダ ファイルはそれを含む JSP に関わりなく同じ情報を表示します。しかし、ヘッダ ページにそれをインクルードする JSP から情報を渡すことによって、ヘッダ ページをより柔軟に使用できます。

JSP request オブジェクトを使用して、インクルードされた JSP に情報を渡すことができます。たとえば、入力としてページのタイトルを定義する属性を取るヘッダ JSP を作成できます。インクルード するページは、request オブジェクト 内で、次のよう にこの属性を設定できます。

```
<html>
<%request.setAttribute("title", "Greetings"); %>
<%@ include file="my_header.jsp" %>
</body>
</html>

<%-- 残りの JSP --%>
...
```

この例では、パラメータではなく属性を使用して、インクルードされるページに値を渡します。属性では、JSP 間でオブジェクト を渡すことができますが、パラメータでは文字列のみを渡すことができる点に注意してください。

ヘッダ ファイル my_header.jsp は、次のようにこの属性にアクセスできます。

```
<html>
<head>
<title><%= request.getAttribute("title")%></title>
</head>
```

属性を処理する機能を追加すれば、インクルードされるファイルを柔軟に使用でき、より広い用途に使用できます。

別の JSP の呼び出し

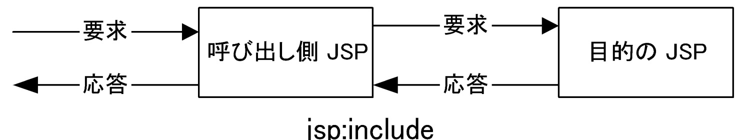
JSP から別の JSP を呼び出すことによってモジュール式 JSP を開発し、それを使用して複雑なアプリケーションを作成できます。JSP から別の JSP を呼び出すとき、2 つの呼び出し方法のいずれかを選択できます。

1 呼び出されたページが処理を完了してから、呼び出し側のページに制御を戻す。この呼び出しを行うには、`jsp:include` アクションを使用します。

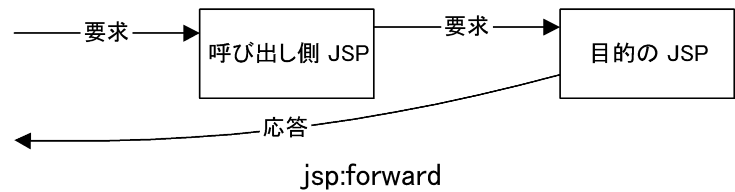
2 呼び出し側のページが呼び出されたページに制御を渡して終了する。この場合、呼び出されたページは呼び出し側のページに制御を戻しません。この呼び出しを行うには、`jsp:forward` アクションを使用します。

`jsp:include` および `jsp:forward` アクションの詳細については、124 ページの「アクション」を参照してください。

これらの 2 つの呼び出し操作を次の図に示します。



jsp:include



jsp:forward

メモ

呼び出されたページをロードした後に変更した場合、そのページを呼び出したときに JRun はそれを再変換します。しかし、呼び出されたページが変更された場合は、呼び出し側の JSP を再変換しません。

既定では、JRun は JSP からクライアントに送信された出力データをバッファに入れます。`jsp:include` または `jsp:forward` アクションを呼び出すと、JRun は JSP の出力バッファをフラッシュします。この出力バッファの詳細については、105 ページの「JSP 出力のバッファ」を参照してください。

次の例は、JSP から別の JSP を呼び出す方法を示しています。また、呼び出された JSP に属性を渡す方法についても説明しています。

1 新しいテキスト ファイルを作成して、次の行を入力します。

```
<% request.setAttribute("Greeting", "Hello World"); %>
<jsp:include page="b.jsp" flush="true"/>
```

ファイルを Web ドキュメント ルートに a.jsp の名前で保存します。

この例では、JSP request オブジェクトを使用して属性 Greeting と値 "Hello World" を渡します。この属性は、b.jsp 内の request オブジェクトから取得できます。request オブジェクトの詳細については、第 9 章を参照してください。

2 b.jsp ファイルを作成します。また、このファイルは Web サーバーのドキュメント ルートに保存する必要があります。b.jsp に次のように入力します。

```
The greeting from b is:<%=request.getAttribute("Greeting")%>
```

この例では、ページ間でデータを受け渡しします。属性の値は文字列に限定されません。どのようなタイプのオブジェクトでも属性として渡すことができます。

Greeting を request オブジェクト内の属性として渡す代わりに、jsp:param アクションを使用してそれをパラメータとして渡すこともできます。この場合、パラメータは呼び出された JSP に文字列として渡されます。この処理は、呼び出された JSP が HTTP post メソッドによって要求された情報を受け取る場合と同様です。

request オブジェクト内でパラメータを使用して b.jsp に情報を渡すことによって、b.jsp をクライアント からの要求に直接応答できるよう にするか、または別の JSP から呼び出せるように構成できます。

次に示す a.jsp は、jsp:param を使用して Greeting パラメータを渡します。

```
<jsp:include page="b.jsp" flush="true">
  <jsp:param name="Greeting" value="Hello World" />
</jsp:include>
```

b.jsp 内で、request.getParameter メソッドを使用してこのパラメータにアクセスします。

```
The greeting from b is:<%=request.getParameter("Greeting")%>
```

呼び出された JSP は、呼び出し側のページと同様に、JSP out オブジェクトを使用してクライアントにデータを返すことができます。しかし、呼び出し側の出力がバッファに入っている場合、呼び出しの前にバッファがフラッシュされます。このフラッシュによって、呼び出されたページは応答のヘッダを設定できなくなります。バッファの詳細については、106 ページの「エラーの処理」を参照してください。

また、呼び出されたページは、request オブジェクトを使用して呼び出し側ページに情報を返すことができます。たとえば、呼び出されたページは何かの値を決定して、setAttribute メソッドを使用してそれを request オブジェクトに書き込むことができます。呼び出し側ページに制御が戻ったとき、呼び出し側ページは request オブジェクトの getAttribute メソッドを使用してその情報にアクセスできます。

JSP 出力のバッファ

既定では、JRun は JSP からクライアントに送信された出力データをバッファに入れます。バッファを使用しているため、応答ヘッダ情報およびその他の出力はバッファがフラッシュされるまではクライアントに送信されません。このフラッシュは、次のいずれかが生じたときに行われます。

- JSP から `jsp:include` アクションによって別の JSP を呼び出した。
- バッファが満杯になった。JRun はバッファが満杯になると自動的にバッファをフラッシュします。
- `out` オブジェクトの `flush` メソッドを呼び出した。

また、`jsp:forward` アクションを呼び出すか、`response` オブジェクトの `redirect` メソッドを使用して要求を転送した場合にもバッファは消去されます。このルールの唯一の例外は、順方向の JSP によって設定されたクッキーが破棄されず、クライアントに送信されることです。バッファが無効になっていても要求を転送できますが、それはまだクライアントに何も返していない場合に限られます。

バッファを使用しているため、ヘッダに依存する操作は、flush メソッドを実行してヘッダをクライアントに送信するまでは無効になります。

メモ
バッファを無効にするか、または JSP の出力バッファのサイズを設定するには、JSP ディレクティブを使用します。`page` ディレクティブの詳細については、第 8 章を参照してください。

タグ ライブラリの使用

JSP は JSP タグと、一般的には HTML であるテンプレート テキストを含み、オプションとして**タグ ライブラリ**にあるカスタム タグを含めることができます。タグ ライブラリを使用すれば、自分の JSP、社内のほかのユーザ、または顧客が使用するカスタム タグを実装することによって、使用できるタグのセットを拡張できます。

JSP で使用するタグ ライブラリを宣言するには、JSP で `taglib` ディレクティブを使用します。ライブラリを識別するには、`taglib` ディレクティブ内で、タグ ライブラリへのパスと、ページ内で使用するタグ接頭辞を指定します。たとえば、次のステートメントでは、接頭辞 `myTags` によって参照されるタグ ライブラリが定義されます。

```
<%@ taglib uri="/myApp/appTags" prefix="myTags" />
```

タグ ライブラリが見つからない場合、JRun では致命的な変換エラーが発生します。JSP 内で接頭辞 `myTag` を使用する別のタグ ライブラリを定義した場合にもエラーが発生します。

taglib ディレクティブの後で、接頭辞 myTags を使用してタグ ライブラリ内のタグを参照できます。次のステートメントでは、タグ ライブラリ内の coolTag を使用します。

```
<myTags:coolTag>
...
</myTags:coolTag>
```

カスタム タグのフォームは、標準 HTML または JSP タグのどのようなフォームでもかまいません。言い換えると、1 つ以上の必須パラメータまたはオプション パラメータを取ることができます。また、タグ本文を指定することもできます。一般的に、タグ ライブラリはその中に含まれているタグを記述する何らかのドキュメントを含んでいます。

タグ ライブラリの作成

JSP でカスタム タグを使用できますが、Java でカスタム タグを開発するにはサーブレット API バージョン 2.2 仕様を使用します。第 22 章 にはカスタム タグおよびタグ ライブラリの作成についての情報が含まれています。

エラーの処理

JSP のエラーは、JSP ライフサイクル内の 2 つのポイントで発生します。

- JSP ソース ファイルから Java クラス ファイルへの変換
- JSP による要求の処理

このセクションでは、そのようなエラーを処理する方法について説明します。

変換エラー

JSP の JSP ソース ファイルから Java クラス ファイルへの変換は、まず Web サーバーがこのファイルに対する最初の要求を受け取ったときに行われます。その後は、ページの最後の変換以降に JSP ソース コード ファイルが変更されたことが検出されたときに変換が行われます。

メモ
JRun 管理コンソールから、それぞれの JSP について、変更された JSP の再変換を有効または無効にすることができます。詳細については、『JRun セットアップ ガイド』を参照してください。

変換が失敗した場合、クライアントの要求が失敗し、対応するエラー メッセージが返されます。たとえば、変換エラーを検出すると、エラー ステータス コード 500 (サーバー エラー) を返します。

要求処理エラー

クライアント 要求の処理中に、JSP の `.class` ファイル、または JSP の `.class` ファイルから 呼び出さ れたコ ード 内で実行時エラ ーが発生するこ と があり ます。要求処理エラーは、Java プログラミング言語の例外メカニズム、および JSP の `exception` オブジェクトを使用してエラーを知らせることによって実装されます。

メモ
エラーの表現フォーマットは、その JSP のために選択したスクリプト言語から独立しています。

要求処理エラーは、それが生成された JSP で捕捉および処理されます。しかし、`page` ディレクティブを使用して**エラー ページ**を指定することもできます。これは、例外を処理する JSP です。この場合、JSP から 捕捉さ れなかった 例外が返さ れたとき 、例外とクライアント要求がエラーページに転送されます。エラー ページを指定しない場合、捕捉さ れなかった 例外が送ら れると JRun はクライアント にエラ ー ステータス コード 500 (サーバー エラー) を返します。

エラー ページとして使用する JSP は、JSP の `page` ディレクティブを使用して `isErrorPage` 属性を設定する必要があります。エラーを生成した JSP は、そのエラーをエラー ページへ転送するとき、エラー ページの JSP `exception` オブジェクトを生成されたエラーに設定します。

`page` ディレクティブの詳細については、116 ページの「page ディレクティブ」を参照してください。`exception` オブジェクトの詳細については、第 9 章を参照してください。

次の例では、errhand.jsp という名前が付けられたエラー ページに例外を転送する JSP を作成します。存在しない JSP をインクルードしようとすると errortest.jsp ページで例外が発生します。errortest.jsp の定義は次のとおりです。

```
<%@ page errorPage="errhand.jsp"%>
<html>
<head><title>Error Test Page</title></head>
<body>
<-- 存在しないページをインクルードすることによって例外を発生させる -->
<jsp:include page="xxxxx.jsp" flush="true"/>

</body>
</html>
```

errhand.jsp の定義は次のとおりです。

```
<%@ page isErrorPage="true" %>
<P>
<HR>There was an error.
<P>

<%-- エラー メッセージを取得し、出力します。--%>
<B>ERROR:</b><BR>
<%
  String sErrMessage = exception.getMessage();
  out.println(sErrMessage);
%>

<%-- 例外の記述を取得し、出力します。--%>
<B>DESCRIPTION:</b><BR>
<%
  String sErrDescr = exception.toString();
  out.println(sErrDescr);
%>

</body>
</html>
```

JSP コンパイラの使用

JSP コンパイラは、JRun が JSP を Java クラス ファイルにコンパイルするために使用する Java ツールです。JRun には、このコンパイラの 2 つのバージョンが含まれています。最初のバージョン、JSP コンパイラは、JRun が JSP へのクライアント要求を処理するときにコンパイルを実行します。2 番目のバージョンの JSPC コンパイラは、JSP をオフラインで、言い換えると Web サーバーのコンテキストの外でコンパイルできるコマンドライン ツールです。

これらのコンパイラの詳細については、第 10 章を参照してください。

JSP の以前のリリースからのアップグレード

JRun バージョン 3.1 は JSP バージョン 1.1 仕様を実装しています。しかし、JSP 仕様の以前のバージョン (0.92、1.0 など) に対応して作成された JSP をバージョン 1.1 に対応するようにアップグレードできます。JSP を JSP の以前のバージョンからアップグレードする方法については、第 13 章を参照してください。

第 8 章

JSP の構文

JSP はテンプレート データ (通常、テキスト と HTML タグ) と JSP 要素により 構成されます。JSP 要素は、サーブレット に変換され、Web サーバーで実行されます。JSP 仕様では、3 種類の JSP 要素が定義されています。

- **ディレクティブ要素** JSP ファイルからサーブレットを作成するためのプロパティを表します。
- **スクリプト要素** オブジェクトを操作し、計算を実行します。
- **アクション要素** オブジェクト の使用、修正、または作成を行ったり 、ページの出力ストリームに書き込みます。

この章では、JSP の基本構文と 、この 3 種類の JSP 要素で使用される構文について説明します。

目次

- JSP 1.1 仕様と JRun の互換性 112
- JSP の基本構文 112
- ディレクティブ 115
- スクリプト要素 122
- アクション 124

JSP 1.1 仕様と JRun の互換性

JRun により実装されている JSP 構文は、JavaServer Pages バージョン 1.1 仕様に完全に準拠しています。したがって、JavaServer Pages 仕様に準拠した JSP はすべて、JRun のページ変換環境と互換性があります。

このセクションで説明する内容は、Sun Microsystems 社の JavaServer Pages バージョン 1.1 仕様に基づいています。この仕様は http://java.sun.com/products/jsp から入手可能です。

ただし、Allaire では、`global.jsa` ファイルの使用など、追加機能をサポートするために JSP 仕様が拡張されています。これらの拡張機能の詳細については、520 ページの第 42 章「JRun 拡張機能の使用」を参照してください。

JSP の基本構文

このセクションでは、次の内容を含む JSP の基本構文について説明します。

- 「JSP テンプレート テキストの挿入」112 ページ
- 「空白文字の使用」113 ページ
- 「開始タグと終了タグの配置」113 ページ
- 「属性値の引用」113 ページ
- 「エスケープ文字」113 ページ
- 「コメントの挿入」114 ページ
- 「JSP における相対 URL の指定」115 ページ

JSP テンプレート テキストの挿入

JSP には通常、HTML テキストと JSP 要素の両方が使用されます。JSP 要素は JRun により解釈されます。JSP 要素からの出力はすべて、HTML テキストとともにクライアントに返されます。

JSP では、テンプレート テキストは JSP 要素の外側にあるテキストなので、JRun によって解釈されることはありません。テンプレート テキストは、変更されることなく、クライアントに直接返されます。JSP にある HTML テキストは、すべてテンプレートテキストと見なされます。

たとえば、<% タグと %> タグの間にある Java コードを除いて、次のページにあるものはすべて、テンプレート テキストと解釈されます。

```
<html>
<head>
<title>Greetings</title>
</head>
<body>
<% for (int i=0; i < 5; i++) out.println("<h1>Hello World!</h1>"); %>
</body>
</html>
```

空白文字の使用

HTML では通常、空白文字は重要ではありません。JSP ファイルのテンプレート コードに含まれる空白文字はすべて、JSP ファイルに入力されているとおりにクライアントに返されます。

開始タグと終了タグの配置

開始タグと終了タグ、およびこれらのタグで囲まれた本文を持つ JSP 要素については、同じファイルにこの 2 つのタグを入れる必要があります。開始タグと終了タグを別々のファイルに入れることはできません。

たとえば、JSP スクリプトレットの構文は <% *scriptlet* %> です。開始タグ (<%) と終了タグ (%>) は両方とも同じファイルになければなりません。

属性値の引用

JSP 要素に対する属性値はすべて、一重引用符または二重引用符を使用して引用する必要があります。たとえば、次の例にある `page` 要素により、属性 `contentType` を使用して、JSP の出力の MIME タイプが `text/plain` に設定されます。

```
<%@  page contentType = "text/plain" %>
```

属性値自体に同じタイプの引用符 (一重引用符または二重引用符) が含まれている場合は、引用符の前にエスケープ文字 (¥) を付けます。エスケープ文字の次に来る文字は、JSP パーサーでは無視されます。属性の引用符には、次のエスケープ シーケンスを使用します。

- ' は ¥' とエスケープします。
- " は ¥" とエスケープします。

また、引用符に HTML 文字参照を使用することもできます。たとえば、属性値では二重引用符の代わりに、文字参照 `"` を挿入します。

エスケープ文字

属性値の引用符をエスケープし、さらに JSP の別の領域では次のような文字をエスケープする必要があります。

- スクリプト要素

  定数 %> は %¥> でエスケープします。

- テンプレート テキスト

  定数 <% は <¥% でエスケープします。

- 属性値

  %> は %¥> でエスケープします。

  <% は <¥% でエスケープします。

コメントの挿入

JSP のコメントには、次の 2 種類を使用できます。クライアントに返されない JSP 自体に対するコメントと、ページの出力の一部としてクライアントに返されるコメントです。

このセクションでは、これらのコメントについて説明します。

JSP に関するコメントの記述

JSP コメントは、JSP 自体に情報を追加するために使用されます。このコメントが JSP の出力の一部としてクライアントに出力されることはありません。

JSP コメントの構文は次のとおりです。

```
<%-- コメント文字列... --%>
```

また、ページで使用されているスクリプト言語のコメント構文を使用して、コメントを追加することもできます。たとえば、ページのスクリプト言語として Java を使用している場合、次の形式でコメントを追加できます。

```
<% /** Java コメント **/ %>
```

JavaScript で記述されたスクリプトにコメントを入れる場合も、同様の構文を使用します。

クライアントへのコメント出力

クライアントに返される JSP の応答出力に表示されるコメントを生成するには、HTML コメント構文を使用します。構文は次のとおりです。

```
<!-- コメント -->
```

コメント内部に JSP 式を入れることにより、動的なコメントを作成できます。次の例では、実行時に評価される式の入ったコメント文字列がクライアントに出力されます。

```
<%! String PageName = "Example Comment Page"; %>
...
<!-- ページに対するコメント:<%= PageName %> -->
```

JSP における相対 URL の指定

JSP 要素では、相対 URL 指定を使用して、別の JSP、Java サーブレット、同じページにある別のエンティティなどを参照できます。参照元 JSP での URL の指定方法に応じて、URL を参照元 JSP を含むアプリケーション、または参照元 JSP の場所に相対させることができます。

例

- `myErrorPage.jsp`

  参照元 JSP の位置に相対する `myErrorPage.jsp` が参照されます。この場合、参照元ページと同じディレクトリにある `myErrorPage.jsp` ページが検索されます。

- `../myErrorpage.jsp`

  参照元 JSP の位置に相対する `myErrorPage.jsp` が参照されます。この場合、参照元ページの親ディレクトリにある `myErrorPage.jsp` ページが検索されます。

- `/errorPages/myErrorpage.jsp`

  参照の先頭に「/」を付けると、参照元 JSP を含むアプリケーションに相対する `myErrorpage.jsp` が検索されます。JSP はすべて、1 つのアプリケーションに含まれます。アプリケーションを設定する手順の一部として、「/」にマッピングされるディレクトリを定義する必要があります。

ディレクティブ

ディレクティブにより、JSP と結果として求められるサーブレットのプロパティを設定できます。ディレクティブによって定義される情報の例には、JSP のスクリプト言語、出力 MIME タイプ、ページによって使用されるタグ ライブラリ、ページで必要とされるインクルード ファイルなどがあります。

通常は、ディレクティブを使用して JSP のスクリプト言語を設定します。既定の設定では、Java が使用されます。しかし、次のディレクティブを使用して、スクリプト言語を JavaScript に設定できます。

```
<%@  page language = "javascript" %>
```

ディレクティブは、JSP の構文解析時に評価されるプリプロセス要素です。ディレクティブは構文解析時に評価されるのに対し、式は実行時に評価されるので、ディレクティブに式は使用できません。したがって、次の例では式 `<%=myVar%>` をコンパイル時に評価できないので、この例にあるディレクティブの使用は正しくありません。

```
<%@ page import="<%=myVar%>" %>
```

また、ディレクティブは構文解釈時に評価されるので、ディレクティブからの出力はありません。つまり、ディレクティブを使用して、クライアントに情報を返すことはできません。

JSP ディレクティブすべての基本構文は次の形式になります。

```
<%@ directive %>
```

JSP では、次の表にある 3 種類のディレクティブがサポートされています。次のセクションでは、これらのディレクティブについて説明します。

ディレクティブ	目的	構文	ページ
page	ページ全体にわたる属性を定義します。	`<%@ page attribute="value" ... %>`	116 ページ
include	JSP にテキストを挿入します。	`<%@ include file = "path" ... %>`	120 ページ
taglib	JSP にタグ ライブラリを含めます。	`<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>`	120 ページ

page ディレクティブ

page ディレクティブにより、JSP 全体に対する 1 つ以上の属性を定義できます。

1 つの JSP で複数の page ディレクティブを使用できます。ただし、import 属性を使用している場合を除き、1 つの属性は 1 回しか参照できません。したがって、重複する属性を持つ複数の page ディレクティブを指定した場合、JSP にある page ディレクティブの最初のインスタンスは認識されますが、この後、この属性を再定義しようとしても無視されます。

import 属性を含む page ディレクティブは複数指定できます。import 属性によって参照されるファイルがすべてインポートされます。

page ディレクティブの構文は次のとおりです。

```
<%@ page attribute = "value" ... %>
```

ここで、

```
attribute は、language | import | contentType | session | buffer |
          autoflush | isThreadSafe | info | errorPage | isErrorpage |
          extends
```

value は、一重引用符、または二重引用符で囲まれた文字列定数

これ以外の属性を含む page ディレクティブでは、例外が発生します。

たとえば、次のディレクティブでは JSP の出力 MIME タイプが HTML に、スクリプト言語が JavaScript に設定されます。

```
<%@  page contentType = "text/html" language = "javascript" %>
```

language

ファイルで使用されるスクリプト言語を定義します。この属性を省略した場合、既定のスクリプト言語 java が使用されます。これは Java プログラミング言語を表します。

使用可能な値は java と javascript です。

import

コンパイルされたページでインポートして使用する必要のあるパッケージを、カンマ区切りリスト で指定します。次に例を示します。

```
<%@  page import = "java.io.*,java.util.Hashtable" %>
```

JSP の言語が Java である場合、import で指定されたファイルのほかに、JRun には次のファイルが常にインポートされます。

- java.lang.*
- javax.servlet.*
- javax.servlet.jsp.*
- javax.servlet.http.*

Java 以外の言語については、既定のインポート リストはありません。Java 以外のスクリプト言語を使用している場合、または Java を使用しているが、ほかのファイルをインポートする必要がある場合は、必要なファイルをインポートしてください。

contentType

MIME タイプを定義します。必要に応じて、JSP により生成される応答の文字セット を定義します。既定の設定では、MIME タイプは text/html で、文字セットは ISO-8859-1 です。

この属性の構文は次のとおりです。

```
<%@  page contentType = "TYPE; charset = CHARSET" %>
```

TYPE は出力 MIME タイプを表します。また、オプションの *CHARSET* には、文字セットを Internet Assigned Numbers Authority (IANA) 値で指定します。

使用可能な MIME タイプの一覧については、次の URL を参照してください。

ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types

CHARSET でサポートされている文字エンコードの一覧については、次の URL を参照してください。

http://java.sun.com/products/jdk/1.1/docs/guide/intl/encoding.doc.html

次の contentType の例では、出力タイプとしてテキスト形式が設定されています。

```
<%@  page contentType = "text/plain" %>
```

session

このページがセッションの一部としてアクセスされるかどうかを指定します。

値に true を指定した場合、JSP オブジェクト session が、このページの現在のセッションとして初期化されます。

値 false は、このページがセッションの一部ではなく、session オブジェクトは使用できないことを表します。JSP の本文で session を参照すると、致命的な変換エラーが発生します。

既定値は true です。

buffer

ページ出力で使用されるバッファリング モデルを表します。サイズ指定に文字列 kb を入れる必要があります。たとえば、ディレクティブでバッファ サイズを 16 KB に設定します。

```
<%@  page buffer = "16kb" %>
```

値 none は、バッファリングは行われず、ページ出力はすべて、直接クライアントに書き込まれることを表します。

バッファ サイズを指定した場合、出力は指定されたサイズ以下のサイズでバッファリングされます。autoFlush 属性の値に応じて、バッファの内容は自動的にフラッシュされるか、または、オーバーフローが発生したときに例外が発生します。

この属性の既定値は buffered で、バッファのサイズは 8 KB です。

autoFlush

値に true を設定した場合、バッファが満杯になると、バッファリングされた出力が自動的にフラッシュされます。false を設定した場合、バッファのオーバーフローを示す例外が発生します。

既定値は true です。buffer が none に設定されている場合は、autoFlush を false に設定できません。

isThreadSafe

ページごとに実装されるスレッド セーフのレベルを指定します。

この属性を false に設定すると、複数の要求を処理するためにページのインスタンスが複数作成されますが、これらのページ インスタンスはそれぞれ、1 つずつスレッドを持つようになります。

true に設定すると、複数の未処理クライアント要求がこのページに対して、同時に送信される可能性があります。JSP によって、ページの共有ステートへのアクセスが適切に同期化されることを保証するには true を設定します。

既定値は true です。

info

JSP に組み込まれる文字列を指定します。この文字列は実行時に取得可能です。アプリケーションからこの文字列にアクセスするには、このサーブレットを表す `Servlet` オブジェクトの `getServletInfo()` メソッドを使用します。

isErrorPage

JSP で、別の JSP からの未処理例外を処理するかどうかを指定します。

値 `true` を設定すると、JSP `exception` オブジェクトが定義されます。JSP が起動されると、このオブジェクトの値がソース JSP からの例外に設定されます。

値 `false` は `exception` オブジェクト が使用できないことを表します。したがって、JSP の本体からこのオブジェクト を参照すると 、致命的な変換エラーが起こります。

既定値は `false` です。

errorPage

そのページで検出されなかった例外がエラー処理のために送信される JSP への URL を指定します。送信先の JSP は、JSP バージョン 1.1 仕様に準拠していなければなりません。

送信先ページでは `page` ディレクティブを使用して、`isErrorPage` 属性を `true` に設定します。

呼び出されると 、送信先 JSP の `exception` オブジェクト に例外に対する 参照が入ります。

メモ

`autoFlush` を `true` に設定したときに、例外を返したページからの出力データがすべてフラッシュされた場合、エラー ページへ例外を返そうとしても、失敗する可能性があります。

---

extends

JSP がサブクラス化する JSP 基本クラスを指定します。次に例を示します。

```
<%@  page extends = "com.myPackage.AServletImplementation" %>
```

既定の設定では、サーブレットはクラス `javax.servlet.http.HttpServlet` のサブクラスとして作成されます。したがって、サーブレットの基本クラスを定義しなければならない特別な理由がない限り、この属性は使用しないでください。

include ディレクティブ

include ディレクティブにより、変換時に JSP ファイルにテキストが挿入されるので、効果的に include ディレクティブを置き換えることができます。このディレクティブの結果、組み込み先 JSP のコンテンツと、組み込まれる JSP のコンテンツの両方が 1 つの JSP に入るようになります。

このディレクティブの構文は次のとおりです。

```
<%@ include file = "path" %>
```

/ で始まる *path* は JSP のアプリケーションに関連付けられています。パス名の先頭に / が付いていない場合、このパスは変換中の JSP のパスに関連付けられていると見なされます。パスの詳細については、115 ページの「JSP における相対 URL の指定」を参照してください。

複数の JSP で共有する機能がある場合は、include ディレクティブが便利です。include ディレクティブの例については、101 ページの「include の実行」を参照してください。

taglib ディレクティブ

taglib ディレクティブを使用して、JSP が含めるタグ ライブラリを宣言できます。タグ ライブラリには JSP で使用可能なカスタム タグが含まれます。

次の例では、タグ coolTag を含むタグ ライブラリが定義されています。このディレクティブの後でこのタグ ライブラリにあるタグを参照するには、接頭辞 myTags を使用します。接頭辞 myTags を使用する JSP に別のタグ ライブラリを定義すると、エラーが発生します。

```
<%@ taglib uri="myApp/appTags" prefix="myTags" />

<myTags:coolTag>
...
</myTags:coolTag>
```

タグ ライブラリに coolTag がないと、致命的な変換エラーが発生します。

taglib ディレクティブの構文は次のとおりです。

```
<%@ taglib uri="path" prefix="tagPrefix" %>
```

uri

タグ ライブラリの場所を、相対パス位置、またはアプリケーションと関連付けられている web.xml ファイルへの検索キーとして指定します。JRun は、まず web.xml ファイルをチェックし、*path* が検索キーであるかどうかを判断します。web.xml ファイルに *path* が見つからない場合、JRun はこれ自体をパス位置と見なします。

タグ ライブラリが見つからないと、致命的な変換エラーが発生します。

path が web.xml ファイルへの検索キーである場合、JRun により web.xml ファイルからこのキーと関連するタグ ライブラリが検索されます。たとえば、アプリケーションの web.xml ファイルで myTagLib という検索キーと、タグ ライブラリ の関連位置を定義するには次のように記述します。

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

path がタグ ライブラリを表す相対パスで、/ で始まっている場合、このパスは JSP のアプリケーションに関連付けられています。パス名の先頭に / が付いていない場合、このパスは変換中の JSP のパスに関連付けられていると見なされます。パスの詳細については、115 ページの「JSP における相対 URL の指定」を参照してください。

prefix

ライブラリにあるカスタム タグを表す接頭辞文字列を定義します。接頭辞 jsp、jspx、java、javax、servlet、sun、および sunw は予約されています。空の接頭辞は使用できません。

スクリプト要素

スクリプト要素により、JSP に含まれる実際のコードが定義されます。このコードは Java または JavaScript (ECMAscript) で記述します。

スクリプト要素で使用されるコードの正確な構文は、JSP ディレクティブを使用して JSP のために指定されたスクリプト言語によって異なります。スクリプト言語の設定の詳細については、116 ページの「page ディレクティブ」を参照してください。

次の表は、JRun でサポートされている 3 種類のスクリプト要素を示します。

要素	目的	構文
宣言	変数など、ページ全体で使用される定義を作成します。	<%! *declaration* %>
スクリプトレット	このページで使用されるスクリプト コードが含まれます。	<% *script code* %>
式	ページ出力をクライアントに送信する前にサーバーで評価されるステートメントを定義します。	<%= *expression* %>

次のセクションで、これらの要素について説明します。

宣言

宣言により、1 つの JSP 全体にわたる定義を行うことができます。JSP で使用される変数やメソッドを定義するには通常、宣言を行います。宣言により、クライアントに出力が書き込まれることはありません。

宣言の構文は次のとおりです。

```
<%! declaration(s) %>
```

次の宣言では Java 変数と関数が定義されます。

```
<%!
  private String foo = null;
  public String getFoo() { return this.foo; }
%>
```

スクリプトレット

スクリプトレット要素では、JSP で使用されるスクリプト コードを指定します。有効なコードをスクリプトレット要素の本文内で指定できます。

スクリプトレットを使用して、JSP の出力ストリームにデータを書き込むことができます。その後、この情報は HTTP 応答とともにクライアントに返されます。通常、このデータは HTML テキスト形式で記述されています。

スクリプトレット内のコードから、application、session など、JSP 用に定義された暗黙的オブジェクトにアクセスできます。これらのオブジェクトの詳細については、第 9 章を参照してください。

page ディレクティブの language 属性を使用して、スクリプト言語を Java、または JavaScript (ECMAScript) に指定できます。既定では、言語は Java に設定されています。詳細については、116 ページの「page ディレクティブ」を参照してください。

スクリプトレット要素のために、一般に使われている構文は次のとおりです。

```
<% script code %>
```

組み込まれた Java コードの例は次のとおりです。

```
<%
  String greeting = request.getParameter("Greeting");
  out.println(greeting);
%>
```

式

式は、ページ出力をクライアントに送信する前にサーバーで評価されるステートメントです。式の結果のデータ タイプは String です。

式から、application、session など、JSP 用に定義された暗黙的オブジェクトにアクセスできます。これらのオブジェクトの詳細については、第 9 章を参照してください。

式の構文は次のとおりです。

```
<%= expression %>
```

次の例にある 2 つの式により、2 つの変数の値が文字列として出力されます。

```
<table>
<tr>
  <td><%= myVar1%></td>
  <td><%= myVar2%></td>
</tr>
</table>
```

アクション

アクションによって、オブジェクトの使用、修正、または作成を行ったり、ページの出力ストリームを修正できます。このセクションでは、JRun でサポートされているすべてのアクションについて説明します。

次の表は、全アクションの一覧です。

要素名	目的	ページ
「jsp:useBean」	JavaBeans のインスタンスを定義します。	124 ページ
「jsp:setProperty」	Bean にある 1 つ以上のプロパティ値を設定します。	126 ページ
「jsp:getProperty」	Bean プロパティの値を、文字列として out オブジェクトに自動的に書き込みます。	128 ページ
「jsp:include」	ある JSP から別の JSP を呼び出します。アクションが完了すると、呼び出し先のページから呼び出し側のページに制御が戻されます。	128 ページ
「jsp:forward」	ある JSP から別の JSP を呼び出します。この呼び出しにより、呼び出し側ページの実行が終了します。	129 ページ
「jsp:param」	ある JSP から別の JSP に要求を転送するときに、名前/値ペアとしてパラメータを HTTP 要求に追加します。	130 ページ
「jsp:plugin」	クライアント ブラウザでアプレットを起動できるようになります。	130 ページ

jsp:useBean

jsp:useBean アクションにより、JSP で JavaBeans がインスタンス化されます。インスタンス化が完了すると、JSP ファイルにある Bean が参照できるようになります。

次の例では、タイプ com.myco.myapp.MyBean の myBean という Bean が定義されます。

```
<jsp:useBean id="myBean" class="com.myco.myapp.MyBean" />
```

jsp:useBean の基本構文は次のとおりです。

```
<jsp:useBean id="name" scope="page|request|session|application"
        typeSpec />
```

ここで、typespec には次のいずれかが入ります。

```
class="className" |
class="className" type="typeName" |
beanName="beanName" type="typeName" |
type="typeName"
```

type または class を指定する必要があります。このとき、class と beanName の両方を指定することはできません。type と class の両方を指定する場合は、class に type が割り当てられるようになっていなければなりません。

属性 beanName は「a.b.c」の形式で表された Bean の名前です。ここに、「a/b/c.ser」の形式でクラスまたはリソース名を指定します。

さらに、次の形式で jsp:useBean に対して本文を指定できます。

```
<jsp:useBean id="name" scope="page|request|session|application"
 typeSpec >
 body
</jsp:useBean>
```

Bean が作成されると、本文が呼び出されます。通常、本文には新しく作成された Bean を修正するために使用されるスクリプトレット、または jsp:setProperty タグが含まれますが、本文の内容は制限されません。

<jsp:useBean> タグには次のような属性があります。

id

指定されたスコープで Bean を識別するために使用される名前と、Bean のスクリプト変数名を指定します。名前では大文字と小文字を区別し、スクリプト言語の変数命名規則に従って指定する必要があります。

scope

必要に応じて、Bean が使用可能なスコープを定義します。

- page この Bean は現在のページで使用できます。これが既定値です。
- request この Bean は getAttribute メソッドを使用して、現在のページにある request オブジェクトから使用できます。現在のクライアント要求が完了すると、この参照は破棄されます。
- session この Bean は getValue メソッドを使用して、現在のページにある session オブジェクトから使用できます。現在のセッションが無効になると、この参照は破棄されます。

注意

JSP で page ディレクティブにより、このページがセッションに含まれていないと指定されている場合、session スコープを指定すると、致命的な変換エラーが発生します。

- application この Bean は getAttribute メソッドを使用して、現在のページにある application オブジェクトから使用できます。

class

Bean の実装を定義する認識可能なクラス名を設定します。クラス名では大文字と小文字は区別されます。

class 属性と beanName 属性を省略する場合、オブジェクトは指定されたスコープ内にある必要があります。

beanName

java.beans.Beans クラスの instantiate メソッドで認識されるように Bean 名を指定します。

この属性の値として、要求時属性式を使用できます。

type

指定されている場合、type はスクリプト変数のタイプを表します。Bean は指定されたタイプへのインスタンスである必要があります。

この属性を使用して、スクリプト変数のタイプを、指定された実装クラスのスクリプト変数に関連させながら、同時に区別できます。このタイプはクラス自体、クラスのスーパークラス、または指定されたクラスにより実装されたインターフェイスになります。

このパラメータを省略した場合、タイプは class 属性の値と同じになります。

jsp:setProperty

jsp:setProperty アクションにより、Bean にある 1 つ以上のプロパティ値が設定されます。このアクションを使用する前に、jsp:useBean を使用して Bean を定義しておく必要があります。

次の例では、*user* という Bean のプロパティに値が設定されます。

```
<jsp:setProperty name="user" property="user" param="username" />
```

次の例では、式を使用してプロパティが設定されます。

```
<jsp:setProperty name="results" property="row" value="<%= i+1 %>" />
```

単純なインデックス付きプロパティは setProperty を使用して設定します。インデックス付きプロパティの値には、配列を割り当てる必要があります。

jsp:setProperty の構文は次のとおりです。

```
<jsp:setProperty name="beanName" prop_expr />
```

ここで、prop_expr は次のいずれかの形式になります。

```
property="*" |
property="propertyName"|
property="propertyName" param="parameterName"|
property="propertyName" value="propertyValue"
```

propertyValue には文字列定数、または式を指定する必要があります。

jsp:setProperty 要素には次のような属性があります。

name

jsp:useBean アクション、またはその他の要素により定義された Bean 名。Bean インスタンスには、設定する必要のあるプロパティが含まれます。jsp:useBean 要素は、同じファイル内にある jsp:setProperty アクションより先に記述する必要があります。

property

設定する必要のある Bean プロパティの名前。

propertyName を * に設定すると、現在の要求パラメータに対して、`jsp:setProperty` が繰り返し実行され、パラメータ名および値タイプが Bean のプロパティ名およびタイプと比較されます。一致したプロパティはそれぞれ、対応するパラメータの値に設定されます。パラメータの値が空白文字列 ("") である場合、対応するプロパティは変更されません。

prop_expr の先頭から 3 つの書式では、文字列として表される値が Bean プロパティに割り当てられます。しかし、Bean プロパティのデータ タイプが `string` 以外である場合、JRun によりタイプ変換が行われます。次の表は、この変換がどのように行われるかを示します。

プロパティ タイプ	変換
boolean または Boolean	`java.lang.Boolean.valueOf(String)` で定義されたとおり
byte または Byte	`java.lang.Byte.valueOf(String)` で定義されたとおり
char または Character	`java.lang.Character.valueOf(String)` で定義されたとおり
double または Double	`java.lang.Double.valueOf(String)` で定義されたとおり
int または Integer	`java.lang.Integer.valueOf(String)` で定義されたとおり
float または Float	`java.lang.Float.valueOf(String)` で定義されたとおり
long または Long	`java.lang.Long.valueOf(String)` で定義されたとおり

prop_expr の 4 つめの書式では、オブジェクトが Bean プロパティに割り当てられます。この場合、オブジェクトは自動的に割り当て先 Bean プロパティのデータタイプに変換されます。

param

Bean プロパティに供給する必要のある値を持つ要求パラメータ名。

アクションは `param` と `value` の両方を持つことはできません。

`param` を省略した場合、要求パラメータ名は Bean プロパティ名と同じであると見なされます。

`param` が `request` オブジェクトに設定されていない場合、または値が空の文字列 ("") である場合、`jsp:setProperty` 要素は何の効力も持ちません。

value

プロパティに割り当てられる値。

アクションは `param` と `value` の両方の値を持つことはできません。

jsp:getProperty

Bean プロパティの値を文字列として、`out` オブジェクトに書き込みます。このアクションを使用する前に、Bean を定義しておく必要があります。

次の例では、*user* Bean の *name* プロパティが書き込まれます。

```
<jsp:getProperty name="user" property="name" />
```

このアクションの構文は次のとおりです。

```
<jsp:getProperty name="name" property="propertyName" />
```

name

このプロパティで使用される Bean インスタンス名を指定します。

このアクションは、Bean が見つからなかった場合に例外を生成します。

property

出力する必要のある値を持つプロパティ名。

jsp:include

現在のページに静的なリソースおよび動的なリソースを含めます。組み込みが完了すると、呼び出し側 JSP で処理が再開されます。

既定では、JSP からクライアントへ送信される出力データはバッファリングされます。バッファを使用しているため、応答ヘッダ情報およびその他の出力はバッファがフラッシュされるまではクライアントに送信されません。組み込まれるページの出力がバッファリングされる場合、組み込みの前にバッファがフラッシュされます。このフラッシュにより、組み込まれたページで応答ヘッダを設定できなくなります。したがって、組み込まれたページで `setCookie` などのメソッドは使用できません。

メモ

バッファを無効にするか、または JSP の出力バッファのサイズを設定するには、JSP ディレクティブを使用します。`page` ディレクティブの詳細については、116 ページの「page ディレクティブ」を参照してください。

次の例では、HTML ページが含められます。

```
<jsp:include page="/templates/copyright.html"/>
```

`jsp:include` の構文は次のとおりです。

```
<jsp:include page="path" flush="true"/>
```

または

```
<jsp:include page="path" flush="true">
  <jsp:param name="paramName" value="paramValue" /> ...
</jsp:include>
```

page

含められるファイルのパスを指定します。/ で始まるパスは JSP のアプリケーションに関連付けられています。パス名の先頭に / が付いていない場合、このパスは変換中の JSP のパスに関連付けられていると見なされます。

パスの詳細については、115 ページの「JSP における相対 URL の指定」を参照してください。

flush

この属性を `true` に設定すると、バッファがフラッシュされます。JSP 1.1 では、値に `false` を指定できません。

既定値は `true` です。

`jsp:include` の 2 番目の書式は、`jsp:param` アクションへの追加です。このアクションにより、目的の JSP により受信される HTTP 要求にパラメータを追加できます。詳細については、130 ページの「jsp:param」を参照してください。

jsp:forward

現在のページと同じアプリケーションで `jsp:forward` アクションにより、JSP または Java サーブレットが呼び出されます。`jsp:forward` により、現在の JSP の実行が終了されます。

ページ出力がバッファリングされた場合、`jsp:forward` アクションを呼び出すか、`response` オブジェクトの `redirect` メソッドを使用して要求を転送した場合に、バッファは消去され、内容が破棄されます。このルールの唯一の例外は、JSP を転送することによって設定されたクッキーが破棄されず、クライアントに送信されることです。

ページ出力がバッファリングされず、ページ出力に何かが書き込まれている場合、`jsp:forward` を使用しようとすると、`IllegalStateException` になります。

メモ

バッファを無効にするか、または JSP の出力バッファのサイズを設定するには、JSP ディレクティブを使用します。`page` ディレクティブの詳細については、116 ページの「page ディレクティブ」を参照してください。

次の例では、ある JSP から別の JSP が呼び出されます。

```
<% String whereTo = "/templates/"+someValue; %>
<jsp:forward page='<%= whereTo %>' />
```

`jsp:forward` の構文は次のとおりです。

```
<jsp:forward page="path" />
```

または

```
<jsp:forward page="path">
  <jsp:param name="paramName" value="paramValue" /> ...
</jsp:forward>
```

page

呼び出されるファイルのパスを指定します。/ で始まるパスは JSP に含まれるアプリケーションに関連付けられています。パス名の先頭に / が付いていない場合、このパスは変換中の JSP のパスに関連付けられていると見なされます。

パスの詳細については、115 ページの「JSP における相対 URL の指定」を参照してください。

jsp:forward の 2 番目の書式は、jsp:param アクションへの追加です。このアクションにより、目的の JSP により受信される HTTP 要求にパラメータを追加できます。詳細については、130 ページの「jsp:param」を参照してください。

jsp:param

jsp:param アクションにより、ある JSP から別の JSP に要求が転送されるときに、名前 / 値ペアとしてパラメータが HTTP 要求に追加されます。このアクションとともに使用できるのは、jsp:include、jsp:forward、および jsp:plugin アクションだけです。

jsp:include または jsp:forward とともに jsp:param を使用すると、目的のページにより、オリジナルの要求パラメータを持つオリジナルの HTTP 要求と、jsp:param で指定された新しいパラメータがすべて受信されます。jsp:param により、要求にすでに入っているパラメータが追加された場合、新しいパラメータ値は既存の値の前に入ります。

たとえば、要求にパラメータ myParm=a が含まれているときに、jsp:param を使用して myParm=b を追加すると、転送された要求には myParm=b, a が入ります。ここで、新しいパラメータがリストの先頭に追加されていることに注意してください。

新しいパラメータのスコープは jsp:include または jsp:forward の目的の JSP です。つまり、含められたページからオリジナルの JSP に戻ってきた後、新しいパラメータと値が要求から削除されます。

jsp:plugin

jsp:plugin アクションにより、クライアント ブラウザでアプレットを呼び出せるようになります。このアクションでは、適切なクライアント ブラウザ依存のコンストラクト (OBJECT または EMBED) を含む HTML テキストが生成されます。この構成体により、Java プラグインがダウンロードされ、続いてアプレットや Bean が実行されます。

この要素は要求元ユーザ エージェントに応じて、<object> タグまたは <embed> タグで置き換えられ、応答の出力ストリームに書き込まれます。次の例では、クライアントにある MyPlugin.class が呼び出されます。

```
<jsp:plugin type=applet code="MyPlugin.class" codebase="/html" >
  <jsp:params>
    <jsp:param name="myplugin" value="Greetings"/>
  </jsp:params>
  <jsp:fallback>
    <p> unable to load Plugin </p>
  </jsp:fallback>
</jsp:plugin>
```

`jsp:plugin` の構文は次のとおりです。

```
<jsp:plugin
  type="bean|applet"
  code="objectCode"
  codebase="objectCodebase"
  { align="alignment" }
  { archive="archiveList" }
  { height="height" }
  { hspace="hspace" }
  { jreversion="jreversion" }
  { name="componentName" }
  { vspace="vspace" }
  { width="width" }
  { nspluginurl="url" }
  { iepluginurl="url" } >
  { <jsp:params>
    { <jsp:param name=" paramName" value="paramValue" /> }+
  </jsp:params> }
  { <jsp:fallback> arbitrary_text </jsp:fallback> } >
</jsp:plugin>
```

中かっこ ({}) で囲まれた要素はオプションです。

type

コンポーネントのタイプを Bean またはアプレットとして指定します。

code

アプレットを含むクラス ファイル名、またはクラスへのパスを指定します。

codebase

`archive` 属性で指定された相対パスを解釈するために使用される基本パスを指定します。この値を省略すると、`<jsp:plugin>` を呼び出した JSP ファイルのパスが使用されます。

詳細については、次の URL にある HTML 4.0 仕様書を参照してください。

`http://www.w3.org/TR/REC-html40/`

align

Bean またはアプレットの場所を指定します。align に次のような値を指定した場合、周囲のテキストに相対させてオブジェクトの場所が決められます。

- bottom　オブジェクトの最下部が、現在のベースラインに対し、垂直方向に揃えられます。これが既定値です。
- middle　オブジェクト の中心が、現在のベースラインに対し 、垂直方向に揃えられます。
- top　オブジェクトの最上部が、現在のテキスト ラインの最上部に対して垂直方向に揃えられます。
- left および right　イメージは現在の左マージンまたは右マージンに浮動します。

詳細については、次の URL にある HTML 4.0 仕様書を参照してください。

http://www.w3.org/TR/REC-html40/

archive

オブジェクトのリソースを含むアーカイブを表す URI を、スペースで区切られたリストとして指定します。一般的に、アーカイブをあらかじめロードしておくと、オブジェクトのロード時間を削減できます。相対 URI として指定されたアーカイブは codebase 属性に相対しているものとして解釈されます。

詳細については、次の URL にある HTML 4.0 仕様書を参照してください。

http://www.w3.org/TR/REC-html40/

height

既定の Bean またはアプレットの高さを書き換えて、指定された値を使用します。値は次のように指定できます。

- ピクセルを表す整数値 (*N*)
- 使用可能な垂直スペースのパーセント値 (*N%*)
- 使用可能な垂直スペースの部分 (*N**)

詳細については、次の URL にある HTML 4.0 仕様書を参照してください。

http://www.w3.org/TR/REC-html40/

hspace

Bean またはアプレットの左右に挿入される空白の量を指定します。値は次のように指定できます。

- ピクセルを表す整数値 (*N*)
- 使用可能な水平スペースのパーセント値 (*N%*)
- 使用可能な水平スペースの部分 (*N**)

詳細については、次の URL にある HTML 4.0 仕様書を参照してください。

http://www.w3.org/TR/REC-html40/

jreversion

コンポーネントが動作するために必要な JRE 仕様のバージョン番号を表します。既定値は `1.1` です。

name

アプレットの名前を指定します。これにより、このアプレットを JavaScript から参照したり、同じページにある別のアプレットを参照できるようになります。

vspace

Bean またはアプレットの上下に挿入される空白の量を指定します。値は次のように指定できます。

- ピクセルを表す整数値 (*N*)
- 使用可能な水平スペースのパーセント値 (*N*%)
- 使用可能な水平スペースの部分 (*N**)

詳細については、次の URL にある HTML 4.0 仕様書を参照してください。

http://www.w3.org/TR/REC-html40/

title

アプレットに関する記述情報を指定します。

詳細については、次の URL にある HTML 4.0 仕様書を参照してください。

http://www.w3.org/TR/REC-html40/

width

Bean やアプレットについて、既定のオブジェクト幅を書き換え、指定された値を使用します。値は次のように指定できます。

- ピクセルを表す整数値 (*N*)
- 使用可能な水平スペースのパーセント値 (*N*%)
- 使用可能な水平スペースの部分 (*N**)

詳細については、次の URL にある HTML 4.0 仕様書を参照してください。

http://www.w3.org/TR/REC-html40/

nspluginurl

Netscape Navigator 用 JRE プラグインをダウンロードできる URL を指定します。既定値は実装によって異なります。

iepluginurl

Internet Explorer 用 JRE プラグインをダウンロードできる URL を指定します。既定値は実装によって異なります。

jsp:param

アプレット コンポーネントまたは JavaBeans コンポーネントにパラメータを設定します。詳細については、130 ページの「jsp:param」を参照してください。

jsp:fallback

プラグインが起動できなかったときに、クライアント ブラウザにより表示されるコンテンツを指定します。プラグインは起動できても、アプレット コンポーネントまたは JavaBeans コンポーネントが見つからなかったり、起動できなかった場合は、プラグイン固有のメッセージがユーザに送られます。

第 9 章

JSP オブジェクト リファレンス

JSP で使用するすべてのスクリプト言語は、Java および Java サーブレット API に組み込まれた特定のオブジェクトへのアクセスを提供する必要があります。オブジェクトへのアクセスとは、オブジェクト メソッドを呼び出し、これらのメソッドに格納されたデータにアクセスすることを意味します。この章では、これらのオブジェクトについて説明し、JSPの作成時にそれらのオブジェクトにアクセスする方法について説明します。

目次

- JSP オブジェクト 136
- application オブジェクト 138
- config オブジェクト 139
- exception オブジェクト 139
- out オブジェクト 140
- pageContext オブジェクト 141
- request オブジェクト 142
- response オブジェクト 143
- session オブジェクト 144

JSP オブジェクト

このセクションでは、JSP 内からアクセス可能なオブジェクトについて説明します。これらのオブジェクトを使用して JSP 内から基本的なタスクを実行できます。次のようなオブジェクトがあります。

- request オブジェクト　クライアントから JSP に送信された HTTP 要求。HTTP 要求には request ヘッダ内のすべての名前 / 値ペアが含まれます。
- response オブジェクト　JSP により生成される HTTP 応答オブジェクト。response オブジェクトを使用して、応答ヘッダとクッキーの名前 / 値ペアなどの情報をクライアントに返すことができます。
- out オブジェクト　クライアントに返される出力ストリーム。JSP によって出力される HTML テキストは通常、out オブジェクトに書き込まれ、クライアントによって解釈されます。

メモ
この章では、これらのオブジェクトの概要についてのみ説明します。オブジェクトとそのメソッドの詳細については、HTML バージョンの Java サーブレット API のマニュアルを参照してください。これらの HTML ファイルは、使用中の JRun インストールディレクトリにある docs/api ディレクトリに配置されています。各オブジェクトは JSP オブジェクト名ではなく、次の表にある Java サーブレット API オブジェクトのタイプにより識別されます。

JSP オブジェクトは実際には、Java サーブレット API からのオブジェクトとして実装されます。JSP オブジェクトにより、これらの Java サーブレット API オブジェクトにアクセスするための簡易メカニズムが使用できます。

次の表は、各 JSP オブジェクト の一覧で、これらのオブジェクト に対応するサーブレット API オブジェクトのタイプをまとめたものです。

JSP オブジェクト	Java サーブレット API オブジェクトのタイプ	オブジェクトの意味
application	javax.servlet.ServletContext	サーブレット設定オブジェクトから取得したサーブレット コンテキスト
config	javax.servlet.ServletConfig	JSP の ServletConfig
exception	java.lang.Throwable	エラー ページが表示される原因となる未処理の例外
out	javax.servlet.jsp.JspWriter	JSP の出力ストリームに書き込みを行うオブジェクト
pageContext	javax.servlet.jsp.PageContext	JSP に対するページ コンテキスト
request	javax.servlet.HttpServletRequest	クライアント要求
response	javax.servlet.HttpServletResponse	クライアントへの応答
session	javax.servlet.http.HttpSession	要求元クライアントのために作成されたセッション オブジェクト

JSP オブジェクトへのアクセスの取得

JRun により JSP が呼び出されると、前のセクションの表に示されている JSP オブジェクトがすべて作成されます。オブジェクト名とオブジェクト メソッド、またはフィールドを参照するだけで、JSP でこれらのオブジェクトにアクセスできます。

メモ
セッションのトラッキングを有効にした場合にのみ、`session` オブジェクトが作成されます。`exception` オブジェクトが作成されるのは、JSP で `page` ディレクティブを使用して、`isErrorPage` を `true` に設定した場合だけです。詳細については、115 ページの「ディレクティブ」を参照してください。

JSP オブジェクトの使用

Java オブジェクトへのインターフェイスは、オブジェクトの public メソッドとフィールドから構成されています。したがって、この章で説明するオブジェクトを使用するには、オブジェクトのメソッドやフィールドを参照してください。

オブジェクト のメソッド 、またはフィ ールド を参照するには、「 ドット 」 アド レス メ カニズムを使用し ます。つまり、オブジェクト名に続けてドット (ピリオド) を入力してから、メソッド名、またはフィールド名を指定し ます。次の例ではこの構文を使用して、JSP の出力に文字列を書き込んでいます。

```
out.println("Greetings from my page!");
```

この例の場合、オブジェクト名は `out` で、メソッドは `println` です。

次の例では、`request` オブジェクトを使用して、JSP への HTTP 要求に含まれている `fName` パラメータと `lName` パラメータの値が取得されます。次に、`out` オブジェクトを使用して、これらの値がクライアントに書き込まれます。

```
<%
  String firstName = request.getParameter("fName");
  String lastName = request.getParameter("lName");
  out.println("Welcome " + firstName + " " + lastName);
%>
```

次の例では、要求からユーザ名が取得され、この名前が JSP の `session` オブジェクトに書き込まれます。これにより、同じセッションにある別の JSP から情報にアクセスできるようになります。この例は、1 つの JSP によりサービスされているすべてのページでクライアント名を表示するようアプリケーションをカスタマイズする方法を示しています。

最初の JSP の `greeting.jsp` では、ユーザ名が `session` オブジェクトに書き込まれます。

```
<%
  String firstName = request.getParameter("fName");
  String lastName = request.getParameter("lName");
  session.setAttribute("fName", firstName);
  session.setAttribute("lName", lastName);
  out.println("Welcome " + firstName + " " + lastName);
%>
```

メモ
クライアントから渡されたパラメータにアクセスするために使用される request メソッドは、getParameter と setParameter により呼び出されます。しかし、session オブジェクトで使用されるメソッドは getAttribute と setAttribute により呼び出されます。パラメータは、文字列としてクライアントへの受け渡しが行われます。属性は、サーバー側スクリプト間でオブジェクトとして渡されるオブジェクトで、サーブレットなどがあります。

session オブジェクトへの情報の書き込みが完了すると、同じセッションにある別の JSP からこの情報へアクセスできるようになります。たとえば、注文の合計を生成する JSP には、次のような行が含まれます。

```
<%
  String firstName = (String) session.getAttribute("fName");
  String lastName = (String) session.getAttribute("lName");
  out.println("Welcome " + firstName + " " + lastName);
%>
```

getAttribute は常に java.lang.Object タイプのオブジェクトを返すので、getAttribute の戻り値をタイプ変換する必要があります。タイプ変換によって、getAttribute から返されたオブジェクトは、送信先の形式に変換されます。この場合は文字列に変換されます。

application オブジェクト

application オブジェクトを使用すると、特定のアプリケーションの全ユーザ間で情報を共有できます。JSP ベースのアプリケーションは、仮想ディレクトリとそのサブディレクトリにあるすべての .jsp ファイルとして定義されます。

構文

application.*Method*(*variables...*)

次の表は、application オブジェクトに共通するメソッドの一部をまとめたものです。

メソッド	説明
getAttribute(String name)	指定された名前を持つ属性を返します。この名前を持つ属性が存在しない場合は、ヌルを返します。
getAttributeNames()	アプリケーション オブジェクト内にある属性名をすべて返します。
getInitParameter(String name)	初期化パラメータの値を返します。パラメータが存在しない場合は、ヌルを返します。
getInitParameterNames()	初期化パラメータの名前を返します。
getServerInfo()	JRun サーブレット エンジンの名前とバージョン番号を返します。

config オブジェクト

config オブジェクト は、JRun JSP コンテナによって生成され、サーブレット が初期化されたときに設定情報をサーブレットに渡します。このサーブレットがアクセスできる設定情報は、初期化パラメータを表す名前 / 値ペアのセットと、このサーブレットが実行されているコンテキストを表す ServletContext オブジェクトです。

構文

config.*Method(variables...)*

次の表は、config オブジェクトに共通するメソッドの一部をまとめたものです。

メソッド	説明
getInitParameter(String name)	初期化パラメータの値を返します。パラメータが存在しない場合は、ヌルを返します。
getInitParameterNames()	各サーブレットの初期化パラメータ名を返します。
getServletName()	サーブレット名を返します。

exception オブジェクト

exception オブジェクトはすべてのエラーと例外を表します。

構文

exception.*Method*(*variables...*)

次の表は、exception オブジェクトに共通するメソッドの一部をまとめたものです。

メソッド	説明
getMessage()	エラー メッセージが含まれている文字列を返します。
printStackTrace()	標準エラーに exception オブジェクトと、そのバックトレースを書き込みます。
toString()	exception オブジェクトについて説明した文字列を返します。

out オブジェクト

out オブジェクト は、JSP の出力ストリ ームに書き 込みを行う オブジェクト を 定義し ます。

構文

out.*Method*(*variables...*)

次の表は、out オブジェクトに共通するメソッドの一部をまとめたものです。

メソッド	説明
clear()	クライアント にコンテンツを 書き 込まずに、出力バッファをクリアします。バッファ がすでにフラッシュされている 場合は、例外が返され ます。
clearBuffer()	クライアント にコンテンツを 書き 込まずに、出力バッファをクリアします。
flush()	クライアント にコンテンツを 書き 込んで、出力バッファをフラッシュし ます。
getBufferSize()	バッファ のサイズをバイト 単位で 返し ます。出力がバッファリングされていない 場合は、0 を 返し ます。
getRemaining()	バッファ 内の空き 容量をバイト 単位で 返し ます。
isAutoFlush()	出力バッファ が自動的にフラッシュされる 場合は「true」を 返し ます。
newLine()	出力に改行文字を 書き 込みます。
print()	改行文字を 付けずに、出力に値を 書き 込みます。
println()	改行文字を 付けて、出力に値を 書き 込みます。

pageContext オブジェクト

pageContext オブジェクトにより、JSP にローカルな情報を格納するメカニズムが提供されます。JSP はそれぞれ専用の pageContext オブジェクトを持っています。このオブジェクトはページが入力されたときに作成され、ページが終了すると破棄されます。pageContext オブジェクトのメソッドを使用すると、JSP の情報にアクセスしたり、ほかのアクションを実行できます。

構文

pageContext.*Method(variables...)*

次の表は、pageContext オブジェクトに共通するメソッドの一部をまとめたものです。

メソッド	説明
findAttribute(String name)	page、request、session (有効である場合)、および application スコープ内の属性値を返します。属性が見つからない場合は、ヌルを返します。
getAttribute(String name)	page スコープ内で、指定された名前と関連付けられているオブジェクトを返します。オブジェクトが見つからない場合は、ヌルを返します。
removeAttribute(String name)	指定された名前を持つ属性を削除します。
setAttribute(String name, java.lang.Object attribute)	指定した名前を持つオブジェクトを pageContext オブジェクトに書き込みます。

request オブジェクト

request オブジェクトにより、HTTP 要求中にクライアントから Web サーバーに渡される値が取得されます。

構文

request.*Method*(*variables...*)

次の表は、request オブジェクトに共通するメソッドの一部をまとめたものです。

メソッド	説明
getCookies()	要求と一緒にクライアントから送られたクッキーをすべて返します。
getHeader(String name)	要求ヘッダの値を文字列として返します。
getAttribute(String name)	指定された属性値を返します。この属性が存在しない場合は、ヌルを返します。
getAttributeNames()	要求に含まれる属性名をすべて返します。
getHeaderNames()	要求に含まれるヘッダ名をすべて返します。
getHeaders(String name)	指定された要求ヘッダの値をすべて返します。
getMethod()	要求の作成に使用された HTTP メソッドに対応する GET、POST、または PUT を返します。
getParameter(String name)	要求に含まれるパラメータの値を返します。パラメータが存在しない場合は、ヌルを返します。
getParameterNames()	要求に含まれるパラメータ名をすべて返します。
getParameterValues(String name)	指定されたパラメータの値をすべて返します。
getQueryString()	要求からクエリ文字列を返します。
getRequestURI()	要求 URL のうち、プロトコル名からクエリ文字列までの部分を返します。
getServletPath()	要求 URL のうち、サーブレットを呼び出す部分を返します。
setAttribute(String name, java.lang.Object o)	属性と、関連する値を要求に書き込みます。

response オブジェクト

`response` はクライアントにデータを送信するオブジェクトです。

構文

response.*Method*(*variables...*)

次の表は、`response` オブジェクトに共通するメソッドの一部をまとめたものです。

メソッド	説明
addCookie(Cookie cookie)	応答にクッキーを書き込みます。
addHeader(String name, String value)	ヘッダを名前/値ペアとして応答に書き込みます。ヘッダが存在する場合、この既存のヘッダ値に値が追加されます。
containsHeader(String name)	指定された応答ヘッダがすでに設定されている場合は、「true」を返します。
sendError(int sc)	指定されたステータス コードを含むエラー応答をクライアントに送信します。
setHeader(String name, String value)	指定された名前と値で、応答ヘッダを設定します。このヘッダがすでに存在している場合、既存のヘッダに設定された値は、新しい値に置き換えられます。

session オブジェクト

session オブジェクトには、特定のユーザ セッションに関する情報が格納されます。ユーザがアプリケーション内で別のページに移動しても、session オブジェクトに格納された変数は破棄されず、ユーザ セッションが継続している間、保持されます。

まだセッションを開始していないユーザからアプリケーションのページが要求された場合は、session オブジェクトが自動的に作成されます。セッションが期限切れになった場合、または中断された場合は、Web サーバーにより、session オブジェクトが破棄されます。

メモ
session 状態は、クッキーをサポートするブラウザに対してのみ保持されます。

構文

session.*Method(variables...)*

次の表は、session オブジェクトに共通するメソッドの一部をまとめたものです。

メソッド	説明
getAttribute(String name)	指定された名前を持つオブジェクトを返します。オブジェクトが見つからない場合は、ヌルを返します。
getAttributeNames()	このセッションに含まれるオブジェクト名をすべて返します。
getCreationTime()	グリニッジ標準時で 1970 年 1 月 1 日午前 0 時を基準に、セッションが作成された時刻をミリ秒単位で返します。
getId()	セッションに対する固有の ID を返します。
getLastAccessedTime()	このセッションに関連付けられている、最後にクライアント要求が行われた時刻を返します。この時刻は、グリニッジ標準時で 1970 年 1 月 1 日午前 0 時を基準に、ミリ秒単位で返します。
getMaxInactiveInterval()	クライアント アクセス間で、JRun がこのセッションを開いた状態にしておく最長時間を秒単位で返します。
removeAttribute(String name)	セッションから属性と値を削除します。
setAttribute(String name, java.lang.Object value)	属性と、関連する値をセッションに書き込みます。

第 10 章

JSP のコンパイル

コンパイラは、JSP を Java クラス ファイルにコンパイルするために JRun によって使用されるツールです。JRun には、JSP と JSPC の 2 つのバージョンのコンパイラが付属しています。最初のバージョン、JSP コンパイラは、Web クライアントが JSP を要求する際にコンパイルを実行します。

2 番目のバージョンの JSPC コンパイラでは、コマンド ラインから JSP をコンパイルできます。JSPC コンパイラは、アプリケーションを公開する前に、JSP をプレコンパイルする場合に役に立ちます。プレコンパイルを実行すると、JRun はクライアントからの最初の要求時にページをコンパイルする必要がなくなるため、システム パフォーマンスが向上します。

この章では、この 2 つのコンパイラの概要について説明します。

目次

- JSP コンパイラ 146
- JSPC コンパイラ 150

JSP コンパイラ

JSP を Java サーブレットに変換する処理の一部として、最初に JSP を Java ソース ファイルに変換します。次に、Java ソース ファイルを Java `.class` ファイルにコンパイルします。このセクションでは、JRun で使用する JSP コンパイラの構成方法について説明します。

JRun コマンド ライン コンパイラおよび JSPC コンパイラの詳細については、150 ページの「JSPC コンパイラ」を参照してください。

JSP コンパイラのプロパティの設定

JRun には `tools.jar` ファイルが付属しており、そのファイルには `javac` コンパイラが含まれています。既定では、JRun によって `javac` コンパイラが使用され、JSP を Java `.class` ファイルにコンパイルします。ただし、JRun 管理コンソール (JMC) を使用して、別のコンパイラを指定したり、既定のコンパイラのプロパティを変更することもできます。

JRun では、コンパイラを Web アプリケーション レベルで構成します。つまり、各 Web アプリケーションごとに異なるコンパイラを指定できます。

JSP コンパイラを構成するには、JMC で、[*サーバー名*] > [Web アプリケーション] > [*アプリケーション名*] > [JavaServer Pages] > [Java コンパイラ] プロパティを使用します。ここで、***サーバー名***には、Web アプリケーションのホストとなる JRun JVM の名前、***アプリケーション名***には、アプリケーションの名前を指定します。`Java compiler` プロパティは、JSP を Java `.class` ファイルにコンパイルするときに使用される、Java コンパイラを含むコマンド ラインを指定します。

メモ

`Java compiler` プロパティが空白の場合、JRun では Sun `javac` コンパイラを使用して、JSP をインプロセスでコンパイルします。

たとえば、次のようにコマンド ラインを指定します。

```
javac -nowarn -classpath %c -d %d %f
```

`Java compiler` のプロパティには、`%c`、`%d`、および `%f` の 3 つのパラメータ プレースホルダを入れる必要があります。それらは JRun によって次の値に置き換えられます。

- `%c` は、コンパイラのクラスパス設定を指定し、次のパスの組み合わせで構成されます。
  - JRun `classpath` 変数の値。このパスは JMC を使用して設定できます。
  - JSP を含む Web アプリケーションの `WEB-INF¥classes` ディレクトリのパス
  - Web アプリケーションの `WEB-INF¥lib` ディレクトリにある JAR ファイルへのパス

次の例に示すように、ディレクトリをコマンド ラインに含めることによって、独自のディレクトリをクラスパスに追加できます。

```
javac -nowarn -classpath ¥myclasses; %c -d %d %f
```

- `%d` は、コンパイル済みファイルが生成されるディレクトリに置き換えられます。
- `%f` は、JSP に対応する Java ファイル名に置き換えられます。

次の例は、Microsoft のコンパイラを指定する方法を示しています。

```
jvc /cp:c %c /dest:%d %f
```

Microsoft のコンパイラを指定すると、基になる Windows 環境を効果的に使用するコードを実装できます。JRun サーブレット エンジンが実行される仮想マシンの種類に依存することなく、Component Object Model (COM) オブジェクトを含むページコンパイル ドキュメントをコンパイルできます。ただし、COM オブジェクトを利用するコンパイル済みページを正しく実行するには、Microsoft Java Virtual Machine (JVM) 上で JRun を実行する必要があります。

JSP のコンパイルの段階は、Web アプリケーションの実行時環境に依存しないため、コンパイラと異なるベンダの JVM を使用できます。したがって、Windows プラットフォームで開発するときに、`javac` ではなく `jview` を使用して、ドキュメントをコンパイルする場合は、コンパイル済みページが COM オブジェクトを参照する限り、ドキュメントを Windows JVM で実行できます。

JSP のコンパイルのバイパス

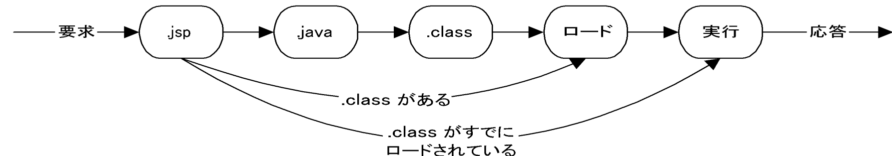
ページがクライアントによって初めて要求される場合、または要求された JSP が最後の要求の後に修正されている場合は、JRun によって JSP がコンパイルされ、そのページの `.class` ファイルが作成されます。ただし、公開するアプリケーションでは、ユーザがコンパイルをバイパスして、常にそのページの `.class` ファイルを JRun にロードさせる場合があります。

次のような理由により、JSP のコンパイルをバイパスします。

- **パフォーマンス**　静的な JSP のコンパイルによって、不必要な処理オーバヘッドがアプリケーションに追加されます。
- **セキュリティ**　JSP はテキスト ファイルであるため、誰でもアプリケーションを使用して読み取りや編集を行うことができます。JSP のコンパイルをバイパスする場合は、アプリケーションを構成する JSP を提供する必要はありません。その代わりに、対応する `.class` ファイルのみを提供します。

JSP のコンパイル プロセス

次の図は、JRun に JSP の要求が渡されたときに実行される手順を示します。



次の手順は、JSP が要求されたときに JRun が実行する動作を示しています。

1 JSP (.jsp ファイル) が解析されて、Java ソース コード (.java ファイル) が作成されます。

2 Java ソース コードが Java クラス (.class ファイル) にコンパイルされます。

3 クラスが Web サーバーのメモリにロードされます。

4 サーブレットが実行されます。

JRun では、JSP のコンパイル時にページの修正時刻が記録されます。次にそのページが要求されたときに、依存チェックが実行され、ページの修正時刻が変更されているかどうかが確認されます。ページが最後のコンパイルから変更されていない場合は、JSP を再コンパイルする必要がないため、修正時刻に変更がない場合は、手順 1 と 2 のコンパイルがバイパスされます。JSP の修正時刻とページが最後にコンパイルされた時刻が異なる場合、JRun によって手順 1 〜 4 が実行され、ページが再コンパイルされます。

JSP の開発、デバッグ、およびテストを行うときに、JSP の編集および修正を頻繁に行うため、アプリケーションの開発中は、修正した JSP を再コンパイルする必要があります。ただし、アプリケーションを公開する時点では、JSP は通常静的であるため、修正されることはありません。

JSP のコンパイルの自動的なバイパス

既定では、JRun では JSP の要求を受信すると、そのページの .jsp ファイルの存在を確認します。.jsp ファイルが存在しない場合は、自動的にそのページの .class ファイルが確認されます。.class ファイルが見つかった場合は、そのファイルがロードされて実行されます。

したがって、静的な JSP の場合、すなわち修正する必要のないページの場合は、JSP をコンパイルしてそのページの .class ファイルを作成し、次に .jsp ファイルを削除できます。これにより、JRun によって自動的にそのページの .class ファイルがロードされます。

メモ

.jsp ファイルの削除は、先に JSP をコンパイルして .class ファイルを作成してから行ってください。次のセクションでは、ページが確実にコンパイルされるようにする 2 つの方法について説明します。

JSP コンパイルの無効化

JSP コンパイルを明示的に無効にするには、`webapp.properties` ファイルを更新して、JSP を処理する JRun サーブレットを修正します。この設定は、個々の Web アプリケーションのコンパイルの有効化または無効化を可能にするため、Web アプリケーションレベルで作成します。

次の手順に従って、JSP コンパイルを無効にします。

1 テキスト エディタを使用して、Web アプリケーションの `WEB-INF/webapp.properties` ファイルを開きます。

2 次の行を追加します。

```
webapp.servlet-mapping.*.jsp=jsprt
```

3 ファイルを保存します。

4 ホット デプロイが有効になっていない場合は、JRun サーバーを再起動します。

これにより、JSP に対する要求を処理するたびに、`jsprt` サーブレット が使用されます。このサーブレットでは、JSP の存在は確認されず、対応する `.class` ファイルのみが検索されます。したがって、公開する Web アプリケーションでは、JSP を実際に公開する必要はありません。JSP はソース コード ファイルであるため、公開するアプリケーションからソース コードを省くことができます。

JSP のコンパイルを無効にすることは通常、アプリケーションの公開時にのみ作成する最適化の方法です。コンパイルを無効にする場合は、すべての JSP に対応する `.class` ファイルを提供する必要があります。したがって、アプリケーションを公開する前に、すべての JSP をプレコンパイルして `.class` ファイルを作成する必要があります。プレコンパイルは、次のいずれかの方法で実行します。

- JSP のコンパイルを有効にして、アプリケーションのすべての JSP を要求します。JSP を要求することにより、すべての JSP が確実にコンパイルされ、すべての JSP に対応する `.class` ファイルが作成されます。

  この方法の欠点は、各 JSP の要求を手作業で作成しなければならないことです。JSP が 1 つでも省かれると、クライアントがページを要求してそのページがコンパイルされていない場合にエラーが発生します。

- JSPC コンパイラを使用して、コマンド ラインからすべての JSP をコンパイルします。

  JSPC コンパイラを使用すると、アプリケーションに関連するすべての JSP を 1 つのコマンドでコンパイルできます。この方法では、すべての JSP が確実にコンパイルされます。JSPC コンパイラの詳細については、150 ページの「JSPC コンパイラ」を参照してください。

JSP のコンパイルの再有効化

JSP のコンパイルを再び有効にするには、次の手順を実行します。

1 テキスト エディタを使用して、Web アプリケーションの `WEB-INF/webapp.properties` ファイルを開きます。

2 次の行を削除します。

```
webapp.servlet-mapping.*.jsp=jsprt
```

3 ファイルを保存します。

4 ホット デプロイが有効でなければ、JRun サーバーを再起動します。

JSPC コンパイラ

JSPCコンパイラは、Web サーバーのコンテキスト の外部で JSP をコンパイルするために使用するコマンドライン ツールです。JSPC コンパイラを使用すると、JSP の要求時に JRun を使用して JSP をコンパイルするのではなく、コマンド ラインから明示的に JSP をコンパイルできます。

メモ
JSPC コンパイラは、JRun での JSP のコンパイルに使用されるコンパイラと同じコンパイラです。唯一異なる点は、JSPC コンパイラはコマンド ラインから起動することです。

いくつかの理由により、作成したサーブレットをコンパイルしなければならない場合があります。たとえば、JSP の開発段階で、ページの Web 要求によってコンパイルを開始するよりも、明示的にページをコンパイルしたほうがエラーを簡単に修正できる場合があります。または、セキュリティ上の理由から、JSP のコンパイルを無効にして、テキスト ベースの `.jsp` ファイルではなく、バイナリの `.class` ファイルのみを配布できます。

JSPC コンパイラに必要なソフトウェア

JSPC コンパイラを使用するには、次のソフトウェアが必要です。

- JRun バージョン 3.0 以降
- JDK 1.1.6 以降、または Microsoft の SDK

JSPC コンパイラの起動

次のコマンドを使用して JSPC コンパイラを起動します (JRun バージョン 3.1 ではパラメータが変更されているので注意してください)。

```
java [-classpath classpath]
          JSPC
          -j jrun_root_dir
          -s server
          -a web_application_name
          [-webroot path_to_web_root]
          [-vghn]
          [-d output_directory]
  [-compiler "compiler_spec"]
 JSP_path
```

classpath

`java` コマンドへのクラスパスを指定します (オプション)。このクラスパスは、コマンド ラインから、または CLASSPATH 環境変数を使用して指定できます。クラスパスには、次のパスを指定する必要があります。

- *JRun のホーム ディレクトリ*`¥lib` ディレクトリ内のすべての JAR ファイル
- *JRun のホーム ディレクトリ*`¥lib¥ext` 内のすべての JAR ファイル
- 使用するコンパイラによって、JDK に提供されているファイルをさらに指定しなければならない場合もあります。

j

JRun のルート ディレクトリを指定します。

s

サーバー名を指定します。

a

Web アプリケーション名を指定します。

webroot

Web アプリケーションのルート ディレクトリへの絶対パス (オプション)。この引数は、`use-webserver-root` プロパティが true に設定されている Web アプリケーションに必要です。この引数は `-w` のように省略できます。

v

JSPC によるコンパイル時に、各 JSP ファイルの名前を表示するように指定します (オプション)。

g

デバッグ メッセージを表示するように指定します (オプション)。

h

JSPC コンパイラのヘルプ メッセージを表示するように指定します (オプション)。

n

対応する `.class` ファイルよりも新しい JSP のみをコンパイルするように指定します (オプション)。

d

JSPC コンパイラが `.class` および `.java` ファイルの出力を書き込む場所を指定します (オプション)。既定では、このディレクトリは現在の作業ディレクトリに設定されます。

通常は、Web アプリケーションのディレクトリ構造の `WEB-INF¥jsp` ディレクトリを指定します。

compiler

コンパイラ、およびコンパイルに使用するコンパイラ設定を含む、引用文字列を指定します (オプション)。このパラメータを省略すると、JRun では Sun `javac` コンパイラを使用して、インプロセスで JSP をコンパイルします。コンパイラの設定の詳細については、146 ページの「JSP コンパイラ」を参照してください。

ここでコンパイラを指定する場合は、146 ページの「JSP コンパイラ」で説明したものと同じ形式を使用します。次に例を示します。

```
java JSPC -compiler "javac -nowarn -classpath %c -d %d %f" ...
```

または

```
java JSPC -compiler "jvc /cp:c %c /dest:%d %f" ...
```

JSP_path

ファイル システム上の JSP の物理パスを、`-a` パラメータで指定された Web アプリケーションへの相対パスで指定します。複数のファイルをスペースで区切って指定できます。ワイルド カード文字を使用して、複数のファイルを指定することもできます。

`-a` によって指定されたディレクトリと `JSP_path` を組み合わせて、コンパイルする JSP を指定します。たとえば、`-a` が `c:¥myapps¥store` にある Web アプリケーションで、コンパイルする JSP が `c:¥myapps¥store¥my.jsp` の場合、ページの *JSP_path* は `my.jsp` です。

複数の JSP をコンパイルする場合は、ワイルド カード文字を `JSP_path` に含めることができます。

メモ

JSPC コンパイラを使用して、`use-webserver-root` プロパティが true に設定されている Web アプリケーションを処理する場合は、`-webroot` 引数を指定する必要があります。たとえば、default JRun サーバーの既定のアプリケーションでは、`use-webserver-root` が true に設定されています。

JSPC コンパイラの例

次に、ページ foo.jsp をコンパイルする例を示します。このページは、c:¥myapps¥store にドキュメント ルート ディレクトリを持つ Web アプリケーションの一部です。この例には、既定の JRun コンパイラを書き換えるコンパイラの指定も含まれています。この例では、JSPC コンパイラを実行するために必要な設定は、CLASSPATH 環境変数に含まれています。

```
java JSPC -j c:¥jrun
          -s qatest
          -a storeapp
          -d c:¥myapps¥store¥WEB-INF¥classes
          -compiler "pathtojikes¥jikes -classpath %c -d %d %f"
          foo.jsp
```

次の例では、既定の JRun コンパイラを使用して、複数の JSP をコンパイルします。JSP_path にワイルドカードが使用されていることに注意してください。

```
java JSPC -j c:¥jrun -s qatest -a storeapp -v -d
    c:¥myapps¥store¥WEB-INF¥classes *.jsp store¥*.jsp
```

第 11 章

JSP でのカスタム タグの作成

JavaServer Pages バージョン 1.1 の仕様書には、タグ ライブラリに関するフレームワークが記述されています。この章では、JSP でカスタム タグ ハンドラをコーディングする方法について説明します。

目次

- 概要 156
- カスタム タグの概念 157
- 構文 159
- 使用方法 163
- サンプル 163
- 高度な使用方法 168

概要

関連する機能のセット のカプセル化、ロジック から のプレ ゼンテーション の分離、および JSP の操作性の強化を行うには、カスタム タグを使用します。JRun は、エンジン自体と、20 以上のカスタム タグの集合である JRun 3.0 タグ ライブラリの両方によって、この機能をサポートする最初の製品の 1 つでした。

本来考えられるように、カスタム タグ ハンドラは Java で書かれており、`Tag` および `BodyTag` インターフェイスから 1 つまたは複数のメソッドを実装して、タグが呼び出されたときに適切な処理を実行します。また、Java 開発者は、タグ ライブラリ記述子 (TLD) ファイルに要素をコーディングし、タグ拡張情報 (TEI) クラスをコーディングして機能を追加する必要があります。カスタム タグのコーディングには、Java、カスタム タグ API、および JSP による開発経験が必要です。Java ベースのカスタム タグ ハンドラのコーディング方法については、JavaServer Pages バージョン 1.1 の仕様書または255 ページの第 22 章「カスタム タグとタグ ライブラリの作成」を参照してください。

JRun Server Tags (JST) は、JSP でカスタム タグ ハンドラの実装に使用できる技術です。Java コードは不要です。JST 技術では、JSP プログラマがカスタム タグの能力を活用できる JSP 構文が採用されています。JST を使用すると、Java で書かれたカスタム タグよりも短期間でアプリケーションを開発できます。ページが JST ページであることを示すには、そのページの名前に拡張子 `.jst` を付けます。

JST と Java ベースのタグ ハンドラの関係は、JSP とサーブレットの関係と同じです。つまり、サーブレットは Web アプリケーション プログラミングに対して Java ベースの強力なソリューションを提供し、JSP はその機能を損ねることなく操作性を向上させる、あるレベルの抽象化を実装します。また、JSP がサーブレットに変換されるように、JST ページは Java ベースのタグ ハンドラに変換されます。この技術によって生成されたタグ ハンドラ クラスは、TLD ファイルを指定するだけで、JSP 1.1 カスタム タグをサポートするすべてのサーブレット エンジンに移植可能です。

カスタム タグの概念

JST を使用すると Java のコーディングから解放されますが、JST ページをコーディングするには、基本的なカスタム タグの概念を認識している必要があります。

- **タグのコーディング**　JSP 開発者はカスタム タグの開始タグと終了タグの両方をコーディングします。終了タグでは、短縮された構文 (例 : `<foo:header/>`) またはスタンドアロン終了タグ (例 : `<foo:header></foo:header>`) のいずれかを使用できます。カスタム タグを使用すると、開始タグと終了タグとの間に本文テキストを入れて本文テキストと対話できます。
- **呼び出しポイント**　タグ ハンドラは開始タグと終了タグを使用して呼び出すことができます。また、本文テキストにもアクセスできます。Java ベースのカスタム タグ ハンドラでこの機能を実装するには、`doStart`、`doEnd`、および `doAfterBody` メソッドをコーディングします。JST によって、`<tagmethod="START|END|AFTER_BODY" %>` ディレクティブによって制御される 1 つのメソッドの機能が自動的に提供されます。ほとんどのカスタム タグでは、`doEnd` メソッドだけを使用するので、既定値は `doEnd` です。ほかのメソッドは、宣言内でメソッドを Java でコーディングすることによって実装できます。
- **本文コンテンツとの対話**　タグ ハンドラでは本文コンテンツの処理、ループ化、または変更が可能です。Java ベースのカスタム タグ ハンドラでこの機能を実装するには、`BodyTagSupport` クラスを拡張し、`doAfterBody` メソッドをコーディングし、`bodyContent` 変数によって本文テキストにアクセスします。JST では、`<%@ tag type="LOOPING|BUFFERED" %>` ディレクティブを含めることによって、このタイプの処理を実行できます。
- **戻りコード**　各メソッドは戻り値を使用して続行方法を示します。Java ベースのカスタム タグ ハンドラでは定数を返します。JST では、暗黙の `returnCode` 変数を使用してこれらの同じ定数を返します。
- **属性**　JSP は属性をカスタム タグに渡すことができます。Java ベースのカスタム タグでは、TLD ファイルで各タグの属性を宣言します。JST では、`tagAttribute` ディレクティブによって属性を宣言します。
- **スクリプト変数**　カスタム タグではスクリプト変数を作成できます。Java ベースのカスタム タグでは、TEI クラスでスクリプト変数を宣言します。JST では、`tagVariable` ディレクティブによってスクリプト変数を宣言します。また、`isValid` メソッドを定義する宣言も使用して属性の検証を実行することもできます。

次の表は、Java ベースのカスタム タグ ハンドラと JSP で書かれたカスタム タグの違いを要約したものです。

Java ベースのカスタム タグ ハンドラ	JSP ベースのカスタム タグ
タグ ハンドラ Java ソース ファイル	JSP ファイル
doStartTag メソッド	<%@ tag method="START" %>
doEndTag メソッド	<%@ tag method="END" %>
doAfterBody メソッド	<%@ tag method="AFTER_BODY" %>
TagSupport クラスから継承します。	<%@ tag type="TAG" %>
BodyTagSupport クラスから継承します。	<%@ tag type="LOOPING|BUFFERED" %>
タグ ハンドラ内に複数のメソッドを実装します。	最初のメソッドは、<% tag method="method-type" %> を使用して実装します。その他のメソッドは、宣言に囲まれた Java ベースのメソッドを使用して実装します。
TLD ファイルで属性を定義します。	<%@ tagAttribute name="name" type="type" required="true|false" rtexpr="true|false" setter="true|false" default="default attribute value" %>
TEI クラスでスクリプト変数を定義します。	<%@ tagVariable (id="id"|name="name") type="type" scope="AT_BEGIN|AT_END|NESTED" %>

これらのトピックの詳細については、255 ページの第 22 章「カスタム タグとタグ ライブラリの作成」を参照してください。

この章の後のセクションでは、JST のコーディングを開始するための情報について説明します。

- 「構文」159 ページ
- 「使用方法」163 ページ
- 「サンプル」163 ページ

構文

JSP でカスタム タグを作成するには、次のディレクティブについて、変更された構文または新しい構文を使用する必要があります。

- 「tag ディレクティブ」159 ページ
- 「tagAttribute ディレクティブ」160 ページ
- 「tagVariable ディレクティブ」161 ページ
- 「taglib ディレクティブ」162 ページ

tag ディレクティブ

tag ディレクティブを使用して、次のタイプの情報を宣言します。

- **Method**　JST ファイルが begin タグで呼び出されるか、end タグで呼び出されるか、または本文コンテンツを反復した後で呼び出されるかを指定します。method 属性を持つ tag ディレクティブで指定します。
- **Type**　タグと本文コンテンツが対話するかどうかを指定します。type 属性を持つ tag ディレクティブで指定します。

method 属性を持つ tag ディレクティブ

JST がいつ呼び出されるかを示すには、method 属性を持つ tag ディレクティブをコーディングします。次の構文を使用します。

```
<%@ tag method="method" %>
```

method 属性には次のいずれかを指定できます。

- START　開始タグが検出されたときに JST ページを呼び出します。START の許容可能な戻りコードは、EVAL_BODY_INCLUDE および SKIP_BODY です。
- END　終了タグが検出されたときに JST ページを呼び出します。END の許容可能な戻りコードは、EVAL_PAGE および SKIP_PAGE です。
- AFTER_BODY　本文の後に JST ページを呼び出します。AFTER_BODY の許容値は、本文のループ化を使用する EVAL_BODY_TAG と、SKIP_BODY です。AFTER_BODY と END の違いは、許容可能な戻り値にあります。また、AFTER_BODY を指定した場合は、`<%@ tag type="LOOPING|BUFFERED" %>` も指定する必要があります。

このディレクティブはオプションです。既定値は END です。

メソッドおよび戻りコードの詳細については、JavaServer Pages バージョン 1.1 の仕様書または255 ページの第 22 章「カスタム タグとタグ ライブラリの作成」を参照してください。

type 属性を持つ tag ディレクティブ

JST が本文コンテンツと対話するかどうかを示すには、type 属性を持つ tag ディレクティブをコーディングします。次の構文を使用します。

```
<%@ tag type="type" %>
```

type

type 属性には次のいずれかを指定できます。

- LOOPING JST は、BodyTag インターフェイスを実装するカスタム タグに変換されます。BODY_TAG を使用すると、本文コンテンツと対話してループ化できます。
- BUFFERED JST は、BodyTag インターフェイスを実装するカスタム タグに変換されます。BODY_TAG を使用すると、本文コンテンツと対話してループ化できます。

<%@ tag method="AFTER_BODY" %> を指定するときに、このディレクティブをコーディングします。

メモ
現在、LOOPING および BUFFERED では同じ結果が生成されます。ただし、JavaServer Pages バージョン 1.2 の仕様書には異なる結果が紹介されています。LOOPING は、本文を反復しますが本文コンテンツを返しません。BUFFERED は、本文コンテンツをループ化して返します。これ以降の JST 技術のリリースでは、この違いが実装されます。詳細については、JavaServer Pages バージョン 1.2 の仕様書を参照してください。

---

tagAttribute ディレクティブ

JST ページに渡される属性を定義するには、tagAttribute ディレクティブをコーディングします。次の構文を使用します。

```
<%@ tagAttribute name="name" type="class" required="true|false"
                 rtexpr="true|false" setter="true|false"
                 default="default attribute value" %>
```

name

必須。属性名を指定します。

type

オプション。たとえば Integer など、属性の Java クラス名を指定します。インポート ステートメント をコーディングしていない場合や、クラスが java.lang でない場合は、完全修飾クラス名を指定します。既定値は String です。

required

オプション。属性が必須かどうかを示します。true または false を指定します。既定値は false です。

rtexpr

オプション。属性に実行時式を含めることができるどうかを示します。true または false を指定します。既定値は false です。

setter

オプション。JRun によって get*AttributeName* メソッドを自動的に作成する場合は true、カスタマイズされた get*AttributeName* メソッドを指定する場合は false を指定します。既定値は true です。

default

オプション。属性の既定値を指定します。

例

次の例は、handle という名前の属性の宣言を示します。

```
<%@ tagAttribute name="handle" type="String"
        required="true" rtexpr="true" %>
```

この tagAttribute ディレクティブによって、JST ページでは次の処理が実行されます。

- handle と呼ばれる専用のインスタンス変数を作成します。
- setHandle(java.lang.String) と呼ばれるメソッドを作成します。
- TagLibraryInfo.getTag メソッド によって返される TagInfo オブジェクト に属性を組み込みます。

tagVariable ディレクティブ

スクリプト 変数を定義するには、tagVariable ディレクティブをコーディングします。次の構文を使用します。

```
<%@ tagVariable (id="attribute"|name="name") type="class"
        scope="scope" %>
```

id

スクリプト変数の名前である値を持つタグ属性の名前を指定します。id 属性または name 属性のいずれかを指定する必要があります。両方を指定することはできません。

name

name 属性ではスクリプト変数の名前を指定します。id 属性または name 属性のいずれかを指定する必要があります。両方を指定することはできません。

type

オプション。たとえば Integer などのスクリプト変数の Java クラス名を指定します。インポート ステートメントをコーディングしていない場合や、クラスが java.lang でない場合は、完全修飾クラス名を指定します。既定値は String です。

scope

オプション。スクリプト変数の範囲を指定します。次のいずれかを指定できます。

- AT_BEGIN JSP では、タグの本文内と JSP の残りの部分でスクリプト変数を使用できます。既定値は AT_BEGIN です。
- AT_END JSP では、JSP の残りの部分でスクリプト変数を使用できます。
- NESTED JSP では、タグの本文内でスクリプト変数を使用できます。

scope 属性の使用方法については、271 ページの「TEI クラスのコーディング」を参照してください。

例

次の例では、tagVariable ディレクティブを使用して、filename という名前のスクリプト変数を作成します。この変数は、タグの本文内と、JSP の残りの部分で使用できます。

```
<%@ tagVariable name="filename" type="String"
        scope="AT_BEGIN" %>
```

taglib ディレクティブ

JST を使用する JSP をコーディングするときは、Java ベースのカスタム タグを使用する JSP のコーディングに必要である場合と同様に、taglib ディレクティブを指定するだけです。唯一の違いは、uri 属性で、JAR ファイルの名前ではなく、.jst ファイルが保存されているディレクトリを指定することです。次の構文を使用します。

```
<%@ taglib prefix="prefix" uri="directory containing jst pages" %>
```

prefix

必須。JSP 開発者が、uri 属性で指定されたディレクトリから JST を呼び出すときに使用する接頭辞を指定します。

uri

必須。JST ページが含まれているディレクトリの名前を指定します。これは Web アプリケーションのルートに相対的です。

例

次の例は、myjst という接頭辞を使用して、JST ページが webapproot/jstdir ディレクトリにあることを指定します。

```
<%@ taglib prefix="myjst" uri="/jstdir" %>
```

使用方法

最も単純な使用例では、拡張子 `.jst` を持つ JSP ファイルを保存して、JRun によって終了タグで呼び出されるカスタム タグとして扱います。これによって、コードの再利用に便利なメカニズムを提供します。ただし、JST 技術には、JavaServer Pages バージョン 1.1 の仕様書に概説されている機能より多くの機能が用意されています。それらの機能を次に示します。

- 開始タグと終了タグでの呼び出し
- 属性
- 本文コンテンツとの対話
- ループ
- スクリプト変数

サンプル

このセクションでは、次について説明します。

- 「単純な例」163 ページ
- 「属性との対話」164 ページ
- 「本文コンテンツとの対話」165 ページ
- 「ループ」166 ページ
- 「スクリプト変数の使用法」167 ページ

単純な例

JST ページ

次のページをコーディングして、アプリケーションのルート ディレクトリに `simple.jst` として保存します。

```
<%-- simple.jst --%>
<p>Hello from the JRun Server Tag!</p>
```

JSP

次のページをコーディングし、アプリケーションのルート ディレクトリに、任意の名前を付けて保存します。`URI="/"` によってルート ディレクトリ内のタグが検索されます。

```
<%@ taglib prefix="t" uri="/" %>

<t:simple/>
```

JST をテストするには、JSP ページをブラウザに表示します。JST ページのテキストが表示されます。

属性との対話

JST ページ

次の JST ページはエラー メッセージの表示に使用されます。前のページに戻るフォームと、2 つの属性が表示されます。

```
<%-- message.jst --%>
<%@ tagAttribute name="messagetext" type="String" required="true"
        rtexpr="true"%>
<%@ tagAttribute name="messagetype" type="String" required="true"
        rtexpr="true"%>

<html>
<body>
<form>
<h1><%= messagetype %></h1>
<p><%= messagetext %></p>
<p>  <INPUT TYPE="button" VALUE="Back"
        onClick="history.back()">
</form>
</body>
</html>
<%
out.flush();
out.close();
%>
```

JSP

次の JSP では、message タグを呼び出しています。

```
<%@ taglib prefix="t" uri="/" %>
<%
  // userPassword 変数を設定していると想定します。
  if(userPassword.length() == 0) {
  // ページをテストするには、前の行をコメントに変え、
  // 次の行をコメントからコードに戻します。
  // if(0 == 0) {
%>
<t:message messagetype="Validation Error"
           messagetext="Please enter a password">
</t:message>
<% } %>
...
```

本文コンテンツとの対話

JST ページ

```
<%-- bodyinteract.jst --%>
<%-- 列挙のインポート --%>
<%@ page import="java.util.Enumeration" %>
<%-- 本文コンテンツを調べるには method="AFTER_BODY" を使用する必要があります。
        --%>
<%@ tag method="AFTER_BODY" %>
<%-- また、本文コンテンツを調べるには、type="BUFFERED" または TYPE="LOOPING"
     を使用する必要があります。--%>
<%@ tag type="BUFFERED" %>

<% // まず、本文コンテンツを取得します。
   String body = bodyContent.getString();
   // 本文の長さを取得します。本文には、HTML タグ文字も含まれているので注意してくだ
        さい。
   int bclength = body.length();
   callerPageContext.getOut().print("<hr>HTML for body contains " +
        bclength + " characters.");
   // ループしないで、本文コンテンツを 1 回だけ調べてください。
   returnValue=SKIP_BODY; %>
```

JSP

```
<html>
<body>
<%@ taglib prefix="t" uri="/" %>
<%@ page import="java.util.*" %>
<h1>Interacting with Body Content</h1>
<t:bodyinteract>
 <hr>
 <p>First line
 <br>Second line
 <br>Third line
</t:bodyinteract>

</body>
</html>
```

ループ

JST ページ

```
<%-- loop.jst --%>
<%-- 列挙のインポート --%>
<%@ page import="java.util.Enumeration" %>
<%-- ループさせるには method="AFTER_BODY" を使用する必要があります。--%>
<%@ tag method="AFTER_BODY" %>
<%-- ループさせるには type="BUFFERED" または TYPE="LOOPING" を使用する必要が
        あります。--%>
<%@ tag type="LOOPING" %>
<%-- 渡されたオブジェクトの属性を宣言します。--%>
<%@ tagAttribute name="thisEnum" type="Enumeration" required="true"
        rtexpr="true"%>
<%@ tagAttribute name="var" type="String" required="true"
        rtexpr="true"%>
<%-- スクリプト変数を定義します。id 属性は var 属性の値から
     スクリプト変数の名前を受け取ります。--%>
<%@ tagVariable id="var" scope="NESTED" %>

<%
  if(thisEnum.hasMoreElements()) {
  // jst には独自の pageContext があるので、
  // callerPageContext を使用して呼び出し側 JSP 内で設定します。
  callerPageContext.setAttribute("header", thisEnum.nextElement());
    returnValue=EVAL_BODY_TAG; // ループ
  }
  else {
    returnValue=SKIP_BODY;
  }
%>

<%-- この例では、複数のメソッドの実装方法も示します。
     JRun では以前のコードは doAfterBody メソッドに変換されます。
    この例では、doAfterBody に加えて実行する
    doStartTag メソッドのコーディング方法を示します。--%>
<%!
public int doStartTag() throws JspException {

  if(thisEnum.hasMoreElements()) {
    // jst には独自の pageContext があるので、
   // callerPageContext.setAttribute を使用して呼び出し側 JSP 内で設定します。
    callerPageContext.setAttribute("header", thisEnum.nextElement());
    // Java ベースのメソッドでは、returnValue を設定する代わりに return を使用
        します。
    return EVAL_BODY_TAG; // 本文テキストの使用を可能にします。
  }
  else {
    return SKIP_BODY;
  }
}
%>
```

JSP

```
<%-- loop.jsp --%>
<html>
<body>
<%@ taglib prefix="t" uri="/" %>
<%@ page import="java.util.*" %>
<h1>Looping through Headers</h1>
<table border="1">
  <tr>
    <th>Name</th>
    <th>Value</th>
  </tr>
<t:loop var="header" thisEnum="<%= request.getHeaderNames() %>">
   <tr>
     <td><%= header %></td>
     <td><%= request.getHeader(header) %></td>
   </tr>
</t:loop>

</table>
</body>
</html>
```

スクリプト変数の使用法

JST ページ

```
<%-- scriptingvar.jst --%>
<%@ tagVariable name="myName" scope="AT_BEGIN" %>

<%-- 呼び出し側ページのコンテキストに値を保存します。--%>
<% callerPageContext.setAttribute("myName", "Christopher"); %>

<p><b>Inside the tag:</b>This tag sets the
myName scripting variable, which
is used later in the calling JSP.
```

JSP

```
<html>
<body>
<%@ taglib prefix="t" uri="/" %>
<h1>Using Scripting Variables</h1>
<p>Before the tag.

<t:scriptingvar/>

<p>After the tag.
<p>The myName variable was set in the custom tag but
because scope="AT_BEGIN", it is available later
in the page.
<p>myName is <%= myName %>

</body>
</html>
```

高度な使用方法

複数のハンドラの実装

既定では、JST ページでは doStart、doEnd、または doAfterBody のいずれかのハンドラを実装します。ただし、Java で追加のハンドラをコーディングして宣言で定義すると、JST ページにハンドラを追加できます。例については、166 ページの「ループ」を参照してください。

ファイル拡張子 .jst の再マッピング

JSP ベースのカスタム タグには、.jst 以外のファイル拡張子を使用できます。この拡張子を変更するには、JRun サーバー用の local.properties ファイルの jsp.jst_suffix プロパティを変更します。次の例では、JST 接尾辞を .jtg に設定します。

```
jsp.jst_suffix=jtg
```

第 12 章

JSP の例

この章では、一般的な機能を実行するために使用される JSP の例について説明します。

目次

- 要求の処理と応答の生成 170
- JSP から別の JSP の呼び出し 171
- セッションのトラッキング 173
- アプリケーション オブジェクトの使用 176
- タグ ライブラリの使用 178

要求の処理と応答の生成

要求の受信、要求からのパラメータの抽出、および応答の生成は JSP の最も基本的なアクションです。この例にある JSP では、名前と口座残高の 2 種類の要求パラメータを使用します。その後、このページにより、入力残高に基づいてクライアントにメッセージが出力されます。

この例の要求 URL は次のとおりです。

```
http://localhost/example1.jsp?bal=222.45&fName=Steven
```

使用している Web サーバーのドキュメント ルート ディレクトリに、この例で使用される JSP の `example1.jsp` を配置します。`example1.jsp` の内容は次のとおりです。

```
<html>
<head><title>Balance Example</title></head>
<body>
<p>
<h1> Do you have enough?</h1>
<br>

<%-- 要求オブジェクトからパラメータを取得します。--%>
<% String firstName = request.getParameter("fName"); %>
<% String balance = request.getParameter("bal"); %>

<%-- bal パラメータを String から double に変換します。--%>
<% double accountBalance = Double.valueOf(balance).doubleValue(); %>

<%-- 出力結果 --%>
Balance for <%=firstName %>:<%=accountBalance %> <br>
<br>

<%-- 残高は十分にあるか? --%>
<% if(accountBalance <= 100.00) { %>
  <b> Get a Job. </b> <br>
<% } %>

</body>
</html>
```

要求パラメータはすべて文字列として渡されるので、入力パラメータ `bal` を double に変換する必要があります。この例では、double で表される残高に対してコンディショナル ロジックを実行します。

JSP から別の JSP の呼び出し

複数の JSP から構成されるアプリケーションを作成する場合があります。別のページを呼び出す JSP を記述する場合、呼び出し方法を制御するために次の 2 つの項目について決めておく必要があります。

- 2 つの JSP の間で情報を渡す方法

  このセクションの例では、JSP `request` オブジェクトを使用して、複数の JSP で情報を共有しています。`request` オブジェクトは、情報が現在処理中の要求に対してのみ重要で、要求の処理が終了したら保持する必要がない場合に使用します。後の例では、要求の処理が完了した後でも情報を保持する必要がある場合に、JSP `session` オブジェクトと `application` オブジェクトを使用して JSP 間で情報を渡しています。

- 呼び出された JSP から呼び出し側 JSP に制御を戻すかどうか

  目的のページで実行が完了してから呼び出し側ページに制御を戻すには、`jsp:include` アクションを使用して、この呼び出しを行います。

  目的のページに制御を渡した後に呼び出し側ページが終了した場合、目的のページから呼び出し側ページに制御が戻されることはありません。この場合、`jsp:forward` アクションを使用して呼び出しを行います。

この例は、前に示した 170 ページの「要求の処理と応答の生成」の例を修正したものです。この例では、入力残高が 100.00 ドル未満の場合に、`example2.jsp` の JSP により `jsp:include` アクションが使用されて `example2a.jsp` が呼び出されます。`example2.jsp` から `example2a.jsp` にパラメータを渡すには、JSP `request` オブジェクトを使用します。`example2.jsp` は次のとおりです。

```
<html>
<head><title>Balance Exmaple</title></head>
<body>
<p>
<h1> Do you have enough?</h1>

<br>
<%-- 要求オブジェクトからパラメータを取得します。--%>
<% String firstName = request.getParameter("fName"); %>
<% String balance = request.getParameter("bal"); %>

<%-- bal パラメータを String から double に変換します。--%>
<% double accountBalance = Double.valueOf(balance).doubleValue(); %>

<%-- 出力結果 --%>
Balance for <%=firstName %>:<%=accountBalance %> <br>

<%-- 残高は十分にあるか? --%>
<% if(accountBalance <= 100.00) { %>
  <% request.setAttribute("needsJob", "true"); %>
 <jsp:include page="example2b.jsp" flush="true"/>
<% } %>

</body>
</html>
```

入力パラメータは request オブジェクトから JSP example2a.jsp に渡されます。このパラメータに従って、出力が条件付けられます。example2a.jsp の内容は次のとおりです。

```
<html>
<body>
<p>

<%-- 要求オブジェクトからパラメータを取得します。--%>
<% String jobStatus = (String) request.getAttribute("needsJob"); %>

<%-- 残高は十分にあるか? --%>
<% if("true".equals(jobStatus)) { %>
You need a job.<br><br>
<h2>Available positions include:</h2>
Software Engineer<br>
QA<br>
Technical Writer<br>
<% } %>

</body>
</html>
```

セッションのトラッキング

HTTP プロトコルはステートレスです。つまり、Web サーバーには複数の要求/応答を通してクライアントをトラッキングできません。しかし、JRun ではセッショントラッキング機能がサポートされているので、Web サイトで情報を格納し、複数の要求を通してクライアントをトラッキングできます。クライアントがアプリケーション内で別のページに移動しても、情報は破棄されず、ユーザ セッションが継続している間、保持されます。

特定のユーザ セッションに関する情報を格納するには、JSP `session` オブジェクトを使用します。ユーザがアプリケーション内で別のページに移動しても、`session` オブジェクトに格納されたデータは破棄されず、ユーザ セッションが継続している間、保持されます。

有効に設定すると、まだセッションを開始していないユーザからアプリケーションのページが要求された場合、`session` オブジェクトが自動的に作成されます。セッションが期限切れになった場合、または中断された場合は、Web サーバーにより、`session` オブジェクトが破棄されます。

JSP から `session` オブジェクトに情報を書き込むことができます。クライアントから要求された別の JSP で、この情報にアクセスすることが可能です。たとえば、ユーザのショッピング カートに ID を割り当てておくと、クライアントによりカートの内容が追加、削除、または修正されたときに、Web サイトからショッピング カートに関する情報にアクセスできます。この ID は `session` オブジェクトに格納できます。

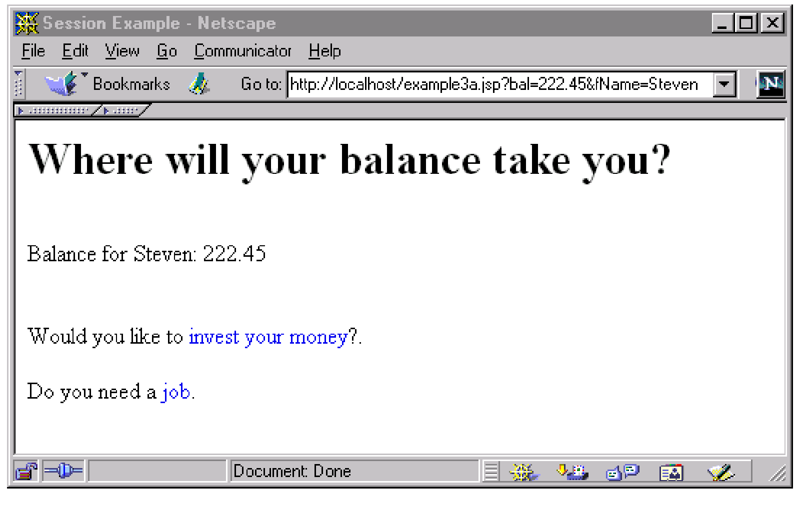
JRun では、セッションのトラッキングにクッキーを使用します。したがって、この例を実行するには、ユーザのブラウザでクッキーをサポートしている必要があります。JRun セッショントラッキングの有効と無効を切り替えるには、JRun 管理コンソールを使用します。セッション トラッキングを制御する方法の詳細については、『JRun セットアップ ガイド』を参照してください。

次の例では、JSP が 3 ページ使用されています。最初のページ `example3a.jsp` では、URL 要求から名前と口座残高が取得され、この情報が `session` オブジェクトに書き込まれます。これで、クライアントがアクセスするすべての JSP から、この情報が使用できます。

`example3a.JSP` には、クライアントからほかの JSP へのアクセスに使用される 2 種類のリンクが表示されます。最初のリンクでは投資オプション、2 番目のリンクでは求人状況がリストに表示されます。この例の要求 URL は次のとおりです。

http://localhost/example3a.jsp?bal=222.45&fName=Steven

次の図は、この URL に対する example3a.jsp の出力を示します。



example3a.jsp は次のとおりです。

```
<html>
<head><title>Session Example</title></head>
<body>
<p>
<h1> Where will your balance take you?</h1>
<br>
<%-- 要求オブジェクトからパラメータを取得します。--%>
<% String firstName = request.getParameter("fName"); %>
<% String balance = request.getParameter("bal"); %>

<%-- bal パラメータを String から double に変換します。--%>
<% double accountBalance = Double.valueOf(balance).doubleValue(); %>

<%-- セッション オブジェクトに入力パラメータを書き込みます。
 セッション オブジェクトは double を格納できません。
 まず、オブジェクトを Double に変換する必要があります。
--%>
<%
  session.setAttribute("userName", firstName);
  Double tempAccountBalance = new Double(accountBalance);
  session.setAttribute("userBalance", tempAccountBalance);
%>

<%-- 出力結果 --%>
Balance for <%=firstName %>:<%=accountBalance %> <br>
<br><br>
Would you like to <A href="example3b.jsp">invest your money</a>?.
<br><br>
Do you need a <a href="example3c.jsp">job</a>.
</body>
</html>
```

example3b.jsp ページにより、session オブジェクトからクライアントの名前と残高が調べられ、投資オプションが提示されるか、または求人を調べるためのプロンプトが表示されます。

```
<html>
<body>

<%-- セッション オブジェクトから名前を取得します。--%>
<% String fName = (String) session.getAttribute("userName"); %>

<h2>Hi <%=fName %> </h2>

<%-- 残高を取得し、double に変換します。--%>
<%
  Double tempBal = (Double)session.getAttribute("userBalance");
  double accountBalance = tempBal.doubleValue();
%>

<%-- 投資に十分な残高があるか? --%>
<% if(accountBalance > 100.00) { %>
  Your balance of $<%= accountBalance %> is sufficient for investing.
  We offer a number of investment opportunities, including: <br>
    <li>Bonds<br>
    <li>CDs<br>
    <li>Mutual funds<br>
<% } %>

<%-- 残高が少なすぎるか? --%>
<% if(accountBalance <= 100.00) { %>
   Your balance is too low for investing.  It looks like you need a <a
        href="example3c.jsp">job</a>.
<% } %>
</body>
</html>
```

The example3c.jsp page greets the user using the name passed in the session object and lists available jobs:

```
<html>
<body>

<%-- セッション オブジェクトからパラメータを取得します。--%>
<% String fName = (String) session.getAttribute("userName"); %>

<h2>Hi <%=fName %></h2>

<h2>Available positions include:</h2>
Software Engineer<br>
QA<br>
Technical Writer<br>
</body>
</html>
```

アプリケーション オブジェクトの使用

前の例では、JSP session オブジェクトを使用し、複数の HTTP 要求を通してクライアント情報をトラッキングする方法について説明しました。もう 1 つのオブジェクトである application オブジェクトを使用すると、アプリケーション全体に関する情報をトラッキングできます。アプリケーション内の JSP からは、application オブジェクトの情報にアクセスできます。

application オブジェクトは通常、指定されたアプリケーションのすべてのユーザ間で情報を共有するために使用されます。たとえば、application オブジェクトに、アプリケーション内の JSP で使用可能な既定の情報を格納できます。

次の例 index.jsp では、application オブジェクト にパラメータが 2 つ設定されます。このページは通常、クライアントにとって Web アプリケーションへの開始ポイントなので、index.jsp を使用してパラメータを設定します。

```
<%-- アプリケーション レベル設定を定義します。--%>
<%
  application.setAttribute("appName", "Application Object Example");
  application.setAttribute("counter","0");
%>

<HTML>
<BODY>
<H1>Application Object Example </H1>

<h2>Display the default application settings</h2>
<%--
 ここで、
 またはアプリケーションの別の JSP ページで、アプリケーション パラメータに
 アクセスするか、変更してください。
--%>
<% String appName = (String) application.getAttribute("appName"); %>
The name of this application is "<%= appName %>"
<br><br>

<% String counter = (String) application.getAttribute("counter"); %>
The counter value = <%= counter %>

</BODY>
</HTML>
```

JRun では、JRun 管理コンソール (JMC) を使用して、application オブジェクトに初期化パラメータを設定することもできます。初期化パラメータは、ユーザがアプリケーションのコンポーネントに最初に要求を作成するときに、その application オブジェクトに書き込まれます。JMC を使用して初期化パラメータを設定する場合は、application.getInitParameter メソッドを使用して、パラメータにアクセスします。

application オブジェクトに初期化パラメータを設定するには、JMC で次のコマンドを使用して、アプリケーション変数エディタを開きます。

サーバー名 > Web アプリケーション > *アプリケーション名* > アプリケーション変数

たとえば、次の 2 つのパラメータを定義します。

- My Application に初期化パラメータ appName を設定
- この初期化パラメータの counter に 0 を設定

次の例 index.jsp では、これらの初期化パラメータにアクセスしています。

```
<HTML>
<BODY>
<H1>Application Object Example </H1>

<h2>Display the default application settings</h2>
<%--
 ここで、
 またはアプリケーションの別の JSP ページで、アプリケーション パラメータにアクセス
        するか、変更してください。
--%>
<% String appName = (String) application.getInitParameter("appName");
        %>
The name of this application is "<%= appName %>"
<br><br>

<% String counter = (String) application.getInitParameter("counter");
        %>
The counter value = <%= counter %>

</BODY>
</HTML>
```

タグ ライブラリの使用

タグ ライブラリを使用すると、JSP 内でカスタム タグを使用できます。タグ ライブラリを定義するには、JSP 内で `taglib` ディレクティブを使用します。タグ ライブラリの定義が完了すれば、このライブラリにあるタグにアクセスできるようになります。

JRun では、*JRun のホーム ディレクトリ*`/servers/lib/jruntags.jar` ファイルにタグ ライブラリの例が添付されています。JSP ファイルに次のステートメントを挿入して、このタグ ライブラリを JRun サーバー上で実行されている Web アプリケーション内で使用できます。

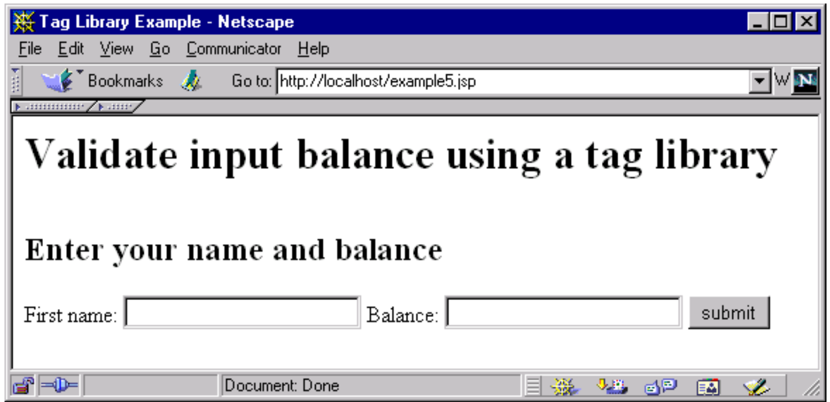
```
<%@ taglib uri="jruntags" prefix="tagLibPrefix" %>
```

ここで、*tagLibPrefix* は JRun タグ ライブラリで、タグのためにユーザが定義した接頭辞です。

`global.properties` ファイルには、上記の `taglib` のディレクティブを使用している `jruntags.jar` ファイルを参照できるマップが含まれています。すべての JRun サーバー上の Web アプリケーションで、ユーザがカスタマイズしたタグ ライブラリを使用可能にするために、同様のマップを作成できます。あるいは、特定の JRun サーバー上の Web アプリケーションで、そのタグ ライブラリを使用可能にするために、そのサーバーの `local.properties` ファイルにマップを追加できます。

この例では、デモ用のタグ ライブラリから 2 つのタグを使用しています。`form` と `input` です。これらのタグを使用すると、HTML フォームを作成し、クライアントがこのフォームに適切な値を入力したことが検証できます。

この例では、`example5.jsp` の JSP により、クライアントが `First Name` フィールドに文字列、`Balance` フィールドに浮動小数点値を入力したことが確認されます。クライアントが入力した値が適切でなかった場合、これらのタグにはエラー メッセージが表示され、クライアントは値を入力し直すことができます。次の図は、`example5.jsp` の出力を示します。



example5.jsp ファイルには次のコードが含まれています。

```
<html>
<head><title>Tag Library Example</title></head>
<body>
<p>
<h1>Validate input balance using a tag library</h1>
<br>

<%-- ライブラリ タグのタグ ライブラリとタグ接頭辞を定義します。--%>
<%@ taglib uri="jruntags" prefix="mylib" %>

<h2>Enter your name and balance</h2>

<%--
 タグ mylib:form を使用して、2 種類の入力を持つフォームを定義します。
 各入力について、入力タイプを定義してください。First Name のタイプは
 text です。Balance のタイプは float です。
 クライアントが入力した値が適切でない場合は、
 エラー メッセージが表示されます。入力内容が正しい場合は、
 要求が example5a.jsp に渡されます。
--%>
<mylib:form name="form1" action="example5a.jsp">
First name:<mylib:input name="fName" type="text" required="true" />
Balance:<mylib:input name="bal" type="float" required="true" />
<input type="submit" value="submit">
</mylib:form>

</body>
</html>
```

送信を実行すると、フォーム データが example5a.jsp ファイルに送信されます。次に、このファイルは request オブジェクトからフォーム情報を取得します。

第 13 章

JSP のアップグレード

このバージョンの JRun では、JSP バージョン 1.1 仕様が完全にサポートされています。この章では、旧バージョンの JSP 仕様にある JSP をアップグレードする方法について説明します。

目次

- 旧リリースからのアップグレード 182
- バージョン 1.1 PR1 からのアップグレード 182
- バージョン 1.1 PD1 からのアップグレード 183
- バージョン 1.0 からのアップグレード 184
- バージョン 0.92 からのアップグレード 185

旧リリースからのアップグレード

このセクションでは、旧バージョンの JSP 仕様にある JSP をアップグレードする方法について説明します。

このセクションで説明する内容は、Sun Microsystems 社の『JavaServer Pages Spececification Version 1.1 Public Release 2』に基づいています。この仕様は http://java.sun.com/products/jsp から入手可能です。

バージョン 1.1 PR1 からのアップグレード

このセクションでは、バージョン 1.1 PR1 から バージョン 1.1 PR2 へのアップグレードに伴う JSP 仕様の変更について説明します。

仕様への変更

- タグ ハンドラが JavaBeans コンポーネントになりました。また、属性は TLD で属性として明確にマークされたプロパティになりました。
- TLD の `attribute` の `type` サブエレメントは、対応する JavaBeans プロパティのタイプにより定義されるため、TLD から削除されました。
- JSP とサーブレットのリリース手段が J2EE とは別になったことを反映して、DTD 名が変更されました。変更後の名前は `web-jsptaglibrary_1_1.dtd` と `web-app_2_2.dtd` です。
- `Tag` 抽象クラスが 2 つのインターフェイスと 2 つのサポート クラスに分割されました。
- コンパイル済み JSP はサポート クラスとともにパッケージされるようになりました。

バージョン 1.1 PD1 からのアップグレード

このセクションでは、バージョン 1.1 PD1 から バージョン 1.1 PR1 へのアップグレードに伴う JSP 仕様の変更について説明します。

仕様への追加

- タグ ライブラリ記述子 (TLD) ファイルが追加されました。
- `jsp:include` および `jsp:forward` に新規パラメータが追加されました。
- `JspException` および `JspError` クラスが追加されました。
- 実行時スタックを提供するために、Tag クラスに親フィールドが追加されました。
- `PageContext` に `pushBody` メソッドと `popBody` メソッドが追加されました。
- プレコンパイル プロトコルが追加されました。
- すべての要求パラメータが `jsp*` の形式に変更されました。

仕様への変更

- `BodyEvaluation` クラスの名前が `BodyJspWriter` に変更され、`JspWriter` のサブクラスになりました。
- `Tag` は抽象クラスになりました。`TagSupport` は削除されました。`NodeData` は TagData に名前が変わりました。
- `doBody` メソッドは `doBeforeBody` と `doAfterBody` に分割されました。
- 本文が評価される回数に関係なく、アクションの呼び出し 1 回に対する `BodyJspWriter` は、最高でも 1 つだけになりました。
- `doStartTag` の戻りタイプは int になりました。
- Tag クラスに `initialize` メソッドと `release` メソッドが追加されました。

仕様からの削除

- `jsp:include` の `flush=false` オプションが削除されました。

バージョン 1.0 からのアップグレード

このセクションでは、バージョン 1.0 から バージョン 1.1 PD1 へのアップグレードに伴う JSP 仕様の変更について説明します。

仕様への追加

- JSP をサーブレットにコンパイルできるようになりました。このサーブレットは、ある JSP エンジンから別の JSP エンジンに移動できます。
- 移植可能タグ拡張メカニズムが追加されました。
- `flush` は `jsp:include` のオプション属性になりました。

仕様への変更

- `jsp:fallback` のコンテンツをクライアントに送信するだけで `jsp:plugin` を実装することはできなくなりました。

バージョン 0.92 からのアップグレード

バージョン 0.92 以降、JSP 仕様の大半は大幅に書き直されたか、変更されています。JRun 3.0 では JSP 1.0 仕様がサポートされています。このセクションでは、JRun 3.0 で実行できるように既存の JSP 0.92 ファイルをアップグレードするために必要な情報について説明します。

仕様への変更

JSP 0.92 仕様は次のように変更されました。

- SSI タグは `<%@ include` ディレクティブに置き換えられました。
- タグで大文字と小文字が区別されるようになりました。
- 標準タグは Java Platform の大文字と小文字の混在規則に従っています。
- `jsp:setProperty` と `jsp:getProperty` が定義されました。
- `<SCRIPT> </SCRIPT>` は `<%!... %>` に置き換えられました。
- 開始タグ、終了タグ、空白タグのように、タグは要素中の実際のタグを意味するだけになりました。用語要素とタグの使用方法は、HTML、XML、SGML などの使用方法と同じになりました。

仕様からの削除

次の項目は JSP 0.92 仕様から削除されました。

- 仕様で NCSA スタイルの SSI について明記されなくなりました。

仕様への追加

JSP 0.92 仕様には、次の項目が追加されました。

- `jsp:request` アクション要素が追加されました。
- 静的なリソースを実行時に含められるようにするため、`jsp:include` アクション要素が追加されました。
- `jsp:plugin` が定義されました。
- バッファリングができるようになりました。
- `page` ディレクティブで複数の属性が収集できるようになりました。`page` ディレクティブの `extends` 属性は、バージョン 0.92 で削除された機能に対応します。

第 14 章

サーバー側インクルードファイルの使用

この章では、拡張子 `.shtml` が付いているサーバー側インクルード (SSI) ファイルと、JRun でのこれらのファイルのサポートについて説明します。

目次

- サーバー側インクルードの使用 188
- Servlet タグ 188
- Include タグ 189

サーバー側インクルードの使用

サーバー側インクルードは、JRun が Web ページにダイナミック コンテンツを提供する 1 つの方法です。SSI はクライアント ブラウザ上ではなく Web サーバー上で使用し、処理するためのタグです。拡張子 .shtml が付いているファイルは SSI タグと HTML が混在しているテキスト ファイルです。

Web サーバーは JRun による SHTML ファイルのサポートを必要としませんが、JRun は SHTML ファイル内で使用できる 2 つの新しいタグのサポートを追加しています。

- <servlet> タグは SHTML ファイルからサーブレットを呼び出します。
- <include> タグは、ほかのファイルのコンテンツを SHTML ファイルにインクルードします。

Servlet タグ

<servlet> タグを使用してサーブレットを呼び出すための一般的な構文は次のとおりです。

```
<servlet name="aliasname" code="classname">
  <param name="paramname1" value="paramvalue1"/>
  <param name="paramname1" value="paramvalue1"/>
  <param name="paramname1" value="paramvalue1"/>
  ...
</servlet>
```

name 属性と code 属性は、いずれか一方だけを使用するようにしてください。<param> タグの使用は任意です。このタグによって、サーブレットに追加のパラメータを渡すことができます。これらのパラメータは通常のパラメータであり、初期化パラメータではありません。サーブレットの初期化パラメータは、JRun 管理コンソール (『JRun セットアップ ガイド』を参照) を使用してサーブレット用に設定した情報から取得されます。

JRun に添付されている SnoopServlet を使用して、<servlet> タグを使ってみましょう。SnoopServlet は、クライアントとサーバーの間の HTTP 接続に関する情報、および渡されたすべてのパラメータを表示します。

次の手順を実行します。

1 新しいテキスト ファイルを作成して、次のように入力します。

```
<servlet name="SnoopServlet">
  <param name="greeting" value="Hello World"/>
  <param name="anotherParam" value="aParamValue"/>
</servlet>
```

2 Web サーバーのドキュメント ルート ディレクトリ、たとえば c:¥inetpub¥wwwroot に、テキスト ファイルを greeting.shtml という名前で保存します。

3 Web ブラウザから greeting.shtml を要求します。URL は http:// *使用するホスト コンピュータ*/greeting.shtml のような形式になります。

greeting.shtml を変更するには、パラメータを変更、追加、または削除します。greeting.shtml への変更を保存した後、もう一度 Web ブラウザからこのページを開いて、実際に変更されていることを確認してください。

Include タグ

SHTML ファイル内で <include> タグを使用してほかのファイルのコンテンツをインクルードします。<include> タグの構文は次のとおりです。

```
<!--#include virtual|file=”filename”-->
```

ファイルのインクルードで使用するパスのタイプを指定するために、キーワード virtual または file を指定する必要があります。filename にはインクルードするパスまたはファイル名を入力します。

インクルード ファイルに特別のファイル名拡張子は必要ありません。

virtual キーワードの使用

virtual キーワードは、パスが「仮想」ディレクトリから始まることを示しています。たとえば、footer.inc という名前のファイルが /myapp という名前の仮想ディレクトリ内にあるとき、次の行によって footer.inc のコンテンツがこの行を含むファイルに挿入されます。

```
<!--#include virtual=”myapp/footer.inc”-->
```

file キーワードの使用

file キーワードは、パスが「相対」パスであることを示しています。相対パスは、インクルード先のファイルを含むディレクトリから始まります。たとえば、ディレクトリ myapp にファイルがあり、ファイル header1.inc が myapp/headers にある場合は、次の行によってファイルに header1.inc が挿入されます。

```
<!--# include file=”headers/header1.inc”-->
```

メモ
インクルードするファイルへのパス headers/header1.inc はインクルード先ファイルを起点とする相対パスです。この include ステートメントを含むスクリプトがディレクトリ /myapp にない場合、このステートメントは無効です。

第 15 章

プレゼンテーション テンプレート

この章では、プレゼンテーション テンプレートと JRun でのサポートについて説明します。

目次

- プレゼンテーション テンプレート (THTML ファイル) の使用 192
- default.template の使用 192
- default.definitions の使用 193
- ファイルの位置 195

プレゼンテーション テンプレート (THTML ファイル) の使用

JRun を実行すると、新しいタイプのサーバー側スクリプト ファイルのサポートが Web サーバーに追加されます。プレゼンテーション テンプレートを使用すると、一定の「外観と使い勝手」を HTML アプリケーションにシームレスに適用できます。拡張子 .thtml によってプレゼンテーション テンプレートを定義します。

THTML ファイルはヘッダと本文しかない HTML ベースのページです。THTML ファイルが要求されると、JRun の template サーブレットは、共通テンプレート ファイル内の特定のタグを、THTML ファイルの head タグと body タグに含まれるテキストに置き換えることによって THTML ファイルを処理します。

テンプレート ファイルの名前は default.template です。default.template ファイルには、すべての THTMLファイルに対する共通の外観と使い勝手が定義されています。

default.template ファイルのほかに、オプションの default.definitions ファイルを使用して、特殊な代入タグに使用する名前を定義することもできます。

default.template の使用

default.template ファイルには、HTML と特殊な代入タグを混合した <subst> が含まれています。<subst> タグは、このタグが示す名前に割り当てられたテキストによって置き換えられるマーカです。たとえば、THTML ファイル greeting.thtml に、次のような <head> タグと <body> タグが含まれているとします。

```
<head>
<title>Greetings from greeting.thtml</title>
</head>
<body>
Hello World from greeting.thtml
</body>
```

さらに、この例の default.template ファイルには、次のタグが含まれているとします。

```
<html>
<head>
<subst data="head"/>
</head >
<body>
A greeting from the requested documents:<subst data=”body”/> <br>
</body>
</html>
```

この場合、JRun の template サーブレットは、次のようなドキュメントを生成します。

```
<html>
<head>
<title>Greetings from greeting.thtml</title>
</head >
```

```
<body>
A greeting from the requested documents:Hello World from greeting.thtml
<br>
</body>
</html>
```

ここで示されているように、default.template ファイルで定義されている <subst data=”head”> および <subst data=”body”> タグは、THTML ファイルの <head> および <body> タグの内容にそれぞれ 置き換えられています。

<subst> タグには、次のいずれかの形式を使用できます。

```
<subst data="[name]"/>
<subst data="[name]"></subst>
```

[name] には代入するデータの名前を指定します。指定した name の値が、作成されたクライアントに対する出力の中で、対応する<subst> タグと置き換わります。

代入は、<head> タグと <body> タグだけに制限されているわけではありません。名前が付いているその他の代入についても、default.definitions ファイルで指定されているように default.template ファイルで定義して使用できます。

メモ
THTML ファイルを正しく機能させるには、default.template ファイルがアクセス可能でなければなりません。THTML ファイルに適用する default.template ファイルのアクセスに関するルールは、195 ページの「ファイルの位置」で説明しています。

---

default.definitions の使用

default.definitions ファイルは、名前 / 値ペアが保持されたプロパティファイルです。このファイルを使用して、<subst> タグの代入に使用される名前を定義できます。次の例は、default.definitions ファイルに名前 / 値ペアを割り当てる方法を示します。

```
MyThought=<P>Missing sweet San Diego</P>
TodaysSaying=<B>Anger is a gift…(Rage Against The Machine)</B><BR>
MyLineBreaks=<PRE>A line break follows<BR>¥nHeres a new line</PRE><BR>
```

それぞれの名前 / 値ペアは、円記号「¥」を使用して新しい行に進まない限り、同じ行に存在することになります (値が 1 行を越える場合は、円記号を使用して新しい行に進みます)。値自体に改行を含める場合は、3 つめの名前 / 値ペアに示すように、「¥n」を使用して指定できます。名前 / 値ペアの指定に関するルールは、java.util.Properties クラスの定義に適用されるルールと同じです。

default.properties ファイルに名前 / 値ペアを設定したら、default.template ファイルの <subst> タグ内でそれらのペアを使用できるようになります。次に例を示します。

```
<html>
<head>
<subst data="head"/>
</head >
<body>
A greeting from the requested documents:<subst data="body"/><br>
Here's a thought:<subst data="MyThought"/>
Here's a saying:<subst data="TodaysSaying"/>
Line Breaks:<subst data="MyLineBreaks"/>
</body>
</html>
```

greeting.thtml が要求されたときに、上記の default.template ファイルと default.definitions ファイルが使用可能な場合は、次の結果ドキュメントがサーバーで生成されます。

```
<html>
<head>
<title>Greetings from greeting.thtml</title>
</head >
<body>
A greeting from the requested documents:Hello World from greeting.thtml
        <br>
Here's a thought:<P>Missing sweet San Diego</P>
Here's a saying:<B>Anger is a gift…(Rage Against The Machine)</B><BR>
Line Breaks:<PRE>A line break follows<BR>
Here's a new line</PRE><BR>
</body>
</html>
```

default.template ファイルに、使用可能な名前 / 値ペアのない <subst> タグが含まれている場合、この <subst> タグは結果ページで空白の文字列に置き換わります。default.definitions ファイルは THTML ファイルの要求時には必要ありません。

ファイルの位置によっては、1 つの default.template ファイルに対して複数の default.definitions ファイルが存在する場合があります。詳細については、次のセクションを参照してください。

ファイルの位置

default.definitions ファイルで定義されている名前 / 値ペアは、同じディレクトリとそのサブディレクトリ内のすべての THTML ファイルに適用されます。ディレクトリ階層に複数の default.definitions ファイルが存在する場合、THTML ファイルには上位階層に存在する最も近い default.definitions ファイルが使用されます。

default.definitions ファイルの名前 / 値ペアは、THTML ファイルの位置を基準に追加されていきます。したがって、名前 / 値ペアを現在の名前 / 値ペア セットに追加しても、追加したペアは同じディレクトリとそのサブディレクトリ内の THTML ファイルにしか適用できません。

ある default.definitions ファイルに定義された名前 / 値ペアは、サブディレクトリの default.definitions ファイルに定義された名前 / 値ペアによって上書きされる可能性もあります。この名前 / 値ペアの上書きは、内容を上書きした default.definitions ファイルと 同じ ディレクトリ および サブディレクトリ に存在する THTML ファイルだけに反映されます。たとえば、c:/foo/ の default.definitions ファイルで drinks=cola と 定義し 、c:/foo/bar/ の default.definitions ファイルで drinks=milk と 定義した場合、drinks=milk を認識するのは、c:/foo/bar/ とそのサブディレクトリ内の THTML ファイルだけになり ます。c:/foo/ 内の THTML ファイルは、drinks=cola を認識します。

default.template ファイルの使用可能なスコープは、同じディレクトリとそのサブディレクトリに存在するすべての THTML ファイルに適用されます。ディレクトリ階層に複数の default.template ファイルが存在する 場合、THTML ファイルは、その上位階層の最も近い default.template ファイルとともに使用されます。

たとえば、次の 3 つのファイルが存在するとします。

```
c:/foo/default.template
c:/foo/bar/doo/default.template
c:/foo/bar/a.thtml
```

この場合、Template サーブレットは、c:/foo/default.template と c:/foo/bar/a.thtml を使用して、クライアントに対する結果ページを生成します。

第 16 章

タグレット

タグレットは、プレゼンテーションおよびアプリケーション ロジック間にもう 1 つの分類レベルを追加します。タグレットを使用すると、HTML プログラマはプレゼンテーション用のタグベースのドキュメントに専念でき、Java プログラマはアプリケーション ロジックに専念できます。

この章では、タグレットの導入について簡単に説明します。また、カスタム タグレット開発の概略についても説明します。

目次

- タグレットとは 198
- SSI タグレットのロードと使用 199

タグレットとは

タグレット は、SHTMLファイルに使用できる独自のタグをアプリ ケーション開発者が柔軟に定義し、実装できるようにする、ユーザ定義のサーバー側のタグです。この章では、サーブレットを呼び出すための 1 対 1 のマッピングを提供する SSI タグレットについて説明します。

SSI タグレット

SSI タグレットは、タグレットとサーブレットの対応を示す 1 対 1 の命名規則を定義することによって、タグを使用してサーブレットを呼び出す手段を提供します。SSI タグレットは、`local.properties` ファイルに定義されているマッピングに基づき、`SSIFilter` によってロードされます。`local.properties` ファイルについては、次のセクションで説明します。

SSI タグレットのロードと使用

local.properties ファイルは、SSIFilter によってデータの事後処理に使用されます。local.properties ファイルは手動でも編集できますが、JRun 管理コンソールには、これらのプロパティの設定に使用できるグラフィカル インターフェイスが用意されています (JRun 管理コンソールの詳細については『 JRun セットアップ ガイド 』を参照)。

local.properties ファイルは、SSI タグレット と サーブレット 間のマッピングを示す、名前 / 値ペアを定義する際に使用します。このファイルは、*JRun のホーム ディレクトリ*/servers/ *サーバー名*というディレクトリに格納されています。*サーバー名*は JRun のサーバー名に対応します。一般的な構文は次のとおりです。

```
ssifilter.<tagname>.dynamictaglet=<servlet class name|servlet alias>
```

tagname には、サーブレットの呼び出しに使用するタグの名前を設定します。servlet class name にサーブレットの完全修飾のクラス名を設定するか、または servlet alias にサーブレットのエイリアスを設定します (Web サーバーの Servlet Engine > Aliases コントロールの JRun 管理コンソールで定義されているとおりに設定します)。tagname の値には、servlet class name または servlet alias のいずれかを使用できます。

たとえば、次の例は、SHTML ファイル内のサーブレット SnoopServlet を呼び出す foo タグを定義しています。

```
ssifilter.foo.DynamicTaglet = SnoopServlet
```

この例では、SnoopServlet がサーブレット のクラス名になります。このマッピングを定義することにより、Web ドキュメントの作成者は次のようにドキュメント内に SnoopServlet を呼び出すことができます。

```
<foo></foo>
```

SSI タグレットでは、次のように サーブレットにパラメータを渡すことができます。

```
<foo name1=value1 name2=value2 name3=value3>
Hello World
</foo>
```

これにより、サーブレット ライターは、サーブレット内の request オブジェクトの getParameter() メソッドを使用して、名前 / 値ペアを取得できるようになります。たとえば、パラメータ name1 の値を取得するには、サーブレットで次のメソッドを実行します。

```
request.getParameter(“name1”)
```

SSI タグレットの本体 (上の例の場合は Hello World) は、次のメソッドで取得できます。

```
request.getAttribute(“taglet.body.foo”)
```

タグレットの名前は、taglet.body.key 文字列の 3 番目のキーです。タグレットの名前が DATETAG のときは、getAttribute("taglet.body.datetag") を使用します。一般に、このメソッド は、先頭の <foo> タグと 末尾の </foo> タグの間にあるすべてのテキストを返します。属性名には小文字を使用することに注意してください。

第 3 部

サーブレットの開発

ここでは、Java サーブレット API を使用してサーブレットを作成する方法について説明します。

Java サーブレットの処理 ... 203
サーブレットのチュートリアル ... 213
サーブレット API の基本情報 ... 221
Java サーブレット API によるプログラミング ... 227
サーブレットの例 ... 237
カスタム タグとタグ ライブラリの作成 ... 255
サーブレット API の変更点 ... 279

第 17 章

Java サーブレット の処理

この章では、サーブレットの基本的な概念について説明します。クラス、同期化、Web アプリケーションについても説明します。

目次

- サーブレットについて ... 204
- Java サーブレット API バージョン 2.2 ... 204
- 基本サーブレット クラスおよびインターフェイス ... 205
- サーブレットのライフサイクル ... 206
- 同期化 ... 208
- サーブレットのチェーン化 ... 210
- Web アプリケーション ... 212

サーブレットについて

サーブレットとは、JRun が Web サーバーとともに実行するサーバー側の Java プログラムです。

JRun で開発されたサーブレットの次のような特長によって、強力な Web ベースのサーバー側アプリケーションの開発が可能になります。

- サーブレットは Java で書かれており、Java プログラミング言語のすべての機能を利用できます。
- サーブレットは CGI よりも効率的なメモリ管理を可能にします。
- サーブレットはコンパイルされているため、非常に高速に実行できます。また、JRun はサーブレット のインスタンスをキャッシュに格納することによってロード時間を最小限にします。
- アプレット ではクライアント 側の Java Virtual Machine (JVM) を制御できませんが、サーブレットでは JRun が実行する JVM を完全に制御できます。
- JRun は Java サーブレット API バージョン 2.2 の仕様書に準拠しています。

サーブレットには、次のような使用法があります。

- HTML ページの作成。オプションとして、外部のデータ ソースからデータにアクセスし、フォーマットし、返すこともできます。
- サーバー側の include を使用して HTML ページにサーブレットを組み込みます。
- サーブレットをチェーン化して、各サーブレットが先行のサーブレットの出力を処理できるようにします。

このほかに、JRun は動的に JSP を Java サーブレットに変換します。

Java サーブレット API バージョン 2.2

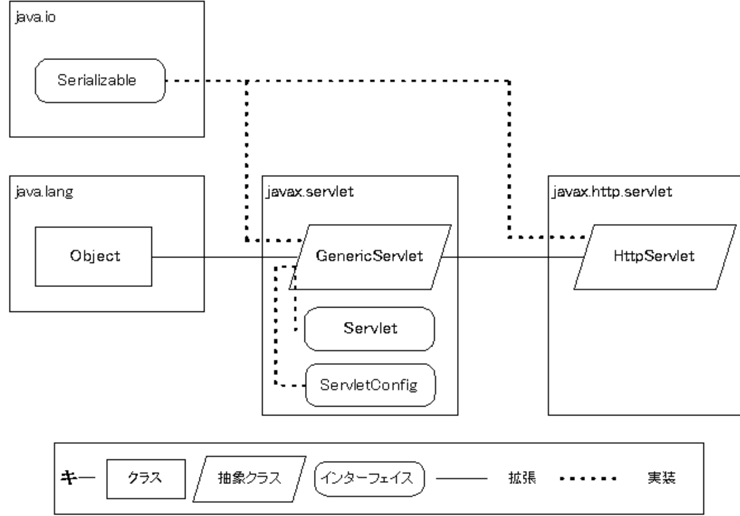
JRun バージョン 3.0 は、Java サーブレット API バージョン 2.2 を完全にサポートします。バージョン 2.2 には次のような特長があります。

- 新しいメソッド
- 認証と承認のサポート
- 要求の送信の機能強化
- 応答バッファ
- Web アプリケーション

サーブレット API の詳細については、http://java.sun.com/products/servlet/ を参照してください。

基本サーブレット クラスおよびインターフェイス

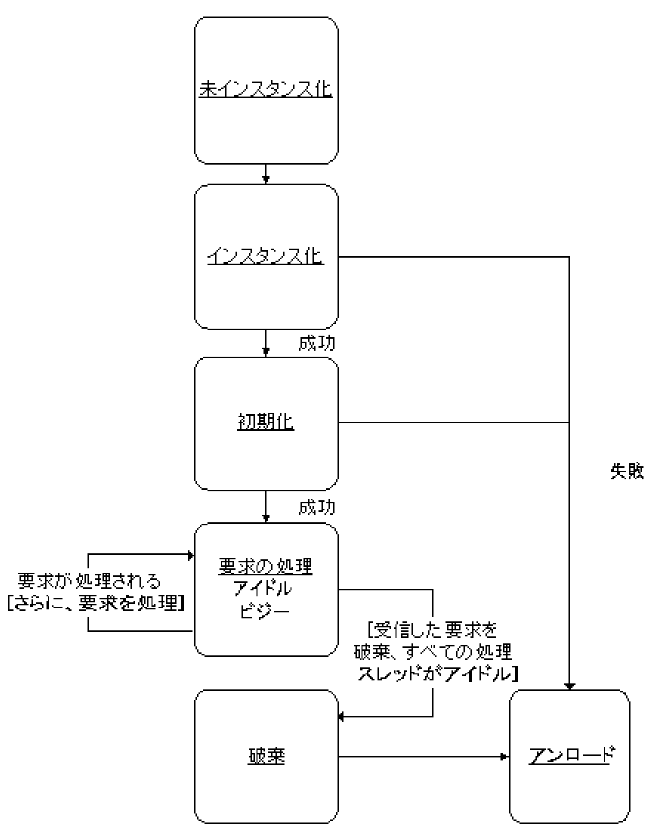
次の図のように、サーブレット API は java.io、java.lang、javax.servlet、および javax.http.servlet パッケージからのクラスおよびインターフェイスを使用します。



サーブレット API のクラス、インターフェイス、および例外の詳細については、221 ページの第 19 章「サーブレット API の基本情報」を参照してください。

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には、次の図のようにいくつかの段階があります。

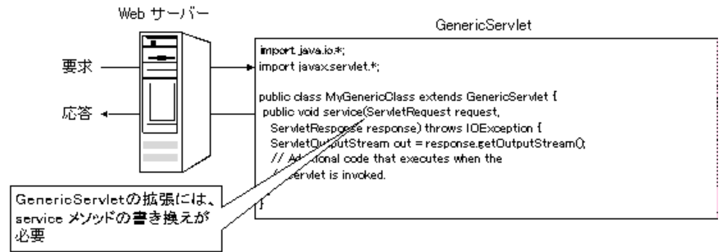


初期化段階で、JRun はサーブレットの `init` メソッドを使用します。`init` メソッドは一度だけ呼び出されます。したがって、サーブレットを呼び出すたびに貴重なリソースを割り当てる必要はありません。`init` メソッドは、サーブレットがそのライフサイクル中に必要とするリソースを割り当てるために使用します。JRun は、起動時 (JMC 内で有効にされている場合) または最初のクライアント要求を受け取ったときに `init` メソッドを呼び出します。

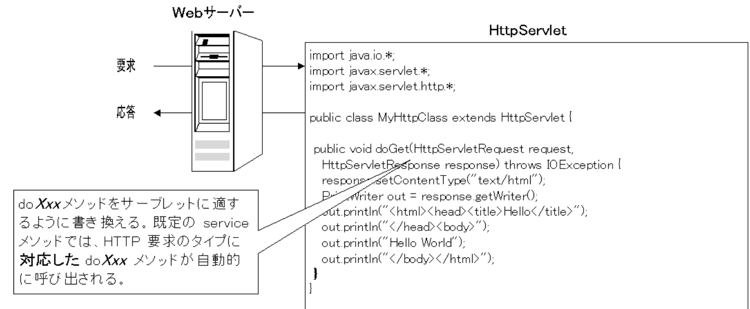
サーブレット クラス内で `init` メソッドを書き換える必要はありません。初期化時に何らかのタイプの処理を実行する場合にのみ、このメソッドを書き換えます。

サーブレットが受信したすべての要求を処理したとき、**要求処理**段階に入ります。JRun は、サーブレットがどのクラスを展開したかによって、要求を `service` メソッド、または **do*Xxx*** メソッドへ送ります。

- GenericServlet　サーブレット が `GenericServlet` を展開した場合、処理は `service` メソッド を書き 換えることによって実行されます。



- HttpServlet　サーブレットが `HttpServlet` を展開した場合、処理は想定している HTTP 要求タイプに対応する **do*Xxx*** メソッドを書き換えることによって実行されます。たとえば、サーブレットで HTTP GET 要求を処理する **doGet** メソッドをコーディングします。



これらのクラスの詳細については、227 ページの第 20 章「Java サーブレット API によるプログラミング」を参照してください。

メモ

JRun は、各サーブレットのインスタンスをそのクラス名ではなく、**エイリアス**によって管理します。サーブレット のエイリ アスを定義していない場合、JRun は動的にエイリアスを作成します。エイリアスを登録名または基準名とも言います。

JRun サーバーをシャットダウンしたとき、または JRun サーバーが変更されたサーブレットを再ロードしたとき、JRun はサーブレットをメモリからアンロードします。アンロード時に、サーブレットは**破棄**段階に入ります。破棄要求の開始時に、JRun は `destroy` メソッド を呼び出します。`destroy` メソッド 内で、サーバーが使用していたリソースの割り当てを解放できます。`destroy` メソッドが呼び出されると、クラスをガーベッジ コレクションに入れることができます。

同期化

通常、サーブレット開発者はサーブレット インスタンスとの関連で同期化を認識していますが、実際の作業を実行するのは**スレッド**です。多くのスレッドが同時に実行されることもあります。

各インスタンスは複数のスレッドで実行される可能性があるため、オブジェクトスコープ変数やその他の共有リソースでの同期化の問題を認識する必要があります。同期化とは、1 つのコードをシングルスレッドとして実行させることです。スレッド管理は、JMC を使用して、各 JVM に別々のスレッド パラメータを指定することによって制御します。

メモ
スレッド管理は高度な操作であるため、本書では扱いません。詳細については、Java スレッド管理に関する市販の解説書を参照してください。

クラススコープ インスタンス変数の同時アクセスを防止するには多くの方法があります。以下はその例です。

- キーワード synchronized を doXxx メソッドのシグネチャに含める (この方法はお勧めしません)
- オブジェクトスコープ変数を更新する行を同期化する
- SingleThreadModel インターフェイスを実装する
- オブジェクトスコープ変数にアクセスするメソッドを同期化する

次のサブセクションでこれらのテクニックについて説明します。

メソッド シグネチャでの synchronized キーワードの使用

メソッド シグネチャで synchronized キーワードを使用することで、メソッド全体へのアクセスを同期化できます。

```
...
public class TestCaller extends HttpServlet
{
  int visitorCounter = 0;

  // コンセプトの説明のみを目的とした例なので、実際には実行しないでください。
public synchronized void doGet(HttpServletRequest servReq,
  HttpServletResponse servRes) throws IOException, ServletException {
...
```

メモ
この例は、コンセプトについて説明するためのものです。この同期化テクニックはパフォーマンスを低下させるため、実際にはほとんど使用しません。

同期化されたコードの使用

synchronizedされたブロックを使用して、オブジェクト スコープ変数へのアクセスを同期化することができます。

```
...
int thisCount;
synchronized(this) {
  // visitorCount はオブジェクトスコープ変数です。
  thisCount = visitorCounter++;
}
out.println("<p>You are visitor number " + thisCount);
...
```

この例では、同期化されたブロック内のカウンタを増分し、それによってオブジェクトスコープ変数へのシングルスレッド アクセスを保証します。

SingleThreadModel インターフェイスの使用

SingleThreadModel インターフェイスは JRun に一群のサーブレット インスタンスを作成して、それぞれのインスタンスについて同時スレッドが service メソッドを実行しないように指示します。

```
...
public class testSync extends HttpServlet
 implements SingleThreadModel {
...
```

JRun は SingleThreadModel を実装するサーブレットの複数のインスタンスを作成するため、オブジェクトスコープ インスタンス変数がすべてのインスタンスについて同じでなければならない場合、このテクニックは使用できません。たとえば、このテクニックをヒット カウンタには使用できません。しかし、オブジェクトスコープインスタンス変数を使用しないサーブレットの場合、またはオブジェクトスコープ インスタンス変数が異なってもよい場合 (例 : バッファ変数、データベース接続) には、SingleThreadModel は効果的なテクニックです。

オブジェクトスコープ変数にアクセスするメソッドの同期化

オブジェクトスコープ変数へのすべてのアクセスが同期化されたメソッドによって行われるようなアクセス方式を実装できます。

```
...
public class TestSync extends HttpServlet {
  int visitorCounter = 0;

public void doGet(HttpServletRequest req, HttpServletResponse res)
  throws IOException, ServletException {
  incrementCount();
...
public synchronized void incrementCount() {
 visitorCount++;
```

```
}
public synchronized int getCount() {
 return visitorCount;
}
...
```

サーブレットのチェーン化

JRun では、特定の複数のサーブレットが常に同じ順序で実行されるように指定できます。先行のサーブレット の出力が後続のサーブレット の入力として渡されます。このテクニックによって、サーブレットを必要な数だけチェーン化できます。たとえば、1 つのサーブレットから一連のデータにアクセスし、それを後続のサーブレットに渡して、そのサーブレットでデータを HTML テーブルの行と列にフォーマットできます。

JRun では、次のいずれかの方法でサーブレットのチェーン化を実装できます。

- Explicit　サーブレットとその順序を明示的に指定します。
- MIME タイプ　後の各サーブレットに MIME タイプを関連付けます。JRun は、先行のサーブレット が関連付けられている MIME を返したときに、後続のサーブレットを自動的に実行します。

サーブレット チェーンの明示的確立

JRun では、サーブレットを明示的な順序でチェーン化できます。

サーブレットを明示的にチェーン化するには

1 先行のサーブレットと後続のサーブレットをコーディングします。
2 JMC を起動します。
3 適切な JRun サーバーを展開します。
4 Web アプリケーション項目を展開します。
5 サーブレットのチェーン化を使用するサーブレットを含んでいる Web アプリケーションをクリックします。
6 [サーブレット定義] パネルを表示します。
7 [編集] をクリックします。
8 チェーン化するサーブレットを定義します。
9 [更新] をクリックします。
10 [サーブレット URL のマッピング] パネルを開きます。
11 [編集] をクリックします。
12 [仮想パス / 拡張子] フィールドで、クライアントが URL で入力した名前を指定して、チェーン内の最初のサーブレットを呼び出します。

13 [呼び出されたサーブレット] フィールド で、チェーンに含めるサーブレット をカンマ区切りリスト を使用して指定します。[サーブレット 定義] パネルで指定したサーブレット名を使用します。

14 [更新] をクリックします。

JMC の使用の詳細については、『JRun セットアップ ガイド』を参照してください。

MIME タイプによるサーブレット チェーン化の確立

JRun では MIME タイプによってサーブレットのチェーン化を実装できます。MIME タイプのチェーン化を使用する場合は、JRun が指定した MIME タイプを使用する応答を検出したときに実行するサーブレットを指定します。

MIME タイプのサーブレットのチェーン化を有効にするには

1 特定の MIME タイプの入力を処理するサーブレットをコーディングします。

2 JMC を起動します。

3 適切な JRun サーバーを展開します。

4 Web アプリケーション項目を展開します。

5 MIME によるサーブレットのチェーン化を有効にする Web アプリケーションをクリックします。

6 [MIME チェーン化] をクリックします。

7 [編集] をクリックします。

[MIME タイプ設定] ウィンドウが表示されます。

8 MIME タイプを指定します。

9 JRun が関連付けられている MIME タイプを使用する応答を検出したときに実行するサーブレットを指定します。

10 [更新] をクリックします。

JMC の使用の詳細については、『JRun セットアップ ガイド』を参照してください。

Web アプリケーション

Web アプリケーションとは、Java サーブレット、JSP、JSP タグ ライブラリ、HTML ページなどのスタティック コンテンツ、およびその他のサーブレットや JSP によって要求されるリソースのコレクションです。Web アプリケーションはあらかじめ定義されたディレクトリ構造に公開されます。詳細については、15 ページの第 2 章「JRun プログラミング モデル」を参照してください。

Web アプリケーション内のサーブレットと JSP は、1 つの ServletContext オブジェクトを共有します。ほかのアプリケーションと直接的にデータを共有することはできませんが、ほかのアプリケーションの ServletContext オブジェクトにアクセスすることによって間接的にデータを共有することはできます。また、同じアプリケーション内のサーブレットは、共通のデータベースをデータ レポジトリとして使用することによってデータを共有できます。

1 つの JRun サーバーで複数の Web アプリケーションをサポートできます。Web アプリケーションを開発するときに検討しなければならない問題の 1 つは、アプリケーションの間の境界をどこに設定するかです。言い換えれば、アプリケーションをほかのアプリケーションと同じ JRun サーバー内に置くことができるか、別の JRun サーバー内に置かなければならないかという問題です。

それを決定する 1 つの要因はエラー処理です。同じ JRun サーバー内のすべての Web アプリケーションは同じプロセス内で実行されるため、1 つのアプリケーションのエラーによって JRun サーバー全体が停止してしまいます。このエラー処理の影響が受け入れられない場合は、アプリケーションを別の JRun サーバーに入れる必要があります。

また、それぞれの JRun サーバーに、特定のアプリケーションによって要求されるサーバー固有の属性を割り当てることができます。たとえば、各 JRun サーバーは異なる JVM を使用できるため、JVM のタイプによってアプリケーションを分離できます。

最後に、アプリケーションのセキュリティは JRun サーバー レベルで決定されます。2 つのアプリケーションが異なるセキュリティ管理システムを要求する場合は、それらのアプリケーションを別の JRun サーバーに配置する必要があります。

Web アプリケーションの詳細については、第 5 章 を参照してください。

第 18 章

サーブレット のチュートリアル

プログラミング言語や API の基礎を学ぶための最も効果的な方法は、簡単なプログラムを開発してみること です。このチュートリ アルでは、`HelloWorld` という サーブレットを開発します。Java サーブレット API に関する学習の進行状況に応じてこのプログラムを更新していきます。最終的には Java サーブレット API のすべての基本的な要素を組み込んだ、簡単に変更できる大規模なプログラムが完成します。このチュートリアルが終わるまでには、API について、また、サーブレット を開発するために必要なツールについての基本的な知識が習得できます。

メモ
このチュートリアルでは、Java を使用してサーブレットを作成する方法について説明します。JSP を使用してサーブレットを作成する方法については、第 7 章を参照してください。

目次

- 第 1 部 214
- 第 2 部 216
- 第 3 部 218

第 1 部

チュートリアルを開始するにあたって、まず次のコードを調べてみます。

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

public class HelloWorld extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title>Hello</title></head><body>");
  out.println("Hello World");
  out.println("</body></html>");
}
}
```

次のことに注意してください。

- 最初の数行は、サーブレットに必要なすべてのパッケージをインポートします。`import javax.servlet.*` と `import javax.servlet.http.*` の行はこれまでに見たことがないかもしれません。これらの項目は Java サーブレット API のパッケージです。これらのパッケージをインポートするのは、すべてのサーブレットがサーブレットインターフェイスを実装する必要があるためです。詳細については、第 19 章「サーブレット API のパッケージ」を参照してください。
- `HelloWorld` プログラムは `HttpServlet` クラスを継承します。HTTP を使用して Web ブラウザと通信するサーブレットのためにこのクラスを継承します。`HttpServlet` は `GenericServlet` クラスを継承しますが、このクラスはあまり使用しません。`GenericServlet` と `HttpServlet` の詳細については、227 ページの第 20 章「Java サーブレット API によるプログラミング」を参照してください。
- `doGet()` メソッドの最初のパラメータは `HttpServletRequest` オブジェクトです。このオブジェクトは、サーブレットがクライアント要求を読み込むことができるように入力ストリームを提供します。このサービスメソッドの 2 番目のパラメータは、`HttpServletResponse` オブジェクトです。このオブジェクトは、サーブレットがクライアントに返す応答を作成できるように出力ストリームを供給します。詳細については、41 ページの「HTTP 要求と応答」を参照してください。
- 出力に HTML タグが含まれるため、`setContentType()` メソッドはサーブレットの出力の MIME タイプを `text/html` に設定します。既定では、Java サーブレットの出力 MIME タイプは `text/html` です。
- `getWriter()` メソッドは `PrintWriter` オブジェクトを返します。このオブジェクトは、サーブレットがブラウザにテキストを返すことができるようにします。
- `out.println()` 行では、HTML 形式を使用することと、文字列「`Hello World`」を含むテキストをブラウザに送信することを指定します。

これから、HelloWorld.java という名前の Java ソース ファイルを作成して、上のコードをこのファイルにコピーします。次に、サーブレット クラス ファイルを作成するために Java コードをコンパイルします。

次のコマンドを使用して HelloWorld.java をコンパイルします。

```
javac -classpath c:¥jruninstalldirectory¥lib¥ext¥servlet.jar
        HelloWorld.java
```

このコマンドは Java サーブレット API からのパッケージをインクルードするための classpath を指定します。この例では、これらのファイルが Microsoft Windows システムの既定の位置にインストールされていることを想定しています。

コードのコンパイルが完了したら、サーバーが HelloWorld.class ファイルにアクセスできるように、このファイルを適切なディレクトリにコピーします。既定ではこのディレクトリは *JRun のインストール ディレクトリ*/servers/default/default-app/WEB-INF/classes です。異なるサーバーを使用する場合は、ファイルを適当なディレクトリにコピーしてください。

サーバー名が *myhost* であれば、次の URL を使用してこのサーブレットを呼び出します。

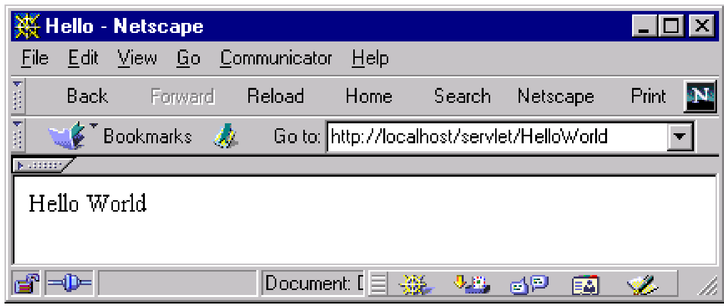
http://localhost/servlet/HelloWorld

JRun URL とファイルの場所の詳細については、70 ページの「Web アプリケーションの開発」を参照してください。

メモ

サーバーがポート 80 以外のポートで動作している場合、URL にそのポート番号を含める必要があります。

次の図はサーブレットの出力を示しています。



第 2 部

このチュートリアルの「第 1 部」の例では、簡単な文字列の値を出力しました。しかし、実際のサーブレットは通常、パラメータを通じて入力を受け入れます。

サーブレットは 2 つのタイプのパラメータを使用します。**要求パラメータ** と**初期化パラメータ** です。このセクションでは要求パラメータについて説明し、「第 3 部」で初期化パラメータについて説明します。

要求パラメータ (単にパラメータと呼ばれることもある) を使用して、サンプル プログラムで単に「Hello World」と出力するのでなく、「Hello <your name>」と出力できます。

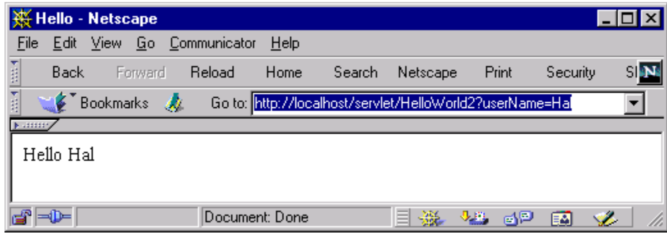
次のコード 例は、getParameter メソッドを使用してパラメータの値を取得する 方法を示しています。取得しようとしているパラメータの名前が定義されていない場合、getParameter メソッドはヌル値を返します。

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

public class HelloWorld2 extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html");
  String userName;
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title>Hello</title></head><body>");
  if ( (userName = req.getParameter("userName")) != null)
    out.println("Hello " + userName);
  else out.println("Hello, who are you?");
  out.println("</body></html>");
}
}
```

ここまではパラメータを取得する方法について説明しました。次は、パラメータを指定する方法について説明します。パラメータを指定する最も基本的な方法は、要求 URL でクエリ文字列を使用する方法です。クエリ文字列を使用して、次のようにパラメータを指定できます。

http://localhost/servlet/HelloWorld2?userName=Hal

次の図はサーブレットの出力を示しています。



パラメータを指定するもう 1 つの方法は、SHTML ファイル内で `<servlet>` タグを使用する方法です。

```
<servlet code="helloWorld2" >
  <param name="userName" value="Hal">
</servlet>
```

このほかに、HTML フォームなどを使用してパラメータを指定できます。HTML フォームは GET または POST メソッドを使用して Web サーバーにデータを送信できます。GET と POST の関係、およびサーブレットでコーディングする方法の詳細については、227 ページの第 20 章「Java サーブレット API によるプログラミング」を参照してください。

第 3 部

これまで、要求パラメータを取得して指定する方法について説明しました。次に、初期化パラメータを使用する例について取り上げます。このセクションでは、初期化パラメータを使用する方法の 1 つと、サーブレットでそれを取得する方法について説明します。

特定の Web ページのアクセス回数を記録するカウンタが必要であると想定します。また、カウンタの値をほかのサーブレット出力と異なるフォント サイズで表示すると想定します。これらのタスクを実行するために、クラスの中に counter と fontSize という 2 つの変数が必要となります。

次のコード例は、初期化パラメータを使用して counter および fontSize を保持する方法を示しています。

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

public class UseInitParms extends HttpServlet {
 //既定値
 private static final int FONTSIZE = 3;
 private static final int COUNTER = 0;

 //既定の変数
 public static int fontSize;
 public static int counter;

  // 引数を持たない init メソッドをオーバロードするのが最適です。
public void init () throws ServletException {
  if ( getInitParameter("fontSize") != null ){
    try{
      this.fontSize =
      Integer.parseInt(getInitParameter("fontSize"));
    }
    catch (NumberFormatException e){
      this.fontSize = this.FONTSIZE;
    }
  }else this.fontSize = this.FONTSIZE;
  if ( getInitParameter("counter") != null ){
    try{
      this.counter =
      Integer.parseInt(getInitParameter("counter"));
    }
    catch (NumberFormatException e){
      this.counter = this.counter;
    }
  }else this.counter = this.counter;
}

public void doGet(HttpServletRequest req, HttpServletResponse resp)
          throws IOException, ServletException {
```

```
  // コンテンツのタイプを設定します。
  resp.setContentType("text/html");

  String userName;
  PrintWriter out = resp.getWriter();
  if (req.getParameter("userName") != null)
    userName = req.getParameter("userName");
  else userName = new String("Stranger");

  out.println("<html><head>Hello " + userName + "</head><body><br>");
  out.println("This page has been accessed <font size=+" + fontSize
        +">" + counter++ + "</font> times<br>");
  out.println("</body></html>");
}
}
```

counter と fontSize はどちらも、クラス定義で既定値が定義されています。しかし、公開担当者とシステム管理者はオプションとして初期化パラメータを使用し、これらの既定値を書き換えることができます。この柔軟性があるため、サイトでは任意の値からカウントを開始でき、また、カウンタ値の表示のために任意のフォント サイズを指定できます。

JMC を使用してサーブレットの初期化パラメータを指定します。JMC の使用の詳細については、『JRun セットアップ ガイド』を参照してください。

このチュートリアルは、Java の基礎について理解していることを前提としているため、定数や変数の宣言については説明しません。

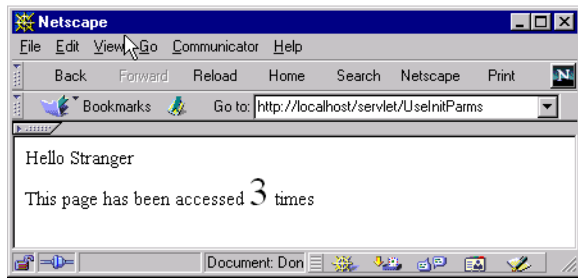
最初のメソッドは init メソッドです。init メソッドは、getInitParameter メソッドを使用して初期化パラメータを取得するため、またグローバル変数を初期化するために使用します。この例では init メソッドを引数なしで使用します (init(ServletConfig config) バージョンを使用する場合、init メソッドの最初の行は super.init(config) の呼び出しである必要があります)。

JRun サーバーが再起動されるまで、またはサーブレットが動的に再ロードされるまで、サーブレットはメモリに保持されます。また、init メソッドはサーブレットがロードされるときに一度だけ呼び出されるため、グローバル変数は 1 回だけ初期化されます。グローバル変数は初期化された後、サーブレットのライフタイムが終わるまで利用可能で、異なるクライアント要求の間で同じステートを保ちます。

したがって、init メソッドのコードによって、公開担当者またはシステム管理者が JMC を使用して有効な初期化パラメータを設定した場合、グローバル変数はそれらの値に設定されます。初期化パラメータが設定されなかった場合、または誤った初期化パラメータが設定された場合は、グローバル変数は既定値 (定数) に設定されます。

doGet メソッドはその HTML 出力内で fontSize および counter 変数を使用します。

次の図はサーブレットの出力を示しています。



これで初期化パラメータを取得する方法は終了です。基本的なサーブレットであれば、自分で作成できるはずです。

第 19 章

サーブレット API の基本情報

この章では、サーブレットの作成およびサーブレット API の基本情報について説明します。

目次

- Java サーブレットのタイプ 222
- サーブレット API のパッケージ 222
- サーブレット API に関するリファレンス情報 225

Java サーブレットのタイプ

サーブレットを検討するときは、`javax.servlet.Servlet` インターフェイスを実装するクラスである必要があります。サーブレット API には、このインターフェイスを実装するクラスと、Java サーブレットを作成するときに拡張するクラスの 2 つが含まれています。

- `GenericServlet` には、プロトコルに依存しない基本サーブレット機能があります。このクラスを拡張して、HTTP 以外のサービスをコーディングしてください。しかし、通常はクラスを `HttpServlet` から拡張します。
- `HttpServlet` は `GenericServlet` を拡張して HTTP 固有の機能を追加します。ほとんどのクラスが `HttpServlet` を拡張します。この章では、クラスで HTTP 処理を実行することを想定します。

サーブレット API のパッケージ

サーブレット API は、Sun Microsystems の Java Software Division によって公開された仕様です。この仕様には、Java サーブレット、および Java サーブレットをサポートするサーバーによって要求されるクラス、メソッド、および動作の概要が記載されています。JRun Version 3.0 は、Java サーブレット API バージョン 2.2 の仕様をサポートしています。

サーブレット API には、次のパッケージが含まれています。

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

メモ
これらのパッケージに関する JavaDoc 形式のドキュメントが JRun の *JRun のインストール ディレクトリ*/docs/api ディレクトリにあります。サーブレット API に関するその他のマニュアルについては、http://java.sun.com/products/servlet/ にアクセスしてください。

---

javax.servlet

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

javax.servlet インターフェイス

次の表は、javax.servlet に含まれるインターフェイスの概要を示しています。

インターフェイス	説明
RequestDispatcher	要求の処理を、ほかのサーブレット、JSP、または HTML ファイルに転送するオブジェクトを定義します。また、応答にほかのサーブレットの出力を含めることもできます。
Servlet	サーブレット インスタンスを初期化して要求を処理し、サーブレット インスタンスを破棄するメソッドを定義します。GenericServlet クラスはこのインターフェイスを実装します。
ServletConfig	サーブレットの init メソッドに情報を渡すオブジェクトを定義します。名前/値ペア、サーブレットの名前、Web アプリケーションの ServletContext オブジェクトの参照が含まれています。
ServletContext	サーブレットがサーブレット コンテナに関する情報にアクセスするために使用するメソッドを定義します。また、アプリケーションの init パラメータが含まれています。1 台の仮想マシン上の Web アプリケーションごとに 1 つの ServletContext があります。
ServletRequest	クライアント要求の情報をカプセル化するオブジェクトを定義します。各インスタンスには、名前/値ペア、属性、および入力ストリームが含まれています。
ServletResponse	クライアントに返す情報をカプセル化するオブジェクトを定義します。バイナリ データまたは文字データを送信できます。
SingleThreadModel	各サーブレット インスタンスに一度に 1 つの要求だけを実行させるオブジェクトを定義します。このインターフェイスの詳細については、JavaDoc API のマニュアルを参照してください。

javax.servlet クラス

次の表は、javax.servlet に含まれるクラスの概要を示しています。

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

javax.servlet 例外

次の表は、javax.servlet に含まれる例外の概要を示しています。

例外	説明
ServletException	サーブレット の問題を知らせる一般的な例外
UnavailableException	サーブレット が使用可能でないことを示します。この例外は、一時的または永久的に使用不能であることを示すために使用できます。

javax.servlet.http

javax.servlet.http パッケージには、HTTP の機能を要求するサーブレットに適用されるインターフェイスおよびクラスが含まれています。次の表では、これらについて説明します。

javax.servlet.http インターフェイス

次の表は、javax.servlet.http に含まれるインターフェイスの概要を示します。

インターフェイス	説明
HttpServletRequest	ServletRequest を HTTP サーブレット 用に拡張します。このオブジェクト は、要求についてクッキー、属性、およびその他の情報にアクセスするために使用します。
HttpServletResponse	ServletResponse を HTTP サーブレット 用に拡張します。このオブジェクト は、ブラウザに応答を送信するために使用します。
HttpSession	複数のページ要求の間で保持されるユーザ情報が含まれています。セッションはセッション ID をキーとして識別され、セッション ID はクッキーまたは URL パラメーターに保存されます。
HttpSessionBinding Listener	オブジェクト がセッションからバインドされたとき、またはバインドを解除されたときに、オブジェクト にそのことを知らせます。このインターフェイスは、セッション固有のリソースを初期化およびクリーンアップするために使用します。

javax.servlet.http クラス

次の表は、javax.servlet.http に含まれるクラスの概要を示しています。

クラス	説明
Cookie	サーブレットをクライアント クッキーの作成、読み取り、および変更に使用します。クッキーを読み取るには、HttpServletRequest オブジェクトを使用します。クッキーを設定するには、HttpServletResponse オブジェクトを使用します。
HttpServlet	HTTP サーブレットを作成するために拡張する抽象クラス。このクラスをすべての Web 指向のサーブレットに使用します。
HttpSessionBindingEvent	HttpSessionBindingListener インターフェイスを実装しているオブジェクトがセッションからバインドまたはバインド解除されたときに、そのオブジェクトに渡されます。2 つのメソッド getName および getSession が含まれています。
HttpUtils	便利なユーティリティ メソッドが含まれています。getRequestURL、parsePostData、parseQueryString などのメソッドが含まれています。

サーブレット API に関するリファレンス情報

JRun には詳細なサーブレット API オンライン文書が添付されています。これは *JRun のインストール ディレクトリ* /docs/api にあります。また、最新の API 文書は http://java.sun.com/products/servlet からアクセスできます。

第 20 章

Java サーブレット API によるプログラミング

この章では、サーブレット API の基本クラスを使用したプログラミングの方法について説明します。基本的なサーブレットの概念については、203 ページの第 17 章「Java サーブレットの処理」を参照してください。その他のコーディング テクニックについては、237 ページの第 21 章「サーブレットの例」を参照してください。

目次

- GenericServlet クラスにおけるメソッドのコーディング 228
- HttpServlet クラスにおけるメソッドのコーディング 232

GenericServlet クラスにおけるメソッドのコーディング

GenericServletクラスでは、Servletインターフェイスを実装することにより 、HTTPではないサーブレット に機能を提供します。HttpServletクラスはGenericServletを書き換えるため、これらのメソッド は HttpServlet を拡張するサーブレット でも使用できます。

アプリケーションで GenericServlet クラスを拡張する場合、このクラスは service メソッドを書き換えます。必要に応じて、getServletInfo、init、destroy メソッドが書き換えられることもあります。さらに、GenericServlet クラスには、サーブレット、要求、およびアプリケーション情報にアクセスするためのメソッドが含まれています。

service メソッドの書き換え

JRun では、サーブレットが要求されるたびに、service メソッドが呼び出されます。GenericServlet を拡張するサーブレットは、service メソッドを書き換えます。

service メソッドで使用できるパラメータは次のとおりです。

- ServletRequest。クライアント要求に関する情報が含まれます。
- ServletResponse。クライアントにデータを返すことができます。

次の例では、serviceメソッド を書き換えます。通常、サーブレット により 拡張されるのは、GenericServlet ではなく、HttpServlet クラスであることに注意してください。

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

public class HelloPlainText extends GenericServlet {

public void service(ServletRequest req, ServletResponse resp)
 throws IOException, ServletException {
  PrintWriter out = resp.getWriter();
  out.println("Hello World.Plain text version.");
}
}
```

getServletInfo メソッド、init メソッド、および destroy メソッドの書き換え

GenericServlet クラスには、サーブレット に特定の機能を提供するために書き換えることができるメソッドが含まれています。

- getServletInfo。 このメソッドを使用して、サーブレットを説明できます。
- init。サーブレットがロードされると、JRun によって呼び出されるメソッドです。
- destroy。サーブレット をアンロード する前に、JRunによって呼び出されるメソッドです。

getServletInfo メソッド のコーディング

getServletInfo はオプションのメソッドで、ほかのクラスから呼び出して、サーブレット の説明にアクセスするために使用できます。このメソッド は引数を取らず、次の例のように String を返します。

```
public String getServletInfo() {
  String infoMessage = "EIS Servlet.Version 1.1";
  return infoMessage;
}
```

init メソッド のコーディング

JRun では、サーブレットが最初にロード されたときに init メソッドが呼び出されるので、データベース接続や、その他のグローバル変数、参照など、1 回だけ行う 必要のある設定や初期化ロジックをコーディングすることもできます。init メソッドを書き換えるためのオプションには、次の 2 つがあります。

- public void init(ServletConfig config)　このバージョンでは、init メソッドを書き換える場合、必ず、先頭行で super.init(config) を呼び出す必要があります。
- public void init()　このバージョンでは、init メソッドを書き換えるために、super.init(config) を呼び出す必要はありません。

次の例では、init メソッドに引数を与えずに使用しています。

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

public class UseInitParms extends HttpServlet {

  //既定値
  private static final int FONTSIZE = 3;
  private static final int COUNTER = 0;

  //既定の変数
  public static int fontSize;
  public static int counter;
```

```
public void init () throws ServletException {
  // 既定のフォント サイズを設定します。
  if (getInitParameter("fontSize") != null){
    try{
      this.fontSize = Integer.parseInt(getInitParameter("fontSize")) ;
    }
    catch (NumberFormatException e){
      this.fontSize = this.FONTSIZE;
    }
  }else this.fontSize = this.FONTSIZE;
  // カウンタを設定します。
  if (getInitParameter("counter") != null){
    try{
      this.counter = Integer.parseInt(getInitParameter("counter")) ;
    }
    catch (NumberFormatException e){
      this.counter = this.counter;
    }
  }else this.counter = this.counter;
}
```

destroy メソッド のコーディング

サーバーからサーブレットをアンロードする前に、JRun により destroy メソッドが呼び出されます。したがって、データベースの切断やステート管理のような、アプリケーション シャットダウン ロジックをコーディングできます。データベースから切断するコード の例は次のとおりです。

```
public void destroy() {
  // dbConnection は init メソッドにより設定された
  // インスタンス変数であるとします。
  if (dbConnection != null) {
    dbConnection.close();
  }
}
```

サーブレット情報、要求情報、およびアプリケーション情報へのアクセス

GenericServlet クラスに含まれる次のメソッドを呼び出して、情報やログ メッセージにアクセスできます。

- getInitParameter および getInitParameterNames これらのメソッドを使用して、サーブレットの初期化パラメータにアクセスできます。
- getServletConfig このメソッドにより ServletConfig オブジェクトが返され、初期化パラメータやコンテキスト 情報にアクセスできます。GenericServlet には、この情報にアクセスするためのメソッドが含まれています。したがって、通常、getServletConfig は使用しません。
- getServletContext このメソッドにより、ServletContext オブジェクトが返されるので JRun と対話するメソッドにアクセスできます。コード例については、250 ページの「サーブレット コンテキストの使用」を参照してください。
- log。ログ ファイルにメッセージが書き込まれます。

初期化パラメータ の使用

次の例では、getInitParameterNames メソッドと getInitParameter メソッドの両方が使用されています。

```
// これは doGet メソッドの一部であると仮定します。
Enumeration eParmNames = getInitParameterNames();
while (eParmNames.hasMoreElements()) {
  String parm = (String) eParmNames.nextElement();
  out.println(" " + parm + ":" + getInitParameter(parm) + "<br>");
}
```

メッセージのロギング

log メソッドを使用して、プログラマが指定したメッセージをサーブレット のホスト になっている JRun サーバーのログ ファイルに書き込むことができます。

メモ
441 ページの第 38 章「ログ」での説明どおり、記録されるログ情報の分量は自動的に変わります。

次の例では、ユーザのアクセス情報がログに記録されます。

```
HttpSession thisSession = req.getSession();
String userName = (String)thisSession.getAttribute("name");
if(userName != null) {
  out.println("<h2>Welcome " + userName + "</h2>");
  // ロギング用の情報を準備
  // この例では、ユーザ名と IP アドレスがログに記録されます。
  String logMsg = userName + ", " + req.getRemoteAddr();
log(logMsg);
}
```

HttpServlet クラスにおけるメソッドのコーディング

HttpServletクラスにより、GenericServletクラスが拡張されます。開発するサーブレット クラスの大半で、HttpServlet クラスが拡張されます。HttpServlet を使用してプログラミングするには、service メソッド、または HTTP 固有の要求処理メソッドのどちらかを書き換える必要があります。

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

これらのメソッドで使用できるパラメータは次のとおりです。

- HttpServletRequest。HTTP ヘッダと、その他のクライアント要求情報が含まれています。
- HttpServletResponse。ブラウザに HTML を返すことができます。

service メソッドの書き換え

service メソッドは、サーブレットが要求されるたびに呼び出されます。

- GenericServlet を拡張するサーブレットの場合、service メソッドを書き換える必要があります。また、init メソッドや destroy メソッドを書き換えることもできます。
- HttpServlet を拡張するサーブレットの場合、service メソッドの既定の実装で、要求が適切な doXxx メソッドに転送されるようになっています。たとえば、HTTP GET 要求が受信されると、service メソッドにより、doGet メソッドが呼び出されます。既定の service メソッドを書き換える場合、このメソッドで HTTP 要求の全種類を処理できるようになっているか、または適切な doXxx メソッドに対して要求をディスパッチするロジックが含まれている必要があります。

doGet メソッドの書き換え

doGet メソッドは、HTTP GET 要求のために呼び出されます。ユーザが URL をタイプ、リンクをクリック、または method=GET を指定するフォームを送信した場合、Web ブラウザにより HTTP GET 要求が送信されます。

次の例で示されているサーブレットは、doGet メソッドを使用して基本情報を表示します。

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

public class DisplayInfo extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title>Display Information");
  out.println("</title></head><body>");
  out.println("<h1>displayInfo Servlet</h1>");
  out.println("<h2>Request Information</h2>");
  out.println("Scheme:" + req.getScheme() + "<br>");
  out.println("Server Name:" + req.getServerName() + "<br>");
  out.println("RemoteAddr:" + req.getRemoteAddr() + "<br>");
  out.println("RemoteHost:" + req.getRemoteHost() + "<br>");
  out.println("Method:" + req.getMethod() + "<br>");
  out.println("Query String:" + req.getQueryString() + "<br>");
  out.println("Request URI:" + req.getRequestURI() + "<br>");
  out.println("Servlet Path:" + req.getServletPath() + "<br>");
  out.println("</body></html>");
}
}
```

doPost メソッドの書き換え

doPost メソッドは、HTTP POST 要求のために呼び出されます。ユーザが method=POST を指定するフォームを送信した場合、Web ブラウザにより HTTP POST 要求が送信されます。

以下に示すのは、POST 要求を通じてサーブレットを呼び出す HTML フォームの例です。

```
<html>
<head>
  <title> Login to the System </title>
</head>

<body bgcolor="Silver">
<h1> Login to the System </h1>

<!-- ログイン フォームの表示 -->
<form action="/servlet/selectionForm" method="POST">
<p>Name: 
<input type="Text" name="myName" size="30">
<p>
<input type="Submit" value="Log In">
</form>
</body>
</html>
```

次の例にあるサーブレットは、doPost メソッドを使用して、渡された値にアクセスしています。

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

public class selectionForm extends HttpServlet {

public void doPost(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  String thisName = "Unknown Name";
  // ログイン情報からユーザ名を取得します。
  String[] attrArray = req.getParameterValues("myName");
  // 呼び出し側フォームには、myName の値が 1 つしかないと想定します。
  if(attrArray != null) {
    thisName = attrArray[0];
  }
  out.println("<html><head><title>Choose Information to Display");
  out.println("</title></head><body>");
  out.println("<h1>Welcome " + thisName + "</h2>");
  out.println("<h2>Choose Information to Display</h2>");
  // ¥ を使用して、二重引用符をエスケープします。
  out.println("<form action=¥"/servlet/displayInfo¥"
        method=¥"post¥">");
  // HTTP 要求情報を表示するチェックボックス
  out.println("<p>Display request information? ");
  out.println("<input type=¥"Checkbox¥" name=¥"requestInfo¥" checked>");
  // クッキー情報を表示するチェックボックス
  out.println("<p>Display cookies? ");
  out.println("<input type=¥"Checkbox¥" name=¥"showCookies¥" checked>");
  out.println("<br>");
  out.println("<input type=¥"Submit¥">");
  out.println("</form>");
  out.println("</body></html>");
}
}
```

ほかの HTTP メソッドの書き換え

次の表で説明するように、HttpServlet クラスには、その他の HTTP 要求タイプをサポート するメソッド が用意されています。HTTP/1.1 ではすべての要求タイプがサポートされていますが、HTTP/1.0 では GET、HEAD、および POST だけがサポートされている点に注意してください。

要求タイプ	メソッド	コメント
DELETE	doDelete	DELETE 要求の詳細については、HTTP のマニュアルを参照してください。
PUT	doPut	PUT 要求の詳細については、HTTP のマニュアルを参照してください。
HEAD	doHead	doGet メソッドを実行しますが、返されるのはヘッダだけです。
OPTIONS	doOptions	既定の実装では、サポートされているオプションの一覧が返されます。通常、このメソッドを書き換える必要はありません。
TRACE	doTrace	既定の実装では、TRACE 要求のヘッダがすべて一覧に表示されます。通常、このメソッドを書き換える必要はありません。

第 21 章

サーブレット の例

この章では、頻繁に使用するサーブレット機能のコード例を示します。

目次

- 制御の受け渡し 238
- セッションのトラッキング 240
- データベースへのアクセス 242
- クッキーの処理 249
- サーブレット コンテキストの使用 250
- ほかのファイルからのコンテンツのインクルード 252

制御の受け渡し

RequestDispatcher オブジェクトの forward メソッドを使用して、ほかのサーブレットまたは JSP に制御を渡すことができます。forward メソッドを使用する場合、呼び出し側のサーブレットから出力ストリームに書き込むことはできません。必要であれば、呼び出し側のサーブレットは、ServletRequest オブジェクトの setAttribute メソッド (JSP の request オブジェクト) を使用して属性を設定することにより、目的のサーブレットに情報を渡します。その場合、ターゲット プログラムは次の方法で属性にアクセスします。

- サーブレットは ServletRequest オブジェクトの getAttribute メソッドによってこれらの属性にアクセスできます。
- JSP は request オブジェクトの getAttribute メソッドによってこれらの属性にアクセスできます。

RequestDispatcher オブジェクト の参照は、getRequestDispatcher メソッド によって取得します。このメソッドは ServletContext オブジェクトと ServletRequest オブジェクトの両方に含まれています。唯一の違いは、ServletRequest.getRequestDispatcher で指定するパス名には先頭のスラッシュが不要なことです。したがって、相対 URL を使用することができます。ServletContext.getRequestDispatcher の場合は、先頭のスラッシュが必要です。次に、ServletContext.getRequestDispatcher を使用した例を示します。

ほかのサーブレットに制御を渡す方法

次の例では、ほかのサーブレットに制御を渡します。

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

public class TestCaller extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  // ServletContext を取得します。
  ServletContext sc = this.getServletContext();
  // RequestDispatcher で目的のサーブレットをラップします。
 RequestDispatcher rd = sc.getRequestDispatcher("/servlet/callMe");
  if (rd !=null) {
    // 制御をサーブレットに渡します。
    try {
   rd.forward(req, resp);
    }
    catch (Exception e) {
      sc.log("Problem invoking servlet.", e);
    }
  }
}
}
```

JSP に制御を渡す方法

RequestDispatcher オブジェクトを使用して、JSP に制御を渡すことができます。

メモ
この方法を使用すると、HTML ページに制御を渡すこともできます。

次の例では、JSP に制御を渡します。

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

public class TestCaller extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
          throws IOException, ServletException {
  // ServletContext を取得します。
  ServletContext sc = this.getServletContext();
  // RequestDispatcher で JSP をラップします。
 RequestDispatcher rd = sc.getRequestDispatcher("/test.jsp");
  if (rd !=null) {
    // JSP に制御を渡します。
    try {
   rd.forward(req, resp);
    }
    catch (Exception e) {
      sc.log("Problem invoking JSP.", e);
    }
  }
}
}
```

セッションのトラッキング

HTTP プロトコルはステートレスであるため、Web アプリケーションは何らかの方法で、要求間でセッション情報を保持する必要があります。したがって、サーブレットが正しく機能するには、どのユーザが要求を送信しているか、そのユーザが Web アプリケーションとの対話を開始しているか、そのユーザが何を実行していたかをサーブレットが把握している必要があります。

単純な実装ではユーザ名が、セキュリティ保護されている実装ではユーザ名とパスワードが、電子商取引システムではショッピング カートがアプリケーションに記憶されます。サーブレット API を使用しないアプリケーションの場合、通常はアプリケーション固有の保存メカニズムを使用してセッション情報を保存し、非表示のフォーム フィールドまたはクッキーを通じてキー値をブラウザに返します。後続の要求では、このキー値を使用して前に保存されているセッション情報を取得します。

Java サーブレット 仕様では HttpSession インターフェイスが定義されています。このインターフェイスによって、実装の詳細を把握せずにセッション管理を行うことができます。HttpSession インターフェイスは HttpServletRequest クラスから取得できます。

getSession メソッドによってセッションを確立し、getAttribute メソッドによってセッションからの値にアクセスします。

セッションの確立

1 セッションで保存する値を確立します。

```
public class SelectionForm extends HttpServlet {

public void service(HttpServletRequest req,
  HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType(“text/html”);
  PrintWriter out = resp.getWriter();
  String thisName = "Unknown Name";
  // ログイン情報からユーザ名を取得します。
  String[] attrArray = req.getParameterValues("myName");
  // 呼び出しフォームには、myName の値が 1 つしかないと想定します。
  if(attrArray != null && attrArray.length > 0) {
    thisName = attrArray[0];
  }
}
...
```

2 HttpServletRequest オブジェクトの getSession メソッドを呼び出すことによって新しいセッションを作成します。

```
// セッションを作成します。
HttpSession thisSession = req.getSession();
```

3 セッションに属性を割り当てます。

```
// この例では、ユーザ名を保存します。
thisSession.setAttribute("name", thisName);
```

セッション情報へのアクセス

1 HttpServletRequest オブジェクトの getSession メソッドを呼び出すことによって session オブジェクトの参照を取得します。

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

public class DisplayInfo extends HttpServlet {

public void service(HttpServletRequest req,
  HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title>Display Information");
  out.println("</title></head><body>");
  out.println("<h1>displayInfo Servlet</h1>");
  // session オブジェクトからログイン ユーザを取得します。
  HttpSession thisSession = req.getSession();
...
```

2 HttpSession オブジェクトの getAttribute メソッドを使用して希望する属性値にアクセスします。必ず結果を適切なタイプに変換してください。

```
String userName = (String)thisSession.getAttribute("name");
```

3 コード内での属性値を使用します。

```
if(userName != null) {
  out.println("<h2>Welcome " + userName + "</h2>");
}
```

データベースへのアクセス

JRun データベースには、Java database connectivity API (JDBC) を使用してアクセスします。JDBC は、Sun のドライバ マネージャと、JDBC ドライバ (JDBC-ODBC ブリッジは Sun により提供、ほかの JDBC ドライバはサードパーティ ベンダにより提供) を使用します。JDBC によるデータベース アクセスに慣れていない場合は、ご使用の JDBC ドライバのマニュアル、または JDBC に関する解説書をお読みください。

JDBC を使用してデータベースにアクセスするには、ユーザ アプリケーションに次のものが必要です。

- **データベース ドライバ** `Class.forName` メソッドを使用してデータベース ドライバのインスタンスを生成します。
- **データベース接続オブジェクト** `DriverManager.getConnection` メソッドを使用して、`Connection` オブジェクトを確立します。
- **データベース URL** データベースの URL には、プロトコル (`jdbc`)、ドライバ サブプロトコル (例 : `odbc`、`sequelink`)、およびデータベースを識別するドライバ依存のデータが含まれています。データベースの URL のフォーマットおよび内容については、ご使用のデータベース ドライバのマニュアルを参照してください。
- **ステートメント オブジェクト** `Statement` オブジェクト内のメソッドを使用して SQL ステートメントを実行します。
- **Result セット** `ResultSet` オブジェクトを使用してデータベースから取得したデータを保存したり、そのデータにアクセスします。

ここでは、次の手段でデータベースにアクセスする方法について説明します。

- JDBC-ODBC ブリッジ
- JDBC ドライバ
- JRun データ ソース サービス

JDBC-ODBC ブリッジの使用

JDBC-ODBC ブリッジは、既存の ODBC ドライバをインターフェイスでつなぎ、データベース アクセスを提供する JDBC ドライバです。Type 1 JDBC ドライバとも呼ばれます。JDK は JDBC-ODBC ブリッジを含んでいるため、データベース アクセスの概念の学習に適しています。しかし、実際の製品開発にはネイティブの JDBC ドライバを使用した方が効率的です。

次の例では、JDBC-ODBC ブリッジを使用して `cfsnippets` データソースにアクセスします。

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

public class DbTest extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
```

```
 throws IOException, ServletException  {
 // まず、セットアップを行います。
// JDBC ドライバ
String dbDriverName = "sun.jdbc.odbc.JdbcOdbcDriver";
// 接続 URL
String dbConnectionURL = "jdbc:odbc:cfsnippets";
// 接続オブジェクト
Connection dbConnection = null;
// ステートメント オブジェクト
 Statement dbStatement = null;
// 実行する SQL ステートメント
String sqlStatement = "Select * from Courses";
// 結果セット オブジェクト
ResultSet dbResultSet = null;

 // サーブレット出力を開始します。
 PrintWriter out = resp.getWriter();
 resp.setContentType("text/html");
 out.println("<html><head><title>DB Test</title></head><body>");
 out.println("<h1>Database Test</h1>");

 // db アクセス コードを開始します。
try {
 // JDBC ドライバ インスタンスを作成します。
 Class.forName(dbDriverName).newInstance();

 // db 接続を作成します。
 dbConnection = DriverManager.getConnection(dbConnectionURL);

 // ステートメント オブジェクトを作成します。
 dbStatement = dbConnection.createStatement();

 // クエリを実行します。
 dbResultSet = dbStatement.executeQuery(sqlStatement);

   // 列ヘッダを表示します。
   ResultSetMetaData rsMetaData = dbResultSet.getMetaData()
   int colCount = rsMetaData.getColumnCount();
   // 結果をテーブルに表示します。
   // テーブルを開始します。
   out.println("<table>");

   // 新しい行を開始します。
   String thisLine = "";
   for (int i = 0; i < colCount; i++) {
     thisLine += "<th>";
     // 1 を基準とした列ヘッダ
     thisLine += rsMetaData.getColumnLabel(i + 1);
     thisLine += "</th>";
   }
```

```
      // ヘッダを表示します。
      out.println("<tr>" + thisLine + "</tr>");

      // 結果セットを表示します。
      int rows = 0;

      // 結果セットをステップスルーします。
      while (dbResultSet.next()) {
        rows++;

        // 行の内容を表示します。
        thisLine = "";
        for (int i = 0; i < colCount; i++) {
          // 1 を基準とした列のインデックス
          thisLine += "<td>";
          thisLine += dbResultSet.getString(i +1);
          thisLine += "</td>";
        } // for を終了します。
        out.println("<tr>" + thisLine + "</tr>");
      } // while を終了します。
        // テーブルを終了します。
        out.println("</table>");
    } // try を終了します。
    catch (Exception e){
      out.println("<p>Exception in main try block");
      e.printStackTrace();
    }

    // すべてにこの処理を実行します。
    finally {
      // クリーンアップ
      try {
        if (dbResultSet != null) {
          dbResultSet.close();
        }
        if (dbStatement != null) {
          dbStatement.close();
        }
        if (dbConnection != null) {
          dbConnection.close();
        }
      }
      catch (SQLException sqlex) {
        out.println("<p>SQL exception in finally block");
        sqlex.printStackTrace();
      }
    } // finally を終了します。

    // サーブレット出力を終了します。
    out.println("</body></html>");
  }
  }
```

JDBC ドライバの使用

JDBC-ODBC ブリッジは ODBC ドライバを通じてデータベースにアクセスします。また、ネイティブの JDBC ドライバを使用することもできます。次の一覧は種々のタイプの JDBC ドライバを示しています。

- **ネイティブ API ドライバ** Type 2 のドライバとも呼ばれます。これは Java コードをネイティブ データベース ライブラリでラップします。Type 2 のドライバを使用するには、JRun はデータベース クライアントの API ライブラリにアクセスできる必要があります。
- **ネット プロトコル ドライバ** Type 3 のドライバとも呼ばれます。これは汎用ネットワーク プロトコルとミドルウェア コンポーネントを使用してデータベースと通信します。
- **ネット プロトコル ドライバ** Type 4 のドライバとも呼ばれます。これはデータベース固有のネイティブ プロトコルを使用してデータベースと通信します。

JDBCドライバを使用しているときに作成するコード は、JDBC-ODBC ブリッジで使用するコード と似ています。唯一の違いは、次のスニペットに示されているようにドライバ クラスとデータベース URL です。

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

public class DbTest extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException,
  ServletException {
  // JDBC ドライバ
 String dbDriverName = "intersolv.jdbc.sequelink.SequeLinkDriver";
  // 接続 URL
 String dbConnectionURL = "jdbc:sequelink://DBSRV:8300/
        [SQL SERVER];Database=cfsnippets";
  // 残りのコードは、JDBC-ODBC ブリッジの例と同じです。
  // 接続オブジェクト
  Connection dbConnection = null;
  // ステートメント オブジェクト
  Statement dbStatement = null;
  // 実行する SQL ステートメント
  String sqlStatement = "Select * from Courses";
  // 結果セット オブジェクト
  ResultSet dbResultSet = null;
  ...
  // db アクセス コードを開始します。
  try {
    // JDBC ドライバ インスタンスを作成します。
    Class.forName(dbDriverName).newInstance();
```

```
        // db 接続を作成します。
        dbConnection = DriverManager.getConnection(dbConnectionURL);

        // ステートメント オブジェクトを作成します。
        dbStatement = dbConnection.createStatement();

        // クエリを実行します。
        dbResultSet = dbStatement.executeQuery(sqlStatement);
    ...
```

JRun データ ソース サービスの使用

JRun データ ソース サービスを使用すると、JRun 内の JDBC データソースを定義できます。コード では、JDBCドライバ情報をハード コード 化する代わり に、データソース名を使用してデータベースを参照します。したがって、JRun でサーブレットを再コンパイルせずにデータ ソース 情報を変更できます。JRun データソースでは、オプションの接続プールに加えて、サーブレットの移植性を強化します。これは、JRun でお勧めするデータベースへのアクセス方法です。

JRun でデータ ソースを定義する方法については、『 JRun セットアップ ガイド 』を参照してください。

次のコード例では、JRun データ ソース サービスを使用して JDBC データ ソース情報にアクセスします。

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

public class DbTest extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  // JRun データ ソース サービスで定義された名前
  String dsName = "cfsnippetsJRun";

  // 接続オブジェクト
  Connection dbConnection = null;
  // ステートメント オブジェクト
  Statement dbStatement = null;
  // 実行する SQL ステートメント
  String sqlStatement = "Select * from Courses";
  // 結果セット オブジェクト
  ResultSet dbResultSet = null;
```

```
// サーブレット出力を開始します。
PrintWriter out = resp.getWriter();
resp.setContentType("text/html");
out.println("<html><head><title>Database Test</title>");
out.println("</head><body>");
out.println("<h1>Database Test</h1>");

// db アクセス コードを開始します。
try {
// JNDI InitialContext オブジェクトを定義します。
InitialContext ctx = new InitialContext();
// InitialContext でデータ ソースを検索します。
DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/" +
      dsName);
dbConnection = ds.getConnection();

  // ステートメント オブジェクトを作成します。
  dbStatement = dbConnection.createStatement();

  // クエリを実行します。
  dbResultSet = dbStatement.executeQuery(sqlStatement);

  // 列ヘッダを表示します。
  ResultSetMetaData rsMetaData = dbResultSet.getMetaData()
  int colCount = rsMetaData.getColumnCount();
  // 結果をテーブルに表示します。
  // テーブルを開始します。
  out.println("<table>");

  // 新しい行を開始します。
  String thisLine = "";
  for (int i = 0; i < colCount; i++) {
    thisLine += "<th>";
    // 1 を基準とした列ヘッダ
    thisLine += rsMetaData.getColumnLabel(i + 1);
    thisLine += "</th>";
  }
  // ヘッダを表示します。
  out.println("<tr>" + thisLine + "</tr>");

  // 結果セットを表示します。
  int rows = 0;

  // 結果セットをステップスルーします。
  while (dbResultSet.next()) {
    rows++;
```

```
        // 行の内容を表示します。
        thisLine = "";
        for (int i = 0; i < colCount; i++) {
          // 1 を基準とした列のインデックス
          thisLine += "<td>";
          thisLine += dbResultSet.getString(i +1);
          thisLine += "</td>";
          out.println("<tr>" + thisLine + "</tr>");
        }
        // テーブルを終了します。
        out.println("</table>");
      }
    }
    catch (Exception e){
      out.println("<p>Exception in main try block");
      e.printStackTrace();
    }
    // すべてにこの処理を実行します。
    finally {
      // クリーン アップ
      try {
        if (dbResultSet != null) {
          dbResultSet.close();
        }
        if (dbStatement != null) {
          dbStatement.close();
        }
        if (dbConnection != null) {
          dbConnection.close();
        }
      }
      catch (SQLException sqlex) {
        out.println("<p>SQL exception in finally block");
        sqlex.printStackTrace();
      }
    }

    // サーブレット出力を終了します。
    out.println("</body></html>");
  }
  }
```

クッキーの処理

クッキーは、サーバー側のアプリケーションが個別のブラウザに情報を保存するために使用する一般的な手段です。ブラウザに保存されたクッキーはサーバー側アプリケーションで取得できます。クッキーを使用することによって、Web アプリケーションで各ブラウザで使用する変数を作成できます。そのような変数にユーザ名または最後にアクセスした日付を含めることができます。クッキーによるセッション記録を有効にした場合、JRun は jsessionid という名前のセッション トラッキング クッキーを作成します (JMC でクッキーの名前を指定することもできます)。

クッキーは、ステートレスである HTTP プロトコルを補完するパーシスタンス メカニズムを提供します。クッキーには一時的クッキーと永久的クッキーがあります。

- **一時的クッキー** ブラウザ インスタンスを終了するまで有効です。一時的クッキーは、セキュリティ保護されたシステムへのアクセスの認証の際にユーザの名前とパスワードを保持するのに適しています。既定では、サーブレット API は一時的クッキーを作成します。
- **永久的クッキー** 期限切れになるか削除されるまで有効です。永久的クッキーは、ユーザ名や最後にアクセスした日付などの情報を保持するのに適しています。永久的クッキーを作成するには、cookie オブジェクトの setMaxAge メソッドを使用します。

クッキーは現在、市販のほとんどのブラウザでサポートされていますが、サポートの内容がブラウザによって異なります。また、クライアントはブラウザの設定によってクッキーのサポートを無効にすることもできます。

クッキーを確立するには

1 クッキーで保存する値を確立します。

```
Date dt = new Date();
String todayString = dt.toString();
```

2 新しい Cookie オブジェクトを作成します。

```
// クッキー インスタンスを作成します。
// この例では、現在の日付と時刻を保存します。
Cookie lastVisit = new Cookie("lastVisit", todayString);
```

3 クッキー オブジェクト をサーブレット の応答オブジェクト に関連付けることによってクッキーをブラウザに返します。

```
resp.addCookie(lastVisit);
```

クッキーにアクセスするには

1 クッキーにアクセスするには、HttpServletRequest オブジェクトの getCookies メソッドを使用します。

```
Cookie[] myCookies = req.getCookies();
```

2 Cookie オブジェクトの getName および getValue メソッドを使用してクッキーおよびその値にアクセスします。この例では、クッキーの名前およびそれに関連付けられている値を表示します。

```
for(int i=0; i<myCookies.length; i++) {
  out.println("Cookie name:" + myCookies[i].getName());
  out.println(" Value:" + myCookies[i].getValue() + "<br>");
}
```

サーブレット コンテキストの使用

ServletContext オブジェクトを使用して、アプリケーションに関する情報を保存したり、次の環境情報にアクセスできます。

- 初期化パラメータ
- MIME タイプ
- バージョン情報
- パス情報

サーブレットは getServletContext メソッドを使用して ServletContext オブジェクトの参照を取得します。

この例では、いくつかのサーブレット コンテキスト情報を表示します。

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

public class GetServletContextInfo extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {

  // 戻り値のタイプを設定します。
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title>Servlet Context</title>");
  out.println("</head><body>");
  out.println("<h1>Servlet Context Information</h1>");

  // ServletContext 情報を取得します。
  ServletContext scntxt = this.getServletContext();
  // サーバー情報
  out.println("Server information:"+ scntxt.getServerInfo() + "<br>");
```

```
    // メジャー/マイナー バージョン
    int majorVersion = scntxt.getMajorVersion();
    int minorVersion = scntxt.getMinorVersion();
    out.println("Major version:" + majorVersion + "<br>");
    out.println("Minor version:" + minorVersion + "<br>");
    // init パラメータがある場合は取得します。
    java.util.Enumeration parmEnum = scntxt.getInitParameterNames();
    if (parmEnum.hasMoreElements()) {
      out.println("<h2>ServletContext Parameters</h2>");
    }
    while (parmEnum.hasMoreElements()) {
      String name = (String)parmEnum.nextElement();
      out.println("<b>"+name+": </b>");
      out.println(scntxt.getInitParameter(name) + "<br>");
    }
    // ServletContext 属性
    java.util.Enumeration attrEnum = scntxt.getAttributeNames();
    if (attrEnum.hasMoreElements()) {
      out.println("<h2>ServletContext Attributes</h2>");
    }
    while (attrEnum.hasMoreElements()) {
      // 常に属性を適切なクラスに割り当てます。
      String attrName = (String)attrEnum.nextElement();
      out.println("<b>" + attrName + ": </b>");
      out.println(scntxt.getAttribute(attrName) + "<br>");
    }
    // サーブレットの実際のパスを取得します。
    String path = req.getServletPath();
    out.println("<b>Full servlet path:</b>");
    out.println(scntxt.getRealPath(path) + "<br>");
    // ServletContext のログを記録します。
    Date now = new Date();
    scntxt.log("Testing ServletContext:" + now);

    out.println("</body></html>");
  }
  }
```

ほかのファイルからのコンテンツのインクルード

次の方法で、サーブレットにコンテンツをインクルードできます。

- RequestDispatcher オブジェクトの include メソッド
- ServletContext オブジェクトの getResource メソッド

RequestDispatcher オブジェクトの使用方法については、238 ページの「制御の受け渡し」を参照してください。

include メソッドの使用

RequestDispatcher オブジェクト の include メソッド を使用して サーブレット に複数のタイプのコンテンツをインクルードできます。

- **テキスト** RequestDispatcher オブジェクトがテキスト ファイルをラップしている場合、include メソッドはテキストを出力ストリームにコピーします。このテキストには HTML タグを含めることができます。
- **サーブレット** RequestDispatcher オブジェクトがサーブレットをラップしている場合、include メソッドはサーブレットを呼び出します。
- **JSP** RequestDispatcher オブジェクトが JSP をラップしている場合、include メソッドは JSP を呼び出します。

include メソッドを使用するとき、呼び出し側サーブレットは include メソッドを呼び出す前と後に ServletOutputStream、PrintWriter、または out に書き込むことができます (out は JSP の場合のみ)。必要な場合は、ServletRequest オブジェクトの setAttribute メソッドを使用してターゲットのサーブレットまたは JSP に情報を渡すことができます。238 ページの「制御の受け渡し」を参照してください。

次の例では、サーブレットをインクルードします。

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

public class TestCallerInclude extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<html><head><title>Calling Another Servlet");
  out.println("</title></head><body>");
  out.println("<h1>Calling Another Servlet</h1>");
  out.println("<p>This text comes from testCallerInclude.");
```

```
  // ServletContext を取得します。
  ServletContext sc = this.getServletContext();
  // RequestDispatcher でサーブレットをラップします。
 RequestDispatcher rd = sc.getRequestDispatcher("/servlet/includeMe");
  if (rd !=null) {
    // サーブレットをインクルードします。
    try {
   // インクルードされたサーブレットの制御は、それ自体のバッファにしかありません。
      // したがって、呼び出し側サーブレットのバッファにアクセスできません。
   rd.include(req, resp);
    }
    catch (Exception e) {
      sc.log("Problem invoking servlet.", e);
    }
  }
  // HTML を終了します。
  out.println("</body></html>");
}
}
```

getResource メソッドの使用

ServletContext オブジェクトの getResource メソッドを使用してサーブレットにコンテンツをインクルードします。getResource メソッドは URL オブジェクトを返します。その後は、この URL オブジェクトを使用してコンテンツにアクセスできます。URL オブジェクトを使用する利点は、ブラウザに返す前にコンテンツを解析できることです。

次の例では、getResource メソッドによってコンテンツをインクルードします。

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

public class TestGetResource extends HttpServlet {

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html");
  ServletOutputStream out = resp.getOutputStream();
  out.println("<html><head><title>Including Content");
  out.println("</title></head><body>");
  out.println("<h1>Including Content through getResource</h1>");
  out.println("<p>This text comes from the calling class.");
```

```
    // ServletContext を取得します。
    ServletContext sc = this.getServletContext();
    try {
      // リソースを取得します。
    URL u = sc.getResource("/includedText.htm");
      if (u !=null) {
        // コンテンツにアクセスし、InputStream に割り当てます。
        InputStream in = (InputStream)u.getContent();
        byte[] buf = new byte[255];
        int numRead = in.read(buf);
        while(numRead != -1){
          out.write(buf, 0, numRead);
          numRead = in.read(buf);
        }
      }else {
        out.println("<p>u was null");
      }
    }
    catch (Exception e) {
      sc.log("Problem including content.", e);
    }
    // HTML を終了します。
    out.println("</body></html>");
  }
  }
```

第 22 章

カスタム タグとタグ ライブラリの作成

この章では、カスタム タグとタグ ライブラリを利用して JSP をより使いやすくする方法について説明します。

目次

- カスタム タグとタグ ライブラリについて 256
- タグ ライブラリのコーディング 257

カスタム タグとタグ ライブラリについて

JavaServer Pages バージョン 1.1 の仕様書には、**タグ ライブラリ**に関するフレームワークが記述されています。タグ ライブラリは、関連する機能のセットをカプセル化するために利用できる強力な機能です。JavaServer Pages バージョン 1.1 の仕様書の詳細については、http://java.sun.com/products/jsp/index.html を参照してください。

タグ ライブラリは、**タグ**または**カスタム タグ**と呼ばれる 1 つ以上の**アクション**から構成され、関連する Java **タグ ハンドラ** クラスでコーディングされている処理が、それぞれのタグによって実行されます。Java 開発者は、それぞれのカスタム タグの機能 (属性を含む) を定義し、タグ ハンドラをコーディングするとともに、タグ ライブラリ記述子 (TLD) ファイルで、それぞれのカスタム タグを定義します。TLD ファイルでは、タグへの本文の組み込み可能性や必須属性など、その他の情報も定義します。

この章では、Java で書かれたタグ ハンドラを作成する方法について説明します。また、JRun には JSP でタグ ハンドラを作成する機能もあります。詳細については、155 ページの第 11 章「JSP でのカスタム タグの作成」を参照してください。

カスタム タグでスクリプト変数を作成する場合、Java 開発者はタグ拡張情報 (TEI、Tag Extra Information) ファイルも作成する必要があります。TEI ファイルは、JSP コードで使用するスクリプト変数とそのスコープを定義する Java クラスです。TEI ファイルを使用して、変換時に属性を検証することもできます。

Java 開発者と JSP 開発者とでは、タグ ライブラリとの対話が次のように異なります。

- Java 開発者は、タグ ライブラリ内のクラスやサポート ファイルのコーディング、文書化、およびパッケージ化を実施します。
- JSP 開発者の視点から見れば、タグ ライブラリ は、特定のタイプの処理の実行時に使用するカスタム タグが格納されているライブラリです。JSP 開発者は、TLD ファイルの格納場所あるいは、`web.xml` ファイルの `taglib` 要素で指定されている URI、ライブラリ内のカスタム タグの名前、タグ属性、スクリプト変数、スクリプト変数の使用法、およびスクリプト変数のスコープを知っておく必要があります。

タグ ライブラリのコーディング

Java 開発者は、タグ ハンドラ、TEI クラス (オプション)、サポート クラス (オプション)、および TLD ファイルをコーディングします。また、ライブラリ 内のカスタム タグの名前、タグ属性、スクリプト変数、スクリプト変数の使用法、およびスクリプト変数のスコープを文書化します。さらに、タグ ライブラリの JAR ファイルを作成し、それを `WEB-INF/lib` 内に配置し、必要があれば、Web アプリケーションの `web.xml` ファイル内に `taglib` 要素をコーディングします。

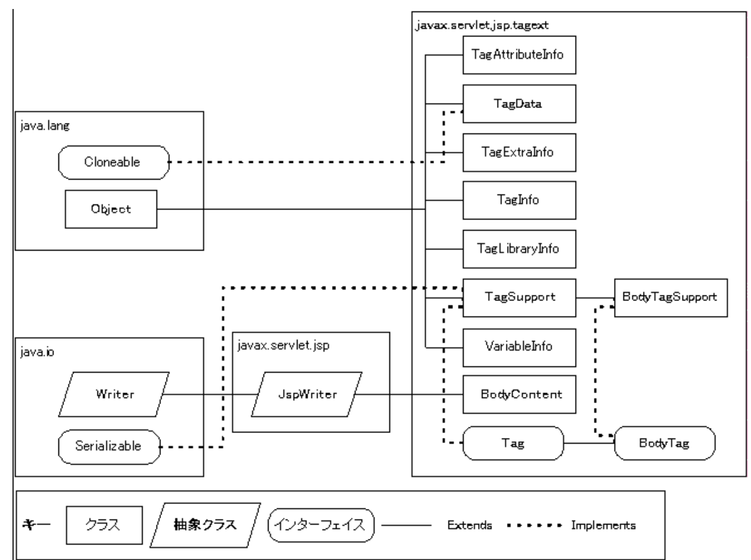
タグ ハンドラおよび TEI クラスは、次のいずれかの格納場所に保存します。

- `WEB-INF/classes` このディレクトリは、予備段階の開発やテストを実施するときにタグ ハンドラと TEI クラスを保存するのに適しています。ただし、タグ ライブラリをパッケージ化するときは、事前にタグ ハンドラと TEI クラスを `WEB-INF/lib` 内の `.jar` ファイルに保存しておく必要があります。
- `WEB-INF/lib` パッケージ化や公開を行う場合は、タグ ハンドラ、TEI クラス、およびその他の関連クラスを `WEB-INF/lib` 内の JAR ファイルに保存する必要があります。

推奨されている TLD ファイルの配置方法など、その他のパッキング情報については、276 ページの「タグ ライブラリのパッケージ化」を参照してください。

クラスとインターフェイス

次の図に示すように、カスタム タグやタグ ライブラリをコーディングするときは、`javax.servlet.jsp.tagext` パッケージ内のクラスとインターフェイスを使用します。



主なクラスとインターフェイスは次のとおりです。

- Tag インターフェイス　関連する開始タグと終了タグによって呼び出される基本的な開始メソッドと終了メソッドを定義します。
- BodyTag インターフェイス　カスタム タグによって本文テキストを操作する場合や、必要に応じて、結果やループを変更する場合に使用する追加メソッドを定義します。
- TagSupport クラス　Tag インターフェイスを実装します。これは、本文テキストと対話しないタグ ハンドラに応じて拡張可能なヘルパ クラスです。
- BodyTagSupport クラス　BodyTag インターフェイスを実装します。これは、本文テキストと対話するタグ ハンドラに応じて拡張可能なヘルパ クラスです。

タグ ライブラリ のプログラミングで使用するクラスとインターフェイスの詳細については、JRun の docs ディレクトリ内にある javadocs を参照してください。

JSP 開発者のカスタム タグの使用法

次の例に示すように、JSP 開発者がタグ ライブラリを有効にするには、taglib ディレクティブを使用し、prefix:tagname の変換を使用してカスタム タグをコーディングします。

```
<%@ taglib prefix="test" uri="DocSamples.tld" %>
<%-- この例では、開始タグと終了タグを組み合わせます。--%>
<test:hello/>
```

カスタム タグ内では本文テキストを使用できますが、TLD ファイル内で本文テキストを禁止することができます。タグ ハンドラでは本文テキストと対話できます。これについては、265 ページの「本文コンテンツとの対話」で説明します。

スクリプト変数と変換時属性の検証を有効にできます。これについては、271 ページの「TEI クラスのコーディング」で説明します。

単純なタグ ハンドラのコーディング

単純なタグ ハンドラは、親元の TagSupport または BodyTagSupport の doStartTag メソッド、または必要な場合には doEndTag メソッドを書き換えます。本文テキストとは対話しません。本文テキストとの対話がないタグ ハンドラでは、TagSupport クラスを拡張する必要があります。TagSupport を拡張すると、doStartTag メソッドと doEndTag メソッドで、定数として定義されている次の戻り値が使用されることに注意してください。

- doStartTag は、次のいずれかの値を返します。
  - EVAL_BODY_INCLUDE 開始タグと終了タグの間の本文テキスト (JSP コードを含む) を受け入れます。ただし、doEndTag メソッドでは本文テキストを使用できないことに注意してください。本文テキストを評価するには、265 ページの「本文コンテンツとの対話」の説明に従って BodyTagSupport を拡張するクラスを作成します。
  - SKIP_BODY 本文テキストを無視します。開始タグと終了タグの間にあるテキストは評価せず、表示しません。
- doEndTag は、次のいずれかの値を返します。
  - EVAL_PAGE ページの評価を続行します。
  - SKIP_PAGE ページの残りを無視します。

次のタグ ハンドラは、doStartTag メソッドと doEndTag メソッドから HTML を出力します。JSP でタグ ハンドラを使用するには、TLD ファイルでタグ ハンドラを定義し、それを JSP から呼び出す必要があります。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class SimpleTag extends TagSupport {
  /**
  * Executes when the tag is started.
  */
public int doStartTag() throws JspException {
  try {
    pageContext.getOut().print("<h2>Hello from doStartTag()</h2>");
    // タグの本文でテキストを使用できるようにします。
    return EVAL_BODY_INCLUDE;
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}

/**
* Executes with the end tag.
*/
public int doEndTag() throws JspException {
  try {
    pageContext.getOut().print("<h2>Hello from doEndTag()</h2>");
    // ページの評価を継続します。
    return EVAL_PAGE;
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}
}
```

TLD ファイルの作成

TLD は、タグ ライブラリを記述する XML 形式のテキスト ファイルです。JRun では、taglib ディレクティブが含まれているページを解釈するために TLD ファイルが参照されます。taglib ディレクティブには、TLD ファイルを指示する uri 属性が含まれています。taglib ディレクティブの詳細については、276 ページの「タグ ライブラリのパッケージ化」を参照してください。

taglib 要素は、TLD ファイルのルート です。このファイルは、次の要素から 構成さ れています。

- tlibversion　タグ ライブラリのバージョン
- jspversion (オプション)　タグ ライブラリが必要とする JSP のバージョン
- shortname　既定のショート ネーム
- uri (オプション)　タグ ライブラリを一意に識別する URI
- info (オプション)　タグ ライブラリの使用情報
- tag　カスタム タグ情報。TLD ファイルは、1 つまたは複数のタグ要素と、外部ツールで使用される 1 つの id 属性を持つことができます。それぞれの要素には、次のサブ要素が含まれています。
  - name : カスタム タグ名
  - tagclass : タグ ハンドラのクラス名
  - teiclass : TEI ファイルのクラス名
  - bodycontent : 本文コンテンツ タイプを識別し ます。有効な値は、tagdependent (SQL ステート メントなどのタグ依存本文コンテンツ)、jsp (JSP および HTML 本文コンテンツ)、および empty (本文コンテンツ使用不可) です。empty を指定した場合、カスタム タグの本文は空です。
  - info (オプション) : カスタム タグ使用情報
  - attribute (オプション) : 属性情報。tag 要素は、attribute 要素を持たない場合と、1 つ以上の attribute 要素を持つ場合があり ます。

TLD ファイルおよび TLD の形式の詳細については、JavaServer Pages バージョン 1.1 の仕様書を参照してください。

TLD ファイルでのタグの定義

カスタム タグは、TLD ファイル内で tag 要素とそのサブ要素によって定義し ます。tag 要素の主な目的は、JSP ファイルで使用されているカスタム タグ名をタグ ハンドラのクラス ファイルに関連付けることです。カスタム タグでスクリプト変数を作成する場合は、teiclass 要素を使用して、タグの TEI クラス ファイルを指定する必要もあり ます。

tag 要素を使用して、属性を定義することもできます。attribute 要素は必須ではありませんが、次のような場合に使用すると便利です。

- 属性が必須の場合は、attribute 要素で <required>true</required> を指定します。
- 属性を実行時スクリプトレット式によって計算できる場合は、attribute 要素で <rtexprvalue>true</rtexprvalue> を指定します。

カスタム タグ属性の詳細については、262 ページの「属性との対話」を参照してください。

TLD ファイルの例

次の TLD ファイルでは、258 ページの「単純なタグ ハンドラのコーディング」で示した SimpleTag クラスに対してタグを定義しています。

```
<?xml version="1.0" ?>

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>JRun Doc Samples</shortname>
  <tag>
    <name>hello</name>
    <tagclass>SimpleTag</tagclass>
    <bodycontent>JSP</bodycontent>
  </tag>
</taglib>
```

単純なカスタム タグの呼び出し

JSP 内でカスタム タグの使用を有効にするには、taglib ディレクティブを使用します。taglib ディレクティブでは、タグ名と一緒に使用して特定のタグを呼び出す接頭辞を指定します。

次の JSP コードでは、TLD ファイルの例で定義されている hello タグを呼び出しています。

```
<html>
<body>
<h1>Simple Custom Tag</h1>
<%@ taglib prefix="test" uri="DocSamples.tld" %>

<test:hello/>

</body>
</html>
```

属性との対話

属性を受け入れるタグ ハンドラをコーディングできます。属性の機能を有効にするには、タグ ハンドラで次のことを行う必要があります。

- 属性ごとにオブジェクト スコープ変数を定義する。
- setter メソッド、また必要な場合は getter メソッドを属性ごとに定義する。setter メソッドの名前は、`set` で始まり、その後に先頭が大文字の変数名を付けます。たとえば、変数名が `foo` であれば、`setFoo` メソッドとなります。

属性の使用法と動作は、次のようにカスタマイズできます。

- **TLD ファイル**　必須属性を指定する場合や、属性で JSP 実行時式を利用する場合は、TLD ファイル内で属性を定義します。詳細については、264 ページの「TLD ファイルでの属性の定義」を参照してください。
- **TEI クラス**　属性をスクリプト変数として使用する場合や、`isValid` メソッドを書き換えて検証を有効にする場合は、TEI クラス内で属性を定義します。詳細については、271 ページの「TEI クラスのコーディング」を参照してください。

TLD ファイルの指定と TEI クラス ファイルには、強力な検証機能があります。ただし、必要な機能は、Bean に類似した setter メソッドでクラス スコープ変数と対話することだけです。たとえば、次のタグ属性の使用法を有効にするとします。

```
...
<test:hello username="Joe"/>
...
```

タグ ハンドラでは、次のコードを実装する必要があります。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestParms extends BodyTagSupport
{
  // 属性と同じ名前を持つオブジェクトスコープの変数です。
 String username;

  // JSP コンパイラによって呼び出される setVariablename メソッドです。
public void setUsername(String username) {
 this.username = username;
}

// 最適な getter メソッドです。
public String getUsername() {
 return username;
}
...
```

タグ ハンドラ全体を示す次の例では、JSP で提供されている include アクション要素を属性を使用してエミュレートしています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import javax.servlet.*;
import java.io.IOException;

public class TestInclude extends TagSupport {
  // 既定値がないので必須です。
  String page;
  // 既定値は true です。
  String flush = "true";

  // setter メソッドです。
public void setPage(String page) {
  this.page = page;
}
public void setFlush(String flush) {
  // 小文字で保存します。
  this.flush = flush.toLowerCase();
}

public int doStartTag() throws JspException {
  // 本文テキストを無視します。
  return SKIP_BODY;
}

  // doEndTag がすべての作業を行います。
public int doEndTag() throws JspException {
  try {
    ServletContext sc = pageContext.getServletContext();
    RequestDispatcher rd = sc.getRequestDispatcher(page);
    if (rd !=null) {
      // アクセス要求および応答
      ServletRequest request = pageContext.getRequest();
      ServletResponse response = pageContext.getResponse();
      // 必要な場合は、バッファを一括消去します。
      if (flush.equals("true")) {
        pageContext.getOut().flush();
      }
      // ファイルをインクルードします。
      rd.include(request, response);
      return EVAL_PAGE;
    }
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
  catch (Exception e) {
    pageContext.getServletContext().log("Error with " + page, e);
  }
  return EVAL_PAGE;
}
}
```

TLD ファイルでの属性の定義

TLD ファイル内で属性を定義するには、attribute 要素を使用します。この要素には、次のサブ要素があります。

- name 属性名
- required 属性が必須かどうかを示します。このサブ要素は、true または false に設定します。
- rtexprvalue この属性の値に関する実行時式をカスタム タグで使用できるかどうかを示します。このサブ要素は、true または false に設定します。

次の TLD エントリの例では、必須属性とオプション属性を設定しています。

```
<?xml version="1.0" ?>

<taglib>
  <tlibversion>0.0</tlibversion>
  <jspversion>1.0</jspversion>
  <shortname>test</shortname>
  <tag>
    <name>include</name>
    <tagclass>TestInclude</tagclass>
    <bodycontent>empty</bodycontent>
  <attribute>
   <name>page</name>
   <required>true</required>
   <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
   <name>flush</name>
   <required>false</required>
   <rtexprvalue>false</rtexprvalue>
  </attribute>
  </tag>
  <tag>
    <name>hello</name>
    <tagclass>HelloTag</tagclass>
    <teiclass>HelloTEI</teiclass>
    <bodycontent>JSP</bodycontent>
  </tag>
</taglib>
```

JSP での属性のコーディング

次の例に示すように、属性はカスタム タグの一部としてコーディングします。

```
<html>
<body>

<%@ taglib prefix="test" uri="test.tld" %>
<h1>Testing Custom Tags with Parameters</h1>

<test:include page="/includedText.htm" flush="true" />

</body>
</html>
```

本文コンテンツとの対話

Tag インターフェイスと BodyTag インターフェイスのどちらを使用しても、テンプレート テキスト、JSP スクリプト要素、およびネストしたカスタム タグをカスタム タグの本文に組み込むことができます。カスタム タグで本文コンテンツと対話しない場合は、TagSupport クラスを拡張し、doStartTag メソッドで EVAL_BODY_INCLUDE を返します。一方、カスタム タグで本文コンテンツの処理、ループ、または変更が必要な場合は、doInitBody メソッドと doAfterBody メソッドがある BodyTagSupport クラスを拡張します。

メモ
TLD ファイル内でカスタム タグの bodycontent 要素に empty を指定すると、本文コンテンツを無効にできます。タグ ハンドラでは、doStartTag メソッドで SKIP_BODY を返すことによって、本文コンテンツを無視できます。

BodyContent オブジェクトは、JspWriter のサブクラスです (JspWriter は、JSP の out 変数のために内部的に使用されるライターです)。BodyContent オブジェクトは、doInitBody、doAfterBody、およびdoEndTag で bodyContent 変数によって使用できます。BodyContent オブジェクトと bodyContent 変数は、大文字の使用法が違う点に注意してください。オブジェクトの内容は、doEndTag メソッドで元の JspWriter と統合できます。BodyContent オブジェクトには、出力を書き出すために使用するメソッドのほかに、このオブジェクトの内容の読み取り、クリア、および取り出しを行うメソッドが含まれています。たとえば、bodyContent.getString を使用すると、ライターの内容を取り出せるほか、必要があれば、その内容を元の JspWriter と統合する前に変更できます。

メモ
bodyContent 変数を使用する前に、ヌルでないことを確認してください。

---

doAfterBody メソッドが EVAL_BODY_TAG を返した場合、JRun は本文の先頭に戻って再実行します。これは、一覧やデータベースの結果セットなどの反復データ全体を繰り返し処理する上で強力な機能です。

単純な例

次の例は、doInitBodyメソッドと doAfterBodyメソッドの簡単な使用法を示したものです。bodyContent の出力を出力ストリームに統合する方法も示されています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestBody extends BodyTagSupport {

public int doStartTag() throws JspException {
  try {
    pageContext.getOut().print("<h2>We're in doStartTag()</h2>");
    return EVAL_BODY_TAG;
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}

public void doInitBody() throws JspException {
 try {
  // これは、doStartTag や doEndTag のライターとは違うことに
  // 注意してください。
  bodyContent.print("<h2>We're in doInitBody()</h2>");
 }
 catch(IOException ioe) {
  throw new JspException(ioe.getMessage());
 }
}

public int doAfterBody() throws JspException {
 try {
  // これは、doStartTag や doEndTag のライターとは違うことに
  // 注意してください。
  bodyContent.print("<h2>We're in doAfterBody()</h2>");
  // return EVAL_BODY_TAG;  // これを使用してループします。
  return SKIP_BODY;
 }
 catch(IOException ioe) {
   throw new JspException(ioe.getMessage());
 }
}
```

```
public int doEndTag() throws JspException {
  try {
  // bodyContent ライターから元のライターに書き込みます。
  pageContext.getOut().print(bodyContent.getString());
  //// バッファが大きい場合は、前の行よりも
  //// 次のコードの方がより効率的です。
  //// 元の（囲んでいる）ライターを取得します。
  // JspWriter jOut = bodyContent.getEnclosingWriter();
  //// 前のライターに本文の出力を追加します。
  //bodyContent.writeOut(jOut);

    // ここで元のライターに戻ります。
    pageContext.getOut().print("<h2>We're in doEndTag()</h2>");
    return EVAL_PAGE;
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}
}
```

ループの例

doAfterBody が EVAL_BODY_TAG を返すようにコーディングすることによって、カスタム タグの本文を繰り返し実行するループを作成できます。

メモ
カスタム タグとループで使用されるスクリプト変数のスコープは、TEI クラスによって制御できます。詳細については、271 ページの「TEI クラスのコーディング」を参照してください。

次の例では、Enumeration タイプの属性を受け入れて、Enumeration オブジェクト内のそれぞれの名前と値 (この例では HTTP ヘッダ) をループによって処理します。

```
import java.util.Enumeration;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestBodyLoopHeaders extends BodyTagSupport {

  Enumeration thisEnum;

public void setThisEnum(Enumeration passedEnum) {
  this.thisEnum = passedEnum;
}

public int doStartTag() throws JspException {
  return EVAL_BODY_TAG;
}
```

```
public void doInitBody() throws JspException {
  if (thisEnum.hasMoreElements()) {
    pageContext.setAttribute("nextElement", thisEnum.nextElement());
  }
}

public int doAfterBody() throws JspException {
  if (thisEnum.hasMoreElements()) {
    pageContext.setAttribute("nextElement", thisEnum.nextElement());
    return EVAL_BODY_TAG;  // ループ
  }else {
    return SKIP_BODY;
  }
}

public int doEndTag() throws JspException {
  try {
    // bodyContent ライターから元のライターに書き込みます。
    pageContext.getOut().print(bodyContent.getString());
    return EVAL_PAGE;
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}
}
```

次の JSP の例では、上記のタグ ハンドラを呼び出しています。

```
<html>
<body>

<%@ taglib prefix="test" uri="DocSamples.tld" %>
<h1>Looping through Headers</h1>
<table border="1">
<tr>
  <th>Name</th>
  <th>Value</th>
</tr>
<test:enumloop thisEnum="<%= request.getHeaderNames() %>">
  <tr>
    <% String header =
         (String)pageContext.getAttribute("nextElement");%>
  <td><%= header %></td>
  <td><%= request.getHeader(header) %></td>
  </tr>
</test:enumloop>
</table>

</body>
</html>
```

ネストしたタグ ハンドラのコーディング

Tag インターフェイスと BodyTag インターフェイスのどちらを実装しているかにかかわらず、カスタム タグをネストできます。タグをネストさせると、ネストしたタグに関するタグ ハンドラでは、findAncestorWithClass メソッドによって親クラスへの参照を取得できます。ネストしたタグ ハンドラでは、この参照を親クラスにタイプ変換することによって、親クラス内のメソッドを呼び出せます。たとえば、親クラスに実装されているメソッドを利用して、ネストしたタグ ハンドラで出力ストリームを書き出せます。

次のコードは、親タグに関するサンプル タグ ハンドラを示したものです。このサンプルには、ネストしたタグに関するタグ ハンドラで出力ストリームを更新するときに呼び出せるメソッドが含まれています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestBodyParent extends TagSupport
{
  String name;

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

public int doStartTag() throws JspException {
  try {
    // 親の名前を表示することから始めます。
    pageContext.getOut().print("<h1>Parent:" + name + "</h1>");
    return EVAL_BODY_INCLUDE;
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}

public int doEndTag() throws JspException {
  try {
    // グループの後にルーラーを追加します。
    pageContext.getOut().print("<hr>");
    return EVAL_PAGE;
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}

public void setNestedName(String name) throws JspException {
  try {
    // ネストされている名前を表示します。
    pageContext.getOut().print("<p>Nested:" + name);
  }
```

```
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}
}
```

次のコードは、ネストしたタグに関するサンプル タグ ハンドラを示したものです。このサンプルでは、親のタグ ハンドラのメソッドを呼び出して、出力ストリームを更新しています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestBodyNest extends TagSupport {
  private String name;
  private TestBodyParent parent = null;
public void setName(String name) {
  this.name = name;
}

public int doStartTag() throws JspException {
  // 親への参照を検索して保存します。
  Tag t = findAncestorWithClass(this, TestBodyParent.class);
  if (t == null) {
    throw new JspException("TestBodyNest must be in TestBodyParent.");
  }else {
    parent = (TestBodyParent)t;
    return EVAL_BODY_INCLUDE;
  }
}

public int doEndTag() throws JspException {
  // 名前を親にコピーします。
  parent.setNestedName(name);
  return EVAL_PAGE;
}
}
```

次の JSP の例では、親タグとネストしたタグを使用しています。

```
<html>
<body>

<%@ taglib prefix="test" uri="DocSamples.tld" %>
<h1>Testing Nested Custom Tags</h1>

<test:bodyparent name="Johnson">
  <test:bodynest name="Lorna"/>
  <test:bodynest name="Gretchen"/>
  <test:bodynest name="Brian"/>
</test:bodyparent>

</body>
</html>
```

タグを使用したスクリプト変数の作成

カスタム タグでは、スクリプト変数を定義できます。スクリプト変数は、JSP 内のスクリプトレットやほかのカスタム タグで使用できます。

メモ
スクリプト 変数の定義ではタグ属性 (ID 属性など) を使用できますが、タグ属性とスクリプト変数には直接の関係はないことに注意してください。

TEI クラスのコーディング

JRun は、変換時に TEI クラスを使用してスクリプト変数を有効にするとともに、必要に応じて属性の検証を実行します。TEI ファイルは、TagExtraInfo クラスを拡張する Java クラスです。getVariableInfo メソッドには、次のシグネチャがあります。

```
public VariableInfo[] getVariableInfo(TagData tagData) { }
```

tagData パラメータには、属性の名前 / 値ペアが格納されています。この名前 / 値ペアを使用してスクリプト変数を定義できます。たとえば、useBean タグでは、id 属性を使用してスクリプト変数を作成しています。

getVariableInfo メソッドで、VariableInfo オブジェクトの配列を作成し、1 つのスクリプト変数につき 1 つの VariableInfo オブジェクトをその配列に格納します。VariableInfo オブジェクトのコンストラクタには、次のパラメータがあります。

- varName は、スクリプト変数名を指定する String 型パラメータです。
- className は、スクリプト変数のクラスを指定する String 型パラメータです。
- declare は、コンストラクタで新規の変数を定義するかどうかを指示する boolean 型パラメータです。
- scope は、スクリプト 変数のスコープを指定する int 型パラメータ です。このパラメータでは、次の表で示すように AT_BEGIN、NESTED、または AT_END を指定します。

同期化とは、ページ コンテキストからオブジェクトを取り出して、それをスクリプト変数に代入する操作です。次の表では、getVariableInfo メソッドで指定した変数に関するスコープ、使用法、および同期化について説明しています。

スコープ	JSP での使用法	JSP との同期化	タグ ハンドラによって設定およびリセットされる場所
AT_BEGIN	タグの本文内と、JSP の残りの部分	このスクリプト変数は、本文の反復のたびにリセットされます。	doInitBody と doAfterBody (BodyTag を実装している場合)。それ以外の場合は、doStartTag または doEndTag。
NESTED	タグの本文内	このスクリプト変数は、本文の反復のたびにリセットされます。	doInitBody と doAfterBody (BodyTag を実装している場合)。それ以外の場合は、doStartTag。
AT_END	JSP の残りの部分	この変数は、本文の実行が完了した後でリセットされます。	doEndTag の後

次の例では、3 つのスクリプト変数を定義する TEI クラスに関する Java コードを示します。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class HelloTEI extends TagExtraInfo
{
public VariableInfo[] getVariableInfo(TagData tagData) {
 VariableInfo[] vars = new VariableInfo[3];

 // ページの本文と残りの部分の中で使用できます。
 vars[0] = new VariableInfo("foo", "java.lang.String", false,
        variableInfo.AT_BEGIN);
 // ページの残りの部分のカスタム タグの後ろで使用できます。
 vars[1] = new VariableInfo("bar", "java.lang.String", true,
        VariableInfo.AT_END);
 // タグ本文でのみ使用できます。
 vars[2] = new VariableInfo("baz", "java.lang.String", true,
        VariableInfo.NESTED);
 return vars;
}
}
```

タグ ハンドラでのスクリプト 変数の有効化

スクリプト変数を pageContext オブジェクトに追加するのは、タグ ハンドラの役割です。タグ ハンドラでは、スクリプト変数に指定されているスコープに応じて、これらの変数をさまざまな方法で定義します。さらに、本文テキストをループ処理するタグ ハンドラでは、doAfterBody メソッドによって、スクリプト変数を更新またはリセットできます。

次の例は、タグ ハンドラでスクリプト変数を設定する方法を示したものです。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class HelloTag extends BodyTagSupport {
  String to;

public void setTo(String to) {
  this.to = to;
}

public int doStartTag() throws JspException {
  try {
    pageContext.getOut().print("Hello " + to);

    // TEI ファイルによってこのスクリプト変数が
    // AT_BEGIN として定義されるので、ここで定義します。
    // それを doInitBody 内で定義し (ループの場合)、
    // doAfterBody 内で変更またはリセットできます。
  pageContext.setAttribute("foo", "foo");
    return EVAL_BODY_TAG;
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}

public void doInitBody() throws JspException {
  // TEI ファイルによってこのスクリプト変数が
  // NESTED として定義されるので、ここで定義します。
  // それを doStartTag 内で定義し (ループの場合)、
  // doAfterBody 内で変更またはリセットできます。
 pageContext.setAttribute("baz", "baz");
}

public int doEndTag() throws JspException {
  try {
    // このタグ ハンドラによって BodyTag が実装されるので
    // (BodyTagSupport を展開することによって)、
    //  本文のライターと元のライターを統合する必要があります。
    pageContext.getOut().print(bodyContent.getString());

    // TEI ファイルによってこのスクリプト変数が
    // AT_END として定義されるので、
    // doEndTag まで自分で定義する必要はありません。
  pageContext.setAttribute("bar", "bar");
    return EVAL_PAGE;
  }
  catch(IOException ioe) {
    throw new JspException(ioe.getMessage());
  }
}
}
```

JSP でのスクリプト 変数の使用法

次の JSP の例では、前出の TEI クラスおよびタグ ハンドラで定義したスクリプト変数を使用しています。

```
<%@ taglib prefix="test" uri="test.tld" %>
<% String foo; %>

<test:hello to="World">
  <%-- baz は NESTED です (本文でのみ使用可能)。--%>
  <%= baz %>
  <%-- foo は AT_BEGIN です (本文とそれ以外でも使用可能)。--%>
  <%= foo %>
</test:hello>
<%= foo %>
<%-- bar は AT_END です (本文の後ろでのみ使用可能)。--%>
<%= bar %>
```

isValid メソッド の使用法

TEI クラスでは、isValid メソッドを書き換えてタグ特有の属性の検証を実施できます。変換時には、JRun から isValid メソッドに TagData インスタンスが渡されます。

isValid メソッドでは、TagData.getAttribute を呼び出して、この値にアクセスできます。TLD ファイル内の属性の定義で実行時式を有効にしている場合は、TagData.REQUEST_TIME_VALUE オブジェクトをチェックできます。このオブジェクトは、カスタム タグの呼び出しが実行時式を使用していることと、検証が不可能であることを示します。実行時式の詳細については、264 ページの「TLD ファイルでの属性の定義」を参照してください。

次の例の TEI ファイルでは、バージョン属性の値が 5 より小さいことを確認しています。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class TestIsValidTEI extends TagExtraInfo
{
  // この例では、スクリプト変数が 1 つのみであると想定しています
  // (この例とは無関係に)。
public VariableInfo[] getVariableInfo(TagData tagData) {
  VariableInfo[] vars = new VariableInfo[1];
  vars[0] = new VariableInfo("foo", "java.lang.String", true,
        VariableInfo.AT_BEGIN);
  return vars;
}

public boolean isValid(TagData data) {
 Object version = data.getAttribute("version");
 // この属性により、実行時式が使用できるので、
 // REQUEST_TIME_VALUE をチェックする必要があります。
```

```
 if (version != null && version != TagData.REQUEST_TIME_VALUE) {
  int iVersion = Integer.parseInt((String)version);
  // バージョンは 5 以下になります。
  if (iVersion > 5) {
   return false;
  }else {
   return true;
  }
 }else {
  return false;
 }
}
}
```

JSP でのタグの使用法

JSP 開発者は、次のように taglib ディレクティブでタグ ライブラリの名前を指定することによってタグ ライブラリを有効にします。

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

この例では、TLD ファイル DocSamples.tld で記述されているタグを有効にしています。このファイルは WEB-INF ディレクトリ内にあります。このライブラリ内のタグを呼び出すには、次の例に示すように <prefix:tagname> の構文を使用します。

```
<test:helloworld/>
```

次の例に示すように、タグは属性を持つことがあります。

```
<test:hellowithparm name="Joe"/>
```

タグは、本文コンテンツを受け入れることができます。次の例に示すように、本文コンテンツには、JSP 構文やその他のカスタム タグが含まれている場合があります。

```
<test:bodyparent name="This Family">
  <test:bodynest name="Lorna"/>
  <test:bodynest name="Gretchen"/>
  <test:bodynest name="Brian"/>
</test:bodyparent>
```

JSP ファイルでのカスタム タグの使用法の詳細については、この章の前の部分にある JSP の例を参照してください。

タグ ライブラリのパッケージ化

アプリケーション開発者は、タグ ライブラリを JSP 開発者に引き渡します。公開可能なタグ ライブラリでは、タグ ハンドラ、TEI クラス、およびその他のサポート クラスを JAR ファイルに組み込む必要があります。この JAR ファイルは、Web アプリケーションの `WEB-INF/lib` ディレクトリに配置する必要があります。

メモ

JAR ファイルを別の場所に格納し、Web アプリケーションの仮想マッピングをその場所に確立することもできます。この場合は、複数の Web アプリケーションで 1 つのタグ ライブラリを共有できます。ただし、Web アプリケーションを公開用にパッケージ化するときは、その JAR ファイルを `WEB-INF/lib` に移動する必要があります。詳細については、『JRun セットアップ ガイド』を参照してください。

パッケージには、JAR ファイルおよび TLD ファイルのほかに、次の情報を記述した文書が含まれます。

- それぞれのタグの名前
- 必須属性とオプション属性
- どの属性で実行時式を使用できるか
- 本文コンテンツを使用できるタグと、使用できないタグの指示
- 必須の親タグ、ネストしたタグ、または本文コンテンツ

TLD ファイルは、JavaServer Pages バージョン 1.1 の仕様書で推奨されている場所である `META-INF/taglib.tld` 内の JAR ファイルへの格納を検討してください。これは、オーサリング ツールの統合のために推奨されている格納場所です。この JAR ファイルに TLD ファイルを格納しない場合は、ファイルの内容を直接見られないように `WEB-INF` ディレクトリの下に TLD ファイルを格納する必要があります。

JSP 開発者が TLD ファイルを参照する方法は、TLD ファイルを `web.xml` で定義しているかどうかによって、次のように異なります。

- `web.xml` に `taglib` 要素が含まれている場合は、`web.xml` ファイル内の `taglib-uri` 要素に対応する `taglib uri` 属性を指定します。
- `web.xml` に `taglib` 要素が含まれていない場合は、TLD ファイルを直接参照する `taglib uri` 属性をコーディングします。

web.xml ファイル内で taglib 要素を定義するには

1 開始を示す taglib 要素をコーディングします。

```
<taglib>
```

2 taglib-uri 要素をコーディングします。この要素では、JSP 開発者が JSP taglib ディレクティブで使用している名前を指定します。

```
<taglib-uri>/eisTaglib</taglib-uri>
```

3 taglib-location 要素をコーディングします。この要素では、TLD ファイルへのパスを指定します。

```
<taglib-location>/WEB-INF/tldHome/eisTaglib.tld</taglib-location>
```

4 終了を示す taglib 要素をコーディングします。

```
</taglib>
```

JSP 開発者が、この例のタグ ライブラリを参照するときは、次の taglib ディレクティブを使用します。

```
<%@ taglib prefix="eis" uri="/eisTagLib" %>
```

第 23 章

サーブレット API の変更点

この章では、サーブレット API の 2.0 から 2.2 への変更点について説明します。

目次

- サーブレット API の 2.0 から 2.1 への変更点 280
- サーブレット API の 2.1 から 2.2 への変更点 283

サーブレット API の 2.0 から 2.1 への変更点

サーブレット API のバージョン 2.1 では、API のいくつかが改良され、機能が拡張されました。

API の改良

サーブレット API バージョン 2.1 は、前のバージョンに次の改良を加えています。

- 「log メソッドにおける変更点」
- 「ServletRequest.getRealPath の廃止」
- 「URL の大文字 / 小文字の一貫性」
- 「init メソッドの書き換え」
- 「New getSession メソッド」
- 「ステータス コードの設定」

このセクションでは、これらの改良について説明します。

log メソッドにおける変更点

`ServletContext.log(Exception e, String msg)` が廃止され、代わりに `ServletContext.log(String message, Throwable t)` が使用されます。

このほかに、サーブレット API 2.1 には `GenericServlet.log(String message, Throwable t)` メソッドが含まれます (前のバージョンの `GenericServlet.log(String message` も使用できます)。

ServletRequest.getRealPath の廃止

`ServletRequest.getRealPath(String path)` が廃止されました。このメソッドへの参照を `ServletContext.getRealPath(String path)` に置換する必要があります。

URL の大文字/小文字の一貫性

サーブレット API 2.1 は、大文字の使用に一貫性を与えるために、次のメソッドについて `Url` を `URL` に変更しました。

- `HttpServletRequest.isRequestedSessionIDFromURL`
- `HttpServletResponse.encodeURL`
- `HttpServletResponse.encodeRedirectURL`

init メソッドの書き換え

`init(ServletConfig config)` を書き換えるとき、`super.init(config)` の呼び出しをコーディングすることによって、`GenericServlet` が常に `config` への参照を保存できるようにしておく必要があります。サーブレット API 2.1 には `init()` メソッド (引数なし) が含まれています。これは `super.init(config)` の呼び出しを必要としません。

New getSession メソッド

サーブレット API 2.1 には HttpServletRequest.getSession() メソッドが含まれています。このメソッドは HttpServletRequest.getSession(true) の代わりに使用できます。

ステータス コード の設定

HttpServletResponse.setStatus(int sc, String sm) が廃止されました。代わりに HttpServletResponse.setStatus(int sc) と HttpServletResponse.sendError(int sc, String msg) を使用します。

既定の getParameter の動作

サーブレット API 2.1 では､複数の値を持つパラメータについて getParameter(String name) を呼び出すと、常に最初の値が返されます。前のバージョンでは、この動作はサーバーによって異なっていました。

サーブレット 参照にアクセスできない

サーブレット API 2.1 では、ServletContext.getServlet(String name) と ServletContext.getServletNames() が廃止されました。

ほかのセッショ ンにアクセスできない

サーブレット API 2.1 では、HttpSession.getSessionContext() が廃止されました。

機能の拡張

サーブレット API 2.1 には、次の機能拡張が含まれています。

- 「要求の送信」
- 「リソースのアクセス」
- 「ネストされた例外」
- 「ServletContext による属性の共有」
- 「セッション タイムアウトの制御」
- 「バージョン情報のアクセス」

このセクションで、これらの機能拡張について説明します。

要求の送信

RequestDispatcher インターフェイスを使用して､処理をほかのサーブレットに転送するか、ほかのサーブレットの出力を呼び出し側のサーブレットの出力に含めることができます。

リソースのアクセス

ServletContext.getResource(String uripath) メソッドを使用して URL オブジェクトを返し、このオブジェクトを使用して実際のリソースにアクセスできます。

ネスト された例外

ServletException 例外をルート例外のラッパーとして使用して、その下の例外を可視化できます。

ServletContext による属性の共有

ServletContext オブジェクトは次のメソッドを追加します。これらのメソッドを使用してサーブレット間で属性を共有できます。

- setAttribute(String name, Object object)
- getAttribute(String name)
- getAttributeNames()
- removeAttribute(String name)

セッション タイムアウト の制御

HttpSession.setMaxInactiveInterval(int interval) メソッドによってセッションの持続時間を制御できます。

バージョン 情報のアクセス

ServletContext.getMajorVersion() および ServletContext.getMinorVersion() メソッドを使用してサーブレット API のバージョンにアクセスできます。

サーブレット API の 2.1 から 2.2 への変更点

サーブレット API バージョン 2.2 には、API のいくつかの改良 (その主な機能拡張の 1 つは Web アプリケーション) およびその他の機能拡張が含まれています。

API の改良

サーブレット API バージョン 2.2 は、前のバージョンに次の改良を加えています。

- 「要求の送信の機能拡張」
- 「転送機能の拡張」

要求の送信の機能拡張

サーブレット API 2.2 には `ServletContext.getNamedDispatcher(String path)` メソッドが含まれています。これを使用して、コンポーネントをその登録されている名前に基づいて送信できます。

また、`ServletRequest.getRequestDispatcher(String path)` メソッドを使用することもできます。これは、ターゲットとして相対 URL を使用します (`ServletContext.getRequestDispatcher(String path)` は完全修飾 URL を使用します)。

転送機能の拡張

`HttpServletResponse.sendRedirect(String url)` は相対 URL をサポートします。

Web アプリケーション

Web アプリケーションは、サーブレット、JSP ドキュメント、HTML ドキュメント、イメージ、およびその他のリソースから構成されています。これらのリソースをあらかじめ定義されているディレクトリ構造に従って公開し、サーブレットをサポートするどの Web サーバーにもそれを公開できるようにします。

2 つのアプリケーションがデータ レポジトリとして共通のデータベースを使用してデータを共有できます。これによって 2 つのアプリケーションが同じ情報にアクセスできます。たとえば、電子商取引 Web サイトが、共通のデータベースを使用する複数のアプリケーションから構成されているとします。顧客はログイン名、パスワード、またはその他の形式の識別子によって識別され、それによって各アプリケーションは、ショッピング カート、支払い情報、住所など、すべてのアプリケーションに共通するユーザ情報にアクセスできます。

アプリケーション間でデータを共有するもう 1 つの手段は、JRun による EJB のサポートです。EJB の使用の詳細については、第 31 章 を参照してください。

1 つの JRun JVM で複数の Web アプリケーションをサポートできます。Web アプリケーションを開発するときに検討しなければならない問題の 1 つは、アプリケーションの間の境界をどこに設定するかです。言い換えれば、アプリケーションをほかのアプリケーションと同じ JRun JVM 内に置くことができるか、別の JVM 内に置く必要があるかという問題です。

それを決定する 1 つの要因はエラー処理です。同じ JRun JVM 内のすべての Web アプリケーションは同じプロセス内で実行するため、1 つのアプリケーションのエラーによって JVM 全体が停止します。このエラー処理結果が容認できない場合は、アプリケーションを別の JRun JVM に配置する必要があります。

また、それぞれの JRun JVM に、特定のアプリケーションによって要求される JVM 固有の属性を割り当てることができます。たとえば、JRun JVM ごとに異なる Java JVM を使用できるため、JVM のタイプを使用してアプリケーションを区別できます。

最後に、アプリケーションのセキュリティは JRun JVM レベルで決定されます。2 つのアプリケーションに別々のセキュリティ管理システムが必要な場合は、それらのアプリケーションを別の JRun JVM に配置する必要があります。

Web アプリケーションの詳細については、Java サーブレット API バージョン 2.2 の仕様書を参照してください。

WAR ファイル

WAR (Web application archive) について説明します。

- WAR ファイルは、Web アプリケーションの公開に使用します。公開すると、WAR ファイルはディレクトリに展開されます。
- 少なくとも次の要素を含むディレクトリ構造です。
  - アプリケーションのルート ディレクトリ
  - `Approot/WEB-INF` ディレクトリ
  - `Approot/WEB-INF/web.xml` ファイル
  - `Approot/WEB-INF/classes` ディレクトリ
  - `Approot/WEB-INF/lib` ディレクトリ

WEB-INF ディレクトリ

`WEB-INF` ディレクトリには `web.xml` ファイル、`lib` ディレクトリ、および `classes` ディレクトリが含まれています。

- `web.xml` ファイル (公開記述子とも言います) には、Web アプリケーションに関する情報が含まれています。この情報には、アプリケーションの init パラメータ、サーブレットの init パラメータ、サーブレットのマッピング、MIME タイプのマッピング、およびセキュリティ情報が含まれます。
- `classes` ディレクトリには、サーブレットの `.class` ファイルが含まれています。
- `lib` ディレクトリには Web アプリケーション内のサーブレットが使用する JAR ファイルとタグ ライブラリが含まれています。

アプリケーションあたり 1 つの ServletContext

Web アプリケーション内のすべてのサーブレットは、1 つの `ServletContext` を共有します。Web アプリケーションの初期パラメータにアクセスするには、`ServletContext.getInitParameter(String name)` および `ServletContext.getInitParameterNames()` メソッドを使用します。

ほかの拡張機能

サーブレット API のバージョン 2.2 には、上記のほかに次の拡張機能が含まれています。

- 「応答バッファ」
- 「複数のヘッダ値のサポート」
- 「一時ディレクトリのサポート」
- 「サーブレット名へのアクセス」
- 「国際化」
- 「セキュリティ」

このセクションで、これらの拡張機能について説明します。

応答バッファ

応答バッファによって、サーブレットがその応答をバッファに入れるかどうかを制御できます。また、バッファのサイズを制御することもできます。`ServletResponse` オブジェクトには応答バッファをサポートするために次のメソッドが含まれています。

- `setBufferSize(int size)`
- `getBufferSize()`
- `isCommitted()`
- `reset()`
- `flushBuffer()`

複数のヘッダ値のサポート

`HttpServletRequest` オブジェクトには、複数の値を持つヘッダを取得するために `getHeaders(String name)` メソッドが含まれています。これは `Accept-Language` ヘッダによって複数言語のサポートを実装するときに特に便利です。このヘッダは複数のヘッダ値を処理できます。

また、`HttpServletRequest.addHeader(String name, String value)`、`HttpServletRequest.addIntHeader(String name, int value)`、および `HttpServletRequest.addDateHeader(String name, long date)` を使用して複数の値を設定することもできます。

一時ディレクトリのサポート

サーブレット API 2.2 では、それぞれのサーブレット コンテキストが専用の一時作業ディレクトリを提供する必要があります。サーブレット コンテキストの一時ディレクトリの位置を決めるときは、ServletContext 属性 javax.servlet.context.tempdir にアクセスします。この属性を持つオブジェクトのタイプは java.io.File です。次のコードと同様のコードを使用します。

```
...
ServletContext sc = this.getServletContext();
File tempDir = (File) sc.getAttribute("javax.servlet.context.tempdir");
...
```

サーブレット名へのアクセス

GenericServlet.getServletName() メソッドを使用してサーブレット登録名にアクセスできます。サーブレットが登録されていない場合、このメソッドはサーブレットのクラス名を返します。

国際化

ServletRequest.getLocale() メソッドを使用してクライアントの Locale オブジェクトにアクセスできます。返されるロケールは、Accept-Language ヘッダに基づきます。このヘッダがない場合、このメソッドはサーバーの既定のロケールを返します。また、ServletRequest.getLocales() メソッドを使用して、許容可能なロケールを指定している Locale オブジェクトの Enumeration にアクセスすることもできます。

HttpServletResponse.setLocale(Locale loc) メソッドを使用してロケールを指定することもできます。

セキュリティ

Web アプリケーションにロールベースのセキュリティ情報を含めることができます。これは、指定したページへのアクセスが特定のロールのユーザにのみ許可されるようにします。

サーブレット API 2.2 には HttpServletRequest.getUserPrincipal() および HttpServletRequest.isUserInRole(String role) メソッドが含まれています。また、HttpServletRequest.isSecure() メソッドも含まれています。このメソッドは、要求が HTTPS を使用している場合に true を返します。

Web アプリケーションでのロールおよびセキュリティの詳細については、Java サーブレット API バージョン 2.2 の仕様書を参照してください。

第 4 部

Enterprise JavaBeans の開発

ここでは、JRun を使用して Enterprise JavaBeans を作成する方法について説明します。

EJB のディレクトリ 289
プロパティ 295
リソース管理 303
Bean の開発 311
Bean 管理パーシスタンス 319
コンテナ管理パーシスタンス 329
Java でのメッセージング 345
EJB クライアントの コーディング 377
高度なテクニック 383
EJB エンジンの使用 393

第 24 章

EJB のディレクトリ

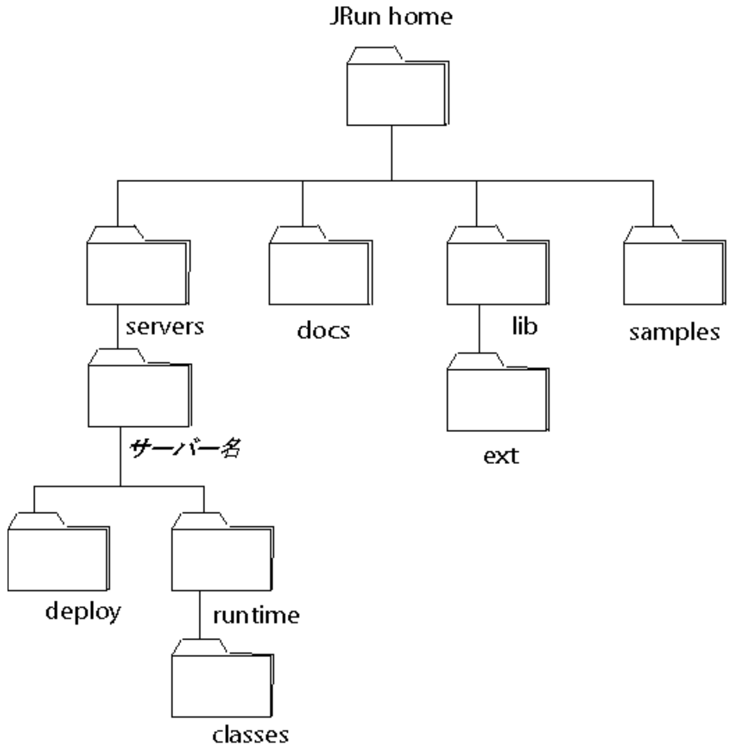
この章では、JRun ディレクトリ構造の、EJB に関係した部分について説明します。JRun ディレクトリ構造の詳細については、『JRun セットアップ ガイド』を参照してください。

目次

- 構造 290

構造

次の図は、JRun を完全にインストールした直後の JRun ディレクトリ構造の、EJB に関係した部分を示します。



この章では、これらのディレクトリのそれぞれについて、図に示した順序で説明します。

JRun home

JRun home は、JRun をインストールしたディレクトリを表します。本書では、JRUN_HOME 変数は絶対パス名を表しています。たとえば、JRun を /opt/jrun にインストールした場合、JRUN_HOME は /opt/jrun になります。JRun を C:¥Program Files¥Allaire¥JRun にインストールした場合は、JRUN_HOME は C:¥Program Files¥Allaire¥JRun になります。

次の表は、JRUN_HOME ディレクトリにインストールされるファイルを示します。

ファイル	説明
overview.htm	JRun 製品の構成情報です。
relnotes.htm	JRun リリース ノートが含まれています。
whatsNew.htm	JRun の現リリースで提供される新機能の要約です。

servers/ *サーバー名* /deploy

deploy ディレクトリは、Bean の公開時に Deploy ツールによって使用されます。Deploy ツールは、Bean 実装、ホーム インターフェイスとリモート インターフェイス、および公開記述子が含まれる JAR ファイルを使用して、ホーム オブジェクトとリモート オブジェクトの実装を作成します。生成された実装とスタブをコンパイルした後、Deploy ツールは結果のクラス ファイルを ejipt_objects.jar ファイルと ejipt_exports.jar ファイルに追加します。Deploy ツールは、runtime.properties ファイルも作成します。409 ページの「Enterprise JavaBeans の公開」を参照してください。起動時に、JRun は deploy ディレクトリ の内容を runtime ディレクトリ に コピーします。

docs

このディレクトリには、JavaDocs、HTML 形式の文書、および PDF 形式の文書など、JRun に関する文書が含まれています。このディレクトリは、最小インストール オプションを使用した場合はインストールされません。

lib

lib ディレクトリには、必要な JAR ファイルとその他のプロパティ ファイルが含まれています。

ファイル	説明
ejipt.jar	サーバーに関連するすべてのクラスが含まれています。
ejipt_tools.jar	Deploy ツールおよび Server ツールのクラスが含まれています。
ejipt_client.jar	クライアントに関連するクラスおよびスタブが含まれています。このファイルは、すべてのクライアントにインストールする必要があります。
ejipt_jms_client.jar	クライアントに関連する JMS のクラスが含まれています。このファイルは、JMS サービスを使用するクライアントにインストールする必要があります。
ejipt_ejbeans.jar	既定の EJB の実装が含まれています。
default_exports.jar	各 ejipt_exports.jar に必要なクラスが含まれています。Deploy ツールによって内部で使用されます。
default_objects.jar	各 ejipt_objects.jar に必要なクラスが含まれています。Deploy ツールによって内部で使用されます。
jrun_ejbeans.jar	単一のサインオンで使用する EJB およびその他のファイルが含まれています。
jrun_exports.jar	単一のサインオンで使用する EJB およびその他のファイルが含まれています。
jrun_objects.jar	公開前の EJB が含まれています。メッセージングやその他の目的で内部で使用されます。
jrun.policy	VM 内のリソースへのアクセスを許可するセキュリティポリシー
ejipt.properties	既定のプロパティ構成が含まれています。295 ページの第 25 章「プロパティ」を参照してください。
/ext	lib ディレクトリには、次に説明する /ext サブディレクトリも含まれています。

lib/ext

ext ディレクトリには、追加の JAR ファイルが含まれます。これらは、Sun JDK の標準拡張機能です。これらのファイルは、便宜上このディレクトリに提供されていますが、Sun から直接入手することもできます。

ファイル	説明
activation.jar	アクティブ化のフレームワーク
ejb.jar	Enterprise JavaBeans API (1.1)
iiop.jar	RMI/IIOP サポートで使用するファイル
jdbc.jar	JDBC API (2.0)
jms.jar	Java Message Service API (1.0)
jndi.jar	Java Naming & Directory Service API (1.2)
jta.jar	Java Transaction API (1.0)
jaxp.jar	Java API for XML parsing (1.0)。このファイルはサーブレット エンジンで使用します。
servlet.jar	Java サーブレット API (2.2)。このファイルはサーブレット エンジンで使用します。

servers/ *サーバー名*/runtime

JRun は runtime ディレクトリを使用して起動時に Bean およびプロパティをロードします。EJB エンジンは、このディレクトリを管理します。通常、ユーザがこのディレクトリのファイルを修正することはお勧めしません。公開された Bean JAR ファイルのほかにも、JRun は起動時に次のファイルを deploy ディレクトリから runtime ディレクトリにコピーします。

ファイル名	説明
ejipt_exports.jar	このファイルには、オブジェクトのスタブが保持されているほか、必要に応じてクライアントにエクスポートされるクラスも追加されています。このファイルは、Deploy ツールによって作成されます。
ejipt_objects.jar	リモート インターフェイスおよび ホーム インターフェイスの実装 (EJB オブジェクト)。このファイルは、Deploy ツールによって作成されます。
runtime.properties	このファイルは、Deploy ツールによって作成されます。

起動時に、deploy ディレクトリの内容が前回の実行時から変更されているかどうかが判断されます。変更されている場合は、JRun はタイム スタンプをチェックし、その時点よりも新しいファイルを runtime ディレクトリにコピーします。

これらのファイル以外に、通常は少なくとも 1 つの *yourbeans*_ejb.jar ファイルが runtime ディレクトリにあります。

起動時に次のファイルが検出されない場合は、これらのファイルも JRun によって作成されます。

ファイル名	説明
ejipt.cache	このファイルは、サーバーによってセッション Bean インスタンスのキャッシュに使用されます。このファイルは、JRun が起動されるたびに消去されます。
instance.store	既定のオブジェクト ストアです。このファイルは、Bean を持続するために使用します。このファイルの内容は自動的に消去されません。

servers/ *サーバー名*/runtime/classes

classes ディレクトリには、load コマンドを発行する際に EJB エンジンによって動的にロードされる Bean クラスが含まれています。ダイナミック Bean ローディングは、テスト および開発の目的でのみ使用してください。ダイナミック Bean ローディングを使用した後は、更新されたクラスを必ず EJB の JAR ファイルに組み込んでください。

メモ
ダイナミック Bean ローディングは、スタンドアロンの EJB エンジンの機能です。

このディレクトリは、Bean 実装クラス専用として使用します。EJB エンジンは、ホーム インターフェイスおよびリモート インターフェイスのダイナミック ローディングは実行しません。詳細については、422 ページの「Bean のダイナミック ローディングの使用」を参照してください。

samples

samples ディレクトリには、サンプル EJB アプリケーションの .java ファイル、make ファイル、および公開記述子ファイルが含まれています。これらのサンプルには、エンティティ Bean とセッション Bean の両方が含まれます。このディレクトリは、最小インストール オプションを使用した場合はインストールされません。提供されるサンプルの詳細については、『JRun サンプル ガイド』を参照してください。

第 25 章

プロパティ

この章では、プロパティを設定する方法、公開記述子の要素、および EJB エンジンがプロパティを適用するタイミングを決定する方法について説明します。

目次

- 概要 296
- サーバー プロパティの設定 296
- コンテナ プロパティの設定 297
- Bean 情報の指定 297
- 例 298

概要

EJB エンジンは標準の `java.util.Properties` によって実装されるプロパティを使用して、その公開環境と実行時環境を設定します。Bean 情報に関しては、EJB エンジンは主に、公開記述子を介して渡された情報をサポートします。

プロパティは、実行時にコンテナおよび公開済み Bean の両方からアクセスされ、サーバー、コンテナまたは公開済み Bean のレベルで書き換えられます。プロパティファイルの作成および保守に任意のテキスト エディタを使用できます。これらのファイルは単純な `key=value` 構文で設定します。設定可能なプロパティのリストについては、JRun JavaDocs ファイルに添付されている API マニュアルの `EjiptProperties` を参照してください。

この章に記載されているメカニズムを使用すると、すべての公開済み Bean がすべてのシステム プロパティ、`ejipt.properties`、コマンド ライン プロパティ、および `local.properties` に読み取り専用でアクセスできるようになります。また、公開済み Bean には、最初に公開記述子で設定した固有のプロパティへの読み取り / 書き込みアクセス権があります。下位レベルのプロパティへのアクセスは常に、上位レベルのプロパティによって書き換えられます。

メモ
JRun 3.0 では、主に `deploy.properties` ファイルおよび Bean プロパティ ファイルでプロパティを設定します。これらの方法は JRun 3.1 でも引き続きサポートされていますが、`deploy.properties` 仕様を `local.properties` に移動し、Bean プロパティを 1 つの公開記述子に移動する方法をお勧めします。

プロパティは、さまざまな方法で設定できます。プロパティ ファイルを直接修正してプロパティを特定の値に永久的に設定することもできます。また、スタンドアロン EJB エンジンのプロパティは、コマンド ライン上で設定できます。公開済み Bean によって実行時にプロパティを設定することもできます。

サーバー プロパティの設定

起動時には、JRun により、まずサーバー プロパティ リストが作成されます。JRun は次の順に複数のソースからプロパティをロードします。

- `ejipt.properties`
- **システム環境** (`java -D` コマンドライン スイッチを介して渡される)
- `global.properties`
- `local.properties`
- `runtime.properties` (Deploy ツールで作成)

JRunを統合J2EEアプリケーション サーバーとして実行する場合は、ejipt.properties から global.properties に、また deploy.properties から local.properties に設定を変更しなければなりません。ファイルを ejipt.properties または deploy.properties から global.properties または local.properties に移動するときは、次の例に示すように ejb 接頭辞を使用します。

```
# No ejb prefix when used in deploy.properties:
# ejipt.userHomeName=default.UserHome
# In local.properties, use the ejb prefix:
ejb.ejipt.userHomeName=default.UserHome
```

スタンドアロン モードで EJB エンジンを実行している場合は、ejipt.properties および deploy.properties にプロパティを引き続き配置する必要があります。たとえば、スタンドアロン モードで実行する EJB のサンプルでは deploy.properties ファイルを使用します。

開発の際に runtime.properties ファイルを直接編集しないでください。その後で Deploy ツールを実行すると、その変更が失われてしまいます。

コンテナ プロパティの設定

EJB エンジンでは、runtime.properties ファイルの ejipt.ejbJars プロパティにリストされている各 JAR ファイルのコンテナを作成します。ejipt.ejbJars プロパティがない場合、deploy ディレクトリ内の ejipt_objects.jar、ejipt_exports.jar および extra_exports.jar ファイルを除くすべての JAR ファイルについてコンテナが作成されます。

EJB エンジンのすべてのコンテナに固有のプロパティ リストがあり、その既定値はサーバー プロパティ リストです。

Bean 情報の指定

Bean 情報は、プロパティ ファイルまたは公開記述子を使用して定義できます。通常、公開記述子は ejb-jar.xml という名前が付けられ、JAR ファイルの META-INF ディレクトリに保存される必要があります。JAR ファイル内にそのようなファイルが検出されると、EJB エンジンは Bean プロパティ ファイルまたは既定プロパティ ファイルではなく記述子を使用します。ejb-jar.xml ファイルがない場合、EJB エンジンは Bean レベルのプロパティ ファイルおよび既定プロパティ ファイルを使用します。XML ベースの公開記述子で必要な要素の詳細については、EJB バージョン 1.1 仕様を参照してください。

メモ
本書の後半では公開記述子の使用法について説明します。ただし、公開記述子で行った設定は Bean プロパティ ファイルでも行うことができます。Bean プロパティ ファイルの詳細については、『JRun 拡張設定ガイド』を参照してください。

公開記述子を定義する場合、次の例のように、env-entry、env-entry-name、および env-entry-value 要素を使用して EJB エンジンに固有のプロパティを含めます。

```
<env-entry>
  <env-entry-name>propertyname</env-entry-name>
  <env-entry-value>propertyvalue</env-entry-value>
</env_entry>
```

EJB エンジン固有のプロパティの先頭には、通常 ejipt が付いています。env-entry 要素の例については、JRUN_HOME/samples/sample2a/Meta-inf/ejb-jar.xml を参照してください。

内部で、EJB エンジンはすべてをプロパティとして保存します。公開済み Bean をロードするときに、EJB エンジンは公開済み Bean ごとに内部プロパティ リストを作成します。このプロパティ リストには、java:comp/env JNDI コンテキストを通じてアクセスできます。既定値は Bean のコンテナのプロパティ リストです。このメカニズムによって、それぞれの公開済み Bean の間ですべてのシステム / サーバー / コンテナ プロパティを読み取り専用で共有できます。

例

次の例は、プロパティを設定するためのさまざまな方法を示しています。この最初の例では、コマンド ラインによってプロパティを書き換えます。2 番目の例では、公開および既定プロパティ ファイルでプロパティを設定します。3 番目の例では、公開済み Bean のレベルでプロパティを設定変更します。4 番目の例では、コマンド ラインから Bean レベルのプロパティを設定変更します。

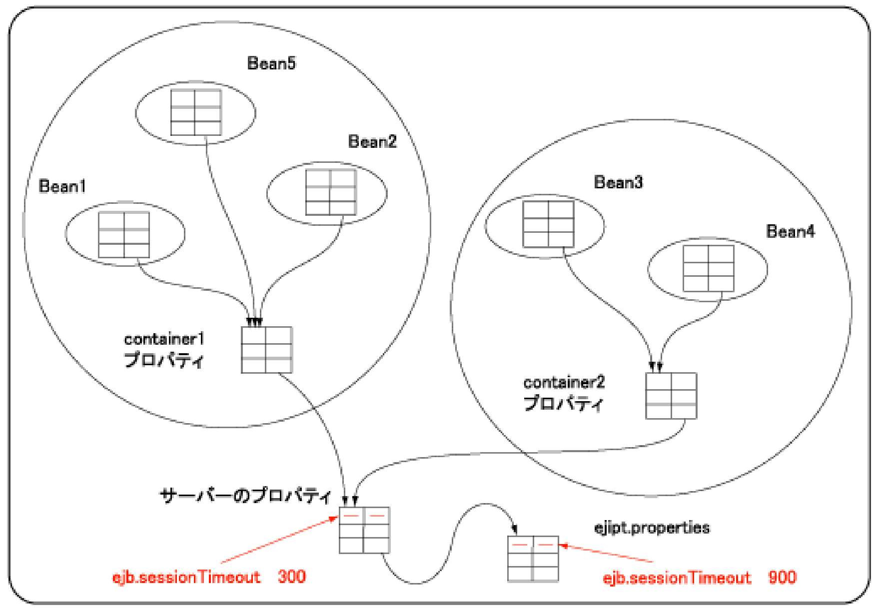
コマンド ラインによる書き換え

この例では、ejb.sessionTimeout プロパティを書き換えます。この例では、JRUN_HOME/lib の ejipt.properties ファイルで ejb.sessionTimeout が 900 (秒) に設定されていると想定します。この例では、EJB エンジンが起動したときにコマンド ラインで -Dejb.sessionTimeout=300 とパラメータを設定することによって、実行時にこの値を 300 に変更します。

```
> cd JRUN_HOME
> java -Dejb.sessionTimeout=300 -classpath lib/ejipt.jar
  allaire.ejipt.Ejipt
```

また、統合 J2EE サーバーの実行時に global.properties の ejb.javaargs プロパティまたは local.properties ファイルでコマンド ライン引数を指定することによって、それらの引数を使用できます。

このコマンド ラインによって、サーバー プロパティ リストに ejb.sessionTimeout プロパティとその値 300 が追加され、次の図に示すように ejipt.properties ファイルから取得した値 900 を書き換えます。



公開済み Bean から EJBContext.getEnvironment().getProperty("ejb.sessionTimeout") 呼び出すと、プロパティ リスト チェーン内でこの値が最初に検出されるため、値 300 が返されます。

プロパティ ファイルによる書き換え

さらに、この例では container1 の ejb.sessionTimeout を書き換え、default.properties ファイルの値を 300 に設定します。また、deploy.properties を使用してサーバー セッション タイムアウトを設定します。この例では、ejipt.properties ファイルで ejb.sessionTimeout が 900 (秒) に設定されていると想定します。

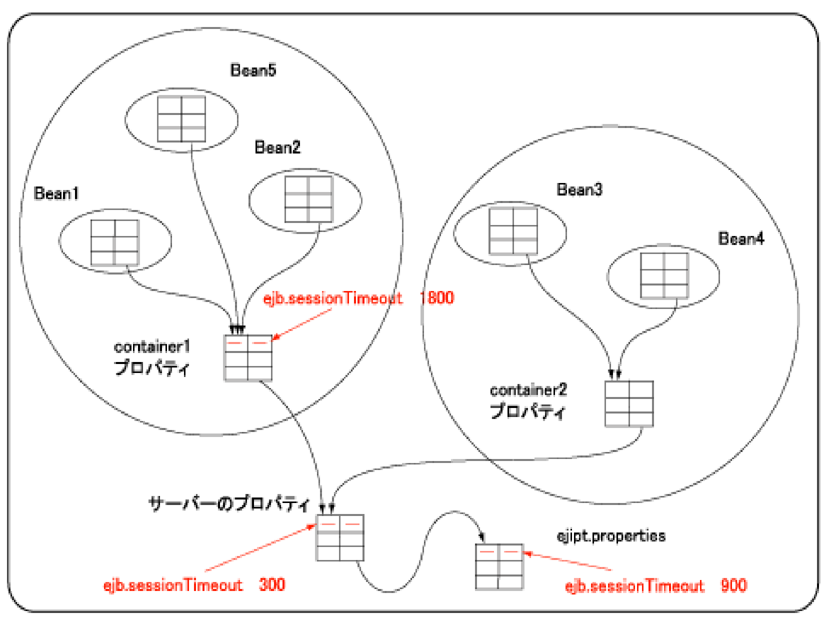
container1 の default.properties に次のプロパティが設定されています。

ejb.sessionTimeout=1800

deploy.properties ファイルには次のプロパティが設定されています。

ejb.sessionTimeout=300

次の図は、結果として作成されたプロパティ リストを示しています。

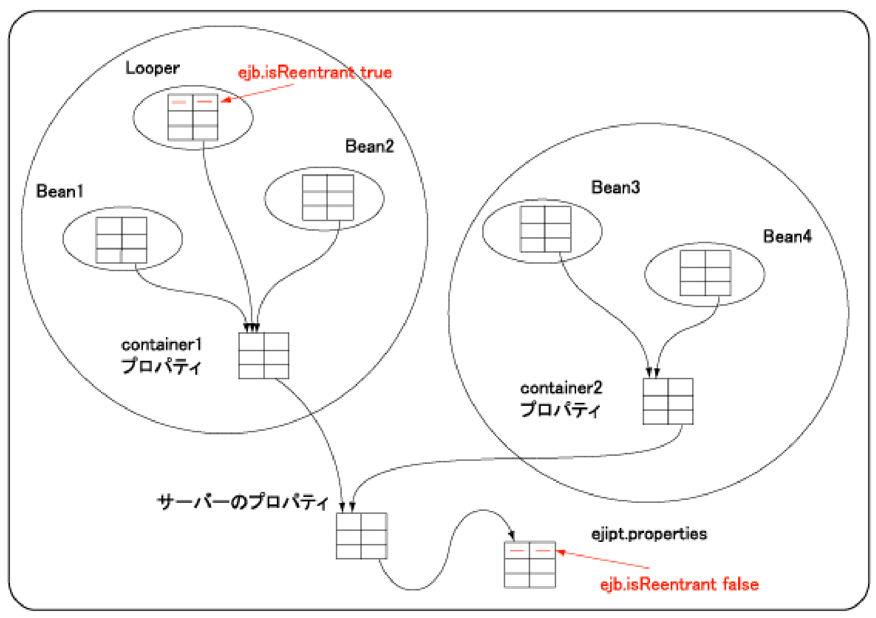


これにより、container1 内にあるすべての Bean の ejb.sessionTimeout 値は 1800 に設定され、それ以外のコンテナ内にある公開済み Bean の ejb.sessionTimeout 値は 300 に設定されます。

Bean プロパティによる書き換え

この例では、公開済み Bean のレベルでプロパティを書き換えます。`ejipt.properties` ファイルでプロパティ `ejb.isReentrant` が `false` に設定されていると想定します (注意 : この特定のプロパティでは、ステートフル セッション Bean とエンティティ Bean の既定の設定は `false` です)。また、`Looper` の名前を持つ公開済み Bean があり、その Looper が再入可能であると想定します。`Looper` でこの動作を妨げないように EJB エンジンに指示するには、`Looper` Bean の公開記述子内で `ejb.isReentrant` を `true` に設定する `env-entry` を確立します。

次の図は、結果として作成されたプロパティ リストを示しています。



`Looper` が公開されると、サーバーはそのプロパティ リストを作成します。プロパティ `ejb.isReentrant=true` は `Looper` のプロパティ リストに追加されます。これにより、`EJBContext.getEnvironment().getProperty("ejb.isReenterant")` を呼び出したとき `Looper` のすべてのインスタンスについて `true` が返され、それ以外の Bean には `false` が返されます。このメカニズムを使用して、プロパティ ファイルで定義されたか XML 記述子ファイルで定義されたかにかかわらず、プロパティを書き換えることができます。

実行時 Bean プロパティによる書き換え

この例では、スタンドアロン EJB エンジンから bean2 という名前の EJB の ejb.sessionTimeout を書き換えます。コマンド ラインから Bean のプロパティを書き換えるには、プロパティの前に Bean のホーム名を付けます。また、このステップでは Bean の公開記述子で設定された任意のプロパティを書き換えます。

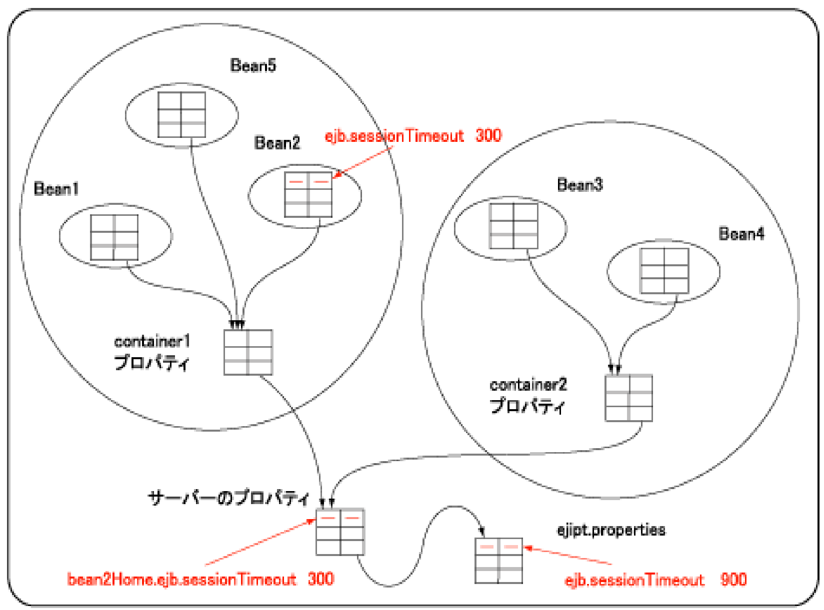
ejipt.properties ファイルで ejb.sessionTimeout が 900 (秒) に設定されており、bean2.properties ファイルで次のように設定されていると想定します。

```
ejb.sessionTimeout=1800
```

スタンドアロン EJB エンジンの実行時のタイムアウト値を 300 に設定するには、スタンドアロン EJB エンジンが起動したときにコマンド ラインで -Dbean2Home.ejb.sessionTimeout=300 パラメータを設定します。この場合、bean2 はホーム名が bean2Home で公開されると想定しています。

```
> cd JRUN_HOME
> java -Dbean2Home.ejb.sessionTimeout=300 -classpath
  lib/ejipt.jar allaire.ejipt.Ejipt
```

次の図は、結果として作成されたプロパティ リストを示しています。



bean2 を除くすべての Bean の ejb.sessionTimeout 値は 900 に設定され、bean2 の ejb.sessionTimeout 値は 300 に設定されます。

第 26 章

リソース管理

この章では、開発者が利用できる種々の EJB に関連するリソースについて説明します。

目次

- 概要 304
- ローカル ホーム オブジェクト 305
- インスタンス マネージャ 306
- データベース接続管理 308
- ローカル キャッシュ /ストア 308
- ロードされたユーザおよびロール 310

概要

JRun では EJB エンジンと対話し、そのリソースを管理するための一連の API が用意されています。`allaire.ejipt` パッケージに用意されているこれらの API のクラスは次のとおりです。

クラス名	API の使用法
`ResourceManager`	データ ベース接続の管理、ローカル ホーム オブジェクト の参照、自動呼び出し の管理を 行います。`ResourceManager` 管理のデータ ベース接続は、スタンド アロン EJB エンジンでのみ使用できることに注意し てく ださい。それ以外の場合は、『 JRun Version 3.1 機能および移行ガイド 』で説明し ているように JRun のデータ ソースを使用し てく ださい。
`InstanceManager`	公開済み Bean のインスタンスのステート についての情報を、関連するコンテキスト から 取得し 、インスタンスのステート の変化についてサーバーに知ら せます。
`StoreManager`	JDBC を 使用し ないパーシスタンスおよびキャッシュ メカニズムを 提供し ます。
`UserManager`	ユーザおよびサーバーでのロールを 管理し 、ユーザにサインイン機能を 提供し ます。

メモ
これらの機能は、アプリケーションを最適化するために必要に応じて実装できます。ただし、これら の機能を使用すると 、アプリ ケーショ ンをほかの J2EE アプリ ケーショ ンサーバーに移植できなくなり ます。

ローカル ホーム オブジェクト

1 つの Bean からローカルの Bean を呼び出す方法は 2 つあります (両方の Bean が同じサーバー上で動作している場合)。通常、呼び出し Bean にローカル JNDI コンテキスト (J2EE 準拠メソッド) を作成させるようにします。もう 1 つの方法は、呼び出し側の Bean が `ResourceManager.getLocalEJBHome(name)` を呼び出すようにする方法です。この場合、`name` はターゲット Bean のプロパティ ファイルで指定されている名前と同じです。

どちらの方法でも、ホーム オブジェクトへの参照が返されます。`getLocalEJBHome()` の呼び出しは単純なハッシュ検索なので、一般的には JNDI を使用する場合よりも速くなります。

メモ

`getLocalEJBHome` を呼び出す JRun 特有の機能およびコードは、ほかの J2EE アプリケーション サーバーに移植することはできません。

次の例では `ResourceManager.getLocalEJBHome` メソッドを呼び出します。

```
...
public void ejbActivate() throws RemoteException {
  _theWeb = ((WebHome)ResourceManager.getLocalEJBHome
    ("sample7c.WebHome")).findTheWeb();
  _loanHome = (LoanHome)ResourceManager.getLocalEJBHome
    ("sample7c.LoanHome");
    }
...
```

詳細については、JRun JavaDocs ファイルに添付されている API マニュアルの `ResourceManager` を参照してください。

インスタンス マネージャ

コンテキスト /Bean インスタンス プール

Bean コンテキストは、公開されている Bean の環境のステートに関する情報を検索するために使用します。コンテキストは Bean インスタンスが作成されたときに作成され、Bean インスタンスが存在する間 Bean に関連付けられ、ほかの Bean インスタンスが使用することはできません。コンテキストには、Bean インスタンスについて、インスタンスのステートが変化したかどうかを示す情報などが記録されています。

利用できるコンテキストの数は、公開記述子内で `ejipt.maxContexts`、`ejipt.maxFreeContexts`、および `ejipt.minFreeContexts` を `env-entries` として設定すること で管理できます。次の例では、これら のプロ パティ がサンプル 5a の公開記述子内でどのように設定されるのかを示しています。

```
...
<env-entry>
  <env-entry-name>ejipt.maxFreeContexts</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>5</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.maxContexts</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>5</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.minFreeContexts</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>0</env-entry-value>
</env-entry>
...
```

これら のプロ パティ の設定に関する 詳細については、JRun JavaDocs ファイルによって提供されている API マニュアルの `EjiptProperties` を参照してください。

`InstanceManager` クラスは、コンテキスト と 相互動作する API を供給します。Bean のインスタンスがサーバーによって作成された最初のインスタンスであるかどうかがわかれば非常に便利です。`InstanceManager.isFirst()` メソッドは、現在のインスタンスがその Bean の最初のインスタンスである場合、`true` を返します。それによって Bean の初期化を行うことができます。このメソッドは通常、`setEntityContext()` または `setSessionContext()` メソッド から 呼び出されます。EJB のサンプル 7c の次の例では、`InstanceManager.isFirst()` メソッドを呼び出します。

```
...
public void setEntityContext(EntityContext context) {
  super.setEntityContext(context);

  if (InstanceManager.isFirst()) {
    initBanks();
  }
}
...
```

InstanceManager.isLast() メソッドは、現在のインスタンスがその Bean の最後のインスタンスである場合、true を返します。これによって、Bean を削除する前にクリーンアップを実行できます。このメソッドは通常、unsetEntityContext() メソッドから呼び出されます。

インスタンスのステートの変化

InstanceManager.isDirty() メソッドは、インスタンスのステートの変化について EJB エンジンに知らせます。setDirty(true) を呼び出すと、ejbStore メソッドが呼び出され、強制的にインスタンスがコンテナに保存されます。setDirty(false) を呼び出すと、コンテナはステートの変化を無視し、データベース内のインスタンスを更新しません。

dirty フラグが設定されていない場合、コンテナはインスタンスを保存するかどうかの判断を試みます。これは、表面的な比較によって行われます。コンテナは == 演算子を使用してフィールドのステートとキャッシュされているステートの比較を繰り返し、ステートが変化したかどうかを判断します。深いレベルでの変化が起こった場合は、Bean メソッド内で dirty フラグを明示的に true に設定して、保存を強制しなければなりません。

メモ
isDirty と setDirty を使用するだけでなく、ejipt.isAlwaysDirty プロパティを true に設定できます。これによって各ビジネス メソッドに従って、コンテナが ejbStore を呼び出します。

詳細については、JRun JavaDocs ファイルによって提供されている API マニュアルの InstanceManager を参照してください。

データベース接続管理

データベース接続は、プールして再使用するリソースです。JRun には、セッション Bean と BMP エンティティによる次の 2 つのデータベース接続のオプションがあります。

- **JRun データ ソース**　JRun 内またはスタンドアロン EJB のもとで実行している EJB では、JNDI を利用して JMC で定義されたデータ ソースにアクセスできます。`java:comp/env/jdbc/`*`datasourcename`* 上で JNDI 検索を行い、`javax.sql.DataSource` に値を返します。このオブジェクトを使用して接続します。`connection.close()` を呼び出すときに、JRun によってプールに接続が返されます。JRun データ ソースはトランザクション処理されないため、XA に準拠しません。
- **XA 準拠データベース接続**　スタンドアロン EJB エンジンは `deploy.properties` ファイルで定義されたデータ ソースにアクセスできます。`ResourceManager.getConnection("`*`datasourcename`*`")` メソッドを呼び出して `java.sql.Connection` を取得します。`connection.close()` を呼び出すときに、JRun によってプールに接続が返されます。EJB スタンドアロン データ ソースはトランザクション処理されるため、XA に準拠します。リモート クライアントからこのタイプの接続にアクセスする場合は、サーバーで `remotelyAccessible` プロパティが true に設定されている必要があります。詳細については、『JRun Version 3.1 機能および移行ガイド』を参照してください。

コンテナ管理パーシスタンスを使用すると、コンテナはデータ ストア接続のすべての管理を処理します。

詳細については、API マニュアルの `ResourceManager`、319 ページの「Bean 管理パーシスタンス」、329 ページの「コンテナ管理パーシスタンス」、および JRun サンプル アプリケーションを参照してください。

ローカル キャッシュ /ストア

アプリケーションに比較的簡単なデータ アクセスの要求があり、リレーショナル データベースを使用できない場合は、`StoreManager` を使用してファイルベースのソリューションを実装します。

`StoreManager` を使用すれば、JDBC を使用しなくてもデータまたはエンティティ Bean をファイル ベースのストアに保持できます。プライマリ キーを使用するとインスタンスを保存し、後で取得できます。ストアをバックアップする既定の `instance.store` ファイルは `runtime` ディレクトリに作成されます。

JRun instance.store

JRun には、instance.store というファイルベースの拡張可能なトランザクション処理パーシスタンス メカニズムが用意されています。リレーショナル データベースの使用が望ましくない場合やリレーショナル データベースが使用できない場合は、このメカニズムを使用してエンティティ Bean を保持します。

Store インターフェイスは、公開済み Bean によるインスタンスのステートの保存に使用できる JDBC を使用しないパーシスタント ストアを表します。このインターフェイスでは、サーバーが使用するパーシスタンス メカニズムのカスタマイズをサポートします。

配信されたものとして instance.store を使用したり、Store インターフェイスを使用して instance.store を拡張できます。DefaultStore 実装をサブクラス化するか、または Store インターフェイスを異なる方法で実装すると、JRun で instance.store メカニズムをカスタマイズできます。

instance.store にストアされているすべてのオブジェクトは 1 つのキーに関連付けられています。既定の実装では、キーは関連付けられたオブジェクトのプライマリ キーのことです。プライマリ キーを使用して、ストアからインスタンスのステートを更新、取得、または削除できます。

Store の既存の実装の詳細については、JRun JavaDocs ファイルに付属する API マニュアルの DefaultStore を参照してください。

プロパティ

JRun には、instance.store を管理するプロパティが 2 つあります。

- ejipt.storeName プロパティを使用してストアの名前を変更したり、ストアの複数のインスタンスを作成できます。JRun では、Bean ごとに固有のストアを持つことができます。ejb-jar.xml の特定の Bean に ejipt.storeName env-entry を設定すると、その Bean は固有のデータ ストアを持つようになります。個々の Bean にこのプロパティを指定すると、実行時ディレクトリ内のストアはその Bean の持続インスタンスを含むストアの名前を持つようになります。local.properties または deploy.properties ファイルでこのプロパティを指定して、コンテナ内で公開されるすべての Bean によって使用されるコンテナにオブジェクト ストアの名前を設定できます。指定されていない場合は、プロパティ の既定の設定は instance.store です。
- ejipt.storeClassName は Store インターフェイスを実装するカスタム クラスを指定するためのプロパティです。カスタム クラスを使用して Store の動作をカスタマイズします。カスタム ストアを実装しているときに、getStore() メソッドを実装することを確認してください。ejipt.storeClassName プロパティの既定の設定は allaire.ejipt.DefaultStore です。

ロードされたユーザおよびロール

`UserManager` を使用すると、公開済み Bean からサーバー内のユーザおよびロールを管理できます。EJB エンジンは、起動時にユーザおよびロールをあらかじめロードしておくことによって、クライアントがサービスの要求を開始したときにそれが利用できるようにします。

`UserManager` は、ユーザとロールの両方が、エンティティ Bean によって固有のプライマリ キーを使用して表されると想定します。`UserManager` は、ユーザを `java.security.Principal` オブジェクトとして追加し、ロールを `java.security.acl.Group` オブジェクトとして追加します。`UserManager` によって、プライマリ キーと `java.security.Principal` および `java.security.acl.Group` オブジェクト間のマッピングを管理します。また、`UserManager` によってログイン セッションを制御することもできます。

`UserManager` クラスを使用する例が『JRun サンプル ガイド』にあります。また、各サンプルの `deploy.properties` ファイルを参照すると、ユーザおよびロールについての知識を得ることができます。詳細については、JRun JavaDocs ファイルに付属する API マニュアルの `UserManager` を参照してください。

第 27 章

Bean の開発

この章では、エンティティ Bean およびセッション Bean の開発プロセスについて説明します。

目次

- 概要 312
- Bean の リモート インターフェイスの作成 312
- Bean のホーム インターフェイスの作成 313
- Bean クラス実装の作成 314
- バージョン管理 317

概要

JRun は、エンティティとセッションの両方の Bean を完全にサポートしています。このため、ビジネスのニーズを満たす強力なソリューションを開発し、公開できます。

JRun では、Bean の開発の際に特定の IDE を必要としません。Java 2 プラットフォームをサポートしているものであれば、任意の IDE を使用できます。

Bean クラス実装、リモート インターフェイス、およびホーム インターフェイスを提供する必要があります。EJB には、この 3 つの要素がすべて必要です。これらの要素では、通常、次の命名規則が使用されます。

- **リモート インターフェイス**　*Beanname* (Balance など)
- **ホーム インターフェイス**　*Beanname*Home (BalanceHome など)
- **クラス実装**　*Beanname*Bean (BalanceBean など)

次の節では、エンティティ Bean とセッション Bean の開発に関するガイドラインおよび例を示します。

Bean の リモート インターフェイスの作成

`javax.ejb.EJBObject` を拡張する Bean のリモート インターフェイスを提供する必要があります。このインターフェイスで定義されるメソッドは、クライアントが呼び出せる唯一のビジネス メソッドです。したがって、リモート インターフェイスで定義される各メソッドに一致するメソッドが、Bean のクラス実装内にあります。引数と戻り値は、有効な RMI データ型であることが必要です。

次のサンプルは、リモート インターフェイスの必要条件を示しています。

```
package ejbeans;

import java.rmi.*;
import javax.ejb.*;

public interface Balance extends EJBObject {
  /* All methods here must have matching methods in BalanceBean.java */
  void save(int value) throws RemoteException;
  void spend(int value) throws RemoteException;
}
```

`throws` 句で定義される例外は、Bean クラス実装で定義される例外に一致し、`RemoteException` を含んでいる必要があります。

メモ
このインターフェイスの実際の実装 (リモート オブジェクト) は、Deploy ツールによって生成されます。

Bean のホーム インターフェイスの作成

さらに、javax.ejb.EJBHome を拡張する Bean のホーム インターフェイスを提供する必要があります。このインターフェイスで定義されるメソッドは、クライアントがリモート オブジェクトへの参照を確立するために使用します。

ホーム インターフェイスは、次のように定義する必要があります。

- **ステートレス セッション Bean**　引数のない 1 つの create メソッド
- **ステートフル セッション Bean**　任意の数の引数のある 1 つまたは複数の create メソッド
- **エンティティ Bean**　任意の数の引数のあるゼロまたはそれ以上の create メソッド

それぞれの create メソッドには、Bean のクラス実装の引数と同じ引数のある、一致する ejbCreate メソッドが 1 つ必要です。引数はどれも有効な RMI データ タイプで、戻り値は次のようになる必要があります。

- **セッション Bean**　無効
- **エンティティ Bean**　Bean のリモート インターフェイス タイプ

エンティティ Bean には、少なくとも 1 つの finder メソッドが必要です。ただし、セッション Bean の場合、finder メソッドは無効です。finder メソッドの引数は、有効な RMI データ型であることが必要です。また、その戻り値は Bean のリモート インターフェイス タイプを単一に、コレクション、または列挙で返す必要があります。

次のサンプル インターフェイスは、エンティティ ホーム インターフェイスの必要条件を示しています。

```
package ejbeans;

import java.rmi.*;
import javax.ejb.*;

public interface BalanceHome extends EJBHome {
  /* All methods here must have matching methods in BalanceBean.java */
  Balance create(int accountId) throws CreateException,
        RemoteException;
  Balance findByPrimaryKey(Integer key) throws FinderException,
        RemoteException;
}
```

throws 句で定義される例外は Bean クラスで定義される例外に一致し、RemoteException が含まれている必要があります。create メソッドの場合、例外として CreateException も含まれている必要があります。

Deploy ツールは自動的にホーム インターフェイスを生成します。

ホーム インターフェイスの詳細については、EJB の例を参照してください。

Bean クラス実装の作成

エンティティ Bean の必要条件は、セッション Bean の場合とかなり異なるため、このセクションでは各タイプについて別々に説明します。各セクションには詳細な説明が含まれています。

エンティティ Bean

エンティティ Bean は、サーバーのシャットダウン中に存続させるオブジェクトを表します。パーシスタンスには、次の 2 つのタイプがあります。

- **Bean 管理パーシスタンス (BMP)** データベース アクセスおよびコールバック メソッドのステートメントの更新をコーディングすることによって、エンティティ Bean の実装がパーシスタンスを管理します。BMP の詳細については、319 ページの第 28 章「Bean 管理パーシスタンス」を参照してください。
- **コンテナ管理パーシスタンス (CMP)** 公開記述子に作成された仕様をコンテナが使用して、データベース アクセスを実行し、自動的にステートメントを更新します。CMP の詳細については、329 ページの第 29 章「コンテナ管理パーシスタンス」を参照してください。

エンティティ Bean のインスタンスを表現するデータは通常、リレーショナル データベースのテーブルの行に格納されています。このデータベースには、JDBC データストアからアクセスします。これらのテーブルは、複数のデータベースにまたがる場合もあります。

エンティティ Bean に関する一般的な必要条件は次のとおりです。

- `javax.ejb.EntityBean` を実装する必要があります。
- パブリック タイプです。
- 抽象タイプは使用できません。

また、実装が必要な定義済みシグネチャを使用する特殊なメソッドもあります。

次の import ステートメントは、エンティティ Bean クラス実装に必ず含めます。

```
import java.rmi.*;
import javax.ejb.*;
```

次のステートメントは、クラスがパブリック タイプで、`EntityBean` インターフェイスを実装するという必要条件を示しています。

```
public class BalanceBean implements EntityBean
```

エンティティ Bean のインスタンスは、関連するコンテキストを格納して、サーバー環境にアクセスできるようにする必要があります。ただし、このフィールドにパーシスタンスは与えません。

```
protected EntityContext _context;
```

setEntityContext メソッドは、必ず定義する必要があります。このメソッドは、Bean インスタンスが作成された後でコンテナによって呼び出され、関連するエンティティ コンテキストを設定します。

```
public void setEntityContext(final EntityContext context) {
    _context = context;
}
```

unsetEntityContext メソッドも定義する必要があります。このメソッドは、Bean インスタンスが削除される前にコンテナによって呼び出され、関連するエンティティ コンテキストを消去します。

```
public void unsetEntityContext() {
  _context = null;
}
```

エンティティ Bean はゼロまたはそれ以上の ejbCreate メソッドを持つことができ、それぞれのメソッドは、ホーム インターフェイスで定義された create メソッドに対応している必要があります。メソッドはパブリック タイプで、ホーム インターフェイスの create メソッドと同じ引数を使用し、プライマリ キー タイプを返す必要があります。BMP を使用するエンティティ Bean については通常、渡されるキーに対する行があるかチェックし、その行がある場合は DuplicateKeyException を返し、その行がない場合は作成するようなロジックをコーディングします。

ejbPostCreate メソッドは、エンティティ Bean のホーム インターフェイスに create メソッドが含まれる場合にのみ必要となります。メソッドはパブリック タイプで、ejbCreate メソッドと同じ引数を使用し、戻り値は void タイプです。ビジネス メソッドが呼び出される前に、ejbPostCreate メソッドを使用して、Bean インスタンスを初期化します。

エンティティ Bean 実装には、ejbFindByPrimaryKey メソッドも含まれている必要があります。このメソッドは、パブリック タイプでなければならず、必ずプライマリ キー タイプを返します。この Bean 実装には、ejbFindAllAccounts のような finder メソッドを追加して含めることができます。

エンティティ Bean は、EntityBean インターフェイスから次のメソッドも実装する必要があります。

- ejbRemove は、データベースからエンティティを削除するために使用します。
- ejbActivate は、Bean インスタンスがオブジェクトに割り当てられる (アクティブになる) ときにコンテナによって呼び出され、Bean インスタンスが準備状態のときに必要な追加リソースを取得できるようになります。ejbLoad がまだ呼び出されていないため、このメソッドにビジネス ロジックが作成されることはありません。
- ejbPassivate は、インスタンスがプールに返される直前にコンテナによって呼び出され、Bean インスタンスは、Bean がインスタンス プール内に存在する間に保持されないリソースを解放できるようになります。

- `ejbLoad` はコンテナによって呼び出され、基礎となるデータベースからインスタンスの状態をロードすることで、状態を同期化するようにインスタンスに指示します。コンテナは、ビジネス メソッドを呼び出す直前に `ejbLoad` メソッドを呼び出します。
- `ejbStore` はコンテナによって呼び出され、インスタンスの状態を基礎データベースに格納することで、状態を同期化するようにインスタンスに指示します。コンテナは、ビジネス メソッドを呼び出した直後に `ejbStore` メソッドを呼び出します。

これらのメソッドに使用するコードは、CMP と BMP では異なります。詳細については、319 ページの第 28 章「Bean 管理パーシスタンス」および329 ページの第 29 章「コンテナ管理パーシスタンス」を参照してください。

Bean のリモート インターフェイスで定義されるメソッドの場合、Bean 実装内にシグネチャが一致するメソッドが存在する必要があります。通常、これらのメソッドはビジネス ロジックを実装します。次の例は、`save` メソッドの実装を示します。

```
public void save(final int value) {
  _value += value;
}
```

セッション Bean

セッション Bean は通常、ビジネス オブジェクト ではなくビジネス ロジックを表します。セッション Bean はデータベースの読み取りと書き込みが可能ですが、Bean インスタンス自体は、サーバーのシャット ダウン後に保持されません。セッション Bean メソッドの代表的な使用法に、1 つまたは複数のエンティティ Bean メソッド の呼び出しをラップすることがあります。これらの連携がトランザクションを形成します。

セッション Bean には、次の 2 つのタイプがあります。

- **ステートレス** 各メソッド呼び出しの後で、すべてのステート情報を削除します。
- **ステートフル** メソッド呼び出しの間もステート情報を維持します。

すべてのセッション Bean には、`javax.ejb.SessionBean` インターフェイスの実装が必要です。

次の import は必ず含める必要があります。

```
import java.rmi.*;
import javax.ejb.*;
```

次のステートメントは、クラスがパブリック タイプで、`SessionBean` インターフェイスを実装する必要条件を示しています。

```
public class LoginSessionBean implements SessionBean
```

Bean インスタンスは、関連するコンテキストを格納して、サーバー環境にアクセスできるようにする必要があります。

```
protected SessionContext _context;
```

setSessionContext メソッドは、必ず定義する必要があります。このメソッドは、Bean インスタンスが作成され後にコンテナによって呼び出され、関連するセッションコンテキストを設定します。

```
public void setSessionContext(final SessionContext context) {
  _context = context;
}
```

ejbCreate メソッドは、ホーム インターフェイスに create メソッドが含まれる場合にのみ必要となります。このメソッドはパブリック タイプで、void タイプの値を返す必要があります。ステートレス セッション Bean の場合、ejbCreate メソッドに引数を使用することはできません。ステートフル セッション Bean にゼロまたはそれ以上の引数を持つ ejbCreate メソッドがいくつか含まれている場合があります。

セッション Bean は、SessionBean インターフェイスから次のメソッドも実装する必要があります。

- ejbRemove は、Bean を削除する直前にコンテナによって呼び出されます。
- ejbActivate は、Bean インスタンスがオブジェクトに割り当てられるときに、コンテナによって呼び出されます。
- ejbPassivate は、インスタンスがプールに返される直前に、コンテナによって呼び出されます。

バージョン管理

EJB エンジンは、Serializable オブジェクトの標準 Java バージョン管理をサポートしています。シリアライズの詳細については、JDK のマニュアルを参照してください。

第 28 章

Bean 管理パーシスタンス

この章では、EJB エンジンによって提供される BMP サポートについて説明します。

目次

- 概要 320
- データ ソースのプロパティ 320
- Bean の必須メソッド 321
- ビジネス メソッド 328

概要

Bean 管理パーシスタンス (BMP) は、持続するデータをきめ細かく制御するために利用します。Bean 管理パーシスタンスを使用すると、データベース ステートメントが、ejbStore や ejbLoad などの Bean 実装の必須メソッド内にハードコード化されます。コンテナ管理パーシスタンス (CMP) と比較すると、BMP ではデータベース スキーマに対して何らかの変更があった場合、Bean 自体を変更することがほとんどなので、データベース間の移植性は低くなります。ただし、J2EE サーバーの場合は異なるテクニックで CMP を実装するので、J2EE アプリケーション サーバー間の移植性は高くなります。

データ ソースのプロパティ

エンティティ Bean の各インスタンスは通常、リレーショナル データベースの行に対応します。つまり、BMP EJB の必須メソッドでは、そのメソッドに適切なデータベースアクセスを実行する必要があることを意味します。これらのデータベース操作の実行時に JRun データ ソースを使用できます。JRun データ ソースを使用すると、JDBC ドライバへの変更からコードが保護され、接続プールも提供されます。

次のスニペットは、JRun データ ソースのアクセスに使用するコードを示します。

```
...
try {
  // コンテキストのインスタンスを作成します。
  Context ctx = new InitialContext();
  // java:comp/env/jdbc/jdbc-datasource-name からデータ ソースを取得します。
  // JMC の JDBC データ ソース名を定義します。
  DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/source1");
  // データベース ロジックを続行します。
  // この例は ejbLoad メソッドからの引用です。
  Connection connection = ds.getConnection();
  try {
    Statement statement = connection.createStatement();
    // _context は EntityContex を保持する Bean レベルの変数です。
    // SetEntityContext メソッドによって設定されます。
    ResultSet results =
      statement.executeQuery("SELECT value FROM account WHERE id = " +
        _context.getPrimaryKey());
    results.next();
    // _value は データベースから取得された
    // 「value」列を保持する Bean レベルの変数です。
    _value = results.getInt(1);
```

```
    results.close();
    statement.close();
  }
  finally {
    connection.close();
  }
}
catch (Exception e) {
  throw new EJBException(e);
}
...
```

メモ
EJB エンジンには、JRun データ ソースの使用に代わるプロパティファイル ベースのメソッドが含まれています。スタンドアロン モードで EJB エンジンを実行する場合、代わりにこのメソッドを使用できます。詳細については、『JRun 拡張設定ガイド』を参照してください。

Bean の必須メソッド

BMP では、さまざまなパーシスタンス機能ポイント (create、load、store など) でコンテナから `EntityBean` インターフェイスの所定のメソッドが呼び出されます。クラス実装には、次のメソッドによるパーシスタンス ロジックが含まれていることが必要です。

- `ejbCreate` および `ejbPostCreate`
- `ejbLoad`
- `ejbStore`
- `ejbRemove`

さらに、エンティティ Bean によって、`ejbFindByPrimaryKey` メソッドがサポートされている必要があります。

後続のセクションでは、BMP を利用する際のクラス実装内の関連コードについて詳しく説明します。

ejbCreate および ejbPostCreate

ejbCreate メソッド は、オブジェクト を 1) 作成できるかどう か 2) 作成する 必要があるかどうかを判別するために使用します。次の ejbCreate の例は、EJB サンプル 2a の BalanceBean からのものです。

```
...
public Integer ejbCreate(int accountId) throws CreateException {
  _value = 0;
  Connection connection = null;

  try {
    Context context = new InitialContext();
    DataSource source =
      (DataSource)context.lookup("java:comp/env/jdbc/source1");
    connection = source.getConnection();

    try {
      // 最初に ID が存在するかどうかをチェックします。
      // 存在する場合は DuplicateKeyException を返します。
      Statement statement = connection.createStatement();
      ResultSet rs =
        statement.executeQuery("SELECT id FROM account WHERE id = " +
          accountId);
      if (rs.next())
        throw new DuplicateKeyException();
      statement.close();

      // ID が存在しない場合は、データを挿入します。
      statement =
        connection.createStatement();
      statement.executeUpdate
        ("INSERT INTO account (id, value) VALUES (" + accountId + ", " +
          _value +")");
      statement.close();
  }
  finally {
    connection.close();
  }
}
catch (NamingException naming) {
  throw new EJBException(naming);
}
catch (SQLException sql) {
  throw new EJBException(sql);
}

return new Integer(accountId);
}
...
```

ejbPostCreate メソッドを使用して、いずれかのビジネス メソッドを呼び出す前に、Bean インスタンスに必要な追加の初期化を実行します。ejbCreate メソッドの引数リストと ejbPostCreate メソッドの引数リストは一致している必要があります。

ejbLoad

コンテナでは、ビジネス メソッドを呼び出す直前に ejbLoad を呼び出します。データストアからエンティティを取得し、フィールドをインスタンス変数にコピーする ejbLoad メソッドにコードを追加します。次の ejbLoad 実装では、データベースに接続し、データを取得し、_value インスタンス変数を設定します。

```
public void ejbLoad() {
  try {
    Context ctx = new InitialContext();
    DataSource ds =
      (DataSource)ctx.lookup("java:comp/env/jdbc/source1");
    Connection connection = ds.getConnection();
    try {
      Statement statement = connection.createStatement();
      ResultSet results =
        statement.executeQuery("SELECT value FROM account WHERE id = "
          + _context.getPrimaryKey());
      results.next();
      _value = results.getInt(1);

      results.close();
      statement.close();
    }
    finally {
      connection.close();
    }
  }
  catch (Exception e) {
    throw new EJBException(e);
  }
}
```

メモ

env-entry 要素または Bean プロパティで ejipt.isDataShared を false に設定した場合、最初の ejbLoad または ejbCreate を使用してインスタンスがすでにロードされていると、コンテナでは ejbLoad を呼び出しません。ejipt.isDataCached を true に設定した場合は、ロールバックの後のビジネス メソッドの前に ejbLoad を呼び出しません。また、アクティブ化の後のビジネス メソッドの前にも ejbLoad を呼び出しません。既定では、ロールバックまたはアクティブ化の後に再ロードします。

ejbStore

コンテナでは、ビジネス メソッドの終了直後に ejbStore を呼び出します。データストアにエンティティを保存する ejbStore メソッドにコードを追加してください。

次の ejbStore 実装では、まずデータベースに接続し、次にオブジェクト のステート によってデータベースを更新しています。

```
public void ejbStore() {0
  try {
    Context ctx = new InitialContext();
    DataSource ds =
      (DataSource)ctx.lookup("java:comp/env/jdbc/source1");
    Connection connection = ds.getConnection();
    try {
      Statement statement = connection.createStatement();
      statement.executeUpdate("UPDATE account SET value = " + _value +
        " WHERE id = " + _context.getPrimaryKey());
       statement.close();
    }
    finally {
      connection.close();
    }
  }
  catch (Exception e) {
    throw new EJBException(e);
  }
}
```

EJB エンジンでは、ステート変化をチェックするときに、Bean インスタンスのメンバについて表面的な一致の比較 (つまり ==) しか行いません。このため、メンバ配列の要素だけが変更されていて、配列自体は変更されていない場合、EJB エンジンではステート変化は検出されません (307 ページの「インスタンスのステートの変化」を参照)。このような最適化が必要なのは、深いデータ構造などで長時間の反復が発生した場合のパフォーマンス低下を避けるためです。

JRun では次のメソッドとプロパティを追加して、ejbStore の呼び出しを制御します。

- 強制的に保存するには、ビジネス メソッドで InstanceManager.setDirty(true) メソッドを呼び出します。
- 変更を無視するように EJB エンジンに指示するには、ビジネス メソッドで InstanceManager.setDirty(false) メソッドを呼び出します。
- EJB エンジンに ejbStore の呼び出しを強制するには、Bean の公開記述子の env-entry 要素で、ejipt.isAlwaysDirty プロパティを true に設定します。

ejbRemove

データ ストアからエンティティ オブジェクトを削除するには、ejbRemove メソッドを呼び出します。次の ejbRemove 実装では、まずデータベースに接続し、次にデータベース内のオブジェクトの削除を試みています。

```
public void ejbRemove() throws RemoveException {
  try {
    Context ctx = new InitialContext();
    DataSource ds =
      (DataSource)ctx.lookup("java:comp/env/jdbc/source1");
    Connection connection = ds.getConnection();
    try {
      Statement statement = connection.createStatement();
      ResultSet results =
        statement.executeQuery("DELETE FROM account WHERE id = " +
          _context.getPrimaryKey());
    }
    finally {
      connection.close();
    }
  }
  catch (Exception e) {
    throw new RemoveException(e.toString());
  }
}
```

finder メソッド

ほとんどのクライアント アプリケーションでは、1 つまたは複数の行を取得し、場合によっては更新することによってデータベースと対話します。行を作成することはめったにありません。エンティティ Bean には、クライアントが 1 行以上の既存のデータにアクセスできる finder メソッドが実装されています。エンティティ Bean のホーム インターフェイスとその Bean 実装で、finder メソッドをコーディングします。

- ホーム インターフェイスでは findBy*Xxx*メソッドを指定します。このメソッドは、Bean のリモート インターフェイスへの参照を返すか、あるいはリモート インターフェイス参照の Enumeration または Collection を返します。
- Bean 実装には、ホーム インターフェイスで命名された finder メソッドごとに、同じシグネチャを持つパブリック ejbFindBy*Xxx* メソッドがあります。Bean 実装の finder メソッドではプライマリ キーを返すか、プライマリ キーの Enumeration または Collection を返します。

1 行の finder メソッド

すべてのエンティティ Bean には、findByPrimaryKey と ejbFindByPrimaryKey メソッドのペアが必要です。このメソッドでは引数としてプライマリ キーを受け入れます。関連付けられている行が、プライマリ キーによって返されるデータ ストアに存在することを確認するか、または FinderException を返します。

次のコード例は、サンプル 2a の BalanceHome の findByPrimaryKey メソッド のホーム インターフェイスを示します。

```
...
Balance findByPrimaryKey(Integer key) throws FinderException,
        RemoteException;
...
```

次のコード例は、サンプル 2a の BalanceBean の ejbFndByPrimaryKey メソッドの Bean 実装を示します。

```
...
public Integer ejbFindByPrimaryKey(Integer key)
        throws FinderException {
  boolean exists = false;
  try {
    Context ctx = new InitialContext();
    DataSource ds =
      (DataSource)ctx.lookup("java:comp/env/jdbc/source1");
    Connection connection = ds.getConnection();
    try {
      Statement statement = connection.createStatement();
      ResultSet results =
        statement.executeQuery("SELECT value FROM account WHERE id = "
          + key);

      if (results.next()) exists = true;
    }
    finally {
      connection.close();
    }
  }
  catch (Exception e) {
    throw new EJBException(e);
  }
  // 行が見つからない場合は例外を返します。
  if (! exists) throw new FinderException();

  // 行が存在していることを示すプライマリ キーを返します。
  return key;
}
...
```

複数行 finder メソッド

アプリ ケーションの必要条件によって、エンティティ Bean に 1 つまたは複数の複数行 finder メソッドを実装するように選択できます。これらのメソッドはオプションであり、引数を取らなかったり、複数の引数を取ったりします。複数行 finder の Bean 実装では、認証を実行してオプションで FinderException を返し、プライマリ キーの Enumeration または Collection を返すか、あるいは Collection を実装するいずれかのクラスの子孫を返します。

次のコード例は、サンプル 7b の BankHome の findAllBanks メソッドのホーム インターフェイスを示します。

```
...
Collection findAllBanks() throws RemoteException;
...
```

次のコード例は、サンプル 7b の BankBean の ejbFindAllBanks メソッドの Bean 実装を示します。

```
...
public Collection ejbFindAllBanks() {
  // 返す変数を定義します。ArrayList は AbstractCollection の子孫であり、
  // Collection を実装していることに注意してください。
  ArrayList names = new ArrayList();

  try {
    Context ctx = new InitialContext();
    DataSource ds =
      (DataSource)ctx.lookup("java:comp/env/jdbc/source1");
    Connection connection = ds.getConnection();
    try {
      Statement stmt = connection.createStatement();
      ResultSet results = stmt.executeQuery("SELECT name FROM bank");

      while (results.next()) {
        names.add(results.getString(1));
      }
      results.close();
      stmt.close();
    }
    finally {
      connection.close();
    }
  }
  catch (NamingException naming) {
    throw new EJBException(naming);
  }
  catch (SQLException sql) {
    throw new EJBException(sql);
  }

  // Collection (この場合は ArrayList) を返します。
  return names;
}
...
```

ビジネス メソッド

ビジネス メソッドは、create メソッドおよび finder メソッド以外で、クライアント アプリケーションによる呼び出しが可能な唯一の EJB メソッドです。リモート インターフェイスと Bean 実装で、エンティティ Bean のビジネス メソッドを定義します。それらのシグネチャと返されるタイプは一致する必要があります。

次のコード例は、サンプル 2a の Balance インターフェイスの getValue、save、および spend メソッドのリモート インターフェイスを示します。

```
package ejbeans;

import java.rmi.*;
import javax.ejb.*;

public interface Balance extends EJBObject {

  int getValue() throws RemoteException;
  void save(int value) throws RemoteException;
  void spend(int value) throws RemoteException;
}
```

次のコード例は、サンプル 2a の BalanceBean 実装におけるこれらのビジネス メソッドの実装を示します。

```
...
public int getValue() {
  return _value;
}

public void save(int value) {
  _value += value;
  ResourceManager.getLogger().logMessage
    ("saving, balance is:" + _value);
}

public void spend(int value) {
  _value -= value;
  ResourceManager.getLogger().logMessage
    ("spending, balance is:" + _value);
}
...
```

第 29 章

コンテナ管理パーシスタンス

この章では、EJB エンジンによって提供される CMP サポートについて説明します。

目次

- 概要 330
- 公開記述子の要素 330
- Bean のメソッド 336
- ストアド プロシージャの呼び出し 343
- CMP での開発者の責任 343

概要

コンテナ管理パーシスタンス (CMP) では、データのアクセス、同期化、および保持をコンテナが行います。ejbLoad、ejbStore、ejbCreate などのメソッドで、データベース ステートメントをコーディングする必要はありません。

JRun で CMP を利用するには Bean の公開記述子の要素を使用します。Bean のステートを保持する役割はコンテナがすべて果たすので、CMP を使用すると Bean 実装の複雑さが大幅に軽減されます。

CMP サポートは、JDBC データ ストアと instance.store のどちらでも使用できます。これらのアクションに関するマッピング情報や実際の SQL ステートメントを Bean の公開記述子で指定します。公開記述子を使用していない場合は、Bean プロパティで指定します。

JDBC のプリペアド ステートメントとストアド プロシージャの呼び出しが両方とも完全にサポートされます。JRun では、ほかの高価なリソースと同様に、効率を最大限に高めるために JDBC ステートメントを自動的にプールして管理します。

実行される SQL ステートメントの順序を詳細に制御できます。パラメータと実行される SQL ステートメントの結果は、どちらも Bean インスタンス フィールドを使用して格納します。

この CMP に対するアプローチに追加される利点として、複数のテーブルだけでなく、時には複数のデータベース内に実際に格納されているデータによって Bean のステートを設定できることが挙げられます。複数のデータベースを利用する場合、一連の SQL ステートメントを指定して、パーシスタンス処理中に実行できます。

公開記述子の要素

CMP を使用する場合に設定する記述子要素がいくつかあります。

- **パーシスタンス タイプ** persistence-type 要素のコンテナを指定して、EJB によって CMP が使用されることを示します。

```
<persistence-type>Container</persistence-type>
```

- **プライマリ キー クラス** prim-key-class 要素によってプライマリ キーのクラスを示します。

```
<prim-key-class>ejbeans.BalanceKey</prim-key-class>
```

- **プライマリ キーフィールド (オプション)** CMP エンティティ Bean でプライマリ キーとしてカスタム プライマリ キー クラスを使用しないで、代わりに java.lang.Integer などの Java プリミティブ ラッパー クラスを使用した場合、primkey-field 要素でプライマリ キー フィールドを指定します。

```
<prim-key-class>java.lang.Integer</prim-key-class>
<primkey-field>id</primkey-field>
```

- **コンテナ管理フィールド** `cmp-field` 要素を使用して、コンテナによってどのフィールドが管理されるかを指定します。コンテナで管理されるのは指定されたフィールドだけです。すべてのプライマリ キー フィールドを指定する必要があります。

```
<cmp-field>
  <field-name>id</field-name>
</cmp-field>
<cmp-field>
  <field-name>value</field-name>
</cmp-field>
```

- **SQL ステートメント** CMP でリレーショナル データベースを使用する場合は、公開記述子で `env-entry` 要素を指定します。これらの要素では、パーシスタンスを保つのに必要な SQLステートメントを指定します。EJB エンジンでは、アクションのタイプごとに一連の `env-entry` 要素が必要です。次の表は、`env-entry` の `env-entry-name` 要素で定義できるパーシスタンス アクションの一覧です。

env-entry-name 要素	説明
ejipt.*action*SQL	データベースと対話する SQL ステートメントまたはストアド プロシージャの呼び出しを指定します。
ejipt.*action*SQL.source	JDBC ソースを示します。これは、JMC で定義された JRun データ ソースと一致する必要があります。
ejipt.*action*SQL.params	SQL ステートメント内の任意のパラメータに関連するフィールドの名前を表示順に示します。各パラメータは、「IN」(既定値)、「OUT」、または「INOUT」で修飾できます。パラメータ情報を指定するほかの方法については、335 ページの「SQL ステートメントのパラメータ」を参照してください。
ejipt.*action*SQL.paramTypes	params プロパティで指定されているフィールドごとに JDBC データ タイプを表示順に示します。
ejipt.*action*SQL.fields	ResultSet に返される要素について、宛先フィールド名を表示順に示します。

次の表は、すべての可能なアクションと CMP 関連の `env-entry-name` 要素の一覧です。

アクション	env-entry-name 要素
`create`	`ejipt.createSQL`
	`ejipt.createSQL.source`
	`ejipt.createSQL.params`
	`ejipt.createSQL.paramTypes`
	`ejipt.createSQL.fields`
`postCreate`	`ejipt.postCreateSQL`
	`ejipt.postCreateSQL.source`
	`ejipt.postCreateSQL.params`
	`ejipt.postCreateSQL.paramTypes`
	`ejipt.postCreateSQL.fields`
`load`	`ejipt.loadSQL`
	`ejipt.loadSQL.source`
	`ejipt.loadSQL.params`
	`ejipt.loadSQL.paramTypes`
	`ejipt.loadSQL.fields`
`store`	`ejipt.storeSQL`
	`ejipt.storeSQL.source`
	`ejipt.storeSQL.params`
	`ejipt.storeSQL.paramTypes`
	`ejipt.storeSQL.fields`
`remove`	`ejipt.removeSQL`
	`ejipt.removeSQL.source`
	`ejipt.removeSQL.params`
	`ejipt.removeSQL.paramTypes`
	`ejipt.removeSQL.fields.`
`findname`	ejipt.find*name*SQL
	ejipt.find*name*SQL.source
	ejipt.find*name*SQL.params
	ejipt.find*name*SQL.paramTypes
	ejipt.find*name*SQL.fields

次の例は、EJB サンプル 3a の公開記述子の env-entry を示します。

```
...
<!--
  CMP properties
-->
<env-entry>
  <env-entry-name>ejipt.createSQL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INSERT INTO account (id, value) VALUES (?, ?)
        </env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.createSQL.source</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>source1</env-entry-value>
  </env-entry>
<env-entry>
  <env-entry-name>ejipt.createSQL.params</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_id, _value</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.createSQL.paramTypes</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INTEGER, INTEGER</env-entry-value>
</env-entry>

<env-entry>
  <env-entry-name>ejipt.loadSQL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>SELECT value FROM account WHERE id =
        ?</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.loadSQL.source</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>source1</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.loadSQL.params</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_id</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.loadSQL.paramTypes</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INTEGER</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.loadSQL.fields</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_value</env-entry-value>
</env-entry>
```

```
<env-entry>
  <env-entry-name>ejipt.storeSQL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>UPDATE account SET value = ?WHERE id =
        ?</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.storeSQL.source</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>source1</env-entry-value>
</env-entry>
  <env-entry>
  <env-entry-name>ejipt.storeSQL.params</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_value, _id</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.storeSQL.paramTypes</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INTEGER, INTEGER</env-entry-value>
</env-entry>
...
```

複数の SQL ステート メント

Bean では、1 つのパーシスタンス アクションについて複数の SQL ステート メントが必要になる場合があり ます。たとえば、1 つの Bean インスタンスで表されるデータを取得するために、複数の SQLステート メント を使用すること があり ます。この場合は、次のように、SQL ステート メントの env-entry が追加されるたびに、env-entry にインデックスが関連付けられます。

```
...
ejipt.actionSQL
ejipt.actionSQL.source
ejipt.actionSQL.params
ejipt.actionSQL.paramTypes
ejipt.actionSQL.fields
ejipt.actionSQL2
ejipt.actionSQL2.source
ejipt.actionSQL2.params
ejipt.actionSQL2.paramTypes
ejipt.actionSQL2.fields
...
ejipt.actionSQLn
ejipt.actionSQLn.source
ejipt.actionSQLn.params
ejipt.actionSQLn.paramTypes
ejipt.actionSQLn.fields
...
```

SQL ステート メントは、プロパティ名にあるインデックス順に昇順に実行されます。SQL ステート メントは無制限に追加できます。

SQL ステートメントのパラメータ

CMP の場合、コンテナには、各メソッドに渡すパラメータに関する情報が必要です。この情報を使用して、SQL ステートメントの疑問符を実際の値に関連付けます。次のテクニックを使用してこれらの関連を指定します。

- **Bean 変数** param および paramTypes env-entry を指定すると、コンテナでは Bean インスタンス変数を使用して、関連付けられている SQL で指定された疑問符を置き換えます。
- **メソッド引数** コンテナで使用されるメソッド引数を指定して、関連付けられている SQL で指定された疑問符を置き換えることもできます。

コンテナで Bean 変数とメソッド引数を組み合わせて SQL ステートメントで指定された疑問符を置き換えるように、これらのメソッドを組み合わせることができます。

Bean 変数を使用した SQL パラメータの置換

param および paramTypes env-entry を指定すると、コンテナでは Bean インスタンス変数を使用して、関連付けられている SQL で指定された疑問符を SQL ステートメントに表示される順序で置き換えます。たとえば、次のような env-entry があります。

```
<env-entry>
  <env-entry-name>ejipt.createSQL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>
    INSERT INTO account (id, value) VALUES (?, ?)
  </env-entry-value>
</env-entry>
```

上記の例は、次の env-entry によって処理できます。

```
<env-entry>
  <env-entry-name>ejipt.createSQL.params</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_id, _value</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.createSQL.paramTypes</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INTEGER, INTEGER</env-entry-value>
</env-entry>
```

Bean インスタンス変数がヌルではないことを確認する必要があります。これは通常、ejbLoad、ejbStore、および ejbPostCreate に関する問題ではありません。ただし、ejbCreate の場合は、指定されたインスタンス変数が初期化されていることを確認するコードを追加する必要があります。たとえば、上記の env-entry を使用するには、次の ejbCreate メソッドをコーディングします。

```
public void ejbCreate(int accountId)
  throws CreateException, RemoteException {
    // CMP 用の Bean インスタンス変数を初期化します。
    _id = accountId;
    _value = 0;
}
```

詳細については、サンプル 3a の ejb-xml.jar および BalanceBean.java ファイルを参照してください。

メソッドの引数を使用した SQL パラメータの置換

メソッドの引数を使用して SQL を置き換える場合は、疑問符の後にメソッド引数内での位置を示す数字を付けて、SQL パラメータをメソッド引数に関連付けます。たとえば、コンテナでは ?1 をメソッドの最初の引数に、?2 をメソッドの 2 番目の引数にというように置き換えます。

このテクニックは、クエリ フィールドが必ずしも Bean インスタンス変数に対応するとは限らない複数行 finder メソッドに最も有効です。たとえば、このテクニックは、次の env-entry に有効です。param および paramTypes env-entry は必要でないことに注意してください。

```
<env-entry>
  <env-entry-name>ejipt.createSQL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>
    INSERT INTO account (id, value) VALUES (?1, ?2)
  </env-entry-value>
</env-entry>
```

この env-entry に対応する ejbCreate メソッドには、次の例に示すように初期化コードは必要ありません。

```
public void ejbCreate(int accountId, int value)
  throws CreateException, RemoteException { }
```

Bean のメソッド

このセクションでは、各必須 Bean メソッドの背後にある処理とパーシスタンスロジックについて詳しく説明します。

ejbCreate および ejbPostCreate

Bean インスタンスを作成する場合、コンテナでは Bean の ejbCreate メソッドを呼び出します。ejbCreate メソッドによって、渡された引数を検証し、Bean 変数を使用して SQL パラメータを置き換える場合は、渡された引数を Bean インスタンス変数にコピーできます。次に、コンテナでは、createSQL ステートメントを実行します。通常はこの createSQL ステートメントによってデータベースに行が挿入されます。プライマリ キーを持つ行がすでに存在する場合は、DuplicateKeyException が返されます。

次に、コンテナでは、Bean の ejbPostCreate メソッドを呼び出します。

コンテナで実行される一連の手順は次のとおりです。

1 Bean インスタンスの ejbCreate を呼び出します。
2 createSQL ステートメントを実行します。
3 プライマリ キーがデータベースに存在している場合は、javax.ejb.DuplicateKeyException が返されます。
4 postCreateSQL を実行し、ejbPostCreate メソッドを呼び出します。

公開記述子で Bean 変数を使用して SQL パラメータを置き換える場合は、次の例に示すように、ejbCreate メソッドによって、渡された引数を使用する Bean フィールドを初期化する必要があります。

```
public void ejbCreate(int accountId)
        throws CreateException, RemoteException {
  _id = accountId;
  _value = 0;
}
```

ejipt.createSQL env-entry には通常、次のようにデータベースに行を追加する INSERT ステートメントが含まれています。

```
<env-entry>
  <env-entry-name>ejipt.createSQL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INSERT INTO account (id, value) VALUES (?, ?)</
        env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.createSQL.source</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>source1</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.createSQL.params</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_id, _value</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.createSQL.paramTypes</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INTEGER, INTEGER</env-entry-value>
</env-entry>
```

ejbLoad

Bean メソッドを実行する前に、コンテナでは、loadSQL ステートメントを実行し、指定されたフィールドにその結果をコピーして、データベースのステートと Bean インスタンスを同期化します。次に、データベースから取得された値に基づいて詳細に初期化を実行できるように、Bean で ejbLoad メソッドを呼び出します。

メモ

env-entry 要素または Bean プロパティで ejipt.isDataShared を false に設定した場合は、最初の ejbLoad または ejbCreate を使用してインスタンスがすでにロードされていると、コンテナでは ejbLoad を呼び出しません。ejipt.isDataCached を true に設定した場合は、ロールバックの後のビジネス メソッド の前に ejbLoad を呼び出しません。また、アクティブ化の後のビジネス メソッドの前にも ejbLoad を呼び出しません。既定では、ロールバックまたはアクティブ化の後に再ロードします。

---

ejipt.loadSQL の値には通常、次のように、Bean インスタンスのデータが格納されている結果セットを返す SELECT ステートメントが含まれています。

```
<env-entry>
  <env-entry-name>ejipt.loadSQL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>SELECT value FROM account WHERE id = ?</
        env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.loadSQL.source</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>source1</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.loadSQL.params</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_id</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.loadSQL.paramTypes</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INTEGER</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.loadSQL.fields</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_value</env-entry-value>
</env-entry>
```

ejbStore

Bean メソッドの終了後に、コンテナでは Bean インスタンスで `ejbStore` メソッドを呼び出します。この手順を使用すると、データベースにデータを格納して Bean インスタンスのステートを同期化する前に、Bean によってインスタンスのパラメータフィールドを準備できます。

JRun では次のようにメソッドとプロパティを追加して、`ejbStore` の呼び出しを制御します。

- 強制的に保存するには、ビジネス メソッドで `InstanceManager.setDirty(true)` メソッドを呼び出します。
- 変更を無視するように EJB エンジンに指示するには、ビジネス メソッドで `InstanceManager.setDirty(false)` メソッドを呼び出します。
- EJB エンジンに `ejbStore` の呼び出しを強制するには、Bean の公開記述子の `env-entry` 要素で `ejipt.isAlwaysDirty` プロパティを `true` に設定します。

`ejipt.storeSQL` の値には通常、次のように、データベースと Bean の現在のステートを同期化する UPDATE ステートメントが含まれています。

```
<env-entry>
  <env-entry-name>ejipt.storeSQL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>UPDATE account SET value = ?WHERE id =
        ?</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.storeSQL.source</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>source1</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.storeSQL.params</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_value, _id</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.storeSQL.paramTypes</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INTEGER, INTEGER</env-entry-value>
</env-entry>
```

`dirty` フラグが設定されていない場合、コンテナでは、表面的な比較によってインスタンスを保存するかどうかを判断します。コンテナは == 演算子を使用してフィールドのステートとキャッシュされているステートを繰り返し比較して、ステートが変化したかどうかを判断します。深いレベルでの変化が起きた場合、Bean メソッド内で `dirty` フラグを明示的に `true` に設定して、保存を強制しなければなりません。

ejbRemove

インスタンスを削除する場合、コンテナでは、インスタンスがデータベースから削除される前に必要なクリーンアップを実行できるように ejbRemove メソッドを呼び出します。ejbRemove では、SQL ステートメントで使用するパラメータを設定する必要があります。

ejipt.removeSQL の値には通常、次のように DELETE ステートメントが含まれています。

```
<env-entry>
  <env-entry-name>ejipt.removeSQL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>DELETE FROM account WHERE id = ?</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.removeSQL.source</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>source1</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.removeSQL.params</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>_id</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>ejipt.removeSQL.paramTypes</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>INTEGER</env-entry-value>
</env-entry>
```

finder メソッド

ほとんどのクライアント アプリケーションでは、1 つまたは複数の行を取得し、場合によっては更新することによってデータベースと対話します。行を作成することはほとんどありません。エンティティ Bean には、クライアントが 1 行以上の既存のデータにアクセスできる finder メソッドが実装されています。エンティティ Bean のホーム インターフェイスとその Bean 実装で、finder メソッドをコーディングします。

- ホーム インターフェイスでは findBy*Xxx*メソッドを指定します。このメソッドは、Bean のリモート インターフェイスへの参照を返すか、あるいはリモート インターフェイス参照の Enumeration または Collection を返します。
- Bean 実装には、ホーム インターフェイスで命名された finder メソッドごとに、同じシグネチャを持つパブリック ejbFindBy*Xxx* メソッドがあります。

ejbFindByPrimaryKey

すべてのエンティティ Bean には、findByPrimaryKey と ejbFindByPrimaryKey メソッドのペアが必要です。ただし、CMP を使用するエンティティ Bean の Bean 実装で、ejbFindByPrimaryKey を実装する必要はありません。ホーム インターフェイスの findByPrimaryKey を定義し、公開記述子の findByPrimaryKeySQL env-entry を指定する必要があります。

エンティティ Bean には、プライマリ キー クラス内のメンバのインスタンス変数がすべて含まれている必要があります。プライマリ キーフィールドのほかに、Bean では返される行の追加の列が含まれるインスタンス変数も定義できます。

ejbFindByPrimaryKey メソッドを呼び出した後、コンテナでは、findByPrimaryKeySQL env-entry によって指定された SQL ステートメントを実行します。最初に、パラメータ引数、またはプロパティで指定されているフィールド、あるいはその両方を SQL ステートメントの各パラメータにマッピングします。ストアド プロシージャを指定した場合は、キーとして渡された値に関連するオブジェクトがデータベースに含まれているかどうかを調べるチェックがストアド プロシージャに含まれている必要があります。データベース内にデータがある場合は、指定されているフィールドに格納した後で、該当する行を 1 つ返す必要があります。この行によって、コンテナでは、プライマリ キークラスの有効なインスタンスを構築できます。プライマリ キーに渡されたメンバがデータベース内のオブジェクトに関連付けられていない場合、SQL は行を返しません。

コンテナでは、新しく作成されたプライマリ キー インスタンスを取り出し、そのインスタンスのオブジェクト テーブルから適切な EJB オブジェクトを検索します。オブジェクトが見つかった場合、そのオブジェクトをホーム インターフェイスの findByPrimaryKey メソッドの結果として返します。EJB オブジェクトが見つからなかった場合、このメソッドでは finder 例外が返されます。

複数行 finder メソッド

CMP の使用時も複数行 finder メソッドを実装できます。finder メソッドごとに、find*SQL プロパティを指定する必要があります。たとえば、ホームインターフェイスで定義された findBigAccounts メソッドがあるとします。

クライアントによってリモート インターフェイスの findBigAccounts メソッドが呼び出されると、finderSQL ステートメントと併用する Bean インスタンス内のパラメータを設定するために、コンテナでは、最初に Bean の ejbfindBigAccounts メソッドを呼び出します。finder メソッドではヌルを返す必要があります。次に、コンテナでは各 finder SQL ステートメントを順番に呼び出します。結果セットが返されるたびに、コンテナでは次の処理が実行されます。

1 Bean インスタンス内のフィールドを設定します。

2 新しいプライマリ キーを作成します。

3 対応するフィールドをインスタンスからプライマリ キーにコピーします。

4 そのプライマリ キーを、返されるプライマリ キーの一覧に追加します。

SQL プロパティは次のようになります。

```
ejipt.findBigAccountsSQL = SELECT id FROM sometable WHERE balance >
        1000000
ejipt.findBigAccountsSQL.fields=custId
```

finder メソッドで基準が設定されている大口預金口座を見つけるために、次の env-entry を使用します。

```
ejipt.findBigAccountsSQL = SELECT id FROM sometable WHERE balance >?
ejipt.findBigAccountsSQL.params=criteria
ejipt.findBigAccountsSQL.paramTypes=double
ejipt.findBigAccountsSQL.fields=custId
```

プロパティ ファイルでは、プライマリ キーにも含まれている Bean のフィールドに対する結果列からのマッピングを指定する必要があります。結果に基づいてプライマリ キーのコレクションが作成されると、EJB エンジンでは、関連するエンティティ オブジェクト、つまり公開時にリモート インターフェイスから生成された実装クラスのインスタンスの検索や作成に取りかかります。

finder が一覧または Java 2 コレクションのどちらを返すかによって、EJB エンジンでは、エンティティ オブジェクトの一覧から抜き出した Enumeration と Collection のいずれかを作成し、それを呼び出し側に返します。

ストアド プロシージャの呼び出し

中かっこ {} で囲まれている SQL ステートメントは、ストアド プロシージャの呼び出しとして扱われます。ストアド プロシージャを使用するときは、必要に応じて、パラメータが IN、OUT、または INOUT のいずれであるかを指定します。指定しない場合は、既定値の IN が使用されます。たとえば、顧客 ID と日付を受け取り、預金残高を返すストアド プロシージャ getBalance があるとします。このプロパティは次のようになります。

```
ejipt.storeSQL={?=getBalance(?,?)}
```

パラメータ定義は次のようになります。

```
ejipt.storeSQL.params=balance:OUT,custId,date
```

次のスニペットに示すように、ストアド プロシージャでは、結果セットを返さずに値を返すことができます。

```
ejipt.postCreateSQL= { call create_balance( ?, ? ) }
ejipt.postCreateSQL.source= source1
ejipt.postCreateSQL.params= id, value:OUT
ejipt.postCreateSQL.paramTypes= INTEGER, INTEGER
```

CMP での開発者の責任

CMP を利用して Bean を開発する場合は、いくつかの重要なガイドラインがあります。パラメータの初期化が必要な場合は、それを ejbCreate、ejbStore、ejbRemove、および finder の各メソッドで提供し、事後処理が必要な場合は、それを ejbLoad メソッドと ejbPostCreate メソッドで提供する必要があります。

すべてのコンテナ管理フィールドは、公開記述子の cmp-field 要素に表示されていなければなりません。

コンテナは、Bean インスタンスのプライマリ キー タイプにアクセスできることが必要です。したがって、CMP を使用する Bean のプライマリ キーについては、次の規則が適用されます。

- プライマリ キー クラスはパブリックです。
- プライマリ キー クラスは、パブリックな既定のコンストラクタを持つ必要があります。
- プライマリ キー クラスでは、equals および hashCode メソッドを実装する必要があります。これらのメソッドがないと、オブジェクト テーブル検索が正しく機能しません。
- プライマリ キーの使用フィールドは、Bean インスタンスのメンバ変数です。
- プライマリ キーの使用フィールドは、公開記述子の cmp-fields 要素で命名されている必要があります。

プロトタイプ プライマリ キークラスを確認するには、EJB サンプル 3a の BalanceKey.java を参照してください。

第 30 章

Java でのメッセージング

この章では、JRun で Java Message Service を使用する方法について説明します。

目次

- 概要 346
- メッセージ コンポーネント 349
- メッセージ タイプ 351

概要

Java Message Service (JMS) 仕様では、メッセージを作成および送受信する一般的な方法を Java 開発者に提供する企業内メッセージング ミドルウェア システムを構築するための API を定義します。JMS は移植可能で、メッセージベースのビジネス アプリケーションをサポートします。

JRun では、EJB エンジン内に JMS がシームレスに統合されています。

この章では、JRun で提供される JMS サービスの使用法について説明します。JMS に関する知識があることが前提になります。JMS の詳細については、http://java.sun.com/products/jms/docs.html にある JMS 仕様を参照してください。

JRun JMS 実装では、Sun JMS バージョン 1.02 仕様の ポイントツーポイント (キューベース) とパブリッシュ / サブスクライブ (トピックベース) の同期、および非同期メッセージングがサポートされています。メッセージにはパーシスタンスを指定できるので、サーバーのシャットダウン時にメッセージが失われません。

トピックベースのメッセージングでは、永続性のあるサブスクライブが使用可能です。これにより、サブスクライバがアクティブではないときに生成されるメッセージを含め、生成されるすべてのメッセージがクライアントにより確実に受信されるようになります。

この章では、**プロデューサ**という 用語はメッセージを送信するクライアント を表します。また、**コンシューマ**という 用語はメッセージを受信するクライアント を表します。ただし、コンシューマ、プロデューサは両方とも **クライアント** と呼ばれます。

メッセージを生成、または処理するために、クライアントは、まずサーバーへの**接続**を確立し、Connection を呼び出して、Session を作成します。クライアントはサーバーと対話し、以前に確立された `Session` オブジェクトを使用してメッセージを生成または処理します。ポイントツーポイント メッセージングとパブリッシュ / サブスクライブでは、`Connection` オブジェクトと `Session` オブジェクトのカスタマイズされた子オブジェクトが使用されます。これらのオブジェクトの詳細については、351 ページの「メッセージ タイプ」を参照してください。

メモ
ポイントツーポイント メッセージングでは、プロデューサは**送信者**、コンシューマは**受信者**と呼ばれます。パブリッシュ / サブスクライブでは、プロデューサは**パブリッシャ**、コンシューマは**サブスクライバ**と呼ばれます。

Bean をコンシューマ (リスナ) として設定するには、`javax.jms.MessageListener` インターフェイスを Bean のリモート インターフェイスにある `implements` 節に入れ、Bean 実装に `javax.jms.MessageListener` インターフェイスと `onMessage` メソッドを実装します。

メッセージングのサポートは、ローカル エンティティ Bean をサポートする EJB エンジンの拡張可能なエンティティ アーキテクチャに基づいて、完全にトランザクション処理されています。これにより、メッセージの保持とロギングで BMP と CMP を使用できます。

JRun では、MessageQueueBean エンティティ Bean を使用して、キュー (ポイントツーポイント) が実装され、TopicDispatcherBean エンティティ Bean を使用してトピック (パブリッシュ / サブスクライブ) が実装されます。このアーキテクチャでは、既定の Bean 実装を書き換えることにより、メッセージング機能をカスタマイズできます。

JRun メッセージング アーキテクチャ

JRun では、次のオブジェクトを通じて JMS が実装されます。

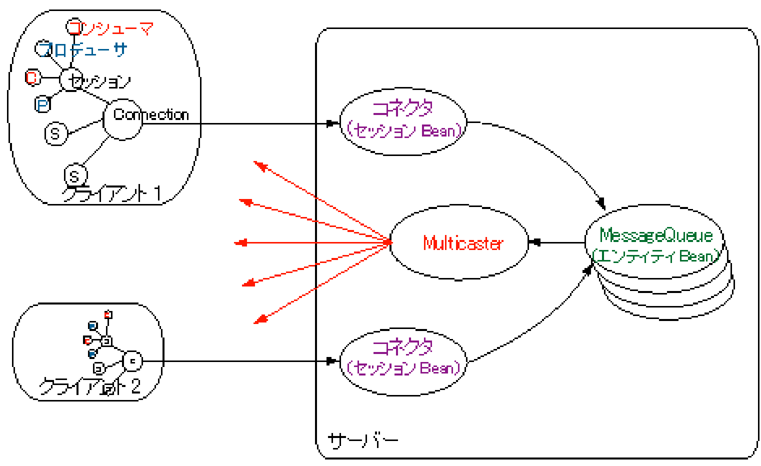
- Connector オブジェクトにより、JMS Connection オブジェクトと JRun MessageQueue オブジェクトの通信が管理されます。
- MessageQueue オブジェクトには、メッセージが保持されます。JRun では、エンティティ Bean を通じてこのオブジェクトが実装されます。
- MultiCaster オブジェクトにより、コンシューマにメッセージが配信されます。JRun では、エンティティ Bean を通じてこのオブジェクトが実装されます。

JRun により、クライアント メッセージは Connection オブジェクトから、サーバーの Connector を通じて指定された MessageQueue に転送されます。メッセージが MessageQueue に追加されると、MultiCaster により、message type と設定されたプロパティがあればそれに従って、メッセージがコンシューマに配布されます。通知が要求されている場合は、それがプロデューサに返されます。

リモート 接続

次の図に示すように、JRun では、リモート JMS 接続のために標準 UDP マルチキャスティングが使用されます。

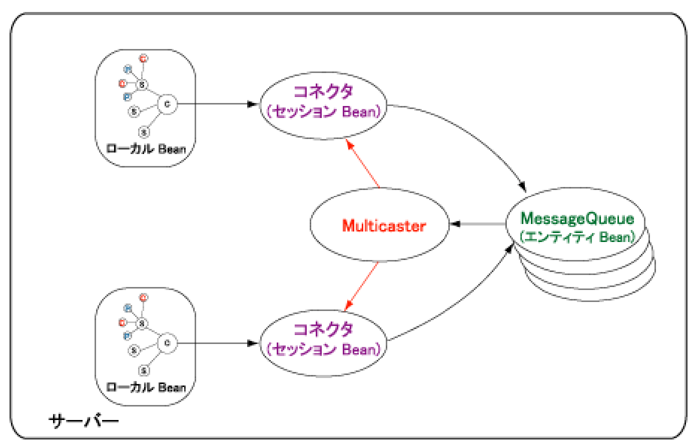
リモート接続



ローカル接続

次の図に示すように、ローカル接続の場合は、Multicaster により、メッセージが Connector を通じて配信されます。

ローカル接続



JMS サポートの有効化

JRun サーバーで JMS サポートを有効にするには、local.properties ファイルで次のプロパティを指定します。

```
ejb.services=ejb,jms
ejb.ejipt.enableMessaging=true
```

また、352 ページの「キューの定義」および363 ページの「トピックの定義」で説明しているように local.properties ファイルでも、キューとトピックをあらかじめ定義する必要があります。

EJB エンジンをスタンドアロン モードで実行している場合は、deploy.properties ファイルにこれらのプロパティを指定します。

メッセージ コンポーネント

メッセージは次の部分に分けられます。

- **ヘッダ**　メッセージの識別と階層のために、クライアントとサーバーの両方で使用される情報
- **プロパティ**　追加のヘッダ プロパティ。プロパティにはアプリケーション固有のプロパティ、標準プロパティ、サーバー固有のプロパティがあります。
- **本文**　メッセージの本文。メッセージの本文には、数種類の定義済みタイプのいずれかを使用できます。

メッセージ ヘッダ フィールド

JRun では、JMS メッセージ ヘッダ フィールドがサポートされていて、これらのフィールドは JMS メッセージ受信者に送信されます。JRun でサポートされている JMS メッセージ ヘッダ フィールドの概要は、次の表のとおりです。

フィールド	内容	設定者
JMSDestination	メッセージの送信先を表す Destination オブジェクト が含まれています。	JRun
JMSDeliveryMode	配信モード が含まれています。有効な値は DeliveryMode.PERSISTENT と DeliveryMode.NON_PERSISTENT です。	JRun
JMSMessageID	固有のメッセージ ID が含まれています。	JRun
JMSTimestamp	メッセージが JRun に送信された時刻が含まれています。	JRun
JMSCorrelationID	応答と、関連する要求をリンクするアプリケーション固有の文字列が含まれています。	JRun
JMSReplyTo	応答の送信先となる Destination オブジェクト が含まれています。応答は要求されていませんが、Destination オブジェクト がこのフィールド に含まれているのは応答が期待されていることを示しています。	JRun
JMSRedelivered	メッセージを再送信するかどうかを表す boolean が含まれています。コンシューマにより、JMSRedelivered が true に設定されたメッセージが受信された場合、このメッセージは以前配信されたが、コンシューマが受信を認めていなかったと考えられます。	クライアント
JMSType	メッセージ タイプを表す String が含まれています。	クライアント

フィールド	内容	設定者
JMSExpiration	メッセージの有効期限を指定する long が含まれています。JRun では、クライアントにより指定された Time - to - Live の値に送信時間の GMT を加えてこの時刻を設定します。	JRun
JMSPriority	メッセージの優先順位が含まれます。優先順位は 0 (最下位) から 9 (最上位) の間で指定します。	JRun

TextInterface や MapInterface などのコンテンツ固有のメッセージ インターフェイスにより拡張された Message インターフェイスにあるメソッドを使用して、ヘッダ フィールドにアクセスします。

メッセージ プロパティ

JRun は JMS 仕様で定義されているオプションの JMSX 接頭辞付きのメッセージ プロパティをサポートしません。しかし、Message オブジェクト メソッドを使用してプロパティを取得および設定できます。たとえば、次のコードの一部分を使用して、メッセージを送信する前にプロパティを設定できます。

```
...
try {
  // ユーザ ID のプロパティを設定します。thisUser String 変数を想定します。
  if(message != null) {
    message.setStringProperty("UserID", thisUser);
    message.setText(text);
    // キューに送信します。メッセージは 5 分間存続します。
    sender.send(_message, delivery, priority, 5 * 60 * 1000);
  }else {
    // 1 つのサーブレットまたは JSP ページでの使用法を想定します。
    out.println("<H1>Message was null</H1>");
  }
}
...
```

次のコードの一部分を使用して、メッセージを受信したときにプロパティを取得できます。

```
final TextMessage message = (TextMessage)(_receiver.receiveNoWait());
// すべてのプロパティを取得します。
Enumeration e = message.getPropertyNames();
if(!e.hasMoreElements()) {
  // 1 つのサーブレットまたは JSP ページでの使用法を想定します。
  out.println("<h1>no properties</H1>");
}
while(e.hasMoreElements()) {
  String prop = (String)e.nextElement();
  out.print("<p> " + prop);
  // すべてのプロパティが Strings であると想定します。
  out.println(":" + message.getStringProperty(prop));
}
```

メッセージ本文の種類

JMS 1.0.2 仕様では、メッセージ本文のフォームについて説明されています。これらのフォームは、Message を拡張したインターフェイスにより定義されます。

次の表は、EJB メッセージ本文インターフェイスの概要についてまとめたものです。

インターフェイス	説明	コメント
StreamMessage	Java プリミティブ値のストリームが含まれています。	このタイプは値の代入と読み込みが順次に行われます。
MapMessage	名前/値ペアのセット が含まれます。名前は String オブジェクト 、値は Java プリミティブ タイプです。	これらには、列挙によって連続的にアクセスするか、名前によってランダムにアクセスします。
TextMessage	String オブジェクト を 1 つ含んでいます。	TextMessage はテキスト メッセージまたは XML 形式のデータを持つメッセージで使用します。
ObjectMessage	Serializable Java オブジェクトが含まれています。	いずれかの JDK 1.2 コレクション クラスを使用できます。
BytesMessage	未解釈のバイト ストリームが含まれています。	通常、このタイプの本文は使用しません。

JRun では、これらのインターフェイスは allaire.ejipt._jms にある類似する名前のクラスを通じて実装されます。

メッセージ タイプ

EJB エンジンではポイントツーポイント メッセージング メカニズムと、パブリッシュ / サブスクライブ メッセージング メカニズムの両方がサポートされています。

ポイントツーポイント

ポイントツーポイントは、キューベースのメッセージング メカニズムです。メッセージは特定のキューに送られます。送信者はメッセージをキューに追加し、受信者はキューからメッセージを抽出します。

ポイントツーポイント メッセージング ソリューションを実装する場合は、次のコードを記述します。

- 送信者
- 受信者

キューの定義

`local.properties` ファイルの次のプロパティを使用してキューを定義します。

```
jms.queue.queue-name.description=queue description
jms.queue.queue-name.display-name=queue display name
```

JRun の起動時に、JNDI にキューを保存します。JNDI 検索を使用してキューにアクセスするコードは次のようになります。

```
_sender =
  _session.createSender((Queue)_context.lookup
    ("java:comp/env/jms/queue-name"));
```

詳細については、次の説明で送信者および受信者の例を参照してください。

EJB エンジンをスタンドアロン モードで実行している場合は、`deploy.properties` ファイルにこれらのプロパティを指定します。

送信者のコーディング

送信者により、メッセージがキューに追加される場合、次のオブジェクトが使用されます。

- `javax.naming.Context`
- `javax.jms.QueueConnectionFactory`
- `javax.jms.QueueConnection`
- `javax.jms.QueueSession`
- `javax.jms.QueueSender`
- 351 ページの「メッセージ本文の種類」で指定されているとおりの `javax.jms.Message` オブジェクトの子オブジェクトの 1 つ。

送信者クラスは、`javax.jms` クラスのいずれからも拡張されません。このクラスでは、明示的にほかのクラスを拡張したり、暗黙的に `java.lang.Object` を拡張できます。

メモ

次のコードをコンパイルおよび実行する場合は、必ずサーバーの `local.properties` ファイルでキューをキュー 1 と定義します。既定では、default JRun サーバーにはキュー 1 が含まれます。ただし、別のサーバーで実行している場合は、キュー 1 を定義しなければなりません。

送信者のコードを記述するには

1 次のパッケージをインポートします。

```
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import java.rmi.*;
```

2 次のように、JMS により使用されるオブジェクトのオブジェクトスコープ変数を宣言します。

```
private Context _context = null;
private QueueConnection _connection = null;
private QueueSession _session = null;
private TextMessage _message = null; // TextMessage を使用する例
private QueueSender _sender = null;
```

3 RMISecurityManager を使用してシステム セキュリティを確立します。

```
static {
  System.setSecurityManager(new RMISecurityManager());
}
```

4 メッセージを送信する前に、JMS 変数を作成して、値を代入します。

```
try {
  // JNDI で設定を行います。
  _context = new InitialContext();

  / JNDI から QueueConnectionFactory を取得します。
  final QueueConnectionFactory factory =
    (QueueConnectionFactory)_context.lookup
      ("java:comp/env/jms/QueueConnectionFactory");
  // ファクトリを使用して、QueueConnection (匿名) を作成します。
  _connection = factory.createQueueConnection();
  // QueueConnection を使用して、QueueSession を作成します。
  _session = _connection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
  // QueueSession を使用して QueueSender を作成します。
  // この例では、local.properties で指定されたキュー 1 を探します。
  // via jms.queue.queue1.description/display-name
  _sender = _session.createSender((Queue)_context.lookup
    ("java:comp/env/jms/queue1"));
  // TextMessage オブジェクトを作成します。
  _message = _session.createTextMessage();
}
catch(NamingException e) {
  System.out.println("Naming Exception:" + e.getMessage());
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

5 メッセージを送信します。このサーブレット例では、フォーム フィールドからメッセージが取得されています。

```
...
int priority = Message.DEFAULT_PRIORITY;
int delivery = Message.DEFAULT_DELIVERY_MODE;

String text = new String("Blank Message");

String[] attrArray = req.getParameterValues("thisMessage");
// 呼び出しフォームには、thisMessage に対する値が 1 つしかないと想定します。
if(attrArray != null) {
  text = attrArray[0];
}
try {
  _message.setText(text);
  // キューに送信します。メッセージは 5 分間存続します。
 _sender.send(_message, delivery, priority, 5 * 60 * 1000);
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
...
```

6 完了したら、JMS オブジェクトを閉じます。

```
try {
  _sender.close();
  _session.close();
  _connection.close();
  _context.close();
}
catch(NamingException e) {
  System.out.println("Naming exception in destroy:" +
      e.getMessage());
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

次のサーブレット例には、ポイントツーポイント メッセージを送信するためのコード全体が含まれています。

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import java.rmi.*;
```

```
public class MySender extends HttpServlet {
  // キュー属性を設定します。
  // init パラメータを介して渡されない限り、既定値を使用します。
  private final String _mode = new String("manual");
  private final String _name = new String("defaultUser");
  // Matches jms.queue.queue1.* properties in local.properties
  private final String _queue = new String("queue1");
  private final String _host = new String("rnielsen");

  // JMS で使用するオブジェクトを設定します。
  private Context _context = null;
  private QueueConnection _connection = null;
  private QueueSession _session = null;
  private TextMessage _message = null;
  private QueueSender _sender = null;

  static {
    System.setSecurityManager(new RMISecurityManager());
  }

public void init(ServletConfig config) throws ServletException {
  super.init(config);

  // JNDI で設定を行います。
  try {
    _context = new InitialContext();

    // JNDI から QueueConnectionFactory を取得します。
    final QueueConnectionFactory factory =
      (QueueConnectionFactory)_context.lookup
        ("java:comp/env/jms/QueueConnectionFactory");
    // ファクトリを使用して QueueConnection (匿名) を作成します。
    _connection = factory.createQueueConnection();
    // QueueConnection を使用して QueueSession を作成します。
    _session = _connection.createQueueSession(false,
      Session.AUTO_ACKNOWLEDGE);
    // QueueSession を使用して QueueSender を作成します。
    // local.properties で指定されたキューを参照します。
    // via jms.queue.queue1.description/display-name
    _sender = _session.createSender((Queue)_context.lookup
      ("java:comp/env/jms/queue1"));
    // TextMessage オブジェクトを作成します。
    _message = _session.createTextMessage();
  }
  catch(NamingException e) {
    System.out.println("Naming Exception:" + e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
}
```

```
// 送信するメッセージを受け取るフォームを表示します。
public void doGet(HttpServletRequest req,
  HttpServletResponse res) throws IOException, ServletException {
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();

  out.println
    ("<html><head><title>Message Sender</title></head><body>");
  out.println("<h1>Message Sender</h1>");
  out.println("<p>Type your message and press Submit</p><hr>");
  out.println("<form action=¥"/servlet/MySender¥" method=¥"post¥">");

  // メッセージを含んでいる Textarea
  out.println("<p>Message:<p>");
  out.println("<TextArea name=¥"thisMessage¥"></textarea>");
  out.println("<p>");
  out.println("<input type=¥"Submit¥" value=¥"Submit Message¥">");
  out.println("</form>");
  out.println("</body></html>");
}

// フォームを読み込んで、メッセージを送信します。
public void doPost(HttpServletRequest req,
  HttpServletResponse res) throws IOException, ServletException {
  // フォームを読み込みます。
  // メッセージはすべて、ここで送信されます。
  // 設定の内容については、init メソッドを参照してください。
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();

  int priority = Message.DEFAULT_PRIORITY;
  int delivery = Message.DEFAULT_DELIVERY_MODE;
  String text = new String("Blank Message");

  String[] attrArray = req.getParameterValues("thisMessage");
  // 呼び出しフォームには、thisMessage に対する値が 1 つしかないと想定します。
  if(attrArray != null) {
    text = attrArray[0];
  }
  try {
    _message.setText(text);
    // キューに送信します。メッセージは 5 分間存続します。
    _sender.send(_message, delivery, priority, 5 * 60 * 1000);
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }

  // 送信確認を表示します。
  out.println("<html><head><title>Message Sent</title></head><body>");
  out.println("<h1>Message Sent</h1>");
  out.println("<p>The following message was sent</p><hr>");
  out.println("<p>" + text);
  out.println("<form action=¥"/servlet/MySender¥" method=¥"get¥">");
```

```
  out.println("<p>");
  out.println("<input type=¥"Submit¥" value=¥"Return¥">");
  out.println("</form>");
  out.println("</body></html>");
}

public String getServletInfo() {
  return "Message Sender";
}

public void destroy() {
  // ヌル メッセージ チェックでない場合はラップします。
  try {
    _sender.close();
    _session.close();
    _connection.close();
    _context.close();
  }
  catch(NamingException e) {
    System.out.println("Naming exception in destroy: " +
      e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
} // メソッドを終了します。
} // サーブレットを終了します。
```

受信者のコーディング

受信者により、メッセージがキューから取得される場合、次のオブジェクトが使用されます。

- javax.naming.Context
- javax.jms.QueueConnectionFactory
- javax.jms.QueueConnection
- javax.jms.QueueSession
- javax.jms.QueueReceiver
- 351 ページの「メッセージ本文の種類」で指定されているとおりの javax.jms.Message オブジェクトの子オブジェクトの 1 つ。

受信者で非同期メッセージを使用する場合は、MessageListener インターフェイスを実装しなければなりません (この例では非同期メッセージは使用しません)。受信者クラスは、javax.jms クラスのいずれからも拡張されませんが、このクラスは明示的にほかのクラスを拡張したり、暗黙的に java.lang.Object を拡張できます。

受信者のコードを記述するには

1 次のパッケージをインポートします。

```
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import java.rmi.*;
```

2 次のように、JMS により使用されるオブジェクトのオブジェクトスコープ変数を宣言します。

```
private Context _context = null;
private QueueConnection _connection = null;
private QueueSession _session = null;
private TextMessage _message = null; // TextMessage を使用する例
private QueueReceiver _receiver = null;
```

3 RMISecurityManager を使用してシステム セキュリティを確立します。

```
static {
  System.setSecurityManager(new RMISecurityManager());
}
```

4 メッセージを受信する前に、JMS 変数を作成して、値を代入します。

```
try {
  _context = new InitialContext();

  // JNDI から QueueConnectionFactory を取得します。
  final QueueConnectionFactory factory =
    (QueueConnectionFactory)_context.lookup
      ("java:comp/env/jms/QueueConnectionFactory");
  // ファクトリを使用して QueueConnection を作成します。
  _connection = factory.createQueueConnection();
  // QueueConnection を使用して QueueSession を作成します。
  _session = _connection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
  // QueueSession を使用して QueueReceiver を作成します。
  // local.properties で指定されたキュー 1 を参照します。
  // via jms.queue.queue1.description/display-name
  _receiver =
    _session.createReceiver((Queue)_context.lookup
      ("java:comp/env/jms/queue1"));
  // TextMessage オブジェクトを作成します。
  _message = _session.createTextMessage();
}
catch(NamingException e) {
  System.out.println("Naming Exception:" + e.getMessage());
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

5 キューからメッセージを取得します。

```
...
// キューからメッセージを取得します。
try {
  // 接続を開始します。
  _connection.start();
  // メッセージ オブジェクトを取得します。
  final TextMessage message =
    (TextMessage)(_receiver.receiveNoWait());
  // メッセージ テキストを取得します。
  // ヌル メッセージをチェックします。
  String text = "";
  if (message != null) {
    text = new String(message.getText());
  }
  else {
    text = "No Message";
  }
  if(text.equals("")) {
    text = "Empty Message";
  }
  out.println(text);
  _connection.stop();
  out.println("</body></html>");
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
...
```

6 完了したら、JMS オブジェクトを閉じます。

```
try {
  _receiver.close();
  _session.close();
  _connection.close();
  _context.close();
}
catch(NamingException e) {
  System.out.println("Naming exception in destroy:" +
     e.getMessage());
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

次のサーブレット例には、ポイントツーポイント メッセージを受信するためのコード全体が含まれています。

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import java.rmi.*;

public class MyReceiver extends HttpServlet {
  // キュー属性を設定します。
  // init パラメータを介して渡されない限り、既定値を使用します。
  private final String _mode = new String("manual");
  private final String _name = new String("defaultUser");
  // Matches jms.queue.queue1.* properties in local.properties
  private final String _queue = new String("queue1");
  private final String _host = new String("rnielsen");

  // JMS で使用するオブジェクトを設定します。
  private Context _context = null;
  private QueueConnection _connection = null;
  private QueueSession _session = null;
  private TextMessage _message = null;
  private QueueReceiver _receiver = null;

  static {
    System.setSecurityManager(new RMISecurityManager());
  }

public void init(ServletConfig config) throws ServletException {
  super.init(config);

  // JNDI で設定を行います。
  try {
    _context = new InitialContext();

    // JNDI から QueueConnectionFactory を取得します。
    final QueueConnectionFactory factory =
      (QueueConnectionFactory)_context.lookup
        ("java:comp/env/jms/QueueConnectionFactory");
    // ファクトリを使用して QueueConnection を作成します。
    _connection = factory.createQueueConnection();
    // QueueConnection を使用して QueueSession を作成します。
    _session = _connection.createQueueSession(false,
      Session.AUTO_ACKNOWLEDGE);
    // QueueSession を使用して QueueReceiver を作成します。
    // local.properties で指定されたキュー１ を参照します。
    // via jms.queue.queue1.description/display-name
```

```
      _receiver =
        _session.createReceiver((Queue)_context.lookup
          ("java:comp/env/jms/queue1"));
      // TextMessage オブジェクトを作成します。
      _message = _session.createTextMessage();
    }
    catch(NamingException e) {
      System.out.println("Naming Exception:" + e.getMessage());
    }
    catch(JMSException e) {
      System.out.println("JMS Exception:" + e.getMessage());
    }
  }

  // 初めて実行したときだけ [メッセージの受信] ボタンを表示します。
  // メッセージを受信するフォームを表示します。
  public void doGet(HttpServletRequest req,
    HttpServletResponse res) throws IOException, ServletException {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println
      ("<html><head><title>Message Receiver</title></head><body>");
    out.println("<h1>Message Receiver</h1>");
    out.println
      ("<p>Press Receive Message to retrieve a message:<p><hr>");
    out.println("<form action=¥"/servlet/MyReceiver¥" method=¥"post¥">");
    out.println("<p>");
    out.println("<input type=¥"Submit¥" value=¥"Receive Message¥">");
    out.println("</form>");
    out.println("</body></html>");
  }

  public void doPost(HttpServletRequest req,
    HttpServletResponse res) throws IOException, ServletException {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println
      ("<html><head><title>Message Receiver</title></head><body>");
    out.println("<h1>Message Receiver</h1>");
    out.println
      ("<p>Press Receive Message to retrieve another message:</p><hr>");
    out.println("<form action=¥"/servlet/MyReceiver¥" method=¥"post¥">");
    // キューからメッセージを取得します。
    try {
      // 接続を開始します。
      _connection.start();
      // メッセージ オブジェクトを取得します。
      final TextMessage message =
        (TextMessage)(_receiver.receiveNoWait());
```

```
      // メッセージ テキストを取得します。
      // ヌル メッセージをチェックします。
      String text = "";
      if (message != null) {
        text = new String(message.getText());
      }
      else {
        text = "No Message";
      }
      if(text.equals("")) {
        text = "Empty Message";
      }
      out.println(text);
      _connection.stop();
      out.println("</body></html>");
    }
    catch(JMSException e) {
      System.out.println("JMS Exception:" + e.getMessage());
    }

    // form/html を完成させます。
    out.println("<p>");
    out.println("<input type=¥"Submit¥" value=¥"Receive Message¥">");
    out.println("</form>");
    out.println("</body></html>");
  }

  public String getServletInfo() {
    return "Message Receiver";
  }

  public void destroy() {
    // ヌル メッセージ チェックでない場合はラップします。
    try {
      _receiver.close();
      _session.close();
      _connection.close();
      _context.close();
    }
    catch(NamingException e) {
      System.out.println("Naming exception in destroy: " +
        e.getMessage());
    }
    catch(JMSException e) {
      System.out.println("JMS Exception:" + e.getMessage());
    }
  } // メソッドを終了します。
} // サーブレットを終了します。
```

パブリッシュ /サブスクライブ

パブリッシュ / サブスクライブはブロードキャスト メカニズムの 1 つです。パブリッシュ / サブスクライブではメッセージはトピックに発行され、自動的に**トピック**のサブスクライバに配布されます。トピックを階層的にしておくと、最上位レベルのサブスクライバはメッセージをすべて受信しますが、サブトピックのサブスクライバはサブトピック メッセージだけを受信します。

トピックの定義

local.properties ファイルの次のプロパティを使用してトピックを定義します。

```
jms.topic.topic-name.description=topic description
jms.topic.topic-name.display-name=topic display name
```

JRun の起動時に、JNDI にトピックを保存します。JNDI 検索を使用してトピックにアクセスするコードは次のようになります。

```
_publisher =
  _session.createPublisher((Topic)_context.lookup
    ("java:comp/env/jms/topic-name");
```

詳細については、次の説明でパブリッシャおよびサブスクライバの例を参照してください。

EJB エンジンをスタンドアロン モードで実行している場合は、deploy.properties ファイルにこれらのプロパティを指定します。

パブリッシャのコーディング

パブリッシャにより、メッセージがトピックに追加される場合は、次のオブジェクトが使用されます。

- javax.naming.Context
- javax.jms.TopicConnectionFactory
- javax.jms.TopicConnection
- javax.jms.TopicSession
- javax.jms.TopicPublisher
- 351 ページの「メッセージ本文の種類」で指定されているとおりの javax.jms.Message オブジェクトの子オブジェクトの 1 つ。

パブリッシャ クラスは、javax.jms クラスのいずれからも拡張されません。このクラスでは、明示的にほかのクラスを拡張したり、暗黙的に java.lang.Object を拡張できます。

メモ

次のコードをコンパイルおよび実行する場合は、JRun サーバーの local.properties ファイルでトピックをトピック 1 と定義することを確認します。

パブリッシャのコードを記述するには

1 次のパッケージをインポートします。

```
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import java.rmi.*;
```

2 次のように、JMS により使用されるオブジェクトのオブジェクトスコープ変数を宣言します。

```
private Context _context = null;
private TopicConnection _connection = null;
private TopicSession _session = null;
private TextMessage _message = null;
private TopicPublisher _publisher = null;
```

3 RMISecurityManager を使用してシステム セキュリティを確立します。

```
static {
  System.setSecurityManager(new RMISecurityManager());
}
```

4 メッセージを送信する前に、JMS 変数を作成して、値を代入します。

```
try {
  _context = new InitialContext();

  final TopicConnectionFactory factory =
    (TopicConnectionFactory)_context.lookup
      ("java:comp/env/jms/TopicConnectionFactory");
  // ファクトリを使用して TopicConnection を作成します。
  _connection = factory.createTopicConnection();
  // TopicConnection を使用して TopicSession を作成します。
  _session = _connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
  // TopicSession を使用して TopicPublisher を作成します。
  _publisher =
    _session.createPublisher((Topic)_context.lookup
      ("java:comp/env/jms/" + _topic));
  // TextMessage オブジェクトを作成します。
  _message = _session.createTextMessage();
}
catch(NamingException e) {
  System.out.println("Naming Exception:" + e.getMessage());
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

5 メッセージを発行します。このサーブレット例では、フォーム フィールドからメッセージが取得されています。

```
...
int priority = Message.DEFAULT_PRIORITY;
int delivery = Message.DEFAULT_DELIVERY_MODE;

String text = new String("Blank Message");

String[] attrArray = req.getParameterValues("thisMessage");
// 呼び出しフォームには、thisMessage に対する値が 1 つしかないと想定します。
if(attrArray != null) {
  text = attrArray[0];
}
try {
  _message.setText(text);
  // トピックを発行します。メッセージは 5 分間存続します。
  _publisher.publish(_message, delivery, priority, 5 * 60 * 1000);
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
...
```

6 完了したら、JMS オブジェクトを閉じます。

```
try {
  _publisher.close();
  _session.close();
  _connection.close();
  _context.close();
}
catch(NamingException e) {
  System.out.println("Naming exception in destroy:" +
     e.getMessage());
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

次のサーブレット例には、トピックを発行するために必要なコードがすべて含まれています。

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import java.rmi.*;
```

```
public class MyPublisher extends HttpServlet {
  // キュー属性を設定します。
  private String _topic = new String("topic1");
  private String _host = new String("rnielsen");

  // JMS で使用するオブジェクトを設定します。
  private Context _context = null;
  private TopicConnection _connection = null;
  private TopicSession _session = null;
  private TextMessage _message = null;
  private TopicPublisher _publisher = null;

  static {
    System.setSecurityManager(new RMISecurityManager());
  }

public void init(ServletConfig config) throws ServletException {
  super.init(config);
  // JNDI で設定を行います。
  try {
    _context = new InitialContext();

    final TopicConnectionFactory factory =
      (TopicConnectionFactory)_context.lookup
        ("java:comp/env/jms/TopicConnectionFactory");
    // ファクトリを使用して TopicConnection を作成します。
    _connection = factory.createTopicConnection();
    // TopicConnection を使用して TopicSession を作成します。
    _session = _connection.createTopicSession(false,
      Session.AUTO_ACKNOWLEDGE);
    // TopicSession を使用して TopicPublisher を作成します。
    _publisher =
      _session.createPublisher((Topic)_context.lookup
        ("java:comp/env/jms/" + _topic));
    // TextMessage オブジェクトを作成します。
    _message = _session.createTextMessage();
  }
  catch(NamingException e) {
    System.out.println("Naming Exception:" + e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
} // init を終了します。

// 発行するメッセージを受け取るフォームを表示します。
public void doGet(HttpServletRequest req,
  HttpServletResponse res) throws IOException, ServletException {

  res.setContentType("text/html");
  PrintWriter out = res.getWriter();
```

```
  out.println
    ("<html><head><title>Message Publisherr</title></head><body>");
  out.println("<h1>Message Publisher</h1>");
  out.println("<p>Type your message and press Submit</p><hr>");
  out.println
    ("<form action=¥"/servlet/MyPublisher¥" method=¥"post¥">");

  // メッセージを含んでいる Textarea
  out.println("<p>Message:<p>");
  out.println("<TextArea name=¥"thisMessage¥"></textarea>");
  out.println("<p>");
  out.println("<input type=¥"Submit¥" value=¥"Submit Message¥">");
  out.println("</form>");
  out.println("</body></html>");
} // doGet を終了します。

// フォームを読み込んで、メッセージを送信します。
public void doPost(HttpServletRequest req,
  HttpServletResponse res) throws IOException, ServletException {
  // フォームを読み込みます。
  // メッセージはすべて、ここで送信されます。
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();

  int priority = Message.DEFAULT_PRIORITY;
  int delivery = Message.DEFAULT_DELIVERY_MODE;

  String text = new String("Blank Message");

  String[] attrArray = req.getParameterValues("thisMessage");
  // 呼び出しフォームには、thisMessage に対する値が 1 つしかないと想定します。
  if(attrArray != null) {
    text = attrArray[0];
  }
  try {
    _message.setText(text);
    // トピックを発行します。メッセージは 5 分間存続します。
    _publisher.publish(_message, delivery, priority, 5 * 60 * 1000);
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }

  // 送信確認を表示します。
  out.println
    ("<html><head><title>Message Published</title></head><body>");
  out.println("<h1>Message Published</h1>");
  out.println("<p>The following message was sent</p><hr>");
  out.println("<p>" + text);
  out.println("<form action=¥"/servlet/MyPublisher¥" method=¥"get¥">");
```

```
  out.println("<p>");
  out.println("<input type=¥"Submit¥" value=¥"Return¥">");
  out.println("</form>");
  out.println("</body></html>");

} // doPost を終了します。

public String getServletInfo() {
  return "Message Publisher";
}

public void destroy() {
  // ヌル メッセージ チェックでない場合はラップします。
  try {
    _publisher.close();
    _session.close();
    _connection.close();
    _context.close();
  }
  catch(NamingException e) {
    System.out.println("Naming exception in destroy: " +
      e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
} // destroy を終了します。
} // サーブレットを終了します。
```

サブスクライバのコーディング

JMS により、関連するトピックに対してトピックが発行されたときに、これがサブスクライバに渡されます。次のオブジェクトを使用して、サブスクライバのコードを記述します。

- `javax.naming.Context`
- `javax.jms.TopicConnectionFactory`
- `javax.jms.TopicConnection`
- `javax.jms.TopicSession`
- `javax.jms.TopicSubscriber`
- 351 ページの「メッセージ本文の種類」で指定されているとおりの `javax.jms.Message` オブジェクトの子オブジェクトの 1 つ。

サブスクライバ クラスには、`MessageListener` インターフェイスと `onMessage` メソッドを実装する必要があります。登録されたサブスクライバ クラスへのトピックに対してメッセージが発行されたときに、JRun により、`onMessage` メソッドが呼び出されます。

サブスクライバ クラスは、`javax.jms` クラスのいずれからも拡張されません。このクラスでは、明示的にほかのクラスを拡張したり、暗黙的に `java.lang.Object` を拡張できます。

メモ
次のコード例にあるサブスクライバ アーキテクチャでは、スタンドアロン メッセージ リスナ クラスを使用して、メッセージが捕捉され、保存されます。その後、このクラスは所有者クラス (この場合はサーブレット) によってインスタンス化されます。このクラスにより、必要に応じてメッセージへのアクセスが行われます。

サブスクライバ (リスナ) のコードを記述するには

1 次のパッケージをインポートします。

```
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import java.rmi.*;
```

2 MessageListener インターフェイスを実装するクラス宣言を作成します。

```
public class MyTopicListener implements MessageListener {
...
```

3 次のように、JMS により使用されるオブジェクトのオブジェクトスコープ変数を宣言します。

```
private Context _context = null;
private TopicConnection _connection = null;
private TopicSession _session = null;
private TopicSubscriber _subscriber = null;
```

4 メッセージを保管するためのオブジェクトを作成します。この例では、Vector を使用しています。

```
// これは単なる例です。リスナは、さまざまなテクニックを使用して
// メッセージを管理することができます。
Vector theMessages = new Vector();
```

5 RMISecurityManager を使用してシステム セキュリティを確立します。

```
static {
  System.setSecurityManager(new RMISecurityManager());
}
```

6 メッセージを受信する前に、JMS 変数を作成して、値を代入します。

```
try {
  _context = new InitialContext();

  final TopicConnectionFactory factory =
    (TopicConnectionFactory)_context.lookup
      ("java:comp/env/jms/TopicConnectionFactory");
  // ファクトリを使用して TopicConnection を作成します。
  _connection = factory.createTopicConnection();
  // TopicConnection を使用して TopicSession を作成します。
  _session = _connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
  // TopicSession を使用して TopicSubscriber を作成します。
  // トピック 1 を使用して、local.properties で次のようなコードによって
     定義されます。
  // jms.topic.topic1.description=Topic for Doc Sample
  // jms.topic.topic1.display-name=DocSampleTopic
  _subscriber =
    _session.createSubscriber((Topic)_context.lookup
      ("java:comp/env/jms/" + _topic));
  // messageListener を設定します。
  _subscriber.setMessageListener(this);
  _connection.start();
}
catch(NamingException e) {
  System.out.println("Naming Exception:" + e.getMessage());
}
catch(JMSException e) {
  System.out.println("JMS Exception:" + e.getMessage());
}
```

7 onMessage メソッド、および関連するメッセージ管理メソッドのコードを記述します。

```
   ...
public void setMessage(String thisMessage) {
  // メッセージを theMessages Vector に追加します。
  theMessages.add(thisMessage);
}

public Enumeration getMessages() {
  // VectorをEnumeration に変換します。
  Enumeration returnThis = theMessages.elements();
  // Enumeration を返します。
  return returnThis;
}
```

```
public void onMessage(final Message message) {
  String text = null;
  try {
    text = ((TextMessage)message).getText();
    setMessage(text);
  }
  catch(JMSException e) {
    System.out.println("JMSException:" + e.getMessage());
  }
}
...
```

8 完了したら、JMS オブジェクトを閉じます。

```
// この例では、メソッドを呼び出して、オブジェクトを閉じます。
// これにより、外部クラスで、メッセージ リスナを管理できるようになります。
public void cleanUp() {
  try {
    _connection.stop();
    _subscriber.setMessageListener(null);
    _subscriber.close();
    _session.close();
    _connection.close();
    _context.close();
  }
  catch(NamingException e) {
    System.out.println("Naming exception in destroy:" +
     e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
}
```

次の例は、メッセージ リスナ クラスと onMessage メソッドを通じてトピックを取得するために必要なコードのすべてです。

```
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import java.rmi.*;

public class MyTopicListener implements MessageListener {
  // トピック属性を設定します。
  // init パラメータを介して渡されない限り、既定値を使用します。
  private String _topic;
  private String _host;

  // メッセージを保持するための変数を設定します。
  Vector theMessages = new Vector();
```

```
  // JMS で使用するオブジェクトを設定します。
  private Context _context = null;
  private TopicConnection _connection = null;
  private TopicSession _session = null;
  private TopicSubscriber _subscriber = null;


  static {
    System.setSecurityManager(new RMISecurityManager());
  }

// 複数の引数を受け入れるコンストラクタ
public MyTopicListener(String topic, String host) {
  // わかりやすくするために、パラメータ エラーのチェックを省略します。
  _topic = topic;
  _host = host;
  // JNDI で設定を行います。
  try {
    _context = new InitialContext();

    final TopicConnectionFactory factory =
      (TopicConnectionFactory)_context.lookup
        ("java:comp/env/jms/TopicConnectionFactory");
    // ファクトリを使用して TopicConnection を作成します。
    _connection = factory.createTopicConnection();
    // TopicConnection を使用して TopicSession を作成します。
    _session = _connection.createTopicSession(false,
      Session.AUTO_ACKNOWLEDGE);
    // TopicSession を使用して TopicSubscriber を作成します。
    // トピック 1 を使用して、local.properties で次のようなコードによって
        定義されます。
    // jms.topic.topic1.description=Topic for Doc Sample
    // jms.topic.topic1.display-name=DocSampleTopic
    _subscriber =
      _session.createSubscriber((Topic)_context.lookup
        ("java:comp/env/jms/" + _topic));
    // messageListener を設定します。
    _subscriber.setMessageListener(this);
    _connection.start();
  }
  catch(NamingException e) {
    System.out.println("Naming Exception:" + e.getMessage());
}
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
} // メソッドを終了します。
```

```
// 既定のコンストラクタ
public MyTopicListener() {
  // 既定のコンストラクタで使用するトピック 1 は、前述のように
  // local.properties で定義しなければなりません。
  this("topic1", "rnielsen");
}

public void setMessage(String thisMessage) {
  // メッセージを theMessages Vector に追加します。
  theMessages.add(thisMessage);
}

public Enumeration getMessages() {
  // VectorをEnumeration に変換します。
  Enumeration returnThis = theMessages.elements();
  // Enumeration を返します。
  return returnThis;
}

public void onMessage(final Message message) {
  String text = null;
  try {
    text = ((TextMessage)message).getText();
    setMessage(text);
  }
  catch(JMSException e) {
    System.out.println("JMSException:" + e.getMessage());
  }
} // メソッドを終了します。

public void cleanUp() {
  try {
    _connection.stop();
    _subscriber.setMessageListener(null);
    _subscriber.close();
    _session.close();
    _connection.close();
    _context.close();
  }
  catch(NamingException e) {
    System.out.println("Naming exception in destroy: " +
      e.getMessage());
  }
  catch(JMSException e) {
    System.out.println("JMS Exception:" + e.getMessage());
  }
} // メソッドを終了します。
} // クラスを終了します。
```

次のサーブレット例では、メッセージ リスナ クラスを使用して、パブリッシュ / サブスクライブ トピックからメッセージにアクセスしています。

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

public class MySubscriber extends HttpServlet {

  // このサーブレットは、メッセージ リスナと通信します。
  // このコンストラクタは既定のトピックを受信します。
  // トピックやホストに渡すこともできます。
  MyTopicListener thisTopicListener = new MyTopicListener();

  // 初回のみ
  // メッセージを受信するフォームを表示します。

public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {

  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();

  out.println("<html><head><title>Message Subscriber</title>");
  out.println("</head><body>");
  out.println("<h1>Message Subscriber</h1>");
  out.println("<p>Press Get Latest Topics to retrieve messages:");
  out.println("</p><hr>");
  out.println("<form action=¥"/servlet/MySubscriber¥"
        method=¥"post¥">");
  out.println("<p>");
  out.println("<input type=¥"Submit¥" value=¥"Get Latest Topics¥">");
  out.println("</form>");
  out.println("</body></html>");
}

// これは、初回を除き、すべて実行されます。
public void doPost(HttpServletRequest req, HttpServletResponse resp)
  throws IOException, ServletException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
```

```
  out.println("<html><head><title>Message Subscriber</title>");
  out.println("</head><body>");
  out.println("<h1>Message Subscriber</h1>");
  // トピックからメッセージを取得します。
  out.println("<p>");
  Enumeration enum = thisTopicListener.getMessages();
  if(!enum.hasMoreElements()) {
    out.println("No messages to retrieve");
  }
  while (enum.hasMoreElements()) {
    String name = (String)enum.nextElement();
    out.println("<hr>" + name);
  }
  out.println("<p>Press Get Latest Topics to retrieve messages:<hr>");
  out.println("<form action=¥"/servlet/MySubscriber¥"
       method=¥"post¥">");

  // form/html を完成させます。
  out.println("<p>");
  out.println("<input type=¥"Submit¥" value=¥"Get Latest Topics¥">");
  out.println("</form>");
  out.println("</body></html>");
}

public String getServletInfo() {
  return "Message Subscriber";
}

public void destroy() {
  // myTopicListener の内容をクリーンアップします。
  thisTopicListener.cleanUp();
  // myTopicListener にヌル値を設定します。
  thisTopicListener = null;
}
}
```

第 31 章

EJB クライアント の コーディング

この章では、クライアント アプリケーションから EJB にアクセスする方法について説明します。特に、この章では、サーブレットを通じて EJB にアクセスする方法を示します。

目次

- 概要 378
- 簡単なアクセス 379
- Web 認証を通じたアクセス保護 380
- カスタム認証によるアクセス保護 381

概要

EJB クライアントとなるのは、Java アプリケーション、アプレット、サーブレット、JSP、およびその他の EJB です。サーブレットおよび JSP クライアントは、クライアントがファイアウォールの外側にいるときに特に便利で、シンクライアント HTML インターフェイスに加えて EJB のすべての利点が提供されます。この章では、JRun、サーブレット、および JSP に焦点を当てて説明します。ただし、この説明はほかのタイプの EJB クライアントにも該当します。

メモ
使用する Bean は 第 27 章 で説明している方法でコーディングされ、第 35 章 で説明している方法で公開されている必要があります。

EJB にアクセスするために、クライアント コードは次のアクションを実行します。

- `InitialContext` オブジェクト を作成し、オプションで、`Properties` オブジェクトを `InitialContext` コンストラクタに渡します。
- JNDI 検索を通じて、ホーム オブジェクト実装への参照を取得します。
- ホーム オブジェクト参照で create または find メソッドを呼び出すことによって、リモート オブジェクト実装への参照を取得します。
- 1 つ以上の EJB ビジネス メソッドを呼び出します。

クライアントは、目的の EJB のセキュリティ レベルに応じてこれらのアクションを実行します。

非認証　保護されていない EJB または Bean メソッドへの簡単なアクセス。

Web 認証　サーブレットまたは JSP から EJB と Bean メソッドへの保護されたアクセス。認証とセッション トラッキングは JRun によって自動的に処理されます。

カスタム認証　EJB と Bean メソッドへの保護されたアクセス。サーブレットには、`InitialContext` インスタンスを作成する場合にプロパティとして渡すユーザ名とパスワードが必要です。

EJB クライアントの詳細については、390 ページの「クライアント アプリケーション」を参照してください。

簡単なアクセス

場合によって、非認証クライアントにアクセスするパブリック EJB やメソッドがあります。保護されていないこのタイプのクライアント アクセスは特に、開発中の EJB へのクライアント アクセスをテストするときに役立ちます。

パブリック アクセスを許可するには、Bean または Bean メソッドの `ejb.allowedIdentities` `env-entry` を `all` に設定します。これによって、クライアントは EJB を検索し、ビジネス メソッドを呼び出すことができます。`ejb.allowedIdentities` の詳細については、389 ページの「セキュリティの無効化」を参照してください。

保護されていない EJB または EJB メソッドにアクセスするには、次の手順を実行します。

1 空のコンストラクタを使用して InitialContext インスタンスを作成します。

```
Context context = new InitialContext();
```

2 EJB のホーム インターフェイスへの参照を検索します。

```
BalanceHome home =
  (BalanceHome)javax.rmi.PortableRemoteObject.narrow
    (context.lookup("java:comp/env/ejb/sample9b.BalanceHome"),
      BalanceHome.class);
```

3 EJB のホーム インターフェイスへの参照を取得します。

```
Balance balance;
int accountNumber;

// アカウント番号はフォーム フィールドから入力されます。
accountNumber = Integer.parseInt(request.getParameter(“acctnum“));
try {
  balance = (Balance)home.findByPrimaryKey(accountNumber);
}
catch(FinderException e) {
...
```

4 必要に応じて EJB メソッドを呼び出します。

```
Integer thisBalance = balance.getBalance();
```

Web 認証を通じたアクセス保護

JRun バージョン 3.1 の シングル サインオン機能により 、Web 認証を通じ て認証さ れたユーザも同じ JRun サーバーでの EJB アクセスに対して認証されます。Web アプリケーションの web.xml ファイルによって、JRun のユーザ情報の取得とユーザ認証の方法を指定します。詳細については、463 ページの第 39 章「Web アプリケーションの認証」と『JRun Version 3.1 機能および移行ガイド』を参照してください。

メモ
このタイプの EJB アクセスはサーブレットと JSP クライアントにのみ有効です。

Web 認証で動作しているサーブレットから EJB および EJB メソッドにアクセスするには、次の手順を実行します。

1 空のコンストラクタを使用して InitialContext インスタンスを作成します。

```
Context context = new InitialContext();
```

2 EJB のホーム インターフェイスへの参照を検索します。

```
BalanceHome home =
  (BalanceHome)javax.rmi.PortableRemoteObject.narrow
    (context.lookup("java:comp/env/ejb/sample9b.BalanceHome"),
      BalanceHome.class);
```

3 EJB のホーム インターフェイスへの参照を取得します。

```
Balance balance;
int accountNumber;

// アカウント番号はフォーム フィールドから入力されます。
accountNumber = Integer.parseInt(request.getParameter(“acctnum“));
try {
  balance = (Balance)home.findByPrimaryKey(accountNumber);
}
catch(FinderException e) {
...
```

4 必要に応じて EJB メソッドを呼び出します。

```
Integer thisBalance = balance.getBalance();
```

上記は379 ページの「簡単なアクセス」と同じサーブレット コーディングの手順です。しかし、この例は、Web 認証が有効で、EJB の ejb-jar.xml に固有のセキュリティが指定されていることを想定しています。

EJB サンプル 9b は、Web 認証を通じて保護された環境で EJB にアクセスする方法を示しています。

カスタム認証によるアクセス保護

また、JRun は EJB のカスタム認証のみをサポートします。既定では、このメカニズムによって、サーバーの user.properties (統合 J2EE アプリ ケーション サーバー) または deploy.properties (スタンドアロン EJB エンジン) ファイルに指定されたユーザ名とパスワードが認証されます。

このメカニズムを拡張して、サイトまたはアプリケーション固有の認証を行うことができます。詳細については、310 ページの「ロードされたユーザおよびロール」と EJB サンプル 2b を参照してください。

カスタム認証メソッドを使用して認証済みコンテキストを作成するには、クライアントは次のプロパティを持つ Properties オブジェクトを作成する必要があります。

プロパティ	値
Context.INITIAL_CONTEXT_FACTORY	allaire.ejipt.ContextFactor
Context.PROVIDER_URL	ejipt://*targetserver*:*targetportnumber*
Context.SECURITY_PRINCIPAL	ユーザ名
Context.SECURITY_CREDENTIALS	パスワード

次のようなコードを使用して、Properties オブジェクトを InitialContext コンストラクタに渡します。

```
...
Properties properties = new Properties();
properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,
  "allaire.ejipt.ContextFactory");
properties.setProperty(Context.PROVIDER_URL,
  "ejipt://targetserver:targetportnumber");
properties.setProperty(Context.SECURITY_PRINCIPAL, username);
properties.setProperty(Context.SECURITY_CREDENTIALS, password);
Context context = new InitialContext(properties);
```

380 ページの「Web 認証を通じたアクセス保護」に示すように、クライアントは、ホーム インターフェイスへの参照を検索することによって引き続きリモート インターフェイスへの参照にアクセスし、ビジネス メソッドを呼び出します。

カスタム認証を使用する EJB にサーブレット や JSP がアクセスするときは、次のよう に、1 つの要求 (要求ログイン) または 1 つのセッション (セッション ログイン) としてログインできます。

- **要求ログイン** EJB コンテキストは 1 つの要求 (`doPost` メソッドなど) の場合に存在します。後続の要求には、もう 1 つの `InitialContext` と再認証を確立する必要があります。
- **セッション ログイン** EJB コンテキストは後続の要求に使用するために保存されます。セッション ログインを使用するには、次の手順を実行します。
  - 次のようなコードを使用して、`allaire.jrun.ejbContext` セッション変数にコンテキスト インスタンスを保存します。

```
...
// プロパティ変数内の値については、
// 前述のコード例を参照してください。
Context context = new InitialContext(properties);

HttpSession session = request.getSession(true);
session.putValue("allaire.jrun.ejbContext", context);
```

  - 後続の要求では、次の例のようにセッション変数から EJB コンテキストにアクセスします。

```
HttpSession session = request.getSession(true);
Context context =
  (Context)request.getSession().getValue("allaire.jrun.ejbContext");
```

  - セッションが終了したら、次の例のようにセッションを無効にします。

```
...
if (request.getParameter("Logout") != null) {
// ログアウト ボタンを押すと、そのセッションが無効になります。
  session.invalidate();
..
```

カスタム認証環境でのセッション ログインの例については、EJB サンプル 9a を参照してください。

第 32 章

高度なテクニック

目次

- 概要 384
- トランザクション 384
- その他の高度な EJB トピック 389
- クライアント アプリケーション 390

概要

この章では、Bean 開発のより高度な機能について説明します。そのため、この章で取り上げるトピックは、ほかの部分では説明しない、さまざまな特徴と利用可能な機能に触れています。

この章で説明するトピックは JRun 関連の説明に限定されており、EJB 仕様については説明していません。また、この章で取り上げる内容について読者が熟知していることを前提としています。

トランザクション

EJB エンジンは、X/Open XA 仕様に基づいて 2 フェーズ コミット トランザクション管理を完全にサポートしています。EJB エンジンは、フラット トランザクションをサポートしていますが、ネストされた (子) トランザクションはサポートしていません。

メモ
JRun では、トランザクションの実行中に新しいトランザクションを開始できますが、これは真のネスト トランザクションではありません。つまり、外部トランザクションの後のロールバックは内部トランザクションのロールバックとはなりません。

コンテナ管理トランザクション (暗黙的トランザクション、CMT) とクライアントまたは Bean 管理トランザクション (明示的トランザクション、BMT) が両方ともサポートされています。

トランザクション属性の設定

EJB を公開する場合は、公開記述子に指定されているように、トランザクション属性によりトランザクションに必要なものを指定します。JRun では、`container-transaction` 要素を使用してトランザクション属性を設定できます。

トランザクション属性は次のとおりです。

- NotSupported　このトランザクション属性を使用してメソッドを呼び出すと、コンテナは既存のトランザクションを停止します。
- Supports　このトランザクション属性を使用してメソッドを呼び出すと、コンテナは、属性があればトランザクションに含めます。
- Required　このメソッドはトランザクション内で呼び出す必要があります。現行のトランザクションがなく、メソッドで CMT を使用している場合、コンテナは新しいトランザクションを開始します。
- RequiresNew　このメソッドを呼び出すと、コンテナは常に新しいトランザクションを開始します。メソッドが終了してログ ファイルに警告メッセージが書き込まれるまで、コンテナは既存のトランザクションを停止します。

- Mandatory　このメソッドはトランザクション内で呼び出す必要があります。現在トランザクションがない場合は、コンテナから例外が返されます。
- Never　このメソッドはトランザクション内で呼び出さないでください。このメソッドをトランザクション内で呼び出すと、コンテナから例外が返されます。

あるメソッドが特定のトランザクション動作を必要とする一方で、同じクラスの別のメソッドが異なるトランザクション動作を必要とする場合があります。メソッドレベルのトランザクション属性は Bean の仕様に従わず、

methodname.ejb.transactionAttribute の形式で env-entry を使用します。次の EJB サンプル 4b からの例では、ejb-entries を使用してトランザクション属性を指定します。

```
<env-entry>
  <env-entry-name>spend.ejb.transactionAttribute</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>required</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>save.ejb.transactionAttribute</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>mandatory</env-entry-value>
</env-entry>
```

次の例は、<container-transaction> 要素における同等の設定を示します。

```
...
<assembly-descriptor>
<container-transaction>
  <method>
    <ejb-name>BalanceBean</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Supports</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>BalanceBean</ejb-name>
    <method-name>save</method-name>
  </method>
  <trans-attribute>Mandatory</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>BalanceBean</ejb-name>
    <method-name>spend</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
...
</assembly-descriptor>
```

これらの設定の結果、save メソッドには Mandatory が、spend メソッドには Required が設定されます。

トランザクション属性はエンティティ Bean の create、find、および remove メソッドに付加することもできます。属性を設定するときは、リモートおよびホーム インターフェイスに表示される名前を使用する必要があります。

env-entry の有効な属性の一覧については、JRun JavaDocs ファイルに付属する EjiptProperties API マニュアルを参照してください。すべての公開記述子エントリの一覧については、EJB バージョン 1.1 の仕様書を参照してください。

コンテナ管理トランザクション

CMT を使用すると、Bean に代わってコンテナがトランザクションを開始してコミットまたはロールバックします。CMT を利用するには、Bean (またはメソッド) のトランザクション属性を次のいずれかに設定する必要があります。

- required
- RequiresNew
- Mandatory

例外のタイプ

コンテナが CMT の例外を処理する方法は、次のように、例外がアプリケーション例外 (確認済みの例外) かシステム例外 (未確認の例外) かによって異なります。

- **アプリケーション例外** java.lang.RuntimeException または java.rmi.RemoteException を拡張する例外を除くすべての例外。アプリケーション例外が発生すると、コンテナからクライアントに例外が返され、次のように、クライアント トランザクションがロールバックされる場合とされない場合があります。
  - EJB メソッドをクライアントと同じトランザクションで実行すると、コンテナはロールバックしないで再び例外を返します。
  - EJB メソッドを独自のトランザクションで実行すると、コンテナはインスタンスが setRollbackOnly メソッドを呼び出したかどうかをチェックします。呼び出していれば、コンテナはトランザクションをロールバックして再び例外を返します。呼び出していなければ、再び例外を返します。
- **システム例外** java.lang.RuntimeException または java.rmi.RemoteException (EJBException を含む) を拡張するすべての例外。コンテナによって EJB メソッドがロールバックされ、次のように、クライアント トランザクションがロールバックされる場合とされない場合があります。
  - EJB メソッドをクライアントと同じトランザクションで実行すると、コンテナはクライアント トランザクションをロールバックし、ロールバックされた例外を返します。
  - EJB メソッドをクライアントとは異なるトランザクションで実行すると、コンテナは、EJB メソッドのトランザクションをロールバックしますが、クライアント トランザクションはロールバックしないで、RemoteException を返します。

例外と CMT のロールバックの関係の詳細については、EJB バージョン 1.1 の仕様書のトランザクションに関する章を参照してください。

ロールバック

あるトランザクションに関係したメソッドから例外が返されると、サーバーからも例外が返され、トランザクションが自動的にロールバックされる場合があります。メソッドで発生した最初の例外はクライアントに対して返されます。

Bean およびクライアント管理トランザクション

このトランザクション管理によって、トランザクションを完全に制御し、クライアントから管理できます。セッション Bean の場合は、Bean 自体から管理できます。

クライアント区分トランザクション

クライアントはトランザクションを明示的に開始するか、コミットするか、またはロールバックすることを選択できます。次のスニペットは、クライアント区分トランザクションを示します。明示的な呼び出しによって、トランザクションを開始して、コミットまたはロールバックすることに注意してください。

```
public void save(final int amount) {
  try{
  final UserTransaction transaction =
      (UserTransaction)_context.lookup
        ("javax.transaction.UserTransaction");

  transaction.begin();
    try {
      _balance.save(amount);
    }
    catch (Exception exception){
   transaction.rollback();
      throw exception;
    }
  transaction.commit();
    return;
  }
  catch (RemoteException remote){
    throw new RuntimeException(remote.toString());
  }
  catch (Exception exception){
    throw new RuntimeException(exception.toString());
  }
}
```

また、トランザクション内で呼び出されるメソッドについて、適切なトランザクションを設定する必要があります。

Bean 管理トランザクション

Bean 管理トランザクション (BMT) はセッション Bean にのみ使用でき、エンティティ Bean では使用できません。BMT を使用する場合、次のように、Bean のプロパティファイルにトランザクション属性を設定する必要があります。

```
<transaction-type>Bean</transaction-type>
```

次のスニペットは、どのようにメソッドを構築するかを示します。

```
public void spend(final int value) throws RemoteException {
  if (_value - value < _min){
    throw new RuntimeException("Min limit reached");
  }
  if (_balance != null){
  final UserTransaction transaction = _context.getUserTransaction();
  transaction.begin();
    try {
      _balance.spend(value);
    }
    catch (final RemoteException remote){
   transaction.rollback();
      throw remote;
    }
  transaction.commit();
  }
  _value -= value;
}
```

ロールバック

アプリケーション例外が返されると、明示的トランザクションは自動的にロールバックされません。ただし、未確認の例外の場合は、明示的なトランザクションにロールバックのマークが付けられ、`TransactionRolledbackException` が返されます。

その他の高度な EJB トピック

デッドロック

EJB エンジンには XA 準拠のロック メカニズムが実装されているので、メソッド の呼び出し時のトランザクションと並行処理管理のために EJB オブジェクトがロックされます。EJB エンジンには、トランザクション内の呼び出しやチェーン内の呼び出しで発生する可能性のあるデッドロックに対応する効率的なデッドロック検出メカニズムも実装しています。デッドロックが検出されると、EJB エンジンはデッドロックを解除し、影響を受けたトランザクションをロールバックするか、またはトランザクション内の呼び出しでない場合は `allaire.ejipt.DeadlockException` を返します。

セキュリティの無効化

EJB セキュリティ チェックを無効にするには、`ejb.allowedIdentities` を定義して `all` に設定する `env-entry` を追加します。これによって、認証されたユーザも未知のユーザも Bean またはメソッドにアクセスできます。EJB セキュリティを使用しない場合は、常にこの `env-entry` を指定してセキュリティ チェックを無効にします。`ejb.allowedIdenties` を `all` に設定すると、すべての呼び出し者が Bean のすべてのメソッドにアクセスできるようになります。`env-entry` の前にメソッド名を付けると、すべての呼び出し者は、そのメソッドにのみアクセスできます。

SSL

Java 2 に準拠した SSL パッケージ、つまりクライアント / サーバー RMI ソケット ファクトリを使用している SSL パッケージであれば、ホーム / オブジェクト ソケット ファクトリ プロパティでクラス名を指定するだけで、EJB エンジンと統合できます。

ローカル Bean

クライアントが JNDI コンテキストを要求すると、使用可能なすべてのホーム オブジェクトの参照がサーバーによってクライアントに送信されます。このため、すべてのクライアントは、サーバーに公開されているすべてのリモートおよびホーム インターフェイスにアクセスできます。

Bean をローカル専用モードで公開することもできます。この場合、ホーム オブジェクトは、どのクライアントにもエクスポートされず、サーバーで同じ場所に配置されている Bean からのみアクセスできます。Bean をローカルとして指定するには、次の env-entry を Bean の公開記述子に含めます。

```
<env-entry>
  <env-entry-name>ejipt.isLocal</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>true</env-entry-value>
</env-entry>
```

ローカル Bean はリモート リソースを消費しません。エンティティ Bean をセッション Bean でラップする場合は、エンティティ Bean をローカルとして指定し、インターフェイスがエクスポートされるのを防ぎます。

クライアント アプリケーション

セッション スコープ

ejipt.sessionScope プロパティはクライアント側プロパティで、ログイン セッションのスコープを指定します。これは、マルチスレッド クライアント のログイン セッションを管理する場合に役立ちます。このプロパティは、クライアントが起動され、サーバーへの接続が確立していないときに、指定する必要があります。

生成されたスタブによって、クラスの初期化時にシステム プロパティからプロパティの設定が選択されます。このプロパティは、コマンド ラインから指定するか、または引数としてプロパティの名前と値を使用して System.setProperty メソッドを呼び出して指定できます。サーバーへの接続が確立する前に、システム プロパティにプロパティを設定する必要があります。設定は変更できません。

クライアントでスタブ クラスがロードされる前にプロパティを指定する必要があります。クライアントは、正しい証明を持つ JNDI コンテキストを作成することによってログインします。

sessionScope の有効値は次のとおりです。

- thread　ログインを取得したスレッドのみがアクセス権を持ちます (既定)。
- thread_group　グループ内の 1 つのスレッドがログインすると、グループ内のほかのすべてのスレッドもログインしたと見なされます。
- vm　1 つのスレッドがログインすると、VM (プロセス) 内のすべてのスレッドもログインしたと見なされます。

クライアント接続の定義

サーバーは、呼び出し側に基づいて、受信した各呼び出しに固有の**呼び出し ID** を割り当てます。ejipt.sessionScope プロパティを vm に設定すると、クライアント VM からのすべての呼び出しは、VM が呼び出しに使用するスレッドの数に関係なく、同じ呼び出し ID と関連付けられます。プロパティを thread に設定すると、VM 内の異なるスレッドによる呼び出しは、異なる呼び出し ID と関連付けられます。異なる VM からの呼び出しを同じ呼び出し ID と関連付けることはできません。

セカンダリ initialcontext をインスタンス化し、ejb.sessionScope を vm に設定する場合は、最初のログインと同じユーザとパスワードを使用してログインしてください。異なるユーザとパスワードを使用してサーバーへのセカンダリ ログイン呼び出しを行うと、サーバーで「ログイン済み」例外が生成されます。

JNDI からログインするたびに、新しい独自の有効期限を持つログイン セッション オブジェクト が作成されます。EJB エンジンは、呼び出し ID ごとにログイン数を追跡し、最後のログインがログアウトまたは期限切れになった場合にのみ呼び出し ID を解放します。

接続数の制限

一部の JRun のバージョンではユーザ接続数が制限されています。たとえば、JRun で EJB 接続数が 3 つに制限されていると、EJB エンジンで一度に最大 3 つの異なる呼び出し IDと関連付けられている呼び出しが処理されます。呼び出し ID は保存されません。つまり、サーバーは一度に 3 つの呼び出しのセットを処理し、次に別の 3 つの呼び出しのセットを処理します。

ただし、「ログインした」呼び出しの場合は例外です。ログイン時、ID はユーザ ID とともにテーブルに登録されます。同じ呼び出し ID に関連付けられているすべての呼び出しは、ユーザがログアウトするか、またはログイン セッションが期限切れになるまで、ログインしたユーザのセキュリティ コンテキスト内で行われます。ログアウトすると、登録が削除され、別の新しいユーザがログインできます。もちろん、すでにログインした呼び出しは、ログアウトしないと、再度ログインできません。

`ejipt.sessionScope` プロパティの値によって、ユーザは、分布範囲の一方の終端で 3 つの異なる VM (クライアント プロセス) としてユーザを定義し、もう一方の終端では同じ VM 内の 3 つの異なるスレッドとして定義できます。繰り返しますが、これは 100 の異なるクライアント VM 呼び出しができないということではなく、EJB エンジンで一度に処理できる呼び出しが 3 つに制限されているということです。

クライアントでセキュリティを使用すると (ログインすると)、一度にログインできるのは 3 人のユーザだけです。ユーザがログアウトしてから次のユーザがログインできるまでに数秒かかります。ユーザ スロットは、前にログインしたユーザがログアウトした場合と、期限切れになった場合にのみ解放されます。

第 33 章

EJB エンジンの使用

目次

- 概要 394
- クラスのロード 394
- クラスパス 394
- スタンドアロン モードでの EJB エンジンの動作 395
- セットアップのトラブルシューティング 396

概要

この章は、EJB エンジンの使い方についての理解を助けることを目的としています。たとえば、EJB エンジンをスタンドアロン モードで実行する方法や、トラブルシューティングのヒントなどについて説明します。

クラスのロード

JRun を起動すると、ejipt_exports.jar が /runtime ディレクトリにコピーされます。JRun クラス サーバーは、常に /runtime ディレクトリから ejipt_exports.jar を取得し、RMI ダイナミック クラス ローダを介してこの JAR を要求元のクライアントに配布します。クラス サーバーは専用 HTTP サーバーです。JRun からエクスポートされたすべてのクラス (スタブを含む) のコードベースが、クラス サーバーの HTTP の URL に自動的に設定されます。

ejipt_exports.jar はクラス サーバーだけに使用されるため、この JAR ファイルをクライアントのクラスパスに含める必要はありません。ただし、唯一の例外として、ダイナミック クラス ローダを使用しない場合は、この JAR ファイルをすべてのクライアントに手作業でコピーし、クライアントのクラスパスに明示的に追加する必要があります。

JRun では、すべてのスタブがエクスポート JAR ファイルに挿入されます。その結果、RMI クラス ローダを使用すると、JAR ファイルだけがクライアントからダウンロードされます。

クライアント 側でアプレット を使用すると 、RMIクラス ローダは必要ではなく 、エクスポート JAR ファイルを複数の JAR ファイルに分割できます。ただし、default_exports.jar 内のクラスは、作成されるすべての JAR ファイルに含まれている必要があります。既定の Bean を使用しない場合は、default_exports.jar の不明瞭なクラスだけを各種のエクスポート JAR ファイルに追加してください。

クラスパス

サーバーの CLASSPATH 環境変数は設定しないようにしてください。標準拡張 JAR ファイル (ejb.jar、jdbc.jar、jms.jar、jndi.jar、および jta.jar) はすべて JRUN_HOME/lib/ext ディレクトリにインストールします。Bean JAR ファイルと、生成されたオブジェクト JAR ファイルはすべて JRun によって自動的にロードされます。CLASSPATH の内容には依存しません。

クライアントでは、プロジェクト固有のクライアント JAR ファイルとともに、ejipt_client.jar をクライアントの CLASSPATH に追加します。プロジェクトのクライアント JAR ファイルには、公開された Bean のリモート インターフェイスとホームインターフェイスも、これらのインターフェイスで使用される追加クラスとともに格納されている必要があります。

スタンドアロン モードでの EJB エンジンの動作

EJB エンジンをスタンドアロン モード (Web アプリケーション、JSP、およびサーブレットを処理できないモード) で実行できます。EJB エンジンをスタンドアロン モードで起動するには、次のコマンドを入力します。

```
cd JRun のインストール ディレクトリ
java -Djava.security.policy=lib/jrun.policy -classpath lib/ejipt.jar
allaire.ejipt.Ejipt
```

EJB エンジンが起動し、コマンド プロンプトを表示します。次のコマンドを使用できます。

- q (quit)　EJB エンジンを停止します。
- l (load)　`/runtime/classes` ディレクトリから Bean 実装を動的に読み取ります。動的な Bean 読み取りの詳細については、422 ページの「Bean のダイナミック ローディングの使用」を参照してください。

また、EJB エンジンをスタンドアロン モードで再起動することもできます。スタンドアロン モードで再起動すると、EJB エンジンは強制的に `/runtime` ディレクトリのファイルだけを使用して起動されます。`/deploy` ディレクトリ の内容は使用されません。EJB エンジンをスタンドアロン モードで起動するには、次のコマンドを入力します。

```
cd JRun のインストール ディレクトリ
java -Djava.security.policy=lib/jrun.policy -classpath lib/ejipt.jar
allaire.ejipt.Ejipt -restart
```

セットアップのトラブルシューティング

Bean を公開し、JRun を起動した後は、クライアントからサーバーに接続できます。接続できない場合は、次の一覧を再確認して、インストールが正しく行われていることを確認します。

許可

サーバーに接続するときは、クライアント VM に接続のアクセス権が必要です。このタスクを処理する最も簡単な方法は、`-Djava.security.policy=`*policyfile*`.policy` クライアントを起動するときのコマンドライン引数として指定することです。ポリシー ファイルの例については、『JRun サンプル ガイド』を参照してください。

標準拡張

JRun 以外の JAR ファイル (`ejb.jar`、`jdbc.jar`、`jms.jar`、`jndi.jar`、`jta.jar` など) はすべて、標準拡張として JRun サーバーとクライアントの両方にインストールする必要があります。いずれかの JRun JAR ファイルを標準拡張として不用意にインストールすると、セキュリティ例外が発生します。`JRUN_HOME/lib/ext` ディレクトリのファイルだけを標準拡張としてインストールするように注意してください。

サーバー クラスパス (スタンドアロン EJB エンジン)

サーバー側では、CLASSPATH 環境変数を空白のままにし、`ejipt.jar` とコマンドライン クラスパスのデータベース ドライバ JAR ファイルだけを使用します。

クライアントのセットアップ

最新の `ejipt_client.jar` をクライアントにコピーします。`ejipt_client.jar` だけを、アプリケーションの クライアント JAR ファイルとともにクライアントのクラスパスに使用してください。なお、アプリケーションのクライアント JAR ファイルには、Bean のリモート インターフェイスおよびホーム インターフェイスが含まれている必要があります。

ほかの EJB エンジン JAR ファイル (`ejipt_exports.jar`、`ejipt_objects.jar`、`default_xxx.jar`) は、コピーしたり操作したりする必要はありません。また、暗黙的または明示的にクラスパスに指定する必要もありません。これらの JAR ファイルは EJB エンジンによって自動的に内部検索され、`ejipt_exports.jar` (スタブを含む) が接続時にクライアントにダウンロードされます。ただし、例外として、JDK 1.1 クライアントを使用する場合はこのルールが適用されません。JDK 1.1 クライアントを使用する場合の詳細な手順については、『JRun サンプル ガイド』の「JDK 1.1 クライアント」を参照してください。

第 5 部

アプリケーションの公開

開発とテストが完了したら、アプリケーションを公開できます。ここでは、Web アプリケーション、EJB、および Web アプリケーションと EJB の両方を含む J2EE アプリケーションの公開方法について説明します。

Web アプリケーションの アセンブルと公開 399
Enterprise JavaBeans の公開 409
J2EE アプリケーションの公開 425

第 34 章

Web アプリケーションのアセンブルと公開

この章では、Web アプリケーションのアセンブルおよび公開に関する概念とタスクについて説明します。

目次

- 概要 400
- 公開用 Web アプリケーション パッケージの作成 402
- Web アプリケーションの公開 404

概要

Java サーブレット API バージョン 2.2 の仕様書では、アプリケーションのアセンブル担当者と公開担当者の役割を定義しています。アプリケーション アセンブル担当者は、アプリケーション開発者が作成したリソースを受け取り、公開可能な WAR ファイルに変換します。EJB の公開については、409 ページの第 35 章「Enterprise JavaBeans の公開」を参照してください。完全な J2EE アプリケーションの公開については、425 ページの第 36 章「J2EE アプリケーションの公開」を参照してください。

これらの役割の実装は、サイトによって異なる場合があります。サイトによっては、アプリケーション開発者、アセンブル担当者、および公開担当者の役割を 1 人で受け持つ場合があります。また、これらの役割が別のグループの Java 開発者、JSP 開発者、アプリケーション アセンブル担当者、および公開担当者に割り当てられているサイトもあります。

メモ
この章では、Java サーブレット API バージョン 2.2 の仕様書に記載されているガイドラインおよびディレクトリ構造を使用して、複数の Web アプリケーションの開発と公開を行うことを想定しています。

Web アプリケーション アセンブルとは

アプリケーション開発者は、サーブレット、JSP ページ、タグ ライブラリ、HTML ファイル、およびアプリケーションの開発とテストに必要なその他のすべての要素をコーディングします。アプリケーション アセンブル担当者は、アプリケーション開発者が作成したアプリケーションを公開可能な Web アプリケーションに変換します。

アプリケーション アセンブル担当者が WAR ファイルを作成する前に、Java 開発者、JSP 開発者、および QA スタッフは、Web アプリケーションの開発に関する次のタスクを完了する必要があります。

タスク	参照先
サーブレットの開発	第 3 部
JSP ページの開発	第 2 部
タグ ハンドラ、TLD ファイル、および TEI クラスの開発	第 22 章
タグ ライブラリおよびその他のユーティリティ クラス用 JAR ファイルの作成	第 22 章
セキュリティおよび計測用のプロパティ ファイルの修正	第 39 章 および 第 40 章
JSP ページで使用する JavaBeans の実装	第 34 章
サーブレットの定義、サーブレットのマッピング、初期化パラメータ、MIME タイプ、およびその他の環境設定の作成	『JRun セット アップ ガイド 』
JMC または手作業による `web.xml` ファイルの作成	『JRun セット アップ ガイド 』

タスク	参照先
Web アプリケーション構造に対応した、集中型のソース管理レポジトリにおけるソース コードとクラス ファイルの管理	サイト固有
完全なテストの実行	サイト固有

アプリケーション アセンブルの詳細については、402 ページの「公開用 Web アプリケーション パッケージの作成」を参照してください。

Web アプリケーションの公開とは

公開担当者は、JMC または JRun WarDeploy ユーティリティとアプリケーション アセンブル担当者が作成した WAR ファイルを使用して、Web アプリケーションを特定の運用環境にインスト ールします。WAR ファイルのインスト ールのほかに、公開担当者は、必要に応じてアプリケーションを運用環境に合わせて設定します。たとえば、公開担当者はセキュリティ ロールをサイト固有のユーザおよびグループにマッピングします。

詳細については、404 ページの「Web アプリケーションの公開」を参照してください。

WAR ファイル

通常、Web アプリケーションは、単一の圧縮 WAR ファイルとして配布します。WAR ファイルには、すべてのディレクトリ構造とアプリケーションを定義するすべてのファイルが含まれています。WAR ファイルは、JAR ファイルと同じツールを使用して作成します。

WAR ファイルは、JMC または JRun Deploy ツールを使用して公開します。これらのツールは、WAR ファイルとサーバー固有のパラメータのセットを受け入れることにより、必要に応じてディレクトリ構造を拡張したり、設定やプロパティ ファイルを更新できます。

公開中、JRun は WAR ファイルを変換し、指定された JRun サーバーで新しいアプリケーションを定義します。WAR ファイルの変換は、JRun を公開するためのメカニズムです。サーバーによって公開条件が異なる場合があります。たとえば、データベース ベンダの実装では、WAR ファイルの内容をデータベースに挿入しなければならない場合があります。

メモ

EJB は、JAR ファイルを使用して公開します。J2EE エンタープライズ アプリケーションをパッケージ化する場合、EJB の JAR ファイルおよび Web アプリケーションの WAR ファイルを、J2EE エンタープライズ アーカイブ (EAR) ファイルの一部としてパッケージ化できます。詳細については、第 35 章および第 36 章を参照してください。

公開用 Web アプリケーション パッケージの作成

アプリケーション アセンブル担当者は、開発結果を公開可能な Web アプリケーションに変換します。Web アプリケーション アセンブル プロセスへの入力には、次の内容が含まれます。

- サーブレット、ユーティリティ、タグ ハンドラ、および TEI クラス用 Java `.class` ファイル
- サードパーティ製のクラスおよびライブラリ
- タグ ライブラリの JAR ファイルに含まれていない場合の TLD ファイル
- `web.xml` ファイル
- アプリケーション固有のプロパティ ファイル
- JavaDocs および使用に関する注意事項を含むマニュアル

アプリケーション アセンブル プロセスからの出力には、次の内容が含まれます。

- 1 つの WAR ファイル
- インストール時の注意事項
- 設定ガイドライン

メモ
サポートされているハードウェア プラットフォーム、オペレーティング システム、Web サーバー、および JVM のすべての組み合わせを使用して、公開可能な WAR ファイルを十分にテストしてください。

JSP ページのコンパイルの無効化

パフォーマンスとセキュリティ上の理由から、JSP ページのコンパイルを無効にして、テキスト ベースの `.jsp` ファイルではなく、バイナリ の `.class` ファイルのみを配布するように指定することもできます。このプロセスは、次の 2 つの手順で構成されます。

- ダイナミック JSP コンパイルの無効化
- JSPC コンパイラを使用した JSP ページのプレコンパイル

これらのタスクの詳細については、第 10 章を参照してください。

WAR ファイルの作成

WAR ファイルは、Java `jar` ユーティリティを使用して作成します。このユーティリティの構文は、次のとおりです。

```
jar options output-file input-files
```

options

次のいずれかまたは複数のオプションを指定します。

- `c` (create)　`.war` ファイルを作成します。
- `f` (file)　`stdout` に出力せず、ファイルを作成します。
- `v` (verbose)　WAR ファイルに追加されたファイルの名前を表示します。
- `0` (zero)　圧縮を実行しません。
- `M` (manifest)　既定のマニフェスト ファイルを作成しません。

output-file

`jar` ユーティリティによって生成されるファイルの名前を指定します。必ず `.war` 接尾辞を使用してください。

input-files

WAR ファイルに追加するファイルをスペースで区切ったリスト。複数のファイルを追加するには、ワイルドカード文字 (*) を使用します。ディレクトリを指定すると、`jar` ユーティリティはサブディレクトリも一緒に追加します。

`jar` ユーティリティの詳細については、Sun 社の Web サイトにアクセスし、*jar* を検索して情報を参照してください。

次の例では、現在のディレクトリを *JRun のインストール ディレクトリ*`/servers/default/default-app` と想定して、JRun 既定のアプリケーションの WAR ファイルを作成します。

```
jar -cf default.war *.*
```

Web アプリケーションの公開

公開担当者は JRun Deploy ツールを使用して、Web アプリケーションをインストールします。JRun Deploy ツールは、JMC またはコマンド ライン インターフェイスから実行できます。

Web アプリケーション公開プロセスの内容は次のとおりです。

- WAR ファイルのディレクトリ構造への展開
- Web アプリケーションの JRun サーバーへの追加
- Web アプリケーションに対するアプリケーション マッピングの定義

詳細については、82 ページの「アプリケーション マッピング」を参照してください。

公開プロセスを開始する前に、次の内容を確認してください。

- WAR ファイルの位置とディレクトリの絶対パス
- 公開した Web アプリケーションを含む JRun サーバーの名前
- Web アプリケーション名 (JRun は、JMC 中およびログに記録するときにこの名前を使用します。)
- アプリケーション ホスト
- この Web アプリケーションにアクセスするためにクライアントにより使用される URL 接頭辞
- Web アプリケーションの公開先ディレクトリの絶対パス（このディレクトリは、必ず常に存在する必要があります。同じ名前のファイルが存在する場合、そのファイルは公開プロセスによって上書きされます。)

JMC の使用

JMC を使用して Web アプリケーションを公開するには、次の手順を実行します。

1 JMC にログオンします。

2 Web アプリケーションの公開先サーバーを展開します。

3 [Web アプリケーション] を展開せずにクリックします。

4 右側ペインで、[アプリケーションの公開] をクリックします。

5 情報を指定します。

6 [公開] をクリックします。

詳細については、『JRun セットアップ ガイド』を参照してください。

コマンド ライン インターフェイスの使用

Web アプリケーション Deploy ツールのコマンド ライン インターフェイスでは、一連のパラメータを使用して WAR ファイルを公開できます。インターフェイスは、プロパティ ファイルおよびコマンド ラインからパラメータを受け入れます。

このセクションでは、コマンド ライン インターフェイスで使用される構文およびプロパティ ファイルの形式について説明します。

構文

WarDeploy ユーティリティを使用すると、Web アプリケーションの公開、削除、および再公開を行うことができます。このユーティリティは、`jrun.jar` ファイルに含まれています。次の構文を使用します。ここでは、`jrun.jar` が現在のシステム クラスパスであると想定しています。

```
java allaire.jrun.tools.WarDeploy -deploy -config=公開用プロパティ ファイル
java allaire.jrun.tools.WarDeploy -remove -config=削除用プロパティ ファイル
java allaire.jrun.tools.WarDeploy -redeploy -config=再公開用プロパティ ファイル
```

プロパティ ファイルの形式

公開を実行するためのプロパティ ファイルの形式は、次のとおりです。

- `deploy.war.path=` *.war ファイルへの絶対パス*
- `deploy.server.name=default |` *サーバー名*
- `deploy.webapp.name=` *アプリケーション名*
- `deploy.context.path=/` *コンテキスト パス*
- `deploy.webapp.rootdir=`*Web アプリケーションのルート ディレクトリ (絶対パス)*
- `deploy.jrun.rootdir=`*JRun のルート ディレクトリ (絶対パス)*

次の例は、公開プロパティを持つファイルを示します。

```
deploy.war.path=c:¥¥testdeploy¥¥rds-app.war
deploy.server.name=default
deploy.webapp.name=testdeploy
deploy.context.path=/testdeploy
deploy.webapp.rootdir=c:¥¥testdeploy
deploy.jrun.rootdir=c:¥¥program files¥¥allaire¥¥jrun
```

メモ
パス名の円記号 (¥) は、2 つの円記号 (¥¥) でエスケープ処理する必要があります。

Web アプリケーションの削除または再公開のためのプロパティ ファイルの形式は次のとおりです。

- `deploy.server.name=default | ` *サーバー名*
- `deploy.webapp.name=` *アプリケーション名*
- `deploy.jrun.rootdir=`*JRun のルート ディレクトリ (絶対パス)*

次の例は、削除または再公開に必要なプロパティを持つファイルを示します。

```
deploy.server.name=default
deploy.webapp.name=testdeploy
deploy.jrun.rootdir=c:¥¥program files¥¥allaire¥¥jrun
```

認証のためのユーザとロールの定義

認証のために、公開担当者は公開する Web アプリケーションのユーザとロールを定義します。詳細については、第 39 章を参照してください。

ホット デプロイおよびオート デプロイの使用

JRun バージョン 3.1 では、Web アプリケーションの動的な更新および自動的な公開が導入されました。この機能には、次のような特徴があります。

- ホット デプロイ - 既存の Web アプリケーション構造に対する物理的変化 (WAR ファイルの更新や web.xml ファイルへの変更など) を検出すると、実行中の Web アプリケーションを自動的に再起動します。
- オート デプロイ - 新しい WAR ファイルを検出すると、新しい Web アプリケーションを自動的に公開します。

ホット デプロイおよびオート デプロイを有効にすれば、Web アプリケーションを変更したり、新しい Web アプリケーションを追加するたびに JRun サーバーを再起動する必要がありません。ただし、JDBC データ ソースの追加など、サーバー レベルでの変更の後は、JRun サーバーを再起動しなければならないことに注意してください。

メモ
この機能は、実際の運用環境では無効にしておくことを特にお勧めします。

JRun 3.1 で実装する Web サーバーおよびアプリケーションでは、この設定を次のように使用しています。

- admin server - 無効 (`admin/local.properties` ファイルは `webapp.hotdeploy.enabled=false` と指定します。)
- jmc-app (admin server) - 無効 (`admin/jmc-app/WEB-INF/webapp.properties` ファイルは `webapp.hotdeploy.enabled=false` と指定します。)
- default application - 有効 (この指定は `global.properties` ファイルの `webapp.hotdeploy.enabled=true` 指定から継承されます。)
- default server - 有効 (この指定は `global.properties` ファイルの `webapp.hotdeploy.enabled=true` 指定から継承されます。)

- default-app - 有効 (この指定は global.properties ファイルの webapp.hotdeploy.enabled=true 指定から継承されます。)
- demo-app - 有効 (この指定は global.properties ファイルの webapp.hotdeploy.enabled=true 指定から継承されます。)
- invoice-app - 有効 (この指定は global.properties ファイルの webapp.hotdeploy.enabled=true 指定から継承されます。)

ホット デプロイ

ホット デプロイは、global.properties および local.properties にある次のプロパティを使用して制御します。

- webapp.hotdeploy.enabled=true|false - ホット デプロイを有効または無効にします。
- webapp.hotdeploy.interval=interval - JRun が変更を確認する間隔を秒単位で指定します。既定値は 5 秒です。この値をゼロに設定すると、ホット デプロイ サービスは無効になります。
- webapp.hotdeploy.onchange=**ディレクトリおよびファイルのカンマ区切りリスト** - JRun が監視するディレクトリおよびファイルを指定します。これらのファイルのいずれかが変更されると、JRun は自動的に Web アプリケーションを再起動し、ログ ファイルに「hot deploy initiating」というメッセージを書き込みます。global.properties の既定の設定は、次のとおりです。

  ```
  webapp.hotdeploy.onchange={webapp.rootdir}/WEB-INF/web.xml,
   {webapp.rootdir}/WEB-INF/webapp.properties,
   {webapp.rootdir}/WEB-INF/lib
  ```

- webapp.hotdeploy.defaultname=default-app war file - 既定のコンテキスト (/ の URL) にマッピングされる WAR ファイルの名前を指定します。既定値は default.war です。ほとんどの場合、Web アプリケーションは /webappname をコンテキストとして使用しますが、既定のアプリケーションは単純にスラッシュを使用します。詳細については、79 ページの第 6 章「JRun によるサーブレットへの要求のマッピング」を参照してください。

変更されたファイルが WAR ファイルの場合、JRun では以前に定義されたルート ディレクトリと URL マッピングを使用して Web アプリケーションを再公開し、再起動します。変更されたファイルが WAR ファイルでない場合、JRun では Web アプリケーションを再起動するだけです。

オート デプロイ

オート デプロイは、`global.properties` および `local.properties` にある `webapp.hotdeploy.autodeploy` プロパティを使用して制御します。このプロパティはディレクトリ を指定し、JRun がそのディレクトリ に新しい WAR ファイルを見つけると、それらを自動的に公開します。JRun では、サーバーが起動するとこの場所を確認し、サーバーの実行中には新しいファイルがないかどうかを監視します。

`global.properties` の既定の設定は、次のとおりです。

```
webapp.hotdeploy.autodeploy={jrun.rootdir}/servers/{jrun.server.name}/
        deploy/*.war
```

WAR ファイルを自動的に公開するときに、JRun は拡張子のない WAR ファイル名を使用して Web アプリケーション名および URL マッピングを作成し、その WAR ファイルを `/servers/server name/`*`Web アプリケーション名`*というディレクトリに展開します。たとえば、`newapp.war` を既定のサーバーの `deploy` ディレクトリ のルート レベルに追加すると、JRun は次のように作成します。

- Web アプリケーション名は newapp
- URL マッピングは `/newapp`
- アプリケーションのルート ディレクトリは*既定サーバーのルート* `/newapp`

JRun で作成されるディレクトリ構造は、WAR ファイルを公開ディレクトリのサブディレクトリに格納することによって制御できます。たとえば、WAR ファイルを `deploy/accounting/payroll.war` に保存すると、JRun では、accounting-payroll という名前で `/accounting/payroll` という URL マッピングを持つ Web アプリケーションが作成されます。

第 35 章

Enterprise JavaBeans の公開

目次

- 概要 410
- 公開記述子のコーディング 411
- JAR ファイルの作成 418
- プロパティ ファイルのコーディング 418
- Deploy ツールの実行 420
- その他のクラスの取り込み 421
- Bean のダイナミック ローディングの使用 422
- 実行時環境 423

概要

Bean の開発が完了し、ホーム インターフェイスおよびリモート インターフェイスを定義したら、Bean を公開できます。ただし、この**公開**という用語は、JRun の EJB では意味が若干異なるため、注意が必要です。サーブレットの場合は、コンパイル、テストを行ってから、最後に公開して配布します。EJB の場合は、コンパイル、テスト公開、テストを行ってから、最後に公開して配布します (アプリケーション アセンブラによって追加された書き換えが含まれることがあります)。

Bean を公開するには、次の手順を実行します。

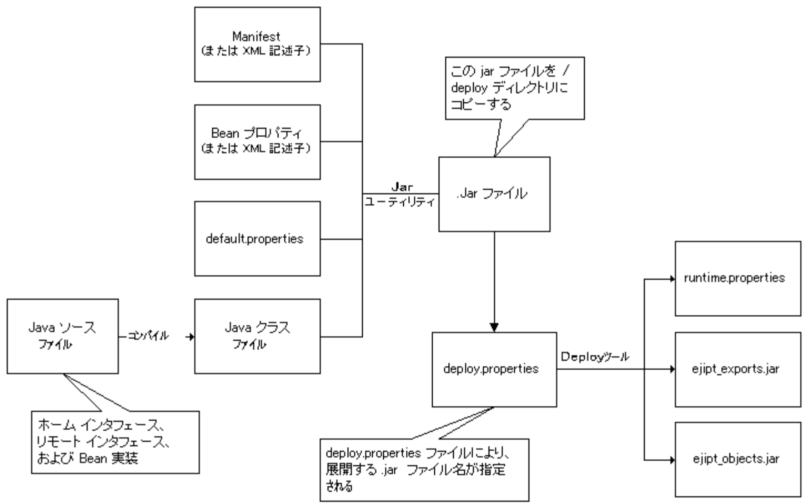
1 リモート インターフェイス、ホーム インターフェイス、および Bean 実装をコンパイルします。

2 公開記述子を作成します。

3 JAR ファイルを作成し、`/deploy` ディレクトリにコピーします。

4 Deploy ツールを実行します。

次の図は、Bean の公開プロセスのコンポーネントを示します。



Deploy ツールへの入力は、次のとおりです。

- 1 つまたは複数の EJB JAR ファイル。各ファイルには、1 つの公開記述子のほか、コンパイルされたホーム インターフェイス、リモート インターフェイス、および Bean 実装も含まれています。
- 一連のプロパティ ファイルからの仕様。これらのプロパティはサーバー レベルの設定を指定するほか、EJB の公開記述子に設定されている要素をオプションで書き換えることもできます。

Deploy ツールからの出力は、`/deploy` サブディレクトリに保存されます。この出力は、JRun の起動時に `/runtime` ディレクトリに自動的にコピーされます。

公開記述子のコーディング

まず Deploy ツールが正しくホームおよびリモート インターフェイス実装を生成し、さらに `runtime.properties` ファイルを生成するように、公開記述子にある要素をコーディングします。既定では、`ejb-jar.xml` という 名前の公開記述子を、XML エディタやテキスト エディタを使用して作成できます。さらに、一部の Java IDE でも公開記述子を自動的に生成します。

メモ
既存の公開記述子を表示するには、`/samples` サブディレクトリを参照してください。既存の公開記述子をコピーし、必要に応じて要素を変更する方法をお勧めします。

基本要素

公開記述子ファイルでは、JAR ファイルにあるすべての EJB を定義する必要があります。Deploy ツールでは、この情報を使用して、ホームおよびリモートの実装を生成します。

Bean タイプの識別

公開記述子にある EJB は、`enterprise-beans` 要素によってラップされます。各 EJB は、`entity` 要素 (エンティティ Bean の場合) または `session` 要素 (セッション Bean の場合) によってラップされます。

Bean の命名

Bean または Bean のインターフェイスの命名に関して条件や制限はありません。命名規則に関して仮定を定義することはできないため、Bean 開発者は、Bean のホーム インターフェイス、リモート インターフェイス、および実装の名前を、Bean の公開記述子で指定する必要があります。たとえば、次のような `home` 要素、`remote` 要素、および `ejb-class` 要素を指定します (推奨表記規則)。

```
<home>ejbeans.BalanceHome</home>
<remote>ejbeans.Balance</remote>
<ejb-class>ejbeans.BalanceBean</ejb-class>
```

エンティティ Bean の プライマリ キー クラス タイプを指定する必要があります。Deploy ツールはこの情報を使用してクラス情報を生成し、コンテナ管理パーシスタンスを使用する場合に、コンテナもこの情報を必要とします。

```
<prim-key-class>java.lang.Integer</prim-key-class>
```

ホーム名

`ejb-name` 要素を使用して、JNDI ネームスペースにある Bean に関連するホーム名を指定します。EJB エンジンはこの指定を使用して、JNDI コンテキストにあるホーム オブジェクトをバインドします。

```
<ejb-name>sample2a.BalanceHome</ejb-name>
```

ステート の管理

session-type 要素は、セッション Bean のステートを管理する方法を指定します。有効な値は、Stateful および Stateless です。

```
<session-type>Stateful</session-type>
```

パーシスタンスおよびコンテナ管理フィ ールド

persistence-type 要素は、エンティティ Bean がパーシスタンスを管理する方法を指定します。有効な値は、bean および container です。さらに、CMP を使用するエンティティ Bean は、cmp-field 要素を使用してコンテナ管理フィールドを識別しなければなりません。次の例は、2 つのコンテナ管理フィールドを持つ CMP エンティティ Bean を示します。

```
<persistence-type>Container</persistence-type>
<cmp-field>
  <field-name>_id</field-name>
</cmp-field>
<cmp-field>
  <field-name>_value</field-name>
</cmp-field>
```

セキュリティ要素

security-role 要素および method-permission 要素を使用して ロールベースのセキュリティを実装すると、1 つの Bean のすべてのメソッドまたは特定のメソッドのいずれか一方の起動を許可されているロールを指定できます。これらの要素は、assembly-descriptor 要素によってラップされている必要があります。

all という特別な値は、認証にかかわらずすべてのユーザを意味します。次の例では、すべてのユーザが create メソッドおよび getValue メソッドを使用できますが、save メソッドについては貯蓄者 (saver) ロールのユーザ、また spend メソッドについては支出者 (spender) ロールのユーザだけが使用できます。

```
<assembly-descriptor>
  <security-role>
    <role-name>spender</role-name>
  </security-role>
  <security-role>
    <role-name>saver</role-name>
  </security-role>
  <security-role>
    <role-name>all</role-name>
  </security-role>
  <method-permission>
    <role-name>spender</role-name>
    <method>
      <ejb-name>sample2a.BalanceHome</ejb-name>
      <method-name>spend</method-name>
    </method>
```

```
  </method-permission>
  <method-permission>
    <role-name>saver</role-name>
    <method>
      <ejb-name>sample2a.BalanceHome</ejb-name>
      <method-name>save</method-name>
    </method>
  </method-permission>
  <method-permission>
    <role-name>all</role-name>
    <method>
      <ejb-name>sample2a.BalanceHome</ejb-name>
      <method-name>create</method-name>
    </method>
  </method-permission>
  <method-permission>
    <role-name>all</role-name>
    <method>
      <ejb-name>sample2a.BalanceHome</ejb-name>
      <method-name>getValue</method-name>
    </method>
  </method-permission>
</assembly-descriptor>
```

env-entry 要素

以前のリリースの JRun EJB エンジンでは、開発者は公開記述子ではなく Bean プロパティ ファイルを指定していました。標準ではありませんが、このテクニックを使用すると、柔軟性が大幅に広がります。JRun 3.1 では、公開記述子が完全にサポートされているほか、env-entry 要素を使用して JRun 固有の Bean プロパティを渡す機能もあります。

env-entry を指定するための基本要素は、次のとおりです。

```
<env-entry>
  <env-entry-name>propertyname</env-entry-name>
  <env-entry-value>propertyvalue</env-entry-value>
</env_entry>
```

たとえば、ejb.sessionTimeout を env-entry として使用することによって、1 つの Bean で、セッション オブジェクトがタイムアウトになるまでの秒数を指定できます。このプロパティを指定しない場合、タイムアウトの既定値 900 (15 分) が適用されます。次の例では、タイムアウトの間隔を 600 に設定しています。

```
<env-entry>
  <env-entry-name>ejb.sessionTimeout</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>600</env-entry-value>
</env-entry>
```

次の例では、ejipt.isTimeoutFromCreate env-entry を使用して、セッション オブジェクトのタイムアウトが作成または最終アクセスの時点のどちらから開始するかを指定します。true または false のいずれかを指定します。このプロパティを指定しない場合、最終アクセスが適用されます。

```
<env-entry>
  <env-entry-name>ejipt.isTimeoutFromCreate</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>false</env-entry-value>
</env-entry>
```

EJB エンジン固有のプロパティの先頭には、通常 ejipt が付いています。env-entry を使用して渡すことのできるプロパティの詳細については、JRun JavaDocs の EjbProperties および EjiptProperties インターフェイスのマニュアルを参照してください。これらのインターフェイスは、各プロパティの内部定数を定義し、プロパティごとの定義を含んでいます。

公開記述子の例

次の例は、EJB サンプル 2b の XML 公開記述子を示します。

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
        EnterpriseJavaBeans 1.1//EN"
        "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
  <description>Custom Authentication</description>
  <display-name>Custom Authentication</display-name>
  <enterprise-beans>
    <entity>
      <display-name>UserBean</display-name>
      <ejb-name>sample2b.UserHome</ejb-name>
      <home>ejbeans.UserHome</home>
      <remote>ejbeans.User</remote>
      <ejb-class>ejbeans.UserBean</ejb-class>
      <prim-key-class>java.lang.String</prim-key-class>
      <persistence-type>Bean</persistence-type>
      <reentrant>False</reentrant>
      <env-entry>
        <env-entry-name>ejipt.maxContexts</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>unspecified</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>ejipt.storeName</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>default</env-entry-value>
      </env-entry>
    </entity>
    <session>
      <display-name>LoginSessionBean</display-name>
      <ejb-name>sample2b.LoginSessionHome</ejb-name>
```

```
    <home>ejbeans.LoginSessionHome</home>
    <remote>ejbeans.LoginSession</remote>
    <ejb-class>ejbeans.LoginSessionBean</ejb-class>
    <session-type>Stateful</session-type>
    <transaction-type>Bean</transaction-type>
    <env-entry>
      <env-entry-name>ejipt.maxContexts</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>unspecified</env-entry-value>
    </env-entry>
    <env-entry>
      <env-entry-name>ejb.sessionTimeout</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>600</env-entry-value>
    </env-entry>
    <env-entry>
      <env-entry-name>create.ejb.runAsMode</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>SYSTEM_IDENTITY</env-entry-value>
    </env-entry>
  </session>
  <entity>
    <display-name>RoleBean</display-name>
    <ejb-name>sample2b.RoleHome</ejb-name>
    <home>ejbeans.RoleHome</home>
    <remote>ejbeans.Role</remote>
    <ejb-class>ejbeans.RoleBean</ejb-class>
    <prim-key-class>java.lang.String</prim-key-class>
    <persistence-type>Bean</persistence-type>
    <reentrant>False</reentrant>
    <env-entry>
      <env-entry-name>ejipt.maxContexts</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>unspecified</env-entry-value>
    </env-entry>
    <env-entry>
      <env-entry-name>ejipt.storeName</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>default</env-entry-value>
    </env-entry>
  </entity>
  <entity>
    <display-name>BalanceBean</display-name>
    <ejb-name>sample2b.BalanceHome</ejb-name>
    <home>ejbeans.BalanceHome</home>
    <remote>ejbeans.Balance</remote>
    <ejb-class>ejbeans.BalanceBean</ejb-class>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <persistence-type>Bean</persistence-type>
    <reentrant>False</reentrant>
```

```
      <env-entry>
        <env-entry-name>ejipt.maxContexts</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>unspecified</env-entry-value>
      </env-entry>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <security-role>
      <role-name>system</role-name>
    </security-role>
    <security-role>
      <role-name>all</role-name>
    </security-role>
    <security-role>
      <role-name>spender</role-name>
    </security-role>
    <security-role>
      <role-name>saver</role-name>
    </security-role>
      <method-permission>
        <role-name>system</role-name>
        <method>
          <ejb-name>sample2b.UserHome</ejb-name>
          <method-name>checkPassword</method-name>
        </method>
      </method-permission>
      <method-permission>
        <role-name>all</role-name>
        <method>
          <ejb-name>sample2b.UserHome</ejb-name>
          <method-name>*</method-name>
        </method>
      </method-permission>
      <method-permission>
        <role-name>all</role-name>
        <method>
          <ejb-name>sample2b.LoginSessionHome</ejb-name>
          <method-name>*</method-name>
        </method>
      </method-permission>
      <method-permission>
        <role-name>all</role-name>
        <method>
          <ejb-name>sample2b.RoleHome</ejb-name>
          <method-name>*</method-name>
        </method>
      </method-permission>
      <method-permission>
        <role-name>spender</role-name>
        <method>
          <ejb-name>sample2b.BalanceHome</ejb-name>
          <method-name>spend</method-name>
        </method>
```

```
      </method-permission>
      <method-permission>
        <role-name>saver</role-name>
        <method>
          <ejb-name>sample2b.BalanceHome</ejb-name>
          <method-name>save</method-name>
        </method>
      </method-permission>
      <method-permission>
        <role-name>all</role-name>
        <method>
          <ejb-name>sample2b.BalanceHome</ejb-name>
          <method-name>create</method-name>
        </method>
       </method-permission>
      <method-permission>
        <role-name>all</role-name>
        <method>
          <ejb-name>sample2b.BalanceHome</ejb-name>
          <method-name>getValue</method-name>
        </method>
      </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

JAR ファイルの作成

JAR ファイルを作成する前に、ファイルをコンパイルする必要があります。Java IDE を使用している場合は、ホーム インターフェイス、リモート インターフェイス、および Bean 実装が含まれているプロジェクトを作成し、これらすべてを一度にコンパイルできます。IDE の多くでは、独自の EJB を公開することもできます。

コマンド ラインでコンパイルを実行している場合に、複数のファイルを最も簡単にコンパイルするには、コンパイルするファイルの名前を含んでいる `sources` というファイルを作成します。各ファイル名は、新しい行に記載する必要があります。詳細については、`javac` のマニュアルを参照してください。

`sources` ファイルを作成したら、次のコマンドを入力し、使用する環境に合わせて `project` および `projectpath` の値を設定します。

```
> cd projectpath
> javac -classpath projectpath; /JRUN_HOME/lib/ejipt.jar @sources
```

ここで次のコマンドを入力して JAR ファイルを作成します (現在の位置が `ejbeans` という名前のプロジェクトの 1 レベル上であり、`ejb-jar.xml` という名前の公開記述子があることを想定します)。

```
> cd projectpath
> jar cmf project_ejb.jar ejbeans/*.class META-INF/ejb-jar.xml
```

JAR ファイルが作成されたら、/deploy ディレクトリに保存します。

プロパティ ファイルのコーディング

プロパティ ファイルは、サーバー レベルのプロパティを指定するために使用します。Deploy ツールはプロパティを多数のソースから受け入れるため、それらの使用順序を理解することが重要です。Deploy ツールは、次の順序でプロパティにアクセスします。

- *JRun のインストール ディレクトリ* `/lib/ejipt.properties` EJB エンジンのすべてのインスタンスで使用するためのシステム全体のプロパティ。このファイルは通常、スタンドアロンの EJB エンジンを実行する場合に使用します。
- システム環境プロパティ `-D` コマンドライン スイッチで指定される、サーバー固有のプロパティ。
- *JRun のインストール ディレクトリ* `/lib/global.properties` すべての JRun サーバーで使用するためのシステム全体のプロパティ。このファイルは通常、統合 J2EE サーバーを実行する場合に使用します。EJB プロパティを `global.properties` 内で指定する場合は、接頭辞 `ejb` を使用します。
- *JRun のインストール ディレクトリ* `/local.properties` サーバー固有のプロパティ。このファイルは通常、統合 J2EE サーバーを実行する場合に使用します。EJB プロパティを `local.properties` 内で指定する場合は、接頭辞 ejb を使用します。
- *jrunserverdirectory*`/deploy/deploy.properties` サーバー固有のプロパティ。サーバー固有のプロパティはこのファイルまたは `local.properties` に格納できます。

ほかのすべてを書き換える deploy.properties を指定すると、リストの後方にあるファイルに指定されたプロパティは、リストの前方にあるファイルのプロパティを書き換えます。

一般的なプロパティ ファイルの指定の内容は、次のとおりです。

- [ejb.]ejipt.classServer.host ホスト名を識別します。ローカルで実行する場合は、この値を localhost のままにすることができます。ただし、リモート クライアントが接続する場合は、この値をローカルホストではなくサーバーのホスト名に設定する必要があります。この値を指定しないと、プロパティの既定値として現在のホストの名前が指定されます。これは通常、deploy.properties (スタンドアロン EJB の場合) または local.properties (統合 J2EE サーバーの場合は、接頭辞 ejb を必ず含む) に指定します。
- [ejb.]ejipt.classServer.port EJB エンジンがクラスにサービスを提供するために使用するポートを指定します。これは通常、ejipt.properties (スタンドアロン EJB の場合) または global.properties (統合 J2EE サーバーの場合は、接頭辞 ejb を必ず含む) に指定します。クライアントはこのプロパティを、InitialContext オブジェクトに渡される PROVIDER_URL に指定します。

```
properties.setProperty(Context.PROVIDER_URL,
    "ejipt://" + server + ":2323");
```

- [ejb.]ejipt.home.port EJB エンジンがホーム オブジェクトにサービスを提供するために使用するポートを指定します。これは通常、ejipt.properties (スタンドアロン EJB の場合) または global.properties (統合 J2EE サーバーの場合は、接頭辞 ejb を必ず含む) に指定します。
- [ejb.]ejipt.ejbJars 公開する JAR ファイルのリストを指定します。プロパティの値は、公開する JAR ファイルのカンマ区切りリストです。リストに指定する JAR ファイルは、/deploy ディレクトリに存在する必要があります。JAR ファイルを指定しない場合、プロパティの既定値として deploy ディレクトリのすべての JAR ファイルが指定されます。各 JAR ファイルは、サーバー内に独自のコンテナを持ちます。これは通常、deploy.properties (スタンドアロン EJB の場合) または local.properties (統合 J2EE サーバーの場合は、接頭辞 ejb を必ず含む) に指定します。
- ejipt.logStackTrace=true 詳細なスタック トレースがログに記録されることを指定します。これは通常、deploy.properties (スタンドアロン EJB の場合) または local.properties (統合 J2EE サーバーの場合は、接頭辞 ejb を必ず含む) に指定します。

プロパティの詳細については、JRun JavaDocs の EjbProperties および EjiptProperties インターフェイスのマニュアルを参照してください。これらのインターフェイスは、各プロパティの内部定数を定義し、プロパティごとの定義を含んでいます。

Deploy ツールの実行

Deploy ツールは、JRun で公開する Bean を作成するために使用します。Deploy ツールは次の作業を実行します。

- 公開記述子のリストにある Bean のホーム実装およびリモート オブジェクト実装を生成します。
- 生成されたオブジェクトにスタブ クラスを作成します。
- deploy.properties ファイルで ejipt.isCompatible=true と指定している場合に限って、JDK 1.1 ベースのクライアントで使用するために必要なスケルトンを作成します。
- JRun が実行時環境を確立するために使用する runtime.properties ファイルを作成します (これはシステム生成ファイルです。手作業で変更しないでください)。

Deploy ツールは、コマンド ラインまたは JMC から実行できます。

コマンド ライン Deploy ツールは、/deploy ディレクトリでのみ動作します。JAR ファイルなどすべての入力が /deploy ディレクトリで使用可能であり、生成された出力がすべて /deploy ディレクトリに配置されるようにする必要があります。JMC を使用して EJB を公開する場合、JAR ファイルはどの位置からでも指定でき、JMC はそれらを /deploy ディレクトリに自動的にコピーします。

Deploy ツールは、deploy.properties ファイルの ejipt.ejbJars プロパティにリストされている Bean JAR ファイルを処理します。このプロパティを指定しないと、Deploy ツールは、/deploy ディレクトリにある ejipt_objects.jar、ejipt_exports.jar、および extra_exports.jar 以外のすべての JAR ファイルを処理します。Deploy ツールの出力には、ejipt_objects.jar、ejipt_exports.jar、および runtime.properties があります。

既定では、Deploy ツールは標準の JDK コンパイラを使用します。ただし、deploy.properties ファイルの ejipt.javac プロパティを書き換えて、別のコンパイラを使用できます。

Deploy ツールを使用して EJB を公開するには、次のコマンドを入力します。

```
> cd jruninstalldirectory
> java -Djava.security.policy=lib/jrun.policy
 -classpath lib/ejipt_tools.jar allaire.ejipt.tools.Deploy
```

JMC を使用して EJB を公開するには、次の手順を実行します。

1 JMC にログオンします。

2 EJB の公開先サーバーを展開します。

3 Enterprise JavaBeans を展開せずにクリックします。

4 [公開] をクリックします。

[Enterprise JavaBeans 公開] パネルに deploy.properties ファイルおよび /deploy ディレクトリの EJB JAR ファイルのリストが表示されます。各 EJB JAR ファイルには、リモート インターフェイス、ホーム インターフェイス、および公開される Bean 実装が含まれています。

5 [ブラウズ] をクリックして、公開される 1 つまたは複数の EJB が含まれている JAR ファイルを選択します。このファイルが /deploy ディレクトリに存在しない場合、JMC 公開プロセスはこれを /deploy ディレクトリにコピーします。

6 (オプション) [公開プロパティ] テキスト領域に表示される deploy.properties ファイルの行を追加または変更します。詳細については、418 ページの「プロパティ ファイルのコーディング」を参照してください。

7 [公開] をクリックします。

JMC は deploy.properties ファイルを更新し、必要に応じて EJB JAR ファイルを /deploy ディレクトリにコピーして、Deploy ツールを呼び出します。

詳細については、『JRun セットアップ ガイド』を参照してください。

再公開

-redeploy オプションを使用すると、新しい Bean または更新された Bean に対してのみオブジェクトの実装が生成されます。Bean を再公開するには、次のコマンドを入力します。

```
> cd JRun のインストール ディレクトリ
> java -Djava.security.policy=lib/jrun.policy -classpath
  lib/ejipt_tools.jar allaire.ejipt.tools.Deploy -redeploy
```

メモ
JMC を使用して再公開することもできます。

その他のクラスの取り込み

JRun を使用して公開すると、公開された EJB オブジェクトに対して生成されるすべてのスタブは、ejipt_exports.jar に挿入されます。JAR のエクスポートに取り込む必要のあるクラスがほかにもある場合は、extra_exports.jar を作成し、これを /deploy ディレクトリにコピーします。EJB エンジンは、/deploy ディレクトリで見つかった extra_exports.jar の内容を ejipt_exports.jar に自動的に組み込みます。

Bean のダイナミック ローディングの使用

Bean のダイナミック ローディングを使用すると、再公開することなく、変更した Bean クラス実装を再コンパイルして実行できます。Bean のダイナミック ローディングでは、JAR ファイル全体を再コンパイルし再公開する必要がないため、開発およびテストに要する時間を節約できます。この機能は、複数の開発者が EJB サーバーで同時に作業をする場合にも役に立ちます。

Bean のダイナミック ローディングは、ホーム インターフェイスおよびリモート インターフェイス用に設計されていません。したがって、Bean のクラス実装の修正によってホーム インターフェイスおよびリモート インターフェイスを変更する必要がある場合は、Bean のダイナミック ローディングを使用できません。

メモ

Bean のダイナミック ローディングは、EJB エンジンをスタンドアロン モードで実行している場合、つまりコマンド ラインで `java allaire.ejipt.Ejipt` コマンドによって起動した場合にのみ可能です。EJB エンジンを JRun で実行している場合は使用できません。スタンドアロン モードの詳細については、395 ページの「スタンドアロン モードでの EJB エンジンの動作」を参照してください。

Bean のダイナミック ローディングを使用するには

1 必要に応じて Bean 実装を修正します。

2 Bean 実装をコンパイルし、`runtime/classes` ディレクトリに保存します。

3 EJB エンジンのコマンド ラインで `load` コマンドを使用して、`runtime/classes` ディレクトリから Bean を再ロードします。

   このコマンドは、現在のインスタンスを無効にして、新しいインスタンスを有効にします。

4 テストが完了したら、変更内容を Bean の JAR ファイルに組み込みます。

実行時環境

EJB を新しい環境に公開する場合、JRun ではサーバーとクライアント環境でサポートファイルが使用できる必要があります。

サーバー環境

EJB エンジンでは、サーバー環境で次のファイルが用意されている必要があります。

ファイル	説明
Java 実行時	サーバーにバージョン 1.2 以降の JDK がインストールされている必要があります。
拡張機能	ejb.jar、jdbc.jar、jms.jar、jndi.jar、および jta.jar ファイルは、JRE の lib/ext ディレクトリにコピーする必要があります (またはシステムのクラスパス上にある必要があります)。この必要条件は、スタンドアロンの EJB エンジンだけを対象にしています。

クライアント環境

クライアントには、JRE がインストールされている必要があります。EJB エンジンは、JDK 1.1 または JDK 1.2 のどちらのクライアントでも実行できます。ただし、リモートアクティブ化と自動エクスポートを使用できるのは、JDK 1.2 のクライアントのみです。

EJB エンジンでは、クライアント環境で次のファイルがインストールされている必要があります。

ファイル	説明
Java Runtime	各クライアントに JRE がインストールされている必要があります。JRE はバージョン 1.1.6 以降で、deploy.properties の ejipt.isCompatible を true に設定する必要があります。特別なプロパティは必要ありません。
拡張機能	ejb.jar、jts.jar、jndi.jar、および jta.jar ファイルは、JRE の lib/ext ディレクトリにコピーする必要があります (またはシステムのクラスパス上にある必要があります)。
ejipt_client.jar	このファイルを JRUN_HOME/lib ディレクトリ内に置きます。
ejipt_jms_client.jar	このファイルを JRUN_HOME/lib ディレクトリ内に置きます。このファイルが必要なのは、クライアントで JMS を使用している場合だけです。

JDK 1.1 のクライアントには、ejipt_exports.jar. へのアクセス権が必要です。JDK 1.1 の制限により、JDK 1.1 のそれぞれのクライアント マシンに ejipt_exports.jar を明示的にインストールする必要があります。

第 36 章

J2EE アプリケーションの公開

この章では、J2EE アプリケーションの公開方法について説明します。

目次

- J2EE アプリケーションの公開について 426
- EAR ファイル 426
- 公開用 J2EE アプリケーションのパッケージ化 427
- J2EE アプリケーションの公開 429

J2EE アプリケーションの公開について

公開担当者は、JMC または JRun EarDeploy ユーティリティのいずれかと、アプリケーション アセンブル担当者が作成した EAR ファイルを使用して J2EE アプリケーションを特定の運用環境にインストールします。EAR ファイルのインストールのほかに、必要に応じてアプリケーションを運用環境用に設定します。たとえば、サーブレットおよび EJB 用に認証とセキュリティを実装しなければならない場合があります。

詳細については、427 ページの「公開用 J2EE アプリケーションのパッケージ化」を参照してください。

EAR ファイル

J2EE アプリケーションは通常、圧縮された単一の EAR ファイルとして公開します。EAR ファイルには、すべてのディレクトリ構造とアプリケーションを定義するすべてのファイルが含まれています。EAR ファイルは JAR ファイルと同じツールを使用して作成します。

JMC または コマンドライン EarDeploy ユーティリティを使用して EAR ファイルを公開します。これらのツールはどちらも EAR ファイルおよびサーバー固有のパラメータのセットを受け入れ、必要に応じてディレクトリ構造を形成したり、設定値やプロパティ ファイルを更新します。

J2EE アプリケーション公開時に、JRun では Ear ファイルに格納されている WAR ファイルが変換され、指定 JRun サーバーに新しいアプリケーションが定義されます。EAR ファイルに格納されているすべての EJB JAR ファイルも公開されます。

EAR ファイルには `META-INF/application.xml` 公開記述子が格納されている必要があります。この公開記述子から、JRun アプリケーション公開ユーティリティに情報が提供されます。

公開用 J2EE アプリケーションのパッケージ化

アプリケーション アセンブル担当者は、開発出力を公開可能な J2EE アプリケーションに変換します。アプリケーション アセンブル プロセスへの入力には、次の内容が含まれます。

- WAR ファイルおよびほかの関連している Web アプリケーション ファイル。第 34 章を参照してください。
- JAR ファイルおよびほかの関連している EJB 関連ファイル 第 35 章 を参照してください。
- `application.xml` ファイル

メモ

J2EE アプリケーションは、EAR ファイル外部との相関関係を持たないようにしてください。Web アプリケーションが仮想マッピングに依存してライブラリ (たとえば、*jruninstalldirectory*/`lib`) にアクセスする場合は、WAR ファイルの作成前に Web アプリケーションの `WEB-INF/lib` ディレクトリに必要なファイルをコピーする必要があります。仮想マッピングの詳細については、66 ページの「Web アプリケーション間でのクラスの共有」を参照してください。

アプリケーション アセンブル プロセスからの出力には、次の内容が含まれます。

- EAR ファイル
- インストール時の注意事項
- 設定ガイドライン

メモ

サポートされているハードウェア プラットフォーム、オペレーティング システム、Web サーバー、および JVM のすべてを組み合わせて使用して、公開可能 EAR ファイルを十分にテストしてください。

---

application.xml ファイルの作成

『J2EE version Specification version 1.2』では、application.xml という名前の XML 公開記述子によってアプリケーションを公開するように定められています。このファイルには、J2EE　アプリケーションのコンポーネントが定義されています。特に、application.xml ファイルでは次の内容が定義されています。

- アプリケーションの Web アプリケーション用の WAR ファイル。WAR ファイルを次のように定義します。

```
<module>
  <web>
    <context-root>sample9a</context-root>
    <web-uri>sample9a.war</web-uri>
  </web>
</module>
```

- アプリ ケーションの EJB 用の JAR ファイル。JAR ファイルを次のよう に定義します。

```
<module>
  <ejb>sample9a_ejb.jar</ejb>
</module>
```

『JRun サンプル ガイド』のサンプル 9a で使用している次のサンプル EAR ファイルでは、Web アプリケーションおよび EJB を定義しています。

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
        Application 1.2//EN" "http://java.sun.com/j2ee/dtds/
        application_1_2.dtd">

<application>
  <display-name>Sample 9a</display-name>
  <description>This sample demonstrates accessing beans from a
        servlet.</description>
  <module>
    <ejb>sample9a_ejb.jar</ejb>
  </module>
  <module>
    <web>
      <context-root>sample9a</context-root>
      <web-uri>sample9a.war</web-uri>
    </web>
  </module>
</application>
```

application.xml ファイルで使用される属性の詳細については、『Java 2 Platform Enterprise Edition Specification, V1.2』を参照してください。

EAR ファイルの作成

Java jar ユーティリティを使用して、EAR ファイルを作成します。

jar ユーティリティの詳細については、403 ページの「WAR ファイルの作成」を参照してください。

次の例では、現在のディレクトリが *jruninstalldirectory*/samples/sample9a である場合の、サンプル 9a の EAR ファイルを作成します。

```
jar cf sample9a.ear -C j2ee-app META-INF
jar uf sample9a.ear -C "%DEPLOY_DIR%" sample9a_ejb.jar
jar uf sample9a.ear sample9a.war
```

J2EE アプリケーションの公開

公開担当者は JRun 公開ツールを使用して J2EE アプリケーションをインストールします。これらのツールは、JMC またはコマンド ライン インターフェイスから実行できます。

J2EE アプリケーション公開プロセスの内容は次のとおりです。

- WAR ファイルのディレクトリ構造への展開
- EJB JAR ファイルの公開
- JRun サーバーへの Web アプリケーションの追加
- Web アプリケーションに対するアプリケーション マッピングの追加

公開プロセスを開始する前に、次の内容を確認してください。

- EAR ファイルの位置とディレクトリの絶対パス
- 公開した Web アプリケーションを格納する JRun サーバーの名前

JMC の使用

JMC を使用して J2EE アプリケーションを公開できます。

JMC を使用して J2EE アプリケーションを公開するには、次の手順を実行します。

1. JMC にログオンします。
2. J2EE アプリケーションの公開先のサーバーを、展開しないでクリックします。
3. [EAR 公開] をクリックします。
4. 情報を指定します。
5. [公開] をクリックします。

詳細については、『JRun セットアップ ガイド』を参照してください。

コマンド ライン インターフェイスの使用

JRun には EarDeploy ツールが用意されており、このツールでは J2EE アプリケーション公開ツール用のコマンド ライン インターフェイスが提供されます。EarDeploy ツールによって、コマンド ライン パラメータのセットを使用して EAR ファイルを公開できます。

EarDeploy には、WarDeploy ツールおよび EJB Deploy ツールが組み込まれています。この機能を活用するには大量のライブラリ セットが必要なので、`-classpath` 引数は非常に長くなります。通常、JMC によって簡単なインターフェイスが提供されます。

構文

コマンド ライン インターフェイスでは、JRun インストール ディレクトリで作業していると想定した場合、次の構文を使用して J2EE アプリケーションの公開、削除、および再公開を実行できます。

```
java -classpath=classpathfiles allaire.jrun.tools.EarDeploy
  -d earfile -s servername -j jruninstalldirectory
```

次の Windows バッチ ファイルの例は、必要なクラスパス ファイルを使用して EarDeploy を実行する方法について示します。

```
@echo off
set JRUN_HOME=c:¥progra~1¥allaire¥jrun

set CP=%CP%;%JRUN_HOME%¥lib¥ext¥ejb.jar
set CP=%CP%;%JRUN_HOME%¥lib¥ext¥jaxp.jar
set CP=%CP%;%JRUN_HOME%¥lib¥ext¥parser.jar
set CP=%CP%;%JRUN_HOME%¥lib¥ext¥servlet.jar
set CP=%CP%;%JRUN_HOME%¥lib¥jrun.jar
set CP=%CP%;%JRUN_HOME%¥lib¥install.jar
set CP=%CP%;%JRUN_HOME%¥lib¥ejipt.jar
set CP=%CP%;%JRUN_HOME%¥lib¥ejipt_tools.jar

java -classpath %CP% allaire.jrun.tools.EarDeploy
        -d %JRUN_HOME%¥samples¥sample9a¥sample9a.ear -s default
        -j %JRUN_HOME%

rem The following line supports arguments for -d, -s, and -j
rem java -classpath %CP% allaire.jrun.tools.EarDeploy %1
```

セキュリティのためのユーザとロールの定義

公開担当者は、公開される J2EE アプリケーションについて、セキュリティ ユーザとロールを定義しなければならない場合があります。ユーザ定義の詳細については、第 39 章を参照してください。

第 6 部

JRun での作業

ここでは、JRun の基本機能を使用する方法について説明します。

Web サーバー接続の監視 433
ログ 441
Web アプリケーションの認証 463
サーブレット メソッド パフォーマンスの監視 485
デバッグとエラー メッセージング 503
JRun の拡張機能 519
JRun と ColdFusion の併用 525

第 37 章

Web サーバー接続の監視

JRun が提供する監視メカニズムを使用して、Web サーバーと JRun の接続に関する統計情報を取得できます。この統計には、処理された要求の数、要求の処理に使用可能なスレッドの数、ヒープ メモリの使用状況が含まれます。

この章では、接続管理メカニズムと、このメカニズムの設定、およびこのメカニズムにより収集される情報の制御について説明します。

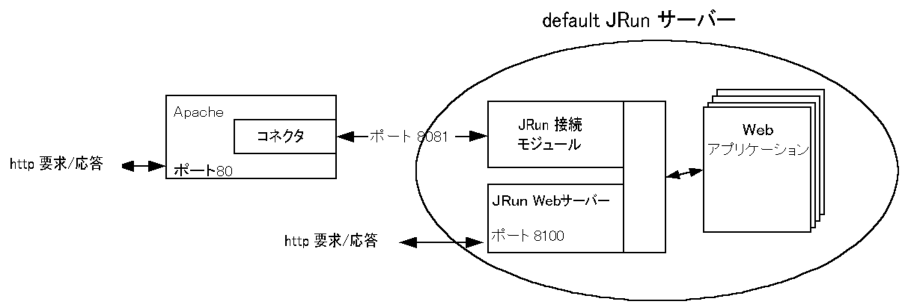
目次

- Web サーバー接続の監視 434
- プロパティの監視 439

Web サーバー接続の監視

JRun では、JRun サーバーと Web サーバーの接続に関するステータス情報が、JRun ログ ファイルに書き込まれます。このステータス情報は JRun サーバーとサードパーティ製 Web サーバーまたは JRun Web サーバー (JWS) の接続から取得できます。

次の図は、サードパーティ製 Web サーバーと JWS の両方に接続されている default JRun サーバーを表しています。



接続ステータス情報が収集されるように JRun を設定するには、JRun サーバーの `local.properties` ファイルにあるプロパティ `logging.loglevel` に指定されたロギング レベルの一覧にキーワード `metrics` を入れます。JRun のロギング メカニズムの詳細については、第 38 章を参照してください。

たとえば、接続ステータス情報を有効にするには、次のプロパティを使用します。

```
logging.loglevel=info,warning,error,metrics
```

次のいずれか、またはすべての接続情報をログ ファイルに書き出すことができます。

- `listenTh`　新しい接続を受信するスレッド
- `idleTh`　新しい要求を待っているスレッド
- `delayTh`　実行を待っているスレッド
- `busyTh`　現在実行しているスレッド
- `totalTh`　ワーカー スレッドの総数
- `delayRq`　同時発生が多発していることにより遅延した要求
- `droppedRq`　却下された要求
- `handledRq`　処理された要求
- `handledMS`　遅延時間 (`delayMs`) を除く、要求のサービスに費やされた時間をミリ秒単位で表したもの
- `delayMs`　遅延状態で経過した時間をミリ秒単位で表したもの
- `bytesIn`　要求から読み込まれたバイト数
- `bytesOut`　応答に書き込まれたバイト数
- `freeMemory`　ヒープ内の空きメモリ量をキロバイト単位で表したもの

- `totalMemory` 使用されているかどうかに関係なく、ヒープ全体のメモリ量をキロバイト単位で表したもの
- `session` 現在アクティブなセッション数
- `sessionsInMem` メモリにあるセッション数

次に示すのは、接続ステータス情報が記録されたログ メッセージの例です。

```
03/20 02:57:53 metrics (jcp+web) Heap=3151KB Listen=1 Idle=0 Queued=0
  Busy=0 Total=0 Requests (count/total ms)=0/0 Delayed=0 TotalDelay=0
  BytesIn=0 BytesOut=0 Sessions (active/in memory)=0/0
```

監視メカニズムの設定

監視メカニズムの設定には、プロパティ ファイルを使用します。JRun 管理コンソール (JMC) では制御できません。

監視メカニズムの既定の設定は JRun のインストールに対応するすべての JRun サーバーに対するもので、`global.properties` ファイルに保存されています。個々の JRun サーバーの設定を変更する場合は、`local.properties` ファイルのそのサーバーの設定を変更します。

`global.properties` ファイルでの既定のプロパティ設定は次のとおりです。

```
monitor.class=allaire.jrun.metrics.MetricsLogger
monitor.interval=60
monitor.format={monitor.combined-format}
monitor.max.history=10
```

これらの設定により、監視メカニズム、監視間隔 (単位は秒)、監視情報の出力形式、監視履歴バッファのサイズを定義するクラスが指定されます。これらの設定では、監視メカニズムにより、60 秒ごとに統計が行われ、最新のものからさかのぼって 10 個の監視サンプルが記録されるように指定されています。

これらのプロパティの詳細については、439 ページの「プロパティの監視」を参照してください。

監視出力形式の設定

global.properties ファイルには、監視メッセージの形式を表すプロパティ monitor.format が含まれています。既定では、monitor.format は次のように設定されています。

```
monitor.format={monitor.combined-format}
```

monitor.combined-format 形式プロパティは、global.properties ファイルで次のように定義されます。

```
monitor.combined-format=(jcp+web)
  Heap={totalMemory}KB
  Listen={{jcp.listenTh}+{web.listenTh}}
  Idle={{jcp.idleTh}+{web.idleTh}}
  Queued={{jcp.delayTh}+{web.delayTh}}
  Busy={{jcp.busyTh}+{web.busyTh}}
  Total={{jcp.totalTh}+{jcp.totalTh}}
  Requests={{jcp.handledRq}+{web.handledRq}}
  Delayed={{jcp.delayRq}+{web.delayRq}}
  TotalDelay={{jcp.delayMs}+{web.delayMs}}
  BytesIn={{jcp.bytesIn}+{web.bytesIn}}
  BytesOut={{jcp.bytesOut}+{web.bytesOut}}
  Sessions (active/in memory)={sessions}/{sessionsInMem}
```

この形式により、メッセージが次のように生成されます。

```
03/20 02:57:53 metrics (jcp+web) Heap=3151KB Listen=1 Idle=0 Queued=0
  Busy=0 Total=0 Requests (count/total ms)=0/0 Delayed=0 TotalDelay=0
  BytesIn=0 BytesOut=0 Sessions (active/in memory)=0/0
```

この設定の構文は次のとおりです。

```
monitor.<label>=(web | jcp | web+jcp)
 String1={[web.| jcp.]statistic1}
 String2={[web.| jcp.]statistic2}
  ...
```

monitor.format プロパティに monitor.label を指定して、ログ ファイルに書き込まれる監視情報の形式を制御します。

monitor.<Label>=(web|jcp|web+jcp)

形式定義に対してラベルを指定するとともに、この形式指定が JWS (web)、サードパーティ製 Web サーバー (jcp)、またはこの両方 (web+jcp) のいずれかから得られた管理情報に対する形式を定義しているのかどうかを指定します。

String={[web.|jcp.]statistic}

ログ ファイルに書き込まれた監視情報出力に入っている文字列の形式を指定します。メッセージの *String=* 部分は、変更されることなくそのままログ ファイルに書き込まれます。JRun により、メッセージの [web.|jcp.]*statistic* 部分が統計値で置き換えられます。この統計情報は、JWS (web) への JRun 接続、またはサードパーティ製 Web サーバー (jcp) からの接続から得られたものです。

ログ ファイルに書き出される接続情報に、次のいずれか、またはすべての統計を入れることができます。これらの統計の多くには、先頭に jcp または web が付きます。これは、この統計がサードパーティ製 Web サーバーから得られたか (jcp)、または JWS から得られたか (web) を表しています。ただし、一部の統計は JRun サーバーと関連付けられた JVM に対するもので、接頭辞は付けられません。

- freeMemory　ヒープ内の空きメモリ量をキロバイト単位で表したもの
- totalMemory　使用されているかどうかに関係なく、ヒープ全体のメモリ量をキロバイト単位で表したもの
- session　現在アクティブなセッション数
- sessionsInMem　メモリにあるセッション数
- [web.|jcp.]busyTh　現在実行しているスレッド
- [web.|jcp.]delayTh　実行待ちスレッド
- [web.|jcp.]idleTh　新しい要求を待っているスレッド
- [web.|jcp.]listenTh　新しい接続を受信するスレッドの数
- [web.|jcp.]totalTh　ワーカー スレッドの総数
- [web.|jcp.]delayRq　同時発生が多発していることにより遅延した要求の数
- [web.|jcp.]droppedRq　却下された要求
- [web.|jcp.]handledRq　処理された要求
- [web.|jcp.]handledMS　遅延時間 (delayMs) を除く、要求のサービスに費やされた時間をミリ秒単位で表したもの
- [web.|jcp.]delayMs　遅延状態で経過した時間をミリ秒単位で表したもの
- [web.|jcp.]bytesIn　要求から読み込まれたバイト数
- [web.|jcp.]bytesOut　応答に書き込まれたバイト数

たとえば、監視メッセージ用に、次の形式を定義したとします。

```
monitor.combined-format=(jcp+web)
  Heap={totalMemory}KB
  Total={{jcp.totalTh}+{web.totalTh}}
```

この定義は、このメッセージ形式が、JWS とサードパーティ製 Web サーバーの両方から得られた情報に適用されることを表しています。この監視メッセージには、ヒープ メモリの量、および JWS とサードパーティ製 Web サーバーの両方で使用されているワーカー スレッドの数を合計したものが含まれます。

JWS とサードパーティ製 Web サーバーの両方で使用されているワーカー スレッド数の合計を表示するには、次の形式で指定します。

```
Total={{jcp.totalTh}+{web.totalTh}}
```

上記のとおり、表示される値は JWS から得られた値と、サードパーティ製 Web サーバーから得られた値の合計です。数式を中かっこ {} で囲むことにより、統計値に対する計算ができることが、この例からわかります。

既定の監視形式

JRun の global.properties ファイルには、次のような定義済み形式が用意されています。このような定義済み形式の 1 つを monitor.format プロパティに指定できます。また、ユーザが形式を作成することもできます。

```
# JWS とサードパーティ製 Web サーバーの両方から得られた監視情報で
# 使用されるメッセージ形式を定義します。
# これは既定の監視形式です。
monitor.combined-format=(jcp+web)
  Heap={totalMemory}KB
  Listen={{jcp.listenTh}+{web.listenTh}}
  Idle={{jcp.idleTh}+{web.idleTh}}
  Queued={{jcp.delayTh}+{web.delayTh}}
  Busy={{jcp.busyTh}+{web.busyTh}}
  Total={{jcp.totalTh}+{web.totalTh}}
  Requests (count/total ms)={{jcp.handledRq}+{web.handledRq}}/
        {{jcp.handledMs}+{web.handledMs}}
  Delayed={{jcp.delayRq}+{web.delayRq}}
  TotalDelay={{jcp.delayMs}+{web.delayMs}}
  BytesIn={{jcp.bytesIn}+{web.bytesIn}}
  BytesOut={{jcp.bytesOut}+{web.bytesOut}}
  Sessions (active/in memory)={sessions}/{sessionsInMem}

# JWS だけから得られる監視情報のメッセージ形式を定義します。
monitor.web-format=(web)
  Heap={totalMemory}KB
  Listen={web.listenTh}
  Idle={web.idleTh}
  Queued={web.delayTh}
  Busy={web.busyTh}
  Total={web.totalTh}
  Requests (count/total ms)={web.handledRq}/{web.handledMs}
  Delayed={web.delayRq}
  TotalDelay={web.delayMs}
  BytesIn={web.bytesIn}
  BytesOut={web.bytesOut}
  Sessions (active/in memory)={sessions}/{sessionsInMem}

# JWS だけから得られる情報のショート メッセージ形式を定義します。
monitor.web-short-format=(web)
  Busy={web.busyTh}
  Total={web.totalTh}
  Requests={web.handledRq}
  TotalDelay={web.delayMs}

# サードパーティ製 Web サーバーだけから得られた監視情報で使用される
# メッセージ形式を定義します。
monitor.jcp-format=(jcp)
  Heap={totalMemory}KB
  Listen={jcp.listenTh}
  Idle={jcp.idleTh}
  Queued={jcp.delayTh}
```

```
  Busy={jcp.busyTh}
  Total={jcp.totalTh}
  Requests (count/total ms)={jcp.handledRq}/{jcp.handledMs}
  Delayed={jcp.delayRq}
  TotalDelay={jcp.delayMs}
  BytesIn={jcp.bytesIn}
  BytesOut={jcp.bytesOut}
  Sessions (active/in memory)={sessions}/{sessionsInMem}

# サードパーティ製 Web サーバーだけから得られた監視情報で使用されるショート
# メッセージ形式を定義します。
monitor.jcp-short-format=(jcp)
  Busy={jcp.busyTh}
  Total={jcp.totalTh}
  Requests={jcp.handledRq}
  TotalDelay={jcp.delayMs}
```

プロパティの監視

local.properties ファイル内の次のプロパティは、接続ステータス情報のコレクションを制御します。

monitor.class

接続管理を定義する JRun クラスを指定します。既定では、このプロパティは allaire.jrun.metrics.MetricsLogger に設定されます。

monitor.interval

監視間隔を秒単位で指定します。既定の間隔は 60 秒です。

monitor.format

ログ ファイルに書き込まれた監視情報の形式を指定します。既定値は {monitor.combined-format} です。メッセージ形式の詳細については、436 ページの「監視出力形式の設定」を参照してください。

monitor.max.history

JRun では、指定された数の接続統計がサンプルとして保持されます。これらのサンプルにアクセスして、一定時間の統計を平均することができます。このプロパティには、平均を求めるために保持しておく必要のある統計サンプルの数を指定します。既定値は 10 です。

monitor.loggername

必要に応じて、接続ステータス情報を受信するために使用されるロガーの名前を指定します。既定の設定では、出力はすべて、JRun サーバーのログ ファイルに書き込まれます。

このプロパティを使用して、出力を受信するためのロガーを作成できます。たとえば、ロガーを使用して、専用のファイルに出力を書き込むことができます。ログ収集とロガーの詳細については、第 38 章を参照してください。

第 38 章

ログ

JRun では、ログ ユーティリティを使用して、アプリケーションの起動時および実行時に生成されたさまざまなタイプのメッセージを出力できます。

この章では、JRun のログ メカニズムについて説明します。

目次

- 概要 442
- 既定のログ設定 444
- 例 446
- ログ メッセージ定義の変更 449
- ログ メカニズムの定義 453
- ログ メッセージの形式 454
- ログ プロパティ 455

概要

アプリケーションの起動時および実行時には、サーバーの状態、エラーの状態、パフォーマンス統計などのログ メッセージが JRun によって生成されます。JRun に用意されているログ ユーティリティを使用して、これらのメッセージを表示できます。

さまざまなタイプのメッセージを処理したり、それらのメッセージをさまざまな出力先に記録できるように JRun を設定できます。また、カスタマイズされたフィルタリングや書き込みを実行する固有のログ コンポーネントも作成できます。

一般的に、JRun のログ メカニズムにはほとんどオーバヘッドがありません。したがって、アプリケーションのパフォーマンスに与える影響は最小限に抑えられています。

メモ

デバッグ情報の生成はアプリケーションのパフォーマンスに影響を与えます。このオプションはアプリケーションのデバッグ時にのみ使用してください。

この章では、既定のログの設定、その設定の変更手順、その変更方法を示す例など、ログ メカニズムについて説明します。

メッセージのタイプ

JRun によって次のタイプのメッセージが生成されます。

- 情報メッセージ。起動時に生成されるメッセージおよびメソッド タイミング メッセージがあります (485 ページの第 40 章「サーブレット メソッド パフォーマンスの監視」を参照)。
- 警告メッセージ
- エラー メッセージ
- デバッグ情報
- システムのメトリックと、Web サーバーと JRun 間の接続についてのメトリック (433 ページの第 37 章「Web サーバー接続の監視」を参照)。

ログ メカニズムを 1 つまたは複数の種類のメッセージを記録するように設定できます。既定では、情報メッセージ、警告メッセージ、およびエラー メッセージが記録されます。

メッセージの出力先

ログ メカニズムによって、メッセージを次の出力先に送信できます。

- 1 つまたは複数のファイル
- クライアントの画面
- 指定されたユーザへの電子メール メッセージ

標準のログ ライター

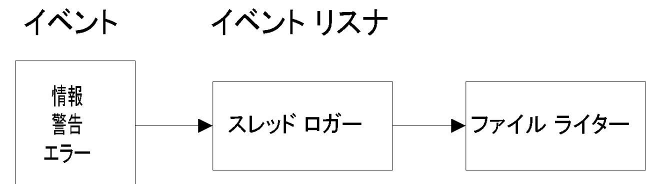
JRun には便利なログ ライターがいくつか用意されています。

- **スレッド ロガー**　このロガーをメインのキューとして使用します。ログ メッセージを作成するコンポーネント (サーブレット) から、メモリ内のキューにメッセージが送信され、制御はすぐにコンポーネントに戻されます。バックグラウンド スレッドによってキューが処理され、ログ メッセージはいくつかのログ ライターに送信されます。この設計によって、ログ作業がファイルに渡されたり、ほかのコンポーネントに電子メールが送信されることによって、サーブレット内のパフォーマンスが維持されます。既定ではスレッド ロガーが使用されます。
- **ディスパッチ ロガー**　イベント タイプに基づいた出力先に、ログ イベントを送り出します。たとえば、1 つのファイル ライターにエラー メッセージを送出し、デバッグ メッセージをほかのファイル ライターに送出するように、ディスパッチ ロガーを設定できます。
- **ファイル ライター**　ディスパッチ ロガーからログ イベントを受信し、ファイルに書き込みます。既定ではファイル ライターはスレッド ロガーと併用します。
- **電子メール ライター**　ディスパッチロガーからログ イベントを受信し、電子メールに送信します。電子メール ライターを使用すると、特定のタイプのメッセージ (エラー メッセージなど) を受信したときに特定のアドレスに電子メールを送信できます。
- **スクリーン ライター**　ディスパッチ ロガーからログ イベントを受信し、システムの標準出力装置 (通常はスクリーン) に送信します。このライターは、デバッグ タイプのメッセージに特に役立ちます。

既定のログ設定

JRun のインストール時に定義された既定の設定では、アプリケーションでログ イベントが発生すると、このイベントはメモリ内のキューに置かれます。その直後、制御がアプリケーションに戻されます。次に、バックグラウンド スレッドにより、キューからイベントが読み込まれ、適切な出力先に各イベントが転送されます。

次の図は、ログ メカニズムの既定の設定を示します。



この既定の設定では、次の作業が行われます。

1 スレッド ロガーがイベント リスナとして機能します。info、warning、または error のイベントが発生すると、そのイベントが記述されているメッセージがスレッド ロガーのキューに書き込まれます。

2 スレッド ロガーのバックグラウンド スレッドにより、キューからイベントが読み込まれ、ファイル ライターに送信されます。

3 ファイル ライターによって、`{jrun.rootdir}¥logs¥{jrun.server.name}-event.log` ファイルにイベントが書き込まれます。

4 ファイルのローテートを行うまでに、ログ ファイルには最大 100,000 バイトの情報を書き込むことができます。ファイルをローテートすると、ファイルの内容は別のログ ファイルに書き込まれます。最大 5 つのローテート ファイルを使用します。この値は変更できます。

ログの設定は `global.properties` ファイルで定義し、サーバーの場合は `local.properties` ファイルで変更します。これ以降は、ログ メカニズムを構成する `global.properties` ファイル内のステートメントを示します。

各プロパティの詳細については、455 ページの「ログ プロパティ」で説明しています。

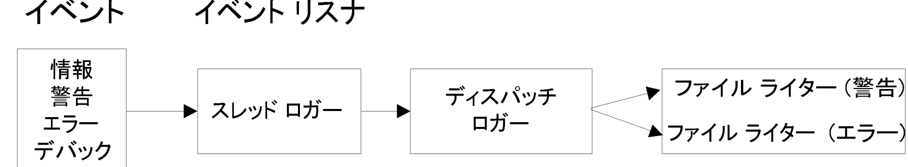
```
#########################################################################
## ログ サービス
#########################################################################

# サービス クラスと情報
logging.class=allaire.jrun.logging.LoggingService
logging.format={date MM/dd HH:mm:ss} {log.level} ({log.name})
     {log.message}
# カンマ区切りリストとしてログ レベルを指定します。
# 値は、debug、info、warning、error、metrics を組み合わせることができます。
logging.loglevel=info,warning,error
# ログ リスナを定義します。
logging.listeners=threadedlogger

# スレッド ロガー。クライアントは、バックグラウンド スレッドによって処理される
# キューにログ イベントを送信します。ログ イベントは、
# 複数のログ リスナに送信できます。
logging.threadedlogger.class=allaire.jrun.logging.ThreadedLogger
logging.threadedlogger.listeners=filelogwriter
# ログに記録するイベントを定義します。
logging.infoevent=allaire.jrun.logging.InfoLogEvent
logging.debugevent=allaire.jrun.logging.DebugLogEvent
logging.warningevent=allaire.jrun.logging.WarningLogEvent
logging.errorevent=allaire.jrun.logging.ErrorLogEvent
# ディスパッチ ロガー。このログ リスナは、各種のログ イベントを
# それに合ったリスナに渡します。
logging.dispatchlogger.class=allaire.jrun.logging.DispatchLogger
logging.dispatchlogger.events={logging.infoevent},{logging.debugevent},
     {logging.warningevent},{logging.errorevent}
logging.dispatchlogger.destinations=filelogwriter,filelogwriter,filelog
     writer,filelogwriter
# スクリーン ログ ライター。すべてのログ イベントが、単に stdout に送信されます。
logging.screenlogger.class=allaire.jrun.logging.ScreenLogWriter
# ファイル ログ ライター。1つのファイル ログ ライターをすべてのイベント タイプに使用したり、
# ディスパッチ ロガーと併用して各イベント タイプがそれぞれ独自のログを使用できます。
# ファイル名は、静的な変数 ({install.rootdir} など) と動的な変数
# ({date}、{hour}、{day}、{month}、{year} など) のどちらでも構成できます。
# 動的な変数を使用すると、ログ イベントのタイムスタンプが変わったときに
# ログ ファイルの名前が変わります。
logging.filelogwriter.class=allaire.jrun.logging.FileLogWriter
logging.filelogwriter.filename={jrun.rootdir}/logs/
     {jrun.server.name}-event.log
logging.filelogwriter.rotationsize=100000
logging.filelogwriter.rotationfiles=5
```

例

このセクションでは、ログ メカニズムの設定例についていくつか説明します。

複数ファイルへのログ メッセージの書き込み

この例では、警告メッセージとエラー メッセージのみを 2 つのファイルに書き込みます。2 つのファイル ライターを設定します。1 つは警告用で、1 つはエラー用です。次の図は、この構成を示します。



これらのステートメントによって、警告メッセージを warning.log というファイルに、エラー メッセージを error.log というファイルに書き込むプロパティが定義されます。既定のプロパティ設定への変更は太字で示します。変更をコメントによって説明します。

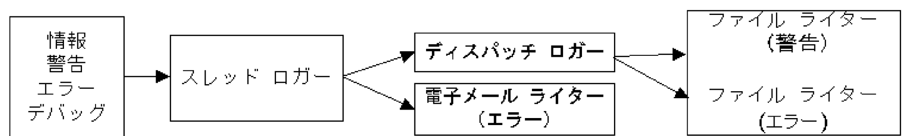
```
logging.class=allaire.jrun.logging.LoggingService
.format={date MM/dd hh:mm:ss} {log.level} ({log.name}) {log.message}

# ログ警告およびエラー メッセージ
logging.loglevel=warning,error

# ログ リスナを定義します。
logging.listeners=threadedlogger

# イベント リスナとして機能する、スレッド ロガーを定義します。
logging.threadedlogger.class=allaire.jrun.logging.ThreadedLogger
# ディスパッチ ロガーをスレッド ロガーのリスナとして定義します。
logging.threadedlogger.listeners=dispatchlogger

# イベントをファイル ライターに転送するディスパッチ リスナを定義します。
logging.dispatchlogger.class=allaire.jrun.logging.DispatchLogger
# イベントを定義します。このステートメント内のイベントの順序に注目してください。
logging.dispatchlogger.events=
    {logging.warningevent},{logging.errorevent}
# 送信先ロガーを定義します。
# この順序は、ステートメント内のイベントの順序に一致します。
logging.dispatchlogger.destinations=fileWarnWriter,fileErrWriter
```

```
# 警告用のファイル ライターを定義します。
logging.fileWarnWriter.class = allaire.jrun.logging.FileLogWriter
logging.fileWarnWriter.filename = {jrun.rootdir}/logs/warning.log
logging.fileWarnWriter.rotationsize = 100000
logging.fileWarnWriter.rotationfiles = 5

# エラー用のファイル ライターを定義します。
logging.fileErrWriter.class = allaire.jrun.logging.FileLogWriter
logging.fileErrWriter.filename = {jrun.rootdir}/logs/errors.log
logging.fileErrWriter.rotationsize = 100000
logging.fileErrWriter.rotationfiles = 5
```

ファイルと電子メールへのログ メッセージの書き込み

この例では、警告メッセージをファイルに、エラー メッセージをファイルと電子メールに書き込みます。

次の図は、この例の設定を示します。



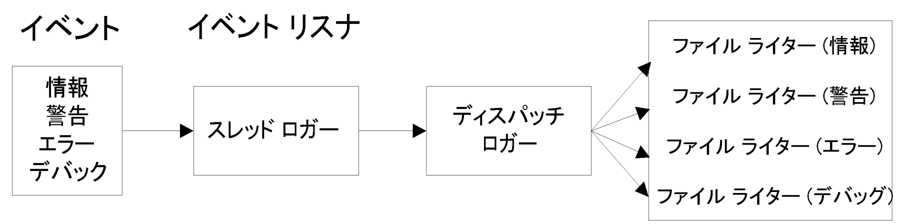
この例では、スレッド ロガーからイベントを受信する 2 つのリスナが定義されています。ディスパッチ ロガーにより、スレッド ロガーからのイベントが受け取られ、定義された 2 つのファイル ライターのどちらかにこれらのイベントが転送されます。電子メール リスナにより、エラー イベントが監視され、これらのイベントに対する電子メール メッセージが生成されます。

これらのステートメントによって、この例のプロパティを定義します。既定のプロパティ設定への変更は太字で示します。変更をコメントによって説明します。

```
logging.class=allaire.jrun.logging.LoggingService
.format={date MM/dd hh:mm:ss} {log.level} ({log.name}) {log.message}

# ログ警告およびエラー メッセージ
logging.loglevel=warning,error

# イベント リスナを指定します。
logging.listeners=threadedlogger

# イベント リスナとして機能する、スレッド ロガーを定義します。
logging.threadedlogger.class=allaire.jrun.logging.ThreadedLogger
# スレッド ロガーのリスナを定義します。
logging.threadedlogger.listeners=dispatchlogger,emailErrWriter

# イベントをファイル ライターに転送するディスパッチ リスナを定義します。
logging.dispatchlogger.class=allaire.jrun.logging.DispatchLogger
# イベントを定義します。このステートメント内のイベントの順序に注目してください。
logging.dispatchlogger.events=
        {logging.warningevent},{logging.errorevent}
# 送信先ロガーを定義します。
# この順序は、ステートメント内のイベントの順序に一致します。
logging.dispatchlogger.destinations=fileWarnWriter,fileErrWriter
# 警告用のファイル ライターを定義します。
logging.fileWarnWriter.class = allaire.jrun.logging.FileLogWriter
logging.fileWarnWriter.filename = {jrun.rootdir}/logs/warning.log
logging.fileWarnWriter.rotationsize = 100000
logging.fileWarnWriter.rotationfiles = 5

# エラー用のファイル ライターを定義します。
logging.fileErrWriter.class = allaire.jrun.logging.FileLogWriter
logging.fileErrWriter.filename = {jrun.rootdir}/logs/errors.log
logging.fileErrWriter.rotationsize = 100000
logging.fileErrWriter.rotationfiles = 5

# 電子メール ライターを定義します。
# 電子メール アドレス、電子メール ホスト、およびメッセージ形式を定義します。
logging.emailErrWriter.class=allaire.jrun.logging.SmtpLogWriter
logging.emailErrWriter.from=JRun-Notification
logging.emailErrWriter.to=someone@mycompany.com
logging.emailErrWriter.host=mymailhost
# 受信するイベントを定義します。
logging.emailErrWriter.loglevel=error
```

ログ メッセージ定義の変更

前のセクションの例で示すように、イベント タイプ メッセージごとに異なるファイルに書き込むようにログ設定を変更できます。次の図は、ディスパッチ ロガーを使用して、ログされたイベントを、異なる複数のファイルに書き込むログ メカニズムを示します。



異なる複数のファイルにログ イベントを書き込むには、イベント タイプに基づいた出力先に、ログ イベントを送り出す**ディスパッチ ロガー**を作成します。この図のように、イベントごとに書き込むファイルを変えるために、個別にファイル ライターを作成できます。

標準出力と標準エラーへのログ出力

既定では、各 JRun サーバーが `System.out-println()` および `System.err-println()` によって書き込まれた情報を受信し、その情報をログ ファイルに書き込みます。ログ ファイルの既定の名前は、次のとおりです。

- `{jrun.rootdir}/logs/{jrun.server.name}-out.log`　標準出力 (`System.out`) に書き込まれる情報。
- `{jrun.rootdir}/logs/{jrun.server.name}-err.log`　標準エラー (`System.err`) に書き込まれる情報。

JRun 自身はこれらのファイルに情報を書き込みません。すべての JRun のログ情報は `{jrun.server.name}-event.log` に書き込まれます。

これらのファイル名と場所を、JRun サーバーの `local.properties` ファイルにあるプロパティを使用して、書き換えることができます。これらのプロパティの詳細については、462 ページの「システム ログ プロパティ」を参照してください。

UNIX では、標準出力と標準エラーに書かれる情報をコンソール ウィンドウに書き込むように JRun を設定することもできます。そのためには、コマンド ラインから `-console` オプションを使用して、JRun を起動します。次の例は、このコマンドを示します。

```
jrun -console -start default
```

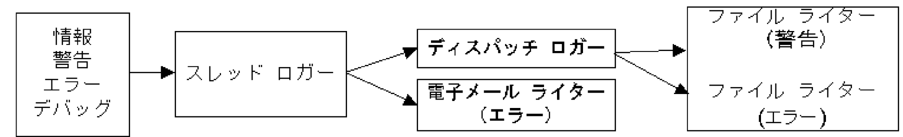
スクリーン ライターの使用法の詳細については、450 ページの「スクリーン ライターの使用」を参照してください。

`jrun` コマンドの詳細については、『JRun セットアップ ガイド』を参照してください。

メモ
ログ ファイルのローテートは標準出力と標準エラーには影響を与えません。

電子メールへのログ

次の図は、ログされたメッセージの電子メールによる通知を追加するようにログ設定を変更できることを示します。



この事例では、2 つの別の出力先にイベントを書き込むためのスレッド ロガーを設定します。ディスパッチ ロガーと 電子メール ライターです。その後、ディスパッチ ロガーはすべてのイベントを別のライターに転送できます。電子メール ライターは、エラー メッセージだけを認識するように設定されています。次に、電子メール ライターは各エラー メッセージを電子メールに書き込みます。

また、この事例では、1 つのイベント (この場合はエラー) を別の 2 種類の出力先 (ファイルや電子メール メッセージ) に送信する方法も示されています。

スクリーン ライターの使用

ログ出力をスクリーンに表示すると、プロジェクトの開発やテスト段階で非常に役立つ場合があります。JRun のデバッグ メッセージの表示のほかに、ログ メソッドをコードに追加し、コンソールにリアル タイムに表示できます。

このセクションでは、スクリーン ログ ライターを使用してログ メッセージをコンソール ウィンドウに出力する方法を説明します。Windows NT/95/98 と UNIX では手順が異なります。

このセクションでは次の用語を使用します。

- **イベント**　メッセージ タイプです。JRun のイベント タイプは、info、warning、error、および debug です。
- **イベント リスナ**　イベントが発生すると、JRun からイベント リスナにそのイベントが転送されます。JRun のスレッド ロガーではイベントをバックグラウンド スレッドでログすることによって最高のパフォーマンスが得られるので、通常はスレッド ロガーをイベント リスナとして使用します。ただし、ライターをイベント リスナとして使用することもできます。

Windows NT/95/98

スクリーン ログ ライターを有効にするには、次の手順を実行します。

1 スクリーン ログ ライターを有効にする JRun サーバーの `local.properties` ファイルを開きます。

2 `java.exe` プロパティを変更して、`{default}` または `javaw.exe` の代わりに `java.exe` へのパスを指定します。

```
# Java Virtual Machine へのパス。ランチャは、必要に応じてシステム PATH を
      使用します。
# 4/17/01 に java.exe に変更しました。
java.exe=C:¥¥jdk1.2.2¥¥bin¥¥java.exe
```

3 `java.System.out` および `java.System.err` プロパティを追加または変更して、参照を行わないようにします。

```
# Primordial 出力およびエラー ストリームの転送
# 標準出力およびエラーがこれらのファイルに転送されないようにするには、
# -console 引数を指定して JRun を実行します。
#  8/2/00 に、スクリーン ロガーが有効にならないように変更しました。
#java.System.out={jrun.rootdir}/logs/{jrun.server.name}-out.log
#java.System.err={jrun.rootdir}/logs/{jrun.server.name}-err.log
java.System.out=
java.System.err=
```

4 `logging.threadedlogger.listeners` プロパティを追加し、`screenlogger` および使用するその他のライターを指定します。

```
# 4/17/01 にスクリーン ロガーを追加しました。
logging.threadedlogger.listeners=screenlogger,filelogwriter
```

ライターは明示的に指定する必要があります。`global.properties` で指定されたライターを有効にする場合は、`{default}` を使用しないでください。

5 プロパティ ファイルを保存します。

6 JRun サーバーをアプリケーションとして再起動します。

メモ
スクリーン ログ ライターは、Windows NT 上で JRun を Windows サービスとして実行している場合は動作しません。JRun を NT サービスとして実行している場合は、[コントロール パネル] の [サービス] でサービスを停止し、[スタート] > [プログラム] > [JRun 3.1] > [*JRun サーバー*] をクリックしてください。別の DOS コンソールにログ エントリが表示されます。スクリーン ログ ライターの出力例については、452 ページの「標準出力」を参照してください。

Unix/Linux

スクリーン ログ ライターを有効にするには、次の手順を実行します。

1 スクリーン ログ ライターを有効にする JRun サーバーの `local.properties` ファイルを開きます。

2 `logging.threadedlogger.listeners` プロパティを追加し、`screenlogger` および使用するその他のライターを指定します。

```
# 4/17/01 にスクリーン ロガーを追加しました。
logging.threadedlogger.listeners=screenlogger,filelogwriter
```

ライターは明示的に指定する必要があります。`global.properties` で指定されたライターを有効にする場合は、`{default}` を使用しないでください。

3 プロパティ ファイルを保存します。

4 JRun サーバーを停止します。たとえば、次のコマンドを発行します。

```
jrun -stop default
```

5 次の例に示すように、`-console` 引数を使用して JRun を再起動します。

```
jrun -console -start default
```

現在のコンソールにログ エントリが表示されます。

標準出力

JRun をバックグラウンドで実行しながらコンソール ウィンドウを開いたら、次のページをブラウザで開いて、デモ サーブレットにアクセスします (ポート番号は異なる場合があります)。

http://localhost:8100/demo/index.html

コンソール ウィンドウの出力は、ログ ファイルの出力と同じです。次に例を示します。

```
09/20 11:14:18 info (JRun) invoker:init
09/20 11:14:18 info (JRun) SimpleServlet:init
09/20 11:14:23 info (JRun) CounterServlet:init
09/20 11:14:26 info (JRun) SnoopServlet:init
```

ログ設定によって、詳細レベルが異なる場合があります。

`java.System.out` プロパティと `java.System.err` プロパティを使用すると、`System.out` メッセージと `System.err` メッセージが `servername-out.log` と `servername-err.log` に出力されます。ただし、JRun のログ アーキテクチャはこれらのファイルには使用されません。

ログ メカニズムの定義

組み込みのログとは異なるログ メカニズムを定義できます。ロガーを作成したら、ロガーに「名前を付けて」定義し、ロガーのプロパティを指定し、サーブレット コードの名前によってそのロガーを要求します。

local.properties ファイルに「ログ グループ」を作成し、ほかのプロパティを使用してロガーの特性を定義し、指定したロガーを定義します。

たとえば、次のステートメントによって、myloggroup というログ グループを定義します。このログ グループでは、デバッグおよび情報レベルのメッセージを mylogfile.log ファイルに書き込みます。各メッセージでは、定義された形式が使用されます。

```
# ログ グループのカンマ区切りリスト
logging.groups=myloggroup

# 'myloggroup' - リスナとそれに対応するロガーを設定します。
# 対応するロガー値は、ログ レベル (debug、info、warning、error、metrics) の
# カンマ区切りリストです。
logging.groups.myloggroup.listeners=mythreadedlogger
logging.groups.myloggroup.mylogger_di=debug,info

# カスタム リスナ
logging.mythreadedlogger.class=allaire.jrun.logging.ThreadedLogger
logging.mythreadedlogger.listeners=myfilelogwriter

# 'myfilelogwriter' - クラス、送信先ファイル、メッセージ形式
logging.myfilelogwriter.class=allaire.jrun.logging.FileLogWriter
logging.myfilelogwriter.filename={jrun.rootdir}/logs/mylogfile.log
logging.myfilelogwriter.format={date MM/dd HH:mm:ss} {log.level}
        {log.message}
```

このステートメントを使用すると、指定されたロガーをサーブレット内で要求できます。

```
Logger logger = LogManager.getLogger("mylogger_di");
```

指定されたロガーのイベントの一覧と一致するログ イベントは (この場合は debug および info)、mylogfile.log に書き込まれます。

ログ メッセージの形式

次の例は、JRun ログから抜き出したものです。

```
05/02 14:48:46 info (JRun) Loading monitor
05/02 14:48:46 warning (license) JRun 3.0 will expire on Jul 15, 2000
05/02 14:48:46 info (JRun) Loading license
05/02 14:48:46 info (JRun) Loading control
05/02 14:48:46 info (license) Enabling unlimited concurrency for JRun
        3.0
05/02 14:48:46 info (control) control listening on *:53000
05/02 14:48:46 info (JRun) Loading ejb
05/02 14:48:52 info (ejb) Loading java:comp/env/ejb/TxnHome...
```

既定のログ メッセージ形式には、日付と時刻、メッセージの種類、メッセージを生成した JRun サービス名、メッセージの内容が含まれます。メッセージの形式は変更できます。

ログ情報の形式

local.properties のプロパティを使用して、ログ メッセージの形式を制御できます。次の例は、既定の形式のログ メッセージです。

```
12/01 04:21:49 info (JRun) default-app Ready.
```

この既定のログ メッセージは次の形式で記述されています。

```
{date MM/dd hh:mm:ss} {log.level} ({log.name}) {log.message}
```

{date MM/dd hh:mm:ss}	日付と時刻の形式で、24 時間形式を使用します。
{log.level}	info、warning、error、またはdebug のイベント タイプです。
({log.name})	かっこ内に表示されたメッセージを生成する JRun サービスです。
{log.message}	メッセージ テキストです。

形式は変更できます。また、メッセージに追加情報を含めることもできます。有効な形式コンポーネントの一覧については、455 ページの「一般プロパティ」の logging.format の説明を参照してください。

次の例では、日、月、およびメッセージだけが含まれるようにメッセージの形式が設定されています。この形式では {log.level} コンポーネントが省略されているので、イベント タイプが 1 つしかないメッセージをファイルに書き込む場合に適しています。

```
logging.format={day} {month} {log.message}
```

次のエラー イベントは、この形式で表示されます。

```
06 12 could not initialize SnoopServlet
```

このエラーは 12 月 6 日に発生し、SnoopServlet サーブレットを JRun が発見できなかったことが原因です。

ログ プロパティ

このセクションでは、JRun ログ メカニズムの設定に使用する次のプロパティのカテゴリについて説明します。

- 「一般プロパティ」455 ページ
- 「スレッド ロガー プロパティ」456 ページ
- 「ディスパッチ ロガー プロパティ」457 ページ
- 「ファイル ライター プロパティ」458 ページ
- 「電子メール ライター プロパティ」461 ページ
- 「スクリーン ライター プロパティ」462 ページ
- 「システム ログ プロパティ」462 ページ

一般プロパティ

このセクションでは、JRun ログ メカニズムを設定して有効にする場合に使用する、一般プロパティについて説明します。

logging.class

ログ サービスのクラスは、`allaire.jrun.logging.LoggingService` です。

logging.format

ログ メッセージの形式です。既定のメッセージ形式は次のとおりです。

```
{date MM/dd HH:mm:ss} {log.level} ({log.name}) {log.message}
```

この形式では、メッセージに日付、時刻、イベント タイプ、イベント 生成者、およびメッセージ テキストが表示されるように設定されています。メッセージ形式を定義する場合、次のコンポーネントが使用できます。

- `{date}`　yyyyMMdd の形式で表された現在の日付
- `{date <format>}`　現在の日付の形式。適切な値については、`{jrun.rootdir}/docs/api` ディレクトリにある Java マニュアルの `java.text.SimpleDateFormat` クラスに関する記述を参照してください。
- `{day}`　日付で、01 ～ 31 の 2 桁の数字
- `{month}`　月で、01 ～ 12 の 2 桁の数字
- `{year}`　年で、4 桁の数字
- `{hour}`　時刻で、00 ～ 23 の 2 桁の数字
- `{julian}`　現在の日付をユリウス暦で表したもの
- `{thread.name}`　現在のスレッド名
- `{thread.hashcode}`　現在のスレッド ハッシュコード
- `{thread.id}`　現在のスレッド ID。この ID は 8 文字の 16 進数形式で表されたハッシュコードです。
- `{log.level}`　ログ イベント タイプ (debug、error、info、warning)

- {log.name}　ログ メッセージを生成する JRun サービスの名前。既定では、この名前はかっこで囲まれます。
- {log.message}　ログ イベント メッセージ

logging.loglevel

カンマ区切りリストとして指定した、ログするイベントのタイプ。使用可能なイベント タイプは、debug、error、info、および warning です。既定のログレベルは、次のように設定されています。

info、warning、error

ログ イベントの一覧が空白の場合は、すべてのログ イベントが記録されます。さらに、ログ メカニズムは Web サーバーと JRun の間の接続についてのメトリック イベントを受信できます (第 37 章を参照)。

logging.listeners

ログ イベント リスナのカンマ区切りリストです。このプロパティを使用して、ログ メカニズムを有効にします。このリストが空白の場合、ログ イベントは記録されません。このリストは、実際のイベント ログを記録しているリスナの論理名に相当します。

JRun のインストール時に行われるログ メカニズムの既定の設定では、このプロパティには既定のスレッド ロガーの論理名 threadedlogger が設定されています。この設定では、スレッド ロガーにより 、すべてのログ イベント が記録され、その後、既定のディスパッチ ロガーに転送されます。ログ メカニズムの既定の設定の説明については、444 ページの「既定のログ設定」を参照してください。

ここで指定された各リスナについて、リスナを定義するためのプロパティを、プロパティ ファイルに指定する必要があります。

スレッド ロガー プロパティ

これらのプロパティによって、logging.listeners プロパティで指定された threadedlogger を定義します。logging.listeners プロパティにほかのロガーを含める場合は、これらのプロパティの threadedlogger をロガーの名前に置き換え、このセクションのプロパティを使用して、ロガーのクラスとリスナを定義する必要があります。

このセクションで説明するプロパティの形式は次のとおりです。

logging.<*loggerName*>.*propertyName* = *propertyValue*

ここで *loggerName* は、前のセクションで説明した logging.listeners プロパティで定義したロガーの論理名です。

既定の設定では、JRun により、次のプロパティが設定された threadedlogger スレッド ロガーが作成されます。

```
logging.threadedlogger.class=allaire.jrun.logging.ThreadedLogger
logging..threadedlogger.listeners=filelogwriter
```

既定では、スレッド ロガーによって 1 つのリスナが定義されます。このリスナによってすべてのイベントがロガーから 1 つのファイル ログ ライターに送信されます。

logging.<loggerName>.class

スレッド ロガー クラスの名前。`allaire.jrun.logging.ThreadedLogger` のプロパティ値を使用する必要があります。

logging.<loggerName>.listeners

スレッド ロガーのキューからイベントを受け取るリスナを指定します。1 つ以上のリスナと、ディスパッチ ロガーか、またはライターを 1 つ定義する必要があります。

既定では、スレッド ロガーに指定されたリスナのみが `filelogwriter` で、これは既定のファイル ライターの論理名です。次に、既定のファイル ライターは 1 つのファイル ライターにすべてのイベントを送ります。ログ メカニズムの既定の設定の説明については、444 ページの「既定のログ設定」を参照してください。

ディスパッチ ロガー プロパティ

ディスパッチ ロガーはログ イベントを受け取り、1 つ以上のログ リスナにこのイベントを転送します。ディスパッチ ロガーを使用すると、イベントのタイプに応じて、イベントを受け取るリスナを選択できるようになります。したがって、4 種類のイベント タイプに対応させて、4 種類のリスナにイベントを転送できます。

このセクションで説明するプロパティの形式は次のとおりです。

`logging.`*`<dispatchLoggerName>.propertyName = propertyValue`*

ここで、*`dispatchLoggerName`* には、使用しているディスパッチ ロガーの論理名が入ります。

既定の設定では、JRun により、次のプロパティを使用して、ディスパッチ ロガーが 1 つ定義されます。

```
logging.dispatchlogger.class=allaire.jrun.logging.DispatchLogger
logging.dispatchlogger.events={logging.infoevent},{logging.debugevent},
        {logging.warningevent},{logging.errorevent}
logging.dispatchlogger.destinations=filelogwriter,filelogwriter,filelog
        writer,filelogwriter
```

logging.<dispatchLoggerName>.class

スレッド ロガー クラスの名前。`allaire.jrun.logging.DispatchLogger` の値を使用する必要があります。

logging.<dispatchLoggerName>.events

カンマ区切りリストとしてディスパッチ ロガーにより送信されるイベント。既定値は次のとおりです。

```
{logging.infoevent},{logging.debugevent},{logging.warningevent},
    {logging.errorevent}
```

ディスパッチ ロガーで転送する必要のあるイベントだけを指定します。

logging.<dispatchLoggerName>.destinations

ディスパッチ ロガーからのイベントの出力先となるリスナのカンマ区切りリスト。リスナは `logging.dispatchlogger.events` プロパティで指定されたイベントの順に指定してください。

既定値は次のとおりです。

```
filelogwriter,filelogwriter,filelogwriter,filelogwriter
```

この設定では、イベントはすべて既定のファイル ライターに書き込まれます。

ファイル ライター プロパティ

このセクションでは、ファイル ライターの設定方法について説明します。ファイル ライターはイベントを受けとって、ファイルに書き込みます。

このセクションで説明するプロパティの形式は次のとおりです。

logging.*<fileLoggerName>.propertyName = propertyValue*

ここで、*fileLoggerName* には、使用しているファイル ライターの論理名が入ります。

JRun のインストール時に行われたログ メカニズムの既定の設定では、次のプロパティを使用して、ファイル ライターが 1 つ定義されています。

```
logging.filelogwriter.class = allaire.jrun.logging.FileLogWriter
logging.filelogwriter.filename = {jrun.rootdir}/logs/
        {jrun.server.name}-event.log
logging.filelogwriter.rotationsize = 100000
logging.filelogwriter.rotationfiles = 5
```

これらのプロパティにより、`filelogwriter` ファイル ライターが作成されます。既定の設定では、すべてのイベントは、JRun のインストール ディレクトリの `logs` ディレクトリにある `{jrun.server.name}-event.log` ファイルに書き込まれます。

logging.<fileLoggerName>.class

ファイル ライター クラスの名前。`allaire.jrun.logging.FileLogWriter` が指定されます。

logging.<fileLoggerName>.filename

このファイル ロガーによってすべてのイベントが書き込まれるファイル名。

既定では、このファイル名は `{jrun.server.name}-event.log` で、JRun インストール ディレクトリの `logs` ディレクトリにあります。ここで、`{jrun.server.name}` には、`default` などの JRun サーバー名が入ります。

logging.<fileLoggerName>.rotationsize

ファイルがローテートされるまでの、ログ ファイルのサイズの最大値 (単位はバイト)。ファイルがローテートされると、ログ メカニズムによる現在のログ ファイルへの書き込みは停止され、新しいログ ファイルが作成されます。すべての新しいイベントは、新しいログ ファイルに書き込まれます。このプロパティを使用して、ログ ファイルの最大サイズを制御できます。

保持するログ ファイルの数を指定するには、ログ メカニズムの `rotationfiles` プロパティを使用します。たとえば、`rotationfiles` を 2 に設定すると、ログ メカニズムによって 2 つのローテート ファイルを持つログ ファイルがログ ライター用に確保されます。すでに 2 つのログ ファイルがあるときに、あるイベントにより、ログ ファイルがローテート サイズを超えてしまった場合、古い方のログ ファイルが削除され、新しいファイルが作成されます。

`rotationsize` の既定値は 100000 です。サイズはバイト、キロバイト (10 k) またはメガバイト (10 m) 単位で指定します。

たとえば、ログ イベントを `event.log` ファイルに書き込むようにファイル ライターを設定し、`rotationfiles=3` および `rotationsize=100000` に設定します。次のログ イベントによってログ ファイルのサイズが 100000 バイトを超えると、ログ ファイルがローテートされ、次の順序のようになります。

- `event_3.log` ファイルがある場合は、このファイルが削除されます。
- `event_2.log` ファイルがある場合は、このファイルの名前が `event_3.log` に変更されます。
- `event_1.log` ファイルがある場合は、このファイルの名前が `event_2.log` に変更されます。
- `event.log` の名前が `event_1.log` に変更されます。
- `event.log` が作成されます。

logging.<fileLoggerName>.rotationfiles

ローテート ファイルの数。上記の「logging.<fileLoggerName>.rotationsize」に関する注意事項を参照してください。

既定値は 5 です。

logging.<fileLoggerName>.format

ログ メッセージの形式を指定します。この設定は、`logging.format` プロパティを使用して指定したログ形式を書き換えます。このプロパティの値の一覧については、455 ページの「一般プロパティ」にある「logging.format」の説明を参照してください。

logging.<fileLoggerName>.heading

JRun により最初にログ ファイルに書き込みが行われるときに、このファイルの先頭に挿入されるログ ファイル ヘッダ。ヘッダには、タイムスタンプなどの動的なプロパティを含むどのようなテキストでも使用できます。次の例では、ログ ファイルに作成日時が書き込まれます。

```
logging.filelogwriter.heading==# 作成日 {date MM/dd hh:mm:ss}
```

また、プロパティ `logging.fileLoggerName.heading.lineN` を使用して、ヘッダを挿入することもできます。`logging.fileLoggerName.heading` プロパティ、または `logging.fileLoggerName.heading.lineN` プロパティのどちらかを使用できます。両方は使用できません。

logging.<fileLoggerName>.heading.lineN

JRun により最初にログ ファイルに書き込みが行われるときに、このファイルの先頭に挿入される複数行のログ ファイル ヘッダの 1 行。プロパティ コンポーネント `lineN` には、ヘッダの行番号を指定します。

ヘッダには、タイムスタンプなどの動的なプロパティを含むどのようなテキストでも使用できます。次の例では、ログ ファイルに作成日時が書き込まれます。

```
logging.filelogwriter.heading.line1=#--------------------------------
logging.filelogwriter.heading.line2=# 作成日{date MM/dd hh:mm:ss}
logging.filelogwriter.heading.line3=# これが JRun ログ ファイルです。
logging.filelogwriter.heading.line4=#--------------------------------
```

行番号は 1 から始まる必要があります。連続する次の行が見つからないと、ログ ライターによるヘッダ行の検索は終了します。

電子メール ライター プロパティ

ログ メッセージを電子メール メッセージに転送するには、電子メール ライターを作成します。このセクションでは、電子メール ライターに設定するプロパティについて説明します。例については、447 ページの「ファイルと電子メールへのログ メッセージの書き込み」を参照してください。

このセクションで説明するプロパティの形式は次のとおりです。

logging.*<mailLoggerName>.propertyName* = *propertyValue*

ここで、*mailLoggerName* には、使用する電子メール ライターの論理名を指定します。

logging.<mailLoggerName>.class

メール ライター クラス名。allaire.jrun.logging.SmtpLogWriter が指定されます。

メモ

電子メール サーバーでは、sun.net.smtp.SmtpClient が必要ですが、すべてのサーバーでこのクライアントが使用できるわけではありません。

---

logging.<mailLoggerName>.from

生成された電子メール アドレスの「from」の部分。この値に空白を入れることはできません。

logging.<mailLoggerName>.to

生成された電子メール アドレスの「to」の部分。この値に空白を入れることはできません。

logging.<mailLoggerName>.host

電子メール ホスト。この値に空白を入れることはできません。

logging.<mailLoggerName>.loglevel

電子メール メッセージを生成するログ イベントのカンマ区切りリスト。指定できる値は info、warning、error、および debug です。

logging.<mailLoggerName>.format

ログ メッセージの形式を指定します。この設定は、logging.format プロパティを使用して指定したログ形式を書き換えます。このプロパティの値の一覧については、455 ページの「一般プロパティ」にある logging.format プロパティの説明を参照してください。

スクリーン ライター プロパティ

このセクションでは、スクリーン ライターの設定方法について説明します。スクリーン ライターはイベントを受け取り、これを標準出力に書き込みます。標準出力は通常、使用しているコンピュータのスクリーンになります。

このセクションで説明するプロパティの形式は次のとおりです。

`logging.<screenLoggerName>.propertyName = propertyValue`

ここで、*screenLoggerName* には、使用するスクリーン ライターの論理名を指定します。

logging.<screenLoggerName>.class

スクリーン ライター クラス名を指定します。

`allaire.jrun.logging.ScreenLogWriter` の値を使用する必要があります。

logging.<screenLoggerName>.format

ログ メッセージの形式を指定します。この設定は、`logging.format` プロパティを使用して指定したログ形式を書き換えます。このプロパティの値の一覧については、455 ページの「一般プロパティ」にある「logging.format」の説明を参照してください。

システム ログ プロパティ

既定では、Web アプリケーションによって標準出力と標準エラーに書かれた情報を、JRun サーバーが取得し、ログ ファイルに書き込みます。次のプロパティを使用して、これらのログ ファイルの名前と場所を設定し、ログ ファイルにスタック トレースを含めるように JRun を設定できます。

java.System.out

標準出力に書き込まれた情報を含むファイルの、名前と場所を設定します。既定では、JRun はこれらの情報を次のファイルに書き込みます。

`{jrun.rootdir}/logs/{jrun.server.name}-out.log`

ここで、`{jrun.server.name}` には、JRun サーバーの名前が入ります。

java.System.err

標準エラーに書き込まれた情報のためのファイルの名前と場所を設定します。既定では、JRun はこれらの情報を次のファイルに書き込みます。

`{jrun.rootdir}/logs/{jrun.server.name}-err.log`

ここで、`{jrun.server.name}` には、JRun サーバーの名前が入ります。

第 39 章

Web アプリケーションの認証

多くの Web サーバーとアプリケーションにとってセキュリティは非常に重要です。セキュリティによって、承認されたユーザだけが Web サイト上のリソースにアクセスできることが保証されます。最新の Java サーブレット API では、Web アプリケーション レベルでのリソースのユーザ アクセス権限を制御する認証メカニズムが定義されています。JRun はこの認証メカニズムをサポートしています。

この章では、JRun 認証メカニズムと認証制限を Web アプリケーションに適用する方法について説明します。

目次

- 認証 464
- アプリケーション認証の設定 469
- サーバー認証メカニズムの制御 478
- 認証プロパティ 482

認証

セキュリティはインターネット上で公開するアプリケーションにとって重要です。インターネット アプリケーションに関するセキュリティ問題に対処するため、Java サーブレット API バージョン 2.2 の仕様書では、Web アプリケーション内部のリソースへのユーザ アクセスを制御するための認証メカニズムが定義されています。JRun では、Java サーブレット API に基づいて、最新のセキュリティ メカニズムをサポートしています。

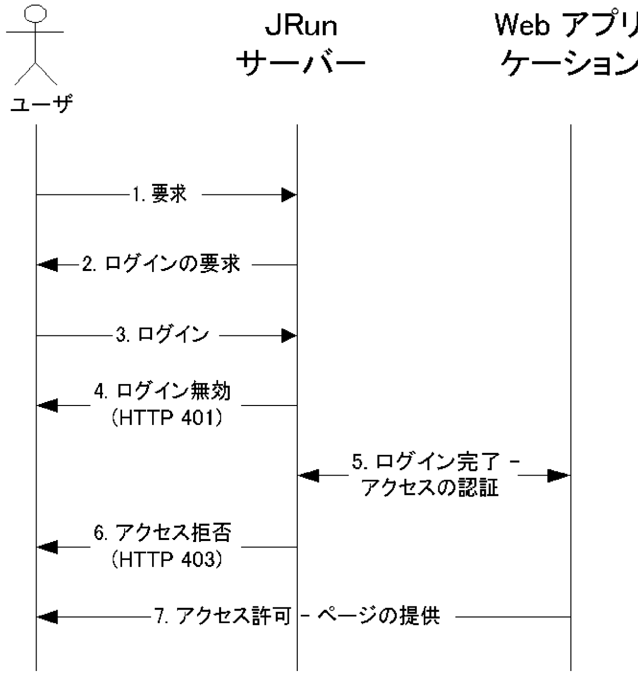
認証メカニズムはロール ベースです。つまり、Web アプリケーションにアクセスするすべてのユーザには 1 つまたは複数のロールが割り当てられます。ロールの例は、manager、developer、および customer です。

アプリケーション開発者は、Web アプリケーション、またはアプリケーションを構成する個々のリソースに用途別ロールを割り当てることができます。JRun では、ユーザに Web アプリケーション リソースへのアクセス権限が与えられる前に、ユーザが認証されていること (つまりログインしていること)、およびユーザにリソースへのアクセス権を持つロールが割り当てられていることが確認されます。Web アプリケーションの非認可アクセスは、非認可アクセスであることを示す HTTP 403 エラーとなります。

認証には、ユーザに関する情報をWeb サイトに保存する必要があります。この情報には、各ユーザに割り 当てられているロールが含まれています。また、ユーザ アクセスを認証する Web サイト には通常、ログイン メカニズムが実装され、パスワード によって各ユーザの ID を検証します。Web サイトでユーザが承認されると、サイトはユーザのロールが判別できます。

認証の例

認証メカニズムがどのように行われるかを示すには、例を示すのが最適です。この例では、developer のロールを割り当てられているユーザだけが Web アプリケーションにアクセスできます。



次に示す手順でこのメカニズムを説明します。

1 ユーザは Web アプリケーション リソースを要求します。Web アプリケーションでは、アプリケーションからページが返される前に要求が認証されなければなりません。

   認証の必要条件は、アプリケーション開発者が Web アプリケーション レベルで設定します。JRun サーバーによって実行される Web アプリケーションでは、認証を個別に有効または無効に設定できます。

2 アプリケーションを実行するアプリケーション サーバーは要求をトラップし、ユーザにログインを要求します。

   ユーザがすでにログインしている場合、この手順は省略されます。

3 ユーザは、ユーザ名とパスワードを指定してログインします。

4 ログインが無効の場合、JRun からユーザに HTTP 401 エラー (非認可アクセス) が返されます。

5 ユーザ名とパスワードが有効であれば、アプリケーション サーバーは、ユーザがアプリケーションへのアクセスに必要なロールが割り当てられているかどうかを判別します。

6 ユーザがリソースへのアクセス権を持っていない場合、JRun からユーザに HTTP 403 エラー (非認可アクセス) が返されます。

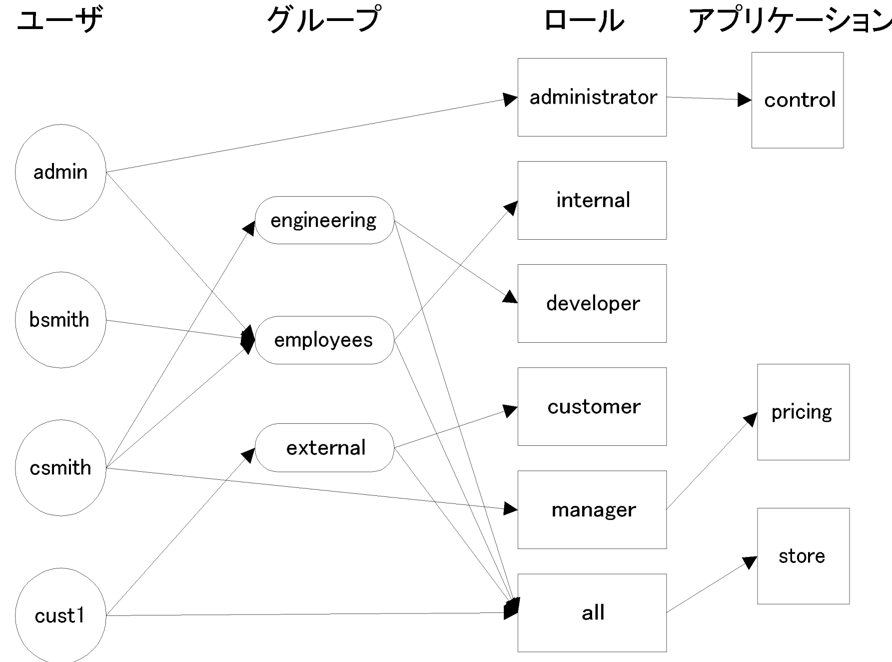
7 ユーザがリソースへのアクセス権を持っている場合、アプリケーション サーバーは要求されたページのサービスを実行します。

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

ユーザ、グループ、およびロール

認証は各ユーザに割り当てられているロールに基づいて行われます。ユーザが Web アプリケーションにアクセスするには、アプリケーションにアクセスする権限を与えられているロールがユーザに割り当てられている必要があります。

ユーザ、グループ、およびロールの 3 つのエンティティから構成される階層によってユーザを配置します。次の図は、この階層を示します。



この図に示すように、ユーザにはグループを割り当てたり、ロールを直接割り当てることができます。グループを使用すると、ユーザを 1 つにまとめて、ユーザ グループ全体に特定のロールを割り当てることができます。

この図では、enigineering、employees、および external の 3 つのグループにユーザが分類されています。ユーザとグループには、特定のロールである、administrator、internal、developer、customer、manager、および all が割り当てられます。

アプリケーション認証とサーバー認証

認証には 2 つの異なる部分があります。Web サイトでアプリケーション認証を行うには、両方とも実装する必要があります。

認証の最初の部分は、アプリケーション レベルで行われます。アプリケーション開発者は、アプリケーションへのアクセス権を持つロールを割り当てます。認証のこの部分は定義段階と考えることができます。つまり、アプリケーション開発者は、アプリケーションにアクセスするのに必要なアクセス ロールを定義します。アプリケーションの認証権の設定の詳細については、469 ページの「アプリケーション認証の設定」を参照してください。

認証の 2 番目の部分は、アプリケーションを実行するアプリケーション サーバーによって行われます。アプリケーション サーバーでは、ユーザの証明を検証します。通常はログイン メカニズムによってユーザを認証し、次に Web アプリケーションに対するユーザのアクセス権を認証します。認証のこの部分を実施段階と考えることができます。認証に関するアプリケーション サーバーの設定の詳細については、478 ページの「サーバー認証メカニズムの制御」を参照してください。

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

サーバーが認証を実施する時期

認証は要求があるたびに行われます。JRun サーバーは Web アプリケーションに対するすべての要求を確認して認証します。

1 台の JRun サーバーで複数のアプリケーションを処理できるので、JRun サーバー内の 1 つのアプリケーションへのアクセス権を持つユーザは、JRun サーバー内の同じアクセス権を持つほかのすべてのアプリケーションにもアクセスできます。

JRun 認証メカニズムの設定

JRun 認証メカニズムを設定するには、プロパティ ファイルを使用します。JRun 管理コンソール (JMC) によって認証を制御することはできません。

JRun の 1 つのインストールに関連付けられているすべての JRun サーバーの認証メカニズムの既定の設定は global.properties ファイルに保存されます。個々の JRun サーバーの既定の設定を変更する場合は、該当するサーバーの local.properties ファイルの設定を変更します。

JRun 認証メカニズムの既定の設定は次のとおりです。

```
# 認証
authentication.service=propfile
authentication.class=allaire.jrun.servlet.ResourceAuthenticator
authentication.propfile.class=allaire.jrun.security.PropertyFileAuthent
        ication
authentication.propfile.filename={jrun.rootdir}/lib/users.properties
```

これらの設定は JRun 認証サービスの名前を propfile と定義し、認証メカニズムを定義する Java クラスの名前を指定して、users.properties ファイルの場所を指定します。JRun サーバーで実装されている認証メカニズムでは、users.properties を使用してユーザ、グループ、およびロールに関する情報を保持します。このファイルの詳細については、478 ページの「既定の JRun 認証メカニズムの使用法」を参照してください。

JRun サーバーの JRun 認証メカニズムを無効にするには、次のように、local.properties にある webapp.services プロパティから認証サービスを削除します。

```
# global.properties から継承された
# 既定の webapp.services プロパティ設定値
# webapp.services=scheduler,logging,session,authentication,jsp,file
# 認証を削除する修正されたプロパティ
webapp.services=scheduler,logging,session,jsp,file
```

このプロパティ設定は、JRun サービスの一覧から JRun 認証メカニズムのサービスを削除し、JRun サーバーの認証が行われないようにします。

アプリケーション認証の設定

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

この情報をアプリケーション サーバーでどのように解釈して適用するかを設定する方法については、478 ページの「サーバー認証メカニズムの制御」を参照してください。

アプリケーションの開発と公開の一環として、次の手順を実行してアプリケーションの認証を設定する必要があります。

1 アプリケーションにアクセス ロールを割り当てる。

2 アプリケーション サーバーがユーザを認証する方法を指定する。

アプリケーション認証を制御する情報は、アプリケーションの公開記述子である `web.xml` ファイルに含まれています。この情報を設定するには、`web.xml` を編集する必要があります。

Web アプリケーションへの認証ロールの割り当て

Web アプリケーションへのアクセスは、ユーザがアプリケーションまたはアプリケーション リソースにアクセスするときに必要なロールを設定することによって制御します。つまり、ユーザがアプリケーションにアクセスするには、アプリケーションにアクセスする権限を持つロールが割り当てられている必要があります。

アクセス制限は、アプリケーション全体に割り当てたり、アプリケーション内部のリソースに割り当てることができます。アプリケーションへのアクセス権を持つロールを定義しているアプリケーションの `web.xml` ファイルの一部を次に示します。

```
<web-app>
  ...

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Store Application</web-resource-name>
      <url-pattern>/store/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
      <description>Sales Info Resource</description>
    </web-resource-collection>
    <auth-constraint>
      <role-name>manager</role-name>
      <description>Managers only</description>
    </auth-constraint>
  </security-constraint>

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

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

- アクセスを制限する Web アプリケーション リソースの URL パターン。

  この例では、URL に /store が含まれているすべてのアプリケーション リソースへのアクセスを制御しています。アプリケーション全体を認証するには、URL パターンを * に設定します。

  <url-pattern>*</url-pattern>

- アプリケーション リソースの HTTP アクセス メソッドが GET および POST であること。http-method 要素を省略すると、すべてのアクセス メソッドが認証されます。
- URL へのアクセス ロールが manager だけであること

アプリケーションにアクセスできるロールの一覧を拡張できます。次の例では、Web アプリケーションへのアクセス権を持つロールに developer のロールを追加します。

```
<auth-constraint>
  <role-name>manager</role-name>
 <role-name>developer</role-name>
  <description>Managers and developers</description>
</auth-constraint>
```

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

```
<web-app>
  ...

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Store Application</web-resource-name>
      <url-pattern>/store/servlet/*</url-pattern>
      <url-pattern>/store/pricing/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
      <description>Sales Info Resource</description>
    </web-resource-collection>
    <auth-constraint>
      <role-name>manager</role-name>
      <description>Managers only</description>
    </auth-constraint>
  </security-constraint>

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

<security-constraint> 要素には次の構文があります。

```
<security-constraint>
  (web-resource-collection+, auth-constraint?, user-data-constraint?)>
</security-constraint>
```

この構文では、+ 記号は、1 つまたは複数の web-resource-collection 要素を定義できることを意味し、? は、auth-constraint および user-data-constraint 要素をオプションで指定できることを示します。

<web-resource-collection> 要素には次の構文があります。

```
<web-resource-collection>
  (web-resource-name, description?, url-pattern*, http-method*)
</web-resource-collection>
```

web-resource-name 要素は必ず指定します。また、description 要素を指定することもできます。* は、url-pattern および http-method 要素を 0 個以上指定できることを示します。

メモ

JRun では、web.xml 内で認証要素 user-data-constraint、security-role、および res-auth の使用をサポートしていません。

サーブレット からのロール情報へのアクセス

サーブレット内から、サーブレットにアクセスするクライアントのロール名とユーザ名にアクセスできます。これにより、ユーザのロールまたはユーザ名に基づいてサーブレットの条件付きで実行できます。たとえば、ユーザが customer のロールに含まれている場合はサーブレットはある処理を実行し、manager のロールに含まれている場合は別の処理を実行します。

サーブレット 内では、Java サーブレット の開発時はサーブレット の HttpServletRequest オブジェクトを使用し、JSP ページの作成時は request オブジェクトを使用して、サーブレット を要求したユーザの情報を取得します。これらのメソッド を次に示します。

- getRemoteUser　ユーザがログインしている場合は、要求を出したユーザのログイン名を、ログインしていない場合は、ヌルを返します。
- isUserInRole　メソッドに渡されたロール名にユーザが含まれていれば、true を返します。
- getUserPrincipal　ユーザがログインしている場合はユーザ名が含まれている java.security.Principal オブジェクトを、ログインしていない場合はヌルを返します。

次の例では、ユーザが Web アプリケーション内のサーブレット mySservlet に対し、all または manager 認証ロールが必要であることを指定しています。

```
<web-app>
  ...

  <servlet>
    <servlet-name>
      myServlet
    </servlet-name>
    <servlet-class>
      myServlet
    </servlet-class>
  <security-role-ref>
      <role-name>
        all
      </role-name>
    </security-role-ref>
    <security-role-ref>
      <role-name>
        manager
      </role-name>
    </security-role-ref>
  </servlet>

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

メモ
この指定では、サーブレット自体の認証を制限しません。認証制限はアプリケーションレベルで指定します。この指定では、サーブレットに対して適用される認証制限を認識することができます。

サーブレット内部では、次の Java サーブレット の例に示すように、ユーザのロールを判別してサーブレットを条件付きで実行できます。

```
public void service(HttpServletRequest req, HttpServletResponse resp)
  throws IOException{
  ...

  if(req.isUserInRole("manager")) {
    // manager として処理します。
  }

  if(req.isUserInRole("all")) {
    // ほかのすべてのユーザとして処理します。
  }
```

JSP ページ内で使用する場合は、次の例を使用してページを条件付きで処理できます。

```
<%
  if(req.isUserInRole("manager")) {
    // manager として処理します。
  }
```

```
  if(req.isUserInRole("all")) {
    // ほかのすべてのユーザとして処理します。
  }
%>
```

サーブレットのもう 1 つのオプションは、サーブレットの web.xml 定義で認証リンクを設定できることです。サーブレット開発者はロールを表す語句を使用でき、その語句をアプリケーション開発者が指定した実際のロール名にリンクさせることができます。アプリケーション開発者がアプリケーションに関連付けられているロールを変更した場合でも、JSP 開発者は web.xml ファイル内のロールのリンクを変更するだけで済みます。

次の例は、ロールのリンクを使用したサーブレットの web.xml 定義を示します。

```
<web-app>
  ...

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

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

この例では、サーブレットは manager のロールを MGR としています。一方、<role-link> タグでは、Web アプリケーション レベルで設定されているサーブレットの MGR ロール指定と manager 指定間のリンクを定義しています。このリンクによって、サーブレットのロールの定義がアプリケーションの定義と関連付けられます。

supervisor のロール名を使用してマネージャのロールを認証するように後で変更された場合は、サーブレットの定義を次のように変更します。

```
<role-name>
  MGR
</role-name>
<role-link>
  supervisor
</role-link>
```

ロールのリンクによって、サーブレットは、`<role-name>` タグで定義されているロールに基づいて常に条件付きで処理されます。このとき、`<role-link>` タグを使用して Web アプリケーションの一部としてこのようなサーブレットを使用すると、サーブレットのロールの定義をアプリケーションの定義と関連付けることができます。ロールのリンク機能により、複数のアプリケーション間でサーブレットを移植できるようになります。

ユーザ認証メソッドの設定

アプリケーション サーバーでアプリケーションに対するユーザのアクセス権を認証するには、サーバーでユーザを識別できることが必要です。ユーザを識別すると、サーバーは、ユーザに割り当てられているロールを識別し、それによってユーザのアクセス権を識別できます。

ユーザが最初に Web アプリケーションにアクセスすると、アプリケーション サーバーはユーザ名とパスワードを使用してログインするようにユーザに要求します。このログイン情報から、サーバーはユーザのロールを判別できます。

アプリケーションの `web.xml` ファイルでは、次の 2 通りの認証メソッドを設定できます。これらのメソッドによって、アプリケーション サーバーがユーザにログイン情報を要求する方法が決まります。

- `BASIC`　HTTP 要求 / 応答メカニズムを使用してユーザをログインさせます。
- `FORM`　アプリケーション開発者がログイン フォームを表示します。

メモ

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

BASIC 認証

`BASIC` 認証は HTTP 要求 / 応答メカニズムを使用して現在のユーザを認証します。このタイプの認証は、次のような一連のイベントによって行われます。

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

2 JRun がユーザのブラウザに HTTP エラー 401 (非認証アクセス) を返します。

3 ブラウザには、ユーザ名とパスワードの入力を要求するプロンプトが表示されます。ブラウザに表示されるログイン プロンプトは次のとおりです。

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

ユーザ名とパスワードが有効であれば、サーバーは、ユーザがアプリケーションにアクセス権があることを認証します。ユーザ名とパスワードが有効な場合は、要求されたページが表示され、それ以外の場合は、JRun からユーザに HTTP 403 エラーが返されます。

ユーザが認証プロセスをキャンセルすると、サーバーから HTTP 401 エラー ページが返されます。

465 ページの「認証の例」の図でもこの手順を示しています。

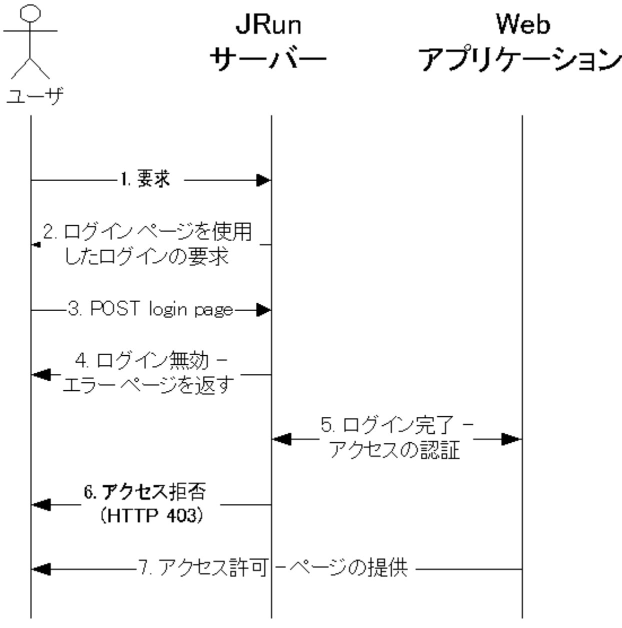
アプリケーションで BASIC 認証を使用するように指定するには、web.xml に次のコードを追加します。

```
<web-app>
  ...

  <login-config>
  <auth-method>
   BASIC
  </auth-method>
  <realm-name>
   Sales
  </realm-name>
  </login-config>

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

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

FORM 認証

FORM 認証により HTML フォームを使用してユーザ名とパスワードを取得できます。次の図は、FORM 認証の手順を示します。



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

```
<web-app>
  ...

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

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

ユーザが Web アプリケーションを要求すると、JRun サーバーは `<form-login-page>` 要素によって指定されているページをユーザに転送します。次の HTML ページ `login.htm` は、このログイン ページの例です。

```
<html>
<head>
<title>My Login Page</title>
</head>
<body>
<center>
<h2>You have requested a secured page.Please login.</h2>

<form method="POST" action="j_security_check">
<table>
<tr><td>User</td><td><input type=text name="j_username"></tr>
<tr><td>Password</td><td><input type=password name="j_password"></tr>
</table>
<br><br>
<input type=submit>
</form>

</center>
</body>
</html>
```

使用するフォームには次の設定が含まれている必要があります。

- **アクション**　フォームは `j_security_check` のアクションを使用して渡す必要があります (POST のみ)。アプリケーションを実行するサーバーは、このアクションを認識してフォームを処理します。
- **ユーザ名**　ユーザ名は `j_username` というフィールドに保存する必要があります。
- **パスワード**　パスワードは `j_password` というフィールドに保存する必要があります。

ユーザ名とパスワードが有効であれば、JRun は、ユーザがアプリケーションにアクセス権があることを認証します。ユーザ名とパスワードが有効な場合は、要求されたページが表示され、それ以外の場合は、JRun からユーザに HTTP 403 エラーが返されます。ユーザ名またはパスワードが無効な場合、`<form-error-page>` タグで指定されているページがユーザに転送されます。

サーバー認証メカニズムの制御

このセクションでは、Web アプリケーションを実行するアプリケーション サーバーの認証メカニズムを設定する方法について説明します。認証メカニズムの選択には、アプリケーションの実行時に使用するサーバーに基づいて、次の 3 つの選択肢があります。

- 既定の JRun 認証メカニズムを使用するアプリケーション サーバー

  JRun を使用して Web アプリケーションを実行します。既定では、JRun アプリケーション サーバーは JRun に用意されている認証メカニズムを使用します。詳細については、478 ページの「既定の JRun 認証メカニズムの使用法」を参照してください。

- カスタム認証メカニズムを使用する JRun アプリケーション サーバー

  JRun を使用して Web アプリケーションを実行しますが、独自の認証メカニズムを定義します。グループやロール情報が含まれている既存のユーザ データベースがあり、そのデータベースを使用してアプリケーションへのユーザ アクセスを認証する場合、このメソッドが多くの場合必要になります。詳細については、481 ページの「JRun によるカスタム認証メカニズムの使用」を参照してください。

- 独自の認証メカニズムを使用するその他のアプリケーション サーバー

  JRun は、Web アプリケーションの認証に関する最新の業界標準仕様 (Java サーブレット API バージョン 2.2 の仕様) に準拠しています。このため、JRun で開発するアプリケーションは、この仕様に準拠しているほかのすべての Web アプリケーション環境に移植できます。

  異なるアプリケーション サーバー上で Web アプリケーションを実行する場合は、アプリケーションの実行に使用するサーバーで独自の認証メカニズムを定義します。詳細については、482 ページの「JRun の外部でのアプリケーションの実行」を参照してください。

既定の JRun 認証メカニズムの使用法

JRun サーバーには、サーバーで実行される Web アプリケーションを認証するための組み込みメカニズムがあります。この認証メカニズムでは、ユーザ、グループ、およびロールに関する情報を 1 つのファイルを使用して記録します。ユーザがアプリケーションにアクセスすると、JRun サーバーはユーザのロールを判別し、そのロールでアプリケーションにアクセスできるかどうかを決めます。

既定の JRun 認証メカニズムは、`global.properties` ファイルに次のコードを設定して定義します。

```
# 認証
authentication.service=propfile
authentication.class=allaire.jrun.servlet.ResourceAuthenticator
authentication.propfile.class=allaire.jrun.security.PropertyFileAuthent
        ication
authentication.propfile.filename={jrun.rootdir}/lib/users.properties
```

authentication.service プロパティでは認証サービス名を propfile と指定します。残りのプロパティでは、認証メカニズムを定義する Java クラスの名前を設定し、users.properties ファイルの場所を設定します。JRun サーバーで実装されている認証メカニズムでは、users.properties を使用してユーザ、グループ、およびロールに関する情報を保持します。

組み込み型の JRun 認証メカニズムでは、すべての JRun サーバーはユーザ、グループ、およびロール情報が格納されている 1 つの users.properties ファイルを共有します。このため、すべてのアプリケーションは同じユーザ認証情報を共有します。

メモ

独自のファイルを使用するように個々の JRun サーバーを設定し、そのサーバー専用のユーザ情報を格納できます。このように設定するには、サーバーの local.properties ファイルを編集してプロパティ authentication.propfile.filename を追加し、users.properties ファイルの場所を指定します。このプロパティの詳細については、482 ページの「local.properties 内のプロパティ」を参照してください。

---

users.properties ファイルは単純なテキスト ファイルであり、編集してユーザ、グループ、およびロールを追加できます。このファイルの構文は JRun プロパティ ファイルの構文と同じです。プロパティ ファイルの構文の詳細については、『JRun セットアップ ガイド』を参照してください。

次の例では、users.properties ファイルに 5 人のユーザ、2 つのグループ、および 3 つのロールを定義しています。

```
# ユーザを定義します。
# パスワードは UNIX のパスワード暗号化を使用して暗号化します。
user.admin=adpexzg3FUZAk
user.ajones=kmBtOv9OZbRE6
user.bsmith=bs.e1isDZSIX.
user.csmith=cszOt89s3eWaU
user.user1=swJBBnSJUNwbQ

# グループを定義します。
group.jrundeveloper=ajones,bsmith,user1
group.all=*

# ユーザとグループにロールを割り当てます。
role.developer=jrundeveloper
role.manager=csmith
role.users=*
```

ユーザ名は次の形式となります。

```
user.userName=UnixCryptPassword
```

userName にはユーザ名を定義します。*UnixCryptPassword* はユーザのパスワードの暗号化フォームです。

この例にあるワイルドカード文字 (*) を使用すると、すべてのユーザにグループ `all` を割り当てたり、すべてのグループにロール `users` に割り当てることができます。

新規ユーザを Web サイトに追加する場合は、`users.properties` を更新してユーザとユーザの暗号化パスワードを追加し、ユーザを適切なグループとロールに追加する必要があります。アプリケーション内からプログラムによってユーザを追加したり、JRun のコマンド ライン ユーティリティを使用できます。

コマンド ライン ユーティリティを使用すると、ユーザと暗号化パスワードを `users.properties` ファイルに追加できます。ただし、この場合も `users.properties` を変更してロールとグループにユーザを追加する必要があります。

メモ
パスワードの長さに制限はありませんが、ユーティリティでは最初の 8 文字だけを暗号化します。

このユーティリティの起動コマンドは、Windows と UNIX システムでは異なります。

Windows

JRun の ホーム ディレクトリ`/lib/jrun.jar` ファイルがシステムの CLASSPATH 環境変数に含まれていることを確認します。次に、Windows コマンド プロンプトで、次のコマンドを使用してこのユーティリティを起動します。

```
java allaire.jrun.security.PropertyFileAuthentication options
```

UNIX

JRun には UNIX でこのユーティリティを起動するためのシェル スクリプトが用意されています。その構文は次のとおりです。

`jrunpasswd` *options*

Windows および UNIX のどちらの場合も、このコマンドに次のオプションを使用できます。

```
command [-verbose] -convert <users.properties file> <pass.properties
        file>
command [-verbose] -add <users.properties file> <username> <password>
command [-verbose] -edit <users.properties file> <username> <password>
command [-verbose] -remove <users.properties file> <username>
```

このコマンドで、プレーン テキストのユーザのパスワードを入力します。ユーティリティによってこれが暗号化され、users.properties ファイルに書き込まれます。JRun 2.3.3 pass.properties ファイルからユーザ名とパスワードを users.properties ファイルにコピーするには、コマンドの入力として pass.properties ファイルを使用します。

たとえば、次のコマンドによって、Bob というユーザおよび Bob の暗号化パスワードを UNIX システムの users.properties に追加します。

```
# jrunpasswd -add /usr/local/jrun/lib/users.properties Bob Bobpassword
User Bob added
```

JRun によるカスタム認証メカニズムの使用

前のセクションで説明した既定の JRun 認証メカニズムは Web サイトによっては適合しない場合があります。たとえば、LDAP データベースなどのユーザに関する情報を保存するためのメカニズムがすでに存在し、それをアプリケーション認証に統合することもあります。

この場合は、JRun インターフェイス allaire.jrun.security.AuthenticationInterface を実装することによって独自の認証メカニズムを定義できます。このインターフェイスには、JRun 認証メカニズムのメソッドが定義されています。このインターフェイスを実装すると、独自の認証メカニズムを定義できます。次の表は、このインターフェイスのメソッドの一覧です。

メソッド	目的
authenticate	パスワードなどの所定の証明を使用してユーザを認証します。
destroy	認証サービスを破棄します。
init	認証サービスを初期化します。
isPrincipalInRole	プリンシパル (ユーザ) に認証領域内でロールが与えられたかどうかを調べます。ユーザ名を取得するときに使用する Principal オブジェクトを返します。

このクラスの情報については、ディレクトリ {JRun root dir}/docs にある JavaDocs を参照してください。

認証インターフェイスを定義したら、JRun サーバーの local.properties にある authentication.*serviceName*.class プロパティを使用して、このインターフェイスを JRun に指定します。

次の例では、JRun サーバーが使用するカスタム認証メカニズムを指定しています。

```
# 認証
authentication.service=myauth
authentication.myauth.class=classFileName
```

JRun の外部でのアプリケーションの実行

JRun は Web アプリケーションの認証に関する最新の業界標準仕様 (Java サーブレット API バージョン 2.2 の仕様) に準拠しています。JRun で開発するアプリケーションは、この仕様に準拠しているほかのすべての Web アプリケーション環境に移植できます。

Web アプリケーションは、アプリケーションにアクセスするためにユーザに割り当てられている必要のあるロールを定義します。一方、アプリケーションを実行するアプリケーション サーバーでは、ロール情報を解釈して適用する必要があります。

したがって、アプリケーションによって参照されるロールを認識するように実行環境の認証メカニズムが設定されている限り、JRun で作成した Web アプリケーションは別の実行環境に移植できる必要があります。

認証プロパティ

このセクションでは、JRun 認証メカニズムを制御するプロパティについて説明します。

local.properties 内のプロパティ

`local.properties` ファイルには、JRun サーバーでアプリケーションを実行するときに認証を制御するために使用するプロパティがいくつか含まれています。各 JRun サーバーは独自の `local.properties` ファイルを持っています。

このセクションで説明するプロパティの形式は次のとおりです。

```
authentication.propertyName = propertyValue
authentication.<ServiceName>.propertyName = propertyValue
```

ここで、*ServiceName* は、authentication.service によって設定された認証サービスの論理名です。

authentication.service

認証サービス名を指定します。この名前は、プロパティ ファイルでクラス名やその他の設定プロパティ (`users.properties` の場所など) を調べるときのキーとして使用されます。

既定のサービス名は `propfile` です。

authentication.<ServiceName>.class

認証クラス名を指定します。既定では、このプロパティは `allaire.jrun.security.PropertyFileAuthentication` に設定されています。

ただし、`allaire.jrun.security.AuthenticationInterface` インターフェイスを実装することによって独自のカスタム認証メカニズムを作成できます。独自の認証メカニズムを定義する場合は、ここでカスタム クラスを指定します。

authentication.<ServiceName>.filename

認証情報のあるデータ ファイル名を指定します。このファイルにはユーザ、グループ、およびロールに関する情報が含まれています。

既定値は `{jrun.rootdir}/lib/users.properties` です。このパス名は、JRun インストール ディレクトリの `lib` ディレクトリに対応します。

users.properties 内のプロパティ

既定では、認証メカニズムは、`users.properties` ファイルに定義されているユーザ、グループ、およびロールに関する情報を使用します。このセクションでは、`users.properties` ファイルのプロパティについて説明します。

`users.properties` ファイルを編集するか、またはコマンド ライン ユーティリティを使用してユーザを作成できます。詳細については、478 ページの「既定の JRun 認証メカニズムの使用法」を参照してください。

user.userName

ユーザの名前とパスワードを指定します。このプロパティの形式は次のとおりです。

`user.`*`userName`*`=`*`UnixCryptPassword`*

userName にはユーザの名前を定義します。*UnixCryptPassword* はユーザのパスワードの暗号化フォームです。認証メカニズムは、Web リソースへのアクセス権を与える前に、この情報を使用してユーザを認証します。

次に、`users.properties` のユーザ定義の例を示します。

```
user.admin=adpexzg3FUZAk
user.ajones=kmBtOv9OZbRE6
user.bsmith=bs.e1isDZSIX.
user.csmith=csz0t89s3eWaU
user.user1=swJBBnSJUNwbQ
```

userName と *UnixCryptPassword* の両方でワイルドカード文字を使用できます。たとえば、次のステートメントを使用すると、ユーザ *developer* のすべてのパスワードが認証されます。

```
user.developer=*
```

注意

この例は、パスワードとして * のプレーン テキスト フォームを示しています。`users.properties` ファイルにパスワードとして * 記号を挿入しないでください。代わりに、* の UNIX の暗号化フォームを挿入する必要があります。

次の例では、ユーザはパスワード `jrun` のすべてのユーザが認証されます。

`user.*=`*`jrun`*

前の例と同様、`users.properties` ファイルにはパスワードとしてプレーン テキスト `jrun` を挿入しないでください。代わりに、`jrun` の UNIX の暗号化フォームを挿入する必要があります。

この例では、ユーザはどのようなパスワードでも認証されます。

```
user.*=*
```

group.groupName

特定のグループに割り当てられたユーザのリストを指定します。グループを使用すると、ユーザを 1 つにまとめてグループ全体の認証を制御できます。このプロパティの形式は次のとおりです。

`group.`*`groupName`*`=`*`memberList`*

groupName にはグループの名前を指定し、*memberList* にはグループ内のユーザのカンマ区切りリストを指定します。ワイルドカード文字を使用してすべてのユーザを 1 つのグループに追加できます。別のグループのメンバーになることはできないので注意してください。

次の例では 2 つのグループが作成されます。

```
group.jrundeveloper=ajones,bsmith,user1
group.all=*
```

role.roleName

ロール名と、そのロール内のユーザまたはグループのリストを指定します。このプロパティの形式は次のとおりです。

`role.`*`roleName`*`=`*`memberList`*

`roleName` にはロール名を指定し、*memberList* にはロール内のユーザおよびグループのカンマ区切りリストを指定します。ワイルドカード文字を使用してすべてのユーザおよびグループを 1 つのロールに追加できます。

次の例では 3 つのロールが作成されます。

```
role.developer=jrundeveloper
role.manager=bsmith
role.user=*
```

ユーザとグループは同じ名前にすることができます。その場合は、ロールのメンバリストに `user` または `group` を接頭辞として付けることができます。次の例では、`jrundeveloper` という名前のユーザとグループの両方がロールに追加されます。

```
role.developer=user.jrundeveloper, group.jrundeveloper
```

`user` または `group` 接頭辞を省略すると、JRun では、まずユーザのリストから名前を検索し、次にグループのリストを検索するので、ユーザ名とグループ名が重複します。このため、ユーザとグループが同じ名前の場合、`user` または `group` 接頭辞を省略すると、JRun は常に名前をユーザ名と見なします。

第 40 章

サーブレット メソッド パフォーマンスの監視

JRun を使用すると、サーブレットのメソッドの実行時間を記録することによって、アプリケーションのボトルネックを識別できます。

目次

- 概要 486
- メソッド タイミングの機能 486
- 例 488
- メソッド タイミングのプロパティ 489
- メッセージの形式 496

概要

アプリケーションのパフォーマンスを向上させるための実用的な戦略では、実行時間の大部分を占めるアプリ ケーション の領域を識別する 必要があり ます。最も 深刻な問題を最適化することによって、アプリケーションの総合的なパフォーマンスを大幅に改善できます。

JRun には、サーブレット 内の個々のメソッド のパフォーマンスを 測定するための メ ソッド タイミング機能が提供されています。作成したサーブレット メソッドだけでなく、サード パーティ ライブラリおよびヘルパ クラスのメソッドの実行時間も測定できます。

メソッド タイミング機能には、次のいくつかの重要な特性があります。

- **低オーバヘッド** この機能は、JRun ログ メカニズムと連動するため、アプリケーション自体から離れて、わずかな処理時間でタイミング情報を記録します。
- **ソース コードへの変更が不要** ソース コードを編集したり修正することなく、JRun プロパティ ファイルを使用してメソッド タイミングを制御できます。この機能では、ソース コードを変更する必要がないため、タイミング機能を追加したり削除するために、アプリケーションを再コンパイルする必要はありません。
- **柔軟性** アプリケーション内の 1 つ以上のサーブレット クラス、またはクラスの 1 つ以上のメソッドに対して、メソッド タイミングを有効または無効にできます。
- **サーブレット、JSP、およびサード パーティ コンポーネントとの連動** メソッド タイミングは、JRun を使用して開発したサーブレットだけでなく、アプリケーションに組み込むすべてのクラスおよびライブラリに対して機能します。

メソッド タイミングの機能

タイミング パラメータの値を指定することによって、メソッド タイミングがアプリケーションをどのように監視するかを設定します。タイミング パラメータの値は、JRun プロパティ ファイルで設定します。パラメータ値は、`global.properties` ファイルで定義します。

一般に、メソッド タイミングは、`global.properties` ファイルを編集するのではなく、アプリケーションを含んでいる JRun サーバーの `local.properties` ファイルを編集することによって、アプリケーションに実装します。サーバー上に 2 つ以上のアプリケーションが存在する場合、アプリケーションに関連付けられた `webapp.properties` ファイルでパラメータを定義できます。

メソッド タイミング機能には、**現在の**メソッド タイミングおよび**呼び出された**メソッド タイミングという 2 つのレベルのメソッド タイミングがあります。現在のメソッド タイミングは、通常、アプリケーションの一部であるメソッドの実行を監視するときに使用します。呼び出されたメソッド タイミングは、現在のメソッド内のメソッド呼び出しの実行を監視します。メソッド タイミングの使用法を理解すると、これらのレベルの違いがはっきりわかります。

メソッドにおけるパフォーマンスの問題を検出するための一般的な戦略では、メソッド タイミングに別の設定を使用して、アプリケーションを 2 度実行します。

- 最初の実行は、現在のメソッド タイミングを使用して、アプリケーションの一部であるすべてのメソッドの実行統計を生成します。このパスの結果には、実行時間の大部分を占めるメソッドを分離する目的があります。
- 2 度目は、呼び出されたメソッド タイミングを使用して、最初のパスによって識別されたメソッド呼び出しの実行統計を生成します。このパスの結果を使用すると、ボトルネックの発生場所を判断できます。

たとえば、アプリケーションに、メソッドを定義するサーブレットが含まれている場合、そのメソッド内のコードは別のメソッドを呼び出します。

```
method abc()
{
...
  ...xyz()...
...
}
```

メソッド abc の現在のメソッド タイミングはメソッドの実行所要時間を生成します。この実行時間には、xyz の所要時間も含まれます。メソッド abc に対して呼び出されたメソッド タイミングはメソッド xyz の実行所要時間を生成します。この結果は、実行時間が主にメソッド abc 内にあるか、またはメソッド xyz にあるかを示します。

次のセクションの例では、両方のタイミングを使用してアプリケーションに関する役立つ情報を得るための方法を説明しています。

JRun ロガーは、パフォーマンス出力を処理します。既定により、JRun はログ ファイルに出力を書き込みますが、クライアントまたは別のファイルに出力を書き込むように指示することもできます。

例

このセクションの例では、メソッド タイミングをアプリケーションに実装して役立つ情報を得るための方法を説明します。

タイミングの有効化と既定のプロパティ値の受け入れ

次のセクション (489 ページの「メソッド タイミングのプロパティ」) で説明するように、global.properties ファイルは、メソッド タイミングを設定する多くのプロパティの既定値を定義します。これらの値は、local.properties ファイルのプロパティを定義し直すことによって変更できます。

この例では、メソッド タイミングを有効にし、global.properties ファイルで定義されたプロパティの既定値を受け入れたときに、メソッド タイミングがサンプル アプリケーションでどのように機能するかを説明しています。

メソッド タイミングを計測するには、次のステートメントを追加することによって、サンプル アプリケーションが存在するサーバー上の local.properties ファイルを編集します。

```
## メソッド タイミングのセクション
timing.enabled=true
```

結果は、次の一覧のとおりです。この一覧を読む以外に、global.properties ファイルに定義されているメソッド タイミングのプロパティを参照することもできます。これらのプロパティについては、490 ページの「global.properties ファイルで定義されたプロパティ」で説明します。

- 唯一の例外として、java.*、javax.*、sun.*、および org.omg.CORBA.* のライブラリにあるメソッドへの呼び出しは、メソッド タイミングから除外されます。
- java.sql.* ライブラリにあるメソッドの呼び出しはメソッド タイミングに含まれます。
- 496 ページの「メッセージの形式」で説明するように、タイミング メッセージは既定の形式に従っています。
- HttpServlet の直接サブクラスで定義されたすべてのメソッドは、タイミングが計測されます。HttpServlet の直接サブクラスのサブクラスでは、タイミングを計測できません。
- javax.servlet.http のメソッドへの呼び出しを含む、HttpServlet の直接サブクラスで定義されたメソッド内で呼び出されたすべてのメソッドに対してタイミングが計測されます。
- アプリケーションによって、SnoopServlet クラスが定義されると、そのクラスのすべてのメソッドに対してタイミングが計測されます。また、除外されたライブラリへの呼び出しを除き、javax.servlet.http.* のメソッドへの呼び出しを含む、これらのメソッド内のすべてのメソッド呼び出しに対してタイミングが計測されます。

指定されたクラスおよびメソッドのタイミングの有効化

前述のとおり、アプリケーションにおけるパフォーマンスのボトルネックを見つける適切な戦略は、アプリ ケーションを 2 度実行すること です。まず、現在のメソッド タイミングを使用して、アプリ ケーションによって使用されるすべてのメソッド の統計を得ます。次に、実行時間の大部分を占めるメソッドの一覧を使用して、もう 1 度アプリケーションを実行します。2 度目は、最初のパスの実行時にメソッド タイミングによって呼び出されたメソッドの統計を提供するプロパティを定義します。この 2 番目のパスは、呼び出されたメソッド タイミングを使用します。

これらのパスの実行時に取得された統計を使用して、どちらのメソッドが実行時間の大部分を使用するかを判断できます。

メソッド タイミングのプロパティ

次の 3 種類のプロパティを使用して、メソッドタイミングを制御します。

- タイミングを有効化または無効化する一般的なプロパティ。どのメソッドを監視の対象または非対象にするかを識別します。
- ログ プロパティは、パフォーマンス メッセージを処理し、メッセージの形式を制御するロガーを指定します。
- クラス プロパティおよびメソッドのプロパティは、タイミングが実行されるクラスおよびメソッドを制御します。

次のセクションでは、これらのプロパティについて説明します。

global.properties ファイルで定義されたプロパティ

global.properties ファイルで定義されたプロパティは、JRun インストールのすべてのサーバー上で実行するすべてのアプリケーションに適用されます。

global.properties ファイルは、JRun とともに提供されているため変更できません。メソッド タイミングに適用される global.properties ファイルのセクションには、次のステートメントが含まれています。

```
#########################################################################
## メソッド タイミング
#########################################################################

# メソッド タイミングを有効にします。
timing.enabled=false

timing.excludecalls=java.*,javax.*,sun.*,org.omg.CORBA.*
timing.includecalls=java.sql.*

# メッセージを処理するために使用されるロガーを定義します。
# 既定では、メッセージはサーブレットを実行する JRun サーバーの
# ログ ファイルに書き込まれます。
# これらのプロパティはメッセージの形式を制御します。

# ロガー名を定義します。
timing.logging.class=simplelogger

# システム ロガーにタイミング メッセージを転送する
# 単純なロガー用のプロパティを指定します。
timing.simplelogger.class=allaire.jrun.methodTimer.SimpleLogger
timing.simplelogger.level=info
timing.simplelogger.entermethod=ENTER
timing.simplelogger.exitmethod=EXIT
timing.simplelogger.beforemethodcall=CALLENTER
timing.simplelogger.aftermethodcall=CALLEXIT
timing.simplelogger.delimiter=,

# メソッド タイミング メッセージをスレッドのローカル記憶領域に転送する
# スレッド ロガーのプロパティを指定します。
# ThreadLogger を
# allaire.jrun.servlets.JRunStats サーブレットと併用して出力を表示します。
timing.threadlogger.class=allaire.jrun.methodTimer.ThreadLogger
# JRunStats サーブレットの設定 (threadlogger と
# 併用されます)
timing.jrunstats.includeSource=true
timing.jrunstats.sourcepath=/WEB-INF/classes;/WEB-INF/jsp
timing.jrunstats.includeZeroTimes=false

# 計測するクラスの名前を指定します。
timing.classes=HttpServlet,SnoopServlet,jsp,jst

# HttpServlet およびすべての直接サブクラスに関する計測設定を指定します。
```

```
timing.HttpServlet.class=javax.servlet.http.HttpServlet
timing.HttpServlet.methods=*
timing.HttpServlet.calls=*,javax.servlet.http.*
timing.HttpServlet.doGet.calls=*
timing.HttpServlet.subclasses=true

# SnoopServlet に関する計測設定を指定します。
timing.SnoopServlet.class=SnoopServlet
timing.SnoopServlet.methods=*
timing.SnoopServlet.calls=*,javax.servlet.http.*

# HttpJSPServlet およびすべての直接サブクラスに関する計測設定を指定します。
timing.jsp.class=allaire.jrun.jsp.HttpJSPServlet
timing.jsp.subclasses=true
timing.jsp.methods=*
timing.jsp.calls=*,javax.servlet.http.*,javax.servlet.*

# JSTTag およびすべての直接サブクラスに関する計測設定を指定します。
timing.jst.class=allaire.jrun.jsp.JSTTag
timing.jst.subclasses=true
timing.jst.methods=*
timing.jst.calls=*
```

一般プロパティ

JRun メソッド タイミングを有効化または無効化するプロパティを生成し、タイミングの対象または非対象となるメソッドを含むライブラリを指定します。次の表では、一般的なプロパティについて説明しています。timing.enabled プロパティを除き、次に説明するすべてのプロパティの例は、global.properties ファイルに含まれるステートメントです。

プロパティ	説明	例
timing.enabled	メソッド タイミングを有効 (true) または無効 (false) に指定します。 global.properties ファイルは、このプロパティの値を false と定義します。	次のステートメントは、メソッド タイミングを有効にします。 timing.enabled=true
timing.excludecalls	タイミングから除外するメソッドのカンマ区切りリストを指定します。アスタリスク (*) ワイルドカード文字を使用して、ライブラリ内のすべてのメソッドへの呼び出しを除外します。	次のステートメントは、指定されたライブラリ内のすべてのメソッドをタイミングから除外します。 timing.excludecalls=java.*,javax.*,sun.*,org.omg.CORBA.*
timing.includecalls	タイミングを計測するメソッドのカンマ区切りリストを指定します。アスタリスク (*) ワイルドカード文字を使用して、ライブラリ内のすべてのメソッドへの呼び出しを含めます。このプロパティは、timing.excludecalls 内の競合する設定を書き換えます。 excludecalls プロパティと includecalls プロパティの要素で始まる完全修飾クラス名 (パッケージ名を含む) は、対象または非対象、あるいはその両方になります。これらのプロパティは、ライブラリ名または特定のクラス名です。	前述の timing.excludecalls ステートメントで使用された場合、次のステートメントは、java.sql ライブラリ内のメソッドへの呼び出しを除く、java ライブラリ内のすべてのメソッドへの呼び出しを除外します。 timing.includecalls=java.sql.*

クラスおよびメソッド プロパティ

このセクションでは、JRun によってタイミングが実行されるクラスおよびメソッドを制御するために使用するプロパティについて説明します。

前のセクションで説明したように、通常、timing.includecalls プロパティと timing.excludecalls プロパティを使用して、タイミングの対象または非対象となるメソッドを含むライブラリを指示します。タイミングを計測するメソッドのクラスおよびサブクラスを指示するには、このセクションで説明するプロパティを使用します。

global.properties ファイルには、次のメソッドへの呼び出しのメソッド タイミングを計測するステートメントが含まれています。

- HttpServlet クラスのすべてのメソッド
- HttpServlet クラスの直接サブクラスにあるすべてのメソッド
- timing.excludecalls プロパティで命名されることによって除外されたメソッドを除く、これらのメソッドによって呼び出されたすべてのメソッド

次の表では、クラスおよびメソッドのプロパティについて説明しています。

プロパティ	説明	例
timing.classes	タイミングを計測するクラスのカンマ区切りリストを指定します。 追加のプロパティは、これらのクラスに関する必要な詳細を提供します。	次のステートメントは、HttpServlet、SnoopServlet、jsp、および jst クラスに対してタイミングが計測されるように指示します。 timing.classes=HttpServlet,SnoopServlet,jsp,jst
timing.<ClassName>.class	タイミングを計測するクラスの完全なパッケージ クラス名を指定します。	次のステートメントには、HttpServlet クラスの完全なパッケージ クラス名が記述されています。 timing.HttpServlet.class=javax.servlet.http.HttpServlet
timing.<ClassName>.methods	タイミングを計測する、指定されたクラスのメソッドのカンマ区切りリストを指定します。 アスタリスク (*) ワイルドカード文字を使用して、複数のメソッドを指定できます。	次のステートメントは、HttpServlet クラスにあるすべてのメソッドに対してタイミングが計測されるように指示します。 timing.HttpServlet.methods=*
timing.<ClassName>.calls	タイミングを計測する、現在のメソッドによって呼び出されたメソッドのカンマ区切りリストを指定します。呼び出されたメソッドは、timing.<ClassName>.methods によって指定されたメソッド内のメソッドから呼び出されます。 アスタリスク (*) ワイルドカード文字を使用して、複数のメソッドを指定できます。	次のステートメントは、javax.servlet.http への呼び出しに対してタイミングが計測されるように指示します。 最初の引数として指定されたアスタリスク (*) は、前述のステートメントで対象または非対象とされたすべてのクラスおよびライブラリがこのステートメントに適用可能なことを示します。javax ライブラリは、timing.excludecalls プロパティでタイミングから除外されたので、メソッドへの呼び出しに含むクラスを指定する必要があります。 timing.HttpServlet.calls=*,javax.servlet.http.*

プロパティ	説明	例
timing.<ClassName>.<MethodName>.calls	timing.<ClassName>.methods によって指定された各メソッドに対して、タイミングを計測する、呼び出されたメソッドのカンマ区切りリストを指定できます。 アスタリスク (*) ワイルドカード文字を使用して、複数のメソッドを指定できます。	次のステートメントは、doGet メソッド内で呼び出された getConnection メソッドのタイミングを計測します。 timing.HttpServlet.doGet.calls=getConnection
timing.<ClassName>.subclasses	<ClassName> の直接サブクラス、または <ClassName> インターフェイス ディレクトリを実装するクラスに対してタイミングを計測するかどうかを指定します。true または false の値を指定します。	次のステートメントによって、HttpServlet クラスのすべての直接サブクラスに対してメソッド タイミングが実行されます。 timing.HttpServlet.subclasses=true

ログ プロパティ

JRun メソッド タイミングは、生成するメッセージの処理を JRun ログ メカニズムに依存します。したがって、メソッド タイミングを設定するための 1 つのステップとして、ロガーに関連するプロパティを設定する必要があります。ロギングの詳細については、441 ページの第 38 章「ログ」を参照してください。

既定により、メソッド タイミングは、メッセージを文字列形式にする simplelogger を使用し、そのメッセージを JRun ロガーに送信します。

既定のメッセージ形式は、現在のメソッドおよび呼び出されたメソッドに制御がいつ転送されたり戻されるかを示します。これらのメッセージを区別するキーワードは、このセクションで説明するプロパティを使用して変更できます。

このセクションでは、メソッド タイミング メッセージに関連付けられたロガーを制御するために使用するプロパティについて説明します。次の表では、ログ プロパティについて説明しています。このセクションのすべてのプロパティの例は、global.properties ファイルに含まれるステートメントの例です。

プロパティ	説明	例
timing.logging.class	タイミングの出力を取得するロガーの名前を指定します。 追加のプロパティは、このロガーに関する必要な詳細を提供します。	次のステートメントは、ロガー名をsimpleloggerと定義します。 timing.logging.class=simplelogger
timing.<LoggerName>.class	タイミング メッセージの形式を定義するロガー クラスを指定し、そのメッセージをログ システムに転送します。通常、allaire.jrun.methodTimer.SimpleLogger クラスを使用します。	次のステートメントは、simplelogger のロガー クラスを指定します。 timing.simplelogger.class=allaire.jrun.methodTimer.SimpleLogger
timing.<LoggerName>.level	ログへのログ イベント タイプを指定します。このタイプのメッセージが生成されると、それらのメッセージはロガーに送信されます。指定可能な値は次のとおりです。debug、error、info (既定)、およびwarning。	次のステートメントによって、info レベルのメッセージが生成されます。 timing.simplelogger.level=info
timing.<LoggerName>.entermethod	現在のクラスの開始時にタイミング メッセージに含めるテキストを指定します。	次のステートメントは、現在のメソッドの開始時にテキスト ENTER を表示します。 timing.simplelogger.entermethod=ENTER
timing.<LoggerName>.exitmethod	現在のメソッドの終了時に生成されるタイミング メッセージに含めるテキストを指定します。	次のステートメントは、現在のメソッドの終了時にテキスト EXIT を表示します。 timing.simplelogger.exitmethod=EXIT
timing.<LoggerName>.beforemethodcall	呼び出されたメソッドの終了時に生成されるタイミングメッセージに含めるテキストを指定します。	次のステートメントは、呼び出されたメソッドの開始時にテキスト CALLENTER を表示します。 timing.simplelogger.beforemethodcall=CALLENTER
timing.<LoggerName>.aftermethodcall	呼び出されたメソッドの終了時に生成されるタイミング メッセージに含めるテキストを指定します。	次のステートメントは、呼び出されたメソッドの終了時にテキスト CALLEXIT を表示します。 timing.simplelogger.aftermethodcall=CALLEXIT
timing.<LoggerName>.delimiter	タイミング メッセージのコンポーネント間に使用する区切り文字を指定します。simplelogger の既定の区切り文字はカンマです。それ以外の場合はスペースです。	次のステートメントは、simplelogger の既定の区切り文字をカンマと定義します。 timing.simplelogger.delimiter=,

メッセージの形式

タイミングを計測するメソッドが呼び出されるたびに、JRun は出力先に 2 つのメッセージを送信します。最初のメッセージには、識別情報、そのメソッド タイミングの開始を示すテキスト、およびメソッドの開始時間が記載されます。2 番目のメッセージには、同じ識別情報とメソッド タイミングの終了を示すテキストのほかに、メソッドのミリ秒単位の終了時間および経過時間が記載されます。

現在のメソッド タイミング メッセージの形式

現在のメソッド呼び出しに関するメッセージの形式には、次のコンポーネントが含まれます。

```
currentTimeMillis loglevel type,className,hashCode,methodName,
methodType,elapsed
```

currentTimeMillis	メッセージの日付と時刻。時刻はミリ秒まで記録されます。
loglevel	タイミング メッセージのレベル。info、warning、error、または debug。
type	メソッド の開始または終了を表す文字列
className	メソッド のクラス名
hashCode	メソッド のハッシュ コード 。 ハッシュ コード は、マルチスレッド 環境で役に立つ、固有オブジェクト 参照です。2 つのスレッド が同じ サーブレット で同時に機能し ている 場合、メソッド タイミング メッセージはインターリーブされます。このため、ハッシュ コード を使用すると、メッセージを区別できます。
methodName	メソッド の名前
methodType	void を 示す V など、戻り 値のタイプを 含むメソッド のシグネチャ
elapsed	メソッド 呼び出し のミリ 秒単位の経過時間 (メソッド)

doGet メソッドを開始したり終了することによって生成されるタイミング メッセージの例を次に示します。

```
11/15 03:41:50 info ENTER,HelloWorld,-100030151,doGet,
(Ljavax/servlet/http/HttpServletRequest;Ljavax/servlet/http
/HttpServletResponse;)V

11/15 03:41:50 info EXIT,HelloWorld,-100030151,doGet,
(Ljavax/servlet/http/HttpServletRequest;Ljavax/servlet/http/
HttpServletResponse;)V,3
```

EXIT メッセージの最後のコンポーネントは、メソッドのミリ秒単位の継続時間です。

呼び出されたメソッド タイミング メッセージの形式

呼び出されたメソッド呼び出しメッセージの形式には、次のコンポーネントが含まれます。このメッセージの形式には、現在のメソッド 呼び出しメッセージのすべてのコンポーネントと次の追加コンポーネントが含まれます。追加コンポーネントは太字で示されています。

currentTime loglevel type,className hashCode,methodName,
methodType,**callClassName,callMethodName,callMethodType,line**,elapsed

コンポーネント	説明
callClassName	タイミングを計測するメソッドのクラス名
callMethodName	現在のメソッドによって呼び出された、呼び出されたメソッドの名前 (methodName)
callMethodType	呼び出されたメソッドのシグネチャ (callMethodName)
line	呼び出されたメソッド呼び出し (callMethodName) を含む、メソッドのステート メントの行番号 (methodName)

JSP で呼び出されたメソッドを開始したり終了することによって生成されたタイミング メッセージの例を次に示します。

- クラス名は、jrun__SampleWithStats2ejsp14 です (この名前は、JRun によりコンパイル済みのファイルに割り当てられます)。
- メソッド名は _jspService です。
- 呼び出されたメソッドのクラス名は java.sql.DriverManager です。
- 呼び出されたメソッド名は getConnection です。
- 呼び出されたメソッドへの呼び出しの行番号は 58 です。
- 経過時間は 828 ミリ秒です。

```
03/16 13:32:12 info (JRun) CALLENTER,jrun__SampleWithStats2ejsp14,
73057990,_jspService,Ljavax/servlet/http/HttpServletRequest;Ljavax/
servlet/http/HttpServletResponse;)V,java.sql.DriverManager,
getConnection,Ljava/lang/String;Ljava/lang/String;Ljava/lang/
String;)Ljava/sql/Connection;,58

03/16 13:32:13 info (JRun) CALLEXIT,jrun__SampleWithStats2ejsp14,
73057990,_jspService,Ljavax/servlet/http/HttpServletRequest;Ljavax/
servlet/http/HttpServletResponse;)V,java.sql.DriverManager,
getConnection,Ljava/lang/String;Ljava/lang/String;Ljava/lang/
String;)Ljava/sql/Connection;,58,828
```

タイミング メッセージの書き込み

既定では、JRun はアプリケーションを実行する JRun サーバーのログ ファイルにタイミング メッセージを書き込みます。これらのメッセージをクライアントまたは別のファイルに書き込むように、JRun に指示することもできます。

クライアント へのタイミング メッセージの書き込み

次の手順に従って、メソッドをクライアントに転送します。

1 `local.properties` ファイルでこのプロパティを設定して、計測を有効にできます。

```
timing.enabled=true
```

このオプションを有効にしない場合、クライアントに返される情報は、サーブレットの合計実行時間のみです。

2 `local.properties` ファイルで次のプロパティを設定します。

```
timing.logging.class=threadlogger
```

3 次のいずれかの手順に従い、サーブレット `JRunStats` を使用して、要求に関する統計情報を表示します。

- 明示的に、または MIME タイプのいずれかを通じて、サーブレット チェーンで使用します。たとえば、次の URL は、`JRunStats` サーブレットと `SnoopServlet` をつなぎます。

```
http://localhost/servlet/SnoopServlet,JRunStats
```

- Java サーブレットの要求ディスパッチャを使用します。

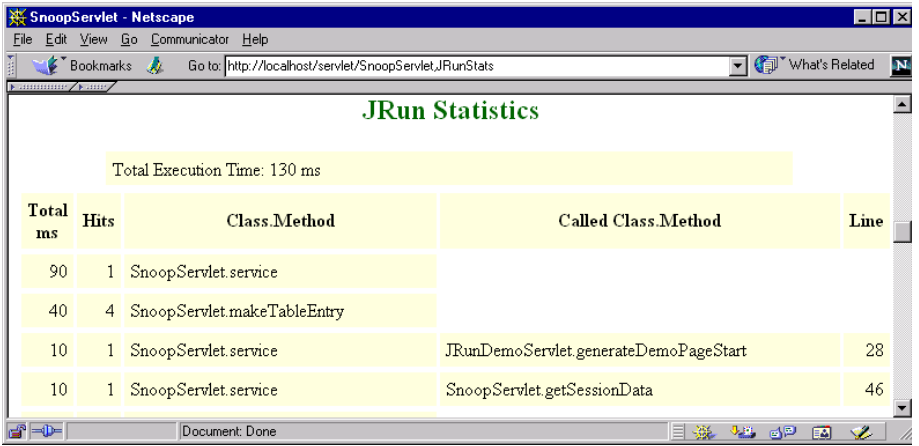
```
RequestDispatcher rd = req.getRequestDispatcher("/servlet/
  JRunStats");
rd.include(req,resp);
```

メソッド タイミングを有効にすると、クライアントに送信される出力情報には、サーブレットの合計実行時間のほかに次の情報が含まれます。

出力	説明
`Total ms`	メソッド呼び出しのミリ秒単位の継続時間
`Hits`	サーブレット内でメソッドが呼び出された回数
`Class.Method`	呼び出しを作成中のサーブレット クラスまたはメソッドの名前
`Called Class.Method`	呼び出されたメソッドの名前。統計情報がメソッド `Class.Method` に関する場合は、空になります。
`Line`	呼び出しが作成された `Class.Method` のソース行番号で、統計情報がメソッド `Class.Method` に関する場合は、空になります。

次の図は、JRun に付属するサーブレット `SnoopServlet` に関するタイミングの出力例を示します。この例で使用されている URL は、次のとおりです。

http://localhost/servlet/SnoopServlet,JRunStats



この例では、情報の最初の行はメソッド呼び出し `SnoopServlet.service` に関する内容です。このタイミング情報は、`SnoopServlet.service` によって呼び出されるメソッドに関するものではなく、`SnoopServlet.service` によって呼び出されるメソッドに関する内容であるため、`Called Class.Method` および `Line` フィールドは空になります。

情報の 3 行目は、`JRunDemoServlet.generateDemoPageStart` メソッドに関する内容です。このメソッドは、`SnoopServlet.service` によって呼び出されます。

ファイルへのタイミング メッセージの書き込み

タイミング メッセージを別のファイルに書き込むように JRun を設定することもできます。

この例では、HelloWorld クラス (現在のメソッド) の doGet メソッドと doPost メソッドのメソッド タイミングを計測し、javax.servlet.http.* のメソッド (呼び出されたメソッド) への呼び出し時間を計測します。プロパティは、fileInstrumentWriter というファイル ライターを定義し、timing.log という名前の付いたファイルにタイミング メッセージを書き込みます。

```
# 計測を有効にします。
timing.enabled=true

# すべてのサーブレットから、計測の対象および対象外となる
# メソッドを指定します。
timing.excludecalls=java.*,javax.*,sun.*,org.omg.CORBA.*
timing.includecalls=java.sql.*

# ログ グループを設定します。
logging.groups=instrumentLog

# instrumentLog ログ グループおよび名前付きロガーを設定します。
logging.groups.instrumentLog.timingNamedLogger=info
logging.groups.instrumentLog.listeners=instLogThreadLog

logging.instLogThreadLog.class=allaire.jrun.logging.ThreadedLogger
logging.instLogThreadLog.listeners=fileInstrumentWriter

# ファイル ライターを定義します。このファイル ライターにより、すべてのイベントが
# 1 つのファイルに書き込まれます。
logging.fileInstrumentWriter.class = allaire.jrun.logging.FileLogWriter
logging.fileInstrumentWriter.filename = {jrun.roodir}/logs/timing.log
logging.fileInstrumentWriter.rotationsize = 100000
logging.fileInstrumentWriter.rotationfiles = 3

# ロガー名を定義します。
timing.logging.class=simplelogger

# ロガー用のクラス ファイルを指定します。
timing.simplelogger.class=allaire.jrun.methodTimer.SimpleLogger
timing.simplelogger.level=info
timing.simplelogger.entermethod=ENTER
timing.simplelogger.exitmethod=EXIT
timing.simplelogger.beforemethodcall=CALLENTER
timing.simplelogger.aftermethodcall=CALLEXIT
timing.simplelogger.delimiter=,
timing.simplelogger.loggername=timingNamedLogger
```

```
# 計測するサーブレットを指定して、タイミングを計測する現在のレベルおよび
# 呼び出されたレベルのメソッドを定義します。この例では、doGet メソッド、doPost メソッド、
# および doGet の呼び出されたすべてのメソッド呼び出しを計測します。
timing.classes=hWorld
timing.hWorld.class=HelloWorld
timing.hWorld.methods=doGet,doPost
timing.hWorld.calls=javax.servlet.http.*
timing.hWorld.doGet.calls=*
```

JSP のタイミングの計測

この例では、JSP ページの実行時間がクライアントに返されます。JRun ログ ファイルに情報を書き込むために、この例を変更することもできます。

次の手順に従って、JSP タイミング情報をクライアントに返します。

1 `local.properties` ファイルで `timing.enabled` プロパティを次のように設定します。

```
timing.enabled=true
```

このオプションを有効にしない場合、サーブレットの合計実行時間のみがクライアントに返されます。

2 次のプロパティを設定します。

```
timing.logging.class=threadlogger
```

3 JSP メソッド タイミング属性を設定します。

```
timing.jsp.class=allaire.jrun.jsp.HttpJSPServlet
timing.jsp.subclasses=true
timing.jsp.methods=*
timing.jsp.calls=*,javax.servlet.*
```

これらの設定によって、`HttpJSPServlet` を直接スーパークラスとするすべてのクラスに対してタイミングが計測されるように指定します。JRun では、すべてのメソッドのパフォーマンスと、既定で `timing.excludecalls` プロパティにより除外されている `javax.servlet.*` 呼び出しを含むメソッド内の呼び出しがすべて計測されます。

4 JRun サーバーを再起動します。

5 サーブレット `JRunStats` を使用して、498 ページの「クライアントへのタイミング メッセージの書き込み」に示しているように、要求に関する統計情報を表示します。

第 41 章

デバッグとエラーメッセージング

アプリケーション開発中の一般的なタスクには、デバッグがあります。この章では、JRun アプリケーションのための最も一般的なデバッグ テクニックについていくつか説明します。

また、JRun には、書き換え可能な組み込みエラー メッセージング メカニズムがあります。この書き換えによって、JRun エラー メッセージを独自のエラー メッセージに置き換えることができます。この章では、JRun によって生成されるエラー メッセージを独自のエラー メッセージに置き換える方法についても説明します。

この章の最後では、J++ や Visual Café などのサードパーティ製 IDE を使用して、JRun アプリケーションを開発する方法について説明します。

目次

- デバッグ 504
- カスタム エラー メッセージング 515
- JRun とサードパーティ製 IDE の併用 517

デバッグ

このセクションでは、JRun を使用して開発した Web アプリケーションをデバッグする際に使用できる、一般的なタスクおよびテクニックについていくつか説明します。次のようなセクションがあります。

- 「JRun によるデバッガの起動」504 ページ
- 「スタック トレース」504 ページ
- 「コア ダンプの処理 (UNIX システムのみ)」507 ページ
- 「メモリ不足エラーの処理」508 ページ
- 「クライアント / サーバー間の通信の監視」509 ページ
- 「追加デバッグ リンク」514 ページ

JRun によるデバッガの起動

JRun アプリケーションをデバッグするための最も基本的な方法の 1 つは、Java デバッガ コマンド `jdb` の制御のもとで JRun を起動する方法です。`jdb` のもとで JRun を起動すると、ブレーク ポイントおよびシングル ステップの設定、スレッドに関する情報の取得、メモリ使用の検査、およびその他のさまざまなデバッグ操作を行うことができます。

`jdb` コマンドを使用して JRun を起動するには、使用するシステムの CLASSPATH 環境変数に、次のファイルを追加する必要があります。

- *JRun のホーム ディレクトリ*`¥lib` 内のすべての JAR ファイル
- *JRun のホーム ディレクトリ*`¥lib¥ext` 内のすべての JAR ファイル

Windows では、次のコマンドを使用して、`jdb` のもとで default JRun サーバーを起動します。

```
jdb JRun c:¥progra~1¥Allaire¥JRun¥servers¥default
```

`jdb` ユーティリティが Windows のディレクトリ名「`program files`」に含まれているスペースを解釈できないこともあるため、省略形の `progra~1` が必要です。UNIX マシンでは、起動する JRun サーバーへのパスを使用します。

スタック トレース

アプリケーションのデバッグに役立つ方法として、スタック トレースを取得する方法があります。Java スタック トレースには、JVM のスレッドおよび監視に関する情報が含まれています。実際に、お客様の問題を解決する方法の一部として、Allaire カスタマ サポートでは、スタック トレースを生成して、電子メールで Allaire に送信していただくようにお客様にお願いしています。

通常、スタック トレースは、デッドロック状態の原因を特定するために使用します。同時に発生する多数の要求を処理するため、JRun はマルチスレッド モデルを使用します。マルチスレッド環境では、スレッドがいつでもリソースを利用できるように、特定のリソースへのアクセスはロックによって制御されます。あるスレッドが 2 番目のスレッドからリソースを取得するために待機しなければならず、2 番目のスレッドも最初のスレッドからリソースを取得するために待機するといった状況が発生します。この状況は、**デッドロック**と呼ばれます。

次の例は、デッドロックを含むスタック トレースからの抜粋です。

```
t@37 waiting to lock object@0xdbdf15c8:"oracle/jdbc/driver/
        OracleCallableStatement" which is locked by t@529
t@529 waiting to lock object@0xdc030470:"oracle/jdbc/driver/
        OracleConnection" which is locked by t@37
```

このように、スレッド 37 および 529 は、お互いに相手のスレッドがリソースを解放するのを待っています。両方のスレッド が待機し ているため、どちら のスレッドも相手のスレッドが要求するリソースを解放することはできません。この場合、両方のスレッドは永久に待ち続けます。

次のセクションでは、JRun でサポート されているオペレーティング システム用のスタック トレースを取得する方法について説明します。

Allaire では、Java スタック トレースの内容の解釈に関する文書を Knowledge Base で公開しています。知識ベースの記事番号 12406 は、次の URL で参照できます。

`http://allaire.com/Support/KnowledgeBase/SearchForm.cfm`

Windows のスタック トレース

Sun または IBM の JVM を使用した Windows システムで稼動する JRun からスタック トレースを取得するには、次の手順を実行します。

1 JRun を実行するための Java 実行可能ファイルとして、`javaw.exe` ではなく、`java.exe` を使用していることを確認します。`java.exe` では JRun サーバーの起動時に DOS ウィンドウが開きますが、`javaw.exe` では Java アプリケーションの実行時に DOS ウィンドウが開きません。

   JMC で [***サーバー名***] > [Java **の設定**] > [Java **実行ファイル**] プロパティを使用して、`java.exe` を設定します。***サーバー名***には、JRun サーバーの名前を指定します。

2 JRun サーバーを起動します。

   空白の DOS ウィンドウが開きます。

3 DOS ウィンドウをアクティブにし、`Ctrl-Break` キーを押して、スタック トレースが `{JRun home dir}/logs/servername-err.log` ファイルに書き込まれるようにします。

メモ
このファイルの名前は、`java.System.err` プロパティで定義されます。ファイル名を変更するには、JRun サーバーの `local.properties` ファイルでこのプロパティを変更します。このプロパティの詳細については、第 41 章 を参照してください。

または、DOS ウィンドウの [閉じる] ボタン (DOS ウィンドウの右上隅にある [X] アイコン) をクリックして、スタック トレースを開始することもできます。DOS ウィンドウは閉じませんが、ポップアップ ウィンドウに次のメッセージが表示されます。

この Windows アプリケーションは [アプリケーション終了] に応答できません。

[アプリケーション終了] を選択すると JRun は停止し、[キャンセル] を選択すると JRun は実行を続けます。

4 メモ帳、ワードパッド、またはほかのテキスト エディタを起動して、`{JRun home dir}/logs/servername-err.log` ファイルに含まれるスタック トレース情報を読み取ります。

UNIX/Solaris スタック トレース

UNIX/Solaris の場合、既定では、スタック トレースを保持する `JRun/logs/{server}-err.log` という名前のファイルが定義されます。`{server}` は、JRun サーバーの名前です。

次の手順に従って、スタック トレースを取得します。

1 `top` コマンドを使用して `java` プロセスのプロセス ID 番号を特定するか、`ps -ef` コマンド の出力時に `grep` コマンド を使用します。次の例では、`grep` および `ps` コマンドを使用しています。

```
ps -ef | grep java
```

2 `kill -QUIT` または `kill -3` コマンドを使用して、quit 信号を `java` プロセス ID に送信します。次に例を示します。

```
kill -3 jrunjavaID
```

一部の UNIX/Solaris プラットフォームでは、使用する JDK によって、Java プロセスを開始したモニタまたはターミナル ウィンドウに Java スタック トレースが直接送信されない場合があります。スタック トレースの出力を別の出力先に転送する機会はありません。唯一の方法として、トレースをモニタまたはウィンドウにダンプして、コピーしたトレースを vi や Emacs などのエディタを実行している別のターミナル ウィンドウに貼り付けます。

Linux

Sun/Blackdown JVM を使用した Linux システムに関するスタック トレースを取得するには、506 ページの「UNIX/Solaris スタック トレース」に記載されている手順を実行してください。

IBM JVM は、quit 信号が送信された JRun サーバーのルート ディレクトリ に `javacore.txt` ファイルを作成して、`kill -QUIT` または `kill -3` に応答します。たとえば、default JRun サーバーの場合、このディレクトリは、既定では `/opt/JRun/servers/default` になります。

次の手順に従って、スタック トレースを取得します。

1 `top` コマンドを使用して `java` プロセスのプロセス ID 番号を特定するか、`ps -ef` コマンド の出力時に `grep` コマンド を使用します。次の例では、`grep` および `ps` コマンドを使用しています。

```
ps -ef | grep java
```

2 `kill -QUIT` または `kill -3` コマンドを使用して、quit 信号を `java` プロセス ID に送信します。次に例を示します。

```
kill -3 jrunjavaID
```

コア ダンプの処理 (UNIX システムのみ)

アプリケーションがコア ダンプを引き起こすと、オペレーティング システムはアプリケーションを起動したディレクトリにコア ファイルを書き込みます。

メモ
コア ダンプは、UNIX システムでのみ発生し、Windows では発生しません。

JRun は Java で記述されているため、JRun でエラーが発生した場合は、常に Java 例外が返され、JRun 自体がコア ダンプを引き起こすことはありません。JRun に関係するコア ダンプは通常、次の理由によって JVM またはアプリケーション自体によって引き起こされます。

1 JVM にバグがある。これは通常、JVM がオペレーティングと対話するときに発生します。JVM 内のバグはコア ダンプを引き起こす可能性があります。

2 JRun アプリケーションが、Java Native Interface (JNI) などを使用してエラーのあるネイティブ コードを使用している。たとえば、ODBC ドライバを呼び出す Type 1 JDBC ドライバを使用してデータベースにアクセスすると、ドライバのネイティブ コード がスレッド セーフではないため、デッド ロック およびコア ダンプを引き起こします。

Allaire では、コア ダンプに関する文書を知識ベースで公開しています。知識ベースの記事番号 15437 は、次の URL で参照できます。

`http://allaire.com/Support/KnowledgeBase/SearchForm.cfm`

メモリ不足エラーの処理

JVM の各インスタンスは、**ヒープ**と呼ばれるスレッドで共有されるすべてのオブジェクトに対してメモリの割り当てを使用します。実行時に、JVM はすべてのクラス インスタンスおよび配列に対してヒープからメモリを割り当てます。作成したサーブレットまたは JSP ページによって、エラー メッセージ `java.lang.OutOfMemoryError` が JRun ログファイルまたはスタック トレースに書き込まれた場合は、使用する JVM の最大ヒープ サイズを増やす必要があります。

メモ
作成するアプリケーションで `OutOfMemoryError` が発生しないように注意してください。一般的に、これらのエラーは予測できません。また、これらのエラーが発生した場合、復旧することもできません。このエラーが発生する前に、JVM ではメモリ不足エラーが発生し、ガーベッジ コレクタがメモリを解放して処理を続けることができなくなります。

`OutOfMemoryError` は、新しいオブジェクトにメモリを割り当てる無限ループなどのプログラミング エラーによって発生する可能性もあります。このタイプのプログラミング エラーは、JVM に割り当てるメモリの量にかかわらずエラーを引き起こします。

既定のヒープ サイズは JVM によって異なります。ただし、ほとんどの JVM では、ヒープの最小サイズおよび最大サイズを変更でき、これらの設定の既定値が用意されています。次の一覧に、Sun と IBM の両方の JVM に最も一般的なヒープ設定を示します。ほとんどの JVM ベンダは、Sun のルールに従ったヒープ サイズの既定値を採用しています。

- Sun の Windows および Solaris バージョン 1.1 JVM　既定の最大ヒープサイズは 16 MB です。ほとんどのサーバー側 Java アプリケーションでは、この既定サイズは十分ではありません。既定の最小ヒープ サイズは 1 MB です。
- Sun の Windows および Solaris バージョン 1.2 および 1.3 JVM　既定の最大ヒープサイズは 64 MB で、既定の最小ヒ ープサイズは 1 MB です。既定の最大ヒ ープサイズは十分ですが、ほとんどのエンタープライズ レベルのサーバー側 Java アプリケーションには、これ以上のサイズが必要です。

  選択する設定は、テスト、試用、およびエラーによって決定してください。この設定はアプリケーションと負荷に大きく依存します。
- IBM バージョン 1.1.8 JVM　既定のヒープ サイズは総物理メモリ容量の半分です。

最大ヒープ サイズを増やすには

1 JMC で、[*サーバー名*] > [JVM の設定] > [Java 引数] プロパティを使用して、ヒープ サイズを設定します。

   - たとえば、1.1 JDK の場合は、次の引数を挿入して、ヒープ サイズを 64 MB に設定します。

     ```
     -mx64m
     ```
   - 1.2 JDK の場合は、次の引数を挿入して、ヒープ サイズを 128 MB に設定します。

     ```
     -Xmx128m
     ```

メモリ 設定の既定の単位はバイト です。数字がキロバイト またはメガバイト で解釈されるように指定するには、接尾辞 `k` または `m` を追加する必要があります。

注意
これらの設定の指定に不正な構文を使用すると、JVM が起動しなくなる可能性があります。これらのパラメータを変更して、JRun の再起動に失敗した場合は、`global.properties` ファイルまたは `local.properties` ファイルの設定を手作業で編集する必要があります。

次の例は、Java バージョン 1.2 の JVM の最小ヒープ サイズ、最大ヒープ サイズ、およびその他の値を設定します。

```
-Xms64m -Xmx128m -Xrs -Djava.compiler=NONE
```

正確な構文および既定値については、それぞれ対応する JDK のマニュアルを参照してください。

メモ
Allaire 知識ベースの記事番号 13940 では、この状況についても説明しています。Allaire 知識ベースの詳細については、`http://allaire.com/Support/KnowledgeBase/SearchForm.cfm` を参照してください。

クライアント/サーバー間の通信の監視

JRun には HTTP スニファ メカニズムが用意されています。このメカニズムにより、Web クライアントと HTTP サーバーの通信を監視して、その情報を JRun ログ ファイルに転送できます。スニファ メカニズムを使用して、HTTP 要求/応答のヘッダ部分とコンテンツ部分の両方を追跡できます。要求、Web サーバーからのクッキー設定、およびその他の要求/応答コンテンツの一部として、クライアントから渡されるパラメータを調べることができるため、スニファ メカニズムはアプリケーションをデバッグする際に役に立ちます。

スニファ メカニズムは HTTP ポートを監視して、クライアント要求を検出します。クライアントがそのポートに対して HTTP 要求を作成すると、スニファ メカニズムはその要求を読み取り、ログに記録して、要求をターゲット Web サーバーに転送します。スニファは、Web サーバーからの応答を待ち、受け取った応答をログに記録して、応答をクライアントに転送します。

スニファ メカニズムが要求を処理するために、要求をスニファの HTTP ポートに転送します。既定では、スニファ ポートは 8101 に設定されます。次に、スニファはその要求を処理するために Web サーバーに転送します。したがって、既定のスニファ ポートへの URL は次のような形式になります。

`http://localhost:8101/`*`resource`*

次の形式を使用すると、スニファ メカニズムをバイパスして、同じ要求を Web サーバーに直接転送できます。

http://localhost/*resource*

スニファ メカニズムは複数の要求を同時に処理できます。その結果、異なる要求からのログ メッセージがログ ファイルにインターリーブされます。このため、スニファ メカニズムでは、ログ ファイルの各行の先頭に固有の要求 ID を付けて要求を識別します。

スニファ メカニズムの設定

スニファ メカニズムを設定するには、プロパティ ファイルを使用します。JRun 管理コンソール (JMC) からスニファ メカニズムを制御することはできません。

JRun の 1 つのインストールに関連するすべての JRun サーバーのスニファ メカニズムのための既定の設定は `global.properties` ファイルに格納されています。通常、`global.properties` ファイルは変更しません。個々の JRun サーバーの設定を変更する場合は、`local.properties` ファイルのそのサーバーの設定を変更します。

`global.properties` ファイルでの既定のプロパティ設定は次のとおりです。

```
sniffer.class=allaire.jrun.http.Sniffer
sniffer.port=8101
sniffer.loglevel=info
sniffer.logcontent=true
sniffer.target.host=localhost
sniffer.target.port=80
```

これらのプロパティは、次のオプションを指定します。

- スニファ メカニズムを定義するクラス
- スニファ メカニズムが要求を監視する HTTP ポート
- JRun ログ ファイルのスニファ メッセージの種類
- 要求 / 応答のヘッダとコンテンツの両方を含むログ情報
- 要求を処理する Web サーバーの名前
- Web サーバーのポート番号

既定では、JRun のスニファ メカニズムは無効になっています。スニファ メカニズムを有効にするには、スニファ メカニズムを JRun サービスの一覧に追加する必要があります。サービスは、`global.properties` または `local.properties` ファイルのいずれかで有効にできます。サービスを `global.properties` で指定すると、すべての JRun でスニファ メカニズムが有効になります。サービスを `local.properties` で指定すると、そのファイルに関連する JRun サーバーについてのみ、スニファ メカニズムが有効になります。

global.propeties を使用してスニファ メカニズムを有効にするには、次の例で示すように、スニファ サービスを jrun.services プロパティに追加します。

```
# 開始するサービスの一覧
jrun.services=scheduler,logging,monitor,license,control,{servlet.servic
        es},{ejb.services},sniffer
```

local.propeties を使用してスニファ メカニズムを有効にするには、global.properties から現在のサービスの一覧をコピーして JRun サーバーの local.properties ファイルに貼り付け、sniffer をサービスの一覧の最後に追加します。

これらのプロパティの詳細については、512 ページの「スニファ メカニズムのプロパティ」を参照してください。

スニファ メカニズムの出力

スニファ メカニズムは出力を JRun サーバーのログ ファイルに書き込みます。次の例はスニファ メカニズムの出力を示しています。この例では、クライアントは次の URL を使用して SnoopServlet を要求します。

```
http://localhost:8101/servlet/SnoopServlet
```

JRun ログ ファイルに書き込まれる情報には次の行が含まれます。

```
03/20 11:58:21 info HTTP Sniffer:Listening on port 8101, target is
        localhost:80
03/20 11:58:38 info SnoopServlet:init
03/20 11:58:50 info HTTP Sniffer:Accepted request 1
03/20 11:58:50 info HTTP Sniffer:Header 1 --> GET /servlet/SnoopServlet
        HTTP/1.0
03/20 11:58:50 info HTTP Sniffer:Header 1 --> Connection:Keep-Alive
03/20 11:58:50 info HTTP Sniffer:Header 1 --> User-Agent:Mozilla/4.7
        (WinNT; I)
03/20 11:58:50 info HTTP Sniffer:Header 1 --> Host:localhost:8101
03/20 11:58:50 info HTTP Sniffer:Header 1 --> Accept:image/gif, image/
        x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
03/20 11:58:50 info HTTP Sniffer:Header 1 --> Accept-Encoding:gzip
03/20 11:58:50 info HTTP Sniffer:Header 1 --> Accept-Language:en
03/20 11:58:50 info HTTP Sniffer:Header 1 -->
        Accept-Charset:iso-8859-1,*,utf-8
03/20 11:58:50 info HTTP Sniffer:Header 1 -->
03/20 11:58:51 info HTTP Sniffer:Header 1 <-- HTTP/1.0 200 OK
03/20 11:58:51 info HTTP Sniffer:Header 1 <-- Server:JRun Web Server/
        3.0
03/20 11:58:51 info HTTP Sniffer:Header 1 <-- Date:Mon, 20 Mar 2000
        16:58:51 GMT
03/20 11:58:51 info HTTP Sniffer:Header 1 <--
        Set-Cookie:jsessionid=953571530946269859;path=/
03/20 11:58:51 info HTTP Sniffer:Header 1 <-- Expires:Thu, 01 Dec 1994
        16:00:00 GMT
03/20 11:58:51 info HTTP Sniffer:Header 1 <-- Connection:Close
```

```
03/20 11:58:51 info HTTP Sniffer:Header 1 <--
        Cache-Control:no-cache="set-cookie,set-cookie2"
03/20 11:58:51 info HTTP Sniffer:Header 1 <-- Content-Type:text/html
03/20 11:58:51 info HTTP Sniffer:Header 1 <--
03/20 11:58:51 info HTTP Sniffer:Content 1 <-- 00000000 30 4C 59 42 41
        32 0D 0A 30 4C 49 45 48 32 30 59   <HTML>..<HEAD><T
03/20 11:58:51 info HTTP Sniffer:Content 1 <-- 00000016 4D 59 41 49 32
        58 65 66 66 77 58 6B 79 7D 63 6B   ITLE>SnoopServle
...
```

この出力では、スニファ メカニズムがポート 8101 で要求を受信し、ポート 80 にその要求を転送しているのがわかります。

スニファ メカニズムは、SnoopSerlvet に対する要求を検出すると、その要求と要求に含まれるすべてのヘッダ情報をログに記録します。要求に対応するログ ファイルの行には、右向矢印 (-->) が含まれています。

Web サーバーからの応答には、ヘッダおよび要求のコンテンツが含まれています。応答に対応するログ ファイルの行には、左向矢印 (<--) が含まれています。応答では、応答のコードと ASCII テキストの両方がログ ファイルに含まれています。

スニファ メカニズムのプロパティ

次のプロパティを使用して、スニファ メカニズムの制御と設定を行います。

sniffer.class

スニファ メカニズムを定義する JRun クラスを指定します。既定では、このプロパティは allaire.jrun.http.Sniffer に設定されます。

sniffer.port

スニファ メカニズムが HTTP 要求を監視するポート番号を指定します。既定のポート番号は 8101 です。

スニファ メカニズムとスニファ メカニズムに関連する Web サーバーは、異なるポート番号を使用する必要があります。したがって、プロパティ sniffer.port と sniffer.target.port には、異なる値を指定する必要があります。

sniffer.loglevel

ログ ファイルへのスニファ出力のログ イベント タイプを指定します。指定できる値は、debug、error、warning、および info です。既定値は info です。

JRun ログ メカニズムは、メッセージのレベルに基づいてメッセージをフィルタ選択できます。詳細については、第 38 章 を参照してください。

sniffer.logcontent

true に設定すると、HTTP 要求 / 応答のメッセージのコンテンツをログに記録することを指定します。false に設定すると、スニファ メカニズムは要求 / 応答ヘッダのみをログに記録します。メッセージのコンテンツはログに記録されませんが、クライアントに転送されます。

既定値は true です。

sniffer.target.host

クライアント要求を処理する Web サーバーのホスト名を指定します。既定値は、`localhost` です。

sniffer.target.port

`sniffer.target.host` によって指定された Web サーバーが、スニファ メカニズムから転送された要求を受信するポート番号を指定します。スニファ メカニズムは、`sniffer.port` によって指定されたポートで要求を認識し、Web サーバーが処理できるように要求をこのポートに転送します。

既定のポート番号は 80 です。

スニファ メカニズムとスニファ メカニズムに関連付けられている Web サーバーは、異なるポート番号を使用する必要があります。したがって、プロパティ `sniffer.port` と `sniffer.target.port` には、異なる値を指定する必要があります。

sniffer.loggername

スニファの出力メッセージを受信するために使用するロガーの名前を任意で指定します。既定では、スニファ出力は、JRun サーバーのログ ファイルに書き込まれます。

このプロパティを使用して、スニファ出力を受信するための独自のロガーを作成できます。たとえば、ロガーを使用して、スニファ出力を独自のファイルに書き込むことができます。ログ収集機能とロガーの詳細については、第 38 章を参照してください。

追加デバッグ リンク

デバッグに関する多くの追加情報およびリンクを参照のために利用できます。次の表で、役立つリンクをいくつか紹介します。

URL	説明
http://www.unix.hp.com/java/hpjmeter/index.html	データのプロファイリングをグラフィカルに表示することで、パフォーマンスのボトルネックを検出できる、プラットフォームに依存しないツール
http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/perfTech.html	Sun が提供するパフォーマンスの調整および分析に関するヒント
http://developer.java.sun.com/developer/technicalArticles/Programming/Stacktrace/index.html	Sun が提供する、Java スタック トレースの紹介
http://www-4.ibm.com/software/os/warp/performance/javatip.htm	IBM が提供する、Java パフォーマンスの調整に関する情報
http://www-4.ibm.com/software/developer/library/tip-heap-size.html	ヒープ サイズの管理および最適化に関する情報
http://www-4.ibm.com/software/developer/library/java2/index.html	IBM が提供する、Java、スレッド、および Linux でのスケジューリングに関する情報
http://www-4.ibm.com/software/developer/library/perf-checklist/index.html	IBM AIX サーバーに関する Java パフォーマンスの問題の検出と修正
http://www-4.ibm.com/software/developer/library/jinsight/index.html	Java プログラムの実行を視覚化するフリー ツール、Jinsight に関する情報
http://www.research.ibm.com/journal/sj/391/viswanathan.html	Java Virtual Machine Profiler Interface に関する情報
http://java.sun.com/people/billf/heap/	`-Xhprof` ファイルを読み取るための Java Heap Analysis Tool (HAT) に関する情報

カスタム エラー メッセージング

JRun はエラーを検出すると、通常、エラー メッセージをクライアントに出力してエラーを知らせますが、これらの JRun エラー メッセージを表示しないようにしたり、独自のメッセージに置き換えることができます。

このセクションでは、次の 3 つの状況でカスタム エラー メッセージをアプリケーションに挿入する方法について説明します。

- コネクタ エラー メッセージ　Web サーバーと JRun の接続でエラーが発生した場合
- HTTP エラー メッセージ　JRun が Web アプリケーションの HTTP エラーを検出した場合
- Java 例外メッセージ　サーブレットまたは JSP ページが Java 例外を生成した場合

次のセクションでは、これらのエラー状態について説明します。

コネクタによる既定のエラー メッセージの変更

JRun をインストールすると、Web サーバーと JRun との接続を作成する、ネイティブ接続モジュールがインストールされます。このシナリオでは、Web サーバーは JRun のクライアントとして機能します。この接続の詳細については、第 2 章を参照してください。

JRun は、次の理由により、この接続でエラーを検出する可能性があります。

- 接続タイムアウト
- JRun が同時に処理する要求数が多すぎる。
- Web サーバーが JRun に接続できない。
- プロトコル エラー

これらの各エラーに対して、JRun は、HTTP 503 エラーおよび エラー メッセージが JRun から発生したことを指定するエラー メッセージをクライアントに出力します。ただし、JRun の既定のエラー メッセージを書き換えて、独自にカスタマイズしたエラー ページを表示できます。

次の一覧では、独自のエラー メッセージをいくつかの異なる Web サーバーに対して設定する方法について説明します。これらの例では、カスタム エラー メッセージはファイル `error.html` に含まれています。

- IIS

  Microsoft IIS Web サーバーを使用している場合は、次の行をファイル `¥inetpub¥scripts¥jrun.ini` に追加します。

  ```
  errorurl=http://hostname/error.html
  ```

  `jrun.ini` を保存して、Web サーバーを再起動します。

- NES

  Netscape NES Web サーバーの場合、次のエントリを Web サーバーの obj.conf ファイルの JRun init 行に追加します。

  ```
  Init fn=jruninit ... errorurl="http://hostname/error.html"
  ```

  コンフィギュレーション ファイルを保存して、Web サーバーを再起動します。

- Apache

  Apache Web サーバーの場合、次の行をファイル httpd.conf の JRunConfig セクションに追加します。

  ```
  JRunConfig errorurl "http://localhost/error.html"
  ```

  コンフィギュレーション ファイルを保存して、Web サーバーを再起動します。

web.xml を使用した HTTP エラー ページの設定

Web アプリケーションの処理中に、JRun はさまざまな種類の HTTP エラーを検出する可能性があります。HTTP エラーを検出すると、JRun ではクライアントに標準エラー メッセージが表示されます。

Web アプリケーションの web.xml ファイルで、独自のエラー メッセージを設定して、JRun によって検出された HTTP エラーを処理できます。これにより、HTTP エラーが発生した場合に、カスタム エラー ページをクライアントに配信できます。

メモ

web.xml ファイルは、1 つのアプリケーションに関連付けられるため、アプリケーションごとに異なるエラー ページを設定できます。ただし、1 つの JRun サーバー内のすべてのアプリケーションに対してグローバル エラー ページを指定することはできません。

HTTP 403 エラーが発生した場合は、アプリケーションの web.xml ファイルの次の部分によってページ 403.html が表示されます。

```
<error-page>
  <error-code>403</error-code>
  <location>/403.html</location>
</error-page>
```

web.xml を使用した Java 例外メッセージの制御

JRun によって検出されるエラーには、Java サーブレットによって返される Java 例外が含まれます。`web.xml` ファイルを使用すると、JRun が特定の種類の Java 例外を検出した場合に、カスタム エラー ページを表示できます。

次の例は、アプリケーションによって生成された Java 例外に対して、`exception.html` を表示するように JRun を設定します。

```
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/exception.html</location>
</error-page>
```

通常、独自のカスタム例外クラスに対してカスタム エラー ページを作成し、`exception-type` タグを使用して例外クラス名を指定します。

JRun とサードパーティ製 IDE の併用

Microsoft のメモ帳などの簡易テキスト エディタを使用して Web アプリケーションを開発することもできますが、ほとんどの開発者は、より高度な開発環境を使用します。いくつかのソフトウェア ベンダから、Web アプリケーション用のサーブレットと JSP ページの開発、デバッグ、およびテストに使用できる Java Integrated Development Environments (IDE) が提供されています。これらの IDE は次のとおりです。

- Symantec Visual Café
- IBM VisualAge for Java
- Microsoft Visual J++
- Borland JBuilder
- Sun Forte

これらの IDE の共通の機能には、エディタ、デバッガ、ウィザード、ユーティリティ、および開発環境を向上させるその他の機能が含まれています。

これらの IDE と JRun を併用して、Web アプリケーションを開発できます。ただし、これらの環境を使用する場合は、JRun と統合するためにいくつかの設定手順を実行する必要があります。Allaire では、これらの IDE に関する文書を知識ベースで公開しています。この文書では、IDE を JRun で使用するための設定方法について説明しています。知識ベースの記事番号 14529 は、次の URL で参照できます。

http://allaire.com/Support/KnowledgeBase/SearchForm.cfm

第 42 章

JRun の拡張機能

Java サーブレット API 仕様と JSP 仕様には、仕様に従ってそれらのテクノロジを実装するための必要条件が定義されていますが、JRun の顧客から仕様に定義されていない追加機能が要求されることがあります。このような要求を満たすために、JRun 拡張機能が用意されました。

この章では、Java サーブレット API 仕様と JSP 構文仕様に追加された拡張機能について説明します。

目次

- JRun 拡張機能の使用 520
- global.jsa ファイルの使用 520
- サーブレット API の拡張機能 523

JRun 拡張機能の使用

Java サーブレット API 仕様と JSP 構文仕様には、いくつかの拡張機能が追加されています。このような拡張機能は次のとおりです。

- `global.jsa` (JRun サーバー アプリケーション) ファイル　JSP に共通のアプリケーション ロジックのための共有ファイルを作成できます。詳細については、520 ページの「global.jsa ファイルの使用」を参照してください。
- サーブレット API 拡張機能　サーブレットから JSP へ要求を転送できます。詳細については、523 ページの「サーブレット API の拡張機能」を参照してください。

メモ
Web アプリケーションでこれらの拡張機能を使用する場合、これらの拡張機能を別の Web アプリケーション サーバーに移植することはできません。

---

global.jsa ファイルの使用

`global.jsa` ファイルを使用すると、1 つのアプリケーションまたはクライアント セッションからアクセスするすべての JSP ファイルに共通のアプリ ケーション ロジックについて、共有ファイルを作成できます。`global.jsa` ファイルにより、このコードを集中管理するための場所が提供されるので、その場所から多数の JSP で共有されるロジックを管理したり変更できます。

`global.jsa` ファイルは、JSP と同じディレクトリに格納する必要があります。

次のイベントにより、JRun がトリガされ、`global.jsa` ファイルが確認されます。

- **アプリケーション開始イベント**　JRun の開始後、`global.jsa` ファイルを含むディレクトリ内の任意の JSP が最初に要求されると、JRun でファイルが読み込まれます。アプリケーション開始イベントを使用すると、同じアプリケーション内にあるすべての JSP で共有されている JSP `application` オブジェクト に情報を書き込めるようになります。
- **セッション開始イベント**　新規のクライアント セッションごとに (つまり、セッション ID ごとに)、`global.jsa` ファイルを含むディレクトリ内の任意の JSP にアクセスする最初のクライアント要求が行われると、ファイルが読み込まれます。セッション開始イベントを使用すると、1 つのクライアントがアクセスするすべての JSP で共有される JSP `session` オブジェクトを初期化できます。

  1 つのアプリケーションに、JSP の複数のディレクトリを持たせることができます。セッション開始イベントは、アプリケーションの JSP に対する最初のクライアント要求でのみトリガされます。この要求によって、クライアントの `session` オブジェクトが作成されます。これ以降は、異なるディレクトリにある JSP に対する要求であったとしても、セッション開始イベントがトリガされることはありません。

たとえば、global.jsa ファイルを含むディレクトリにある /index.jsp への最初のクライアント要求が行われると、セッション開始イベントがトリガされます。ただし、同じ session オブジェクトとセッション ID を持つ同じクライアントが、続けて別のディレクトリにある /store/checkout.jsp にアクセスした場合、このディレクトリに global.jsa ファイルが含まれていても、/store/global.jsa のセッション開始イベントはトリガされません。

- **アプリケーション終了イベント**　アプリケーションが終了すると、JRun により、JSP を供給するすべてのディレクトリにある global.jsa ファイルの有無がチェックされます。ファイルが見つかった場合、そのファイルが読み込まれます。
- **セッション終了イベント**　セッションが終了すると、JRun により、JSP を供給するすべてのディレクトリにある global.jsa ファイルの有無がチェックされます。ファイルが見つかった場合、そのファイルが読み込まれます。

JRun をトリガして、global.jsa ファイルをチェックするイベントにはそれぞれ、イベント トリガが検知されたときに実行される関連メソッドが用意されています。これらのメソッドを編集して、global.jsa ファイルにロジックを挿入できます。次の表はイベントと対応するメソッドの一覧です。

イベント	メソッド
アプリケーション開始	applicationInit()
セッション開始	sessionInit(HttpSession session)
アプリケーション終了	applicationDestroy()
セッション終了	sessionDestroy(HttpSession session)

各メソッドでは、次の JSP オブジェクトにアクセスできます。

- sessionInit と sessionDestroy では、session オブジェクトにアクセスできます。
- applicationInit と applicationDestroy では、application オブジェクトにアクセスできます。

global.jsa ファイルの例

次の例では、global.jsa ファイルにある各メソッド の構文とメソッド 定義の例を示します。

```
<%! public void sessionInit(HttpSession session) {
  System.err.println("session init:" + session.getId());
  session.setAttribute("IDString","Session ID:" + session.getId());
}%>

<%! public void sessionDestroy(HttpSession session) {
  System.err.println("session destroy:" + session.getId());
}%>

<%! public void applicationInit() {
  application.setAttribute("appName", "MyApp");
}%>

<%! public void applicationDestroy() {
  System.out.println("Application terminated:" + (String)
        application.getAttribute("appName"));
}%>
```

global.jsa ファイルの有効化

JRun による global.jsa ファイルに対するチェックを有効にするには、次の手順を実行します。

1 JRun 管理コンソール (JMC) で、[*サーバー名*] > [Web アプリケーション] > [*アプリケーション名*] > [JavaServer Pages] プロパティを選択します。対応するフォームが JMC の右側に開きます。

2 [編集] ボタンをクリックします。

3 [global.jsa (JSP 1.1) の検索] チェックボックスをオンにします。

このプロパティを有効にすると、JRun によって global.jsa ファイルの存在がチェックされます。無効にした場合は、global.jsa ファイルのチェックは行われません。JRun のインストール時は、このプロパティは無効になっています。

JSP で global.jsa ファイルが使用されていない場合、このチェックボックスをオフのままにしておくと、必要のない処理オーバヘッドを避けることができます。

JMC の詳細については、『JRun セットアップ ガイド 』を参照してください。

サーブレット API の拡張機能

JRun には、サーブレットの JSP ドキュメントをプログラム内から呼び出す機能があり、その機能により JSP ドキュメントがサポートされています。

allaire.jrun.servlet.JRunResponse クラス

このクラスにより、`javax.servlet.http.HttpServletResponse` インターフェイスが実装され、サーブレットでページを呼び出し、必要に応じてコンテキストを渡せるようにするメソッドが追加されます。このクラスは JRun の以前のリリースとの下位互換性を保つためにあり、`com.livesoftware.jrun.JRunServletResponse` として参照できます。

メモ
新しいアプリケーションの開発または古いアプリケーションの書き替えの場合は、239 ページの「JSP に制御を渡す方法」で説明しているように、`JRunResponse.callPage` を使用しないで `RequestDispatcher.forward` または `RequestDispatcher.include` を呼び出してください。

---

callPage(String fileName, HttpServletRequest req)

このメソッドを使用して、サーブレット内部から JSP を提供します。`request` オブジェクトを使用して、このメソッドに一部のコンテキストを渡すことができます。ファイルは、このページがキャッシュされないことをブラウザに伝えるヘッダディレクティブとともに返されます。

パラメータ

`fileName`　出力を生成し、コンテンツを表示するために使用されるファイルを表す URL 名。「/」で始まる名前は、ドキュメント ルートに対する相対パスであると見なされます。先頭が「/」でない場合は、現在の要求の呼び出しに使用された URL への相対パスと見なされます。

`req`　このメソッドを呼び出すサーブレットの `HttpServletRequest` オブジェクト。`request` オブジェクトのコンテキストでは、コンテンツは通常、Bean として渡されます。

サーブレットの service メソッド内部から JSP を呼び出す例は次のとおりです。

```
service(HttpServletRequest req, HttpServletResponse res)
  throws IOException, ServletException {
  JRunServletRequest jrunReq = (JRunServletRequest)req;
  // 要求オブジェクトに属性を保存します。
  jrunReq.setAttribute("greeting", "Hello World");
  JRunServletResponse jrunRes = (JRunServletResponse)res;
  jrunRes.callPage("/a.JSP", jrunReq);
}
```

このサーブレット の例では、次の例に示す HTTPServletRequest.getAttribute メソッドを使用して request-scope 属性にアクセスできる a.jsp という JSP ページを呼び出しています。

```
<% String greeting = (String)request.getAttribute("greeting"); %>
```

別のサーブレットへの制御の受け渡しの詳細については、238 ページの「制御の受け渡し」を参照してください。

第 43 章

JRun と ColdFusion の併用

ColdFusion には、JRun と統合するためのタグが 2 つあります。

- CFSERVLET　ColdFusion ページからサーブレットを呼び出します。
- CFOBJECT　ColdFusion ページから EJB を呼び出します。

この章では、CFSERVLET および CFOBJECT の使用方法について説明します。

目次

- JRun および ColdFusion 526
- CFSERVLET の使用.. 526
- CFOBJECT の使用.. 532

JRun および ColdFusion

ColdFusion には、JRun と統合するためのタグが 2 つあります。

- CFSERVLET　ColdFusion ページからサーブレットを呼び出します。
- CFOBJECT　ColdFusion ページから EJB を呼び出します。

この章で提供される情報は、次の条件を満たしていることを前提としています。

- ColdFusion と JRun の両方が正常にインストールされている。
- CFML のコーディング方法を理解している。
- Java サーブレットのコーディング方法を理解している。
- EJB クライアントのコーディング方法を理解している。
- JRun コネクタ ウィザードをすでに実行し、JRun サーバーのコネクタ ポート (JRun プロキシ ポートとも呼ばれる) を確認している。

CFSERVLET の使用

ColdFusion サーバーでは CFSERVLET タグを処理するとき、JRun に要求を送信します。JRun サーブレット エンジンではサーブレットを処理し、変更されたサーブレット属性値を含むサーブレットの出力とともに、制御を ColdFusion サーバーに戻します。

CFSERVLET タグを使用すると、ColdFusion ページから Java サーブレットを呼び出すことができます。サーブレットを呼び出すには、ColdFusion ページに次の情報が必要です。

- **サーブレット名**　JMC で指定されたサーブレット名、またはサーブレット クラス名のいずれかを指定します。
- **プロキシ ポート**　Web サーバーが JRun との通信に使用するポートを指定します。このポート番号は、JRun コネクタ ウィザードで Web サーバーと JRun 間の通信を設定するときに指定します。
- **属性**　CFSERVLET を使用すると、CFSERVLETPARAM タグを使用して属性値を渡せます。

これ以降の部分では、CFSERVLET タグの使用方法について説明します。

CFML テンプレートにおけるサーブレットの呼び出し

CFSERVLET タグを使用すれば、CFML ページから簡単にサーブレットを呼び出すことができます。CFSERVLET タグは、JRun エンジンで Java サーブレットを実行します。このタグは CFSERVLETPARAM タグと併用します。CFSERVLETPARAM タグは、サーブレットにパラメータまたは属性、あるいはその両方が設定されている場合、そのサーブレットにデータを渡すためのタグです。

構文

```
<CFSERVLET
 CODE="class name of servlet"
 JRUNPROXY="proxy server"
 TIMEOUT="timeout in seconds"
 WRITEOUTPUT="Yes" or "No"
 DEBUG="Yes" or "No">
 <CFSERVLETPARAM
 NAME="parameter name" or "attribute name"
 VALUE="value"
 >
 ...
</CFSERVLET>
```

サーブレットを初めて使用するユーザは、サーブレットの名前を指定する必要があり、リモート ホストで実行している場合は IP アドレスも指定する必要があるので、属性に関する次の説明は非常に重要です。

CODE

必須。実行する Java サーブレットのクラス名。

JRUNPROXY

オプション。JRun エンジンが実行されているリモート マシンを指定します。既定では、JRun エンジンは、ColdFusion を実行しているホスト で実行されます。リモート ホストの名前を指定する場合は、リモート ホストの IP アドレスの後にコロンを入力し、その後に JRun が受信しているポート番号を指定します。この JRun サーバー コネクタ ポートは、JRun コネクタ ウィザードで指定されます。

例

次の例は、SnoopServlet という最も単純な形式でこのタグを使用する方法を示します。

```
<HTML>
<HEAD>
<TITLE>CFSERVLET</TITLE>
</HEAD>
<BASEFONT FACE="Arial, Helvetica" SIZE=2>
<BODY bgcolor="#FFFFD5">

<H3>CFSERVLET</H3>
<P>
<!--- JRUNPROXY is Web server IP:proxy port.
      To determine proxy port, look at the
      jcp.endpoint.main.port property in the
      local.properties file for the JRun server
      you want to access. --->
<CFSERVLET code="SnoopServlet"
  JRUNPROXY="127.0.0.1:53003"
  TIMEOUT="10" >
</CFSERVLET>

</BODY>
</HTML>
```

このページは、ほかの CFML ページと同じようにブラウザで表示できます。

パラメータと属性が設定されているサーブレットの呼び出し

CFSERVLETPARAM タグを使用すると、パラメータと属性が設定されているサーブレットを CFML テンプレートから呼び出せます。このタグでは参照または値を使用できます。

サーブレット の例

次のサーブレットは、パラメータと属性の両方を使用しています。このサーブレットは属性値を変更し、対応する CFML 変数に新しい値を返します。

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

public class MySimpleServlet extends HttpServlet {

public void doGet (HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException {

  // コンテンツ タイプを設定して、PrintWriter を作成します。
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  // 応答のデータを書き込みます。

  // パラメータの名前を取得します。
  Enumeration enumNames = req.getParameterNames();

  // 各パラメータの名前と値を while ループに書き込みます。
  out.println("<P>Here are the pass-by-value parameters.");
  while ( enumNames.hasMoreElements() ) {
    String strName = (String) enumNames.nextElement();
    out.println("<br> " + strName + ":" + req.getParameter(strName));
  }

  // パラメータ名を知っている場合は、それを名前で参照します。
  out.println("<P>Here are the pass-by-value parameters, again.");
  out.println("<br> Town:" + req.getParameter("town"));
  out.println("<br> State:" + req.getParameter("state"));

  // 属性の名前を取得します。
  Enumeration enumAttrNames = req.getAttributeNames();
  out.println("<P>Here are the attributes.");
  while ( enumAttrNames.hasMoreElements() ) {
    String attrName = (String) enumAttrNames.nextElement();
    out.println("<br> " + attrName + ":" + req.getAttribute(attrName));
  }
  // 属性名を知っている場合は、それを名前で参照します。
  out.println("<P>Here are the attributes, again.");
  out.println("<br> aString:" + req.getAttribute("aString"));
  out.println("<br> aDate:" + req.getAttribute("aDate"));
  out.println("<br> aDouble:" + req.getAttribute("aDouble"));
```

```
// aString 属性を取得して、それを StringObject に割り当てます。
Object StringObject = req.getAttribute("aString");
if(StringObject != null) {
  String myString = (String)StringObject;
  // 印刷します。
  out.println("<br> The string value is:¥"" + myString + "¥"");
  // 文字列を修正します。
  myString = "New value from servlet";

  // 修正した文字列を印刷します。
  out.println("<br> The string value is:¥"" + myString + "¥"");

  // 元のオブジェクトを修正します。
  StringObject = myString;

  // StringObject の値を CFML 変数に返します。
  req.setAttribute("aString", StringObject);
}else{
  out.println("<h2>StringObject is null</H2>");
}
// aDate 属性を取得して、DateObject オブジェクトに割り当てます。
Object DateObject = req.getAttribute("aDate");

// DateObject オブジェクト が Date タイプかどう かをチェックします。そう でない場合は、
// 例外を返します。
if(DateObject != null) {
  if(DateObject instanceof java.util.Date) {
    // (Date) タイプ変換を使用して、myDate にその値を設定することによって、
    // JVM に DateObject が Date クラスを持っていることを知らせます。
    // メモ : Date は Java のクラスです。
    Date myDate = (Date)DateObject;
    // 印刷します。
    out.println("<br>Date value:¥"" + myDate.toString() + "¥"");

    // 修正して返します。
    myDate = new Date(System.currentTimeMillis());

    // 印刷します。
    out.println("<br>MODIFIED date:¥"" + myDate.toString() + "¥"");

    // 元のオブジェクトを修正します。
    DateObject = myDate;

    // DateObject の新しい値を CFML 変数に返します。
    req.setAttribute("aDate", DateObject);
  }else{
    throw new ServletException("Illegal type for aDate - expected
      type java.util.Date and instead the type was:"+
      DateObject.getClass().getName());
  }
}else{
  out.println("<h2>DateObject is null</H2>");
}
```

```
  // 渡した実数値を変更するには、
  // aDouble 属性を取得して、RealObject オブジェクトに割り当てます。
  Object RealObject = req.getAttribute("aDouble");

  // (Double) タイプ変換を使用して、myDouble にその値を設定することによって、
  // JVM に RealObject が Double クラスを持っていることを知らせます。
  // メモ : double は Java のクラスです。つまり、double はスカラー タイプです。
  if(RealObject != null) {
    Double myDouble = (Double)RealObject;

    // toString メソッドを使用して、myDouble の値を印刷します。
    out.println("<br> Value of Parameter that was passed in:" +
        myDouble.toString() + "¥"");

    // myDouble オブジェクトからスカラー値 (実数/倍精度) を抽出し、
    // それを dVal に割り当てます。
    double dVal = myDouble.doubleValue();

    // 渡された実数値に 100 を加えます。
    dVal += 100.0;

    // 新しい値を double タイプの myDouble に割り当てます。
    myDouble = new Double(dVal);

    // 修正した値を印刷します。
    out.println("<br> Double:" + myDouble.toString() + "¥"");

    // 元のオブジェクトを修正します。
    RealObject = myDouble;

    // サーブレットの aDouble 属性から CF 変数を返します。
    req.setAttribute("aDouble", RealObject);
  }else{
    out.println("<h2>RealObject is null</H2>");
  }

  // out.println("</BODY>");
  out.close();
}

public String getServletInfo() {
  return "A simple servlet";
  }
}
```

CFML の例

次の CFML テンプレートでは、前述のサンプル Java サーブレットを呼び出します。Java サーブレットで参照する各パラメータおよび属性は、CFSERVLETPARAM タグを使用して渡す必要があります。

パラメータを渡すには、属性 NAME と VALUE を使用します。NAME はパラメータの名前、VALUE はその値を表します。属性を渡すには、属性 NAME、VARIABLE、および TYPE を使用します。NAME は Java サーブレットの属性名、VARIABLE は CFML ページで対応する変数名、TYPE は変数のデータ タイプを表します。TYPE は、値のタイプが整数、ブール値、または実数値 (浮動小数点) の場合にのみ指定します。

```
<HTML>
<HEAD>
<TITLE>CFSERVLET</TITLE>
</HEAD>

<BASEFONT FACE="Arial, Helvetica" SIZE=2>
<BODY  bgcolor="#FFFFD5">

<H3>CFSERVLET</H3>

<CFPARAM name="aUser" default="Mary Horvath">
<CFPARAM name="aReal" default="44.6">
<CFPARAM name="theDate" default="#now()#">

<p>These are the values of the attributes that are to be passed into the
        servlet.</p>
<cfoutput>
<ul>
<li>User:#aUser#
<li>Real Number:#aReal#
<li>Date:#theDate#
</ul>
</cfoutput>
<p>Call MySimpleServlet to demonstrate the following:
<ul>
<li>How to pass parameters.
<li>How to pass attributes.
<li>How to change the value of an attribute within a servlet and return
        it to the CFML page.
```

```
</ul>
<CFSERVLET jrunproxy="52000" code="MySimpleServlet" debug="Yes">
<cfservletparam name="aString" variable="aUser" type="STRING">
<cfservletparam name="aDouble" variable="aReal" type="REAL">
<cfservletparam name="aDate" variable="theDate" >
<cfservletparam name="town" value="Concord">
<cfservletparam name="state" value="MA">
</cfservlet>
<p>Here are the modified values of the attributes:</p>
<CFOUTPUT>
<P>User:#aUser#
<P>Number:#aReal#
<P>Date:#theDate#
</CFOUTPUT>

</BODY>
</HTML>
```

CFOBJECT の使用

CFOBJECT タグを使用して EJB にアクセスする場合は、このタグを使用して JRun との通信に必要なオブジェクトへの参照を取得します。オブジェクトへの参照を取得すると、ColdFusion ファイルから EJB にアクセスして、その EJB のメソッドを呼び出すことができます。CFOBJECT を使用するには、次の手順を実行します。

- JRun に EJB を公開します。
- ColdFusion Administrator で Java 設定を定義します。
- CFML ファイルをコーディングします。

EJB の公開

アクセス対象の EJB は、JRun サーバーにすでに公開されていなければなりません。EJB 公開の詳細については、第 35 章を参照してください。ここでは、*JRun インストールルート* /samples/sample1a/ejbeans にある Balance エンティティ Bean にアクセスします。このサンプルを実行するには、まず『JRun サンプル ガイド』で説明しているサンプル 1a をコンパイルして公開する必要があります。

ColdFusion Administrator における Java 設定の定義

CFOBJECT を使用して EJB と対話する前に、次の設定を ColdFusion Administrator の Java ペインで定義する必要があります。

- 使用する Java VM。次のように指定します。
  - *jdkpath*¥jre¥bin¥classic¥jvm.dll
  - JVM の起動時に使用するクラスパス。次の項目が含まれていなければなりません。
  - `ejipt_client.jar` ファイルのファイル名とパス
  - `/runtime` ディレクトリにある EJB の `.jar` ファイルの名前とパス。このファイルは Deploy ツールにより作成され、JRun の起動時に `/deploy` ディレクトリから `/runtime` ディレクトリにコピーされます。

    次にサンプル クラスパス指定の例を示します。

    ```
    c:¥program files¥allaire¥jrun¥lib¥ejipt_client.jar;
      c:¥program files¥allaire¥jrun¥servers¥default¥runtime¥
      sample1a_ejb.jar
    ```
- JVM 初期化パラメータ。セキュリティ ポリシーと Java セキュリティ マネージャ クラスを次のように指定します。

  ```
  java.security.policy=c:¥program files¥allaire¥jrun¥lib¥
       jrun.policy;java.security.manager=java.rmi.RMISecurityManager
  ```

EJB を公開し、ColdFusion Administrator で設定を定義したら、サーバー マシンを再起動してください。

CFML ファイルのコーディング

CFML ファイルから EJB にアクセスするには、次の手順を実行します。

- `javax.naming.context` への参照を取得します。この参照は、プロパティで使用する定数値へのアクセスに使用します。
- `java.util.Hashtable` への参照を取得します。この参照は、プロパティの定義に使用します。CFOBJECT を呼び出したら、`init` メソッドを呼び出して `java.util.HashTable` コンストラクタを呼び出します。
- `javax.naming.InitialContext` への参照を取得します。CFOBJECT を呼び出したら、プロパティを指定して `init` メソッドを呼び出し、`java.util.InitialContext` コンストラクタを呼び出します。初期化が終了すると、この参照は EJB の検索に使用されます。
- EJB ホーム オブジェクトへの参照を取得します。
- EJB インスタンスを作成するか、既存の EJB インスタンス (既存の EJB インスタンスはエンティティ Bean にのみ適用) にアクセスします。
- EJB のメソッドを呼び出します。

次の CFML ファイルでは、サンプル 1a で定義された Balance Bean にアクセスして、save メソッドを呼び出します。

```
<html>
<head>
  <title>CFOBJECT Test</title>
</head>

<body>
<H1>CFOBJECT TEst</H1>
<!-- Context オブジェクトを作成して、静的なフィールドで取得します。--->
<CFOBJECT
  action=create
  name=ctx
  type="JAVA"
  class="javax.naming.Context">

<!-- Properties オブジェクトを作成して、明示的コンストラクタを呼び出します。--->
<CFOBJECT
  action=create
  name=prop type="JAVA"
  class="java.util.Hashtable">

<!--- CFOBJECT によって与えられる初期化メソッドを呼び出して、
        HashTable コンストラクタを呼び出します。--->
<cfset prop.init()>

<!-- プロパティを指定します。--->
<cfset prop.put(ctx.INITIAL_CONTEXT_FACTORY,
        "allaire.ejipt.ContextFactory")>
<cfset prop.put(ctx.PROVIDER_URL, "ejipt://rnielsen:2323")>
<cfset prop.put(ctx.SECURITY_PRINCIPAL, "chief")>
<cfset prop.put(ctx.SECURITY_CREDENTIALS, "pass")>

<!-- InitialContext を作成します。--->
<CFOBJECT
  action=create
  name=initContext
  type="JAVA"
  class="javax.naming.InitialContext">

<!--- CFOBJECT によって与えられる初期化メソッドを呼び出して、
      InitialContext コンストラクタにプロパティを渡します。--->
<cfset initContext.init(prop)>
```

```
<!--- ホーム オブジェクトへの参照を渡します。--->
<cfset home = initContext.lookup("sample1a.BalanceHome")>
<!--- エンティティ Bean の新しいインスタンスを作成します。
      (ハードコード化された預金口座番号)
      または、検索メソッドを使用して既存のエンティティ Bean を探します。--->
<cfset balance = home.create(123)>

<!--- エンティティ Bean 内でメソッドを呼び出します。
      この例では、500 ドルを預金するようにハードコード化されています。--->
<cfset balance.save(500)>

<!--- コンテキストを終了します。--->
<cfset initContext.close()>

</body>
</html>
```

索引

記号

->　114
<!--　114
--%>　114
<%　123
<%--　114
<%!　122
<%=　123
<%@　115
== 演算子　339
2 フェーズ コミット トランザクション管理　53, 384

A

activation.jar　293
admin JRun サーバー　19
Allaire
　お問い合わせ先　xxv
　テクニカル サポート　xxv
allaire.ejipt.DefaultStore　309
allaire.ejipt.Ejipt　299
allaire.ejipt.tools.Deploy　420
allaire.jrun.ejbContext セッション変数　382
allaire.jrun.security.AuthenticationInterface　481
allaire.jrun.security.PropertyFileAuthentication　480
allowedIdentities　379, 389
`application.xml` ファイル　426
assembly-descriptor 要素　412
AT_BEGIN スコープ　271
AT_END スコープ　271

B

Bean インスタンス プール　306
Bean 管理トランザクション　388
Bean 管理パーシスタンス (BMP)　53, 320
Bean コンテキスト　306
Bean のダイナミック ローディング　422
Bean プロパティ
　Bean 情報　297
　書き換え　301
　公開記述子　296
　説明　411
BodyContent オブジェクト　265
BodyTagSupport クラス　258
　定義　258
　本文コンテンツとの対話　265
BodyTag インターフェイス　258
Borland JBuilder　517
BytesMessage インターフェイス　351

C

CFOBJECT タグ　532
CFSERVLET タグ　526
Class.forName メソッド　242
`classes` ディレクトリ　294
ClusterCATS　9
cmp-field 要素　343, 412
ColdFusion
　CF Administrator における Java 設定　533
　EJB アクセス　532
　サーブレット アクセス　526
Collection、複数行 finder 結果　342
config オブジェクト　44, 139
　「ServletConfig オブジェクト」も参照
container-transaction 要素　384
CreateException、エンティティ ホーム インターフェイス　313
create アクション　332
create メソッド　52

D

`default.definitions` ファイル　195
`default.properties` ファイル　300
`default.template` ファイル　192
default_exports.jar　292, 394
default_objects.jar　292
default JRun サーバー　19
　Web アプリケーション　63
　既定の Web アプリケーション　69
DefaultStore クラス　309
DELETE SQL ステートメント　340
Deploy ツール　55
　Bean ホーム インターフェイス　313
　Bean リモート インターフェイス　312
　`deploy` ディレクトリ　291
　`runtime.properties` ファイル　297
　実行　420
`deploy` ディレクトリ　293, 418
destroy メソッド　207
　書き換え　229
　コーディング　230
dirty フラグ　339
doAfterBody メソッド　265
`docs` ディレクトリ　291
doDelete メソッド　235
doEndTag メソッド
　本文コンテンツ　265
　戻り値　259
doGet メソッド　232
doHead メソッド　235
doInitBody メソッド　265
doOptions メソッド　235

doPost メソッド　233
doPut メソッド　235
doStartTag メソッド　259
doTrace メソッド　235
DriverManager.getConnection
　　メソッド　242

E

EarDeploy ツール　430
EAR ファイル
　J2EE アプリケーションの公
　　開　426
　作成　429
ejb.allowedIdentities　379, 389
ejb.isReentrant
　Bean プロパティの例　301
　true value　301
ejb.jar　293, 394, 423
ejb.sessionTimeout　413
ejbActivate メソッド
　エンティティ Bean　315
　セッション Bean　317
ejb-class 要素　411
EJBContext.getEnvironment().getP
　　roperty メソッド　301
ejbCreate メソッド　322, 336
　エンティティ Bean　315
　セッション Bean　317
ejbeans.jar　292
ejbFindByPrimaryKey
　　メソッド　326, 341
ejb-jar.xml ファイル
　がない場合　297
　公開記述子　411
ejbLoad メソッド　323, 338
　ejbActivate メソッド　315
　エンティティ Bean　316
ejb-name 要素　411
EJBObject　312
ejbPassivate メソッド
　エンティティ Bean　315
　セッション Bean　317
ejbPostCreate メソッド　315, 322,
　　337
ejbRemove メソッド　325, 340
　エンティティ Bean　315
　セッション Bean　317
ejbStore メソッド　324, 339
EJB エンジン
　JRun 統合　47
　スタンドアロン モード　395
EJB クライアントのセットアッ
　　プ　396
ejipt　306
ejipt.cache　294
ejipt.classServer.host　419
ejipt.classServer.port　419
ejipt.createSQL　332
ejipt.createSQL.fields　332
ejipt.createSQL.params　332
ejipt.createSQL.paramTypes　332
ejipt.createSQL.source　332
ejipt.ejbJars　420
ejipt.ejbJars プロパティ　297
ejipt.ejbJars、定義　419
ejipt.findnameSQL　332
ejipt.findnameSQL.fields　332
ejipt.findnameSQL.params　332
ejipt.findnameSQL.paramTypes　3
　　32
ejipt.findnameSQL.source　332
ejipt.home.port　419
ejipt.isAlwaysDirty プロパ
　　ティ　324, 339
ejipt.isCompatible　420
ejipt.isDataCached　323, 338
ejipt.isDataShared　323, 338
ejipt.isTimeoutFromCreate　414
ejipt.jar　292, 299
ejipt.javac　420
ejipt.loadSQL　332, 338
ejipt.loadSQL.fields　332
ejipt.loadSQL.params　332
ejipt.loadSQL.paramTypes　332
ejipt.loadSQL.source　332
ejipt.logStackTrace　419
ejipt.maxContexts　306
ejipt.maxFreeContexts　306
ejipt.minFreeContexts　306
ejipt.postCreateSQL　332
ejipt.postCreateSQL.fields　332
ejipt.postCreateSQL.params　332
ejipt.postCreateSQL.paramTypes
　　332
ejipt.postCreateSQL.source　332
ejipt.properties
　Bean プロパティによる
　　書き換え　301
　プロパティ ファイルによる
　　書き換え　300
ejipt.properties
　　ファイル　292
　ユーザとロール　54
ejipt.removeSQL　332, 340
ejipt.removeSQL.fields.　332
ejipt.removeSQL.params　332
ejipt.removeSQL.paramTypes　33
　　2
ejipt.removeSQL.source　332
ejipt.sessionScope　390
ejipt.storeClassName　309
ejipt.storeName　309
ejipt.storeSQL　332, 339
ejipt.storeSQL.fields　332
ejipt.storeSQL.params　332
ejipt.storeSQL.paramTypes　332
ejipt.storeSQL.source　332
ejipt_client.jar　292, 423
　クラスパス　394
　セットアップ　396
ejipt_ejbeans.jar　292
ejipt_exports.jar　297, 420, 423
　default_exports.jar　292
　Deploy ツール　291
　JRun の起動　394
　セットアップ　396
　内容　293
ejipt_jms_client.jar　292, 423
ejipt_objects.jar　297, 420
　default_objects.jar　292
　Deploy ツール　291
　トラブルシューティング セット
　　アップ　396
　内容　293
ejipt_tool.jar　292
EjiptProperties オブジェクト　296
enterprise-beans 要素　411
Enterprise JavaBeans
　API の使用　50
　client classloader issues　394
　ColdFusion による
　　アクセス　532
　J2EE アプリケーション　7
　JAR ファイル　26
　定義　48
　クライアント クラスパスの
　　問題点　394
　クライアントのセット
　　アップ　396
　サーブレットからの
　　アクセス　378
　サポートされる仕様　xx
　定義　7
　プロパティ ファイル　26
　ローカル モード　389
　機能　26
　必要条件　26
EntityBean インターフェイス　314
EntityContext インター
　　フェイス　314
entity 要素　411

Enumeration、複数行
finder 結果 342
env-entry 要素 413
EVAL_BODY_INCLUDE doStartTag
戻り値 259
EVAL_BODY_TAG doAfterBody
戻り値 265
EVAL_PAGE doEndTag
戻り値 259
exception オブジェクト 139
exception オブジェクト、
作成 137
extra_exports.jar 297, 420, 421
`ext` ディレクトリ 293

F

findAncestorWithClass
メソッド 269
findByPrimaryKeySQL
env-entry 341
findByPrimaryKey
メソッド 326
CMP 341
エンティティ Bean 315
ホーム インターフェイスでの
例 313
FinderException 313
FinderException、エンティティ
ホーム インターフェイス 313
finder メソッド
CMP 340
エンティティ Bean 313, 325
エンティティ Bean の
複数行 327
findname アクション 332
forward メソッド 238

G

GenericServlet クラス
定義 222
パラメータのアクセス 231
メソッドの書き換え 207, 228
メソッドのコーディング 228
メッセージのロギング 231
GenericServlet クラスのログ
メソッド 231
getAttribute メソッド
セッション値 240
呼び出し側サーブレットと JSP
属性 238
getConnection メソッド 308
getInitParameterNames
メソッド 231
getInitParameter メソッド 219,
231
getLocalEJBHome メソッド 305
getParameter メソッド
既定の動作 281
例 216
getRequestDispatcher
メソッド 238
getResource メソッド 253
getServletConfig メソッド 231
getServletContext メソッド 231
getServletInfo メソッド 229
getSession メソッド 240, 281
getStore メソッド 309
getVariableInfo メソッド 271
`global.jsa` ファイル 520
イベント 521
メソッド 521
有効化 522
例 522
`global.properties` ファイル
Web サーバー接続 436
メソッド タイミング 486
ログ設定 444
定義 28

H

home name 要素 411
home 要素 411
HTML ページ、制御の
受け渡し 239
HttpServletRequest
オブジェクト 41
HttpServletRequest クラス 240
HttpServletRequest
パラメータ 232
HttpServletResponse
オブジェクト 41
HttpServletResponse
パラメータ 232
HttpServlet クラス
doDelete メソッド 235
doGet メソッド 232
doHead メソッド 235
doOptions メソッド 235
doPost メソッド 233
doPut メソッド 235
doTrace メソッド 235
service メソッド 232
定義 222
メソッドの書き換え 207
メソッドのコーディング 232
HttpSession オブジェクト、「セッション オブジェクト」を参照
HTTP サーバー 394
HTTP 要求 / 応答
アクセス 41
サポート 34

I

IBM VisualAge for Java 517
IDE 29
iiop.jar 293
<include> タグ 189
include ディレクティブ
JSP 101
使用 120
include メソッド 252
initialContext オブジェクト 390
init メソッド
API のバージョン 2.1 の
変更点 280
書き換え 229
コーディング 229
サーブレットのライフ
サイクル 206
認証 481
例 219
INOUT パラメータ タイプ 343
INSERT ステートメント 337
`instance.store` ファイル 53,
309
起動時に作成 294
InstanceManager.isFirst
メソッド 306
InstanceManager.isLast
メソッド 307
InstanceManager.setDirty
メソッド 324, 339
InstanceManager クラス 306
invoker サーブレット
Web アプリケーション 74
使用法 83
IN パラメータ タイプ 343
isAlwaysDirty プロパティ 307,
324, 339
isFirst メソッド 306
isLast メソッド 307
isLocal property 389
isValid メソッド 274

J

J2EE アプリケーション 5
3 階層モデル 5
EarDeploy による公開 430
JMC による公開 429
コンポーネント 7
パッケージ化 427

ユーザとロール　430
jar ユーティリティ
EAR ファイル作成　429
EJB　418
WAR ファイルの作成　403
構文　403
java.security.acl.Group　54, 310
java.security.Principal　54, 310
java.sql.Connection　308
java.System.err　462
java.System.out　462
java.util.Properties　296
java:comp/env JNDI
コンテキスト　298
javac
JSP のコンパイル　146
サーブレットのコンパイル　215
Java IDE　29
Java Message Service
API　50
ejipt_jms_client.jar　292
アーキテクチャ　347
概要　346
キューの定義　352
コンポーネント　349
サポートされる仕様　xx
トピックの定義　363
パブリッシュ / サブスクライブ　363
ポイントツーポイント　351
メッセージ ヘッダ フィールド　349
有効化　348
機能　26
定義　26
Java Naming および Directory Interface、「JNDI」を参照
JavaScript　39
JavaServer Pages、「JSP」を参照
Java Transaction Server
サポートされる仕様　xx
javax.ejb.DuplicateKeyException　337
javax.ejb.EJBHome　313
javax.ejb.EJBObject　312
javax.ejb.EntityBean　314
javax.ejb.SessionBean　316
javax.jms.MessageListener　346
javax.servlet.http パッケージ　224
javax.servlet.jsp.tagext
パッケージ　257
javax.servlet パッケージ　222
Java サーブレット API
2.0 と 2.2 の間の変更点　279
javax.servlet.http
パッケージ　224
javax.servlet パッケージ　222
JRunServletResponse
クラス　523
JRun サポート　204
JRun の HTML バージョン　136
Web のアプリケーション認証　464
インターフェイス　205
クラス　205
サポート　222
サポートされる仕様　xx
定義　222
パッケージ　222
リファレンス情報　225
Javaサーブレット API
HttpSession インターフェイス　240
Java データベース API、「JDBC」を参照
Java トランザクション API　50
Java の利点　36
Java、利点　4
jaxp.jar　293
JDBC
API　50
エンティティ Bean データのアクセス　314
ストアド プロシージャの呼び出し　330
データ タイプ　331
データベース アクセス　242
ドライバ　242
ネイティブ API ドライバ　245
ネイティブ プロトコル ドライバ　245
ネット プロトコル ドライバ　245
プリペアド ステート メント　330
jdbc.jar　293, 394, 423
JDBC-ODBC ブリッジ　242
JDK 1.1　396, 420, 423
JDK 1.2　26, 55, 423
jms.jar　293, 394, 423
JMS、「Java Message Service」を参照
JNDI
API　50
Bean ホーム名　411
Bean、登録　52
EJB クライアント検索　378
コンテキスト　246, 411
コンテキスト、Bean の呼び出し　305
jndi.jar　293, 394, 423
JRun
3 階層モデル　5
ClusterCATS　9
J2EE アプリケーション　5
Web サーバー　22
Web サーバーの対話　84
アーキテクチャ モデル　5
開発者コミュニティ　xxii
開発者センター　xxii
開発者リソース　xxii
機能　xx
製品のラインナップ　xxi
データ ソース サービス　246
プログラミング モデル　15
ユーザ タイプ　13
設定　27
`jrun.policy` ファイル　292
jrun_ejbeans.jar　292
jrun_exports.jar　292
`JRUN_HOME` ディレクトリ　55
JRUN_HOME 変数　291
jrun_objects.jar　292
JRun Advanced 版　xxi
JRun Connection Module　59
JRun Developer 版　xxi
JRun Enterprise 版　xxi
jrunpasswd ユーティリティ　480
JRun Professional 版　xxi
JRun Server Tags (JST)　156
JRunServletResponse クラス　523
JRunStats サーブレット
JSP のタイミングの計測　501
クライアントへのタイミングメッセージ　498
JRun Web サーバー
admin JRun サーバー　20
default JRun サーバー　20
使用　23
定義　17
ドキュメント ルート ディレクトリ　23
JRun 管理コンソール (JMC)
J2EE アプリケーションの公開　429
Web アプリケーションの公開　404
EJB 公開　420
JRun サーバー　16
admin　19
EJB　16
JSP　16

JVM 21
Web アプリケーション 16, 63
Web サーバー 17
既定値 19
機能 18
サーブレット 16
複数の Web アプリケーション 60
プロセス モデル 18
JRun デモ アプリケーション 20
JRun 管理コンソール (JMC) 27
JSP
.class ファイル 95
.java ファイル 95
include ディレクティブ 101
Java サーブレット 92
jsp:forward アクション 103
jsp:include アクション 103
JSPC コンパイラ 150
JSP コンパイラ 146
MIME タイプ 117
out オブジェクト 99
request オブジェクト 99
response オブジェクト 99
Web アプリケーションに追加 72
依存チェック 101
エラーページ 107, 119
エラー処理 107
オブジェクト 99
カスタム タグの使用 275
構文 111
コンパイラ 145
コンパイル、無効化 402
作成 93
サポートされる仕様 xx
式 98
仕様 181
条件ロジック 98
仕様のアップグレード 181
スクリプティング 92
スクリプト言語 117
属性 100
タグ ライブラリ 105
定義 7
ディレクトリ 96
内容 92
認証 471
バッファリング 105, 118
パラメータ 100
パラメータ、受け渡し 104
ファイル インポート 117
変換 95
変数 97
メソッド タイミング 501
呼び出し 103
例 169–179
例、Hello World 93
jsp:forward アクション 129
定義 103
jsp:getProperty アクション 128
jsp:include アクション 128
定義 103
jsp:param アクション 130
jsp:plugin アクション 132
jsp:setProperty アクション 126
jsp:useBean アクション 126
JSPC コンパイラ
CLASSPATH 環境変数 153
「JSP コンパイラ」も参照
JSP ページの入力 152
依存チェック 152
オプション 151
起動 151
クラスパス 151
出力ファイルの場所 152
仕様 152
デバッグ メッセージ 151
ヘルプ メッセージ 151
要件 150
例 153
jsprt サーブレット 149
JspWriter クラス 265
JSP オブジェクト
API オブジェクトへのマッピング 136
application 138
config 139
exception 139
JSP で使用 137
out 99, 140
pageContext 141
request 142
response 99, 143
ServletConfig 231
ServletContext 212, 231
session 144
アクセス 137
使用 99
セッション、メソッド 138
要求 99
JSP コンパイラ
JMC 146
「JSPC コンパイラ」も参照
jsprt サーブレット 149
依存チェック 148
既定値 146
クラスパス 146
コマンド ライン 146
コンパイル プロセス 148
再コンパイル 148
自動的にバイパス 148
設定 146
バイパス 147
プレースホルダ 146
プロパティ 146
無効 149
有効化 150
JSP 属性値のエスケープシーケンス 113
JSP との同期化 272
JSP の構文
HTML テキスト 112
include ディレクティブ 120
jsp:forward アクション 129
jsp:getProperty アクション 128
jsp:include アクション 128
jsp:param アクション 130
jsp:plugin アクション 130
jsp:setProperty アクション 126
jsp:useBean アクション 124
JSP コメント 114
page ディレクティブ 116
taglib ディレクティブ 120
URL 115
アクション 124
エスケープ文字 113
空白文字 113
クライアントへのコメント 114
コメント 114
式 123
スクリプト要素 122
スクリプトレット 123
宣言 122
属性の引用 113
タグの配置 113
ディレクティブ 115
テンプレート テキスト 112
jta.jar 293, 394, 423
JTA インターフェイス 53
JVM ヒープ サイズ 508
JVM、サポートするバージョン 21

L

LDAP 50
lib ディレクトリ 292
loadSQL ステートメント 338
load アクション 332
local.properties ファイル
EJB プロパティ 418
Web アプリケーションクラスパス 65

サーバー プロパティ　297
接続ステータス　439
タグレット マッピング　198
ログ メッセージの形式　454
定義　28
logging.class　455
logging.format　455
logging.listeners　456
logging.loglevel　456

M

Mandatory トランザクション属性　385, 386
MapMessage インターフェイス　351
MessageListener インターフェイス
Bean 実装　346
サブスクライバ クラス　368
非同期メッセージ　357
method-permission 要素　412
Microsoft Visual J++　517
MIME タイプ
JSP　117
既定のサーブレット出力　214
チェーン化されたサーブレット　211
Multicaster　348

N

NESTED スコープ　271
Never トランザクション属性　385
NotSupported トランザクション属性　384

O

ObjectMessage インターフェイス　351
onMessage メソッド
実装　346
トピックの発行　368
out オブジェクト　140
OUT パラメータ タイプ　343

P

pageContext オブジェクト　141
page ディレクティブ　116
pass.properties ファイル　481
persistence-type 要素　412
postCreate アクション　332
PrintWriter インターフェイス　42
PropertyFileAuthentication ユーティリティ　480

R

readme.txt ファイル　291
relnotes.htm ファイル　291
RemoteException、Bean リモートインターフェイス　312
Remote Method Invocation API　50
remote 要素　411
remove アクション　332
RequestDispatcher インターフェイス　281
RequestDispatcher オブジェクト
コンテンツのインクルード　252
制御の受け渡し　238
request オブジェクト　142, 199
Required トランザクション属性　384, 386
RequiresNew トランザクション属性　384, 386
ResourceManager
getConnection メソッド　308
getLocalEJBHome メソッド　305
定義　304
response オブジェクト　143
ResultSet オブジェクト　242
RMI ダイナミック クラス ローダ　394
rotationfiles プロパティ　459
rotationsize プロパティ　459
runtime.properties ファイル　293
Deploy の出力　420
ejipt.ejbJars プロパティ　297
定義　420
runtime/classes ディレクトリ　422
runtime ディレクトリ　293

S

samples ディレクトリ　294
security.policy　396
security-role 要素　412
SELECT ステートメント　338
service メソッド　207, 232
servlet.jar　293
ServletConfig オブジェクト　44, 231
「アプリケーション オブジェクト」も参照
ServletContext オブジェクト　44, 212, 231
getRequestDispatcher メソッド　238
getResource メソッド　253
「アプリケーション オブジェクト」も参照
定義　250
ServletOutputStream インターフェイス　42
ServletRequest オブジェクト　238
ServletRequest パラメータ　228
ServletResponse パラメータ　228
<servlet> タグ　188, 217
SessionBean インターフェイス　316
SessionContext　316
session-type 要素　412
session オブジェクト　144
session 要素　411
setAttribute メソッド　238
setDirty メソッド　324, 339
setEntityContext メソッド　306, 315
setRollbackOnly メソッド　386
setSessionContext メソッド　306, 317
SHTML、「SSI」を参照
SKIP_BODY doStartTag EVAL_BODY_INCLUDE doStartTag 戻り値　259
SKIP_PAGE doEndTag 戻り値　259
SQL ステートメント、複数　334
SSI
<include> タグ　189
local.properties ファイル　198
<servlet> タグ　188
概要　39
サポート　188
使用　188
定義　188
SSIFilter　199
SSI タグレット　199
SSL　389
Statement オブジェクト　242
StoreManager　308
store アクション　332
StreamMessage インターフェイス　351
<subst> タグ
default.definitions ファイル　193
forms　193
Sun Fort　517
Supports トランザクション属性　384
Symantec Visual Cafe　517

System.setProperty メソッド　390

T

tagAttribute ディレクティブ　160
TagExtraInfo クラス
- TEI ファイル　271
- コーディング　271
- 属性の定義　262

taglib ディレクティブ　162
- uri 属性　121, 260
- 使用　120
- 例　258

TagSupport クラス　258
tagVariable ディレクティブ　161
tag ディレクティブ　159
TEI クラス、「TagExtraInfo クラス」を参照
TextMessage インターフェイス　351
THTML、「プレゼンテーションテンプレート」を参照
TLD ファイル　260
tools.jar　146
transaction.begin method　388
transaction.commit method　388
transaction.rollback method　388

U

UDP マルチキャスティング　347
unsetEntityContext　307, 315
UPDATE ステートメント　339
URI の要求、「要求 URI」を参照
URL パターン　82
URL マッピング、Web アプリケーション　64
URL、JSP の構文　115
UserManager　310
`users.properties` ファイル
- filename　483
- group.groupName プロパティ　484
- role.roleName プロパティ　484
- user.userName プロパティ　483
- Web のアプリケーション認証　479
- グループ、追加　479
- 更新　480
- コマンド ライン ユーティリティ　480
- 定義　479
- 場所　479
- パスワード、暗号化　480
- パスワード、追加　479
- ユーザ、追加　479
- ロール、追加　479

W

WarDeploy ユーティリティ　405
WAR ファイル
- WarDeploy ユーティリティを使用した公開　405
- Web アプリケーション　25, 76
- 公開　76
- 作成　76, 403

`web.xml` ファイル
- auth-constraint 要素　470
- auth-method 要素　475
- form-error-page 要素　476
- form-login-config 要素　476
- form-login-page 要素　476
- HTTP エラー メッセージ　516
- Java 例外　517
- role-link 要素　473
- role-name 要素　470
- security-constraint 要素　470
- security-role-ref element　472
- taglib 要素　276
- url パターン要素　470
- web-resource-collection 要素　471
- Web アプリケーション　24
- タグ ライブラリ　121
- 認証例　469

`webapp.properties` ファイル
- JSP コンパイルの無効化　149
- クラスパス　65
- 定義　29

Web アプリケーション
- default JRun サーバー　63
- EJB、追加　75
- HTML ファイル、追加　72
- invoker サーブレット　74
- J2EE アプリケーション　7
- Java サーブレット API 仕様と〜　24
- JRun サーバー　60
- JSP、追加　72
- URL マッピング　64
- WAR ファイル　25
- `web.xml` ファイル　24
- `web.xml` ファイルの定義　62
- WEB-INF ディレクトリ　61
- アプリケーションのルートディレクトリ　24
- 移植性　24
- 一時ディレクトリ　61
- エラー処理　212
- オート デプロイ　406
- 開発　70
- 既定値　80
- 既定の Web アプリケーション　67
- クラスの共有　66
- クラスパス　64
- 公開　77, 404
- 公開記述子　24, 62
- 公開、定義　401
- コンポーネント　62
- コンポーネントの追加　71
- サーブレット　73
- 作成　70
- 使用　59
- セキュリティ　212
- タグ ライブラリ、追加　75
- 定義　6, 17, 212
- ディレクトリ構造　24, 60
- ディレクトリ、追加　71
- パッケージ化　76
- 分散　67
- ホット デプロイ　406
- 利点　58
- ルート ディレクトリ　60

Web サーバー
- JRun　22
- JRun コネクタ　8
- JRun サーバー　17
- JRun への接続　22
- 対話　84
- 定義　17
- ネットワーク ポート　22
- バインド アドレス　22
- 要求の処理　22

Web サーバー接続管理
- 既定の形式　438
- 形式　436
- 出力　436
- 設定　435
- 統計　434
- プロパティ　439
- 有効化　434

Web のアプリケーション認証
- authentication.service　482
- authentication.<ServiceName>.class　482
- authentication.<ServiceName>.filename　483
- BASIC 認証　474
- FORM 認証　476
- HTTP アクセス メソッド　470
- `web.xml` ファイル　469
- アクセス ロール　470

アプリケーション認証 467
アプリケーション リソース 464
カスタム認証 481
既定のメカニズム 478
クラス名 482
グループ 466
グループ、定義 484
グループ、例 479
サードパーティのサーバー認証 482
サーバー認証 467
サービス名 482
サーブレット 471
設定 468
認証、BASIC 474
認証、FORM 476
パスワード 480
パスワード、ワイルド カード文字 483
プロパティ 482
無効 468
ユーザ 466
ユーザ認証メソッド、設定 474
ユーザ、定義 483
ユーザ、例 479
要求 467
リソースの URL パターン 470
リソース、URL パターン 470
例 465, 469
ロール 466
ロールのリンク 473
ロール、定義 484
ロール、リンク 473
ロール、例 479
ロール、割り当て 469

X

X/Open XA 仕様 384
XA 準拠データ ソース 308
XA 仕様 384

あ

アクション 124
jsp:forward 129
jsp:getProperty 128
jsp:include 128
jsp:param 130
jsp:plugin 130
jsp:setProperty 126
jsp:useBean 124
「カスタム タグ」も参照
アクセス制御リスト (ACL) 54
アプリケーション アセンブル担当者
J2EE アプリケーション 426
Web アプリケーション 400
アプリケーション
オブジェクト 44
「ServletContext オブジェクト」も参照
アプリケーションの例外 386
アプリケーション マッピング 80
暗黙的サーブレット
マッピング 83

い

イベント リスナ、定義 (ロギング) 450
イベント、定義 (ロギング) 450
インスタンス プール 315
インスタンス マネージャ 306

え

エラー処理 212
エラーページ
JSP 107
Web サーバー 85
エラー メッセージ
HTTP 516
Java 例外 517
カスタマイズ 515
コネクタ 515
ログ 442
エンティティ Bean
概要 314
ローカル モード 389
エンティティ オブジェクト 49
エンティティ ベースのキャッシュ 49

お

オート デプロイ 406

か

ガーベッジ コレクション 49
開発者リソース xxiv
開発ツール 11
拡張性 8
カスタム タグ
JSP 156, 275
JSP での属性のコーディング 265
TLD ファイルでの定義 260
概要 256
スクリプト変数 271
「タグ ライブラリ」も参照
呼び出し 261
仮想マッピング 427
(株) アイ・ティ・フロンティア
Web サイト xxii
環境プロパティ 52
監視ユーティリティ 10
管理 48

き

既定の Web アプリケーション
JWS 69
URL マッピング 67
アプリケーションのルートディレクトリ 69
クラスパス 70
使用 69
ディレクトリ構造 69
特徴 67
マッピング 80
ルート ディレクトリ 67
～の動機 68
～への要求 68
キュー
～からのメッセージの取得 357
定義 (メッセージ) 352
～にメッセージの送信 352
許可、トラブルシューティング
セットアップ 396

く

クッキー 250
クライアント / サーバー接続の監視 509
クライアント区分トランザクション 387
クライアントのセットアップ
EJB 394
トラブルシューティング 396
クラスのロード 394
クラスパス
EJB 299, 394
Web アプリケーション 64
グループ、Web アプリケーションの認証 466
グローバル変数 219

け

警告ログ メッセージ 442
形式、ログ メッセージ、既定 454
現在のメソッド タイミング 486

こ

コア ダンプ 507
公開
EAR ファイル 426
EJB 410

WAR ファイル 401
オート デプロイ 406
概要 397
ホット デプロイ 406
公開記述子
application.xml ファイル 426
ejb-jar.xml ファイル 297
「web.xml ファイル」も参照
コーディング 411
定義 62, 296
公開された Bean のライフサイクル 52
公開担当者 400
コネクタ 8
Web アプリケーションの対話 80
サーバー 347
ソース コード 23
コネクタ管理、「Web サーバー接続管理」を参照
コマンド ラインによる書き換え 299
コンシューマ、定義 346
コンテキスト
Bean 306
Bean インスタンス 316
公開された Bean 52
コンテキスト インスタンス プール 306
コンテキスト パス 81
コンテナ 48
コンテナ管理トランザクション 384
コンテナ管理パーシスタンス (CMP) 53, 330
プライマリ キー クラス タイプ 411
コンテナ プロパティ 297
コンパイラ
Deploy ツール 420
javac 215, 418
JSP 146
JSPC 150
コンポーネントのライフサイクル 48

さ

サーバー側インクルード、「SSI」を参照
サーバー常駐データの共有 49
サーバー プロパティ、設定 296
サーブレット
CGI との比較 37
ColdFusion によるアクセス 526
EJB アクセス 378
GenericServlet クラス 207, 222, 231
HttpServletRequest クラス 240
HttpServlet クラス 207, 222, 232
invoker サーブレット 74
javax.servlet.http パッケージ 224
javax.servlet パッケージ 222
JRun でのライフサイクル 206
SingleThreadModel インターフェイス 209
SSI タグレットによる起動 199
Web アプリケーションに追加 73
Web アプリケーション、登録 73
アプリケーション情報 471
オブジェクトスコープ変数 209
クラススコープ変数 208
サーブレット コンテキスト、例 250
サポートされる仕様 xx
シャットダウン 230
スレッド、管理 208
スレッド、同期化 208
制御の受け渡し、例 238
セッション トラッキング、例 240
セットアップと初期化 229
定義 7
同期化 210
パラメータの受け渡し 199
要求 / 応答プロセス 34
要求パラメータ 216
呼び出し 35
利点 35
サーブレット API、「Java サーブレット API」を参照
サーブレットのチェーン化 210
サーブレットのライフサイクル 206
サーブレット パス 81
サーブレット マッピング 80
再公開オプション 421

し

システムの例外 386
システム ログ プロパティ 462
実行 ID 54
実行時 Bean プロパティによる書き換え 302
実行モード 54
認証 54
情報ログ メッセージ 442
初期化パラメータ
retrieving 219
例 218, 231

す

スクリーン ライター
使用 450
定義 443
トラブルシューティング 451
プロパティ 462
スクリプティング JSP 92
スクリプト要素 122
スクリプトレット 123
スタック トレース 504
Linux 506
Solaris 506
UNIX 506
Windows 505
デッドロック 504
例 505
ログ ファイル 462
スタブ
Deploy ツール 291
ejipt.exports.jar 293
スタンドアロン モード、EJB エンジン 395
ステートの管理 412
ステートフル セッション Bean 316, 412
ステートレス セッション Bean 316, 412
ストア インターフェイス 309
ストアド プロシージャの呼び出し 343
スニファ メカニズム 509
スレッド管理
SingleThreadModel インターフェイス 209
制御 208
スレッド ロガー
設定 444
定義 443
プロパティ 456

せ

セキュリティ 9
EJB 412
EJB、無効化 389
Web アプリケーション 212

「Web アプリケーションの認証」も参照
サポート 54
明示的サーブレット マッピング 83
セッション Bean 316
ステートフル 316
ステートレス 316
ローカル モードのエンティティ Bean 389
セッション オブジェクト 43
「HttpSession オブジェクト」も参照
作成 137
メソッド 138
例 137
セッション スコープ 390
セッション トラッキング 10, 137, 240
セッション ログイン 382
セッション、JSP 118
セッション、メッセージング 346
接続数 53
接続、サーバー 346
設定可能プール 53

そ

属性
getAttribute メソッド 238
JSP 100
JSP、アクセス 101

た

タグ
<include> 189
<servlet> 188, 217
<subst> 193
タグ インターフェイス 258
タグ ハンドラ
概要 257
スクリプト変数、作成 271
スクリプト変数、追加 272
属性との対話 262
「タグ ライブラリ」も参照
ネストしたタグ ハンドラ 269
本文コンテンツとの対話 265
例 258
例、ループ 267
タグ ライブラリ
JSP 105, 120
JSP での使用法 275
taglib ディレクティブ 120
`web.xml` ファイル 121
概要 256
「カスタム タグ」も参照
使用 106
タグ接頭辞 121
ディレクトリ 105
場所 121
パッケージ化 276
タグ ライブラリ記述子、「TLD ファイル」を参照
タグレット 198
タグ、「カスタム タグ」を参照

ち

チェック済みの例外 386

て

ディスパッチ ロガー
定義 443
ファイルへのメッセージのログ 449
プロパティ 457
ディレクティブ
include 120
JSP 115
page 116
taglib 120
データ ソースのプロパティ 320
データベース アクセス 242
データベース接続 53
JDBC-ODBC ブリッジ 242
JDBC ドライバ 245
JRun データ ソース サービス 246
管理 308
テクニカル サポート xxv
デッドロック 389
デバッグ
URL リンク 514
起動 504
クライアント / サーバーの監視 509
コア ダンプ 507
スタック トレース 504
スニファ メカニズム 509
メモリ不足エラー 508
デバッグ ログ メッセージ 442
電子メール
メッセージのログ 450
ログ メッセージの出力先、例 447
電子メール ライター
定義 443
プロパティ 461

と

同期化、スクリプト変数 272
ドット アドレス メカニズム 137
トピック
サブスクライバ 368
定義 363
パブリッシャ 363
トランザクション 53
トランザクション管理 384

に

認証
EJB 412
「Web アプリケーションの認証」も参照

ね

ネスト (子) トランザクション 384

は

パーシスタンス
JRun サポート 53
公開記述子 412
パーシスタント ストア 309
バージョン管理 317
バインド アドレス 22
パス情報 81
バッファリング
JSP 118
jsp:forward 129
jsp:include 128
JSP 出力 105
自動フラッシュ 118
パフォーマンス
アプリケーション 486
サーブレット メソッド 486
デバッグ ログ メッセージ 442
パブリッシュ / サブスクライブ管理 50, 363
パラメータ
HttpServletRequest 232
HttpServletResponse 232
JSP 100
ServletRequest 228
ServletResponse 228
初期化 231
初期化、例 218
要求 216

ひ

ヒープ サイズ 508
非同期メッセージ 49
標準エラー
ログ ファイル名と場所 462

ログ メッセージ 449
標準拡張、トラブルシューティング
セットアップ 396
標準出力
ログ ファイル名と場所 462
ログ メッセージ 449
表面的な比較 339
分散型 2 フェーズ コミット
トランザクション管理 49

ふ

ファイル
個別にメッセージをログ 449
ローテート サイズ、ログ 459
ローテート、標準出力と標準エラーへのログ 450
ローテート、ログ 459
ログ メッセージの出力先、
例 446
ファイル サービス 84
ファイル ライター
既定のログ設定 444
定義 443
プロパティ 458
フェイルオーバ 9
複数行 finder メソッド 327, 342
複数の SQL ステートメント 334
プライマリ キー クラス 341
プライマリ キー クラス
タイプ 411
フラット トランザクション 384
プレゼンテーション テンプレート
<subst> タグ、フォーム 193
概要 39
使用 192
<subst> タグ、数値の定義 193
定義 192
プロパティ
authentication.service 482
authentication.<ServiceName>.class 482
authentication.<ServiceName>.filename 483
EJB 296
EJB Deploy ツールの使用法 418
group.groupName 484
jrun.classpath 65
managing instance.store 309
monitor.class 439
monitor.format 439
monitor.interval 439
monitor.loggername 439
monitor.max.history 439
role.roleName 484
sniffer.class 512
sniffer.logcontent 512
sniffer.loggername 513
sniffer.loglevel 512
sniffer.port 512
sniffer.target.host 513
sniffer.target.port 513
timing.classes 493
timing.<ClassName>.calls 493
timing.<ClassName>.class 493
timing.<ClassName>.<MethodName>.calls 494
timing.<ClassName>.methods 493
timing.<ClassName>.subclasses 494
timing.enabled 492
timing.excludecalls 492
timing.includecalls 492
timing.<LoggerName>.aftermethodcall 495
timing.<LoggerName>.beforemethodcall 495
timing.<LoggerName>.class 495
timing.<LoggerName>.delimiter 495
timing.<LoggerName>.entermethod 495
timing.<LoggerName>.exitmethod 495
timing.<LoggerName>.level 495
timing.logging.class 495
user.classpath 65
user.userName 483
起動する、～で作成された
リスト 296
コンテナ 297
サーバー 296
システム ログ 462
スクリーン ライター 462
スレッド ロガー 456
ディスパッチ ロガー 457
電子メール ライター 461
ファイルによる書き換え 300
ファイル ライター 458
ログ 455
プロパティ ファイル 28
分散ガーベッジ コレクション 49

へ

ページ コンテキスト 情報 42
変数
オブジェクトスコープ 209
グローバル 219

ほ

ポイントツーポイント
メッセージング 50, 351
ホーム インターフェイス 291
ejipt_objects.jar 293
作成 313
ホーム インターフェイスの create
メソッド 313
ポリシー ファイル 396

ま

マルチポート 49

み

未確認の例外 386

め

メソッド
applicationDestroy 521
applicationInit 521
authenticate 481
callPage 523
Class.forName 242
destroy 207, 481
destroy、書き換え 229
destroy、コーディング 230
doAfterBody 265
doDelete、書き換え 235
doEndTag 259
doGet、書き換え 232
doHead、書き換え 235
doInitBody 265
doOptions、書き換え 235
doPost、書き換え 233
doPut、書き換え 235
doStartTag 259
doTrace、書き換え 235
DriverManager.getConnection
242
findAncestorWithClass 269
forward 238
GenericServlet での
コーディング 228
getAttribute、セッション
値 240
getAttribute、呼び出し側サーブレットと JSP 属性 238
getInitParameter 219, 231
getInitParameterNames 231
getParameter 216
getRemoteUser 471

getRequestDispatcher 238
getResource 253
getServletConfig 231
getServletContext 231
getServletInfo 229
getSession 240
getUserPrincipal 471
getVariableInfo 271
HttpServlet での
コーディング 232
include 252
init 206, 219, 481
init、コーディング 229
isPrincipalInRole 481
isUserInRole 471
isValid 274
service 207
sessionDestroy 521
sessionInit 521
setAttribute 238
サービス、書き換え 232
ログ 231
メソッド タイミング
JSP 501
概要 486
クラスの取り込み 493
現在の 486
サブクラスの取り込み 494
出力、クライアントへの
指示 498
設定 486
特定のメソッドの取り込み 493
プロパティ 489
プロパティ、クラスおよび
メソッド 492
プロパティ、ログ 494
メソッドの除外 492
メソッドの取り込み 492
有効化 492
呼び出された 486
例、既定値 488
ログ 494
メッセージ コンポーネント 349
メトリック ログ メッセージ 442
メモリ不足エラー 508

ゆ

ユーザ
EJB UserManager クラス 310
Web のアプリケーション
認証 466
ユーザ Bean 54
ユーザ認証 54
ユーザ タイプ 13

よ

要求 / 応答プロセス 41
要求 URI 81
要求パラメータ 216
要求ログイン 382
呼び出されたメソッド
タイミング 486
呼び出し ID 52, 390

り

リモート インターフェイス 291
ejipt_objects.jar 293
作成 312
リモート オブジェクト 313

れ

例
getInitParameterNames
メソッド 231
getInitParameter メソッド 231
getParameter メソッド 216
init メソッド 219
JSPC コンパイラ 153
taglib ディレクティブ 258
TLD ファイルでの属性の
定義 264
Web のアプリケーション
認証 465, 469
クッキーの処理 249
コンテンツのインクルード 252
初期化パラメータ 218
スタック トレース 505
タグ ハンドラ 258
タグ ハンドラ、ループ 267
データベース アクセス 242
ファイルと電子メールへの
メッセージへ
ログ メッセージ 447
ファイルへのメッセージの
ログ 446
メソッド タイミング、既定 488
例外、処理 42

ろ

ローカル Bean 389
ローカル キャッシュ 308
ローカル ストア 308
ローカル ホーム オブジェクト 305
ローカルホスト、
ejipt.classServer.host
プロパティ 419
ローカル モード、EJB 389
ローテート ファイル、ログ 444
ロード バランス 9
ロール
EJB UserManager クラス 310
security 要素 412
Web のアプリケーション
認証 466
ロール Bean 54
ロガーの定義 453
ログ
group 453
設定 444
プロパティ 455
メソッド タイミング 494
ロガーの定義 453
ログイン セッション 310, 390
ログ ファイル
既定の最大サイズ 444
既定の設定 444
ローテート 459
ローテート サイズ 459
ローテート、標準出力と
標準エラー 450
ログ メッセージ
debug 442
error 442
info 442
warning 442
出力先 442
出力先、変更 449
書式 454
スクリーンへの書き込み 450
電子メールへの書き込み 450
電子メールへの書き込み、
例 447
ファイルへの書き込み、例 446
メトリック 442
ログ ライター 443