



# **JBoss Enterprise Application Platform 5**

## **管理設定ガイド**

JBoss Enterprise Application Platform 5 向け  
エディション 5.1.2



# JBoss Enterprise Application Platform 5 管理設定ガイド

---

JBoss Enterprise Application Platform 5 向け  
エディション 5.1.2

Community JBoss

**編集者**

Documentation Group Red Hat

## 法律上の通知

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## 概要

本書は JBoss Enterprise Application Platform 5 およびその修正リリースに対する管理設定ガイドです。

## 目次

本書の範囲 .....	12
第1章 はじめに .....	13
1.1. JBOSS ENTERPRISE APPLICATION PLATFORM ユースケース .....	14
パート I. JBOSS ENTERPRISE APPLICATION PLATFORM インフラストラクチャー .....	15
第2章 JBOSS ENTERPRISE APPLICATION PLATFORM 5 アーキテクチャー .....	16
2.1. JBOSS ENTERPRISE APPLICATION PLATFORM のブートストラップ .....	17
2.2. ホットデプロイメント .....	17
パート II. JBOSS ENTERPRISE APPLICATION PLATFORM 5 の設定 .....	19
第3章 ネットワーク .....	20
3.1. IPV6 のサポート .....	20
第4章 EJB3 サービスと ENTERPRISE APPLICATION .....	21
4.1. セッション BEAN .....	21
4.2. エンティティ BEAN (JAVA PERSISTENCE API) .....	23
4.2.1. persistence.xml ファイル .....	25
4.2.2. 代替りのデータベースを利用 .....	26
4.2.3. デフォルトの Hibernate オプション .....	27
4.3. メッセージ駆動型 BEAN .....	27
4.4. EJB3 サービスのパッケージとデプロイ .....	28
4.4.1. EJB3 JAR のデプロイ .....	29
4.4.2. EJB3 JAR で EAR をデプロイ .....	29
第5章 ロギング .....	33
5.1. ロギングのデフォルト .....	33
5.2. コンポーネント固有のロギング .....	34
5.2.1. Hibernate でのSQLロギング .....	34
5.2.2. トランザクションサービスのロギング .....	34
第6章 デプロイメント .....	35
6.1. デプロイ可能なアプリケーションのタイプ .....	35
6.1.1. 展開デプロイメント .....	37
6.2. 標準のサーバープロファイル .....	37
6.2.1. プロファイルの変更 .....	38
6.3. コンテキストルート .....	39
第7章 マイクロコンテナ .....	41
第8章 JNDI ネーミングサービス .....	42
8.1. JNDI の概要 .....	42
8.1.1. 名前 .....	42
8.1.2. コンテキスト .....	43
8.1.2.1. InitailContext を使ってコンテキストを取得 .....	43
8.2. JBOSS NAMING SERVICE アーキテクチャー .....	44
8.3. NAMING INTIALCONTEXT ファクトリ .....	46
8.3.1. 標準のネーミングコンテキストファクトリ .....	47
8.3.2. org.jboss.naming.NamingContextFactory .....	48
8.3.3. クラスタ環境でのネーミングディスカバリ .....	48
8.3.4. HTTP InitialContext Factory 実装 .....	49
8.3.5. Login InitialContext Factory 実装 .....	50

8.3.6. ORBInitialContextFactory	50
8.4. HTTP 経由の JNDI	51
8.4.1. HTTP 経由で JNDI にアクセス	51
8.4.2. HTTPS で JNDI にアクセス	54
8.4.3. HTTP 経由の JNDI アクセスでセキュリティを確保	57
8.4.4. セキュリティ保護されていないコンテキストを読み取り専用にして JNDI へのセキュアなアクセスを確保	58
8.5. その他のネーミング MBean	61
8.5.1. JNDI バインディングマネージャー	61
8.5.2. org.jboss.naming.NamingAlias MBean	62
8.5.3. org.jboss.naming.ExternalContext MBean	63
8.5.4. org.jboss.naming.JNDIView MBean	65
8.6. J2EE および JNDI - アプリケーションコンポーネント環境	67
8.6.1. ENC の使用規則	68
8.6.1.1. 環境エントリ	69
8.6.1.2. EJB参照	70
8.6.1.3. jboss.xml および jboss-web.xml でのEJB 参照	72
8.6.1.4. EJB のローカル参照	73
8.6.1.5. リソースマネージャー接続ファクトリの参照	75
8.6.1.6. jboss.xml と jboss-web.xml でのリソースマネージャー接続ファクトリの接続	76
8.6.1.7. リソース環境の参照	77
8.6.1.8. リソース環境参照と jboss.xml、 jboss-web.xml	78
<b>第9章 WEB サービス</b>	<b>79</b>
9.1. WEB サービスの必要性	79
9.2. WEB サーバーとは	79
9.3. DOCUMENT/LITERAL	80
9.4. DOCUMENT/LITERAL (BARE)	80
9.5. DOCUMENT/LITERAL (WRAPPED)	81
9.6. RPC/LITERAL	81
9.7. RPC/ENCODED	83
9.8. WEB サービスのエンドポイント	83
9.9. POJO (PLAIN OLD JAVA OBJECT)	83
9.10. WEB アプリケーションとしてのエンドポイント	83
9.11. エンドポイントのパッケージ化	84
9.12. 生成された WSDL にアクセス	84
9.13. EJB3 ステートレスセッション BEAN (SLSB)	84
9.14. エンドポイントプロバイダー	85
9.15. WEBSERVICECONTEXT	86
9.16. WEB サービスのクライアント	86
9.16.1. サービス	86
9.16.1.1. サービスの使用	87
9.16.1.2. ハンドラーリゾルバー	87
9.16.1.3. Executor	88
9.16.2. 動的なプロキシ	88
9.16.3. WebServiceRef	89
9.16.4. Dispatch	90
9.16.5. 非同期の呼び出し	91
9.16.6. Oneway 呼び出し	92
9.17. 共通 API	92
9.17.1. ハンドラーフレームワーク	92
9.17.1.1. 論理ハンドラー	93
9.17.1.2. プロトコルハンドラー	93

9.17.1.3. サービスエンドポイントのハンドラー	93
9.17.1.4. サービスクライアントのハンドラー	93
9.17.2. メッセージコンテキスト	93
9.17.2.1. メッセージコンテキストへのアクセス	94
9.17.2.2. 論理メッセージコンテキスト	94
9.17.2.3. SOAP メッセージコンテキスト	94
9.17.3. 障害の処理	94
9.18. DATABINDING	95
9.18.1. アノテーションが付けられていないクラスで JAXB を使用	95
9.19. 添付	95
9.19.1. MTOM/XOP	95
9.19.1.1. 対応 MTOM パラメータータイプ	95
9.19.1.2. エンドポイントごとに MTOM を有効にする方法	96
9.19.2. SwaRef	96
9.19.2.1. SwaRef と JAX-WS エンドポイントを併用	97
9.19.2.2. WSDL から開始	98
9.20. ツール	98
9.20.1. Bottom-Up (wsprovide を使用)	99
9.20.2. Top-Down (wsconsume を使用)	101
9.20.3. クライアント側	102
9.20.4. コマンドライン & Ant タスク参照	105
9.20.5. JAX-WS バインディングのカスタマイズ	105
9.21. WEB サービスの拡張	105
9.21.1. WS-Addressing	105
9.21.1.1. 仕様	105
9.21.1.2. エンドポイントの処理	106
9.21.1.3. クライアントの処理	106
9.21.2. WS-Security	108
9.21.2.1. エンドポイント設定	108
9.21.2.2. サーバー側 WSSE 宣言 (jboss-wsse-server.xml)	109
9.21.2.3. クライアント側 WSSE 宣言 (jboss-wsse-client.xml)	110
9.21.2.3.1. クライアント側キーストアの設定	110
9.21.2.4. BouncyCastle JCE プロバイダーのインストール	111
9.21.2.5. Username Token 認証JBOSSCC-50	112
9.21.2.5.1. セキュアトランスポート	115
9.21.2.6. X509 Certificate Token	115
9.21.2.7. JAAS の統合	118
9.21.2.8. POJO エンドポイント認証および権限付与JBOSSCC-50	120
9.21.3. XML レジストリ	122
9.21.3.1. Apache jUDDI の設定	122
9.21.3.2. JBoss JAXR の設定	123
9.21.3.3. JAXR サンプルコード	123
9.21.3.4. トラブルシューティング	126
9.21.3.5. リソース	127
9.22. JBOSSWS の拡張	127
9.22.1. 登録商標のアノテーション	127
9.22.1.1. EndpointConfig	127
9.22.1.2. WebContext	128
9.22.1.3. SecurityDomain	129
9.23. WEB サービス付録	129
9.24. 参考文献	130
<b>第10章 JBOSS AOP</b>	<b>131</b>

10.1. 主な用語	131
10.2. JBOSS AOP でアスペクトを作成	132
10.3. JBOSS AOP でアスペクトを適応	133
10.4. AOP アプリケーションのパッケージ	134
10.5. JBOSS ASPECTMANAGER サービス	135
10.6. SUN JDK を使用した JBOSS ENTERPRISE APPLICATION PLATFORM におけるロード時の変換	136
10.7. JROCKIT	137
10.8. JBOSS ENTERPRISE APPLICATION PLATFORM 環境でロード時のパフォーマンスを改善	138
10.9. AOP をクラスローダーヘスコープ	138
10.9.1. スコープされたクラスローダーの一部としてデプロイ	138
10.9.2. スコープされたデプロイメントへ添付	138
<b>第11章 トランザクション管理</b>	<b>140</b>
11.1. 概要	140
11.2. 必須設定	140
11.3. トランザクションリソース	143
11.4. 最終リソースコミット最適化 (LRCO: LAST RESOURCE COMMIT OPTIMIZATION)	144
11.5. トランザクションタイムアウトの処理	144
11.6. リカバリ設定	144
11.7. TRANSACTION SERVICE のよくある質問	144
11.8. JTS MODULE の利用	145
11.9. XTS モジュールの利用	146
11.10. トランザクション管理コンソール	147
11.11. コンポーネント (トライアル)	147
11.12. ソースコードとアップグレード	147
<b>第12章 リモータリング</b>	<b>149</b>
12.1. 背景	149
12.2. JBOSS REMOTING の設定	149
12.2.1. MBean	149
12.2.2. POJO	150
12.3. マルチホーム化されたサーバー	151
12.4. アドレス変換	152
12.5. 設定ファイルの場所	153
12.6. 詳細情報	153
<b>第13章 JBOSS MESSAGING</b>	<b>154</b>
<b>第14章 JBOSS ENTERPRISE APPLICATION PLATFORM で別のデータベースを使用</b>	<b>155</b>
14.1. 代替りのデータベースの使用法	155
14.2. JDBC ドライバーのインストール	155
14.2.1. Sybase に関する注記	156
14.2.1.1. JAVA サービスの有効化	156
14.2.1.2. CMP の設定	156
14.2.1.3. Java クラスのインストール	157
14.2.1.4. Sybase v15.0.3 向けにデフォルトの @@textsize を増加	157
14.2.2. JDBC DataSource の設定	157
14.3. 一般的なデータベース関連のタスク	158
14.3.1. セキュリティとプーリング	158
14.3.2. JMS サービス用にデータベースを変更	158
14.3.3. CMP サービスで外部キーのサポート	158
14.3.4. Java Persistence API 用にデータベースダイアレクトを指定	158
14.3.5. 外部データベースを使用するよう他の JBoss Enterprise Application Platform サービスを変更	159
14.3.5.1. 簡単な方法	159



14.3.5.2. より柔軟な方法	160
14.3.6. Oracle DataBases に関する注記	160
<b>第15章 DATASOURCE の設定</b>	<b>162</b>
15.1. DATASOURCE の種類	162
15.2. DATASOURCE パラメーター	163
15.3. データソースの例	168
15.3.1. 汎用データソースの例	168
15.3.2. リモート使用向けのDataSource の設定	170
15.3.3. ログインモジュールを使用するようDataSource を設定	171
<b>第16章 プーリング</b>	<b>173</b>
16.1. 戦略	173
16.2. トランザクションスティッキネス	173
16.3. ORACLE での回避法	173
16.4. プールアクセス	174
16.5. プールの充填	174
16.6. アイドル接続	174
16.7. 停止接続	175
16.7.1. 有効な接続チェック	175
16.7.2. SQL クエリ中のエラー	175
16.7.3. プールの変更／終了／フラッシュ	175
16.7.4. その他プーリング	175
<b>第17章 よくある質問 (FAQ)</b>	<b>176</b>
17.1. ORACLE XA の問題	176
<b>パート III. クラスターリングガイド</b>	<b>177</b>
<b>第18章 概要とクイックスタート</b>	<b>178</b>
18.1. クイックスタートガイド	178
18.1.1. 最初の準備	179
18.1.2. JBoss Enterprise Application Platform クラスターの起動	180
18.1.3. Web アプリケーションクラスターリングのクイックスタート	182
18.1.4. EJB セッション Bean クラスターリングのクイックスタート	183
18.1.5. エンティティクラスターリングのクイックスタート	183
<b>第19章 クラスターリングの概念</b>	<b>185</b>
19.1. クラスター定義	185
19.2. サービスアーキテクチャー	186
19.2.1. クライアント側インターセプターアーキテクチャー	186
19.2.2. 外部ロードバランサーアーキテクチャー	187
19.3. 負荷分散ポリシー	188
19.3.1. クライアント側インターセプターアーキテクチャー	188
19.3.2. 外部ロードバランサーアーキテクチャー	189
<b>第20章 ビルディングブロックのクラスター化</b>	<b>190</b>
20.1. JGROUPS とのグループ通信	190
20.1.1. チャネルファクトリサービス	191
20.1.1.1. 標準のプロトコルスタック設定	191
20.1.1.2. プロトコルスタック設定の変更	193
20.1.1.3. JBoss Messagingのプロトコルスタック設定	193
20.1.2. JGroups 共有トランスポート	194
20.2. JBOSS CACHE による分散キャッシング	195
20.2.1. JBoss Enterprise Application Platform の CacheManager サービス	195

20.2.1.1. 標準のキャッシュ設定	196
20.2.1.2. キャッシュ設定エイリアス	198
20.3. HAPARTITION サービス	199
20.3.1. DistributedReplicantManager サービス	201
20.3.2. DistributedState サービス	202
20.3.3. HAPartition のカスタム使用	202
<b>第21章 クラスター化 JNDI サービス</b>	<b>203</b>
21.1. 仕組み	203
21.2. クライアントの設定	205
21.2.1. Enterprise Application Platform 内部で実行されているクライアントの場合	205
21.2.1.1. EJB および WAR からのHA-JNDI リソースへのアクセス -- 環境ネーミングコンテキスト	206
21.2.1.2. 単に jndi.properties ファイルに指定するのではなく、プログラムを使用して行う理由	207
21.2.1.3. 正しくない HA-JNDI ヘバインドされたことを知る方法	207
21.2.2. Enterprise Application Platform 外部で実行されているクライアントの場合	207
21.3. JBOSS の設定	209
21.3.1. 2 つ目の HA-JNDI サービスの追加	212
<b>第22章 クラスター化されたセッション EJB</b>	<b>214</b>
22.1. EJB 3.0 のステートレスセッション BEAN	214
22.2. EJB 3.0 でのステートフルセッション BEAN	215
22.2.1. EJB アプリケーション設定	215
22.2.2. ステートレプリケーションの最適化	217
22.2.3. CacheManager サービスの設定	217
22.3. EJB 2.X のステートレスセッション BEAN	220
22.4. EJB 2.X のステートフルセッション BEAN	221
22.4.1. EJB アプリケーション設定	221
22.4.2. ステートレプリケーションの最適化	222
22.4.3. The HASessionStateService サービスの設定	222
22.4.4. クラスター再起動の処理	223
22.4.5. JNDI ルックアッププロセス	224
22.4.6. SingleRetryInterceptor	224
<b>第23章 クラスター化した ENTITY EJB</b>	<b>225</b>
23.1. EJB 3.0 内の エンティティ BEAN	225
23.1.1. 分散型キャッシュの設定	225
23.1.2. キャッシュ用の エンティティ bean の設定	228
23.1.3. クエリ結果のキャッシュ	230
23.2. EJB 2.X 内の ENTITY BEAN	234
<b>第24章 HTTP サービス</b>	<b>235</b>
<b>第25章 JBOSS MESSAGING のクラスタリングに関する注意</b>	<b>236</b>
<b>第26章 クラスター化されたデプロイメントのオプション</b>	<b>237</b>
26.1. クラスター化シングルトンサービス	237
26.1.1. HASingleton デプロイメントオプション	238
26.1.1.1. HASingletonDeployer サービス	238
26.1.1.2. HASingletonController を使用した POJO デプロイメント	239
26.1.1.3. バリアを使用した HASingleton デプロイメント	240
26.1.2. マスターノードの決定	241
26.1.2.1. HA シングルトンの選出ポリシー	242
26.2. ファーミングデプロイメント	242
<b>第27章 JGROUPS サービス</b>	<b>245</b>

27.1. JGROUPS チャンネルのプロトコルスタックを設定	245
27.1.1. 一般的な設定プロパティ	247
27.1.2. トランスポートプロトコル	248
27.1.2.1. UDP 設定	248
27.1.2.2. TCP 設定	251
27.1.2.3. TUNNEL の設定	252
27.1.3. ディスカバリプロトコル	253
27.1.3.1. PING	253
27.1.3.2. TCPGOSSIP	254
27.1.3.3. TCPPING	254
27.1.3.4. MPING	255
27.1.4. 障害検出プロトコル	256
27.1.4.1. FD	256
27.1.4.2. FD_SOCKET	256
27.1.4.3. VERIFY_SUSPECT	257
27.1.4.4. FD と FD_SOCKET	257
27.1.5. 信頼できる配信プロトコル	258
27.1.5.1. UNICAST	258
27.1.5.2. NAKACK	259
27.1.6. グループメンバーシップ (GMS)	259
27.1.7. フロー制御 (FC)	260
27.2. 断片化 (FRAG2)	262
27.3. 状態の転送	262
27.4. 分散ガベージコレクション (STABLE)	263
27.5. マージ (MERGE2)	263
27.6. その他設定の問題	264
27.6.1. JGroups チャンネルの特定インターフェースへのバインド	264
27.6.2. JGroups チャンネルの分離	265
27.6.2.1. アプリケーションサーバーインスタンス同士を分離	265
27.6.2.2. 同じ AS インスタンスセット上の他のサービスに対してチャンネルを分離	266
27.6.2.2.1. グループ名の変更	266
27.6.2.2.2. マルチキャストアドレスとポートの変更	266
27.6.2.2.3. マルチキャストポートの変更	267
27.6.2.3. OS UDP のバッファ制限を設定してUDPのパフォーマンスを改善	268
27.6.3. JGroups のトラブルシューティング	269
27.6.3.1. ノードがクラスターを形成しない	269
27.6.3.2. FD でハートビートが不明な原因	269
<b>第28章 JBOSS CACHE の設定とデプロイメント</b>	<b>271</b>
28.1. 主な JBOSS CACHE 設定オプション	271
28.1.1. CacheManager 設定の編集	271
28.1.2. キャッシュモード	276
28.1.3. トランザクションの処理	277
28.1.4. 並行アクセス	277
28.1.5. JGroups の統合	278
28.1.6. エビクション	279
28.1.7. キャッシュローダー	279
28.1.7.1. Web セッションキャッシュと SFSB キャッシュの CacheLoader 設定	280
28.1.8. バディレプリケーション	281
28.2. 独自の JBOSS CACHE インスタンスをデプロイ	283
28.2.1. CacheManager サービスによるデプロイメント	283
28.2.1.1. CacheManager へのアクセス	283
28.2.2. -service.xml ファイルからのデプロイメント	285

28.2.3. -jboss-beans.xml ファイルからのデプロイメント	286
<b>パート IV. レガシーの EJB サポート</b>	<b>289</b>
<b>第29章 JBOSS 上の EJB</b>	<b>290</b>
EJB コンテナ設定とアーキテクチャー	290
29.1. EJB クライアント側のビュー	290
29.1.1. EJB プロキシ設定の指定	293
29.2. EJB サーバー側のビュー	297
29.2.1. 分離呼び出し：トランスポートの仲介	298
29.2.2. HA JRMPInvoker - クラスターされた RMI/JRMP トランスポート	301
29.2.3. HA HttpInvoker - クラスターされた RMI/HTTP トランスポート	302
29.3. EJB コンテナ	303
29.3.1. EJBDeployer MBean	303
29.3.1.1. EJB デプロイメントの検証	304
29.3.1.2. EJB をコンテナにデプロイ	305
29.3.1.3. コンテナ設定情報	305
29.3.1.3.1. container-name 要素	309
29.3.1.3.2. call-logging 要素	309
29.3.1.3.3. invoker-proxy-binding-name 要素	310
29.3.1.3.4. sync-on-commit-only 要素	310
29.3.1.3.5. insert-after-ejb-post-create	310
29.3.1.3.6. call-ejb-store-on-clean	310
29.3.1.3.7. container-interceptors 要素	310
29.3.1.3.8. instance-pool 要素	310
29.3.1.3.9. container-pool-conf 要素	310
29.3.1.3.10. instance-cache 要素	311
29.3.1.3.11. container-cache-conf 要素	311
29.3.1.3.12. persistence-manager 要素	313
29.3.1.3.13. web-class-loader 要素	313
29.3.1.3.14. locking-policy 要素	314
29.3.1.3.15. commit-option と optiond-refresh-rate 要素	314
29.3.1.3.16. security-domain 要素	315
29.3.1.3.17. cluster-config	315
29.3.1.3.18. depends 要素	316
29.3.2. コンテナプラグインのフレームワーク	316
29.3.2.1. org.jboss.ejb.ContainerPlugin	317
29.3.2.2. org.jboss.ejb.Interceptor	317
29.3.2.3. org.jboss.ejb.InstancePool	318
29.3.2.4. org.jboss.ejb.InstanceCache	319
29.3.2.5. org.jboss.ejb.EntityPersistenceManager	320
29.3.2.6. org.jboss.ejb.EntityPersistenceStore インターフェース	324
29.3.2.7. org.jboss.ejb.StatefulSessionPersistenceManager	327
29.4. エンティティ BEAN のロックとデッドロックの検出	328
29.4.1. JBoss にロック機能が必要な理由	328
29.4.2. エンティティ Bean のライフサイクル	328
29.4.3. デフォルトのロッキング動作	329
29.4.4. プラグ可能なインターセプターとロックポリシー	329
29.4.5. デッドロック	330
29.4.5.1. デッドロックの検出	331
29.4.5.2. ApplicationDeadlockException のキャッチ	331
29.4.5.3. ロック情報の表示	332
29.4.6. 詳細設定と最適化	332

29.4.6.1. 存続期間の短いトランザクション	333
29.4.6.2. アクセスの順序付け	333
29.4.6.3. 読み取り専用 bean	333
29.4.6.4. Read-Only メソッドを明示的に定義	333
29.4.6.5. Instance Per Transaction ポリシー	334
29.4.7. クラスター内での実行	335
29.4.8. トラブルシューティング	335
29.4.8.1. ロッキング動作が機能しない場合	335
29.4.8.2. IllegalStateException	335
29.4.8.3. ハングとトランザクションタイムアウト	335
29.5. EJB タイマー設定	336
<b>第30章 CMP エンジン</b>	<b>338</b>
30.1. サンプルコード	338
30.1.1. CMP デバッグロギングの有効化	339
30.1.2. サンプルの実行	340
30.2. JBOSSCMP-JDBC の構造	341
30.3. エンティティ BEAN	343
30.3.1. エンティティマッピング	345
30.4. CMP フィールド	350
30.4.1. CMP フィールド宣言	350
30.4.2. CMP フィールドのカラムマッピング	350
30.4.3. Read-only フィールド	352
30.4.4. エンティティアクセスの監査	353
30.4.5. 依存値クラス(DVC: Dependent Value Class)	354
30.5. コンテナ管理リレーション	358
30.5.1. CMR-Field 抽象アクセッサー	358
30.5.2. リレーションシップの宣言	359
30.5.3. 関係マッピング	360
30.5.3.1. リレーションシップロールマッピング	362
30.5.3.2. 外部キーマッピング	364
30.5.3.3. 関係テーブルのマッピング	365
30.6. クエリ	366
30.6.1. Finder と Select 宣言	366
30.6.2. EJB-QL 宣言	367
30.6.3. EJB-QL を SQL マッピングにオーバーライド	368
30.6.4. JBossQL	369
30.6.5. DynamicQL	371
30.6.6. DeclaredSQL	372
30.6.6.1. パラメーター	375
30.6.7. EJBQL 2.1 と SQL92 のクエリ	375
30.6.8. BMP カスタムファインダー	376
30.7. 最適化ローディング	377
30.7.1. ローディングのシナリオ	377
30.7.2. Load Groups	379
30.7.3. Read-ahead	379
30.7.3.1. on-find	379
30.7.3.1.1. Left join read ahead	381
30.7.3.1.2. D#findByPrimaryKey	381
30.7.3.1.3. D#findAll	382
30.7.3.1.4. A#findAll	382
30.7.3.1.5. A#findMeParentGrandParent	384
30.7.3.2. on-load	384

30.7.3.3. none	386
30.8. ローディングプロセス	387
30.8.1. コミットオプション	387
30.8.2. 一括読み込みプロセス	387
30.8.3. 遅延ローディングプロセス	389
30.8.3.1. リレーションシップ	389
30.8.4. 遅延ローディングの結果セット	392
30.9. トランザクション	393
30.10. 楽観的ロッキング	395
30.11. エンティティコマンドおよびプライマリキー生成	399
30.11.1. 既存のエンティティコマンド	400
30.12. デフォルト	402
30.12.1. jbosscomp-jdbc.xml defaults 宣言例	404
30.13. データソースのカスタマイズ	406
30.13.1. タイプマッピング	406
30.13.2. 関数マッピング	409
30.13.3. マッピング	409
30.13.4. ユーザータイプマッピング	410
<b>パート IV. パフォーマンスチューニング</b>	<b>412</b>
<b>第31章 JBOSS ENTERPRISE APPLICATION PLATFORM 5 のパフォーマンスチューニング</b>	<b>413</b>
31.1. はじめに	413
31.2. ハードウェアチューニング	413
31.2.1. CPU (Central Processing Unit: 中央処理装置)	413
31.2.2. RAM (Random Access Memory: ランダムアクセスメモリ)	414
31.2.3. ハードディスク	414
31.3. オペレーティングシステムのパフォーマンスチューニング	414
31.3.1. ネットワーキング	415
31.4. JVM のチューニング	415
31.5. アプリケーションのチューニング	415
31.5.1. インストルメンテーション	416
31.6. JBOSS ENTERPRISE APPLICATION PLATFORM のチューニング	416
31.6.1. メモリ使用率	416
31.6.1.1. VFS チューニング	417
31.6.1.1.1. VFS キャッシュのチューニング	418
31.6.1.1.2. アノテーションスキンのチューニング	419
31.6.2. データベース接続	420
31.6.3. クラスタリングのチューニング	421
31.6.3.1. 十分なネットワークバッファの確保	421
31.6.3.2. クラスター間トラフィックの分離	421
31.6.3.3. JGroups のメッセージバンドリング	421
31.6.3.4. セッションキャッシュのパディレプリケーションを有効化	422
31.6.3.5. Web セッションレプリケーションのボリュームを低減	423
31.6.3.6. EJB3 ステートフルセッション Bean レプリケーションのボリュームを低減	423
31.6.3.7. JPA/Hibernate の 2 次キャッシングの注意	423
31.6.3.8. JMX による JGroups の監視	424
31.6.4. その他の主な設定	425
<b>パート VI. 添付資料</b>	<b>426</b>
<b>付録A ベンダー固有のデータソース定義</b>	<b>427</b>
A.1. デプロイヤーの場所と名前	427
A.2. DB2	427

---

A.3. ORACLE	431
A.3.1. Oracle 10g JDBC ドライバーの変更	435
A.3.2. Oracle 10g に対するタイプマッピング	435
A.3.3. 基盤の Oracle Connection オブジェクトをリトリート	435
A.3.4. Oracle 11g の制限	435
A.4. SYBASE	435
A.4.1. Sybase の制限	436
A.5. MICROSOFT SQL SERVER	437
A.5.1. Microsoft JDBC ドライバー	439
A.5.2. JSQL ドライバー	441
A.5.3. jTDS JDBC ドライバー	442
A.5.4. "Invalid object name 'JMS_SUBSCRIPTIONS' Exception	444
A.6. MYSQL DATASOURCE	444
A.6.1. ドライバーのインストール	444
A.6.2. MySQL Local-TX Datasource	445
A.6.3. 名前付きパイプを利用した MySQL	445
A.7. POSTGRESQL	446
A.8. INGRES	448
<b>付録B ログギング情報とレシピ</b> .....	<b>449</b>
B.1. ログレベルの説明	449
B.2. アプリケーション別にログファイルを分類	450
B.3. カテゴリ出力のリダイレクト	451
<b>付録C 改訂履歴</b> .....	<b>452</b>

## 本書の範囲

本書の第一の目的は、設定とアーキテクチャーの両方の観点より標準の JBoss Enterprise Application Platform 5.0 のアーキテクチャーコンポーネントについて説明することです。標準の JBoss ディストリビューションユーザーは、標準コンポーネントの設定方法を理解できるようになるでしょう。本書は JavaEE の紹介やアプリケーションで JavaEE を使用する方法について書かれたものではなく、JBoss サーバーアーキテクチャー内部の詳細や、JavaEE コンテナ実装の設定方法や拡張方法に焦点が置かれています。

JBoss の開発者として、標準コンポーネントの統合やアーキテクチャーを十分理解でき、ご利用のインフラストラクチャーで必要な標準コンポーネントを拡張、又は入れ替えることができるようになるでしょう。また、このマニュアルでは、JBoss サーバーの構築とデバッグの方法と共に JBoss ソースコードの取得の方法も案内しています。



## 第1章 はじめに

JBoss Enterprise Application Platform 5 は新しい JBoss Microcontainer 上に構築されています。JBoss Microcontainer は、POJO (Plain Old Java Object) の直接デプロイや設定、ライフサイクルをサポートする軽量コンテナです。このスタンドアローンの JBoss Microcontainer プロジェクトは、JBoss Enterprise Application Platform 4.x で使用された JBoss JMX Microkernel を置き換えるものとなっています。

JBoss Microcontainer は JBoss Aspect Oriented Programming フレームワーク (JBoss AOP) とスムーズに統合することができます。JBoss AOP については [10章JBoss AOP](#) に説明されています。JBoss Enterprise Application Platform 5 でも JMX に対するサポートは継続されるため、従来の Microkernel 向けに書かれた MBean サービスも動作するはずです。

JBoss Enterprise Application Platform 5.0.0.GA 以降のバージョン上で実行可能で、興味深い技術を多数保持するサンプル Java EE 5 アプリケーションが本ディストリビューションで提供される Seam Booking Application となっています。このサンプルアプリケーションは、JBoss Enterprise Application Platform 5 上で稼働する以下の技術を利用しています。

- EJB3
- ステートフルセッション Bean
- ステートレスセッション Bean
- JPA (Hibernate 検証あり)
- JSF
- Facelets
- Ajax4JSF
- Seam

JBoss Enterprise Application Platform 5 にある主要機能の多くは、次のようなスタンドアロン JBoss プロジェクトを統合して提供されます。

- JBoss Enterprise Application Platform 5 に含まれる JBoss EJB3 は、Enterprise Java Beans (EJB) 仕様の最新バージョンを実装します。EJB 3.0 は EJB 仕様が大幅に改良、簡素化されています。EJB 3.0 の目標は、開発の簡素化やテスト駆動型アプローチの推進を図るとともに、複雑な EJB API に対するコード化ではなく POJO (Plain Old Java Object) の書き込みに焦点を置いています。
- JBoss Messaging は、JBoss Enterprise Application Platform 5 のデフォルトのメッセージプロバイダーとして提供される高パフォーマンス JMS プロバイダーです。JBoss ESB インフラストラクチャーのバックボーンでもあります。JBoss Messaging は、JBoss Enterprise Application Platform 4.2 のデフォルトの JMS プロバイダーである JBossMQ を完全に書き直したものです。
- JBoss Cache は、従来のツリー構造のノードベースキャッシュと、PojoCache の2種類のキャッシュを提供します。PojoCache はインメモリでレプリケーション型のトランザクションキャッシュシステムで、レプリケーションや永続性アスペクトのユーザー管理をアクティブに行わなくてもユーザーが簡単な POJO を透過的に操作できるようにします。
- JBossWS 3.x は、Java EE 互換 Web サービス JAXWS-2.x を提供する JBoss Enterprise Application Platform 5 の Web サービススタックです。

- JBoss Transactions は、JBoss Enterprise Application Platform 5 のデフォルトのトランザクションマネージャーです。JBoss Transactions は業界で定評のある技術を基盤とし、分散トランザクションのリーダーとして 18 年の実績を持っています。JBoss Transactions は最も相互操作性に優れた実装の 1 つです。
- JBoss Web は JBoss Enterprise Application Platform 5 の Web コンテナで、APR (Apache Portable Runtime) と Tomcat ネイティブ技術を含む Apache Tomcat を基盤とした実装となっており、Apache Http サーバー以上のパフォーマンス特性やスケーラビリティを実現します。

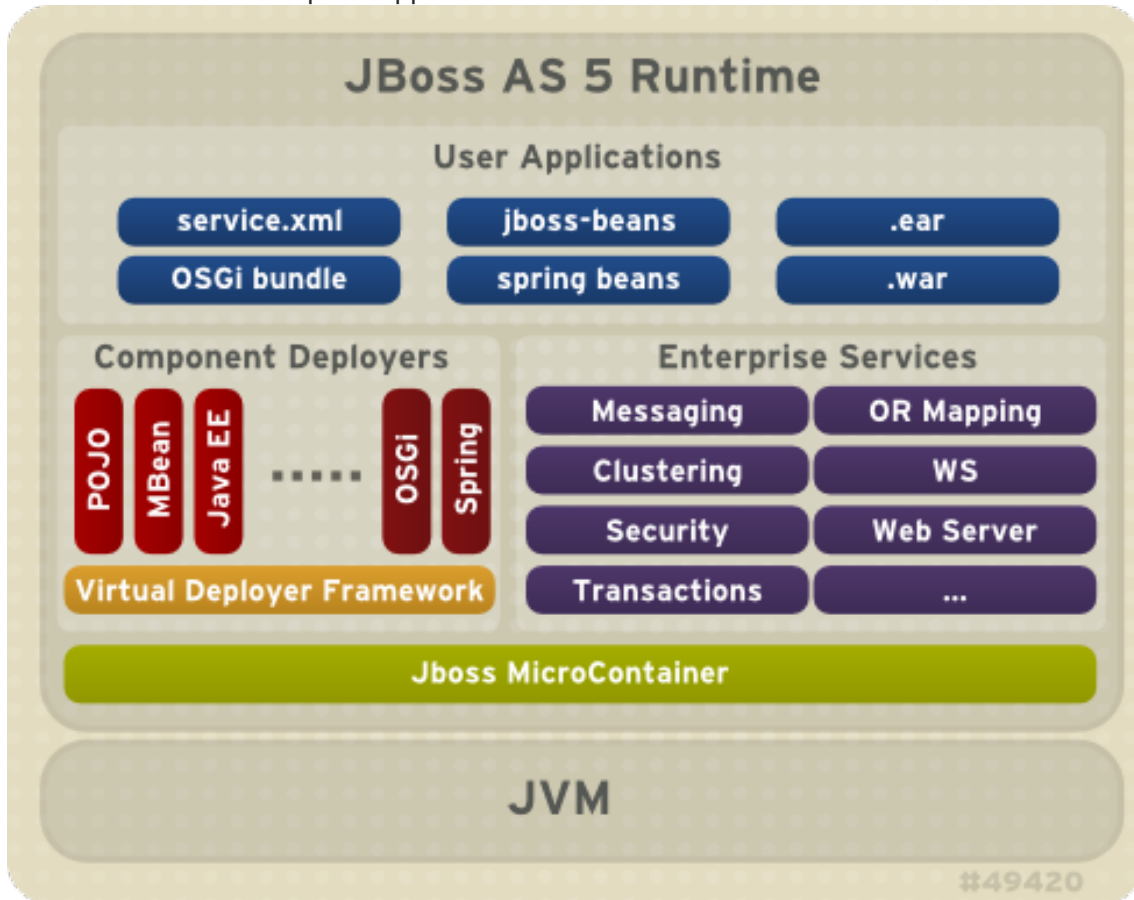
## 1.1. JBOSS ENTERPRISE APPLICATION PLATFORM ユースケース

- 99% の Web アプリケーションがデータベースを活用している場合
- クラスタ化される可能性の高いミッションクリティカルな Web アプリケーション
- JSP/サーブレットを持つ簡単な Web アプリケーションが Tomcat 組み込みの JBoss Enterprise Application Platform にアップグレードされる場合
- JSP/サーブレットを持ち、Struts、Java Server Faces、Cocoon、Tapestry、Spring、Expresso、Avalon、Turbine などの Web フレームワークを使用する中級ウェブアプリケーション。
- JSP/サーブレット、SEAM、EJB (Enterprise Java Bean)、JMS (Java Messaging)、キャッシングなどを持つ複雑な Web アプリケーション
- アプリケーション間共通のミドルウェア (JMS、Corba、JMX など)

## パート I. JBOSS ENTERPRISE APPLICATION PLATFORM イン フラストラクチャー

## 第2章 JBOSS ENTERPRISE APPLICATION PLATFORM 5 アーキテクチャー

次の図は、JBoss Enterprise Application Server とそのコンポーネントの概略を表しています。



JBoss Enterprise Application Platform 5 のディレクトリ構造は 4.x シリーズと似ていますが、明らかな違いがいくつかあります。

```
-jboss-as - the path to your JBoss Enterprise Application Server.
|-- bin - contains start scripts and run.jar
|-- client - client jars
|-- common/lib - static jars shared across server profile
|-- docs - schemas/dtds, examples
|-- lib - core bootstrap jars
|   lib/endorsed - added to the server JVM java.endorsed.dirs path
|-- server - server profile directories. See Section 3.2
               for details of the server profiles included in this
release.
```

```
-seam - the path to JBoss SEAM application framework
|-- bootstrap
|-- build
|-- examples - examples demonstrating uses of SEAM's features
|-- extras
|-- lib - library directory
|-- seam-gen - command-line utility used to generate simple skeletal
SEAM code to get your project started
|-- ui -
```

```
-resteasy - RESTEasy - a portable implementation of JSR-311 JAX-RS
Specification
|-- embedded-lib
|-- lib
|-- resteasy-jaxrs.war
```

## 2.1. JBOSS ENTERPRISE APPLICATION PLATFORM のブートストラップ

JBoss Enterprise Application Platform 5 のブートストラップは `org.jboss` の JBoss Enterprise Application Platform 4.x バージョンと似ています。メインエントリーポイントが `org.jboss.system.server.Server` 実装をロードします。JBoss Enterprise Application Platform 4.x では JMX ベースのマイクロカーネルでしたが、JBoss Enterprise Application Platform 5 では JBoss マイクロコンテナになりました。

JBoss Enterprise Application Platform 5 のデフォルトの `org.jboss.system.server.Server implementation` は `org.jboss.bootstrap.microcontainer.ServerImpl` です。この実装はカーネル基本ブートストラップの拡張で、`BasicXMLDeployer` を使用して `{jboss.server.config.url}/bootstrap.xml` 記述子にて宣言されたブートストラップ bean から MC をブートします。また、`ServerImpl` レジスターは、`org.jboss.bootstrap.spi.Bootstrap` インターフェースを実装する Bean のコールバックをインストールします。`bootstrap/profile*.xml` 設定には、`Bootstrap` インターフェースを実装する `ProfileServiceBootstrap` Bean が含まれます。

`org.jboss.system.server.profileservice.ProfileServiceBootstrap` は、`org.jboss.bootstrap.spi.Bootstrap` インターフェースの実装で現在のプロファイルに紐付けられたデプロイメントをロードします。`PROFILE` はロードされたプロファイルの名前で、`server -c` コマンドライン引数に対応します。デフォルトの `PROFILE` は `default` です。

## 2.2. ホットデプロイメント

JBoss Enterprise Application Platform 5 のホットデプロイメントは、`ProfileService` に関連する `Profile` 実装によって制御されます。`deploy/hdscanner-jboss-beans.xml` よりデプロイされた `HDScanner` Bean はアプリケーションディレクトリ内容の変更についてプロファイルサービスをクエリし、更新された内容を再デプロイします。その後、削除された内容をアンデプロイし、`ProfileService` より新しいデプロイメント内容を現在のプロフィールへ追加します。

ホットデプロイメントを一時的、あるいは永続的に無効にしたい場合は、以下のいずれかのメソッドを利用してください。2 番目のメソッドは最も簡単に元に戻すことができるため、一時的にホットデプロイメントを無効にしたい場合は、この 2 番目のメソッドの利用が最適です。

- デプロイメントから `hdscanner-jboss-beans.xml` ファイルを削除します。
- `hdscanner-jboss-beans.xml` ファイルを編集し、`scanEnabled` 属性を `false` へ変更します。

以下に、ホットデプロイメントを無効にした `hdscanner-jboss-beans.xml` ファイルを抜粋しています。

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Hot deployment scanning
```

```
$Id: hdscanner-jboss-beans.xml 98983 2010-01-04 13:35:41Z
emuckenhuber $
-->
<deployment xmlns="urn:jboss:bean-deployer:2.0">

    <!-- Hotdeployment of applications -->
    <bean name="HDScanner"
class="org.jboss.system.server.profileservice.hotdeploy.HDScanner">
        <property name="deployer"><inject
bean="ProfileServiceDeployer"/></property>
        <property name="profileService"><inject
bean="ProfileService"/></property>
        <property name="scanPeriod">5000</property>
        <property name="scanThreadName">HDScanner</property>
        <property name="scanEnabled">false</property>
    </bean>

    ...(snip)...

</deployment>
```

## パート II. JBOSS ENTERPRISE APPLICATION PLATFORM 5 の 設定

## 第3章 ネットワーク

デフォルトでは、Enterprise Platform は起動時にネットワークアドレスすべてにバインドします。起動時にUDPアドレスだけでなくバインドアドレスを指定することができます。

表3.1 ネットワーク関連のスタートアップオプション

<b>-b [IP-ADDRESS]</b>	アプリケーションサーバーのバインド先のアドレスを指定します。指定されていない場合は、アプリケーションサーバーは全アドレスにバインドします。
<b>-u [IP-ADDRESS]</b>	UDPマルチキャストアドレス（オプション）。指定されていない場合はTCPを利用します。
<b>-m [MULTICAST_PORT_ADDRESSES]</b>	UDPマルチキャストポート。JGroups のみが使用します。

### 3.1. IPV6 のサポート

今後対応される予定ではありますが、Enterprise Application Platform 5 では IPv6 には対応していません。



## 第4章 EJB3 サービスと ENTERPRISE APPLICATION

EJB3 (Enterprise Java Bean 3.0) は、Java EE 5 アプリケーションのコアとなるコンポーネントモデルを提供します。EJB3 bean は、管理コンポーネントでトランザクション、セキュリティ、永続性、ネーミング、依存性注入など Java EE 5 サーバーコンテナが提供するサービスすべてを有効活用できるよう自動的にワイヤーされている管理コンポーネントです。この管理コンポーネントにより、開発者はビジネスロジックに集中し、横断的関心事 (cross-cutting concern) はコンテナに設定として任せることができます。アプリケーション開発者として、コンポーネントの作成や破棄はご自身で行う必要があります。Java EE コンテナから EJB3 bean の名前のみで EJB3 bean を呼び出しでき、適用した設定済みのコンテナサービスすべてを使いメソッドを呼び出すことができます。Java EE コンテナの内部、外部を問わず EJB3 bean へアクセスできます。

JBoss Enterprise Application Platform 5 はカスタマイズなしに EJB3 に対応しています。JBoss Enterprise Application Platform 4.2 は J2EE サーバーであるため、EJB3 機能セットすべてに対応していない点に注意してください。

EJB3 コンポーネントのプログラミングモデルに関する詳細は、本書の対応範囲を超えています。EJB3 インターフェースとアノテーションのほとんどが Java EE 5 規格の一部であるため、Java EE 5 準拠の全アプリケーションサーバーと同じです。興味のある方は EJB3 仕様か、EJB3 関連の本が多数ありますのでそちらを参照し、EJB3 プログラミングについて学習してみてください。

本章では、JBoss Enterprise Application Platform 固有となる、EJB3 設定の問題について見ていきます。例えば、JBoss Enterprise Application Platform における EJB3 コンポーネントの JNDI 命名規則、エンティティ bean 向けの Hibernate 永続エンジンのオプション設定、JBoss EJB3 デプロイヤーのカスタムオプションについて触れています。

### 4.1. セッション BEAN

セッション bean は、ローカルおよびリモートクライアントにトランザクションサービスを提供する際に幅広く利用されています。セッション bean を記述するには、インターフェースと実装クラスが必要になります。

```
@Local
public interface MyBeanInt {
    public String doSomething (String para1, int para2);
}

@Stateless
public class MyBean implements MyBeanInt {

    public String doSomething (String para1, int para2) {
        ... implement the logic ...
    }

}
```

セッション bean メソッドを呼び出すと、サーバーにあるセキュリティーマネージャーとトランザクションマネージャーがメソッド実行を自動的に管理します。メソッド上のアノテーションを使うことで、メソッド毎にトランザクションあるいはセキュリティープロパティを指定することができます。多くのクライアントで、セッション bean インスタンスを再利用することができます。サーバーが2つのクライアント間で bean の内部ステータスを管理するかどうかにより、セッション bean はステートレスかステートフルか決まります。Bean にリモートビジネスインターフェースがある場合、現在の JVM の

外にあるクライアントは、EJB3 bean を呼び出すことができます。これらはすべて bean 上の標準のアノテーションを使い設定可能です。トランザクションあるいはセキュリティプロパティは、Bean がビジネスインターフェースから呼び出された場合のみ有効である点に注意してください。

セッション bean を定義後、クライアントはどのようにそのセッション bean への参照を取得するのでしょうか？前述したとおり、クライアントは、EJB3 コンポーネントを作成あるいは破棄しないため、サーバーが管理している既存のインスタンスへの参照をサーバーに要求するだけです。これは、JNDI 経由でできます。JBoss AS では、セッション bean に対するデフォルトのローカル JNDI 名は bean クラスのデプロイメントパッケージにより変わります。

- Bean が **JBoss\_DIST/default/deploy** にあるスタンドアローンの JAR ファイルにデプロイされた場合、Bean はローカルの JNDI 名 **MyBean/local** 経由でアクセス可能です。ここでは、先ほど示したように、**MyBean** は bean の実装クラス名です。JBoss AS では Local の JNDI は、**java:comp/env/** を起点とする JNDI 名となります。
- Bean を含む JAR ファイルが EAR ファイルにパッケージされている場合、Bean に対するローカルの JNDI 名は **myapp/MyBean/local** です。この **myapp** は EAR アーカイブファイルのルート名となっています (例: **myapp.ear**。EJB3 bean の EAR パッケージについては本書で後述されていますので参照してください)。

Bean インタフェースに **@Remote** のアノテーションがついており、デプロイされているサーバーの外から Bean にアクセスする場合は、もちろん、**local** を **remote** に変更すべきです。以下は、**myapp.ear** にパッケージされた Web アプリケーションで MBean bean の参照を取得し、管理メソッドを呼び出すためのコードスニペットです (例: サーブレットあるいは JSF バッキング bean)。

```
try {
    InitialContext ctx = new InitialContext();
    MyBeanInt bean = (MyBeanInt) ctx.lookup("myapp/MyBean/local");
} catch (Exception e) {
    e.printStackTrace ();
}

... ..

String result = bean.doSomething("have fun", 1);

... ..
```

クライアントが JNDI から取得したものは必然的に、Bean インスタンスの「スタブ」あるいは「プロキシ」となります。クライアントがメソッドを呼び出すと、プロキシは、リクエストをサーバーまでどのようにルーティングし、レスポンスをあわせてマーシャリングするか見出します。

デフォルトの JNDI 名が気に入らない場合は、いつでも bean 実装クラスの **@LocalBinding** アノテーションを使い Bean に独自の JNDI バインディングを指定することが可能です。JNDI バインディングは常に、**java:comp/env/** スペースの下では Local となります。例えば、以下の bean クラス定義は、**java:comp/env/MyService/MyOwnName** という JNDI 名の下で利用可能な bean インスタンスが返ってきます。

```
@Stateless
@LocalBinding (jndiBinding="MyService/MyOwnName")
public class MyBean implements MyBeanInt {
```

```

    public String doSomething (String para1, int para2) {
        ... implement the logic ...
    }
}

```

### 注記

Java EE 5 は、明示的に JNDI ルックアップを行わずにアノテーションを使うことで、EJB3 bean インスタンスを直接、Web アプリケーションに注入できるようになります。この動作は、JBoss Enterprise Application Platform 4.2 では未対応ですが、JBoss Enterprise Platform は、JBoss Seam と呼ばれる統合フレームワークを提供しています。JBoss Seam では、Java EE 5 が提供しているものをはるかに超える新たなレベルで、EJB3 / JSF 統合が行えます。詳細は、本プラットフォームに同梱されている JBoss Seam のリファレンスガイドを参照してください。

## 4.2. エンティティ BEAN (JAVA PERSISTENCE API)

EJB3 セッション bean により、データを実装しトランザクションメソッドでビジネスロジックにアクセスできるようになります。実際データベースにアクセスするには、EJB3 エンティティ bean とエンティティマネージャー API が必要です。これらはまとめて、Java Persistence API (JPA) と呼ばれています。

EJB3 エンティティ Bean は Plain Old Java Object (POJO) で、関係データベースのテーブルにマッピングします。例えば、以下のエンティティ bean クラスは、Customer という名前の関係テーブルにマッピングします。このテーブルには、name、age、signupdate の 3 つカラムがあり、bean の各インスタンスは、テーブルの各行に対応します。

```

@Entity
public class Customer {

    String name;

    public String getName () {
        return name;
    }

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

    int age;

    public int getAge () {
        return age;
    }

    public void setAge (int age) {
        this.age = age;
    }

    Date signupdate;
}

```

```

    public Date getSignupdate () {
        return signupdate;
    }

    public void setSignupdate (Date signupdate) {
        this.signupdate = signupdate;
    }
}

```

単純なデータプロパティ以外に、エンティティ bean には@OneToMany、@ManyToOne、@ManyToMany などの関係マッピングアノテーションを持つ、その他のエンティティ bean への参照も含むことができます。これらのエンティティオブジェクトの関係は、自動的に外部キーとしてデータベース内に設定されます。例えば、以下の例では Customer テーブル内の各レコードには該当するレコードが Account テーブルに 1 つ、Order テーブルに複数あり、Employee テーブルの各レコードには、該当するレコードが Customer テーブルに複数存在します。

```

@Entity
public class Customer {

    ... ..

    Account account;

    @OneToOne
    public Account getAccount () {
        return account;
    }

    public void setAccount (Account account) {
        this.account = account;
    }

    Employee salesRep;

    @ManyToOne
    public Employee getSalesRep () {
        return salesRep;
    }

    public void setSalesRep (Employee salesRep) {
        this.salesRep = salesRep;
    }

    Vector <Order> orders;

    @OneToMany
    public Vector <Order> getOrders () {
        return orders;
    }

    public void setOrders (Vector <Order> orders) {
        this.orders = orders;
    }
}

```

}

EntityManager API を使い、エンティティ bean の作成、更新、削除、クエリを行うことができます。EntityManager はプロセス中に基盤のデータベーステーブルを透過的に更新します。

@PersistenceContext アノテーションを使い EJB セッション bean で EntityManager オブジェクトを取得することができます。

```
@PersistenceContext
EntityManager em;

Customer customer = new Customer ();
// populate data in customer

// Save the newly created customer object to DB
em.persist (customer);

// Increase age by 1 and auto save to database
customer.setAge (customer.getAge() + 1);

// delete the customer and its related objects from the DB
em.remove (customer);

// Get all customer records with age > 30 from the DB
List <Customer> customers = em.query (
    "select c from Customer where c.age > 30");
```

EntityManager API の詳細用途に関しては、本書の対象外となっています。興味のある方は、JPA 文書あるいは Hibernate EntityManager の文書を参照してください。

#### 4.2.1. persistence.xml ファイル

EntityManager API は優れていますが、サーバーはエンティティオブジェクトを保存、更新、クエリすべきデータベースがどれなのか、どのように把握するのでしょうか？性能や問題解決を向上できるように、基盤となる object-relational-mapping エンジンやキャッシュをどのように設定したらいいでしょう。persistence.xml ファイルで、EntityManager を自由自在に設定することができます。

persistence.xml ファイルは、JPA の標準設定ファイルです。このファイルは、エンティティ bean を含む JAR ファイルの META-INF ディレクトリに入れる必要があります。persistence.xml ファイルは、persistence-unit をスコープ化された現在の classloader 内に一意の名前で定義する必要があります。プロバイダー属性は、基盤となる JPA EntityManager 実装を指定します。JBoss Enterprise Application Platform では、デフォルトの JPA プロバイダーは、Hibernate で唯一対応で推奨しているプロバイダーです。jta-data-source は、この永続ユニットのマッピング先のデータベースの JNDI 名を参照します。ここでは、java:/DefaultDS は、JBoss Enterprise Application Platform に埋め込まれている HSQL DB を参照しています。JBoss AS 用に別のデータベースを設定する方法は、[14章 JBoss Enterprise Application Platform で別のデータベースを使用](#) を参照してください。

```
<persistence>
  <persistence-unit name="myapp">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
```

```

    <properties>
        ...
    </properties>
</persistence-unit>
</persistence>

```

### 注記

永続ユニットのインスタンスが複数、同じアプリケーションで定義されている可能性があるため、通常 `@PersistenceContext` アノテーションに対して明示的にどのユニットを注入したいかを指示する必要があります。例えば、`@PersistenceContext(name="myapp")` は、`myapp` という名前の永続ユニットから `EntityManager` を注入します。

しかし、スコープ設定された独自のクラスローダーに EAR アプリケーションをデプロイし、全アプリケーションで1つの永続ユニットのみ定義されている場合、`@PersistenceContext` の `"name"` を省略可能です。EAR パッケージとデプロイメントについては、本章で後述していますのでそちらを参照してください。

`persistence.xml` のプロパティ要素には、基盤永続プロバイダー向けの設定プロパティを含めることができます。JBoss Enterprise Application Platform は Hibernate を EJB3 永続プロバイダーとして利用するため、Hibernate オプションをここで渡すことができます。詳細については、Hibernate と Hibernate EntityManager 文書を参照してください。ここでは、永続エンジンの SQL 方言を HSQL に設定し、アプリケーション起動時にエンティティ bean からテーブルを作成し、アプリケーション停止時にこのテーブルを破棄するという例を提示しています。

```

<persistence>
  <persistence-unit name="myapp">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>

```

#### 4.2.2. 代わりのデータベースを利用

ビルトインされている HSQL DB 以外の別のデータベースを利用して、エンティティ bean をバックアップする場合、データベースのデータソースをまず定義し、JNDI にてそのデータソースを登録する必要があります。これには、`deploy` ディレクトリの `*-ds.xml` ファイルを使います。様々なデータベース向けの `*-ds.xml` ファイル例は、当サーバーの `JBOSS_DIST/docs/examples/jca` ディレクトリから入手できます。

次に、`persistence.xml` で、`jta-data-source` 属性を変更し、JNDI の新規データソースを参照する必要があります (例: デフォルトの `mysql-ds.xml` を使い、MySQL 外部データベースを設定する場合は `java:/MySQLDS`)。

ほとんどの場合、Hibernate は自動的に接続先のデータベースを検出試行した後、そのデータベースに

適した SQL を自動選択します。しかし、特に利用頻度の少ないデータベースサーバーなど、常に検出機能が動作するわけでない点がわかっています。そのため、`persistence.xml` で `hibernate.dialect` プロパティを明示的に設定するよう推奨しています。以下に JBoss プラットフォームで正式に対応しているデータベースサーバーの Hibernate 方言を示しています。

- Oracle 9i および 10g: `org.hibernate.dialect.Oracle9Dialect`
- Microsoft SQL Server 2005: `org.hibernate.dialect.SQLServerDialect`
- PostgreSQL 8.1: `org.hibernate.dialect.PostgreSQLDialect`
- MySQL 5.0: `org.hibernate.dialect.MySQL5Dialect`
- DB2 8.0: `org.hibernate.dialect.DB2Dialect`
- Sybase ASE 12.5: `org.hibernate.dialect.SybaseDialect`

### 4.2.3. デフォルトの Hibernate オプション

Hibernate には多数の設定プロパティがあります。`persistence.xml` ファイルで指定しないプロパティについては、JBoss AS が妥当なデフォルト値セットを提供します。デフォルトの Hibernate プロパティの値は、**JBOSSE\_DIST/server/default/deploy/ejb3.deployer/MEAT-INF/persistence.properties** ファイルで指定されます。以下は、JBoss AS 4.2 に同梱されている **persistence.properties** ファイルです。コメントアウトされたオプションに注意してください。このファイルでは、**persistence.xml** ファイルで利用可能なプロパティにどのようなものがあるか、簡単に提示しています。

```
hibernate.transaction.manager_lookup_class=org.hibernate.transaction.JBoss
TransactionManagerLookup
#hibernate.connection.release_mode=after_statement
#hibernate.transaction.flush_before_completion=false
#hibernate.transaction.auto_close_session=false
#hibernate.query.factory_class=org.hibernate.hql.ast.ASTQueryTranslatorFactory
#hibernate.hbm2ddl.auto=create-drop
#hibernate.hbm2ddl.auto=create
hibernate.cache.provider_class=org.hibernate.cache.HashtableCacheProvider
# Clustered cache with TreeCache
#hibernate.cache.provider_class=org.jboss.ejb3.entity.TreeCacheProviderHook
#hibernate.treecache.mbean.object_name=jboss.cache:service=EJB3EntityTreeCache
#hibernate.dialect=org.hibernate.dialect.HSQLDialect
hibernate.jndi.java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
hibernate.jndi.java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
hibernate.bytecode.use_reflection_optimizer=false
# I don't think this is honored, but EJB3Deployer uses it
hibernate.bytecode.provider=javassist
```

## 4.3. メッセージ駆動型 BEAN

メッセージ駆動型 bean は、特別な EJB3 bean で、スタブからのプロキシメソッドの呼び出しではなく JMS メッセージを使いサービスリクエストを受信します。そのため、メッセージ駆動型 bean の重要な設定パラメーターは、リッスンする JMS メッセージキューはどれか指定するためのものです。キューに受信メッセージがある場合、サーバーは bean の `onMessage()` メソッドを呼び出し、このメソッドをメッセージ内で渡し処理を行います。bean クラスは `@MessageDriven` アノテーションで JMS キューを指定します。このキューはローカルの JNDI `java:comp/env/` 名前空間に登録されます。

```
@MessageDriven(activationConfig =
{
    @ActivationConfigProperty(propertyName="destinationType",
        propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination",
        propertyValue="queue/MyQueue")
})
public class MyJmsBean implements MessageListener {

    public void onMessage (Message msg) {
        // ... do something with the msg ...
    }

    // ... ...
}
```

メッセージ駆動型 bean がデプロイされると、受信メッセージキューが存在しない場合は自動で作成されます。Bean にメッセージを送信するには、標準の JMS API を使ってください。

```
try {
    InitialContext ctx = new InitialContext();
    queue = (Queue) ctx.lookup("queue/MyQueue");
    QueueConnectionFactory factory =
        (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
    cnn = factory.createQueueConnection();
    sess = cnn.createQueueSession(false,
        QueueSession.AUTO_ACKNOWLEDGE);

} catch (Exception e) {
    e.printStackTrace ();
}

TextMessage msg = sess.createTextMessage(...);

sender = sess.createSender(queue);
sender.send(msg);
```

JMS 仕様あるいは書籍を参照し、JMS API のプログラミングの方法について学習してみてください。

## 4.4. EJB3 サービスのパッケージとデプロイ

EJB3 bean クラスは通常の JAR ファイルでパッケージされます。標準設定ファイル (セッション bean は `ejb-jar.xml`、エンティティ bean は `persistence.xml` など) は、JAR の中にある `META-INF` ディレクトリにあります。JBoss AS ではスタンドアローンサービスとして、あるいはエンタープライズアプリ



ケーションの一部 (EAR アーカイブ内) として、EJB3 bean をデプロイすることができます。この章では、この2つのデプロイメントオプションについて説明します。

#### 4.4.1. EJB3 JAR のデプロイ

JAR ファイルを **JBOSS\_DIST/server/default/deploy/** ディレクトリに置くと、自動的にサーバーがピックアップし処理を行います。JAR ファイルに定義されている EJB3 bean はすべて、**MyBean/local** など JNDI 名を使ってサーバーの内部あるいは外部にデプロイされている他のアプリケーションにて利用できるようになります。この **MyBean** は、セッション bean の実装クラス名です。このデプロイメントは、**JBOSS\_DIST/server/default/ejb3.deployer/** の JBoss EJB3 デプロイヤーを使い行われます。デフォルトの EJB3 エンティティマネージャーを設定する際に前述した、**META-INF/persistence.properties** ファイルは、EJB3 デプロイヤーの中に置かれています。

EJB3 デプロイヤーは自動的にクラスパスの JAR をスキャンし、EJB3 アノテーションを検索します。EJB3 アノテーションを持つクラスが見つかった場合、EJB3 サービスとしてこれらのクラスをデプロイします。しかし、多数の JAR がデプロイされている大規模なアプリケーションをお使いの場合は、クラスパスにあるすべての JAR をスキャンするのに大変時間がかかります。

**JBOSS\_DIST/server/default/ejb3.deployer/META-INF/jboss-service.xml** ファイルでは、EJB3 デプロイヤーに EJB3 bean が含まれていないとわかっている JAR を無視するよう指示することができます。JBoss AS で同梱されている EJB 以外の JAR ファイルは **jboss.ejb3:service=JarsIgnoredForScanning** MBean サービスにすでにリストアップされています。

```
... ..
<mbean code="org.jboss.ejb3.JarsIgnoredForScanning"
      name="jboss.ejb3:service=JarsIgnoredForScanning">
  <attribute name="IgnoredJars">
    snmp-adaptor.jar,
    otherimages.jar,
    applet.jar,
    jcommon.jar,
    console-mgr-classes.jar,
    jfreechart.jar,
    juddi-service.jar,
    wsdl4j.jar,
    ... ..
    servlets-webdav.jar
  </attribute>
</mbean>
... ..
```

サーバーがスキャンで時間を無駄にしないように、お使いのアプリケーションにある EJB3 以外の JAR をこのリストに追加することができます。こうすることで、場合によってはアプリケーションの起動時間が大幅に短縮されることがあります。

#### 4.4.2. EJB3 JAR で EAR をデプロイ

Java EE アプリケーションは、EAR アーカイブとしてデプロイされます。EAR アーカイブは、通常 Web ページ、サーブレット、その他の Web 関連のコンポーネント、WAR コンポーネントにサービス (例：データアクセスやトランザクション) を提供する1つ以上の EJB3 JAR、アプリケーションが必要とするサポートライブラリ JAR 向けの WAR アーカイブを含む JAR ファイルです。EAR ファイルには **application.xml** や **jboss-app.xml** など、配備記述子があります。以下に一般的な EAR アプリケーションの基本構造を示しました。

```

myapp.ear
|+ META-INF
    |+ applications.xml and jboss-app.xml
|+ myapp.war
    |+ web pages and JSP /JSF pages
    |+ WEB-INF
        |+ web.xml, jboss-web.xml, faces-config.xml etc.
        |+ lib
            |+ tag library JARs
        |+ classes
            |+ servlets and other classes used by web pages
|+ myapp.jar
    |+ EJB3 bean classes
    |+ META-INF
        |+ ejb-jar.xml and persistence.xml
|+ lib
    |+ Library JARs for the EAR

```

JBoss AS では他の多くのアプリケーションサーバーとは違い、WAR ファイルのコンポーネントが EJB3 サービスにアクセスできるように web.xml ファイルで EJB 参照を宣言する必要がありません。本章で前述したように、JNDI から直接参照を取得することができます。

一般的な application.xml ファイルは以下の通りです。このファイルは、WAR と EJB3 JAR アーカイブを EAR で宣言し、アプリケーションの Web コンテンツルートを定義します。当然、同じ EAR アプリケーション内に複数の EJB3 モジュールを持たせることができます。application.xml ファイルはオプションで、このアプリケーション内で利用する JAR ファイルの共有クラスパスを定義することも可能です。JAR ファイルの場所は、デフォルトで JBoss AS の lib になっていますが、他のアプリケーションサーバーでは異なる可能性もあります。

```

<application>
  <display-name>My Application</display-name>

  <module>
    <web>
      <web-uri>myapp.war</web-uri>
      <context-root>/myapp</context-root>
    </web>
  </module>

  <module>
    <ejb>myapp.jar</ejb>
  </module>

  <library-directory>lib</library-directory>

</application>

```

jboss-app.xml ファイルで、EAR アプリケーション向けに JBoss 固有のデプロイメントを設定できます。例えば、EAR 内でモジュールのデプロイメントの順番を指定、SAR (ARchive for MBean) や HAR (Hibernate ARchive for Hibernate objects) など JBoss 固有のアプリケーションモジュールを EAR にデ

プロイ、このアプリケーションで利用可能なセキュリティドメインと JMX MBean を提供します。  
jboss-app.xml で利用可能な属性については、DTD : [http://www.jboss.org/j2ee/dtd/jboss-app\\_4\\_2.dtd](http://www.jboss.org/j2ee/dtd/jboss-app_4_2.dtd) を参照してください。

jboss-app.xml の一般的な用途は、他のアプリケーションとの名前の衝突を避けるため、この EAR ファイルを独自にスコープ設定されたクラスローダーにデプロイすべきかを設定することです。お使いの EAR アプリケーションが、独自にスコープ設定されたクラスローダー内にデプロイされ、EJB3 JAR に永続ユニットが1つのみ定義されている場合、@PersistenceContext アノテーションに永続ユニット名を渡す必要なく、@PersistenceContext EntityManager を使い EntityManager をセッション bean に注入することができます。以下のjboss-app.xml は、EAR アプリケーションに対しスコープ設定されたクラスローダー myapp:archive=myapp.ear を指定しています。

```
<jboss-app>
  <loader-repository>
    myapp:archive=myapp.ear
  </loader-repository>
</jboss-app>
```

EAR デプロイメントは、JBOSS\_DIST/server/default/deploy/ear-deploy.xml で設定します。このファイルには、以下のような3つの属性が含まれています。

```
<server>
  <mbean code="org.jboss.deployment.EARDeployer"
    name="jboss.j2ee:service=EARDeployer">
    <!--
      A flag indicating if ear deployments should
      have their own scoped class loader to isolate
      their classes from other deployments.
    -->
    <attribute name="Isolated">false</attribute>

    <!--
      A flag indicating if the ear components should
      have in VM call optimization disabled.
    -->
    <attribute name="CallByValue">false</attribute>

    <!--
      A flag the enables the default behavior of
      the ee5 library-directory. If true, the lib
      contents of an ear are assumed to be the default
      value for library-directory in the absence of
      an explicit library-directory. If false, there
      must be an explicit library-directory.
    -->
    <attribute name="EnablelibDirectoryByDefault">true</attribute>
  </mbean>
</server>
```

Isolated パラメーターを true に設定した場合、EAR デプロイメントはすべて、デフォルトでスコープ設定されたクラスローダーを持つことになります。jboss-app.xml でクラスローダーを定義する必要は

ありません。CallByValue 属性は、全 EJB 呼び出しをリモートの呼び出しとして処理すべきかを指定します。リモート呼び出しは、ローカルの参照渡しと比較すると、大幅にパフォーマンスが低下しています。理由は、リモート呼び出しのオブジェクトは、シリアル化、デシリアル化される必要があるためです。アプリケーションの多くは、WAR および EJB3 JAR を同じサーバー上でデプロイするため、この値はデフォルトで false になっており、サーバーはローカルの参照渡しを使い同じ JVM 内にある EJB メソッドを呼び出します。EnablelibDirectoryByDefault 属性は、EAR アーカイブの lib ディレクトリが共有ライブラリ JAR のデフォルトの場所とすべきかを指定します。

## 第5章 ログイン

ログインとは、エラーの解決やプラットフォームのコンポーネントの状況を監視するにあたり最も重要なツールです。**log4j** は、Java 開発者にとって馴染みがあり、柔軟なフレームワークを提供しています。

「[ログインのデフォルト](#)」には、当プラットフォームに対するデフォルトのログイン動作をカスタマイズする方法について説明しています。その他のカスタマイゼーションについては、「[コンポーネント固有のログイン](#)」を参照してください。[付録B ログイン情報とレシピ](#)で、ログインレシピを提供しており、必要に応じてカスタマイズ可能です。

### 5.1. ログインのデフォルト

**log4j** の設定は `$JBOSS_HOME/server/PROFILE/conf/jboss-log4j.xml` の配備記述子からロードされます。**log4j** は、アペンダーを使いログインの動作を制御します。アペンダーは情報をどこにログするか、どのようにログするか指示を出します。**jboss-log4j.xml** ファイルには、FILE、CONSOLE、SMTPなど、多くのサンプルアペンダーが含まれています。

表5.1 log4j 設定の一般的な指示子

設定オプション	詳細
<b>appender</b>	主なアペンダー。名前、実装クラスを提供します。
<b>errorHandler</b>	特にアペンダーが何らかの理由でログを記述できない場合に、外部クラスを委譲し、ロガーに渡される例外を処理します。
<b>param</b>	アペンダータイプ固有のオプション。この例では、 <code>&lt;param&gt;</code> はファイル名でFILE appender のログを格納します。
<b>layout</b>	ログインのフォーマットを制御。これを微調整することで、ご利用になりたいログ構文解析ソフトウェアと連携させます。

#### 例5.1 サンプルのアペンダー

```
<appender name="FILE"
class="org.jboss.logging.appender.DailyRollingFileAppender">
  <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="File" value="${jboss.server.log.dir}/server.log"/>
  <param name="Append" value="true"/>
  <!-- In AS 5.0.x the server log threshold was set by a system
property.
In 5.1 and later, the system property sets the priority on the root
logger (see <root/> below)
  <param name="Threshold" value="${jboss.server.log.threshold}"/> -->

  <!-- Rollover at midnight each day -->
  <param name="DatePattern" value="'.'yyyy-MM-dd"/>
  <layout class="org.apache.log4j.PatternLayout">
    <!-- The default pattern: Date Priority [Category] (Thread) Message\n
-->
```

```
<param name="ConversionPattern" value="%d %-5p [%c] (%t) %m%n"/>
</layout>
</appender>
```

**log4j**設定の詳細情報については、<http://logging.apache.org/log4j/1.2/>を参照してください。

## 5.2. コンポーネント固有のロギング

プラットフォームのコンポーネントによっては、別のロギングオプション、あるいは追加でロギングのカスタマイズメカニズムを利用できます。

### 5.2.1. Hibernate でのSQLロギング

Hibernate では、SQL ステートメントのロギングを可能にする方法は2種類あります。これらのステートメントは、アプリケーション開発のテストフェーズやデバッグフェーズで最も役立ちます。

1 つ目は、コードで明示的に有効にする方法です。

```
SessionFactory sf = new Configuration()
    .setProperty("hibernate.show_sql", "true")
    // ...
    .buildSessionFactory();
```

もう 1 つの方法は、特別な機能を使ってHibernate にて設定を行うことで、SQL メッセージを全て **log4j** に送信することができます。

```
log4j.logger.org.hibernate.SQL=DEBUG, SQL_APPENDER
log4j.additivity.org.hibernate.SQL=false
```

**additivity** オプションは、これらのログメッセージが親ハンドラーにも反映されるかどうかを制御しますが、このオプションは任意です。

### 5.2.2. トランザクションサービスのロギング

Enterprise Platform に含まれるTransactionManagerService は、スタンドアローンの Transaction Service と違った形でロギングを処理します。特に、**jbossjta-properties.xml** にある `com.arjuna.common.util.logger` property の値をオーバーライドし、**log4j\_releveller** ロガーを強制的に利用します。トランザクションコードの **INFO** レベルのメッセージは、**DEBUG** メッセージのような動作をします。そのため、フィルターレベルが **DEBUG** の場合、これらのメッセージはログファイルにのみ表示されます。その他のログメッセージはすべて通常通りの動作となっています。

## 第6章 デプロイメント

JBoss Enterprise Application Platform にアプリケーションをデプロイするには、アプリケーションを **\$JBOSS\_HOME/server/default/deploy** ディレクトリにコピーします。**default** は **all** や **minimal** などの他のサーバープロファイルに置き換えることができます (サーバープロファイルについては後ほど説明します)。JBoss Enterprise Application Platform は常に deploy ディレクトリをスキャンし、新しいアプリケーションや既存アプリケーションへの変更を検知します。これにより、JBoss Enterprise Application Platform を実行したままアプリケーションの**ホットデプロイメント**をオンザフライで実行できます。

### 6.1. デプロイ可能なアプリケーションのタイプ

JBoss Enterprise Application Platform 4.x には、指定のデプロイメントタイプを処理するデプロイヤーがあり、それがデプロイメントを処理する唯一のデプロイヤーでした。JBoss Enterprise Application Platform 5 では、メタデータよりランタイムコンポーネントを作成するデプロイヤーによって処理されるまで複数のデプロイヤーがデプロイメントに関連するメタデータを変換します。

デプロイメントには、コンポーネントのメタデータがデプロイメントに追加されるようにする記述子が含まれなければなりません。JBoss Enterprise Application Platform のデフォルトでデプロイヤーが存在するデプロイメントタイプは次の通りです。

#### WAR

WAR アプリケーションアーカイブ (例、myapp.war) は JAR ファイル内に Java EE Web アプリケーションをパッケージングします。これには、サーブレットクラス、ビューページ、ライブラリ、さらには **web.xml**、**faces-config.xml**、**jboss-web.xml** といった、WEB-INF の配備記述子が含まれます。

#### EAR

WAR アプリケーションアーカイブ (例、myapp.war) は JAR ファイル内に Java EE 企業向けアプリケーションをパッケージングします。通常、Web モジュールの WAR ファイル、EJB モジュールの JAR ファイル、application.xml や jboss-app.xml などの META-INF 配備記述子が含まれます。

## 注記

EJB3 仕様に従い、永続ユニットがEAR ファイルの外部にあり、bean がこの永続ユニットをEAR内に投入しようとする、EARへの永続ユニットのデプロイは失敗します。この仕様にあわせ、EARファイル内にパッケージされた永続ユニットをデプロイする必要があります。

しかし、JBoss EAP 永続ユニットはEAR外でも存在することはできます。この動作を可能にするには、該当の JBoss AS サーバプロファイルの **deployers/ejb3.deployer/META-INF/jpa-deployer-jboss-beans.xml** ファイルにある **PersistenceUnitDependencyResolver** beanのbean クラスを変更します。

```
<!--
Can be DefaultPersistenceUnitDependencyResolver for spec
compliant resolving,
InterApplicationPersistenceUnitDependencyResolver for
resolving beyond EARs,
or DynamicPersistenceUnitDependencyResolver which
allows configuration via JMX.
-->
<bean name="PersistenceUnitDependencyResolver"
class="org.jboss.jpa.resolvers.DynamicPersistenceUnitDependen
cyResolver"/>
```

このbean のデフォルト値は**DynamicPersistenceUnitDependencyResolver**です。このリボルバーにより、仕様に準拠した動作を指定することができ、JMX Console の MBean からこれらの動作を追加で監視可能です。仕様に準拠しない JBoss 変数を使うには、bean を**InterApplicationPersistenceUnitDependencyResolver**に設定します。

## JBoss Microcontainer

JBoss Microcontainer (MC) Bean アーカイブ (拡張子は .beans や .deployer など) は、**META-INF/jboss-beans.xml** 記述子と共に POJO デプロイメントを JAR ファイルにパッケージ化します。この形式は、JBoss Enterprise Application Platform のコンポーネントデプロイヤによって頻繁に使用されます。

**\*-jboss-beans.xml** は MC Bean 定義を使用してデプロイすることができます。deploy ディレクトリまたは lib ディレクトリ内に適切な JAR ファイルがあれば、スタンドアローン XML を使用して MC Bean をデプロイすることができます。

## SAR

SAR アプリケーションアーカイブ (myservice.sar など) は、JBoss サービスを JAR ファイルにパッケージ化します。主に、MC Bean スタイルのデプロイメントサポートのために更新されていない JBoss Enterprise Application Platform の内部サービスによって使用されます。

**\*-service.xml** は MBean サービス定義を使用してデプロイすることができます。deploy ディレクトリまたは lib ディレクトリ内に適切な JAR ファイルがあれば、XML ファイルに指定された MBean を起動することができます。JMS キューなど、POJO スタイルのデプロイメントサポートのために更新されていない JBoss Enterprise Application Platform の内部サービスをデプロイする方法になります。

## DataSource



**\*-ds.xml** ファイルは外部データベースとの接続を定義します。これにより内部 JNDI を介して JBoss Enterprise Application Platform 内の全てのアプリケーションとサービスによるデータソースの再利用が可能になります。

## HAR

HAR ファイルは、アプリケーション用に Hibernate オブジェクトを定義します。これは SAR ファイルに似ていますが、META-INF ディレクトリに Hibernate クラスとマッピングファイル、\*-hibernate.xml 配備記述子を含みます。

### 注記

**\*-hibernate.xml** は、**jboss-service.xml** と同じ形式を取ります。

#### 例6.1 Hibernate 配備記述子 (\*-hibernate.xml)

```
<hibernate-configuration xmlns="urn:jboss:hibernate-
  deployer:1.0">
  <session-factory name="java:/hibernate/SessionFactory"

  bean="jboss.test.har:service=Hibernate,testcase=TimersUnitT
  estCase">
    <property name="datasourceName">OracleDS</property>
    <property
  name="dialect">org.hibernate.dialect.OracleDialect</propert
  y>
    <depends>jboss:service=Naming</depends>
    <depends>jboss:service=TransactionManager</depends>
  </session-factory>
</hibernate-configuration>
```

## \*AR

EJB が含まれる JAR ファイルや、他のサービスオブジェクトを直接 JBoss Enterprise Application Platform にデプロイすることもできます。JAR ファイルとして認識される拡張子の一覧は、**conf/bootstrap/deployers.xml** の JARStructure Bean コンストラクタセットに指定されます。

### 6.1.1. 展開デプロイメント

WAR、EAR、MC Bean、SAR のデプロイメントパッケージは、META-INF や WEB-INF のようなディレクトリ内の特殊な XML 配備記述子を持つ JAR ファイルです。JBoss Enterprise Application Platform により、これらのアーカイブを JAR ファイルではなく展開ディレクトリとしてデプロイすることができるようになります。これにより、アプリケーション全体を再デプロイすることなくオンザフライで Web ページなどに変更を加えることができます。サーバーを再起動せずに展開ディレクトリを再デプロイする必要がある場合は、配備記述子 (WAR 内での **WEB-INF/web.xml**、EAR 内での **META-INF/application.xml**) を **touch** してタイムスタンプを更新します。

## 6.2. 標準のサーバースプロファイル

各プロファイルは **\$JBOSS\_HOME/server/PROFILE/** という名前のディレクトリに格納されます。各サーバープロファイルのディレクトリを確認すると、プロファイルに含まれるサービスやアプリケーション、ライブラリを確認することができます。



#### 注記

**\$JBOSS\_HOME/server/PROFILE** ディレクトリの実際の内容は、サーバープロファイルのサービス実装によって異なります。また、管理層や組み込みサーバーも進化しているため、ディレクトリの内容が変更する可能性があります。

### all

**all** プロファイルは、クラスタリングサポートや他のエンタープライズ拡張を提供しています。

### production

**production** サーバープロファイルは **all** プロファイルを基に実稼働環境向けに最適化された設定を提供します。

### minimal

企業向けサービスを使用せずにコアサーバーコンテナを起動します。**minimal** サーバープロファイルは必要なサービスのみを含むJBoss Enterprise Application Platform をカスタマイズ構築する際に基盤として使います。

### default

**default** サーバープロファイルはアプリケーション開発者が最も頻繁に使用するプロファイルで、標準の Java EE 5.0 プログラミング API (Annotations、JPA、EJB3 など) をサポートします。



#### 注記

プロファイルがコマンドラインあるいは設定ファイルで指定されていない場合、**default** サーバープロファイルを使います。

### standard

**standard** サーバープロファイルは Java EE に準拠するようテストされたプロファイルです。既存設定との主な違いは、デフォルトでコールバイバリュー (call-by-value) とデプロイメント分離が有効になっていることと、**rmiiop** と **juddi** (**all** 設定より取得) のサポートが有効になっていることです。

### web

**web** サーバープロファイルは、JBoss Web 向けに実験的に作成された軽量設定で、JavaEE 6 web サーバープロファイルの開発に従っていきます。**servlet/jsp** コンテナ以外は JTA/JCA と JPA のサポートを提供します。HTTP ポートからのみサーバーへのアクセスが許可されるよう制限されています。この設定は JavaEE 認定の設定ではなく、今後のリリースで変更になる可能性が高いことにご留意ください。

サービスの詳細や各サーバープロファイルでサポートされる API については本書にて説明していきます。

#### 6.2.1. プロファイルの変更

サーバーが利用するプロファイルを変更したい場合、サーバーがコマンドラインで開始するか、サービスとして開始するかによりメソッドが変わります。

サーバーがコマンドラインで開始された場合、**run.sh -c profile** のように **-c** パラメーターで必要なプロファイルを指定します。例えば、Red Hat Enterprise Linux では **run.sh -c all**、Microsoft Windows では **run.bat -c all** コマンドが **all** サーバードプロファイルでサーバーを起動します。

サーバーがサービスとして起動された場合は、サービスを使ったプロファイルが再構成し、サービスの停止、開始を行います。プロファイル指定箇所の詳細については『インストールガイド』の『サービスとして Enterprise Application Platform を起動』の章を参照してください。

## 6.3. コンテキストルート

コンテキストルートは、デプロイ済みアプリケーションの URL を決定します。デフォルトでは、コンテキストルートはアプリケーションのディレクトリあるいはアーカイブ構造と同じとなっています。例えば、JSP ページで **hello** ディレクトリを含む、**application.war** アーカイブをデプロイした場合、**hello** アプリケーションのコンテキストルートは、**/application/home** となります。

### 手順6.1 デフォルトのコンテキストルートを書き直し

必要であればコンテキストルートを変更することが可能です。コンテキストルートを書きなおす場合、新規コンテキストルートを定義し、サーバーが新規コンテキストを利用できるようにしなければなりません。

1. 新規のコンテキストルートを定義するには、新しい値を持つ **context-root** 要素をアプリケーションの配備記述子に追加します。
  - Web アプリケーションのコンテキストアプリケーションを変更するには、**context-root** 要素を **jboss-web.xml** ファイルに追加します。

#### 例6.2 コンテキストルートを定義した jboss-web.xml 例

```
<?xml version="1.0"?>
<jboss-web>
  <context-root>/application-root</context-root>
</jboss-web>
```

localhost のアプリケーション用 URL アドレスは  
**http://localhost:8080/application-root**

です。

- サーブレットのコンテキストルートを変更するには、**web.xml** ファイルの **url-pattern** 要素を変更します。

#### 例6.3 コンテキストルートを定義した web.xml 例

```
<?xml version="1.0"?>
<servlet-mapping>
  <servlet-name>MapRenderer</servlet-name>
  <url-pattern>/servlet-root</url-pattern>
</servlet-mapping>
```

localhost のサーブレット用 URL アドレスは  
`http://localhost:8080/application-root/servlet-root`

です。

2. `REWRITE_CONTEXT_CHECK` 変数が **false** に設定されているサーバーを起動するには、**`run.sh`** -  
**`Dorg.apache.catalina.connector.Response.REWRITE_CONTEXT_CHECK=false`** コマンドを実行します。

## 第7章 マイクロコンテナ

JBoss Enterprise Application Platform 5.0 は、標準的な Java EE 環境を提供するため、マイクロコンテナを使用して企業向けサービスと Servlet/JSP コンテナ、EJB コンテナ、デプロイヤー、管理ユーティリティを統合します。その他のサービスが必要な場合はサービスを Java EE 上にデプロイすると必要な機能を提供できます。同様に、必要でないサービスはサーバープロファイル設定を変更して削除することができます。サービスデプロイメントの段階で異なるクラスローディングモデルにプラグすれば、Tomcat や GlassFish などの他の環境でさえもマイクロコンテナを使用して同様の操作を行うことができます。

JBoss Microcontainer は大変軽量で POJO に対応するため、Java ME ランタイム環境にサービスをデプロイするために使用することもできます。これにより、モバイルアプリケーションは完全な JEE アプリケーションサーバーを必要とせず企業向けサービスを利用できるため、モバイルアプリケーションの可能性を新たに広げることになります。他の軽量コンテナと同様に、JBoss Microcontainer は依存関係の挿入を使用して個別の POJO をワイヤーし、サービスを作成します。情報の場所によりアノテーションまたは XML を使用して設定を行います。テスト環境を設定する JUnit を拡張するヘルパークラスにより、テストが大変簡単になりました。これにより、コードを数行使用するだけでテストメソッドより POJO やサービスへアクセスできるようになりました。



### 注記

マイクロコンテナアーキテクチャーに関する詳細情報は、[docs.redhat.com](https://docs.redhat.com) のマイクロコンテナユーザガイドを参照してください。

## 第8章 JNDI ネーミングサービス

ネーミングサービスはエンタープライズ Java アプリケーションにて重要な役割を果たしており、アプリケーションサーバーでオブジェクトやサービスの場所を検索する際に利用する中核インフラストラクチャーを提供しています。また、このサービスは、アプリケーションサーバーの外部にあるクライアントがアプリケーションサーバー内にあるサービスを検索するためのメカニズムでもあります。アプリケーションコードが JBoss Enterprise Application Platform インスタンス内部、外部のいずれであっても、このアプリケーションコードは **queue/IncomingOrders** という名称のメッセージキューに接続するだけで、キューの設定詳細に関しては考慮する必要がありません。

クラスター環境では、ネーミングサービスは価値が高くなっています。サービスのクライアントは、どのマシンに常駐しているか把握する必要なしに、クラスターから **ProductCatalog** セッション bean をルックアップできる必要があります。大規模なクラスター化サービス、ローカルリソース、あるいはアプリケーションコンポーネントのいずれかが必要かによって、JNDI ネーミングサービスは、コードが名前別にシステム内のオブジェクトを検索できるよう接点としての機能を提供します。

### 8.1. JNDI の概要

JNDI は、Java 開発キットに同梱されている標準の Java API です。JNDI は、DNS、LDAP、Active Directory、RMI レジストリ、COS レジストリ、NIS、ファイルシステムなど、既存の各種ネーミングサービスに対して共通のインターフェースを提供しています。JNDI API はネーミングサービスのアクセスに使用するクライアント API と、ユーザーがネーミングサービスに対して JNDI 実装を作成できるようにするサービスプロバイダーインターフェース (SPI: service provider interface) とに論理的に分割されます。

SPI 層は、中核 JNDI クラスが共通の JNDI クライアントインターフェースを使いネーミングサービスを公開できるように、ネーミングサービスプロバイダーが実装しなければならない抽象化層で、ネーミングサービスに対する JNDI の実装は *JNDI プロバイダー* と呼んでいます。JBoss ネーミングは SPI クラスをベースとする JNDI 実装例になります。J2EE コンポーネント開発者には JNDI SPI は必要ないので注意してください。

主な JNDI API パッケージは、**javax.naming** パッケージで、この中にはインターフェースが5個、クラスが10個、例外が数個含まれています。主なクラス1つ (**InitialContext**) と、インターフェース2つ (**Context** および **Name**) が存在します。

#### 8.1.1. 名前

名前の概念は JNDI では欠かせない重要なものとなります。ネーミングシステムにより、従うべき名前の構文が決定します。ユーザーはネーミングシステムの構文を使い、名前の文字列表現をコンポーネントに構文解析することができるようになります。名前はオブジェクトの検索にネーミングシステムで使われます。最もシンプルに言い替えると、ネーミングシステムは単に一意名を持つオブジェクトの集合なのです。ネーミングシステムでオブジェクトを検索する場合、ネーミングシステムに名前を提示するとネーミングシステムはその名前を持つオブジェクトストアを返します。

例として、UNIX ファイルシステムの命名規則を見ていきます。各ファイルは、ファイルシステムの root を起点とした相対パスで命名され、それぞれのコンポーネントはスラッシュ ("/") で区切られます。このファイルのパスは左から右の順になります。たとえば、パス名 **/usr/jboss/readme.txt** は、ファイルシステムの root にある **usr** ディレクトリ配下の **jboss** ディレクトリ内にある **readme.txt** ファイルを指します。JBoss Enterprise Application Platform のネーミングは、Unix 形式の命名空間を命名規則として採用しています。

**javax.naming.Name** インターフェースは、汎用名をコンポーネントの順に表します。合成名 (複数の名前空間にわたる名前) の場合も、複合名の場合も (単一階層のネーミングシステム内で利用される名前) あります。名前のコンポーネントには番号がつけられます。コンポーネント N を持つ名前のイン

デックスは、0 から N 未満の範囲となっています。最も重要なコンポーネントは、インデックス 0 にあります。また、空の名前にはコンポーネントがありません。

合成名は、複数の名前空間にわたる一連のコンポーネント名です。合成名の例として、**scp** などの Unix コマンドで通常使われるホスト名とファイルの組み合わせなどでしょう。例えば、以下のコマンドは、**localfile.txt** を、**ahost.someorg.org** ホストにある **tmp** ディレクトリの **remotefile.txt** ファイルにコピーします。

```
scp localfile.txt ahost.someorg.org:/tmp/remotefile.txt
```

複合名は、階層名前空間から抽出されます。複合名の各コンポーネントは原子名 (atomic name) で、それ以上小さなコンポーネントには分類できない文字列になります。Unix ファイルシステムのファイルパス名が複合名の一例です。**ahost.someorg.org:/tmp/remotefile.txt** は、DNS と Unix ファイルシステムの名前空間にわたる複合名となっています。また、複合名のコンポーネントは、**ahost.someorg.org** と **/tmp/remotefile.txt** で、コンポーネントは、ネーミングシステムの名前空間からの文字列名となっています。コンポーネントが階層名前空間から来るものであれば、このコンポーネントは、**javax.naming.CompoundName** クラスを使って原子パーツにさらに構文解析することができます。JNDI API は、合成名の **Name** インターフェースの実装として、**javax.naming.CompositeName** クラスを提供します。

### 8.1.2. コンテキスト

**javax.naming.Context** インターフェースは、ネーミングサービスとやりとりを行うにあたり主要なインターフェースとなっています。**Context** インターフェースは、名前とオブジェクトのバインディングセットを表します。全コンテキストは紐付けられた命名規則があり、それにより、コンテキストが **javax.naming.Name** インスタンスで文字列名を解析する方法を決定します。名前とオブジェクトのバインディングを作成するには、**Context** のバインドメソッドを呼出し、名前とオブジェクトを引数として指定します。このオブジェクトは、あとで **Context** のロックアップメソッドを使った名前でもトリブすることができます。通常、**Context** で、名前とオブジェクトのバインド、名前のアンバインド、名前とオブジェクトの全バインディングの一覧を取得する操作を行うことができます。**Context** にバインドするオブジェクトは、それ自体が **Context** タイプになり得ます。バインドされた **Context** オブジェクトは、バインドメソッドが呼び出されたところで **Context** のサブコンテキストとして参照されます。

例として、パス名が **/usr** で、Unix ファイルシステムではコンテキストとなるファイルディレクトリを見てみましょう。別のファイルディレクトリを起点に指定されたファイルディレクトリはサブコンテキスト (通常サブディレクトリと呼ばれる) となります。**/usr/jboss** とのパス名を持つファイルディレクトリは、**usr** のサブコンテキストである **jboss** コンテキストを指定します。別の例では、**org** などの DNS ドメインはコンテキストで、別の DNS ドメインを起点に指定された DNS ドメインはサブコンテキストの別例となっています。DNS ドメイン **jboss.org** では、DNS 名は右から左に解析されるため、DNS ドメイン **jboss** が **org** のサブコンテキストとなります。

#### 8.1.2.1. InitialContext を使ってコンテキストを取得

ネーミングサービスの操作はすべて **Context** インターフェースの実装で行われます。したがって、使おうとしているネーミングサービスの **Context** を取得する方法が必要になります。**javax.naming.InitialContext** クラスは **Context** インターフェースを実装するので、ネーミングサービスとのやりとりを開始することができます。

**InitialContext** を作成する場合、この環境からのプロパティを使って初期化されます。JNDI は以下のソース 2 つからの値を順にマージすることで、各プロパティの値を決定します。

- コンストラクターの環境パラメーターから最初に発生したプロパティ、(適切なプロパティに対して) アプレットパラメーターとシステムプロパティ

- クラスパス上にある全 **jndi.properties** リソースファイル

上記の 2 ソース両方にある各プロパティに関して、プロパティの値が次のように確定されます。プロパティが JNDI ファクトリー一覧を指定する標準の JNDI プロパティのいずれかの場合、すべての値がコロンで区切られた単一リストに連結されます。その他のプロパティは最初に見付かった値のみが使用されます。JNDI 環境プロパティを指定する方法としては、**jndi.properties** ファイルを使う方法をお勧めします。このファイルによりコードが JNDI プロバイダー固有の情報を外部に配置することができるようになるため、JNDI プロバイダーを変更してもコードの変更や再コンパイルを必要としません。

**InitialContext** クラスが内部で使用する **Context** 実装は、ランタイム時に決定されます。デフォルトポリシーは、**javax.naming.spi.InitialContextFactory** 実装のクラス名を含む環境プロパティ **java.naming.factory.initial** を使います。ご利用中のネーミングサービスプロバイダーから **InitialContextFactory** クラス名を取得します。

例8.1「**jndi.properties** ファイルの例」で、クライアントアプリケーションがポート 1099 のローカルホストで稼働する JBossNS サービスに接続する際に利用するサンプルの **jndi.properties** ファイルを提供しています。クライアントアプリケーションは、アプリケーションクラスパスにある **jndi.properties** ファイルが必要となるでしょう。これらは、JBossNS JNDI 実装が必要とするプロパティです。その他の JNDI プロバイダーには、別のプロパティと値があります。

#### 例8.1 jndi.properties ファイルの例

```
### JBossNS properties
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

## 8.2. JBOSS NAMING SERVICE アーキテクチャー

JBoss Naming Service (JBossNS) アーキテクチャーは、**javax.naming.Context** インターフェースの Java ソケット/RMI ベース実装となっています。JBossNS はリモートからアクセス可能なクライアント/サーバー実装で、この実装は最適化されており、ソケットを必要とせずに、JBossNS サーバーが稼働している同じ VM からアクセスすることができます。グローバルシングルトンとして利用可能なオブジェクト参照を使うことで同じ VM アクセスが発生します。図8.1「JBoss Naming Service アーキテクチャーで主要なコンポーネント」では、JBossNS 実装およびその関係において重要となるキーの一部を図解しています。



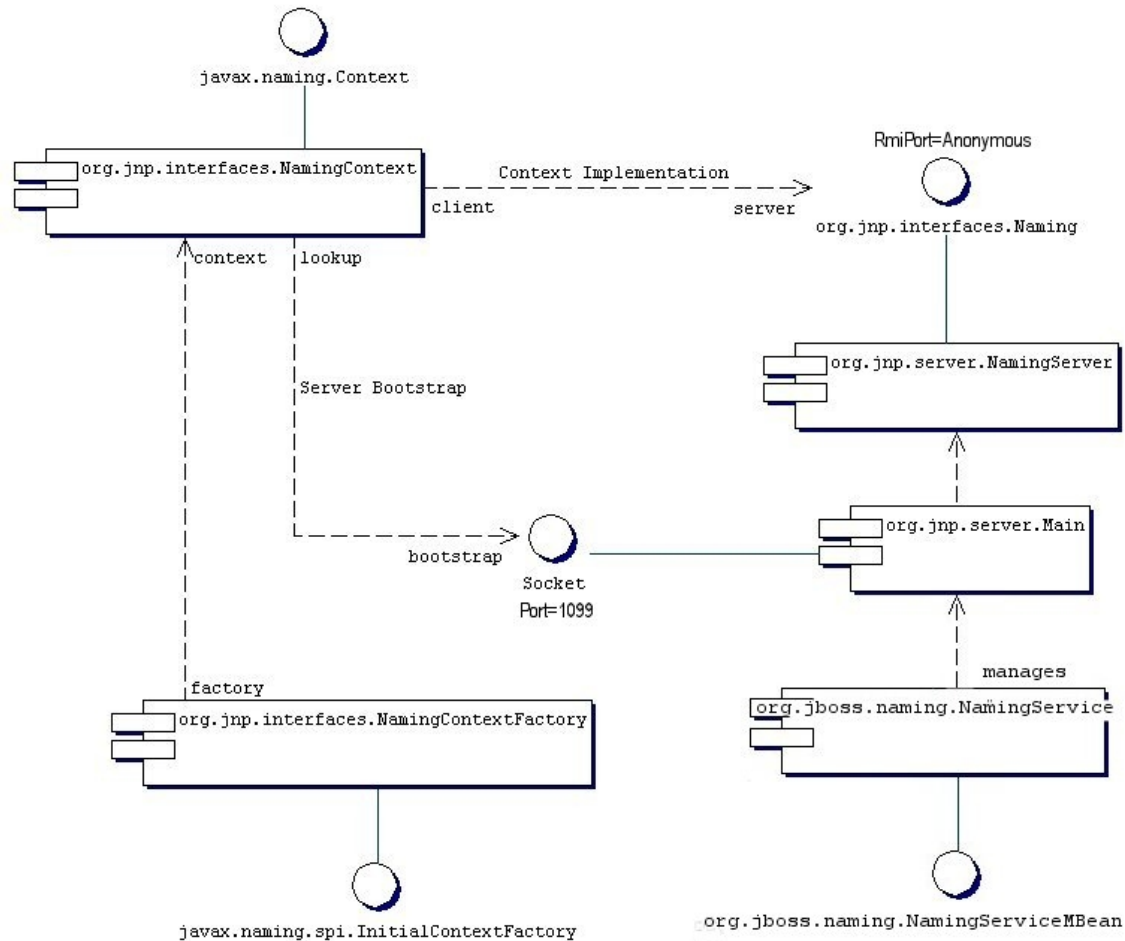


図8.1 JBoss Naming Service アーキテクチャーで主要なコンポーネント

まず、**NamingService** MBeanから見ていきます。**NamingService** MBean は、JNDI ネーミングサービスを提供しており、J2EE テクノロジーコンポーネントで広範利用されている主要サービスです。**NamingService** の設定可能な属性を以下に示します。

- **Port:** **NamingService**のポートをリッスンする jnp プロトコル。指定がない場合は、RMI レジストリのデフォルトポートと同じ 1099 がデフォルトになります。
- **RmiPort:** RMI ネーミング実装がエクスポートされるRMI ポート。指定がない場合、デフォルトは0で、利用可能なポートはどれでも利用するという意味です。
- **BindAddress:** **NamingService** がリッスンする特定のアドレス。ソケットアドレスの1つに関する接続リクエストしか受け付けない **java.net.ServerSocket** のマルチホームホスト上で利用可能です。
- **RmiBindAddress:** **NamingService** の RIM サーバー部分がリッスンする特定のアドレスです。ソケットアドレスの1つに関する接続リクエストしか受け付けない **java.net.ServerSocket** のマルチホームホスト上で利用可能です。指定されていない場合、**BindAddress** が指定されている場合、**RmiBindAddress** は **BindAddress** 値にデフォルト設定されます。
- **Backlog:** 受信接続の表示 (接続要求) に対する最大長は **backlog** パラメーターに設定します。キューがフルの状態では接続表示を受けると、その接続は拒否されます。
- **clientSocketFactory:** オプションのカスタム **java.rmi.server.RMIClientSocketFactory** 実装クラス名。指定されていない場合、デフォルトの **RMIClientSocketFactory** が利用されます。

- **ServerSocketFactory**: オプションのカスタム `java.rmi.server.RMIServerSocketFactory` 実装名。指定されていない場合、デフォルトの `RMIServerSocketFactory` が利用されます。
- **JNPServerSocketFactory**: オプションのカスタム `javax.net.ServerSocketFactory` 実装クラス名。JBoss Naming Service **Naming** インターフェースのダウンロードをブートストラップするために使用する `ServerSocket` のファクトリ。指定されていない場合、`javax.net.ServerSocketFactory.getDefault()` メソッド値が利用されます。

`java:comp` コンテキストへアクセスするスレッドのコンテキストクラスローダーに基づき、このコンテキストへのアクセスが隔離されるように、**NamingService** も `java:comp` を作成します。こうすることで J2EE 仕様で必要なアプリケーションコンポーネントのプライベート ENC を提供します。このような隔離は、`javax.naming.Reference` を `org.jboss.naming.ENCFactory` を使用するコンテキストに `javax.naming.ObjectFactory` としてバインドすることで行うことができます。クライアントが `java:comp` やサブコンテキストのルックアップを行う場合、**ENCFactory** はスレッドコンテキスト `ClassLoader` をチェックし、`ClassLoader` をキーとして使いマップ内でルックアップを実行します。

クラスローダーインスタンスに対しコンテキストインスタンスが存在しない場合、**ENCFactory** マップ内にコンテキストインスタンスが作成されクラスローダーと紐付けられます。そのため、実行中のコンポーネントスレッドに紐付いた固有の `ClassLoader` を受け取る各コンポーネントにより、正確にアプリケーションコンポーネントの ENC を隔離されるか左右されます。

**NamingService** はその機能を `org.jnp.server.Main` MBean に委譲します。MBean が重複する理由には、JBoss Naming Service が最初はスタンドアローンの JNDI 実装として開始され、そのまま実行することができるという点が挙げられます。**NamingService** MBean は **Main** インスタンスを JBoss サーバーに埋め込むため、JBoss サーバーと同じ VM と JNDI を利用しても、ソケットのオーバーヘッドは発生しません。実際は、**NamingService** の設定可能な属性は、JBoss Naming Service **Main** MBean の設定可能な属性なのです。**NamingService** MBean 上の属性設定は、単に **NamingService** 内の **Main** MBean にある該当属性を設定するだけです。**NamingService** が起動すると、包含する **Main** MBean を開始し JNDI のネーミングサービスを有効にします。

さらに、**NamingService** は、JMX detyped 呼出し操作により **Naming** インターフェースの操作を公開します。これにより、任意のプロトコルに対する JMX アダプタを経由でネーミングサービスにアクセスできるようになります。HTTP が呼出し操作を使ってネーミングサービスにアクセスする方法について、一例を本章で後述します。

**Main** MBean が開始されると、以下のタスクを実行します。

- `org.jnp.naming.NamingService` インスタンスをインスタンス化し、これをローカルの VM サーバーインスタンスとして設定します。これは、JBoss サーバーの VM 内に作成された `org.jnp.interfaces.NamingContext` インスタンスのいずれかが使用し TCP/IP での RMI 呼出しが回避されるようにします。
- 設定された `RmiPort`、`ClientSocketFactory`、`ServerSocketFactory` 属性を使って、**NamingServer** インスタンスの `org.jnp.naming.interfaces.Naming` RMI インターフェースをエクスポートします。
- `BindAddress` と `Port` 属性により与えられたインターフェースをリッスンするソケットを作成します。
- ソケットで接続を受け取るスレッドを生成します。

## 8.3. NAMING INITIALCONTEXT ファクトリ

JBoss JNDI プロバイダーは現在、様々な**InitialContext** ファクトリ実装に対応しています。

### 8.3.1. 標準のネーミングコンテキストファクトリ

最もよく使われるファクトリは、**org.jnp.interfaces.NamingContextFactory**実装で、このプロパティには以下が含まれます。

- **java.naming.factory.initial**: 最初に使うコンテキストファクトリを指定するための環境プロパティ名。プロパティの値は、最初のコンテキストを作成するファクトリクラスの完全修飾クラス名にする必要があります。これが指定されていない場合、**InitialContext** オブジェクトが作成されると**javax.naming.NoInitialContextException** がスローされます。
- **java.naming.provider.url**: クライアントが使う JBoss JNDI サービスプロバイダーの場所を指定するための環境プロパティ名。**NamingContextFactory** クラスはこの情報を使いどの JBossNS サーバーに接続するかを把握します。このプロパティの値はURL文字列としてください。JBossNSでは、URL 形式は**jnp://host:port/[jndi\_path]**となっています。URL の **jnp:** の部分はプロトコルで、これは JBoss が使うソケット/RMI ベースのプロトコルを参照します。URL の **jndi\_path** の部分は、root コンテキストを起点としたオプションのJNDI名となっています (例: **apps** や **apps/tmp**)。ホストコンポーネント以外はすべてオプションとなっています。デフォルトのポート値が 1099 であるため、以下の例はすべて同じです。
  - **jnp://www.jboss.org:1099/**
  - **www.jboss.org:1099**
  - **www.jboss.org**
- **java.naming.factory.url.pkgs**: URL コンテキストファクトリでローディングする際に使うパッケージプレフィックス一覧を指定するための環境プロパティ名。このプロパティの値はファクトリクラスのクラス名につけるプレフィックス一覧をコンマで区切るものとします。このファクトリクラスは、URL コンテキストファクトリを作成します。JBoss JNDI プロバイダーでは、これは **org.jboss.naming.org.jnp.interfaces** でなければなりません。このプロパティは、JBoss JNDI プロバイダーの**jnp:** と **java:** URL コンテキストファクトリを検索する際に必須となっています。
- **jnp.socketFactory**: ブートストラップソケットを作成するのに利用する **javax.net.SocketFactory** 実装の完全修飾クラス名。デフォルト値は、**org.jnp.interfaces.TimedSocketFactory**で、**TimedSocketFactory** は、接続と読み込みタイムアウトの使用に対応する単純な **SocketFactory** 実装となっています。この2つのプロパティは以下により指定されます。
- **jnp.timeout**: 接続のタイムアウト(ミリ秒単位)。デフォルト値が0の場合、VM TCP/IP がタイムアウトするまで、接続が停止されます。
- **jnp.sotimeout**: 接続されたソケットの読み込みタイムアウト (ミリ秒単位)。デフォルト値が0の場合、読み込みが停止します。これは新しく接続されたソケットで**Socket.setSoTimeout** に渡される値となっています。

クライアントがこれらの利用可能なJBossNS プロパティを使って **InitialContext** を作成すると**org.jnp.interfaces.NamingContextFactory** オブジェクトを使い今後の操作で利用される**Context** インスタンスを作成します。**NamingContextFactory** は **javax.naming.spi.InitialContextFactory** インターフェースのJBossNS 実装となっており、**NamingContextFactory** クラスに**Context**を作成が求められると、グローバル JNDI 名前空間にあるコンテキスト名と**InitialContext** 環境を使い **org.jnp.interfaces.NamingContext** インスタンス

スを作成します。**NamingContext** インスタンスが実際にJBossNS サーバーに接続するタスクを実行し、**Context** インターフェースを実装しています。この環境からの**Context.PROVIDER\_URL** 情報は、どのサーバーから**NamingServer** RMI 参照を取得するか示しています。

**NamingContext** インスタンスと**NamingServer** インスタンスの紐付けは、最初の **Context** 操作が実行される際に遅延モードで行われます。**Context** 操作が実行され**NamingContext** に紐付けられている **NamingServer** がない場合、その環境プロパティが**Context.PROVIDER\_URL**を定義しているかどうか確認します。**Context.PROVIDER\_URL** は、**Context** が使う予定の JBossNS サーバーのホストとポートを定義します。プロバイダ URL がある場合、**NamingContext** は**NamingContext** クラスの静的マップを確認することで、まずホストとポートの組み合わせで入力される **Naming** インスタンスがすでに作成されていないか確認します。ホストとポートのペアでインスタンスがすでに取得されている場合、単に既存の**Naming** インスタンスを使います。該当のホストとポートに対して、**Naming** インスタンスがすでに作成されていない場合、**NamingContext** は**java.net.Socket**を使ってホストとポートを接続し、ソケットから**java.rmi.MarshalledObject**を読み込み、get メソッドを呼び出すことでサーバーから**Naming** RMI スタブをリトリブします。新しく取得した**Naming** インスタンスは、ホストとポートの組み合わせ配下にある **NamingContext** サーバーマップでキャッシュ化されます。コンテキストが紐付けられた JNDI 環境にてプロバイダー URL が指定されていない場合、**NamingContext** は単に**Main** MBean で設定された VM **Naming** インスタンスを使用します。

**Context** インターフェースの **NamingContext** 実装は、全操作を **NamingContext** に紐付けられた **Naming** インスタンスに委譲します。**Naming** インターフェースを実装する **NamingServer** クラスは**Context** ストアとして **java.util.Hashtable** を使用します。特定の JBossNS サーバーに対する JNDI 名はそれぞれ違っており、各 JNDI 名に固有の**NamingServer**が1つずつ存在します。**NamingServer** インスタンスを参照する時点で有効になる一時的な **NamingContext** インスタンスは0またはそれ以上存在します。**NamingContext** の目的は、**NamingContext**に渡される JNDI 名の変換を管理する **Naming** インターフェースアダプターに対し **Context** として動作することです。JNDI名は相対あるいは URL になり得るので、参照先の JBossNS サーバーのコンテキスト内で絶対名に変換される必要があります。この変換は**NamingContext** の主要機能となります。

### 8.3.2. org.jboss.naming.NamingContextFactory

このバージョンの **InitialContextFactory** 実装は jnp バージョンの単純拡張で、jnp バージョンと違うのは、パブリックスレッドのローカル変数内で

**InitialContextFactory.getInitialContext(Hashtable env)** メソッドに渡す最後の設定を格納する点です。これは、**UserTransaction** ファクトリなどのEJB ハンドルやその他の JNDI を検知するオブジェクトは作成された時点で効力を発揮する JNDI コンテキストを継続的に追跡しますが、この実装はこれらのEJBハンドルやその他のJNDI-sensitive オブジェクトにより利用されます。vm 境界を越えシリアル化された後でも、この環境をオブジェクトにバインドさせたい場合、**org.jboss.naming.NamingContextFactory**を利用してください。現在のVM **jndi.properties** あるいはシステムプロパティにて定義された環境が必要な場合は、**org.jnp.interfaces.NamingContextFactory** のバージョンを使ってください。

### 8.3.3. クラスター環境でのネーミングディスカバリ

クラスター化された JBoss 環境で実行している場合、**Context.PROVIDER\_URL** 値を指定せず、クライアントが利用可能なネーミングサービスをネットワークにクエリするよう選択することができます。これは、**all** サーバードプロファイルあるいは

は、**org.jboss.ha.framework.server.ClusterPartition** と

**org.jboss.ha.jndi.HANamingService** がデプロイされている同等のサーバードプロファイルで稼働している JBoss サーバーでのみ機能します。このディスカバリプロセスでは、ディスカバリアドレス/ポートへマルチキャストリクエストパケットを送信し、ノードからの応答を待機します。この応答は、**Naming** インターフェースの HA-RMI バージョンとなっています。以下の **InitialContext** プロパティにより、ディスカバリ設定が変わってきます。

- **jnp.partitionName**: クラスターパーティション名のディスカバリは制限される必要があります。複数のクラスターを持つ環境で稼働している場合、特定のクラスターを対象にネーミングディスカバリを行いたい場合があるかもしれません。デフォルト値なし。これは、クラスターの応答はどれでも受け付けることになります。
- **jnp.discoveryGroup**: ディスカバリクエリの送信先マルチキャスト IP アドレス。デフォルトは、230.0.0.4。
- **jnp.discoveryPort**: ディスカバリクエリの送信先ポート。デフォルトは 1102。
- **jnp.discoveryTimeout**: ディスカバリクエリの応答待機時間 (ミリ秒単位)。デフォルト値は、5000 (5 秒)。
- **jnp.disableDiscovery**: ディスカバリプロセスを回避する必要がある場合にたてるフラグ。 **Context.PROVIDER\_URL** が指定されていない場合、あるいは指定 URL に有効なネーミングサービスが見付からない場合、ディスカバリは起こります。 **jnp.disableDiscovery** フラグが true の場合、ディスカバリは試行されません。

### 8.3.4. HTTP InitialContext Factory 実装

JNDI ネーミングサービスは HTTP 経由でアクセス可能です。JNDI クライアントから見ると、これは JNDI **Context** インターフェースを継続利用しており、透過的な変更になります。**Context** インターフェース経由の操作は、(JMX 呼び出し操作を使ってリクエストを NamingService に渡す) サブレットへの HTTP ポストに変換されます。HTTP をアクセスプロトコルとして使用する利点には、HTTP を許可する設定のプロキシやファイアウォールを通過するアクセスの他、セキュリティをベースとした標準サブレットロールを使って JNDI サービスへのアクセスセキュリティを保つ機能などがあります。

HTTP で JNDI にアクセスするには、**org.jboss.naming.HttpNamingContextFactory** をファクトリ実装として利用します。このファクトリのサポート **InitialContext** 環境プロパティの完全セットは以下の通りです：

- **java.naming.factory.initial**: 初期コンテキストファクトリを指定する環境プロパティ名。**org.jboss.naming.HttpNamingContextFactory** でなければなりません。
- **java.naming.provider.url** (あるいは **Context.PROVIDER\_URL**): これは JNDI ファクトリの HTTP URL に設定する必要があります。完全な HTTP URL は、JBoss サブレットコンテナの公開 URL に **/invoker/JNDIFactory** を付け加えたものになるでしょう。例は以下の通りです。
  - **http://www.jboss.org:8080/invoker/JNDIFactory**
  - **http://www.jboss.org/invoker/JNDIFactory**
  - **https://www.jboss.org/invoker/JNDIFactory**

最初の例は、ポート 8080 を使ってサブレットにアクセスし、2 番目は標準の HTTP ポート 80 を使い、3 番目は SSL 暗号化接続を使い標準の HTTPS ポート 443 へアクセスします。

- **java.naming.factory.url.pkgs**: 全 JBoss JNDI プロバイダーに対し **org.jboss.naming:org.jnp.interfaces** にしなければなりません。このプロパティは、JBoss JNDI プロバイダーの **jnp:** や **java:** URL コンテキストファクトリの場所を検索するのに必要となります。

**HttpNamingContextFactory** によって返された JNDI **Context** 実装は、JMX バスを介して呼出しを **NamingService** への転送し、HTTP 経由で返信をマーシャルするブリッジサブレットに呼出し

を委譲するプロキシです。このプロキシが動作するには、ブリッジサーブレットの URL は何かを把握している必要があります。この値は、JBoss Webサーバーがよく知られている公開インターフェースの場合、サーバー側ですでにバインドされている場合もあります。JBoss Web サーバーがファイアウォールやプロキシ 1 つ以上を介す場合、プロキシは必要な URL はどれか把握できません。このような場合は、プロキシをクライアント VM にて設定する必要があるシステムプロパティ値と関連付けます。HTTP 上で JNDI の操作に関する詳細情報は、「[HTTP 経由で JNDI にアクセス](#)」を参照してください。



#### 注記

クラスターのパーティションがデフォルトのパーティション名を利用する場合、ディスカバリプロセスは他のクラスターを無視します。そのため、複数のクラスターを利用する場合は、一意のパーティション名 `props.put("jnp.partitionName", "ClusterBPartition")` を指定するようにしてください。

### 8.3.5. Login InitialContext Factory 実装

JAAS は、リモートクライアントの認証メソッドとして適しています。しかし、JAAS を使わない他のアプリケーションサーバー環境からの移行が簡素化、容易化されるよう、JBoss では **InitialContext** を使うことでセキュリティ証明を渡すことができます。JAAS はバックグラウンドで使用されていますが、クライアントアプリケーションで JAAS インターフェースは明示的に利用されません。

この機能を提供するファクトリクラスは、

**org.jboss.security.jndi.LoginInitialContextFactory** で、このファクトリのサポート **InitialContext** 環境プロパティの全セットは以下の通りです。

- **java.naming.factory.initial**: 最初のコンテキストファクトリを指定するための環境プロパティ名。 **org.jboss.security.jndi.LoginInitialContextFactory** でなければなりません。
- **java.naming.provider.url**: **NamingContextFactory** プロバイダー URL に設定する必要があります。 **LoginInitialContext** は単に、既存の **NamingContextFactory** の動作に JAAS ログインを追加する **NamingContextFactory** のラッパーというだけのことです。
- **java.naming.factory.url.pkgs**: 全 JBoss JNDI プロバイダーに対し **org.jboss.naming:org.jnp.interfaces** にしなければなりません。このプロパティは、JBoss JNDI プロバイダーの **jnp:** や **java:** URL コンテキストファクトリの場所を検索するのに必要となります。
- **java.naming.security.principal** (あるいは **Context.SECURITY\_PRINCIPAL**): 認証主体。これは **java.security.Principal** 実装あるいは主体名を示す文字列のいずれかです。
- **java.naming.security.credentials** (あるいは **Context.SECURITY\_CREDENTIALS**): 主体認証に使われる証明書。例: パスワード、セッションキーなど
- **java.naming.security.protocol**: (**Context.SECURITY\_PROTOCOL**) : JAAS ログインモジュール名を提供し主体や証明書の認証に使用します。

### 8.3.6. ORBInitialContextFactory

Sun の CosNaming を使用する場合、デフォルトとは違ったネーミングコンテキストファクトリを使用する必要があります。CosNaming は **deploy/iiop-service.xml?** で設定される ORB を使用せずに JNDI で ORB を検索します。 **org.jboss.iiop.naming.ORBInitialContextFactory** にグローバ

ルなコンテキストファクトリを設定する必要があります。これにより ORB を JBoss の ORB に設定します。この設定は **conf/jndi.properties** ファイル内で行います。

```
# DO NOT EDIT THIS FILE UNLESS YOU KNOW WHAT YOU ARE DOING
#
java.naming.factory.initial=org.jboss.iiop.naming.ORBInitialContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

また、アプリケーションクライアントで CosNaming を利用する場合、**ORBInitialContextFactory** を利用する必要があります。

## 8.4. HTTP 経由の JNDI

ソケットブートストラッププロトコルを持つレガシーな RMI/JRMP に加え、JBoss は HTTP 経由の JNDI ネーミングサービスへのアクセスに対するサポートも提供しています。

### 8.4.1. HTTP 経由で JNDI にアクセス

この機能は、**http-invoker.sar** により提供されており、**http-invoker.sar** の構造は以下の通りです。

```
http-invoker.sar
+- META-INF/jboss-service.xml
+- invoker.war
| +- WEB-INF/jboss-web.xml
| +- WEB-
INF/classes/org/jboss/invoke/http/servlet/InvokerServlet.class
| +- WEB-
INF/classes/org/jboss/invoke/http/servlet/NamingFactoryServlet.class
| +- WEB-
INF/classes/org/jboss/invoke/http/servlet/ReadOnlyAccessFilter.class
| +- WEB-INF/classes/roles.properties
| +- WEB-INF/classes/users.properties
| +- WEB-INF/web.xml
| +- META-INF/MANIFEST.MF
+- META-INF/MANIFEST.MF
```

**jboss-service.xml** 記述子は、**HttpInvoker** と **HttpInvokerHA** MBean を定義します。これらのサービスは、HTTP経由でJMX バスにある適切な対象の MBean へ送信されるメソッド呼出しルーティングを処理します。

**http-invoker.war** Web アプリケーションには、HTTP トランスポートの詳細を処理するサーブレットが含まれています。**NamingFactoryServlet** は、JBoss JNDIネーミングサービス **javax.naming.Context** 実装に対する作成リクエストを処理します。**InvokerServlet** は、RMI/HTTP クライアントによる呼出しを処理します。**ReadOnlyAccessFilter** により、JNDIネーミングサービスのセキュリティを確保し、未認証のクライアントに読み取り専用のアクセスを提供する JNDI コンテキストを作成することができるようになります。



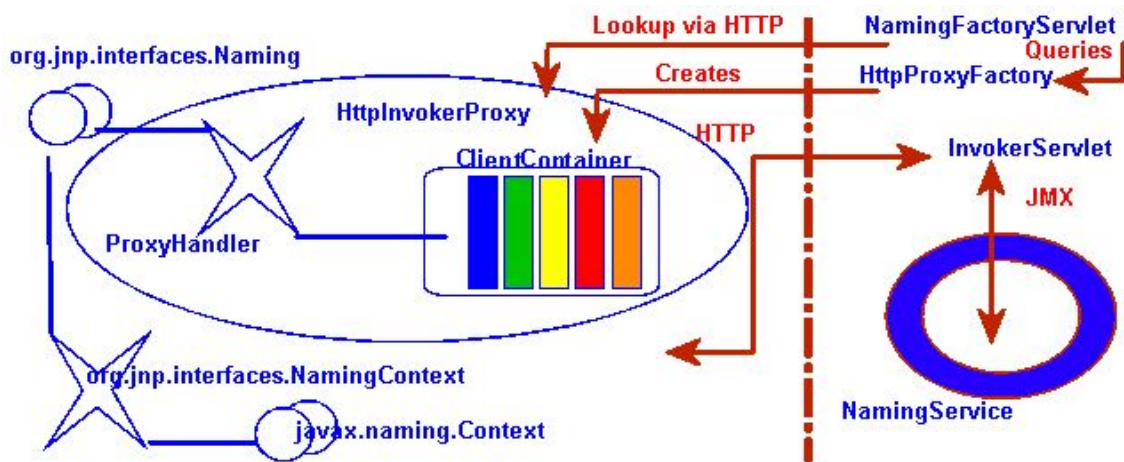


図8.2 JNDI コンテキストに対する HTTP invoker のプロキシ／サーバーの構成

設定について説明する前に、**http-invoker** サービスの操作について見てみましょう。図8.2「JNDI コンテキストに対する HTTP invoker のプロキシ／サーバーの構成」で、JBoss JNDIプロキシの構造とJBoss サーバー側の**http-invoker**コンポーネントとの関係に関する論理的な見解を示しています。このプロキシは、**Context.INITIAL\_CONTEXT\_FACTORY**プロパティが**org.jboss.naming.HttpNamingContextFactory**に設定され、**Context.PROVIDER\_URL**プロパティが**NamingFactoryServlet**のHTTP URL に設定されている**InitialContext**を使い、**NamingFactoryServlet** から取得します。結果として出たプロキシは **Context** 実装を提供する**org.jnp.interfaces.NamingContext** インスタンスに組み込まれます。

プロキシは**org.jboss.invocation.http.interfaces.HttpInvokerProxy**のインスタンスで、**org.jnp.interfaces.Naming**インターフェースを実装します。内部的に**HttpInvokerProxy**は、HTTP ポスト経由で **Naming** インターフェースメソッド呼出しを**InvokerServlet** にマーシャルする invoker を含みます。**InvokerServlet** は、これらのポストを JMX 呼出しを**NamingService**に変換し、HTTP ポストのレスポンスにて呼出しの応答をプロキシに返します。

設定値によっては、これらのコンポーネントをすべて連携させるよう設定する必要があります。図8.3「設定ファイルと JNDI/HTTP コンポーネント間の関係」で、設定ファイルと該当のコンポーネントの関係を示しています。



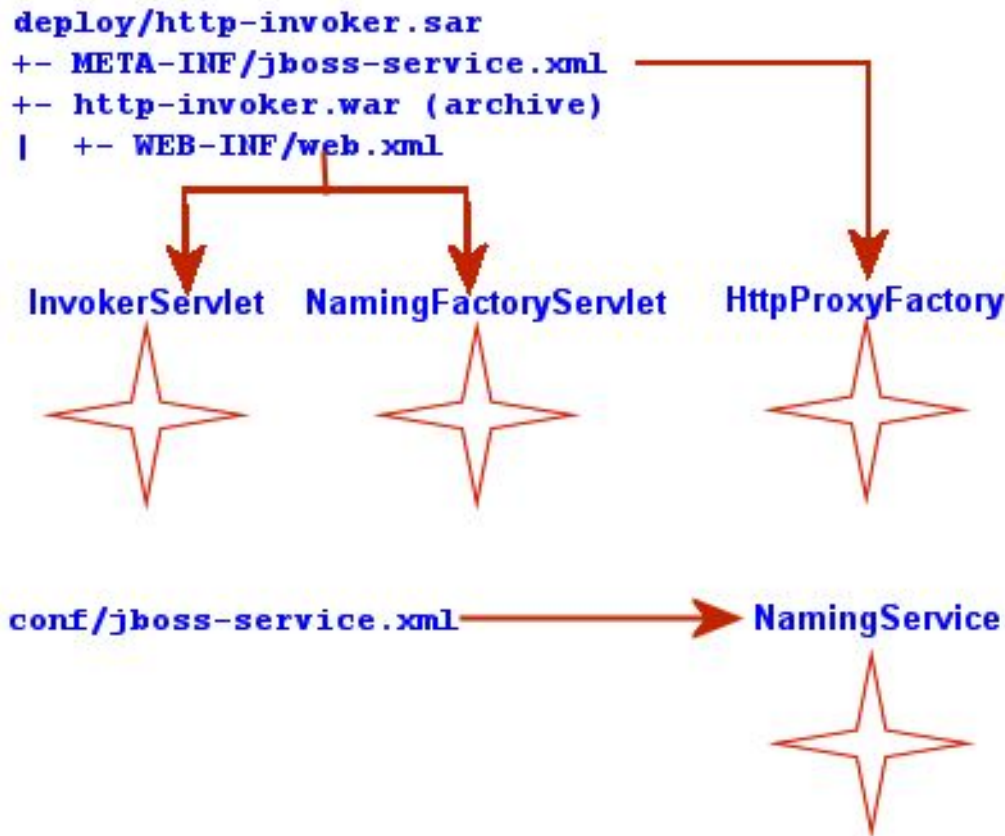


図8.3 設定ファイルと JNDI/HTTP コンポーネント間の関係

http-invoker.sar/META-INF/jboss-service.xml 記述子

は、NamingServiceのHttpInvokerProxy を作成するHttpProxyFactory を定義します。HttpProxyFactory に対する設定が必要な属性には以下が含まれます：

- **InvokerName:** conf/jboss-service.xml記述子にて定義される NamingService のJMX ObjectName。JBoss ディストリビューションで使用する標準設定は jboss:service=Naming。
- **InvokerURL** あるいは **InvokerURLPrefix + InvokerURLSuffix + UseHostName** : 完全な HTTP URL をInvokerURL 属性を使い、InvokerServlet に指定することができます。あるいは、URLのホスト名に依存しない部分を特定し、HttpProxyFactory を入力できます。**InvokerURL**値の例として、http://jboss-host1.dot.com:8080/invoker/JMXInvokerServletなどが挙げられます。これは以下に分割できます。
  - **InvokerURLPrefix:** ホスト名の前につく URL のプレフィックス。通常、http:// あるいは https:// (SSL が使われている場合) となっています。
  - **InvokerURLSuffix:** ホスト名の後につくURL のサフィックス。これには Web サーバーのポート番号、InvokerServletにデプロイされるパスが含まれます。例えば、InvokerURLSuffixのInvokerURL の値は、引用符なしの:8080/invoker/JMXInvokerServlet となります。ポート番号は Web コンテナサービス設定により決まります。InvokerServlet へのパスは、http-invoker.sar/invoker.war/WEB-INF/web.xml の記述子にて指定されます。
  - **UseHostName:** 完全 InvokerURL のホスト名部分を構築する際、ホスト名をホスト IP アドレスの位置に使用するか示すフラグ。True の場合、InetAddress.getLocalHost().getHostName メソッドが使われます。True でな

ければ、`InetAddress.getLocalHost().getHostAddress()` メソッドが利用されます。

- **ExportedInterface**: プロキシがクライアントに対し公開する `org.jnp.interfaces.Naming` インターフェース。このプロキシの実際のクライアントは、JBoss JNDI実装の `NamingContext` クラスで、JBoss JNDIプロバイダーを使う場合、JNDI クライアントは `InitialContext` ルックアップから取得します。
- **JndiName**: プロキシがバインドされる場合の JNDI の名前。このインターフェースが JNDI にバインドされないように、この項目は空白／空の文字列に設定する必要があります。JNDI を使ってそれ自体をブートストラップすることはできません。これは、`NamingFactoryServlet` の役割になります。

`http-invoker.sar/invoker.war/WEB-INF/web.xml` 記述子は、初期化パラメーターと共に `NamingFactoryServlet` と `InvokerServlet` のマッピングを定義します。JNDI/HTTP に関連する `NamingFactoryServlet` の設定は `JNDIFactory` のエントリで以下を定義します。

- **HttpProxyFactory** MBean 名をマッピングする `namingProxyMBean` の初期化パラメーター。HTTP ポストへの応答時に返す `Naming` プロキシを取得するために、`NamingFactoryServlet` がこのパラメーターを利用します。デフォルトの `http-invoker.sar/META-INF/jboss-service.xml` 設定は、名前が `jboss:service=invoker, type=http, target=Naming` となります。
- `Naming` プロキシ値をクエリするための `namingProxyMBean` 属性名を定義するプロキシ初期化パラメーター。属性名が `Proxy` にデフォルト設定されています。
- **JNDIFactory** 設定のサーブレットマッピング。セキュリティ保護されていないマッピングのデフォルト設定は `/JNDIFactory/*` となっています。これは、`http-invoker.sar/invoker.war` のコンテキスト root を起点とした相対パスでデフォルトでは、`.war` のサフィックスを取った WAR 名となっています。

JNDI/HTTP に関連する `InvokerServlet` 設定は `JMXInvokerServlet` で以下を定義します。

- **InvokerServlet** のサーブレットマッピング。セキュリティ保護されていないマッピングのデフォルト設定は `/JMXInvokerServlet/*` で、これは、`http-invoker.sar/invoker.war` のコンテキスト root を起点とした相対パスでデフォルトでは、`.war` のサフィックスを取った WAR 名となっています。

## 8.4.2. HTTPS で JNDI にアクセス

HTTP/SSL 経由で JNDI にアクセスできるようにするには、Web コンテナにある SSL コネクタを有効にする必要があります。この詳細については「Tomcat 用にサーブレットコンテナを統合する」で説明しています。HTTPS URL を JNDI プロバイダー URL として使用するクライアントの簡単な例を使って HTTPS の使いかたを説明します。この例には SSL コネクタの設定を用いますので、SSL コネクタの設定に関する詳細を除き必要なものはすべて含まれています。

また、HTTPS URL を使用するよう `HttpProxyFactory` 設定構成も提供しています。次の例ではこの設定を提供するためのサンプル例をインストールする `http-invoker.sar/jboss-service.xml` 記述子のセクションを示します。標準 HTTP 設定に相対的に変更されたものは `InvokerURLPrefix` および `InvokerURLSuffix` の属性のみで 8443 ポートを使って HTTPS URL を設定します。

```
<!-- Expose the Naming service interface via HTTPS -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
      name="jboss:service=invoker,type=https,target=Naming">
```

```

<!-- The Naming service we are proxying -->
<attribute name="InvokerName">jboss:service=Naming</attribute>
<!-- Compose the invoker URL from the cluster node address -->
<attribute name="InvokerURLPrefix">https://</attribute>
<attribute name="InvokerURLSuffix">:8443/invoker/JMXInvokerServlet
</attribute>
<attribute name="UseHostName">true</attribute>
<attribute name="ExportedInterface">org.jnp.interfaces.Naming
</attribute>
<attribute name="JndiName"/>
<attribute name="ClientInterceptors">
  <interceptors>
    <interceptor>org.jboss.proxy.ClientMethodInterceptor
  </interceptor>
    <interceptor>org.jboss.proxy.SecurityInterceptor
  </interceptor>

  <interceptor>org.jboss.naming.interceptors.ExceptionInterceptor
  </interceptor>
    <interceptor>org.jboss.invocation.InvokerInterceptor
  </interceptor>
  </interceptors>
</attribute>
</mbean>

```

少なくとも、HTTPS を使った JNDI クライアントは HTTPS URL プロトコルハンドラーを設定する必要があります。HTTPS 用の Java Secure Socket Extension (JSSE) を使います。JSSE の文書は HTTPS 使用の要件という面では詳しく説明されていないため、[例8.2「トランスポートとしてHTTPSを使う JNDI クライアント」](#)にて説明されているクライアントサンプルを設定するには、以下の手順をふむ必要があります。

- HTTPS URL のプロトコルハンドラーは Java で利用できるようにする必要があります。JSSE リリースには、**com.sun.net.ssl.internal.www.protocol** パッケージの HTTPS ハンドラーがあります。HTTPS URL を利用可能にするには、標準の URL プロトコルハンドラーの検索プロパティ **java.protocol.handler.pkgs** にこのパッケージを含める必要があります。Ant スクリプトに **java.protocol.handler.pkgs** プロパティを設定します。
- SSL を機能させるには、JSSE セキュリティプロバイダーをインストールする必要があります。これは、JSSE jars を拡張パッケージとしてインストールするか、プログラムでインストールしてください。この例では煩わしい作業が少ないプログラムでの手法を使っています。**ExClient** コードの 18 行目で、実施方法が説明されています。
- JNDI プロバイダー URL は HTTPS をプロトコルとして利用する必要があります。**ExClient** コードの 24-25 行目で、ポート番号 8443 が割り当てられたローカルホストへの HTTP/SSL 接続を指定しています。このホスト名とポートは、Web コンテナの SSL コネクタで定義されています。
- サーバー証明に対する HTTPS URL のホストネームの検証は無効にする必要があります。デフォルトでは、JSSE HTTPS プロトコルハンドラーが、HTTPS URL のホスト名部分をサーバー証明の共通名をもとに厳密に検証を行います。これは、セキュリティがかかっている Web サイトに接続すると Web ブラウザーが行うチェックと同じです。特定のホスト名ではなく **"Chapter 8 SSL Example"** の共通名を使用する自己署名サーバー証明を採用しており、これは開発環境やイントラネットで一般的に利用されている場合が多いです。JBoss **HttpInvokerProxy** は、**org.jboss.security.ignoreHttpsHost** システムプロパティが

存在するか、さらにTrueの値を持っているかを確認するデフォルトのホスト名をオーバーライドします。Ant スクリプトでは`org.jboss.security.ignoreHttpsHost` プロパティをTrueに設定します。

### 例8.2 トランスポートとしてHTTPSを使う JNDI クライント

```
package org.jboss.chap3.ex1;

import java.security.Security;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[]) throws Exception
    {
        Properties env = new Properties();
        env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
            "org.jboss.naming.HttpNamingContextFactory");
        env.setProperty(Context.PROVIDER_URL,
            "https://localhost:8443/invoker/JNDIFactorySSL");

        Context ctx = new InitialContext(env);
        System.out.println("Created InitialContext, env=" + env);

        Object data = ctx.lookup("jmx/invoker/RMIAdaptor");
        System.out.println("lookup(jmx/invoker/RMIAdaptor): " + data);
    }
}
```

クライアントのテストを実施するには、まず3章の例を構築し **chap3** 設定ファイルセットを作成します。

```
[examples]$ ant -Dchap=naming config
```

次に、**naming** 設定ファイルセットを使って JBoss サーバーを起動します。

```
[bin]$ sh run.sh -c naming
```

最後に、以下を使い **ExClient** を実行します。

```
[examples]$ ant -Dchap=naming -Dex=1 run-example
...
run-example1:

[java] Created InitialContext, env={java.naming. \
provider.url=https://localhost:8443/invoker/JNDIFactorySSL, java.naming. \
factory.initial=org.jboss.naming.HttpNamingContextFactory}
[java] lookup(jmx/invoker/RMIAdaptor): org.jboss.invocation.jrmp. \
interfaces.JRMPInvokerP
roxy@cac3fa
```

### 8.4.3. HTTP 経由の JNDI アクセスでセキュリティを確保

HTTP 経由で JNDI にアクセスする利点の 1 つに、標準の Web 宣言セキュリティを使うことで JNDI **InitialContext** ファクトリやネーミング操作へ簡単にかつセキュアにアクセスできる点が挙げられます。サーバー側で JNDI/HTTP トランスポートの処理が 2 つのサーブレットで実装されているため、これが可能となっています。前述したように、これらのサーブレットは、**default** と **all** サーバースタックファイルの deploy ディレクトリにある **http-invoker.sar/invoker.war** ディレクトリに含まれています。**invoker.war/WEB-INF/web.xml** の記述子を編集し、セキュリティ保護がされていないサーブレットマッピングを削除することで、JNDI へのセキュアなアクセスが可能になります。例えば、[例8.3「JNDI サーブレットへのアクセスをセキュアにするための web.xml 記述子の例」](#)の **web.xml** 記述子では、ユーザ認証がされており、**HttpInvoker** のロールを持つ場合のみ、**invoker.war** サーブレットへのアクセスが可能です。

#### 例8.3 JNDI サーブレットへのアクセスをセキュアにするための web.xml 記述子の例

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <!-- ### Servlets -->
  <servlet>
    <servlet-name>JMXInvokerServlet</servlet-name>
    <servlet-class>
      org.jboss.invocation.http.servlet.InvokerServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet>
    <servlet-name>JNDIFactory</servlet-name>
    <servlet-class>
      org.jboss.invocation.http.servlet.NamingFactoryServlet
    </servlet-class>
    <init-param>
      <param-name>namingProxyMBean</param-name>
      <param-
value>jboss:service=invoker,type=http,target=Naming</param-value>
    </init-param>
    <init-param>
      <param-name>proxyAttribute</param-name>
      <param-value>Proxy</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>
  <!-- ### Servlet Mappings -->
  <servlet-mapping>
    <servlet-name>JNDIFactory</servlet-name>
    <url-pattern>/restricted/JNDIFactory/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>JMXInvokerServlet</servlet-name>
    <url-pattern>/restricted/JMXInvokerServlet/*</url-pattern>
  </servlet-mapping>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>HttpInvokers</web-resource-name>
```

```

        <description>An example security config that only allows
users with
        the role HttpInvoker to access the HTTP invoker
servlets </description>
        <url-pattern>/restricted/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>HttpInvoker</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>JBoss HTTP Invoker</realm-name>
</login-config>    <security-role>
    <role-name>HttpInvoker</role-name>
</security-role>
</web-app>

```

**web.xml** 記述子が定義するのは、どのサーブレットがセキュアか、どのロールであればセキュリティ保護されているサーブレットにアクセス可能かのみです。これに加え、war 認証、権限付与を処理するセキュリティドメインを定義する必要があります。定義方法は**jboss-web.xml** 記述子で行い、**http-invoker**セキュリティドメインの使用例を以下に示しています。

```

<jboss-web>
    <security-domain>java:/jaas/http-invoker</security-domain>
</jboss-web>

```

**security-domain** 要素は、認証、権限付与に使う JAAS ログインモジュール設定用のセキュリティドメイン名を定義しています。

#### 8.4.4. セキュリティ保護されていないコンテキストを読み取り専用にして JNDI へのセキュアなアクセスを確保

JNDI/HTTP ネーミングサービスで利用できる別の機能として、未認証のユーザには読み取り専用モードでアクセスできるコンテキストを定義する機能です。認証層を使うサービスには、これは重要となる可能性があります。例えば、**SRPLoginModule** は認証に利用する SRP サーバーインターフェースを検索する必要があります。本項ではここから、JBoss Enterprise Application Platform においてどのように読み取り専用がどのように動作するかを説明していきます。

まず、**invoker.sar/WEB-INF/web.xml**で**ReadOnlyJNDIFactory**を宣言すると、**/invoker/ReadOnlyJNDIFactory**にマッピングします。

```

<servlet>
    <servlet-name>ReadOnlyJNDIFactory</servlet-name>
    <description>A servlet that exposes the JBoss JNDI Naming service stub
        through http, but only for a single read-only context. The
return content
        is serialized MarshalledValue containing the
org.jnp.interfaces.Naming
        stub.
    </description>

```

```

    <servlet-
class>org.jboss.invocation.http.servlet.NamingFactoryServlet</servlet-
class>
    <init-param>
        <param-name>namingProxyMBean</param-name>
        <param-
value>jboss:service=invoker, type=http, target=Naming, readonly=true</param-
value>
    </init-param>
    <init-param>
        <param-name>proxyAttribute</param-name>
        <param-value>Proxy</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

<!-- ... -->

<servlet-mapping>
    <servlet-name>ReadOnlyJNDIFactory</servlet-name>
    <url-pattern>/ReadOnlyJNDIFactory/*</url-pattern>
</servlet-mapping>

```

このファクトリは、invoker へ接続する必要がある JNDI スタブのみを提供します。ここでは、invoker は **jboss:service=invoker, type=http, target=Naming, readonly=true** となっており、**http-invoker.sar/META-INF/jboss-service.xml** ファイルにて宣言されます。

```

    <mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
        name="jboss:service=invoker, type=http, target=Naming, readonly=true">
        <attribute name="InvokerName">jboss:service=Naming</attribute>
        <attribute name="InvokerURLPrefix">http://</attribute>
        <attribute
name="InvokerURLSuffix">:8080/invoker/readonly/JMXInvokerServlet</attribut
e>
        <attribute name="UseHostName">true</attribute>
        <attribute
name="ExportedInterface">org.jnp.interfaces.Naming</attribute>
        <attribute name="JndiName"></attribute>
        <attribute name="ClientInterceptors">
            <interceptors>

<interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.naming.interceptors.ExceptionInterceptor</intercept
or>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
            </interceptors>
        </attribute>
    </mbean>

```

クライアント側のプロキシは、サーバー側にある特定の invoker サブレットに応答する必要があります。ここでの設定は**/invoker/readonly/JMXInvokerServlet**への実際の呼出しが含まれていま

す。実際には、これは読み取り専用フィルタが付けられた標準の **JMXInvokerServlet** となっています。

```
<filter>
  <filter-name>ReadOnlyAccessFilter</filter-name>
  <filter-
class>org.jboss.invocation.http.servlet.ReadOnlyAccessFilter</filter-
class>
  <init-param>
    <param-name>readOnlyContext</param-name>
    <param-value>readonly</param-value>
    <description>The top level JNDI context the filter will
enforce
operations
or
this
unrestricted
    read-only access on. If specified only Context.lookup
    will be allowed on this context. Another other operations
    lookups on any other context will fail. Do not associate
    filter with the JMXInvokerServlets if you want
    access. </description>
  </init-param>
  <init-param>
    <param-name>invokerName</param-name>
    <param-value>jboss:service=Naming</param-value>
    <description>The JMX ObjectName of the naming service mbean
</description>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>ReadOnlyAccessFilter</filter-name>
  <url-pattern>/readonly/*</url-pattern>
</filter-mapping>

<!-- ... -->
<!-- A mapping for the JMXInvokerServlet that only allows invocations
      of lookups under a read-only context. This is enforced by the
      ReadOnlyAccessFilter
      -->
<servlet-mapping>
  <servlet-name>JMXInvokerServlet</servlet-name>
  <url-pattern>/readonly/JMXInvokerServlet/*</url-pattern>
</servlet-mapping>
```

**readOnlyContext** パラメーターが**readonly**に設定されていると、**ReadOnlyJNDIFactory**から JBoss にアクセスする場合、**readonly** コンテキストのデータにのみアクセスができるようになります。使用方法を示すコードを以下に示します。

```
Properties env = new Properties();
env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
    "org.jboss.naming.HttpNamingContextFactory");
env.setProperty(Context.PROVIDER_URL,
    "http://localhost:8080/invoker/ReadOnlyJNDIFactory");
```



```
Context ctx2 = new InitialContext(env);
Object data = ctx2.lookup("readonly/data");
```

読み取り専用コンテキスト以外のオブジェクトをルックアップしようとするとう失敗します。JBoss には **readonly** コンテキストのデータは含まれていませんので、ご自身で作成しない限り **readonly** コンテキストの使い道はありません。

## 8.5. その他のネーミング MBean

JBoss 内の JBossNS サーバーに組み込み設定されている **NamingService** MBean 以外に、JBoss に同梱されているネーミング関連の MBean サービスがいくつか存在します。 **JndiBindingServiceMgr**、 **NamingAlias**、 **ExternalContext**、 **JNDIView** などとなっています。

### 8.5.1. JNDI バインディングマネージャー

JNDI バインディングマネージャーサービスにより、アプリケーションコードで利用できるよう JNDI にオブジェクトを素早くバインドできるようになります。バインディングサービスの MBean クラスは **org.jboss.naming.JNDIBindingServiceMgr** となっており、 **BindingsConfig** 属性を1つ含んでいます。この属性は、 **jndi-binding-service\_1\_0.xsd** スキーマをに基づいた XML ドキュメントを受付けます。 **BindingsConfig** 属性のコンテンツは JbossXB フレームワークを使いアンマッシュルされます。以下は MBean の定義で、JNDI バインディングマネージャーサービスの最も基本的なフォーム利用について示しています。

```
<mbean code="org.jboss.naming.JNDIBindingServiceMgr"
      name="jboss.tests:name=example1">
  <attribute name="BindingsConfig" serialDataType="jbx"b">
    <jndi:bindings xmlns:xs="http://www.w3.org/2001/XMLSchema-
instance"
                  xmlns:jndi="urn:jboss:jndi-binding-service:1.0"
                  xs:schemaLocation="urn:jboss:jndi-binding-service
\
      resource:jndi-binding-service_1_0.xsd">
      <jndi:binding name="bindexample/message">
        <jndi:value trim="true">
          Hello, JNDI!
        </jndi:value>
      </jndi:binding>
    </jndi:bindings>
  </attribute>
</mbean>
```

これは、JNDI 名 **bindexample/message** でテキスト文字列 **"Hello, JNDI!"** をバインドします。アプリケーションは、他の JNDI 値の場合と同様にこの値を検索します。 **trim** 属性は、前後のホワイトスペースを無視するよう指定します。この属性は単に説明のためだけに利用しているため、デフォルト値は **True** となっています。

```
InitialContext ctx = new InitialContext();
String          text = (String) ctx.lookup("bindexample/message");
```

文字列の値自体はあまり興味深いものではありません。JavaBeans プロパティエディターを利用できる場合、 **type** 属性を使って希望のクラス名を指定することができます。

```
<jndi:binding name="urls/jboss-home">
  <jndi:value type="java.net.URL">http://www.jboss.org</jndi:value>
</jndi:binding>
```

**editor** 属性を使って、使用するプロパティエディターを指定することができます。

```
<jndi:binding name="hosts/localhost">
  <jndi:value editor="org.jboss.util.propertyeditor.InetAddressEditor">
    127.0.0.1
  </jndi:value>
</jndi:binding>
```

より複雑な構造については、BossXB対応のスキーマを使う場合もあります。以下の例では、**java.util.Properties** オブジェクトのマッピング方法を説明しています。

```
<jndi:binding name="maps/testProps">
  <java:properties xmlns:java="urn:jboss:java-properties"
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xs:schemaLocation="urn:jboss:java-properties \
resource:java-properties_1_0.xsd">
    <java:property>
      <java:key>key1</java:key>
      <java:value>value1</java:value>
    </java:property>
    <java:property>
      <java:key>key2</java:key>
      <java:value>value2</java:value>
    </java:property>
  </java:properties>
</jndi:binding>
```

### 8.5.2. org.jboss.naming.NamingAlias MBean

**NamingAlias** MBean は、単純なユーティリティサービスで、別のJNDI 名で JNDI **javax.naming.LinkRef** の形式でエイリアスを作成することができます。これは Unix ファイルシステムのシンボリックリンクに似ています。エイリアスを作成するには、**NamingAlias** MBeanの設定を **jboss-service.xml** 設定ファイルに追加します。**NamingAlias** サービスの設定属性は以下の通りです：

- **FromName:** JNDI 下で**LinkRef**がバインドされる場所
- **ToName:** エイリアスの to name。これは、**LinkRef** が参照するターゲット名となっており、この名前はURL、**InitialContext**を起点とし解決される相対名、あるいはこの名前の最初の文字がドット (.)の場合リンクがバインドされているコンテキストを起点とした相対名となっています。

以下の例では、JNDI 名**QueueConnectionFactory** を**ConnectionFactory** という名前にマッピングしています。

```
<mbean code="org.jboss.naming.NamingAlias"
name="jboss.mq:service=NamingAlias,fromName=QueueConnectionFactory">
```

```

    <attribute name="ToName">ConnectionFactory</attribute>
    <attribute name="FromName">QueueConnectionFactory</attribute>
</mbean>

```

### 8.5.3. org.jboss.naming.ExternalContext MBean

**ExternalContext** MBean は、外部の JNDI コンテキストを JBoss サーバーの JNDI 名前空間に連携することができます。外部とは、JBoss サーバー VM の内部で動作する JBossNS ネーミングサービスの外部にあるネーミングサービスを指しています。JNDI プロバイダの root コンテキストがシリアル化されていない場合でも、LDAP サーバー、ファイルシステム、DNS サーバーなどを組み込むことができます。ネーミングサービスがリモートアクセスに対応している場合、リモートクライアントに対してもこの連携を用意することができます。

外部の JNDI ネーミングサービスを組み込むには、**ExternalContext** MBean サービスの設定を **jboss-service.xml** 設定ファイルに追加する必要があります。**ExternalContext** サービスの設定可能な属性は以下の通りです。

- **JndiName**: 外部コンテキストがバインドされる JNDI 名。
- **RemoteAccess**: リモートクライアントが外部の **InitialContext** を作成できるように、**Serializable** フォームを使って外部の **InitialContext** をバインドする必要があるかを示す boolean フラグ。リモートクライアントが JBoss JNDI **InitialContext** を介して外部のコンテキストを検索する場合、**ExternalContext** MBean に渡したものと同一 env プロパティを使い、外部の **InitialContext** のインスタンスを効果的に作成します。クライアントが **new InitialContext(env)** をリモートから実行できる場合のみ、機能します。これには、コンテキストにアクセスしているリモート VM にて、env の **Context.PROVIDER\_URL** 値が解決できなければなりません。これは、LDAP の例で機能するはずです。しかし、ファイルシステムの例では、ファイルシステムのパスが共通のネットワークパスを指定していない限り機能しない場合が多いでしょう。このプロパティが設定されていないと、デフォルトで **False** に設定されます。
- **CacheContext**: **cacheContext** フラグ。True に設定されている場合、MBean の起動後 MBean が停止するまで in memory オブジェクトとして格納されているときにのみ、外部の **Context** が作成されます。**cacheContext** が **false** に設定されている場合、MBean プロパティと **InitialContext** クラスを使って検索する度に外部の **Context** が作成されます。クライアントがキャッシュされていない **Context** を検索する場合、このクライアントは **Context** 上で **close()** を呼び出し、リソース漏れを防ぐはずです。
- **InitialContext**: 使用する **InitialContext** 実装の完全修飾クラス名。以下のいずれかである必要があります: **javax.naming.InitialContext**, **javax.naming.directory.InitialDirContext** あるいは **javax.naming.ldap.InitialLdapContext**。**InitialLdapContext** の場合、null **Controls** アレイを使用します。デフォルトは、**javax.naming.InitialContext** となっています。
- **Properties**: **Properties** 属性には 外部の **InitialContext** に対する JNDI プロパティが含まれます。ここでの入力値は **jndi.properties** ファイルに入るテキストと同等のものでなければなりません。
- **PropertiesURL**: 外部のプロパティファイルから外部の **InitialContext** に対する **jndi.properties** 情報を設定します。これは、URL、文字列あるいはクラスパスのリソース名となっており、以下に例を示します。
  - **file:///config/myldap.properties**

- http://config.mycompany.com/myldap.properties
- /conf/myldap.properties
- myldap.properties

以下のMBean 定義では、**external/ldap/jboss**名の配下で、外部の LDAP コンテキストをJBoss JNDI名前空間にバインドしています。

```
<!-- Bind a remote LDAP server -->
<mbean code="org.jboss.naming.ExternalContext"

name="jboss.jndi:service=ExternalContext,jndiName=external/ldap/jboss">
  <attribute name="JndiName">external/ldap/jboss</attribute>
  <attribute name="Properties">
    java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
    java.naming.provider.url=ldap://ldaphost.jboss.org:389/o=jboss.org
    java.naming.security.principal=cn=Directory Manager
    java.naming.security.authentication=simple
    java.naming.security.credentials=secret
  </attribute>
  <attribute name="InitialContext"> javax.naming.ldap.InitialLdapContext
</attribute>
  <attribute name="RemoteAccess">true</attribute>
</mbean>
```

この設定では、以下のコードを使ってJBoss VM内から**ldap://ldaphost.jboss.org:389/o=jboss.org**にある外部のLDAP にアクセスすることができます。

```
InitialContext iniCtx = new InitialContext();
LdapContext ldapCtx = iniCtx.lookup("external/ldap/jboss");
```

この場合、**RemoteAccess** プロパティが true に設定されたため、JBoss サーバー VMの外部にある同じコードを利用すると動作します。False に設定されている場合、リモートクライアントは、外部の **InitialContext** を再構築できない **ObjectFactory** と共に**Reference** オブジェクトを受け取るため、動作しません。

```
<!-- Bind the /usr/local file system directory -->
<mbean code="org.jboss.naming.ExternalContext"

name="jboss.jndi:service=ExternalContext,jndiName=external/fs/usr/local">
  <attribute name="JndiName">external/fs/usr/local</attribute>
  <attribute name="Properties">

    java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
    java.naming.provider.url=file:///usr/local
  </attribute>
  <attribute
name="InitialContext">javax.naming.IntialContext</attribute>
</mbean>
```

この設定は、ローカルのファイルシステムディレクトリ**/usr/local** を**external/fs/usr/local**名配下にあるJBoss JNDI名前空間へバインディングしています。

この設定では、以下のコードを使って JBoss VM 内から **file:///usr/local** にある外部のファイルシステムコンテキストへアクセス可能です。

```
InitialContext iniCtx = new InitialContext();
Context ldapCtx = iniCtx.lookup("external/fs/usr/local");
```

#### 8.5.4. org.jboss.naming.JNDIView MBean

JNDIView MBean では、ユーザは JMX Agent View のインターフェースを使うことで JBoss サーバー内に存在するように、JNDI 名前空間のツリーを参照することができます。JNDIView MBean を使って JBoss JNDI 名前空間を参照するには、http インターフェースを使い JMX Agent View に接続します。これはデフォルト設定で、**http://localhost:8080/jmx-console/** となっています。このページでは、登録済みの MBean をドメイン別に分類し一覧になっているセクションが表示されます。図8.4「設定済みの JBoss MBeans を JMX Console で表示」で示すような表示になるはずです。

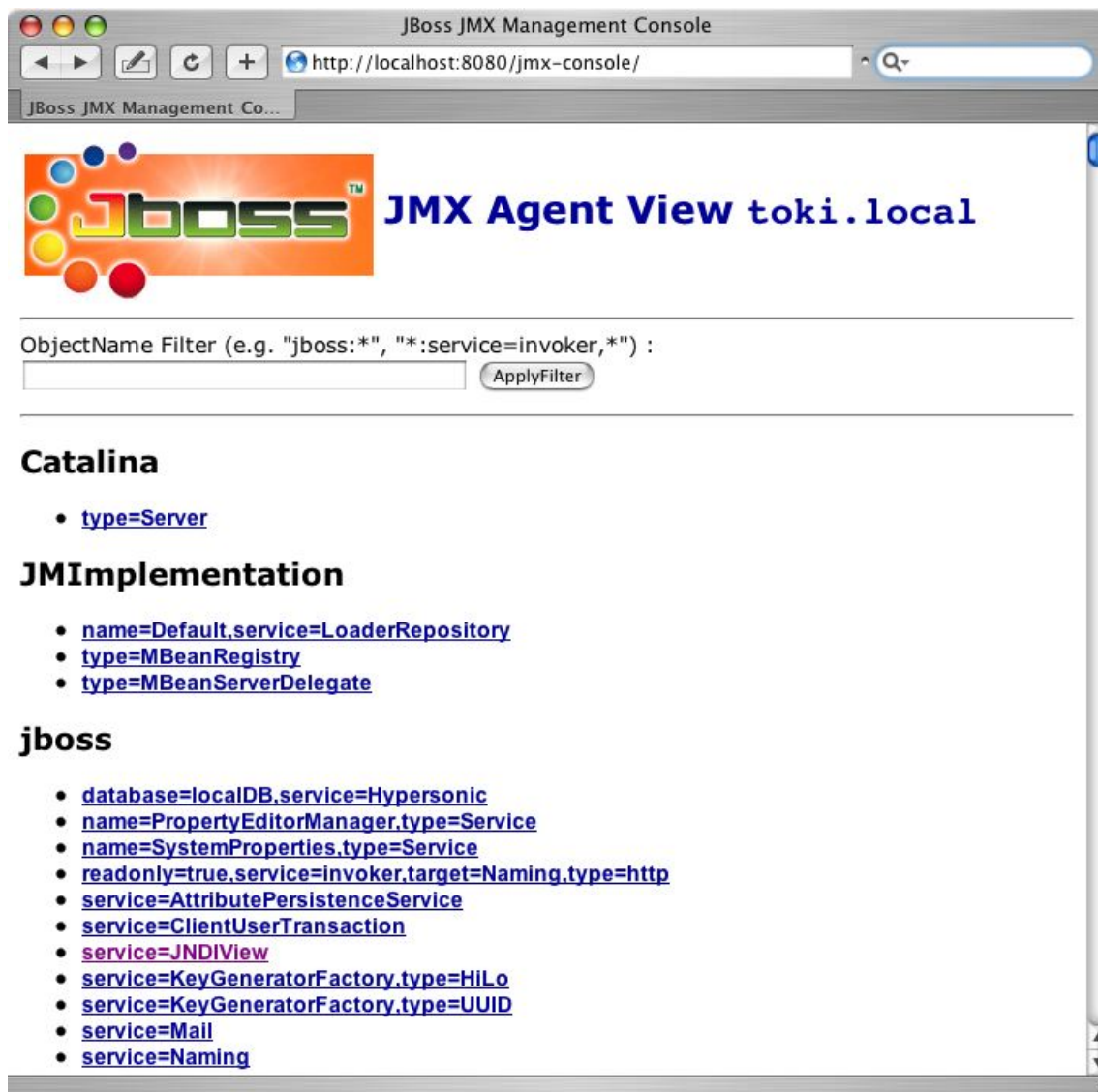


図8.4 設定済みの JBoss MBeans を JMX Console で表示

JNDIView リンクを選択すると、JNDIView MBean ビューに移動し、JNDIView MBean の操作一覧が表示されます。これは、図8.5「JNDIView MBean を JMX Console で表示」で表示されているようなビューになるはずです。

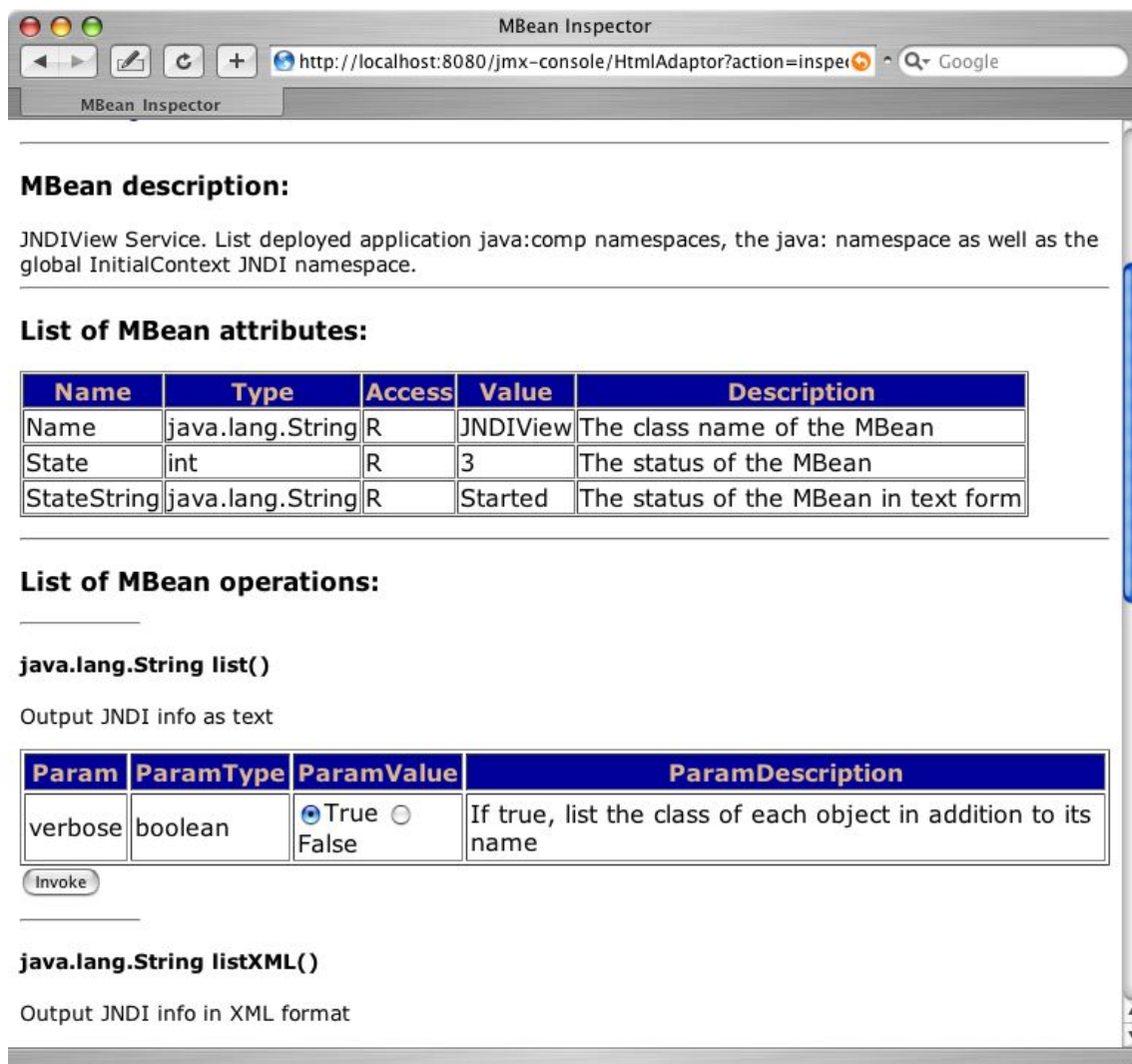


図8.5 JNDIView MBean を JMX Console で表示

リスト操作により、シンプルなテキストビューを使い、JBoss サーバーのJNDI 名前空間を HTML ページとしてダンプします。例としてリスト操作を呼び出すことで、[図8.6「JNDIView のリスト操作出力を JMX Console で表示」](#)で示されているようなビューが生成されます。



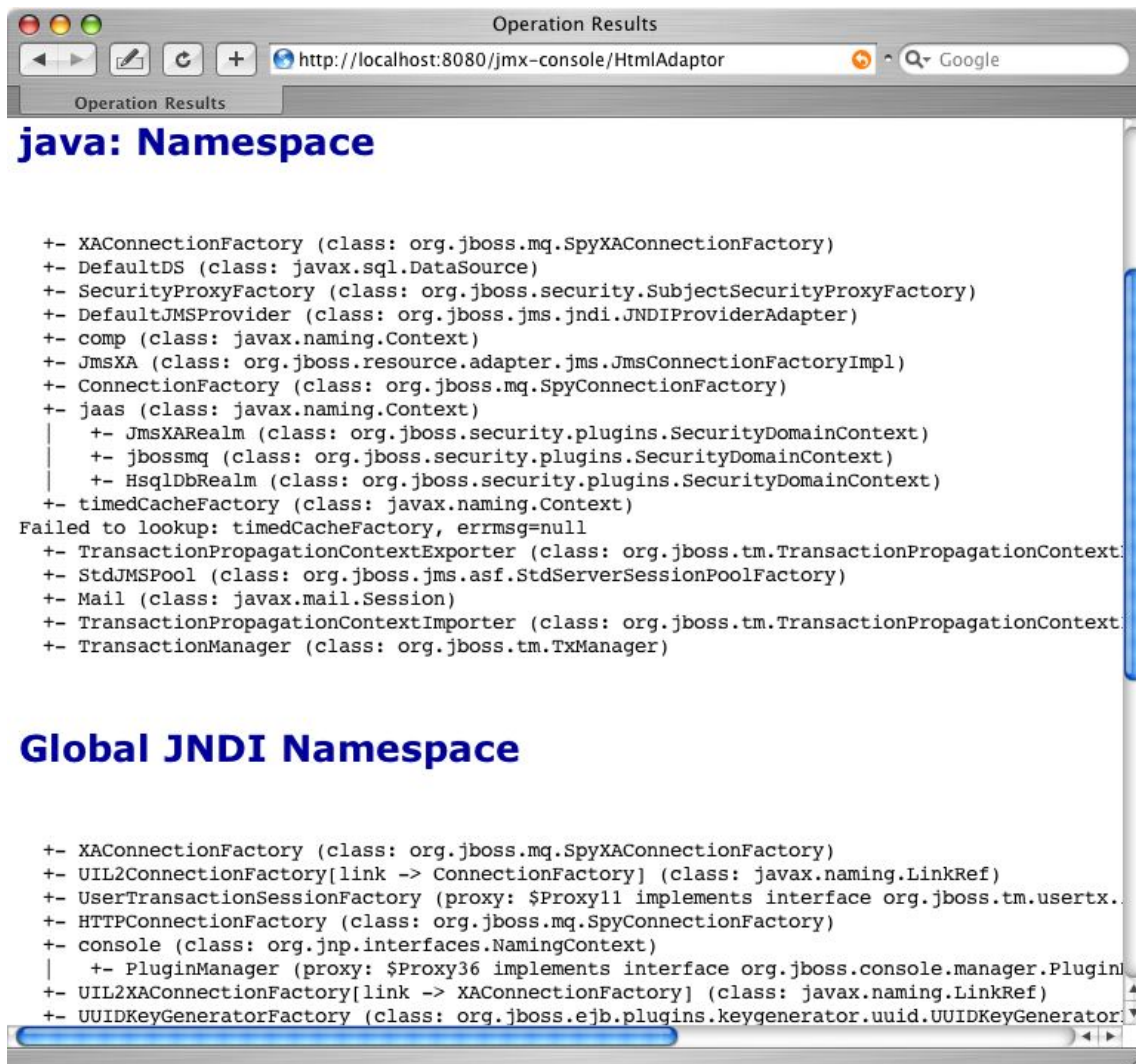


図8.6 JNDIView のリスト操作出力をJMX Console で表示

## 8.6. J2EE および JNDI - アプリケーションコンポーネント環境

JNDI は J2EE 仕様の基本となります。重要となる使用法のひとつとして、コードがデプロイされる環境からの J2EE コンポーネントコードの分離があります。アプリケーションコンポーネントの環境を使用することでアプリケーションコンポーネントのソースコードにアクセスあるいは、変更を加えずにアプリケーションコンポーネントをカスタマイズすることができるようになります。アプリケーションコンポーネント環境は ENC (エンタープライズネーミングコンテキスト) と呼ばれています。JNDI コンテキストの形式でコンテナコンポーネントに対して ENC を使用できるようにするのはアプリケーションコンポーネントコンテナの役割になります。次のようにして、J2EE のライフサイクル内で関連するパーティシパントによって ENC は活用されます。

- アプリケーションコンポーネントのビジネスロジックは、コード化し ENC からの情報にアクセスする必要があります。コンポーネントプロバイダーは、コンポーネントに対して標準の配備記述子を使い、必要とされる ENC エントリを指定します。このエントリは、実行時にコンポーネントが必要とする情報とリソースの宣言をしています。
- このコンテナが提供するツールでは、コンポーネントのデプロイヤーを使うことで、コンポーネント開発者が作成した ENC 参照をこの参照を満たすデプロイメント環境エンティティへマッピングできます。
- コンポーネントデプロイヤーは、コンテナツールを使い最終的なデプロイメントができるようコンポーネントを準備します。

- コンポーネントコンテナは、デプロイメントパッケージ情報を使ってランタイム時に完全なコンポーネントENC を構築します。

J2EE プラットフォームでJNDI 利用する上での完全仕様については、J2EE 1.4 仕様の5章を参照してください。

アプリケーションコンポーネントのインスタンスは JNDI API を使い ENC を検索します。さらに、このインスタンスは、引数なしのコンストラクタを使って `javax.naming.InitialContext` オブジェクトを作成してから `java:comp/env` 名配下にあるネーミング環境を検索します。アプリケーションコンポーネントの環境エントリは ENC に直接格納されるか、あるいはサブコンテキスト内に格納されます。例8.4「ENCアクセスのサンプルコード」でコンポーネントがENC にアクセスする際に使うコードの典型的な行を示しています。

#### 例8.4 ENCアクセスのサンプルコード

```
// Obtain the application component's ENC
Context iniCtx = new InitialContext();
Context compEnv = (Context) iniCtx.lookup("java:comp/env");
```

アプリケーションコンポーネント環境とは、アプリケーションサーバーコンテナの制御スレッドがアプリケーションコンポーネントとやりとりを行っている際にコンポーネントのみがアクセスできるローカル環境です。つまり、EJB Bean1 は EJB Bean2 のENC 要素にアクセスできず、逆方向でも同様にアクセスできなくなっています。同様に、Web アプリケーションWeb1 は、Web アプリケーションWeb2、Bean1、あるいはBean2 のENC 要素にアクセスできません。また、任意のクライアントコードはアプリケーションサーバー VM 内外を問わず実行されている場合、コンポーネントの `java:comp` JNDI コンテキストにアクセスできません。ENC の目的は、コンポーネントのデプロイ環境タイプに関係なくアプリケーションコンポーネントが依存できる分離型の読み取り専用名前空間を提供することです。各コンポーネントが独自のENC コンテンツを定義するため、ENC は他のコンポーネントから分離する必要があります。たとえば、コンポーネントA と B が同名定義していても別のオブジェクトを参照している可能性があります。また、別の例を挙げますと、EJB Bean1 は環境エントリ `java:comp/env/red` を定義し、RGB 色の赤に対して 16 進数を参照している場合があります、一方で Web アプリケーションWeb1 は、同名をデプロイメント環境言語ロケールの赤表示にバインドしている可能性もあります。

JBoss にはよく利用されるネーミングスコープレベルが3つあります。`java:comp`配下の名前、`java:`配下の名前、その他の名前です。前述したように、`java:comp` コンテキストとそのサブコンテキストは、特定のコンテキストに紐付けられたアプリケーションコンポーネントに対してのみ利用可能になっています。`java:`直下のサブコンテキストやオブジェクトバインディングは、JBoss サーバーの下層マシン内でのみ表示され、リモートクライアントからは見えません。コンテキストあるいはオブジェクトがシリアル化に対応していれば、その他のコンテキストやオブジェクトバインディングはリモートクライアントに提供されます。これらのネーミングスコープの分離の実施方法については「JBoss Naming Service アーキテクチャー」を参照してください。

`java:` コンテキストへのバインディングを制限すると便利な例は、関連データベースプールが常駐する JBoss サーバー内でのみ利用可能な `javax.sql.DataSource` の接続ファクトリでしょう。一方で、EJB ホームインターフェースは、リモートクライアントからアクセスできるはずのグローバルな表示名にバインドされるでしょう。

#### 8.6.1. ENC の使用規則

JNDI を API として使用することで、アプリケーションコンポーネントから相当量の情報を外部化します。アプリケーションコンポーネントが情報のアクセスに使用する JNDI 名は EJB コンポーネント用の標準 `ejb-jar.xml` 配備記述子および Web コンポーネント用の標準 `web.xml` 配備記述子内に宣言さ



れます。次のような様々な種類の情報が JNDI 内に格納、検索されます。

- **env-entry** 要素により宣言された通りの環境エントリ
- **ejb-ref** と **ejb-local-ref** 要素により宣言された通りのEJB 参照
- **resource-ref** 要素が宣言した通りのリソースマネージャー接続ファクトリ参照
- **resource-env-ref** 要素に定義された通りのリソース環境参照

配備記述子要素のタイプにはそれぞれ、配下に情報がバインドされる JNDI コンテキストの名前に関する JNDI 使用規則があります。また、標準の配備記述子要素に加え、アプリケーションコンポーネントによって使用される JNDI 名をデプロイメント環境の JNDI 名にマッピングする JBoss サーバー固有の配備記述子要素があります。

#### 8.6.1.1. 環境エントリ

環境エントリは、コンポーネント ENC に格納される情報の中で最もシンプルな形式で、Unix や Windows にあるようなオペレーティングシステムの環境変数に似ています。環境エントリは name-to-value バインディングで、コンポーネントが値を外部化し、名前を使って値を参照することができます。

環境エントリは、標準の配備記述子で **env-entry** を使い宣言されます。**env-entry**要素には以下の子要素が含まれます：

- エントリの詳細を提供する任意の **description** 要素
- **java:comp/env**に相対となるエントリ名を渡す **env-entry-name** 要素
- エントリ値の java タイプを与える **env-entry-type** 要素。このエントリ値のタイプは以下のいずれかでなければなりません。
  - **java.lang.Byte**
  - **java.lang.Boolean**
  - **java.lang.Character**
  - **java.lang.Double**
  - **java.lang.Float**
  - **java.lang.Integer**
  - **java.lang.Long**
  - **java.lang.Short**
  - **java.lang.String**
- 文字列としてエントリ値を与える **env-entry-value** 要素

**ejb-jar.xml** 配備記述子からの **env-entry** フラグメントに関する例が例8.5「**ejb-jar.xml env-entry のフラグメント例**」に提供されています。**env-entry**は完全名かつ値の指定であるため、JBoss 固有の配備記述子要素はありません。例8.6「**ENC env-entry アクセスコード**」では、配備記述子で宣言されて

いる **maxExemptions** 値、**taxRate** 値、**env-entry** 値へアクセスするための、サンプルコードを提示しています。

#### 例8.5 ejb-jar.xml env-entry のフラグメント例

```
<!-- ... -->
<session>
  <ejb-name>ASessionBean</ejb-name>
  <!-- ... -->
  <env-entry>
    <description>The maximum number of tax exemptions allowed
  </description>
    <env-entry-name>maxExemptions</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>15</env-entry-value>
  </env-entry>
  <env-entry>
    <description>The tax rate </description>
    <env-entry-name>taxRate</env-entry-name>
    <env-entry-type>java.lang.Float</env-entry-type>
    <env-entry-value>0.23</env-entry-value>
  </env-entry>
</session>
<!-- ... -->
```

#### 例8.6 ENC env-entry アクセスコード

```
InitialContext iniCtx = new InitialContext();
Context envCtx = (Context) iniCtx.lookup("java:comp/env");
Integer maxExemptions = (Integer) envCtx.lookup("maxExemptions");
Float taxRate = (Float) envCtx.lookup("taxRate");
```

#### 8.6.1.2. EJB参照

EJBやWeb コンポーネントは通常、他のEJBとやりとりを行います。配下にEJB ホームインターフェースがバインドされるJNDI名はデプロイメント時に決定されるため、デプロイヤーによりリンクされるEJBへの参照をコンポーネント開発者が宣言する方法が必要となります。EJB参照は、この要件を満たしています。

EJB 参照は、アプリケーションコンポーネントのネーミング環境内のリンクで、デプロイされたEJB ホームインターフェースを指定しています。アプリケーションコンポーネントによって使用される名前は、そのコンポーネントをデプロイメント環境にある実際の EJB ホーム名から隔離する論理リンクになります。J2EE 規格では、エンタープライズ bean への参照はすべてアプリケーションコンポーネント環境の **java:comp/env/ejb** コンテキスト内にまとめるよう推奨しています。

EJB 参照は、**ejb-ref** 要素を使い配備記述子で宣言します。各 **ejb-ref** 要素は、参照アプリケーションコンポーネントが参照されるエンタープライズ bean に対して持つインターフェース要件を記述します。**ejb-ref** 要素には以下の子要素が含まれます。

- 参照の目的を示すオプションの**description** 要素。

- **ejb-ref-name** 要素で、**java:comp/env** コンテキストに対して相対となる参照名を指定します。推奨の**java:comp/env/ejb** コンテキスト配下に参照を置くには、**ejb-ref-name** 値に対して、**ejb/link-name**形式を使います。
- EJB の種類を指定する**ejb-ref-type** 要素。**Entity** あるいは **Session**でなければなりません。
- EJB ホームインターフェースの完全修飾クラス名を提供する**home** 要素。
- EJB リモートインターフェースの完全修飾クラス名を提供する**remote** 要素。
- オプションの **ejb-link** 要素で同じ EJB JAR または同じ J2EE アプリケーションユニットにある別のエンタープライズ bean に参照をリンクします。**ejb-link** 値は参照される bean の **ejb-name** になります。同じ **ejb-name**を持つエンタープライズ bean が複数ある場合、値はパス名前を使用して、参照コンポーネントを含む **ejb-jar** ファイルの場所を指定します。パス名は参照 **ejb-jar** ファイルに相対的となります。Application Assembler は参照される bean の **ejb-name** を # で区切ってパス名に追加します。これにより同じ名前を持つ複数の bean が一意に識別可能となります。

EJB 参照は、宣言に**ejb-ref** 要素を含むアプリケーションコンポーネントにスコープ設定されます。つまり、EJB 参照は実行時には他のアプリケーションコンポーネントからアクセスできず、他のアプリケーションコンポーネントは名前の衝突を起こさず**ejb-ref**要素を同じ**ejb-ref-name** で定義できます。例8.7「[ejb-jar.xml 内 ejb-ref 記述子の部分例](#)」では、**ejb-ref**の使用方法を説明する**ejb-jar.xml** フラグメントを提示しています。例8.7「[ejb-jar.xml 内 ejb-ref 記述子の部分例](#)」にて宣言されている**ShoppingCartHome** 参照にアクセスするためのコード例は、例8.8「[ENC ejb-ref アクセスコード](#)」に提供されています。

#### 例8.7 ejb-jar.xml 内 ejb-ref 記述子の部分例

```
<!-- ... -->
<session>
  <ejb-name>ShoppingCartBean</ejb-name>
  <!-- ...-->
</session>

<session>
  <ejb-name>ProductBeanUser</ejb-name>
  <!-- ...-->
  <ejb-ref>
    <description>This is a reference to the store products entity
  </description>
    <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>org.jboss.store.ejb.ProductHome</home>
    <remote> org.jboss.store.ejb.Product</remote>
  </ejb-ref>
</session>

<session>
  <ejb-ref>
    <ejb-name>ShoppingCartUser</ejb-name>
    <!-- ...-->
    <ejb-ref-name>ejb/ShoppingCartHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>org.jboss.store.ejb.ShoppingCartHome</home>
```

```

        <remote> org.jboss.store.ejb.ShoppingCart</remote>
        <ejb-link>ShoppingCartBean</ejb-link>
    </ejb-ref>
</session>

<entity>
    <description>The Product entity bean </description>
    <ejb-name>ProductBean</ejb-name>
    <!--...-->
</entity>

<!--...-->

```

### 例8.8 ENC ejb-ref アクセスコード

```

InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ShoppingCartHome home = (ShoppingCartHome)
    ejbCtx.lookup("ShoppingCartHome");

```

### 8.6.1.3. jboss.xml および jboss-web.xml でのEJB 参照

JBoss 固有の**jboss.xml** EJB 配備記述子はEJB 参照に2通りに作用します。まず、**session** の **jndi-name** の子要素と**entity** 要素により、ユーザがEJB ホームインターフェースに対してデプロイメントJNDI 名を指定することができます。EJB に対し **jndi-name** を**jboss.xml** で指定しない場合、ホームインターフェースは、**ejb-jar.xml** **ejb-name** 値の配下にバインドされます。例えば、[例 8.7 「ejb-jar.xml 内 ejb-ref 記述子の部分例」](#)にある**ShoppingCartBean** の**ejb-name**を持つセッションEJB では、**jboss.xml** **jndi-name** の指定がないため、ホームインターフェースはJNDI 名の配下にバインドされます。

**ejb-ref**関連の**jboss.xml** 記述子に関する2つ目の用途は、コンポーネントのENC **ejb-ref** の参照先を設定します。**ejb-link** 要素を使って別の企業アプリケーションにあるEJBを参照することはできません。ご利用中の**ejb-ref** が外部のEJBにアクセスする必要がある場合、**jboss.xml** **ejb-ref/jndi-name** 要素を使って、配備されたEJB ホームのJNDI 名を指定することができます。

**jboss-web.xml** 記述子は、Web アプリケーションのENC **ejb-ref** の参照先を設定するためだけに利用されます。JBoss **ejb-ref**のコンテンツモデルは以下の通りです。

- **ejb-jar.xml** あるいは**web.xml**標準記述子内の**ejb-ref-name** 要素に相当する **ejb-ref-name** 要素。
- デプロイメント環境でEJB ホームインターフェースのJNDI 名を指定する **jndi-name** 要素。

[例8.9 「jboss.xml ejb-ref の部分例」](#) は、以下の使用点を説明する **jboss.xml** 記述子の部分例となっています。

- **ProductBeanUser** **ejb-ref** のリンク先は、**jboss/store/ProductHome**のデプロイメント名に設定されます。
- **ProductBean** のデプロイメントJNDI 名は、**jboss/store/ProductHome**と設定されます。

## 例8.9 jboss.xml ejb-ref の部分例

```

<!-- ... -->
<session>
  <ejb-name>ProductBeanUser</ejb-name>
  <ejb-ref>
    <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
    <jndi-name>jboss/store/ProductHome</jndi-name>
  </ejb-ref>
</session>

<entity>
  <ejb-name>ProductBean</ejb-name>
  <jndi-name>jboss/store/ProductHome</jndi-name>
  <!-- ... -->
</entity>
<!-- ... -->

```

## 8.6.1.4. EJB のローカル参照

EJB2.0 は、値セマンティクスによるRMI呼出しを使わないローカルインターフェースを追加していました。これらのインターフェースは、参照セマンティクスによる呼出しを使うため、RMI のシリアル化のオーバーヘッドが起こりません。EJB ローカル参照は、デプロイされた EJB ローカルホームインターフェースを指すアプリケーションコンポーネントのネーミング環境内のリンクになります。アプリケーションコンポーネントが使う名前は、論理リンクとなっており、デプロイメント環境にあるEJB ローカルホームの実際名からコンポーネントを分離します。J2EE 仕様では、アプリケーションコンポーネント環境の `java:comp/env/ejb` コンテキスト内に、エンタープライズ bean への参照をすべて整理するよう推奨しています。

EJB ローカル参照は配備記述子内の **ejb-local-ref** 要素を使って宣言されます。各 **ejb-local-ref** 要素は、参照されるエンタープライズ bean に対する、参照アプリケーションのインターフェース要件を記述しています。**ejb-local-ref** 要素には以下の子要素が含まれます。

- 参照の目的を示すオプションの **description** 要素。
- **ejb-ref-name** 要素で、`java:comp/env` コンテキストに対して相対となる参照名を指定します。推奨の `java:comp/env/ejb` コンテキスト配下に参照を置くには、**ejb-ref-name** 値に対して、**ejb/link-name**形式を使います。
- EJB の種類を指定する **ejb-ref-type** 要素。**Entity** あるいは **Session** でなければなりません。
- EJB ローカルホームインターフェースの完全修飾クラス名を提供する **local-home**。
- EJB ローカルインターフェースの完全修飾クラス名を提供する **local** 要素。
- **ejb-jar** ファイルまたは、同じJ2EE アプリケーションユニット内にある別のエンタープライズ bean への参照にリンクする **ejb-link** 要素。**ejb-link** 値は、参照されたbean の **ejb-name** となっています。同じ **ejb-name** を持つエンタープライズ bean が複数存在する場合、この値はパス名を使い、参照コンポーネントを含む **ejb-jar** ファイルの場所を指定します。Application Assembler は、#で区切られたパス名に参照 bean の **ejb-name** を付けます。これにより、同名を持つ bean が複数あっても一意に識別できるようになります。JBoss では **ejb-link** 要素を指定し、ローカル参照と対応するEJBを一致させる必要があります。

EJB ローカル参照は、宣言に **ejb-local-ref** 要素を含むアプリケーションコンポーネントにスコープ化されます。つまり、EJB ローカル参照は、ランタイム時に他のアプリケーションコンポーネントからアクセスできず、別のアプリケーションコンポーネントは名前の衝突なしに同じ **ejb-ref-name** を持つ **ejb-local-ref** 要素を定義する場合があるのです。例8.10「[ejb-jar.xml ejb-local-ref 記述子の部分例](#)」では、**ejb-local-ref** の用途について例示する **ejb-jar.xml** のフラグメントが提供されています。例8.11「[ENC ejb-local-ref アクセスコード](#)」のコード例にて、例8.10「[ejb-jar.xml ejb-local-ref 記述子の部分例](#)」で宣言される **ProbeLocalHome** 参照へのアクセスについて例示しています。

#### 例8.10 ejb-jar.xml ejb-local-ref 記述子の部分例

```
<!-- ... -->
<session>
  <ejb-name>Probe</ejb-name>
  <home>org.jboss.test.perf.interfaces.ProbeHome</home>
  <remote>org.jboss.test.perf.interfaces.Probe</remote>
  <local-
home>org.jboss.test.perf.interfaces.ProbeLocalHome</local-home>
  <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
  <ejb-class>org.jboss.test.perf.ejb.ProbeBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Bean</transaction-type>
</session>
<session>
  <ejb-name>PerfTestSession</ejb-name>
  <home>org.jboss.test.perf.interfaces.PerfTestSessionHome</home>
  <remote>org.jboss.test.perf.interfaces.PerfTestSession</remote>
  <ejb-class>org.jboss.test.perf.ejb.PerfTestSessionBean</ejb-
class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
  <ejb-ref>
    <ejb-ref-name>ejb/ProbeHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>org.jboss.test.perf.interfaces.SessionHome</home>
    <remote>org.jboss.test.perf.interfaces.Session</remote>
    <ejb-link>Probe</ejb-link>
  </ejb-ref>
  <ejb-local-ref>
    <ejb-ref-name>ejb/ProbeLocalHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-
home>org.jboss.test.perf.interfaces.ProbeLocalHome</local-home>
    <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
    <ejb-link>Probe</ejb-link>
  </ejb-local-ref>
</session>
<!-- ... -->
```

#### 例8.11 ENC ejb-local-ref アクセスコード

```
InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ProbeLocalHome home = (ProbeLocalHome) ejbCtx.lookup("ProbeLocalHome");
```

### 8.6.1.5. リソースマネージャー接続ファクトリの参照

リソースマネージャー接続ファクトリの参照により、アプリケーションコンポーネントコードはリソースマネージャー接続ファクトリの参照と呼ばれる論理名を使うことで、リソースファクトリを参照することができます。リソースマネージャー接続ファクトリの参照は、標準の配備記述子にて **resource-ref** 要素により定義されています。**Deployer** は、**jboss.xml** と **jboss-web.xml** 記述子を使って、リソースマネージャー接続ファクトリの参照を実際の運用環境にあるリソースマネージャー接続ファクトリにバインドします。

各 **resource-ref** 要素は、1つのリソースマネージャー接続ファクトリの参照を記述します。**resource-ref** 要素は、以下の子要素で構成されます：

- 参照の目的を示すオプションの **description** 要素。
- **java:comp/env** コンテキストを起点とした参照名を指定する **res-ref-name** 要素。どのサブコンテキストに **res-ref-name** を配置するかを決定するリソースタイプに基づいた命名規則について、次の段落にて説明します。
- リソースマネージャー接続ファクトリの完全修飾名を指定する **res-type** 要素。
- アプリケーションコンポーネントコードがリソースのサインオンをプログラムで実行するか、あるいはコンテナがデプロイヤー提供の主体マッピング情報に基づいたリソースへサインオンするかを示す **res-auth** 要素。**Application** あるいは **Container** のいずれかでなければなりません。
- 任意の **res-sharing-scope** 要素。現在、JBoss では対応していません。

J2EE 仕様では、リソース管理タイプごとに異なるサブコンテキストを使うことで、リソース管理接続ファクトリの参照をすべて、アプリケーションコンポーネント環境のサブコンテキスト内にまとめるよう推奨されています。サブコンテキスト名に推奨されるリソース管理タイプは次の通りです。

- **JDBC DataSource** 参照は、**java:comp/env/jdbc** のサブコンテキストで宣言する必要があります。
- **JMS** 接続ファクトリは **java:comp/env/jms** サブコンテキストで宣言する必要があります。
- **JavaMail** 接続ファクトリは **java:comp/env/mail** サブコンテキストにて宣言する必要があります。
- **URL** 接続ファクトリは、**java:comp/env/url** サブコンテキストにて宣言する必要があります。

例8.12「web.xml resource-ref 記述子のフラグメント」は、**resource-ref** 要素の用途を示す **web.xml** 記述子のフラグメント例です。例8.13「ENC resource-ref アクセスのサンプルコード」は、アプリケーションコンポーネントが **resource-ref** により宣言されている **DefaultMail** リソースへアクセスする際に使うコードを提供しています。

#### 例8.12 web.xml resource-ref 記述子のフラグメント

```
<web>
  <!-- ... -->
  <servlet>
    <servlet-name>AServlet</servlet-name>
    <!-- ... -->
```

```

</servlet>
<!-- ... -->
<!-- JDBC DataSources (java:comp/env/jdbc) -->
<resource-ref>
  <description>The default DS</description>
  <res-ref-name>jdbc/DefaultDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
<!-- JavaMail Connection Factories (java:comp/env/mail) -->
<resource-ref>
  <description>Default Mail</description>
  <res-ref-name>mail/DefaultMail</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
<!-- JMS Connection Factories (java:comp/env/jms) -->
<resource-ref>
  <description>Default QueueFactory</description>
  <res-ref-name>jms/QueueFactory</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
</web>

```

#### 例8.13 ENC resource-ref アクセスのサンプルコード

```

Context initCtx = new InitialContext();
javax.mail.Session s = (javax.mail.Session)
initCtx.lookup("java:comp/env/mail/DefaultMail");

```

#### 8.6.1.6. jboss.xml と jboss-web.xml でのリソースマネージャー接続ファクトリの接続

JBoss **jboss.xml** EJB 配備記述子と **jboss-web.xml** Web アプリケーションの配備記述子の目的は、**jboss-web.xml** 要素により定義された論理名からJBossでデプロイされるリソースファクトリのJNDI名へのリンクを提供することです。**jboss.xml** あるいは **jboss-web.xml** 記述子に **resource-ref** 要素を提供することでリンク可能です。JBoss **resource-ref** 要素には、以下の子要素が含まれています：

- **res-ref-name**要素。これは、**ejb-jar.xml** あるいは **web.xml** の標準記述子にある該当の **resource-ref** 要素の **res-ref-name** と一致しなければなりません。
- 任意の **res-type** 要素。リソースマネージャー接続ファクトリの完全修飾クラス名を指定します。
- **jndi-name**要素。これは、JBossにデプロイされるように、リソースファクトリのJNDI名を指定します。
- **res-url**要素。これは、**resource-ref**がタイプ **java.net.URL** の場合にURL文字列を指定します。



例8.14 「jboss-web.xml resource-ref 記述子の例」は、jboss-web.xml のサンプル記述子を提示しています。このフラグメントでは例8.12 「web.xml resource-ref 記述子のフラグメント」にあるresource-ref要素のサンプルマッピングが示されています。

#### 例8.14 jboss-web.xml resource-ref 記述子の例

```
<jboss-web>
  <!-- ... -->
  <resource-ref>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <jndi-name>java:/DefaultDS</jndi-name>
  </resource-ref>
  <resource-ref>
    <res-ref-name>mail/DefaultMail</res-ref-name>
    <res-type>javax.mail.Session</res-type>
    <jndi-name>java:/Mail</jndi-name>
  </resource-ref>
  <resource-ref>
    <res-ref-name>jms/QueueFactory</res-ref-name>
    <res-type>javax.jms.QueueConnectionFactory</res-type>
    <jndi-name>QueueConnectionFactory</jndi-name>
  </resource-ref>
  <!-- ... -->
</jboss-web>
```

#### 8.6.1.7. リソース環境の参照

リソース環境の参照は、論理名を使ってリソース (例：JMS デスティネーション) と紐付いた管理オブジェクトを参照する要素です。リソース環境の参照は、標準の配備記述子にて**resource-env-ref**要素で定義されています。**Deployer**は、jboss.xml と jboss-web.xml記述子を使うことで、リソース環境の参照を対象の運用環境にある実際の管理オブジェクトの場所にバインドします。

**resource-env-ref** 要素はそれぞれ、参照される管理オブジェクトに対し参照するアプリケーションコンポーネントが必要とする要件を記述します。**resource-env-ref** 要素には、以下の子要素が含まれます：

- 参照の目的を示すオプションの**description** 要素。
- **java:comp/env** コンテキストを起点とした参照名を指定する**resource-env-ref-name** 要素。規則は関連のリソースファクトリタイプに対応するサブコンテキスト内にその名前を配置します。たとえば、**MyQueue** という名前の JMS キュー参照は **jms/MyQueue** の **resource-env-ref-name** を持っているはずです。
- 参照されるオブジェクトの完全修飾クラス名を指定する**resource-env-ref-type** 要素。例えば、JMS キューの場合、値は**javax.jms.Queue**となります。

例8.15 「ejb-jar.xml resource-env-ref のフラグメント例」は、セッション bean による**resource-ref-env**要素の宣言例を示しています。例8.16 「ENC resource-env-ref アクセスコード」では、**resource-env-ref**により宣言された**StockInfo** キューを検索する方法を示すコードを提供しています。

#### 例8.15 ejb-jar.xml resource-env-ref のフラグメント例

```

<session>
  <ejb-name>MyBean</ejb-name>
  <!-- ... -->
  <resource-env-ref>
    <description>This is a reference to a JMS queue used in the
      processing of Stock info
    </description>
    <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
  </resource-env-ref>
  <!-- ... -->
</session>

```

#### 例8.16 ENC resource-env-ref アクセスコード

```

InitialContext iniCtx = new InitialContext();
javax.jms.Queue q = (javax.jms.Queue)
envCtx.lookup("java:comp/env/jms/StockInfo");

```

#### 8.6.1.8. リソース環境参照と jboss.xml、jboss-web.xml

JBoss **jboss.xml** EJB 配備記述子と **jboss-web.xml** Web アプリケーションの配備記述子の目的は、**resource-env-ref-name** 要素により定義された論理名からJBoss でデプロイされる管理オブジェクトのJNDI 名へのリンクを提供することです。**jboss.xml** あるいは**jboss-web.xml** 記述子に**resource-env-ref** 要素を提供することでリンク可能です。JBoss **jboss-web.xml** 要素には、以下の子要素が含まれています：

- **resource-env-ref-name** 要素。**ejb-jar.xml** あるいは **web.xml** の標準記述子からの該当の**resource-env-ref** 要素にある**resource-env-ref-name** と一致しなければなりません。
- JBoss でデプロイされるようにリソースのJNDI 名を指定する**jndi-name** 要素。

例8.17 「[jboss.xml resource-env-ref 記述子のフラグメント例](#)」では、**StockInforesource-env-ref**に関するマッピングサンプルを示す**jboss.xml** 記述子の例を提供しています。

#### 例8.17 jboss.xml resource-env-ref 記述子のフラグメント例

```

<session>
  <ejb-name>MyBean</ejb-name>
  <!-- ... -->
  <resource-env-ref>
    <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
    <jndi-name>queue/StockInfoQueue</jndi-name>
  </resource-env-ref>
  <!-- ... -->
</session>

```

## 第9章 WEB サービス

Web サービスは、現在の Web コマースの運営方法を決める主要因となっています。Web サービスは、大小のデータのチャンクを送信しアプリケーション同士が通信できるようにします。

Web アプリケーションは、コンピューターネットワーク上やワールドワイドウェブ上のアプリケーションのやりとりをサポートするソフトウェアアプリケーションです。通常、Web サービスは、オブジェクト、コンピュータープログラム、ビジネスプロセス、データベースなどへマップする XML ドキュメントを介してやりとりを行います。アプリケーションは XML サードパーティ形式のメッセージを Web サービスに送信し、Web サービスはこのメッセージを対応するプログラムへ送信します。応答は要件を基に受信され、Web サービスは受信したメッセージを XML 形式で必要とするプログラムやアプリケーションへ送信します。Web サービスは、サプライチェーン管理やビジネス統合などさまざまな目的に使用できます。

JBossWS は JBoss Enterprise Application Platform の一部として同梱される Web サービスのフレームワークです。Java Platform Enterprise Edition 5 (Java EE 5) をターゲットとし、Java に Web サービスを実装するためのプログラミングモデルやランタイムアーキテクチャーを定義する JAX-WS 仕様を実装します。JAX-RPC もサポートされますが (J2EE 1.4 の Web サービス仕様)、JBossWS は JAX-WS に主力を置きます。



### 警告

JAX-RPCは、JBoss Web Services CXF スタックには対応していません。

### 9.1. WEB サービスの必要性

Web サービス技術を適切に導入すると、企業向けシステムの通信を向上することができます。よく設計されたコントラクトに焦点を置くと、開発者はサービス機能の抽象ビューを確立することができます。コントラクトが書かれた標準的な方法を考慮すると、サードパーティシステムとの通信を実現できるため、最終的にビジネス間の統合をサポートします。プロバイダーとコンシューマーが合意するコントラクトの内容はすべて明確で標準的です。また、実装間の依存関係を軽減するため、主要な変更を行わなくても他のコンシューマーが提供されるサービスを簡単に使用することができます。

Web サービスの相互運用性はサービスの再使用と構成を向上するため、内部異種サブシステムの Web サービス技術を取り入れる企業システムには別の利点があります。異なるソフトウェア言語を使用して別の企業部門によって開発されるため、Web サービスによって機能全体を書き直す必要がなくなります。

### 9.2. WEB サーバーとは

Web サービスは、ソフトウェアシステム通信すべてに対するソリューションではありません。

現在、疎結合で粗粒度の通信、メッセージ (文書) の交換に利用するために Web サービスが利用されています。近年、多くの仕様 (WS-\*) が検討され、信頼できるメッセージングやメッセージレベルセキュリティ、クロスサービストランザクションなど、標準的な WS 関連の上級アスペクトの確立が最終的に承諾されました。Web サービス仕様には、サービスコントラクト参照を集めるレジストリ概念も含まれており、サービス実装の検索を簡単に行うことができます。

そのため、Web サービスの技術プラットフォームはリモートプロシーチャー呼び出しを行う最新の方法としてだけでなく、複雑な企業向け通信に適合します。

## 9.3. DOCUMENT/LITERAL

ドキュメントスタイルの Web サービスでは、2 者が XML スキーマで適切に定義される複雑なビジネス文書の交換に同意することになります。たとえば、一方が発注書の記載があるドキュメントを送信し、相手側がその発注書の状況を記載する文書で応答(即座または後に)します。SOAP メッセージのペイロードは XML スキーマに対して認証可能な XML ドキュメントです。このドキュメントは SOAP バインディングのスタイル属性で定義されます。

```
<binding name='EndpointInterfaceBinding' type='tns:EndpointInterface'>
  <soap:binding style='document'
transport='http://schemas.xmlsoap.org/soap/http' />
  <operation name='concat'>
    <soap:operation soapAction='' />
    <input>
      <soap:body use='literal' />
    </input>
    <output>
      <soap:body use='literal' />
    </output>
  </operation>
</binding>
```

ドキュメントスタイルの Web サービスでは、各メッセージのペイロードは XML スキーマでの複合タイプで定義されます。

```
<complexType name='concatType'>
  <sequence>
    <element name='String_1' nillable='true' type='string' />
    <element name='long_1' type='long' />
  </sequence>
</complexType>
<element name='concat' type='tns:concatType' />
```

そのため、メッセージの部分はスキーマからの要素を参照しなければなりません。

```
<message name='EndpointInterface_concat'>
  <part name='parameters' element='tns:concat' />
</message>
```

次のメッセージ定義は無効となります。

```
<message name='EndpointInterface_concat'>
  <part name='parameters' type='tns:concatType' />
</message>
```

## 9.4. DOCUMENT/LITERAL (BARE)

Bare とは Java ドメインからの実装詳細になります。抽象 コントラクト (swdl+スキーマなど) 内や SOAP メッセージレベルでは bare エンドポイントは認識されません。bare のエンドポイントまたはクライアントはドキュメントのペイロード全体を表す Java bean を使用します。

```
@WebService
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
```

```

public class DocBareServiceImpl
{
    @WebMethod
    public SubmitBareResponse submitPO(SubmitBareRequest poRequest)
    {
        ...
    }
}

```

ペイロードを表している Java bean は JAXB アノテーションを含んでおり、このアノテーションがメッセージ上でペイロードがどのように表されるかを定義しているのが要点になります。

```

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SubmitBareRequest",
namespace="http://soapbinding.samples.jaxws.ws.test.jboss.org/", propOrder
= { "product" })
@XmlRootElement(namespace="http://soapbinding.samples.jaxws.ws.test.jboss.
org/", name = "SubmitPO")
public class SubmitBareRequest
{

    @XmlElement(namespace="http://soapbinding.samples.jaxws.ws.test.jboss.org/
", required = true)
    private String product;

    ...
}

```

## 9.5. DOCUMENT/LITERAL (WRAPPED)

Wrapped とは Java ドメインからの実装詳細になります。抽象 コントラクト (swdl+スキーマなど) 内や SOAP メッセージレベルでは wrapped エンドポイントは認識されません。wrapped のエンドポイントやクライアントは個別のドキュメントペイロードプロパティを使用します。wrapped はデフォルトであるため、明示的に宣言する必要はありません。

```

@WebService
public class DocWrappedServiceImpl
{
    @WebMethod
    @RequestWrapper (className="org.somepackage.SubmitPO")
    @ResponseWrapper (className="org.somepackage.SubmitPOResponse")
    public String submitPO(String product, int quantity)
    {
        ...
    }
}

```



### 注記

JBossWS では、要求および応答のラッパーアノテーションは必要ありません。目的にあったデフォルトを使用し要求に従い生成されます。

## 9.6. RPC/LITERAL

RPC ではエンドポイント操作に名前を付けるラッパー要素があります。親 RPC の子要素は個別のパラメーターです。SOAP ボディーは次のような簡単なルールによって構築されます。

- ポートタイプ操作名がエンドポイントメソッド名を定義する。
- メッセージ部分はエンドポイントメソッドパラメーターである。

TPC は SOAP バインディングのスタイル属性によって定義されます。

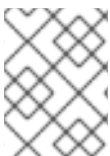
```
<binding name='EndpointInterfaceBinding' type='tns:EndpointInterface'>
  <soap:binding style='rpc'
transport='http://schemas.xmlsoap.org/soap/http' />
  <operation name='echo'>
    <soap:operation soapAction='' />
    <input>
      <soap:body namespace='http://org.jboss.ws/samples/jsr181pojo'
use='literal' />
    </input>
    <output>
      <soap:body namespace='http://org.jboss.ws/samples/jsr181pojo'
use='literal' />
    </output>
  </operation>
</binding>
```

RPC スタイルの Web サービスでは、portType が操作の名前を付けます (例: エンドポイント上の java メソッド)。

```
<portType name='EndpointInterface'>
  <operation name='echo' parameterOrder='String_1'>
    <input message='tns:EndpointInterface_echo' />
    <output message='tns:EndpointInterface_echoResponse' />
  </operation>
</portType>
```

個別のメッセージ部分によって操作パラメーターが定義されます。

```
<message name='EndpointInterface_echo'>
  <part name='String_1' type='xsd:string' />
</message>
<message name='EndpointInterface_echoResponse'>
  <part name='result' type='xsd:string' />
</message>
```



### 注記

SOAP メッセージペイロード全体を認証できる複雑なタイプは、XML スキーマにはありません。

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
  @WebMethod
```

```

    @WebResult(name="result")
    public String echo(@WebParam(name="String_1") String input)
    {
        ...
    }
}

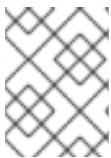
```

RPC パラメーターの要素名は JAX-WS Annotations#javax.jws.WebParam、RPC 戻り値の要素名は JAX-WS Annotations#javax.jws.WebResult を使用して定義することができます。

## 9.7. RPC/ENCODED

SOAP エンコーディングスタイルは [SOAP-1.1](#) 仕様の 5 章により定義されます。相互運用性に関する固有の問題が存在し、これは修正ができません。[Basic Profile-1.0](#)は [4.1.7 SOAP encodingStyle Attribute](#) でこのエンコーディングスタイルを禁止しています。JBossWS は、RPC/Encoded に基本対応しています。これは、Literal エンコーディングに未対応の SOAP スタックとの簡単な相互運用シナリオに対し現状提供されています。具体的には、JBossWS は以下に対応しません。

- 要素の参照
- bean プロパティとしての soap 配列



### 注記

本項は、JBoss Web Services CXF スタックと連携する際に利用すべきではありません。

## 9.8. WEB サービスのエンドポイント

JAX-WS は Web サービスエンドポイントの Web サービスエンドポイントの開発モデルを大幅に簡略化します。エンドポイント実装 bean は JAX-WS アノテーションが付けられサーバーにデプロイされます。サーバーは自動的にクライアントが消費する抽象コントラクトを (wsdl+スキーマなど) を生成して発行します。すべてのマーシャルおよびアンマーシャルは JAXB に委任されます。

## 9.9. POJO (PLAIN OLD JAVA OBJECT)

簡単な POJO エンドポイント実装を見てみましょう。エンドポイントに関連するメタデータはすべて JSR-181 アノテーションより提供されます。

```

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
    @WebMethod
    public String echo(String input)
    {
        ...
    }
}

```

## 9.10. WEB アプリケーションとしてのエンドポイント

JAX-WS java サービスエンドポイント (JSE) は Web アプリケーションとしてデプロイされます。

```
<web-app ...>
  <servlet>
    <servlet-name>TestService</servlet-name>
    <servlet-
class>org.jboss.test.ws.jaxws.samples.jsr181pojo.JSEBean01</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>TestService</servlet-name>
      <url-pattern>/*</url-pattern>
    </servlet-mapping>
  </web-app>
```

## 9.11. エンドポイントのパッケージ化

JSR-181 java サービスエンドポイント (JSE) は Web アプリケーションとして **\*.war** ファイルにパッケージ化されます。

```
<war warfile="${build.dir}/libs/jbossws-samples-jsr181pojo.war"
webxml="${build.resources.dir}/samples/jsr181pojo/WEB-INF/web.xml">
  <classes dir="${build.dir}/classes">
    <include
name="org.jboss/test/ws/samples/jsr181pojo/JSEBean01.class"/>
  </classes>
</war>
```



### 注記

エンドポイント実装 bean と **web.xml** ファイルのみが必要となります。

## 9.12. 生成された WSDL にアクセス

正しくデプロイされたサービスエンドポイントはサービスエンドポイントマネージャーに表示されます。サービスエンドポイントマネージャーには生成された WSDL へのリンクもあります。

```
http://yourhost:8080/jbossws/services
```

また、JBoss ツールを使用してオフラインで抽象コントラクトを生成することも可能です。詳細については [Top Down \(wsconsume の利用\)](#) を参照してください。

## 9.13. EJB3 ステートレスセッション BEAN (SLSB)

JAX-WS プログラミングモデルは [Plain old Java Object \(POJO\)](#) のエンドポイント上と同じアノテーションセットを EJB3 ステートレスセッション bean でもサポートします。EJB-2.1 エンドポイントには JAX-RPC プログラミングモデルを使い対応しています。

```
@Stateless
@Remote(EJB3RemoteInterface.class)
@RemoteBinding(jndiBinding = "/ejb3/EJB3EndpointInterface")

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJB3Bean01 implements EJB3RemoteInterface
```



```
{
    @WebMethod
    public String echo(String input)
    {
        ...
    }
}
```

上記には、リモートインターフェース上とエンドポイント操作上の両方で1つのメソッドを公開する EJB-3.0 ステートレスセッション bean があります。

### エンドポイントのパッケージ化

JSR-181 EJB サービスのエンドポイントは通常の ejb デプロイメントとしてパッケージ化されます。

```
<jar jarfile="${build.dir}/libs/jbossws-samples-jsr181ejb.jar">
  <fileset dir="${build.dir}/classes">
    <include
name="org/jboss/test/ws/samples/jsr181ejb/EJB3Bean01.class"/>
    <include
name="org/jboss/test/ws/samples/jsr181ejb/EJB3RemoteInterface.class"/>
  </fileset>
</jar>
```

### 生成された WSDL にアクセス

正しくデプロイされたサービスエンドポイントはサービスエンドポイントマネージャーに表示されます。サービスエンドポイントマネージャーには生成された WSDL へのリンクもあります。

```
http://yourhost:8080/jbossws/services
```

また、JBossWS ツールを使用してオフラインで抽象コントラクトを生成することも可能です。詳細については [Top Down \(wsconsumeを利用\)](#) を参照してください。

## 9.14. エンドポイントプロバイダー

JAX-WS のサービスは一般的にネイティブの Java サービスエンドポイントインターフェース (SEI) を実装し、おそらく WSDL ポートタイプから直接あるいはアノテーションを使うことによってマッピングされます。

Java SEI は、XML ベースのメッセージで使用するため Java オブジェクトとその XML 表記間で行われる変換の詳細を隠す Java 中心の高レベルの抽出を提供します。ただし、XML メッセージレベルでサービスの動作が可能となる方が望ましい場合もあります。この場合プロバイダーインターフェースは SEI に代替を提供し、XML メッセージレベルでの動作を好むサービスによる実装が可能になります。

Provider ベースのサービスインスタンスの呼び出しメソッドはサービスに対して受け取られる各メッセージごとに呼びだされます。

```
@WebServiceProvider
@ServiceMode(value = Service.Mode.PAYLOAD)
public class ProviderBeanPayload implements Provider<Source>
{
    public Source invoke(Source req)
    {
```

```

        // Access the entire request PAYLOAD and return the response PAYLOAD
    }
}

```

注記、 `Service.Mode.PAYLOAD` はデフォルトとなり、明示的に宣言される必要はありません。また、 `Service.Mode.MESSAGE` を使って SOAP 全体にアクセスすることもできます (MESSAGE で Provider は SOAP ヘッダーも見ることができます)。

## 9.15. WEBSERVICECONTEXT

**WebServiceContext** はエンドポイントが初期化される時点で設定できるインジェクションが可能なソースとして扱われます。 **WebServiceContext** オブジェクトは次に、同じエンドポイントオブジェクト宛ての複数の要求処理に同時使用されているスレッド数に関係なく、スレッドローカル情報を使用して正しい情報を返します。

```

@WebService
public class EndpointJSE
{
    @Resource
    WebServiceContext wsCtx;

    @WebMethod
    public String testGetMessageContext()
    {
        SOAPMessageContext jaxwsContext =
        (SOAPMessageContext)wsCtx.getMessageContext();
        return jaxwsContext != null ? "pass" : "fail";
    }
    ...
    @WebMethod
    public String testGetUserPrincipal()
    {
        Principal principal = wsCtx.getUserPrincipal();
        return principal.getName();
    }

    @WebMethod
    public boolean testIsUserInRole(String role)
    {
        return wsCtx.isUserInRole(role);
    }
}

```

## 9.16. WEB サービスのクライアント

### 9.16.1. サービス

**Service** は WSDL サービスを表す抽象になります。WSDL サービスは関連ポートの集合で、それぞれが特定のプロトコルにバインドするポートタイプから構成されます。また、特定のエンドポイントアドレスで使用可能になります。

ほとんどのクライアントの場合、WSDL から生成されるスタブセットを使って開始します。これらのうちのひとつはサービスになり、そのサービスと動作させるためにそのクラスのオブジェクトを作成します (以下の「静的な場合」を参照)。

### 9.16.1.1. サービスの使用

#### 静的な場合

ほとんどのクライアントは WSDL ファイルで開始し、**wsconsume** のような jbossws ツールを使っていくつかのスタブを生成します。これは通常、膨大なファイル数となり、そのうちのひとつがツリーのトップになります。これがサービス実装クラスです。

生成される実装クラスは、1 つは引数なしでもうひとつは 2 つの引数を持ちそれぞれ wsdl の場所 (java.net.URL) とサービス名 (javax.xml.namespace.QName) を表す 2 つのパブリックコンストラクターを持つことになると思います。

通常、引数なしのコンストラクタを使用します。この場合、WSDL の場所とサービス名は WSDL にあります。これらは生成されるクラスを修飾する WebServiceClient アノテーションから黙示的に設定されます。

次のコードスニペットは生成されたクラスから生成されるコンストラクターを示しています。

```
// Generated Service Class

@WebServiceClient(name="StockQuoteService",
    targetNamespace="http://example.com/stocks",
    wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{

    public StockQuoteService()
    {
        super(new URL("http://example.com/stocks.wsdl"), new
QName("http://example.com/stocks", "StockQuoteService"));
    }

    public StockQuoteService(String wsdlLocation, QName serviceName)
    {
        super(wsdlLocation, serviceName);
    }

    ...
}
```

[ダイナミックプロキシ](#) の章には、サービスからのポート取得方法やポート上での操作の呼び出し方法について説明されています。XML ペイロードを直接作業する必要がある場合や SOAP メッセージ全体の XML 表記を作業する必要がある場合は、[ディスパッチ](#) をご覧ください。

#### 動的な場合

動的な場合、何も生成されないと Web サービスクライアントは **Service.create** を使って Service インスタンスを作成します。次のコードでこのプロセスを示します。

```
URL wsdlLocation = new URL("http://example.org/my.wsdl");
QName serviceName = new QName("http://example.org/sample", "MyService");
Service service = Service.create(wsdlLocation, serviceName);
```

このような JBossWS の使用方法は推奨されていません。

### 9.16.1.2. ハンドラーリゾルバー

JAX-WS はハンドラーとして知られるメッセージ処理モジュール用の柔軟なプラグインフレームワークを提供しており、JAX-WS ランタイムシステムの機能拡張に使用することができます。 [Handler Framework](#) ではハンドラーフレームワークについて詳細に説明されています。 **Service** インスタンスは、サービス毎やポート毎、プロトコルバインディング毎にハンドラーセットを設定することができる **getHandlerResolver** メソッドと **setHandlerResolver** メソッドのペアで **HandlerResolver** へのアクセスを提供します。

**Service** インスタンスがプロキシまたは **Dispatch** インスタンスの作成に使用されると、現在そのサービスで登録されているハンドラーリゾルバーが要求されるハンドラーチェーンの作成に使用されます。続いて起こる **Service** インスタンス用に設定されるハンドラーリゾルバーへの変更は前に作成されたプロキシあるいは **Dispatch** のインスタンスにあるハンドラーには影響しません。

### 9.16.1.3. Executor

**Service** インスタンスは **java.util.concurrent.Executor** で設定することができます。次にエグゼキューターを使ってアプリケーションが要求するすべての非同期のコールバックが呼び出されます。 **Service** の **setExecutor** と **getExecutor** のメソッドを使ってサービス用に設定されるエグゼキューターを編集して読み出すことができます。

### 9.16.2. 動的なプロキシ

**Service** の **getPort** メソッドの 1 つを使ってクライアントプロキシのインスタンスを作成することができます。

```
/**
 * The getPort method returns a proxy. A service client
 * uses this proxy to invoke operations on the target
 * service endpoint. The <code>serviceEndpointInterface</code>
 * specifies the service endpoint interface that is supported by
 * the created dynamic proxy instance.
 */
public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)
{
    ...
}

/**
 * The getPort method returns a proxy. The parameter
 * <code>serviceEndpointInterface</code> specifies the service
 * endpoint interface that is supported by the returned proxy.
 * In the implementation of this method, the JAX-WS
 * runtime system takes the responsibility of selecting a protocol
 * binding (and a port) and configuring the proxy accordingly.
 * The returned proxy should not be reconfigured by the client.
 */
public <T> T getPort(Class<T> serviceEndpointInterface)
{
    ...
}
```

サービスエンドポイントインターフェース (SEI) は通常ツールを使って生成されます。詳細については [Top Down \(Using wsconsume\)](#) を参照してください。

生成される静的な **Service** は通常、型指定されたメソッドも提供しポートを取得します。これらのメソッドは SEI を実装する動的なプロキシも返します。

```
@WebServiceClient(name = "TestEndpointService", targetNamespace =
"http://org.jboss.ws/wsref",
    wsdlLocation = "http://localhost.localdomain:8080/jaxws-samples-
webserviceref?wsdl")
public class TestEndpointService extends Service
{
    ...

    public TestEndpointService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    @WebEndpoint(name = "TestEndpointPort")
    public TestEndpoint getTestEndpointPort()
    {
        return (TestEndpoint)super.getPort(TESTENDPOINTPORT,
TestEndpoint.class);
    }
}
```

### 9.16.3. WebServiceRef

**WebServiceRef** アノテーションは Web サービスに対する参照の宣言に使用されます。JSR-250 にある **javax.annotation.Resource** で範例されるリソースパターンに従います。

**WebServiceRef** アノテーションには 2 通りの使い方があります。

1. タイプが生成されたサービスクラスである参照を定義します。この場合、タイプと値要素はいずれも生成されたサービスクラス型を参照します。さらに、その参照タイプがアノテーションが適用されるフィールド／メソッド宣言で推測できる場合、タイプおよび値要素はデフォルトの値 (**Object.class**) とすることもできます。タイプが推測できない場合は、最低でもタイプ要素がデフォルト以外の値でなければなりません。
2. タイプが SEI の参照を定義します。この場合、その参照タイプがアノテーションが付けられたフィールド／メソッド宣言から推測できればタイプ要素はデフォルトでも構いませんが、値要素は常に存在しなければならず、生成されたサービスクラスタイプ (**javax.xml.ws.Service** のサブタイプ) を参照しなければなりません。wsdlLocation 要素が存在する場合、参照され生成されたサービスクラスの **WebService** アノテーションに指定された theWSDL の場所情報よりも優先されます。

```
public class EJB3Client implements EJB3Remote
{
    @WebServiceRef
    public TestEndpointService service4;

    @WebServiceRef
    public TestEndpoint port3;
}
```

### WebServiceRef のカスタマイズ

JBoss Enterprise Application Platform 5.0 は **WebServiceRef** アノテーションへのオーバーライドや拡張を多数提供します。以下のようなものが含まれます。

- コンテナ管理ポートを解決するために使用されるべきポートの定義
- Stub オブジェクトに対するデフォルトの Stub プロパティ設定の定義
- 最終的に使用される WSDL ドキュメントの URL 定義

例:

```
<service-ref>
  <service-ref-name>OrganizationService</service-ref-name>
  <wsdl-override>file:/wsdlRepository/organization-service.wsdl</wsdl-
override>
</service-ref>
..
<service-ref>
  <service-ref-name>OrganizationService</service-ref-name>
  <config-name>Secure Client Config</config-name>
  <config-file>META-INF/jbossws-client-config.xml</config-file>
  <handler-chain>META-INF/jbossws-client-handlers.xml</handler-chain>
</service-ref>

<service-ref>
  <service-ref-name>SecureService</service-ref-name>
  <service-class-
name>org.jboss.tests.ws.jaxws.webservicesref.SecureEndpointService</service
-class-name>
  <service-qname>
{http://org.jboss.ws/wsref}SecureEndpointService</service-qname>
  <port-info>
    <service-endpoint-
interface>org.jboss.tests.ws.jaxws.webservicesref.SecureEndpoint</service-
endpoint-interface>
    <port-qname>{http://org.jboss.ws/wsref}SecureEndpointPort</port-
qname>
    <stub-property>
      <name>javax.xml.ws.security.auth.username</name>
      <value>kermit</value>
    </stub-property>
    <stub-property>
      <name>javax.xml.ws.security.auth.password</name>
      <value>thefrog</value>
    </stub-property>
  </port-info>
</service-ref>
```

#### 9.16.4. Dispatch

XML Web サービスはサービスとサービスクライアント間の通信に XML メッセージを使用します。高レベルの JAX-WS API は Java メソッド呼び出しとそれに対応する XML メッセージの変換詳細を隠すよう設計されていますが、XML メッセージレベルでの動作が望ましい場合があります。Dispatch インターフェースはこのモードでの通信に対するサポートを提供しています。

Dispatch は 2 種類の使用モードに対応しています。 `javax.xml.ws.Service.Mode.MESSAGE` と `javax.xml.ws.Service.Mode.PAYLOAD` はそれぞれ次のように識別されます。

### Message

このモードでは、クライアントアプリケーションは直接プロトコル固有のメッセージ構造と動作します。例、SOAP プロトコルバインディングと併用する場合、クライアントアプリケーションは直接 SOAP メッセージと動作します。

### Message Payload

このモードでは、クライアントアプリケーションはメッセージ自体ではなくメッセージのペイロードと動作します。例、SOAP プロトコルバインディングと併用する場合、クライアントアプリケーションは SOAP メッセージ全体ではなく SOAP Body の内容と動作します。

Dispatch はクライアントが XML としてメッセージのペイロードまたはメッセージを構築し、目的のメッセージまたはペイロードの構造の詳細な知識を必要とする低レベルの API になります。Dispatch はあらゆるタイプのメッセージまたはメッセージペイロードの入出力に対応する汎用クラスになります。

```
Service service = Service.create(wsdlURL, serviceName);
Dispatch dispatch = service.createDispatch(portName, StreamSource.class,
Mode.PAYLOAD);

String payload = "<ns1:ping
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

payload = "<ns1:feedback
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
Source retObj = (Source)dispatch.invoke(new StreamSource(new
StringReader(payload)));
```

### 9.16.5. 非同期の呼び出し

**BindingProvider** インターフェースはクライアントによる使用を目的としたプロトコルバインディングを提供するコンポーネントを表し、プロキシによって実装、**Dispatch** インターフェースによって拡張されます。

**BindingProvider** インスタンスは非同期の動作機能を提供することができます。使用した場合、動作が完了するとレスポンスコンテキストは更新されないなど、呼び出し時に非同期の動作呼び出しが **BindingProvider** インスタンスから分離されます。代わりに、別のレスポンスコンテキストが **Response** インターフェースを使って利用できるようになります。

```
public void testInvokeAsync() throws Exception
{
    URL wsdlURL = new URL("http://" + getServerHost() + ":8080/jaxws-
samples-asynchronous?wsdl");
    QName serviceName = new QName(targetNS, "TestEndpointService");
    Service service = Service.create(wsdlURL, serviceName);
    TestEndpoint port = service.getPort(TestEndpoint.class);

    Response response = port.echoAsync("Async");

    // access future
```

```
String retStr = (String) response.get();
assertEquals("Async", retStr);
}
```

### 9.16.6. Oneway 呼び出し

@Oneway は特定の Web メソッドが入力メッセージのみで出力をもたないことを示します。一般的に one-way (一方向) メソッドは実際のビジネスメソッドを実行する前に、制御のスレッドを呼び出し元のアプリケーションに返します。

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl
{
    private static String feedback;
    ...
    @WebMethod
    @Oneway
    public void ping()
    {
        log.info("ping");
        feedback = "ok";
    }
    ...
    @WebMethod
    public String feedback()
    {
        log.info("feedback");
        return feedback;
    }
}
```

## 9.17. 共通 API

本項は [Web サービスエンドポイント](#) と [Web サービスクライアント](#) に同等に当てはまる概念について説明しています。

### 9.17.1. ハンドラフレームワーク

ハンドラフレームワークは JAX-WS プロトコルバインディングによってクライアントおよびサーバー側両方のランタイムで実装されます。バインディングプロバイダーとして集合的に知られるプロキシおよび Dispatch インスタンスはそれぞれプロトコルバインディングを使ってその抽出機能を特定のプロトコルにバインドします。

クライアントおよびサーバー側ハンドラーはハンドラーチェーンとして知られる順序付けされた一覧に並べられます。ハンドラーチェーン内のハンドラ群はメッセージが送信または受信される度に呼び出されます。インバウンドメッセージはバインディングプロバイダー処理の前に処理されます。アウトバウンドメッセージはあらゆるバインディングプロバイダー処理の後に処理されます。

ハンドラーは、インバウンドメッセージとアウトバウンドメッセージにアクセスして編集、およびプロパティセットの管理を行うメソッドを提供するメッセージコンテキストで呼び出されます。メッセージコンテキストのプロパティはハンドラー間での通信やハンドラーとクライアントおよびサービス実装間での通信を容易にするために使用することができます。異なるメッセージコンテキストタイプで異なるタイプのハンドラーが呼び出されます。



### 9.17.1.1. 論理ハンドラー

メッセージコンテキストプロパティおよびメッセージペイロードでのみ動作するハンドラです。論理ハンドラーはプロトコルにとらわれないため、メッセージのプロトコル固有部分には作用できません。論理ハンドラーは `javax.xml.ws.handler.LogicalHandler` を実装するハンドラーになります。

### 9.17.1.2. プロトコルハンドラー

メッセージコンテキストプロパティおよびプロトコル固有メッセージでのみ動作するハンドラです。プロトコルハンドラーは特定のプロトコルに固有となるため、メッセージのプロトコル特定部分にアクセスして変更することができます。プロトコルハンドラーは `javax.xml.ws.handler.LogicalHandler` 以外の `javax.xml.ws.handler.Handler` から生じたあらゆるインターフェースを実装するハンドラーになります。

### 9.17.1.3. サービスエンドポイントのハンドラー

サービスエンドポイントで、ハンドラーは `@HandlerChain` アノテーションを使って定義されます。

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
public class SOAPEndpointSourceImpl
{
    ...
}
```

ハンドラーチェーンファイルの場所は 2 種類の形式に対応しています。

1. externalForm での完全な java.net.URL。 (例、 <http://myhandlers.foo.com/handlerfile1.xml>)
2. ソースファイルまたはクラスファイルからの相対パス。 (例、 `bar/handlerfile1.xml`)

### 9.17.1.4. サービスクライアントのハンドラー

クライアント側で、ハンドラーは SEI で `@HandlerChain` アノテーションを使用して、あるいは API を使って動的に設定することができます。

```
Service service = Service.create(wsdlURL, serviceName);
Endpoint port = (Endpoint)service.getPort(Endpoint.class);

BindingProvider bindingProvider = (BindingProvider)port;
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(new LogHandler());
handlerChain.add(new AuthorizationHandler());
handlerChain.add(new RoutingHandler());
bindingProvider.getBinding().setHandlerChain(handlerChain); // important!
```

## 9.17.2. メッセージコンテキスト

`MessageContext` はすべての JAX-WS メッセージコンテキスト群のスーパーインターフェースになります。追加のメソッドおよび定数で `Map<String,Object>` を拡張しハンドラーチェーン内のハンドラ群が一連の処理に関連する状態を共有できるようにするプロパティセットを管理します。たとえば、任意のハンドラーは、そのハンドラーチェーン内の 1 つまたは複数の他のハンドラーが続いて `get` メソッド経由で取得する可能性のあるメッセージコンテキスト内に 1 プロパティを挿入するために `put` メソッドを使うことができます。

プロパティは APPLICATION または HANDLER としてスコープされます。特定のエンドポイントに関連するすべてのハンドラーに対してすべてのプロパティを使用できます。例えば、論理ハンドラーがメッセージコンテキストにプロパティを挿入すると、実行中チェーンのすべてのプロトコルハンドラーはそのプロパティを使用できます。APPLICATION にスコープされたプロパティは、クライアントアプリケーションやサービスエンドポイント実装に対しても使用できます。プロパティのデフォルトスコープは HANDLER です。

#### 9.17.2.1. メッセージコンテキストへのアクセス

ユーザーは、`@WebService` アノテーションよりハンドラーまたはエンドポイントのメッセージコンテキストへアクセスすることができます。

#### 9.17.2.2. 論理メッセージコンテキスト

`LogicalMessageContext` は呼び出し時に **Logical Handlers** へ渡されます。`LogicalMessageContext` は、メッセージペイロードを取得し変更するメソッドを用いて `MessageContext` を拡張し、メッセージのプロトコル固有のアспектへのアクセスは提供しません。プロトコルバインディングは、論理メッセージコンテキストより使用できるメッセージのコンポーネントを定義します。SOAP バインディングは、SOAP バインディングにデプロイされた論理ハンドラは SOAP ボディーの内容にはアクセス可能で、SOAP ヘッダにはアクセスできないことを定義します。XML/HTTP バインディングは、論理ハンドラーがメッセージの XML ペイロード全体にアクセスできることを定義します。

#### 9.17.2.3. SOAP メッセージコンテキスト

呼び出し時に `SOAPMessageContext` は **SOAP handlers** へ渡されます。`SOAPMessageContext` は、SOAP メッセージペイロードを取得し変更するメソッドを用いて `MessageContext` を拡張します。

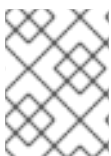
### 9.17.3. 障害の処理

実装は `SOAPFaultException` をスローすることがあります。

```
public void throwSoapFaultException()
{
    SOAPFactory factory = SOAPFactory.newInstance();
    SOAPFault fault = factory.createFault("this is a fault string!", new
    QName("http://foo", "FooCode"));
    fault.setFaultActor("mr. actor");
    fault.addDetail().addChildElement("test");
    throw new SOAPFaultException(fault);
}
```

または、アプリケーション固有のユーザー例外の場合は次のようになります。

```
public void throwApplicationException() throws UserException
{
    throw new UserException("validation", 123, "Some validation error");
}
```



#### 注記

後者の場合、JBossWS は必要な障害ラッパーの bean がデプロイの一部にない場合ランタイム時に生成します。

## 9.18. DATABINDING

### 9.18.1. アノテーションが付けられていないクラスで JAXB を使用

JAXB は Java バインディングの Java アノテーションに大きく影響されます。現在、外部バインディング設定はサポートしていません。

これをサポートするため、JAXBContext 作成中に RuntimeInlineAnnotationReader 実装を指定できる AXB RI 機能で構築しました (JAXBRIContext を参照)。

この機能は、「JAXB アノテーション導入」と呼ばれ、通常消費のために使用することができます。例えば、SVN よりチェックアウトやビルド、あるいは使用することができます。

- <http://anonsvn.jboss.org/repos/jbossws/projects/jaxbintros/>

詳細なドキュメントは次をご覧ください。

- [JAXB Introductions](#)

## 9.19. 添付

JBoss-WS4EE は SwA (SOAP with Attachments) と呼ばれる廃止された添付に依存しています。SwA には WS-I Basic Profile では禁止されている soap/encoding を使用する必要があります。JBossWS は WS-I AP 1.0 と MTOM へのサポートを提供します。

### 9.19.1. MTOM/XOP

本セクションでは、特定タイプのコンテンツを持つ XML 情報セットの連続化をより効率的に行う方法として Message Transmission Optimization Mechanism (MTOM) および XML-binary Optimized Packaging (XOP) について説明していきます。関連する仕様は以下の通りです。

- [SOAP Message Transmission Optimization Mechanism \(MTOM\)](#)
- [XML-binary Optimized Packaging \(XOP\)](#)

#### 9.19.1.1. 対応 MTOM パラメータータイプ

image/jpeg	java.awt.Image
text/xml	javax.xml.transform.Source
application/xml	javax.xml.transform.Source
application/octet-stream	javax.activation.DataHandler

上記の表では対応エンドポイントパラメーターのタイプ一覧を示しています。推奨される手段は [javax.activation.DataHandler](#) クラスを使用してバイナリデータをサービスエンドポイントパラメーターとして表す方法です。



## 注記

Microsoft エンドポイントはあらゆるデータを `application/octet-stream` として送信する傾向にあります。この曖昧性に容易に対処できる唯一の Java タイプは `javax.activation.DataHandler` になります。

### 9.19.1.2. エンドポイントごとに MTOM を有効にする方法

サーバー側で、MTOM 処理は `@BindingType` アノテーションで有効にされます。JBossWS は SOAP1.1 および SOAP1.2 を処理します。いずれも MTOM があってもなくても使用できます。

#### MTOM が有効化されたサービス実装

```
package org.jboss.test.ws.jaxws.samples.xop.doclit;

import javax.ejb.Remote;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.ws.BindingType;

@Remote
@WebService(targetNamespace = "http://org.jboss.ws/xop/doclit")
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT, parameterStyle =
SOAPBinding.ParameterStyle.BARE)
@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
(1)
public interface MTOMEndpoint
{
    ...
}
```

#### 1. MTOM 有効の SOAP 1.1 バインディング ID

#### MTOM が有効化されたクライアント

Web サービスクライアントは前述と同じ方法を使用するか、または MTOM の有効化を **Binding API** に依存することができます (`org.jboss.test.ws.jaxws.samples.xop.doclit.XOPTestCase` からの抜粋):

```
...
Service service = Service.create(wsdlURL, serviceName);
port = service.getPort(MTOMEndpoint.class);

// enable MTOM
binding = (SOAPBinding)((BindingProvider)port).getBinding();
binding.setMTOMEnabled(true);
```



## 注記

デプロイメントのデフォルト設定に JBossWS 設定テンプレートを使用します。

### 9.19.2. SwaRef

[WS-I Attachment Profile 1.0](#) は [swaRef](#) 使って MIME 添付の部分参照するメカニズムを定義します。

このメカニズムでは、タイプ `ws:swaRef` の XML 要素のコンテンツは MIME 添付として送信され、SOAP Body 内側の要素は [RFC 2111](#) で定義されるように CID URI スキームでこの添付への参照を格納します。

### 9.19.2.1. SwaRef と JAX-WS エンドポイントを併用

JAX-WS エンドポイントはすべてのマーシャル／アンマーシャルを JAXB API に委任します。**DataHandler** タイプに対して SwaRef エンコーディングを有効にする最もシンプルな方法は、以下のようパイロード bean に `@XmlAttachmentRef` アノテーションを付ける方法です。

```
/**
 * Payload bean that will use SwaRef encoding
 */
@XmlRootElement
public class DocumentPayload
{
    private DataHandler data;

    public DocumentPayload()
    {
    }

    public DocumentPayload(DataHandler data)
    {
        this.data = data;
    }

    @XmlElement
    @XmlAttachmentRef
    public DataHandler getData()
    {
        return data;
    }

    public void setData(DataHandler data)
    {
        this.data = data;
    }
}
```

エンドポイントをラップしたドキュメントでは、サービスエンドポイントインターフェースで `@XmlAttachmentRef` アノテーションを指定することさえ可能です。

```
@WebService
public interface DocWrappedEndpoint
{
    @WebMethod
    DocumentPayload beanAnnotation(DocumentPayload dhw, String test);

    @WebMethod
    @XmlAttachmentRef
    DataHandler parameterAnnotation(@XmlAttachmentRef DataHandler data,
    String test);
}
```

次にメッセージは CID で添付部分を参照することになります。

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
  <env:Header/>
  <env:Body>
    <ns2:parameterAnnotation
      xmlns:ns2='http://swaref.samples.jaxws.ws.test.jboss.org/'>
      <arg0>cid:0-1180017772935-32455963@ws.jboss.org</arg0>
      <arg1>Wrapped test</arg1>
    </ns2:parameterAnnotation>
  </env:Body>
</env:Envelope>
```

### 9.19.2.2. WSDL から開始

コントラクトから開始する方法を選択した場合は、SwaRef エンコーディングを使用すべきあらゆる要素宣言が単純に wsi:swaRef スキーマタイプを参照することを確認する必要があります。

```
<element name="data" type="wsi:swaRef"
  xmlns:wsi="http://ws-i.org/profiles/basic/1.1/xsd"/>
```

いずれの wsi:swaRef スキーマタイプも次に DataHandler にマッピングされます。

## 9.20. ツール

JBossWS で提供される JAX-WS ツールはさまざまな方法で 사용할 ことができます。まず、サーバー側デプロイメントについて考察してからクライアントを見ていきます。Web Service Endpoint (サーバー側) を開発する場合、Java (bottom-up 開発) から開始するか、サービスを定義する抽出コントラクト (WSDL) から開始するか (top-down 開発) を選択するオプションがあります。これが新しいサービスなら (既存コントラクトがない) bottom-up の方が時間的に早くなり、サービスが正しく起動して実行するようクラスにいくつかアノテーションを追加するだけになります。しかし、既に定義されているコントラクトがあるサービスを開発する場合は、提供されているツールがアノテーション付きコードを生成するため top-down の方法を選択した方がずっと簡単になります。

Bottom-up を使用した場合:

- すでに存在する EJB3 bean を Web Service として公開
- 新しいサービスを提供する、コントラクトの生成が必要

Top-down を使用した場合:

- 旧式クライアントとの互換性を維持しながら既存 Web サービスの実装を置き換える
- サードパーティによって指定されるコントラクトに適合するサービスを公開する (例: 定義済みのプロトコルを使ってコールバックを行うベンダー)
- 前もって手動で開発した WSDL と XML スキーマを厳守するサービスを作成する

次の JAX-WS コマンドラインツールが JBossWS に含まれています。

コマンド	内容
------	----

<a href="#">wsprovide</a>	移植性のある JAX-WS アーチファクトを生成し、抽象規定を提供します。bottom-up 開発で使用されます。
<a href="#">wsconsume</a>	抽象規定 (SWDL と Schema ファイル) を摂取して、サーバーおよびクライアントの両方にアーチファクトを生成します。top-down およびクライアント開発に使用されます。
<a href="#">wsrunclient</a>	JBossWS クラスパスを使って Java クライアント (1 つの main メソッドを持つ) を実行します。

### 9.20.1. Bottom-Up (wsprovide を使用)

bottom-up の方法では使用するサービスに Java コードを開発してそれに JAX-WS アノテーションを付ける必要があります。これらのアノテーションはサービス用に生成されるコントラクトのカスタマイズに使用できます。たとえば、マッピングするオペレーション名は何にしても構いません。しかし、アノテーションはすべて目的に合ったデフォルトを持つので、`@WebService` アノテーションのみが必要になります。

次のように単一クラスを作成するなどできるだけシンプルにすることができます。

```
package echo;

@javax.jws.WebService
public class Echo
{
    public String echo(String input)
    {
        return input;
    }
}
```

JSE または EJB3 のデプロイメントはこのクラスを使って構築することができるため、JBossWS にデプロイが必要となる Java コードはこれのみになります。WSDL および「ラッパークラス」と呼ばれるその他すべての Java アーチファクトはデプロイ時にユーザーによって生成されます。これは JAX-WS の仕様外で、オフラインツールを使用したラッパークラスの生成が必要となります。これはベンダーの実装問題に対処するため、開発者に対して一連の追加ステップによる負担をかけないようにするために生成しています。ただし、他のアプリケーションサーバーに対しても移植性のあるデプロイメントにしたい場合は、ツールを使って生成されるクラスを追加する必要があります。

これが [wsprovide](#) ツールの主要な目的となり、移植性のある JAX-WS アーチファクトを生成します。さらに、サービスに抽象規定 (WSDL ファイル) を「提供」するために使用することもできます。「-w」オプションを使って [wsprovide](#) を呼び出して、これを取得することができます。

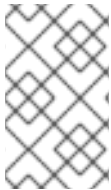
```
$ javac -d . -classpath jboss-jaxws.jar Echo.java
$ wsprovide -w echo.Echo
Generating WSDL:
EchoService.wsdl
Writing Classes:
echo/jaxws/Echo.class
echo/jaxws/EchoResponse.class
```

WSDL をよく見てみると EchoService という名前のサービスがわかります。

```
<service name='EchoService'>
  <port binding='tns:EchoBinding' name='EchoPort'>
    <soap:address location='REPLACE_WITH_ACTUAL_URL' />
  </port>
</service>
```

期待通り、このサービスはあるオペレーション、「echo」を定義しています。

```
<portType name='Echo'>
  <operation name='echo' parameterOrder='echo'>
    <input message='tns:Echo_echo' />
    <output message='tns:Echo_echoResponse' />
  </operation>
</portType>
```



### 注記

**JBossWS の開発時にはこのツールを実行する必要はないことを念頭においてください。** これは、移植性のあるアーチファクトやサービス用の抽象規定を生成する場合にのみ必要になります。

JBoss Enterprise Application Platform 上にデプロイメントに対する POJO エンドポイントを作成してみましょう。簡単な **web.xml** を作成する必要があります。

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>Echo</servlet-name>
    <servlet-class>echo.Echo</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Echo</servlet-name>
    <url-pattern>/Echo</url-pattern>
  </servlet-mapping>
</web-app>
```

これで **web.xml** および単一クラスを使用して WAR を作成できるようになりました。

```
$ mkdir -p WEB-INF/classes
$ cp -rp echo WEB-INF/classes/
$ cp web.xml WEB-INF
$ jar cvf echo.war WEB-INF
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
```



```
adding: WEB-INF/classes/echo/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/Echo.class(in = 340) (out= 247)(deflated 27%)
adding: WEB-INF/web.xml(in = 576) (out= 271)(deflated 52%)
```

次に war をデプロイします。

```
cp echo.war <replaceable>$JBOSS_HOME</replaceable>/server/default/deploy
```

JBossWS はデプロイ時に内部的に [wsprovide](#) を呼び出し、WSDL を生成します。デプロイメントに成功すれば、デフォルトの設定を使用していることになり、<http://localhost:8080/echo/Echo?wsdl> にあるはずです。

移植性のある JAX-WS デプロイメントの場合、先に生成されるラッパークラスをデプロイメントに追加することができます。

### 9.20.2. Top-Down (wsconsume を使用)

top-down の開発はサービスの抽象規定で開始し、これは WSDL ファイルおよび 0 またはそれ以上のスキーマファイルを含みます。[wsconsume](#) ツールは次にこの規定を消費するため使用され、それを定義するアノテーションが付けられた Java クラス (およびオプションでソース) を生成します。



#### 注記

[wsconsume](#) は Unix システム上の symlinks との動作には問題があるようです。

bottom-up の例から WSDL ファイルを使用して、このサービスを厳守する新しい Java 実装を生成することが可能です。「-k」オプションは単にクラスを提供するだけでなく、生成される Java ソースファイルを維持するために [wsconsume](#) に渡されます。

```
$ wsconsume -k EchoService.wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

次の表では各生成ファイルの目的を示します。

ファイル	目的
Echo.java	Service Endpoint Interface
Echo_Type.java	要求メッセージに対するラッパー bean

EchoResponse.java	レスポンスメッセージに対するラッパー bean
ObjectFactory.java	JAXB XML レジストリ
package-info.java	JAXB パッケージアノテーションのホルダー
EchoService.java	JAX-WS クライアントによってのみ使用される

Service Endpoint Interface をよく見てみると、同じコントラクトに対して評価するのに、bottom-up の例での手動で記述されたクラスにあるアノテーションより明示的であるのに気づくはずです。

```
@WebService(name = "Echo", targetNamespace = "http://echo/")
public interface Echo
{
    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "echo", targetNamespace = "http://echo/",
        className = "echo.Echo_Type")
    @ResponseWrapper(localName = "echoResponse", targetNamespace =
        "http://echo/", className = "echo.EchoResponse")
    public String echo(@WebParam(name = "arg0", targetNamespace = "")
        String arg0);
}
```

ここにはないのは (パッケージング以外) その実装クラスのみで、上記のインターフェースを使って記述することができます。

```
package echo;

@javax.jws.WebService(endpointInterface="echo.Echo")
public class EchoImpl implements Echo
{
    public String echo(String arg0)
    {
        return arg0;
    }
}
```

### 9.20.3. クライアント側

クライアント側について詳細に説明する前に、Web サービスの中心となる分離の概念を理解することが重要となります。このような方法で利用可能ですが、Web サービスは内部 RPC に対して最も適応しているわけではなく、これに関しては (CORBA や RMI など) さらに優れた技術が存在します。Web サービスは相互運用性のある粗粒子の通信向けに開発されています。Web サービスのやりとりを行うパーティシパントが、特定の場所で特定のオペレーティングシステムを使用し、もしくは特定のプログラミング言語で記述されているとの前提、保証はありません。そのため、クライアントとサーバーの実装を明確に分離することが重要となります。抽象コントラクトの定義のみを共通になるようにするだけでいいのです。何らかの理由でソフトウェアがこの原理に従わない場合は、Web サービスを使用すべきではありません。このような理由により、クライアントが同じサーバーで稼働している場合でも、クライアントの開発技法としてトップダウンアプローチを用いることが推奨されます。

[wsprovide](#) によってオフラインで生成されたものではなくデプロイされた WSDL を使用して、トップダウンの項のプロセスを再度確認してみましょう。soap:address に対する正しい値を取得するためだけ

にこれを実行します。この値はコンテナ設定の内容が基になるため、デプロイ時に算出されなければなりません。WSDL ファイルを編集することもできますが、パスが正しくなるよう注意してください。

オフラインバージョン:

```
<service name='EchoService'>
  <port binding='tns:EchoBinding' name='EchoPort'>
    <soap:address location='REPLACE_WITH_ACTUAL_URL' />
  </port>
</service>
```

オンラインバージョン:

```
<service name="EchoService">
  <port binding="tns:EchoBinding" name="EchoPort">
    <soap:address
location="http://localhost.localdomain:8080/echo/Echo"/>
  </port>
</service>
```

[wsconsume](#) を用いてオンラインでデプロイしたバージョンを使用する:

```
$ wsconsume -k http://localhost:8080/echo/Echo?wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

**EchoService.java** は top-down セクションでは確認しなかったクラスでした。WSDL の取得先となる場所を格納する方法に注目してください。

```
@WebServiceClient(name = "EchoService", targetNamespace = "http://echo/",
wsdlLocation = "http://localhost:8080/echo/Echo?wsdl")
public class EchoService extends Service
{
    private final static URL ECHOSERVICE_WSDL_LOCATION;

    static
    {
        URL url = null;
        try
        {
            url = new URL("http://localhost:8080/echo/Echo?wsdl");
        }
        catch (MalformedURLException e)
        {
        }
    }
}
```

```

        e.printStackTrace();
    }
    ECHOSERVICE_WSDL_LOCATION = url;
}

public EchoService(URL wsdlLocation, QName serviceName)
{
    super(wsdlLocation, serviceName);
}

public EchoService()
{
    super(ECHOSERVICE_WSDL_LOCATION, new QName("http://echo/",
"EchoService"));
}

@WebEndpoint(name = "EchoPort")
public Echo getEchoPort()
{
    return (Echo)super.getPort(new QName("http://echo/", "EchoPort"),
Echo.class);
}
}

```

ご覧のように、上記は JAX-WS、**javax.xml.ws.Service** 内のメインクライアントエントリポイントを拡張するクラスを生成しました。**Service** を直接使用できる一方、設定情報を提供してくれるためこちらの方がずっと簡単になります。本当に注意しなければならないメソッドは **getEchoPort()** メソッドのみで、**Service Endpoint Interface** のインスタンスを返します。これであらゆる WS オペレーションが返されるインターフェースでメソッドを呼び出すだけで呼び出し可能になります。



### 注記

実稼働アプリケーションでリモート WSDL URL へ参照することは推奨されません。参照すると Service Object をインスタンス化する度にネットワーク I/O が発生するためです。保存したローカルコピーに対してツールを使用するか、URL バージョンのコンストラクタを使用して新しい WSDL の場所を提供するようにします。

あとはクライアントを記述してコンパイルするだけです。

```

import echo.*;
..
public class EchoClient
{
    public static void main(String args[])
    {
        if (args.length != 1)
        {
            System.err.println("usage: EchoClient <message>");
            System.exit(1);
        }

        EchoService service = new EchoService();
        Echo echo = service.getEchoPort();
    }
}

```

```

        System.out.println("Server said: " + echo.echo(args[0]));
    }
}

```

これで [wsrunclient](#) ツールを使って簡単に実行できるようになります。これは必要なクラスパスで java を呼び出す便利なツールです。

```

$ wsrunclient EchoClient 'Hello World!'
Server said: Hello World!

```

起動時のオペレーションのエンドポイントアドレス変更は簡単です。以下のように `ENDPOINT_ADDRESS_PROPERTY` を設定します。

```

...
EchoService service = new EchoService();
Echo echo = service.getEchoPort();

/* Set NEW Endpoint Location */
String endpointURL = "http://NEW_ENDPOINT_URL";
BindingProvider bp = (BindingProvider)echo;
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
    endpointURL);

System.out.println("Server said: " + echo.echo(args[0]));
...

```

#### 9.20.4. コマンドライン & Ant タスク参照

- [wsconsume 参照ページ](#)
- [wsprovide 参照ページ](#)
- [wsrunclient 参照ページ](#)

#### 9.20.5. JAX-WS バインディングのカスタマイズ

バインディングカスタマイズの導入

- <http://java.sun.com/webservices/docs/2.0/jaxws/customizations.html>

バインディングカスタマイズのファイル用スキーマは次をご覧ください。

- [バインディングのカスタマイズ](#)

### 9.21. WEB サービスの拡張

#### 9.21.1. WS-Addressing

本項では、[WS-Addressing](#) を使用してステートフルサービスのエンドポイントを提供する方法について説明しています。

##### 9.21.1.1. 仕様

WS-Addressing は W3C Candidate Recommendation 17 August 2005 にある次の仕様の組み合わせで定義されます。WS-Addressing API は [JSR-224 - Java API for XML-Based Web Services \(JAX-WS\)](#) で標準化されます。

- [Web Services Addressing 1.0 - Core](#)
- [Web Services Addressing 1.0 - SOAP Binding](#)

### 9.21.1.2. エンドポイントの処理



#### 注記

以下の情報は、JBoss Web Services CXF スタックと併用するべきではありません。

次のエンドポイント実装はステートフルの一般的なショッピングカートアプリケーション用のオペレーションセットを持っています。

```
@WebService(name = "StatefulEndpoint", targetNamespace =
"http://org.jboss.ws/samples/wsaddressing", serviceName = "TestService")
@Addressing(enabled=true, required=true)
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class StatefulEndpointImpl implements StatefulEndpoint,
ServiceLifecycle
{
    @WebMethod
    public void addItem(String item)
    { ... }

    @WebMethod
    public void checkout()
    { ... }

    @WebMethod
    public String getItems()
    { ... }
}
```

JAX-WS 2.1 で定義された `javax.xml.ws.soap.Addressing` アノテーションを使用してサーバー側のアドレッシングハンドラーを有効にします。

### 9.21.1.3. クライアントの処理

クライアントは JAX-WS 2.1 API の `javax.xml.ws.soap.AddressingFeature` を使用して WS-Addressing を有効にします。

```
Service service = Service.create(wsdlURL, serviceName);
port1 = (StatefulEndpoint)service.getPort(StatefulEndpoint.class, new
AddressingFeature());
```

ステートフルエンドポイントに接続しているクライアント

```
public class AddressingStatefulTestCase extends JBossWSTest
{
```

```

...
public void testAddItem() throws Exception
{
    port1.addItem("Ice Cream");
    port1.addItem("Ferrari");

    port2.addItem("Mars Bar");
    port2.addItem("Porsche");
}

public void testGetItems() throws Exception
{
    String items1 = port1.getItems();
    assertEquals("[Ice Cream, Ferrari]", items1);

    String items2 = port2.getItems();
    assertEquals("[Mars Bar, Porsche]", items2);
}
}

```

## SOAP メッセージの交換

以下で交換されている SOAP メッセージを確認できます。

```

<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
<wsa:To>uri:jbossws-samples-wsaddr/TestService</wsa:To>
<wsa:Action>http://org.jboss.ws/addressing/stateful/action</wsa:Action>
<wsa:ReferenceParameters>
<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</wsa:ReferenceParameters>
</env:Header>
<env:Body>
<ns1:addItem xmlns:ns1='http://org.jboss.ws/samples/wsaddr'>
<String_1>Ice Cream</String_1>
</ns1:addItem>
</env:Body>
</env:Envelope>

<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
<wsa:To>http://www.w3.org/2005/08/addressing/anonymous</wsa:To>
<wsa:Action>http://org.jboss.ws/addressing/stateful/actionReply</wsa:Action>
</env:Header>
<env:Body>
<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</env:Header>
<env:Body>
<ns1:addItemResponse xmlns:ns1='http://org.jboss.ws/samples/wsaddr'>
</ns1:addItemResponse>
</env:Body>
</env:Envelope>

...

<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
<wsa:To>uri:jbossws-samples-wsaddr/TestService</wsa:To>

```

```

<wsa:Action>http://org.jboss.ws/addressing/stateful/action</wsa:Action>
<wsa:ReferenceParameters>
<ns1:clientid xmlns:ns1='http://somens'>clientid-1</ns1:clientid>
</wsa:ReferenceParameters>
</env:Header>
<env:Body>
<ns1:getItems xmlns:ns1='http://org.jboss.ws/samples/wsaddr' />
</env:Body>
</env:Envelope>

<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
<wsa:To>http://www.w3.org/2005/08/addressing/anonymous</wsa:To>
<wsa:Action>http://org.jboss.ws/addressing/stateful/actionReply</wsa:Action>
</env:Header>
<env:Body>
<ns1:getItemsResponse xmlns:ns1='http://org.jboss.ws/samples/wsaddr'>
<result>[Ice Cream, Ferrari]</result>
</ns1:getItemsResponse>
</env:Body>
</env:Envelope>

```

### 9.21.2. WS-Security

WS-Security はメッセージレベルのセキュリティを処理します。承認、暗号化、Web サービスのデジタル署名処理の標準化を行います。SSL などのトランスポートセキュリティモデルとは異なり、WS-Security はセキュリティを直接 Web サービスメッセージの要素に適用します。結果、あらゆるメッセージモデル (ポイントツーポイント、マルチホップリレーなど) が使用できるようになるため ご利用中の Web サービスの柔軟性が向上されます。

本章ではシンプルな SOAP メッセージの署名と暗号化を行うため WS-Security の使用方法を説明します。

#### 仕様

WS-Security は次の仕様の組み合わせで定義されます。

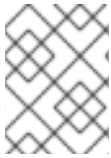
- [SOAP Message Security 1.0](#)
- [Username Token Profile 1.0](#)
- [X.509 Token Profile 1.0](#)
- [W3C XML Encryption](#)
- [W3C XML Signature](#)
- [Basic Security Profile 1.0 \(起草段階\)](#)

#### 9.21.2.1. エンドポイント設定

JBoss WS はハンドラーを使用して WS-security でエンコード化されたリクエストを識別しセキュリティコンポーネントを呼び出すことでメッセージの署名と暗号化を行います。セキュリティ処理を可能にするため、クライアントおよびサーバー側は該当するハンドラー設定を含む必要があります。事前定



義されている [JAX-WS Endpoint Configuration](#) または [JAX-WS Client Configuration](#) をそれぞれ参照することが推奨されます。



### 注記

エンドポイント設定と WSSE 宣言の両方を設定する必要があります。個別に設定していきます。

#### 9.21.2.2. サーバー側 WSSE 宣言 (jboss-wsse-server.xml)

この例ではメッセージボディを署名するためクライアントおよびサーバーの両方を設定しています。また、クライアントとサーバーはいずれも相手からこれを必要とします。このため、クライアントまたはサーバーいずれかのセキュリティ配備記述子を削除すると、もう一方がそのメッセージは正しいセキュリティ要件を満たしていなかったことを伝える障害を送出するのがわかります。

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.jboss.com/ws-security/config
http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
(1) <key-store-file>WEB-INF/wsse.keystore</key-store-file>
(2) <key-store-password>jbossws</key-store-password>
(3) <trust-store-file>WEB-INF/wsse.truststore</trust-store-file>
(4) <trust-store-password>jbossws</trust-store-password>
(5) <config>
(6)   <sign type="x509v3" alias="wsse"/>
(7)   <requires>
(8)     <signature/>
    </requires>
  </config>
</jboss-ws-security>
```

1. これは、使用したいキーストアは war ファイルにある **WEB-INF/wsse.keystore** であることを指定します。
2. ストアパスワードは「jbossws」であることを指定しています。パスワードは {EXT} と {CLASS} のコマンドを使って暗号化することができます。使用方法についてはサンプルをご覧ください。
3. これは、使用したいキーストアは war ファイルにある **WEB-INF/wsse.truststore** であることを指定します。
4. 信頼できるストアパスワードも「jbossws」であることを指定しています。パスワードは {EXT} と {CLASS} のコマンドを使って暗号化することができます。使用方法についてはサンプルをご覧ください。
5. root config ブロックはここから開始します。root config ブロックはこの war ファイル内のすべてのサービスに対してデフォルト設定となります。
6. サーバーはすべてのレスポンスのメッセージボディに署名しなければならないという意味になります。Type は X.509v3 証明書 (標準の証明書) を使用しているという意味です。エイリアスオプションは、署名に使用する証明書とキーの組み合わせは「wsse」エイリアス配下のキーストアにあることを示しています。
7. オプションの requires ブロックはここから開始します。このブロックはサーバーがメッセージを受信するときに満たされる必要のある全セキュリティ要件を指定します。

8. この war ファイル内の Web サービスはすべて署名すべきメッセージボディを必要とします。

デフォルトではエンドポイントは WS-Security 設定を使用しません。専用の `@EndpointConfig` アノテーションを使って設定名を指定します。使用できる設定名の一覧は [JAX-WS\\_Endpoint\\_Configuration](#) をご覧ください。

```
@WebService
@EndpointConfig(configName = "Standard WSSecurity Endpoint")
public class HelloJavaBean
{
    ...
}
```

### 9.21.2.3. クライアント側 WSSE 宣言 (jboss-wsse-client.xml)

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.jboss.com/ws-security/config
http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
(1) <config>
(2)     <sign type="x509v3" alias="wsse"/>
(3)     <requires>
(4)         <signature/>
        </requires>
    </config>
</jboss-ws-security>
```

1. root config ブロックはここから開始します。root config ブロックはすべての Web サービスクライアントに対してデフォルトの設定となります (Call、Proxy オブジェクト)。
2. クライアントは送信するすべての要求のメッセージボディに署名しなければならないという意味になります。Type は X.509v3 証明書 (標準の証明書) を使用しようとしていることを意味します。エイリアスオプションは署名に使用する証明書とキーの組み合わせが「wsse」エイリアス配下のキーストアにあることを示しています。
3. オプションの requires ブロックはここから開始します。このブロックはクライアントがレスポンスを受信するときに満たされなければならない全セキュリティ要件を指定します。
4. すべての Web サービスクライアントは署名されているレスポンスメッセージを受信しなければならないという意味になります。

#### 9.21.2.3.1. クライアント側キーストアの設定

クライアントアプリケーションは代わりに wsse System プロパティを使用するため、キーストアや信頼できるストアは指定しませんでした。これが Web または ejb クライアントの場合なら (war ファイルまたは ejb ファイル内の webservice クライアントの意)、クライアントの記述子にこれらを指定することになります。

以下に JBoss WS の例からの抜粋を示します。

```
<sysproperty key="org.jboss.ws.wsse.keyStore"
value="${tests.output.dir}/resources/jaxrpc/samples/wssecurity/wsse.keystore"/>
<sysproperty key="org.jboss.ws.wsse.trustStore"
value="${tests.output.dir}/resources/jaxrpc/samples/wssecurity/wsse.trusts
```

```

tore"/>
<sysproperty key="org.jboss.ws.wsse.keyStorePassword" value="jbossws"/>
<sysproperty key="org.jboss.ws.wsse.trustStorePassword" value="jbossws"/>
<sysproperty key="org.jboss.ws.wsse.keyStoreType" value="jks"/>
<sysproperty key="org.jboss.ws.wsse.trustStoreType" value="jks"/>

```

## SOAP メッセージの交換

下記には、セキュリティヘッダーの詳細が省略されている受信 SOAP メッセージがあります。SOAP ボディは平文のままですがセキュリティヘッダーが署名されているため送信中に操作されないという仕組みです。

```

<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header>
    <wsse:Security env:mustUnderstand="1" ...>
      <wsu:Timestamp wsu:Id="timestamp">...</wsu:Timestamp>
      <wsse:BinarySecurityToken ...>
        ...
      </wsse:BinarySecurityToken>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        ...
      </ds:Signature>
    </wsse:Security>
  </env:Header>
  <env:Body wsu:Id="element-1-1140197309843-12388840" ...>
    <ns1:echoUserType xmlns:ns1="http://org.jboss.ws/samples/wssecurity">
      <UserType_1 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <msg>Kermit</msg>
      </UserType_1>
    </ns1:echoUserType>
  </env:Body>
</env:Envelope>

```

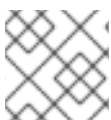
### 9.21.2.4. BouncyCastle JCE プロバイダーのインストール

下記の情報は オリジナルは[The Legion of the Bouncy Castle](#) より提供されています。

**java.security** プロパティファイルにエントリを追加すると静的登録よりプロバイダーを環境の一部として設定することができます。 **java.security** プロパティセキュリティファイルは、**\$JAVA\_HOME/jre/lib/security/java.security** にあります (**\$JAVA\_HOME** は JDK および JRE ディストリビューションの場所に置き換えます)。このファイルに詳細手順が記載されていますが、主に次の行を追加することになります。

```
security.provider.<n>=org.bouncycastle.jce.provider.BouncyCastleProvider
```

<n> はプロバイダーを配置したい場所になります。



#### 注記

Sun 提供のプロバイダーが 1 番目にならないと問題が発生する可能性があります。

プロバイダー jar を格納する場所はユーザーが決めることができますが、jdk5 の場合は **\$JAVA\_HOME/jre/lib/ext** に格納するのがよいでしょう。ウィンドウの下は通常 Java の JRE インストールと JDK インストールになるはずですが、正しくインストールされたにも拘らず動作しない場合

は、プロバイダーインストールが使用されていない可能性が高いでしょう。

### 9.21.2.5. Username Token 認証 **JBOSSCC-50**

Username Token を使ってクライアントを認証する必要がある場合、JAAS 統合は受け取ったトークンと設定済みのJBoss JAAS セキュリティドメインを照合します。

#### 例9.1 ユーザー名トークンの基本設定

この機能を実装するには、<jboss-ws-security>要素を以下の情報を含んだ**jboss-wsse-client.xml** に貼り付ける必要があります。

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                    xsi:schemaLocation="http://www.jboss.com/ws-
security/config
http://www.jboss.com/ws-
security/schema/jboss-ws-security_1_0.xsd">
  <config>
    (1)   <username/>
    (2)   <timestamp ttl="300"/>
  </config>
</jboss-ws-security>
```

2 行目では、メッセージ内に<timestamp> 要素が存在していなければならない、そのメッセージは300秒よりも古くはないと、指定しています。時間の制限 (秒) を使ってリプレイ攻撃を回避します。

その後、両方のヘッダーが一致するように、**jboss-wsse-server.xml** ファイルにて同じ<timestamp> 要素と **seconds** 属性を指定する必要があります。また、<requires/> 要素を指定しこの条件を強化する必要があります。

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                    xsi:schemaLocation="http://www.jboss.com/ws-
security/config
http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd">
  <config>
    <timestamp ttl="300"/>
    <requires/>
  </config>
</jboss-ws-security>
```



## 警告

この設定例では、シンプルテキストのユーザー情報がSOAPヘッダで送信されることとなります。JBossWS Secure Transportの実装を検討してください。

## パスワードダイジェスト、Nonce、タイムスタンプ

例9.1「ユーザー名トークンの基本設定」では、平文としてクライアントのパスワードが送信されます。さらにリプレイ攻撃から保護されるようダイジェストパスワード、*nonce*、タイムスタンプの組み合わせを使うことが可能です。

パスワードダイジェストを有効にするには、例9.2「パスワードダイジェストの有効化」で説明されているように以下の項目を実装する必要があります：

## 例9.2 パスワードダイジェストの有効化

`jboss-wsse-client.xml` ファイルの<username> 要素にて：

- *digestPassword* 属性を有効化
- *nonces* および *timestamps* を有効化

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/ws-security/config
    http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
  <config>
    (3)   <username digestPassword="true" useNonce="true"
    useCreated="true"/>
          <timestamp ttl="300"/>
        </config>
  </jboss-ws-security>
```

また、`login-config.xml` ファイルで、Username Token モジュールオプションを実装する必要があります。

## 例9.3 UsernameTokenCallback モジュール

```
<application-policy name="JBossWSDigest">
  <authentication>
    <login-module
      code="org.jboss.security.auth.spi.UsersRolesLoginModule"
      flag="required">
      <module-option name="usersProperties">META-INF/jbossws-
        users.properties</module-option>
      <module-option name="rolesProperties">META-INF/jbossws-
        roles.properties</module-option>
      <module-option name="hashAlgorithm">SHA</module-option>
      <module-option name="hashEncoding">BASE64</module-option>
```

```

        <module-option name="hashUserPassword">false</module-option>
        <module-option name="hashStorePassword">true</module-option>
        <module-option
name="storeDigestCallback">org.jboss.ws.extensions.security.auth.callbac
k.UsernameTokenCallback</module-option>
        <module-option name="unauthenticatedIdentity">anonymous</module-
option>
    </login-module>
</authentication>
</application-policy>

```

より高度なカスタムのログインモジュールを使い、リプレイ攻撃に対するセキュリティを高めたい場合があるかもしれません。以下を実装していると、ご自身のカスタムログインモジュールを利用することができます。

- **UsernameTokenCallback** コールバックをログインモジュールにプラグします。
- **org.jboss.security.auth.spi.UsernamePasswordLoginModule** を継承します。
- 例9.3「**UsernameTokenCallback モジュール**」で示されているように、ハッシュ属性 (**hashAlgorithm**、**hashEncoding**、**hashUserPassword**、**hashStorePassword**)を設定します。

#### 拡張調整 : Nonce ファクトリ

Nonce が作成され、その後サーバー側でチェック、格納される方法により、リプレイ攻撃に対するセキュリティレベルが左右されます。現在、JBossWS では、サーバー側で受け取ったトークンをキャッシュ化しないという Nonce ストアの基本実装が同梱されています。

**NonceFactory** と **NonceStore** インターフェースを実装することでご利用中のモジュールにより複雑な実装をプラグすることができます。これらのインターフェース `org.jboss.ws.extensions.security.nonce` パッケージにあります。

これらのインターフェースを含めると、**jboss-wsse-server.xml** ファイルの `<nonce-factory-class>` 要素を使ってご利用中のファクトリクラスを指定します。

#### 拡張調整 : タイムスタンプの検証

**wsse:Security** ヘッダーに **Timestamp** がある場合、ヘッダー検証にあたり時間比較における誤差の許容範囲はありません。メッセージが少しでも未来の時間に作成されている場合、メッセージの有効期限が切れている場合は拒否されます。 `<timestamp-verification>` と呼ばれる新しい要素が **wsse** 設定では利用可能です。例9.4「`<timestamp-verification>` 設定」では、`<timestamp-verification>` 要素に必要な属性について説明しています。

#### 例9.4 `<timestamp-verification>` 設定

`<timestamp-verification>` 要素属性により、**'Timestamp'** ヘッダーの **'Created'** あるいは **'Expires'** 要素を検証時に利用する許容範囲 (秒) を指定できます。

```

<jboss-ws-security xmlns='http://www.jboss.com/ws-security/config'
                    xmlns:xsi='http://www.w3.org/2001/XMLSchema-
instance'
                    xsi:schemaLocation='http://www.jboss.com/ws-
security/config
http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd'>

```



```
<timestamp-verification createdTolerance="5" warnCreated="false"
expiresTolerance="10" warnExpires="false" />
</jboss-ws-security>
```

### **createdTolerance**

どの程度先のメッセージであれば承認されるか、その秒数。デフォルト値は **0** になります。

### **expiresTolerance**

Expired と分類されてからメッセージが拒否されるまでの秒数。デフォルト値は **0** になります。

### **warnCreated**

'Created' が未来の値のメッセージを承認した場合、警告メッセージをログに残すかどうかを指定します。デフォルト値は、**true**です。

### **warnExpires**

'Expired' の値が過去のメッセージを承認した場合、警告メッセージをログに残すかどうかを指定します。デフォルト値は、**true**です。



### 注記

**warnCreated** および **warnExpires** 属性を使い、通常は拒否されているメッセージで承認したものを特定することができます。このデータを使うことで、クライアントのメッセージを拒否することなしにサーバータイムと同期されていないクライアントを特定することができます。

#### 9.21.2.5.1. セキュアトランスポート

#### 9.21.2.6. X509 Certificate Token

X509v3 証明書を使うことで、メッセージの署名と暗号化を両方行うことができます。

#### 暗号化

暗号化の設定には、[例9.5「X509 暗号化設定」](#)の項目を指定する必要があります。クライアント、サーバーともに同じ設定となります。

#### 例9.5 X509 暗号化設定

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                    xsi:schemaLocation="http://www.jboss.com/ws-
security/config
http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd">
(1) <key-store-file>WEB-INF/bob-sign_enc.jks</key-store-file>
    <key-store-password>password</key-store-password>
    <key-store-type>jks</key-store-type>
    <trust-store-file>WEB-INF/wsse10.truststore</trust-store-file>
    <trust-store-password>password</trust-store-password>
```

```

    <config>
      <timestamp ttl="300"/>
(2)    <sign type="x509v3" alias="1" includeTimestamp="true"/>
(3)    <encrypt type="x509v3"
        alias="alice"
        algorithm="aes-256"
        keyWrapAlgorithm="rsa_oaep"
        tokenReference="keyIdentifier" />
(4)    <requires>
        <signature/>
        <encryption/>
      </requires>
    </config>
  </jboss-ws-security>

```

サーバー設定には、以下の暗号化情報が含まれます：

1. キーストアおよびトラストストア情報：各ストアの場所、パスワード、ストアの種類
2. 署名設定：使用する証明書と鍵のペアのエイリアスを提供する必要があります。***includeTimestamp*** は、改竄を防ぐためにtimestamp を署名するか指定します。
3. 暗号化設定：使用する証明書とキーのペアのエイリアスを提供する必要があります。詳細については、[アルゴリズム](#) を参照してください。
4. オプションのセキュリティ要件：受信メッセージは両方、署名、暗号化する必要があります。

## 動的暗号化

複数のクライアントに返信する場合、サービスプロバイダは、正しい公開鍵を使って宛先に基づいて、メッセージを暗号化する必要があります。JBossWS で WS-Security をネイティブ実装すると、受信メッセージで受け取り (確認された) 署名から正しい鍵を取得し利用します。

### 例9.6 動的暗号化の設定

動的暗号化を設定するには、サーバー側 (1) で暗号化のエイリアスを指定せず、署名が必要である (2) と宣言します。

```

<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://www.jboss.com/ws-
security/config
  http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd">
  <key-store-file>WEB-INF/bob-sign_enc.jks</key-store-file>
  <key-store-password>password</key-store-password>
  <key-store-type>jks</key-store-type>
  <trust-store-file>WEB-INF/wsse10.truststore</trust-store-file>
  <trust-store-password>password</trust-store-password>

  <config>
    <timestamp ttl="300"/>

```



```

(1)      <sign type="x509v3" alias="1" includeTimestamp="true"/>
        <encrypt type="x509v3"
              algorithm="aes-256"
              keyWrapAlgorithm="rsa_oaep"
              tokenReference="keyIdentifier" />
        <requires>
(2)      <signature/>
          <encryption/>
        </requires>
      </config>
</jboss-ws-security>

```

## アルゴリズム

<encrypt> 要素が宣言されていれば必ず、非対称暗号化および対称暗号化が行われます。生成された対称暗号鍵を使ってメッセージデータを暗号化します。この鍵は、受取側の公開鍵で暗号化 (ラップ) された後に SOAP ヘッダーに書き込まれます。暗号化アルゴリズムとキーラップアルゴリズム両方を設定することが可能です。

対応している暗号化アルゴリズムには以下が含まれます：

- AES 128 (aes-128) (デフォルト)
- AES 192 (aes-192)
- AES 256 (aes-256)
- Triple DES (triple-des)

対応のキーラップアルゴリズムには以下が含まれます：

- RSA v1.5 (rsa\_15) (デフォルト)
- RSA OAEP (rsa\_oaep)



## 注記

(aes-256などの) 強力なアルゴリズムを実行するには、[Unlimited Strength Java\(TM\) Cryptography Extension](#)のインストールが必要となる可能性があります。国によってはアプリケーションで利用できる暗号強度に制限が課せられている場合もありますので、責任を持って各管轄に適した暗号化レベルを選択するようにしてください。

## 暗号化トークンの参照

相互運用できるように、利用する暗号化トークンへの参照タイプを設定する必要があるかもしれません。例えば、ローカルのバイナリセキュリティトークンはJBossWS でデフォルトの参照タイプとして利用されていますが、Microsoft Indigo はこのトークンへの直接参照に対応していません。

この参照を設定するには、<encrypt> 要素の **tokenReference** 属性を指定します。 **tokenReference** 属性の値は以下の通りです：

- **directReference** (デフォルト)
- **keyIdentifier** : X509 SubjectKeyIdentifier参照を使いトークンデータを指定します。

- **x509IssuerSerial** : X509 とシリアル番号でエンドエンティティ (EE: End Entity) 証明書を一意に識別します。



## 注記

X509 Token Profile に関する包括的な情報は、[Oasis.org docs portal](http://Oasis.org/docs_portal)の『WSS X501 Certificate Token Profile 1.0』文書でご覧いただけます。

## ターゲットの設定

JBossWS では、署名あるいは暗号化を必要とする要素を正確に制御することができます。これにより、同じサービス上でやりとりがされているが、(e-メールアドレスなど)セキュリティレベルが高くないその他の情報ではなく、重要なデータのみ (クレジットカード番号など) を暗号化することができます。設定方法は、暗号化したい SOAP 要素の修飾名 (qname) を指定します。デフォルトの動作は、SOAP のボディ全体を暗号化します。

```
<encrypt type="x509v3" alias="alice">
  <targets>
    <target type="qname">{http://www.my-company.com/cc}CardNumber</target>
    <target type="qname">{http://www.my-company.com/cc}CardExpiration</target>
    <target type="qname" contentOnly="true">{http://www.my-company.com/cc}CustomerData</target>
  </targets>
</encrypt>
```

## ペイロードのキャリッジリターン (CR)

XML パーサーが特殊文字を解析する仕方が原因で、キャリッジリターン (\r) を含む署名済みのメッセージペイロードで署名照合エラーが発生する可能性があります。この問題を回避するには、ペイロードを送る前にカスタムエンコーディングを実装するようにします。そうするとユーザーはメッセージを暗号化するか、あるいはJBossWS によりメッセージのメッセージの正準正規化を強制的に行うことができます。

MessageContextで**true**に設定されている場合、org.jboss.ws.DOMContentCanonicalNormalization プロパティはペイロードを正規化できます。このプロパティは、クライアント側で呼び出す直前にエンドポイント実装に設定する必要があります。

## 9.21.2.7. JAAS の統合

WS-Security を実装すると、JAAS 統合により J2EE の宣言セキュリティを行うことができます。ユーザーのアイデンティティや認証情報は、サーバーのwsse 設定ファイルにあるパラメーターに従い、受信メッセージの wsse ヘッダーから取得します。特定のセキュリティドメイン用に設定された JAAS ログインモジュールを委譲することで認証と権限付与が行われます。

## Username Token

Username Token Profile により、呼出し元のユーザー名、パスワードを指定することができます。wsse サーバーの設定ファイルを使って、設定済みのログインモジュールにて認証、権限付与を行う際にこれらの情報を活用することができます。

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/ws-security/config
    http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
```

```

<config>
  <username/>
  <authenticate>
    <usernameAuth/>
  </authenticate>
</config>
</jboss-ws-security>

```



### 注記

JBossWS 3.0.2 Native 以前では、指定時は呼出し側の主体と認証情報を設定するために、Username Token が常に使われていました。つまり、認証タグが指定されずに User Token が利用されている場合、後方互換ができるようこの動作も保持されます。

## X.509 Certificate Token

JBossWS の以前のバージョンでは、指定時は呼出し側の主体と認証情報を設定するために、Username Token が常に使われていました。つまり、<authenticate> 要素が指定され User Token が使われている場合、後方互換ができるようこの動作が残されています。

```

<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/ws-security/config
    http://www.jboss.com/ws-security/schema/jboss-ws-
security_1_0.xsd">
  <key-store-file>META-INF/bob-sign.jks</key-store-file>
  <key-store-password>password</key-store-password>
  <key-store-type>jks</key-store-type>
  <trust-store-file>META-INF/wsse10.truststore</trust-store-file>
  <trust-store-password>password</trust-store-password>
  <config>
    <sign type="x509v3" alias="1" includeTimestamp="false"/>
    <requires>
      <signature/>
    </requires>
    <authenticate>
(1)    <signatureCertAuth
certificatePrincipal="org.jboss.security.auth.certs.SubjectCNMapping"/>
    </authenticate>
  </config>
</jboss-ws-security>

```

オプションの **certificatePrincipal** 要素 (1) はクラスを使い X509 証明の属性から主体をリトリブします。選択したクラスは、**CertificatePrincipal** を継承する必要があります。属性が指定されていない場合に使うデフォルトのクラスは、**org.jboss.security.auth.certs.SubjectDNMapping**となっています。

例9.7「BaseCertLoginModule セキュリティドメイン」に記載されているように、設定済みのセキュリティドメインにBaseCertLoginModuleを正しく設定する必要があります。

### 例9.7 BaseCertLoginModule セキュリティドメイン

以下は、**CertRolesLoginModule** を持つセキュリティドメインのコード例で、(指定の **jboss-roles.properties** ファイルを使うことで)認証も有効にします。

■

```
<application-policy name="JBossWSCert">
  <authentication>
    <login-module
      code="org.jboss.security.auth.spi.CertRolesLoginModule" flag="required">
      <module-option name="rolesProperties">jboss-
        roles.properties</module-option>
      <module-option name="unauthenticatedIdentity">anonymous</module-
        option>
      <module-option
        name="securityDomain">java:/jaas/JBossWSCert</module-option>
      </login-module>
    </authentication>
  </application-policy>
```

BaseCertLoginModule は、中央キーストアを使いユーザーを認証します。例

9.8「BaseCertLoginModule キーストア」で示されているように、

`org.jboss.security.plugins.JaasSecurityDomain` MBean を使ってこのストアを設定します。

#### 例9.8 BaseCertLoginModule キーストア

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
  name="jboss.security:service=SecurityDomain">
  <constructor>
    <arg type="java.lang.String" value="JBossWSCert"/>
  </constructor>
  <attribute name="KeyStoreURL">resource:META-
    INF/keystore.jks</attribute>
  <attribute name="KeyStorePass">password</attribute>
  <depends>jboss.security:service=JaasSecurityManager</depends>
</mbean>
```

認証時に、指定した **CertificatePrincipal** マッピングクラスは、紐付けられた wsse ヘッダーから取得した主体を使いキーストアにアクセスします。証明書があり、wsse ヘッダーで指定したものと同じであれば、ユーザー認証に成功します。

#### 9.21.2.8. POJO エンドポイント認証および権限付与 **JBOSSCC-50**

WS-Security が取得した証明書は通常、他のセキュリティがかかったリソースを呼び出す場合、EJB エンドポイントあるいは POJO エンドポイントに対して使います。POJO エンドポイントの確認に認証、権限付与ができるようになります。



#### 重要

EJB コンテナがデプロイされた bean のセキュリティ要件を処理するため、EJB ベースのエンドポイントに対し認証や権限付与を有効化すべきではありません。

#### 手順9.1 POJO 認証や権限付与を有効化

この手順では、POJO エンドポイントの認証や権限付与を有効にする際に必要となる追加設定について説明しています。

## 1. Web アーカイブにてセキュリティドメインを定義

POJO を含む WAR にてセキュリティドメインを定義する必要があります。

/WEB-INF フォルダの jboss-web 配備記述子で<security-domain> を指定します。

```
<jboss-web>
  <security-domain>java:/jaas/JBossWS</security-domain>
</jboss-web>
```

## 2. jboss-wsse-server.xml <authorize> 要素を設定します。

<config> 要素内の<authorize> 要素を指定します。

<config> 要素をグローバル、ポート固有、あるいはオペレーション固有のいずれかに定義することが可能です。

<authorize> 要素には、<unchecked/> 要素あるいは、1 つ以上の<role>要素を含める必要があります。各<role> 要素には、有効な RoleName 名を含む必要があります。

認証タイプは、Unchecked と役割ベース認証の2種類から選択し実装することができます。

### Unchecked 認証

この認証手順でユーザーのユーザー名やパスワードを照合しますが、役割の確認はそれ以上行われません。ユーザーのユーザー名やパスワードが無効の場合、リクエストは拒否されます。

#### 例9.9 Unchecked 認証

```
<jboss-ws-security>

  <config>
    <authorize>
      <unchecked/>
    </authorize>
  </config>

</jboss-ws-security>
```

### 役割ベースの認証

Unchecked 認証のようにユーザー名とパスワードを使いユーザー認証が行われます。ユーザーのユーザー名やパスワードが照合されると、ユーザーの認証情報が再確認され、<role> 要素で指定された役割のうち少なくとも1つがユーザーに割り当てられるようにします。



#### 注記

ユーザー名とパスワードあるいは証明書がリクエストメッセージで提示されていない場合でも、認証、権限付与が進められます。このシナリオでは、セキュリティドメインのログインモジュールが匿名のアイデンティティで設定されている場合、認証が進む場合があります。

#### 例9.10 役割ベースの認証

```
<jboss-ws-security>
```

```

    <config>
      <authorize>
        <role>friend</role>
        <role>family</role>
      </authorize>
    </config>
  </jboss-ws-security>

```

### 9.21.3. XML レジストリ

J2EE 5.0 は JAXR (Java API for XML Registries) のサポートを義務付けています。J2EE 5.0 認定の Application Server を持つ XML Registry を含むのは任意です。JBoss EAP には JBoss は Apache jUDDI レジストリである UDDI v2.0 規格のレジストリが同梱されています。また、Apache Scout の統合により JAXR Capability Level 0 (UDDI レジストリ) にも対応しています。

「[XML レジストリ](#)」は JBoss での jUDDI レジストリ設定方法を説明し、いくつかのサンプルコードで jUDDI レジストリの発行および問い合わせに JAXR API を使用する概略を示します。

#### 9.21.3.1. Apache jUDDI の設定

jUDDI レジストリの構成は **all** サーバプロファイルの **juddi-service.sar** アーカイブにデプロイされる MBean Service により行われます。このサービスの設定は **juddi-service.sar** 内の META-INF ディレクトリにある **jboss-service.xml** で行うことができます。

変更可能な設定アイテムを個別に見ていきます。

DataSources の設定

```

<!-- Datasource to Database -->
<attribute name="DataSourceUrl">java:/DefaultDS</attribute>

```

データベーステーブル (起動時に作成されるべきか、停止時にドロップされるべきか、起動時にドロップされるべきかなど)

```

<!-- Should all tables be created on Start-->
<attribute name="CreateOnStart">false</attribute>
<!-- Should all tables be dropped on Stop-->
<attribute name="DropOnStop">true</attribute>
<!-- Should all tables be dropped on Start-->
<attribute name="DropOnStart">false</attribute>

```

JNDI でバインドされるべき JAXR Connection ファクトリ (バインドされるべきか? どの名前の配下で?)

```

<!-- Should I bind a Context to which JaxrConnectionFactory bound-->
<attribute name="ShouldBindJaxr">true</attribute>

<!-- Context to which JaxrConnectionFactory to bind to. If you have remote
clients, please bind it to the global namespace(default behavior).
To just cater to clients running on the same VM as JBoss, change to
java:/JAXR -->
<attribute name="BindJaxr">JAXR</attribute>

```



## その他共通の設定

jUDDI レジストリへのアクセスを承認されたユーザーを追加します。(1 行に sql 挿入ステートメントを追加する)

```
Look at the script META-INF/ddl/juddi_data.ddl for more details. Example
for a user 'jboss'
```

```
INSERT INTO PUBLISHER (PUBLISHER_ID,PUBLISHER_NAME,
EMAIL_ADDRESS,IS_ENABLED,IS_ADMIN)
VALUES ('jboss','JBoss User','jboss@xxx','true','true');
```

### 9.21.3.2. JBoss JAXR の設定

本セクションでは、JAXR API の実行に必要とされる設定について説明していきます。JAXR 設定は JVM に渡される System プロパティに依存します。必要とされる System プロパティは次の通りです。

```
javax.xml.registry.ConnectionFactoryClass=org.apache.ws.scout.registry.
ConnectionFactoryImpl
jaxr.query.url=http://localhost:8080/juddi/inquiry
jaxr.publish.url=http://localhost:8080/juddi/publish
scout.proxy.transportClass=org.jboss.jaxr.scout.transport.SaaJTransport
```

ホスト名を「localhost」から UDDI サービス/JBoss Server のホスト名に変更するのを忘れないようにしてください。

System プロパティを次のようにして JVM に渡すことができます。

- クライアントコードが JBoss の内側で実行している場合は (おそらくサーブレットか EJB)、**run.sh** または **run.bat** のスクリプトにある System プロパティを **"-D"** オプションを使って java プロセスに渡す必要があります。
- クライアントコードが外部 JVM で実行している場合は、「-D」オプションとして java プロセスにプロパティを渡すか、明示的にクライアントコードに設定する (推奨しません) ことができます。

```
System.setProperty(propertyname, propertyvalue);
```

### 9.21.3.3. JAXR サンプルコード

API には JAXR Publish API と JAXR Inquiry API の 2 つのカテゴリがあります。すべての JAXR クライアントコードが使用する重要な JAXR インターフェースは次の通りです。

- J2EE 5.0 JavaDoc の [javax.xml.registry.RegistryService](#) より抜粋: 「これは JAXR プロバイダーにより実装される第一インターフェースになります。レジストリクライアントはこのインターフェースをレジストリへの Connection から取得することができます。JAXR プロバイダーにより実装される各種の機能固有インターフェースを見つけるためにクライアントによって使用されるメソッドを提供します。」
- J2EE 5.0 JavaDoc の [javax.xml.registry.BusinessLifeCycleManager](#) より抜粋: 「Registry Service により公開される BusinessLifeCycleManager インターフェースはビジネスレベル API の一部として Registry のライフサイクル管理機能を実装します。Connection インターフェー

スはクライアントの代わりにその状態とコンテキストを維持するため提供される認証情報はないので注意してください。」

- J2EE 5.0 JavaDoc の [javax.xml.registry.BusinessQueryManager](#) より抜粋: 「Registry Service により公開される BusinessQueryManager インターフェースはビジネススタイルのクエリーインターフェースを実装します。 集中クエリーインターフェースとも呼ばれます。」

JAXR API の使用中に行われる共通のプログラミングタスクをいくつか見てみます。

レジストリへの JAXR Connection を取得します。

```
String queryurl = System.getProperty("jaxr.query.url",
    "http://localhost:8080/juddi/inquiry");
String puburl = System.getProperty("jaxr.publish.url",
    "http://localhost:8080/juddi/publish");
..
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL", queryurl);
props.setProperty("javax.xml.registry.lifeCycleManagerURL", puburl);

String transportClass = System.getProperty("scout.proxy.transportClass",
    "org.jboss.jaxr.scout.transport.SaajTransport");
System.setProperty("scout.proxy.transportClass", transportClass);

// Create the connection, passing it the configuration properties
factory = ConnectionFactory.newInstance();
factory.setProperties(props);
connection = factory.createConnection();
```

レジストリと認証を行います。

```
/**
 * Does authentication with the uddi registry
 */
protected void login() throws JAXRException
{
    PasswordAuthentication passwdAuth = new PasswordAuthentication(userid,
passwd.toCharArray());
    Set creds = new HashSet();
    creds.add(passwdAuth);

    connection.setCredentials(creds);
}
```

Business を保存します。

```
/**
 * Creates a Jaxr Organization with 1 or more services
 */
protected Organization createOrganization(String orgname) throws
JAXRException
{
    Organization org = blm.createOrganization(getIString(orgname));
    org.setDescription(getIString("JBoss Inc"));
    Service service = blm.createService(getIString("JBoss JAXR Service"));
```



```

        service.setDescription(getIString("Services of XML Registry"));
        //Create serviceBinding
        ServiceBinding serviceBinding = blm.createServiceBinding();
        serviceBinding.setDescription(blm.createInternationalString("Test
Service Binding"));

        //Turn validation of URI off
        serviceBinding.setValidateURI(false);
        serviceBinding.setAccessURI("http://testjboss.org");
        ...
        // Add the serviceBinding to the service
        service.addServiceBinding(serviceBinding);

        User user = blm.createUser();
        org.setPrimaryContact(user);
        PersonName personName = blm.createPersonName("Anil S");
        TelephoneNumber telephoneNumber = blm.createTelephoneNumber();
        telephoneNumber.setNumber("111-111-7777");
        telephoneNumber.setType(null);
        PostalAddress address = blm.createPostalAddress("111", "My Drive",
"BuckHead", "GA", "USA", "1111-111", "");
        Collection postalAddresses = new ArrayList();
        postalAddresses.add(address);
        Collection emailAddresses = new ArrayList();
        EmailAddress emailAddress = blm.createEmailAddress("anil@apache.org");
        emailAddresses.add(emailAddress);

        Collection numbers = new ArrayList();
        numbers.add(telephoneNumber);
        user.setPersonName(personName);
        user.setPostalAddresses(postalAddresses);
        user.setEmailAddresses(emailAddresses);
        user.setTelephoneNumbers(numbers);

        ClassificationScheme cScheme = getClassificationScheme("ntis-
gov:naics", "");
        Key cKey = blm.createKey("uuid:C0B9FE13-324F-413D-5A5B-2004DB8E5CC2");
        cScheme.setKey(cKey);
        Classification classification = blm.createClassification(cScheme,
"Computer Systems Design and Related Services", "5415");
        org.addClassification(classification);
        ClassificationScheme cScheme1 = getClassificationScheme("D-U-N-S", "");
        Key cKey1 = blm.createKey("uuid:3367C81E-FF1F-4D5A-B202-3EB13AD02423");
        cScheme1.setKey(cKey1);
        ExternalIdentifier ei = blm.createExternalIdentifier(cScheme1, "D-U-N-S
number", "08-146-6849");
        org.addExternalIdentifier(ei);
        org.addService(service);

        return org;
    }

```

Business を問い合わせます。

```

/**
 * Locale aware Search a business in the registry

```

```
*/
public void searchBusiness(String bizname) throws JAXRException
{
    try
    {
        // Get registry service and business query manager
        this.getJAXREssentials();

        // Define find qualifiers and name patterns
        Collection findQualifiers = new ArrayList();
        findQualifiers.add(FindQualifier.SORT_BY_NAME_ASC);
        Collection namePatterns = new ArrayList();
        String pattern = "%" + bizname + "%";
        LocalizedString ls = blm.createLocalizedString(Locale.getDefault(),
pattern);
        namePatterns.add(ls);

        // Find based upon qualifier type and values
        BulkResponse response = bqmc.findOrganizations(findQualifiers,
namePatterns, null, null, null, null);

        // check how many organisation we have matched
        Collection orgs = response.getCollection();
        if (orgs == null)
        {
            log.debug(" -- Matched 0 orgs");
        }
        else
        {
            log.debug(" -- Matched " + orgs.size() + " organizations -- ");

            // then step through them
            for (Iterator orgIter = orgs.iterator(); orgIter.hasNext();)
            {
                Organization org = (Organization)orgIter.next();
                log.debug("Org name: " + getName(org));
                log.debug("Org description: " + getDescription(org));
                log.debug("Org key id: " + getKey(org));
                checkUser(org);
                checkServices(org);
            }
        }
    }
    finally
    {
        connection.close();
    }
}
```

これ以外の JAXR API を使用するコード例については、リソースセクションにあるリソースを参照してください。

#### 9.21.3.4. トラブルシューティング

- **JAXR からレジストリに接続できません。** inquiry を確認して JAXR ConnectionFactory に渡された url を公開してください。

- **jUDDI レジストリに接続できません。** jUDDI 設定を確認して server.log にエラーがないかチェックしてください。また、jUDDI レジストリは「all」構成でのみ使用可能になるので注意してください。
- **jUDDI レジストリに対して認証できません。** 章の前半の説明通りに、認証ユーザーを jUDDI データベースに追加しましたか？
- **クライアントと UDDI レジストリの間を移動中の SOAP メッセージを表示させたいのですが。** tcpmon ツールを使って移動中のメッセージを表示してください。 [TCPMon](#)

#### 9.21.3.5. リソース

- [JAXR Tutorial and Code Camps](#)
- [J2EE 1.4 Tutorial](#)
- [J2EE Web Services by Richard Monson-Haefel](#)

## 9.22. JBOSSWS の拡張

本セクションは JAX-WS に対する登録商標の JBoss 拡張について説明しています。

### 9.22.1. 登録商標のアノテーション

標準のアノテーションセットについては [JAX-WS Annotations](#) をご覧ください。

#### 9.22.1.1. EndpointConfig

```
/**
 * Defines an endpoint or client configuration.
 * This annotation is valid on an endpoint implementaion bean or a SEI.
 */
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = { ElementType.TYPE })
public @interface EndpointConfig
{
    ...
    /**
     * The optional config-name element gives the configuration name that
     must be present in
     * the configuration given by element config-file.
     *
     * Server side default: Standard Endpoint
     * Client side default: Standard Client
     */
    String configName() default "";
    ...
    /**
     * The optional config-file element is a URL or resource name for the
     configuration.
     *
     * Server side default: standard-jaxws-endpoint-config.xml
     * Client side default: standard-jaxws-client-config.xml
     */
}
```

```

    */
    String configFile() default "";
}

```

### 9.22.1.2. WebContext

```

/**
 * Provides web context specific meta data to EJB based web service
 endpoints.
 *
 * @author thomas.diesler@jboss.org
 * @since 26-Apr-2005
 */
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = { ElementType.TYPE })
public @interface WebContext
{
    ...
    /**
     * The contextRoot element specifies the context root that the web
     service endpoint is deployed to.
     * If it is not specified it will be derived from the deployment short
     name.
     *
     * Applies to server side port components only.
     */
    String contextRoot() default "";
    ...
    /**
     * The virtual hosts that the web service endpoint is deployed to.
     *
     * Applies to server side port components only.
     */
    String[] virtualHosts() default {};

    /**
     * Relative path that is appended to the contextRoot to form fully
     qualified
     * endpoint address for the web service endpoint.
     *
     * Applies to server side port components only.
     */
    String urlPattern() default "";

    /**
     * The authMethod is used to configure the authentication mechanism for
     the web service.
     * As a prerequisite to gaining access to any web service which are
     protected by an authorization
     * constraint, a user must have authenticated using the configured
     mechanism.
     *
     * Legal values for this element are "BASIC", or "CLIENT-CERT".
     */
    String authMethod() default "";
}

```

```

/**
 * The transportGuarantee specifies that the communication
 * between client and server should be NONE, INTEGRAL, or
 * CONFIDENTIAL. NONE means that the application does not require any
 * transport guarantees. A value of INTEGRAL means that the application
 * requires that the data sent between the client and server be sent in
 * such a way that it can't be changed in transit. CONFIDENTIAL means
 * that the application requires that the data be transmitted in a
 * fashion that prevents other entities from observing the contents of
 * the transmission. In most cases, the presence of the INTEGRAL or
 * CONFIDENTIAL flag will indicate that the use of SSL is required.
 */
String transportGuarantee() default "";

/**
 * A secure endpoint does not by default publish it's wsdl on an
unsecure transport.
 * You can override this behaviour by explicitly setting the
secureWSDLAccess flag to false.
 *
 * Protect access to WSDL. See http://jira.jboss.org/jira/browse/JBWS-723
 */
boolean secureWSDLAccess() default true;
}

```

### 9.22.1.3. SecurityDomain

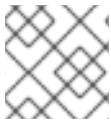
```

/**
 * Annotation for specifying the JBoss security domain for an EJB
 */
@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME)
public @interface SecurityDomain
{
    /**
     * The required name for the security domain.
     *
     * Do not use the JNDI name
     *
     * Good: "MyDomain"
     * Bad: "java:/jaas/MyDomain"
     */
    String value();

    /**
     * The name for the unauthenticated principal
     */
    String unauthenticatedPrincipal() default "";
}

```

## 9.23. WEB サービス付録



## 注記

この情報は JBoss Web Services CXF Stack で利用可能です。

[JAX-WS Endpoint Configuration](#)

[JAX-WS Client Configuration](#)

[JAX-WS Annotations](#)

## 9.24. 参照文献

1. [JSR-224 - Java API for XML-Based Web Services \(JAX-WS\) 2.0](#)
2. [JSR 222 - Java Architecture for XML Binding \(JAXB\) 2.0](#)
3. [JSR-250 - Common Annotations for the Java Platform](#)
4. [JSR 181 - Web Services Metadata for the Java Platform](#)

## 第10章 JBoss AOP

JBoss AOP は 100% Pure Java のアスペクト指向フレームワークで、すべてのプログラミング環境で使うことができ、アプリケーションサーバーと密に統合することが可能です。通常のオブジェクト指向プログラムでの対応が難しい場合でも、アスペクトはコードベースを簡単にモジュール化することができます。アプリケーションロジックやシステムコードからクリーンに分離することができ、統合ポイントをソフトウェアへ公開する素晴らしい方法を提供します。JDK 1.5 アノテーションを組み合わせると、コード生成にアノテーションのみを使用せずに、クリーンでプラグ可能な方法で Java 言語を拡張できます。

JBoss AOP はフレームワークであるだけでなく、アノテーションやポイントカット表現経由で適用され、ランタイム時に動的に適用されるパッケージ済みのアスペクトセットでもあります。これには、キャッシング、非同期通信、トランザクション、セキュリティ、リモーティングなど、多数含まれます。

アスペクトは、メソッド、クラス、オブジェクト階層、オブジェクトモデル全体に散在する共通の機能です。構造があるような動作をしますが、従来のオブジェクト指向技術では構造をコードで表現することはできません。

例えば、メトリックスは共通のアスペクトの 1 つです。アプリケーションより有用なログを生成するには、コード全体に通知メッセージを散在させなければなりません。しかし、メトリックスはクラスやオブジェクトモデルが考慮されるものではありません。顧客やアカウントを表すわけではなく、ビジネスルールを実現しません。メトリックスは直交で、実際のアプリケーションとは無関係です。

### 10.1. 主な用語

#### ジョインポイント

ジョインポイントとは Java プログラム内のポイントのことです。メソッドの呼び出し、コンストラクタの実行、フィールドのアクセスはすべてジョインポイントとなります。ジョインポイントは、イベントがメソッド呼び出し、コンストラクタコール、フィールドアクセスなどである特定の Java イベントと考えることができます。

#### 呼び出し

呼び出しは、ランタイム時のジョインポイントをカプセル化する JBoss AOP クラスです。呼び出されたメソッドやメソッドの引数などの情報が含まれます。

#### アドバイス

アドバイスは、特定のジョインポイントが実行された時に呼び出されるメソッドのことです (メソッドが呼び出された時にトリガーされる動作など)。インターセプションを行うコードと考えることもできます。アドバイスは「イベントハンドラー」とも類似しています。

#### ポイントカット

ポイントカットは AOP の表現言語です。正規表現が文字列と一致するように、ポイントカット表現もジョインポイントと一致します。

#### イントロダクション

イントロダクションは Java クラスのタイプや構造を変更します。既存のクラスにインターフェースを実装するよう強制、あるいはアノテーションを追加するために使用できます。

#### アスペクト

アスペクトは、アドバイス、ポイントカット定義、ミックスイン、その他 JBoss AOP コンストラクトの数をカプセル化するプレーン Java クラスです。

## インターセプター

インターセプターは **invoke** という名前のアドバイスのみを持つアスペクトです。インターフェースを実装するようクラスを強制してコードをチェックしたい場合に実装できるインターフェースです。インターセプターは移植可能で、EJB や JMX MBean など他の JBoss 環境で再使用することができます。

AOP では、メトリックスなどの機能は**横断的関心事 (crosscutting concern)** と呼ばれます。これは、オブジェクトモデルの複数のポイントを「横断」しながらも明確に異なる動作であるためです。開発方法では、横断的関心事を抽象化し、カプセル化することが推奨されます。

アプリケーションにコードを追加し、特定メソッドの呼び出しに掛かる時間を測定するとしましょう。プレーン Java では、以下のようなコードになるはずです。

```
public class BankAccountDAO
{
    public void withdraw(double amount)
    {
        long startTime = System.currentTimeMillis();
        try
        {
            // Actual method body...
        }
        finally
        {
            long endTime = System.currentTimeMillis() - startTime;
            System.out.println("withdraw took: " + endTime);
        }
    }
}
```

このコードでも動作しますが、いくつかの問題があります。

1. **try/finally** ブロックのコードを手作業でベンチマーク対象のメソッドやコンストラクターすべてに追加する必要があるため、メトリックスの有効化や無効化が大変難しくなります。
2. プロファイリングコードをアプリケーション全体に散在させるべきではありません。**try/finally** ブロック内にタイミングが含まれるようにしなければならないため、コードが膨張して読みにくくなります。
3. この機能を拡張してメソッドや失敗数が含まれるようにし、これらの統計を高性能なレポートメカニズムに登録したい場合、多くのファイルを変更する必要があります。

コードベース全体にメトリックスが分散されるため、メトリックスに対するこのアプローチは維持や拡張が大変難しくなります。多くの場合、メトリックスをクラスに追加するのに OOP が最良の方法であるとは言えません。

アスペクト指向プログラミングは、このような動作機能をカプセル化する方法を提供します。これにより、メトリックスのような動作をコードの「周り」に追加することができます。例えば、コードの実際のボディーを実行する前に **BankAccountDAO** への呼び出しがメトリックスアスペクトを通過するように指定するプログラム制御を AOP は提供します。

## 10.2. JBOSS AOP でアスペクトを作成

簡単に説明すると、すべての AOP フレームワークは、横断的関心事を実装する方法とプログラムコンストラクト (プログラミング言語またはタグのセット) の 2 つを定義し、コードスニペットの適用方法



を指定します。JBoss AOP、横断的関心事、ユーザーがどのように JBoss Enterprise Application Platform でメトリクスアスペクトを実装できるか見てみましょう。

独自の Java クラスにメトリクス機能をカプセル化することが JBoss AOP にメトリクスアスペクトを作成する最初のステップになります。以下のコードは、最初のコード例の

**BankAccountDAO.withdraw()** メソッドにある **try/finally** ブロックを JBoss AOP インターセプタークラスの実装である **Metrics** に抽出します。

次のリストは JBoss AOP インターセプターへのメトリクス実装を表しています。

```
01. public class Metrics implements org.jboss.aop.advice.Interceptor
02. {
03.     public Object invoke(Invocation invocation) throws Throwable
04.     {
05.         long startTime = System.currentTimeMillis();
06.         try
07.         {
08.             return invocation.invokeNext();
09.         }
10.         finally
11.         {
12.             long endTime = System.currentTimeMillis() - startTime;
13.             java.lang.reflect.Method m =
14. ((MethodInvocation)invocation).method;
15.             System.out.println("method " + m.toString() + " time: " +
16. endTime + "ms");
17.         }
18.     }
19. }
```

JBoss AOP では、Metrics クラスが **withdraw()** をラップします。呼び出しコードが **withdraw()** を呼び出すと、AOP フレームワークがメソッドコールをパーツに分解し、パーツを呼び出しオブジェクトにカプセル化します。その後、呼び出しコードと実際のメソッドボディーの間に存在するすべてのアスペクトをフレームワークが呼び出します。

AOP フレームワークによるメソッドコールの分解が終了すると、3 行目で **Metrics** の呼び出しメソッドを呼び出します。8 行目は実際のメソッドをラップし、実際のメソッドへ委譲します。**try/finally** ブロックを使用してタイミングを実行します。13 行目は **Invocation** オブジェクトからメソッドコールのコンテキスト情報を取得します。14 行目はメソッド名と算出されたメトリクスを表示します。

独自のオブジェクト内に **Metrics** コードが存在することにより、後で追加の測定を簡単に拡張し取得することができます。アスペクトにカプセル化されたメトリクスの適用方法を見てみましょう。

### 10.3. JBOSS AOP でアスペクトを適応

アスペクトを適用するには、アスペクトコードの実行時を定義します。実行のポイントは **ポイントカット** と呼ばれます。ポイントカットは正規表現と似ています。正規表現は文字列とマッチさせますが、ポイントカット表現はアプリケーション内のイベントや **ポイント** とマッチさせます。例えば、有効なポイントカット定義は「JDBC メソッド **executeQuery()** への呼び出しに対し、SQL 構文を検証するアスペクトを呼び出す」のようになります。

フィールドアクセスやメソッドコール、コンストラクタコールなどがエントリポイントとなります。スローされた例外などがイベントとなります。AOP 実装によっては、ポイントカットの指定にクエリに似た言語を使用するものもあります。その他の AOP 実装はタグを使用します。JBoss AOP は両方を

使用します。

次のリストは JBoss AOP における Metrics 例のポイントカットの定義方法を示しています。

```

1 <interceptor name="SimpleInterceptor" class="com.mc.Metrics"/>
2 <bind pointcut="execution (public void com.mc.BankAccountDAO-
>withdraw(double amount))" >
3     <interceptor-ref name="SimpleInterceptor" />
4 </bind>
5 <bind pointcut="execution (* com.mc.billing.->(..))">
6     <interceptor-ref name="com.mc.Metrics" />
7 </bind>

```

1 行目は **interceptor** クラスへのインターセプター名のマッピングを定義しています。2-4 行目は、特定の **BankAccountDAO.withdraw()** メソッドに **metrics** アスペクトを適用するポイントカットを定義します。5-7 行目は **com.mc.billing** パッケージにおける全クラス的全メソッドへ **metrics** アスペクトを適用する一般的なポイントカットを定義します。XML を希望しない場合は、オプションのアノテーションマッピングもあります。詳細は JBoss AOP の参照文書をご覧ください。

JBoss AOP には Java アプリケーションの様々なポイントやイベントを定義するために使用できるポイントカット表現のセットが多くあります。ポイントを実装した後、アスペクトを適用することができます。アプリケーション内の特定の Java クラスにアスペクトを適用することができますが、複雑な複合的なポイントカットを使用し、1 つの表現内に複数のクラスを指定することもできます。

この例の通り、AOP では横断的動作を 1 つのオブジェクトにまとめ、ビジネスロジックに関係ない機能でコードを複雑化することなく簡単に適応することができます。共通の横断的関心事は 1 つの場所で維持拡張できます。

**BankAccountDAO** クラス内のコードはプロファイルされた事を認識しません。プロファイリングはアスペクト指向プログラマーが直交した関心事 (orthogonal concern) として考えるものの一部です。本章の始めにあるオブジェクト指向プログラミングのコードスニペットでは、プロファイリングがアプリケーションコードの一部となっています。AOP ではこのコードを削除することが可能です。近年のミドルウェアは透明性を約束していますが、AOP も確実に透明性を提供します。

直交した動作は開発後に追加することもできます。オブジェクト指向コードでは、監視とプロファイリングは開発時に追加されなければなりません。AOP では、開発者や管理者がコードに触れることなく必要に応じて簡単に監視やメトリクスを追加することができます。これは目立った機能ではありませんが、この分離により、横断するコードの上下にアスペクトを階層化することができるため、AOP の重要機能の一部になります。階層化により、必要な時に機能を追加削除できます。例えば、ベンチマークを実行する時のみメトリクスを有効にし、実稼働時に削除するとします。AOP では、コードを編集、再コンパイル、再パッケージすることなく、これを実行することができます。

## 10.4. AOP アプリケーションのパッケージ

AOP アプリケーションを JBoss Enterprise Application Platform にデプロイするには、AOP アプリケーションをパッケージ化する必要があります。AOP は SAR (MBean) と同様にパッケージ化されます。XML ファイルを署名 **\*-aop.xml** とパッケージと共に直接 **deploy/** ディレクトリにデプロイするか (これにより **jboss-aop.deployer** ファイルにある **base-aop.xml** が利用できます)、クラスを格納する JAR ファイルに XML ファイルが含まれるようにすることができます。JAR に XML ファイルが含まれるようにする場合、ファイルの拡張子は **.aop** でなければなりません。また、**jboss-aop.xml** ファイルは **META-INF** ディレクトリに含まれなければなりません (例: **META-INF/jboss-aop.xml**)。

JBoss Enterprise Application Platform 5 では、使用するスキーマを指定しないと情報が正しく構文解析されないため、必ず指定してください。スキーマを指定するには、次のように **xmlns="urn:jboss:aop-beans:1:0"** 属性をルート **aop** 要素に追加します。

```
<aop xmlns="urn:jboss:aop-beans:1.0">
</aop>
```

.aop JAR ファイルを使った既定の例を超えるものを作成するには、XML バインディング設定を格納する AOP ファイルが含まれるトップレベルのデプロイメントを作成します。例えば、EAR ファイルまたは WAR ファイルに AOP ファイルを保持することができます。AOP ファイル内にある **META-INF/jboss-aop.xml** ファイルに指定されたバインディングは、WAR ファイル全体のすべてのクラスに影響します。

EAR ファイル内の AOP ファイルを取得するには、次の例のように Java モジュールとして **.ear/META-INF/application.xml** にリストされなければなりません。

```
<?xml version='1.0' encoding='UTF-8'?>
<!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>AOP in JBoss example</display-name>
  <module>
    <java>example.aop</java>
  </module>
  <module>
    <ejb>aopexampleejb.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>aopexample.war</web-uri>
      <context-root>/aopexample</context-root>
    </web>
  </module>
</application>
```

### 重要

JBoss Enterprise Application Platform 5 では、**.ear** ファイルの内容は **application.xml** にある一覧の順番通りにデプロイされます。ロード時の組み入れを使用する場合は、**ejb** クラスや **servlet** などがロードされる前にバインディングがシステムに存在するよう、アドバイスされたクラスがデプロイされる前に **example.aop** ファイルにリストされているバインディングがデプロイされなければなりません。これには、**application.xml** の最初に AOP ファイルを記載します。**.sar** ファイルや **.war** ファイルなど、他のタイプのアーカイブは最初にデプロイされるため、特別な処置は必要ありません。

## 10.5. JBOSS ASPECTMANAGER サービス

AspectManager サービスは、**http://localhost:8080/jmx-console** の JMX コンソールを使用してランタイム時に管理することができます。AspectManager サービスは ObjectName **jboss.aop:service=AspectManager** で登録されています。起動時に設定を行いたい場合は、設定ファイルを編集する必要があります。

JBoss Enterprise Application Platform 5 では、**AspectManager** サービスは JBoss Microcontainer bean を使用して設定されます。設定ファイルは **jboss-as/server/PROFILE/conf/bootstrap/aop.xml** です。**AspectManager** サービスは次の XML に

てデプロイされます。

```
<bean name="AspectManager"
class="org.jboss.aop.deployers.AspectManagerJDK5">

  <property name="jbossIntegration"><inject bean="AOPJBossIntegration"/>
</property>

  <property name="enableLoadtimeWeaving">false</property>
  <!-- only relevant when EnableLoadtimeWeaving is true.
  When transformer is on, every loaded class gets transformed.
  If AOP can't find the class, then it throws an exception.
  Sometimes, classes may not have all the classes they reference.
  So, the Suppressing is needed. (For instance, JBoss cache in the
  default configuration) -->

  <property name="suppressTransformationErrors">true</property>

  <property name="prune">true</property>

  <property name="include">org.jboss.test., org.jboss.injbossaop.
</property>

  <property name="exclude">org.jboss.</property>
  <!-- This avoids instrumentation of hibernate cglib enhanced proxies

  <property name="ignore">*$EnhancerByCGLIB$*</property> -->

  <property name="optimized">true</property>

  <property name="verbose">false</property>
  <!-- Available choices for this attribute are:
org.jboss.aop.instrument.ClassicInstrumentor (default)
org.jboss.aop.instrument.GeneratedAdvisorInstrumentor -->

  <!-- <property
name="instrumentor">org.jboss.aop.instrument.ClassicInstrumentor</property
-->

  <!-- By default the deployment of the aspects contained in
  ../deployers/jboss-aop-jboss5.deployer/base-aspects.xml
  are not deployed. To turn on deployment uncomment this property
  <property name="useBaseXml">true</property>-->
</bean>
```

**AspectManager** サービスのクラス変更については後ほど説明します。クラスを変更するには、**bean** 要素の **class** 属性の内容を置き換えます。

## 10.6. SUN JDK を使用した JBOSS ENTERPRISE APPLICATION PLATFORM におけるロード時の変換

JBoss Enterprise Application Platform は JDK と特別に統合し、ロード時の変換を実行します。本項では、この使用法を説明します。

JBoss Enterprise Application Platform 5 と Sun JDK でロード時の変換を実行したい場合は、次の手順に従ってください。

- **enableLoadtimeWeaving** 属性/プロパティを **true** に設定します。JBoss Application Server はデフォルトでは AOP ファイルのロード時のバイトコード操作を実行しないため、この設定を行う必要があります。 **suppressTransformationErrors** が **true** に設定されている場合、バイトコード変換に失敗するとエラー警告のみが生成されます。JBoss デプロイメントにクラスが参照する全てのクラスが存在しないことがあるため、このフラグが必要となります。
- **pluggable-instrumentor.jar** を JBoss AOP ディストリビューションの **lib/** ディレクトリから JBoss Enterprise Application Platform の **bin/** ディレクトリへコピーします。
- 次に **run.sh** または **run.bat** (OS によって異なります) を編集し、下記を **JAVA\_OPTS** 環境変数に追加します。

```
set JAVA_OPTS=%JAVA_OPTS% -Dprogram.name=%PROGNAME% -
javaagent:pluggable-instrumentor.jar
```



### 重要

AspectManager サービスのクラスは、**-javaagent** オプションが機能する **org.jboss.aop.deployers.AspectManagerJDK5** または **org.jboss.aop.deployment.AspectManagerServiceJDK5** でなければなりません。

## 10.7. JROCKIT

JRockit は、「[Sun JDK を使用した JBoss Enterprise Application Platform におけるロード時の変換](#)」に記載されている **-javaagent** スイッチもサポートします。このスイッチを使用する場合、「[Sun JDK を使用した JBoss Enterprise Application Platform におけるロード時の変換](#)」の手順に従ってください。JRockit はクラスがロードされる際のインターセプトに対する独自のフレームワークもっており、**-javaagent** スイッチよりも高速である場合があります。特別な JRockit フックを使用してロード時の変換を実行する場合は、次の手順に従ってください。

- **enableLoadtimeWeaving** 属性/プロパティを **true** に設定します。JBoss アプリケーションサーバーはデフォルトでは AOP ファイルのロード時のバイトコード操作を実行しないため、この設定を行う必要があります。 **suppressTransformationErrors** に設定されている場合、バイトコード変換に失敗するとエラー警告のみが生成されます。JBoss デプロイメントにクラスが参照する全てのクラスが存在しないことがあるため、このフラグが必要となります。
- **jrockit-pluggable-instrumentor.jar** を JBoss AOP ディストリビューションの **lib/** ディレクトリから JBoss Enterprise Application Platform の **bin/** ディレクトリへコピーします。
- 次に **run.sh** または **run.bat** (OS によって異なります) を編集し、下記を **JAVA\_OPTS** 環境変数と **JBOSS\_CLASSPATH** 環境変数に追加します。

```
# Setup JBoss specific properties

JAVA_OPTS="$JAVA_OPTS -Dprogram.name=$PROGNAME \
-
Xmanagement:class=org.jboss.aop.hook.JRockitPluggableClassPreProcessor"
```

```
JBoss_CLASSPATH="$JBoss_CLASSPATH:jrockit-pluggable-
instrumentor.jar"
```

- JRockit の特別なフックに対応するため、**AspectManager** サービスのクラスを JBoss Enterprise Application Platform 5 上で **org.jboss.aop.deployers.AspectManagerJRockit** に設定するか、**org.jboss.aop.deployment.AspectManagerService** に設定します。

## 10.8. JBOSS ENTERPRISE APPLICATION PLATFORM 環境でロード時のパフォーマンスを改善

JBoss Enterprise Application Platform におけるロード時のウィーピングのパフォーマンスを調整するルールは、スタンドアローン Java の場合と同じです。pruning、optimized、include、exclude などのスイッチは、本章の始めで説明した **jboss-5.x.x.GA/server/xxx/conf/aop.xml** ファイルより設定できます。

## 10.9. AOP をクラスローダーヘスコープ

JBoss のすべてのデプロイメントは、デフォルトでアプリケーションサーバー全体に対してグローバルになっています。そのため、deploy ディレクトリにある EAR、SAR、JAR はデプロイされた別のアーカイブからクラスを可視できます。同様に、AOP のバインディングは仮想マシン全体に対してグローバルになっています。**global**の可視性はトップレベルデプロイメント毎に無効にすることができます。

### 10.9.1. スコープされたクラスローダーの一部としてデプロイ

次のプロセスは、今後の JBoss AOP バージョンで変更する可能性があります。スコープされたアーカイブの一部として AOP ファイルをデプロイする場合、**.aop/META-INF/jboss-aop.xml** ファイル内に適用されるバインディングなどは、スコープされたアーカイブ内でクラスのみ適用され、アプリケーションサーバーのそれ以外のものには適用されません。**-aop.xml** ファイルをサービスアーカイブ(SAR)の一部としてデプロイする方法もあります。この場合でも、SAR がスコープされていると、**-aop.xml** ファイルに格納されたバインディングは SAR ファイルの内容のみに適用されます。現在、スタンドアローンの **-aop.xml** ファイルをデプロイあるいは、すでに添付されたデプロイメントに添付することはできません。スタンドアロン **-aop.xml** ファイルはアプリケーションサーバー全体のクラスに適用されます。

### 10.9.2. スコープされたデプロイメントへ添付

クラスローダー分離を使用するアプリケーションがある場合、クラスを準備していれば後に AOP ファイルをデプロイメントに添付できます。**jboss-app.xml** ファイルを使用しスコープされた EAR ファイルとスコープされたローダーレポジトリ **jboss.test:service=scoped** があるとします。

```
<jboss-app>
  <loader-repository>
    jboss.test:service=scoped
  </loader-repository>
</jboss-app>
```

この場合、アスペクトや設定が含まれる AOP ファイルを後でデプロイし、デプロイメントをスコープされた EAR に添付することができます。これを実行するには、AOP ファイルの **META-INF/jboss-aop.xml** にある **loader-repository** を使用します。

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
  <loader-repository>jboss.test:service=scoped</loader-repository>

  <!-- Aspects and bindings -->
</aop>
```

上記は、前に説明した AOP ファイルを EAR の一部としてデプロイする方法と同じ効果がありますが、アスペクトをスコープされたアプリケーションにホットデプロイすることが可能です。

## 第11章 トランザクション管理

本章では、JBoss Transaction Service の主要設定オプションに関し簡単に説明します。詳細情報は、『JBoss Transactions Administration Guide』を参照してください。

### 11.1. 概要

JBoss Enterprise Application Platform のトランザクションサポートは、JBossTransaction Serviceによって提供されます。JBossTransaction Service は、十分に考慮・開発されたモジュラー、規格ベースかつ、設定自在のトランザクションマネージャーです。デフォルトでは、サーバーはインストールされたJBossTransaction Service のローカル専用JTAモジュールにて稼働します。このモジュールは、EJBコンテナなどの他の内部コンポーネントによる使用、さらにはアプリケーションによる直接使用に向けた標準的なJTA API の実装を提供します。関係データベースやメッセージキューなど1つ以上のXA リソースマネージャーが関係するACID トランザクションの調整に適しています。

JBoss Transaction Service トランザクションモジュールについても、JBoss Enterprise Application Platform に2つオプションが追加されています。これらのモジュールをデプロイすると、必要であれば次のような追加機能を提供することができます。

#### JBoss Transaction Service JTS

トランザクションマネージャーは、リモート IIOP メソッドの呼び出し上でトランザクションコンテキストを分散でき、複数の JVM に対応する単一の分散トランザクションを作成することができます。複数のサーバーにわたる大型のアプリケーションや、CORBA ベースのシステムで実行されているトランザクションビジネスロジックとの規格ベースの相互運用に対して有用です。このモジュールの機能は標準的なJTA API から利用できるため、トランザクションビジネスロジックへの変更を必要とせず、簡単に代替の導入ができます。この機能の有効化に関する詳細情報は「[JTS Module の利用](#)」を参照してください。

#### JBoss Transaction Service XTS

*WS-AtomicTransaction (WS-AT)* や *WS-BusinessActivity (WS-BA)* 仕様を実装する XML ベースの Transaction Manager です。この追加モジュールは JTA または JTS が提供するコアトランザクションサポートや JBossWS Native が提供する Web サービス機能は使用します。JBossTransaction Service XTS はアプリケーションとしてサーバー内にデプロイされます。アプリケーションは JTS とほぼ同じ要領で WS-AT を使用して標準的な分散 ACID トランザクションを提供することもあります。CORBA ではなく Web サービストランスポートを使用します。WS-BA 実装は、さらに代替の補償ベーストランザクションモデルを提供します。このトランザクションモデルは、長期実行される疎結合のビジネスプロセスの調整に適しています。XTS は、通常はローカル WS-AT 実装や WS-BA 実装が内部でのみアクセスする *WS-Coordination (WS-C)* サービスも実装します。しかし、この WS-C サービスは、別の JBoss サーバーインスタンスや JBoss 以外のコンテナに作成された WS-AT や WS-BA トランザクションのリモート調整を提供するためにも使用されます。XTS を有効にするには、「[XTS モジュールの利用](#)」を参照してください。

### 11.2. 必須設定

JBossTS JTA のデフォルト設定は、トランザクションマネージャー独自のプロパティファイルやアプリケーションサーバーのデプロイメント設定を組み合わせで管理されます。設定ファイルは **`$JBOSS_HOME/server/PROFILE/conf/jbossts-properties.xml`** にあります。設定ファイルには、最も一般的に使用されるプロパティのデフォルト設定が格納されています。その他の設定について多数、付属の JBossTransaction Service 管理ガイドに説明されています。各設定にはハードコードされたデフォルトがありますが、設定ファイルがないとシステムが正しく機能しない場合があります。追加設定は **`$JBOSS_HOME/server/PROFILE/deploy/transaction-jboss-beans.xml`** にある Microcontainer bean 設定の一部として行うこともできます。これにより、トランザクションマネージャーがサーバープロファイル全体の設定に結合されるため、随時トランザクション設定ファイルの設



定よりもアプリケーションサーバー固有の値が優先されます。特に、Service Binding Manager を使用してポートのバインディング情報を設定し、選択された他のプロパティより優先させます。設定プロパティはサーバーの初期化時に JBossTransaction Service によって読み取られ、その後設定ファイルに加えられた変更は、サーバーが再起動するまで有効になりません。

表11.1 JBoss Transaction Service で最も重要なプロパティ

プロパティ名	デフォルト値	詳細
transactionTimeout	300 秒	<p>秒単位のデフォルト時間で、この時間が経過した後トランザクションはタイムアウトし、ロールバックされます。ご利用中の環境やワークロードにあわせて、これを調節してください。</p> <p>トランザクションが非同期的に処理される点に驚かれるかもしれませんが、設計的にこのように決定され、コードによりユーザ側で対応する必要があります。</p>
objectStoreDir		<p>トランザクションデータがログ化されるディレクトリ。システム障害が発生した場合にトランザクションを完了するにはトランザクションログが必要となり、このログは信頼性のあるストレージに設置される必要があります。通常、トランザクション毎に1ファイル生成され、各ファイルはサイズ的には数キロバイトとなっています。パフォーマンスが最適化されるよう、これらのファイルはディレクトリツリー全体で分散されています。RAIDコントローラを使っている場合、データベースの格納デバイスとほぼ同じ方法でライトスルーキャッシュ用に設定する必要があります。トランザクションがロールバックされる場合あるいは、リソースを1つしか含まない場合、トランザクションログの書き込みは自動的にスキップされます。</p>
max-pool-size		<p>Java EE Connector Architecture コンテナは、リカバリが実行されるEISに対し、専用の物理接続を開いた状態に保ちます。そのため、<b>max-pool-size</b> を（接続可能な最大数－1）を設定してください。</p>

表11.2 JBoss Transaction Service の追加プロパティ

プロパティ名	デフォルト値	詳細
com.arjuna.common.util.logging.DebugLevel	<b>0x00000000</b> はロギングなしと同等。	<p>トランザクションマネージャーのコードベースに対する内部ログの閾値を決定します。サーバー全体の <b>log4j</b> ロギング設定から独立しており、外部からのログ入力が表示されないように抑える役割を果たします。デフォルト値が有効であれば、INFO や WARN メッセージは表示され、この設定により最適なパフォーマンスが実現されます。<b>0xffffffff</b> により完全なデバッグロギングが有効になり、この設定をするとログファイルのサイズが大きくなります。</p> <p>内部の <b>DebugLevel</b> チェックを渡すログメッセージがサーバーのロギングシステムに渡されさらに処理されます。理論的には、完全なデバッグ機能はオンの状態にでき、<b>log4j</b> を使ってロギングの切/入が可能です。実際にはこれはパフォーマンスに影響を与えます。</p>
com.arjuna.ats.arjuna.coordinator.commitOnePhase	<b>YES</b>	トランザクションに単一のリソースのみが登録されている場合に、トランザクションマネージャーは自動的に 1 フェーズコミットの最適化をトランザクション完了プロトコルへ適用しするかを決定します。デフォルトでは、トランザクションログの書き込みを回避するため、この設定は有効になっています。
com.arjuna.ats.arjuna.objectstore.transactionSync	<b>ON</b>	<p>この設定は、トランザクションの終了中にディスクへのトランザクションログのフラッシングを制御します。デフォルト値では、各コミットトランザクションに対して <b>FileDescriptor.sync</b> 呼び出しが実行されます。リカバリ保証や ACID プロパティの提供に、この動作は必要となります。これらの機能がアプリケーションにとって重要でない場合、このプロパティを向こうにすることでパフォーマンスが向上しますが、通常、トランザクションを全く利用しないようにアプリケーションを記述するほうが良いため、この方法は推奨されません。</p>

プロパティ名	デフォルト値	詳細
com.arjuna.ats.arjuna.xa.nodeldentifier  com.arjuna.ats.jta.xaRecoveryNode		これらのプロパティはトランザクションリカバリシステムの動作を決定します。サーバーのクラッシュが発生した時にリカバリが行われトランザクションが正しく解決されるようにするため、適切な設定が必要となります。詳細は、JBoss Transaction 管理ガイドのリカバリの章を参照してください。
com.arjuna.ats.arjuna.coordinator.enableStatistics	NO	トランザクション統計の収集を有効にします。この統計は、 <b>FileDescriptor.sync</b> あるいは該当の JMX MBean でメソッドを使うことで参照できます。デフォルトでは無効になっています。

### 11.3. トランザクションリソース

Transaction Service は、様々なリソースマネージャーによって提供される **XAResource** 実装を利用しトランザクションステートの更新を調整します。リソースマネージャーは、データベース、メッセージキュー、サードパーティ JCA リソースアダプターなどを含むことがあります。JBoss Enterprise Application Platform での使用が認定された JDBC データベースドライバーやデータベースの一覧は <http://www.jboss.com/products/platforms/application/supportedconfigurations/> を参照してください。規格準拠の JDBC ドライバーの多くは正しく機能するはずですが、ベンダーによって XA 仕様の解釈が異なるため、認定を受けていないドライバーを使う場合は徹底的なテストを行う必要があります。

データベース接続プールは、アプリケーションサーバーの Datasource ファイルで設定します。これらのファイルは、**-ds.xml** などのファイル名にします。<xa-datasource> プロパティを使うデータソースは自動的にトランザクションマネージャーとやりとりを行います。JNDI でこのようなデータソースを検索し、**getConnection** を呼び出すことで取得した接続は自動的に進行中のトランザクションに参加します。これは、データアクセスに対してトランザクション保証が必要な場合、推奨されるユースケースとなっています。

XA トランザクションに対応できないデータベースを利用している場合、<local-xa-datasource> を使い接続プールをデプロイできます。この種類のデータソースは、「**最終リソースコミット最適化 (LRCO: Last Resource Commit Optimization)**」を使い管理トランザクションに参加し、より制限のあるトランザクション保証を提供します。<no-tx-datasource> から取得した接続は、トランザクションマネージャーとやり取りを行わないため、このような接続で実施される作業については、JDBC API を使いアプリケーション側から明示的にコミットあるいはロールバックを行う必要があります。

多くのデータベースでは、XA リソースマネージャーとして機能できるようになるまでに追加設定が必要となります。データベース設定に関するベンダー固有の情報は、[付録A ベンダー固有のデータソース定義](#)にあります。追加設定の説明については、ご利用中のデータベースに同梱されているデータベースアドミニストレーターや文書を参照してください。さらに、XA リカバリの適切な設定方法に関する情報は、JBoss Transactions 管理ガイドを参照してください。

JBoss Messaging は XA 対応ドライバーを提供しており、XA トランザクションに参加できます。詳細については、JBoss Messaging ユーザガイドを参照してください。

## 11.4. 最終リソースコミット最適化 (LRCO: LAST RESOURCE COMMIT OPTIMIZATION)

XA トランザクションプロトコルは、2 フェーズのコミットプロトコルを用いることで ACID プロパティを提供するように設計されていますが、モデルが常に正しいというわけではありません。時には、XA 非対応のリソースマネージャーがトランザクションに参加できるようにする必要がある場合もあります。これは通常、分散トランザクションに対応していないデータストアを使う場合に該当します。

このような場合、*Last Resource Commit Optimization (LRCO)*と呼ばれる技術を使うことができます。これは、*Last Resource Gambit*と呼ばれることもあります。この技術を使用すると、1 フェーズ対応リソースがトランザクションの **prepare** フェーズの最後に処理され、その時にコミットが試行されます。コミットに成功すると、トランザクションログが書き込まれ、残りのリソースはフェーズ 2 コミットに進みます。最後のリソースのコミットに失敗すると、トランザクションはロールバックされます。このプロトコルにより、ほとんどのトランザクションは普通通りに完了しますが、エラーの種類によっては不整合なトランザクション結果が生じる原因となります。そのため、他に手段がない場合のみ LRCO を使ってください。トランザクションに単一の `<local-tx-datasource>` が使用される場合、LRCO が自動的に適用されます。その他の場合、特別なマーカインターフェースを使用して最終リソースを指定することが可能です。詳細は JBoss Transactions のプログラマーガイドを参照してください。

同じトランザクションに 1 フェーズリソースを複数使うと、トランザクション的に安全ではないため、推奨されません。JBoss Transaction Service は、2 番目のこのようなリソースをエラーとして登録するよう試行し、トランザクションを終了します。この種のエラーは JBoss Application Server のレガシーバージョンから移行する際に最も頻繁に見られます。できる限り、`<local-tx-datasource>` は `<xa-datasource>` に変換しこの問題を解決します。

## 11.5. トランザクションタイムアウトの処理

リソースの無限ロックを防ぐため、`<xa-datasource>` トランザクションマネージャーは指定の間隔後に完了しなかったインフライトトランザクションを停止します。これは、**TransactionReaper** が調整するバックグラウンドプロセスのセットによって停止が実行されます。リーパーは、範囲内で動作している可能性があるスレッドへ割り込まずにトランザクションをロールバックします。これにより、任意コードを実行するスレッドへの割り込みが原因で発生する不安定性を防ぐことができます。また、ビジネスロジックスレッドがネットワーク I/O 呼び出しなど割り込みできない操作を実行している場合、トランザクションを適時に停止できるようにします。しかし、この方法はマルチスレッド化されたトランザクションの処理を想定していないコードで、予想外の動作が実行される原因となることがあります。予想外のトランザクション状況の変化により、他のトランザクション対応コンポーネントなどから警告やエラーメッセージが出力されることがありますが、トランザクションの結果へは影響しないはずです。トランザクションタイムアウトの値を適切に調節することでこの問題を最小限に抑えることができます。詳細については、[15章 DataSource の設定](#) を参照してください。

## 11.6. リカバリ設定

設定を強固なものにするため、障害、リカバリに対応するため JBoss Transaction Service を正しく設定することが重要です。これについて、『JBoss Transactions Administration Guide』"Resource Recovery in JBoss Transaction Service"の章にて詳細に説明しています。

## 11.7. TRANSACTION SERVICE のよくある質問

本章では、JBoss Transaction Service の設定に関する問題で最も一般的なものをいくつか挙げていきます。

**問：** デバッグロギングを有効にしましたが何もログに残されていません。

**答：** JBossTS は 2 レベルのフィルターを使ってログステートメントを送信します。

1. ログは、JBoss Transaction Service の独自のロギング抽象層を経由します。
2. ログは、JBoss Enterprise Application Platform's **log4j** のロギングシステムを経由します。

ログステートメントを表示するには両方のフィルターを通る必要があります。よくある間違いとしてロギングシステムの片方だけを有効にすることです。詳細情報は、[表11.2「JBoss Transaction Service の追加プロパティ」](#) を参照してください。

**問：** サーバーログに **WARN Adding multiple last resources is disallowed.** が表示されトランザクションが停止されました。なぜでしょうか。

**答：** おそらく `<local-xa-datasource>` を使っており、1 フェーズ対応のパーティシパントを1つ以上使おうとしている可能性があります。この設定は回避すべきです。詳細情報については「[最終リソースコミット最適化 \(LRCO: Last Resource Commit Optimization\)](#)」を参照してください。さらに懸念がありましたら、Global Support Services にお問い合わせください。

**問：** サーバーが予期せず停止しました。現在、再稼働していますが、ログに次のようなメッセージが記録されています:**WARN [com.arjuna.ats.jta.logging.loggerI18N] [com.arjuna.ats.internal.jta.resources.arjunacore.norecoveryxa] Could not find new XAResource to use for recovering non-serializable XAResource.**

**答：** リカバリ用に全リソースマネージャーを設定していないのかもしれませんが。リソースマネージャーをリカバリさせる設定に関する詳細は、JBoss Transaction Administration Guide のリカバリの章を参照してください。

**問：** トランザクションに時間がかかり、時々予期しないことが発生します。サーバーログには次が含まれています:**WARN [arjLoggerI18N] [BasicAction\_58] - Abort of action id ... invoked while multiple threads active within it.**

**答：** タイムアウトの時間を過ぎるとトランザクションはロールバックされることがあります。これは、バックグラウンドのスレッドにより実行されますが、割り込みを予期しているアプリケーションコードが混乱することがあります。詳細は、「[トランザクションタイムアウトの処理](#)」を参照してください。

上記以外の質問がある場合、他の JBoss Transaction ガイドを参照いただくか、Global Support Services にご連絡ください。

## 11.8. JTS MODULE の利用

様々なサーバーのビジネスロジック間でトランザクションを伝搬する必要がある場合、JTS API を使うことができます。直接利用することができますが、通常は標準の JTA クラス経由でアクセスします。デフォルトであるローカルのための JTA 実装と完全互換となっており、必要なクラスはすでに設置されています。JTA と JTS モジュール間の移動は `jbossts-properties.xml` ファイルを変更するだけです。

サンプルの `jbossts-properties.xml` ファイル

は、`$JBOSS_HOME/docs/examples/transactions/` ディレクトリに置かれています。

`transactions-jboss-beans.xml` など、他のファイルに加える必要のある変更については、同じ

ディレクトリにある **README.txt** を参照してください。全ステップを自動で実行するために、ANT スクリプトが提供されていますが、このスクリプトを実行する前に **README.txt** を注意して参照していただき、既存の設定のバックアップを取るよう推奨しています。

JTS では、CORBA ORB サービスも含めるにはサーバー設定が必要です。例で参照されている "all" サーバープロファイルから始めるといいでしょう。JTS や JTA のどちらを選ぶかにより、サーバー全体に影響を与えます。JTS は追加リソースが必要ないため、必要な場合のみ使ってください。

アプリケーションの起動時、JTA を使う設定になっているサーバーは、以下のようなログファイルを出力します：

```
INFO [TransactionManagerService] JBossTS Transaction Service (JTA
version - ...)
```

JTS が有効な場合、このようなメッセージが表示されます。

```
INFO [TransactionManagerService] JBossTS Transaction Service (JTS
version - ...)
```

## 11.9. XTS モジュールの利用

JBoss Transaction Service の Web Service コンポーネントである XTS をインストールし、Enterprise Application Platform でホストされる Web サービスに WS-AT や WS-BA サポートを提供することができます。このモジュールは、**\$JBOSS\_HOME/docs/examples/transactions/** に *Service Archive* (.sar) としてパッケージ化されています。

### 手順11.1 XTS モジュールのインストール

1. **jbossxts.sar/** と呼ばれる **\$JBOSS\_HOME/server/[name]/deploy/** ディレクトリにサブディレクトリを作成します。
2. この新しいディレクトリに、ZIP アーカイブの .sar を解凍します。
3. JBoss Enterprise Application Platform を再起動し、このモジュールを有効にします。

サーバーは JBossWS Nativeに加え、JTA あるいは JTS モジュールを利用する必要があります。



#### 注記

現在のところ XTS は、CXF などの別の JBossWS バックエンドと連携するようにはなっていません。デフォルトの XTS 設定はデプロイメントの多くに手気市訂鯨。EAP 設定から自動的にネットワークインターフェースやポートバインディングに関する情報を検出します。トランザクションコーディネーターを別のホストで使う必要のあるアプリケーションをデプロイメントする場合のみ、手動での設定変更が必要です。詳細情報は、JBoss Web Service Transactions のプログラマーガイドを参照してください。

開発者は、XTS Service Archive に含まれている **jbossxts-api.jar** にリンクすることもできますが、クラスローディングの問題を回避するにはこのファイルをご利用中のアプリケーションとパッケージ化するべきではありません。これ以外の JAR ファイルにはすべて内部実装クラスが含まれているため、直接利用するべきではありません。



詳細な設定情報については、**\$JBOSS\_HOME/docs/examples/transactions/README.txt**をご覧ください。JBoss Web Services Transactions ユーザーガイドには、ご利用中のアプリケーションでの XTS 利用に関する情報が含まれています。

## 11.10. トランザクション管理コンソール

Transaction Management Console は、**\$JBOSS\_HOME/docs/example/transactions/**に含まれる簡単な GUI ツールですが、これは未対応で試験的なプロトタイプとして提供されています。Transaction Management Console の機能や、用途に関する情報は、**README.txt** を参照してください。

## 11.11. コンポーネント (トライアル)

JBoss Enterprise Application Platform に含まれる JBoss Transaction Service の対応コンポーネントに加え、継続的に機能強化作業が行われており、最終的に本製品の今後のリリースに追加される可能性があります。それまでの間、これらのプロトタイプコンポーネントは、<http://jboss.org> のコミュニティー Web サイトから入手できます。



### 警告

これらの機能が正しく動作するかは保証されておらず、Enterprise Application Platform のサポート契約にも含まれていないため注意する必要がありますが、利用可能な拡張機能によっては、開発初期段階のプロジェクトには役立つかもしれません。これらプロトタイプをダウンロードするユーザーは、「[ソースコードとアップグレード](#)」に従ってモジュールの互換性に関する制限を認識するようにしてください。

### txbridge

Web Services トランザクションの範囲内で、EJB といった従来のトランザクションコンポーネントを呼び出す機能が必要な場合もあります。反対に従来のトランザクション系アプリケーションによっては、トランザクショナルな Web サービスを呼び出す必要がある可能性もあります。Transaction Bridge (txbridge) は、この2種類のトランザクショナルサービスを連携する仕組みを提供しています。

### BA フレームワーク

XTS API は、非常に低いレベルで稼働しており、開発者は WS-BA を絡めたトランザクションインフラストラクチャーの作業を行う必要があります。BA フレームワークは、ハイレベルのアノテーションを用意しており、JBoss Transaction Service がこのようなインフラストラクチャーに対応できるようにしています。そのため、開発者は、ビジネスロジックにさらに焦点をあてることができます。

## 11.12. ソースコードとアップグレード

トランザクション関連の問題の多くは、サーバーからデバッグロギング情報を提示いただいてから、Global Support Services で検証されます。

しかし、独自のツールをお使いいただき、ご自身でソースコードをデバッグあるいはレビューすること

も可能です。<http://anonsvn.jboss.org/repos/labs/labs/jbosstm/>の Subversion レポジトリを使い、ソースコードをダウンロードすることができます。Enterprise Application Platform は起動時に以下のような文字列を利用し Transaction Service のバージョンを出力します。

```
INFO [TransactionManagerService] JBossTS Transaction Service (JTA
version - tag:JBOSSTS_4_6_1_GA_CP02) - JBoss Inc.
```

**tag** 要素は、Subversion レポジトリの /tags/ 以下のツリーに対応します。バージョンは、Enterprise Application Platform 自体のバージョンではなく、Enterprise Application Platform にて利用される JBoss Transaction Service コンポーネントのバージョンであることに注意してください。ソースから Enterprise Application Platform を構築する場合は、**component-matix/pom.xml** ファイルの文字列 **version.jboss.jbossts** を検索しバージョンを確認することができます。



#### 警告

ご利用中の Enterprise Application Platform で提供されている JBossTS のバージョン以外をインストールすることはサポート対象外となっています。JBoss Transaction Service のコンポーネントは別にパッケージされていますが、Enterprise Application Platform が提供しているバージョン以外はサポート対象外となります。



## 第12章 リモートイング

JBoss Remoting の主な目的は、プラグ可能なトランスポートとマーシャラーを使用するネットワークベースの呼び出しと関連サービスの多くに単一の API を提供することです。JBoss Remoting のAPI は、同期および非同期のリモート呼び出し、プッシュ型およびプル型のコールバック、リモートイングサーバーの自動ディスカバリなどを行う機能を提供します。これにより、異なるニーズに対して異なるトランスポートを追加できるようにしながら、リモート呼び出しに同じ API を維持するようにします。また、異なるニーズに対してコードの変更でなく設定の変更のみが必要となります。

Remoting にはそのまま使用できる複数のトランスポート (bisocket、http、rmi、ソケット、サーブレット、これらの SSL が有効になったもの)、標準かつ圧縮データマーシャラー、標準の jdk シリアル化と JBoss Serialization を切り替える設定可能機能を提供します。リモートクラスローディングも可能で、接続障害通知の大規模な機能も持っています。さらに、単一の JVM に配置されるクライアント／サーバー呼び出しに対し参照渡し最適化を実行し、マルチホーム化されたサーバーを実装します。

Enterprise Application Platform では Remoting は EJB2、EJB3、Messaging サブシステムのトランスポート層を提供します。Remoting の設定の多くは事前に決定され、固定されていますが、Remoting 設定の変更方法を知っていると便利な場合があります。

### 12.1. 背景

Remoting サーバーは、トランスポート固有のサーバー呼び出しをラップし設定するコネクタによって構成されます。コネクタは次のような InvokerLocator スtring によって示されます。

```
socket://bluemonkeydiamond.com:8888/?timeout=10000&serialization=jboss
```

この例では、ソケットトランスポートを使用しているサーバーはホスト bluemonkeydiamond.com の 8888 番ポートでアクセス可能で、ソケットタイムアウト値 10000 と JBoss Serialization を使用するようサーバーが設定されています。Remoting クライアントは InvokerLocator を使用して一定のサーバーに接続できます。

Enterprise Application Platform では、Remoting サーバーとクライアントは表面よりかなり深い位置で作成され、設定ファイルからのみアクセス可能です。また、SLSB などが JNDI ディレクトリからダウンロードされた場合、InvokerLocator のコピーもダウンロードされるため、適切な Remoting サーバーへの連絡方法を認識します。**サーバーとクライアントが InvokerLocator を共有するため、InvokerLocator のパラメーターがクライアントとサーバーの両方を設定することに注意してください。**

### 12.2. JBOSS REMOTING の設定

Remoting コネクタの作成や設定に使用できる XML ファイルには 2 つの種類があります。コネクタを MBean として定義するには、ファイル名が \*-service.xml 形式のファイルを使用します。コネクタを POJO として定義するには、ファイル名が \*-jboss-beans.xml 形式のファイルを使用します。

#### 12.2.1. MBean

JBoss Messaging JMS サブシステムでは、Remoting サーバーはファイル remoting-bisocket-service.xml で設定されます。以下はこのファイルを省略したものになります。

```
<mbean code="org.jboss.remoting.transport.Connector"
      name="jboss.messaging:service=Connector,transport=bisocket"
      display-name="Bisocket Transport Connector">
  <attribute name="Configuration">
    <config>
```

```

        <invoker transport="bisocket">
            <attribute name="marshaller"
isParam="true">org.jboss.jms.wireformat.JMSWireFormat</attribute>
            <attribute name="unmarshaller"
isParam="true">org.jboss.jms.wireformat.JMSWireFormat</attribute>
            <attribute name="serverBindAddress">${jboss.bind.address}
</attribute>
            <attribute name="serverBindPort">4457</attribute>
            <attribute name="callbackTimeout">10000</attribute>
            ...
        </invoker>
        ...
    </config>
</attribute>
</mbean>

```

この設定ファイルから次のような情報を得ることができます。

- このサーバーは bisocket トランスポートを使用
- ホスト \${jboss.bind.address}; の 4457 番ポートで稼働
- JBoss Messaging は独自のマーシャリングアルゴリズムを使用

InvokerLocator はこのファイルに由来しています。ここで重要なのは、**InvokerLocator** にパラメーターが含まれるかを「**isParam**」属性が判断するということです。「isParam」が省略されるか、false に設定されると、パラメーターが適用されるのはサーバーのみでクライアントには通信されません。Remoting サーバーの InvokerLocator に bluemonkeydiamond.com という \${jboss.bind.address} がある場合、次のようになります。

```

bisocket://bluemonkeydiamond.com:4457/?marshaller=
org.jboss.jms.wireformat.JMSWireFormat&
unmarshaller=org.jboss.jms.wireformat.JMSWireFormat

```

InvokerLocator には「callbackTimeout」パラメーターは含まれていません。

## 12.2.2. POJO

**org.jboss.remoting.ServerConfiguration** POJO のように同じコネクタを設定することができます。

```

<bean name="JBMConnector"
class="org.jboss.remoting.transport.Connector">
    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
        (name="jboss.messaging:service=Connector,transport=bisocket",
exposedInterface=org.jboss.remoting.transport.ConnectorMBean.class,
        registerDirectly=true)</annotation>
    <property name="serverConfiguration"><inject
bean="JBMConfiguration"/></property>
</bean>

<!-- Remoting server configuration -->
<bean name="JBMConfiguration"
class="org.jboss.remoting.ServerConfiguration">

```

```

    <constructor>
      <parameter>bisocket</parameter>
    </constructor>

    <!-- Parameters visible to both client and server -->
    <property name="invokerLocatorParameters">
      <map keyClass="java.lang.String" valueClass="java.lang.String">
        <entry>
          <key>serverBindAddress</key>
          <value>
            <value-factory bean="ServiceBindingManager"
method="getStringBinding">
              <parameter>JBMConnector</parameter>
              <parameter>${host}</parameter>
            </value-factory>
          </value>
        </entry>
        <entry>
          <key>serverBindPort</key>
          <value>
            <value-factory bean="ServiceBindingManager"
method="getStringBinding">
              <parameter>JBMConnector</parameter>
              <parameter>${port}</parameter>
            </value-factory>
          </value>
        </entry>
        ...
        <entry><key>marshaller</key>
<value>org.jboss.jms.wireformat.JMSWireFormat</value></entry>
        <entry><key>unmarshaller</key>
<value>org.jboss.jms.wireformat.JMSWireFormat</value></entry>
      </map>
    </property>

    <!-- Parameters visible only to server -->
    <property name="serverParameters">
      <map keyClass="java.lang.String" valueClass="java.lang.String">
        <entry><key>callbackTimeout</key> <value>10000</value></entry>
      </map>
    </property>

    ...
  </bean>

```

このバージョンでは、設定情報は JBMConfiguration **ServerConfiguration** POJO に表現され、JBMConnector **org.jboss.remoting.transport.Connector** POJO に挿入されます。構文は Microcontainer の構文ですが、本章の範囲外となりますので、詳細は [7章 マイクロコンテナ](#) を参照してください。また、MBean バージョンのバリエーションの 1 つが ServiceBindingManager の使用になりますが、これも本章の範囲外となります。@org.jboss.aop.microcontainer.aspects.jmx.JMX アノテーションによって、JBMConnector が「jboss.messaging:service=Connector,transport=bisocket」という名前の MBean として表示されるようになります。

## 12.3. マルチホーム化されたサーバー

Remoting は複数のインターフェースにバインドされるサーバーを作成することができます。このファシリティの適応例として、1つのサーバーをインターネットに面するインターフェースと LAN に面する別のインターフェースにバインドすることができます。例えば、前述の POJO 例に POJO を追加して (1) 変更することができます。

```
<bean name="homes1" class="java.lang.StringBuffer">
  <constructor>
    <parameter class="java.lang.String">
      <value-factory bean="ServiceBindingManager"
method="getStringBinding">
        <parameter>JBMConnector:bindingHome1</parameter>
        <parameter>${host}:${port}</parameter>
      </value-factory>
    </parameter>
  </constructor>
</bean>

<bean name="homes2" class="java.lang.StringBuffer">
  <constructor factoryMethod="append">
    <factory bean="homes1"/>
    <parameter>
      <value-factory bean="ServiceBindingManager"
method="getStringBinding">
        <parameter>JBMConnector:bindingHome2</parameter>
        <parameter>!${host}:${port}</parameter>
      </value-factory>
    </parameter>
  </constructor>
</bean>
```

これにより、StringBuffer の値が「external.acme.com:5555!internal.acme.com:4444」のようになります (JBMConnector:bindingHome1 and JBMConnector:bindingHome2 のServiceBindingManager 設定値による)。また、「serverBindAddress」パラメーターと「serverBindPort」パラメーターが次に置き換えられます (2)。

```
<entry>
  <key>homes</key>
  <value><value-factory bean="homes2" method="toString"/></value>
</entry>
```

これにより、StringBuffer が文字列「external.acme.com:5555!internal.acme.com:4444」に変換され、JBMConnector に挿入されます。結果、InvokerLocator は次のようになります。

```
bisocket://multihome/?homes=external.acme.com:5555!internal.acme.com:
4444&marshaller=org.jboss.jms.wireformat.JMSWireFormat&
unmarshaller=org.jboss.jms.wireformat.JMSWireFormat
```

## 12.4. アドレス変換

ファイアウォールを変換するアドレスよりサーバーへアクセスしなければならない場合があります。Remoting サーバーは、バインドするアドレス／ポートとクライアントが使用するアドレス／ポートの両方で設定することができます。追加で「clientConnectAddress」と「clientConnectPort」の2つのパラメーターが使用されます。「serverBindAddress」と「serverBindPort」の値はサーバーの作成に

使用されます。「clientConnectAddress」と「clientConnectPort」の値は InvokerLocator で使用され、クライアントにサーバーの場所を伝えます。また、マルチホームサーバー向けの「connecthome」パラメーターもあります。この場合、サーバーの設定に「home」が使用され、「connecthome」がクライアントにサーバーの場所を伝えます。

## 12.5. 設定ファイルの場所

サポートされるサブシステムの実際の Remoting 設定ファイルは次の通りです。

EJB2: **`$JBOSS_HOME/server/$CONFIG/deploy/remoting-jboss-beans.xml`**

EJB3: **`$JBOSS_HOME/server/$CONFIG/deploy/ejb3-connectors-jboss-beans.xml`**

JBM: **`$JBOSS_HOME/server/$CONFIG/deploy/messaging/remoting-bisocket-service.xml`**

## 12.6. 詳細情報

その他の詳細情報は、<http://www.jboss.org/jbossremoting/docs/guide/2.5/html/index.html> の Remoting ガイドを参照してください。

## 第13章 JBOSS MESSAGING

JBoss Messaging の使用に関する最新情報は、[http://www.redhat.com/docs/en-US/JBoss\\_Enterprise\\_Application\\_Platform/](http://www.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/) の『JBoss Messaging User Guide』を参照してください。

## 第14章 JBOSS ENTERPRISE APPLICATION PLATFORM で別のデータベースを使用

### 14.1. 代わりのデータベースの使用法

JBoss はデフォルトのデータベースとして Hypersonic データベースを使用します。Hypersonic データベースの使用は開発やプロトタイピングには便利ですが、実稼働環境で使用するには、通常は別のデータベースが必要となります。本章では、JBoss Enterprise Application Platform が代替データベースを使用するよう設定する方法を説明します。JBoss Enterprise Application Platform 上で正式サポートされるデータベースすべてを対象とします。認定データベースの総合一覧は、<http://www.jboss.com/products/platforms/application/supportedconfigurations/>を参照してください。

本章では、JBoss Enterprise Application Platform 内の全サービスをサポートするため代替データベースを使用する方法について説明します。これには、EJB や JMS など全システムレベルのサービスが含まれます。JBoss Enterprise Application Platform にデプロイされている各アプリケーション (WAR や EAR など) に対して、適切なデータソース接続を設定すれば、バックエンドのデータベースも使用することができます。

外部データベースのインストールは本書の対象外となっています。ご利用中のデータベースのベンダーが提供するツールを使い、空のデータベースを設定してください。データベースへ接続するために当プラットフォームが使用するデータソースを作成するには、データベース名、接続 URL、ユーザー名、パスワードが必要になります。

### 14.2. JDBC ドライバーのインストール

選択した外部データベースを使うには、ご利用中のデータベースに JDBC ドライバーもインストールする必要があります。JDBC ドライバーは、JAR ファイルで **\$JBOSS\_HOME/server/PROFILE/lib** ディレクトリに設置する必要があります。また、**PROFILE** を利用予定のサーバープロファイルで置き換えます。

JBoss Enterprise Application Platform が起動すると、このファイルがロードされるため、JBoss Enterprise Application Platform が稼働中であれば、一旦終了し再起動してください。適切な JDBC ドライバーについては以下の一覧を参照してください。認定済みの JBoss Enterprise Application Platform データベースドライバーの総合一覧について

は、<http://www.jboss.com/products/platforms/application/supportedconfigurations/#JEAP5-0> を参照してください。このリンクが壊れている場合、JIRA の本書の項目から報告してください。ただし、Red Hat ではこういった外部リンクの管理は行っておりませんので、ご注意ください。ご利用中のデータベース向けの最新版ドライバーについてはデータベースベンダーにお問い合わせください。

#### JBDC ドライバーのダウンロード先

##### MySQL

<http://www.mysql.com/products/connector/> からダウンロード

##### PostgreSQL

<http://jdbc.postgresql.org/> からダウンロード

##### Oracle

[http://www.oracle.com/technology/software/tech/java/sqlj\\_jdbc/index.html](http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html) からダウンロード

##### IBM

<http://www-306.ibm.com/software/data/db2/java/> からダウンロード

## Sybase

Sybase jConnect 製品ページ <http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect> からダウンロード



### 注記

このドライバーで Sybase データベースを使うと、ドライバーの **PreparedStatement** クラスでの制限が原因で **MaxParams** 属性を **481** 以上に設定することができません。

## Microsoft

MSDN Web サイト <http://msdn.microsoft.com/data/jdbc/> からダウンロード

### 14.2.1. Sybase に関する注記

JBoss のサービスによっては、作成されたデフォルトテーブルに対し null の値を使用することもあるため、Sybase Adaptive Server はデフォルトで null を許可するように設定する必要があります。

```
sp_dboption db_name, "allow nulls by default", true
```

その他のオプションについては、Sybase のマニュアルを参照してください。

さらに、データベースに格納される文字や画像の値は非常に大きくなる可能性があります。選択した一覧に文字と画像の値が両方含まれる場合、返されるデータの長さ制限は、**@@textsize** のグローバル変数で決定されます。この変数のデフォルト設定は、Adaptive Server へアクセスするのに利用するソフトウェアによって変わります。JDBC ドライバーについては、デフォルト値は 32 キロバイトとなっています。

#### 14.2.1.1. JAVA サービスの有効化

JMS、CMP、タイマーなど、Sybase で設定された Java サービスを使用するには、Sybase Adaptive Server 上で Java が有効になっている必要があります。Java を有効にする方法：

```
sp_configure "enable java",1
```

詳細は、Sybase のマニュアルを参照してください。

Sybase Adaptive Server に対して、Java が有効でない場合、コンソールに以下のエラーメッセージが表示される可能性があります。

```
com.sybase.jdbc2.jdbc.SybSQLException: Cannot run this command because
Java services are not
    enabled. A user with System Administrator (SA) role must
reconfigure the system to enable Java
```

#### 14.2.1.2. CMP の設定

System Adaptive Server Enterprise で、ユーザ定義の Java オブジェクトに CMP (Container Managed



Persistence: コンテナ管理の永続性) を使用するには、ご利用中のデータベースに Java クラスをインストールする必要があります。システムテーブル **sysxtypes** には、各拡張 Java-SQL データタイプに対して 1 つの行が含まれており、このテーブルは、Java が有効な Adaptive Server のみに利用されます。Java のために有効になっている Adaptive Server のみに使用されます。**installjava** プログラムを使用して java クラスをインストールしてください。

```
installjava -f <jar-file-name> -S<sybase-server> -U<super-user> -P<super-pass> -D<db-name>
```

別のオプションについては、Sybase の **installjava** マニュアルを参照してください。

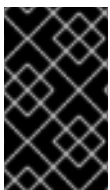
### 14.2.1.3. Java クラスのインストール

1. Java クラスをインストールするには、必要な権限を持つスーパーユーザーになる必要があります。
2. インストールする JAR ファイルは圧縮せずに作成する必要があります。
3. インストールしてサーバー内で使用する Java クラスは JDK 1.2.2 でコンパイル されている必要があります。より新しい JDK で1つのクラスをコンパイルしても、**installjava** ユーティリティを使用してサーバーにインストールできますが、そのクラスの使用を試す時に、`java.lang.ClassFormatError` 例外が発生します。これは、Sybase Adaptive Server が内部で古い JVM を使用しているため、それと同じもので Java クラスをコンパイルする必要があるからです。

### 14.2.1.4. Sybase v15.0.3 向けにデフォルトの @@textsize を増加

データベースから返されるデフォルトの最大テキストサイズの値に問題があると、JBoss Messaging 上でテストの一部が失敗してしまう可能性があります。この問題を修正するには、デフォルトの **@@textsize** を 32768 (バイト) から 2147483647 (バイト) に変更してください。

<connection-url> ディレクティブの中にある **sybase-ds.xml** に変更を加えます。サンプルの **sybase-ds.xml** ファイルは、**JBOSS\_DIST/jboss-as/docs/examples/jca/** に置かれています。



#### 重要

ディレクティブ値としてデータベースへの全 URL を指定します。*[domain]* を Sybase Database をホストしているドメイン名か IP アドレスと置き換え、*[port]* をリクエストを受け取るよう設定されたポートと置き換えてください。

```
<connection-url>jdbc:sybase:[domain]:[port]/db_name?SQLINITSTRING=set  
TextSize 2147483647</connection-url>
```

### 14.2.2. JDBC DataSource の設定

データソースは、簡素化された JCA Datasource 設定仕様に対応します。

データソースは、別のデプロイ可能なアプリケーションやリソースと共に **\$JBOSS\_HOME/server/PROFILE/deploy** ディレクトリに設置する必要があります。このファイルは、**DBNAME-ds.xml**の標準ネーミングスキームを使います。

認定済みのデータベースすべてに対するデータソース例は、`$JBOSS_HOME/docs/examples/jca` ディレクトリに保存されています。ご利用中のデータベースに対応するデータソースを編集し、アプリケーションサーバーを再起動する前に `deploy/` ディレクトリにコピーします。

データソース設定に関する情報については、[15章 \*DataSource の設定\*](#) を参照してください。ご利用中のデータベースに対応させるには、少なくとも `connection-url`、`user-name`、`password` を変更する必要があります。

## 14.3. 一般的なデータベース関連のタスク

### 14.3.1. セキュリティとプーリング

`ResourceAdapter` に `<reauthentication-support>` がある場合を除き、複数のセキュリティ ID を使用すると各 ID に対してサブプールが作成されます。



#### 注記

最小および最大プールサイズはサブプールごとの容量になります。ID の数が多い場合はこれらのパラメーターに注意してください。

### 14.3.2. JMS サービス用にデータベースを変更

JBoss Enterprise Application Platform の JMS サービスは関係データベースを使用してメッセージを永续します。パフォーマンス向上のため、JMS サービスを変更して外部データベースを有効活用するようにします。それには、外部データベースに応じてファイル

`$JBOSS_HOME/server/PROFILE/deploy/messaging/$DATABASE-persistence-service.xml` をファイル `$DATABASE-persistence-service.xml` に置き換える必要があります。

- MySQL: `mysql-persistence-service.xml`
- PostgreSQL: `postgresql-persistence-service.xml`
- Oracle: `oracle-persistence-service.xml`
- DB2: `db2-persistence-service.xml`
- Sybase: `sybase-persistence-service.xml`
- MS SQL Server: `mssql-persistence-service.xml`

### 14.3.3. CMP サービスで外部キーのサポート

次に、`fk-constraint` プロパティが `true` になるよ

う、`$JBOSS_HOME/server/PROFILE/conf/standardjbosscmp-jdbc.xml` ファイルを変更する必要があります。JBoss Enterprise Application Platform 上でサポートする全ての外部データベースに対して変更する必要があります。このファイルは、JBoss Enterprise Application Platform にデプロイされた EJB2 CMP Bean のデータベース接続設定を構成します。

```
<fk-constraint>true</fk-constraint>
```

### 14.3.4. Java Persistence API 用にデータベースダイアレクトを指定

Java Persistence API (JPA) エンティティマネージャーは EJB3 エンティティ Bean をバックエンドデータベースに保存できます。Hibernate は JBoss Enterprise Application Platform に JPA 実装を提供します。Hibernate は、付録にある下記データベースのダイアレクトなど多くのデータベースで動作する、ダイアレクト自動検知メカニズムを持っています。代替データベースに特定のダイアレクトが必要な場合は、**\$JBOSS\_HOME/server/PROFILE/deployers/ejb3.deployer/META-INF/jpa-deployers-jboss-beans.xml** ファイルにデータベースファイルを設定することができます。このファイルを設定するには、マップエントリ **hibernate.dialect** に関連するタグセットをアンコメントし、設定するデータベースに応じて次の値に変更する必要があります。

- Oracle 10g: **org.hibernate.dialect.Oracle10gDialect**
- Oracle 11g: **org.hibernate.dialect.Oracle10gDialect**
- Microsoft SQL Server 2005: **org.hibernate.dialect.SQLServerDialect**
- Microsoft SQL Server 2008: **org.hibernate.dialect.SQLServerDialect**
- PostgreSQL 8.2.3: **org.hibernate.dialect.PostgreSQLDialect**
- PostgreSQL 8.3.7: **org.hibernate.dialect.PostgreSQLDialect**
- MySQL 5.0: **org.hibernate.dialect.MySQL5InnoDBDialect**
- MySQL 5.1: **org.hibernate.dialect.MySQL5InnoDBDialect**
- DB2 9.1: **org.hibernate.dialect.DB2Dialect**
- Sybase ASE 15: **org.hibernate.dialect.SybaseASE15Dialect**

### 14.3.5. 外部データベースを使用するよう他の JBoss Enterprise Application Platform サービスを変更

JMS や CMP や JPA 以外にも、まだ他の JBoss サービス群を外部データベースと組み合わせる必要があります。それには2つの方法があります。1つは、簡単ですが柔軟性がありません。もう1つは、柔軟性に富みますがより多くのステップを必要とします。ここで、それらの2つのアプローチを別々に説明していきましょう。

#### 14.3.5.1. 簡単な方法

簡単な方法とは、外部データベース用の JNDI 名を **DefaultDS** に変更することです。ほとんどの JBoss サービスはデフォルトで **DefaultDS** を使用するように設定されています。そのため、DataSource 名を変更するだけで、各サービス用に個別設定を変更する必要はありません。

JNDI 名を変更するには、外部データベースの **\*-ds.xml** ファイルを開き、**jndi-name** プロパティの値を **DefaultDS** に変更します。例えば、**mysql-ds.xml** では、**MySQLDS** を **DefaultDS** などに変更することになります。この後、**DefaultDS** 定義の重複を避けるために、**\$JBOSS\_HOME/server/PROFILE/deploy/hsqldb-ds.xml** ファイルを削除する必要があります。

**messaging/\$DATABASE-persistence-service.xml** ファイルでは、**PersistenceManagers** MBean の **depends** タグのデータソース名も **DefaultDS** に変更する必要があります。例えば、**mysql-persistence-service.xml** ファイルの場合、**MySQLDS** を **DefaultDS** に変更します。

```
<mbean
```

```
code="org.jboss.messaging.core.jmx.JDBCPersistenceManagerService"
```

```
name="jboss.messaging:service=PersistenceManager"
xmbbean-dd="xmdesc/JDBCPersistenceManager-xmbbean.xml">
```

```
<depends>jboss.jca:service=DataSourceBinding,name=DefaultDS</depends>
```

#### 14.3.5.2. より柔軟な方法

外部 datasource を **DefaultDS** に変更することは便利です。しかし、**DefaultDS** が常にファクトリデフォルトの HSQL DB を ポイントすることを想定してしまうアプリケーションがある場合、このやり方はアプリケーションを破損する可能性があります。また、**DefaultDS** 目的地を変更することは、全ての JBoss サービスが外部データベースを使用することを強制することになります。その場合、一部のサービス用のみに外部データベースを使用したい時に問題になるでしょう。

より安全で柔軟に JBoss Enterprise Application Platform サービスを外部データベースに組み合わせるには、標準の JBoss サービスすべての **DefaultDS** を、手作業で **\*-ds.xml** ファイル内で定義されたデータソース JNDI 名 (**mysql-ds.xml** 内の **MySQLDS** など) に変更します。**DefaultDS** を含むファイルの完全一覧は次の通りです。これら全てを更新して全 JBoss サービスで外部データベースを使用するか、一部を更新してサービスごとに異なる組み合わせのデータソースを使用するようにします。

- **\$JBOSS\_HOME/server/PROFILE/conf/login-config.xml**: このファイルは Java EE コンテナで管理するセキュリティサービス内で使用されます。
- **\$JBOSS\_HOME/server/PROFILE/conf/standardjbosscmp-jdbc.xml**: このファイルは EJB コンテナ内で CMP Bean を設定します。
- **\$JBOSS\_HOME/server/PROFILE/deploy/ejb2-timer-service.xml**: このファイルは EJB タイマーサービスを設定します。
- **\$JBOSS\_HOME/server/PROFILE/deploy/juddi-service.sar/META-INF/jboss-service.xml**: このファイルは UUDI サービスを設定します。
- **\$JBOSS\_HOME/server/PROFILE/deploy/juddi-service.sar/juddi.war/WEB-INF/jboss-web.xml**: このファイルは UUDI サービスを設定します。
- **\$JBOSS\_HOME/server/PROFILE/deploy/juddi-service.sar/juddi.war/WEB-INF/juddi.properties**: このファイルは UUDI サービスを設定します。
- **\$JBOSS\_HOME/server/PROFILE/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml**: このファイルは UUDI サービスを設定します。
- **\$JBOSS\_HOME/server/PROFILE/deploy/messaging/messaging-jboss-beans.xml** および **\$JBOSS\_HOME/server/PROFILE/deploy/messaging/persistence-service.xml**: これらのファイルは前述の通り JMS 永続サービスを設定します。

#### 14.3.6. Oracle DataBases に関する注記

本章で説明する設定は、JBoss Enterprise Application Platform がサーバーの起動時に外部データベース内で必要なテーブルを自動的に作成することを前提としています。ほとんどの場合で問題ありませんが、Oracle のようにデータベースによっては、同じデータベースサーバーを使用して複数の JBoss Enterprise Application Platform インスタンスをサポートすると軽度な問題が発生することがあります。

Oracle データベースは **schemaname.tablename** 形式のテーブルを作成します。JBoss Enterprise Application Platform で必要となる **TIMERS** テーブルと **HILOSEQUENCES** テーブルが別のスキーマ上に

既に存在する場合、スキーマ上には作成されません。この問題を回避するには、**\$JBOSS\_HOME/server/PROFILE/deploy/ejb2-timer-service.xml** ファイルを編集し、テーブル名を **TIMERS** から **schemaname2.tablename** のように変更します。

```
<mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"
name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
  <!-- DataSourceBinding ObjectName -->
  <depends optional-attribute-name="DataSource">
    jboss.jca:service=DataSourceBinding,name=DefaultDS
  </depends>
  <!-- The plugin that handles database persistence -->
  <attribute name="DatabasePersistencePlugin">
    org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin
  </attribute>
  <!-- The timers table name -->
  <attribute name="TimersTable">TIMERS</attribute>
</mbean>
```

同様に、**\$JBOSS\_HOME/server/PROFILE/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml** ファイルも編集し、テーブル名を **HILOSEQUENCES** から **schemaname2.tablename** のように変更します。

```
<!-- HiLoKeyGeneratorFactory --> <mbean
code="org.jboss.ejb.plugins.keygenerator.hilo.HiLoKeyGeneratorFactory"
name="jboss:service=KeyGeneratorFactory,type=HiLo">

  <depends>jboss:service=TransactionManager</depends>

  <!-- Attributes common to HiLo factory instances -->

  <!-- DataSource JNDI name -->
  <depends optional-attribute-
name="DataSource">jboss.jca:service=DataSourceBinding,name=DefaultDS</depe
nds>

  <!-- table name -->
  <attribute name="TableName">HILOSEQUENCES</attribute>
```

## 重要

Oracle 11g R1 で稼働する場合、Oracle JDBC driver version 11.1.0.7.0 が原因で、JBoss Messaging Test Suite が **SQLException** ("Bigger type length than Maximum") を返し問題が発生します。

これは、Oracle JDBC ドライバー 11.1.0.7.0 の再発バグが原因となっています。

Oracle 11g R1、Oracle RAC 11g R1、Oracle RAC 11g R2 では、Oracle JDBC ドライバーのバージョン 11.2.0.1.0 を使うように推奨されます。

## 第15章 DATASOURCE の設定



### 警告

デフォルトの永続化設定は Hypersonic (HSQLDB) でそのまま機能するため、JBoss Enterprise Platform はカスタマイズなしにそのまま稼働することができます。ただし、**Hypersonic** は実稼働環境で対応していないため、実稼働環境で利用すべきではありません。

Hypersonic Database の既知の問題は以下のとおりです：

- トランザクション分離がない
- スレッドおよびソケットがリークする (`connection.close()` はリソースの整理を行いません)
- 持続品質 (ログは通常、障害後は壊れてしまい自動回復ができなくなります)
- データベースの破損
- 負荷がかけられた際の安定性 (過剰なデータを処理すると、データベース処理は停止されます)
- クラスター化環境では実行不可

サポートが必要であれば、『スタートガイド』の「別のデータベースを利用」の章を確認してください。

DataSource は `<datasources>` 要素内に定義されます。正確な要素は、必要とされるデータソースの種類により変化します。

### 15.1. DATASOURCE の種類

#### DataSource 定義

##### `<no-tx-datasource>`

JTA トランザクションには関与せず、`java.sql.Driver` を使用します。

##### `<local-tx-datasource>`

2 相コミットには対応しておらず、`java.sql.Driver` を使用します。単一データベースあるいは非 XA 対応リソースに適しています。

##### `<xa-datasource>`

2 相コミットをサポートせず、`javax.sql.XADataSource` を使用します。

## 15.2. DATASOURCE パラメーター

### 一般的な DataSource パラメーター

#### <mbean>

標準的な JBoss MBean デプロイメント

#### <depends>

この **ConnectionFactory** または **DataSource** デプロイメントが依存する MBean サービスの **ObjectName** です。

#### <jndi-name>

DataSource がバインドされる JNDI 名です。

#### <use-java-context>

jndi-name のプレフィックスを *java:* とすべきかを示す Boolean 値です。このプレフィックスは JBoss Enterprise Application Platform の仮想マシン内からしか DataSource にアクセスできなくなります。デフォルトは **true** です。

#### <user-name>

データソースへの接続作成時に使用されるユーザー名です。

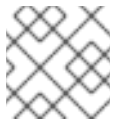


#### 注記

セキュリティ設定されている場合は利用されません。

#### <password>

Datasource への接続が作成される際に使用されるパスワードです。



#### 注記

セキュリティ設定されている場合は利用されません。

#### <transaction-isolation>

接続のデフォルトトランザクション分離です。指定しないとデータベースが提供するデフォルト値が使用されます。

#### <transaction-isolation> に対し可能な値

- TRANSACTION\_READ\_UNCOMMITTED
- TRANSACTION\_READ\_COMMITTED
- TRANSACTION\_REPEATABLE\_READ
- TRANSACTION\_SERIALIZABLE
- TRANSACTION\_NONE

**<new-connection-sql>**

新規接続に対して実行される SQL ステートメントです。接続スキーマの設定などに使用できます。

**<check-valid-connection-sql>**

有効であるかを確認するため、プールからチェックアウトされる前に実行される SQL ステートメントです。この SQL ステートメントが失敗すると接続が閉じられ、新しい接続が作成されます。

**<valid-connection-checker-class-name>**

ベンダー固有のメカニズムを使用して接続が有効かを確認するクラスです。

**<exception-sorter-class-name>**

ベンダー固有のメッセージを解析し、SQL エラーが致命的でこの接続を破棄するかを判断するクラスです。空の場合、致命的エラーとして処理されるエラーはありません。

**<track-statements>**

閉じられていない Statement や ResultSet を監視し、閉じられていない場合に警告を発行するかを指定します。デフォルト値は **NOWARN** です。

**<prepared-statement-cache-size>**

後続のリクエストで閉じずに再利用する、接続毎の prepared ステートメントの数です。ステートメントは *Least Recently Used (LRU)* キャッシュに保存されます。デフォルト値は **0** で、これはキャッシュが保持されないということです。

**<share-prepared-statements>**

**<prepared-statement-cache-size>** が非ゼロの場合、同じトランザクション内の 2 つのリクエストが同じステートメントを返すべきであるかを指定します。デフォルトは **FALSE** です。

**例15.1 <share-prepared-statements>の利用**

目的は、ドライバーが自動コミットセマンティックをローカルトランザクションに適応してしまうこのドライバーの動作を回避することです。

```

    Connection c = dataSource.getConnection(); // auto-commit ==
false
    PreparedStatement ps1 = c.prepareStatement(...);
    ResultSet rs1 = ps1.executeQuery();
    PreparedStatement ps2 = c.prepareStatement(...);
    ResultSet rs2 = ps2.executeQuery();

```

これは、prepared ステートメントが同じであることを想定しています。ドライバーによっては **ps2.executeQuery()** が自動的に **rs1** を閉じてしまうため、背後に 真の prepared ステートメントが2つ必要となります。これは、クエリの再実行が自動的に新しいトランザクションを開始する、自動コミットセマンティックのみが目的でなければなりません。仕様に準拠するドライバーの場合、**TRUE** を設定して同じ真の prepared ステートメントを共有することができます。

**<set-tx-query-timeout>**

トランザクションがタイムアウトするまでの残り時間を基にクエリのタイムアウトを有効にするか指定します。デフォルトは **FALSE** です。



### <query-timeout>

クエリがタイムアウトするまでの最大時間 (秒)。<set-tx-query-timeout>を **TRUE** に設定することでこの値をオーバーライドすることができます。

### <type-mapping>

conf/standardjbosscmp.xml 内の型マッピングへのポインターです。この要素は、<metadata>の子要素です。JBoss4 からのレガシー。

### <validate-on-match>

プール外の接続を確認した場合など、JCA 層が管理された接続と一致する場合にこの接続を検証するかどうか指定します。<background-validation>を追加することで、これは必要なくなります。通常、<background-validation> に **TRUE** を指定するとともに、<validate-on-match> に **TRUE** を指定する必要はありません。デフォルト値は、**TRUE**です。

### <prefill>

接続プールに最小接続数を事前に入力を試みるかを指定します。supporting pool (OnePool) のみがこの機能をサポートします。このプールが事前入力に対応しない場合、ログに警告が記録されます。デフォルトは **TRUE** となっています。

### <background-validation>

バックグラウンドの接続検証は、接続を検証する際の RDBMS システム上の全体的な負荷を軽減します。この機能を使用すると、EAP はプール内の現在の接続が別のスレッド (ConnectionValidator) であるかをチェックします。<background-validation-minutes>は、**TRUE** に設定されているこの値により変わります。デフォルトは **FALSE** です。

### <background-validation-millis>

バックグラウンドの接続検証は、接続を検証する際の RDBMS システム上の全体的な負荷を軽減します。このパラメーターを設定すると、JBoss はプール内の現在の接続を個別のスレッド (ConnectionValidator) として検証しようとします。パラメーターの値が ConnectionValidator の実行周期 (ミリ秒単位) を定義します (この値は <idle-timeout-minutes> とは違う値でなければなりません)。

### <idle-timeout-minutes>

アイドル状態の接続が閉じられるまでの最大時間 (分単位) を示します。値が0 の場合はタイムアウトが無効となります。デフォルトは **15** です。

### <track-connection-by-tx>

トランザクションの最後に接続をプールに戻すの代わりに、接続をトランザクションにロックすべきかを指定します。以前のリリースでは、ローカル接続ファクトリのデフォルト値は **true** で、XA 接続ファクトリのデフォルト値は **false** でした。現在、ローカル接続ファクトリと XA 接続ファクトリ両方のデフォルト値が **true** となり、要素は廃止されました。

### <interleaving>

XA 接続ファクトリのインターリービングを有効にします。

### <background-validation-minutes>

ConnectionValidator が実行される頻度 (分単位) になります。デフォルトは **15** になります。



## 注記

<min-pool-size> を最小プールサイズセットに指定していない限り、これを、<idle-timeout-minutes>よりも小さい値に設定する必要があります。

### <url-delimiter>, <url-property>, <url-selector-strategy-class-name>

データベースのフェールオーバーを処理するパラメーター。JBoss Enterprise Application Platform 現在、これらのパラメーターは主なデータソース設定の一部として設定されています。以前のバージョンでは、<url-delimiter> は <url-delimeter> として表示されていました。

### <stale-connection-checker-class-name>

異常な接続を通知する **SQLException** が **org.jboss.resource.adapter.jdbc.StateConnectionException** の例外をスローするかを決定する **org.jboss.resource.adapter.jdbc.StateConnectionChecker** 実装です。

### <max-pool-size>

プール内で接続可能な最大数です。定義されていない場合は、デフォルトは **10** となっています。datasource の定義例 (**\$JBOSS\_HOME/server/PROFILE/hsqldb-ds.xml**) の値は、20 に設定されています。

### <min-pool-size>

プールで保持される最小接続数です。<prefill> が **TRUE** でない限り、プールが<min-pool-size>まで埋まった時点で最初に利用されるまで、プールは空ままとなります。アイドルによるタイムアウトが原因で、プールサイズが<min-pool-size>以下に下がった場合、プールは<min-pool-size>まで補充されます。デフォルトは、**0**となっています。

### <blocking-timeout-millis>

すべての接続がチェックアウトされた時、接続が使用できるまで待機する時間になります。デフォルトは **30000** (30 秒) になります。

### <use-fast-fail>

以前の試行が失敗あるいはフェールオーバーが開始された場合でも、プールからの接続取得を試行するかを指定します。これは、SQL 検証の実行に時間とリソースが多くかかるようなパフォーマンスの問題に対処するためです。デフォルトは、**FALSE**となっています。

## javax.sql.XDataSource を使用するためのパラメーター

### <connection-url>

JDBC ドライバー接続の URL スtring です。

### <driver-class>

**java.sql.Driver** を実装する JDBC ドライバークラスです。

### <connection-property>

**java.sql.Driver** から取得した接続を設定するために使用されます。

#### 例15.2 例 : <connection-property>

```
<connection-property name="char.encoding">UTF-8</connection-  
property>
```

`javax.sql.XADataSource` を使用するためのパラメーター

#### <xa-datasource-class>

`XADataSource` を実装するクラスです。

#### <xa-datasource-property>

`XADataSource` の設定に使用されるプロパティです。

#### 例15.3 例： <xa-datasource-property> の宣言

```
<xa-datasource-property name="IfxWAITTIME">10</xa-datasource-  
property>  
<xa-datasource-property name="IfxIFXHOST">myhost.mydomain.com</xa-  
datasource-property>  
<xa-datasource-property name="PortNumber">1557</xa-datasource-  
property>  
<xa-datasource-property name="DatabaseName">mydb</xa-datasource-  
property>  
<xa-datasource-property name="ServerName">myserver</xa-datasource-  
property>
```

#### <xa-resource-timeout>

0 でない場合、`XAResource.setTimeout()` を経過した秒数になります。

#### <isSameRM-override-value>

`FALSE` に設定した場合、Oracle データベースの問題の一部を修正します。

#### <no-tx-separate-pools>

トランザクショナル接続と非トランザクショナル接続を別々にプールします。



#### 警告

このオプションを使うと、実際のプールが2つ作成されるため、合計のプール容量が`max-pool-size`の2倍になります。

Oracle の問題を修正するために利用されます。

#### セキュリティパラメーター

**<application-managed-security>**

**getConnection** に渡されたユーザー名とパスワードを使用するか、アプリケーションによる **createConnection** リクエストを使用します。

**<security-domain>**

**conf/login-module.xml** に設定されている確認されたログインモジュールを使用します。

**<security-domain-and-application>**

**conf/login-module.xml** に設定されている確認されたログインモジュールや、アプリケーション (JMS のキューやトピックなど) によって提供される他の接続要求情報を使用します。

**JCA レイヤーにてXAをリカバリするためのパラメーター****<recover-user-name>**

リカバリ操作を行うことのできる認証情報を持つユーザー

**<recover-password>**

リカバリ操作を行うことのできるユーザー (認証情報付き) のパスワード

**<recover-security-domain>**

リカバリ用のセキュリティドメイン

**<no-recover>**

リカバリからのデータソースを除外

JCA レイヤーにてXAをリカバリするためのパラメーターのフィールドに、non-recover と対となるフォールバック値 (<user-name>,< password> and <security-domain>) を入力する必要があります。

**15.3. データソースの例**

データベース固有の例については、[付録A ベンダー固有のデータソース定義](#) を参照してください。

**15.3.1. 汎用データソースの例****例15.4 汎用データソースの例**

```
<datasources>
<local-tx-datasource>
  <jndi-name>GenericDS</jndi-name>
  <connection-url>[jdbc: url for use with Driver class]</connection-
url>
  <driver-class>[fully qualified class name of java.sql.Driver
implementation]</driver-class>
  <user-name>x</user-name>
  <password>y</password>
  <!-- you can include connection properties that will get passed in
the DriverManager.getConnection(props) call-->
  <!-- look at your Driver docs to see what these might be -->
```

```

    <connection-property name="char.encoding">UTF-8</connection-
property>
    <transaction-isolation>TRANSACTION_SERIALIZABLE</transaction-
isolation>

    <!--pooling parameters-->
    <min-pool-size>5</min-pool-size>
    <max-pool-size>100</max-pool-size>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <idle-timeout-minutes>15</idle-timeout-minutes>
    <!-- sql to call when connection is created
<new-connection-sql>some arbitrary sql</new-connection-sql>
-->

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
<check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
-->

    <set-tx-query-timeout></set-tx-query-timeout>
    <query-timeout>300</query-timeout> <!-- maximum of 5 minutes for
queries -->

    <!-- pooling criteria.  USE AT MOST ONE-->
    <!-- If you don't use JAAS login modules or explicit login
getConnection(usr,pw) but rely on user/pw specified above,
don't specify anything here -->

    <!-- If you supply the usr/pw from a JAAS login module -->
    <security-domain>MyRealm</security-domain>

    <!-- if your app supplies the usr/pw explicitly getConnection(usr,
pw) -->
    <application-managed-security></application-managed-security>

    <!--Anonymous depends elements are copied verbatim into the
ConnectionManager mbean config-->
    <depends>myapp.service:service=DoSomethingService</depends>

</local-tx-datasource>

<!-- you can include regular mbean configurations like this one -->
<mbean code="org.jboss.tm.XidFactory"
name="jboss:service=XidFactory">
    <attribute name="Pad">true</attribute>
</mbean>

<!-- Here's an xa example -->
<xa-datasource>
    <jndi-name>GenericXADS</jndi-name>
    <xa-datasource-class>[fully qualified name of class implementing
javax.sql.XADataSource goes here]</xa-datasource-class>
    <xa-datasource-property name="SomeProperty">SomePropertyValue</xa-
datasource-property>

```

```

    <xa-datasource-property
name="SomeOtherProperty">SomeOtherValue</xa-datasource-property>

    <user-name>x</user-name>
    <password>y</password>
    <transaction-isolation>TRANSACTION_SERIALIZABLE</transaction-
isolation>

    <!--pooling parameters-->
    <min-pool-size>5</min-pool-size>
    <max-pool-size>100</max-pool-size>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <idle-timeout-minutes>15</idle-timeout-minutes>
    <!-- sql to call when connection is created
<new-connection-sql>some arbitrary sql</new-connection-sql>
-->

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
<check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
-->

    <!-- pooling criteria.  USE AT MOST ONE-->
    <!-- If you don't use JAAS login modules or explicit login
getConnection(usr,pw) but rely on user/pw specified above,
don't specify anything here -->

    <!-- If you supply the usr/pw from a JAAS login module -->
    <security-domain></security-domain>

    <!-- if your app supplies the usr/pw explicitly getConnection(usr,
pw) -->
    <application-managed-security></application-managed-security>

</xa-datasource>

</datasources>

```

### 15.3.2. リモート使用向けのDataSource の設定

JBoss EAP はリモートのクライアントからDataSource へのアクセスに対応しています。クライアントが JNDI より DataSource をルックアップできるように変更するには、[例15.5「リモート利用向けのDataSource の設定」](#)を参照してください。これは、**use-java-context=false** を指定します。

#### 例15.5 リモート利用向けの DataSource の設定

```
<datasources>
  <local-tx-datasource>
    <jndi-name>GenericDS</jndi-name>
    <use-java-context>false</use-java-context>
    <connection-url>...</connection-url>
    ...
  </local-tx-datasource>
</datasources>
```

これにより、DataSource がデフォルトの `java:/GenericDS` ではなく **GenericDS** という JNDI 名でバインドされるため、EAP サーバーと同じ Virtual Machine へのルックアップが制限されます。



### 注記

実稼働環境では、`<use-java-context>` 設定の利用は推奨されません。接続プールへのリモートアクセスが必要となり、接続がシリアル化可能でないため予期せぬ問題が発生する可能性があります。また、(クラッシュやネットワーク障害など)信頼性に欠ける場合、接続リークの原因となる可能性があるため、トランザクション伝搬はサポートされません。DataSource へのリモートでアクセスするには、リモートセッション Bean ファサードからのアクセスが推奨されます。

## 15.3.3. ログインモジュールを使用するよう DataSource を設定

### 手順15.1 ログインモジュールを使用するよう DataSource を設定

1. データソースについて、`<security-domain-parameter>` を XML ファイルに追加します。

```
<datasources>
  <local-tx-datasource>
    ...
    <security-domain>MyDomain</security-domain>
    ...
  </local-tx-datasource>
</datasources>
```

2. `login-config.xml` ファイルにアプリケーションポリシーを追加します。

認証セクションにはログインモジュールの設定が含まれるようにしなければなりません。例えば、データベースパスワードを暗号化する場合、**SecureIdentityLoginModule** ログインモジュールを使用します。

```
<application-policy name="MyDomain">
  <authentication>
    <login-module
      code="org.jboss.resource.security.SecureIdentityLoginModule"
      flag="required">
      <module-option name="username">scott</module-option>
      <module-option name="password">-170dd0fbd8c13748</module-
option>
      <module-option
        name="managedConnectionFactoryName">jboss.jca:service=LocalTxCM,name=
=OracleDSJAAS</module-option>
      </login-module>
    </authentication>
  </application-policy>
```

3. Web アプリケーションからデータソース接続をフェッチするつもりであれば、Web アプリケーションに対する認証を有効化し、**Subject** が生成されるようにする必要があります。
4. ユーザーが無名で接続する機能が必要な場合、 application-policy に別のログインモジュールを追加しセキュリティ証明書を生成します。
5. **UsersRolesLoginModule** モジュールをチェーンの最初に追加します。**usersProperties** と **rolesProperties** パラメーターをダミーファイルに移動することができます。

```
<login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
  <module-option name="unauthenticatedIdentity">nobody</module-
option>
  <module-option
name="usersProperties">props/users.properties</module-option>
  <module-option
name="rolesProperties">props/roles.properties</module-option>
</login-module>
```



## 第16章 プーリング

### 16.1. 戦略

JBossJCA は **ManagedConnectionPool** を使用してプーリングを実行します。**ManagedConnectionPool** は、選択された戦略やその他のプーリングパラメーターに応じて、サブプールで構成されます。

xml	mbean	内部名	詳細
	ByNothing	OnePool	同等の接続の単一プール。
<application-managed-security/>	ByApplication	PoolByCRI	allocateConnection() より接続プロパティを使用。
<security-domain/>	ByContainer	PoolBySubject	サブジェクト (事前設定のサブジェクトまたは EJB/Web ログインサブジェクト) 毎のプール。
<security-domain-and-application/>	ByContainerAndApplication	PoolBySubjectAndCri	サブジェクト毎と接続プロパティの組み合わせ。



#### 注記

xml 名を見るとセキュリティに関連しているように見えますが、違います。

<security-domain-and-application/> では、サブジェクトは常に CRI の createConnection(user, password) からのユーザーやパスワードより優先されます。

```
(
  ConnectionRequestInfo
)
```

### 16.2. トランザクションスティッキネス

<track-connection-by-tx/> フラグを用いると、プール (またはサブプール) からの同じ接続をトランザクション中に強制的に再利用することができます。



#### 注記

これは「ローカル」トランザクションのみサポートされる動作となります。JBoss Enterprise Application Platform 5 ではデフォルトでトランザクションスティッキネスが有効になっているため、この要素は廃止されました。XA ユーザーは <interleaving/> 要素で明示的にインターリービングを有効にできます。

### 16.3. ORACLE での回避法

Oracle は、JTA トランザクション内部および外部でのXA 接続の使用を好みません。この問題を回避するには、`<no-tx-separate-pools/>` を使用して異なるコンテキストに対して個別のサブプールを作成します。

## 16.4. プールアクセス

プールは同時使用のためにあります。

同時に (またはプールからの接続を使用して) 最大 `<max-pool-size/>` のスレッド数がプール内に存在することができます。

スレッド数がこの制限に達すると、[No Managed Connections Available](#) をスローする前にスレッドはプールを使用するため `<blocking-timeout-seconds/>` の期間待機します。

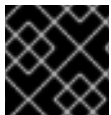
例外をスローする前にプールが接続の取得を再試行するようにするには、`<allocation-retry/>` 要素と `<allocation-retry-wait-millis/>` 要素を使用します。

## 16.5. プールの充填

プール内の接続数はプールの大きさによって制御されます。

- `<min-pool-size/>` - 接続数がこの値を下回ると新しい接続が作成されます。
- `<max-pool-size/>` - この値を越える数の接続は作成されません。
- `<prefill/>` - 4.0.5 に機能要求が実装されました。注意: OnePool? または ByNothing? プーリング基準のみがこの機能をサポートします。

プールの充填は、アプリケーションスレッドをブロックするのではなく、個別の「Pool Filler」スレッドによって行われます。



### 重要

最初の利用では、接続プールを充填は `min-pool-size` までに制限されます。

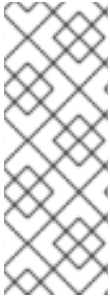
## 16.6. アイドル接続

ピーク期間の終了後に未使用の接続をリープ処理したい場合など、アイドル状態の接続を閉じるよう接続を設定することができます。これには、`<idle-timeout-minutes/>` を使用します。

アイドル状態の確認は、LRU (Least Recently Used) 方式で個別の「Idle Remover」によって実行されます。 `idle-timeout-minutes` の期間使用されていない接続に対し、 `idle-timeout-minutes` を 2 で割った周期で確認が実行されます。

プール自体は MRU (Most Recently Used) 方式で操作します。これにより、余分な接続を簡単に特定することができます。

アイドル状態の接続を閉じたため、プールにある接続の数が `min-pool-size` を下回ってしまった場合、新規の接続が作成されます。



## 注記

長期実行しているトランザクションが存在し、インターリーピングを使用している場合 (track-connection-by-tx を使用しない場合など)、アイドルタイムアウトの値がトランザクションタイムアウトの値より大きくなるようにしてください。インターリーピングを使用している場合、他のユーザーが使用できるよう接続がプールに戻されます。戻された接続が使用されなかった場合、トランザクションがコミットされる前に削除される候補となります。

## 16.7. 停止接続

JDBC プロトコルは接続が切断されていると自然な `connectionErrorOccured()` イベントを提供しません。停止された接続や切断された接続のチェックをサポートするには、複数のプラグインがあります。

### 16.7.1. 有効な接続チェック

最も簡単な形式は「クイック」SQL ステートメントを実行するだけです：

```
<check-valid-connection-sql>select 1 from dual</check-valid-connection-sql>
```

これは、接続をアプリケーションに渡す前に行います。これに失敗した場合、選択できる接続がなくなるまで他の接続が選択され、選択できる接続がなくなった時点で新しい接続が作成されます。

より性能が高いと思われるチェックを実行するにはベンダー固有の機能を使用します。例えば、以下で Oracle または MySQL の `pingDatabase()` を行います。

```
<valid-connection-checker-class-name/>
```

### 16.7.2. SQL クエリ中のエラー

クエリ中に接続が切断されたかをチェックするには、通常の `SQLExceptions` ではなく、`SQLException` のエラーコードやエラーメッセージより `FATAL` エラーを確認します。これらのコードやメッセージはベンダー固有の場合があります。例：

```
<exception-sorter-class-name>org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</exception-sorter-class-name>
```

**FATAL** エラーに関しては、接続が終了します。

### 16.7.3. プールの変更／終了／フラッシュ

- プールの [変更または flush\(\)](#)
- プールを終了するか、アンデプロイすると、最初にフラッシュされます。

### 16.7.4. その他プーリング

[サードパーティのプール](#) - 知識があり理解している場合のみ参照してください。

## 第17章 よくある質問 (FAQ)

### 17.1. ORACLE XA の問題

以下を確認してください。

1. conf/jboss-service.xml の XidFactory が pad=true になっていますか。
2. oracle-xa-ds.xml に <track-connection-by-tx/> がありますか (JBoss Enterprise Application Platform 5.x ではデフォルトで有効にあり、要素が廃止されたため必要ありません)。
3. oracle-xa-ds.xml に <isSameRM-override-value>>false</isSameRM-override-value> がありますか。
4. oracle-xa-ds.xml に <no-tx-separate-pools/> がありますか。
5. jbosscomp-jdbc.xml が使用している Oracle と同じバージョンを指定していますか。
6. 接続する Oracle サーバーに XA がありますか。

XA サポート向けの Oracle データベースを設定すると、Oracle データベースが XA リソースをサポートするよう設定することができます。これにより、JDBC 2.0 対応の Oracle ドライバーを使用することができます。Oracle データベースを XA 向けに初期化するには、次の手順に従ってください。

Oracle JServer がご利用中のデータベースにインストールされているようにしてください。インストールされていない場合は、Oracle データベース設定アシスタントを使用して追加する必要があります。

「Change an Existing DB」を選択し、Oracle JServer を追加したいデータベースを指定します。

「Next」、「Oracle JServer」、「Finish」を順に選択します。以前設定したデータベース設定が Oracle JServer に適していない場合、あるいは十分でない場合は、追加のパラメーターを入力するように求められます。データベース設定ファイル **init.ora** は **\oracle\admin\  
<your\_db\_name>\pfile** ディレクトリにあります。データベース上で **initxa.sql** を実行します。デフォルトでは、このスクリプトファイルは **\oracle\ora81\javavm\install** にあります。ファイル実行中にエラーが発生した場合は、手作業でファイルから SQL ステートメントを実行する必要があります。DBA Studio を使用してパッケージと **JAVA\_XA** という名前のパッケージボディーを **SYS** スキーマに作成し、このパッケージのシノニム (名前は同様に **JAVA\_XA**) を **PUBLIC** スキーマに作成します。

詳しい説明は [Configuring and using XA distributed transactions in WebSphere Studio - Oracle Exception section](#) を参照してください。

## パート III. クラスタリングガイド

## 第18章 概要とクイックスタート

クラスタリングにより、アプリケーションクライアントに単一のビューを提供しつつ、1つのアプリケーションを複数の並列サーバー (クラスターノード) で実行することができるようになります。負荷は複数のサーバーに分散されるため、1つまたは複数のサーバーに障害が発生した場合でも、障害が発生しなかったクラスターノードからアプリケーションにアクセスすることができます。クラスターにノードを追加するだけでパフォーマンスを向上させることができるクラスタリングはスケーラブルな企業向けアプリケーションに欠かせません。また、クラスタリングインフラストラクチャーが高可用性を実現するのに必要な冗長性をサポートするため、クラスタリングは可用性の高い企業アプリケーションを実現するためにも重要です。

**production** サーバードプロファイルの一部として、JBoss Enterprise Application Platform にはそのまま使用できるクラスタリングサポートが含まれています。**production** サーバードプロファイルには次のサポートが含まれています。

- スケーラブルでフォールトトレラントな JNDI 実装 (HA-JNDI)。
- 次を含む Web 層クラスタリング。
  - ステートレプリケーションによる Web セッションステートの高可用性。
  - ハードウェアロードバランサーやソフトウェアロードバランサーとの統合が可能( mod\_jk や JK ベースのソフトウェアロードバランサーとの特殊な統合など)。
  - クラスター全体のシングルサインオンサポート。
- ステートフル Bean およびステートレス Bean に対する、EJB3 と EJB2 両方の EJB セッション Bean クラスタリング
- JPA/Hibernate エンティティに対する分散キャッシュ。
- ノード上で Bean に変更があった時にクラスター全体でキャッシュエントリを無効化し、クラスター全体におけるローカル EJB2 エンティティキャッシュの整合性を維持するフレームワーク。
- 分散 JMS キューと JBoss Messaging によるトピック。
- クラスターの複数のノード上でサービスかアプリケーションをデプロイし、1つのノード上のみでアクティブにすることを HA シングルトンと呼びます。
- **Farm** サービスにて、デプロイされたコンテンツの同期化をクラスターのすべてのノード上で維持します。

本 **クラスタリングガイド** は、JBoss Enterprise Application Platform におけるクラスタリング機能の使用方法を深く理解することを目的としています。最初に、JBoss Enterprise Application Platform のクラスタリングをお試しいただくため、基本的な「クイックスタート」の手順を紹介し、JBoss Enterprise Application Platform クラスタリングの仕組みを理解していただくため、基礎的な情報を提供します。次に、クラスター機能を使用して JEE サービスをクラスターする方法について詳しく説明します。最後に、JGroups と JBoss Cache の上級設定、JBoss Enterprise Application Platform クラスタリングの基礎となる技術について詳しく説明します。

### 18.1. クイックスタートガイド

本項の目的は、JBoss Enterprise Application Platform のクラスタリングを使用するために最低必要な情報を提供することです。本項に記載されている内容の多くは、本項以降で詳しく説明されています。

### 18.1.1. 最初の準備

JBoss Enterprise Application Platform クラスターがサーバーのセットとして動作するよう準備するには、次の手順に従います。

- **JBoss Enterprise Application Platform をすべてのサーバーにインストールします。** JBoss のダウンロードを各サーバー上のファイルシステムに展開するのが最も簡単な方法です。

複数の JBoss Enterprise Application Platform インスタンスを単一のサーバーで実行したい場合は、完全な JBoss ディストリビューションをファイルシステムの複数の場所にインストールするか、**production** サーバードプロファイルのコピーを作成します。例えば、JBoss ディストリビューションのルートが **/var/jboss** に展開された場合、次を行います。

```
$ cd /var/jboss/server
$ cp -r production node1
$ cp -r production node2
```

- **各ノードに対して、ソケットをバインドするアドレスを決定します。** クラスター化の有無に関係なく、JBoss を起動する際に、ソケットがトラフィックをリッスンするアドレスを JBoss に伝える必要があります (デフォルトは **localhost** で、安全ですがクラスター内ではあまり実用的ではありません)。そのため、アドレスを決める必要があります。
- **マルチキャストが動作しているか確認します。** デフォルトでは、JBoss Enterprise Application Platform はほとんどのクラスター内での通信で UDP マルチキャストを使用します。各サーバーのネットワーク設定がマルチキャストをサポートし、サーバー間のスイッチやルーターに対してマルチキャストサポートが有効になっているようにしてください。1つのサーバー用で複数のノードを実行したい場合は、サーバーのルーティングテーブルにマルチキャストルートが含まれているようにしてください。マルチキャストが動作しているかを確認する JGroups の分析ツールの使用方法など、詳細は <http://www.jgroups.org> の JGroups ドキュメントを参照してください。



#### 注記

JBoss Enterprise Application Platform のクラスターリングは UDP マルチキャストの使用を必要としません。クラスター内での通信で TCP ユニキャストを使用するよう再設定することもできます。

- **ノード毎に一意的な整数 "ServerPeerID" を決定します。** これは JBoss Messaging のクラスターリングに必要で、JBoss Messaging を実行しない場合は必要ありません (サーバードプロファイルの **deploy** ディレクトリから JBM を削除する場合など)。JBM を実行する場合、クラスターの各ノードには "ServerPeerID" と呼ばれる固有の整数 ID が必要になります。



#### 重要

簡単な「1, 2, 3, ..., x」のようなネーミングスキームで構いませんが、値は **0** から **1023** の範囲でなければなりません。この範囲外の値になると、ServerPeer 起動サービスで `java.lang.IllegalArgumentException` が発生します。

「JBoss Enterprise Application Platform クラスターの起動」にて、ServerPeerID の使用方法について説明します。

上記が必須の手順となりますが、クラスターをネットワークで実行されている他の JBoss Enterprise Application Platform クラスターから適切に分離するため、次の2つのオプション手順が推奨されます。



- クラスターに対して固有の名前を選択します。JBoss Enterprise Application Platform クラスターのデフォルトの名前は「DefaultPartition」です。ご自分の環境の各クラスターに対して異なる名前を付けてください (「QAPartition」や「BobsDevPartition」など)。名前に「Partition」を使用する必要はありませんが、慣例的に使用されます。クラスターの周囲で送信されるすべてのメッセージに名前が含まれるため、パフォーマンスを考慮して短い名前を付けるようにしてください。選択した名前の使用方法については次の項で取り上げます。
- クラスターに対して固有のマルチキャストアドレスを選択します。デフォルトでは、JBoss Enterprise Application Platform はほとんどのクラスター内の通信に UDP マルチキャストを使用します。実行する各クラスターに対して異なるマルチキャストアドレスを選択してください。一般的に、マルチキャストアドレスには **239.255.x.y** の形式を使用するのがよいでしょう。マルチキャストアドレスの割り当てについては、<http://www.29west.com/docs/THPM/multicast-address-assignment.html> を参照してください。選択したアドレスの使用方法については、次の項で説明します。

クラスターの分離についての詳細は、「[JGroups チャンネルの分離](#)」を参照してください。

### 18.1.2. JBoss Enterprise Application Platform クラスターの起動

JBoss サーバークラスターを開始する一番簡単な方法は、各インスタンスに **-c production** コマンドラインオプションを使用して同じローカルネットワーク上で複数の JBoss インスタンスを開始することです。これらのサーバースタンスは相互に検出を行い、自動的に 1 つのクラスターを形成します。

ここで、いくつかの異なるシナリオを見てみましょう。すべてのシナリオで 2 ノードクラスターを 1 つ作成し、最初のノードは **1**、2 つ目のノードは **2** とします。クラスターの名前を「DocsPartition」とし、**239.255.100.100** をマルチキャストアドレスとして使用するようにします。シナリオは例の提示を目的としています。例として最も単純であるため 2 ノードのクラスターを使用しますが、2 ノードがサイズの的に最適であるとは限りません。

#### ● シナリオ 1: 個別のマシン上のノード

このシナリオは実稼働環境では最も一般的なシナリオです。マシン名は「node1」と「node2」で、node1 の IP アドレスは **192.168.0.101**、node2 の IP アドレスは **192.168.0.102** とします。node1 の ServerPeerID は **1**、node2 の ServerPeerID は **2** とし、各マシンの **/var/jboss** に JBoss がインストールされているとします。

node1 で JBoss を開始します。

```
$ cd /var/jboss/bin
$ ./run.sh -c production -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.101 -Djboss.messaging.ServerPeerID=1
```

node2 でも同様ですが、**-b** の値と ServerPeerID の値が異なります。

```
$ cd /var/jboss/bin
$ ./run.sh -c production -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.102 -Djboss.messaging.ServerPeerID=2
```

**-c** スイッチは、クラスタリングサポートが含まれる **production** 設定の使用を指示します。  
**-g** スイッチは、クラスター名を設定します。**-u** スイッチはクラスター内での通信で使用されるマルチキャストアドレスを設定します。**-b** スイッチはソケットがバインドされるアドレスを設定します。**-D** スイッチは、JBoss Messaging が固有の ID を取得するシステムプロパティ **jboss.messaging.ServerPeerID** を設定します。



## ● シナリオ 2: 単一のマルチホームサーバー上の 2 ノード

同じマシンで複数のノードを実行することは開発環境では一般的で、実稼働環境でもシナリオ 1 を併用して使用されます (すべてのノードを単一マシン上の実稼働クラスタで実行することは、マシン自体が単一障害点となるため通常推奨されません)。このシナリオでは、マシンがマルチホーム化されています (複数の IP アドレスを持っている状態)。これにより、各JBoss インスタンスを異なるアドレスにバインドできるため、ノードがソケットを開ける際の競合を防ぐことができます。

単一のマシンにアドレスとして **192.168.0.101** と **192.168.0.102** が割り当てられ、シナリオ 1 と同様に 2 つの JBoss インスタンスが同じアドレスと ServerPeerIDs を使用するとします。シナリオ 1 との違いは、Enterprise Application Platform インスタンスが独自のワークエリアを持つようにすることです。そのため、**production** 設定を使用せずに、前項で **production** よりコピーした **node1** 設定と **node2** 設定を使用するようにします。

最初のインスタンスを起動するため、コンソールウィンドウを開いて次を実行します。

```
$ cd /var/jboss/bin
$ ./run.sh -c node1 -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.101 -Djboss.messaging.ServerPeerID=1
```

2 つ目のインスタンスも同様ですが、**-b** 値、**-c** 値、ServerPeerID が異なります。

```
$ cd /var/jboss/bin
$ ./run.sh -c node2 -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.102 -Djboss.messaging.ServerPeerID=2
```

## ● シナリオ 3: マルチホーム化されていない単一サーバー上の 2 ノード

シナリオ 2 と似ていますが、マシンの IP アドレスは 1 つのみになります。2 つのプロセスはソケットと同じアドレスやポートにバインドできないため、2 つのインスタンスに異なるポートを使用するよう JBoss に伝える必要があります。これには、**jboss.service.binding.set** システムプロパティを設定して ServiceBindingManager サービスを設定します。

最初のインスタンスを起動するため、コンソールウィンドウを開いて次を実行します。

```
$ cd /var/jboss/bin
$ ./run.sh -c node1 -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.101 -Djboss.messaging.ServerPeerID=1 \
  -Djboss.service.binding.set=ports-default
```

2 つ目のインスタンスには次を実行します。

```
$ cd /var/jboss/bin
$ ./run.sh -c node2 -g DocsPartition -u 239.255.100.100 \
  -b 192.168.0.101 -Djboss.messaging.ServerPeerID=2 \
  -Djboss.service.binding.set=ports-01
```

これは、最初のノード上にある ServiceBindingManager に標準のポートセット (1099 番ポート上の JNDI など) を使用するよう指示します。2 つ目のノードは、標準のポート番号から 100 を引いた番号 (1199 番ポート上の JNDI など) を持つ各ポートのデフォルトである ports-01 を使

用します。ServiceBindingManager 設定の全体については、**conf/bindingservice.beans/META-INF/bindings-jboss-beans.xml** ファイルを参照してください。

この設定は異なるポートを使用し、管理が複雑になるため、実稼働環境への使用は推奨されません。クラスタリングを使用したいのにワークステーションをマルチホーム化できない開発環境では一般的なシナリオとなります。



### 注記

技術的に、**ports-default** はデフォルト値であるため、node1 のコマンドラインに **-Djboss.service.binding.set=ports-default** を含める必要はありませんが、慣れていないユーザーはすべてのサーバーで統一されたコマンドライン引数を使用した方が簡単でしょう。

これで終了です。JBoss Enterprise Application Platform サーバーのクラスタを開始し、実行することができるようになります。

## 18.1.3. Web アプリケーションクラスタリングのクイックスタート

JBoss Enterprise Application Platform は、クラスタにある 1 つ以上のノードに各ユーザーの **HttpSession** ステートのバックアップコピーが保存されるクラスタ化された Web セッションをサポートします。セッションを処理する 1 次ノードに障害が発生あるいは停止した場合、クラスタの別のノードがバックアップコピーにアクセスし、セッションの後続要求を処理することができます。Web 層クラスタリングの詳細は『HTTP コネクター負荷分散ガイド』を参照してください。

Web 層クラスタリングには 2 つの側面があります。

- **外部ロードバランサーの設定。** Web アプリケーションは、JBoss Enterprise Application Platform インスタンスのクラスタ全体で HTTP 要求を負荷分散するため、外部ロードバランサーを必要とします (その理由については、「[外部ロードバランサーアーキテクチャー](#)」を参照)。JBoss Enterprise Application Platform 自体は HTTP ロードバランサーとして機能しないため、ハードウェアロードバランサーかソフトウェアロードバランサーを設定する必要があります。選択できるロードバランサーは多く存在するため、ロードバランサーの設定方法については本クイックスタートの範囲外となります。よく使用される mod\_jk ソフトウェアロードバランサーの設定についての詳細は、『HTTP コネクター負荷分散ガイド』を参照してください。
- **クラスタリングに対する Web アプリケーションの設定。** これには、特定の Web アプリケーションに対する希望のクラスタリング動作を JBoss に伝える必要があります。それには、空の **distributable** 要素をアプリケーションの **web.xml** ファイルに追加します。

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
          version="2.5">

    <distributable/>

</web-app>
```

上記を実行すれば、ほとんどのアプリケーションでデフォルトの JBoss Enterprise Application Platform の Web セッションクラスタリング動作を実行できるはずです。詳細設定オプションについては、『HTTP コネクター負荷分散ガイド』を参照してください。

#### 18.1.4. EJB セッション Bean クラスタリングのクイックスタート

JBoss Enterprise Application Platform は、クラスター全体で Bean の要求が分散されるクラスターされた EJB セッション Bean をサポートします。ステートフル Bean については、Bean ステートのバックアップコピーは 1 つ以上のクラスターノードで維持されるため、特定のセッションを処理しているノードに障害が発生あるいは停止した場合に高可用性を提供できます。EJB2 Bean と EJB3 Bean 両方のクラスタリングがサポートされます。

EJB3 セッション Bean の場合、**org.jboss.ejb3.annotation.Clustered** アノテーションをステートフル Bean またはステートレス Bean の Bean クラスに追加します。

```
@javax.ejb.Stateless
@org.jboss.ejb3.annotation.Clustered
public class MyBean implements MySessionInt {

    public void test() {
        // Do something cool
    }
}
```

EJB2 セッション Bean や、アノテーションでなく XML 設定を使用したい EJB3 Bean の場合、JBoss 固有の配備記述子である **jboss.xml** の Bean のセクションに、**clustered** 要素を追加します。

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>example.StatelessSession</ejb-name>
      <jndi-name>example.StatelessSession</jndi-name>
      <clustered>true</clustered>
    </session>
  </enterprise-beans>
</jboss>
```

詳細の設定オプションについては、[22章 クラスター化されたセッション EJB](#) を参照してください。

#### 18.1.5. エンティティクラスタリングのクイックスタート

JBoss Enterprise Application Platform 5 のクラスタリングで大幅に改善された機能の 1 つが、Hibernate 3.3 で導入された 2 次エンティティキャッシングに対して新しい Hibernate/JBoss Cache 統合を使用することです。JPA/Hibernate コンテキストでは、2 次キャッシュは、トランザクションの範囲外で保持されたコンテンツを持つキャッシュを参照します。データベースの読み取り回数を減らすと、2 次キャッシュのパフォーマンスが改良されることがあります。常に 2 次キャッシュを有効にしてアプリケーションの負荷をテストし、無効にしてから特定のアプリケーションに対して有益であるか確認するようにしてください。

複数の JBoss Enterprise Application Platform インスタンスを使用して JPA/Hibernate アプリケーションを実行し、2 次キャッシュを使用する場合、クラスター対応のキャッシュを使用しなければなりません。クラスター対応のキャッシュを使用しないと、サーバー B 上のアクティビティがエンティティを更新した後でもサーバー A 上のキャッシュは古いデータを保持することになります。

JBoss Enterprise Application Platform は、JBoss Cache を基にしたクラスター対応の 2 次レベルを提

供します。JBoss Enterprise Application Platform の標準の Hibernate ベース JPA プロバイダーが JBoss Cache での 2 次キャッシングを有効にするには、**persistence.xml** を次のように設定します。

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="somename" transaction-type="JTA">
    <jta-data-source>java:/SomeDS</jta-data-source>
    <properties>
      <property name="hibernate.cache.use_second_level_cache"
value="true"/>
      <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
      <property name="hibernate.cache.region.jbc2.cachefactory"
value="java:CacheManager"/>
      <!-- Other configuration options ... -->
    </properties>
  </persistence-unit>
</persistence>
```

これにより、Hibernate が JBoss Cache ベースの 2 次キャッシュを使用するよう指示しますが、キャッシュするエンティティは指定されていません。キャッシュするエンティティを指示するには、**org.hibernate.annotations.Cache** アノテーションをエンティティクラスに追加します。

```
package org.example.entities;

import java.io.Serializable;
import javax.persistence.Entity;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Cache(usage=CacheConcurrencyStrategy.TRANSACTIONAL)
public class Account implements Serializable {
```

詳細設定オプションや、JPA でない Hibernate アプリケーションを同様に設定する方法については、[23章 クラスター化した Entity EJB](#) を参照してください。



## 注記

クラスターリングは JPA/Hibernate の 2 次キャッシュのオーバーヘッドを大幅に増加させます。そのため、2 次キャッシングがクラスター化されていないアプリケーションに有益であってもクラスター化されたアプリケーションにも有益であるとは限りません。クラスター化された 2 次キャッシングが総合的に有益である場合、頻繁に変更されるエンティティタイプはクラスター化されていない場合に有益でもクラスター化されていると有益でないこともあります。常にアプリケーションの負荷をテストするようにしてください。

## 第19章 クラスタリングの概念

次項では、JBoss のクラスタリングサービスの基礎となる基本概念について説明します。この概念を先に理解してから残りのクラスタリングガイドを読むようにするとよいでしょう。

### 19.1. クラスタ定義

クラスタは、お互いにやりとりし、目的を共有する一連のノードです。JBoss Application Server クラスタ (パーティションとも呼ばれます) では、ノードは JBoss Enterprise Application Server インスタンスです。ノード間の通信は、クラスタ内のノードを追跡する中核的な機能を提供し、クラスタメンバー間のメッセージを安定的に交換する JGroups **Channel** とともに JGroups グループ通信ライブラリによって処理されます。同じ設定と名前を持つ JGroups Channel は動的にお互いのノードを検索し、グループを形成することができます。このため、同じネットワークの 2 つの AS インスタンスで「`run -c production`」を実行するだけで、ノードがクラスタを形成するようになります。各 Enterprise Application Platform は同じデフォルト設定で **Channel** (実際には複数) を開始し、その結果、お互いのノードが動的に検索され、クラスタが形成されます。ノードは、他のクラスタメンバーと同じ設定と名前で **Channel** を開始または停止するだけでいつでもクラスタに動的に追加、あるいはクラスタから動的に削除できます。

同じ Enterprise Application Platform インスタンス上で、別サービスが独自の **Channel** を作成することができます。Enterprise Application Platform 5 の **production** サーバードプロファイルを標準起動すると、2 つの異なるサービスが合計 4 つのチャンネルを作成します。JBoss Messaging が 2 つのチャンネルを作成し、HAPartition と呼ばれるコアの汎用クラスタリングサービスも 2 つのチャンネルを作成します。クラスタ化された Web アプリケーションやクラスタ化された EJB3 SFSB、クラスタ化された JPA/Hibernate エンティティキャッシュをデプロイすると、追加のチャンネルが作成されます。Enterprise Application Platform が接続するチャンネルは、大きく 3 つに分類されます (HAPartition サービスが使用する汎用チャンネル、特別用途のキャッシングやクラスタ全体のステートレプリケーション向けに JBoss Cache が作成したチャンネル、JBoss Messaging が使用する 2 つのチャンネル)。

したがって、2 つの Enterprise Application Platform 5.0.x インスタンスで **run -c production** を実行すると、お互いのチャンネルが検索され、概念的な **cluster** が形成されます。これは 2 ノードクラスタと考えると分かりやすいですが、実際には 4 つのチャンネル、したがって 4 つの 2 ノードクラスタであると理解することが重要です。

同じネットワーク上でサービスがクラスタリングしたい異なるセットのサーバーがあるとします。☒ [19.1「クラスタとサーバーノード」](#) は、3 つのセットに分割された JBoss サーバードプロファイルのネットワーク例です。3 番目のセットにはノードが 1 つしかありません。このようなトポロジは、ノードのセット内でチャンネル設定と名前が一致するようクラスタを形成するようにして設定します。この場合、チャンネル設定と名前が他のチャンネル設定と名前のマッチと異なるようにし、同じネットワークの別のチャンネルとも異なるようにします。Enterprise Application Platform はこの設定をなるべく簡単にします。クラスタ化するサーバーは、同じ値をコマンドラインの **-g** (パーティション名) と **-u** (マルチキャストアドレス) 起動スイッチに渡すことだけが必要になります。各サーバーセットには異なる値を選択する必要があります。「JGroups 設定」と「JGroups チャンネルの分離」の項に、検索したいピアのみを見つけるよう Enterprise Application Platform を設定する方法が説明されています。



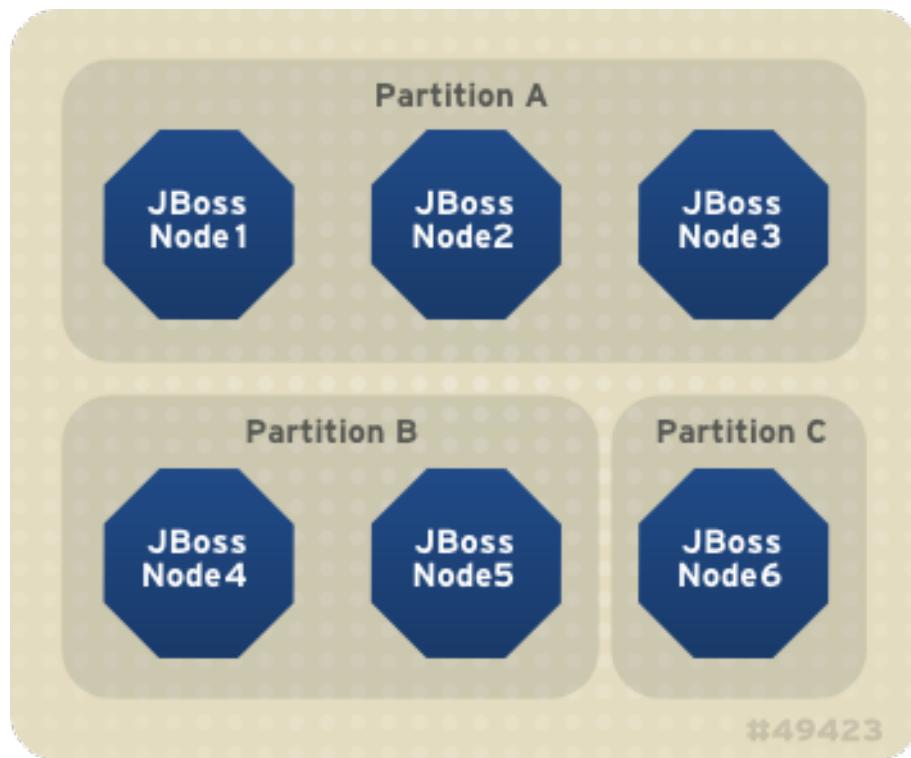


図19.1 クラスターとサーバーノード

## 19.2. サービスアーキテクチャー

各ノードで JGroups 設定によって定義されるクラスタリングのトポグラフィはシステム管理者にとって非常に重要となります。アプリケーション開発者の場合は、クライアントアプリケーションの観点から見るクラスターアーキテクチャーの方が関心が高いと思います。JBoss Enterprise Application Platform ではクライアント側インターセプター (スマートプロキシまたはスタブとも呼ばれます) と外部ロードバランサーの 2 つの基本クラスタリングアーキテクチャーが使用されています。アプリケーションが使用するアーキテクチャーは、クライアントのタイプによって異なります。

### 19.2.1. クライアント側インターセプターアーキテクチャー

JNDI、EJB、JMS、RMI、JBoss Remoting などの JBoss Enterprise Application Platform によって提供されるリモートサービスの多くは、クライアントによるリモートプロキシオブジェクトの取得 (ルックアップやダウンロードなど) を必要とします。プロキシオブジェクトはサーバーによって生成され、サービスのビジネスインターフェースを実装します。次にクライアントはプロキシオブジェクトに対してローカルメソッド呼び出しを行います。プロキシは自動的に呼び出しをネットワーク全体にルーティングし、サーバー内で管理されるサービスオブジェクトに対して呼び出されます。プロキシオブジェクトは、適切なサーバーノードやマーシャル呼び出しパラメーター、アンマーシャル呼び出し結果を検索する方法を見だし、結果を呼び出し側クライアントに返します。クラスター化された環境では、サーバーによって生成されたプロキシオブジェクトには、クラスター内の複数のノードに呼び出しをルーティングする方法を認識するインターセプターが含まれています。

プロキシのクラスタリング論理はクラスターに関する最新のナレッジに対応しています。例えば、使用可能な全サーバーノードの IP アドレス、ノード全体に負荷を分散させるアルゴリズム (次項を参照)、目的のノードが使用できない場合に要求をフェイルオーバーする方法などを認識しています。各サービス要求を処理する際にクラスタトポロジが変更されると、サーバーノードはクラスター内の最終変更にてプロキシを更新します。例えば、クラスターからノードがドロップアウトすると、次にクラスタのアクティブなノードに接続した時に各プロキシが新しいトポロジで更新されます。プロキシのクラスタリング論理で行われる操作はすべてクライアントアプリケーションに対して透過的となります。[図19.2「クラスタリング用クライアント側インターセプター \(プロキシ\) アーキテクチャー」](#) は、クライアント側のインターセプタークラスタリングアーキテクチャーになります。

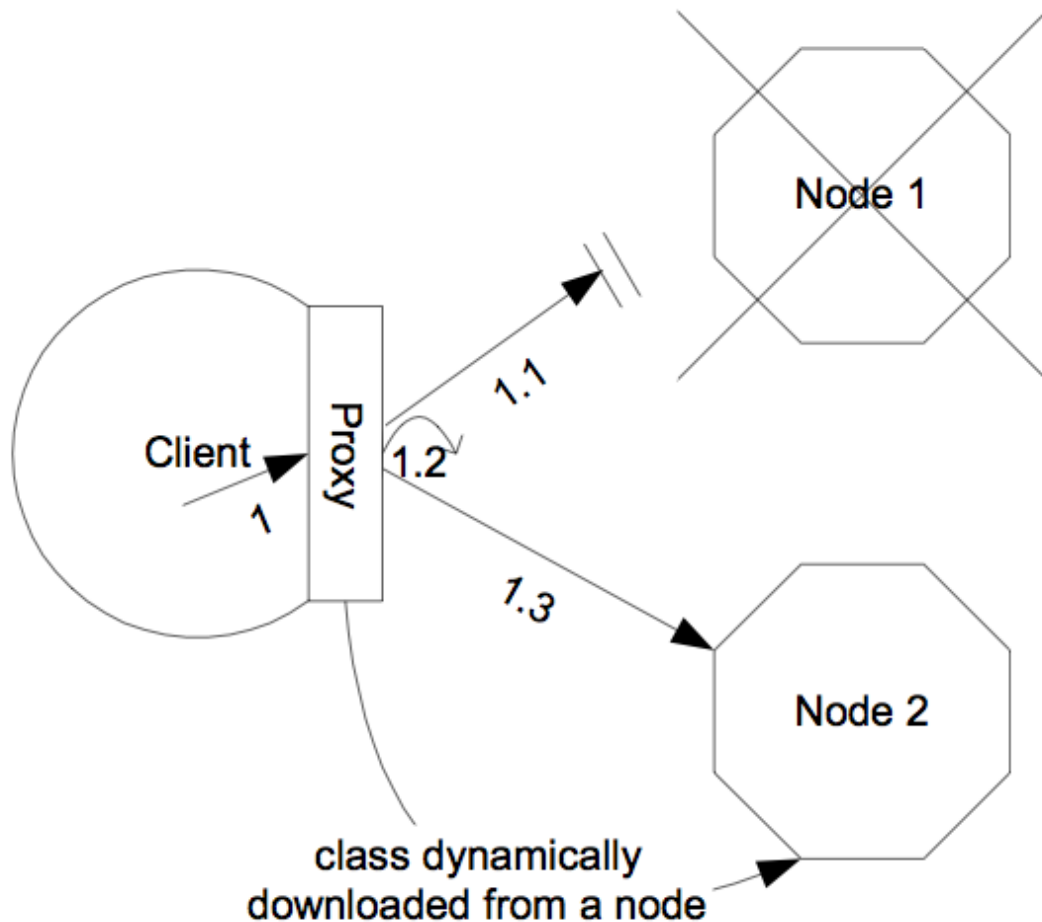


図19.2 クラスターリング用クライアント側インターセプター (プロキシ) アーキテクチャー

### 19.2.2. 外部ロードバランサーアーキテクチャー

HTTP ベースの JBoss サービスはクライアントによるダウンロードを必要としません。クライアント (Web ブラウザーなど) は、HTTP プロトコルを使用して直接要求を送信、あるいは 応答を受信します。この場合、外部ロードバランサーがすべての要求を処理し、クラスター内複数のノードに要求を送信する必要があります。クライアントはロードバランサーへの接続方法のみ認識している必要があります。クライアントはロードバランサーへのコンタクト方法のみ理解する必要があり、ロードバランサー背後の JBoss Enterprise Application Platform インスタンスについては認識しません。ロードバランサーは論理的にはクラスターの一部ですが、クライアントや JBoss Enterprise Application Platform インスタンスと同じプロセスで実行されていないため、「外部」と呼ばれます。ロードバランサーはソフトウェアかハードウェアに実装できます。ハードウェアロードバランサーのベンダーは数多く存在します。mod\_jk Apache モジュールはソフトウェアロードバランサーの良い例です。外部ロードバランサーはクラスター設定を理解する独自のメカニズムを実装し、独自の負荷分散ポリシーやフェイルオーバーポリシーを提供します。図19.3「クラスターリング用外部ロードバランサーアーキテクチャー」は外部ロードバランサーのクラスターリングアーキテクチャーになります。

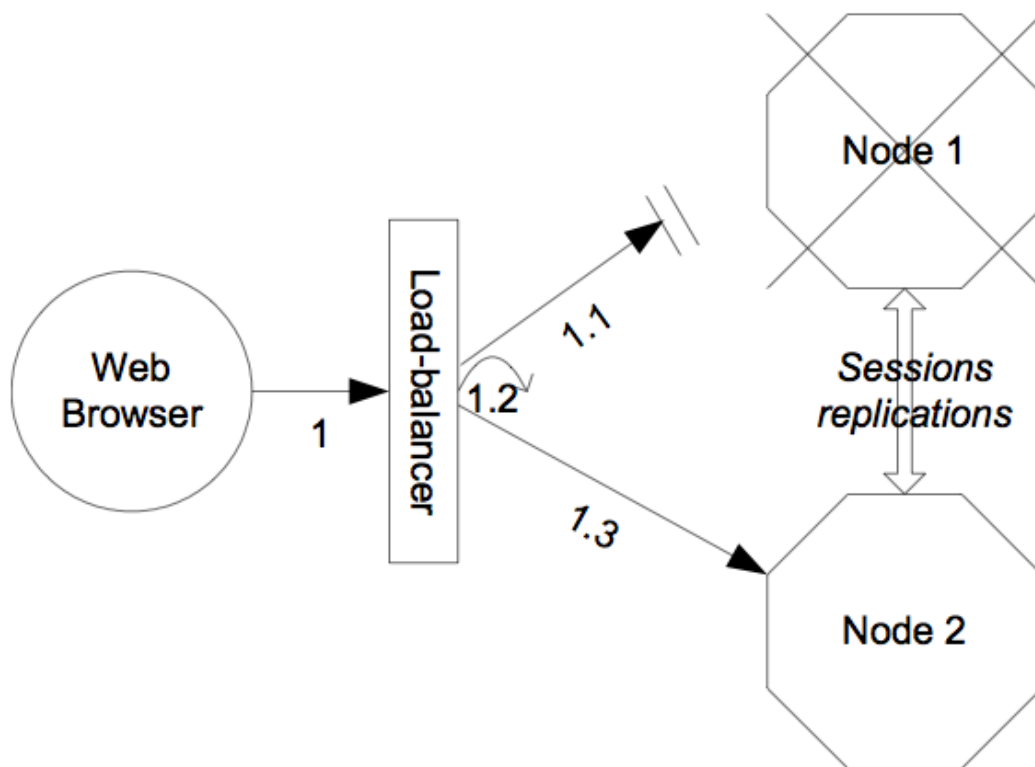


図19.3 クラスタリング用外部ロードバランサーアーキテクチャ

外部ロードバランサーに伴う問題は、ロードバランサー自体が単一点障害になり得ることです。クラスターサービス全体が高可用性を維持するよう注意深く監視する必要があります。

## 19.3. 負荷分散ポリシー

JBoss クライアント側インターセプター (スタブ) とロードバランサーはいずれも負荷分散ポリシーを使用して新しい要求を送信するサーバーノードを確定します。この項では JBoss Enterprise Application Platform 内で使用可能な負荷分散ポリシーについて説明します。

### 19.3.1. クライアント側インターセプターアーキテクチャ

JBoss Enterprise Application Platform 5 では、クライアント側インターセプターアーキテクチャが使用されている時に次の負荷分散オプションを使用できます。クライアント側スタブは目的のサービスを提供するすべてのノードの一覧を保持します。各要求に対してこの一覧からノードを選択することが負荷分散ポリシーの役割です。各ポリシーは 2 つの実装クラスを持っています。1 つは、レガシー分離呼び出しアーキテクチャを使用する EJB2 のようなレガシーサービスが使用するための実装クラスです。もう 1 つは AOP ベースの呼び出しを使用する EJB3 のようなサービスのための実装クラスになります。

- Round-Robin: 各呼び出しは、ノードの一覧の順に新しいノードへ配信されます。最初のターゲットノードは一覧より無作為に選択されます。  
`org.jboss.ha.framework.interfaces.RoundRobin` (レガシー) と  
`org.jboss.ha.client.loadbalance.RoundRobin` (EJB3) によって実装されます。
- Random-Robin: 各呼び出しに対して一覧よりターゲットノードを無作為に選択します。  
`org.jboss.ha.framework.interfaces.RandomRobin` (レガシー) と  
`org.jboss.ha.client.loadbalance.RandomRobin` (EJB3) によって実装されます。
- First Available: 使用可能なターゲットノードの 1 つがメインターゲットとして選択されてから



全てのコールに使用されます。選択されるメンバーはクラスター内のメンバーの一覧から無作為に選択されます。ターゲットノードの一覧が変更されると (1 ノードが起動または停止したため)、現在選択されているノードが使用可能である場合を除き、ポリシーによって新しいターゲットノードが選択されます。各クライアント側プロキシは、他のプロキシに関係なく独自のターゲットノードを選択します。そのため、特定のクライアントが同じターゲットサービス (EJB など) に対して 2 つのプロキシをダウンロードする場合、各プロキシは独立してターゲットを選択します。これが「セッションアフィニティ」または「スティッキーセッション」を提供するポリシーの例となります (ターゲットノードは確立されると変更されないため)。

**org.jboss.ha.framework.interfaces.FirstAvailable** (レガシー) や

**org.jboss.ha.client.loadbalance.aop.FirstAvailable** (EJB3) によって実装されます。

- First Available Identical All Proxies: 「First Available」ポリシーと同じ動作になりますが、選択されるターゲットノードは同じターゲットサービスに関連付けられた同じクライアント側 VM のすべてのプロキシで共有されます。そのため、特定のクライアントが同じターゲットサービス (EJB など) に対して 2 つのプロキシをダウンロードする場合、各プロキシは同じターゲットを使用します。

**org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies** (レガシー) や

**org.jboss.ha.client.loadbalance.aop.FirstAvailableIdenticalAllProxies** (EJB3) によって実装されます。

上記のいずれもが **org.jboss.ha.framework.interfaces.LoadBalancePolicy** インターフェースを実装します。ユーザーは特殊な動作を必要とする場合にこの単純なインターフェースの独自の実装を自由に作成できます。後半のセクションでは、さまざまなサービスによって使用される負荷分散ポリシーの設定方法について説明します。

### 19.3.2. 外部ロードバランサーアーキテクチャー

JBoss Enterprise Application Platform 5 では新しく "TransactionSticky" 負荷分散ポリシーが導入されました。これらのポリシーは、トランザクションの範囲内で発生するすべての呼び出しを同じノードにルーティングする (ノードが存在する場合) など、新しい動作を標準のポリシーに追加します。レガシー分離呼び出しアーキテクチャーがベースであるため、EJB3 のような AOP ベースのサービスでは使用できません。

- Transaction-Sticky Round-Robin: Round-Robin のトランザクションスティッキー変異です。**org.jboss.ha.framework.interfaces.TransactionStickyRoundRobin** によって実装されます。
- Transaction-Sticky Random-Robin: Random-Robin のトランザクションスティッキー変異です。**org.jboss.ha.framework.interfaces.TransactionStickyRandomRobin** によって実装されます。
- Transaction-Sticky First Available: First Available のトランザクションスティッキー変異です。**org.jboss.ha.framework.interfaces.TransactionStickyFirstAvailable** によって実装されます。
- Transaction-Sticky First Available Identical All Proxies: First Available Identical All Proxies のトランザクションスティッキー変異です。**org.jboss.ha.framework.interfaces.TransactionStickyFirstAvailableIdenticalAllProxies** によって実装されます。

上記はすべて簡単なインターフェースの実装となります。ユーザーは特殊な動作を必要とする場合に独自の実装を自由に作成することができます。後半の項では、さまざまなサービスによって使用される負荷分散ポリシーの設定方法について説明します。

## 第20章 ビルディングブロックのクラスター化

JBoss Enterprise Application Platform のクラスターリング機能は、コア機能の多くを提供するローレベルライブラリの上にビルドされています。図20.1「JBoss Enterprise Application Platform のクラスターリングアーキテクチャー」に主な機能が記載されています。

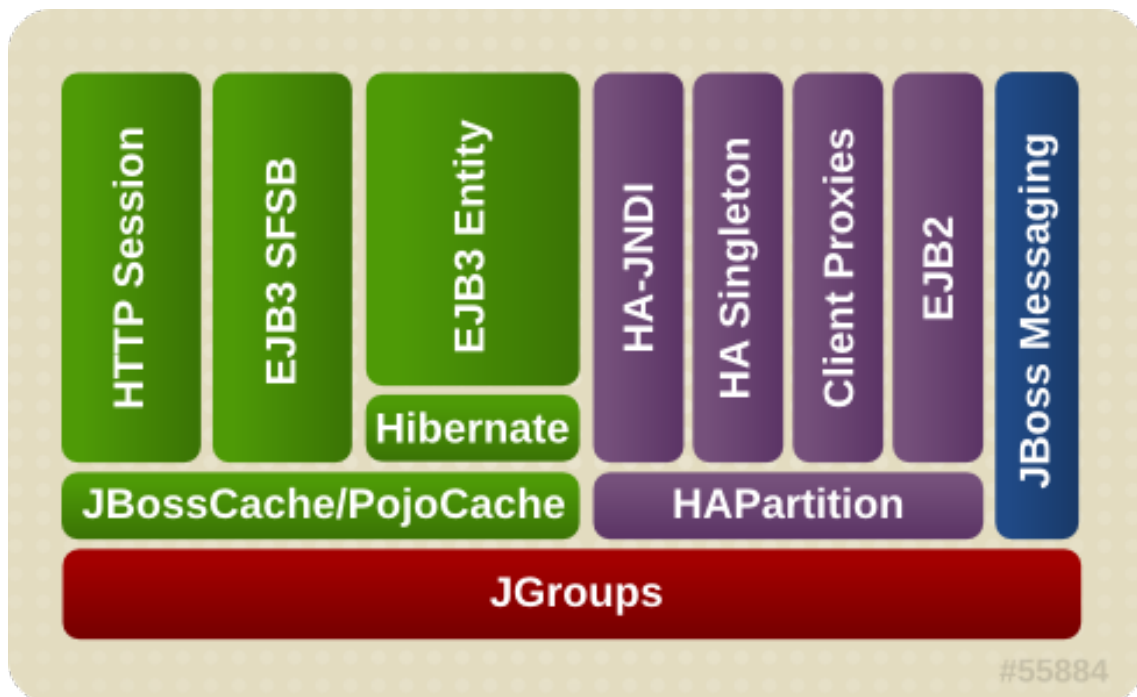


図20.1 JBoss Enterprise Application Platform のクラスターリングアーキテクチャー

**JGroups** は、信頼できるポイントツーポイント通信やポイントツーマルチポイント通信向けのツールキットです。JGroups は JBoss Enterprise Application Platform クラスターにおけるノード間のクラスター関連の通信に使用されます。

**JBoss Cache** は、大変柔軟なクラスター化されたトランザクションキャッシングライブラリです。Enterprise Application Platform のクラスターリングサービスの多くが、メモリ内にステートをキャッシュする必要があります。この際、再作成 (Web セッションの内容など) できない場合にステートのバックアップコピーが別のノードで使えるよう、高可用性を維持し、クラスターの各ノードにキャッシュされたデータの一貫性を維持する必要があります。JBoss Cache は、JBoss Enterprise Application Platform のクラスター化サービスのこのような懸念に対応します。JBoss Cache は JGroups を使用してグループ通信の要件に対応します。**POJO Cache** は JBoss Cache のコアの延長で、クラスターされた Web セッションステートのきめ細かいレプリケーションをサポートするため JBoss Enterprise Application Platform によって使用されます。JBoss Enterprise Application Platform が JBoss Cache や POJO Cache をどのように使用するかについては、「[JBoss Cache による分散キャッシング](#)」を参照してください。

**HAPartition** は JGroups の上部にあるアダプターで、複数のサービスが JGroups チャネルを使用できるようにします。HAPartition は、クラスターメンバ上で実行されている HAPartition ベースのサービスに対する分散レジストリをサポートします。クラスターメンバーシップやクラスターされたサービスのレジストリが変更した時に、リスナーに通知を行います。HAPartition の詳細は「[HAPartition サービス](#)」を参照してください。

他の高レベルクラスターリングサービスは JBoss Cache か HAPartition を使用します。HA-JNDI は JBoss Cache と HAPartition の両方を使用します。例外は JBoss Messaging のクラスターリング機能で、JGroups と直接対話します。

### 20.1. JGROUPS とのグループ通信

JGroups は、JBoss Enterprise Application Platform クラスタに対する基礎のグループ通信サポートを提供します。ピアとグループ通信する必要がある JBoss Enterprise Application Platform は JGroups **Channel** を取得し、通信するために使用します。 **Channel** は、どのノードがグループのメンバーであるかを管理し、ノードの障害を検出して、すべてのグループメンバに対してメッセージを失わないよう先着 (FIFO) 順にメッセージを発信します。また、フロー制御を提供し、低速のメッセージ受信者が高速のメッセージ送信者によって圧倒されないようにします。

JGroups **Channel** の特徴は、JGroups **Channel** を構成する **プロトコル** のセットによって決まります。各プロトコルはグループ通信タスク全体の 1 つのアスペクトを処理します。例えば、**UDP** プロトコルは UDP データグラムの送受信の詳細に対応します。**UDP** プロトコルを使用する **Channel** は UDP ユニキャストとマルチキャストと通信することができます。**TCP** プロトコルを使用する **Channel** は、すべてのメッセージに対して TCP ユニキャストを使用することができます。JGroups は多くのプロトコルをサポートしていますが (詳細は「[JGroups チャネルのプロトコルスタックを設定](#)」を参照)、Enterprise Application Platform には多くのニーズに対応するチャネル設定のデフォルトセットが同梱されています。

デフォルトでは、Enterprise Application Platform の JGroups チャネルがすべて UDP マルチキャストを利用します (TCP ベースの JBoss Messaging チャネルは例外)。サーバーに対するデフォルトのマルチキャストタイプを変更するには、**\$JBoss\_HOME/bin/** で **run.conf** を作成し、そのファイルを開き **JAVA\_OPTS="\$JAVA\_OPTS -Djboss.default.jgroups.stack=<METHOD>"** を追加します。

### 20.1.1. チャネルファクトリサービス

JBoss Enterprise Application Platform 5 と以前のリリースとの大きな違いは、クラスタリングサービスが必要とする JGroups Channel (分散 HttpSession キャッシュによって使用されるチャネルなど) が、コンシュームするサービスの設定の一部として詳細に設定にされなくなり、コンシュームするサービスによって直接インスタンス化されなくなったことです。その代わりに、名前付きチャネル設定のレジストリおよび **Channel** インスタンスのファクトリとして、新しい **ChannelFactory** サービスが使用されるようになりました。チャネルが必要なサービスは **ChannelFactory** よりチャネルをリクエストし、希望する設定の名前を渡します。

ChannelFactory サービスは **server/production/deploy/cluster/jgroups-channelfactory.sar** にデプロイされます。ChannelFactory サービスは起動時に名前 (UDP や TCP など) によって識別される様々な標準 JGroups 設定が含まれる **server/production/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml** ファイルを構文解析します。チャネルが必要なサービスはチャネルファクトリにアクセスし、特定の名前付き設定を持つチャネルを要求します。

#### 注記

複数のサービスが ChannelFactory より同じ設定名を持つチャネルをリクエストした場合、同じ基礎のチャネルへの参照はこれらのサービスには提供されません。各サービスへ独自のチャネルが提供されますが、チャネルの設定はすべて同じになります。この場合、これらのチャネルが同じマルチキャストアドレスやポートを使用する場合など、チャネル同士がグループを形成しないようにする方法が論理的な問題となります。そのため、コンシュームするサービスがチャネルに接続し、サービスに固有の **cluster\_name** 引数を **Channel.connect(String cluster\_name)** メソッドに渡します。ネットワーク上で受信したメッセージが意図されたものであるかを判断する要素の 1 つとして、チャネルは **cluster\_name** を使用します。

#### 20.1.1.1. 標準のプロトコルスタック設定

Enterprise Application Platform 5 に同梱される標準のプロトコルスタック設定は以下のようになります。すべての設定が実際に使用されるわけではありません。同梱される多くはデフォルトのサーバープ

ロファイルを変更したいユーザー向けの設定となります。変更されていない本来の Enterprise Application Platform 5 の **production** サーバプロファイルで実際に使用される設定は、**udp**、**jbm-control**、**jbm-data** になります。JBoss Messaging 以外のクラスタリングサービスは **udp** を使用します。

新しいスタック設定を追加するには、新しい **stack** 要素を **server/production/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml** ファイルに追加します。このファイルを編集すると、既存設定の動作を変更できます。追加や変更を行う前に、Enterprise Application Platform に同梱される標準設定を確認してみてください。ニーズに合った標準設定があるかもしれません。また、設定を編集する前にどのサービスがこの設定を利用しているか把握し、影響を受ける全サービスに対して、変更したい設定が適切であるよう確認します。特定のサービスに対して変更が不適切な場合、新しい設定を作成して一部のサービスが新しい設定を使用するよう変更した方がよいかもしれません。

- **udp**

UDP マルチキャストベースのスタックは、異なるチャンネルによって共有されることを目的としています。同期グループ RPC の待ち時間が長くなるため、メッセージのバンドルは無効になっています。非同期 RPC のみを大量に呼び出すサービス (REPL\_ASYNC 向けに設定された JBoss Cache など) の場合、次のようにキャッシュが **udp-async** スタックを使用するよう設定するとパフォーマンスを改善できるでしょう。同期 RPC のみを呼び出すサービス (REPL\_SYNC や INVALIDATION\_SYNC 向けに設定された JBoss Cache など) の場合、フロー制御が含まれない次の **udp-sync** スタックを使用するとパフォーマンスを改善できるでしょう。

- **udp-async**

前述のデフォルトの **udp** スタックと同じですが、トランスポートプロトコルでメッセージバンドルが有効になっています (**enable\_bundling=true**)。メッセージのバインドによりパフォーマンスが改善できる大量の非同期 RPC (REPL\_ASYNC 向けに設定された大量の JBoss Cache インスタンスなど) を呼び出すサービスに対して有効です。

- **udp-sync**

フロー制御やメッセージバンドルがない UDP マルチキャストベースのスタックです。同期呼び出しが使用され、メッセージの量 (メッセージ比率とサイズ) が大きくない場合、**udp** の代わりに使用することができます。持続率が高い状態でメッセージを送信する場合、メモリが不足することがあるため、この設定は使用しないでください。

- **tcp**

フロー制御とメッセージバンドルを持つ TCP ベースのスタックです。TCP スタックは通常、ネットワークで IP マルチキャストが使用できない場合に使用します (ルーターがマルチキャストを破棄する場合など)。

- **tcp-sync**

フロー制御やメッセージバインドがない TCP ベースのスタックです。TCP スタックは通常、ネットワークで IP マルチキャストが使用できない場合に使用します (ルーターがマルチキャストを破棄する場合など)。同期呼び出しが使用され、メッセージの量 (メッセージ比率とサイズ) が大きくない場合、前述の **tcp** の代わりにこの設定を使用するようにしてください。高い持続率でメッセージを送信する場合、メモリが不足することがあるため、この設定は使用しないでください。

- **jbm-control**

JBoss Messaging の制御チャネル向けに最適化されたスタックです。デフォルトでは、前述の **udp** スタックがデフォルトで使用する UDP トランスポートプロトコル設定と同じ設定が使用されます。これにより、JBoss Messaging の制御チャネルが、他の標準 JBoss Enterprise Application Platform のクラスタされたサービスが使用するソケットやネットワークバッファ、スレッドプールを使用できるようになります (「[JGroups 共有トランスポート](#)」を参照してください)。

- **jbm-data**

JBoss Messaging データチャネル向けに最適化された TCP ベースのスタックです。

### 20.1.1.2. プロトコルスタック設定の変更

デフォルトでは、JBoss Messaging 以外のクラスタリングサービスはすべて **udp** プロトコルスタック設定を利用します。TCPベースの設定を利用したい場合、system property `jboss.default.jgroups.stack` を **tcp** の値 (**-Djboss.default.jgroups.stack=tcp**) に設定してください。この変更により、JGroups チャネルを使うサービスのほとんどをTCPベースの設定に構成します。**tcp** をデフォルトのプロトコルスタックにするには、Linux プラットフォームでは `$JBOSS_HOME/bin/run.conf` にある `JAVA_OPTS` の環境変数に、また Windows プラットフォームでは `$JBOSS_HOME/bin/run.conf.bat` にある環境変数をこのシステムプロパティを追加します。

**tcp** スタックは、ピア検出に (MPING層経由で) UDP マルチキャストを使います。こうすることで、スタックが環境固有のホスト設定を避け、カスタマイズなしに使うことができます。UDP マルチキャストを利用できない場合、非UDPベースのピア検出層 (TCPPING層) に変更し、必要と考えられるクラスターノードのアドレス/ポートを設定する必要があります。**jgroups-channelfactory-stacks.xml** からプロトコルスタック設定を変更できます。このファイルには、両ピア検出層の定義を含みます。デフォルトでは、MPING層の定義はアンコメントされ、TCPPING層はコメントされています。非UDPベースのピア検出に切り替えるには、MPING層をコメント、TCPPING層をアンコメントし、設定します。MPINGおよびTCPPINGの詳細情報は、「[ディスカバリプロトコル](#)」を参照してください。

### 20.1.1.3. JBoss Messagingのプロトコルスタック設定

JBoss Messaging はデフォルトで **jbm-control** および **jbm-data** プロトコルスタック設定を利用します。**jbm-control** プロトコルスタックは、完全にUDPベースとなっており、**jbm-data** はUDPマルチキャストを採用するMPING検出プロトコルを使います。そのため、JBoss Messaging でTCPベースの設定のみを使いたい場合は、JBoss Messaging コントロールチャネルを **jbm-control** スタックの代わりに **tcp** プロトコルスタックを使用するように設定し、**jbm-data** プロトコルスタックがMPING層の代わりにTCPPING層を利用するように変更しなければなりません。

JBoss Messaging コントロールチャネルを **tcp** プロトコルスタックを使うように設定するには、**deploy/messaging/RDMS-persistence-service.xml** ファイルを開き (*RDMS* の値はメッセージの永続化に利用している関係データベース管理システムにより変わります)、`org.jboss.messaging.core.jmx.MessagingPostOfficeService` mbean の **ControlChannelName** 属性値を **tcp** に変更します。

```
<!--<attribute name="ControlChannelName">jbm-control</attribute>-->
<attribute name="ControlChannelName">tcp</attribute>
```

MPING層ではなくTCPPING層を使うように **jbm-data** プロトコルスタック定義を変更するには、**/server/PROFILE/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml** を開き、[例20.1「TCPPING 定義を持つjbm-data プロトコルスタックの定義」](#) に示されているようにMPING層を同等のTCPPING層に置き換えます。

**例20.1 TCPPING 定義を持つjbm-data プロトコルスタックの定義**



```

<!--
<MPING timeout="3000"
    num_initial_members="3"
    mcast_addr="${jboss.jgroups.tcp.mping_mcast_addr:230.11.11.11}"
    mcast_port="${jgroups.tcp.mping_mcast_port:45700}"
    ip_ttl="${jgroups.udp.ip_ttl:2}"/>

-->
<TCPPING timeout="5000"

initial_hosts="${jbm.data.tcpping.initial_hosts:localhost[7900],localhos
t[7901]}"
    port_range="1"
    num_initial_members="3"/>

```

定義済みポートが他のTCPPING層で使われているポートと矛盾しないようにしてください。システムプロパティ-Djbm.data.tcpping.initial\_hosts を使い、JAVA\_OPTSからこの層の初期ホストを設定することができます。

### 20.1.2. JGroups 共有トランスポート

Enterprise Application Platform 上で実行される JGroups ベースのクラスタリングサービスの数が年を追って増加したため、これらのチャネルが使用するリソース (特にソケットとスレッド) を共有する必要性が問題となりました。Enterprise Application Platform 5 の **production** 設定は、起動時に 4 つの JGroups チャネルに接続します。分散可能な Web アプリケーション、クラスターされた EJB3 SFSB、クラスターされた JPA/Hibernate 2 次キャッシュがすべて使用されると合計で 7 つまたは 8 つの JGroups チャネルが接続されます。チャネルの数が多くリソースが大幅に消費されることがあるため、ネットワーク環境が設定にクラスター分離を必要とする場合、設定が大変困難になります。

Enterprise Application Platform 5 より、JGroups がチャネル間のトランスポートプロトコルインスタンスの共有をサポートするようになりました。JGroups チャネルは、個別のプロトコルのスタックにより構成され、各プロトコルはチャネル動作の 1 つのアスペクトを担当しています。トランスポートプロトコルは、実際にメッセージをネットワーク上で送信、あるいはネットワークより受信するプロトコルです。共有が望ましいリソース (ソケットとスレッドプール) はトランスポートプロトコルにより管理されているため、チャネル間でトランスポートプロトコルを共有すると、JGroups のリソースを効率的に共有することができます。

トランスポートプロトコルの共有を設定するには、**singleton\_name="someName"** 属性をプロトコルの設定に追加します。トランスポートプロトコル設定が同じ **singleton\_name** 値を使用するすべてのチャネルがトランスポートを共有します。スタック内の他のプロトコルは共有されません。図 20.2 「共有トランスポートを使用するサービス」では、仮想マシンで 4 つのサービスが実行され、各サービスが独自のチャネルを持っています。3 つのサービスがトランスポートを共有し、1 つのサービスは独自のトランスポートを使用しています。

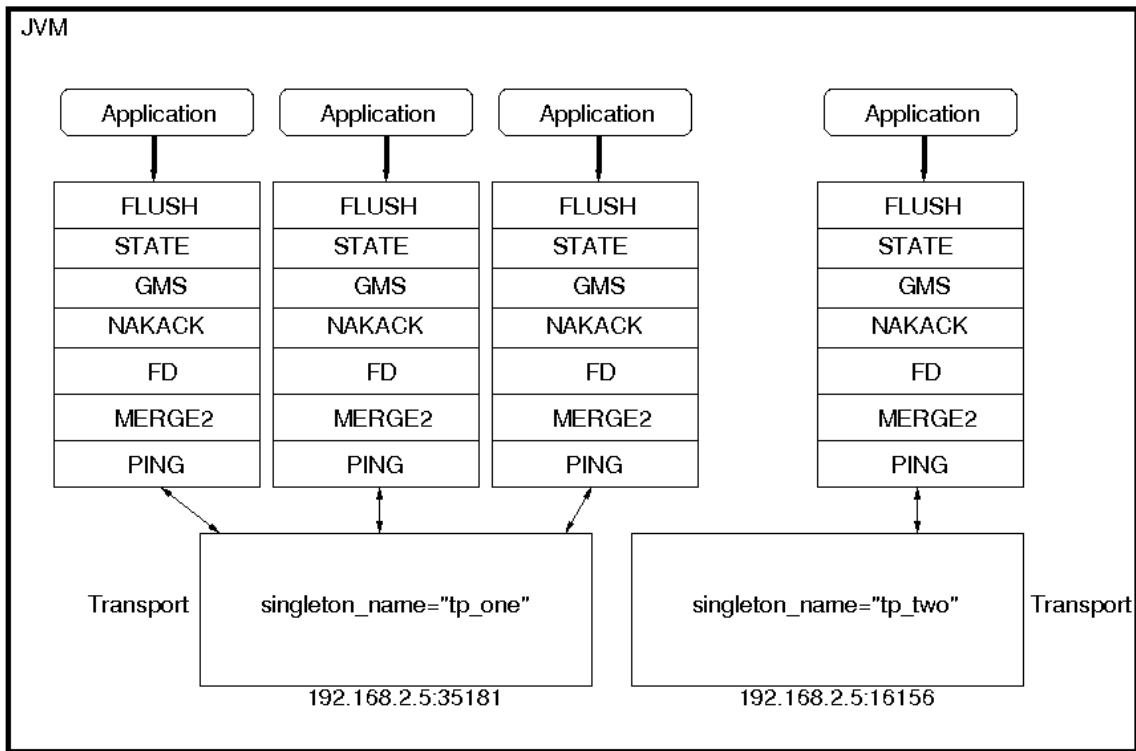


図20.2 共有トランスポートを使用するサービス

Enterprise Application Platform 5 の ChannelFactory が使用するプロトコルスタック設定にはすべて **singleton\_name** が設定されています。スタックのチャンネルを作成する前に **singleton\_name** が含まれていない ChannelFactory にスタックを追加すると、「unnamed\_customStack」のように「unnamed\_」ストリングにスタック名を連結して ChannelFactory が合成的に **singleton\_name** を作成します。

## 20.2. JBOSS CACHE による分散キャッシング

JBoss Cache は、アプリケーションサーバー環境やスタンドアローンで使えるフル機能の分散キャッシュフレームワークです。JBoss Cache は、以下を含む JBoss Enterprise Application Platform クラスターの多くの標準クラスター化サービスが使用する基礎の分散キャッシングサポートを提供します。

- クラスター化された Web アプリケーションセッションのレプリケーション
- クラスター化された EJB3 ステートフルセッション Bean のレプリケーション
- JPA および Hibernate エンティティのクラスター化されたキャッシング
- クラスター化されたシングルサインオン
- HA-JNDI レプリケートツリー
- DistributedStateService

アプリケーションのカスタム使用のために独自の JBoss Cache インスタンスや POJO Cache インスタンスを作成することもできます。詳細は、[28章 JBoss Cache の設定とデプロイメント](#) を参照してください。

### 20.2.1. JBoss Enterprise Application Platform の CacheManager サービス

JBoss Enterprise Application Platform の標準のクラスター化されたサービスの多くは JBoss Cache を

使用してクラスタ全体で一貫的なステートを維持します。異なるサービス (Web セッションクラスタリングや JPA/Hibernate エンティティの 2 次レベルキャッシングなど) は異なる JBoss Cache インスタンスを使用し、各キャッシュは使用するサービスに合わせて設定されます。Enterprise Application Platform 4 では、各キャッシュが個別に **deploy/** ディレクトリへデプロイされたため、次のような欠点がありました。

- エンドユーザーが必要ないキャッシュもデプロイされ、各キャッシュが JGroup チャネルを作成しました。例えば、クラスター化された EJB3 SFSB がなくても、保存されるキャッシュが開始されました。
- キャッシュはキャッシュを使用するサービスの内部的な詳細であるため、ファーストクラスのデプロイメントとすべきではありません。
- サービスは JMX ルックアップよりキャッシュを検索するのですが、管理インターフェースを公開する目的以外で JMX を使用することは、JBoss Enterprise Application Platform 5 の方針に反します。

JBoss Enterprise Application Platform 5 では、散在するキャッシュデプロイメントが、**\$JBoss\_HOME/server/production/deploy/cluster/jboss-cache-manager.sar** よりデプロイされる新しい **CacheManager** サービスに置き換えられました。CacheManager はファクトリで、JBoss Cache インスタンスのレジストリです。キャッシュが必要なサービスは、名前を用いてキャッシュマネージャーにキャッシュを要求します。キャッシュマネージャーがキャッシュを作成し (キャッシュが作成されていなかった場合)、返します。キャッシュマネージャーは作成したキャッシュの参照を保持するため、同じキャッシュ設定名をリクエストするすべてのサービスが同じキャッシュを共有します。サービスは使用を終えたキャッシュをキャッシュマネージャーに解放します。キャッシュマネージャーは各キャッシュを使用するサービスの数を把握し、すべてのサービスがキャッシュを解放した後にキャッシュを停止し破棄します。

#### 20.2.1.1. 標準のキャッシュ設定

次の標準 JBoss Cache 設定が JBoss Enterprise Application Platform 5 に同梱されます。ニーズに合わせて別の設定を追加するか、これらの設定を編集してキャッシュの動作を調整することができます。追加や変更を行うには、**deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml** ファイルを編集します (詳細は「[CacheManager サービスによるデプロイメント](#)」を参照)。ただし、これらの設定は使用目的に合わせて最適化されているため、本ガイドの各サービスに対する章に明確に説明がない限り、変更しない方がよいでしょう。

- **standard-session-cache**

Web セッションに使用される標準のキャッシュです。

- **field-granularity-session-cache**

FIELD 粒度の Web セッションに使用される標準のキャッシュです。

- **sfsb-cache**

EJB3 SFSB キャッシングに使用される標準のキャッシュです。

- **ha-partition**

Web の階層のクラスター化されたシングルサインオン、HA-JNDI、分散ステートによって使用されます。

- **mvcc-entity**



JBoss Cache の MVCC ロッキングを使用する JPA/Hibernate エンティティ/コレクションキャッシングに適切な設定です (下記の注記を参照)。

- **optimistic-entity**

JBoss Cache の楽観的ロッキングを使用する JPA/Hibernate のエンティティ/コレクションキャッシングに適切な設定です (下記の注記を参照)。

- **pessimistic-entity**

JBoss Cache の悲観的ロッキングを使用する JPA/Hibernate のエンティティ/コレクションキャッシングに適切な設定です (下記の注記を参照)。

- **mvcc-entity-repeatable**

「mvcc-entity」と同じですが、`READ_COMMITTED` ではなく JBoss Cache の `REPEATABLE_READ` 分離レベルを使用します (下記の注記を参照)。

- **pessimistic-entity-repeatable**

「pessimistic-entity」と同じですが、`READ_COMMITTED` ではなく JBoss Cache の `REPEATABLE_READ` 分離レベルを使用します (下記の注記を参照)。

- **local-query**

JPA/Hibernate のクエリ結果のキャッシングに適切な設定です。クエリの結果はレプリケートしません。このキャッシュのクエリ結果が有効であることを検証するために Hibernate が使用するタイムスタンプデータを保存しないでください。

- **replicated-query**

JPA/Hibernate のクエリ結果のキャッシングに適切な設定です。クエリの結果をレプリケートします。このキャッシュのクエリ結果が有効であることを検証するために Hibernate が使用するタイムスタンプデータを保存しないでください。

- **timestamps-cache**

JPA/Hibernate のクエリ結果キャッシングの一部としてキャッシュされるタイムスタンプデータに適切な設定です。クエリ結果自体が **local-query** のような非リプリケーションキャッシュを使用する場合でも、クエリ結果キャッシングが使用される時はレプリケートされたタイムスタンプのキャッシュが必要となります。

- **mvcc-shared**

JPA/Hibernate のエンティティ、コレクション、クエリ結果、タイムスタンプキャッシングで共有されるキャッシュに適切な設定です。最も非効率なモードである `REPL_SYNC` キャッシュモードが必要となるため、推奨される設定ではありません。また、起動時にフルステート転送が必要となるため、高価になることがあります。JBoss 4 では共有キャッシュのみが使用できたため、後方互換性を理由に維持されています。JBoss Cache の MVCC ロックを使用します。

- **optimistic-shared**

JPA/Hibernate のエンティティ、コレクション、クエリ結果、タイムスタンプキャッシングで共有されるキャッシュに適切な設定です。最も非効率なモードである `REPL_SYNC` キャッシュモードが必要となるため、推奨される設定ではありません。また、起動時にフルステート転送が必要となるため、高価になることがあります。JBoss 4 では共有キャッシュのみが使用できたため、後方互換性を理由に維持されています。JBoss Cache の楽観的ロックを使用します。

- **pessimistic-shared**

JPA/Hibernate のエンティティ、コレクション、クエリ結果、タイムスタンプキャッシングで共有されるキャッシュに適切な設定です。最も非効率なモードである REPL\_SYNC キャッシュモードが必要となるため、推奨される設定ではありません。また、起動時にフルステート転送が必要となるため、高価になることがあります。JBoss 4 では共有キャッシュのみが使用できたため、後方互換性を理由に維持されています。JBoss Cache の悲観的ロックを使用します。

- **mvcc-shared-repeatable**

「mvcc-shared」と同じですが、READ\_COMMITTED ではなく JBoss Cache の REPEATABLE\_READ 分離レベルを使用します (下記の注記を参照)。

- **pessimistic-shared-repeatable**

「pessimistic-shared」と同じですが、READ\_COMMITTED ではなく JBoss Cache の REPEATABLE\_READ 分離レベルを使用します (下記の注記を参照)。



#### 注記

JBoss Cache のロックスキームに関する詳しい情報は「[並行アクセス](#)」を参照してください。



#### 注記

JPA/Hibernate の 2 次レベルキャッシングでは、アプリケーションが EntityManager/Hibernate セッションからエンティティを追放または削除し、同じトランザクション内で繰り返し再度読み取りする場合のみ REPEATABLE\_READ が有効です。そうでない場合、セッションの内部キャッシュが repeatable-read のセマンティックを提供します。

### 20.2.1.2. キャッシュ設定エイリアス

キャッシュの登録名以外の名前でルックアップできるようにするなど、CacheManager はキャッシュのエイリアスもサポートします。エイリアスは、異なるキャッシュ設定名が設定されているサービス間でキャッシュを共有するのに便利です。また、Enterprise Application Platform 4 よりポートされるレガシー EJB3 アプリケーション設定のサポートにも便利です。

エイリアスは **jboss-cache-manager-jboss-beans.xml** ファイルの「CacheManager Bean」を編集して設定することができます。次の設定は、Enterprise Application Platform 5 の標準的なエイリアスになります。

```
<bean name="CacheManager" class="org.jboss.ha.cachemanager.CacheManager">
    . . .
    <!-- Aliases for cache names. Allows caches to be shared across
         services that may expect different cache configuration names. -->
    <property name="configAliases">
        <map keyClass="java.lang.String" valueClass="java.lang.String">
            <!-- Use the HAPartition cache for ClusteredSSO caching -->
            <entry>
                <key>clustered-sso</key>
                <value>ha-partition</value>
            </entry>
        </map>
    </property>
</bean>
```

```

        </entry>
        <!-- Handle the legacy name for the EJB3 SFSB cache -->
        <entry>
            <key>jboss.cache:service=EJB3SFSBClusteredCache</key>
            <value>sfsb-cache</value>
        </entry>
        <!-- Handle the legacy name for the EJB3 Entity cache -->
        <entry>
            <key>jboss.cache:service=EJB3EntityTreeCache</key>
            <value>mvcc-shared</value>
        </entry>
    </map>
</property>

    . . .

</bean>

```

## 20.3. HAPARTITION サービス

HAPartition は、Enterprise Application Platform クラスターリングでのさまざまなタスクに使用される汎用的なサービスです。要するに、HAPartition は、1 つまたは複数のクラスタメンバに対する RPC 呼び出しの実行/受信をサポートする JGroups **Channel** の上部に構築された抽象層です。HAPartition を使用するサービスは単一の **Channel** や多重 RPC 呼び出しを共有できるため、設定を簡易化することができ、各サービスが独自の **Channel** を作成するランタイムオーバーヘッドが発生しなくなります。また、HAPartition はどのクラスターリングサービスがどのクラスタメンバで実行されているかを記録する分散レジストリもサポートし、クラスタメンバーシップやクラスター化されたサービスレジストリが変更されたときに該当するリスナに通知します。さらに、HAPartition は、スマートなクライアントサイドクラスタプロキシ、EJB 2 SFSB レプリケーションおよびエンティティキャッシュ管理、ファージング、HA-JNDI、HA シングルトンなどの、本書でこれから説明する多くの重要なクラスターリングサービスを形成します。

次のスニペットは標準の JBoss Enterprise Application Platform ディストリビューションにパッケージされた **HAPartition** サービスの定義を表しています。この設定は、**server/production/deploy/cluster/hapartition-jboss-beans.xml** ファイルにあります。

```

<bean name="HAPartitionCacheHandler"
class="org.jboss.ha.framework.server.HAPartitionCacheHandlerImpl">
    <property name="cacheManager"><inject bean="CacheManager"/></property>
    <property name="cacheConfigName">ha-partition</property>
</bean>
<bean name="HAPartition"
class="org.jboss.ha.framework.server.ClusterPartition">
    <depends>jboss:service=Naming</depends>
    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX

    (name="jboss:service=HAPartition,partition=${jboss.partition.name:DefaultP
artition}",
exposedInterface=org.jboss.ha.framework.server.ClusterPartitionMBean.class
, registerDirectly=true)</annotation>

    <!-- ClusterPartition requires a Cache for state management -->

    <property name="cacheHandler"><inject bean="HAPartitionCacheHandler"/>
</property>

```

```

<!-- Name of the partition being built -->

<property name="partitionName">${jboss.partition.name:DefaultPartition}
</property>

<!-- The address used to determine the node name -->

<property name="nodeAddress">${jboss.bind.address}</property>

<!-- Max time (in ms) to wait for state transfer to complete. Increase
for large states -->

<property name="stateTransferTimeout">30000</property>

<!-- Max time (in ms) to wait for RPC calls to complete. -->

<property name="methodCallTimeout">60000</property>

<!-- Optionally provide a thread source to allow async connect of our
channel -->

<property name="threadPool"><inject
bean="jboss.system:service=ThreadPool"/></property>
<property name="distributedStateImpl">
<bean name="DistributedState"
class="org.jboss.ha.framework.server.DistributedStateImpl">

    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX

(name="jboss:service=DistributedState,partitionName=${jboss.partition.name
:DefaultPartition}",
exposedInterface=org.jboss.ha.framework.server.DistributedStateImplMBean.c
lass, registerDirectly=true)</annotation>

    <property name="cacheHandler"><inject bean="HAPartitionCacheHandler"/>
</property>
</bean>
</property>
</bean>

```

上記の多くは定型的な定義ですが、次にエンドユーザーに適切な主なポイントについて説明します。上記では、**HAPartitionCacheHandler** と **HAPartition** の2つの Bean が定義されています。

**HAPartition** Bean は次の設定プロパティを公開します。

- **PartitionName** はクラスター名を指定する任意の属性です。デフォルト値は **DefaultPartition** です。-g (または --partition) コマンドラインスイッチを使用して、サーバー起動時にこの値を設定します。



### 注記

MCBean:ServerConfig プロファイルサービスコンポーネントにある `partitionName` を使う場合、システムは当プロパティに対して `null` 値を返します。MCBean:HAPartition 管理のコンポーネントからの `PartitionName` を使い正しい値を取得します。

- **nodeAddress** は未使用のため、無視しても問題ありません。
- **stateTransferTimeout** は初期のアプリケーションステート転送のタイムアウト値 (ミリ秒単位) を指定します。ステート転送は、サービスの起動時にすでに実行されているクラスタメンバから初期のアプリケーションステートのシリアル化されたコピーを取得するプロセスを意味します。デフォルト値は **30000** です。
- **methodCallTimeout** は、別のクラスタメンバーからグループ RPC への応答を取得する時のタイムアウト値 (ミリ秒単位) を指定します。デフォルト値は **60000** です。

**HAPartitionCacheHandler** は HAPartition と JBoss Cache の統合を手助けする小さなユーティリティサービスです (「[JBoss Enterprise Application Platform の CacheManager サービス](#)」を参照)。HAPartition は JBoss Cache を使用する DistributedState と呼ばれる子サービスを公開します (「[DistributedState サービス](#)」参照)。**HAPartitionCacheHandler** は、DistributedState のキャッシュが使用する JGroups **Channel** と HAPartition が直接使用する **Channel** 間の設定が一貫性を保つようにします。

- **cacheConfigName** は、HAPartition 関連のキャッシュに使用する JBoss Cache 設定の名前になります。HAPartition が使用するべきである JGroups プロトコルスタック設定の名前も間接的に指定します。JGroups プロトコルスタックの設定方法は、「[JGroups の統合](#)」を参照してください。

ノードがクラスターを形成するには、ノードの **PartitionName** が同じでなければなりません。また、**HAPartitionCacheHandler** の **cacheConfigName** が同じ JBoss Cache 設定を指定しなければなりません。どちらかの要素を一部のノードで変更すると、クラスターが適切に動作しなくなります。

ご使用のブラウザでクラスター内いずれかの JBoss インスタンスの JMX コンソール (<http://hostname:8080/jmx-console/> など) にアクセスしてから

**jboss:service=HAPartition,partition=DefaultPartition** MBean (-g 起動スイッチを使用する場合はご使用のパーティション名が反映されるよう MBean 名を変更) をクリックすると現在のクラスタ情報を表示することができます。現在のクラスタメンバーの IP アドレス一覧は **CurrentView** フィールドに表示されます。



### 注記

1 つの JBoss サーバーインスタンスを複数の HAPartitions に同時に置くことは技術的には可能ですが、管理が複雑になるため一般的には推奨されません。

## 20.3.1. DistributedReplicantManager サービス

**DistributedReplicantManager** (DRM) サービスは、**HAPartition.getDistributedReplicantManager()** メソッドにより HAPartition ユーザーが使用できる HAPartition サービスのコンポーネントの 1 つです。通常、JBoss Enterprise Application Platform のユーザーは直接 DRM を利用することはありません。ここでは、Enterprise Application Platform のクラスタリング内部がどのように動作するかを詳しく知りたいユーザー向けに説明します。

DRM は HAPartition ユーザーが特定のキーでオブジェクトを登録できるようにします。これにより、クラスターのメンバーによりそのキーで登録したオブジェクトのセットを呼び出し側が利用できるようになります。DRM は通知のメカニズムも提供するため、レジストリの内容が変更された時にリスナへ通知を行うこともできます。

JBoss Enterprise Application Platform の DRM には主に 2 つの使用法があります。

- クラスター化されたスマートプロキシ

クラスター化された EJB の名前など、クラスター化されたスマートプロキシを必要とするサービスの名前がキーとなります (「[クライアント側インターセプターアーキテクチャ](#)」を参照)。各ノードが DRM に保存する値オブジェクトは「ターゲット」と呼ばれます。スマートプロキシのトランスポート層はこのターゲットを使用してノードへコンタクトします (RMI スタブ、HTTP URL、JBoss Remoting の **InvokerLocator** など)。クラスター化されたスマートプロキシを構築するファクトリが DRM にアクセスして、プロキシに挿入すべきである「ターゲット」のセットを取得し、クラスター内のすべてのノードと通信できるようにします。

- HASingleton

高可用シングルトンとして機能する必要があるサービスの名前がキーとなります (HASingleton の章を参照)。各ノードが DRM に保存する値オブジェクトは、トークンとして動作するストリングで、ノードにサービスがデプロイされていることを示します。そのため、HA シングルトンサービスの「マスター」ノードの候補になります。

両方の場合で、オブジェクトが登録されたキーが特定のクラスタ化サービスを識別します。クラスター内のすべてのノードがすべてのキーでオブジェクトを登録する必要はありません。特定ノードにデプロイされたサービスのみがサービスのキーを用いて登録を行います。サービスがクラスター全体で均一にデプロイされる必要はありません。そのため、サービスがデプロイされたノードの認識など、DRM はクラスター関連のサービスの「トポロジ」を理解するメカニズムとして有用です。

### 20.3.2. DistributedState サービス

**DistributedState** サービスは、**HAPartition.getDistributedState()** メソッドによって HAPartition ユーザーが利用できる HAPartition サービスのレガシーコンポーネントです。このサービスはクラスターの周囲に任意アプリケーションステートの連携管理を提供します。このサービスは後方互換性を維持するためにサポートされていますが、新しいアプリケーションではこのサービスを使用せず、より高度な JBoss Cache を代わりに使用してください。

JBoss Enterprise Application Platform 5 では、**DistributedState** サービスは基礎の JBoss Cache インスタンスへ委譲します。

### 20.3.3. HAPartition のカスタム使用

カスタムサービスも HAPartition を利用してクラスターとのやりとりに対応することができます。通常、**org.jboss.ha.framework.server.HAServiceImpl** ベースクラスを拡張するのが一番簡単な方法です。JMX の登録や通知のサポートが必要な場合は **org.jboss.ha.jmx.HAServiceMBeanSupport** クラスを拡張します。

## 第21章 クラスター化 JNDI サービス

JNDI はアプリケーションサーバーによって提供される最も重要なサービスの 1 つです。JBoss HA-JNDI (高可用性 JNDI) サービスにより以下の機能が JNDI に提供されます。

- ネーミング処理の透過的なフェイルオーバー。HA-JNDI ネーミングコンテキストは特定の JBoss Enterprise Application Platform インスタンス上の HA-JNDI サービスに接続され、サービスは失敗するか、シャットダウンされます。HA-JNDI クライアントは別の Enterprise Application Platform インスタンスに透過的にフェイルオーバーできます。
- ネーミング処理の負荷分散。HA-JNDI ネーミングコンテキストはクラスター内のすべての HA-JNDI サーバーで要求を自動的に負荷分散します。
- HA-JNDI サーバーの自動クライアント検出 (マルチキャストを使用)
- クラスター全体の JNDI ツリーの統合ビュー。クライアントはクラスター内の任意のノードで実行されている HA-JNDI サービスに接続し、他のノードの JNDI でバインドされたオブジェクトを検索できます。これは以下の 2 つのメカニズムによって実現されます。
  - クラスター間ルックアップ。クライアントはルックアップを実行でき、サーバー側 HA-JNDI サービスはクラスター内のノードにある通常の JNDI でバインドされたものを検索できます。
  - 複製されたクラスター全体のコンテキストツリー。HA-JNDI サービスにバインドされたオブジェクトは、クラスター全体で複製され、そのオブジェクトのコピーはクラスターの各ノードの VM で利用できます。

JNDI は、他の多くのインターセプタベースのクラスタリングサービスに対する主要なコンポーネントです。これらのサービスはそれ自体を JNDI に登録し、クライアントがプロキシをルックアップしてそのサービスを使用できるようにします。HA-JNDI はクライアントがこれらのプロキシをルックアップする高可用性方法を持つようにします。ただし、HA-JNDI の使用 (または非使用) はルックアップされるオブジェクトのクラスタリングの動作にまったく影響を及ぼしません。

- EJB がクラスター対象として設定されていない場合は、HA-JNDI を使用して EJB をルックアップしても、EJB にクラスタリング機能 (EJB 読み出しの負荷分散、透過的なフェイルオーバー、状態レプリケーション) が追加されません。
- EJB がクラスター対象として設定された場合は、HA-JNDI の代わりに通常の JNDI を使用して EJB をルックアップしても、Bean プロキシのクラスタリング機能は削除されません。

### 21.1. 仕組み

JBoss のクライアント側の HA-JNDI ネーミングコンテキストは、クライアント側のインターセプタのアーキテクチャが基になっています (はじめにとクイックスタートの章を参照)。クライアントは HA-JNDI プロキシオブジェクトを取得し (**InitialContext** オブジェクトより)、プロキシよりリモートサーバーの JNDI ルックアップサービスを呼び出します。クライアントは **InitialContext** オブジェクトによって使用されるネーミングプロパティを設定し、HA-JNDI プロキシを要求します。この詳細は「[クライアントの設定](#)」で取り上げています。**InitialContext** に適切なネーミングプロパティが提供されていることを確認する必要がある以外は、ネーミングコンテキストが HA-JNDI を使用していることはクライアントに対して透過的です。

サーバー側では、HA-JNDI サービスはクラスター全体のコンテキストツリーの管理を行います。クラスター全体のツリーはクラスター内にノードが 1 つ存在していれば常に使用することができます。クラスター内の各ノードもノード自体のローカル JNDI コンテキストツリーを管理します。各ノードの HA-JNDI サービスはローカル JNDI コンテキストツリーにバインドされたオブジェクトを検索でき、他のノード上のローカルツリーにバインドされたオブジェクトを検索するためクラスター全体の RPC を



作成することができます。アプリケーションはオブジェクトをいずれかのツリーにバインドできますが、実際はほとんどのオブジェクトがローカルの JNDI コンテキストツリーへバインドされます。このアーキテクチャーの設計原理は次の通りです。

- すでに JNDI 実装がローカルであることを前提としていたので、アプリケーションに伴う移行関連の問題を回避し、設定ファイルを少し調整するだけでそのままクラスタリングを機能するようにしたかったこと。
- 同種のクラスターでは、この構成によってネットワークトラフィック量が軽減され、同じタイプのオブジェクトが各ノードの同じ名前にバインドされること。
- このような設計により、基礎となるすべてのクラスターコードが新しい **InitialContext** を使用してバインディングをルックアップしたり作成するため、HA-JNDI サービスは任意のサービスとなります。

サーバー側では、**new InitialContext()** の呼び出しによって取得されたネーミングコンテキストがローカル専用のクラスタワイドでない JNDI コンテキストにバインドされます。このため、すべての EJB ホームなどはクラスター全体の JNDI コンテキストにはバインドされませんが、各ホームはローカルの JNDI にバインドされます。

リモートクライアントが HA-JNDI を使用してルックアップを行う時、グローバルなクラスター全体のコンテキスト内にオブジェクトが見つからない場合 HA-JNDI はローカル JNDI コンテキスト委譲します。詳細なルックアップルールは次の通りです。

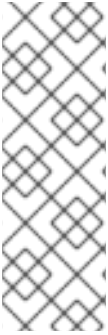
- クラスタ全体の JNDI ツリーにバインディングがある場合はそれを返します。
- クラスタ全体のツリーにバインディングがない場合、ルックアップのクエリをローカルの JNDI サービスに委任して受信した応答があればそれを返信します。
- 応答を受信しなかった場合、HA-JNDI サービスはクラスター内の他すべてのノードに対しローカルの JNDI サービスがそのようなバインディングを持っているか問い合わせ、受信するセットからの応答を返します。
- バインディングを持つローカルの JNDI サービスがない場合、最終的には **NameNotFoundException** が発行されます。

実際には、オブジェクトはクラスター全体の JNDI ツリーにほとんどバインドされず、ローカルの JNDI ツリーにバインドされます。たとえば、EJB がデプロイされた場合、そのプロキシは常に HA-JNDI ではなくローカルの JNDI にバインドされます。したがって、HA-JNDI を使用して行われた EJB ホームルックアップは常にローカルの JNDI インスタンスに委任されます。

## 注記

異なる Bean (同じタイプでも異なるクラスターに参加している) が同じ JNDI 名を使用する場合、各 JNDI サーバーは同じ名前の論理的に異なる「ターゲット」バインドを持つこととなります (ノード 1 の JNDI は Bean A のバインディング、ノード 2 の JNDI は同じ名前でも Bean B のバインディングを持つ)。必然的に、クライアントがこの名前で HA-JNDI クエリを実行すると、このクエリはクラスターの任意の JNDI サーバーで呼び出され、ローカルでバインドされたスタブを返します。しかし、このスタブはクライアントが求めるスタブとは異なる可能性があります。したがって、常にクラスター全体で論理的に異なるバインディングに異なる名前を使用することがベストプラクティスとなります。





## 注記

クラスターの少数のノードでのみバインディングが使用できる場合 (例えば、クラスターにおける小さいサブセットのノードのみに Bean がデプロイされた場合)、このバインディングを所有しない HA-JNDI サーバーをルックアップする可能性が高くなるため、クラスターにあるすべてのノードにルックアップを転送する必要があります。したがって、ローカルでバインディングが使用できる場合よりもクエリにかかる時間が長くなります。そのため、JNDI クエリの結果はできる限りクライアントでキャッシュするようにしてください。



## 注記

HA-JNDI を使用したい場合、現在非 JNP JNDI 実装 (例: LDAP) をローカルの JNDI 実装として使用することはできません。しかし、**ExternalContext** MBean を用いる JNDI フェデレーションを使用すると、非 JBoss JNDI ツリーを JBoss JNDI の名前空間へバインドすることができます。また、クラスター全体や HA-JNDI や JNP のスクラップに 1 つの集中 JNDI サーバーを使用することもできます。

## 21.2. クライアントの設定

クライアントが HA-JNDI を使用するよう設定するには、新しい **InitialContext** が作成された時に適切なネーミングプロパティ環境が使用できるようにします。この方法は、クライアントが JBoss Enterprise Application Platform 内で実行されているか別の VM で実行されているかによって異なります。

### 21.2.1. Enterprise Application Platform 内部で実行されているクライアントの場合

Enterprise Application Platform 内部から HA-JNDI にアクセスしたい場合、JNDI プロパティをコンストラクターに渡して明示的に **InitialContext** を設定する必要があります。次のコードは HA-JNDI にバインドされたネーミングコンテキストの作成方法について示しています。

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
// HA-JNDI is listening on the address passed to JBoss via -b
String bindAddress = System.getProperty("jboss.bind.address",
"localhost");
p.put(Context.PROVIDER_URL, bindAddress + ":1100"); // HA-JNDI address and
port.
return new InitialContext(p);
```

Context.PROVIDER\_URL プロパティは **deploy/cluster/hajndi-jboss-beans.xml** ファイルに設定された HA-JNDI サービスを示しています (「JBoss の設定」を参照)。デフォルトでは、このサービスは **jboss.bind.address** システムプロパティより命名されたインターフェースをリスンします。 **jboss.bind.address** システムプロパティは、JBoss Enterprise Application Platform 起動時に **-b** コマンドラインオプションへ割り当てた値に設定されます (指定がない場合は **localhost**)。上記のコードはこのプロパティへのアクセス例になります。

しかし、すべての場合で動作するわけではなく、特に複数の JBoss Enterprise Application Platform インスタンスを同じマシン上で実行し、同じ IP アドレスへバインドし、異なるポートを使用するよう設定した場合は動作しません。下記のように Context.PROVIDER\_URL を指定せず、**jnp.partitionName** プロパティを指定して **InitialContext** が VM 内の HA-JNDI を静的に見つけるようにするのが最も安全なやり方です。

```

Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
// HA-JNDI is registered under the partition name passed to JBoss via -g
String partitionName = System.getProperty("jboss.partition.name",
"DefaultPartition");
p.put("jnp.partitionName", partitionName);
return new InitialContext(p);

```

この例は **jboss.partition.name** システムプロパティを使用して、HA-JNDI サービスが動作するパーティションを特定します。このシステムプロパティは、JBoss Enterprise Application Platform 起動時に **-g** コマンドラインオプションへ割り当てた値に設定されます (指定がない場合は **DefaultPartition**)。

**jndi.properties** ファイルをデプロイメントに配置し、Enterprise Application Platform の **conf/jndi.properties** ファイルを編集して作業を単純化しないようにしてください。いずれの場合でもほぼ間違いなくアプリケーションが破損し、場合によってはサーバー全体が破損します。クライアント設定を外部化するには、**jndi.properties** という名前でないプロパティファイルをデプロイし、そのファイルの内容をロードする **Properties** オブジェクトをプログラムを使用して作成するのが 1 つの方法です。

#### 21.2.1.1. EJB および WAR からのHA-JNDI リソースへのアクセス -- 環境ネーミングコンテキスト

HA-JNDI が EJB またはサーブレットである場合、リソースのルックアップを設定する最も危害を少なくする方法は、ルックアップを実行する Bean や Web アプリケーションの環境ネーミングコンテキストへリソースをバインドすることです。ローカルマッピングでなく HA-JNDI を使用するようバインディングを設定することができます。JMS 接続ファクトリとキューの例は次の通りです (最も一般的なユースケースとなります)。

ejb-jar.xml の Bean 定義や war の web.xml 内で、接続ファクトリと送信先の 2 つのリソース参照マッピングを定義する必要があります。

```

<resource-ref>
  <res-ref-name>jms/ConnectionFactory</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

<resource-ref>
  <res-ref-name>jms/Queue</res-ref-name>
  <res-type>javax.jms.Queue</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

```

これらの例を使用して、ルックアップを実行する Bean は `java:comp/env/jms/ConnectionFactory` をルックアップして接続ファクトリを取得し、`java:comp/env/jms/Queue` をルックアップしてキューを取得できます。

JBoss 固有の配備記述子 (EJB は `jboss.xml`、WAR は `jboss-web.xml`) の中でこれらの参照を HA-JNDI が使用する URL へマップする必要があります。

```

<resource-ref>

```

```

    <res-ref-name>jms/ConnectionFactory</res-ref-name>
    <jndi-name>jnp://${jboss.bind.address}:1100/ConnectionFactory</jndi-name>
</resource-ref>

<resource-ref>
    <res-ref-name>jms/Queue</res-ref-name>
    <jndi-name>jnp://${jboss.bind.address}:1100/queue/A</jndi-name>
</resource-ref>

```

URL は、Bean と同じノード上で稼働している HA-JNDI サーバーへの URL になります。Bean が利用できる場合、ローカル HA-JNDI サーバーも利用できるはずです。ルックアップはクラスターの全ノードを自動的にクエリし、JMS リソースを持つノードを特定します。

上記で使用される `${jboss.bind.address}` 構文は、URL を判定する時に `jboss.bind.address` システムプロパティの値を使用するよう JBoss に伝えます。このシステムプロパティは、JBoss Enterprise Application Platform の起動時に `-b` コマンドラインオプションで割り当てた値をシステムプロパティ自体に設定します。

### 21.2.1.2. 単に `jndi.properties` ファイルに指定するのではなく、プログラムを使用して行う理由

JBoss Enterprise Application Platform の内部ネーミング環境は、編集してはならない `conf/jndi.properties` ファイルによって制御されます。

Enterprise Application Platform 内に他の `jndi.properties` ファイルをデプロイしないでください。他のファイルをデプロイすると、`jndi.properties` ファイルがあってはならないクラスパス上に存在してしまうことがあるため、サーバーの内部操作を妨害する可能性があります。例えば、HA-JNDI に設定された `jndi.properties` が EJB デプロイメントに含まれる場合、サーバーが EJB プロキシを JNDI ヘバインドすると、プロキシが属するローカル JNDI ツリーではなく、レプリケートされた HA-JNDI ツリーにバインドする可能性が高いでしょう。

### 21.2.1.3. 正しくない HA-JNDI ヘバインドされたことを知る方法

jmx コンソールにて、`jboss:service=JNDIView` MBean 上で `list` 操作を実行します。結果の最後の方に「HA-JNDI Namespace」の内容が一覧表示されます。通常、一覧には何も表示されませんが、明示的にバインドしなかったデプロイメントが表示された場合、不適切な `jndi.properties` ファイルがクラスパスにあると考えられます。例は、[Problem with removing a Node from Cluster](#) を参照してください。

## 21.2.2. Enterprise Application Platform 外部で実行されているクライアントの場合

JNDI クライアントは HA-JNDI クラスターを認識しなければなりません。JNDI サーバーの一覧 (HA-JNDI クラスター内のノードなど) を `jndi.properties` ファイル内の `java.naming.provider.url` JNDI 設定に渡すことができます。各サーバーノードは IP アドレスと JNDI ポート番号によって識別されます。サーバーノードはコンマで区切ります (サーバーとポートの設定方法については「[JBoss の設定](#)」を参照)。

```

java.naming.provider.url=server1:1100,server2:1100,server3:1100,server4:1100

```

初期化の際、JNP クライアントコードは一覧の各サーバーノードとの通信を 1 つずつ試行します。そして、通信できたサーバーがあった時点で試行を停止します。そして、そのサーバーのノードより HA-JNDI スタブをダウンロードします。



## 注記

JNP クライアントのルックアッププロセスには負荷分散の動作はありません。単にプロバイダー一覧を確認してスタブを取得するために利用できる最初のサーバーを使用します。HA-JNDI プロバイダー一覧はクラスタ内の HA-JNDI ノードのサブセットが含まれている必要があります。HA-JNDI スタブがダウンロード可能になると、利用可能なサーバーすべての情報がスタブによって含まれます。サーバーのセットが含まれるようにし、最低でも一覧のサーバーが 1 つ使用できるようにすると良いでしょう。

ダウンロードしたスマートスタブには、現在実行されているノードの一覧と、必要な場合にネーミング要求を負荷分散し別のノードにフェイルオーバーするロジックが含まれます。また、サーバーに対して JNDI 呼び出しが行われる度にプロキシインターセプタのターゲット一覧が更新されます (最後の呼び出し後に一覧が変更された場合のみ)。

プロパティの文字列 `java.naming.provider.url` が空の場合や表示のサーバーすべてにアクセスできない場合、JNP クライアントはネットワーク上のマルチキャスト呼び出しより HA-JNDI サーバーをディスカバリしようとします (自動ディスカバリ)。JNDI サーバーノード上で自動ディスカバリを設定する方法は、「JBoss の設定」を参照してください。自動ディスカバリにより、設定がなくてもクライアントは有効な HA-JNDI サーバーを取得することができる場合があります。自動ディスカバリが動作させるには、クライアントとサーバークラスター間のネットワークセグメントを設定し、このようなマルチキャストデータグラムを伝搬するようにしなければなりません。



## 注記

デフォルトでは、自動ディスカバリ機能はマルチキャストグループのアドレス 230.0.0.4 と 1102 番ポートを使用します。

`java.naming.provider.url` プロパティだけでなく、他のプロパティセットも指定することもできます。次の一覧は、新規の `InitialContext` を作成する時に指定できるクラスタリング関係のクライアント側プロパティになります (通常の JNDI と共に使用されるすべての標準的な非クラスタリング関連環境プロパティも使用できます)。

- **java.naming.provider.url:** クラスター内の HA-JNDI プロバイダーノード群の IP アドレスとポート番号の一覧を提供します。クライアントはプロバイダーを 1 つずつ確認し、最初に応答するプロバイダーを使用します。
- **jnp.disableDiscovery: true** が指定されると自動ディスカバリ機能が無効になります。デフォルトでは **false** が指定されます。
- **jnp.partitionName:** 異なるクラスター (パーティション) にバインドされる複数の HA-JNDI サービスが実行されている環境でこのプロパティを使用すると、クライアントが希望するパーティションのサーバーからの自動ディスカバリ応答のみを受け入れるようにできます。自動ディスカバリ機能を使用しない場合は (`jnp.disableDiscovery` が `true` の場合など)、このプロパティは使用されません。デフォルトでは、このプロパティは設定されず、クラスターパーティション名に関係なく最初に応答する HA-JNDI サーバーが自動のディスカバリ機能によって選択されます。
- **jnp.discoveryTimeout:** 自動ディスカバリパケットへの応答に対するコンテキストの待ち時間を指定します。デフォルトは 5000 ミリ秒です。
- **jnp.discoveryGroup:** 自動ディスカバリに使用されるマルチキャストグループアドレスを指定します。デフォルト値は 230.0.0.4 です。サーバー側 HA-JNDI サービスで設定された `AutoDiscoveryAddress` の値を一致しなければなりません。サーバー側の HA-JNDI サービス

は、デフォルトでは **-u** 起動スイッチより指定されたアドレス上でリッスンします。そのため、サーバー側で **-u** が使用される場合 (推奨通り)、クライアント側で `jnp.discoveryGroup` を設定する必要があります。

- **jnp.discoveryPort**: 自動ディスカバリに使用されるマルチキャストポートを指定します。デフォルト値は 1102 です。サーバー側 HA-JNDI サービスで設定された `AutoDiscoveryPort` の値と一致しなければなりません。
- **jnp.discoveryTTL**: 自動ディスカバリ IP マルチキャストパケットの TTL (time-to-live) を指定します。この値は、ネットワーク装置がマルチキャストパケットをドロップする前にマルチキャストパケットが伝搬できるネットワークホップ数を表します。名前とは異なり、時間の単位を表すものではありません。

## 21.3. JBOSS の設定

`$JBOSS_HOME/server/all/deploy/cluster` ディレクトリの `hajndi-jboss-beans.xml` ファイルには、HA-JNDI サービスを有効にする次の Bean が含まれています。

```
<bean name="HAJNDI" class="org.jboss.ha.jndi.HANamingService">

    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
      (name="jboss:service=HAJNDI",
        exposedInterface=org.jboss.ha.jndi.HANamingServiceMBean.class)
    </annotation>

    <!-- The partition used for group RPCs to find locally bound objects
on other nodes -->
    <property name="HAPartition"><inject bean="HAPartition"/></property>

    <!-- Handler for the replicated tree -->
    <property name="distributedTreeManager">
      <bean
class="org.jboss.ha.jndi.impl.jbc.JBossCacheDistributedTreeManager">
        <property name="cacheHandler"><inject
bean="HAPartitionCacheHandler"/></property>
      </bean>
    </property>

    <property name="localNamingInstance">
      <inject bean="jboss:service=NamingBeanImpl"
property="namingInstance"/>
    </property>

    <!-- The thread pool used to control the bootstrap and auto
discovery lookups -->
    <property name="lookupPool"><inject
bean="jboss.system:service=ThreadPool"/></property>

    <!-- Bind address of bootstrap endpoint -->
    <property name="bindAddress">${jboss.bind.address}</property>
    <!-- Port on which the HA-JNDI stub is made available -->
    <property name="port">
      <!-- Get the port from the ServiceBindingManager -->
      <value-factory bean="ServiceBindingManager"
method="getIntBinding">
```

```

        <parameter>jboss:service=HAJNDI</parameter>
        <parameter>Port</parameter>
    </value-factory>
</property>

<!-- Bind address of the HA-JNDI RMI endpoint -->
<property name="rmiBindAddress">${jboss.bind.address}</property>

<!-- RmiPort to be used by the HA-JNDI service once bound. 0 =
ephemeral. -->
<property name="rmiPort">
    <!-- Get the port from the ServiceBindingManager -->
    <value-factory bean="ServiceBindingManager"
method="getIntBinding">
        <parameter>jboss:service=HAJNDI</parameter>
        <parameter>RmiPort</parameter>
    </value-factory>
</property>

<!-- Accept backlog of the bootstrap socket -->
<property name="backlog">50</property>

<!-- A flag to disable the auto discovery via multicast -->
<property name="discoveryDisabled">>false</property>
<!-- Set the auto-discovery bootstrap multicast bind address. If not
specified and a BindAddress is specified, the BindAddress will be
used. -->
<property name="autoDiscoveryBindAddress">${jboss.bind.address}
</property>
<!-- Multicast Address and group port used for auto-discovery -->
<property
name="autoDiscoveryAddress">${jboss.partition.udpGroup:230.0.0.4}
</property>
<property name="autoDiscoveryGroup">1102</property>
<!-- The TTL (time-to-live) for autodiscovery IP multicast packets -
->
<property name="autoDiscoveryTTL">16</property>

<!-- The load balancing policy for HA-JNDI -->
<property
name="loadBalancePolicy">org.jboss.ha.framework.interfaces.RoundRobin</pro
perty>

<!-- Client socket factory to be used for client-server
RMI invocations during JNDI queries
<property name="clientSocketFactory">custom</property>
-->
<!-- Server socket factory to be used for client-server
RMI invocations during JNDI queries
<property name="serverSocketFactory">custom</property>
-->
</bean>

```

この Bean は、異なるプロパティに挿入された複数の他のサービスを持っていることが分かります。

- **HAPartition** は、HA-JNDI のクラスター化プロキシの管理や、別のノードにてローカルでバインドされたオブジェクトを検索するグループ RPC の作成に使用されるコアのクラスタリング



サービスを許可します。詳細は、「[HAPartition サービス](#)」を参照してください。

- **distributedTreeManager** は、レプリケートされたツリーのハンドラーを許可します。標準的なハンドラは、JBoss Cache を使用してレプリケートされたツリーを管理します。挿入された **HAPartitionCacheHandler** Bean を使用して JBoss Cache インスタンスが読み出しされます。詳細は「[HAPartition サービス](#)」を参照してください。
- **localNamingInstance** はローカル JNDI サービスへの参照を許可します。
- **LookupPool** は、ブートストラップや自動ディスカバリのルックアップを処理するスレッドを提供するために使用するスレッドプールを許可します。

上記の依存関係が注入されたサービスの他に、HA-JNDI Bean に使用できる設定属性は次の通りです。

- **bindAddress** は、JNP クライアントからネーミングプロキシダウンロードリクエストをリッスンするために HA-JNDI サーバーがバインドするアドレスを指定します。デフォルト値は **jboss.bind.address** システムプロパティの値になります。**jboss.bind.address** システムプロパティが設定されていない場合は、**localhost** がデフォルト値になります。JBoss 起動時に **-b** コマンドラインスイッチが使用されると **jboss.bind.address** システムプロパティが設定されます。
- **port** は、JNP クライアントからネーミングプロキシダウンロード要求をリッスンするために HA-JNDI サーバーがバインドするポートを指定します。値は **conf/bootstrap/bindings.xml** に設定されている ServiceBindingManager Bean より取得されます。デフォルト値は **1100** です。
- **backlog** は、サービスが JNP クライアントからのネーミングプロキシダウンロード要求をリッスンする TCP サーバーソケットに対する受信接続指示の最大キュー長を指定します。デフォルト値は **50** です。
- **rmiBindAddress** は、ネーミングプロキシからの RMI 要求 (JNDI ルックアップに対してなど) をリッスンするために HA-JNDI サーバーがバインドするアドレスを指定します。デフォルト値は **jboss.bind.address** システムプロパティの値になります。**jboss.bind.address** システムプロパティが設定されていない場合は、**localhost** がデフォルト値になります。JBoss 起動時に **-b** コマンドラインスイッチが使用されると **jboss.bind.address** システムプロパティが設定されます。
- **rmiPort** は、ダウンロードされたスタブと通信するためにサーバーがバインドするポートを指定します。**conf/bootstrap/bindings.xml** に設定された ServiceBindingManager Bean より値が取得されます。デフォルト値は **1101** です。設定された値がない場合、オペレーティングシステムが自動的にポートを割り当てます。
- **discoveryDisabled** は、自動ディスカバリマルチキャストリスナの設定を無効にするブール変数フラグです。デフォルトは **false** です。
- **autoDiscoveryAddress** JNDI 自動ディスカバリのためにリッスンするマルチキャストアドレスを指定します。デフォルト値は、**jboss.partition.udpGroup** システムプロパティの値になります。このシステムプロパティが設定されていない場合は、**230.0.0.4** がデフォルト値となります。JBoss 起動時に **-u** コマンドラインスイッチが使用されると **jboss.partition.udpGroup** システムプロパティが設定されます。
- **autoDiscoveryGroup** は、JNDI 自動ディスカバリパケットのためにリッスンするポートを指定します。デフォルト値は **1102** です。

- **AutoDiscoveryBindAddress** は HA-JNDI が自動ディスカバリ要求パケットをリッスンするインターフェースを設定します。この属性が指定されておらず **bindAddress** が指定されている場合、**bindAddress** が使用されます。
- **autoDiscoveryTTL** は自動ディスカバリ IP マルチキャストパケットの TTL (time-to-live) を指定します。この値は、ネットワーク装置がマルチキャストパケットをドロップする前にマルチキャストパケットが伝搬できるネットワークホップ数を表します。名前とは異なり、時間の単位を表すものではありません。
- **loadBalancePolicy** は、クライアントプロキシに含まれるべきである `LoadBalancePolicyImplementation` のクラス名を指定します。詳細は、紹介とクイックスタート章を参照してください。
- **clientSocketFactory** は、クライアントソケットの作成に使用される `java.rmi.server.RMIClientSocketFactory` の完全修飾クラス名を指定する任意の属性です。デフォルトは **null** です。
- **serverSocketFactory** は、サーバーソケットの作成に使用される `java.rmi.server.RMIServerSocketFactory` の完全修飾クラス名を指定する任意の属性です。デフォルトは **null** です。

### 21.3.1.2 つ目の HA-JNDI サービスの追加

異なる HAPartitions を使用する複数の HA-JNDI サービスを起動することも可能です。例えば、1つのノードが多数の論理クラスターの一部である場合などに使用することができます。この場合、各サービスにも別々のポートや IP アドレスを設定するようにしてください。例えば、設定するサンプルのクラスターに HA-JNDI を接続してバインドするポートを変更したい場合、Bean 記述子は次のようになります (標準的なデプロイメントと変わらないプロパティは省略します)。

```
<-- Cache Handler for secondary HAPartition -->
<bean name="SecondaryHAPartitionCacheHandler"

class="org.jboss.ha.framework.server.HAPartitionCacheHandlerImpl">
    <property name="cacheManager"><inject bean="CacheManager"/>
</property>
    <property name="cacheConfigName">secondary-ha-
partition</property>
</bean>

<-- The secondary HAPartition -->
<bean name="SecondaryHAPartition"
class="org.jboss.ha.framework.server.ClusterPartition">

    <depends>jboss:service=Naming</depends>

    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="jboss:service=HAPartition,partition=SecondaryPartition",

exposedInterface=org.jboss.ha.framework.server.ClusterPartitionMBean.class
, registerDirectly=true)</annotation>

    <property name="cacheHandler"><inject
bean="SecondaryHAPartitionCacheHandler"/></property>

    <property name="partitionName">SecondaryPartition</property>
```



```
.....
</bean>

<bean name="MySpecialPartitionHAJNDI"
class="org.jboss.ha.jndi.HANamingService">

    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="jboss:service=HAJNDI,partitionName=SecondaryPartition",
    exposedInterface=org.jboss.ha.jndi.HANamingServiceMBean.class)
</annotation>

    <property name="HAPartition"><inject bean="SecondaryHAPartition"/>
</property>

    <property name="distributedTreeManager">
        <bean
class="org.jboss.ha.jndi.impl.jbc.JBossCacheDistributedTreeManager">
            <property name="cacheHandler"><inject
bean="SecondaryHAPartitionPartitionCacheHandler"/></property>
        </bean>
    </property>

    <property name="port">56789</property>

    <property name="rmiPort">56790</property>

    <property name="autoDiscoveryGroup">56791</property>

.....
</bean>
```

## 第22章 クラスター化されたセッション EJB

セッション EJB はリモート呼び出しサービスを提供します。このセッションはクライアント側のインターセプターアーキテクチャに基づいてクラスター化されます。クラスター化されたセッション Bean のクライアントアプリケーションは、若干の変更を除きセッション Bean のクラスター化されていないバージョンのクライアントと同じです。クライアント側ではコードの変更や再コンパイルは必要ありません。それでは、EJB 3.0 と EJB 2.x のサーバーアプリケーションそれぞれでクラスター化されたセッション Bean を設定する方法について見てみましょう。

### 22.1. EJB 3.0 のステートレスセッション BEAN

ステートレスセッション Bean のクラスター化は、ステートが関係しないため、最も簡単なケースだと言えます。呼び出しは、クラスターの参加しているノード (この特定の Bean がデプロイされたすべてのノード) すべてへ呼び出しの負荷を分散することができます。

EJB 3.0 でステートレスセッション Bean をクラスターするには、**@Clustered** アノテーションを用いて、Bean クラスにアノテーションを付けます。このアノテーションには、負荷分散ポリシーと使用するパーティションの両方を上書きするオプションのパラメーターが含まれます。

```
public @interface Clustered
{
    String partition() default "${jboss.partition.name:DefaultPartition}";
    String loadBalancePolicy() default "LoadBalancePolicy";
}
```

- **partition** は、Bean が参加するクラスターの名前を指定します。**@Clustered** アノテーションで、個別の Bean に対してデフォルトパーティション **DefaultPartition** を上書きできますが、**jboss.partition.name** システムプロパティを使用するとすべての Bean に対して上書きできます。
- **loadBalancePolicy** は、クラスターのノード上の呼び出しを Bean スタブが分散する方法を示し、**org.jboss.ha.client.loadbalance.LoadBalancePolicy** を実装するクラス名を定義します。デフォルト値 **LoadBalancePolicy** は、セッション Bean タイプのデフォルトポリシーを示す特別なトークンです。ステートレスセッション Bean のデフォルトポリシーは **org.jboss.ha.client.loadbalance.RoundRobin** です。独自の実装を使用してデフォルト値を上書き、あるいは使用できるポリシーのリストより選択することができます。

#### **org.jboss.ha.client.loadbalance.RoundRobin**

無作為のターゲットから開始し、負荷分散を最大限にするため、常にリスト上で次に利用できるターゲットを選択しようとします。

#### **org.jboss.ha.client.loadbalance.RandomRobin**

以前に選択されたターゲットを考慮せずに、無作為にターゲットを選択します。

#### **org.jboss.ha.client.loadbalance.aop.FirstAvailable**

ターゲットが選択されると、同じターゲットを選択しようとします (これ以上負荷分散が行われません)。ステートフルセッション Bean など、「スティッキーセッション」動作が望まれる場合に便利です。

#### **org.jboss.ha.client.loadbalance.aop.FirstAvailableIdenticalAllProxies**

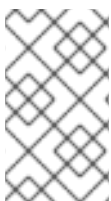
**FirstAvailable** と似ていますが、希望のターゲットがすべてのプロキシで共有されます。

以下は、クラスター化された EJB 3.0 のステートレスセッション bean の実装例になります。

```
@Stateless
@Clustered
public class MyBean implements MySessionInt
{
    public void test()
    {
        // Do something cool
    }
}
```

**@Clustered** アノテーションを使用する代わりに、jboss.xml のセッション Bean に対してクラスタリングを有効にすることもできます。

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>NonAnnotationStateful</ejb-name>
      <clustered>true</clustered>
      <cluster-config>
        <partition-name>FooPartition</partition-name>
        <load-balance-
policy>org.jboss.ha.framework.interfaces.RandomRobin</load-balance-policy>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```



### 注記

**<clustered>True</clustered>** 要素は conf/standardjboss.xml ファイルの **<configuration-name>Clustered Stateless SessionBean</configuration-name>** 要素のエイリアスにすぎません。

この Bean の設定では、Bean がクラスタリング機能のサポートを必要とすることを示す **<clustered>** 要素のみが必須です。オプションの **<cluster-config>** 要素のデフォルト値は **@Clustered** アノテーションからの対応するプロパティと一致します。

## 22.2. EJB 3.0 でのステートフルセッション BEAN

ステートフルセッション Bean のクラスター化は、JBoss がステート情報を管理する必要があるためステートレスセッション bean よりも複雑になります。すべてのステートフルセッション Bean のステートはレプリケートされ、Bean のステートが変更されるたびにクラスター全体で同期されます。

### 22.2.1. EJB アプリケーション設定

EJB 3.0 でステートフルセッション Bean をクラスター化するには、EJB 3.0 のステートレスセッション

ン Bean をクラスター化したように、Bean 実装クラスに **@Cluster** アノテーションのタグを付ける必要があります。ステートレスセッション Bean と異なり、ステートフルセッション Bean のメソッド呼び出しは、デフォルトで **org.jboss.ha.client.loadbalance.aop.FirstAvailable** を使用して負荷分散されます。このポリシーを使用してメソッド呼び出しが無作為に選択されたノードに従います。

**@org.jboss.ejb3.annotation.CacheConfig** アノテーションを Bean に適応してデフォルトのキャッシング動作を上書きすることもできます。**@CacheConfig** アノテーションの定義は次の通りです。

```
public @interface CacheConfig
{
    String name() default "";
    int maxSize() default 10000;
    long idleTimeoutSeconds() default 300;
    boolean replicationIsPassivation() default true;
    long removalTimeoutSeconds() default 0;
}
```

- **name** は、「[CacheManager サービスの設定](#)」で説明した **CacheManager** サービスで登録されたキャッシュ設定の名前を指定します。デフォルトでは、**sfsb-cache** 設定が使用されます。
- **maxSize** は、LRU アルゴリズムを使用して Bean の非活性化を開始する前にキャッシュできる Bean の最大数を指定します。
- **idleTimeoutSeconds** はキャッシュが Bean を非活性化する前に Bean が未使用の状態である最大時間を指定します (maxSize Bean がキャッシュされているかは無関係)。
- **removalTimeoutSeconds** は、キャッシュが Bean を削除するまで Bean を使用しない最大時間を指定します。
- **replicationIsPassivation** は、キャッシュでレプリケーションを非活性化と同等であると見なし、Bean に対して **@PrePassivate** と **@PostActivate** のコールバックを呼び出すかどうかを指定します。デフォルトでは真であり、レプリケーションには Bean のシリアル化が関係するため、シリアル化の準備とシリアル化からの回復がコールバックメソッドを実装する一般的な理由となります。

以下は、クラスター化された EJB 3.0 のステートレスセッション Bean の実装例になります。

```
@Stateful
@Clustered
@CacheConfig(maxSize=5000, removalTimeoutSeconds=18000)
public class MyBean implements MySessionInt
{
    private int state = 0;

    public void increment()
    {
        System.out.println("counter: " + (state++));
    }
}
```

ステートレス Bean と同様に、**@Clustered** アノテーションは省略でき、クラスタリング設定は **jboss.xml** に適用されます。

```

<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>NonAnnotationStateful</ejb-name>
      <clustered>true</clustered>
      <cache-config>
        <cache-max-size>5000</cache-max-size>
        <remove-timeout-seconds>18000</remove-timeout-seconds>
      </cache-config>
    </session>
  </enterprise-beans>
</jboss>

```

### 22.2.2. ステートレプリケーションの最適化

レプリケーションプロセスにはコストがかかるため、オプションで Bean クラスに `org.jboss.ejb3.cache.Optimized` を実装することによりレプリケーションの動作を最適化できます。

```

public interface Optimized
{
    boolean isModified();
}

```

Bean をレプリケートする前に、Bean が **Optimized** メソッドを実装するかをコンテナーがチェックします。この場合、コンテナーが **isModified()** メソッドを呼び出し、メソッドが **true** を返した場合のみ Bean をレプリケートします。Bean が変更されていない場合 (またはユーザーの希望により十分な変更でないためレプリケーションが必要ない場合) は、**false** を返すことができます。この場合、レプリケーションは発生しません。

### 22.2.3. CacheManager サービスの設定

JBoss Cache は EJB 3.0 のステートフルセッション bean のセッションステートレプリケーションサービスを提供します。[「JBoss Enterprise Application Platform の CacheManager サービス」](#) で説明されている **CacheManager** サービスは JBoss Cache インスタンスのファクトリでもありレジストリでもあります。次の通り、ステートフルセッション Bean はデフォルトで **CacheManager** より **sfsb-cache** 設定を使用します。

```

<bean name="StandardSFSBCacheConfig"
class="org.jboss.cache.config.Configuration">

  <!-- No transaction manager lookup -->

  <!-- Name of cluster. Needs to be the same for all members -->
  <property name="clusterName">${jboss.partition.name:DefaultPartition}-
SFSBCache</property>
  <!--
    Use a UDP (multicast) based stack. Need JGroups flow control (FC)
    because we are using asynchronous replication.
  -->
  <property name="multiplexerStack">${jboss.default.jgroups.stack:udp}
</property>
  <property name="fetchInMemoryState">true</property>

  <property name="nodeLockingScheme">PESSIMISTIC</property>

```

```

<property name="isolationLevel">REPEATABLE_READ</property>
<property name="useLockStriping">false</property>
<property name="cacheMode">REPL_ASYNC</property>

<!--
    Number of milliseconds to wait until all responses for a
    synchronous call have been received. Make this longer
    than lockAcquisitionTimeout.
-->
<property name="syncReplTimeout">17500</property>
<!-- Max number of milliseconds to wait for a lock acquisition -->
<property name="lockAcquisitionTimeout">15000</property>
<!-- The max amount of time (in milliseconds) we wait until the
state (ie. the contents of the cache) are retrieved from
existing members at startup. -->
<property name="stateRetrievalTimeout">60000</property>

<!--
    SFSBs use region-based marshalling to provide for partial state
    transfer during deployment/undeployment.
-->
<property name="useRegionBasedMarshalling">false</property>
<!-- Must match the value of "useRegionBasedMarshalling" -->
<property name="inactiveOnStartup">false</property>

<!-- Disable asynchronous RPC marshalling/sending -->
<property name="serializationExecutorPoolSize">0</property>
<!-- We have no asynchronous notification listeners -->
<property name="listenerAsyncPoolSize">0</property>

<property name="exposeManagementStatistics">true</property>

<property name="buddyReplicationConfig">
    <bean class="org.jboss.cache.config.BuddyReplicationConfig">

        <!-- Just set to true to turn on buddy replication -->
        <property name="enabled">false</property>

        <!--
            A way to specify a preferred replication group. We try
            and pick a buddy who shares the same pool name (falling
            back to other buddies if not available).
        -->
        <property name="buddyPoolName">default</property>

        <property name="buddyCommunicationTimeout">17500</property>

        <!-- Do not change these -->
        <property name="autoDataGravitation">false</property>
        <property name="dataGravitationRemoveOnFind">true</property>
        <property name="dataGravitationSearchBackupTrees">true</property>

        <property name="buddyLocatorConfig">
            <bean
class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
                <!-- The number of backup nodes we maintain -->

```

```

        <property name="numBuddies">1</property>
        <!-- Means that each node will *try* to select a buddy on
            a different physical host. If not able to do so
            though, it will fall back to colocated nodes. -->
        <property name="ignoreColocatedBuddies">true</property>
    </bean>
</property>
</bean>
</property>
<property name="cacheLoaderConfig">
    <bean class="org.jboss.cache.config.CacheLoaderConfig">
        <!-- Do not change these -->
        <property name="passivation">true</property>
        <property name="shared">false</property>

        <property name="individualCacheLoaderConfigs">
            <list>
                <bean class="org.jboss.cache.loader.FileCacheLoaderConfig">
                    <!-- Where passivated sessions are stored -->
                    <property
name="location">${jboss.server.data.dir}${/}sfsb</property>
                    <!-- Do not change these -->
                    <property name="async">false</property>
                    <property name="fetchPersistentState">true</property>
                    <property name="purgeOnStartup">true</property>
                    <property name="ignoreModifications">false</property>
                    <property name="checkCharacterPortability">false</property>
                </bean>
            </list>
        </property>
    </bean>
</property>

<!-- EJBs use JBoss Cache eviction -->
<property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
        <property name="wakeupInterval">5000</property>
        <!-- Overall default -->
        <property name="defaultEvictionRegionConfig">
            <bean class="org.jboss.cache.config.EvictionRegionConfig">
                <property name="regionName">/</property>
                <property name="evictionAlgorithmConfig">
                    <bean
class="org.jboss.cache.eviction.NullEvictionAlgorithmConfig"/>
                </property>
            </bean>
        </property>
        <!-- EJB3 integration code will programmatically create other regions
as beans are deployed -->
    </bean>
</property>
</bean>

```

## エビクション

デフォルトの SFSB キャッシュはエビクションをサポートするよう設定されます。EJB3 SFSB コンテナは JBoss Cache エビクションメカニズムを使用して SFSB 非活性化を管理します。Bean がデブ

ロイされた場合、EJB コンテナはプログラムを使用してキャッシュにエビクションリージョンを追加します (1 つの Bean タイプに対して 1 つのリージョン)。

## CacheLoader

JBoss Cache CacheLoader も設定し、SFSB 非活性化を再度サポートします。Bean がキャッシュから除外されると、キャッシュローダーは永続ストアに Bean を非活性化します (この場合は **\$JBoss\_HOME /server/all/data/sfsb** ディレクトリのファイルシステムに行います)。JBoss Cache は、様々な CacheLoader 実装に対応しており、これらの実装はあらゆる永続ストアタイプへデータを格納する方法を認識します。詳細については、JBoss Cache のドキュメントを参照してください。ただし、CacheLoaderConfiguration を変更する場合は、共有されたストア (共有されたデータベースの単一のスキーマなど) を使用しないようにしてください。クラスター内の各ノードは独自の永続ストアを持つ必要があります。そうでないと、ノードは個別でクラスター化された Bean を非活性化および活性化するため、ノードがそれぞれのデータを破損してしまいます。

## バディレプリケーション

バディレプリケーションを使用すると、クラスターのすべてのサーバーではなく、クラスターで設定可能な数のバックアップサーバー (バディ) ヘステートがレプリケートされます。バディレプリケーションを有効にするには、**buddyReplicationConfig** プロパティ Bean の次のプロパティを調整します。

- **enabled** を **true** に設定します。
- **nbuddyPoolName** を使用してクラスター内でノードの論理サブグループを形成します。可能な場合、同じバディプールのノードよりバディが選択されます。
- 各ノードがステートをレプリケートするバックアップノードの数に合わせ、**buddyLocatorConfig.numBuddies** プロパティを調整します。

## 22.3. EJB 2.X のステートレスセッション BEAN

EJB 2.x Bean をクラスター化するには、**jboss.xml** 記述子を変更し、**<clustered>** タグが含まれるようにする必要があります。

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>nextgen.StatelessSession</ejb-name>
      <jndi-name>nextgen.StatelessSession</jndi-name>
      <clustered>true</clustered>
      <cluster-config>
        <partition-name>DefaultPartition</partition-name>
        <home-load-balance-
policy>org.jboss.ha.framework.interfaces.RoundRobin</home-load-balance-
policy>
        <bean-load-balance-
policy>org.jboss.ha.framework.interfaces.RoundRobin</bean-load-balance-
policy>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```



- **partition-name** は、Bean が参加するクラスターの名前を指定します。デフォルト値は **DefaultPartition** です。デフォルトのパーティション名は、**jboss.partition.name** システムプロパティを使用してシステム全体で設定することもできます。
- **home-load-balance-policy** は、クラスターのノードへの呼び出しを分散するためにホームスタブが使用するクラスを指定します。デフォルトでは、プロキシは **RoundRobin** を用いて呼び出しの負荷を分散します。
- **bean-load-balance-policy** は、クラスターのノードへの呼び出しを分散するために Bean スタブが使用するクラスを指定します。デフォルトでは、プロキシは **RoundRobin** を用いて呼び出しの負荷を分散します。

## 22.4. EJB 2.X のステートフルセッション BEAN

ステートフルセッション Bean のクラスター化は、JBoss がステート情報を管理する必要があるためステートレスセッション bean よりも複雑になります。すべてのステートフルセッション Bean のステートはレプリケートされ、Bean のステートが変更されるたびにクラスター全体で同期されます。JBoss Enterprise Application Platform は **HASessionStateService** Bean を使用してクラスター化された EJB 2.x ステートフルセッション Bean の分散セッションステートを管理します。この項では、セッション Bean 設定と **HASessionStateService** Bean 設定の両方について説明します。

### 22.4.1. EJB アプリケーション設定

EJB アプリケーションでは、各ステートフルセッション Bean に対する **jboss.xml** 記述子ファイルを変更し、**<clustered>** タグを追加する必要があります。

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>nextgen.StatefulSession</ejb-name>
      <jndi-name>nextgen.StatefulSession</jndi-name>
      <clustered>True</clustered>
      <cluster-config>
        <partition-name>DefaultPartition</partition-name>
        <home-load-balance-
policy>org.jboss.ha.framework.interfaces.RoundRobin</home-load-balance-
policy>
        <bean-load-balance-
policy>org.jboss.ha.framework.interfaces.FirstAvailable</bean-load-
balance-policy>
        <session-state-manager-jndi-name>/HASessionState/Default</session-
state-manager-jndi-name>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```

この Bean 設定では、Bean がクラスターで動作することを示す **<clustered>** タグが必須です。**<cluster-config>** 要素はオプションであり、そのデフォルトの属性値は上記と同じ設定で指定されます。

**<session-state-manager-jndi-name>** タグは、この Bean が使用する **HASessionStateService** サービスの JNDI 名を提供するために使用されます。

残りのタグの詳細はステートレスセッション Bean と同一となります。クラスター化されたステートフルセッション Bean のホームインタフェースでのアクションはデフォルトでラウンドロビン方式により負荷分散されます。クライアントに対して Bean のリモートスタブが利用できるようになると、呼び出しはラウンドロビン方式で負荷分散されなくなり、一覧の 1 番目のノードに固定されます。

### 22.4.2. ステートレプリケーションの最適化

レプリケーションプロセスにはコストがかかるため、オプションで Bean クラスに以下のシグネチャでメソッドを実装することによりレプリケーションの動作を最適化できます。

```
public boolean isModified();
```

Bean をレプリケートする前に、Bean がこのメソッドを実装するかをコンテナがチェックします。この場合、コンテナが `isModified()` メソッドを呼び出し、メソッドが `true` を返した場合のみ Bean をレプリケートします。Bean が変更されていない場合 (またはユーザーの希望により十分な変更でないためレプリケーションが必要ない場合) は、`false` を返すことができます。この場合、レプリケーションは発生しません。

### 22.4.3. The `HASessionStateService` サービスの設定

`HASessionStateService` Bean は `$JBoss_HOME/server/PROFILE/deploy/cluster/ha-legacy-jboss-beans.xml` ファイルで定義されます。

```
<bean name="HASessionStateService"
      class="org.jboss.ha.hasessionstate.server.HASessionStateService">

  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="jboss:service=HASessionState",
    exposedInterface=org.jboss.ha.hasessionstate.server.
    HASessionStateServiceMBean.class,
    registerDirectly=true)</annotation>

  <!-- Partition used for group RPCs -->
  <property name="HAPartition"><inject bean="HAPartition"/></property>

  <!-- JNDI name under which the service is bound -->
  <property name="jndiName">/HASessionState/Default</property>
  <!-- Max delay before cleaning unreclaimed state.
    Defaults to 30*60*1000 => 30 minutes -->
  <property name="beanCleaningDelay">0</property>

</bean>
```

`HASessionStateService` Bean の設定属性は次の通りです。

- **HAPartition** は HA-JNDI がクラスター内の通信に使用する HAPartition サービスを挿入する必須属性です。
- **JndiName** は、この `HASessionStateService` サービスがバインドされる JNDI 名を指定するオプションの属性です。デフォルト値は `/HAPartition/Default` になります。
- **BeanCleaningDelay** はオプションの属性で、指定された時間 (ミリ秒単位) が経過した後に `HASessionStateService` が変更されていないステートを消去します。Bean を所有する

ノードがクラッシュすると、その兄弟ノードがこの Bean を所有することになります。兄弟ノードのコンテナキャッシュはそれを認識しないため、Bean の消去設定に基づいて消去することはできません。そのため、**HASessionStateService** が時々クリーンアップを行う必要があります。デフォルト値は **30\*60\*1000** ミリ秒 (30 分) です。

#### 22.4.4. クラスター再起動の処理

HA スマートクライアントアーキテクチャについては「[クライアント側インターセプターアーキテクチャ](#)」で説明しました。デフォルトの HA スマートプロキシクライアントはクラスター内に 1 つのノードがある限りフェイルルオーバーを行うことができます。クラスターが完全にシャットダウンされた場合、プロキシは孤立しクラスター内の使用可能なノード群の情報を失うことになります。この状態からプロキシを復帰させる方法はありません。プロキシはノードが再起動されたら JNDI/HA-JNDI から新しい対象をルックアップする必要があります。

このような再起動の障害からの透過的に回復できるよう、RetryInterceptor をプロキシクライアント側のインターセプタースタックに追加することができます。EJB に対してこれを有効にするには、RetryInterceptor を含む invoker-proxy-binding を設定します。jboss.xml 設定の例は次の通りです。

```
<jboss>
  <session>
    <ejb-name>nextgen_RetryInterceptorStatelessSession</ejb-name>
    <invoker-bindings>
      <invoker>
        <invoker-proxy-binding-name>clustered-retry-stateless-rmi-
invoker</invoker-proxy-binding-name>
        <jndi-name>nextgen_RetryInterceptorStatelessSession</jndi-name>
      </invoker>
    </invoker-bindings>
    <clustered>true</clustered>
  </session>
  <invoker-proxy-binding>
    <name>clustered-retry-stateless-rmi-invoker</name>
    <invoker-mbean>jboss:service=invoker,type=jrmpha</invoker-mbean>
    <proxy-factory>org.jboss.proxy.ejb.ProxyFactoryHA</proxy-factory>
    <proxy-factory-config>
      <client-interceptors>
        <home>
          <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
          <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
          <interceptor>org.jboss.proxy.ejb.RetryInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </home>
      </client-interceptors>
    </proxy-factory-config>
  </invoker-proxy-binding>
  <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
    <interceptor>org.jboss.proxy.ejb.RetryInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
  </bean>
```

```
</client-interceptors>
</proxy-factory-config>
</invoker-proxy-binding>
</jboss>
```

## 22.4.5. JNDI ルックアッププロセス

HA プロキシを回復するために、RetryInterceptor は JNDI でルックアップを行います。つまり、内部的に新しい InitialContext が作成され、JNDI ルックアップが行われます。ただし、ルックアップが成功するには、ネーミングサーバーを見つけるよう InitialContext を適切に設定する必要があります。

RetryInterceptor は以下のステップを経て適切なネーミング環境プロパティを調べます。

1. また独自の静的 retryEnv フィールドも調べます。このフィールドは RetryInterceptor.setRetryEnv(Properties) の呼び出しを用いてクライアントコードより設定できますが、この設定方法には 2 つの欠点があります。1 つ目の欠点は、JBoss 固有の呼び出しをクライアントコードに導入することによって移植性が減少することです。もう 1 つの欠点は、静的フィールドが使用されているため 1 つの JVM に対して 1 つの設定しか使用できないことです。
2. retryEnv フィールドが null の場合は、org.jboss.naming.NamingContextFactory クラスによって ThreadLocal にバインドされた環境プロパティがチェックされます。このクラスをネーミングコンテキストファクトリとして使用するには、jndi.properties でプロパティ java.naming.factory.initial=org.jboss.naming.NamingContextFactory を適切に設定します。この利点は org.jboss.naming.NamingContextFactory を jndi.properties ファイルの設定オプションで簡単に使用できるため、java コードが影響を受けないことです。欠点はネーミングプロパティが ThreadLocal に格納されるため、InitialContext を最初に作成したスレッドに対してのみ可視の状態にあることです。
3. 上記のいずれの方法でもネーミング環境プロパティが設定されない場合はデフォルトの InitialContext が使用されます。ネーミングサーバーへ接続できない場合、デフォルトでは HA-JNDI ネーミングサーバーを検索するため InitialContext がマルチキャストディスカバリへフォールバックしようとします。HA-JNDI のマルチキャストディスカバリの詳細は、[21章 クラスター化 JNDI サービス](#) を参照してください。

## 22.4.6. SingleRetryInterceptor

RetryInterceptor は多くのケースで役に立ちますが、成功するまで JNDI で HA プロキシをルックアップの継続をする点が欠点です。何らかの理由で成功しない場合、この処理が永遠に続き RetryInterceptor をトリガーした EJB 呼び出しが戻らないことになります。多くのクライアントアプリケーションでは対応できません。結果として JBoss で RetryInterceptor はクラスター化された EJB のデフォルトのクライアントインターセプタースタックの一部にはなりません。

以前のリリースでは、新しい再試行インターセプター org.jboss.proxy.ejb.SingleRetryInterceptor が導入されました。このバージョンは RetryInterceptor のように動作しますが、JNDI での HA プロキシの再ルックアップを一度しか行いません。この再ルックアップに失敗すると、再試行インターセプタが使用されていない場合のように EJB 呼び出しに失敗します。SingleRetryInterceptor はクラスター化された EJB のデフォルトクライアントインターセプターの一部となりました。

SingleRetryInterceptor の欠点は、サーバーを利用できないクラスターの再起動時に再試行が行われると再試行に失敗し、それ以降再試行が行われなくなることです。

## 第23章 クラスター化した ENTITY EJB

JBoss Enterprise Application Platform クラスターでは、エンティティ Bean のインスタンスは全ノードで同期される必要があります。エンティティ Bean がリモートサービスを提供する場合、そのサービスメソッドも負荷分散される必要があります。

### 23.1. EJB 3.0 内の エンティティ BEAN

EJB 3.0 では、エンティティ Bean は主に永続データモデルとして機能し、リモートサービスを提供しません。そのため、EJB 3.0 内のエンティティ Bean クラスター化サービスは負荷分散ではなく、主に分散キャッシングやレプリケーションに対応します。

#### 23.1.1. 分散型キャッシュの設定

データベースへのラウンドトリップを避ける為に、ご使用のエンティティ用のキャッシュを使用することができます。JBoss EJB 3.0 エンティティ bean は、2 次キャッシュをサポートする Hibernate によって実装されます。2 次キャッシュは以下のような機能を提供します。

- キャッシュが有効になっている エンティティ Bean インスタンスをエンティティマネージャーを用いてデータベースへ永続化させると、エンティティはキャッシュの中に挿入されます。
- エンティティ Bean インスタンスを更新し、その変更をエンティティマネージャーよりデータベースに保存すると、エンティティはキャッシュ内で更新されます。
- エンティティ Bean インスタンスをエンティティマネージャーを用いてデータベースから削除すると、そのエンティティはキャッシュから削除されます。
- エンティティマネージャーを介してデータベースからキャッシュ化したエンティティをロードする時、そのエンティティがデータベース内にまだ存在しない場合、そのエンティティはキャッシュに挿入されます。

キャッシングエンティティのリージョンと同様に、2 次キャッシュにもキャッシングコレクション、クエリ、タイムスタンプ向けのリージョンが含まれています。JBoss EJB 3.0 の実装に使用される Hibernate 設定は JBoss Cache を基礎となる 2 次キャッシュ実装として使用します。

2 次キャッシュは、EJB3 デプロイメントの **persistence.xml** より設定されます。

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/persistence"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="tempdb" transaction-type="JTA">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.cache.use_second_level_cache"
value="true"/>
      <property name="hibernate.cache.use_query_cache" value="true"/>
      <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
      <!-- region factory specific properties -->
      <property name="hibernate.cache.region.jbc2.cachefactory"
value="java:CacheManager"/>
      <property name="hibernate.cache.region.jbc2.cfg.entity" value="mvcc-
entity"/>
    </properties>
  </persistence-unit>
</persistence>
```

```
<property name="hibernate.cache.region.jbc2.cfg.collection"
value="mvcc-entity"/>
</properties>
</persistence-unit>
</persistence>
```

### hibernate.cache.use\_second\_level\_cache

エンティティとコレクションの 2 次キャッシングを有効にします。

### hibernate.cache.use\_query\_cache

クエリの 2 次キャッシングを有効にします。

### hibernate.cache.region.factory\_class

リージョン固有のキャッシング動作を指示する **RegionFactory** 実装を定義します。Hibernate には、共有と多重の 2 種類の JBoss Cache ベース 2 次キャッシュが同梱されています。

共有リージョンファクトリは全てのキャッシュリージョンに同じキャッシュを使用します。これは、以前の Hibernate バージョンのレガシー CacheProvider 実装の動作と同様です。

Hibernate には 2 つの共有リージョンファクトリ実装が同梱されています。

### org.hibernate.cache.jbc2.SharedJBossCacheRegionFactory

全てのキャッシュリージョンに対し、新たにインスタンス化された CacheManager より単一の JBoss Cache 設定を使用します。

表23.1 SharedJBossCacheRegionFactory のその他のプロパティ

プロパティ	デフォルト	詳細
hibernate.cache.region.jbc2.cfg.shared	treecache.xml	JBoss Cache 設定が含まれるクラスパスまたはファイルシステムリソースです。
hibernate.cache.region.jbc2.cfg.jgroups.stacks	org/hibernate/cache/jbc2/builder/jgroups-stacks.xml	JGroups プロトコルスタック設定が含まれるクラスパスまたはファイルシステムリソースです。

### org.hibernate.cache.jbc2.JndiSharedJBossCacheRegionFactory

全てのキャッシュリージョンに対し、既存の JNDI へバインドされた既存の CacheManager より単一の JBoss Cache 設定を使用します。

表23.2 JndiSharedJBossCacheRegionFactory のその他のプロパティ

プロパティ	デフォルト	詳細
hibernate.cache.region.jbc2.cfg.shared	<b>Required</b>	共有 <b>Cache</b> インスタンスがバインドされる JNDI 名です。

多重リージョンファクトリは、各キャッシュリージョンに対して最適化された設定を使用し、個別のキャッシュインスタンスを使用します。

表23.3 多重リージョンファクトリ実装の一般的なプロパティ

プロパティ	デフォルト	詳細
hibernate.cache.region.jbc2.cfg.entity	optimistic-entity	エンティティキャッシュリージョンに使用される JBoss Cache 設定です。この他に、mvcc-entity、pessimistic-entity、mvcc-entity-repeatable、optimistic-entity-repeatable、pessimistic-entity-repeatable を設定することができます。
hibernate.cache.region.jbc2.cfg.collection	optimistic-entity	コレクションキャッシュリージョンに使用される JBoss Cache 設定です。通常、コレクションキャッシュリージョンはエンティティキャッシュリージョンと同じ設定を使用します。
hibernate.cache.region.jbc2.cfg.query	local-query	クエリキャッシュリージョンに使用される JBoss Cache 設定です。デフォルトでは、キャッシュされたクエリ結果はレプリケートされません。この他に、replicated-query を設定することができます。
hibernate.cache.region.jbc2.cfg.timestamps	timestamps-cache	タイムスタンプキャッシュリージョンに使用される JBoss Cache 設定です。クエリキャッシングが使用される場合、クエリキャッシュがレプリケートしなくても、対応するタイムスタンプキャッシュはレプリケートしなければなりません。タイムスタンプキャッシュリージョンはクエリキャッシュと同じキャッシュを共有してはいけません。

Hibernate には 2 つの共有リージョンファクトリ実装が同梱されています。

#### org.hibernate.cache.jbc2.MultiplexedJBossCacheRegionFactory

キャッシュリージョン毎に、新たにインスタンス化された CacheManager より個別の JBoss Cache 設定を使用します。

表23.4 MultiplexedJBossCacheRegionFactory のその他のプロパティ

プロパティ	デフォルト	詳細
hibernate.cache.region.jbc2.configs	org/hibernate/cache/jbc2/builder/jbc2-configs.xml	JBoss Cache 設定が含まれるクラスパスまたはファイルシステムリソースです。
hibernate.cache.region.jbc2.cfg.jgroups.stacks	org/hibernate/cache/jbc2/builder/jgroups-stacks.xml	JGroups プロトコルスタック設定が含まれるクラスパスまたはファイルシステムリソースです。

### org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory

キャッシュリージョン毎に、JNDI にバインドされた CacheManager より個別の JBoss Cache 設定を使用します。「JBoss Enterprise Application Platform の CacheManager サービス」を参照してください。

表23.5 JndiMultiplexedJBossCacheRegionFactory のその他のプロパティ

プロパティ	デフォルト	詳細
hibernate.cache.region.jbc2.cachefactory	<b>Required</b>	<b>CacheManager</b> インスタンスがバインドされる JNDI 名です。

これで、分散型キャッシングの EJB 3.0 エンティティ bean をサポートするための JBoss Cache を設定したことになります。キャッシュサービスを使用するには、まだ個別の エンティティ bean を設定する必要があります。

#### 23.1.2. キャッシュ用の エンティティ bean の設定

次に、キャッシュするエンティティを設定する必要があります。上述の設定であっても、デフォルトでは何もキャッシュしないことになっています。キャッシュされる必要がある エンティティ bean にタグ付けするため **@org.hibernate.annotations.Cache** アノテーションを使用します。

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
public class Account implements Serializable
{
    // ...
}
```

大体の目安として、あまり変更がなく頻繁に読み込まれるオブジェクトをキャッシュします。適切な JBoss Cache 設定ファイル **jboss-cache-manager-jboss-beans.xml** など) にある各エンティティ Bean に対してキャッシュを細かく調整することができます。例えば、キャッシュのサイズを指定することができます。キャッシュ内にあるオブジェクトが多すぎる場合、キャッシュは設定通りに最も古いオブジェクトや使用頻度が最も低いオブジェクトをエビクトして新しいオブジェクトのスペースを確保することができます。例えば、**persistence.xml** で指定された **region\_prefix** が **myprefix** である場合、**com.mycompany.entities.Account** エンティティ bean のキャッシュリージョンのデフォルト名は **/myprefix/com/mycompany/entities/Account** になります。



```

<bean name="..." class="org.jboss.cache.config.Configuration">
    ...
    <property name="evictionConfig">
        <bean class="org.jboss.cache.config.EvictionConfig">
            <property name="wakeupInterval">5000</property>
            <!-- Overall default -->
            <property name="defaultEvictionRegionConfig">
                <bean class="org.jboss.cache.config.EvictionRegionConfig">
                    <property name="regionName">/</property>
                    <property name="evictionAlgorithmConfig">
                        <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                            <!-- Evict LRU node once we have more than this number of
nodes -->
                                <property name="maxNodes">10000</property>
                                <!-- And, evict any node that hasn't been accessed in this
many seconds -->
                                    <property name="timeToLiveSeconds">1000</property>
                                    <!-- Don't evict a node that's been accessed within this
many seconds.
                                Set this to a value greater than your max expected
transaction length. -->
                                        <property name="minTimeToLiveSeconds">120</property>
                                </bean>
                            </property>
                        </bean>
                    </property>
                </bean>
            </property>
            <property name="evictionRegionConfigs">
                <list>
                    <bean class="org.jboss.cache.config.EvictionRegionConfig">
                        <property
name="regionName">/myprefix/com/mycompany/entities/Account</property>
                        <property name="evictionAlgorithmConfig">
                            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                                <property name="maxNodes">10000</property>
                                <property name="timeToLiveSeconds">5000</property>
                                <property name="minTimeToLiveSeconds">120</property>
                            </bean>
                        </property>
                    </bean>
                    ...
                </list>
            </property>
        </bean>
    </property>
</bean>

```

エンティティ Bean クラスに対してキャッシュリージョンを指定しないと、上記の通り、**defaultEvictionRegionConfig** を使用してそのクラスのすべてのインスタンスがキャッシュされます。エンティティクラスの完全修飾名より自動的に作成するのではなく、**@Cache** アノテーションは、エンティティが保存されるキャッシュリージョンを指定できるようにする任意の属性「region」を公開します。

```

@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL, region = "Account")
public class Account implements Serializable

```

```
{
  // ...
}
```

除外設定は以下のようになります。

```
<bean name="..." class="org.jboss.cache.config.Configuration">
  ...
  <property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
      <property name="wakeupInterval">5000</property>
      <!-- Overall default -->
      <property name="defaultEvictionRegionConfig">
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/</property>
          <property name="evictionAlgorithmConfig">
            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
              <property name="maxNodes">5000</property>
              <property name="timeToLiveSeconds">1000</property>
              <property name="minTimeToLiveSeconds">120</property>
            </bean>
          </property>
        </bean>
      </property>
      <property name="evictionRegionConfigs">
        <list>
          <bean class="org.jboss.cache.config.EvictionRegionConfig">
            <property name="regionName">/myprefix/Account</property>
            <property name="evictionAlgorithmConfig">
              <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                <property name="maxNodes">10000</property>
                <property name="timeToLiveSeconds">5000</property>
                <property name="minTimeToLiveSeconds">120</property>
              </bean>
            </property>
          </bean>
          ...
        </list>
      </property>
    </bean>
  </property>
</bean>
```

### 23.1.3. クエリ結果のキャッシュ

EJB3 のクエリ API を使用すると、指定されたクエリの結果 (エンティティ bean の 1 次キーの集まりやスカラー値の集まりなど) を 2 次キャッシュに保存できます。ここでは、bean に名前付きクエリのアノテーションを付ける簡単な例を説明し、Hibernate でクエリをキャッシュするための Hibernate 固有のヒントも提供します。

最初に persistence.xml で Hibernate がクエリキャッシュを有効にするよう指定する必要があります。

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

次に、エンティティに関連付けられた名前付きクエリを作成し、そのクエリの結果をキャッシュすることを Hibernate に指示します。

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL, region = "Account")
@NamedQueries(
{
    @NamedQuery(
        name = "account.bybranch",
        query = "select acct from Account as acct where acct.branch = ?1",
        hints = { @QueryHint(name = "org.hibernate.cacheable", value = "true")
    }
})
public class Account implements Serializable
{
    // ... ..
}
```

@NamedQueries、@NamedQuery、および @QueryHint アノテーションはすべて javax.persistence パッケージに含まれます。EJB3 クエリの使用法とクエリをキャッシュするよう EJB3 に指示する方法の詳細については、Hibernate と EJB3 のドキュメントを参照してください。

デフォルトでは、Hibernate は JBoss Cache のクエリ結果を <region\_prefix>/org/hibernate/cache/StandardQueryCache という名前の領域に保存します。これに基づき、クエリ結果に対して異なるエビクション処理を設定できます。**persistence.xml** でリージョンのプレフィックスが myprefix に設定されている場合は、以下のようなエビクション処理を作成できます。

```
<bean name="..." class="org.jboss.cache.config.Configuration">
    ... ..
    <property name="evictionConfig">
        <bean class="org.jboss.cache.config.EvictionConfig">
            <property name="wakeupInterval">5000</property>
            <!-- Overall default -->
            <property name="defaultEvictionRegionConfig">
                <bean class="org.jboss.cache.config.EvictionRegionConfig">
                    <property name="regionName"/></property>
                    <property name="evictionAlgorithmConfig">
                        <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                            <property name="maxNodes">5000</property>
                            <property name="timeToLiveSeconds">1000</property>
                            <property name="minTimeToLiveSeconds">120</property>
                        </bean>
                    </property>
                </bean>
            </property>
            <property name="evictionRegionConfigs">
                <list>
                    <bean class="org.jboss.cache.config.EvictionRegionConfig">
                        <property
name="regionName">/myprefix/Account</property>
                        <property name="evictionAlgorithmConfig">
                            <bean
```

```

class="org.jboss.cache.eviction.LRUAlgorithmConfig">
    <property name="maxNodes">10000</property>
    <property
name="timeToLiveSeconds">5000</property>
    <property
name="minTimeToLiveSeconds">120</property>
    </bean>
</property>
</bean>
<bean class="org.jboss.cache.config.EvictionRegionConfig">
    <property
name="regionName">/myprefix/org/hibernate/cache/StandardQueryCache</proper
ty>
    <property name="evictionAlgorithmConfig">
    <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
    <property name="maxNodes">100</property>
    <property name="timeToLiveSeconds">600</property>
    <property
name="minTimeToLiveSeconds">120</property>
    </bean>
    </property>
    </bean>
</list>
</property>
</bean>
</property>
</bean>

```

上記の `@NamedQuery.hints` 属性はベンダー固有の `@QueryHints` の配列を値とします。Hibernate は `"org.hibernate.cacheRegion"` クエリヒントを許可します。この値はデフォルトの `/org/hibernate/cache/StandardQueryCache` の代わりに使用するキャッシュリージョンの名前です。例は次の通りです。

```

@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL, region = "Account")
@NamedQueries(
{
    @NamedQuery(
        name = "account.bybranch",
        query = "select acct from Account as acct where acct.branch = ?1",
        hints =
        {
            @QueryHint(name = "org.hibernate.cacheable", value = "true"),
            @QueryHint(name = "org.hibernate.cacheRegion", value = "Queries")
        }
    )
})
public class Account implements Serializable
{
    // ... ...
}

```

関連する除外設定:

```

<bean name="..." class="org.jboss.cache.config.Configuration">
    ... ..
    <property name="evictionConfig">
        <bean class="org.jboss.cache.config.EvictionConfig">
            <property name="wakeupInterval">5000</property>
            <!-- Overall default -->
            <property name="defaultEvictionRegionConfig">
                <bean class="org.jboss.cache.config.EvictionRegionConfig">
                    <property name="regionName">/</property>
                    <property name="evictionAlgorithmConfig">
                        <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                            <property name="maxNodes">5000</property>
                            <property name="timeToLiveSeconds">1000</property>
                            <property name="minTimeToLiveSeconds">120</property>
                        </bean>
                    </property>
                </bean>
            </property>
            <property name="evictionRegionConfigs">
                <list>
                    <bean class="org.jboss.cache.config.EvictionRegionConfig">
                        <property
name="regionName">/myprefix/Account</property>
                        <property name="evictionAlgorithmConfig">
                            <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                                <property name="maxNodes">10000</property>
                                <property
name="timeToLiveSeconds">5000</property>
                                <property
name="minTimeToLiveSeconds">120</property>
                            </bean>
                        </property>
                    </bean>
                    <bean class="org.jboss.cache.config.EvictionRegionConfig">
                        <property
name="regionName">/myprefix/Queries</property>
                        <property name="evictionAlgorithmConfig">
                            <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                                <property name="maxNodes">100</property>
                                <property name="timeToLiveSeconds">600</property>
                                <property
name="minTimeToLiveSeconds">120</property>
                            </bean>
                        </property>
                    </bean>
                    ... ..
                </list>
            </property>
        </bean>
    </property>
</bean>

```

## 23.2. EJB 2.X 内の ENTITY BEAN

最初に、2.x エンティティ Bean はクラスター化しない方がよいこと特記しておきます。クラスター化すると、クラスター化されたりリモートオブジェクトに対するリモートオブジェクトとして使用するには粒度が細かすぎる要素を公開し、重大なデータ同期化に関する問題が発生する原因となります。読み専用ノードや、読み書き可能なノード 1 つと他の読み専用ノードをキャッシュ無効化サービスで同期化するような特殊な場合以外は、EJB 2.x エンティティ Bean をクラスター化しないでください。

クラスター化したエンティティ bean を使用するには、クラスター化された HA-JNDI から EJB 2.x リモート bean 参照をルックアップすること以外、アプリケーションが行わなければならないことは特にありません。

EJB 2.x エンティティ bean をクラスター化するには、**<clustered>** 要素をアプリケーションの **jboss.xml** 記述子ファイルに追加する必要があります。以下に標準的な **jboss.xml** ファイルを示します。

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>nextgen.EnterpriseEntity</ejb-name>
      <jndi-name>nextgen.EnterpriseEntity</jndi-name>
      <clustered>True</clustered>
      <cluster-config>
        <partition-name>DefaultPartition</partition-name>
        <home-load-balance-
policy>org.jboss.ha.framework.interfaces.RoundRobin</home-load-balance-
policy>
        <bean-load-balance-
policy>org.jboss.ha.framework.interfaces.FirstAvailable</bean-load-
balance-policy>
      </cluster-config>
    </entity>
  </enterprise-beans>
</jboss>
```

EJB 2.x エンティティ bean はロードバランス化したりリモート起動の為にクラスター化されます。全ての bean インスタンスは、全てのノード上で同じコンテンツを持つように同期化されます。

しかし、クラスター化された EJB 2.x エンティティ Bean は分散型ロック機構や、分散型キャッシュを持ちません。同期化させるには、データベースレベルで行レベルロックを使用するか (CMP 仕様の **<row-lock>** 参照)、JDBC ドライバーのトランザクション分離レベルを **TRANSACTION\_SERIALIZABLE** に設定するしか方法はありません。サポートされる分散型ロック機構や分散型キャッシュがないため、エンティティ Bean は、デフォルトでコミットオプション「B」を使用します。(standardjboss.xml とコンテナ設定 Clustered CMP 2.x EntityBean、Clustered CMP EntityBean、Clustered BMP EntityBean を参照)。エンティティ Bean が読み込み専用でない場合は、コミットオプション「A」の使用は推奨できません。



### 注記

BMP (Bean Managed Persistence: Bean 管理の永続性) を使用している場合は、ご自身で同期化を実装する必要があります。

## 第24章 HTTP サービス

**mod\_jk** および **mod\_cluster**などの HTTP サービスのインストールや設定については、『HTTP Connectors Load Balancing Guide』にて詳細に説明しています。

## 第25章 JBOSS MESSAGING のクラスタリングに関する注意

クラスタ化された環境で JBoss Messaging を使用する場合の最新情報は、  
[http://www.redhat.com/docs/en-US/JBoss\\_Enterprise\\_Application\\_Platform/](http://www.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/) の『JBoss Messaging User Guide』を参照してください。



## 第26章 クラスター化されたデプロイメントのオプション

### 26.1. クラスター化シングルトンサービス

クラスター化されたシングルトンサービス (HA シングルトンとも呼ばれます) はクラスター内の複数のノードにデプロイされたサービスですが、ノードの 1 つでのみサービスを提供します。シングルトンサービスを実行するノードは通常マスターノードと呼ばれます。

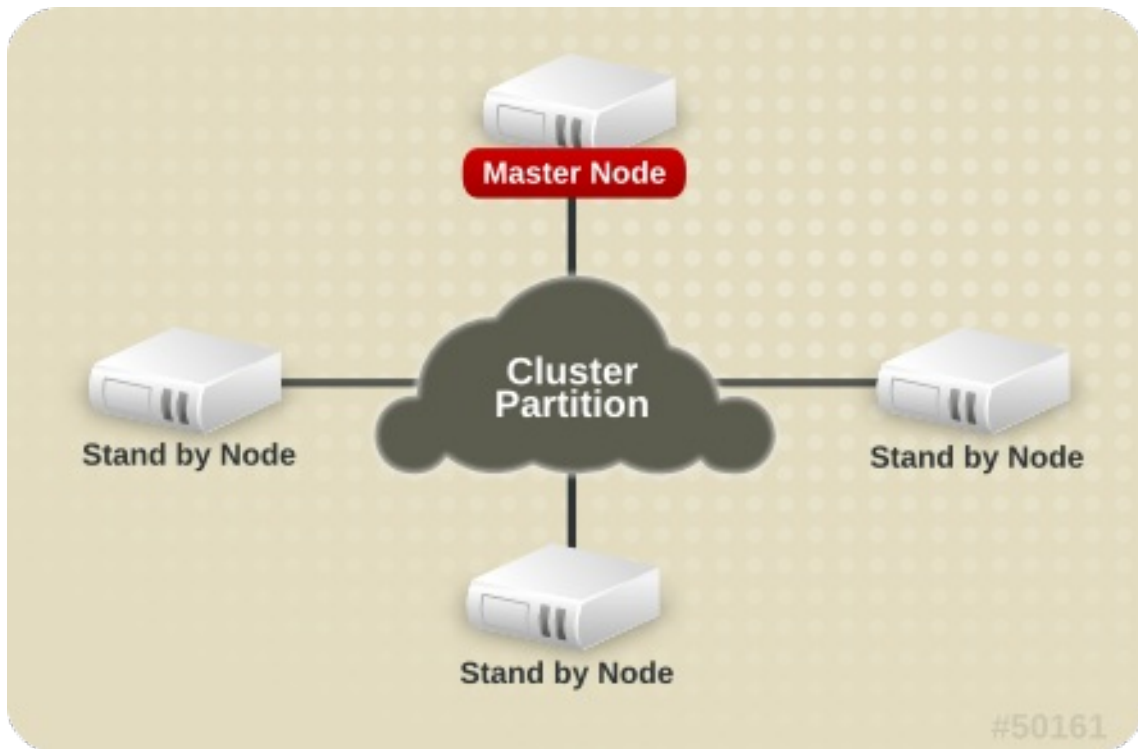


図26.1 マスターノードで障害が発生する前のトポロジ

マスターで障害が発生あるいはシャットダウンすると、残りのノードから別のマスターが選択され、その新しいマスターでサービスが再び起動されます。したがって、あるマスターが停止し、別のマスターが引き継ぐまでの短い時間を除き、サービスは常に 1 つのノードによって提供されます。

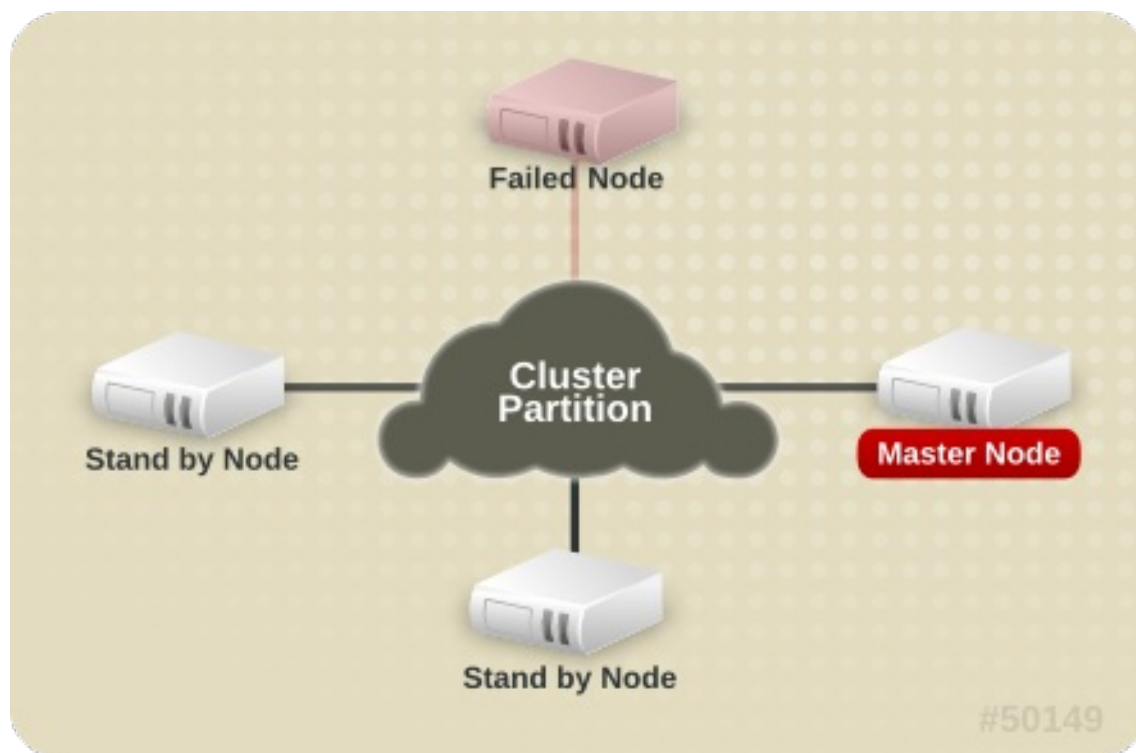


図26.2 マスターノードで障害が発生した後のトポロジ

### 26.1.1. HASingleton デプロイメントオプション

JBoss Enterprise Application Platform は、クラスター化されたシングルトンサービスのデプロイメントを容易にする複数の方法をサポートします。この項では、さまざまな方針について説明します。これらすべての方法は「はじめに」で説明した HAPartition サービスの上に構築されます。また、**HAPartition** を使用してクラスター内の異なるノードが起動および停止した時に通知を行います。これらの通知に基づいて、クラスター内の各ノードは独自に (また一貫的に) マスターノードであるかを判断し、サービスの提供を開始するかを決定します。

#### 26.1.1.1. HASingletonDeployer サービス

HA シングルトンをデプロイする最も単純で最も一般的に使用されている方法は、通常のデプロイメント (war、ear、jar やその他デプロイに通常含めるもの) を利用して、**deploy** の代わりに **\$JBOSS\_HOME/server/production/deploy-hasingleton** ディレクトリにデプロイすることです。**deploy-hasingleton** ディレクトリは **deploy** や **farm** ディレクトリに含まれないため、そのコンテンツは Enterprise Application Platform インスタンスの起動時に自動的にデプロイされません。このディレクトリのコンテンツをデプロイするのは **HASingletonDeployer** bean (これ自体は **deploy/deploy-hasingleton-jboss-beans.xml** ファイルによってデプロイされます) という特殊なサービスです。HASingletonDeployer サービスはそれ自体が HA シングルトンであり、マスターになった時にその提供されたサービスは **deploy-hasingleton** のコンテンツをデプロイし、マスターでなくなったとき (通常はサーバーのシャットダウン時) にそのサービスは **deploy-hasingleton** のコンテンツをアンデプロイします。

したがって、デプロイメントを **deploy-hasingleton** に格納することにより、デプロイメントはクラスター内のマスターノードにのみデプロイされます。マスターノードが正常にシャットダウンした場合は、シャットダウンの一部としてデプロイメントが正常にデプロイ解除されます。マスターノードで障害が発生した場合やマスターノードがシャットダウンされた場合はこのデプロイメントがマスターとして引き継いだノードにデプロイされます。

**deploy-hasingleton** の使用は非常に簡単ですが、以下の 2 つの欠点があります。

- **deploy-hasingleton** のサービスにはホットデプロイメント機能がありません。 **deploy-hasingleton** にデプロイされたサービスを再デプロイするにはサーバーを再起動する必要があります。
- マスターノードで障害が発生し、別のノードがマスターを引き継いだ場合、シングルトンサービスはサービスを提供する前にデプロイメントプロセス全体を経る必要があります。サービスのデプロイメントの複雑さや実行する開始アクティビティの種類によって異なりますが、処理にしばらく時間がかかることがあります。処理中はサービスが提供されません。

### 26.1.1.2. HASingletonController を使用した POJO デプロイメント

サービスが POJO である場合 (ear、war、jar などの J2EE デプロイメントでない場合) は、HASingletonController というサービスと共にサービスをデプロイして HA シングルトンに変えることができます。HASingletonController は HAPartition サービスと連携してクラスターを監視し、そのサービスのマスターノードであるかどうか確認します。マスターノードである場合は、サービスに対してサービスの提供を開始するメソッドが呼び出されます。マスターノードでなくなると、サービスに対してサービスの提供を中止するメソッドが呼び出されます。まず、下図を見てみましょう。

最初に HA シングルトンを作成する POJO があります。POJO は、サービスの提供を開始する時に呼び出すことができるパブリックメソッドと、サービスの提供を停止する時に呼び出す別のメソッドを公開する必要があります。

```
public interface HASingletonExampleMBean
{
    boolean isMasterNode();
}

public class HASingletonExample implements HASingletonExampleMBean
{
    private boolean isMasterNode = false;

    public boolean isMasterNode()
    {
        return isMasterNode;
    }

    public void startSingleton()
    {
        isMasterNode = true;
    }

    public void stopSingleton()
    {
        isMasterNode = false;
    }
}
```

上図の例では、メソッド名が **startSingleton** と **stopSingleton** になっていますが、メソッドには任意の名前を付けることができます。

次に、制御を行う HASingletonController とサービスを共にデプロイします。通常、以下の **META-INF/jboss-service.xml** と共に .sar ファイルにパッケージされています。

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">
    <!-- This bean is an example of a clustered singleton -->
```

```

<bean name="HASingletonExample"
class="org.jboss.ha.examples.HASingletonExample">
  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="jboss:service=HASingletonExample",

exposedInterface=org.jboss.ha.examples.HASingletonExampleMBean.class)
</annotation>
</bean>

<bean name="ExampleHASingletonController"
class="org.jboss.ha.singleton.HASingletonController">
  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="jboss:service=ExampleHASingletonController",

exposedInterface=org.jboss.ha.singleton.HASingletonControllerMBean.class,
  registerDirectly=true)</annotation>
  <property name="HAPartition"><inject bean="HAPartition"/></property>
  <property name="target"><inject bean="HASingletonExample"/></property>
  <property name="targetStartMethod">startSingleton</property>
  <property name="targetStopMethod">stopSingleton</property>
</bean>
</deployment>

```

クラスター化されたシングルトンサービスです。

deploy-ha-singleton と比較したこの方法の主な利点は、上記の例を **deploy** または **farm** に格納できるため、ホットデプロイメントやファームデプロイメントを実行できることです。また、このサービス例が複雑で時間がかかる起動要件を持っている場合、create() メソッドまたは start() メソッドで実装することが可能です。サービスがデプロイされると、JBoss は即座に create() と start() を呼び出します。ノードがマスターノードになるまで待機しないため、コントローラーが startSingleton() を実装するまで待機すれば、サービスはすぐに利用できる状態になります。

上記の例にはありませんが、**HASingletonController** は目的の起動メソッドや停止メソッドに対するオプションの引数をサポートします。引数を指定するには、起動メソッドの場合は **targetStartMethodArgument** プロパティを使用し、停止メソッドの場合は **TargetStopMethodArgument** プロパティを使用します。現在、ストリング値のみがサポートされています。

### 26.1.1.3. バリアを使用した HASingleton デプロイメント

deploy-hasingleton のコンテンツがデプロイまたはデプロイ解除された時に (現在のノードがマスターになった時など) 起動または停止する deploy または farm 内に通常デプロイされるサービスは、バリアーサービスの依存関係のみを指定する必要があります。

```
<depends>jboss.ha:service=HASingletonDeployer,type=Barrier</depends>
```

この仕組みは、BarrierController が HASingletonDeployer と共にデプロイされ、そこから JMX 通知をリッスンします。BarrierController はシステムの JMX 通知を受信するためにサブスクライブできる比較的単純な Mbean です。また、受け取った通知を使用して、動的に作成された Barrier という Mbean のライフサイクルを管理します。BarrierController がデプロイされると、Barrier はインスタンス化されてから登録され、CREATE ステートになります。その後、一致する JMX 通知を受け取ると BarrierController が Barrier を起動または終了します。したがって、他のサービスは通常の <depends> タグを使用して Barrier Bean に依存するだけで良いのです。これらのサービスは Barrier と並行して開始または停止します。BarrierController がアンデプロイされると、Barrier も破棄されます。

これにより `deploy-hasingleton` を使用した方法とは別の方法が提供され、`deploy-hasingleton` のコンテンツをすべてのノードに手動でコピーする必要がありますがファームリングを使用してサービスを分散できます。

その一方でバリアー依存のサービスはすべてのノードでインスタンス化および作成されます (つまり、任意の `create()` メソッドが呼び出されます) が、マスターノードでのみ開始されます。これは複数のノードのいずれかに `deploy-hasingleton` ディレクトリのコンテンツのみをデプロイ (インスタンス化/作成/開始) する `deploy-hasingleton` の方法とは異なります。

したがって、バリアーに依存するサービスは `create()` のステップ内で必要最低限の作業を行うか、まったく作業を行わないようにし、`start()` を使用して作業を行うようにする必要があります。



### 注記

`Barrier` は依存するサービスの起動や停止を制御しますが、**`BarrierController`** が破棄あるいはデプロイ解除された時のみ発生するサービスの破棄については制御しません。そのため、サービスの通常のデプロイ解除操作として **`Barrier`** を使用し、破棄しなければならないサービスを制御しようとしても (**`EJBContainer`** など)、思い通りには動作しません。

## 26.1.2. マスターノードの決定

多くのクラスター化シングルトン管理方法は、クラスターの各ノードが独自にクラスター内の変更へ応じ、マスターノードであるかを正しく判断することに依存しています。実際、どのように実現されるのでしょうか。

クラスターの各メンバーに対して、`HAPartition` サービスは現在のクラスターメンバーの順序化されたリストである `CurrentView` という属性を保持します。ノードがクラスターに参加し、クラスターから離れると、`JGroups` によってクラスターの残りの各メンバーが更新されたビューを受け取るようにします。現在のビューは `JMX` コンソールに移動し、**`jboss:service=DefaultPartition`** MBean の `CurrentView` 属性を確認します。クラスターの各メンバーは同じビューを持ち、メンバーの順番も同じです。

たとえば、ノード A から D までの 4 つのノードクラスターを持ち、現在のビューを {A, B, C, D} と表すことができるとします。通常、ビューのノードの順序はノードがクラスターに参加した順序を反映します (ただし、常にそうなるとは限らず、該当しない場合もあります)。

また、何らかの理由で B を除くクラスターにデプロイされた `Foo` という名前のシングルトンサービス (**`HASingletonController`** など) があるとします。**`HAPartition`** サービスはどのサービスがどこにデプロイされているかをビューの順序で表すレジストリをクラスター全体で保持します。したがって、クラスター内の各ノードでは、**`HAPartition`** サービスは `Foo` サービスに関するビューが {A, C, D} (B はなし) であることを認識します。

`Foo` サービスのクラスタートポロジに変更があると、**`HAPartition`** サービスは `Foo` へのコールバックを呼び出し、新しいトポロジについて通知します。例えば、`Foo` が D で起動した時に、A、C、D で実行されているすべての `Foo` サービスが、`Foo` の新しいビューが {A, C, D} であると通知するコールバックを受け取ったとします。この場合、コールバックにより、各ノードが独自にマスターであるかを判断できる十分な情報をノードに提供します。「[HA シングルトンの選出ポリシー](#)」の通り、各ノードの `Foo` サービスは **`HAPartition`** の **`HASingletonElectionPolicy`** を使用してマスターであるかを判断します。

A に障害が発生あるいはシャットダウンすると、`Foo` の新しいビューが {C, D} であるというコールバックを C と D の `Foo` が受け取ります。この時、C がマスターになります。A が再起動すると、`Foo` の新しいビューが {C, D, A} であるというコールバックを A、C、D が受け取りますが、C が引き続きマスターとなります。A が前のマスターであっても、再度マスターになるわけではありません。

### 26.1.2.1. HA シングルトンの選出ポリシー

**HASingletonElectionPolicy** オブジェクトは、クラスタートポロジの変更後、HA シングルトンの代わりに利用可能なノードのリストからマスターノードを選出します。

```
public interface HASingletonElectionPolicy
{
    ClusterNode elect(List<ClusterNode> nodes);
}
```

JBoss Enterprise Application Platform には 2 つの選出ポリシーが同梱されます。

#### **HASingletonElectionPolicySimple**

このポリシーはマスターノードを相対年代で選択します。希望の年代は、利用可能ノードリストのインデックスに相当する **position** プロパティより設定します。**position = 0** はデフォルトで最も古いノードを意味し、**position = 1** は 2 番目に古いノードを意味します。**position** に負の数字を使用してノードの新しさを示すこともできます。利用可能ノードのリストは環状リストであると考えてみてください。**position = -1** は最新のノードを意味し、**position = -2** は 2 番目に新しいノードを意味します。

```
<bean class="org.jboss.ha.singleton.HASingletonElectionPolicySimple">
    <property name="position">-1</property>
</bean>
```

#### **PreferredMasterElectionPolicy**

このポリシーは、**HASingletonElectionPolicySimple** を拡張し、希望のノードを設定できるようにします。**host:port** または **address:port** と指定された **preferredMaster** プロパティは、使用できる場合にマスターとなるべきノードを特定します。希望のノードが使用できない場合、選出ポリシーが前述の通り動作します。

```
<bean class="org.jboss.ha.singleton.PreferredMasterElectionPolicy">
    <property name="preferredMaster">server1:12345</property>
</bean>
```

## 26.2. ファーミングデプロイメント

アプリケーションをクラスター内にデプロイする最も簡単な方法はファーミングサービスを使用することです。ファーミングサービスを使用すると、アプリケーション (アーカイブファイルまたは展開形式の EAR、WAR、SAR など) をクラスターメンバの **all/farm/** ディレクトリへデプロイすることができます。アプリケーションは自動的に同じクラスター内の全ノードで複製されます。ノードが後にクラスタに参加する場合、クラスター内のファーミングデプロイされたアプリケーションがすべてプラインされ起動時にローカルにデプロイされます。実行中のクラスター化されたサーバーノードの **farm/** ディレクトリよりアプリケーションを削除すると、アプリケーションはローカルでアンデプロイされ、他すべてのクラスター化されたサーバーノードにある **farm/** ディレクトリより削除されます (アンデプロイがトリガされます)。

JBoss Enterprise Application Platform の **all** 設定ではファーミングがデフォルトで有効になっているため、自分で設定する必要はありません。必須の **farm-deployment-jboss-beans.xml** 設定ファイルと **timestamps-jboss-beans.xml** 設定ファイルは **deploy/cluster** ディレクトリにあります。



す。カスタム設定でファームングを有効にしたい場合は JBoss deploy ディレクトリ **\$JBoss\_HOME/server/\$your\_own\_config/deploy/cluster** にこれらのファイルをコピーするだけです。カスタム設定ではクラスタリングが有効になっているようにしてください。

ファームングサービスをカスタマイズする必要はほとんどありませんが、**FarmProfileRepositoryClusteringHandler** Bean よりカスタマイズすることができます。プロパティとデフォルト値は次のようになります。

```
<bean name="FarmProfileRepositoryClusteringHandler"
      class="org.jboss.profileservice.cluster.repository.
      DefaultRepositoryClusteringHandler">

  <property name="partition"><inject bean="HAPartition"/></property>
  <property name="profileDomain">default</property>
  <property name="profileServer">default</property>
  <property name="profileName">farm</property>
  <property name="immutable">false</property>
  <property name="lockTimeout">60000</property><!-- 1 minute -->
  <property name="methodCallTimeout">60000</property><!-- 1 minute -->
  <property name="synchronizationPolicy"><inject
  bean="FarmProfileSynchronizationPolicy"/></property>
</bean>
```

- **partition** は必須の属性で、 クラスター内の通信でファームサービスが使用する HAPartition サービスを挿入します。
- **profile[Domain|Server|Name]** は、 このハンドラーが目的とするサーバープロファイルを特定するのに使用されます。
- **immutable** は、 ノードがコンテンツの変更をクラスターへプッシュすることを許可するかを示します。 **true** の値は、 **synchronizationPolicy** を **org.jboss.system.server.profileservice.repository.clustered.sync.ImmutableSynchronizationPolicy** に設定することに相当します。
- **lockTimeout** はクラスター全体のロック取得に対して待機する時間 (ミリ秒単位) を定義します。
- **methodCallTimeout** は、 リモートクラスターノード上の呼び出しに対して待機する時間 (ミリ秒単位) を定義します。
- **synchronizationPolicy** は、 クラスターへの参加を試みるノードやノードの統合より、 どのようにクラスターコンテンツを追加、 再生、 更新、 削除するかを決定します。 ポリシーは「権限のある」ノード上で確認されます (クラスター上のサービスのマスターノードなど)。 再生 (Reincarnation) は、 新たに起動されたノードの **farm** ディレクトリに以前ファームングサービスによって削除されたサービスが存在することで、 実行されていない時に削除されると起動するノードに残存することがあります。 デフォルトの **synchronizationPolicy** は次のように定義されます。

```
<bean name="FarmProfileSynchronizationPolicy"
      class="org.jboss.profileservice.cluster.repository.
      DefaultSynchronizationPolicy">
  <property name="allowJoinAdditions"><null/></property>
  <property name="allowJoinReincarnations"><null/></property>
  <property name="allowJoinUpdates"><null/></property>
  <property name="allowJoinRemovals"><null/></property>
  <property name="allowMergeAdditions"><null/></property>
```

```
<property name="allowMergeReincarnations"><null/></property>
<property name="allowMergeUpdates"><null/></property>
<property name="allowMergeRemovals"><null/></property>
<property name="developerMode">false</property>
<property name="removalTrackingTime">2592000000</property><!-- 30
days -->
<property name="timestampService"><inject
bean="TimestampDiscrepancyService"/></property>
</bean>
```

- **allow[Join|Merge][Additions|Reincarnations|Updates|Removals]** は、要求への固定応答を定義し、結合されたノードや統合されたノードからの追加、再生、更新、削除を可能にします。
- **developerMode** は、すべての変更を許可する寛大な同期化ポリシーを有効します。  
developerModeを有効にすることは、上記の各プロパティを **true** に設定することに相当するため、開発環境用の設定となります。
- **removalTrackingTime** は、再生の検知に使用するため、ポリシーが削除されたアイテムを記憶すべき期間 (ミリ秒単位) を定義します。
- **timestampService** は、クラスターの現在および過去のメンバーに対するシステムクロックの不一致を推測し追跡します。デフォルト実装は **timestamps-jboss-beans.xml** に定義されます。



## 第27章 JGROUPS サービス

JGroups は JBoss Enterprise Application Platform クラスターに対する基礎のグループ通信サポートを提供します。クラスター化されたサービスと Groups との対話については、「[JGroups とのグループ通信](#)」に説明されています。本章では、この対話の詳細に重点を置き、設定の詳細やトラブルシューティングのヒントを中心に説明します。

本章は完全な JGroups ドキュメントではありません。JGroups に関する詳細を知りたい場合は、以下を参照してください。

- <http://jgroups.org/ug.html> の JGroups プロジェクトドキュメント。
- <https://www.jboss.org/community/wiki/JGroups> に掲載されている jboss.org の JGroups wiki ページ。

本章の最初の項では、多くの JGroups 設定オプションの詳細を取り上げます。JBoss Enterprise Application Platform にはデフォルトの JGroups 設定のセットが同梱されています。ほとんどのアプリケーションはデフォルト設定でそのまま動作します。ネットワーク要件やパフォーマンス要件が特別なアプリケーションをデプロイする場合のみ設定を変更する必要があります。

### 27.1. JGROUPS チャネルのプロトコルスタックを設定

JGroups のフレームワークは、クラスターでノード間のピアツーピア通信を可能にするサービスを提供します。通信は通信チャンネル上で発生します。チャンネルはネットワーク通信プロトコルのスタックより構築され、各チャンネルはチャンネル全体の動作に特定機能を追加します。プロトコルによって提供される主な機能には、トランスポート、クラスターディスカバリ、メッセージの順序付け、無損失メッセージ配信、失敗したピアの検知、クラスターメンバーシップの管理サービスなどが含まれます。

図27.1「JGroups 内のプロトコルスタック」は、各メンバのチャンネルが JGroups プロトコルのスタックで構成される概念的クラスターです。

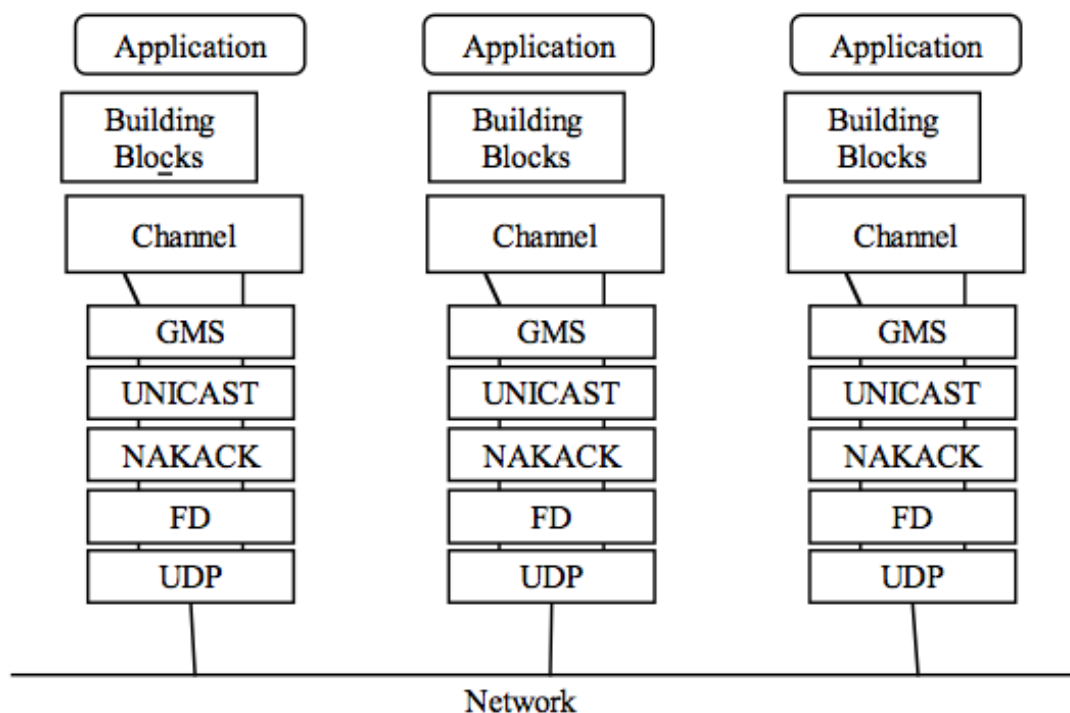


図27.1 JGroups 内のプロトコルスタック

本項では、チャンネルへ追加する動作タイプにより、最も頻繁に使用されるプロトコルについて取り上げます。各プロトコルによって公開される主な設定属性についても説明しますが、これらの属性は熟練ユーザーのみが変更すべきであるため、プロトコルの目的を理解することに重点を置きます。

JBoss Enterprise Application Platform で使用される JGroups 設定は、**\$JBoss\_HOME/server/production/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml** ファイル内でネストされた要素として表示されます。このファイルは **ChannelFactory** サービスによって構文分析され、このコンテンツを使用して、正しく設定されたチャンネルを必要とするクラスター化サービスへ提供します。**ChannelFactory** サービスの詳細は、「[チャンネルファクトリサービス](#)」を参照してください。

以下は、**jgroups-channelfactory-stacks.xml** のプロトコルスタック設定例になります。

```
<stack name="udp-async"
    description="Same as the default 'udp' stack above, except
message bundling
                    is enabled in the transport protocol
(enable_bundling=true).
                    Useful for services that make high-volume
asynchronous
                    RPCs (e.g. high volume JBoss Cache instances
configured
                    for REPL_ASYNC) where message bundling may
improve performance.">
    <config>
        <UDP
            singleton_name="udp-async"
            mcast_port="${jboss.jgroups.udp_async.mcast_port:45689}"
            mcast_addr="${jboss.partition.udpGroup:228.11.11.11}"
            tos="8"
            ucast_recv_buf_size="200000000"
            ucast_send_buf_size="640000"
            mcast_recv_buf_size="250000000"
            mcast_send_buf_size="640000"
            loopback="true"
            discard_incompatible_packets="true"
            enable_bundling="true"
            max_bundle_size="64000"
            max_bundle_timeout="30"
            ip_ttl="${jgroups.udp.ip_ttl:2}"
            thread_naming_pattern="cl"
            timer.num_threads="12"
            enable_diagnostics="${jboss.jgroups.enable_diagnostics:true}"

diagnostics_addr="${jboss.jgroups.diagnostics_addr:224.0.0.75}"
            diagnostics_port="${jboss.jgroups.diagnostics_port:7500}"

            thread_pool.enabled="true"
            thread_pool.min_threads="8"
            thread_pool.max_threads="200"
            thread_pool.keep_alive_time="5000"
            thread_pool.queue_enabled="true"
            thread_pool.queue_max_size="1000"
            thread_pool.rejection_policy="discard"

            oob_thread_pool.enabled="true"
```

```

        oob_thread_pool.min_threads="8"
        oob_thread_pool.max_threads="200"
        oob_thread_pool.keep_alive_time="1000"
        oob_thread_pool.queue_enabled="false"
        oob_thread_pool.rejection_policy="discard"/>
<PING timeout="2000" num_initial_members="3"/>
<MERGE2 max_interval="100000" min_interval="20000"/>
<FD_SOCKET/>
<FD timeout="6000" max_tries="5" shun="true"/>
<VERIFY_SUSPECT timeout="1500"/>
<BARRIER/>
<pbcast.NAKACK use_mcast_xmit="true" gc_lag="0"
        retransmit_timeout="300,600,1200,2400,4800"
        discard_delivered_msgs="true"/>
<UNICAST timeout="300,600,1200,2400,3600"/>
<pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
        max_bytes="400000"/>
<VIEW_SYNC avg_send_interval="10000"/>
<pbcast.GMS print_local_addr="true" join_timeout="3000"
        shun="true"
        view_bundling="true"
        view_ack_collection_timeout="5000"
        resume_task_timeout="7500"/>
<FC max_credits="2000000" min_threshold="0.10"
        ignore_synchronous_response="true"/>
<FRAG2 frag_size="60000"/>
<!-- pbcast.STREAMING_STATE_TRANSFER/ -->
<pbcast.STATE_TRANSFER/>
<pbcast.FLUSH timeout="0" start_flush_timeout="10000"/>
</config>
</stack>

```

**<config>** 要素にはすべての JGroups 設定データが含まれます。この情報は、概念的にソケットと似ている JGroups チャネルの設定に使用され、クラスターのピア間の通信を管理します。**<config>** 要素内部の各要素は特定の JGroups プロトコルを定義します。各プロトコルは 1 つの機能を実行し、これらの機能の組み合わせによってチャネル全体の特性が定義されます。今後の項では、一般的に使用されるプロトコルやオプションについて説明し、各プロトコルに使用できるオプションについて取り上げます。

### 27.1.1. 一般的な設定プロパティ

次のプロパティが以下で説明するすべての JGroups プロトコルによって公開されます。

- **stats** は、AS の JMX コンソールや JGroups Probe ユーティリティなどのツールによって公開される操作に対するランタイム統計をプロトコルが収集するかを判断します。収集される統計はプロトコルによって異なります。デフォルトは **true** になります。



#### 注記

JBoss Application Server 3.x および 4.x で使用された JGroups バージョンのすべてのプロトコルは **down\_thread** 属性と **up\_thread** 属性を公開しました。JBoss Application Server 5 とそれ以降のバージョンに含まれる JGroups のバージョンはこれら属性を使用しません。また、これらの属性がプロトコルに対して設定されると、**WARN** メッセージがサーバーログに書き込まれます。

## 27.1.2. トランスポートプロトコル

トランスポートプロトコルは、メッセージをネットワークに送信、あるいはメッセージをネットワークから受信します。また、トランスポートプロトコルは使用されるスレッドプールを管理し、受信メッセージをプロトコルスタックのより高いアドレスに送信します。JGroups はトランスポートプロトコルとして **UDP**、**TCP**、**TUNNEL** をサポートします。



### 注記

**UDP**、**TCP**、**TUNNEL** プロトコルは相互に排他的です。各 JGroups **Config** エLEMENT には 1 つのトランスポートプロトコルのみを指定できます。

### 27.1.2.1. UDP 設定

UDP は JGroups の推奨プロトコルです。UDP はマルチキャスト (または特別な設定の場合は複数のユニキャスト) を使用してメッセージの送受信を行います。クラスターサービスのトランスポートプロトコルに UDP を選択する場合は、JGroups **config** 要素内の **UDP** サブ要素に設定する必要があります。例は次の通りです。

```
<UDP
  singleton_name="udp-async"
  mcast_port="${jboss.jgroups.udp_async.mcast_port:45689}"
  mcast_addr="${jboss.partition.udpGroup:228.11.11.11}"
  tos="8"
  ucast_recv_buf_size="20000000"
  ucast_send_buf_size="640000"
  mcast_recv_buf_size="25000000"
  mcast_send_buf_size="640000"
  loopback="true"
  discard_incompatible_packets="true"
  enable_bundling="true"
  max_bundle_size="64000"
  max_bundle_timeout="30"
  ip_ttl="${jgroups.udp.ip_ttl:2}"
  thread_naming_pattern="cl"
  timer.num_threads="12"
  enable_diagnostics="${jboss.jgroups.enable_diagnostics:true}"

diagnostics_addr="${jboss.jgroups.diagnostics_addr:224.0.0.75}"
diagnostics_port="${jboss.jgroups.diagnostics_port:7500}"

  thread_pool.enabled="true"
  thread_pool.min_threads="8"
  thread_pool.max_threads="200"
  thread_pool.keep_alive_time="5000"
  thread_pool.queue_enabled="true"
  thread_pool.queue_max_size="1000"
  thread_pool.rejection_policy="discard"

  oob_thread_pool.enabled="true"
  oob_thread_pool.min_threads="8"
  oob_thread_pool.max_threads="200"
  oob_thread_pool.keep_alive_time="1000"
  oob_thread_pool.queue_enabled="false"
  oob_thread_pool.rejection_policy="discard"/>
```

JGroups のトランスポート設定には、複数の属性を使用できます。最初に、**UDP** プロトコルに使用できる属性を取り上げた後、**TCP** プロトコルや **TUNNEL** プロトコルが使用する属性について説明します。

**UDP** プロトコル特有の属性は次の通りです。

- **ip\_mcast** は IP マルチキャスト機能を使用するかどうかを指定します。デフォルト値は **true** です。 **false** に設定された場合は、1つのマルチキャストパケットではなく複数のユニキャストパケットが送信されます。 **UDP** プロトコルより送信されるパケットはすべて UDP データグラムです。
- **mcast\_addr** はグループ (クラスターなど) との通信に対するマルチキャストアドレス (クラス D) を指定します。JBoss AS の標準的なプロトコルスタック設定は、システムプロパティ **jboss.partition.udpGroup** の値 (設定されている場合) をこの属性に使用します。JBoss Application Server の起動時に **-u** コマンドラインスイッチを使用すると、この値を設定します。この設定属性を使用して JGroups チャンネルを適切に別のチャンネルより分離する方法については、[「JGroups チャンネルの分離」](#) を参照してください。この属性を省略すると、デフォルト値は **228.11.11.11** になります。
- **mcast\_port** はグループとマルチキャスト通信するポートを指定します。この設定属性を使用して JGroups チャンネルを適切に別のチャンネルより分離する方法については、[「JGroups チャンネルの分離」](#) を参照してください。この属性を省略すると、デフォルト値は **45688** になります。
- **mcast\_send\_buf\_size**、**mcast\_recv\_buf\_size**、**ucast\_send\_buf\_size**、**ucast\_recv\_buf\_size** は、JGroups がオペレーティングシステムより要求するソケットの送受信バッファサイズを指定します。バッファサイズを大きくすると、バッファのオーバーフローによるパケット落ちを防ぐことができます。しかし、ソケットバッファサイズはオペレーティングシステムレベルで制限されるため、希望のバッファサイズを得るにはオペレーティングシステムレベルでの設定が必要となることがあります。詳細は [「OS UDP のバッファ制限を設定してUDPのパフォーマンスを改善」](#) を参照してください。
- **bind\_port** はユニキャスト受信ソケットがバインドされるポートを指定します。デフォルトは **0** で、短命ポートを使用します。
- **port\_range** は、**bind\_port** によって特定されたポートがない場合に試行するポート数を指定します。デフォルトは **1** で、**bind\_port** のみが試行されます。
- **ip\_ttl** は、IP マルチキャストパケットの TTL (生存期間) を指定します。TTL は、マルチキャストネットワークで一般的に使用される用語ですが、この値はネットワーク装置がパケットを破棄するまでパケットがネットワークを通過する回数を表すためこの用語は実際には適切ではありません。
- **tos** はユニキャストおよびマルチキャストデータグラムを送信するトラフィッククラスを指定します。

すべてのトランスポートプロトコルで共通しているため、**TCP** や **TUNNEL** で同じ意味を持つ属性は次の通りです。

- **singleton\_name** はトランスポートプロトコル設定に固有の名前を提供します。アプリケーションサーバーの **ChannelFactory** によって使用され、同じトランスポートプロトコルを使用する異なるチャンネルによるトランスポートプロトコルインスタンスの共有をサポートします。[「JGroups 共有トランスポート」](#) を参照してください。
- **bind\_addr** はメッセージの送受信を行うインターフェースを指定します。デフォルトでは、JGroups はシステムプロパティ **jgroups.bind\_addr** の値を使用します。**-b** コマンドライン

スイッチを使用して設定することもできます。JGroups ソケットのバインドに関する詳細は、[「その他設定の問題」](#) を参照してください。

- **receive\_on\_all\_interfaces** はこのノードがすべてのインターフェースでマルチキャストをリッスンするかを指定します。デフォルト値は **false** です。マルチキャストの受信については **bind\_addr** プロパティよりも優先されます。ただし、マルチキャストの送信には引き続き **bind\_addr** (設定されている場合) が使用されます。
- **send\_on\_all\_interfaces** は、マシンに複数のネットワークインターフェースコントローラがある場合に、このノードがすべてのネットワークインターフェースコントローラから UDP パケットを送信するかどうかを指定します。同じマルチキャストメッセージが N 回送信されることを意味するため、注意して使用してください。
- **receive\_interfaces** は、マルチキャストを受信するインターフェースの一覧を指定します。マルチキャスト受信ソケットはこれらのインターフェースすべてでリッスンします。IP アドレスまたはインターフェース名をカンマで区切ったリストになります (例: **192.168.5.1,eth1,127.0.0.1**)。
- **send\_interfaces** は、マルチキャストを送信するインターフェースの一覧を指定します。マルチキャスト送信ソケットはこれらのインターフェースすべてでリッスンします。IP アドレスまたはインターフェース名をカンマで区切ったリストになります (例: **192.168.5.1,eth1,127.0.0.1**)。そのため、同じマルチキャストメッセージが N 回送信されるため、注意して使用してください。
- **enable\_bundling** は、メッセージのバンドルを有効にするかを指定します。 **true** の場合、 **max\_bundle\_size** バイトが累積されるか **max\_bundle\_time** ミリ秒が経過するまで (いずれか早く発生する期間) トランスポートプロトコルが送信メッセージをキューに入れます。次にキューに入ったメッセージを 1 つの大きなメッセージにまとめてから送信します。このメッセージは受信側で元に戻されます。デフォルト値は **false** です。

メッセージのバンドルは、送信元が受信先からの応答待機をブロックしない場合 (**REPL\_ASYNC** に対して設定された JBoss Cache インスタンスなど)、大量のメッセージ送信に使用されるチャネルに対してパフォーマンスの大幅な改善を期待できます。送信元が応答待機をブロックする必要がある場合、アプリケーションの待ち時間が大幅に長くなるため、JBoss Cache インスタンスが **REPL\_SYNC** に設定された場合など、特定の場合でのみ推奨されます。

- **loopback** は、グループにメッセージを送信するスレッド自体が、配信のためメッセージをスタックに戻すかを指定します (グループに送信されるメッセージは常に送信するノードにも配信されます)。 **false** の場合、送信するスレッドはメッセージを持ちません。トランスポートプロトコルがネットワークからメッセージを読み取るために待機し、配信にメッセージ配信プールスレッドの 1 つを使用します。デフォルトは **false** ですが、ネットワークインターフェースが停止した時にチャネルがメッセージを受信できるようにするため **true** が推奨されます。
- **discard\_incompatibe\_packets** は、別バージョンの JGroups を使用するピアから送信されたパケットを破棄するか指定します。クラスター内の各メッセージには JGroups バージョンのタグが付けられます。 **discard\_incompatible\_packets** が **true** に設置されると、別バージョンの JGroups からの受信されたメッセージは静かに破棄されます。これ以外の場合は警告がログに記録されます。メッセージは配信されません。デフォルト値は **false** です。
- **enable\_diagnostics** は、アドレス **diagnostics\_addr** とポート **diagnostics\_port** 上でマルチキャストソケットを開け、JGroup の **Probe ユーティリティ** が送信した分析要求をリッスンするかを指定します。
- 通常の受信メッセージをスタックに運ぶため JGroups が使用するスレッドプールの動作は、様々な **thread\_pool** 属性が設定します。これらの属性は、



`java.util.concurrent.ThreadPoolExecutorService` のインスタンスに対してコンストラクターの引数を提供します。上記の例では、プールの最低またはコアサイズが 8 スレッドで、最大サイズは 200 になります。9 つ以上のプールスレッドが作成されると、メッセージの伝達より返されたスレッドは、伝達する新しいメッセージが割り当てられるまで最大 5000 ミリ秒待機し、その後終了します。メッセージを伝達するスレッドがない場合、ソケット外でメッセージを読み取る個別のスレッドがメッセージをキューに入れます。キューは最大 1000 メッセージを保持することができます。キューがいっぱいの場合、ソケット外でメッセージを読み取るスレッドはメッセージを破棄します。

- 受信メッセージをプロトコルスタックへ運ぶために使用される `java.util.concurrent.ThreadPoolExecutorService` を設定するという面で、`oob_thread_pool` 属性は `thread_pool` 属性と似ています。この場合、OOB (帯域外、Out Of Band) メッセージと呼ばれる特別なタイプのメッセージを運ぶためにプールが使用されます。OOB メッセージは、NAKACK や UNICAST などのプロトコルの順番配信要件を免除されるため、NAKACK や UNICAST が特定送信元のメッセージをキューに入れている場合でも、OOB メッセージはスタックへ配信されます。OOB メッセージは JGroups プロトコルによって頻繁に内部で使用され、アプリケーションによって使用されることもあります。例えば、JBoss Cache が **REPL\_SYNC** モードである場合、2 フェーズコミットプロトコルの第 2 フェーズに OOB メッセージを使用します。

### 27.1.2.2. TCP 設定

JGroups ベースのクラスターは TCP 接続でも動作します。クラスターサイズが大きくなると、TCP は UDP より多くのネットワークトラフィックを生成します。TCP は基本的にはユニキャストのプロトコルです。マルチキャストメッセージを送信するために、JGroups は複数の TCP ユニキャストを使用します。TCP をトランスポートプロトコルとして使用するには、JGroups `config` 要素内の **TCP** 要素を定義してください。TCP 要素の例は次の通りです。

```
<TCP singleton_name="tcp"
    start_port="7800" end_port="7800"/>
```

TCP 要素固有の属性は次の通りです。

- **start\_port** と **end\_port** は、サーバーがバインドする TCP ポートの範囲を定義します。サーバーソケットは、**start\_port** 以降の最初に利用できるポートへバインドされます。**end\_port** の前に利用可能なポートが見つからない場合 (他のソケットによってポートが使用されている場合など)、サーバーは例外をスローします。**end\_port** の指定がない場合や、**end\_port** が **start\_port** よりも低い場合、ポートの範囲に上限が適用されません。**start\_port** と **end\_port** が同じ場合、指定されたポートが使用できないと **start\_port** が失敗するため、JGroups は指定されたポートの使用を強制されます。デフォルト値は **7800** です。**0** を設定すると、オペレーティングシステムがポートを選択します (**TCCPING** にノードと必要ポートをリストする必要があるため、**MPING** や **TCPGOSSIP** のディスカバリプロトコルを使用する場合にのみ動作します)。
- TCP の **bind\_port** は **start\_port** のエイリアスとして動作します。内部に設定されると **start\_port** を設定します。
- **recv\_buf\_size**, **send\_buf\_size** は受信と送信のバッファサイズを定義します。バッファのオーバーフローでパケットが破棄される可能性を低減するため、受信バッファサイズは大きめにした方がよいでしょう。
- **conn\_expire\_time** はトラフィックが受信されていない場合、ここに指定した時間 (ミリ秒単位) を経過すると reaper によって接続を閉じることができます。

- **reaper\_interval** は reaper を実行する間隔を指定します (ミリ秒単位)。いずれの値も 0 の場合、reap は行われません。いずれかの値が > 0 になる場合、reap が有効になります。デフォルトでは、reaper\_interval が 0 です (つまり、reaper を実行しません)。
- **sock\_conn\_timeout** は、ソケット作成に関する最大時間をミリ秒単位で指定します。初めて検索を実行し、ピアがハングする場合は、いつまでも待たずにタイムアウト後に他のメンバに対して ping を実行します。これにより、メンバーがまったく見つからない可能性が少なくなります。デフォルト値は 2000 です。
- **use\_send\_queues** は各接続に別の送信キューを使用するかどうかを指定します。これにより、ピアがハングしたときに書き込みブロックが回避されます。デフォルト値は true です。
- **external\_addr** は、他のグループメンバにブロードキャストする外部 IP アドレス (ローカルアドレスと異なる場合) を指定します。これは、NAT (Network Address Translation) とファイアウォールの背後にあるプライベートネットワーク上のノードを使用し、そのノードに外部から見えるアドレス (バインドされるローカルアドレスとは異なります) のみを使用してルーティングできる場合に役に立ちます。したがって、ノードは、引き続きローカルアドレスにバインドできる一方で外部アドレスをブロードキャストするよう設定できます。これにより、ファイアウォール外部のノードはファイアウォール内部のノード (ただし、外部アドレスのもののみ) に引き続きルーティングできるため、TUNNEL プロトコル (つまり、中央ゴシップルーターの要件) を使用する必要がなくなります。external\_addr を設定しない場合、ファイアウォール背後のノードはプライベートアドレスをそのアドレスにルーティングできない他のノードにブロードキャストします。
- **skip\_suspected\_members** は、ユニキャストメッセージを該当すると思われるメンバに送信するかどうかを指定します。デフォルト値は true です。
- **tcp\_nodelay** は TCP\_NODELAY を指定します。デフォルトでは TCP はメッセージに nagle を使用します。つまり、概念的に複数の小さいメッセージは大きいメッセージにまとめられます。同期クラスターメソッド呼び出しを行う場合は、メッセージバンドルを無効にするのに加えて nagle を無効にする必要があります (**enable\_bundling** を false に設定します)。nagle を無効にするには、**tcp\_nodelay** を true に設定します。デフォルト値は false です。



#### 注記

UDP プロトコルの項で説明したすべてのプロトコルに共通する属性も TCP に適用されます。

### 27.1.2.3. TUNNEL の設定

TUNNEL プロトコルはメッセージの送信を処理するため外部ルーターを使用します。外部ルーターは **org.jgroups.stack.GossipRouter** メインクラスを実行する Java プロセスです。各ノードはこのルーターに登録する必要があります。すべてのメッセージがこのルーターに送信されてからその目的地に転送されます。TUNNEL の方法はファイアウォールの背後にあるノードとの通信を設定するために使用できます。ノードはファイアウォールを通過して **GossipRouter** への TCP 接続を確立することができます (80番ポートの使用可)。ファイアウォールの多くは、ファイアウォール内部のホストへの TCP 接続を外部ホストが開始することを許可しないため、ルーターによるファイアウォール背後のノードへのメッセージ送信にもこの接続が使用されます。TUNNEL 設定は、次のように JGroups **<config>** 要素内の **TUNNEL** 要素で定義されます。

```
<TUNNEL singleton_name="tunnel"
      router_port="12001"
      router_host="192.168.5.1"/>
```

TUNNEL 要素内で使用できる属性は以下の通りです。



- **router\_host** は GossipRouter が稼働しているホストを指定します。
- **router\_port** は GossipRouter がリッスンしているポートを指定します。
- **reconnect\_interval** は、接続が確立できない場合に **TUNNEL** が **GossipRouter** への接続を試行する間隔 (ミリ秒単位) を指定します。デフォルトは **5000** です。



#### 注記

UDP プロトコルの項で説明したすべてのプロトコルに共通する属性も **TUNNEL** に適用されます。

### 27.1.3. ディスカバリプロトコル

ノード上のチャンネルが最初に接続すると、互換性のあるチャンネルを実行する他のノードやコーディネータ (新しいノードをグループへ参加させるノード) として機能しているノードを判断する必要があります。ディスカバリプロトコルは、クラスターのアクティブなノードを検索し、どのノードがコーディネータであるか判断するために使用されます。そして、コーディネータの GMS と通信し、新たに接続したノードをグループに追加するグループメンバーシッププロトコル (GMS) にその情報が提供されます (グループメンバーシッププロトコルについての詳細は「[グループメンバーシップ \(GMS\)](#)」を参照してください)。

ディスカバリプロトコルは、クラスターのスプリットを検知するために結合プロトコル (「[マージ \(MERGE2\)](#)」参照) も支援します。

ディスカバリプロトコルはトランスポートプロトコルの上に位置しているため、使用するトランスポートプロトコルに応じて異なるディスカバリプロトコルの使用を選択することができます。また、JGroups **<config>** エレメント内のサブ要素として設定することもできます。

#### 27.1.3.1. PING

PING は PING 要求を IP マルチキャストアドレスにマルチキャストするか、ゴシップルーターに接続することによって動作する検索プロトコルです。したがって、PING は通常、UDP または TUNNEL トランスポートプロトコルの上部に位置します。各ノードはパケット {C, A} で応答します。ここで、C はコーディネータのアドレス、A は自身のアドレスを示します。ミリ秒単位のタイムアウトの経過後または num\_initial\_members の応答後、参加するノードは応答からコーディネータを調べ、JOIN 要求をコーディネータに送信します。誰も応答しない場合は、グループの最初のメンバと見なされます。

以下は IP マルチキャスト用の PING 設定の例です。

```
<PING timeout="2000"
  num_initial_members="3"/>
```

以下は Gossip Router に接続するための PING 設定の別の例です。

```
<PING gossip_host="localhost"
  gossip_port="1234"
  timeout="2000"
  num_initial_members="3"/>
```

**PING** 要素内で使用できる属性は以下の通りです。

- **timeout** はなんらかの応答を待機する最大ミリ秒数を指定します。デフォルト値は 3000 です。

- **num\_initial\_members** は、待機する最大応答数を指定します (タイムアウトが経過していない場合)。デフォルト値は 2 です。
- **gossip\_host** GossipRouter が稼働しているホストを指定します。
- **gossip\_port** GossipRouter がリスンしているポートを指定します。
- **gossip\_refresh** は GossipRouter からのリースの間隔を指定します (ミリ秒単位)。デフォルト値は 20000 です。
- **initial\_hosts** は、ディスカバリで ping されるアドレスやポートをコンマで区切った一覧です (例: **host1[12345],host2[23456]**)。デフォルトは **null** で、マルチキャストディスカバリが使用されます。 **initial\_hosts** を指定した場合、ウェルノウンホストの一部だけでなく可能なクラスターメンバーすべてをリストに含める必要があります。すべてをリストしないと、 **MERGE2** クラスタースプリットディスカバリが確実に動作しません。

**gossip\_host** と **gossip\_port** の両方が定義されると、クラスターは初期のディスカバリに GossipRouter を使用します。 **initial\_hosts** が指定されると、クラスターはディスカバリにその静的アドレス一覧を ping します。これ以外は、クラスターはディスカバリに IP マルチキャストリングを使用します。



#### 注記

ディスカバリフェーズが返されるのは **timeout** ミリ秒が経過した場合、または **num\_initial\_members** 応答が受信された場合です。

### 27.1.3.2. TCPGOSSIP

TCPGOSSIP プロトコルは GossipRouter とのみ動作します。基本的には有効な **gossip\_host** と **gossip\_port** の属性を持つ PING プロトコル設定と同じように動作します。UDP および TCP 両方のトランスポートプロトコルの上で動作します。例を示します。

```
<TCPGOSSIP timeout="2000"
    num_initial_members="3"
    initial_hosts="192.168.5.1[12000],192.168.0.2[12000]"/>
```

TCPGOSSIP 要素内で使用できる属性は以下の通りです。

- **timeout** はなんらかの応答を待機する最大ミリ秒数を指定します。デフォルト値は 3000 です。
- **num\_initial\_members** は、待機する最大応答数を指定します (タイムアウトが経過していない場合)。デフォルト値は 2 です。
- **initial\_hosts** は GossipRouter が登録するアドレスやポートのコンマで区切りリストしたものです (例: **host1[12345],host2[23456]**)。

### 27.1.3.3. TCPPING

TCPPING プロトコルは既知のメンバーのセットをとり、ディスカバリのため ping を行います。これは基本的には静的な設定です。TCP の上で動作します。次に JGroups **config** 要素内の **TCPPING** 設定要素の例を示します。

```
<TCPPING timeout="2000"
    num_initial_members="3"/>
```

```
initial_hosts="hosta[2300],hostb[3400],hostc[4500]"
port_range="3">
```

**TCPPING** 要素で使用できる属性は以下の通りです。

- **timeout** はなんらかの応答を待機する最大ミリ秒数を指定します。デフォルト値は 3000 です。
- **num\_initial\_members** は、待機する最大応答数を指定します (タイムアウトが経過していない場合)。デフォルト値は 2 です。
- **initial\_hosts** は ping を行うアドレスのコンマ区切りリストです (例: **host1[12345],host2[23456]**)。
- **port\_range** は、初回のメンバーシップを取得する時に、**initial\_hosts** パラメータに指定されたポートよりプローブされる連続ポートの数を指定します。上記の **port\_range** と **initial\_hosts** の値を例とすると、**TCPPING** 層は **hosta[2300]**、**hosta[2301]**、**hosta[2302]**、**hostb[3400]**、**hostb[3401]**、**hostb[3402]**、**hostc[4500]**、**hostc[4501]**、**hostc[4502]** へ接続しようとしています。この設定オプションにより、ポートの組み合わせをすべて記述しなくても同じホストの複数のポートへ ping することができます。TCP プロトコル設定で、**end\_port** が **start\_port** より大きい場合、**TCPPING port\_range** がこの差異になるようにし、許可範囲内でバインドされるポートに関係なくノードが ping されるようにすることが推奨されます。

#### 27.1.3.4. MPING

**MPING** は IP マルチキャストを使用して最初のメンバーシップをディスカバリします。ネットワーク上でのディスカバリメッセージの送受信をトランスポートプロトコルへ委譲する他とディスカバリプロトコルと違い、**MPING** はソケットを開いてマルチキャストディスカバリメッセージを送受信します。そのため、すべてのトランスポートに使用できますが、**TCP** で最も頻繁に使用されます。通常、**TCP** は可能性があるグループメンバーすべてを明示的にリストする必要がある **TCPPING** を必要とし、標準のメッセージトランスポートに **TCP** が必要で、ディスカバリに UDP マルチキャストイングが許可される場合に使用されます。

```
<MPING timeout="2000"
  num_initial_members="3"
  bind_to_all_interfaces="true"
  mcast_addr="228.8.8.8"
  mcast_port="7500"
  ip_ttl="8"/>
```

**MPING** 要素で使用できる属性は以下の通りです。

- **timeout** はなんらかの応答を待機する最大ミリ秒数を指定します。デフォルト値は 3000 です。
- **num\_initial\_members** は待機する最大応答数を指定します (タイムアウトが経過しない場合)。デフォルト値は 2 です。
- **bind\_addr** はメッセージの送受信を行うインターフェースを指定します。デフォルトでは、JGroups はシステムプロパティ **jgroups.bind\_addr** の値を使用します。**-b** コマンドラインスイッチを使用して設定することもできます。JGroups ソケットのバインドに関する詳細は、[「その他設定の問題」](#) を参照してください。
- **bind\_to\_all\_interfaces** は **bind\_addr** を上書きして multihome ノードにある全インターフェースを使用します。

- **mcast\_addr, mcast\_port, ip\_ttl** 属性は UDP プロトコル設定にある関連属性と同じものになります。

#### 27.1.4. 障害検出プロトコル

障害検出プロトコルは障害が発生したノードの検出に使用されます。障害が発生したノードが検出されると、検証フェーズが開始されます。このフェーズが終了してもノードがまだダウンしていると思なされた場合は、クラスターがそのビューを更新し、メッセージが障害が発生したノードに送信されないようにします。ノードがクラスターの一部でなくなったことが JGroups を使用するサービスに報告されます。障害検出プロトコルは JGroups **<config>** 要素内のサブ要素として設定されます。

##### 27.1.4.1. FD

**FD** は「ハートビート」メッセージに基づく障害検出プロトコルです。このプロトコルでは各ノードが定期的に近くのノードに ping する必要があります。近くのノードが応答しないと、呼び出しするノードは **SUSPECT** メッセージをクラスターに送ります。現在のグループコーディネータは、この疑いのあるノードが実際に停止しているか (**VERIFY\_SUSPECT**) 任意に検証できます。この検証段階の後にまだノードがダウンしている場合、コーディネータがクラスターのビューを更新します。**FD** 設定の例は次の通りです。

```
<FD timeout="60000"
    max_tries="5"
    shun="true"/>
```

**FD** 要素で利用できる属性は以下の通りです。

- **timeout** は are-you-alive メッセージに対する応答を待機する最大ミリ秒数を指定します。デフォルト値は 3000 です。
- **max\_tries** はノードが疑われるまでそのノードから失った are-you-alive メッセージの数を指定します。デフォルト値は 2 です。
- **shun** は、障害のあるノードが正式にグループに再参加せずにそのグループへのメッセージ送信を不可にするかどうかを指定します。回避されたノードはディスカバリプロセスを通じてクラスターに再参加しなければなりません。チャンネルが回避されると自動的に再参加やステート移動が発生するよう JGroups を設定することができます (これは JBoss Application Server のデフォルトの動作です)。



#### 注記

ノードからの定期的なトラフィックはノードが生存している証拠となります。このため、長期間ノードに定期的なトラフィックが検出されなかった場合のみハートビートメッセージが送信されます。

##### 27.1.4.2. FD\_SOCK

**FD\_SOCK** は、グループメンバー間で作成された TCP ソケットのリングに基づく障害検出プロトコルです。グループ内の各メンバーは近くのメンバーに接続し (最後のメンバーは最初のメンバーに接続します)、リングを形成します。ノード B の近くにあるノード A は、正常に閉じられなかった TCP ソケットを検出した時にノード B を疑い、ノード B のクラッシュが原因であると想定します (ノードがグループから退出しようとする、近くのノードに連絡し、近くのノードが疑われないようにします)。

最も単純な **FD\_SOCK** 設定は属性を使用しません。JGroups **<config>** 要素に空の **FD\_SOCK** 要素を宣言できます。

```
<FD_SOCK/>
```

**FD\_SOCK** 要素に使用できる属性は次の通りです。

- **bind\_addr** はメッセージの送受信を行うインターフェースを指定します。デフォルトでは、JGroups はシステムプロパティ **jgroups.bind\_addr** の値を使用します。-b コマンドラインスイッチを使用して設定することもできます。JGroups ソケットのバインドに関する詳細は、[「その他設定の問題」](#) を参照してください。

#### 27.1.4.3. VERIFY\_SUSPECT

このプロトコルは、疑いのあるメンバが実際にダウンしているかそのメンバーに対してもう一度 ping を実行して確認します。この確認は、クラスターのコーディネータによって実行されます。疑いのあるメンバはダウンしていることが確認されるとクラスタグループから破棄されます。このプロトコルの目的は、間違った疑いを最小限に抑えることです。以下に例を示します。

```
<VERIFY_SUSPECT timeout="1500"/>
```

**VERIFY\_SUSPECT** 要素に使用できる属性は次の通りです。

- **timeout** は、疑いのあるメンバーの応答を待つ時間を指定します。この待ち時間を過ぎるとダウンしていると見なされます。

#### 27.1.4.4. FD と FD\_SOCK

FD と FD\_SOCK (それぞれ独立して取得) では、障害検出レイヤーは安定していません。では、これらの障害検出プロトコル間の違いを把握し、これらがお互いどのように補完するかについて理解していきましょう。

- **FD**
  - 過負荷の状態にあるマシンは、are-you-alive 応答の送信時にパフォーマンスが低下することがあります。
  - デバッグ/プロファイラーでサスペンドされたときに、メンバーに問題があるのではと疑いがかけられます。
  - タイムアウトを小さくすると、誤検出の可能性が大きくなり、ネットワークトラフィックが増大します。
  - タイムアウトが大きいと、クラッシュしたメンバーが一定の時間検出されず、破棄されません。
- **FD\_SOCK:**
  - TCP 接続はまだオープンなのでデバッグでサスペンドされても問題ありません。
  - 同様の理由により、高ロードも問題とはなりません。
  - TCP 接続が中断された場合メンバーに疑いがかけられ、停止しているメンバーは検出されません。
  - また、クラッシュしたスイッチは、接続が TCP タイムアウト (2~20 分。TCP/IP スタック実装によって異なる) になるまで検出されません。

障害検出層の目的は、実際の障害を迅速に報告し、誤った疑いをかけないようにすることです。2つの解決法が存在します。

1. デフォルトでは、JGroups は `KEEP_ALIVE` で `FD_SOCKET` ソケットを設定します。これは、TCP が、トラフィックを 2 時間受信していないソケットにハートビートを送信することを意味します。TCP 接続を適切に閉じずにホストがクラッシュすると (または中間スイッチまたはルーターがクラッシュすると)、2 時間 (プラス数分) 後にこれを検出することになります。これは、当然接続をまったく閉じないよりもいいですが (`KEEP_ALIVE` が無効な場合)、それほど役に立たないことがあります。したがって、最初の解決法は、`KEEP_ALIVE` のタイムアウト値を小さくすることです。これは、ほとんどのオプションではカーネル全体に対してのみ行えます。したがって、この値を 15 分まで小さくすると、すべての TCP ソケットが影響を受けます。
2. 2 つ目の解決法は `FD_SOCKET` と `FD` を組み合わせることです。FD のタイムアウトは TCP タイムアウトよりもかなり小さく設定できます。これは 1 つのプロセスごとに個別に設定できます。ソケットが正常に閉じられていない場合は `FD_SOCKET` が疑いがあることを示すメッセージを生成します。ただし、クラッシュしたスイッチまたはホストの場合は、FD によって、ソケットが結果的に閉じられ、疑いがあることを示すメッセージが生成されるようになります。以下に例を示します。

```
<FD_SOCKET/>
<FD timeout="6000" max_tries="5" shun="true"/>
<VERIFY_SUSPECT timeout="1500"/>
```

この例では、オペレーティングシステムがすべてのソケットを閉じるため、近くにあるソケットがクラッシュなどで正常に閉じられなかった時にメンバーが疑われます。ただし、ホストまたはスイッチがクラッシュすると、ソケットは閉じられません。FD は 60 秒後 (6000 ミリ秒) 後に近くのメンバを疑います。この例では、デバッガのブレークポイントでシステムが停止した場合、`timeout` が経過した後にはデバッグされたノードが疑われます。

FD と `FD_SOCKET` の組み合わせにより、安定した障害検出層が提供されます。このため、この技術は JBoss Application Server に含まれる JGroups 設定全体で使用されます。

### 27.1.5. 信頼できる配信プロトコル

JGroups スタック内の信頼できる配信プロトコルはデータパケットが実際に正しい順序 (FIFO) で目的地となるノードに配信されるようにします。肯定的および否定的な配信確認 (ACK と NAK) が信頼できるメッセージ配信の基盤となります。ACK モードでは、送信側が受信側からの確認が受信されるまでメッセージを再送します。NAK モードでは、受信側がギャップを発見した場合に再送を要求します。

#### 27.1.5.1. UNICAST

UNICAST プロトコルはユニキャストメッセージに使用されます。肯定的な確認を使用し (ACK)、JGroups `config` 要素のサブ要素として設定されます。JGroups スタックが TCP トランスポートプロトコルで設定された場合は、TCP 自体がユニキャストメッセージの FIFO 配信を保証するため UNICAST は必要ありません。次に、UNICAST プロトコルの設定例を示します。

```
<UNICAST timeout="300,600,1200,2400,3600"/>
```

UNICAST 要素で設定できる属性は 1 つのみです。

- `timeout` は、再送信のタイムアウト (ミリ秒単位) を指定します。例えば、タイムアウトが **100,200,400,800** の場合、最初 100 ミリ秒後に ACK を受信しなかったら送信元がメッセー

ジを再送信し、2回目の再送信を行う前に200ミリ秒待ちます。最初のタイムアウト値を小さくすると、損失したメッセージの再送信が迅速に行われますが、実際にメッセージを損失しなかった場合は、メッセージが複数回送信されます(メッセージは送信されても、**ACK** がタイムアウト前に受信されなかった場合)。UDP データグラムが頻繁に損失しないようネットワークが調整されている場合、大きな値 (**1000, 2000, 3000**) を設定するとパフォーマンスは改善されます。損失が頻発するネットワークに大きな値を設定すると、損失したメッセージが再送信されるまで後のメッセージが発信されないため、パフォーマンスに悪影響を与えます。

### 27.1.5.2. NAKACK

**NAKACK** プロトコルはマルチキャストメッセージに使用されます。否定的な確認 (**NAK**) を使用します。このプロトコルでは、各メッセージは連続番号のタグが付けられます。受信者はこの連続番号を追跡して順序通りにメッセージを配信します。受信した連続番号でギャップが検出されると、周期的に損失したメッセージの再送信を送信者に依頼するタスクを受信先が設定します。損失したメッセージが受信されると、このタスクはキャンセルされます。**NAKACK** プロトコルは JGroups **<config>** 要素配下の **pbcast.NAKACK** サブ要素として設定されます。設定例は次の通りです。

```
<pbcast.NAKACK max_xmit_size="60000" use_mcast_xmit="false"
  retransmit_timeout="300,600,1200,2400,4800" gc_lag="0"
  discard_delivered_msgs="true"/>
```

**pbcast.NAKACK** 要素で設定できる属性は以下の通りです。

- **retransmit\_timeout** は、一連のタイムアウト時間(ミリ秒単位)を指定します。ここで指定したタイムアウト時間以降に、損失したメッセージが受信されないと再送信が要求されます。
- **use\_mcast\_xmit** は送信者が再送を要求しているノードだけではなくクラスター全体に再送信すべきであるか判断します。送信者のネットワーク層でパケット落ちが発生する傾向にある場合に便利で、各ノードごとに再送信する必要がありません。
- **max\_xmit\_size** は複数メッセージの損失が報告がされた時のバンドル再送信の最大サイズ(バイト単位)を指定します。
- **discard\_delivered\_msgs** は受信者ノードで配信されたメッセージを破棄するかを指定します。デフォルトでは、ノードが配信されたメッセージを保存するため、元の送信者がクラッシュあるいはグループを退出してもすべてのノードが損失したメッセージを再送信することができます。ただし、送信者のみにそのメッセージを再送するよう依頼する場合は、このオプションを有効にして配信されたメッセージを破棄することができます。
- **gc\_lag** は、周期的なクリーンアッププロトコル(「[分散ガベージコレクション \(STABLE\)](#)」参照)がすべてのピアがメッセージを受信したと提示した場合でも、再送信のためにメモリに保持するメッセージの数を指定します。デフォルト値は **20** です。

### 27.1.6. グループメンバーシップ (GMS)

JGroups スタック内のグループメンバーシップサービス (GMS) はアクティブなノードの一覧を管理しています。クラスターへの参加および退出の要求を処理します。また、障害検出プロトコルから送信される **SUSPECT** メッセージも処理します。クラスター内のすべてのノードや、JBoss Cache および HAPartition などの関心のあるサービスは、グループメンバーシップが変更されると通知を受けます。グループメンバーシップサービスは JGroups **config** 要素配下の **pbcast.GMS** サブ要素に設定されます。設定例は次の通りです。

```
<pbcast.GMS print_local_addr="true"
  join_timeout="3000"
  join_retry_timeout="2000"
```



```
shun="true"
view_bundling="true"/>
```

**pbcast.GMS** 要素で設定できる属性は以下の通りです。

- **join\_timeout** 新しいノードの JOIN 要求が成功するの待つ最長ミリ秒数を指定します。指定ミリ秒数経過後は再び試行します。
- **join\_retry\_timeout** は、JOIN が失敗した後、再試行するまでに待機する時間 (ミリ秒単位) を指定します。
- **print\_local\_addr** は起動時のノード自体のアドレスを標準出力にダンプするかを指定します。
- **shun** は、メンバーノードではないクラスタービューを受信した場合にノード自身が shun (切断) を行うかどうかを指定します。
- **disable\_initial\_coord** は、最初にチャネルへ接続する際に、ノードがクラスターコーディネーターになることを阻止するか指定します。現在のコーディネーターがグループから退出した場合、最初のチャネル接続の後にノードがコーディネーターになることを阻止することはできません。
- **view\_bundling** は、同時に到着した複数の JOIN 要求や LEAVE 要求をまとめて同時に処理し、1つの新しいビューのみがすべての変更を反映するようにするかを指定します。これは、各要求を別々に処理するよりも効率的です。

### 27.1.7. フロー制御 (FC)

フロー制御(FC) プロトコルはノード間でデータ送信率をデータ受信率へ適応させようとします。送信側ノードが速すぎる場合、受信側ノードが対応できないため、メモリ不足になるか、再送信されたパケットがドロップされることがあります。JGroups では、フロー制御はクレジットベースのシステムで実装されます。送受信ノードは同じクレジット数 (バイト) を開始時に持っています。送信側は送信するメッセージのバイト数をクレジットから引きます。受信側は受信するメッセージのバイトをクレジットに足します。送信側のクレジットがしきい値まで減少すると、受信側は送信側にクレジットを送ります。送信側のクレジットがなくなると、受信側からクレジットを受け取るまで送信側はブロックします。フロー制御プロトコルは JGroups **config** 要素配下の **FC** サブ要素に設定されます。設定例は次の通りです。

```
<FC max_credits="2000000"
    min_threshold="0.10"
    ignore_synchronous_response="true"/>
```

**FC** 要素で使用できる属性は以下の通りです。

- **max\_credits** はクレジット (バイト単位) の最大数を指定します。この値は JVM ヒープサイズより小さくしてください。
- **min\_credits** は、受信側が送信側にクレジットを送付する前に受け取らなければならない最低バイト数を指定します。
- **min\_threshold** は、**min\_credits** を算出するために使用する **max\_credits** の割合 (パーセント) を指定します。この設定は **min\_credits** 属性よりも優先されます。
- **ignore\_synchronous\_response** は、アプリケーションへメッセージを運んだスレッドが、クレジットをブロックせずに FC より送信メッセージを運び戻すことができるようにするかを指定します。同期応答とは、通常これらのメッセージが受信 RPC タイプのメッセージの応答



であることを意味しています。JGroups スレッドが FC のブロックメッセージを運べないようになると、特定のデッドロックを防ぐことができるため、**true** に設定することが推奨されます。

## 注記

最も低速の受信先が処理できる最速の速さでグループメッセージを送信しなければならない場合、グループ通信に FC が必要となります。例えば、ノード A、B、C、D で構成されるクラスターがあるとして、D は低速で (オーバーロードの可能性)、他のノードはすべて高速だとします。A がグループメッセージを送信する場合、TCP 接続より A-A (理論上)、A-B、A-C、A-D のように送信されます。

例えば、A がクラスターへ 1 億通のメッセージを送信するとしましょう。TCP のフロー制御は個別に A-B、A-C、A-D へ適用されますが、A-BCD へグループとしては適用されません。そのため、A、B、C は 1 億通のメッセージを受信しますが、D は 100 万通のメッセージのみを受信します (そのため、TCP が独自の再送信を処理しても **NAKACK** が必要となるのです)。

JGroups は、元の送信側 **S** がダウンし、**S** が送信したメッセージの再送信をノードがリクエストしたときに備え、メモリ内のすべてのメッセージをバッファする必要があります。すべてのメンバは受信したすべてのメッセージをバッファするため、安定したメッセージ (すべてのノードが可視できるメッセージ) を時々消去する必要があります (消去のプロセスは **STABLE** プロトコルによって管理されます。詳細は「[分散ガベージコレクション \(STABLE\)](#)」を参照してください)。

上記のケースでは、低速なノード D によってグループが 1M を超えるメッセージを消去することができず、各メンバが 99M メッセージをバッファします。ほとんどの場合、これにより OOM 例外が発生します。TCP のスライドウィンドウプロトコルではウィンドウがいっぱいである場合書き込みがブロックされますが、上記のケースでは A-D よりも A-B と A-C の方が高速であると見なすことに注意してください。

そのため、最も低速な受信側 (D) が処理できる速度でメッセージを送信する必要があるため、TCP を使用していても FC が必要となります。

## 注記

これは、アプリケーションが JGroups チャネルをどのように使用するかによって異なります。上記の例では、D が対応できないため、A が送信レートを下げるアプリケーションの場合に FC は必要なくなります。

このようなアプリケーションの良い例として、同期グループ RPC 呼び出しを行うアプリケーションがあります。ここで同期とは、呼び出しするスレッドがグループのすべてのメンバからの応答待機をブロックすることを意味します。このようなアプリケーションでは、呼び出しを行う A のスレッドが D からの応答の待機をブロックし、全体的な呼び出しの速度を自然に低下させます。

REPL\_SYNC に設定された JBoss Cache クラスターは同期グループ RPC 呼び出しを行うアプリケーションの良い例です。チャネルが REPL\_SYNC に設定されたキャッシュに対してのみ使用される場合は、プロトコルスタックから FC を削除することをお勧めします。

また、当然クラスターが 2 つのノードのみから構成される場合は、FC を TCP ベースのプロトコルスタックに含めることは必要ありません。単一のピアツーピア関係を超えるグループは存在せず、TCP の内部フロー制御はこれを適切に処理します。

FC が必要ではない別のケースは、バディレプリケーションと単一バディに対して設定された JBoss Cache により使用されるチャネルを使用する場合です。このようなチャネルは多くの点で 2 ノードクラスターのように動作し、メッセージはもう 1 つのノード (バディ) とのみ交換されます (すべてのメンバに送信されるデータグラビテーションに関する他のメッセージが存在する場合がありますが、適切に設計されたバディレプリケーションのケースではそのようなことはほとんどありません。ただし、FC を削除する場合はアプリケーションの負荷をテストしてください)。

## 27.2. 断片化 (FRAG2)

このプロトコルは、特定のサイズよりも大きいメッセージを断片化し、受信者側で非断片化します。ユニキャストメッセージとマルチキャストメッセージの両方に使用でき、JGroups **config** 要素の **FRAG2** サブ要素で設定されます。以下に設定例を示します。

```
<FRAG2 frag_size="60000"/>
```

FRAG2 要素で設定できる属性は以下の通りです。

- **frag\_size** は、断片化が発生する前の最大メッセージサイズ (バイト単位) を指定します。これよりサイズが大きいメッセージは断片化されます。UDP トランスポートを使用するスタックの値は 64 キロバイト未満 (最大 UDP データグラムサイズ) でなければなりません。TCP ベースのスタックの値は、FC プロトコルの **max\_credits** の値未満でなければなりません。

## 注記

TCP プロトコルにより断片化が提供されますが、FC が使用される場合は JGroups の断片化プロトコルが必要です。これは、**FC.max\_credits** よりも大きいメッセージを送信する場合は、FC プロトコルがブロックするからです。したがって、FRAG2 内の **frag\_size** は常に **FC.max\_credits** よりも小さく設定する必要があります。

## 27.3. 状態の転送

状態転送サービスはその状態を既存ノード (クラスターコーディネーター) から新たにジョインしているノードに転送します。JGroups **Config** エlement配下の **pbcaster.STATE\_TRANSFER** サブ要素で設定します。設定できる属性はありません。次に設定例を示します。

```
<pbcaster.STATE_TRANSFER/>
```

## 27.4. 分散ガベージコレクション (STABLE)

JGroups クラスター内では、全てのノードは障害が発生した場合の再送に備えて受け取ったメッセージを全て格納しなければなりません。しかし、全メッセージを永久に保存したらメモリがなくなります。分散ガベージコレクションサービスは、すべてのノードによって読まれたメッセージを定期的に消去し、各ノードのメモリから削除します。分散ガベージコレクションサービスは JGroups **config** 要素下の **pbcaster.STABLE** サブ要素に設定されます。設定例は次の通りです。

```
<pbcaster.STABLE stability_delay="1000"
    desired_avg_gossip="5000"
    max_bytes="400000"/>
```

**pbcaster.STABLE** 要素で設定できる属性は以下の通りです。

- **desired\_avg\_gossip** はガベージコレクションが実行される間隔を指定します (ミリ秒単位)。間隔的なガベージコレクションを無効にするには **0** を設定します。
- **max\_bytes** はクラスターがガベージコレクション実行を始動させる前に受信する最大バイト数を指定します。受信バイトを基にしたガベージコレクションを無効にするには **0** を設定します。
- **stability\_delay** は、ガベージコレクション実行の最後にノードが **STABILITY** メッセージを送信する前に適用されるランダム遅延の最大時間 (ミリ秒単位) を指定します。この遅延により、**STABLE** タスクを同時に実行している別のノードが変更を先に送信できるようにします。**max\_bytes** を併用する場合、この属性に小さい値を設定する必要があります。



### 注記

トラフィック量の高いクラスターの場合は **max\_bytes** 属性を設定します。

## 27.5. マージ (MERGE2)

ネットワークエラーが発生すると、クラスターは複数の異なるパーティションに区切られる可能性があります。JGroups には、パーティションのコーディネーターがお互いに通信し、再び単一のクラスターを形成できるようにする MERGE サービスがあります。フロー制御サービスは JGroups **Config** 要素配下の **MERGE2** サブ要素で設定します。以下に例を示します。

```
<MERGE2 max_interval="10000"
    min_interval="2000"/>
```

**MERGE2** 要素で使用できる属性は以下の通りです。

- **max\_interval** は MERGE メッセージを送信する前に待機する最大時間 (ミリ秒単位) を指定します。

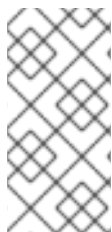
- **min\_interval** MERGE メッセージを送信する前に待機する最低時間 (ミリ秒単位) を指定します。

MERGE メッセージを定期的を送信するため、JGroups は **min\_interval** と **max\_interval**の間の無作為な値を選択します。



#### 注記

アプリケーションがチャンネルを使用して維持するアプリケーションステートはマージ中に JGroups によってマージされません。これはアプリケーションによって実行されます。



#### 注記

**MERGE2** を **TCPPING** とともに使用する場合は、マージプロセスが適切に実行されるように **initial\_hosts** 属性に再びマージできるすべてのノードを含める必要があります。そうしないと、マージプロセスがすべてのサブグループを検出せず、リストされていないメンバーのみで構成されるサブグループが無視される可能性があります。

## 27.6. その他設定の問題

### 27.6.1. JGroups チャンネルの特定インターフェースへのバインド

上記の「トランスポートプロトコル」セクションでは、JGroups がソケットをバインドするインターフェースを設定する方法について簡単に説明しました。ここではこのトピックについて詳しく説明します。

最初に、システムプロパティ **jgroups.bind\_addr** (または廃止された初期の名前である **bind.address**) が設定されている場合に XML 設定ファイルの **bind\_addr** 要素に設定された値が JGroups によって無視されることを理解することが重要です。このシステムプロパティは XML プロパティよりも優先されます。JBoss Application Server が **-b** (または **--host** スイッチを使用して起動されるとアプリケーションサーバーは **jgroups.bind\_addr** を指定された値に設定します。 **-b** が指定されないとアプリケーションサーバーはデフォルトでほとんどのサービスを **localhost** にバインドします。

JGroups がインターフェースにバインドする方法を管理するベストプラクティスは何ですか？

- JGroups を他のサービスとして同じインターフェースにバインドします。単純に **-b** を使用します。

```
./run.sh -b 192.168.1.100 -c production
```

- サービス (JBoss Web など) を 1 つのインターフェースにバインドしますが、JGroups には異なるインターフェースを使用します。

```
./run.sh -b 10.0.0.100 -Djgroups.bind_addr=192.168.1.100 -c production
```

特別に設定されたシステムプロパティは **-b** 値よりも優先されます。これは一般的な使用パターンであり、一方のネットワークにクライアントトラフィックを割り当て、もう一方のネットワークにクラスター間トラフィックを割り当てます。

- サービス (JBoss Web など) をすべてのインターフェースにバインドします。これは以下のよう  
に実行することができます。

```
./run.sh -b 0.0.0.0 -c production
```

しかし、これを実行すると JGroups がすべてのインターフェースにバインドしなくなります。代わりに JGroups はマシンのデフォルトのインターフェースにバインドします。JGroups がすべてのインターフェースで送受信するよう指示する方法については、「[トランスポートプロトコル](#)」の章を参照してください。

- サービス (JBoss Web など) をすべてのインターフェースにバインドしますが、JGroups インターフェースは指定します。

```
./run.sh -b 0.0.0.0 -Djgroups.bind_addr=192.168.1.100 -c production
```

ここでも、特別に設定したシステムプロパティは **-b** 値よりも優先されます。

- 異なるチャンネルに異なるインターフェースを使用します。

```
./run.sh -b 10.0.0.100 -Djgroups.ignore.bind_addr=true -c production
```

この設定では、JGroups が **jgroups.bind\_addr** システムプロパティを無視し、代わりに XML で指定された値を使用します。**bind\_addr** 属性を希望のインターフェースに設定するにはさまざまな XML 設定ファイルを編集する必要があります。

## 27.6.2. JGroups チャンネルの分離

JBoss Application Server 内には、独立して JGroups チャンネルを作成する多くのサービスが存在します。これらのサービスには複数の JBoss Cache サービス (**HttpSession** レプリケーション、EJB3 ステートフルセッション Bean レプリケーション、EJB3 エンティティレプリケーション)、2 つの JBoss Messaging チャンネル、他多くの JBossHA サービスの基礎となる汎用クラスター化サービスである **HAPartition** などが含まれます。

これらのチャンネルは目的のピアとのみ通信することが重要です。他のサービスで使用されるチャンネルや、グループの一部ではないマシンで開かれた同じサービスのチャンネルとは通信しません。ノードがお互いに適切に通信しないことは、JBoss Enterprise Application Platform クラスターリングでユーザーが直面する最も一般的な問題の 1 つです。

JGroups チャンネルが通信する相手はグループ名によって定義されます。UDP ベースのチャンネルの場合、マルチキャストアドレスとマルチキャストポートによって定義されます。JGroups チャンネルを分離すると、異なるチャンネルに異なるグループ名、マルチキャストアドレス (場合によってはマルチキャストポート) が使用されます。

### 27.6.2.1. アプリケーションサーバーインスタンス同士を分離

本項では、同じ環境内で複数の独立したクラスターが実行される場合の問題点を取り上げます。例えば、単一の実稼働クラスター、ステージングクラスター、QA クラスターか、複数のクラスターが QA テスト環境または開発チーム環境にあるとします。

JGroups クラスターをネットワーク上の他のクラスターから分離するには、次を行う必要があります。

- 異なるクラスターのチャンネルが異なるグループ名を使用するようにしてください。これは、JBoss の起動に使用されるコマンドライン引数で制御できます。詳細は、[「グループ名の変更」](#)を参照してください。

- 異なるクラスターのチャネルが異なるマルチキャストアドレスを使用するようにしてください。JBoss の起動に使用されるコマンドライン引数を用いると簡単に制御することができます。
- Linux、Windows、Solaris、HP-UX のいずれかで実行していない場合、各クラスターのチャネルが異なるマルチキャストポートを使用する必要があります。これは、異なるグループ名を使用するよりも複雑ですが、コマンドラインで制御することができます。「[マルチキャストポートの変更](#)」を参照してください。サーバーが Linux、Windows、Solaris、HP-UX のいずれかで稼働していれば異なるポートを使用する必要はありません。

### 27.6.2.2. 同じ AS インスタンスセット上の他のサービスに対してチャネルを分離

本項では、一般的なケースを取り上げます。例えば、3つのマシンで構成されたクラスターがあり、各マシンに HAPartition と Web セッションクラスタリング向けの JBoss Cache がデプロイされているとします。HAPartition チャネルは JBoss Cache チャネルと通信してはなりません。これらのチャネルを適切に分離するのは比較的単純で、ユーザーによる変更を必要とせずに、通常はアプリケーションサーバーによって処理されます。

同じアプリケーションサーバーインスタンスのセット上で異なるサービスのチャネルを分離するには、各チャネルに独自のグループ名がなければなりません。JBoss Application Server に同梱される設定はこの条件に対応していますが、JGroups を直接使用するカスタムサービスを作成した場合は固有のグループ名を使用する必要があります。JBoss Cache のカスタム設定を作成した場合、**clusterName** 設定プロパティに固有の値を指定するようにしてください。

JBoss Application Server 5 以前のリリースでは、同じアプリケーションサーバーで実行されている異なるチャネルも固有のマルチキャストポートを使用しなければなりませんでした。JBoss AS 5 で導入された JGroups 共有トランスポート (「[JGroups 共有トランスポート](#)」参照) により、複数のチャネルが同じトランスポートプロトコルとソケットを使用することが一般的になりました。これにより、共有トランスポートの利点の1つである設定の簡易化が実現しました。しかし、JGroups プロトコルスタックのカスタム設定を独自に作成する場合は、トランスポートプロトコルのマルチキャストポートが他のプロトコルスタックが使用するポートと同じにならないようにしてください。

#### 27.6.2.2.1. グループ名の変更

JGroups チャネルのグループ名は、チャネルを開始するサービスによって設定されます。標準のクラスター化サービスの場合は、JBoss の起動時に **-g** (または **--partition**) スイッチを使用するだけで固有のグループ名を作成できます。

```
./run.sh -g QAPartition -b 192.168.1.100 -c production
```

このスイッチは、**jboss.partition.name** システムプロパティを設定します。このプロパティをコンポーネントの1つとして利用し、すべての標準的なクラスタリング設定ファイルのグループ名を設定します。例は以下の通りです。

```
<property name="clusterName">${jboss.partition.name:DefaultPartition}-SFSBCache</property>
```

#### 27.6.2.2.2. マルチキャストアドレスとポートの変更

**-u** (または **--udp**) コマンドラインスイッチを使用すると、すべての標準的な AS サービスが開いた JGroups チャネルによって使用されるマルチキャストアドレスを制御できます。

```
/run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c production
```

このスイッチは **jboss.partition.udpGroup** システムプロパティを設定します。これは、JBoss AS の標準プロトコルスタック設定すべてにて参照されます。

```
<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}" ....
```



### 注記

グループ名が異なるチャンネルが同じマルチキャストアドレスとポートを共有する場合は、各チャンネルの下位のレベルの JGroups プロトコルが他のグループ向けのメッセージを確認および処理し、結果的に破棄します。これにより少なくともパフォーマンスが低下し、異常な動作が起こることがあります。

#### 27.6.2.2.3. マルチキャストポートの変更

オペレーティングシステムによっては (Mac OS X など) 異なる **-g** や **-u** の値を使うだけではクラスターを分離できないことがあります。この場合、異なるクラスターで実行しているチャンネルも異なるマルチキャストポートを使用する必要があります。マルチキャストポートの設定は **-g** や **-u** ほど簡単ではありません。デフォルトでは、**production** 設定で実行されている JBoss AS インスタンスは、JGroups UDP トランスポートプロトコルの異なるインスタンスを最大で 2 つ使用します。そのため、2 つのマルチキャストソケットを開きます。コマンドラインでシステムプロパティを使用するとソケットが使用するポートを制御できます。例は次の通りです。

```
/run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c production \\  
-Djboss.jgroups.udp.mcast_port=12345 -  
Djboss.messaging.datachanneludpport=23456
```

**jboss.messaging.datachanneludpport** プロパティは、JBoss Messaging の **DATA** チャンネルにある **MPING** プロトコルによって使用されるマルチキャストポートを制御します。

**jboss.jgroups.udp.mcast\_port** プロパティは、他すべてのクラスター化サービスが共有する UDP トランスポートプロトコルによって使用されるマルチキャストポートを制御します。

**\$JBASS\_HOME/server/production/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml** に含まれる JGroups プロトコルスタック設定には、標準的な JBoss AS ディストリビューションが実際に使用しない他のサンプルプロトコルスタック設定がいくつか含まれています。このような設定もシステムプロパティを使用してマルチキャストポートを設定します。そのため、このようなプロトコルスタック設定の 1 つを使用するよう AS サービスを設定した場合、適切なシステムプロパティを使用してコマンドラインからポートを制御するようにしてください。





## 注記

アドレスを変更するだけで十分なはずですが、マルチキャストソケットの処理は、JVM がオペレーティングシステムの動作の違いをアプリケーションに隠すことのできない場合の 1 つとなります。 `java.net.MulticastSocket` クラスは異なるオーバーロードしたコンストラクタを提供します。オペレーティングシステムによっては、1 つのコンストラクタバリエーションを使用すると特定のマルチキャストポートへ提供されたパケットは、リッスンしているマルチキャストアドレスに関係なくポート上のすべてのリスナに配信されます。これを、「プロミスキャストラフィック」問題と呼びます。プロミスキャストラフィック問題が発生するオペレーティングシステムの多くでは (Linux、Solaris、HP-UX)、この問題を回避できる異なるコンストラクタバリエーションを JGroups が使用することができます。しかし、一部のオペレーティングシステム (Mac OS X) では、別のコンストラクタバリエーションを使用するとマルチキャストが正しく動作しません。そのため、このようなオペレーティングシステムでは、異なるマルチキャストポートを異なるクラスターに設定することが推奨されます。

### 27.6.2.3. OS UDP のバッファ制限を設定して UDP のパフォーマンスを改善

デフォルトでは、JBoss Enterprise Application Platform の JGroups チャネルは IP マルチキャストを利用するため UDP トランスポートプロトコルを使用します。しかし、UDP の欠点の 1 つは TCP 提供の信用できる配信保証がないことです。「[信頼できる配信プロトコル](#)」で説明したプロトコルは、JGroups が UDP メッセージの配信を保証できるようにしますが、これらのプロトコルは Java によって実装され、オペレーティングシステムのネットワーク層には実装されません。UDP ベースの JGroups のパフォーマンスを最大限にするため、UDP データグラムの損失を制限して JGroups によるメッセージの再送信を制限することが重要です。

UDP データグラム損失の最も一般的な原因の 1 つは、ソケットの受信バッファが小さすぎることです。JGroups がオペレーティングシステムに「要求」する受信バッファサイズを指定するには、UDP プロトコルの `mcast_recv_buf_size` 設定属性と `ucast_recv_buf_size` 設定属性が使用されますが、オペレーティングシステムが提供する実際のバッファサイズはオペレーティングシステムレベルの最大値によって制限されます。このような値は大変小さいことがほとんどです。

表27.1 デフォルトの最大 UDP バッファサイズ

オペレーティングシステム	デフォルトの最大 UDP バッファ (バイト単位)
Linux	131071
Windows	既知の制限なし
Solaris	262144
FreeBSD、 Darwin	262144
AIX	1048576

上記の制限を増やすために使用されたコマンドは、オペレーティングシステム固有のコマンドです。下記の表は、最大バッファを 25 メガバイトに増加するために必要なコマンドを表しています。すべての場合で root 権限が必要になります。

表27.2 最大 UDP バッファサイズを変更するコマンド



オペレーティングシステム	コマンド
Linux	<b>sysctl -w net.core.rmem_max=26214400</b>
Solaris	<b>ndd -set /dev/udp udp_max_buf 26214400</b>
FreeBSD、 Darwin	<b>sysctl -w kern.ipc.maxsockbuf=26214400</b>
AIX	<b>no -o sb_max=8388608</b> (AIX は 1 メガバイト、 4 メガバイト、 8 メガバイトのみを許可)

### 27.6.3. JGroups のトラブルシューティング

#### 27.6.3.1. ノードがクラスターを形成しない

使用しているマシンが IP マルチキャストを使用するよう正しく設定されていることを確認します。これを検出するために使用できるテストプログラムには `McastReceiverTest` と `McastSenderTest` の 2 つがあります。**\$JBoss\_HOME/server/production/lib** ディレクトリに移動し、`McastReceiverTest` を起動します。例は次の通りです。

```
[lib]$ java -cp jgroups.jar org.jgroups.tests.McastReceiverTest -
mcast_addr 224.10.10.10 -port 5555
```

その後、別のウィンドウで**`McastSenderTest`**を起動します。

```
[lib]$ java -cp jgroups.jar org.jgroups.tests.McastSenderTest -mcast_addr
224.10.10.10 -port 5555
```

特定のネットワークインターフェースカード (NIC) にバインドしたい場合は、**`-bind_addr 192.168.0.2`**を使用します。ここで、`192.168.0.2` はバインドする NIC の IP アドレスです。送信側と受信側の両方でこのパラメーターを使用します。

**`McastSenderTest`** ウィンドウに入力すると、**`McastReceiverTest`** ウィンドウに出力が表示されるはずです。これが不可能な場合は、送信側で **`-ttl 32`** を使用してください。さらに失敗した場合は、IP マルチキャストを正しく設定する方法についてシステム管理者に問い合わせ、システム管理者に、選択したインターフェースでマルチキャストが動作するよう確認するか、マシンに複数のインターフェースがある場合は、正しいインターフェースを確認してください。マルチキャストがクラスター内の各マシンで正しく動作している場合は、送信側をあるマシン、受信側を別のマシンに割り当てることによって上記のテストを繰り返してネットワークをテストできます。

#### 27.6.3.2. FD でハートビートが不明な原因

ハートビート ACK を時間 T (タイムアウトと `max_tries` によって定義される) の間受け取らないため、メンバーが FD によって疑われることがあります。これには複数の理由が考えられます。たとえば、A、B、C、D のクラスターでは以下の場合に C が疑われることがあります (A は B、B は C、C は D、D は A に対して ping を実行します)。

- B または C の CPU 稼働率が T 秒以上 100% である場合。したがって、C がハートビート ACK を B に送信しても、B は CPU 稼働率が 100% であるためそれを処理できません。
- B または C がガベージコレクションを実行している場合 (上記と同様)。
- 上記の 2 つのケースの組み合わせ
- ネットワークでパケットが失われる場合。これは通常、ネットワークに大量のトラフィックが存在し、スイッチがパケットの破棄を開始したときに発生します (通常は最初にブロードキャスト、次に IP マルチキャスト、最後に TCP パケットが失われます)。
- B または C がコールバックを処理している場合。C がチャンネルを介してリモートからのメソッド呼び出しを受信し、その処理に T+1 秒かかるものとします。この時間の間、C はハートビートを含む他のすべてのメッセージを処理しません。したがって B はハートビート ACK を受信しないため、C を疑うことになります。

## 第28章 JBOSS CACHE の設定とデプロイメント

JBoss Cache は、JBoss Enterprise Application クラスタの標準クラスタ化サービスの多くで使用される分散キャッシュサポートの基盤を提供します。カスタムキャッシュの要件に対応するようにアプリケーションに JBoss Cache をデプロイすることもできます。本章では、JBoss Cache で使用できる主な設定オプションについて説明し、これらのオプションが Enterprise Application Platform によって提供される標準クラスタ化サービスが使用する JBoss Cache にどのように関連するかについて焦点を置きます。また、Enterprise Application Platform にカスタムキャッシュをデプロイするためのオプションについても説明します。

独自のアプリケーションによる直接利用を目的に JBoss Cache をデプロイしたい場合は、[Red Hat Documentation portal](#)にある JBoss Cache ドキュメントを読んでからデプロイすることが強く推奨されます。

また、標準の JBoss Enterprise Application Platform のクラスター化サービスがどのように JBoss Cache を使用するかについては、「[JBoss Cache による分散キャッシング](#)」を参照してください。

### 28.1. 主な JBOSS CACHE 設定オプション

JBoss Enterprise Application Platform には、標準のクラスター化サービスのユースケースに適したデフォルトの JBoss Cache 設定が同梱されます (Web セッションレプリケーションや JPA/Hibernate キャッシングなど)。標準のクラスター化サービスに関係するほとんどのアプリケーションは、同梱されている状態のままデフォルト設定で動作します。ネットワークやパフォーマンスの要件が特別なアプリケーションをデプロイする場合のみ調整が必要になります。本項では、選択できる主な設定の概要を説明します。ここでは包括的な内容は取り上げないため、デフォルト設定以外も確認されたいユーザーは、[http://www.redhat.com/docs/en-US/JBoss\\_Enterprise\\_Application\\_Platform/](http://www.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/) の JBoss Cache ドキュメントを参照してください。

本項の JBoss Cache 設定例の多くは、XML より `org.jboss.cache.config.Configuration` オブジェクトグラフをビルドするのに JBoss Microcontainer スキーマを使用します。JBoss Cache は独自のカスタム XML スキーマを持っていますが、他の内部 Enterprise Application Platform サービスとの整合性を保つため、標準 JBoss Enterprise Application Platform の CacheManager サービスは JBoss Microcontainer スキーマを使用します。

主な設定オプションについて説明する前に、それらのオプションを使用するような場合について見てみましょう。

#### 28.1.1. CacheManager 設定の編集

「[JBoss Enterprise Application Platform の CacheManager サービス](#)」の通り、標準の JBoss Enterprise Application Platform のクラスター化サービスは CacheManager サービスを JBoss Cache インスタンスのファクトリとして使用します。そのため、キャッシュ設定を変更する場合、CacheManager サービスの編集が必要になるでしょう。



#### 注記

CacheManager を、ユーザー独自のアプリケーションが直接使用するカスタムキャッシュのファクトリとして使用することもできます。詳細は「[CacheManager サービスによるデプロイメント](#)」を参照してください。

CacheManager は `$JBOSS_HOME/server/PROFILE/deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml` ファイルにて設定できます。編集する可能性の高い要素は、「CacheConfigurationRegistry」Bean でしょう。この Bean は

CacheManager が認識する名前付きの JBC 設定すべてのレジストリを保持しています。このファイルへの編集の多くは、新しい JBoss Cache 設定の追加や既存設定のプロパティの変更が関係します。

以下は、編集された「CacheConfigurationRegistry」Bean 設定になります。

```
<bean name="CacheConfigurationRegistry"
class="org.jboss.ha.cachemanager.DependencyInjectedConfigurationRegistry">

    <!-- If users wish to add configs using a more familiar JBC config
format
        they can add them to a cache-configs.xml file specified by this
property.
        However, use of the microcontainer format used below is
recommended.
        <property name="configResource">META-INF/jboss-cache-
configs.xml</property>
        -->

    <!-- The configurations. A Map<String name, Configuration config> --
>
    <property name="newConfigurations">
        <map keyClass="java.lang.String"
valueClass="org.jboss.cache.config.Configuration">

        <!-- The standard configurations follow. You can add your own and/or
edit these. -->

        <!-- Standard cache used for web sessions -->
        <entry><key>standard-session-cache</key>
        <value>
            <bean name="StandardSessionCacheConfig"
class="org.jboss.cache.config.Configuration">

                <!-- Provides batching functionality for caches that don't want
to
                    interact with regular JTA Transactions -->
                <property name="transactionManagerLookupClass">
                    org.jboss.cache.transaction.BatchModeTransactionManagerLookup
                </property>

                <!-- Name of cluster. Needs to be the same for all members -->
                <property
name="clusterName">${jboss.partition.name:DefaultPartition}-
SessionCache</property>
                <!-- Use a UDP (multicast) based stack. Need JGroups flow control
(FC)
                    because we are using asynchronous replication. -->
                <property
name="multiplexerStack">${jboss.default.jgroups.stack:udp}</property>
                <property name="fetchInMemoryState">true</property>

                <property name="nodeLockingScheme">PESSIMISTIC</property>
                <property name="isolationLevel">REPEATABLE_READ</property>
                <property name="cacheMode">REPL_ASYNC</property>
            </bean>
        </value>
        </entry>
    </map>
    </property>
</bean>
```

```

        .... more details of the standard-session-cache configuration
    </bean>
</value>
</entry>

<!-- Appropriate for web sessions with FIELD granularity -->
<entry><key>field-granularity-session-cache</key>
<value>

    <bean name="FieldSessionCacheConfig"
class="org.jboss.cache.config.Configuration">
        .... details of the field-granularity-standard-session-cache
configuration
    </bean>

</value>

</entry>

... entry elements for the other configurations

</map>
</property>
</bean>

```

実際の JBoss Cache 設定は、標準の JBoss Cache 設定形式でなく、JBoss Microcontainer のスキーマを使用して指定されます。JBoss Cache が標準設定形式の 1 つを構文分析すると、**org.jboss.cache.config.Configuration** タイプの Java Bean と、さらに複雑なサブ設定 (キャッシュローティング、エビクション、バディレプリケーションなど) に対する子 Java Bean のツリーを作成します。このような XML の構文分析や Java Bean の作成を JBC に委譲する代わりに、Enterprise Application Platform のマイクロコンテナに直接このタスクを実行してもらいます。これにより、マイクロコンテナは設定 Bean を認識します。今後の Enterprise Application Platform 5.x リリースでは、外部管理ツールで JBC 設定を管理できるようにするため便利です。

設定形式は、Enterprise Application Platform に同梱される標準の設定を見ればすぐ理解できるはずです。設定には主要な要素がすべて含まれています。JBoss Cache 設定を構成する様々な Java Bean のタイプやプロパティについては JBoss Cache の javadoc を参照してください。ここで、包括的な例を紹介します：

```

<bean name="StandardSFSBCacheConfig"
class="org.jboss.cache.config.Configuration">

    <!-- No transaction manager lookup -->

    <!-- Name of cluster. Needs to be the same for all members -->
    <property name="clusterName">${jboss.partition.name:DefaultPartition}-
SFSBCache</property>
    <!-- Use a UDP (multicast) based stack. Need JGroups flow control (FC)
        because we are using asynchronous replication. -->
    <property name="multiplexerStack">${jboss.default.jgroups.stack:udp}
</property>
    <property name="fetchInMemoryState">true</property>

    <property name="nodeLockingScheme">PESSIMISTIC</property>
    <property name="isolationLevel">REPEATABLE_READ</property>
    <property name="cacheMode">REPL_ASYNC</property>

```

```

<property name="useLockStriping">false</property>

<!-- Number of milliseconds to wait until all responses for a
      synchronous call have been received. Make this longer
      than lockAcquisitionTimeout.-->
<property name="syncReplTimeout">17500</property>
<!-- Max number of milliseconds to wait for a lock acquisition -->
<property name="lockAcquisitionTimeout">15000</property>
<!-- The max amount of time (in milliseconds) we wait until the
      state (ie. the contents of the cache) are retrieved from
      existing members at startup. -->
<property name="stateRetrievalTimeout">60000</property>

<!--
      SFSBs use region-based marshalling to provide for partial state
      transfer during deployment/undeployment.
-->
<property name="useRegionBasedMarshalling">false</property>
<!-- Must match the value of "useRegionBasedMarshalling" -->
<property name="inactiveOnStartup">false</property>

<!-- Disable asynchronous RPC marshalling/sending -->
<property name="serializationExecutorPoolSize">0</property>
<!-- We have no asynchronous notification listeners -->
<property name="listenerAsyncPoolSize">0</property>

<property name="exposeManagementStatistics">true</property>

<property name="buddyReplicationConfig">
  <bean class="org.jboss.cache.config.BuddyReplicationConfig">

    <!-- Just set to true to turn on buddy replication -->
    <property name="enabled">false</property>

    <!-- A way to specify a preferred replication group. We try
          and pick a buddy who shares the same pool name (falling
          back to other buddies if not available). -->
    <property name="buddyPoolName">default</property>

    <property name="buddyCommunicationTimeout">17500</property>

    <!-- Do not change these -->
    <property name="autoDataGravitation">false</property>
    <property name="dataGravitationRemoveOnFind">true</property>
    <property name="dataGravitationSearchBackupTrees">true</property>

    <property name="buddyLocatorConfig">
      <bean
class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
        <!-- The number of backup nodes we maintain -->
        <property name="numBuddies">1</property>
        <!-- Means that each node will *try* to select a buddy on
              a different physical host. If not able to do so
              though, it will fall back to colocated nodes. -->
        <property name="ignoreColocatedBuddies">true</property>

```

```

        </bean>
    </property>
</bean>
</property>
<property name="cacheLoaderConfig">
    <bean class="org.jboss.cache.config.CacheLoaderConfig">
        <!-- Do not change these -->
        <property name="passivation">true</property>
        <property name="shared">false</property>

        <property name="individualCacheLoaderConfigs">
            <list>
                <bean
class="org.jboss.cache.loader.FileCacheLoaderConfig">
                    <!-- Where passivated sessions are stored -->
                    <property
name="location">${jboss.server.data.dir}${/}sfsb</property>
                    <!-- Do not change these -->
                    <property name="async">false</property>
                    <property
name="fetchPersistentState">true</property>
                    <property name="purgeOnStartup">true</property>
                    <property
name="ignoreModifications">false</property>
                    <property
name="checkCharacterPortability">false</property>
                </bean>
            </list>
        </property>
    </bean>
</property>

<!-- EJBs use JBoss Cache eviction -->
<property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
        <property name="wakeupInterval">5000</property>
        <!-- Overall default -->
        <property name="defaultEvictionRegionConfig">
            <bean class="org.jboss.cache.config.EvictionRegionConfig">
                <property name="regionName"/></property>
                <property name="evictionAlgorithmConfig">
                    <bean
class="org.jboss.cache.eviction.NullEvictionAlgorithmConfig"/>
                </property>
            </bean>
        </property>
        <!-- EJB3 integration code will programatically create
            other regions as beans are deployed -->
    </bean>
</property>
</bean>

```

基本的に、XML は **org.jboss.cache.config.Configuration** Java Bean の作成と、この Bean のプロパティ数の設定を指定します。ほとんどのプロパティは単純なタイプですが、**buddyReplicationConfig** や **cacheLoaderConfig** などは、複数タイプの Java Bean を値とします。

次に、主な設定オプションの一部を見てみましょう。

### 28.1.2. キャッシュモード

JBoss Cache の **cacheMode** 設定属性は、2 つの関連アスペクトを 1 つのプロパティにします。

#### クラスター更新の処理

1 ノード上のキャッシュインスタンスがローカルステートを変更する際にどのように他のクラスターへ通知するかを制御します。これには、3 つのオプションがあります。

- **同期** では、キャッシュインスタンスがピアへメッセージを送信して変更を通知し、ピアが同じ変更を適用したか確認できるまで待機してから返信します。JTA トランザクションの一部として変更された場合、トランザクションコミット中の 2 フェーズコミットプロセスの一部として実行されます。確認が受信されるまでロックは保留されます。すべてのノードからの確認を待つと遅延が発生しますが、クラスター周囲の整合性が保たれます。クラスターのすべてのノードがキャッシュデータにアクセスする可能性があるため整合性を維持する必要性が高い場合、同期モードが必要となります。
- **非同期** では、キャッシュインスタンスがピアにメッセージを送信し変更を通知すると同じ変更を適用したか確認する前に即座に返答します。キャッシュコンテンツを変更したスレッド以外のスレッドが送信メッセージを処理するわけでは**ありません**。変更を行うスレッドはクラスターへのメッセージ送信処理を行います。処理時間が同期の通信よりも短いだけです。送信を行うキャッシュのみがデータへアクセスし、クラスターメッセージを使い送信ノードに問題があった場合などバックアップコピーを提供するといった、セッション複製などの場合に、非同期モードは最も有用です。非同期のメッセージングは、後に別のノードにフェールオーバーするユーザー要求が期限切れステートになるリスクがありますが、同期モードよりも非同期モードの方がパフォーマンス的に利点が多いため、多くのセッションタイプアプリケーションでこのリスクは受け入れられています。
- **ローカル** では、キャッシュインスタンスは全くメッセージを送信しません。JGroups チャネルもキャッシュによって使用されません。JBoss Cache にはクラスタリング以外にも多くの便利な機能を持っており、クラスタで使用されなくても大変有用なキャッシングライブラリとして機能します。また、クラスター内でもキャッシュされたデータの一部で整合性を保つ必要がありません。このような場合、Local モードはパフォーマンスを改善することができます。JPA/Hibernate クエリ結果セットのキャッシングがこの例となります。Hibernate の 2 次キャッシングロジックは、別のメカニズムを使用して 2 次キャッシュの陳腐化したクエリ結果セットを無効にするため、JBoss Cache はクエリ結果セットキャッシュに対してクラスターの周囲でメッセージを送信する必要はありません。

#### レプリケーションと無効化

この側面は、キャッシュがローカルステートを変更した時にクラスターの周囲へ送信されたメッセージの内容に対処します (クラスターの他のキャッシュがどのようにして変更を反映するかなど)。

- **レプリケーション** では、送信ノードの新しいステートを反映するため、他のノードがステートを更新する必要があります。送信ノードは変更されたステート含める必要があるため、メッセージの負荷が大きくなります。他のノードが別の方法でステートを取得できない場合、レプリケーションが必要となります。
- **無効化** では、他のノードがローカルステートより変更されたステートを削除する必要があります。無効化は、ステート自体ではなく変更されたステートのキャッシュキーのみを送信する必要があるため、クラスター更新メッセージの負荷を軽減します。しかし、削除されたステートを別のソースから取り出すことができる場合のみのオプションとなります。キャッシュされたステートをデータベースより再読み取りできるため、クラスター化された JPA/Hibernate エンティティキャッシュに最適なオプションです。



これら 2 つの側面が組み合わされ、**cacheMode** 設定属性に対する 5 つの有効な値が形成されます。

- **LOCAL** はクラスターメッセージが必要ないことを意味します。
- **REPL\_SYNC** は同期されたレプリケーションメッセージが送信されることを意味します。
- **REPL\_ASYNC** は、非同期のレプリケーションメッセージが送信されることを意味します。
- **INVALIDATION\_SYNC** は、同期された無効化メッセージが送信されることを意味します。
- **INVALIDATION\_ASYNC** は、非同期の無効化メッセージが送信されることを意味します。

### 28.1.3. トランザクションの処理

JBoss Cache は JTA トランザクションマネージャーと統合し、キャッシュへのトランザクションのアクセスを許可します。JBossCache がトランザクションの存在を検出すると、トランザクションが存在する間ロックが保持され、トランザクションがロールバックするとキャッシュへの変更が元に戻されます。他のノードに変更を知らせるために送信されたクラスター全体へのメッセージは延期され、トランザクションコミットの一部としてバッチで送信されます (チャティネスを低減します)。

トランザクションマネージャーと統合するには、**transactionManagerLookupClass** 設定属性を設定します。この属性は、ローカルトランザクションマネージャーを検索するために JBoss Cache が使用できるクラスの完全修飾クラス名を指定します。JBoss Enterprise Application Platform 内では、この属性は次の 2 つの値のいずれかを持ちます。

- **org.jboss.cache.transaction.JBossTransactionManagerLookup**

この値は、アプリケーションサーバーで実行されている標準のトランザクションマネージャーを検索します。JTA トランザクションにキャッシングを参加させたい場合、デプロイするカスタムキャッシュにこの値を使用します。

- **org.jboss.cache.transaction.BatchModeTransactionManagerLookup**

Web セッションと EJB SFSB キャッシングに使用されるキャッシュ設定にこの値が使用されます。JBoss Cache に同梱される、**BatchModeTransactionManager** と呼ばれる疑似の **TransactionManager** を指定します。このトランザクションマネージャーは真の JTA トランザクションマネージャーではないため、JBoss Cache 以外には使用しないでください。JBoss Enterprise Application Platform で使用すると、要求中に実行されるエンドユーザートランザクションに絡まることなくセッションレプリケーションのユースケースに対して JBoss Cache のトランザクション動作の利点を提供することができます。

### 28.1.4. 並行アクセス

JBoss Cache はスレッドセーフのキャッシング API で、独自の効率的なメカニズムを使用して並行アクセスを制御します。並行アクセスは **nodeLockingScheme** 設定属性と **isolationLevel** 設定属性によって設定されます。

**nodeLockingScheme** には次の 3 つを選択できます。

- **MVCC** (多版型同時実行制御: multi-versioned concurrency control) は、最新のデータベース実装が利用するロッキングスキームで、共有データへの高速で安全な並行アクセスを管理します。JBoss Cache 3.x は MVCC の革新的な実装をデフォルトのロッキングスキームとして使用します。MVCC は並行アクセスに対して次の機能を提供します。
  - ライターをブロックしないリーダー

- 高速で失敗するライター

並行ライターにデータのバージョン化やコピー化を使用して実現します。ライターが共有ステートをコピーする間、リーダーが共有ステートの読み取りを継続し、バージョン ID の値を上げます。バージョンがまだ有効であることを確認してから (別の並行ライターが先にステートを変更しなかった場合など)、共有ステートを書き戻します。

JPA/Hibernate エントリキャッシングには MVCC の選択が推奨されます。

- **PESSIMISTIC** (悲観的) ロッキングでは、読み取りや書き込みを行う前にスレッドやトランザクションがノード上で排他的ロックか非排他的ロックを取得します。どちらのロックが取得されるかは、**isolationLevel** (下記参照) によって決まりますが、ほとんどの場合で読み取りには非排他的ロック、書き込みには排他的ロックが取得されます。ライターが排他的ロックを完了してリリースするまでリーダースレッドがブロックされるため (書き込みがトランザクションの一部である場合は長期化する可能性があります)、悲観的ロッキングは MVCC よりかなり多くのオーバーヘッドを必要にし、並行性も低下します。また、読み取りが継続することにより、書き込みの遅延が発生します。

JBoss Cache 3.0 で廃止された PESSIMISTIC より MVCC の方が通常は適切ですが、JBoss Enterprise Application Platform 5.0.0 では PESSIMISTIC がデフォルトとなっています。これは、セッションのユースケースには同じキャッシュの場所へアクセスする並行スレッドがなく、MVCC を使用する利点がありません。

- **OPTIMISTIC** (楽観的) ロッキングは、キャッシュにアクセスする各リクエストやトランザクションの「ワークスペース」を作成して PESSIMISTIC の並行性を向上しようとします。リクエストやトランザクションによるデータへのアクセスは (読み取りも含む)、ワークスペースへコピーされるため、オーバーヘッドが増加します。すべてのデータがバージョン化され、非トランザクションリクエストやトランザクションのコミットが終了すると、ワークスペースのデータバージョンがメインキャッシュと比較され、合致しない場合は例外が発行されます。合致する場合は、ワークスペースへの変更がメインキャッシュに適用されます。

OPTIMISTIC ロッキングは廃止されましたが、後方互換性を維持するために提供されます。ユーザーは低負荷で同じ利点を持つ MVCC を使用することが推奨されます。

**isolationLevel** 属性には、データベーススタイルの分離レベルのセマンティックに対応する **READ\_COMMITTED** と **REPEATABLE\_READ** の 2 つの値が使用できます。旧バージョンの JBoss Cache は 5 つのデータベース分離レベルすべてをサポートしていました。サポート対象外の分離レベルを設定すると、もっとも近いサポート対象レベルにアップグレードまたはダウングレードされます。

**REPEATABLE\_READ** はデフォルトの分離レベルで、旧バージョンの JBoss Cache との互換性を維持します。**READ\_COMMITTED** の分離は若干弱くなりますが、パフォーマンス面では **REPEATABLE\_READ** よりも優れています。

### 28.1.5. JGroups の統合

各 JBoss Cache インスタンスは内部で JGroups **Channel** を使用してグループ通信を処理します。JBoss Enterprise Application Platform 内部では、Enterprise Application Platform の JGroups Channel ファクトリサービスをキャッシュの **Channel** のソースとして使用することが強く推奨されます。本項では、チャンネルファクトリよりチャンネルを取得するようキャッシュを設定する方法について説明します。別の方法でチャンネルを設定したい場合は、JBoss Cache のドキュメントを参照してください。

#### CacheManager サービスより取得したキャッシュ

最も簡単な方法になります。CacheManager サービスは既にチャンネルファクトリへの参照を持っているため、使用する JGroups プロトコルスタック設定の名前のみを設定する必要があります。

**jboss-cache-manager-jboss-beans.xml** ファイル (「[CacheManager サービスによるデプロイメント](#)」を参照) よりキャッシュを設定する場合は、次をキャッシュ設定に追加します。値はプロトコルスタック設定の名前になります。

```
<property name="multiplexerStack">udp</property>
```

#### -jboss-beans.xml ファイルよりデプロイされたキャッシュ

JBoss Microcontainer の **-jboss-beans.xml** ファイル (「[-jboss-beans.xml ファイルからのデプロイメント](#)」を参照) よりキャッシュをデプロイする場合は、チャンネルファクトリサービスへの参照を挿入し、プロトコルスタック設定を指定する必要があります。

```
<property name="runtimeConfig">
  <bean class="org.jboss.cache.config.RuntimeConfig">
    <property name="muxChannelFactory"><inject bean="JChannelFactory"/>
  </property>
  </bean>
</property>
<property name="multiplexerStack">udp</property>
```

#### -service.xml ファイルよりデプロイされたキャッシュ

キャッシュ MBean を **-service.xml** ファイル (「[-service.xml ファイルからのデプロイメント](#)」を参照) よりデプロイする場合は、**CacheJmxWrapper** が MBean のクラスになります。このクラスは、**MuxChannelFactory** MBean 属性を公開します。この属性へチャンネルファクトリサービスの依存関係を挿入し、**MultiplexerStack** 属性よりプロトコルスタック名を設定します。

```
<attribute name="MuxChannelFactory"><inject bean="JChannelFactory"/>
</attribute>
<attribute name="MultiplexerStack">udp</attribute>
```

### 28.1.6. エビクション

エビクションは、キャッシュがデータ (通常、頻繁に使用されないデータ) を削除してメモリ制御を行えるようにします。カスタムキャッシュにエビクションを設定するには、使用できるオプションすべてを JBoss Cache ドキュメントで確認してください。JPA/Hibernate キャッシングへの設定は、<http://www.jboss.org/jbosscustering/docs/hibernate-jbosscache-guide-3.pdf> の「Using JBoss Cache as a Hibernate Second Level Cache」ガイドにあるエビクションの章を参照してください。Web セッションキャッシュにはエビクションを設定しないでください。分散可能セッションマネージャがエビクションに対応します。EJB 3 SFSB キャッシュに対しては、Enterprise Application Platform の標準の **sfsb-cache** 設定 (「[JBoss Enterprise Application Platform の CacheManager サービス](#)」参照) にあるエビクション設定を使用してください。各 Bean の設定に含まれている値を使用して EJB コンテナがエビクションを設定します。

### 28.1.7. キャッシュローダー

キャッシュローディングは、JBoss Cache がデータをメモリだけでなく永続ストアに保存できるようにします。データは、永続ストアのデータがメモリに反映されないオーバーフローか、メモリに保存されているデータのスーパーセットとなります。スーパーセットとなる場合、メモリにあるデータやメモリからエビクトされたデータは永続ストアに反映されます。どちらが使用されるかは、JBoss Cache のキャッシュロード設定にある **passivation** フラグの設定によって決まります。値が **true** の場合、データがメモリ内キャッシュからエビクトされた時書き込まれるオーバーフロー領域として永続ストアが機能します。

カスタムキャッシュにキャッシュローディングを設定したい場合は、使用できるすべてのオプションを JBoss Cache ドキュメントで確認してください。JPA/Hibernate キャッシュにキャッシュローディングを設定しないでください。データベース自体が永続ストアとして機能するため、キャッシュローダーの追加は不要になります。

Web セッションと EJB3 SFSB のキャッシングに使用されるキャッシュは非活性化を使用します。次に、これらのキャッシュに対するキャッシュローダー設定について説明します。

### 28.1.7.1. Web セッションキャッシュと SFSB キャッシュの CacheLoader 設定

HttpSession と SFSB の非活性化は、JBoss Cache のキャッシュローダー非活性化に依存して、非活性化されたセッションを保存し読み出します。そのため、Web アプリケーションのクラスター化セッションマネージャーや Bean の EJB コンテナによって使用されるキャッシュインスタンスを設定し、キャッシュローダーの非活性化を有効にしなければなりません。

ほとんどの場合、標準の Web セッションキャッシュや SFSB キャッシュに対してキャッシュローダーを変更する必要はありません。標準の JBoss Enterprise Application Platform の設定で適切なはずです。次は、デフォルトを変更したいユーザー向けの内容になります。

**standard-session-cache** 設定のキャッシュローダー設定を例にします。

```
<property name="cacheLoaderConfig">
  <bean class="org.jboss.cache.config.CacheLoaderConfig">
    <!-- Do not change these -->
    <property name="passivation">true</property>
    <property name="shared">>false</property>

    <property name="individualCacheLoaderConfigs">
      <list>
        <bean
class="org.jboss.cache.loader.FileCacheLoaderConfig">
          <!-- Where passivated sessions are stored -->
          <property
name="location">${jboss.server.data.dir}${/}session</property>
          <!-- Do not change these -->
          <property name="async">>false</property>
          <property name="fetchPersistentState">true</property>
          <property name="purgeOnStartup">true</property>
          <property name="ignoreModifications">>false</property>
          <property
name="checkCharacterPortability">>false</property>
        </bean>
      </list>
    </property>
  </bean>
</property>
```

説明

- **passivation** プロパティは **true** でなければなりません。
- **shared** プロパティは **false** でなければなりません。共有の永続ストアにセッションを非活性化しないでください。非活性化すると、別のノードがセッションをアクティベートした時にセッションが永続ストアより削除され、以前同じセッションを非活性化した別のノードにあるメモリからも削除されます。バックアップコピーを紛失します。

- **individualCacheLoaderConfigs** プロパティはキャッシュローダ設定のリストを許可します。JBC によって、キャッシュローダーをチェーンできるようになります。詳細は JBoss Cache のドキュメントを参照してください。セッション非活性化のユースケースでは、単一のキャッシュローダーで十分です。
- キャッシュローダー設定 Bean 上の **class** 属性は、キャッシュローダー実装の設定クラスを参照しなければなりません (**org.jboss.cache.loader.FileCacheLoaderConfig** または **org.jboss.cache.loader.JDBCCLoaderConfig**)。使用できる CacheLoader 実装の詳細は JBoss Cache のドキュメントを参照してください。JDBCCLoader を使用したい場合 (FileCacheLoader が使用するファイルシステムではなくデータベースへ永続させるため)、前述の **shared** プロパティに関するコメントを参照してください。共有データベースは使用しないようにしてください。最低でもデータベースの共有テーブルは使用しないでください。クラスタの各ノードは独自の保存場所が必要です。
- FileCacheLoaderConfig の **location** プロパティは、非活性化されたセッションが保存されるファイルシステムツリーのルートノードを定義します。デフォルトでは、JBoss Enterprise Application Platform 設定の **data** ディレクトリに保存されます。
- 非活性化されたセッションが迅速に永続ストアへ書き込まれるようにするため、**async** を **false** にしなければなりません。
- キャッシュの開始時に別のノードより転送されるセッションバックアップコピーのセットに非活性化されたセッションが含まれるようにするため、**fetchPersistentState** プロパティを **true** にしなければなりません。
- 過去にサーバーがシャットダウンした時に残存した期限切れのセッションデータが現在のデータセットを破損しないようにするため、**purgeOnStartup** を **true** にしなければなりません。
- **ignoreModifications** を **false** にする必要があります。
- マイナーなパフォーマンスの最適化を行うため、**checkCharacterPortability** を **false** にしなければなりません。

### 28.1.8. バディレプリケーション

バディレプリケーションは、クラスタの全てのインスタンスへのデータレプリケーションを抑制できるようにする JBoss Cache の機能です。各インスタンスはクラスタで 1 つ以上の「バディ」を選択し、特定のバディのみをレプリケートします。他のインスタンスがクラスタへ追加された時にメモリやネットワークトラフィックへの影響がないため、スケーラビリティが向上します。

他のノードにあるキャッシュがローカルにないデータを必要とする場合、クラスタの別のノードに提供を求めることができます。コピーを持っているノードは、「データグラビテーション」と呼ばれるプロセスの一部としてデータを提供します。新しいノードがデータの所有者となり、データのバックアップコピーをバディへ置きます。データをグラビテーションできるということは、データへの全要求がコピーを持っているノード上で発生しなくてもよいことを意味します。すべてのノードがデータへの要求を処理できます。しかし、データグラビテーションは負荷が高いため、頻繁に発生しないようにしなければなりません。同じデータを使用しているノードが障害を起こすか、シャットダウンして、対象のクライアントが別ノードへのフェールオーバーを強制された場合のみデータグラビテーションが発生するようにするのが理想的です。これにより、特定セッションへのすべての要求が通常単一のサーバーによって処理される場合、セッションアフィニティを持つセッションタイプのアプリケーション (スティッキーセッション) に対するバディレプリケーションが有効になります。

バディアプリケーションを Web セッションキャッシュと EJB3 SFSB キャッシュに対して有効にすることができます。他の標準クラスタ化サービス (JPA/Hibernate キャッシングなど) が使用するキャッシュ設定にバディレプリケーションを追加しないようにしてください。バディレプリケーション向けに

特別設計されていないサービスでは、バディレプリケーションが導入されてもまず適切に動作することはないでしょう。

バディレプリケーションの設定は比較的単純です。例として、CacheManager サービスの **standard-session-cache** 設定より、バディレプリケーション設定のセクションを見てみましょう。

```
<property name="buddyReplicationConfig">
  <bean class="org.jboss.cache.config.BuddyReplicationConfig">

    <!-- Just set to true to turn on buddy replication -->
    <property name="enabled">true</property>

    <!-- A way to specify a preferred replication group. We try
         and pick a buddy who shares the same pool name (falling
         back to other buddies if not available). -->
    <property name="buddyPoolName">default</property>

    <property name="buddyCommunicationTimeout">17500</property>

    <!-- Do not change these -->
    <property name="autoDataGravitation">false</property>
    <property name="dataGravitationRemoveOnFind">true</property>
    <property name="dataGravitationSearchBackupTrees">true</property>

    <property name="buddyLocatorConfig">
      <bean
class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
        <!-- The number of backup copies we maintain -->
        <property name="numBuddies">1</property>
        <!-- Means that each node will *try* to select a buddy on
             a different physical host. If not able to do so
             though, it will fall back to colocated nodes. -->
        <property name="ignoreColocatedBuddies">true</property>
      </bean>
    </property>
  </bean>
</property>
```

設定対象となる主な項目は次の通りです。

- **buddyReplicationEnabled** — バディレプリケーションを実行したい場合は **true**、クラスターのすべてのノードでデータをレプリケートする場合は **false** を設定します。 **false** の場合、バディレプリケーションの他の設定は関係ありません。
- **numBuddies** — 各ノードがステータスをレプリケートするバックアップノードの数です。
- **buddyPoolName** — クラスター内でノードの論理サブグループ化を有効にします。可能な場合、同じバディプール内のノードよりバディが選択されます。

**ignoreColocatedBuddies** スイッチは、キャッシュがバディを検索する時に、可能な場合はキャッシュと同じ物理ホスト上にあるバディを選択しないようにします。同じマシン上で稼働しているサーバーのみが見つかった場合、そのサーバーバディとして使用します。

**autoDataGravitation**、**dataGravitationRemoveOnFind**、**dataGravitationSearchBackupTrees** の設定は変更しないでください。これらの変更を変更すると、セッションレプリケーションが正しく動作しません。

## 28.2. 独自の JBOSS CACHE インスタンスをデプロイ

アプリケーションがカスタム使用できるよう JBoss Enterprise Application Platform 内部にユーザー独自の JBoss Cache インスタンスをデプロイすることは比較的一般的です。本項では、キャッシュをデプロイする方法を複数取り上げます。

### 28.2.1. CacheManager サービスによるデプロイメント

JBoss Cache を使用する標準の JBoss クラスタ化サービスは、Enterprise Application Platform の CacheManager サービスからキャッシュへの参照を取得します (「[JBoss Enterprise Application Platform の CacheManager サービス](#)」参照)。エンドユーザーアプリケーションも下記の方法で同様の動作をさせることができます。

「[CacheManager 設定の編集](#)」は CacheManager の「CacheConfigurationRegistry」Bean の設定になります。新しい設定を追加するには、追加する要素を Bean の **newConfigurations** <map> 内に追加します。

```
<bean name="CacheConfigurationRegistry"
class="org.jboss.ha.cachemanager.DependencyInjectedConfigurationRegistry">
    .....
    <property name="newConfigurations">
        <map keyClass="java.lang.String"
valueClass="org.jboss.cache.config.Configuration">

            <entry><key>my-custom-cache</key>
                <value>
                    <bean name="MyCustomCacheConfig"
class="org.jboss.cache.config.Configuration">
                        .... details of the my-custom-cache configuration
                    </bean>
                </value>
            </entry>
            .....
        </map>
    </property>
</bean>
```

設定例は、「[CacheManager 設定の編集](#)」を参照してください。

#### 28.2.1.1. CacheManager へのアクセス

CacheManager にキャッシュ設定を追加したら、次に CacheManager への参照をアプリケーションに提供します。これには 3 つの方法があります。

- 依存関係の挿入

アプリケーションが設定に JBoss Microcontainer を使用する場合、サービスに CacheManager を挿入するのが最も簡単な方法となります。

```
<bean name="MyService" class="com.example.MyService">
    <property name="cacheManager"><inject bean="CacheManager"/>
</property>
</bean>
```

- JNDI ルックアップ

CacheManager を JNDI でルックアップすることもできます。 **java:CacheManager** 以下にバインドされます。

```
import org.jboss.ha.cachemanager.CacheManager;

public class MyService {
    private CacheManager cacheManager;

    public void start() throws Exception {
        Context ctx = new InitialContext();
        cacheManager = (CacheManager)
ctx.lookup("java:CacheManager");
    }
}
```

### ● CacheManagerLocator

JBoss Enterprise Application Platform は、CacheManager へのアクセスに使用されるサービスロケーターオブジェクトも提供します。

```
import org.jboss.ha.cachemanager.CacheManager;
import org.jboss.ha.framework.server.CacheManagerLocator;

public class MyService {
    private CacheManager cacheManager;

    public void start() throws Exception {
        CacheManagerLocator locator =
CacheManagerLocator.getCacheManagerLocator();
        // Locator accepts as param a set of JNDI properties to help
in lookup;
        // this isn't necessary inside the Enterprise Application
Platform
        cacheManager = locator.getCacheManager(null);
    }
}
```

CacheManager への参照を取得すれば、使用方法は簡単です。希望する設定の名前を渡してキャッシュにアクセスします。CacheManager はキャッシュを起動しません。キャッシュの起動はアプリケーションが行いますが、キャッシュサーバーで実行されている他のアプリケーションによって既に起動されていることがあります。キャッシュを共有することもできます。アプリケーションはキャッシュの使用を終了しても、停止しません。キャッシュが使用されていないことを CacheManager に伝え、キャッシュを要求したすべての呼び出し元がキャッシュを解放した後、CacheManager がキャッシュを停止します。

```
import org.jboss.cache.Cache;
import org.jboss.ha.cachemanager.CacheManager;
import org.jboss.ha.framework.server.CacheManagerLocator;

public class MyService {
    private CacheManager cacheManager;
    private Cache cache;

    public void start() throws Exception {
        Context ctx = new InitialContext();
```



```

        cacheManager = (CacheManager) ctx.lookup("java:CacheManager");

        // "true" param tells the manager to instantiate the cache if
        // it doesn't exist yet
        cache = cacheManager.getCache("my-cache-config", true);

        cache.start();
    }

    public void stop() throws Exception {
        cacheManager.releaseCache("my-cache-config");
    }
}

```

POJO キャッシュのインスタンスへアクセスするために CacheManager を使用することもできます。

```

import org.jboss.cache.pojo.PojoCache;
import org.jboss.ha.cachemanager.CacheManager;
import org.jboss.ha.framework.server.CacheManagerLocator;

public class MyService {
    private CacheManager cacheManager;
    private PojoCache pojoCache;

    public void start() throws Exception {
        Context ctx = new InitialContext();
        cacheManager = (CacheManager) ctx.lookup("java:CacheManager");

        // "true" param tells the manager to instantiate the cache if
        // it doesn't exist yet
        pojoCache = cacheManager.getPojoCache("my-cache-config", true);

        pojoCache.start();
    }

    public void stop() throws Exception {
        cacheManager.releaseCache("my-cache-config");
    }
}

```

### 28.2.2. -service.xml ファイルからのデプロイメント

JBoss Enterprise Application Platform 4.x と同様に、**-service.xml** ファイルより JBoss Cache インスタンスを MBean サービスとしてデプロイすることもできます。JBoss 4.x と異なる点は、主に **mbean** 要素にある **code** 属性の値です。JBoss Enterprise Application Platform 4.x での値は **org.jboss.cache.TreeCache** ですが、JBoss Enterprise Application Platform 5.x では **org.jboss.cache.jmx.CacheJmxWrapper** になりました。例は次の通りです。

```

<?xml version="1.0" encoding="UTF-8"?>

<server>
    <mbean code="org.jboss.cache.jmx.CacheJmxWrapper"
          name="foo:service=ExampleCacheJmxWrapper">

        <attribute name="TransactionManagerLookupClass">

```

```

        org.jboss.cache.transaction.JBossTransactionManagerLookup
    </attribute>

    <attribute name="MuxChannelFactory"><inject bean="JChannelFactory"/>
</attribute>

    <attribute name="MultiplexerStack">udp</attribute>
    <attribute name="ClusterName">Example-EntityCache</attribute>
    <attribute name="IsolationLevel">REPEATABLE_READ</attribute>
    <attribute name="CacheMode">REPL_SYNC</attribute>
    <attribute name="InitialStateRetrievalTimeout">15000</attribute>
    <attribute name="SyncReplTimeout">20000</attribute>
    <attribute name="LockAcquisitionTimeout">15000</attribute>
    <attribute name="ExposeManagementStatistics">true</attribute>

</mbean>
</server>

```

**CacheJmxWrapper** 自体はキャッシュではありません (データの保存などは不可能)。名前が示す通り、JMX との統合を処理する **org.jboss.cache.Cache** 周囲のラッパーになります。

**CacheJmxWrapper** は、**CacheJmxWrapperMBean** MBean インターフェースの **Cache** 属性を公開します。キャッシュが必要なサービスは、この属性よりキャッシュへの参照を取得します。

### 28.2.3. -jboss-beans.xml ファイルからのデプロイメント

**-jboss-beans.xml** ファイルの JBoss Microcontainer スキーマを使用して POJO (Plain Old Java Object) が記述されている場合、JBoss Enterprise Application Platform 5 は **-service.xml** で説明した MBean サービスのデプロイメント同様に、POJO で構成されるサービスをデプロイすることができます。このようなファイルを作成したら、直接 **deploy** ディレクトリにデプロイするか、ear または sar にパッケージ化することができます。例は次の通りです。

```

<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

    <!-- First we create a Configuration object for the cache -->
    <bean name="ExampleCacheConfig"
        class="org.jboss.cache.config.Configuration">

        <!-- Externally injected services -->
        <property name="runtimeConfig">
            <bean name="ExampleCacheRuntimeConfig"
                class="org.jboss.cache.config.RuntimeConfig">
                <property name="transactionManager">
                    <inject bean="jboss:service=TransactionManager"
                        property="TransactionManager"/>
                </property>
                <property name="muxChannelFactory"><inject
bean="JChannelFactory"/></property>
            </bean>
        </property>

        <property name="multiplexerStack">udp</property>
        <property name="clusterName">Example-EntityCache</property>
        <property name="isolationLevel">REPEATABLE_READ</property>
    </bean>

```

```

        <property name="cacheMode">REPL_SYNC</property>
        <property name="initialStateRetrievalTimeout">15000</property>
        <property name="syncReplTimeout">20000</property>
        <property name="lockAcquisitionTimeout">15000</property>
        <property name="exposeManagementStatistics">true</property>

    </bean>

    <!-- Factory to build the Cache. -->
    <bean name="DefaultCacheFactory"
class="org.jboss.cache.DefaultCacheFactory">
        <constructor factoryClass="org.jboss.cache.DefaultCacheFactory" />
    </bean>

    <!-- The cache itself -->
    <bean name="ExampleCache" class="org.jboss.cache.Cache">
        <constructor factoryMethod="createCache">
            <factory bean="DefaultCacheFactory"/>
            <parameter class="org.jboss.cache.config.Configuration"><inject
bean="ExampleCacheConfig"/></parameter>
            <parameter class="boolean">>false</false>
        </constructor>
    </bean>

    <bean name="ExampleService" class="org.foo.ExampleService">
        <property name="cache"><inject bean="ExampleCache"/></property>
    </bean>

</deployment>

```

上記の多くが JBoss Cache **Configuration** オブジェクトの作成になります。これは、CacheManager サービスの設定と同じになります (「[CacheManager 設定の編集](#)」参照)。この例では、CacheManager サービスをキャッシュファクトリとして使用していないため、独自のファクトリ Bean を作成し、キャッシュの作成に使用します (「ExampleCache」Bean)。そして、「ExampleCache」が必要とするサービス (架空) へ挿入されます。

上記の例で使用されている **RuntimeConfig** オブジェクトを見てください。マイクロコンテナが可視できる **TransactionManager** や JGroups **ChannelFactory** などの外部リソースは、この **RuntimeConfig** へ依存関係が挿入されます。この例では、Enterprise Application Platform における他の配備記述子の一部に、参照された Bean が既に記述されていることを前提としています。

上記の設定を使用すると、JMX は「ExampleCache」キャッシュを表示できません。ここで、キャッシュが JMX にバインドされる別の方法を見てみましょう。

```

<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

    <!-- First we create a Configuration object for the cache -->
    <bean name="ExampleCacheConfig"
        class="org.jboss.cache.config.Configuration">

        .... same as above

    </bean>

```

```
<bean name="ExampleCacheJmxWrapper"
class="org.jboss.cache.jmx.CacheJmxWrapper">

    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
    (name="foo:service=ExampleCacheJmxWrapper",

exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
    registerDirectly=true)
    </annotation>

    <property name="configuration"><inject bean="ExampleCacheConfig"/>
</property>

</bean>

<bean name="ExampleService" class="org.foo.ExampleService">
    <property name="cache"><inject bean="ExampleCacheJmxWrapper"
property="cache"/></property>
</bean>

</deployment>
```

この例では、「ExampleCacheJmxWrapper」 Bean により設定からキャッシュが作成されます。**CacheJmxWrapper** はキャッシュの MBean インターフェースを提供する JBoss Cache クラスです。`<annotation>` 要素を追加すると、JBoss Microcontainer **@JMX** アノテーションを Bean へバインドします。その結果、デプロイメントプロセスの一部として JBoss Enterprise Application Platform が Bean を JXM に登録します。

実際の基礎の **org.jboss.cache.Cache** インスタンスは、**cache** プロパティにて **CacheJmxWrapper** より使用できます。この例は、キャッシュを「ExampleService」に挿入するための使用方法を表しています。

## パート IV. レガシーの **EJB** サポート

## 第29章 JBOSS 上の EJB

### EJB コンテナ設定とアーキテクチャー

JBoss EJB コンテナのアーキテクチャーは、モジュール式のプラグインアプローチを採用しています。EJB コンテナの主なアспектを開発者がカスタマイズしたプラグインやインターセプターで置き換えることが可能です。このアプローチは、ニーズにあわせて EJB コンテナ動作のカスタマイズを微調整することができます。EJB コンテナ動作のほとんどが EJB JAR **META-INF/jboss.xml** 記述子やデフォルトのサーバー全体で対応する **standardjboss.xml** 記述子を使い設定が可能です。コンテナアーキテクチャーについて学習していくとともに、この章全体で様々な設定機能を見ていきます。

### 29.1. EJB クライアント側のビュー

EJB コンテナの説明ですが、ホームおよびリモートプロキシを使った EJB のクライアントビューについてまず見ていきます。コンテナプロバイダーが EJB 実装向けに **javax.ejb.EJBHome** および **javax.ejb.EJBObject** を生成します。クライアントは、EJB bean インスタンスを直接参照するわけではなく、bean ホームインターフェースを実装する **EJBHome** と bean リモートインターフェースを実装する **EJBObject** を参照します。図29.1「JBoss における EJBHome プロキシの構成」では、EJB ホームプロキシの構成と EJB デプロイメントとの関係について説明しています。

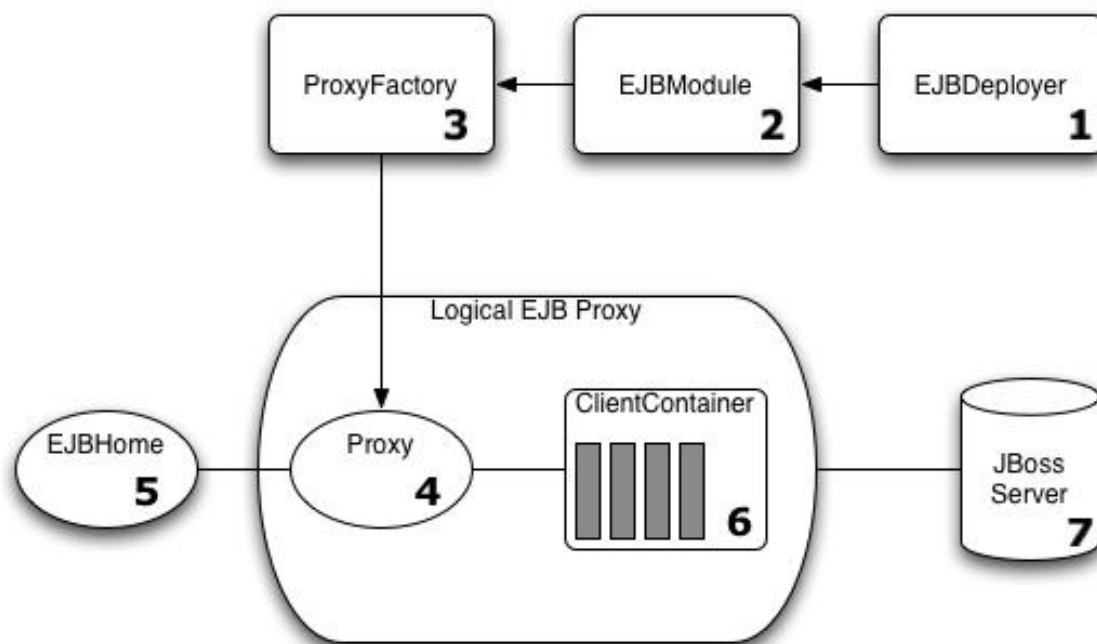


図29.1 JBoss における EJBHome プロキシの構成

この図で番号付きの項目は以下のとおりです。

1. **EJBDeployer** (**org.jboss.ejb.EJBDeployer**) を呼び出し、EJB JAR をデプロイします。**EJBModule** (**org.jboss.ejb.EJBModule**) を作成し、デプロイメントメタデータをカプセル化します。
2. **EJBModule** ライフサイクルの作成フェーズで、**EJBModuleInvoker-proxy-bindings** メタデータを元に EJB ホームとリモートインターフェースプロキシを管理する **EJBProxyFactory** (**org.jboss.ejb.EJBProxyFactory**) を作成します。EJB には複数のプロキシファクトリを関連付けることができ、これから、この定義方法について見ていきます。
3. **ProxyFactory** は論理プロキシを構築し、ホームを JNDI にバインドします。論理プロキシは、動的な **Proxy** (**java.lang.reflect.Proxy**)、プロキシが公開する EJB のホームイン

ターフェース、**ClientContainer** (`org.jboss.proxy.ClientContainer`) 形式の **ProxyHandler** (`java.lang.reflect.InvocationHandler`) 実装、クライアント側のインターセプターで構成されています。

4. **EJBProxyFactory** で作成したプロキシは、標準のダイナミックプロキシです。**EJBModule** メタデータで定義されているように、EJB ホームとリモートインターフェースをプロキシ化するシリアル可能なオブジェクトです。プロキシは、このプロキシに関連付けられた **ClientContainer** ハンドラーを使って、強く型付けされた EJB インターフェースで出されたリクエストを型付けされていない呼び出しに変換します。これは、クライアントをルックアップする EJB ホームインターフェースとして JNDI にバインドされた動的プロキシインスタンスです。クライアントが EJB ホームをルックアップすると、ホームプロキシは **ClientContainer** とそのインターセプターとともに、クライアント VM へ移植されます。動的なプロキシを使うと、他の多くの EJB コンテナでは必要となる EJB 固有の編集手順を行わなくてもよくなります。
5. EJB ホームインターフェースは、`ejb-jar.xml` 記述子で宣言し、**EJBModule** メタデータから利用可能です。動的プロキシの主なプロパティは、これらのプロキシを公開するインターフェースを実装するために表示されます。これは、Java の強く型付けされたシステムにおいては当てはまります。プロキシは、ホームインターフェースのどれにでもキャストでき、またプロキシ化するインターフェースの全詳細を提供するプロキシを反映することができます。
6. プロキシは、**ClientContainer** ハンドラーへのインターフェースのいずれかを使った呼び出しを委譲します。ハンドラーで必要とされる唯一のメソッドは、**public Object invoke(Object proxy, Method m, Object[] args) throws Throwable** です。**EJBProxyFactory** は、**ClientContainer** を作成し、これを **ProxyHandler** として割り当てます。**ClientContainer** のステータスには、**InvocationContext** (`org.jboss.invocation.InvocationContext`) とインターセプターのチェーン (`org.jboss.proxy.Interceptor`) が含まれています。**InvocationContext** は以下を含みます。
  - **Proxy** が関連付けられている EJB コンテナ MBean の JMX **ObjectName**
  - EJB 向けの `javax.ejb.EJBMetaData`
  - EJB ホームインターフェースの JNDI 名
  - トランSPORT固有の呼び出し (`org.jboss.invocation.Invoker`)
 インターセプターチェーンは、EJB ホームとリモートインターフェースの動作を組み立てる機能ユニットで構成されています。これは、`jboss.xml` 記述子についての説明時に見ていきますが、EJB の設定可能なアスペクトで、インターセプターのマークアップが **EJBModule** メタデータに含まれています。インターセプター (`org.jboss.proxy.Interceptor`) は、様々な EJB タイプ、セキュリティ、トランザクション、トランSPORTを処理します。独自のインターセプターの追加も可能です。
7. プロキシに関連付けられたトランSPORT固有の呼び出しは、EJB メソッド呼び出しのトランSPORT詳細を処理するサーバー側の分離呼び出しへ紐付けされています。分離呼び出しは、JBoss サーバー側のコンポーネントです。

クライアント側のインターセプターの設定は `jboss.xml` `client-interceptors` を使って行います。**ClientContainer** 呼び出しメソッドが呼び出されると、型なしの **Invocation** (`org.jboss.invocation.Invocation`) を作成しリクエストをカプセル化します。これがインターセプターチェーンに渡されます。チェーンの最後のインターセプターがトランSPORTハンドラーとなり、サーバーへリクエストを送信し、返信を取得する方法を把握しており、トランSPORT固有の内容に対応します。

クライアントのインターセプター設定の利用例にあるよう

に、**server/production/standardjboss.xml** のデフォルトのステートレスセッション bean 設定を見てみましょう。例29.1「[Standard Stateless SessionBean 設定からのクライアントインターセプター](#)」は、Standard Stateless SessionBean が参照する **stateless-rmi-invoker** クライアントのインターセプター設定について示しています。

### 例29.1 Standard Stateless SessionBean 設定からのクライアントインターセプター

```
<invoker-proxy-binding>
  <name>stateless-rmi-invoker</name>
  <invoker-mbean>jboss:service=invoker,type=jrmp</invoker-mbean>
  <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
    <proxy-factory-config>
      <client-interceptors>
        <home>

<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
  <interceptor call-by-value="false">
    org.jboss.invocation.InvokerInterceptor
  </interceptor>
  <interceptor call-by-value="true">
    org.jboss.invocation.MarshallingInvokerInterceptor
  </interceptor>
        </home>
      </bean>
    </client-interceptors>
  </proxy-factory-config>
</invoker-proxy-binding>

<interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
  <interceptor call-by-value="false">
    org.jboss.invocation.InvokerInterceptor
  </interceptor>
  <interceptor call-by-value="true">
    org.jboss.invocation.MarshallingInvokerInterceptor
  </interceptor>
  </bean>
</client-interceptors>
</proxy-factory-config>
</invoker-proxy-binding>

<container-configuration>
  <container-name>Standard Stateless SessionBean</container-name>
  <call-logging>>false</call-logging>
  <invoker-proxy-binding-name>stateless-rmi-invoker</invoker-proxy-binding-name>
  <!-- ... -->
</container-configuration>
```



これは、ステートレス bean の設定をオーバーライドする EJB JAR **META-INF/jboss.xml** がない場合に利用される、ステートレスセッション bean 向けのクライアントインターセプター設定です。各クライアントインターセプターが提供する機能は以下の通りです。

- **org.jboss.proxy.ejb.HomeInterceptor**: **getHomeHandle**、**getEJBMetaData** を処理し、クライアント VM 内でローカルにて **EJBHome** インターフェースのメソッドを削除します。その他のメソッドは、次のインターセプターに伝搬されます。
- **org.jboss.proxy.ejb.StatelessSessionInterceptor**: クライアント VM 内でローカルにて **EJBObject** インターフェースの **toString**、**equals**、**hashCode**、**getHandle**、**getEJBHome**、**isIdentical** メソッドを処理します。その他のメソッドは、次のインターセプターに伝搬されます。
- **org.jboss.proxy.SecurityInterceptor**: 現在のセキュリティコンテキストをメソッド呼び出しと関連付け、他のインターセプターやサーバーで利用できるようにします。
- **org.jboss.proxy.TransactionInterceptor**: アクティブなトランザクションと呼び出しメソッドの呼び出しを関連付け、他のインターセプターが利用できるようにします。
- **org.jboss.invocation.InvokerInterceptor**: メソッド呼び出しのディスパッチをトランスポート固有の呼び出しにカプセル化します。クライアントがサーバーと同じ VM で実行しているか、またこのような場合はオプションで参照渡しへの呼び出しをルーティングするか把握しています。クライアントがサーバー VM の外にある場合、このインターセプターは、この呼び出しを呼び出しコンテキストが紐付いたトランスポート呼び出しに委譲します。[例29.1「Standard Stateless SessionBean 設定からのクライアントインターセプター」](#) 設定の場合、これは、**jboss:service=invoker,type=jrmp** (**JRMPInvoker** サービス) が紐付いている呼び出しスタブとなります。

**org.jboss.invocation.MarshallingInvokerInterceptor**: VM 内の呼び出しを最適化しないように **InvokerInterceptor** を継承します。これを使いメソッド呼び出しに **call-by-value** セマンティクスを強制的に利用させます。

### 29.1.1. EJB プロキシ設定の指定

EJB 呼び出しトランスポートとクライアントのプロキシインターセプタースタックを指定するには、EJB JAR **META-INF/jboss.xml** 記述子あるいはサーバーの **standardjboss.xml** 記述子のいずれかにある **invoker-proxy-binding** を定義する必要があります。各種デフォルトの EJB コンテナ設定や標準の RMI/JRMP や RMI/IIOP トランスポートプロトコル向けに **standardjboss.xml** 記述子に定義されているデフォルトの **invoker-proxy-bindings** が複数あります。現在のデフォルトプロキシ設定は以下のとおりです。

- **entity-rmi-invoker**: エンティティ bean 向けの RMI/JRMP 設定
- **clustered-entity-rmi-invoker**: クラスター化されたエンティティ bean 向けの RMI/JRMP 設定
- **stateless-rmi-invoker**: ステートレスセッション bean 向けの RMI/JRMP 設定
- **clustered-stateless-rmi-invoker**: クラスター化されたステートレスセッション bean 向けの RMI/JRMP 設定
- **stateful-rmi-invoker**: クラスター化されたステートフルセッション bean 向けの RMI/JRMP 設定
- **clustered-stateful-rmi-invoker**: クラスター化されたステートフルセッション bean 向けの RMI/JRMP 設定

- **message-driven-bean**: メッセージ駆動型 bean 向けの JMS 呼び出し
- **singleton-message-driven-bean**: シングルトンメッセージ駆動型 bean 向けの JMS 呼び出し
- **message-inflow-driven-bean**: メッセージインフロー駆動型 bean の JMS 呼び出し
- **jms-message-inflow-driven-bean**: 標準のメッセージ駆動型 bean 向けの JMS インフロー呼び出し
- **iiop**: セッション bean およびエンティティ bean と共に利用する RMI/IIOP

新規のプロトコルバインディングを導入し、プロキシファクトリあるいはクライアント側のインターセプタースタックをカスタマイズするには、新たに**invoker-proxy-binding**を定義する必要があります。プロキシ設定を指定する際の完全な **invoker-proxy-binding** DTD 部分については、[図 29.2 「invoker-proxy-binding スキーマ」](#) にあります。

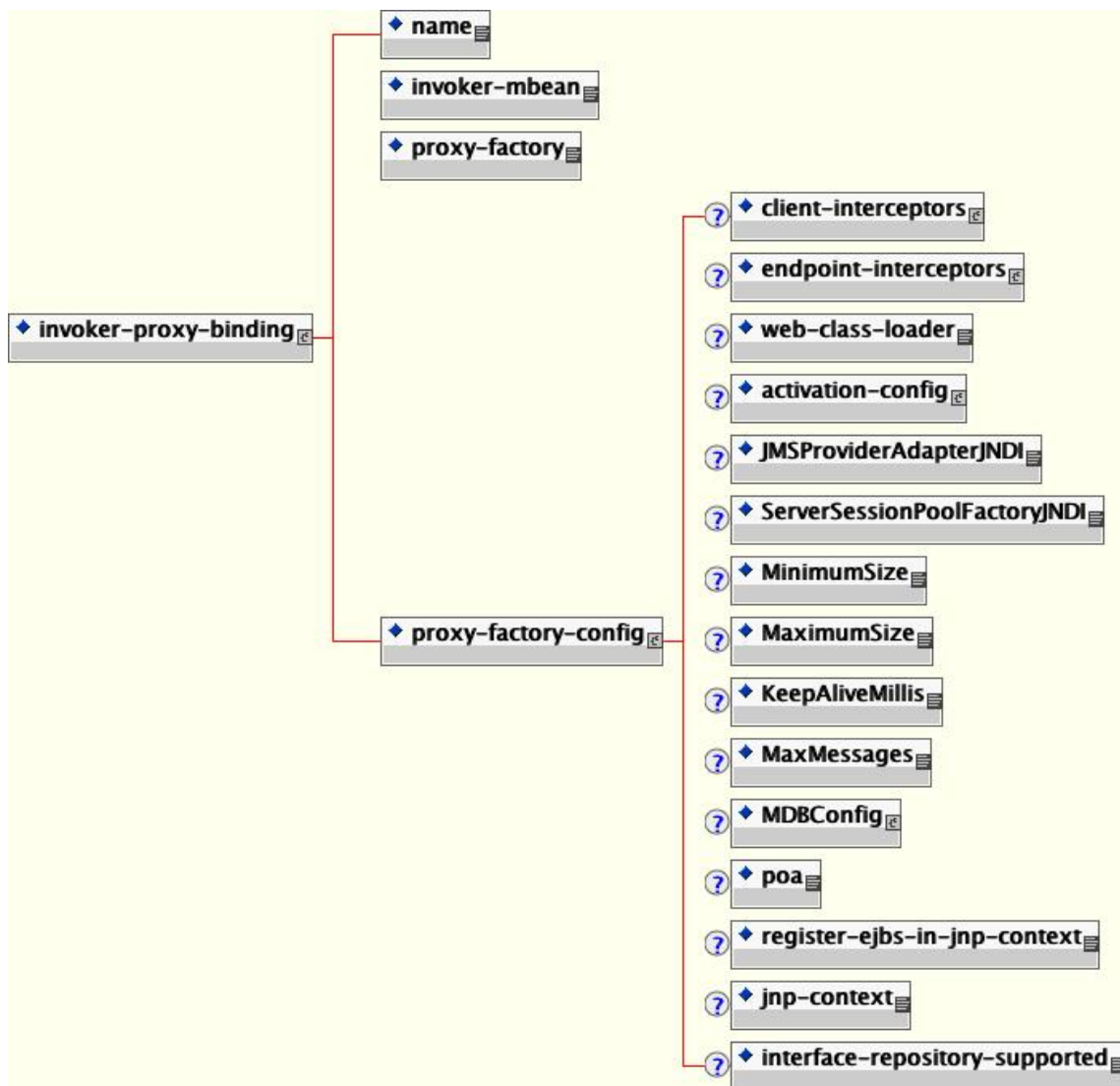


図29.2 invoker-proxy-binding スキーマ

**invoker-proxy-binding** の子要素は以下の通りです。

- **name**: **name** 要素は、**invoker-proxy-binding**の一意名を渡します。追加のプロキシバインディングを指定するためにデフォルトのプロキシバインディングと EJB デプロイメントレベルが設定されている場合は、この名前を使い、EJB コンテナ設定からのバインディングを参照します。サーバー側の EJB コンテナ設定を制御する **jboss.xml** 要素をみていただくとこれがどのように行われているかがわかります。

- **invoker-mbean: invoker-mbean** 要素は、プロキシ呼び出しが紐付けられる分離呼び出し MBean サービスの JMX **ObjectName** 文字列を渡します。
- **proxy-factory: proxy-factory** 要素は、**org.jboss.ejb.EJBProxyFactory** インターフェースを実装する必要があるプロキシファクトリの完全修飾クラス名を指定します。**EJBProxyFactory** は、プロキシの設定、プロトコル固有の呼び出し、コンテキストの関連付けといった処理を行います。**EJBProxyFactory** インターフェースの現在の JBoss 実装は以下を含みます。
  - **org.jboss.proxy.ejb.ProxyFactory**: RMI/JRMP 固有のファクトリ
  - **org.jboss.proxy.ejb.ProxyFactoryHA**: クラスター RMI/JRMP 固有のファクトリ
  - **org.jboss.ejb.plugins.jms.JMSContainerInvoker**: JMS 固有のファクトリ
  - **org.jboss.proxy.ejb.IORFactory**: RMI/IIOP 固有のファクトリ
- **proxy-factory-config: proxy-factory-config** 要素は、**proxy-factory** 実装の追加情報を指定します。残念ながら、各種要素は構造化されていません。プロキシファクトリの各型を適用しているのは、一部の要素のみとなっています。子要素は、3つの呼び出しプロトコル (RMI/RJMP、RMI/IIOP、JMS) に分類されます。

RMI/JRMP 固有のプロキシファクトリ

り、**org.jboss.proxy.ejb.ProxyFactory**、**org.jboss.proxy.ejb.ProxyFactoryHA** については、以下の要素を適用します。

- **client-interceptors: client-interceptors** は、ホームとリモートを定義します。また、オプションで複数の値を持つプロキシインターセプタースタックも定義します。
- **web-class-loader**: 動的なクラスローディングができるように、web クラスローダーは、プロキシに関連付けられるべき **org.jboss.web.WebClassLoader** インスタンスを定義します。

以下の **proxy-factory-config** は、RMI でアクセスしたエンティティ bean 用です。

```
<proxy-factory-config>
  <client-interceptors>
    <home>
      <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
  <interceptor call-by-value="false">
    org.jboss.invocation.InvokerInterceptor
  </interceptor>
  <interceptor call-by-value="true">
    org.jboss.invocation.MarshallingInvokerInterceptor
  </interceptor>
    </home>
  <bean>

<interceptor>org.jboss.proxy.ejb.EntityInterceptor</interceptor>
  <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
  <interceptor call-by-value="false">
    org.jboss.invocation.InvokerInterceptor
```

```

        </interceptor>
        <interceptor call-by-value="true">
            org.jboss.invocation.MarshallingInvokerInterceptor
        </interceptor>
    </bean>
</list-entity>

<interceptor>org.jboss.proxy.ejb.ListEntityInterceptor</interceptor>
    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
    <interceptor call-by-value="false">
        org.jboss.invocation.InvokerInterceptor
    </interceptor>
    <interceptor call-by-value="true">
        org.jboss.invocation.MarshallingInvokerInterceptor
    </interceptor>
</list-entity>
</client-interceptors>
</proxy-factory-config>

```

RMI/IIOP 固有のプロキシファクトリ **org.jboss.proxy.ejb.IORFactory** については、以下の要素を適用します。

- **web-class-loader**: 動的なクラスローディングができるように、web クラスローダーは、プロキシに関連付けられるべき **org.jboss.web.WebClassLoader** インスタンスを定義します。
- **poa**: 移植可能なオブジェクトアダプターの用途。有効な値は、**per-servant** と **shared** となっています。
- **register-ejbs-in-jnp-context**: EJB が JNDI で登録されるべきか指定するフラグ
- **jnp-context**: EJB を登録する JNDI コンテキスト
- **interface-repository-supported**: これは、デプロイ済みの EJB が独自の CORBA インターフェースレポジトリを持つか否かを指定します。

以下は、IIOP でアクセスした EJB の **proxy-factory-config** です。

```

<proxy-factory-config>
    <web-class-loader>org.jboss.iiop.WebCL</web-class-loader>
    <poa>per-servant</poa>
    <register-ejbs-in-jnp-context>true</register-ejbs-in-jnp-context>
    <jnp-context>iiop</jnp-context>
</proxy-factory-config>

```

JMS 固有のプロキシファクトリ、**org.jboss.ejb.plugins.jms.JMSContainerInvoker** については、以下の要素を適用します。

- **MinimumSize**: これは、MDB 処理の最小プールサイズを指定します。デフォルト値は 1 です。
- **MaximumSize**: これは、JMS のデスティネーションで許容できる同時 MDB 数の上限を指定します。デフォルト値は、15 です。
- **MaxMessages**: これは、**javax.jms.QueueConnection** と **javax.jms.TopicConnection** インターフェースの **createConnectionConsumer** メソッド

ドに対する `maxMessages` パラメーター値だけでなく、`javax.jms.TopicConnection` の `createDurableConnectionConsumer` メソッドに対する `maxMessages` パラメーター値を指定します。これは1度のサーバーセッションに割り当て可能な最大メッセージ数です。この値は、JMS プロバイダーが対応していると表明していない場合は、デフォルト値から変更すべきではありません。

- **KeepAliveMillis**: これは、セッションプール内のセッション間のキープアライブ間隔をミリ秒単位で指定します。デフォルト値は、30000 (30 秒) です。
- **MDBConfig**: MDB JMS 接続動作の設定。要素野中で対応しているものは、以下のとおりです。
  - **ReconnectIntervalSec**: JMS サーバーへの接続回復を試行するまでの待機時間 (秒単位)
  - **DeliveryActive**: MDB が起動時に有効かどうか。デフォルトは True となっています。
  - **DLQConfig**: MDB のデッドレターキューの設定。メッセージの再送が多すぎる場合に利用されます。
  - **JMSProviderAdapterJNDI**: `java:/` 名前空間の JMS プロバイダーアダプターの JNDI 名。これは、MDB には必須で、`org.jboss.jms.jndi.JMSProviderAdapter` を実装する必要があります。
  - **ServerSessionPoolFactoryJNDI**: JMS プロバイダーのセッションプールファクトリの `java:/` 名前空間にあるセッションプールの JNDI 名。これは MDB には必須で、`org.jboss.jms.asf.ServerSessionPoolFactory` を実装する必要があります。

例29.2「[JMSContainerInvoker proxy-factory-config 例](#)」では、`standardjboss.xml` 記述子から抜粋した、サンプルの `proxy-factory-config` を提示しています。

### 例29.2 JMSContainerInvoker proxy-factory-config 例

```
<proxy-factory-config>
  <JMSProviderAdapterJNDI>DefaultJMSProvider</JMSProviderAdapterJNDI>

  <ServerSessionPoolFactoryJNDI>StdJMSPool</ServerSessionPoolFactoryJNDI>
    <MinimumSize>1</MinimumSize>
    <MaximumSize>15</MaximumSize>
    <KeepAliveMillis>30000</KeepAliveMillis>
    <MaxMessages>1</MaxMessages>
    <MDBConfig>
      <ReconnectIntervalSec>10</ReconnectIntervalSec>
      <DLQConfig>
        <DestinationQueue>queue/DLQ</DestinationQueue>
        <MaxTimesRedelivered>10</MaxTimesRedelivered>
        <TimeToLive>0</TimeToLive>
      </DLQConfig>
    </MDBConfig>
  </proxy-factory-config>
```

## 29.2. EJB サーバー側のビュー

EJB 呼び出しはすべて JBoss サーバーがホストする EJB コンテナに行き着く必要があります。本章では、呼び出しが JBoss サーバー VM に移動し、JMX バスを経由して EJB コンテナへ行き着く方法を見ていきます。

### 29.2.1. 分離呼び出し：トランスポートの仲介

MBean サービスの RMI 互換インターフェースを公開するといったコンテキストで分離呼び出しアーキテクチャについて前述しました。ここでは、分離呼び出しをつかい、EJB コンテナホームと bean インターフェースをクライアントにどのように公開するか見て行きましょう。呼び出しアーキテクチャの全体像は、[図29.3「トランスポート呼び出しサーバー側のアーキテクチャ」](#) で提示されています。

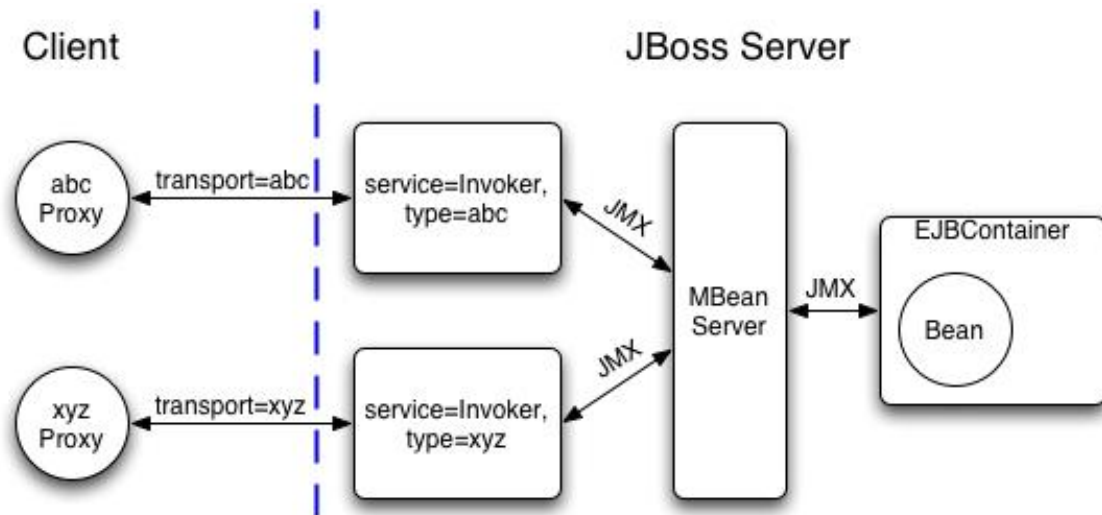


図29.3 トランスポート呼び出しサーバー側のアーキテクチャ

ホームプロキシの各タイプには、呼び出しへのバインドや関連付けられたトランスポートプロトコルがあります。コンテナには、複数の呼び出しプロトコルが同時に有効になっている場合があります。`jboss.xml` ファイルでは、**invoker-proxy-binding-name** は、**invoker-proxy-binding/name** 要素にマッピングします。**container-configuration** レベルでは、これは、コンテナにデプロイされる EJB で使うデフォルトの呼び出しを指定します。bean レベルでは、**invoker-bindings** が1つ以上の呼び出しを指定し、EJB コンテナ MBean とあわせて利用します。

任意の EJB デプロイメント1つに複数の呼び出しを指定した場合、ホームプロキシに一意の JNDI バインディング先を渡す必要があります。これは、**invoker/jndi-name** 要素値で指定できます。1つの EJB に対して複数の呼び出しが存在する場合に問題がもう1つあります。それは、EJB が別の bean を呼び出した際に取得したリモートのホームとインターフェースを処理する方法です。クライアントが呼び出しを開始するのに使ったプロキシと、渡されたリモートホームとインターフェースがそれぞれ互換がきくように、このようなインターフェースは、外部の EJB の呼び出しに利用したのと同じ呼び出しを使う必要があります。**invoker/ejb-ref** 要素を使うと、参照呼び出し型と一致する **ejb-ref** の参照先 EJB ホームに対し、プロトコルに依存しない ENC **ejb-ref** からホームプロキシバインディングにマッピングすることができます。

カスタムの **JRMPInvoker** MBean の使用しセッション bean 向けに圧縮ソケットを圧縮可能にする例は、testsuite の **org.jboss.test.jrmp** パッケージにあります。以下の例では、カスタムの **JRMPInvoker** 設定と、ステートレスセッション bean へのマッピングについて示しています。

```
<server>
  <mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
    name="jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory"
```

```

>
    <attribute name="RMIObjectPort">4445</attribute>
    <attribute name="RMIClientSocketFactory">
        org.jboss.test.jrmp.ejb.CompressionClientSocketFactory
    </attribute>
    <attribute name="RMIServerSocketFactory">
        org.jboss.test.jrmp.ejb.CompressionServerSocketFactory
    </attribute>
</mbean>

</server>

```

ここでは、ポート 4445 にバインドし、トランスポートレベルで圧縮ができるようカスタムのソケットファクトリを使用するように、デフォルトの **JRMPInvoker** がカスタマイズされています。

```

<?xml version="1.0"?>
<!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<!-- The jboss.xml descriptor for the jrmp-comp.jar ejb unit -->
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>StatelessSession</ejb-name>
            <configuration-name>Standard Stateless
SessionBean</configuration-name>
            <invoker-bindings>
                <invoker>
                    <invoker-proxy-binding-name>
                        stateless-compression-invoker
                    </invoker-proxy-binding-name>
                    <jndi-name>jrmp-compressed/StatelessSession</jndi-
name>
                </invoker>
            </invoker-bindings>
        </session>
    </enterprise-beans>

    <invoker-proxy-bindings>
        <invoker-proxy-binding>
            <name>stateless-compression-invoker</name>
            <invoker-mbean>

jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory
            </invoker-mbean>
            <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-
factory>
            <proxy-factory-config>
                <client-interceptors>
                    <home>

<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

```

```

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </home>
    <bean>
        <interceptor>

org.jboss.proxy.ejb.StatelessSessionInterceptor
    </interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </bean>
    </client-interceptors>
</proxy-factory-config>
</invoker-proxy-binding>
</invoker-proxy-bindings>
</jboss>

```

**StatelessSession EJB invoker-bindings** 設定は、**stateless-compression-invoker** を JNDI 名 **jrmp-compressed/StatelessSession** の配下にバインドされたホームインターフェースと共に利用するように指定しています。**stateless-compression-invoker** は、先ほど宣言したカスタムの JRMP 呼び出しにリンクされます。

以下の例 (**org.jboss.test.hello** testsuite パッケージ) は、**HttpInvoker** を使い、RMI/HTTP プロトコルを利用できるように、ステートレスセッション bean を設定しています。

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>HelloWorldViaHTTP</ejb-name>
            <jndi-name>helloworld/HelloHTTP</jndi-name>
            <invoker-bindings>
                <invoker>
                    <invoker-proxy-binding-name>
                        stateless-http-invoker
                    </invoker-proxy-binding-name>
                </invoker>
            </invoker-bindings>
        </session>
    </enterprise-beans>
    <invoker-proxy-bindings>
        <!-- A custom invoker for RMI/HTTP -->
        <invoker-proxy-binding>
            <name>stateless-http-invoker</name>
            <invoker-mbean>jboss:service=invoker,type=http</invoker-mbean>
            <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-
factory>
            <proxy-factory-config>
                <client-interceptors>
                    <home>

```



```

<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </home>
    <bean>
        <interceptor>
org.jboss.proxy.ejb.StatelessSessionInterceptor
        </interceptor>

<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </bean>
    </client-interceptors>
    </proxy-factory-config>
    </invoker-proxy-binding>
    </invoker-proxy-bindings>
</jboss>

```

これは、カスタムの **stateless-http-invoker** という名前の **invoker-proxy-binding** を定義します。これは、分離呼び出しとして **HttpInvoker** MBean を使います。 **jboss:service=invoker,type=http** の名前は、 **http-invoker.sar/META-INF/jboss-service.xml** 記述子にあるように、 **HttpInvoker** MBean のデフォルト名で、このサービス記述子の一部については以下に示しています。

```

<!-- The HTTP invoker service configuration -->
<mbean code="org.jboss.invocation.http.server.HttpInvoker"
    name="jboss:service=invoker,type=http">
    <!-- Use a URL of the form
http://<hostname>:8080/invoker/EJBInvokerServlet
    where <hostname> is InetAddress.getHostname value on which the
server
    is running. -->
    <attribute name="InvokerURLPrefix">http://</attribute>
    <attribute
name="InvokerURLSuffix">:8080/invoker/EJBInvokerServlet</attribute>
    <attribute name="UseHostName">true</attribute>
</mbean>

```

クライアントプロキシは、EJB 呼び出しのコンテンツを **HttpInvoker** サービス設定で指定している **EJBInvokerServlet** URL に掲載します。

### 29.2.2. HA JRMPInvoker - クラスターされた RMI/JRMP トランスポート

**org.jboss.invocation.jrmp.server.JRMPInvokerHA** サービスは、**JRMPInvoker** の拡張で、クラスター対応の呼び出しとなっています。**JRMPInvokerHA** は、**JRMPInvoker** の全属性に完全対応しています。つまり、ポート、インターフェース、ソケットトランスポートのカスタマイズバインディ

ングは、クラスター化された RMI/JRMP でも利用できることになります。HA RMI プロキシのクラスタリングアーキテクチャーと実装に関する追加情報は、JBoss クラスタリングの文書を参照してください。

### 29.2.3. HA HttpInvoker - クラスターされた RMI/HTTP トランスポート

RMI/HTTP 層は、クラスター環境で呼び出しのソフトウェアロードバランシングが可能となります。HTTP 呼び出しの HA 対応拡張が追加され、HA-RMI/JRMP クラスタリングからその機能の多くを借りることができます。

HA-RMI/HTTP を有効にするには、EJB コンテナに対して呼び出しを設定する必要があります。これは、**jboss.xml** 記述子、あるいは **standardjboss.xml** 記述子で設定できます。例29.3「HA-RMI/HTTP に対する jboss.xml ステートレスセッション設定」は、**org.jboss.test.hello** testsuite パッケージから抜粋したステートレスセッション設定例を示しています。

#### 例29.3 HA-RMI/HTTP に対する jboss.xml ステートレスセッション設定

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>HelloWorldViaClusteredHTTP</ejb-name>
      <jndi-name>helloworld/HelloHA-HTTP</jndi-name>
      <invoker-bindings>
        <invoker>
          <invoker-proxy-binding-name>
            stateless-httpHA-invoker
          </invoker-proxy-binding-name>
        </invoker>
      </invoker-bindings>
      <clustered>true</clustered>
    </session>
  </enterprise-beans>
  <invoker-proxy-bindings>
    <invoker-proxy-binding>
      <name>stateless-httpHA-invoker</name>
      <invoker-mbean>jboss:service=invoker,type=httpHA</invoker-
mbean>
      <proxy-factory>org.jboss.proxy.ejb.ProxyFactoryHA</proxy-
factory>
      <proxy-factory-config>
        <client-interceptors>
          <home>

          <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>

          <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

          <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

          <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
          </home>
          <bean>
            <interceptor>
              org.jboss.proxy.ejb.StatelessSessionInterceptor
```

```

        </interceptor>

    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>

    <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>

    <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </bean>
    </client-interceptors>
</proxy-factory-config>
</invoker-proxy-binding>
</invoker-proxy-bindings>
</jboss>

```

**stateless-httpHA-invoker** invoker-proxy-binding は、**jboss:service=invoker,type=httpHA** 呼び出しサービスを参照しています。このサービスは、以下のように設定することができます。

```

<mbean code="org.jboss.invocation.http.server.HttpInvokerHA"
    name="jboss:service=invoker,type=httpHA">
    <!-- Use a URL of the form
        http://<hostname>:8080/invoker/EJBInvokerHAServlet
        where <hostname> is InetAddress.getHostname value on which the
server
        is running.
    -->
    <attribute name="InvokerURLPrefix">http://</attribute>
    <attribute
name="InvokerURLSuffix">:8080/invoker/EJBInvokerHAServlet</attribute>
    <attribute name="UseHostName">true</attribute>
</mbean>

```

呼び出しプロキシが使う URL は、クラスターノードでデプロイされているような、**EJBInvokerHAServlet** マッピングです。クラスター全体の **HttpInvokerHA** インスタンスは、http URL 候補となるものを集め、クライアント側のプロキシが、フェールオーバーおよび／もしくはロードバランシングに利用できるようにしています。

## 29.3. EJB コンテナ

EJB コンテナは、特定の EJB クラスを管理するコンポーネントです。JBoss では、デプロイされた EJB の設定ごとに **org.jboss.ejb.Container** のインスタンスが 1 つ作成されます。インスタンス化される実際のオブジェクトは、**Container** のサブクラスで、コンテナインスタンスの作成は、**EJBDeployer** MBean が管理します。

### 29.3.1. EJBDeployer MBean

**org.jboss.ejb.EJBDeployer** MBean は、EJB コンテナを作成します。デプロイの準備ができた EJB JAR があるとし、**EJBDeployer** は、EJB のタイプごとに、必要な EJB コンテナを 1 つずつ作成し開始します。**EJBDeployer** の設定可能な属性は以下のとおりです。

- **VerifyDeployments**: EJB 検証を実行すべきか示す boolean フラグ。これにより、デプロイメントユニットの EJB が EJB 2.1 仕様に準拠しているか検証します。これを true に設定すると、デプロイメントが有効か確認できるため便利です。

- **VerifierVerbose:** 検証プロセスから出される検証の失敗／警告をどの程度詳しく出すか制御する boolean。
- **StrictVerifier:** 厳密な検証を有効化あるいは無効化する boolean。厳密な検証が有効であれば、検証レポートにエラーがない場合にのみ、EJB はデプロイされます。
- **CallByValue:** 値渡し of セマンティクスをデフォルトで使用すべきであると指示をだす boolean フラグ。
- **ValidateDTDs:** 宣言した DTD に対して、**ejb-jar.xml** と **jboss.xml** 記述子が有効であることを検証するべきかを示す boolean フラグ。これを true にすると、配備記述子が有効であることを確認できるため便利です。
- **MetricsEnabled:** **metricsEnabled=true** 属性で印付けされているコンテナインターセプターが設定に含めるべきかを示す boolean フラグ。これにより、コンテナインターセプター設定 (これは On/Off が可) を含むメトリクスタイプのインターセプターを定義できるようになります。
- **WebServiceName:** Web サービス MBean の JMX ObjectName 文字列。これにより、動的クラスローディングに対応可能にする EJB クラスの動的クラスローディングに対応できるようになります。
- **TransactionManagerServiceName:** JTA トランザクションマネージャーの JMX ObjectName 文字列。これには、**javax.transaction.TransactionManager** インスタンスを返す **TransactionManager** と呼ばれる属性を持たせる必要があります。

デプロイヤーには、**deploy** と **undeploy** という、2つの中心的なメソッドが含まれています。**deploy** メソッドは、EJB JAR、あるいは有効な EJB JAR と同じ構造のディレクトリ (開発目的で便利) を参照している URL を取得します。デプロイメントが行われると、同じ URL 上で **undeploy** が呼び出されるとアンデプロイすることができます。すでにデプロイ済みの URL で **deploy** が呼び出されると、**undeploy** が呼び出され、その URL のデプロイメントが行われます。JBoss は実装およびインターフェースクラス両方の再デプロイメントに対応しており、変更されたクラスはどれでも読み込みます。これにより、稼働中のサーバーを停止することなく EJB の開発と更新を行うことができます。

EJB JAR のデプロイメント中に、**EJBDeployer** とこれに関連付けられたクラスは、EJB を検証、一意の EJB ごとにコンテナを作成、デプロイメント設定情報でコンテナを初期化するという、3つの機能を果たします。以下の章で、各機能について説明していきます。

### 29.3.1.1. EJB デプロイメントの検証

**EJBDeployer** の **VerifyDeployments** 属性が true の場合、デプロイヤーはデプロイメントの EJB を検証します。この検証で、EJB が EJB 仕様に準拠しているか確認します。これは、EJB デプロイメントユニットに、必要なホームとリモート、ローカルホーム、ローカルインターフェースが含まれているかの検証も行います。また、これらのインターフェースに表示されるオブジェクトが正しい型で、また実装クラスに必要なメソッドが存在しているかも確認します。EJB 開発者やデプロイヤーは適切な EJB JAR を正しく構築する際に多くの手順を踏む必要があるため間違いを起こしやすいので、便利な動作としてデフォルトで有効になっています。検証の段階で間違いを突き止め、どこを修正すべきか示すようなエラーを出しデプロイメントが失敗するように試みています。

おそらく、EJB 記述する上で最も問題となる部分は、bean 実装とリモートとホームのインターフェースの間、さらに配備記述子設定との間が分離される点です。これらの各種要素を簡単に同期しないようにできます。この問題を回避できるようにするツールの 1 つに **XDoclet** があります。このツールは、EJB bean 実装クラスの中でカスタムの **JavaDoc** のようなタグを使い、関連の bean インターフェース、配備記述子、関連オブジェクトを生成することができます。さらなる詳細については、**XDoclet** ホームページ、<http://sourceforge.net/projects/xdoclet> を参照してください。

### 29.3.1.2. EJB をコンテナにデプロイ

**EJBDeployer** が行う最も重要な役割は、EJB コンテナを作成し、EJB をコンテナにデプロイすることです。デプロイメントフェーズでは、EJB JAR 内で EJB を反復し、**ejb-jar.xml** および **jboss.xml** 配備記述子で記述されているように、bean クラスやメタデータを抽出します。EJB JAR の各 EJB に対して、以下の手順を実行します。

- EJB の型 (ステートレス、ステートフル、BMP エンティティ、CMP エンティティ、あるいはメッセージ駆動) に従い、**org.jboss.ejb.Container** のサブクラスを作成します。コンテナに一意的な **ClassLoader** が割り当てられ、この **ClassLoader** からローカルリソースをロードすることができます。 **ClassLoader** の一意性を利用し、標準の **java:comp** JNDI 名前空間と他の J2EE コンポーネントを分類することができます。
- **jboss.xml** と **standardjboss.xml** 記述子を統合したものから取ったコンテナの設定可能属性をすべて設定します。
- コンテナに設定されているようにコンテナインターセプターを作成し、追加します。
- アプリケーションオブジェクトとコンテナを関連付けます。このアプリケーションオブジェクトは、J2EE エンタープライズアプリケーションを表し、複数の EJB および Web コンテキストを含む場合があります。

EJB がすべて正常にデプロイされると、アプリケーションが開始され、その後、順に全コンテナを開始し EJB をクライアント側で利用できるようにします。EJB のいずれかがデプロイメントに失敗すると、デプロイメントの例外がスローされ、そのデプロイメントモジュールが失敗します。

### 29.3.1.3. コンテナ設定情報

JBoss は、EJB コンテナの設定のすべて、あるいはその多くを **jboss\_4\_0.dtd** を構成する XML ファイルを使い外部に置いています。コンテナ設定情報に関連する DTD は、[図29.4「コンテナ設定関連の jboss\\_4\\_0 DTD 要素」](#)にあります。

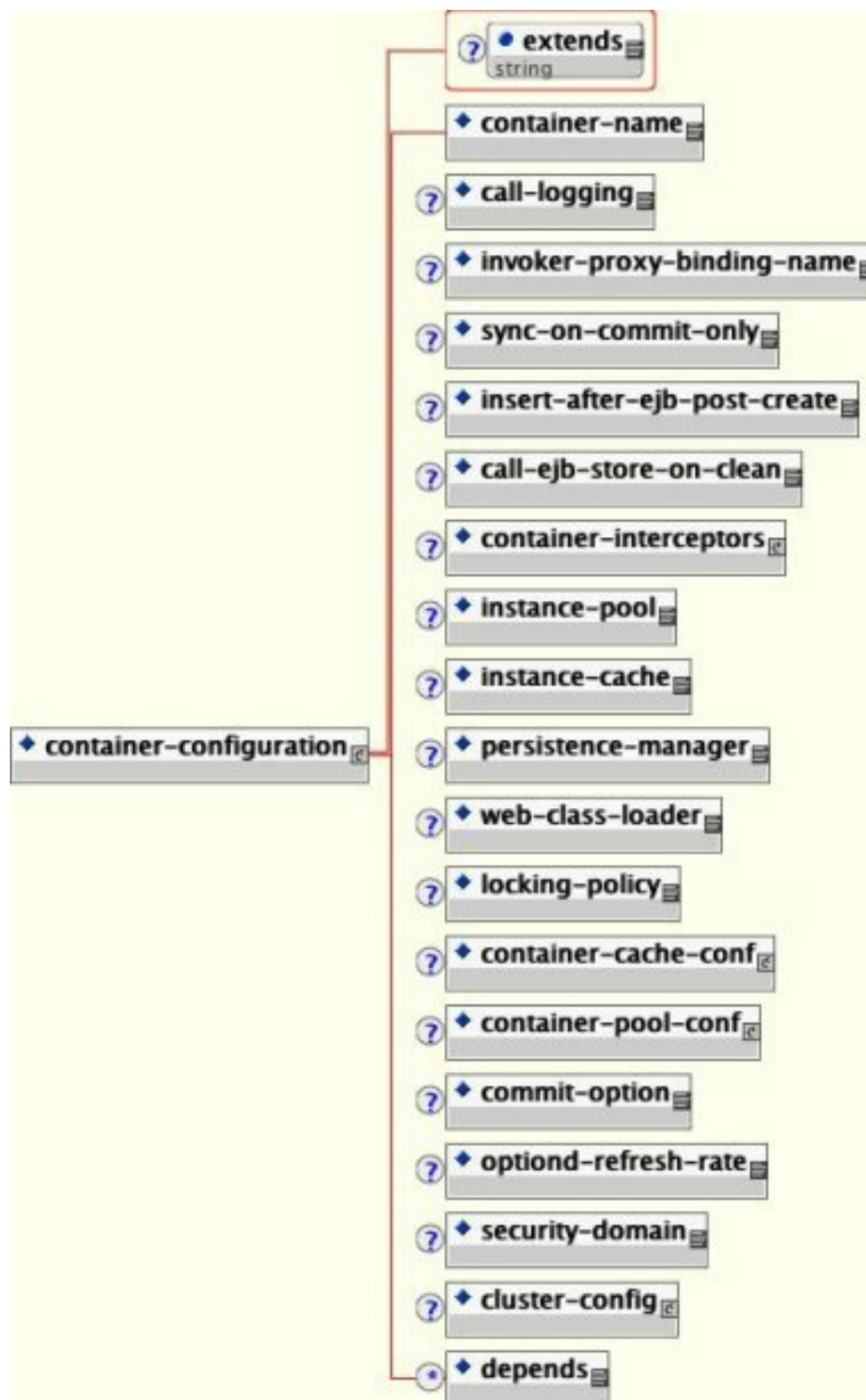


図29.4 コンテナ設定関連の jboss\_4\_0 DTD 要素

**container-configuration** 要素とそのサブ要素は **container-name** 要素で提示されているように、コンテナのタイプ毎にコンテナ構成の設定を指定します。各設定は、デフォルトの呼び出しタイプ、コンテナインターセプターのマークアップ、インスタンスキャッシュ／プール、そのサイズ、永続マネージャー、セキュリティなどの情報を指定します。これには、JBoss コンテナアーキテクチャを詳細まで把握していなければならない情報が大量にあるので、JBoss では EJB の4つのタイプに対する標準設定を同梱しています。この設定ファイルは、**standardjboss.xml** と呼ばれ、EJB を使う設定ファイルセットの **conf** ディレクトリに置かれています。以下は、**standardjboss.xml** からの **container-configuration** 例となっています。

```

<container-configuration>
  <container-name>Standard CMP 2.x EntityBean</container-name>
  <call-logging>false</call-logging>
  <invoker-proxy-binding-name>entity-rmi-invoker</invoker-proxy-binding-
name>
  <sync-on-commit-only>false</sync-on-commit-only>
  <insert-after-ejb-post-create>false</insert-after-ejb-post-create>
  <call-ejb-store-on-clean>true</call-ejb-store-on-clean>
  <container-interceptors>

<interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</intercep
tor>
  <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
  <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>

<interceptor>org.jboss.ejb.plugins.CallValidationInterceptor</interceptor>
  <interceptor metricsEnabled="true">
    org.jboss.ejb.plugins.MetricsInterceptor
  </interceptor>

<interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor</intercepto
r>
  <interceptor>
org.jboss.resource.connectionmanager.CachedConnectionInterceptor
  </interceptor>

<interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</inter
ceptor>

<interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</inter
ceptor>
  </container-interceptors>
  <instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-
pool>
  <instance-
cache>org.jboss.ejb.plugins.InvalidableEntityInstanceCache</instance-
cache>
  <persistence-
manager>org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager</persistence-
manager>
  <locking-
policy>org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock</locking-
policy>
  <container-cache-conf>
    <cache-
policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cache-
policy>

```

```

<cache-policy-conf>
  <min-capacity>50</min-capacity>
  <max-capacity>1000000</max-capacity>
  <overager-period>300</overager-period>
  <max-bean-age>600</max-bean-age>
  <resizer-period>400</resizer-period>
  <max-cache-miss-period>60</max-cache-miss-period>
  <min-cache-miss-period>1</min-cache-miss-period>
  <cache-load-factor>0.75</cache-load-factor>
</cache-policy-conf>
</container-cache-conf>
<container-pool-conf>
  <MaximumSize>100</MaximumSize>
</container-pool-conf>
<commit-option>B</commit-option>
</container-configuration>

```

これらの2つの例は、コンテナ設定のオプションがいかに拡張できるかを説明しています。コンテナ設定情報は、2つのレベルで指定可能です。1つは、設定ファイルセットのディレクトリに含まれている **standardjboss.xml** ファイル内で、2つ目は、EJB JAR レベルで指定可能です。EJB JAR **META-INF** ディレクトリに **jboss.xml** ファイルを置くことで、**standardjboss.xml** ファイルのコンテナ設定をオーバーライドするか、あるいは新しく名前をつけたコンテナ設定を指定することも可能です。こうすることで、コンテナの設定が大幅に柔軟になります。前述したように、コンテナ設定属性はすべて、外部に置かれているため、変更も可能です。知識の豊富な開発者は、インスタンスプールやキャッシュなど、特別なコンテナコンポーネントを実装し、標準のコンテナ設定と簡単に統合し、特定のアプリケーションや環境で動作の最適化を図ることさえ可能です。

EJB デプロイメントがどのようにコンテナ設定を選択するかは、**jboss/enterprise-beans/<type>/configuration-name** 要素が明示的か暗黙的かにより変わります。**configuration-name** 要素は、**container-configurations/container-configuration** へのリンクとなっており、参照 EJB に利用するコンテナ設定はどれかを指定します。これは、**configuration-name** 要素から **container-name** 要素をリンクしています。

EJB 定義に **container-configuration** 要素を含めることで、EJB のクラスごとにコンテナ設定を指定することができます。通常、新規コンテナ設定の定義がサポートされていても、完全な定義を行わないようです。**jboss.xml** レベルの **container-configuration** は、一般的に、**standardjboss.xml** 記述子にある **container-configuration** の1つあるいは複数の部分をオーバーライドすることで利用しています。これは、既存の **standardjboss.xml** **container-configuration/container-name** 名を **container-configuration/extends** 属性の値として参照している、**container-configuration** を指定することで実行できます。以下の例は、**Standard Stateless SessionBean** 設定の拡張である、新規の **Secured Stateless SessionBean** 設定を定義しています。

```

<?xml version="1.0"?>
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>EchoBean</ejb-name>
      <configuration-name>Secured Stateless
SessionBean</configuration-name>
      <!-- ... -->
    </session>
  </enterprise-beans>
  <container-configurations>
    <container-configuration extends="Standard Stateless SessionBean">

```



```

        <container-name>Secured Stateless SessionBean</container-name>
        <!-- Override the container security domain -->
        <security-domain>java:/jaas/my-security-domain</security-
domain>
    </container-configuration>
</container-configurations>
</jboss>

```

EJB がデプロイメントユニットの EJB JAR にコンテナ設定仕様を提供しない場合、コンテナファクトリは EJB のタイプを元に **standardjboss.xml** 記述子からコンテナ設定を選択します。そのため、実際は、暗黙的な **configuration-name** 要素が EJB 型ごとに存在し、デフォルトのコンテナ設定へ EJB 型をマッピングしたものは以下の通りになります。

- container-managed persistence entity version 2.0 = Standard CMP 2.x EntityBean
- container-managed persistence entity version 1.1 = Standard CMP EntityBean
- bean-managed persistence entity = Standard BMP EntityBean
- stateless session = Standard Stateless SessionBean
- stateful session = Standard Stateful SessionBean
- message driven = Standard Message Driven Bean

デフォルトの bean タイプを使いたい場合は、EJB が利用するコンテナ設定はどれかを指定する必要があります。おそらく、**configuration-name** を含み、必要なものを備え持った記述子を提供していますが、これは単にスタイルの問題です。

EJB が利用するコンテナ設定を指定する方法を理解いただき、デプロイメントユニットレベルのオーバーライドの定義が可能になったため、次の章で **container-configuration** 子要素を見ていきます。要素の多くは、インターフェースクラス実装を指定しますが、このインターフェース実装の設定は他の要素から影響を受けるため、設定要素に手を加える前に、**org.jboss.metadata.XmlLoadable** インターフェースについて理解する必要があります。

**XmlLoadable** インターフェースは、シンプルなインターフェースでメソッド 1 つしか含んでいません。インターフェースの定義は以下の通りです。

```

import org.w3c.dom.Element;
public interface XmlLoadable
{
    public void importXml(Element element) throws Exception;
}

```

クラスは、このインターフェースを実装し、そのクラス設定を XML 文書にて指定できるようになります。文書の root 要素は、**importXml** メソッドに渡されるはずで、以下の章にてコンテナ設定の要素について説明していくため、これについていくつか例を参照できるはずで、

#### 29.3.1.3.1. container-name 要素

**container-name** 要素は、設定ごとに一意名を指定します。コンテナ設定に関して、EJB の **configuration-name** 要素を **container-name** の値に設定することで、EJB は特定のコンテナ設定と関連付けられます。

#### 29.3.1.3.2. call-logging 要素

**call-logging** 要素は、**LogInterceptor** はコンテナへのメソッド呼び出しをログするべきかを指示する値に boolean (true あるいは false) が来るであろうと予測します。これは、細かく設定可能なロギング API を提供する log4j に移行するうえで若干古くなってきています。

#### 29.3.1.3.3. invoker-proxy-binding-name 要素

**invoker-proxy-binding-name** 要素は、利用するデフォルトの呼び出し名を指定します。bean レベルの **invoker-bindings** 指定がない場合、**invoker-proxy-binding-name** 要素の値と **invoker-proxy-binding** の名前が一致する **invoker-proxy-binding** を使い、ホームとリモートのプロキシを作成します。

#### 29.3.1.3.4. sync-on-commit-only 要素

これは、パフォーマンスの最適化について設定します。このパフォーマンスの最適化を行うと、エンティティ bean のステータスがコミット時のみにデータベースと同期されるようになります。通常、トランザクション内の bean のステータスはすべて、finder メソッドが呼び出された時、あるいは remove メソッドが呼び出されたときなどに同期されなければなりません。

#### 29.3.1.3.5. insert-after-ejb-post-create

これは別のエンティティ bean の最適化ですが、これは、**ejbPostCreate** メソッドが呼び出されるまで、新規エンティティ bean に対するデータベースの挿入コマンドが実行されません。こうすることで、デフォルトの挿入後にアップデートを行うことなく、通常の CMP フィールドと CMR フィールドを1回の挿入で設定することができます。結果、関係フィールドに null 値を可能にするための要件を削除できます。

#### 29.3.1.3.6. call-ejb-store-on-clean

この仕様によると、トランザクションでインスタンスが変更されなかった場合にもトランザクションがコミットされた場合、コンテナはエンティティ bean インスタンスで **ejbStore** メソッドを呼び出す必要があります。これを false に設定すると、JBoss はダーティオブジェクトに対して **ejbStore** だけを呼び出すようになります。

#### 29.3.1.3.7. container-interceptors 要素

**container-interceptors** 要素は、1つ以上のインターセプター要素を指定し、このインターセプター要素はコンテナにメソッドインターセプターチェーンとして設定されます。インターセプター要素の値は、**org.jboss.ejb.Interceptor** インターフェース実装の完全修飾名です。コンテナインターセプターは、**linked-list** 構造を形成しており、この構造を使い EJB メソッドの呼び出しを渡します。**MBeanServer** がコンテナにメソッド呼び出しを渡すと、チェーンにある最初のインターセプターが呼び出されます。最後のインターセプターは、bean にあるビジネスメソッドを呼び出します。**Interceptor** インターフェースについては、本章でコンテナプラグインフレームワークについて触れる際に後述します。一般的に、セキュリティ、トランザクション、永続性、スレッドセーフに関する EJB コントラクトはインターセプターから派生しているため、既存の標準 EJB インターセプター設定を変更する際は注意が必要です。

#### 29.3.1.3.8. instance-pool 要素

**instance-pool** 要素は、**org.jboss.ejb.InstancePool** インターフェースの完全修飾クラス名を指定し、コンテナ **InstancePool** として利用します。InstancePool インターフェースについては、本章でコンテナプラグインフレームワークについて触れる際に後述します。

#### 29.3.1.3.9. container-pool-conf 要素

`container-pool-conf` を `InstancePool` 実装クラスに渡します。この `InstancePool` 実装クラスは、`XmlLoadable` インターフェイスが実装された場合に `instance-pool` 要素により渡されます。現在の JBoss `InstancePool` 実装は、図29.5「`container-pool-conf` 要素の DTD」で提示している要素へ対応可能にする `org.jboss.ejb.plugins.AbstractInstancePool` クラスから来ています。

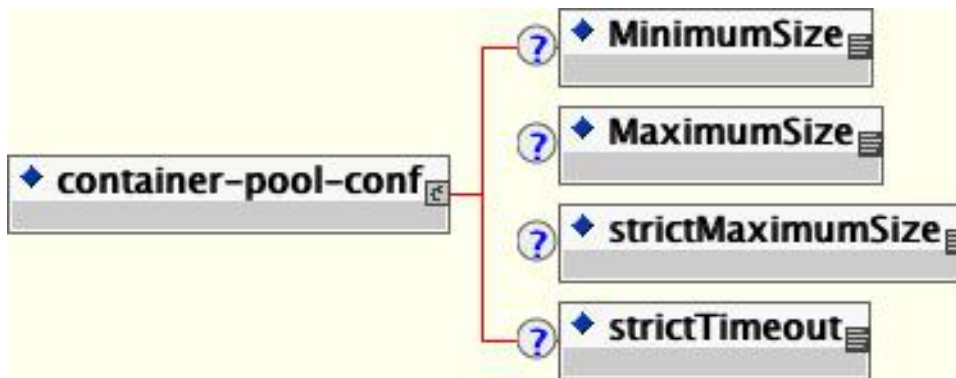


図29.5 `container-pool-conf` 要素の DTD

- **MinimumSize:** JBoss では現在 `InstancePool` を `MinimumSize` 値にシードしませんが、`MinimumSize` 要素は、プールに保存できるインスタンスの最小数を渡します。
- **MaximumSize:** `MaximumSize` は、プールインスタンスの上限数を指定します。`MaximumSize` のデフォルトの用途は、考えているものと少し違うかもしれません。プール `MaximumSize` は、EJB インスタンスの最大利用可能数ですが、同時リクエストの数が `MaximumSize` の値を超えた場合、さらにインスタンスを作成することができます。
- **strictMaximumSize:** EJB の最大並行処理数をプールの `MaximumSize` に制限したい場合、`strictMaximumSize` 要素を `true` に設定する必要があります。`strictMaximumSize` が `true` の場合、`MaximumSize` EJB インスタンスのみがアクティブとなります。`MaximumSize` が有効なインスタンスがある場合、インスタンスがプールに開放されるまでその後に続くリクエストはブロックされます。`strictMaximumSize` のデフォルト値は、`false` です。
- **strictTimeout:** リクエストがインスタンスのプールオブジェクトの待機をブロックする時間は、`strictTimeout` 要素で制御されます。`strictTimeout` は、`MaximumSize` が有効なインスタンスがある場合にプールヘインスタンスを返すまでの待機時間をミリ秒単位で定義します。0以下の値の場合は待機時間なしとなります。リクエストのインスタンス待機時間がタイムアウトした場合、`java.rmi.ServerException` が生成され呼び出しが中断されます。これは `Long` として解析されるため、最大の許容待機時間は 9,223,372,036,854,775,807 あるいは約 292,471,208 年で、これがデフォルト値となっています。

### 29.3.1.3.10. instance-cache 要素

`instance-cache` 要素は、`org.jboss.ejb.InstanceCache` インターフェイス実装の完全修飾名を指定します。この要素は、ID が関連付けられている EJB タイプがエンティティ bean とステートフルセッション bean のみであるため、これらの bean に対してのみ意味をなします。`InstanceCache` インターフェイスについては、コンテナプラグインフレームワークについて本章で後述する際に触れます。

### 29.3.1.3.11. container-cache-conf 要素

`container-cache-conf` 要素は、`XmlLoadable` インターフェイスに対応している場合、`InstanceCache` 実装に渡されます。現在の JBoss `InstanceCache` 実装はすべて、`org.jboss.ejb.plugins.AbstractInstanceCache` クラスから来ています。このクラスは、`XmlLoadable` インターフェイスに対応し、インスタンスキャッシュストアとして利用する

`org.jboss.util.CachePolicy` 実装の完全修飾名として `cache-policy` 子要素を利用します。`cache-policy-conf` 子要素は、`XmlLoadable` インターフェースに対応していれば、`CachePolicy` 実装に渡されます。対応していない場合は、`cache-policy-conf` は確認なしに無視されます。

現在の `cache-policy-conf` 子要素アレイに対応している、`standardjboss.xml` 設定が利用する `CachePolicy` の JBoss 実装が2つあります。これらの実装クラスは、`org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy` は `org.jboss.ejb.plugins.LRUStatefulContextCachePolicy` となっています。`LRUEnterpriseContextCachePolicy` は、エンティティ bean コンテナに、`LRUStatefulContextCachePolicy` はステートフルセッション bean コンテナにより利用されます。このキャッシュポリシーは両方、[図29.6「container-cache-conf 要素の DTD」](#)にある以下の `cache-policy-conf` 子要素に対応しています。

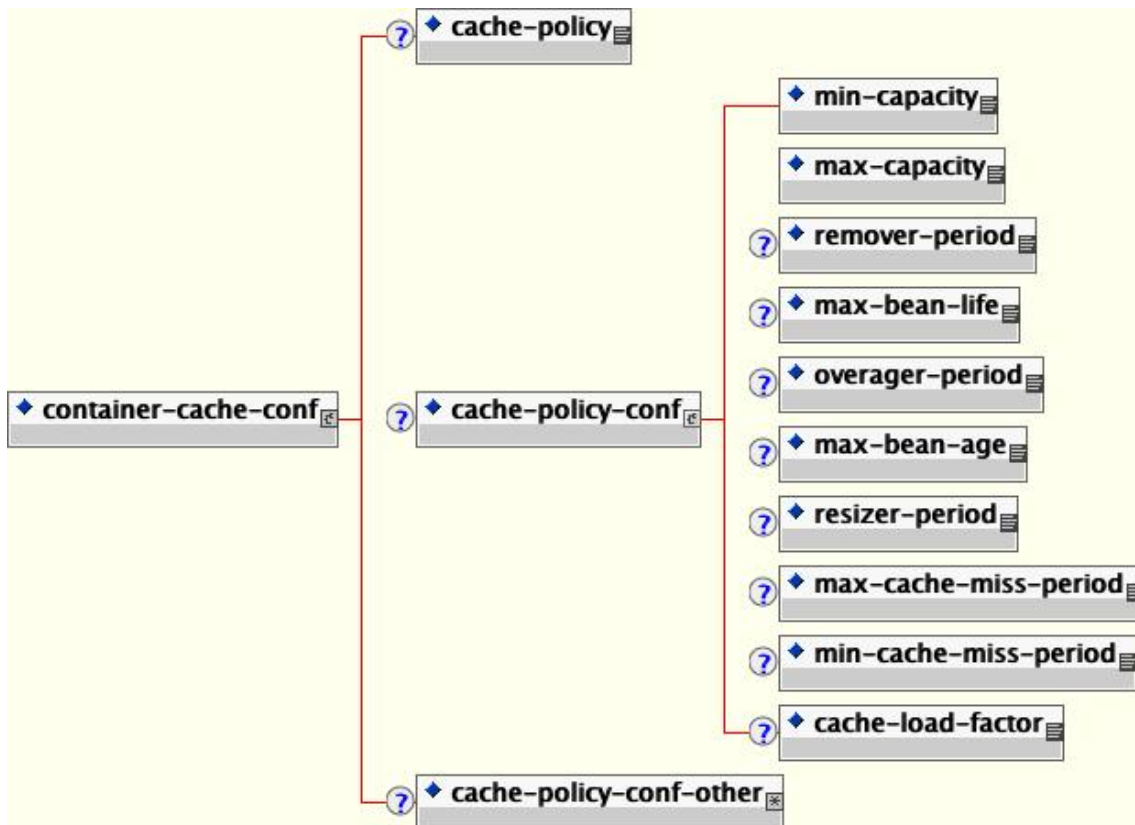


図29.6 container-cache-conf 要素の DTD

- **min-capacity:** このキャッシュの最小容量を指定します。
- **max-capacity:** このキャッシュの最大容量を指定しますが、`min-capacity` より小さい数値には設定できません。
- **overager-period:** overager タスクを実行する間隔を秒単位で指定します。overager タスクの目的は、キャッシュに `max-bean-age` 要素値の一定年齢以上となっている bean が含まれているか確認することです。この基準に合致する bean はすべて非活性化 (passivate) されます。
- **max-bean-age:** overager プロセスにより非活性化が行われるまで bean がどの程度アクティブでない状態でいられるか上限を指定します。
- **resizer-period:** resizer タスクを実行する間隔を秒単位で指定します。resizer タスクの目的は、以下の方法で、残り3つの要素値をもとにキャッシュの容量を縮小／拡大します。resizer タスクが実行されると、キャッシュミスの現在の間隔をチェックし、この間隔が `min-cache-`

**miss-period** の値より小さい場合は、**cache-load-factor** を使い、キャッシュは最大で **max-capacity** の値まで拡大されます。キャッシュミスの間隔が **max-cache-miss-period** 値より小さい場合は、このキャッシュは **cache-load-factor** を使い縮小されます。

- **max-cache-miss-period**: キャッシュミスがキャッシュの容量が縮小されるとシグナル送信すべき間隔を指定します。キャッシュが縮小されるまで許容最小の誤り指数と同じになります。
- **min-cache-miss-period**: キャッシュミスがキャッシュの容量が拡大されるとシグナル送信すべき間隔を指定します。キャッシュが拡大されるまで許容される最大の誤り指数と同じになります。
- **cache-load-factor**: キャッシュ容量を縮小／拡大する因数を指定します。因数は 1 未満にしてください。キャッシュが縮小される場合、容量が削減されるので bean がキャッシュする容量の現在の割合は **cache-load-factor** 値と同等になります。キャッシュが拡大される場合、新しい容量は **current-capacity \* 1/cache-load-factor** として確定されます。実際の拡大因数はキャッシュミス数に基づく内部アルゴリズムに応じて 2 まで上げることができます。キャッシュミス率が高いほど実際の拡大因数は 2 に近くなります。

**LRUStatefulContextCachePolicy** は残りの子要素にも対応しています。

- **remover-period**: remover タスクを実行する間隔を秒単位で指定します。remover タスクは **max-bean-life** 秒を超える間隔でアクセスされていない非活性化された bean を削除します。このタスクはユーザーにより削除されなかったステートフルセッション bean で非活性化ストアがいっぱいになってしまわないようにします。
- **max-bean-life**: bean がアクティブでない状態で存在できる最大期間を秒単位で指定します。この期間を超えると、bean は非活性化ストアから削除されます。

別のキャッシュポリシー実装は **org.jboss.ejb.plugins.NoPassivationCachePolicy** クラスで、これは単にインスタンスの非活性化を行いません。明示的に削除しない限りインスタンスを破棄することのないインメモリの **HashMap** 実装を使用します。このクラスはいずれの **cache-policy-conf** 設定要素もサポートしません。

### 29.3.1.3.12. persistence-manager 要素

**persistence-manager** 要素の値は永続管理実装の完全修飾クラス名を指定します。実装タイプは EJB のタイプに依存します。ステートフルセッション bean の場合は **org.jboss.ejb.StatefulSessionPersistenceManager** インターフェースの実装、そして BMP エンティティ bean の場合は、**org.jboss.ejb.EntityPersistenceManager** インターフェースの実装でなければならず、CMP エンティティ bean の場合は **org.jboss.ejb.EntityPersistenceStore** インターフェースの実装でなければなりません。

### 29.3.1.3.13. web-class-loader 要素

**web-class-loader** 要素は、**WebService** MBean と併用する **org.jboss.web.WebClassLoader** のサブクラスを指定し、デプロイされた ear、EJB JAR および WAR から動的にリソースとクラスをローディングができるようにします。**WebClassLoader** は **Container** に関連づけられ、その親として **org.jboss.mx.loading.UnifiedClassLoader** を持つ必要があります。**getURLs()** メソッドをオーバーライドし、ローカルのローディングに使用するものとは別のリモートローディング用の URL セットを返します。

**WebClassLoader** には、サブクラスによりオーバーライドされるはずのメソッドが 2 つあります。そのメソッドは、**getKey()** と **getBytes()** です。2 番目のメソッドはこの実装では空命令で、iiop モジュールで使用されるクラスローダーなどのように、バイトコード生成機能がついたサブクラスにより上書きされるはずですが。

**WebClassLoader** サブクラスは **WebClassLoader(ObjectName containerName, UnifiedClassLoader parent)** コンストラクターと同じ署名を持つコンストラクターを持っていないければなりません。

#### 29.3.1.3.14. locking-policy 要素

**locking-policy** 要素は、利用する EJB ロック実装の完全修飾クラス名を渡します。このクラスは **org.jboss.ejb.BeanLock** インターフェースを実装しなければなりません。現在の JBoss バージョンには次が含まれます。

- **org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock**: この実装は、公平な FIFO キューでトランザクションロックが開放されるのを待機するスレッドを格納します。トランザクション以外のスレッドもこの待機キューに置かれます。このクラスはキューから次の待機トランザクションをポップしてそのトランザクションに関連づけられ待機中のスレッドにのみ通知します。**QueuedPessimisticEJBLock** は標準設定で使用される現在のデフォルトになります。
- **org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLockNoADE**: デッドロック検出が無効になっている点以外は **QueuedPessimisticEJBLock** と同じ動作となります。
- **org.jboss.ejb.plugins.lock.SimpleReadWriteEJBLock**: このロックにより同時に複数の読み取りロックができるようになります。ライターがこのロックを要求すると、それ以降、トランザクションに読み取りロックをまだ持たない読み取りロックのリクエストは、ライターがすべて終了するまでブロックされます。その後、待機中のリーダーはすべて (再入可能設定 / `methodLock` に従い) 同時に開始します。プロモートするリーダーは、他の待機中のライターよりも先に、書き込みロックをまず試行します。プロモート中のリーダーが存在する場合、矛盾読み取り例外がスローされます。当然、ライターは書き込みロックを得る前にすべての読み取りロックが解放されるのを待たなければなりません。
- **org.jboss.ejb.plugins.lock.NoLock**: トランザクションコンテナの設定ごとにインスタンスとともに使用されるアンチロッキングポリシーです。

ロック機能とデッドロック検出については「[エンティティ bean のロックとデッドロックの検出](#)」で詳細に説明します。

#### 29.3.1.3.15. commit-option と optiond-refresh-rate 要素

**commit-option** 値は EJB エンティティ bean の永続ストレージコミットオプションを指定します。**A**、**B**、**C**、**D** のいずれかにならなければなりません。

- **A**: コンテナがトランザクション間の bean のステータスをキャッシュします。このオプションはコンテナが永続ストアにアクセスしている唯一のユーザーであると仮定しています。この仮定により、必要な場合に限ってのみコンテナが永続ストレージからのインメモリステータスを同期できるようになります。見つかった bean で最初のビジネスメソッドが実行される前か、bean が非活性化され別のビジネスメソッドにサービスを提供するため再度アクティブにされた後に、同期が行われます。この動作はビジネスメソッドがトランザクションコンテキスト内で実行されるかどうかには依存しません。
- **B**: このコンテナはトランザクション間の bean のステータスをキャッシュします。ただし、オプション **A** とは異なり、このコンテナは永続ストアへ独占的にアクセスするとの仮定は行いません。したがって、コンテナは各トランザクション最初にインメモリのステータスを同期します。このため、トランザクションコンテキストの外側で実行しているビジネスメソッド (トランザクション属性が `Never`、`NotSupported` または `Supports`) はキャッシュされた (また無効である可能性がある) bean のステータスにアクセスしますが、トランザクションコンテキスト内で実行しているビジネスメソッドにとってコンテナが bean をキャッシュしてもあまり役には立ちません。



- **C:** コンテナは bean インスタンスをキャッシュしません。インメモリのステータスは各トランザクションの起動時に同期されなければなりません。トランザクションの外側で実行しているビジネスメソッドの場合、同期はいまだ実行されますが **ejbLoad** が呼び出し側と同じトランザクションコンテキスト内で実行します。
- **D:** EJB 仕様には記載されていない JBoss 固有のコミットオプションです。オプション **A** のようにトランザクション間で bean のステータスがキャッシュされる遅延読み取りスキームですが、ステータスが定期的に永続ストアのステータスと再同期されます。再ロードの間隔はデフォルトで 30 秒ですが、**optiond-refresh-rate** 要素を使って設定することができます。

### 29.3.1.3.16. security-domain 要素

**security-domain** 要素は **org.jboss.security.AuthenticationManager** と **org.jboss.security.RealmMapping** のインターフェースを実装するオブジェクトの JNDI 名を指定します。特定のデプロイメントの EJB すべてが同じ方法でセキュリティを保てるように、**jboss root** 要素の下にある **security-domain** を指定する方がより一般的となります。しかし、各 bean 設定に対してセキュリティドメインを設定することもできます。

### 29.3.1.3.17. cluster-config

**cluster-config** 要素によりコンテナ設定を使用するすべての EJB に対してクラスター固有の設定を指定できるようになります。クラスター設定の指定はコンテナ設定レベルか個別の EJB デプロイメントレベルで行うことができます。

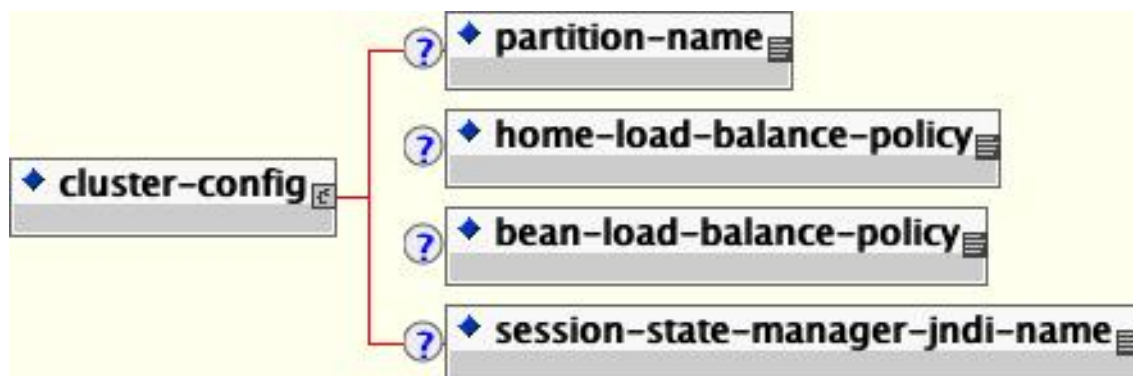


図29.7 cluster-config と関連要素

- **partition-name:** **partition-name** 要素は、クラスタリング情報の交換にコンテナが使用する **org.jboss.ha.framework.interfaces.HAPartition** インターフェースを検索する場所を指示します。これは **HAPartition** がバインドされる場所の完全 JNDI 名ではなく、目的のクラスターを管理している **ClusterPartitionMBean** サービスの **PartitionName** 属性に該当するはずですが。実際の **HAPartition** バインディングの JNDI 名は **partition-name** 値に **/HASessionState/** を追加した形式となります。デフォルト値は **DefaultPartition** です。
- **home-load-balance-policy:** **home-load-balance-policy** 要素はホームプロキシで出された呼び出しの負荷分散に使用する Java クラス名を指示します。このクラスは **org.jboss.ha.framework.interface.LoadBalancePolicy** インターフェースを実装しなければなりません。デフォルトポリシーは **org.jboss.ha.framework.interfaces.RoundRobin** になります。
- **bean-load-balance-policy:** **bean-load-balance-policy** 要素は bean プロキシで呼び出しの負荷分散に使用する java クラス名を指示します。このクラスは **org.jboss.ha.framework.interface.LoadBalancePolicy** インターフェースを実装し

なければなりません。エンティティ bean およびステートフルセッション bean の場合デフォルトは `org.jboss.ha.framework.interfaces.FirstAvailable` で、ステートレスセッション bean の場合は `org.jboss.ha.framework.interfaces.RoundRobin` になります。

- **session-state-manager-jndi-name: session-state-manager-jndi-name** 要素はクラスター内のステータスセッション管理のバックエンドとしてコンテナが使用する `org.jboss.ha.framework.interfaces.HASessionState` の名前を指示します。  
`partition-name` 要素とは異なり、`HASessionState` 実装がバインドされる場所の JNDI 名になります。デフォルトの場所は `/HASessionState/Default` です。

#### 29.3.1.3.18. depends 要素

**depends** 要素はコンテナまたは EJB が依存するサービスの JMX **ObjectName** を渡します。その他サービスに対する明示的な依存性を指定すると、必要となるサービス起動後のデプロイメントの順序に依存せずに済みます。

### 29.3.2. コンテナプラグインのフレームワーク

JBoss EJB コンテナはコンテナ動作の実装をさまざまな側面から変更できるようにするフレームワークパターンを使用しています。コンテナ自体は各種の動作コンポーネント同士を接続する以外は特に重要な動作は行いません。コンテナ設定を変更することで新しい実装をプラグインできるため、動作コンポーネントの実装はプラグインとして参照されます。一般的に変更される可能性のあるプラグイン動作の例としては、永続管理、オブジェクトのプール、オブジェクトのキャッシュ、コンテナ呼び出し、インターセプターなどがあります。`org.jboss.ejb.Container` クラスには 4 つのサブクラスがあり、それぞれ特定の bean タイプを実装しています。

- **org.jboss.ejb.EntityContainer**: `javax.ejb.EntityBean` タイプを処理します。
- **org.jboss.ejb.StatelessSessionContainer**: ステートレス `javax.ejb.SessionBean` タイプを処理します。
- **org.jboss.ejb.StatefulSessionContainer**: ステートフル `javax.ejb.SessionBean` タイプを処理します。
- **org.jboss.ejb.MessageDrivenContainer**: `javax.ejb.MessageDrivenBean` タイプを処理します。

EJB コンテナは、コンテナの動作の多くをコンテナプラグインとして知られるコンポーネントに委譲します。コンテナプラグインポイントを構成するインターフェースは以下のとおりです。

- `org.jboss.ejb.ContainerPlugin`
- `org.jboss.ejb.ContainerInvoker`
- `org.jboss.ejb.Interceptor`
- `org.jboss.ejb.InstancePool`
- `org.jboss.ejb.InstanceCache`
- `org.jboss.ejb.EntityPersistenceManager`
- `org.jboss.ejb.EntityPersistenceStore`
- `org.jboss.ejb.StatefulSessionPersistenceManager`



コンテナの主な役割は、コンテナのプラグイン群を管理することです。つまり、プラグインがその機能を実装するために必要とするすべての情報を必ず得られるようにします。

### 29.3.2.1. org.jboss.ejb.ContainerPlugin

**ContainerPlugin** インターフェースは全コンテナプラグインインターフェースの親インターフェースになります。これによりコールバックができるようになり、コンテナが各プラグインに対して、プラグインが代わりに作業を行っているコンテナへのポインターを提供できるようになります。以下に **ContainerPlugin** インターフェースを示します。

#### 例29.4 org.jboss.ejb.ContainerPlugin インターフェース

```
public interface ContainerPlugin
    extends Service, AllowedOperationsFlags
{
    /** co
     * This callback is set by the container so that the plugin
     * may access its container
     *
     * @param con the container which owns the plugin
     */
    public void setContainer(Container con);
}
```

### 29.3.2.2. org.jboss.ejb.Interceptor

**Interceptor** インターフェースにより各 EJB メソッド呼び出しが通過しなければならないメソッドインターセプターのチェーンを構築できるようになります。**Interceptor** インタフェースは以下に示しています。

#### 例29.5 org.jboss.ejb.Interceptor インターフェース

```
import org.jboss.invocation.Invocation;

public interface Interceptor
    extends ContainerPlugin
{
    public void setNext(Interceptor interceptor);
    public Interceptor getNext();
    public Object invokeHome(Invocation mi) throws Exception;
    public Object invoke(Invocation mi) throws Exception;
}
```

コンテナ設定で定義されるインターセプターはすべて **EJBDeployer** によって作成されコンテナインターセプターチェーンに追加されます。最後のインターセプターは EJB bean 実装とやりとりを行うため、デプロイヤーではなく、コンテナ自体によって追加されます。

チェーン内のインターセプターの順序が重要になります。特定の **EnterpriseContext** インスタンスに結びつけられていないインターセプターをキャッシュやプールと相互作用するインターセプターより前に配置するために、この順序が存在します。

**Interceptor** インターフェースの実装はリンクされた一覧のような構造をとり、**Invocation** オブジェクトはこの構造を通過します。invoker が **Invocation** を JMX バス経由でコンテナに渡すと、チェーン内の 1 番目のインターセプターが呼び出されます。最後のインターセプターは bean 上でビジネスメソッドを呼び出します。通常、bean タイプやコンテナ設定によりチェーン内には約 5 つのインターセプターがあります。**Interceptor** セマンティックは単純なものから複雑なものまで様々です。シンプルなインターセプターの例としては **LoggingInterceptor** があげられます。また、複雑な例は **EntitySynchronizationInterceptor** になるでしょう。

インターセプターパターンの主な利点のひとつがインターセプター配列の柔軟性にあります。もうひとつの利点はインターセプター間の機能的な相違が明確であることです。たとえば、トランザクションとセキュリティの論理はそれぞれ **TXInterceptor** と **SecurityInterceptor** 間で明確に区別されます。

インターセプターのいずれかが失敗した場合、その呼び出しはその時点で終了されます。これがセマンティックの迅速なフェールタイプになります。たとえば、セキュリティー設定された EJB に適切なパーミッションなしにアクセスした場合、その呼び出しはトランザクションが開始されるか、あるいはインスタンスキャッシュが更新される前に **SecurityInterceptor** として失敗します。

### 29.3.2.3. org.jboss.ejb.InstancePool

**InstancePool** を使用して、いずれの ID にも関連付けられていない EJB インスタンスを管理します。プールは実際には関連付けされていない bean インスタンス群および関連データを集約する **org.jboss.ejb.EnterpriseContext** オブジェクトのサブクラスを管理します。

#### 例29.6 org.jboss.ejb.InstancePool インターフェース

```
public interface InstancePool
    extends ContainerPlugin
{
    /**
     * Get an instance without identity. Can be used
     * by finders and create-methods, or stateless beans
     *
     * @return Context /w instance
     * @exception RemoteException
     */
    public EnterpriseContext get() throws Exception;

    /** Return an anonymous instance after invocation.
     *
     * @param ctx
     */
    public void free(EnterpriseContext ctx);

    /**
     * Discard an anonymous instance after invocation.
     * This is called if the instance should not be reused,
     * perhaps due to some exception being thrown from it.
     *
     * @param ctx
     */
    public void discard(EnterpriseContext ctx);

    /**
     * Return the size of the pool.
     */
}
```

```

    *
    * @return the size of the pool.
    */
    public int getCurrentSize();

    /**
    * Get the maximum size of the pool.
    *
    * @return the size of the pool.
    */
    public int getMaxSize();
}

```

その構成によって、コンテナは特定サイズのプールに再利用したインスタンスを含むよう選択するか、必要に応じてインスタンスのインスタンス化および初期化をするよう選択することもできます。

**InstanceCache** 実装はプールを使用して有効化を行うための空きインスタンスの獲得を行います。また、プールを利用してインターセプターによって Home インターフェースメソッドに使用するためのインスタンス獲得します (create および finder の呼び出し)。

#### 29.3.2.4. org.jboss.ejb.InstanceCache

コンテナ **InstanceCache** 実装はアクティブな状態にある EJB インスタンスをすべて処理します。つまり ID が付けられている bean インスタンスということになります。エンティティおよびステートフルセッションの bean のみがメソッド呼び出し間のステータスを持つ bean タイプであるため、これらの bean のみがキャッシュされます。エンティティ bean のキャッシュキーは bean プライマリキーになります。ステートフルセッション bean のキャッシュキーはセッション ID になります。

##### 例29.7 org.jboss.ejb.InstanceCache インターフェース

```

public interface InstanceCache
    extends ContainerPlugin
{
    /**
    * Gets a bean instance from this cache given the identity.
    * This method may involve activation if the instance is not
    * in the cache.
    * Implementation should have O(1) complexity.
    * This method is never called for stateless session beans.
    *
    * @param id the primary key of the bean
    * @return the EnterpriseContext related to the given id
    * @exception RemoteException in case of illegal calls
    * (concurrent / reentrant), NoSuchObjectException if
    * the bean cannot be found.
    * @see #release
    */
    public EnterpriseContext get(Object id)
        throws RemoteException, NoSuchObjectException;

    /**
    * Inserts an active bean instance after creation or activation.
    * Implementation should guarantee proper locking and O(1)
    complexity.
    */
}

```

```

    *
    * @param ctx the EnterpriseContext to insert in the cache
    * @see #remove
    */
    public void insert(EnterpriseContext ctx);

    /**
     * Releases the given bean instance from this cache.
     * This method may passivate the bean to get it out of the cache.
     * Implementation should return almost immediately leaving the
     * passivation to be executed by another thread.
     *
     * @param ctx the EnterpriseContext to release
     * @see #get
     */
    public void release(EnterpriseContext ctx);

    /**
     * Removes a bean instance from this cache given the identity.
     * Implementation should have O(1) complexity and guarantee
     * proper locking.
     *
     * @param id the primary key of the bean
     * @see #insert
     */
    public void remove(Object id);

    /**
     * Checks whether an instance corresponding to a particular
     * id is active
     *
     * @param id the primary key of the bean
     * @see #insert
     */
    public boolean isActive(Object id);
}

```

アクティブなインスタンスの一覧管理の他、**InstanceCache** はインスタンスの有効化および非活性化も行います。特定の ID を持つインスタンスが要求され、このインスタンスが現在アクティブでない場合、**InstanceCache** は **InstancePool** を使用して空きインスタンスを取得し、その後で永続マネージャーがインスタンスを有効にします。同様に、**InstanceCache** がアクティブなインスタンスの非活性化を行うと決定した場合、永続マネージャーを呼び出しインスタンスの非活性化を行い、そのインスタンスを **InstancePool** に解放する必要があります。

#### 29.3.2.5. org.jboss.ejb.EntityPersistenceManager

**EntityPersistenceManager** は **EntityBeans** の永続化を行います。これには次が含まれます。

- ストレージ内で EJB インスタンスを作成
- 特定のプライマリキーの状態を EJB インスタンスにロード
- 特定の EJB インスタンスの状態を格納

- ストレージから EJB インスタンスを削除
- EJB インスタンスの状態を有効化
- EJB インスタンスの状態を非活性化

#### 例29.8 org.jboss.ejb.EntityPersistenceManager インターフェース

```
public interface EntityPersistenceManager
    extends ContainerPlugin
{
    /**
     * Returns a new instance of the bean class or a subclass of the
     * bean class.
     *
     * @return the new instance
     */
    Object createBeanClassInstance() throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbCreate
    method
     * on the instance and to handle the results properly wrt the
    persistent
     * store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     */
    void createEntity(Method m,
        Object[] args,
        EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbPostCreate
    method
     * on the instance and to handle the results properly wrt the
    persistent
     * store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     */
    void postCreateEntity(Method m,
        Object[] args,
        EntityEnterpriseContext instance)
        throws Exception;

    /**
```

```

* This method is called when single entities are to be found. The
* persistence manager must find out whether the wanted instance is
* available in the persistence store, and if so it shall use the
* ContainerInvoker plugin to create an EJBObject to the instance,
which
* is to be returned as result.
*
* @param finderMethod the find method in the home interface that
was
* called
* @param args any finder parameters
* @param instance the instance to use for the finder call
* @return an EJBObject representing the found entity
*/
Object findEntity(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
throws Exception;

/**
* This method is called when collections of entities are to be
* found. The persistence manager must find out whether the wanted
* instances are available in the persistence store, and if so it
* shall use the ContainerInvoker plugin to create EJBObjects to
* the instances, which are to be returned as result.
*
* @param finderMethod the find method in the home interface that
was
* called
* @param args any finder parameters
* @param instance the instance to use for the finder call
* @return an EJBObject collection representing the found
* entities
*/
Collection findEntities(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
    throws Exception;

/**
* This method is called when an entity shall be activated. The
* persistence manager must call the ejbActivate method on the
* instance.
*
* @param instance the instance to use for the activation
*
* @throws RemoteException thrown if some system exception occurs
*/
void activateEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
* This method is called whenever an entity shall be load from the
* underlying storage. The persistence manager must load the state
* from the underlying storage and then call ejbLoad on the
* supplied instance.

```

```

*
* @param instance the instance to synchronize
*
* @throws RemoteException thrown if some system exception occurs
*/
void loadEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
 * This method is used to determine if an entity should be stored.
 *
 * @param instance the instance to check
 * @return true, if the entity has been modified
 * @throws Exception thrown if some system exception occurs
 */
boolean isModified(EntityEnterpriseContext instance) throws
Exception;

/**
 * This method is called whenever an entity shall be stored to the
 * underlying storage. The persistence manager must call ejbStore
 * on the supplied instance and then store the state to the
 * underlying storage.
 *
 * @param instance the instance to synchronize
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void storeEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
 * This method is called when an entity shall be passivate. The
 * persistence manager must call the ejbPassivate method on the
 * instance.
 *
 * @param instance the instance to passivate
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void passivateEntity(EntityEnterpriseContext instance)
throws RemoteException;

/**
 * This method is called when an entity shall be removed from the
 * underlying storage. The persistence manager must call ejbRemove
 * on the instance and then remove its state from the underlying
 * storage.
 *
 * @param instance the instance to remove
 *
 * @throws RemoteException thrown if some system exception occurs
 * @throws RemoveException thrown if the instance could not be
removed
 */

```

```

    void removeEntity(EntityEnterpriseContext instance)
    throws RemoteException, RemoveException;
}

```

### 29.3.2.6. org.jboss.ejb.EntityPersistenceStore インターフェース

EJB 2.1 仕様に従い、JBoss はコンテナ管理の永続性 (CMP: Container Managed persistence) と bean 管理永続性 (BMP: Bean Managed Persistence) の 2 つの エンティティ bean 永続セマンティックをサポートしています。CMP 実装は **org.jboss.ejb.EntityPersistenceStore** インターフェースの実装を使用します。デフォルトではこれが

**org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager** で CMP2 永続エンジンのエントリポイントになります。 **EntityPersistenceStore** インターフェースを以下に示します。

#### 例29.9 org.jboss.ejb.EntityPersistenceStore インターフェース

```

public interface EntityPersistenceStore
    extends ContainerPlugin
{
    /**
     * Returns a new instance of the bean class or a subclass of the
     * bean class.
     *
     * @return the new instance
     *
     * @throws Exception
     */
    Object createBeanClassInstance()
        throws Exception;

    /**
     * Initializes the instance context.
     *
     * <p>This method is called before createEntity, and should
     * reset the value of all cmpFields to 0 or null.
     *
     * @param ctx
     *
     * @throws RemoteException
     */
    void initEntity(EntityEnterpriseContext ctx);

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for handling the results
     * properly wrt the persistent store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     * @return The primary key computed by CMP PM or null for BMP
     *
     * @throws Exception
     */
}

```



```

    Object createEntity(Method m,
Object[] args,
EntityEnterpriseContext instance)
    throws Exception;

/**
 * This method is called when single entities are to be found. The
 * persistence manager must find out whether the wanted instance
 * is available in the persistence store, if so it returns the
 * primary key of the object.
 *
 * @param finderMethod the find method in the home interface that
was
 * called
 * @param args any finder parameters
 * @param instance the instance to use for the finder call
 * @return a primary key representing the found entity
 *
 * @throws RemoteException thrown if some system exception occurs
 * @throws FinderException thrown if some heuristic problem occurs
 */
Object findEntity(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
    throws Exception;

/**
 * This method is called when collections of entities are to be
 * found. The persistence manager must find out whether the wanted
 * instances are available in the persistence store, and if so it
 * must return a collection of primaryKeys.
 *
 * @param finderMethod the find method in the home interface that
was
 * called
 * @param args any finder parameters
 * @param instance the instance to use for the finder call
 * @return an primary key collection representing the found
 * entities
 *
 * @throws RemoteException thrown if some system exception occurs
 * @throws FinderException thrown if some heuristic problem occurs
 */
Collection findEntities(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
    throws Exception;

/**
 * This method is called when an entity shall be activated.
 *
 * <p>With the PersistenceManager factorization most EJB
 * calls should not exists However this calls permits us to
 * introduce optimizations in the persistence store. Particularly
 * the context has a "PersistenceContext" that a PersistenceStore
 * can use (JAWS does for smart updates) and this is as good a

```

```
* callback as any other to set it up.
* @param instance the instance to use for the activation
*
* @throws RemoteException thrown if some system exception occurs
*/
void activateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is called whenever an entity shall be load from the
 * underlying storage. The persistence manager must load the state
 * from the underlying storage and then call ejbLoad on the
 * supplied instance.
 *
 * @param instance the instance to synchronize
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void loadEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is used to determine if an entity should be stored.
 *
 * @param instance the instance to check
 * @return true, if the entity has been modified
 * @throws Exception thrown if some system exception occurs
 */
boolean isModified(EntityEnterpriseContext instance)
    throws Exception;

/**
 * This method is called whenever an entity shall be stored to the
 * underlying storage. The persistence manager must call ejbStore
 * on the supplied instance and then store the state to the
 * underlying storage.
 *
 * @param instance the instance to synchronize
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void storeEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is called when an entity shall be passivate. The
 * persistence manager must call the ejbPassivate method on the
 * instance.
 *
 * <p>See the activate discussion for the reason for
 * exposing EJB callback * calls to the store.
 *
 * @param instance the instance to passivate
 *
 * @throws RemoteException thrown if some system exception occurs
 */
```

```

void passivateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is called when an entity shall be removed from the
 * underlying storage. The persistence manager must call ejbRemove
 * on the instance and then remove its state from the underlying
 * storage.
 *
 * @param instance the instance to remove
 *
 * @throws RemoteException thrown if some system exception occurs
 * @throws RemoveException thrown if the instance could not be
removed
 */
void removeEntity(EntityEnterpriseContext instance)
    throws RemoteException, RemoveException;
}

```

**EntityPersistenceManager** インターフェースの BMP デフォルト実装は、**org.jboss.ejb.plugins.BMPPersistenceManager** です。BMP 永続マネージャーはすべての永続ロジックがエンティティ bean 自体の中にあるためかなりシンプルになります。永続マネージャーの唯一の役割はコンテナのコールバックを行うことです。

### 29.3.2.7. org.jboss.ejb.StatefulSessionPersistenceManager

**StatefulSessionPersistenceManager** はステートフル **SessionBeans** の永続性を処理します。これには次のようなものが含まれます。

- ストレージ内でステートフルセッションを作成
- ストレージからステートフルセッションを有効化
- ステートフルセッションをストレージへ非活性化
- ストレージからステートフルセッションを削除

**StatefulSessionPersistenceManager** インターフェースを以下に示しています。

#### 例29.10 org.jboss.ejb.StatefulSessionPersistenceManager インターフェース

```

public interface StatefulSessionPersistenceManager
    extends ContainerPlugin
{
    public void createSession(Method m, Object[] args,
        StatefulSessionEnterpriseContext ctx)
        throws Exception;

    public void activateSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException;

    public void passivateSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException;
}

```

```
public void removeSession(StatefulSessionEnterpriseContext ctx)
    throws RemoteException, RemoveException;

public void removePassivated(Object key);
}
```

デフォルトの **StatefulSessionPersistenceManager** インターフェース実装は **org.jboss.ejb.plugins.StatefulSessionFilePersistenceManager** になります。その名前の通り、**StatefulSessionFilePersistenceManager** はファイルシステムを使い、ステートフルセッション bean の永続化を行います。特に永続マネージャーは **.ser** 拡張子を持つセッション ID と bean 名から成る名前を持つフラットファイル内で bean をシリアル化します。永続マネージャーは有効化をするときに bean の状態を復元し、非活性化を行うときに bean の **.ser** ファイルからその状態をそれぞれ格納します。

## 29.4. エンティティ BEAN のロックとデッドロックの検出

本項では、エンティティ bean ロックとは何か、JBoss ではどのようにしてエンティティ bean へのアクセスが行われ、ロックされるのかについて説明します。また、システムでエンティティ bean を使用する際に遭遇する可能性がある問題とその対処方法の他、デッドロックについては正式に定義して検討を行ってみます。最後に、エンティティ bean のロックという観点からシステムの微調整について見ていきます。

### 29.4.1. JBoss にロック機能が必要な理由

ロック機能とは、データの整合性を保護するためのものです。時に、極めて重要なデータの更新を行うことができるユーザーは一度に確実に一人のみにする必要があります。また、読み取りや書き込みが同時に起こることでデータが破損しないよう、システム内の機密オブジェクトに対するアクセスがシリアル化される必要がある場合もあります。従来はデータベースがこうした機能をトランザクションスコープや表と行のロック機能で提供しています。

エンティティ bean はオブジェクト指向のインターフェースをリレーショナルデータに提供するのに最適の方法となります。これに加え、更新が確実に必要となるまでキューッシュして遅延させることでデータベースの負荷を低減しパフォーマンスを改善することができるため、データベースの効率性を最大限に引き出すことが可能になります。ただし、キャッシングではデータの整合性が問題となるため、従来のデータベースで使用していたようなトランザクション分離プロパティを提供するにはなんらかのアプリケーションサーバーレベルのロック機能が必要となります。

### 29.4.2. エンティティ Bean のライフサイクル

JBoss デフォルトの設定では、特定のエンティティ bean のインスタンスの中で、メモリ内で有効なインスタンスは1つのみです。これはすべてのキャッシュ設定およびすべての **commit-option** タイプに適用されます。このインスタンスのライフサイクルはそれぞれの commit-option とは異なります。

- **コミットオプション A** の場合、このインスタンスはキャッシュされトランザクション間で使用されます。
- **コミットオプション B** の場合は、このインスタンスはキャッシュされトランザクション間で使用されますが、トランザクションの終わりでダーティと印がつけられます。つまり、新しいトランザクションの開始時に **ejbLoad** を呼び出す必要がでてきます。
- **コミットオプション C** の場合は、このインスタンスはダーティの印が付けられてキャッシュから解放され、さらにトランザクションの終わりに非活性化用にマークが付けられます。

- コミットオプション **D** なら、バックグラウンドの更新スレッドが定期的に **ejbLoad** をキャッシュ内の古い bean で呼び出します。これ以外、このオプションは **A** と同じように動作します。

bean に非活性化のマークが付けられると、bean は非活性化キューに配置されます。各エンティティ bean コンテナには定期的に非活性化キューに配置されている bean を非活性化する非活性化スレッドがあります。同じプライマリキーの bean へのアクセスをアプリケーションが要求すると、bean は非活性化キューから引き出され再利用されます。

例外またはトランザクションロールバックでは、エンティティ bean インスタンスは完全にキャッシュから破棄されます。非活性化キューに配置されることやインスタンスプールによって再利用されることもありません。非活性化キューをのぞき、エンティティ bean インスタンスのプーリングはありません。

### 29.4.3. デフォルトのロック動作

エンティティ bean ロック機能はエンティティ bean インスタンスから完全に分離されます。ロック機能の論理は別のロックオブジェクト内で完全に隔離され、管理されます。一度に許可されるアクティブな特定エンティティ bean のインスタンスはひとつだけのため、JBoss はデータの整合性確保および EJB 仕様に対する準拠の目的で 2 種のロックを採用しています。

- **Method Lock**: メソッドロックによって特定の Entity Bean 上で呼び出し可能なのは必ず一度に 1 つのスレッド実行のみであるようにします。これは EJB 仕様では必須となっています。
- **Transaction Lock**: トランザクションロックを使うと、特定の Entity Bean へのアクセスできるのは必ず一度に 1 トランザクションのみにします。これによりアプリケーションサーバーレベルでのトランザクションの ACID プロパティを確保します。デフォルトでは特定の Entity Bean のアクティブなインスタンスは一度に 1 つとなるため、JBoss はこのインスタンスをダーティな読み込みおよび書き込みから保護しなければなりません。このため、デフォルトのエンティティ bean ロック機能の動作はそれが完了するまで 1 トランザクション内のエンティティ bean をロックします。これはトランザクション内のエンティティ bean でなんらかのメソッドが呼び出されると、保持しているトランザクションがコミットを行うまたはロールバックされるまでは他のトランザクションはこの bean へのアクセスできないということになります。

### 29.4.4. プラグ可能なインターセプターとロックポリシー

基本的なエンティティ bean のライフサイクルおよび動作は **standardjboss.xml** 記述子で定義されるコンテナ設定により定義されることがわかりました。今度は **Standard CMP 2.x EntityBean** 設定の **container-interceptors** 定義を見てみることにします。

```
<container-interceptors>

<interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</interceptor>
  <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
  <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
  <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>

<interceptor>org.jboss.ejb.plugins.CallValidationInterceptor</interceptor>
  <interceptor
metricsEnabled="true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
  <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
```

```

<interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor</interceptor>

<interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</interceptor>

<interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</interceptor>
</container-interceptors>

```

上記に示すインターセプターはエンティティ bean のほとんどの動作を定義します。以下に本項に関連のあるインターセプターについて記します。

- **EntityLockInterceptor:** このインターセプターの役割は呼び出しが実行される前に取得されなければならないロックをすべてスケジュールすることです。このインターセプターは非常に軽量ですべてのロック機能動作をプラグ可能なロックポリシーに委譲します。
- **EntityInstanceInterceptor:** このインターセプターの役割はキャッシュ内でエンティティ bean を検索する、または新しいエンティティ bean を作成することです。また、このインターセプターは一度にメモリ内に存在する bean のアクティブなインスタンスが必ず 1 つのみであるようにします。
- **EntitySynchronizationInterceptor:** このインターセプターの役割は基盤となるストレージでキャッシュの状態を同期することです。EJB 仕様の **ejbLoad** および **ejbStore** のセマンティックでこれを行います。トランザクションが存在すると、トランザクションの分離によって引き起こされます。コールバックを基盤となるトランザクションモニターに JTA インターフェース経由で登録します。トランザクションがない場合、このポリシーは呼び出しから戻ると状態を格納します。その仕様の同期ポリシー **A**、**B**、**C** の他 JBoss 固有のコミットオプション **D** もここで処理されます。

#### 29.4.5. デッドロック

本項ではデッドロックの問題を発見して解決する方法について説明しています。何が Mbean をデッドロックしているか、使用しているアプリケーション内でデッドロックをどのようにして検出するか、そしてデッドロックをどのようにしたら解決できるのかなどについて説明していきます。デッドロックは複数のスレッドが共有リソース上でロックしている場合に発生する可能性があります。[図29.8「デッドロックの定義例」](#)では、シンプルなデッドロックの発生例を示します。以下では、**Thread 1** は **Bean A** のロック、**Thread 2** は **Bean B** のロックを持っています。しばらくすると、**Thread 1** は **Thread 2** が **Bean B** を持っているためそれをロックしてブロックしようとします。同様に **Thread 2** は **Thread 1** が **A** のロックを持っているためそれをロックしてブロックしようとします。この時点で、両スレッドはデッドロックとなり、他のスレッドですでにロックされているリソースへのアクセスを待機することになります。

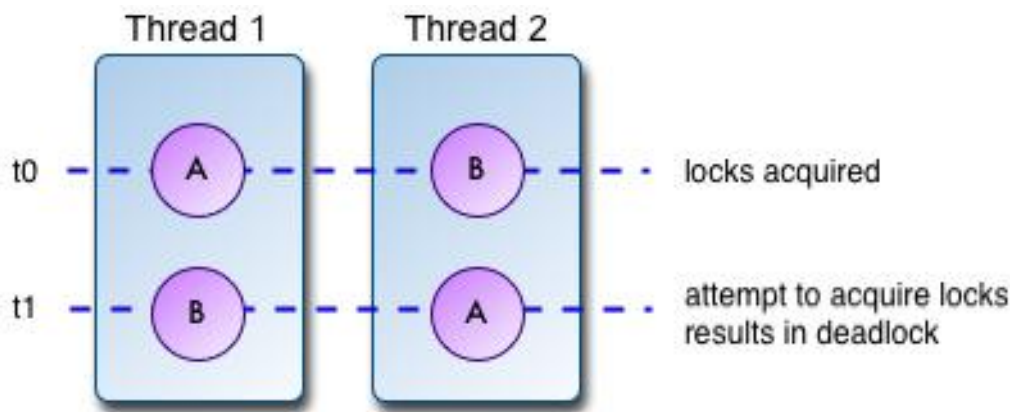


図29.8 デッドロックの定義例

JBoss のデフォルトとなるロックポリシーは、トランザクションが完了するまでそのトランザクションのコンテキスト内に呼び出しが発生する場合に Entity bean をロックすることです。このため、多くのエンティティ bean にアクセスする長期実行のトランザクションがある場合や、bean へのアクセス順序に注意を怠った場合にデッドロックは非常に簡単に発生します。デッドロックの問題を回避するためさまざまな技術や詳細設定を使用することができます。これらについては本項で後述します。

29.4.5.1. デッドロックの検出

幸い、JBoss はデッドロックの検出が可能です。JBoss には待機中のトランザクションや、ブロック中のトランザクションに関するグローバルな内部グラフがあります。エンティティ bean ロックを取得できないとスレッドが判断すると必ず、bean 上で現在ロックを保持しているトランザクションを割り出してそれ自体をブロック中のトランザクションのグラフに追加します。グラフの具体例を表29.1「ブロックされているトランザクションテーブルの例」に示しています。

表29.1 ブロックされているトランザクションテーブルの例

ブロックしている TX	ロックを保持する TX
Tx1	Tx2
Tx3	Tx4
Tx4	Tx1

スレッドが実際にブロックを行う前に、デッドロックの問題があるかどうかを検出しようとします。検出はブロックトランザクショングラフを検討することによって行われます。グラフを検討するたびにブロックされているトランザクションの記録を残していきます。グラフ内に複数回にわたりブロックされているノードを検出すると、デッドロックが発生していると認識して **ApplicationDeadlockException** を送出します。この例外によりトランザクションのロールバックが行われて、このロールバックによりトランザクションが保持しているすべてのロックが解放されることになります。

29.4.5.2. ApplicationDeadlockException のキャッチ

JBoss はアプリケーションのデッドロックを検出できるため、 **ApplicationDeadlockException** が原因で呼び出しが失敗する場合にアプリケーションがトランザクションを再試行できるように独自のアプリケーションを記述してください。残念ながら、この例外は **RemoteException** 内の深部に組み込まれるため、独自のキャッチブロックで検索する必要があります。例えば、

-

```

try {
    // ...
} catch (RemoteException ex) {
    Throwable cause = null;
    RemoteException rex = ex;
    while (rex.detail != null) {
        cause = rex.detail;
        if (cause instanceof ApplicationDeadlockException) {
            // ... We have deadlock, force a retry of the transaction.
            break;
        }
        if (cause instanceof RemoteException) {
            rex = (RemoteException)cause;
        }
    }
}
}

```

### 29.4.5.3. ロック情報の表示

**EntityLockMonitor** MBean サービスにより基本的なロックの統計数値を表示できる他、 トランザクションロック表の状態を出力することもできます。 このモニターを有効にするには、**conf/jboss-service.xml** 内でその設定のコメントを外します。

```

<mbean code="org.jboss.monitor.EntityLockMonitor"
       name="jboss.monitor:name=EntityLockMonitor"/>

```

**EntityLockMonitor** には設定可能属性はありません。 次のような読み取り専用属性があります。

- **MedianWaitTime:** ロックの取得にスレッドが待機しなかった全時間の中央値です。
- **AverageContentenders:** ロックを待機しなかった全スレッドの合計 対 競合合計数の比率になります。
- **TotalContentions:** トランザクションロックを取得するのに待機しなかったスレッドの合計数です。これは、スレッドが別のトランザクションに関連づけられているロックの取得を試行すると発生します。
- **MaxContentenders:** トランザクションロックの取得を待機していたスレッドの最大数です。

以下のような操作も可能です。

- **clearMonitor:** この操作はすべてのカウンターをゼロにしてロックモニターの状態をリセットします。
- **printLockMonitor:** この操作は bean の **ejbName**、 ロックの待機に要された合計時間、ロックが待たされた回数、ロック待機がタイムアウトとなったトランザクション数を一覧表示する全 EJB ロックの表を出力します。

### 29.4.6. 詳細設定と最適化

エンティティ bean のデフォルトのロック機能動作はデッドロックとなる可能性があります。エンティティ bean へアクセスするとその bean をトランザクション内にロックするため、アプリケーションのパフォーマンスあるいはスループットに関してかなり大きな問題が出てくる可能性があります。本セク



ションでは、パフォーマンスの最適化およびデッドロック発生率の軽減に利用できる各種設定や手法について見ていきます。

#### 29.4.6.1. 存続期間の短いトランザクション

トランザクションの存続期間をできるだけ短くして微調整を行うようにします。トランザクションの存続期間が短いほど、同時アクセスによる衝突やアプリケーションのスループット上昇の可能性が低くなります。

#### 29.4.6.2. アクセスの順序付け

エンティティ bean へのアクセスの順序付けを行うとデッドロック発生率の低減に役立ちます。つまり、システム内のエンティティ bean が必ず正確に同じ順序でアクセスされるようにするというのです。ほとんどの場合、ユーザーアプリケーションはこの方法を使用するには複雑過ぎるためより詳細にわたる設定が必要となってきます。

#### 29.4.6.3. 読み取り専用 bean

エンティティ bean は読み取り専用として印を付けることができます。beanが読み取り専用として印が付けられると、トランザクションの一部となることはなくなります。つまり、トランザクション的にロックされることが絶対に無くなるということになります。このオプションをつけてコミットオプション **D** を使用すると、読み取り専用 bean のデータが外部ソースによって更新されることがある場合などに非常に役立つことがあります。

bean に読み取り専用の印を付けるには、**jboss.xml** 配備記述子内で **read-only** フラグを使用します。

##### 例29.11 jboss.xml を使いエンティティ bean に読み取り専用と印付ける方法

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>MyEntityBean</ejb-name>
      <jndi-name>MyEntityHomeRemote</jndi-name>
      <read-only>True</read-only>
    </entity>
  </enterprise-beans>
</jboss>
```

#### 29.4.6.4. Read-Only メソッドを明示的に定義

エンティティ bean のデフォルトのロック動作を一通り理解すると、「データを変更しないのになぜ bean をロックするのだろうか?」という疑問が湧くかもしれません。JBoss ではエンティティ bean で読み取り専用となるメソッドを定義することができるため、こうしたタイプのメソッドのみが呼びだされる場合はトランザクション内で bean をロックしなくなります。**jboss.xml** 配備記述子内でこうした読み取り専用メソッドを定義することができます。メソッド名にワイルドカードの使用が可能です。次にすべての getter メソッドと **anotherReadOnlyMethod** を読み取り専用として宣言している例を示します。

##### 例29.12 エンティティ bean メソッドを読み取り専用として定義

```
<jboss>
  <enterprise-beans>
```

```

<entity>
  <ejb-name>nextgen.EnterpriseEntity</ejb-name>
  <jndi-name>nextgen.EnterpriseEntity</jndi-name>
  <method-attributes>
    <method>
      <method-name>get*</method-name>
      <read-only>true</read-only>
    </method>
    <method>
      <method-name>anotherReadOnlyMethod</method-name>
      <read-only>true</read-only>
    </method>
  </method-attributes>
</entity>
</enterprise-beans>
</jboss>

```

#### 29.4.6.5. Instance Per Transaction ポリシー

Instance Per Transaction ポリシーは JBoss のデフォルトロックポリシーが要因となるデッドロックやスループット関連の問題を完全に解消してくれる高度な設定になります。デフォルトのエンティティ Bean ロックポリシーは 1 つの bean にアクティブなインスタンスを 1 つしか許可しません。Instance Per Transaction ポリシーはトランザクションごとに bean の新しいインスタンスを割り当てこのインスタンスをトランザクションの終わりにドロップすることによって先の要件をなくすることができます。各トランザクションは bean のコピーを持っているため、トランザクションベースのロックは必要ありません。

このオプションは非常に便利ですが、現在、欠点も存在します。まず、このオプションのトランザクション分離動作は **READ\_COMMITTED** と同じです。必要とされない場合、反復可能読み取りを作成する可能性があります。つまり、トランザクションは古い bean のコピーを持っている可能性があるということです。次に、この設定オプションは現在、ejbLoad がトランザクションの開始時に発生しなければならないためパフォーマンスの浪費となる可能性があるコミットオプションの **B** または **C** が必要になります。ただし、ご使用のアプリケーションが現在いずれにせよこのコミットオプションの **B** または **C** を必要とする場合はこの方法を選択すべきでしょう。JBoss 開発者は現在、コミットオプション **A** も許可できる方法を模索しています (このオプションのキャッシュ機能を使用できるようにする)。

JBoss には **Instance Per Transaction CMP 2.x EntityBean** と **Instance Per Transaction BMP EntityBean** という名前のコンテナ設定があり、このコンテナ設定は、このロックポリシーを実装する standardjboss.xml で定義されています。この設定を使用するには、以下に示すように jboss.xml 配備記述子内で bean と併用するようこのコンテナ設定名を参照するだけです。

#### 例29.13 Instance Per Transaction ポリシーの使用例

```

<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>MyCMP2Bean</ejb-name>
      <jndi-name>MyCMP2</jndi-name>
      <configuration-name>
        Instance Per Transaction CMP 2.x EntityBean
      </configuration-name>
    </entity>
    <entity>

```

```

        <ejb-name>MyBMPBean</ejb-name>
        <jndi-name>MyBMP</jndi-name>
        <configuration-name>
            Instance Per Transaction BMP EntityBean
        </configuration-name>
    </entity>
</enterprise-beans>
</jboss>

```

### 29.4.7. クラスター内での実行

現在、クラスター内のエンティティ bean 向けロック機能は配信されていません。この機能はデータベースに委譲されており、アプリケーション開発者側で対応する必要があります。クラスター化されたエンティティ bean の場合、コミットオプションの **B** または **C** と行ロックメカニズムとを組み合わせで使用することをお勧めします。CMP の場合、行ロック設定オプションがあります。このオプションは bean がデータベースからロードされると SQL select for update を使用します。コミットオプション **B** または **C** を付けると、クラスター全体に使用可能なトランザクションロックを実装します。BMP の場合は、BMP の **ejbLoad** メソッド内で明示的にその select for update 呼び出しを実装する必要があります。

### 29.4.8. トラブルシューティング

本項ではロックに関してよく見られる問題とその解決方法について説明します。

#### 29.4.8.1. ロッキング動作が機能しない場合

ロック機能が動作していないように感じ、その bean への同時アクセスが見られ、結果ダーディな読み取りとなってしまった JBoss ユーザーの多くがいらっしゃいます。これについて、共通する原因を以下にいくつか示します。

- カスタムの **container-configurations** がある場合はこれらの設定が更新されていることを確認してください。
- equals および hashCode をカスタムまたは総合プライマリキーのクラス群から確実に正しく実装していることを確認してください。
- カスタムのプライマリキーあるいは複雑なプライマリキーのクラスが確実に正しくシリアル化していることを確認してください。よくある誤りのひとつに、プライマリキーがアンマーシャルされるときにメンバー変数の初期化が実行されると思い込んでいる場合があります。

#### 29.4.8.2. IllegalStateException

"removing bean lock and it has tx set!" というメッセージが付いた IllegalStateException は通常、カスタムのプライマリキーまたは複雑なプライマリキークラスに **equals** と **hashCode** のいずれかまたは両方が正しく実装されていない、あるいはプライマリキークラスがシリアル化において正しく実装されないことを意味します。

#### 29.4.8.3. ハングとトランザクションタイムアウト

トランザクションのタイムアウトは、JBoss で長期にわたり解決されていないバグです。このバグとは、トランザクションロールバックの印のみが付いていて実際にはロールバックされないのです。この処理の役目は呼び出しスレッドに委譲されています。エンティティ bean ロックなどは解放されることがないため呼び出しスレッドが無制限にハングしてしまう場合などにこれが大きな問題の要因となる可能

性があります。この問題の対処法は最適とは言えませんが、無制限にハングする可能性があるトランザクション内では単純に何も行わないようにするしかありません。よくある誤りのひとつにインターネット全体に接続を確立してしまう、あるいはトランザクション内で web-crawler を実行してしまうことがあげられます。

## 29.5. EJB タイマー設定

J2EE タイマーサービスにより EJB オブジェクトは指定した時間にタイマーコールバックの登録を行うことができるようになります。タイマーイベントは監査、報告、あるいはその他クリーンアップなど今後必要になってくる作業を行うために使用することができます。タイマーイベントは永続化されるようになっており、サーバーに障害が発生した場合であっても実行されなければなりません。EJB タイマーに対するコーディングは J2EE 仕様の標準部分となるため、このプログラミングモデルについてはここでは触れません。代わりに、ご使用の環境に最適な状態でタイマーを稼働させる方法を理解するために JBoss でのタイマーサービスの設定について見ていくことにします。

EJB タイマーサービスは **ejb-deployer.xml** ファイル内のいくつかの関連 MBean によって設定されます。主要 MBean は **EJBTimerService** MBean です。

```
<mbean code="org.jboss.ejb.txtimer.EJBTimerServiceImpl"
name="jboss.ejb:service=EJBTimerService">
  <attribute
name="RetryPolicy">jboss.ejb:service=EJBTimerService,retryPolicy=fixedDelay</attribute>
  <attribute
name="PersistencePolicy">jboss.ejb:service=EJBTimerService,persistencePolicy=database</attribute>
  <attribute
name="TimerIdGeneratorClassName">org.jboss.ejb.txtimer.BigIntegerTimerIdGenerator</attribute>
  <attribute
name="TimedObjectInvokerClassName">org.jboss.ejb.txtimer.TimedObjectInvokerImpl</attribute>
</mbean>
```

**EJBTimerService** には次のような設定可能な属性があります

- **RetryPolicy**: 再試行ポリシーを実装する MBean の名前です。この MBean は **org.jboss.ejb.txtimer.RetryPolicy interface** に対応する必要があります。JBoss は **FixedDelayRetryPolicy** という実装を 1 つ提供しており、これについては後ほど説明していきます。
- **PersistencePolicy**: タイマーイベント保存用の永続方法を実装する MBean の名前です。この MBean は **org.jboss.ejb.txtimer.PersistencePolicy** インターフェースに対応しなければなりません。JBoss は **NoopPersistencePolicy** と **DatabasePersistencePolicy** の 2 つの実装を提供しており、これについては後ほど説明していきます。
- **TimerIdGeneratorClassName**: タイマー ID 生成方法を提供するクラスの名前になります。このクラスは **org.jboss.ejb.txtimer.TimerIdGenerator** インターフェースに対応していなければなりません。JBoss は **org.jboss.ejb.txtimer.BigIntegerTimerIdGenerator** 実装を提供しています。
- **TimedObjectInvokerClassname**: タイマーメソッド呼び出し方法を提供するクラスの名前になります。このクラスは **org.jboss.ejb.txtimer.TimedObjectInvoker** インターフェースを実装していなければなりません。JBoss は **org.jboss.ejb.txtimer.TimedObjectInvokerImpl** 実装を提供しています。

使用される再試行ポリシー MBean 定義を次に示します。

```
<mbean code="org.jboss.ejb.txtimer.FixedDelayRetryPolicy"
      name="jboss.ejb:service=EJBTimerService,retryPolicy=fixedDelay">
  <attribute name="Delay">100</attribute>
</mbean>
```

再試行ポリシーは設定値を 1 つとります。

- **Delay:** 失敗したタイマー実行を再試行するまでの遅延時間 (ミリ秒) になります。デフォルトの遅延時間は 100 ミリ秒です。

EJB タイマーを永続化する必要がない場合、**NoopPersistence** ポリシーを使用することができます。この MBean はデフォルトではコメントアウトされていますが、有効にすると次のようになります。

```
<mbean code="org.jboss.ejb.txtimer.NoopPersistencePolicy"
      name="jboss.ejb:service=EJBTimerService,persistencePolicy=noop"/>
```

タイマーを使用するほとんどのアプリケーションがタイマーの永続化を必要とします。この場合には **DatabasePersistencePolicy** MBean を使用してください。

```
<mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"
      name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
  <!-- DataSource JNDI name -->
  <depends optional-attribute-
name="DataSource">jboss.jca:service=DataSourceBinding,name=DefaultDS</depends>
  <!-- The plugin that handles database persistence -->
  <attribute
name="DatabasePersistencePlugin">org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin</attribute>
</mbean>
```

- **DataSource:** タイマーデータの書き込み先となる DataSource の MBean です。
- **DatabasePersistencePlugin:** 永続方法を実装するクラスの名前になります。これは **org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin** になるはずです。

## 第30章 CMP エンジン

本章では JBoss でのコンテナ管理による永続性 (CMP) の使い方について説明します。EJB CMP モデルに関する基本的な知識があることを前提とし、JBoss CMP エンジンの操作方法を中心に解説します。特に、JBoss における CMP アプリケーションの設定と最適化を学びます。CMP の基本コンセプトなど初歩的な部分については、**Enterprise Java Beans, Fourth Edition** (O'Reilly 2004)を参照してください。

### 30.1. サンプルコード

本章はサンプルを使いながら進行します。架空の犯罪組織に関する情報を格納する犯罪ポータルアプリケーションを使用します。図30.1「犯罪ポータルのクラス例」にデータモデルを示します。

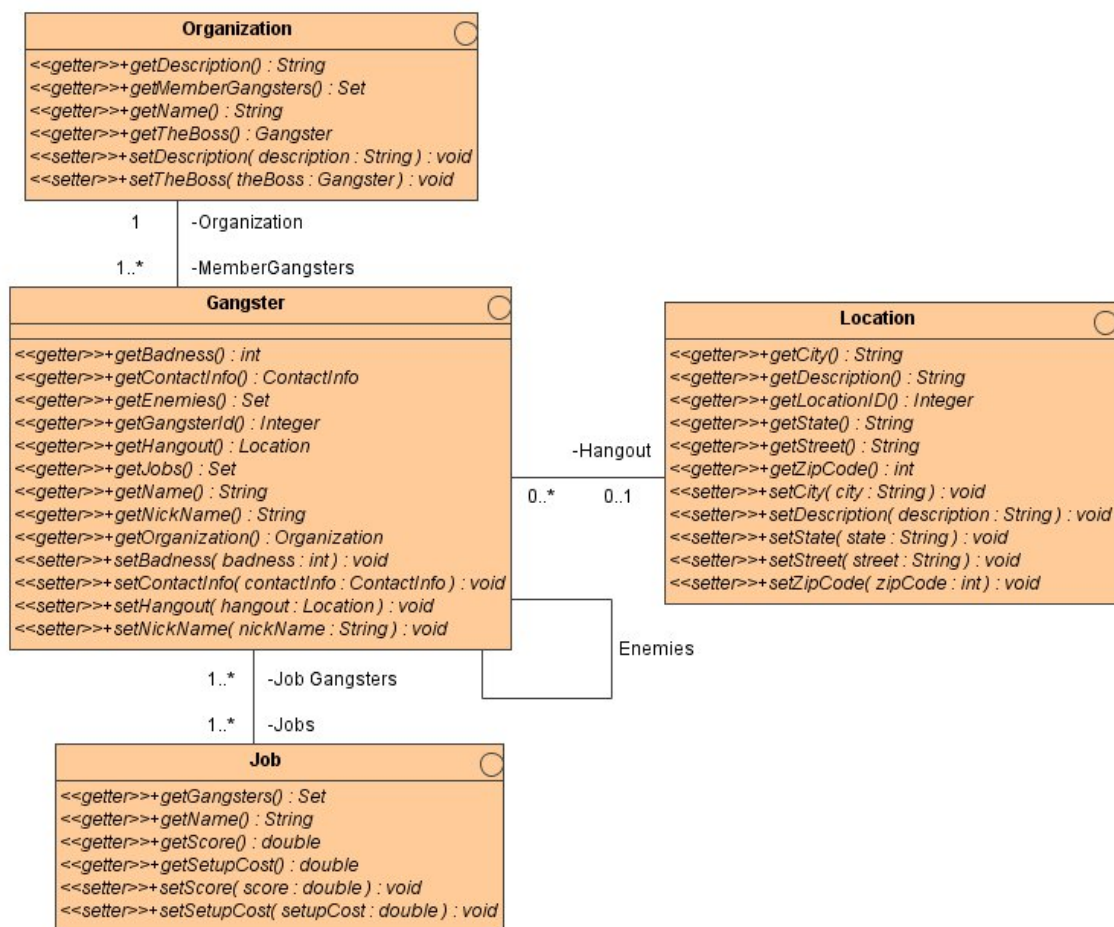


図30.1 犯罪ポータルのクラス例

犯罪ポータルのソースコードはサンプルコードの `src/main/org/jboss/cmp2` にあります。サンプルコードをビルドするには、以下のようにして Ant を実行します。

```
[examples]$ ant -Dchap=cmp2 config
```

このコマンドで JBoss サーバーにアプリケーションの構築およびデプロイを行います。JBoss サーバーを起動する場合、あるいはすでに起動している場合、次のようなデプロイメントメッセージが表示されるはずです。

```
15:46:36,704 INFO [OrganizationBean$Proxy] Creating organization Yakuza,
Japanese Gangsters
15:46:36,790 INFO [OrganizationBean$Proxy] Creating organization Mafia,
```

```

Italian Bad Guys
15:46:36,797 INFO  [OrganizationBean$Proxy] Creating organization Triads,
Kung Fu Movie Extras
15:46:36,877 INFO  [GangsterBean$Proxy] Creating Gangster 0 'Bodyguard'
Yojimbo
15:46:37,003 INFO  [GangsterBean$Proxy] Creating Gangster 1 'Master'
Takeshi
15:46:37,021 INFO  [GangsterBean$Proxy] Creating Gangster 2 'Four finger'
Yuriko
15:46:37,040 INFO  [GangsterBean$Proxy] Creating Gangster 3 'Killer' Chow
15:46:37,106 INFO  [GangsterBean$Proxy] Creating Gangster 4 'Lightning'
Shogi
15:46:37,118 INFO  [GangsterBean$Proxy] Creating Gangster 5 'Pizza-Face'
Valentino
15:46:37,133 INFO  [GangsterBean$Proxy] Creating Gangster 6 'Toothless'
Toni
15:46:37,208 INFO  [GangsterBean$Proxy] Creating Gangster 7 'Godfather'
Corleone
15:46:37,238 INFO  [JobBean$Proxy] Creating Job 10th Street Jeweler Heist
15:46:37,247 INFO  [JobBean$Proxy] Creating Job The Greate Train Robbery
15:46:37,257 INFO  [JobBean$Proxy] Creating Job Cheap Liquor Snatch and
Grab

```

サンプルの bean はデプロイを解除するとテーブルが削除されるよう設定されているため、JBoss サーバーを起動すると必ず設定ターゲットに戻り、サンプルデータの再ロードとアプリケーションの再デプロイを行う必要があります。

### 30.1.1. CMP デバッグロギングの有効化

本章のテストで有用なフィードバックを取得するためには、テストを実行する前に CMP サブシステムのログレベルを上げておきます。デバッグロギングを有効にするには、次のカテゴリを **log4j.xml** ファイルに追加します。

```

<category name="org.jboss.ejb.plugins.cmp">
    <priority value="DEBUG"/>
</category>

```

また、**CONSOLE** アペンダーのしきい値を下げ、コンソールにデバッグレベルのメッセージが記録されるように設定します。次の変更も **log4j.xml** ファイルに適用する必要があります。

```

<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
    <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
    <param name="Target" value="System.out"/>
    <param name="Threshold" value="DEBUG" />

    <layout class="org.apache.log4j.PatternLayout">
        <!-- The default pattern: Date Priority [Category] Message
-->
        <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}]
%m%n"/>
    </layout>
</appender>

```

CMP エンジンが完全に起動していることを確認するには、以下に示すように

**org.jboss.ejb.plugins.cmp** カテゴリでカスタムの **TRACE** レベル優先度を有効にする必要があります。

```
<category name="org.jboss.ejb.plugins.cmp">
    <priority value="TRACE" class="org.jboss.logging.XLevel"/>
</category>
```

### 30.1.2. サンプルの実行

最初のテストターゲットでは、本章で説明するカスタマイズ機能を示します。これらのテストを実行するには、次の ant ターゲットを実行します。

```
[examples]$ ant -Dchap=cmp2 -Dex=test run-example
```

```
22:30:09,862 DEBUG [OrganizationEJB#findByPrimaryKey] Executing SQL:
SELECT t0_OrganizationEJ
B.name FROM ORGANIZATION t0_OrganizationEJB WHERE
t0_OrganizationEJB.name=?
22:30:09,927 DEBUG [OrganizationEJB] Executing SQL: SELECT desc, the_boss
FROM ORGANIZATION W
HERE (name=?)
22:30:09,931 DEBUG [OrganizationEJB] load relation SQL: SELECT id FROM
GANGSTER WHERE (organi
zation=?)
22:30:09,947 DEBUG [StatelessSessionContainer] Useless invocation of
remove() for stateless s
ession bean
22:30:10,086 DEBUG [GangsterEJB#findBadDudes_ejbql] Executing SQL: SELECT
t0_g.id FROM GANGST
ER t0_g WHERE (t0_g.badness > ?)
22:30:10,097 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT
t0_GangsterEJB.id FRO
M GANGSTER t0_GangsterEJB WHERE t0_GangsterEJB.id=?
22:30:10,102 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT
t0_GangsterEJB.id FRO
M GANGSTER t0_GangsterEJB WHERE t0_GangsterEJB.id=?
```

これらのテストでは、マッピングの問題に関して各種の finder、selector、object を実行させます。本章ではこれらのテストについて見ていきます。

その他メインターゲットは「[最適化ローディング](#)」に記載されている最適化ローディング設定を行うテスト式を実行します。これでログ機能が正しく設定され、read-ahead テストは実行したクエリーに関する便利な情報を表示するようになります。log4j.xml ファイルに対する変更を認識させるのに JBoss サーバーを起動する必要はありませんが、反映されるのに 1 分ほどかかる場合があります。readahead クライアントの実際に実行したものを以下に示します。

```
[examples]$ ant -Dchap=cmp2 -Dex=readahead run-example
```

readahead クライアントを実行すると、テスト中に実行される SQL クエリはすべて JBoss サーバーコンソールに表示されます。出力を分析する場合、実行されるクエリ数、選択したカラム、ロードされる行数に着目してください。次に readahead からの JBoss サーバーコンソール出力で read-ahead none の部分を示します。



```

22:44:31,570 INFO  [ReadAheadTest]
#####
### read-ahead none
###
22:44:31,582 DEBUG [GangsterEJB#findAll_none] Executing SQL: SELECT
t0_g.id FROM GANGSTER t0_
g ORDER BY t0_g.id ASC
22:44:31,604 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,615 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,622 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,628 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,635 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,644 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,649 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,658 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,670 INFO  [ReadAheadTest]
###
#####
...

```

このサンプルについては、最適化ローディングの設定の際に再度検証します。

## 30.2. JBOSSCMP-JDBC の構造

**jbosscmp-jdbc.xml** 記述子を使って JBoss エンジンの動作を制御します。サーバー設定ファイルセットにある **conf/standardjbosscmp-jdbc.xml** 記述子でグローバルに行うか、**META-INF/jbosscmp-jdbc.xml** 記述子で EJB JAR デプロイメントごとに実行することができます。

**jbosscmp-jdbc.xml** 記述子の DTD は **JBOSS\_DIST/docs/dtd/jbosscmp-jdbc\_4\_0.dtd** にあります。この DTD 用のパブリック doctype :

```

<!DOCTYPE jbosscmp-jdbc PUBLIC
"-//JBoss//DTD JBOSSCMP-JDBC 4.0//EN"
"http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_4_0.dtd">

```

トップレベルの要素は、[図30.2 「jbosscmp-jdbc コンテンツモデル」](#) で示しています。

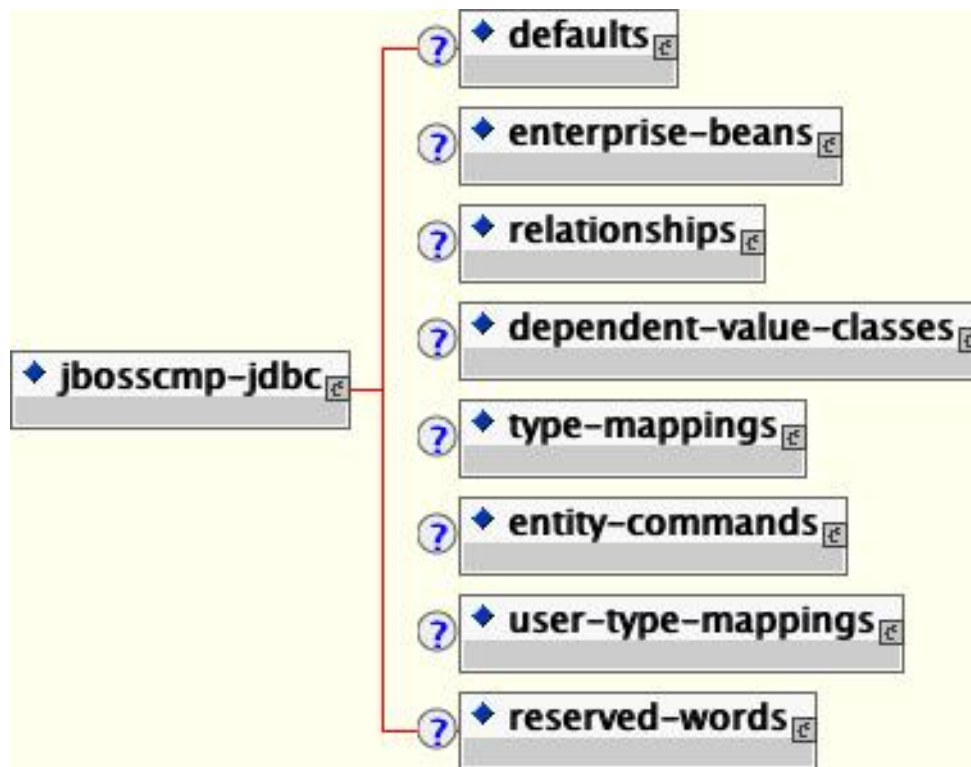


図30.2 jbossCMP-jdbc コンテンツモデル

- **defaults:** デフォルトセクションではエンティティ bean を制御する動作のデフォルト動作/設定を指定できます。このセクションを使用するとエンティティ bean セクションにて共通動作に必要とされる情報量を低減します。デフォルトコンテンツについては「[デフォルト](#)」を参照してください。
- **enterprise-beans:** **enterprise-beans** 要素は `ejb-jar.xml` `enterprise-beans` 記述子で定義されるエンティティ bean のカスタマイズが可能です。これについては「[エンティティ Bean](#)」で詳しく説明しています。
- **relationships:** **relationships** 要素は、エンティティの関係に関するローディング動作やテーブルのカスタマイズを行うことができます。これについては「[コンテナ管理リレーション](#)」で詳しく説明しています。
- **dependent-value-classes:** **dependent-value-classes** 要素では 依存値クラス (DVC: Dependent value class) のテーブルへのマッピングをカスタマイズすることができます。依存値クラスについては「[依存値クラス\(DVC: Dependent Value Class\)](#)」(DVC) で詳しく説明しています。
- **type-mappings:** **type-mappings** 要素は、SQL テンプレートや関数マッピングとともに、データベース用に Java から SQL の型マッピングを定義します。これについては「[データソースのカスタマイズ](#)」で詳しく説明しています。
- **entity-commands:** **entity-commands** 要素は、永続ストアでのエンティティインスタンスの作成方法を把握しているエンティティ作成コマンドインスタンスの定義ができます。これについては「[エンティティコマンドおよびプライマリキー生成](#)」で詳しく説明しています。
- **user-type-mappings:** **user-type-mappings** 要素はマッパークラスを使った列へのユーザータイプのマッピングを定義します。マッパーとはいわば仲介役です。格納時、ユーザータイプのインスタンスを取得してカラム値に変換します。ロードする場合は、カラム値を取得してそれをユーザータイプのインスタンスに変換します。ユーザータイプのマッピングについては「[ユーザータイプマッピング](#)」で詳しく説明しています。

- **reserved-words: reserved-words** 要素はテーブル生成時にエスケープすべき 1 つ以上の予約語を定義します。各予約語は word 要素の内容として指定されます。

### 30.3. エンティティ BEAN

まず、犯罪ポータル CMP エンティティ bean の一つを検証しながら、JBoss におけるエンティティ bean について説明します。ローカルの CMP エンティティ bean として実装される gangster bean を検証してみましょう。JBoss は、ローカルのエンティティ bean と同じようにパフォーマンスがアップするようにリモートのエンティティ bean に参照渡しセマンティクスを提供できますが、ローカルのエンティティ bean の使用を強く推奨します。

まず、必要なホームインターフェースから見ていきます。ここでは CMP フィールドのみを集中して検証するため、CMP フィールドを処理するメソッドだけを示します。

```
// Gangster Local Home Interface
public interface GangsterHome
    extends EJBLocalHome
{
    Gangster create(Integer id, String name, String nickName)
        throws CreateException;
    Gangster findByPrimaryKey(Integer id)
        throws FinderException;
}
```

ローカルインターフェースはクライアントがやり取りに使用するインターフェースです。ここでも CMP フィールドアクセッサのみを含みます。

```
// Gangster Local Interface
public interface Gangster
    extends EJBLocalObject
{
    Integer getGangsterId();

    String getName();

    String getNickName();
    void setNickName(String nickName);

    int getBadness();
    void setBadness(int badness);
}
```

最後に、実際の gangster bean が入ります。サイズに関わらず、実際に必要なコードは限られます。クラスの大部分は create メソッドです。

```
// Gangster Implementation Class
public abstract class GangsterBean
    implements EntityBean
{
    private EntityContext ctx;
    private Category log = Category.getInstance(getClass());
    public Integer ejbCreate(Integer id, String name, String nickName)
        throws CreateException
    {
        log.info("Creating Gangster " + id + " '" + nickName + "' "+
```

```

name);
    setGangsterId(id);
    setName(name);
    setNickName(nickName);
    return null;
}

public void ejbPostCreate(Integer id, String name, String nickName) {
}

// CMP field accessors -----
--
public abstract Integer getGangsterId();
public abstract void setGangsterId(Integer gangsterId);
public abstract String getName();
public abstract void setName(String name);
public abstract String getNickName();
public abstract void setNickName(String nickName);
public abstract int getBadness();
public abstract void setBadness(int badness);
public abstract ContactInfo getContactInfo();
public abstract void setContactInfo(ContactInfo contactInfo);
//...

// EJB callbacks -----
--
public void setEntityContext(EntityContext context) { ctx = context;
}

public void unsetEntityContext() { ctx = null; }
public void ejbActivate() { }
public void ejbPassivate() { }
public void ejbRemove() { log.info("Removing " + getName()); }
public void ejbStore() { }
public void ejbLoad() { }
}

```

ここで足りないのは **ejb-jar.xml** 配備記述子のみです。実際の bean クラスの名前は **GangsterBean** ですが、このエンティティをここでは **GangsterEJB** と呼んでいます。

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-
J2EE_Certification_and_Standards_Compliance" version="2.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/.ejb-
jar_2_1.xsd">
    <display-name>Crime Portal</display-name>

    <enterprise-beans>
        <entity>
            <display-name>Gangster Entity Bean</display-name>
            <ejb-name>GangsterEJB</ejb-name>
            <local-home>org.jboss.cmp2.crimeportal.GangsterHome</local-
home>
            <local>org.jboss.cmp2.crimeportal.Gangster</local>

```

```

    <ejb-class>org.jboss.cmp2.crimeportal.GangsterBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>gangster</abstract-schema-name>

    <cmp-field>
      <field-name>gangsterId</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>name</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>nickName</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>badness</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>contactInfo</field-name>
    </cmp-field>
    <primkey-field>gangsterId</primkey-field>

    <!-- ... -->
  </entity>
</enterprise-beans>
</ejb-jar>

```

これが EJB 2.x CMP エンティティ bean であることを表すために CMP バージョン **2.x** を指定しているので注意してください。抽象スキーマ名は **gangster** に設定しました。「[クエリ](#)」で EJB-QL クエリーを確認する際にこれが重要となります。

### 30.3.1. エンティティマッピング

エンティティの JBoss 設定は **jbosscmp-jdbc.xml** ファイル内の **entity** 要素で宣言されます。このファイルは EJB JAR の **META-INF** ディレクトリに配置され、CMP マッピング設定用の全オプション設定情報を含んでいます。各 bean の **entity** 要素はトップレベルの **jbosscmp-jdbc** 要素の配下にある **enterprise-beans** 要素内にグループ化されます。スタブアウトしたエンティティ設定を以下に示します。

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jbosscmp-jdbc PUBLIC
  "-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
  "http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_2.dtd">
<jbosscmp-jdbc>
  <defaults>
    <!-- application-wide CMP defaults -->
  </defaults>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- overrides to defaults section -->
      <table-name>gangster</table-name>
      <!-- CMP Fields (see CMP-Fields) -->
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>

```

```
        <!-- Load Groups (see Load Groups)-->
        <!-- Queries (see Queries) -->
    </entity>
</enterprise-beans>
</jbosscmp-jdbc>
```

**ejb-name** 要素は、**ejb-jar.xml** ファイルにあるものとエンティティの仕様を適合させる必要があります。その要素の残りはグローバルか、又はアプリケーションレベルの CMP デフォルトと bean に特有の CMP マッピング詳細を上書きします。アプリケーションデフォルトは **jbosscmp-jdbc.xml** ファイルの **defaults** セクションから出るものであり、グローバルデフォルトは現在のサーバー設定ファイルセットの **conf** ディレクトリにある **defaults** セクションから出るものです。**defaults** セクションについては、「[デフォルト](#)」で説明があります。[図30.3「エンティティ要素のコンテンツモデル」](#) は完全な **entity** コンテンツモデルを示しています。

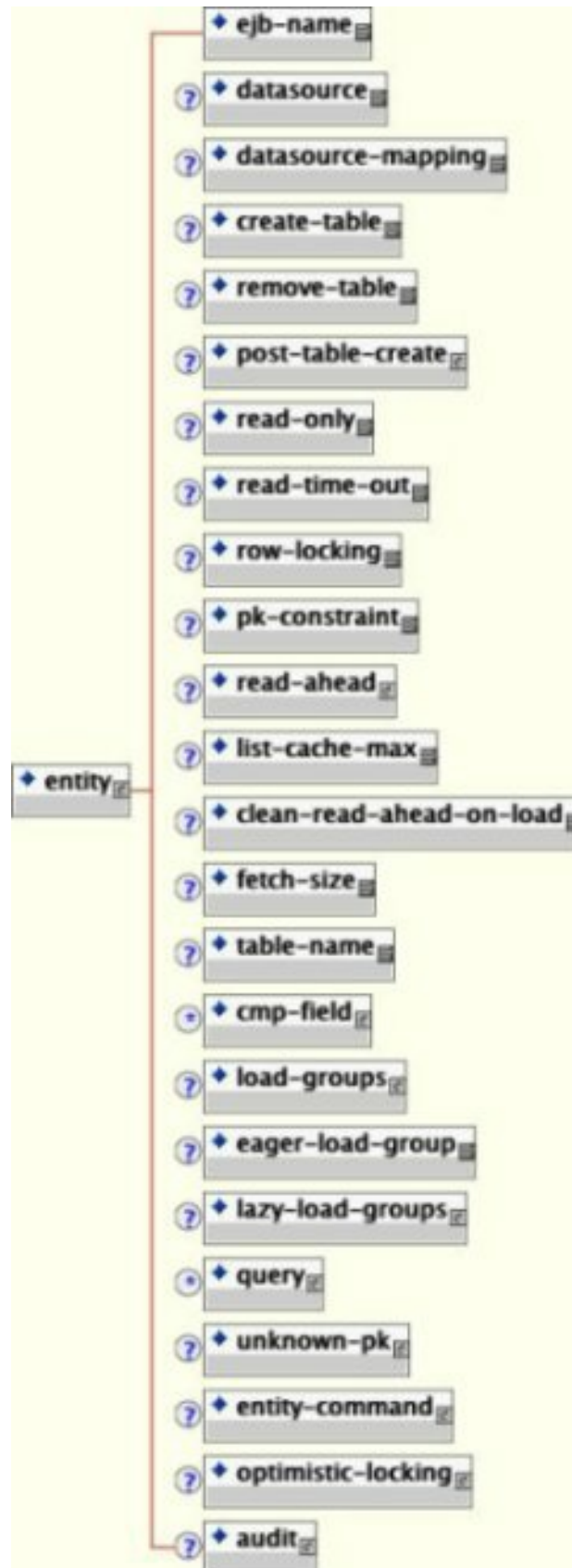


図30.3 エンティティ要素のコンテンツモデル

各エンティティの要素の詳細は次の通りになります。

- **ejb-name**: この必須要素は、この設定が適用される EJB 名です。この要素は `ejb-jar.xml` ファイル内のエンティティの `ejb-name` と一致しなければなりません。
- **datasource**: この任意要素はデータソースの検索に使用される `jndi-name` になります。エン



ティティまたは関係テーブルで使用するデータベース接続はすべてデータソースから取得されます。エンティティに対して異なるデータソースを指定すると、finder および ejbSelect がクエリできるドメインを大幅に制約することになるため推奨しません。デフォルトセクションでオーバーライドされない限り `java:/DefaultDS` がデフォルトです。

- **datasource-mapping:** この任意要素は **type-mapping** 名を指定し、Java タイプが SQL タイプにマッピングされる方法、および EJB-QL 関数がデータベース固有の関数にマッピングされる方法を指定します。タイプのマッピングについては「[マッピング](#)」で説明します。デフォルトセクションでオーバーライドされない限り **HypersonicSQL** がデフォルトです。
- **create-table:** この任意要素が true の場合 JBoss はエンティティに対してテーブルの作成を試行しなければならないことを指定します。アプリケーションがデプロイされると、JBoss はテーブル作成の前にすでにテーブルが存在しているかどうか確認します。テーブルが見つかった場合、ログが記録されるだけでテーブルは作成されません。このオプションは、テーブルの構成が頻繁に変更する開発初期段階で非常に役立ちます。デフォルトセクションでオーバーライドされない限り、デフォルトは false です。
- **alter-table:** スキーマの自動作成に **create-table** が使用される場合、**alter-table** を使ってエンティティ bean に対する変更を反映しスキーマを最新状態に保つことができます。Alter table は次のような特定のタスクを実行します。
  - 新しいフィールドが作成されます。
  - 使用されなくなったフィールドが削除されます。
  - 宣言されている長さより短い文字列フィールドは宣言されている長さまで延長されます(すべてのデータベースでサポートされているわけではありません)。
- **remove-table:** この任意の要素が true の場合、JBoss は各エンティティおよび関連性をマッピングした各関係テーブルに対してテーブルのドロップを試行します。アプリケーションがアンデプロイされると、JBoss はこのテーブルをドロップを試行します。このオプションは、テーブルの構成が頻繁に変更される開発初期段階で非常に役立ちます。デフォルトセクションでオーバーライドされない限り、デフォルトは false です。
- **post-table-create:** この任意要素はデータベーステーブルの作成直後に実行されるべき任意の SQL ステートメントを指定します。このコマンドは **create-table** が true でテーブルが以前に存在しない場合にのみ実行されます。
- **read-only:** この任意の要素が true の場合、bean プロバイダーはいずれのフィールドの値を変更することも許可されないことを指定します。読み取り専用のフィールドはデータベースに格納もしくは挿入されません。プライマリキーフィールドが読み取り専用である場合、create メソッドが **CreateException** を送出します。set アクセッサが読み取り専用フィールドで呼び出されると、**EJBException** をスローします。読み取り専用フィールドは最後の更新などデータベースのトリガーで入力されるフィールドに便利です。**read-only** オプションは **cmp-field** ごとにオーバーライドが可能です。詳しくは「[Read-only フィールド](#)」で説明します。**defaults** セクションでオーバーライドされない限り、デフォルトは false です。
- **read-time-out:** この任意の要素は読み取り専用フィールドでの読み取りが有効である期間をミリ秒単位で表します。値が 0 の場合、常にトランザクション開始時に値が再ロードされることになり、値が -1 の場合は値がタイムアウトすることがないという意味になります。このオプションも **cmp-field** ごとにオーバーライドが可能です。**read-only** が false の場合、この値は無視されます。デフォルトは **defaults** セクションでオーバーライドされない限り -1 です。
- **row-locking:** この任意の要素が true の場合、JBoss はトランザクションでロードされたすべての列をロックすることを指定します。ほとんどのデータベースがエンティティをロードする



際に **SELECT FOR UPDATE** 構文を使ってこれを実装していますが、実際の構文はこのエンティティで使用されるデータソースマッピング内の **row-locking-template** で確定されます。**defaults** セクションでオーバーライドされない限り、デフォルトは **false** です。

- **pk-constraint**: この任意の要素が **true** の場合、JBoss はテーブル作成時にプライマリキー制約を追加することを指定します。**defaults** セクションでオーバーライドされない限り、デフォルトは **true** です。
- **read-ahead**: この任意の要素はエンティティの **cmr-fields** およびクエリ結果のキャッシュ化を制御します。このオプションについては「[Read-ahead](#)」で説明します。
- **fetch-size**: この任意の要素は基礎となるデータストアへの 1 往復で読み込むエンティティ数を指定します。**defaults** セクションでオーバーライドされない限り、デフォルトは 0 です。
- **list-cache-max**: この任意の要素はこのエンティティで追跡可能な read-list 数を指定します。このオプションについては **on-load** で説明します。**defaults** セクションでオーバーライドされない限り、デフォルトは 1000 です。
- **clean-read-ahead-on-load**: 先行読み込み (read ahead) キャッシュからエンティティがロードされる場合、JBoss は先行読み込みキャッシュから使用されるデータを削除することができます。デフォルトは **false** になります。
- **table-name**: この任意の要素はこのエンティティ用のデータを保持するテーブル名になります。各エンティティインスタンスはこのテーブルの 1 列内に格納されます。デフォルトは **ejb-name** になります。
- **cmp-field**: この任意の要素で **ejb-jar.xml** の **cmp-field** を永続ストア上でマッピングする方法を定義することができます。このオプションについては「[CMP フィールド](#)」で説明します。
- **load-groups**: この任意の要素はフィールドのロードグループ化を宣言する CMP フィールドの 1 つ以上のグループ化を指定します。このオプションについては「[Load Groups](#)」で説明します。
- **eager-load-groups**: このオプションの要素は eager load group として 1 つ以上のロードグループ化を定義します。このオプションについては「[一括読み込みプロセス](#)」で説明します。
- **lazy-load-groups**: この任意の要素は lazy load group として 1 つ以上のロードグループ化を定義します。これについては「[遅延ローディングプロセス](#)」で説明します。
- **query**: この任意の要素は finder と selector の定義を指定します。これについては「[クエリ](#)」で説明します。
- **unknown-pk**: この任意の要素により **java.lang.Object** の未知のプライマリキータイプを永続ストアにマッピングする方法を定義することができます。
- **entity-command**: この任意の要素によりエンティティ作成コマンドインスタンスを定義することができます。一般的にこれを使いカスタムのコマンドインスタンスを定義することで、プライマリキー生成を可能にします。これについては「[エンティティコマンドおよびプライマリキー生成](#)」で詳しく説明しています。
- **optimistic-locking**: この任意の要素は楽観的なロッキングに対して使用するストラテジーを定義します。これについては「[楽観的ロッキング](#)」で説明します。
- **audit**: この任意の要素は監査予定の CMP フィールドを定義します。詳しくは「[エンティティアクセスの監査](#)」で説明します。

## 30.4. CMP フィールド

CMP フィールドは、JavaBean プロパティアクセッサの規定に従う抽象 getter と setter のメソッドとして bean クラス上で宣言されます。たとえば、gangster bean には **name** CMP フィールドへアクセスできるように **getName()** と **setName()** メソッドがあります。本項では、これらの宣言 CMP フィールドの設定方法、永続性や動作の制御方法について検証します。

### 30.4.1. CMP フィールド宣言

CMP フィールドの宣言は **ejb-jar.xml** ファイル内で始まります。たとえば、gangster bean では **name**、**nickName**、**badness** が **ejb-jar.xml** ファイルで次のように宣言されます。

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <cmp-field><field-name>gangsterId</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>nickName</field-name></cmp-field>
      <cmp-field><field-name>badness</field-name></cmp-field>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

J2EE 配備記述子はオブジェクトの関係マッピング詳細やその他の設定については宣言しないので注意してください。単純に CMP フィールドの宣言です。

### 30.4.2. CMP フィールドのカラムマッピング

CMP フィールドの関係マッピング設定は **jbosscmp-jdbc.xml** ファイルで行います。この構成は配下に **cmp-field** 要素と追加設定詳細を持つエンティティ **element** がある **ejb-jar.xml** に似ています。

以下に gangster bean の基本的なカラム名とデータタイプマッピングを示します。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <table-name>gangster</table-name>

      <cmp-field>
        <field-name>gangsterId</field-name>
        <column-name>id</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>name</field-name>
        <column-name>name</column-name>
        <not-null/>
      </cmp-field>
      <cmp-field>
        <field-name>nickName</field-name>
        <column-name>nick_name</column-name>
        <jdbc-type>VARCHAR</jdbc-type>
```

```

    <sql-type>VARCHAR(64)</sql-type>
  </cmp-field>
  <cmp-field>
    <field-name>badness</field-name>
    <column-name>badness</column-name>
  </cmp-field>
</entity>
</enterprise-beans>
</jbosscmp-jdbc>

```

jbosscmp-jdbc.xml の cmp-field 要素の完全コンテンツモデルを以下に示します。

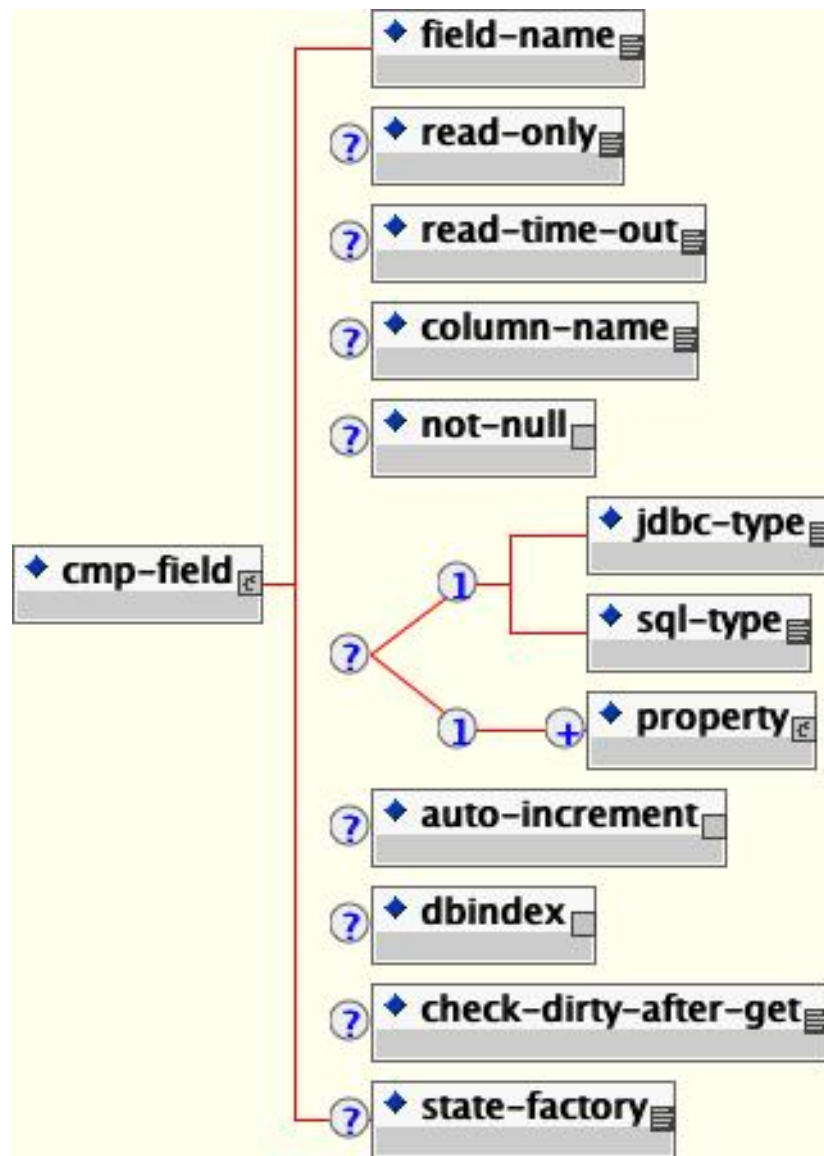


図30.4 JBoss エンティティ要素のコンテンツモデル

各要素の詳細は次の通りになります。

- **field-name**: この必須の要素は設定中の **cmp-field** 名です。ejb-jar.xml ファイル内でこのエンティティ用に宣言される **cmp-field** の **field-name** 要素と一致しなければなりません。
- **read-only**: これは対象となるフィールドが読み取り専用であることを宣言するものです。このフィールドは JBoss ではデータベースに書き込まれません。読み取り専用フィールドについては「[Read-only フィールド](#)」で説明します。

- **read-only-timeout**: read-only フィールドの値が有効と判断した期間をミリ秒単位で表します。
- **column-name**: この任意の要素は **cmp-field** のマッピング先となるカラム名です。 **field-name** 値を使用するのがデフォルトです。
- **not-null**: この任意の要素は、このエンティティ用の表の自動作成時に JBoss が NOT NULL をカラム宣言の末尾に追加すべきことを示します。プライマリキーフィールドとプリミティブ用のデフォルトは not null です。
- **jdbc-type**: JDBC で準備されたステートメント内にパラメーターを設定するか、JDBC 結果セットからデータをロードする場合に使用される JDBC タイプです。有効なタイプは **java.sql.Types** で定義されます。 **sql-type** が指定された場合にのみ必要です。デフォルトの JDBC タイプは **datasourcemapping** 内のデータベースタイプに基づきます。
- **sql-type**: このフィールドの create テーブルステートメントで使用される SQL タイプです。有効な SQL タイプはデータベースベンダーによって異なります。 **jdbc-type** が指定された場合にのみ必要です。デフォルトの SQL タイプは **datasourcemapping** 内のデータベースタイプによって決定されます。
- **property**: この任意の要素により dependent value class CMP フィールドのプロパティが永続ストアにどのようにマッピングされるか定義することができるようになります。これについては「[依存値クラス\(DVC: Dependent Value Class\)](#)」で詳しく説明します。
- **auto-increment**: この任意のフィールドがあると、データベース層ごとに自動的に増分されることを示します。これはフィールドを生成されたカラムやかい部で操作されたカラムにマッピングするために使用します。
- **dbindex**: この任意のフィールドがあると、サーバーはデータベース内の該当するカラムにインデックスを作成することを示します。このインデックス名は **fieldname\_index** になります。
- **check-dirty-after-get**: この値はプリミティブタイプとそのベシック **java.lang** 不変ラッパー (**Integer**、**String** など) に **false** をデフォルト設定します。不変となる可能性があるオブジェクトに対しては、JBoss は **get** オペレーションのあとダーティの可能性ありとしてそのフィールドをマークします。オブジェクトでのダーティチェックにおける金額的な負担が大きすぎる場合は、**check-dirty-after-get** を **false** に設定して最適化することができます。
- **state-factory**: このフィールドにダーティチェックを行うことができる状態ファクトリオブジェクトのクラス名を指定します。状態ファクトリクラスは **CMPFieldStateFactory** インターフェースを実装する必要があります。

### 30.4.3. Read-only フィールド

JBoss では **cmp-field** 宣言で **read-only** と **read-time-out** の要素を設定することで読み取り専用 CMP フィールドを有効にします。これらの要素はエンティティレベルでの動作と同じです。フィールドが読み取り専用の場合、**INSERT** または **UPDATE** のステートメントで使用されることはありません。プライマリキーフィールドが read-only の場合、create メソッドは **CreateException** を送出します。set アクセッサが読み取り専用フィールドで呼びだされると、**EJBException** をスローします。読み取り専用フィールドは最後の更新などデータベーストリガーで入力されるフィールドに便利です。読み取り専用 CMP フィールド宣言のサンプルを次に示します。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <cmp-field>
```

```

        <field-name>lastUpdated</field-name>
        <read-only>true</read-only>
        <read-time-out>1000</read-time-out>
    </cmp-field>
</entity>
</enterprise-beans>
</jbosscmp-jdbc>

```

#### 30.4.4. エンティティアクセスの監査

エンティティセクションの **audit** 要素により、エンティティ bean とアクセス権限の監査の方法を指定できます。これが確立した呼び出し ID となるように、セキュリティドメイン下で エンティティ bean にアクセスする場合にのみ可能となります。audit 要素のコンテンツモデルは図30.5「jbosscmp-jdbc.xml 監査要素のコンテンツモデル」でご覧になれます。

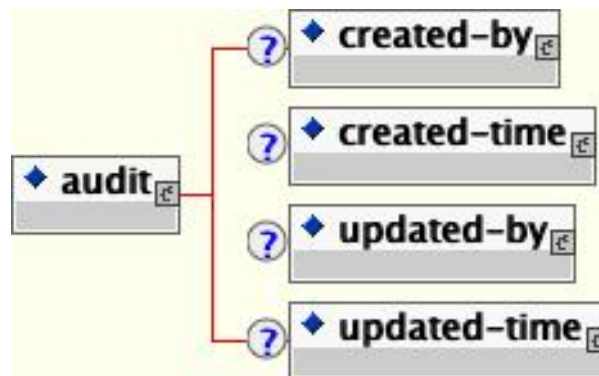


図30.5 jbosscmp-jdbc.xml 監査要素のコンテンツモデル

- **created-by**: この任意の要素はエンティティを作成した呼び出し側が指定の **column-name** または cmp **field-name**のいずれかに保存されるよう指定します。
- **created-time**: この任意の要素はエンティティの作成時間が指定の **column-name** または cmp **field-name**のいずれかに保存されるよう指定します。
- **updated-by**: この任意の要素はエンティティを最後に変更した呼び出し側が指定の **column-name** あるいは CMP **field-name**のいずれかに保存されるよう指示します。
- **updated-time**: この任意の要素はエンティティの最終更新時間が指定の **column-name** または CMP **field-name**のいずれかに保存されるよう指定します。

各要素に対して、**field-name** が与えられる場合は、アクセスされたエンティティ bean の指定 CMP フィールドに、該当する監査情報が格納されるようになっています。エンティティ上に該当する CMP フィールドが宣言される必要はありません。一致するフィールド名がある場合は、該当する CMP フィールドの抽象 getter と setter を使ってアプリケーション内の監査フィールドにアクセスすることができます。これ以外は、監査フィールドが作成されて内部的にエンティティに追加されます。監査フィールド名を使って EJB-QL クエリ内の監査情報にアクセスすることはできますが、エンティティアクセッサから直接アクセスすることはできません。

一方、**column-name** が指定されている場合、該当する監査情報は指定のエンティティテーブルのカラムに格納されます。JBoss がテーブルを作成すると **jdbc-type** と **sql-type** 要素を使用して、ストレージタイプを定義することができます。

特定のカラム名を持つ監査情報の宣言を以下に示します。

```

<jbosscmp-jdbc>

```

```

<enterprise-beans>
  <entity>
    <ejb-name>AuditChangedNamesEJB</ejb-name>
    <table-name>cmp2_audit_changednames</table-name>
    <audit>
      <created-by>
        <column-name>createdby</column-name>
      </created-by>
      <created-time>
        <column-name>createdtime</column-name>
      </created-time>
      <updated-by>
        <column-name>updatedby</column-name></updated-by>
      <updated-time>
        <column-name>updatedtime</column-name>
      </updated-time>
    </audit>
  </entity>
</enterprise-beans>
</jbosscomp-jdbc>

```

### 30.4.5. 依存値クラス(DVC: Dependent Value Class)

DVC (dependent value class) とは、文字列や数値など自動認識されるコアタイプ以外の **cmp-field** タイプである Java クラスの識別に使用する用語です。デフォルトでは、DVC はシリアル化され、シリアル化形式がデータベースの 1 カラムに格納されます。ここでは説明しませんが、シリアル化形式でのクラスの長期保管には既知の問題がいくつかあります。

JBoss は DVC 内部データを 1 つ以上のカラムへ保存することができます。これはレガシーの JavaBean やデータベース構造に対応する場合に便利です。非常にフラットな構造を持つデータベースを検索するのは珍しくありません(**SHIP\_LINE1**、**SHIP\_LINE2**、**SHIP\_CITY**などのフィールドや請求書送付先の追加フィールドセットを持つ **PURCHASE\_ORDER** テーブルなど)。この他一般的なデータベース構造としてはエリアコード、外線、内線の別フィールドを持つ電話番号、もしくは複数のフィールドにわたる人名などが挙げられます。DVC では複数のカラムを 1 つの論理フィールドにマッピングすることが可能です。

JBoss は、マッピングされる DVC がシンプルプロパティの JavaBeans ネーミング仕様に準ずること、またデータベース内に格納される各プロパティが **getter** と **setter** の両方のメソッドを持っていることが必要となります。さらに、bean は連続化が可能でなければならない、引数なしのコンストラクターを持っていなければなりません。プロパティはシンプルタイプのマッピングされない DVC あるいはマッピングされる DVC のいずれでも構いませんが、EJB にはなりません。DVC マッピングは **dependent-value-classes** 要素内の **dependent-value-class** 要素に指定されます。

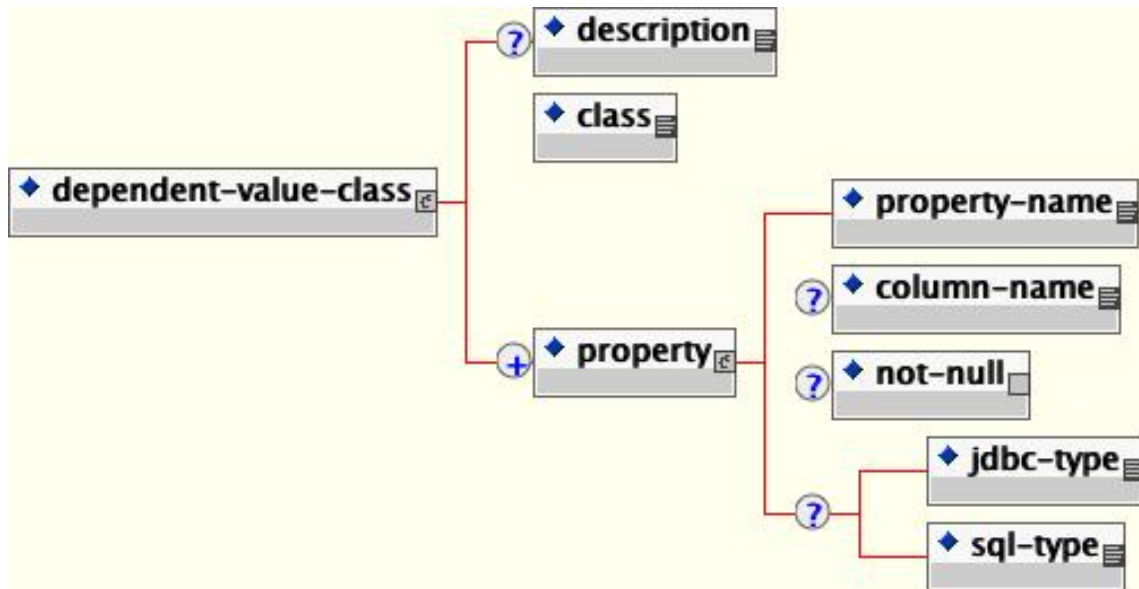


図30.6 jbossCMP-jdbc dependent-value-class 要素モデル

次にシンプルな **ContactInfo** DVC クラスの例を示します。

```

public class ContactInfo
    implements Serializable
{
    /** The cell phone number. */
    private PhoneNumber cell;

    /** The pager number. */
    private PhoneNumber pager;

    /** The email address */
    private String email;

    /**
     * Creates empty contact info.
     */
    public ContactInfo() {
    }

    public PhoneNumber getCell() {
        return cell;
    }

    public void setCell(PhoneNumber cell) {
        this.cell = cell;
    }

    public PhoneNumber getPager() {
        return pager;
    }

    public void setPager(PhoneNumber pager) {
        this.pager = pager;
    }
}
  
```



```

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email.toLowerCase();
    }

    // ... equals, hashCode, toString
}

```

連絡先情報には電話番号が含まれ、これは別の DVC クラスで表されます。

```

public class PhoneNumber
    implements Serializable
{
    /** The first three digits of the phone number. */
    private short areaCode;

    /** The middle three digits of the phone number. */
    private short exchange;

    /** The last four digits of the phone number. */
    private short extension;

    // ... getters and setters

    // ... equals, hashCode, toString
}

```

これら 2 クラスの DVC マッピングは比較的簡単です。

```

<dependent-value-classes>
  <dependent-value-class>
    <description>A phone number</description>
    <class>org.jboss.cmp2.crimeportal.PhoneNumber</class>
    <property>
      <property-name>areaCode</property-name>
      <column-name>area_code</column-name>
    </property>
    <property>
      <property-name>exchange</property-name>
      <column-name>exchange</column-name>
    </property>
    <property>
      <property-name>extension</property-name>
      <column-name>extension</column-name>
    </property>
  </dependent-value-class>

  <dependent-value-class>
    <description>General contact info</description>
    <class>org.jboss.cmp2.crimeportal.ContactInfo</class>
    <property>
      <property-name>cell</property-name>
      <column-name>cell</column-name>
    </property>
  </dependent-value-class>
</dependent-value-classes>

```



```

        </property>
        <property>
            <property-name>pager</property-name>
            <column-name>pager</column-name>
        </property>
        <property>
            <property-name>email</property-name>
            <column-name>email</column-name>
            <jdbc-type>VARCHAR</jdbc-type>
            <sql-type>VARCHAR(128)</sql-type>
        </property>
    </dependent-value-class>
</dependent-value-classes>

```

各 DVC は **dependent-value-class** 要素で宣言します。DVC は、クラス要素で宣言される Java クラスタイプで識別されます。各プロパティの永続化はプロパティ要素で宣言されます。この仕様は **cmp-field** 要素に基づくため、すぐに理解できるようになっています。この制限も今後のリリースでは削除される予定です。現在は、ローカルエンティティの場合はプライマリーフィールド、リモートエンティティの場合はエンティティハンドルを格納する設定が提案されています。

**dependent-value-classes** セクションは内部構造とデフォルトのクラスマッピングを定義します。未知のタイプのフィールドがある場合、JBoss は登録済みの DVC リストを検索します。DVC が見つかったら、カラムセットにこのフィールドを永続化します。それ以外の場合は、単一の列にシリアル化形式で格納されます。JBoss では DVC の継承をサポートしていません。つまり、この検索は宣言タイプのフィールドのみを対象とします。DVC は他の DVC から構築することができ、JBoss が DVC を発見した場合、DVC ツリー構造をカラムセットにフラット化します。起動時に DVC 回路を発見した場合、JBoss は **EJBException** を送出します。プロパティのカラム名はデフォルトでは **cmp-field** の後に下線とプロパティのカラム名が付きます。プロパティが DVC の場合、このプロセスが繰り返されます。たとえば、**info** とつけられた **ContactInfo** を使用する **cmp-field** の場合は、

```

info_cell_area_code
info_cell_exchange
info_cell_extension
info_pager_area_code
info_pager_exchange
info_pager_extension
info_email

```

自動生成されるカラム名は長すぎる上、認識しにくくなります。カラムのデフォルトマッピングは、以下のようにエンティティ要素でオーバーライド可能です

```

<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <cmp-field>
                <field-name>contactInfo</field-name>
                <property>
                    <property-name>cell.areaCode</property-name>
                    <column-name>cell_area</column-name>
                </property>
                <property>
                    <property-name>cell.exchange</property-name>
                    <column-name>cell_exch</column-name>
                </property>
            </cmp-field>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>

```

```

        <property>
            <property-name>cell.extension</property-name>
            <column-name>cell_ext</column-name>
        </property>

        <property>
            <property-name>pager.areaCode</property-name>
            <column-name>page_area</column-name>
        </property>
        <property>
            <property-name>pager.exchange</property-name>
            <column-name>page_exch</column-name>
        </property>
        <property>
            <property-name>pager.extension</property-name>
            <column-name>page_ext</column-name>
        </property>

        <property>
            <property-name>email</property-name>
            <column-name>email</column-name>
            <jdbc-type>VARCHAR</jdbc-type>
            <sql-type>VARCHAR(128)</sql-type>
        </property>
    </cmp-field>
</entity>
</enterprise-beans>
</jbosscmp-jdbc>

```

エンティティのプロパティ情報をオーバーライドする場合、**cell.areaCode**のようなフラットな観点からプロパティを参照してください。

## 30.5. コンテナ管理リレーション

コンテナ管理リレーション (CMR: Container Managed Relationship) は、CMP 2.0 の力強い新機能です。プログラマーは EJB 1.0 導入時（当然、データベースも登場）からエンティティオブジェクト間のリレーションシップを作成していますが、CMP 2.0 以前は関連エンティティのプライマリキーを抽出するため各リレーションのコードを書き、擬似外部キーフィールドに格納しなければなりませんでした。シンプルなリレーションシップはコード化するのも簡単ですが、参照整合性のある複雑なリレーションシップはコード化作業に相当の時間が必要でした。CMP 2.0 があれば、手作業でリレーションをコード化する必要はなくなります。コンテナは、参照整合性のある 1対1、1対多、または多対多のリレーションシップを管理できます。ただし、ローカルインターフェース間でしか定義できないのが唯一の制限事項です。つまり、同じアプリケーションサーバー内でもアプリケーションが異なる 2 つのエンティティではリレーションシップを作成できないのです。

コンテナ管理リレーションの作成には、基本的に 2 種類の方法があります。**cmr-field** 抽象アクセッサを作成し、**ejb-jar.xml** ファイルでリレーションシップを宣言します。次の 2 つの項では、このステップを説明します。

### 30.5.1. CMR-Field 抽象アクセッサ

CMR-Field 抽象アクセッサの署名は **cmp-fields** と同じですが、ただし、値を 1 つ持つリレーションシップは、それに関連するエンティティのローカルインターフェースを返す必要があります。また、複数の値を持つリレーションシップは **java.util.Collection**（または **java.util.Set**）オブジェ

クトしか返すことができません。たとえば、organization と gangster の 1 対多のリレーションシップを宣言するには、**OrganizationBean** クラスで organization から gangster への関係を宣言します。

```
public abstract class OrganizationBean
    implements EntityBean
{
    public abstract Set getMemberGangsters();
    public abstract void setMemberGangsters(Set gangsters);
}
```

また gangster から organization へのリレーションシップを **GangsterBean** クラスで宣言することもできます。

```
public abstract class GangsterBean
    implements EntityBean
{
    public abstract Organization getOrganization();
    public abstract void setOrganization(Organization org);
}
```

各 bean は CMR フィールドで宣言されますが、リレーションシップの 2 つの bean のうち一つにアクセッサセットを設定してください。CMP フィールドは getter と setter いずれのメソッドも必要です。

### 30.5.2. リレーションシップの宣言

**ejb-jar.xml** ファイルでのリレーションシップの宣言は複雑でエラーが発生しやすくなっています。CMR フィールドの配備記述子管理は XDoclet などのツール使用を推奨していますが、記述子について理解しておくことが重要です。以下で organization/gangster リレーションシップの宣言について説明します。

```
<ejb-jar>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Organization-Gangster</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>org-has-gangsters </ejb-
relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>OrganizationEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>memberGangsters</cmr-field-name>
          <cmr-field-type>java.util.Set</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        gangster-belongs-to-org
      </ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <cascade-delete/>
      <relationship-role-source>
        <ejb-name>GangsterEJB</ejb-name>
```

```

        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>organization</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
</relationships>
</ejb-jar>

```

これを見ればわかるように、トップレベルの **relationships** 要素内で **ejb-relation** 要素付きのリレーションシップを宣言しています。リレーションシップには **ejb-relation-name** 要素で名前が付けられます。これは **jbosscmp-jdbc.xml** ファイルで名前によってロールを参照するので重要です。各 **ejb-relation** には2つの **ejb-relationship-role** 要素があります（リレーションシップの各サイドに一つずつ）。**ejb-relationship-role** タグは以下のようになります。

- **ejb-relationshiprole-name**: ロールを識別し、**jbosscmp-jdbc.xml** ファイルをマッピングするデータベースとマッチさせるのに使用する任意の要素です。リレーションシップの各サイドに対するロール名はそれぞれ異なるものにしてください。
- **multiplicity**: これはリレーションシップのこちら側の多重度を示します。**One** または **Many** が値です。この例の場合、1つの **organization** から複数の **gangster** への関係なので、**organization** の多重度は **One** で、**gangster** は **Many** になります。XML 要素と同様、大文字小文字が区別されますので注意してください。
- **cascade-delete**: この任意の要素が有効な場合、親エンティティが削除されると、子エンティティも削除されます。カスケードの削除は、リレーションシップの他サイドの多重度が1であるロールにのみ有効です。デフォルトでは、カスケードの削除はオフに設定されています。
- **relationship-role-source**
  - **ejb-name**: 必須の要素で、ロールを持つエンティティの名称を付与します。
- **cmr-field**
  - **cmr-field-name**: エンティティの CMR フィールドの名前がある場合は、その名称のことです。
  - **cmr-field-type**: フィールドがコレクションタイプであれば、これは CMR フィールドのタイプです。**java.util.Collection** または **java.util.Set** になります。

CMR フィールドの抽象アクセサを追加し、リレーションシップを宣言すると、リレーションシップが機能するようになります。次のセクションでは、リレーションシップのデータベースマッピングについて解説します。

### 30.5.3. 関係マッピング

リレーションシップは外部キー、または別の関係テーブルを使ってマッピングできます。1対1、および1対多のリレーションシップはデフォルトで外部キーマッピングスタイルを使用し、多対多のリレーションシップではリレーションテーブルマッピングスタイルのみを使用します。リレーションシップのマッピングは **ejb-relation** 要素を使い **jbosscmp-jdbc.xml** 記述子の **relationships** セクションで宣言します。リレーションシップは **ejb-jar.xml** ファイルの **ejb-relation-name** で識別されます。**jbosscmp-jdbc.xml** の **ejb-relation** 要素のコンテンツモデルを [図30.7「jbosscmp-jdbc.xml ejb-relation 要素のコンテンツモデル」](#)

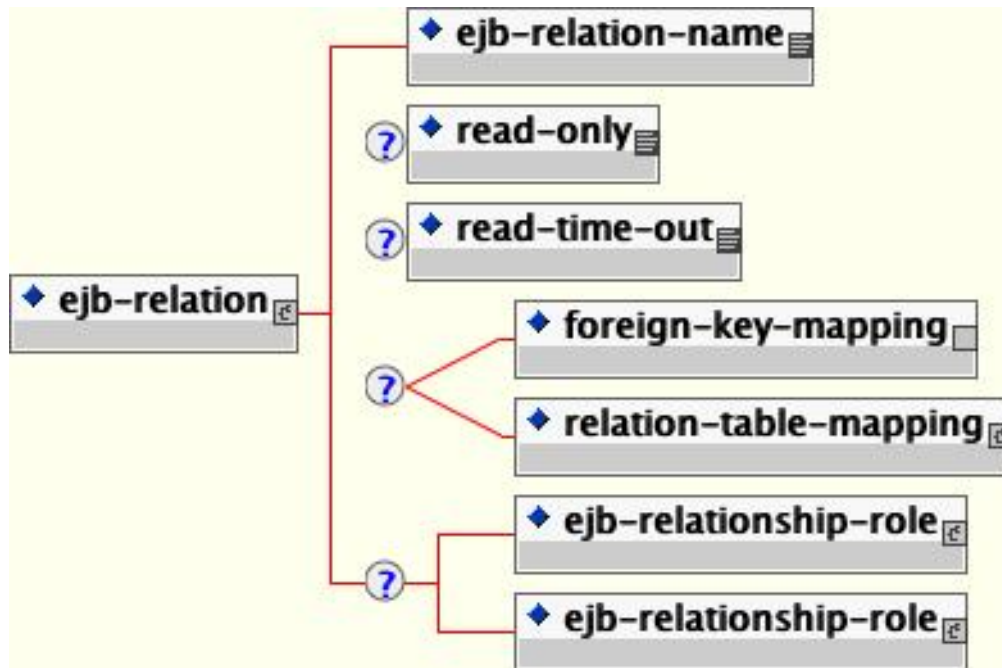


図30.7 jbosscmp-jdbc.xml ejb-relation 要素のコンテンツモデル

**Organization-Gangster** リレーションシップのリレーションシップマッピング宣言の基本テンプレートは次のようになります。

```

<jbosscmp-jdbc>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Organization-Gangster</ejb-relation-name>
      <foreign-key-mapping/>
      <ejb-relationship-role>
        <ejb-relationship-role-name>org-has-gangsters</ejb-
relationship-role-name>
        <key-fields>
          <key-field>
            <field-name>name</field-name>
            <column-name>organization</column-name>
          </key-field>
        </key-fields>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>gangster-belongs-to-org</ejb-
relationship-role-name>
        <key-fields/>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</jbosscmp-jdbc>
  
```

マッピングされるリレーションシップの **ejb-relation-name** を宣言すれば、リレーションシップは **read-only** と **read-time-out** 要素を使用して読み取り専用として宣言することができます。これはエンティティ要素の相手側と同じセマンティクスを持ちます。

**ejb-relation** 要素は **foreign-key-mapping** 要素、または **relation-table-mapping** 要素を含む必要があります。詳しくは「外部キーマッピング」および「関係テーブルのマッピング」に説明します。次のセクションに説明するように、このエレメントは **ejb-relationship-role** 要素のペアも

含みます。

### 30.5.3.1. リレーションシップロールマッピング

2つの **ejb-relationship-role** 要素はそれぞれリレーションシップのエンティティ特有のマッピング情報を含みます。**ejb-relationship-role** 要素のコンテンツモデルを 図30.8 「jbosscomp-jdbc **ejb-relationship-role** 要素のコンテンツモデル」 に示します。

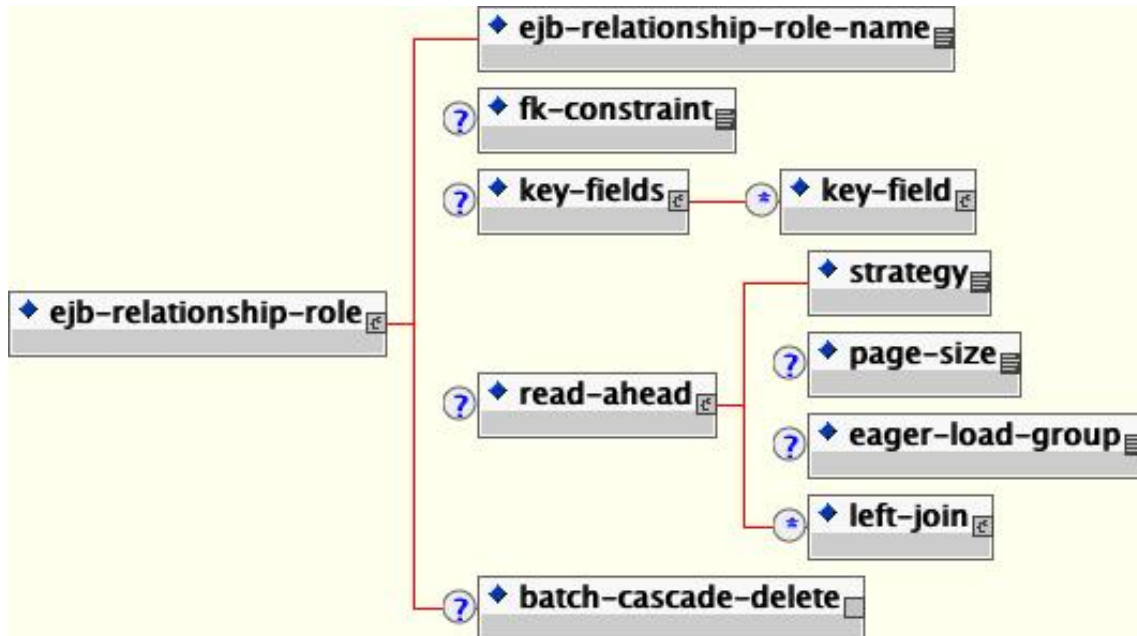


図30.8 jbosscomp-jdbc **ejb-relationship-role** 要素のコンテンツモデル

主な要素の詳細は次の通りになります。

- **ejb-relationship-role-name**: 必須であるこの要素は、この設定が適用されるロール名を付与します。名称は **ejb-jar.xml** ファイルでこのリレーションシップに対して宣言されたロールのうち一つの名称と一致させる必要があります。
- **fk-constraint**: この任意の要素は true/false の値で、リレーションシップのこちら側のテーブルに対して外部キー制約を追加するかどうかを示すものです。デプロイメント実行中に JBoss がプライマリテーブルとこのプライマリテーブルに関連するテーブルを両方作成した場合、JBoss は制約のみを追加生成します。
- **key-fields**: 現行エンティティのプライマリキーフィールドのマッピングについて、関係テーブル、または関連するオブジェクトのいずれにマッピングするかを指定します。**key-fields** 要素は、現在のエンティティのプライマリキーフィールドごとに **key-fields** 要素を1つ含む必要があります。リレーションのこちら側に外部キーマッピングが必要なければ、**key-fields** 要素を空にしておくこともできます。この例は、1対多のリレーションシップの多サイドになります。この要素の詳細については以下に説明します。
- **read-ahead**: この任意の要素は、リレーションシップのキャッシュを制御します。このオプションについては、「[リレーションシップ](#)」で説明します。
- **batch-cascade-delete**: このリレーションシップにおけるカスケード削除は、SQL ステートメント1つで実行しなければならないことを示します。この場合、このリレーションシップは **ejb-jar.xml** で **batch-delete** としてマークしておく必要があります。

上記で述べたように、**key-fields** 要素は、現行のエンティティのプライマリキーフィールドごとに **key-fields** 要素を1つ含みます。**key-field** 要素は、エンティティの **cmp-field** 要素と同じ構文

を使います。ただし、**key-field** は **not-null** オプションをサポートしません。**relation-table** のキーフィールドはテーブルのプライマリキーなので、自動的に not null になります。一方で、外部キーフィールドはデフォルトで null 可能にしておく必要があります。CMP 仕様は **ejbPostCreate** メソッドによってデータベースに挿入し、**ejbPostCreate** による CMR の変更箇所をピックアップするため、更新する必要があります。EJB 仕様では **ejbPostCreate** まではリレーションシップを修正できないため、外部キーは最初は null に設定されます。削除の場合にも同様の問題が発生します。この挿入動作は **jboss.xmlinsert-after-ejb-post-create** コンテナ設定フラグを使用して変更することもできます。デフォルトで **insert-after-ejb-post-create** を使用する新しい bean 設定の作成を以下の例に示します。

```
<jboss>
  <!-- ... -->
  <container-configurations>
    <container-configuration extends="Standard CMP 2.x EntityBean">
      <container-name>INSERT after ejbPostCreate
Container</container-name>
      <insert-after-ejb-post-create>true</insert-after-ejb-post-
create>
    </container-configuration>
  </container-configurations>
</jboss>
```

non-null の外部キーを操作するには、non-null の CMP フィールドに外部キー要素をマッピングする方法もあります。この場合、関連する CMP フィールドセッターを使用して **ejbCreate** の外部キーフィールドを生成するだけです。

key-fields 要素のコンテンツモデルを 図30.9 「jbosscmp-jdbc key-fields 要素のコンテンツモデル」に示します。

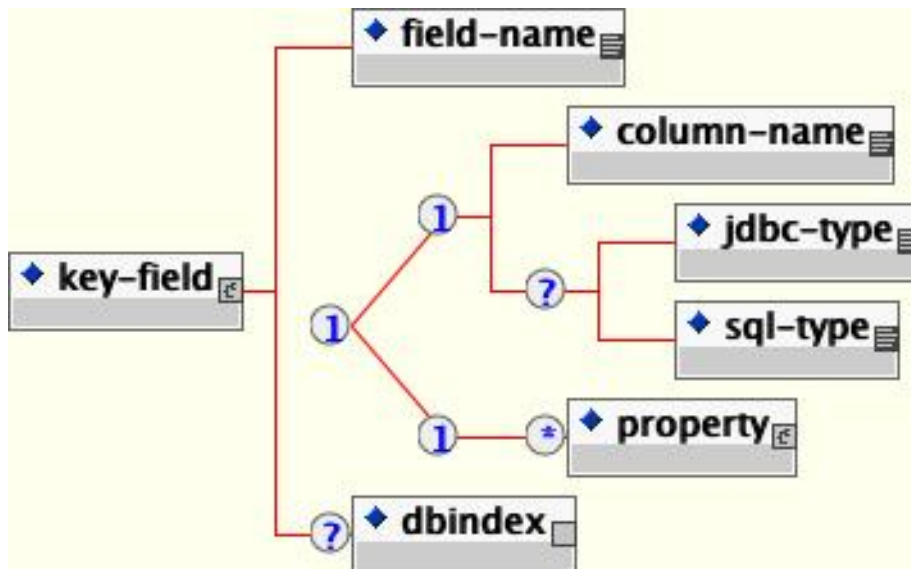


図30.9 jbosscmp-jdbc key-fields 要素のコンテンツモデル

**key-field** 要素に含まれる各種要素の詳細については、以下に説明します。

- **field-name**: この必須要素は、マッピングを適用するフィールドを識別します。この名称は、現行エンティティのプライマリキーフィールドに一致させる必要があります。
- **column-name**: プライマリキーフィールドが格納される列の名称を指定する場合、このエレメントを使用します。このリレーションシップが **foreign-key-mapping** を使用する場合、この列は関連するエンティティのテーブルに追加されます。このリレーションシップが **relation-**



**table-mapping**を使用する場合、この列は **relation-table**に追加されます。この要素はマッピング依存値クラスには対応しません。代わりにプロパティ要素を使用します。

- **jdbc-type**: JDBC **PreparedStatement**内にパラメーターを設定する、または JDBC **ResultSet**からデータをロードする場合に使用される JDBC タイプです。有効なタイプは **java.sql.Types**に定義されます。
- **sql-type**: このフィールドの **create** テーブルステートメントで使用される SQL タイプです。有効なタイプは、データベースベンダーのみにて制限されます。
- **property**: 依存値クラスのプライマリキーフィールドのマッピングを指定する場合、この要素を使用します。
- **dbindex**: この任意フィールドがあると、サーバーはデータベース内の該当カラムにインデックスを作成する必要があることを示します。このインデックス名は **fieldname\_index** になります。

### 30.5.3.2. 外部キーマッピング

外部キーマッピングは、1 対 1、および 1 対多のリレーションシップで最も一般的なマッピングスタイルですが、多対多のリレーションシップには対応していません。外部キーマッピングは **key-mapping** 要素を **ejb-relation** 要素に追加して宣言するだけです。

前項に記載したように、外部キーマッピングがある場合、**ejb-relationship-role** で宣言した **key-fields** は関連エンティティのテーブルに追加されます。**key-fields** 要素が空の場合、外部キーはそのエンティティに対して作成されません。1 対多のリレーションシップでは、多の側（例の **Gangster**）は空の**key-fields** 要素が必要で、1 の側の（例の **Organization**）要素は **key-fields** マッピングが必要です。1 対 1 のリレーションシップでは 1 つ、あるいは両方のロールに外部キーがあります。

外部キーマッピングはリレーションシップの方向に従属していません。つまり、1 対 1 の両方向のリレーションシップ（一方のみにアクセッサーがある）では、1 つあるいは両方のロールに外部キーがあるのです。**Organization-Gangster** リレーションシップの完全な外部キーマッピングを外部キー要素を太字で強調表示して以下に示します。

```
<jbosscmp-jdbc>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Organization-Gangster</ejb-relation-name>
      <foreign-key-mapping/>
      <ejb-relationship-role>
        <ejb-relationship-role-name>org-has-gangsters</ejb-
relationship-role-name>
        <key-fields> <key-field> <field-name>name</field-name>
<column-name>organization</column-name> </key-field> </key-fields>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>gangster-belongs-to-org</ejb-
relationship-role-name>
        <key-fields/>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</jbosscmp-jdbc>
```



### 30.5.3.3. 関係テーブルのマッピング

関係テーブルマッピングは、1 対 1、および 1 対多のリレーションシップでは一般的ではありませんが、多対多のリレーションシップに対応する唯一のマッピングスタイルです。関係テーブルは **relation-table-mapping** 要素を使用して定義します。コンテンツモデルを以下に示します。

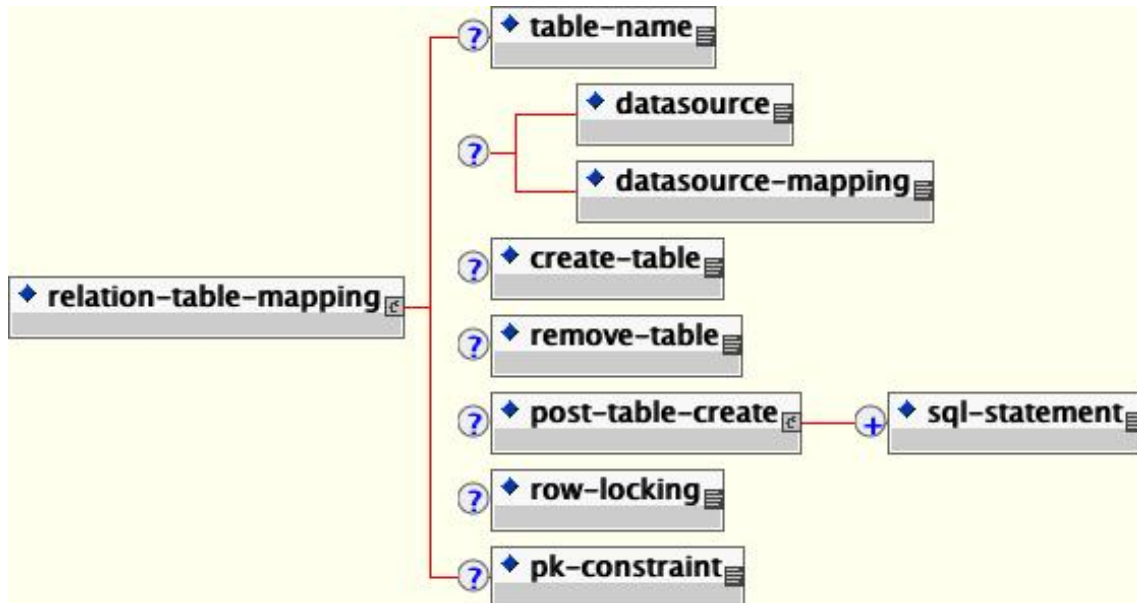


図30.10 jbosscmp-jdbc relation-table-mapping 要素のコンテンツモデル

**Gangster-Job** リレーションシップの relation-table-mapping が以下に示していますが、テーブルマッピング要素を太字で強調表示しています。

#### 例30.1 jbosscmp-jdbc.xml Relation-table マッピング

```

<jbosscmp-jdbc>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Gangster-Jobs</ejb-relation-name>
      <relation-table-mapping>
        <table-name>gangster_job</table-name>
      </relation-table-mapping>
      <ejb-relationship-role>
        <ejb-relationship-role-name>gangster-has-jobs</ejb-
relationship-role-name>
        <key-fields>
          <key-field>
            <field-name>gangsterId</field-name>
            <column-name>gangster</column-name>
          </key-field>
        </key-fields>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>job-has-gangsters</ejb-
relationship-role-name>
        <key-fields>
          <key-field>
            <field-name>name</field-name>
            <column-name>job</column-name>
          </key-field>
        </key-fields>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</jbosscmp-jdbc>
  
```

```

        </key-fields>
    </ejb-relationship-role>
</ejb-relation>
</relationships>
</jbossCMP-jdbc>

```

**relation-table-mapping** 要素は **entity** 要素で使用可能なオプションのサブセットを含みます。この要素に関する詳細な説明をここで繰り返します。

- **table-name**: この任意の要素は、このリレーションシップのデータを格納するテーブル名を付与します。デフォルトのテーブル名は、エンティティと **cmr-field** 名に基づいて付与されます。
- **datasource**: この任意の要素は、データソースのルックアップに使用される **jndi-name** になります。データベース接続はすべてデータソースから取得されます。エンティティにさまざまなデータソースを持たせると、finder および **ejbSelect** でクエリできるドメインを大幅に制約することになるため推奨しません。
- **datasourcemapping**: この任意の要素では、使用する **type-mapping** の名称を指定することができます。
- **create-table**: この任意の要素が true の場合、JBoss はエンティティのテーブルの作成を試行しなければなりません。アプリケーションをデプロイすると、JBoss はテーブル作成の前にすでにテーブルが存在していないか確認します。テーブルが見つかった場合、ログに記録されるだけでテーブルは作成されませんこのオプションは、開発の初期段階でテーブル構造が頻繁に変わる場合に非常に役立ちます。
- **post-table-create**: この任意要素はデータベーステーブルの作成直後に実行されるべき任意の SQL ステートメントを指定します。このコマンドは **create-table** が true でテーブルが以前に存在しない場合にのみ実行されます。
- **remove-table**: この任意の要素が true の場合、アプリケーションがアンデプロイされると、**relation-table** のドロップを試行します。このオプションは、開発の初期段階でテーブル構造が頻繁に変わる場合に非常に役立ちます。
- **row-locking**: この任意の要素が true の場合、トランザクションでロードされたすべての行を JBoss がロックすることを意味します。ほとんどのデータベースがエンティティのロード時に **SELECT FOR UPDATE** 構文を使用してこれを実装しますが、実際の構文は、このエンティティで 사용되는 **datasource-mapping** 内の **row-locking-template** で決まります。
- **pk-constraint**: この任意の要素は、true であれば、テーブル作成時に JBoss がプライマリキー制約を追加することを意味します。

## 30.6. クエリ

エンティティ beans は 2 種類のクエリに対応します。finders と selects です。finder は bean のクライアントに対するエンティティ bean でクエリを行います。select メソッドは、エンティティ実装に対するプライベートのクエリステートメントを提供するように作られています。定義されたホームインターフェースと同じタイプのエンティティの検索のみに制限される finder とは異なり、select メソッドはどんなタイプのエンティティでも、あるいはエンティティの一つのフィールドだけでも返します。EJB-QL はプラットフォーム固有の方法で finder、および select メソッドを指定するクエリ言語です。

### 30.6.1. Finder と Select 宣言

finder の宣言は CMP 2.0 でも変わりません。finder はエンティティのホームインターフェース（ローカル、またはリモート）で宣言します。ローカルホームインターフェースで定義された finder は `RemoteException` を送出しないように設定されています。以下のコードは **GangsterHome** インターフェースで **findBadDudes\_ejbql** finder を宣言します。ejbql 接尾辞ここでは必要ありません。これは単に、異なるタイプのクエリ仕様を分類するため便宜上使用する命名規則です。

```
public interface GangsterHome
    extends EJBLocalHome
{
    Collection findBadDudes_ejbql(int badness) throws FinderException;
}
```

select メソッドはエンティティ実装クラスで宣言し、CMP と CMR フィールドの抽象アクセッサと同様、パブリックで抽象的であり、**FinderException** を送出する必要があります。以下のコードで select メソッドを宣言します。

```
public abstract class GangsterBean
    implements EntityBean
{
    public abstract Set ejbSelectBoss_ejbql(String name)
        throws FinderException;
}
```

### 30.6.2. EJB-QL 宣言

select または finder メソッド（**findByPrimaryKey** を除く）には **ejb-jar.xml** ファイルで定義した EJB-QL クエリが必要です。EJB-QL クエリは、エンティティ要素に含まれるクエリ要素で宣言します。以下に **findBadDudes\_ejbql** および **ejbSelectBoss\_ejbql** クエリの宣言を示します。

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <query>
        <query-method>
          <method-name>findBadDudes_ejbql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <ejb-ql><![CDATA[
          SELECT OBJECT(g) FROM gangster g WHERE g.badness > ?1
        ]]></ejb-ql>
      </query>
      <query>
        <query-method>
          <method-name>ejbSelectBoss_ejbql</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
        <ejb-ql><![CDATA[
          SELECT DISTINCT underling.organization.theBoss FROM
```

```
gangster underling WHERE underling.name = ?1 OR underling.nickName = ?1
    ]]></ejb-ql>
    </query>
  </entity>
</enterprise-beans>
</ejb-jar>
```

EJB-QL は SQL に似ていますが、驚くような違いがあります。以下に EJB-QL に関する注意事項を紹介します。

- EJB-QL は型付言語で、つまり類似する型の比較しかできません（例：ストリングはストリングとの比較しかできないなど）。
- 同値比較では、変数（一価パス）は左側に記載する必要があります。以下に例を示します。

```
g.hangout.state = 'CA' Legal
'CA' = g.shippingAddress.state NOT Legal
'CA' = 'CA' NOT Legal
(r.amountPaid * .01) > 300 NOT Legal
r.amountPaid > (300 / .01) Legal
```

- パラメーターは `java.sql.PreparedStatement` のようにベース 1 インデックスを使用します。
- パラメーターは比較の右側に必ず記載されます。以下に例を示します。

```
gangster.hangout.state = ?1 Legal
?1 = gangster.hangout.state NOT Legal
```

### 30.6.3. EJB-QL を SQL マッピングにオーバーライド

EJB-QL クエリは `jbosscmp-jdbc.xml` ファイルでオーバーライドできます。finder、または select は EJB-QL 宣言に必要ですが、**ejb-ql** 要素は空のままにしておいて構いません。現在、SQL は JBossQL、DynamicQL、DeclaredSQL、または BMP スタイルカスタムの **ejbFind** メソッドでオーバーライドできます。EJB-QL のオーバーライドは EJB の仕様に対して規格外の拡張子であるため、この拡張子を使用するとアプリケーションのポータビリティ（移植性）が制限されます。BMP カスタムファインダーを除く EJB-QL のオーバーライドはすべて、`jbosscmp-jdbc.xml` ファイルの **query** 要素を使用して宣言されます。コンテンツモデルを [図30.11 「jbosscmp-jdbc query 要素のコンテンツモデル」](#) に示します。

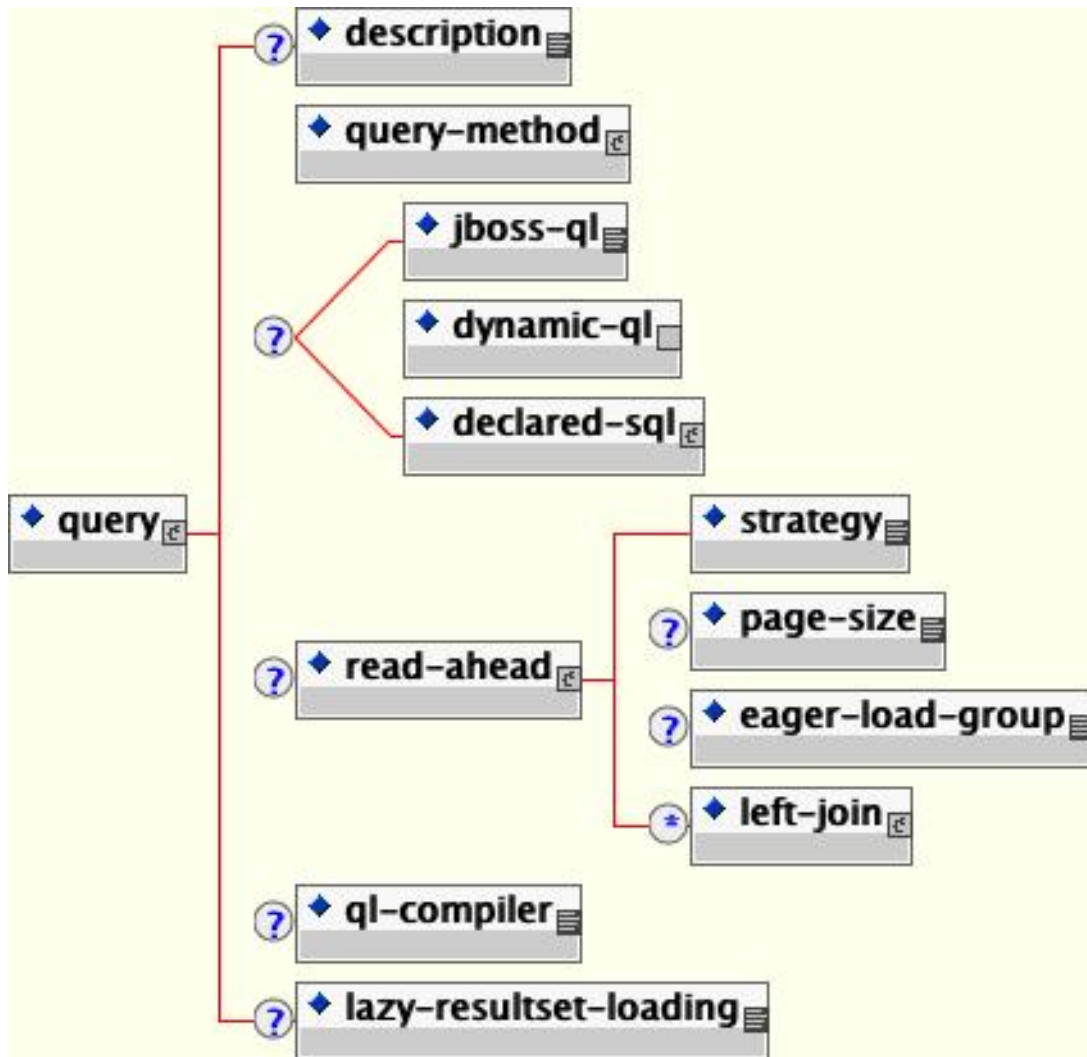


図30.11 jbossCMP-jdbc query 要素のコンテンツツモデル

- **description**: クエリに対するオプションの記述子です。
- **query-method**: 必須の要素で、設定するクエリメソッドを指定します。これは **ejb-jar.xml** ファイルでこのエンティティに対して宣言された **query-method** と一致する必要があります。
- **jboss-ql**: EJB-QL クエリの代わりに使用される JBossQL クエリです。JBossQL については「JBossQL」で説明します。
- **dynamic-ql**: メソッドがダイナミッククエリであり、EJB-QL クエリではないことを示します。ダイナミッククエリについては、「DynamicQL」で説明しています。
- **declared-sql**: このクエリは EJB-QL の代わりに宣言した SQL を使用します。宣言した SQL については、「DeclaredSQL」で説明しています。
- **read-ahead**: この任意の要素により、クエリで参照したエンティティ付きで使用する追加フィールドのロードを最適化します。これについては、「最適化ローディング」で詳しく説明しています。

#### 30.6.4. JBossQL

JBossQL は EJB-QL のスーパーセットであり、EJB-QL の不足機能を補完するため作られています。柔軟性のある構文のほか、新しい関数、キーワード、文節が加わりました。現時点では、JBossQL は **ORDER BY**、**OFFSET** および **OFFSET** 句、**IN** および **LIKE** 演算子のパラメー

ター、**COUNT**、**MAX**、**MIN**、**AVG**、**SUM**、**UCASE** および **LCASE** 関数をサポートしています。また、クエリは select メソッドに対する **SELECT** 句の関数を含みます。

JBossQL は JBossQL クエリを含む **jboss-ql** 要素付きの **jbosscomp-jdbc.xml** ファイルで宣言されます。以下に JBossQL 宣言の例を紹介します。

```
<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>findBadDudes_jbossql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <jboss-ql><![CDATA[ SELECT OBJECT(g) FROM gangster g
WHERE g.badness > ?1 ORDER BY g.badness DESC ]]></jboss-ql>
      </query>
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>
```

これに対応する生成された SQL は単純でわかりやすくなっています。

```
SELECT t0_g.id
FROM gangster t0_g
WHERE t0_g.badness > ?
ORDER BY t0_g.badness DESC
```

JBossQL には **LIMIT** と **OFFSET** 関数を使い、ファインダ結果をブロックで読み出す機能も備わっています。たとえば、実行した大量のジョブを繰り返すには、以下の **findManyJobs\_jbossql** finder を定義します。

```
<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>findManyJobs_jbossql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <jboss-ql><![CDATA[ SELECT OBJECT(j) FROM jobs j OFFSET ?
1 LIMIT ?2 ]]></jboss-ql>
      </query>
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>
```

### 30.6.5. DynamicQL

DynamicQL はランタイム生成と JBossQL クエリの実行を可能にします。DynamicQL クエリメソッドは抽象メソッドで、JBossQL とクエリ引数をパラメーターとして取り込みます。JBoss は JBossQL をコンパイルし、生成済み SQL を実行します。以下は、州一覧の中の任意の州に存在する gangster を選択する JBossQL クエリを生成するものです。

```
public abstract class GangsterBean
    implements EntityBean
{
    public Set ejbHomeSelectInStates(Set states)
        throws FinderException
    {
        // generate JBossQL query
        StringBuffer jbossQl = new StringBuffer();
        jbossQl.append("SELECT OBJECT(g) ");
        jbossQl.append("FROM gangster g ");
        jbossQl.append("WHERE g.hangout.state IN (");

        for (int i = 0; i < states.size(); i++) {
            if (i > 0) {
                jbossQl.append(", ");
            }

            jbossQl.append("?").append(i+1);
        }

        jbossQl.append(") ORDER BY g.name");

        // pack arguments into an Object[]
        Object[] args = states.toArray(new Object[states.size()]);

        // call dynamic-ql query
        return ejbSelectGeneric(jbossQl.toString(), args);
    }
}
```

DynamicQL select メソッドには有効な select メソッド名が複数ありますが、このメソッドではストリングとオブジェクトアレイをパラメーターとして取る必要があります。DynamicQL は **dynamic-ql** 要素付きの **jbosscmp-jdbc.xml** ファイルで宣言します。以下に **ejbSelectGeneric** の宣言を示します。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>ejbSelectGeneric</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.Object[]</method-param>
          </method-params>
        </query-method>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

```

        <dynamic-ql/>
      </query>
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>

```

### 30.6.6. DeclaredSQL

DeclaredSQL は旧 JAWS CMP 1.1 エンジンの finder 宣言をベースにしていますが、CMP 2.0 用にアップデートされています。通常、この宣言を使い、EJB-QL や JBossQL では表示できない **WHERE** 句を使ったクエリを制限しています。declared-sql 要素のコンテンツモデルは [図30.12 「jbosscomp-jdbc declared-sql 要素のコンテンツモデル」](#) に示しています。

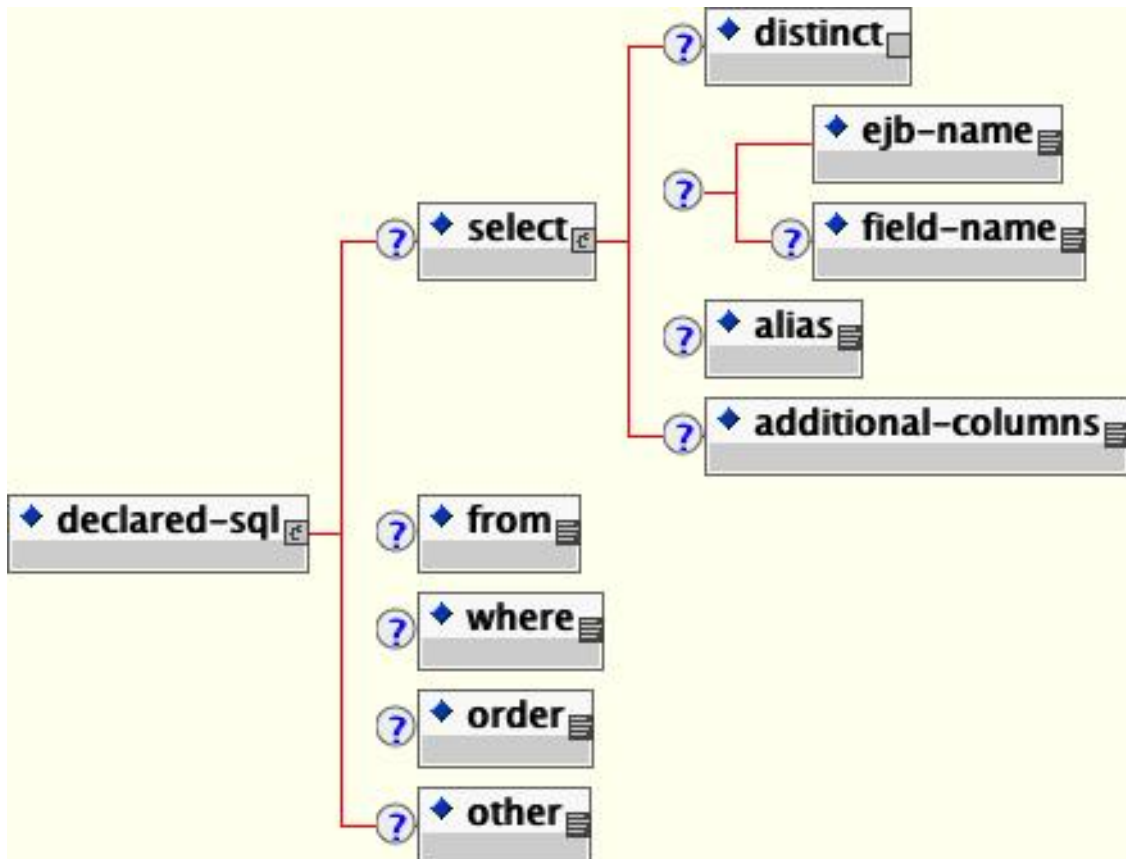


図30.12 jbosscomp-jdbc declared-sql 要素のコンテンツモデル

- **select:** **select** 要素は選択する対象を指定するもので、以下の要素で構成されています。
  - **distinct:** この空の要素がある場合、JBoss は **DISTINCT** キーワードを追加して **SELECT** 句を生成します。デフォルトでは、メソッドが `java.util.Set` を返した場合、**DISTINCT** を使用するように設定されています。
  - **ejb-name:** 選択するエンティティの **ejb-name** を示します。この項目は、クエリーが `select` メソッドの場合のみ必須です。
  - **field-name:** 指定したエンティティから選択される CMP フィールドの名称です。デフォルトではエンティティ全体を選択するよう設定されています。
  - **alias:** メインの `select` テーブルに使用するエイリアスを指定します。デフォルトは **ejb-name** を使用します。



- **additional-columns**: finder で任意のカラムの命令を実行、あるいは select の集約関数に対応するために選択されるカラムを宣言します。
- **from: from** 要素は、生成済みの **FROM** 句に追加する SQL を宣言します。
- **where: where** 要素は クエリー用の **WHERE** 句を宣言します。
- **order: order** 要素はクエリー用の **ORDER** 句を宣言します。
- **other: other** 要素はクエリーの最後に追加する SQL を宣言します。

DeclaredSQL 宣言の例を以下に示します。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>findBadDudes_declaredsql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <declared-sql>
          <where><![CDATA[ badness > {0} ]]></where>
          <order><![CDATA[ badness DESC ]]></order>
        </declared-sql>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

生成される SQL は次のようになります。

```
SELECT id
FROM gangster
WHERE badness > ?
ORDER BY badness DESC
```

ご覧のように、JBoss はこのエンティティにプライマリーキーを選択するのに必要な **SELECT** および **FROM** 句を生成します。ご希望であれば、自動生成された **FROM** 句の末尾に追加される **FROM** 句を指定することができます。以下に **FROM** 句を追加した DeclaredSQL 宣言の例を示します。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>ejbSelectBoss_declaredsql</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
```

```

        <declared-sql>
        <select>
            <distinct/>
            <ejb-name>GangsterEJB</ejb-name>
            <alias>boss</alias>
        </select>
        <from><![CDATA[, gangster g, organization o]]></from>
        <where><![CDATA[
            (LCASE(g.name) = {0} OR LCASE(g.nick_name) = {0})
AND
            g.organization = o.name AND o.the_boss = boss.id
        ]]></where>
        </declared-sql>
    </query>
</entity>
</enterprise-beans>
</jbosscmp-jdbc>

```

生成される SQL は次のようになります。

```

SELECT DISTINCT boss.id
FROM gangster boss, gangster g, organization o
WHERE (LCASE(g.name) = ? OR LCASE(g.nick_name) = ?) AND
      g.organization = o.name AND o.the_boss = boss.id

```

**FROM** 句はコンマで始まりますので注意してください。これは、コンテナが生成された **FROM** 句の末尾に宣言した **FROM** 句を追加するためです。**FROM** 句は SQL **JOIN** ステートメントで始まる場合もあります。select メソッドなので、選択されるエンティティを宣言する **select** 要素が必要です。エイリアスもクエリに対して宣言されますので、注意してください。エイリアスが宣言されない場合、**table-name** をエイリアスとして使用します。つまり、**SELECT** 句が **table\_name.field\_name** スタイルのカラム宣言になります。データベースベンダーによってはこの構文をサポートしていませんので、エイリアスの宣言を推奨します。任意で **distinct** 要素により **SELECT** 句は **SELECT DISTINCT** 宣言を使用します。DeclaredSQL 宣言は CMP フィールドを選択する select メソッドでも使用することができます。

ここで **Organization** が運営している場所の郵便番号を返す select をオーバーライドする場合の例を以下に示します。

```

<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>OrganizationEJB</ejb-name>
            <query>
                <query-method>
                    <method-
name>ejbSelectOperatingZipCodes_declaredsql</method-name>
                    <method-params>
                        <method-param>java.lang.String</method-param>
                    </method-params>
                </query-method>
                <declared-sql> <select> <distinct/> <ejb-
name>LocationEJB</ejb-name> <field-name>zipCode</field-name>
<alias>hangout</alias> </select> <from><![CDATA[ , organization o,
gangster g ]]></from> <where><![CDATA[ LCASE(o.name) = {0} AND o.name =
g.organization AND g.hangout = hangout.id ]]></where> <order><![CDATA[

```

```

hangout.zip ]]></order> </declared-sql>
    </query>
  </entity>
</enterprise-beans>
</jbosscmp-jdbc>

```

対応する SQL は次のようになります。

```

SELECT DISTINCT hangout.zip
  FROM location hangout, organization o, gangster g
 WHERE LCASE(o.name) = ? AND o.name = g.organization AND g.hangout =
 hangout.id
        ORDER BY hangout.zip

```

### 30.6.6.1. パラメーター

DeclaredSQL はまったく新しいパラメーター処理システムを採用し、エンティティと DVC パラメーターをサポートします。パラメーターは中括弧に入れ、1 ベースの EJB-QL パラメーターとは別の 0 ベースのインデックスを使用します。パラメーターには以下の 3 種類のカテゴリがあります。Simple、DVC と entity です。

- **simple:** シンプルパラメーターは既知（マッピング済みの）DVC やエンティティ以外のあらゆるタイプを指します。シンプルパラメーターは **{0}** などの引数のみを含みます。シンプルパラメーターが設定されている場合、パラメーターを設定する JDBC タイプは、エンティティに対する **datasourcemapping** で決定します。未知の DVC はシリアル化され、パラメーターとして設定されます。WHERE 句では BLOB 値の使用をサポートしていないデータベースがほとんどですので、注意してください。
- **DVC:** DVC パラメーターは既知の（マッピング済み）DVC になります。DVC パラメーターはシンプルプロパティ（別の DVC ではないもの）を逆参照する必要があります。たとえば、タイプ **ContactInfo** の CVS プロパティがある場合、有効なパラメーターの宣言は **{0.email}** および **{0.cell.areaCode}** になり、**{0.cell}** にはなりません。パラメーターの設定に使用する JDBC タイプは、プロパティのクラスタイプとエンティティの **datasourcemapping** により決まります。パラメーターの設定に使用する JDBC タイプは、**dependent-value-class** 要素でこのプロパティに対して宣言した JDBC タイプになります。
- **entity:** エンティティパラメーターは、アプリケーションにある任意のエンティティになります。エンティティパラメーターは、DVC プライマリキーフィールドのシンプルプライマリキーフィールド、またはシンプルプロパティを逆参照する必要があります。たとえば、タイプ **Gangster** のパラメーターがある場合、有効なパラメーター宣言は **{0.gangsterId}** になります。タイプ **ContactInfo** のプライマリキーフィールドの名前が付いたエンティティがある場合、**valid parameter** 宣言は **{0.info.cell.areaCode}** になります。エンティティのプライマリキーの構成要素であるフィールドのみ逆参照できます（今後のバージョンではこの制限がなくなる予定です）。パラメーターの設定に使用する JDBC タイプは、エンティティ宣言でこのフィールドに対して宣言した JDBC タイプになります。

### 30.6.7. EJBQL 2.1 と SQL92 のクエリ

デフォルトのクエリーコンパイラは、EJB-QL 2.1 または the SQL92 規格に完全対応していません。このうちいずれかが必要な場合、クエリコンパイラを交換してください。デフォルトのコンパイラは **standardjbosscmp-jdbc.xml** に指定しています。

```

<defaults>
  ...

```

```
<ql-compiler>org.jboss.ejb.plugins.cmp.jdbc.JDBCEJBQLCompiler</ql-compiler>
...
</defaults>
```

SQL92 コンパイラーを使用するには、単に**ql-compiler** 要素で SQL92 コンパイラーを指定するだけです。

```
<defaults>
...
<ql-compiler>org.jboss.ejb.plugins.cmp.jdbc.EJBQLToSQL92Compiler</ql-compiler>
...
</defaults>
```

これにより、システム全体の bean すべてに対し、クエリコンパイラーが変更されます。また、各要素に対するクエリコンパイラーを **jbosscmp-jdbc.xml** で指定することもできます。初期のクエリの一つを使用した例を以下に紹介します。

```
<query>
  <query-method>
    <method-name>findBadDudes_ejbql</method-name>
    <method-params>
      <method-param>int</method-param>
    </method-params>
  </query-method>
  <ejb-ql><![CDATA[
    SELECT OBJECT(g)
    FROM gangster g
    WHERE g.badness > ?1]]>
  </ejb-ql>
  <ql-compiler>org.jboss.ejb.plugins.cmp.jdbc.EJBQLToSQL92Compiler</ql-compiler>
</query>
```

SQL92 クエリコンパイラーの制限事項で重要な点は、使用する **read-ahead** ストラテジに関わらず、エンティティの全フィールドを選択することです。たとえば、クエリが**on-loadread-ahead** ストラテジで設定されている場合、最初のクエリには全フィールドが含まれ、プライマリキーフィールドだけが **ResultSet** から読み込まれます。次に、他のフィールドは先読みキャッシュにロードされます。デフォルトのロードグループ\* を備えた **on-findread-ahead** は 期待通りに動作します。

### 30.6.8. BMP カスタムファインダー

JBoss は bean 管理による永続カスタムファインダーもサポートします。カスタムの finder メソッドがホーム、またはローカルホームインターフェースで宣言した finder と一致する場合、JBoss は **ejb-jar.xml** または **jbosscmp-jdbc.xml** ファイルで宣言したその他の実装に優先して、このカスタムファインダーを呼び出します。以下のシンプルな例に、プライマリキーの集まりによるエンティティを示します。

```
public abstract class GangsterBean
    implements EntityBean
{
    public Collection ejbFindByPrimaryKeys(Collection keys)
    {
```

```

        return keys;
    }
}

```

データベースに接続せずにプライマリーキーを実際のエンティティオブジェクトに変換できるため、非常に便利な finder です。問題なのは、データベースに存在しないプライマリーキーでエンティティオブジェクトを作成できるという点です。無効なエンティティでメソッドが呼び出された場合、`NoSuchEntityException` が送出されます。生成されたエンティティ bean は finder を実装している EJB 仕様に反するのも問題点の一つです。JBoss EJB verifier は `StrictVerifier` 属性が `false` に設定されない限り、このようなエンティティをデプロイできません。

## 30.7. 最適化ローディング

最適化ローディングは、数が最も少ないクエリーにおける処理を完了させるのにデータのロードを必要最小限に抑えることが目的です。JBoss の調整は、ロードプロセスに関する知識によります。このセクションでは、JBoss ロードプロセス内部とその設定について説明します。ロードプロセスを調整するには、ロードシステムに関する深い知識が必要であるため本章を繰り返し読むことをおすすめします。

### 30.7.1. ローディングのシナリオ

使用シナリオを参照するのが、ローディングについて調べる最も簡単な方法です。最も一般的なシナリオはエンティティのコレクションを検索し、操作を実行する検索結果を反復することです。以下の例に `gangsters` をすべて含む html テーブルの生成を示します。

```

public String createGangsterHtmlTable_none()
    throws FinderException
{
    StringBuffer table = new StringBuffer();
    table.append("<table>");

    Collection gangsters = gangsterHome.findAll_none();
    for (Iterator iter = gangsters.iterator(); iter.hasNext();) {
        Gangster gangster = (Gangster) iter.next();
        table.append("<tr>");
        table.append("<td>").append(gangster.getName());
        table.append("</td>");
        table.append("<td>").append(gangster.getNickName());
        table.append("</td>");
        table.append("<td>").append(gangster.getBadness());
        table.append("</td>");
        table.append("</tr>");
    }

    return table.toString();
}

```

このコードが1つのトランザクションで呼び出され、ローディングの最適化がすべて無効になっていると仮定します。`findAll_none` の呼び出しで、JBoss は以下のクエリを実行します。

```

SELECT t0_g.id
FROM gangster t0_g
ORDER BY t0_g.id ASC

```

サンプルデータベースの 8 つの gangster それぞれにアクセスするため、JBoss は以下の 8 つのクエリを実行します。

```
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=0)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=1)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=2)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=3)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=4)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=5)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=6)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=7)
```

このシナリオには 2 つ問題があります。1 つは、JBoss が **findAll** に対して 1 つのクエリを、また、見つかった各要素にアクセスするのに 1 つのクエリを実行するため、過剰な数のクエリが実行されます。この動作の原因は、JBoss コンテナ内でのクエリ結果の処理方法と関係があります。クエリが実行されると、選択された実際のエンティティ bean が返されるように見えますが、JBoss は実際は一致するエンティティのプライマリキーを戻すだけで、そのエンティティに対してメソッドが呼び出されるまで、エンティティをロードしません。これは **n+1** 問題として知られ、以降のセクションで説明する read-ahead ストラテジで対処します。

2 つ目の問題は、未使用フィールドの値が不必要にロードされることです。JBoss は **hangout** および **organization** フィールドをロードしますが、これはアクセスされることはありません。(分かりやすくするために、複雑な **contactInfo** フィールドを無効にしています)。

次の表でクエリを実行したものを示しています。

表30.1 非最適化クエリ実行

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia

id	name	nick_name	badness	hangout	organization
7	Corleone	Godfather	6	7	Mafia

### 30.7.2. Load Groups

ローディングシステムの設定および最適化は、エンティティにて名前付きのロードグループの宣言から始まります。ロードグループには、1 回の操作でロードされる、外部キー (Organization-Gangster 例の **Gangster** など) を持つ CMR フィールドおよび CMP フィールドの名前が入っています。設定例を次に示します。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <load-groups>
        <load-group>
          <load-group-name>basic</load-group-name>
          <field-name>name</field-name>
          <field-name>nickName</field-name>
          <field-name>badness</field-name>
        </load-group>
        <load-group>
          <load-group-name>contact info</load-group-name>
          <field-name>nickName</field-name>
          <field-name>contactInfo</field-name>
          <field-name>hangout</field-name>
        </load-group>
      </load-groups>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

この例では、**basic** および **contact info** という2つのロードグループが宣言されています。ロードグループは相互排他的である必要はないことに注意してください。たとえば、両方のロードグループに **nickName** フィールドが入っています。JBoss は、宣言されたロードグループに加えて、エンティティ内の外部キーを持つすべての CMR フィールドおよび CMP フィールドを含んでいる \* (スターグループ) というグループを自動的に追加します。

### 30.7.3. Read-ahead

JBoss での最適化ローディングは read-ahead と呼ばれます。これは、ロードするエンティティの行と、その次の数行を読み込む技術のことで、こういった背景から read-ahead (先読み) という表現となっています。JBoss は主に2つのストラテジー (**on-find** および **on-load**) を実装して、前項で挙げたローディングの問題を最適化します。read-ahead 中にロードされた余分なデータは、すぐにメモリ内のエンティティオブジェクトと関連付けられません。なぜなら、JBoss においてエンティティは実際にアクセスされるまで実体化されないからです。その代わり、これはプレロードキャッシュに格納され、エンティティにロードされるまで、あるいはトランザクションの終わりが現れるまで、そこにとどまります。以降のセクションでは、read-ahead ストラテジーについて説明します。

#### 30.7.3.1. on-find

**on-find** ストラテジーは、クエリが呼び出されると、追加のカラムを読み込みます。クエリが **on-find** 最適化されている場合、クエリが実行されると JBoss は次のクエリを実行します。

```
SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
ORDER BY t0_g.id ASC
```

必要なデータはすべてプレロードキャッシュ内に入るので、クエリ結果を反復している間、追加のクエリを実行する必要はありません。このストラテジーは、少量のデータを返すクエリに対しては有効ですが、メモリに大量の結果セットをロードしようとしている場合には非常に効率的ではありません。次のテーブルは、このクエリーの実行を示したものです。

表30.2 on-find 最適化クエリ実行

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

クエリーの **read-ahead** ストラテジーと **load-group** は、**query** 要素に定義されます。**read-ahead** ストラテジーが **query** 要素に宣言されていない場合は、**entity** 要素または **defaults** 要素に宣言されたストラテジーが使用されます。**on-find** 設定は次のとおりです。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!--...-->
      <query>
        <query-method>
          <method-name>findAll_onfind</method-name>
          <method-params/>
        </query-method>
        <jboss-ql><![CDATA[
          SELECT OBJECT(g)
          FROM gangster g
          ORDER BY g.gangsterId
        ]]></jboss-ql>
```



```

        <read-ahead>
            <strategy>on-find</strategy>
            <page-size>4</page-size>
            <eager-load-group>basic</eager-load-group>
        </read-ahead>
    </query>
</entity>
</enterprise-beans>
</jboss-cmp-jdbc>

```

**on-find** ストラテジーには、選択されたすべてのエンティティについて追加データをロードしなければならないという問題があります。Web アプリケーションでは通常、ページ上で固定された数の結果だけがレンダリングされます。プレロードされたデータはトランザクションの長さだけ有効であり、1 トランザクションは Web HTTP の 1 ヒットに制限されるので、プレロードされたデータの大半は使用されません。次項で説明する **on-load** ストラテジーでは、この問題の影響を受けません。

#### 30.7.3.1.1. Left join read ahead

Left join read ahead は、**on-findread-ahead** ストラテジーが強化されたものです。1 つの SQL クエリで、ベースインスタンスからのフィールドだけでなく、CMR ナビゲーションによりベースインスタンスから到達できる関連インスタンスもプレロードすることができます。CMR ナビゲーションの深さに対する制限はありません。また、ナビゲーションおよびリレーションシップタイプのマッピングで使用する CMR フィールドのカーディナリティに対する制限もありません。すなわち、外部キーと関連テーブルマッピングスタイルの両方がサポートされます。それでは、例をいくつか見ていきましょう。エンティティとリレーションシップの宣言を以下に示しています。

#### 30.7.3.1.2. D#findByPrimaryKey

エンティティ **D** があるとします。**findByPrimaryKey** に対して生成される一般的な SQL は、次のようになります。

```
SELECT t0_D.id, t0_D.name FROM D t0_D WHERE t0_D.id=?
```

**findByPrimaryKey** の実行中に、2 つのコレクション値 CMR フィールド **bs** および **cs** もプレロードすることにします。

```

<query>
    <query-method>
        <method-name>findByPrimaryKey</method-name>
        <method-params>
            <method-param>java.lang.Long</method-param>
        </method-params>
    </query-method>
    <jboss-ql><![CDATA[SELECT OBJECT(o) FROM D AS o WHERE o.id = ?1]]>
</jboss-ql>
    <read-ahead>
        <strategy>on-find</strategy>
        <page-size>4</page-size>
        <eager-load-group>basic</eager-load-group>
        <left-join cmr-field="bs" eager-load-group="basic"/>
        <left-join cmr-field="cs" eager-load-group="basic"/>
    </read-ahead>
</query>

```

**left-join** では、一括読み込みを行いたいリレーションを宣言します。生成される SQL は次のようになります。

```
SELECT t0_D.id, t0_D.name,
       t1_D_bs.id, t1_D_bs.name,
       t2_D_cs.id, t2_D_cs.name
FROM D t0_D
     LEFT OUTER JOIN B t1_D_bs ON t0_D.id=t1_D_bs.D_FK
     LEFT OUTER JOIN C t2_D_cs ON t0_D.id=t2_D_cs.D_FK
WHERE t0_D.id=?
```

固有の id を持つ **D** では、それに関連するすべての **B** と **C** をプリロードし、それをデータベースからではなく先読みキャッシュからロードするインスタンスにアクセスすることができます。

### 30.7.3.1.3. D#findAll

同様に、**D** への **findAll** メソッドは **D** 関連をすべて選択しますが、この **findAll** メソッドを最適化することができます。通常の **findAll** クエリは次のようになります。

```
SELECT DISTINCT t0_o.id, t0_o.name FROM D t0_o ORDER BY t0_o.id DESC
```

リレーションをプレロードするには、**left-join** 要素をクエリーに追加するだけで結構です。

```
<query>
  <query-method>
    <method-name>findAll</method-name>
  </query-method>
  <jboss-ql><![CDATA[SELECT DISTINCT OBJECT(o) FROM D AS o ORDER BY o.id
DESC]]></jboss-ql>
  <read-ahead>
    <strategy>on-find</strategy>
    <page-size>4</page-size>
    <eager-load-group>basic</eager-load-group>
    <left-join cmr-field="bs" eager-load-group="basic"/>
    <left-join cmr-field="cs" eager-load-group="basic"/>
  </read-ahead>
</query>
```

生成される SQL は次のとおりです。

```
SELECT DISTINCT t0_o.id, t0_o.name,
               t1_o_bs.id, t1_o_bs.name,
               t2_o_cs.id, t2_o_cs.name
FROM D t0_o
     LEFT OUTER JOIN B t1_o_bs ON t0_o.id=t1_o_bs.D_FK
     LEFT OUTER JOIN C t2_o_cs ON t0_o.id=t2_o_cs.D_FK
ORDER BY t0_o.id DESC
```

これで、このシンプルな **findAll** クエリは、各 **D** オブジェクトに対して、関連する **B** および **C** オブジェクトをプレロードするようになりました。

### 30.7.3.1.4. A#findAll

それでは、より複雑な設定を見てみましょう。ここでは、いくつかのリレーションと共にインスタンス **A** をプレロードしようと思います。

- CMR フィールド **parent** で **A** から到達するその親 (セルフリレーション)
- CMR フィールド **b** で **A** から到達する **B**、および CMR フィールド **c** で **B** から到達する関連の **C**
- 今回は CMR フィールド **b2** で **A** から到達する **B**、およびそれに関連付けられた、CMR フィールド **c** で **B** から到達する **C**

参考までに、標準のクエリーは次のとおりです。

```
SELECT t0_o.id, t0_o.name FROM A t0_o ORDER BY t0_o.id DESC FOR UPDATE
```

次のメタデータは、プレロードプランを記述したものです。

```
<query>
  <query-method>
    <method-name>findAll</method-name>
  </query-method>
  <jboss-ql><![CDATA[SELECT OBJECT(o) FROM A AS o ORDER BY o.id DESC]]>
</jboss-ql>
  <read-ahead>
    <strategy>on-find</strategy>
    <page-size>4</page-size>
    <eager-load-group>basic</eager-load-group>
    <left-join cmr-field="parent" eager-load-group="basic"/>
    <left-join cmr-field="b" eager-load-group="basic">
      <left-join cmr-field="c" eager-load-group="basic"/>
    </left-join>
    <left-join cmr-field="b2" eager-load-group="basic">
      <left-join cmr-field="c" eager-load-group="basic"/>
    </left-join>
  </read-ahead>
</query>
```

生成 SQL は次のようになります。

```
SELECT t0_o.id, t0_o.name,
       t1_o_parent.id, t1_o_parent.name,
       t2_o_b.id, t2_o_b.name,
       t3_o_b_c.id, t3_o_b_c.name,
       t4_o_b2.id, t4_o_b2.name,
       t5_o_b2_c.id, t5_o_b2_c.name
FROM A t0_o
  LEFT OUTER JOIN A t1_o_parent ON t0_o.PARENT=t1_o_parent.id
  LEFT OUTER JOIN B t2_o_b ON t0_o.B_FK=t2_o_b.id
  LEFT OUTER JOIN C t3_o_b_c ON t2_o_b.C_FK=t3_o_b_c.id
  LEFT OUTER JOIN B t4_o_b2 ON t0_o.B2_FK=t4_o_b2.id
  LEFT OUTER JOIN C t5_o_b2_c ON t4_o_b2.C_FK=t5_o_b2_c.id
ORDER BY t0_o.id DESC FOR UPDATE
```

この設定では、別のデータベースをロードすることなく、見つかった **A** のインスタンスから CMR をナビゲートすることができます。

### 30.7.3.1.5. A#findMeParentGrandParent

次に別のセルフリレーションの例を示します。あるインスタンス、その親、祖父母および曾祖父母を 1 つのクエリでプレロードするメソッドを書くことにします。これを行うのに、ネストされた **left-join** 宣言を使用します。

```
<query>
  <query-method>
    <method-name>findMeParentGrandParent</method-name>
    <method-params>
      <method-param>java.lang.Long</method-param>
    </method-params>
  </query-method>
  <jboss-ql><![CDATA[SELECT OBJECT(o) FROM A AS o WHERE o.id = ?1]]>
</jboss-ql>
  <read-ahead>
    <strategy>on-find</strategy>
    <page-size>4</page-size>
    <eager-load-group>*</eager-load-group>
    <left-join cmr-field="parent" eager-load-group="basic">
      <left-join cmr-field="parent" eager-load-group="basic">
        <left-join cmr-field="parent" eager-load-group="basic"/>
      </left-join>
    </left-join>
  </read-ahead>
</query>
```

生成される SQL は次のようになります。

```
SELECT t0_o.id, t0_o.name, t0_o.secondName, t0_o.B_FK, t0_o.B2_FK,
t0_o.PARENT,
      t1_o_parent.id, t1_o_parent.name,
      t2_o_parent_parent.id, t2_o_parent_parent.name,
      t3_o_parent_parent_parent.id, t3_o_parent_parent_parent.name
FROM A t0_o
      LEFT OUTER JOIN A t1_o_parent ON t0_o.PARENT=t1_o_parent.id
      LEFT OUTER JOIN A t2_o_parent_parent ON
t1_o_parent.PARENT=t2_o_parent_parent.id
      LEFT OUTER JOIN A t3_o_parent_parent_parent
      ON t2_o_parent_parent.PARENT=t3_o_parent_parent_parent.id
WHERE (t0_o.id = ?) FOR UPDATE
```

**left-join** メタデータを取り除けば、次のようになります。

```
SELECT t0_o.id, t0_o.name, t0_o.secondName, t0_o.B2_FK, t0_o.PARENT FOR
UPDATE
```

### 30.7.3.2. on-load

**on-load** ストラテジーは、あるエンティティがロードされるときに、いくつかのエンティティの追加データをブロックでロードするもので、要求されたエンティティから開始し、選択された順序で次のいくつかのエンティティをロードします。このストラテジーは、find または select の結果が順方向にアクセスされるという理論に基づきます。クエリーが実行されると、JBoss は、見つかったエンティティの順番をリストキャッシュに格納します。後で、エンティティのいずれかがロードされるとき、JBoss は

このリストを使用して、ロードするエンティティのブロックを決定します。キャッシュに格納するリスト数は、エンティティの **list-cachemax** 要素で指定します。この戦略は、**on-find** ストラテジーでデータがロードされないという問題があるときにも使用されます。

**on-find** ストラテジーと同様に、**on-load** は **read-ahead** 要素で宣言します。この例の **on-load** 設定を次に示します。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <query>
        <query-method>
          <method-name>findAll_onload</method-name>
          <method-params/>
        </query-method>
        <jboss-ql><![CDATA[
          SELECT OBJECT(g)
          FROM gangster g
          ORDER BY g.gangsterId
        ]]></jboss-ql>
        <read-ahead>
          <strategy>on-load</strategy>
          <page-size>4</page-size>
          <eager-load-group>basic</eager-load-group>
        </read-ahead>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

このストラテジーによって、finder メソッドのクエリーは変更されません。

```
SELECT t0_g.id
FROM gangster t0_g
ORDER BY t0_g.id ASC
```

しかし、結果セットを反復するので、データのロード方法は異なります。ページサイズが 4 の場合、JBoss は、エンティティの **name**、**nickName**、**badness** フィールドをロードするのに、次の 2 つのクエリーを実行するだけでよくなります。

```
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=4) OR (id=5) OR (id=6) OR (id=7)
```

次の表は、これらのクエリーの実行を示したものです。

表30.3 on-load 最適化クエリー実行

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

### 30.7.3.3. none

**none** ストラテジーは、実は反ストラテジーです。このストラテジーによって、システムはデフォルトの lazy-load コードに戻ります。具体的には、どのデータも先読みせず、見つかったエンティティの順序を記憶しません。この結果、この章の先頭で紹介したクエリーとパフォーマンスになります。none ストラテジーは、read-ahead 要素で宣言します。**read-ahead** 要素に **page-size** 要素または **eager-load-group** が含まれる場合、それは無視されます。次の例で none ストラテジーが宣言されています。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <query>
        <query-method>
          <method-name>findAll_none</method-name>
          <method-params/>
        </query-method>
        <jboss-ql><![CDATA[
          SELECT OBJECT(g)
          FROM gangster g
          ORDER BY g.gangsterId
        ]]></jboss-ql>
        <read-ahead>
          <strategy>none</strategy>
        </read-ahead>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

## 30.8. ロードイングプロセス

前項では、いくつかの手順で「エンティティがロードされる場合」というフレーズを使用しています。エンティティを指定した commit オプションおよびトランザクションの現在のステータスにより、エンティティがロードされる時が決まるため、これについては意図的あいまいにしておきました。次項では、commit オプションとロードイングプロセスについて説明します。

### 30.8.1. コミットオプション

ロードイングプロセスの中核を成すのは commit オプションであり、これはエンティティのデータの有効期限を制御します。JBoss では、**A**、**B**、**C**、**D**という 4 つの commit オプションをサポートしています。最初の 3 つは Enterprise JavaBeans 仕様で説明されていますが、最後のものは JBoss に固有のもので、各 commit オプションの詳細は次のとおりです。

- **A**: JBoss は、それがデータベースの唯一のユーザーであるとみなします。したがって、JBoss は各トランザクション間のエンティティの現行値をキャッシュ化でき、大幅に性能が向上するでしょう。そのため、JBoss で管理されるデータは JBoss 外では変更できません。たとえば、別のプログラムでデータを変更するか、(JBoss 内であっても) ダイレクト JDBC を利用してデータを変更すると、データベースの状態が矛盾してしまいます。
- **B**: JBoss は、データベースのユーザーが複数存在するとみなしますが、トランザクション間のエンティティに関するコンテキスト情報を保持します。このコンテキスト情報は、エンティティのロードイングを最適化するのに使用されます。これはデフォルトの commit オプションです。
- **C**: JBoss は、トランザクションの終わりにすべてのエンティティコンテキスト情報を破棄します。
- **D**: これは JBoss 固有の commit オプションです。このオプションは、指定された期間のみデータが有効であるという点を除けば、commit オプション **A** と似ています。

commit オプションは、**jboss.xml** ファイルで宣言します。このファイルについて詳しくは [29章JBoss 上の EJB](#) を参照してください。以下の例は、アプリケーション内のすべてのエンティティ bean に対して commit オプションを **A** に変更しています。

#### 例30.2 jboss.xml Commit オプション宣言

```
<jboss>
  <container-configurations>
    <container-configuration>
      <container-name>Standard CMP 2.x EntityBean</container-
name>
      <commit-option>A</commit-option>
    </container-configuration>
  </container-configurations>
</jboss>
```

### 30.8.2. 一括読み込みプロセス

エンティティがロードされると、JBoss はロードする必要があるフィールドを決定しなければなりません。デフォルトでは、JBoss は、このエンティティを選択した最終クエリーの **eager-load-group** を使用します。クエリーでエンティティが選択されていない場合、または最終クエリーが **none read-**

ahead ストラテジーを使用していた場合は、JBoss はエンティティに宣言されたデフォルトの **eager-load-group** を使用します。次の設定例では、gangster エンティティ bean に対するデフォルト **eager-load-group** として **basic** ロードグループが設定されています。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <load-groups>
        <load-group>
          <load-group-name>most</load-group-name>
          <field-name>name</field-name>
          <field-name>nickName</field-name>
          <field-name>badness</field-name>
          <field-name>hangout</field-name>
          <field-name>organization</field-name>
        </load-group>
      </load-groups>
      <eager-load-group>most</eager-load-group>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

一括読み込みプロセスは、トランザクション内のエンティティでメソッドが初めて呼び出されたときに開始します。ロードプロセスの詳細は次のとおりです。

1. エンティティコンテキストがまだ有効な場合はローディングは必要なく、したがってローディングプロセスは完了です。commit オプション **A** を使用している場合、または commit オプション **D** を使用していてデータがまだタイムアウトしていない場合には、エンティティコンテキストは有効になります。
2. エンティティコンテキストに残っているデータがフラッシュされます。これにより、古いデータが新しいデータに入り込むことはなくなります。
3. プライマリキー値がプライマリキーフィールドに戻ります。プライマリキーオブジェクトは、実際にはフィールドとは無関係で、ステップ 2 でのフラッシュ後に再ロードする必要があります。
4. このエンティティのプレロードキャッシュ内のすべてのデータが、フィールドにロードされます。
5. まだロードする必要がある追加のフィールドを JBoss が決定します。通常、ロードするフィールドは、エンティティの eager-load グループによって決まりますが、**on-find** または **on-load** 先読みストラテジーを持つクエリーまたは CMR フィールドを使用してエンティティを特定した場合には、オーバーライドすることができます。すべてのフィールドがすでにロードされていた場合、ロードプロセスはステップ 7 に進んでください。
6. クエリーを実行して必要なカラムを選択します。このエンティティが **on-load** ストラテジーを使用している場合、**「on-load」** で説明したとおりに 1 ページのデータがロードされます。現行のエンティティのデータがコンテキストに格納され、他のエンティティのデータはプレロードキャッシュに格納されます。
7. エンティティの **ejbLoad** メソッドが呼び出されます。



### 30.8.3. 遅延ローディングプロセス

遅延ローディングは、一括読み込みと対になっています。あるフィールドを一括読み込みしない場合は、それを遅延ローディングにしなければなりません。bean の未ロードフィールドへのアクセスが行われると、JBoss は、そのフィールドとそれが属する **lazy-load-group** のフィールドをすべてロードします。JBoss は set join を実行し、その後、すでにロードされているフィールドがあれば削除します。設定例を次に示します。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <load-groups>
        <load-group>
          <load-group-name>basic</load-group-name>
          <field-name>name</field-name>
          <field-name>nickName</field-name>
          <field-name>badness</field-name>
        </load-group>
        <load-group>
          <load-group-name>contact info</load-group-name>
          <field-name>nickName</field-name>
          <field-name>contactInfo</field-name>
          <field-name>hangout</field-name>
        </load-group>
      </load-groups>
      <!-- ... -->
      <lazy-load-groups>
        <load-group-name>basic</load-group-name>
        <load-group-name>contact info</load-group-name>
      </lazy-load-groups>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

この設定で bean プロバイダーが **getName()** を呼び出すと、JBoss は、**name**、**nickName**、**badness**を、それらがまだロードされていないとみなしてロードします。bean プロバイダーが **getNickName()**、**name**、**nickName**、**badness**、**contactInfo**、**hangout** がロードされます。遅延ローディングプロセスの詳細は次のとおりです。

1. このエンティティのプレロードキャッシュ内のすべてのデータが、フィールドにロードされます。
2. プレロードキャッシュによってフィールド値がロードされていた場合には、遅延ロードプロセスは終了します。
3. JBoss は、このフィールドを含むすべての遅延ロードグループを見つけ出し、このグループに対して set join を実行し、すでにロードされていたフィールドがあれば削除します。
4. クエリーを実行して必要なカラムを選択します。基本ロードプロセスと同様に、JBoss はエンティティのブロックをロードする場合があります。現行のエンティティのデータがコンテキストに格納され、他のエンティティのデータはプレロードキャッシュに格納されます。

#### 30.8.3.1. リレーションシップ

CMR フィールドはフィールドでもありクエリーでもあるため、リレーションシップは、遅延ローディングでは特別なケースです。フィールドとしては、**on-load** でブロックロードすることが可能で、現在求められているエンティティの値および次のいくつかのエンティティに対する CMR フィールドの値がロードされることを意味します。クエリーとしては、**on-find** を使用して、関連するエンティティのフィールド値をプレロードすることができます。

ここでも、最も簡単にローディングについて詳しく調べるため、使用シナリオを考察していきます。この例では、各 `gangster` とその `hangout` が入った HTML テーブルが生成されます。コード例は次のとおりです。

### 例30.3 リレーションシップの遅延ローディングのコード例

```
public String createGangsterHangoutHtmlTable()
    throws FinderException
{
    StringBuffer table = new StringBuffer();
    table.append("<table>");
    Collection gangsters = gangsterHome.findAll_onfind();
    for (Iterator iter = gangsters.iterator(); iter.hasNext(); ) {
        Gangster gangster = (Gangster)iter.next();

        Location hangout = gangster.getHangout();
        table.append("<tr>");
        table.append("<td>").append(gangster.getName());
        table.append("</td>");
        table.append("<td>").append(gangster.getNickName());
        table.append("</td>");
        table.append("<td>").append(gangster.getBadness());
        table.append("</td>");
        table.append("<td>").append(hangout.getCity());
        table.append("</td>");
        table.append("<td>").append(hangout.getState());
        table.append("</td>");
        table.append("<td>").append(hangout.getZipCode());
        table.append("</td>");
        table.append("</tr>");
    }

    table.append("</table>");return table.toString();
}
```

この例では、`gangster` の `findAll_onfind` クエリーの設定は **on-find** セクションから変わっていません。**Location** エンティティおよび **Gangster-Hangout** リレーションシップの設定は次のとおりです。

### 例30.4 `jbossCMP-jdbc.xml` リレーションシップの遅延ローディング設定

```
<jbossCMP-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>LocationEJB</ejb-name>
      <load-groups>
        <load-group>
          <load-group-name>quick info</load-group-name>
```

```

        <field-name>city</field-name>
        <field-name>state</field-name>
        <field-name>zipCode</field-name>
    </load-group>
</load-groups>
<eager-load-group/>
</entity>
</enterprise-beans>
<relationships>
    <ejb-relation>
        <ejb-relation-name>Gangster-Hangout</ejb-relation-name>
        <foreign-key-mapping/>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                gangster-has-a-hangout
            </ejb-relationship-role-name>
            <key-fields/>
            <read-ahead>
                <strategy>on-find</strategy>
                <page-size>4</page-size>
                <eager-load-group>quick info</eager-load-group>
            </read-ahead>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                hangout-for-a-gangster
            </ejb-relationship-role-name>
            <key-fields>
                <key-field>
                    <field-name>locationID</field-name>
                    <column-name>hangout</column-name>
                </key-field>
            </key-fields>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
</jboss-cmp-jdbc>

```

JBoss は、finder に対して次のクエリーを実行します。

```

SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
ORDER BY t0_g.id ASC

```

hangout にアクセスすると、JBoss 次の 2 つのクエリーを実行して、hangout の **city**、**state**、**zip** フィールドをロードします。

```

SELECT gangster.id, gangster.hangout,
       location.city, location.st, location.zip
FROM gangster, location
WHERE (gangster.hangout=location.id) AND
      ((gangster.id=0) OR (gangster.id=1) OR
       (gangster.id=2) OR (gangster.id=3))
SELECT gangster.id, gangster.hangout,

```

```

        location.city, location.st, location.zip
FROM gangster, location
WHERE (gangster.hangout=location.id) AND
      ((gangster.id=4) OR (gangster.id=5) OR
       (gangster.id=6) OR (gangster.id=7))

```

次の表でクエリを実行したものを示しています。

表30.4 on-find 最適化されたリレーションシップのクエリ実行

id	name	nick_name	badness	hangout	id	city	st	zip
0	Yojimbo	Bodyguard	7	0	0	San Fran	CA	94108
1	Takeshi	Master	10	1	1	San Fran	CA	94133
2	Yuriko	Four finger	4	2	2	San Fran	CA	94133
3	Chow	Killer	9	3	3	San Fran	CA	94133
4	Shogi	Lightning	8	4	4	San Fran	CA	94133
5	Valentino	Pizza-Face	4	5	5	New York	NY	10017
6	Toni	Toothless	2	6	6	Chicago	IL	60661
7	Corleone	Godfather	6	7	7	Las Vegas	NV	89109

#### 30.8.4. 遅延ローディングの結果セット

デフォルトでは、マルチオブジェクトの `finder` または `select` メソッドが実行されると、JDBC 結果セットがすぐさま最後まで読み込まれます。クライアントは、**EJBLocalObject** または CMP フィールド値のコレクションを受け取り、この後それを反復することができます。結果セットが大きい場合は、この方法は効率的ではありません。場合によっては、クライアントがコレクションから対応する値の読み込みを試行するまでは、結果セットにある次の行の読み込みを遅らせるほうがよい場合があります。**lazy-resultset-loading** 要素を使用することで、クエリーに対するこの動作を起こさせることができます。

```

<query>
  <query-method>
    <method-name>findAll</method-name>
  </query-method>

```

```
<jboss-ql><![CDATA[select object(o) from A o]]></jboss-ql>
<lazy-resultset-loading>true</lazy-resultset-loading>
</query>
```

遅延ローディングの結果セットを使用する場合に気をつけなければならないことがいくつかあります。遅延ローディングされた結果セットに関連付けられた **Collection** を扱う場合には、特別な注意が必要です。**iterator()** への最初の呼び出しによって、**ResultSet** から読み込むという特別な **Iterator** を返します。この **Iterator** が使い果たされるまで、その後の **iterator()** に対する呼び出しや **add()** メソッドに対する呼び出しは除外されます。**remove()** および **size()** メソッドはそのまま期待通りに機能します。

## 30.9. トランザクション

この章で紹介した例はすべて、トランザクション内で動作するように定義されたものです。トランザクションはプレロードされたデータの存続期間を決めるため、トランザクションの粒度は最適化ローディングにおいて主要な要素です。トランザクションが完了、コミット、またはロールバックした場合は、プレロードキャッシュ内のデータが失われます。これは結果的に性能の面で深刻な影響を引き起こす可能性があります。

(結果セットを小さく保つため) 先頭の 4 人のギャングを選択する **on-find** 最適化クエリーを使用した例を用いて、トランザクションなしで実行した場合の性能上の影響をデモンストレーションし、これをラッパートランザクションなしで実行します。コード例は次のとおりです。

```
public String createGangsterHtmlTable_no_tx() throws FinderException
{
    StringBuffer table = new StringBuffer();
    table.append("<table>");

    Collection gangsters = gangsterHome.findFour();
    for(Iterator iter = gangsters.iterator(); iter.hasNext(); ) {
        Gangster gangster = (Gangster)iter.next();
        table.append("<tr>");
        table.append("<td>").append(gangster.getName());
        table.append("</td>");
        table.append("<td>").append(gangster.getNickName());
        table.append("</td>");
        table.append("<td>").append(gangster.getBadness());
        table.append("</td>");
        table.append("</tr>");
    }

    table.append("</table>");
    return table.toString();
}
```

この finder の結果、次のクエリーが実行されます。

```
SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
WHERE t0_g.id < 4
ORDER BY t0_g.id ASC
```

通常はこれが実行される唯一のクエリーになりますが、このコードはトランザクション内で実行されていないために、finder が戻るとすぐに、プレロードされたデータはすべて破棄されます。そのため、

CMP フィールドにアクセスすると、JBoss は次の 4 つのクエリー (各グループに対して 1 つ) を実行します。

```
SELECT id, name, nick_name, badness
  FROM gangster
 WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
  FROM gangster
 WHERE (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
  FROM gangster
 WHERE (id=2) OR (id=3)
SELECT name, nick_name, badness
  FROM gangster
 WHERE (id=3)
```

実際にはこれよりも悪い事態になります。JBoss は、各クエリーを 3 回、アクセスされた各 CMP フィールドに対して 1 回ずつ実行します。その理由は、CMP フィールドアクセッサの呼び出しと次の呼び出しの間で、プレロードされた値が破棄されるからです。

次の図は、クエリーの実行を示したものです。

id§	name§	nick name§	badness§
0§	Yojimbo	Bodyguard	7
1§	Takeshi	Master	10
2§	Yuriko	Four finger	4
3§	Chow	Killer	9

図30.13 トランザクションなし on-find 最適化クエリー実行

データベースからロードされるデータ量のせいで、先読みなしにした場合よりもパフォーマンスが悪くなります。ロードされる行の数は、次の式で求められます。

$$n + n - 1 + n - 2 + \dots + 1 = \frac{n * (n + 1)}{2} = O(n^2)$$

例の中のトランザクションはエンティティに対する 1 回の呼び出しによって境界が引かれるため、これはすべて起こります。これは「トランザクション内でどのようにコードを実行するか」という重要な問題を提起します。その答えは、コードを実行する場所によって変わります。EJB (セッション、エンティティ、またはメッセージ駆動型) で実行する場合は、**assembly-descriptor** で **Required** または **RequiresNewtrans-attribute** をメソッドに付けなければなりません。コードを EJB で実行していない場合は、ユーザートランザクションが必要です。次のコードでは、ユーザートランザクションによって、宣言されたメソッドに対する呼び出しをラップしています。

```
public String createGangsterHtmlTable_with_tx()
    throws FinderException
{
    UserTransaction tx = null;
    try {
```

```

        InitialContext ctx = new InitialContext();
        tx = (UserTransaction) ctx.lookup("UserTransaction");
        tx.begin();

        String table = createGangsterHtmlTable_no_tx();

        if (tx.getStatus() == Status.STATUS_ACTIVE) {
            tx.commit();
        }
        return table;
    } catch (Exception e) {
        try {
            if (tx != null) tx.rollback();
        } catch (SystemException unused) {
            // eat the exception we are exceptioning out anyway
        }
        if (e instanceof FinderException) {
            throw (FinderException) e;
        }
        if (e instanceof RuntimeException) {
            throw (RuntimeException) e;
        }

        throw new EJBException(e);
    }
}

```

## 30.10. 楽観的ロックング

JBoss には、エンティティ bean の 楽観ロックングに対するサポートがあります。楽観ロックングによって、同じエンティティ bean の複数のインスタンスを同時にアクティブにすることができます。楽観ロックングポリシー選択に基づいて一貫性を強化します。楽観ロックングポリシー選択では、修正されたデータのデータベースへのコミット時間の書き込みに使用するフィールドセットを定義します。楽観ローディングの一貫性チェックでは、選択されたフィールドセットの値が、現行のトランザクションを開始した時点で存在していたデータベース内の値と同じであることを表明します。これは、値のアサーションを含む **select for UPDATE WHERE ...** ステートメントを使用して行われます。

楽観ロックングポリシー選択は、**jbosscmp-jdbc.xml** 記述子の **optimistic-locking** 要素を使用して指定します。**optimistic-locking** 要素のコンテンツモデルを次に示し、その後に各要素の説明をします。

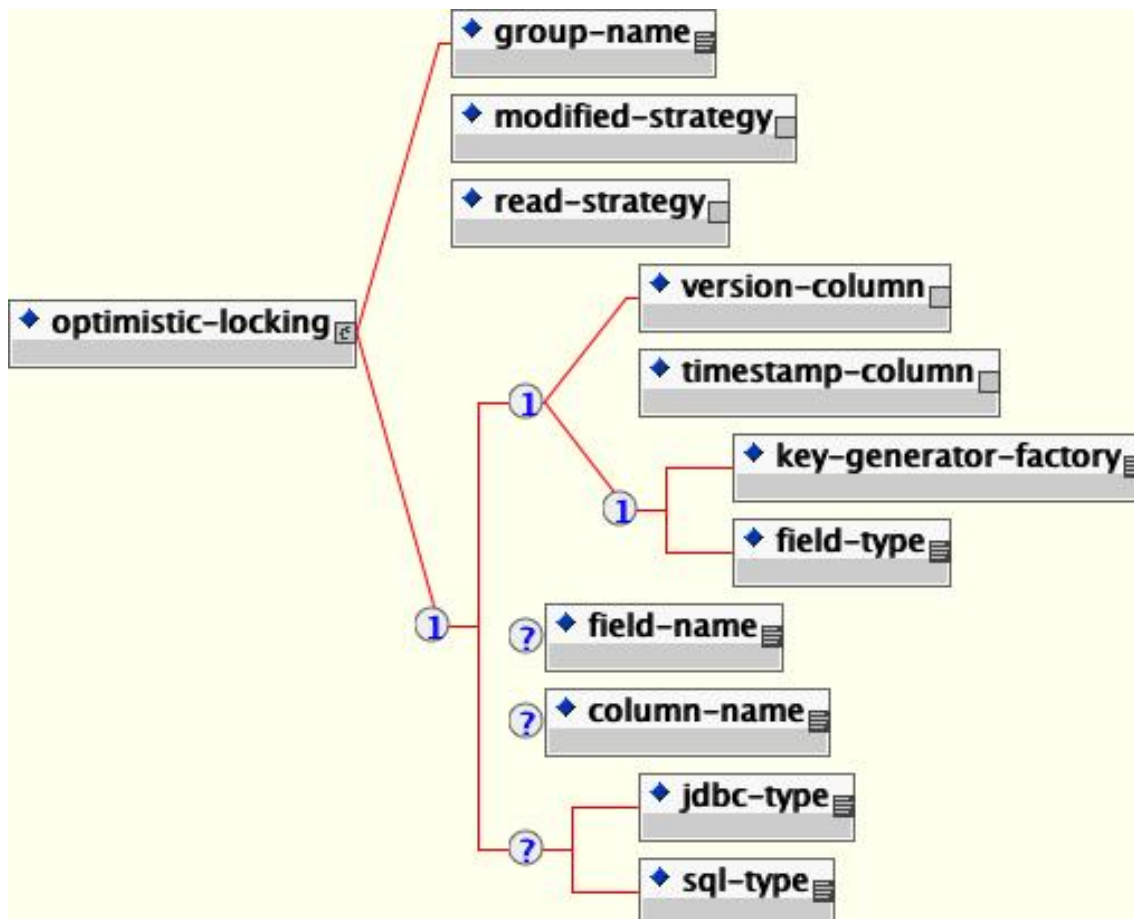


図30.14 jbossCMP-jdbc optimistic-locking 要素コンテンツモデル

- **group-name**: この要素は、**load-group** のフィールドに基づく楽観ロックであることを指定します。この要素のこの値は、エンティティの **load-group-name** のいずれかと一致していなければなりません。このグループ内のフィールドが楽観ロックに使用されます。
- **modified-strategy**: この要素は、変更フィールドをベースにした楽観ロックであることを指定します。この戦略は、トランザクション中に修正されたフィールドが楽観ロックに使用されることを意味します。
- **read-strategy**: この要素は、読み込まれたフィールドに基づく楽観ロックであることを指定します。この戦略は、トランザクション内の読み込み/変更されたフィールドが楽観ロックに使用されることを意味します。
- **version-column**: この要素は、バージョンカラム戦略に基づく楽観ロックであることを指定します。この要素を指定すると、楽観ロック用にエンティティ bean に **java.lang.Long** タイプの追加のバージョンフィールドが追加されます。エンティティの更新ごとに、このフィールドの値が増えます。**field-name** 要素は、CMP フィールドの名前の指定を可能にするものであるのに対し、**column-name** 要素は、対応するテーブルのカラム指定ができるようになります。
- **timestamp-column**: この要素は、タイムスタンプのカラム戦略に基づく楽観ロックであることを指定します。この要素を指定すると、楽観ロック用にエンティティ bean に **java.util.Date** タイプのバージョンフィールドが別途追加されます。エンティティの更新ごとに、このフィールドの値を現在時刻に設定します。**field-name** 要素は、CMP フィールドの名前の指定を可能にするものであるのに対し、**column-name** 要素は、対応するテーブルのカラム指定ができるようになります。
- **key-generator-factory**: この要素は、キー生成に基づく楽観ロックであることを指定します。この要素の値



は、**org.jboss.ejb.plugins.keygenerator.KeyGeneratorFactory**実装の JNDI 名です。この要素を指定すると、楽観ロッキング用にエンティティ bean にバージョンフィールドが別途追加されます。フィールドのタイプは **field-type** 要素で指定しなければなりません。エンティティの更新ごとに、キージェネレーターから新しい値を取得することでキーフィールドを更新します。**field-name** 要素は、CMP フィールドの名前の指定を可能にするものであるのに対し、**column-name** 要素は、対応するテーブルのカラム指定を可能にします。

すべての楽観ロッキングストラテジーを記した **jbosscmp-jdbc.xml** 記述子のサンプルを次に示します。

```
<!DOCTYPE jbosscmp-jdbc PUBLIC
    "-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_2.dtd">
<jbosscmp-jdbc>
  <defaults>
    <datasource>java:/DefaultDS</datasource>
    <datasource-mapping>Hypersonic SQL</datasource-mapping>
  </defaults>
  <enterprise-beans>
    <entity>
      <ejb-name>EntityGroupLocking</ejb-name>
      <create-table>true</create-table>
      <remove-table>true</remove-table>
      <table-name>entitygrouplocking</table-name>
      <cmp-field>
        <field-name>dateField</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>integerField</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>stringField</field-name>
      </cmp-field>
      <load-groups>
        <load-group>
          <load-group-name>string</load-group-name>
          <field-name>stringField</field-name>
        </load-group>
        <load-group>
          <load-group-name>all</load-group-name>
          <field-name>stringField</field-name>
          <field-name>dateField</field-name>
        </load-group>
      </load-groups>
      <optimistic-locking>
        <group-name>string</group-name>
      </optimistic-locking>
    </entity>
    <entity>
      <ejb-name>EntityModifiedLocking</ejb-name>
      <create-table>true</create-table>
      <remove-table>true</remove-table>
      <table-name>entitymodifiedlocking</table-name>
      <cmp-field>
        <field-name>dateField</field-name>
      </cmp-field>
```

```

    <cmp-field>
      <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
      <modified-strategy/>
    </optimistic-locking>
  </entity>
  <entity>
    <ejb-name>EntityReadLocking</ejb-name>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
    <table-name>entityreadlocking</table-name>
    <cmp-field>
      <field-name>dateField</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
      <read-strategy/>
    </optimistic-locking>
  </entity>
  <entity>
    <ejb-name>EntityVersionLocking</ejb-name>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
    <table-name>entityversionlocking</table-name>
    <cmp-field>
      <field-name>dateField</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
      <version-column/>
      <field-name>versionField</field-name>
      <column-name>ol_version</column-name>
      <jdbc-type>INTEGER</jdbc-type>
      <sql-type>INTEGER(5)</sql-type>
    </optimistic-locking>
  </entity>
  <entity>
    <ejb-name>EntityTimestampLocking</ejb-name>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
    <table-name>entitytimestamplocking</table-name>
    <cmp-field>

```

```

        <field-name>dateField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
        <timestamp-column/>
        <field-name>versionField</field-name>
        <column-name>ol_timestamp</column-name>
        <jdbc-type>TIMESTAMP</jdbc-type>
        <sql-type>DATETIME</sql-type>
    </optimistic-locking>
</entity>
<entity>
    <ejb-name>EntityKeyGeneratorLocking</ejb-name>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
    <table-name>entitykeygenlocking</table-name>
    <cmp-field>
        <field-name>dateField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>integerField</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>stringField</field-name>
    </cmp-field>
    <optimistic-locking>
        <key-generator-factory>UUIDKeyGeneratorFactory</key-
generator-factory>
        <field-type>java.lang.String</field-type>
        <field-name>uuidField</field-name>
        <column-name>ol_uuid</column-name>
        <jdbc-type>VARCHAR</jdbc-type>
        <sql-type>VARCHAR(32)</sql-type>
    </optimistic-locking>
</entity>
</enterprise-beans>
</jbossCMP-jdbc>

```

## 30.11. エンティティコマンドおよびプライマリーキー生成

永続ストアにエンティティを挿入するのに使用するエンティティ作成コマンドオブジェクトをカスタム実装することで、エンティティ bean 外でのプライマリーキー生成に対応するようになりました。利用可能なコマンドのリストは、**jbossCMP-jdbc.xml** 記述子の **entity-command** 要素で指定します。デフォルトの **entity-command** は、**jbossCMP-jdbc.xml** の **defaults** 要素で指定できます。それぞれの **entity** 要素では、独自の **entity-command** を指定することによって、デフォルトの **entity-command** をオーバーライドすることができます。**entity-command** および子要素のコンテンツモデルを次に示します。



図30.15 jbosscmp-jdbc.xml entity-command 要素モデル

各 **entity-command** 要素では、エンティティ生成実装を指定します。**name** 属性では、**entity-commands** セクションで定義されたコマンドを defaults および entity 要素で参照できるようにする名前を指定します。**class** 属性では、キー生成をサポートする **org.jboss.ejb.plugins.cmp.jdbc** 実装を指定します。データベースベンダー固有のコマンドは通常、挿入実行の副次的結果としてデータベースがプライマリーキーを生成する場合は

**org.jboss.ejb.plugins.cmp.jdbc.JDBCIdentityColumnCreateCommand** を、生成されたキーをコマンドで挿入しなければならない場合は

**org.jboss.ejb.plugins.cmp.jdbc.JDBCInsertPKCreateCommand** をサブクラス化します。

オプションの **attribute** 属性は、エンティティコマンド実装クラスに対して利用可能になる任意名/値プロパティペアの指定を可能にするものです。**attribute** 属性には、名前プロパティを指定する必須の **name** 属性があり、**attribute** 属性のコンテンツはプロパティの値です。属性値

は、**org.jboss.ejb.plugins.cmp.jdbc.metadata.JDBCEntityCommandMetaData.getAttribute(文字列)** メソッドを使いアクセスすることができます。

### 30.11.1. 既存のエンティティコマンド

以下は、**standardjbosscmp-jdbc.xml** 記述子内にある現行の **entity-command** 定義です。

- default: (org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand)**  
**JDBCCreateEntityCommand** は、**standardjbosscmp-jdbc.xml** defaults 要素で参照される **entity-command** であるため、デフォルトの要素作成になります。この **entity-command** は、割り当てられたプライマリーキー値を使用して **INSERT INTO** クエリーを実行します。
- no-select-before-insert:**  
**(org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand)** これ  
 は、**jbosscmp-jdbc:service=SQLExceptionProcessor** サービスをポイントする属性 **name="SQLExceptionProcessor"** を指定することによって insert の select をスキップする default の変化形です。**SQLExceptionProcessor** サービスは、**boolean isDuplicateKey(SQLException e)** 操作を提供し、一意制約違反がないか判断することができます。
- pk-sql (org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCPkSqlCreateCommand)**  
**JDBCPkSqlCreateCommand** は、**pk-sql** 属性により与えられる **INSERT INTO** クエリーステートメントを実行して、次のプライマリーキー値を取得します。シーケンスサポートのあるデータベースが主要な使用目的です。
- mysql-get-generated-keys:**  
**(org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCMySQLCreateCommand)**  
**JDBCMySQLCreateCommand** は、MySQL ネイティブ **java.sql.Statement** インターフェイス実装から **getGeneratedKeys** メソッドを使用して **INSERT INTO** クエリーを実行し生成されたキーをフェッチします。
- oracle-sequence:**  
**(org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCOracleCreateCommand)**  
**JDBCOracleCreateCommand** は、**RETURNING** 句とともにシーケンスを使用して単一のス

ステートメントでキーを生成する、Oracle と併用する作成コマンドです。これには、シーケンスのカラム名を指定する必須の **sequence** 要素があります。

- **hsqldb-fetch-key:**  
(org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCMySQLCreateCommand)  
JDBCMySQLCreateCommand は、**CALL IDENTITY()** ステートメントの実行後に **INSERT INTO** クエリーを実行して生成されたキーをフェッチします。
- **sybase-fetch-key:**  
(org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCSybaseCreateCommand)  
JDBCSybaseCreateCommand は、**SELECT @@IDENTITY** ステートメントの実行後に **INSERT INTO** クエリーを実行して生成されたキーをフェッチします。
- **mssql-fetch-key:**  
(org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCSQLServerCreateCommand)  
**IDENTITY** カラムからの値を使用する Microsoft SQL Server 用の **JDBCSQLServerCreateCommand** です。デフォルトでは、**SELECT SCOPE\_IDENTITY()** を使用して、トリガーの影響を減らします。たとえば V7 などの **pk-sql** 属性でオーバーライドできます。
- **informix-serial:**  
(org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCInformixCreateCommand)  
JDBCInformixCreateCommand は、Informix ネイティブ **java.sql.Statement** インターフェース実装から **getSerial** メソッドを使用した後で **INSERT INTO** クエリーを実行して生成されたキーをフェッチします。
- **postgresql-fetch-seq:**  
(org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCPostgreSQLCreateCommand) シーケンスの現在値をフェッチする PostgreSQL 用の **JDBCPostgreSQLCreateCommand** です。オプションの **sequence** 属性を使用して、シーケンス名を変更することができます。デフォルトは **table\_pkColumn\_seq** です。
- **key-generator:**  
(org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCKeyGeneratorCreateCommand)  
JDBCKeyGeneratorCreateCommand は、**key-generator-factory** により参照されるキージェネレーターからプライマリキーの値を取得した後に **INSERT INTO** クエリーを実行します。**key-generator-factory** 属性では **org.jboss.ejb.plugins.keygenerator.KeyGeneratorFactory** 実装の JNDI バインディングの名前を指定しなければなりません。
- **get-generated-keys:**  
(org.jboss.ejb.plugins.cmp.jdbc.jdbc3.JDBCGetGeneratedKeysCreateCommand)  
JDBCGetGeneratedKeysCreateCommand は、自動生成されたキーを取り出す機能を持つ **prepareStatement(String, Statement.RETURN\_GENERATED\_KEYS)** を使用して構築されたステートメントを使用して **INSERT INTO** クエリーを実行します。生成されたキーは、**PreparedStatement.getGeneratedKeys** メソッドを呼び出すことによって得られます。これは JDBC3 サポートが必要なため、サポートする JDBC ドライバーがある JDK1.4.1+でのみ利用可能です。

生成されたキーが既知のプライマリキー **cmp-field** にマッピングされる、**hsqldb-fetch-keyentity-command** を使用した設定例を次に示します。

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
```

```

    <ejb-name>LocationEJB</ejb-name>
    <pk-constraint>>false</pk-constraint>
    <table-name>location</table-name>

    <cmp-field>
      <field-name>locationID</field-name>
      <column-name>id</column-name>
      <auto-increment/>
    </cmp-field>
    <!-- ... -->
    <entity-command name="hsqldb-fetch-key"/>

  </entity>
</enterprise-beans>
</jbosscomp-jdbc>

```

明示的な **cmp-field** なしに不明なプライマリキーを使用した別例を次に示します。

```

<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>LocationEJB</ejb-name>
      <pk-constraint>>false</pk-constraint>
      <table-name>location</table-name>
      <unknown-pk>
        <unknown-pk-class>java.lang.Integer</unknown-pk-class>
        <field-name>locationID</field-name>
        <column-name>id</column-name>
        <jdbc-type>INTEGER</jdbc-type>
        <sql-type>INTEGER</sql-type>
        <auto-increment/>
      </unknown-pk>
      <!-- ... -->
      <entity-command name="hsqldb-fetch-key"/>
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>

```

## 30.12. デフォルト

JBoss グローバルデフォルトは、**standardjbosscomp-jdbc.xml** file of the **server/<server-name>/conf/** ファイルに定義されています。それぞれのアプリケーションで、**jbosscomp-jdbc.xml** ファイル内のグローバルデフォルトをオーバーライドすることができます。デフォルトオプションは、設定ファイルの **defaults** 要素に入っています。コンテンツモデルを次に示します。

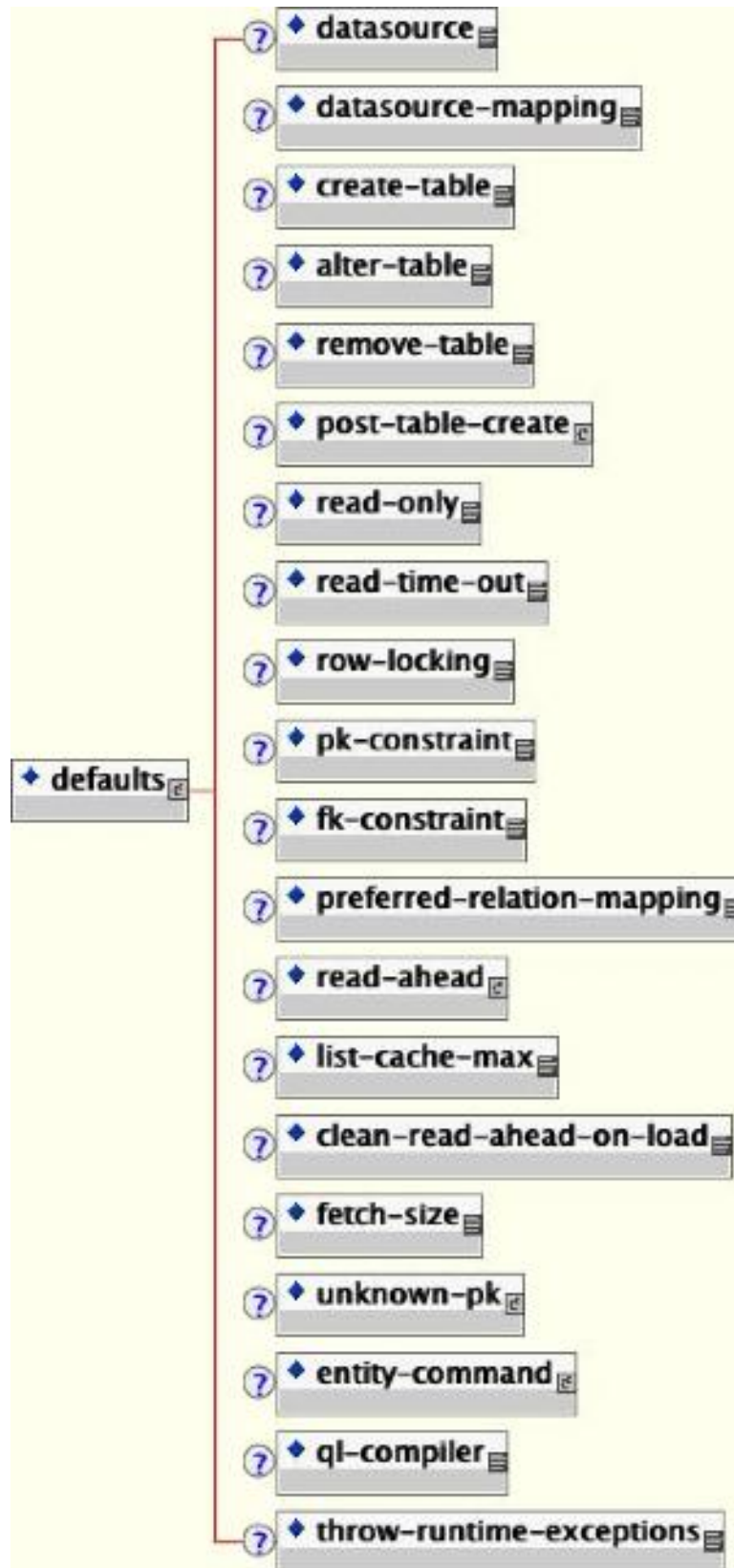


図30.16 jbosscmp-jdbc.xml defaults コンテンツモデル

defaults セクションの例は次のとおりです。

```
<jbosscmp-jdbc>
  <defaults>
    <datasource>java:/DefaultDS</datasource>
    <datasource-mapping>Hypersonic SQL</datasource-mapping>
    <create-table>true</create-table>
```

```

        <remove-table>false</remove-table>
        <read-only>false</read-only>
        <read-time-out>300000</read-time-out>
        <pk-constraint>true</pk-constraint>
        <fk-constraint>false</fk-constraint>
        <row-locking>false</row-locking>
        <preferred-relation-mapping>foreign-key</preferred-relation-
mapping>
        <read-ahead>
            <strategy>on-load</strategy>
            <page-size>1000</page-size>
            <eager-load-group>*</eager-load-group>
        </read-ahead>
        <list-cache-max>1000</list-cache-max>
    </defaults>
</jbossCMP-jdbc>

```

### 30.12.1. jbossCMP-jdbc.xml defaults 宣言例

各オプションは、エンティティ、リレーションシップ、またはその両方に適用することができ、特定のエンティティまたはリレーションシップでオーバーライドすることができます。各オプションの詳細は次のとおりです。

- **datasource**: このオプションの要素は、データソースのルックアップに使用される jndi-name になります。エンティティまたは **relation-table** で使用されるデータベース接続はすべてデータソースから得られます。エンティティにさまざまなデータソースを持たせると、finder および ejbSelect でクエリーできるドメインを大きく制約することになるため推奨しません。
- **datasource-mapping**: このオプションの要素は、Java タイプを SQL タイプにマッピングする方法、および EJB-QL 関数をデータベース固有の関数にマッピングする方法を決める **type-mapping** の名前を指定します。タイプマッピングについては、「[マッピング](#)」で説明しています。
- **create-table**: この任意要素が true の場合、JBoss はエンティティのテーブルの作成を試行しなければなりません。アプリケーションがデプロイされると、JBoss は、テーブル作成の前にすでにテーブルが存在しているかどうか確認します。テーブルが見つかった場合は、ログに記録が取られ、テーブルは作成されません。このオプションは、開発の初期段階でテーブル構造が頻繁に変わる場合に非常に役立ちます。デフォルトは false です。
- **alter-table**: スキーマの自動作成に **create-table** が使用される場合、**alter-table** を使ってエンティティ bean に対する変更を反映しスキーマを最新状態に保つことができます。Alter table は次のような特定のタスクを実行します。
  - 新しいフィールドが作成されます。
  - 使用されなくなったフィールドが削除されます。
  - 宣言されている長さより短い文字列フィールドは宣言されている長さまで延長されます(すべてのデータベースでサポートされているわけではありません)。
- **remove-table**: この任意の要素が true の場合、JBoss は、各エンティティおよびリレーションシップにマッピングした各関連テーブルに対して、テーブルのドロップを試行します。アプリケーションのデプロイが解除されたときに、JBoss はテーブルのドロップを試行します。このオプションは、開発の初期段階でテーブル構造が頻繁に変わる場合に非常に役立ちます。デフォルトは false です。



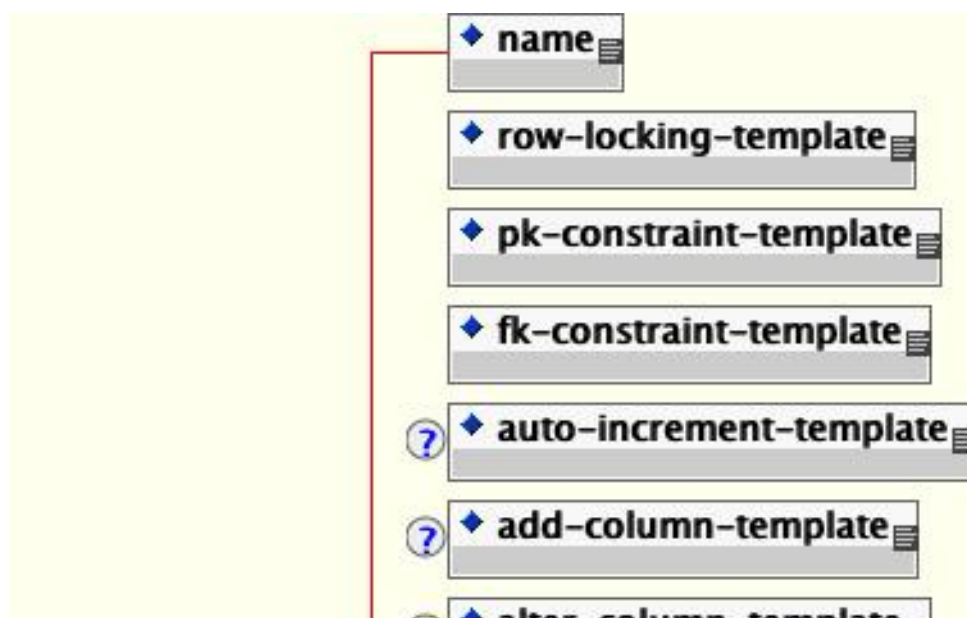
- **read-only:** この任意の要素は、true の場合、bean プロバイダーがフィールドの値を変更ができなくなります。読み取り専用のフィールドは、データベースに格納もしくは挿入されません。プライマリキーフィールドが読み取り専用である場合、create メソッドは **CreateException** を送出します。**read-only** フィールドで set アクセッサが呼び出されると、**EJBException** をスローします。読み取り専用フィールドは、最終更新など、データベーストリガーによって入力されるフィールドで役立ちます。**read-only** オプションは、フィールドごとにオーバーライドすることが可能です。デフォルトは false です。
- **read-time-out:** この任意の要素では、読み取り専用フィールドで読み取りが有効である時間をミリ秒単位で表します。値が 0 の場合にはトランザクション開始時に必ずその値が再読み込みされ、値が -1 の場合には値はタイムアウトしません。このオプションも、CMP フィールドごとにオーバーライドすることが可能です。**read-only** が false の場合、この値は無視されます。デフォルトは -1 です。
- **row-locking:** この任意の要素が true の場合、トランザクションでロードされたすべての行を JBoss がロックすることを指定します。ほとんどのデータベースがエンティティのロード時に **SELECT FOR UPDATE** 構文を使用してこれを実装しますが、実際の構文は、このエンティティで使用される **datasource-mapping** 内の **row-locking-template** で決まります。デフォルトは false です。
- **pk-constraint:** この任意の要素では、true の場合テーブル作成時に JBoss がプライマリキー制約を追加することを指定します。デフォルトは false です。
- **preferred-relation-mapping:** この任意の要素は、リレーションシップに対する希望のマッピングスタイルを指定します。**preferred-relation-mapping** 要素は、**foreign-key** または **relation-table** のいずれかでなければなりません。
- **read-ahead:** この任意の要素はエンティティの CMR フィールドおよびクエリ結果のキャッシュ化を制御します。このオプションについては「[Read-ahead](#)」で説明します。
- **list-cache-max:** この任意の要素は、このエンティティで追跡できる **read-lists** の数を指定します。このオプションについては、「[on-load](#)」で説明しています。デフォルトは 1000 です。
- **clean-read-ahead-on-load:** 先行読み込み (read ahead) キャッシュからエンティティがロードされる場合、JBoss は先行読み込みキャッシュから使用されるデータを削除することができます。デフォルトは **false** になります。
- **fetch-size:** この任意の要素は、基盤データストアへの 1 往復で読み込むエンティティの数を指定します。デフォルトは 0 です。
- **unknown-pk:** この任意の要素によって、**java.lang.Object** の未知のプライマリキータイプを永続ストアにマッピングする方法を定義することができます。
- **entity-command:** この任意の要素によって、エンティティ作成用のデフォルトコマンドを定義することができます。これについては、「[エンティティコマンドおよびプライマリキー生成](#)」で詳しく説明しています。
- **ql-compiler:** この任意の要素によって、代替のクエリーコンパイラを指定することができます。代替のクエリーコンパイラについては、「[EJBQL 2.1 と SQL92 のクエリ](#)」で説明しています。
- **throw-runtime-exceptions:** この属性では、true に設定された場合、データベースへの接続エラーは、チェック例外としてではなく、ランタイム **EJBException** として表示されるべきであることを示します。

## 30.13. データソースのカスタマイズ

JBoss には、Cloudscape、DB2、DB2/400、Hypersonic SQL、InformixDB、InterBase、MS SQLSERVER、MS SQLSERVER2000、MySQL、Oracle7、Oracle8、Oracle9i、PointBase、PostgreSQL、PostgreSQL 7.2、SapDB、SOLID、Sybase など、多くのデータベースに対してあらかじめ定義されたタイプマッピングがあります。提供されているマッピングを使用したくない場合、または自分のデータベース用のマッピングが提供されていない場合は、新しいマッピングを定義する必要があります。提供されているマッピングに誤りを見つけた場合、あるいは新しいデータベース用に新しいマッピングを作成した場合は、SourceForge で JBoss プロジェクトページにパッチを掲載してください。

### 30.13.1. タイプマッピング

データベースのカスタマイズは、`jbosscmp-jdbc.xml` 記述子の `type-mapping` セクションで行います。type-mapping 要素のコンテンツモデルは、[図30.17 「jbosscmp-jdbc type-mapping 要素のコンテンツモデル」](#) に示してあります。



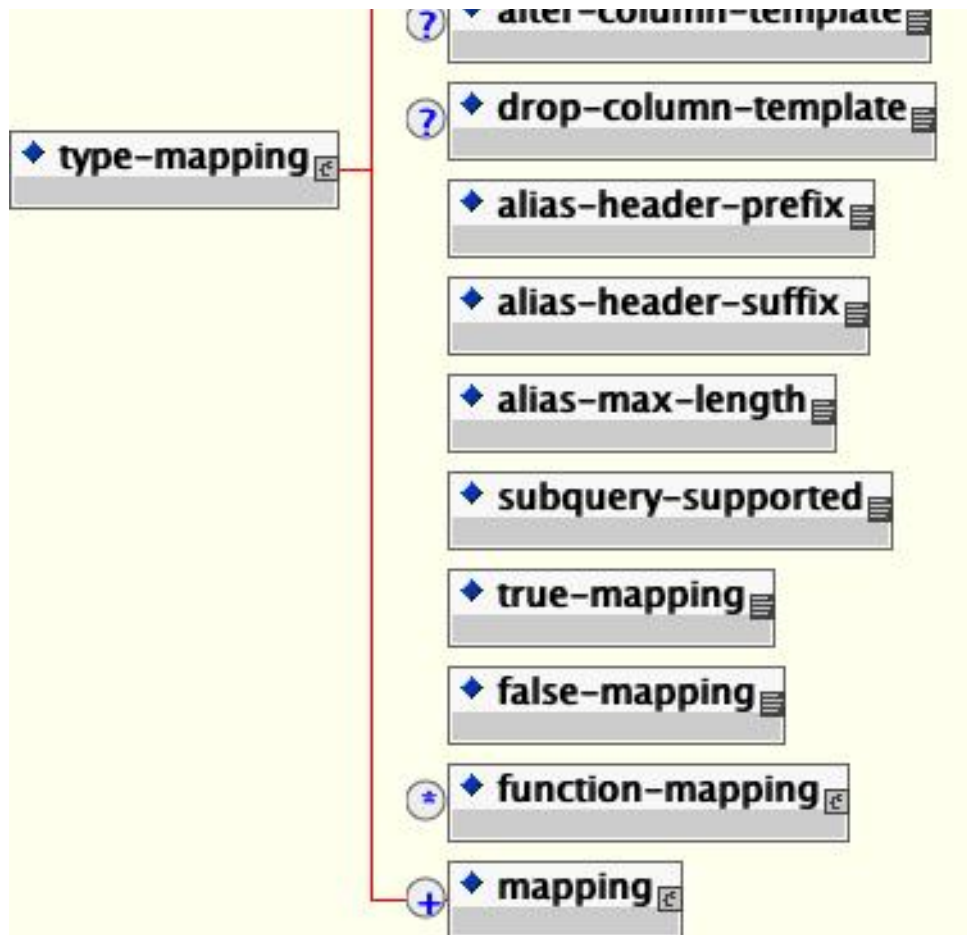


図30.17 jbossCMP-jdbc type-mapping 要素のコンテンツモデル

各要素は次のとおりです。

- **name:** この必須の要素は、データベースのカスタマイズを識別する名前を指定します。これは、defaults およびエンティティ内に見られる **datasource-mapping** 要素によるマッピングを参照するのに使用されます。
- **row-locking-template:** この必須の要素により、選択された行で行ロックを作成するのに使用する **PreparedStatement** テンプレートが与えられます。テンプレートは、次の3つの引数をサポートしなければなりません。

1. select 句
2. from 句。テーブルの順序は、現在のところ対応していません。
3. where 句

select ステートメントで行レベルのロッキングがサポートされない場合は、この要素を空にしなければなりません。行レベルのロッキングの最も一般的な形式は、たとえば **SELECT ?1 FROM ?2 WHERE ?3 FOR UPDATE** などのような select for update です。

- **pk-constraint-template:** この必須の要素は、create table ステートメントにおいてプライマリキー制約を作成するのに使用する **PreparedStatement** テンプレートを渡します。このテンプレートは、次の2つの引数をサポートしなければなりません。
1. プライマリキー制約名。これは常に **pk\_{table-name}** です。
  2. コンマで区切ったプライマリキー列名のリスト

create table ステートメントでプライマリー制約句がサポートされない場合は、この要素を空にしなければなりません。プライマリー制約の最も一般的な形式は **CONSTRAINT ?1 PRIMARY KEY (?2)** です。

- **fk-constraint-template:** これは、別のステートメントで外部キー制約を作成するのに使用するテンプレートです。テンプレートは、次の 5 つの引数をサポートしなければなりません。

1. テーブル名
2. 外部キー制約名。これは常に **fk\_{table-name}\_{cmr-field-name}** です。
3. コンマで区切った外部キー列名のリスト
4. 参照テーブル名
5. コンマで区切った参照プライマリーキーカラム名のリスト

データソースが外部キー制約をサポートしない場合は、この要素を空にしなければなりません。外部キー制約の最も一般的な形式は **ALTER TABLE ?1 ADD CONSTRAINT ?2 FOREIGN KEY (?3) REFERENCES ?4 (?5)** です。

- **auto-increment-template:** これは、自動インクリメントカラムを指定するための SQL テンプレートを宣言します。
- **add-column-template:** **alter-table** が true であるとき、この SQL テンプレートは、既存のテーブルに列を追加するための構文を指定します。デフォルト値は **ALTER TABLE ?1 ADD ?2 ?3** です。各パラメーターは次のとおりです。

1. テーブル名
2. カラム名
3. カラムタイプ

- **alter-column-template:** **alter-table** が true であるとき、この SQL テンプレートは、既存のテーブルから列をドロップするための構文を指定します。デフォルト値は **ALTER TABLE ?1 ALTER ?2 TYPE ?3** です。各パラメーターは次のとおりです。

1. テーブル名
2. カラム名
3. カラムタイプ

- **drop-column-template:** **alter-table** が true であるとき、この SQL テンプレートは、既存のテーブルからカラムをドロップするための構文を指定します。デフォルト値は **ALTER TABLE ?1 DROP ?2** です。各パラメーターは次のとおりです。

1. テーブル名
2. カラム名

- **alias-header-prefix:** この必須の要素により、エイリアスヘッダーの作成に使用する接頭辞が付与されます。エイリアスヘッダーは、EJB-QL コンパイラーにより生成されるテーブルエイリアスの先頭に付けられ、名前の重複を防ぎます。エイリアスヘッダーは **alias-header-prefix + int\_counter + alias-header-suffix** のように構成されます。エイリアスヘッダーの例は、**alias-header-prefix** が **"t"**、**alias-header-suffix** が **"\_"** の場合で **t0\_** になります。

- **alias-header-suffix**: この必須の要素により、生成されるエイリアスヘッダーの接尾辞部分が付与されます。
- **alias-max-length**: この必須の要素により、生成されるエイリアスヘッダーの最大許容長が渡されます。
- **subquery-supported**: この必須の要素は、この **type-mapping** サブクエリーを true とするか false とするかを指定します。一部の EJB-QL 演算子は既存のサブクエリーにマッピングされます。**subquery-supported** が false の場合、EJB-QL コンパイラーは left join を使用し、null になります。
- **true-mapping**: この必須の要素は、EJB-QL クエリーにおける true の識別を定義します。例としては、**TRUE**、**1**、**(1=1)** があります。
- **false-mapping**: この必須の要素は、EJB-QL クエリーにおける false の識別を定義します。例としては、**FALSE**、**0**、**(1=0)** があります。
- **function-mapping**: この任意の要素は、EJB-QL 関数から SQL 実装へのマッピングを 1 つ以上指定します。詳しくは「[関数マッピング](#)」を参照してください。
- **mapping**: この必須の要素は、Java タイプから対応する JDBC および SQL タイプへのマッピングを指定します。詳しくは「[マッピング](#)」を参照してください。

### 30.13.2. 関数マッピング

function-mapping 要素モデルを次に示します。

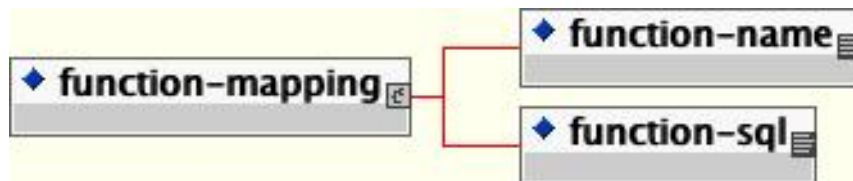


図30.18 jbossCMP-jdbc function-mapping 要素コンテンツモデル

許容される子エレメントは次のとおりです。

- **function-name**: この必須の要素は、**concat**、**substring** などの EJB-QL 関数名を渡します。
- **function-sql**: この必須の要素は、基盤データベースに適した関数用の SQL を渡します。**concat** 関数の例としては、**(?1 || ?2)**、**concat(?1, ?2)**、**(?1 + ?2)** があります。

### 30.13.3. マッピング

**type-mapping** とは、マッピングする Java クラスタイプとデータベースタイプ間のマッピングセットのことです。タイプマッピングは **mapping** 要素により定義されます。このコンテンツモデルについては、[図30.19「jbossCMP-jdbc マッピング要素のコンテンツモデル」](#)で紹介しています。

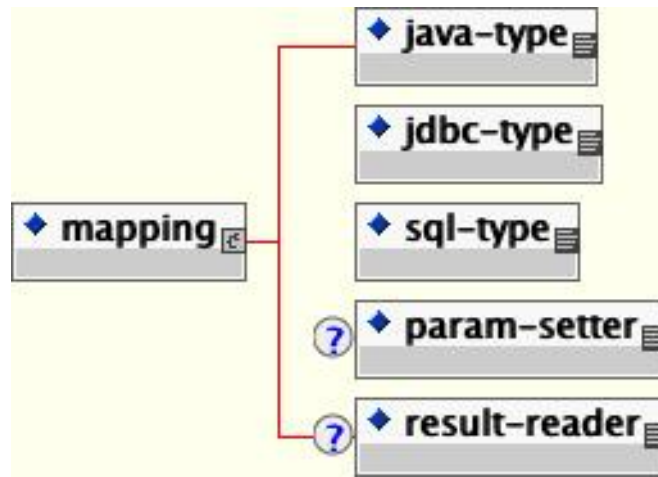


図30.19 jbossCMP-jdbc マッピング要素のコンテンツモデル

JBoss は各タイプのマッピングを見つけられない場合、オブジェクトをシリアル化し、`java.lang.Object` マッピングを使用します。次に、マッピング要素の 3 つの子要素について説明します。

- **java-type**: この必須の要素は、Java クラスの完全修飾名を渡しマッピングします。クラスが `java.lang.Short` などのプリミティブラッパークラスである場合、マッピングもプリミティブタイプに適用します。
- **jdbc-type**: この必須の要素により、JDBC `PreparedStatement` でのパラメーターの設定時または JDBC `ResultSet` からのデータのロード時に使用する JDBC タイプが与えられます。有効なタイプは `java.sql.Types` に定義されます。
- **sql-type**: この必須の要素により、create table ステートメントで使用する SQL タイプが与えられます。データベースベンダーのみが有効なタイプに制限を加えます。
- **param-setter**: この任意の要素は、このマッピングの `JDBCParameterSetter` 実装の完全修飾名を指定します。
- **result-reader**: この任意の要素は、このマッピングに対し `JDBCResultSetReader` 実装の完全修飾名を指定します。

Oracle9i での mapping 要素の例を示します。

```

<jbossCMP-jdbc>
  <type-mappings>
    <type-mapping>
      <name>Oracle9i</name>
      <!--...-->
      <mapping>
        <java-type>java.lang.Short</java-type>
        <jdbc-type>NUMERIC</jdbc-type>
        <sql-type>NUMBER(5)</sql-type>
      </mapping>
    </type-mapping>
  </type-mappings>
</jbossCMP-jdbc>

```

#### 30.13.4. ユーザータイプマッピング

ユーザータイプマッピングは、`org.jboss.ejb.plugins.cmp.jdbc.Mapper` インターフェースのインスタンスを指定することで JDBC カラムタイプからカスタム CMP フィールドタイプへのマッピングを可能にします。その定義を次に示します。

```
public interface Mapper
{
    /**
     * This method is called when CMP field is stored.
     * @param fieldValue - CMP field value
     * @return column value.
     */
    Object toColumnValue(Object fieldValue);

    /**
     * This method is called when CMP field is loaded.
     * @param columnValue - loaded column value.
     * @return CMP field value.
     */
    Object toFieldValue(Object columnValue);
}
```

典型的な使用事例は、整数タイプの、タイプセーフ Java 列挙型インスタンスへのマッピングです。**user-type-mappings** 要素のコンテンツモデルは、1 つ以上の **user-type-mapping** 要素から成ります。このコンテンツモデルは図30.20「**user-type-mapping** コンテンツモデル」に示してあります。

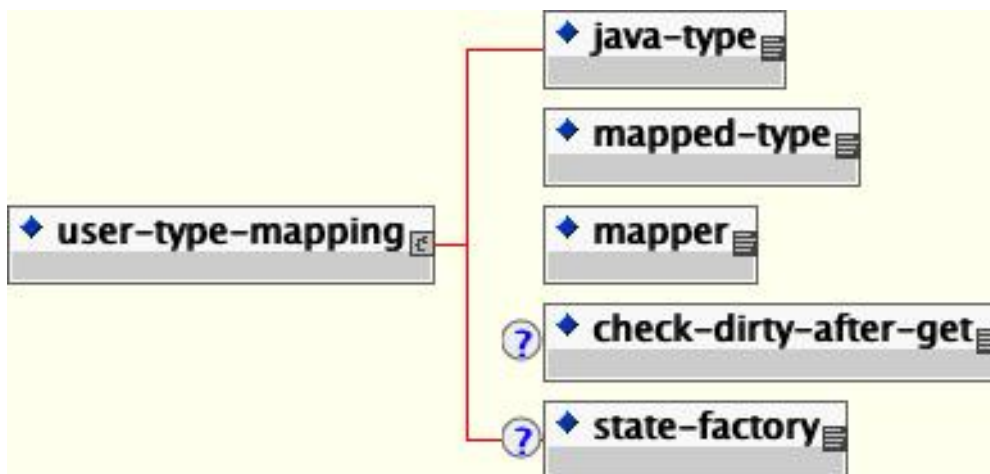


図30.20 user-type-mapping コンテンツモデル

- **java-type**: マッピングにおける CMP フィールドタイプの完全修飾名。
- **mapped-type**: マッピングにおけるデータベースタイプの完全修飾名。
- **mapper**: **java-type** と **mapped-type** 間の変換を処理する **Mapper** インターフェース実装の完全修飾名。



## パート **IV.** パフォーマンスチューニング



## 第31章 JBOSS ENTERPRISE APPLICATION PLATFORM 5 のパフォーマンスチューニング

### 31.1. はじめに

アプリケーションやサーバーのチューニングをせず、アプリケーションを開発し、Enterprise Application Platform にデプロイするだけでは最良のパフォーマンスは保証されません。パフォーマンスをチューニングするにはアプリケーションや Enterprise Application Platform のパフォーマンスを最良の状態に保ちながらアプリケーションが不必要なリソースを消費しないようにする必要があります。

パフォーマンスチューニングでは、アプリケーションの設計、ハードウェア／ネットワークプロファイル、オペレーティングシステム、アプリケーションソフトウェアの開発、テスト、デプロイメントのすべてが重要となります。パフォーマンスのボトルネックは、アプリケーションだけでなくこれらの要素が原因となる可能性があります。最近の調査によると、パフォーマンスの問題のほとんどはミドルウェアやオペレーティングシステムでなくアプリケーションが原因となっていることが分かりました。これには、コンピューターソフトウェアやハードウェア、ネットワークなどの信頼性を向上させた技術開発が関連している可能性もあります。

アプリケーションの設計を改善し、実装前にアプリケーションのパフォーマンスを再検討することは、実装後にボトルネックが発生しないようにするために大変重要です。パフォーマンスの再検討を行うには、テスト環境を設置してテストを実行し、テスト結果を分析する必要があります。効率的に再検討を行うには、アプリケーションのワークロードがピークになる時間を特定し、通常のワークロード時との差異を判断しなければなりません。ワークロードのピーク時は、特定の日や週ならびに、月、四半期、年間の特定期間である可能性があります。ピーク時のワークロードは一定期間の平均値よりはるかに大きい場合があるため、ピーク時のワークロードを平均値で判断しない方がよいでしょう。システム要件は、ワークロードの平均ではなくワークロードのピークによって制約されます。チューニングの際、満足のいくパフォーマンスが得られるまで、追加のテストとシステムのチューニングを実行することが推奨されます。

### 31.2. ハードウェアチューニング

JBoss Enterprise Application Platform におけるアプリケーションのパフォーマンスに見合った適切なハードウェア設定を開発するには、選択したハードウェア設定が他のアプリケーションやオペレーティングシステム全体のパフォーマンスに与える影響を理解する必要があります。

ハードウェアのパフォーマンスチューニングの問題を理解するには、システムのハードウェア構造を理解することも大変重要です。

#### 31.2.1. CPU (Central Processing Unit: 中央処理装置)

CPU とは以下で構成されるコンピュータの中央処理装置のことです。

- 命令を受信し、受信した命令の種類を判定する制御ユニット。
- 中間の処理情報を一時的に格納する CPU レジスター。
- 後続の実行可能タスクの場所を保持するプログラムカウンタ
- 現在実行しているタスクを格納する命令レジスター
- CPU が現在処理しているデータを保持する制限されたメモリである CPU キャッシュ。

CPU の構造を理解すると、CPU の仕様や動作を特定しやすくなります。AMD の CPU の詳細は、[http://www.amd.com/us-en/Processors/ProductInformation/0,,30\\_118,00.html](http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118,00.html) を参照してください。

Intel の CPU の詳細は、 [http://www.intel.com/products/processor/index.htm?id=subhdr+prod\\_proc](http://www.intel.com/products/processor/index.htm?id=subhdr+prod_proc) を参照してください。

### 31.2.2. RAM (Random Access Memory: ランダムアクセスメモリ)

RAM (ランダムアクセスメモリ) は、実行しているプログラムやデータを保持するために使用できる次のレベルのストレージです。RAM チップは CPU キャッシュより大きなストレージを提供し、コンピュータのパフォーマンスを向上できます。頻繁に使用されるデータやプログラムを RAM に格納すると、ハードディスクからよりも速く読み出しできるため、パフォーマンスを大幅に向上できます。

バッファークッシュを管理するようデータベース管理システムをチューニングする場合など、RAM は重要になります。この場合、他のアプリケーションやオペレーティングシステム全体のパフォーマンスに悪影響を与えないよう注意しながら、アプリケーションが迅速にアクセスできるよう頻繁に使用されるデータベース情報を RAM に格納します。

### 31.2.3. ハードディスク

CPU や RAM とは異なり、ハードディスクドライブは情報やデータを保持するのに電源が不要です。停電が発生すると、CPU や RAM に格納された情報は損失します。ハードディスクに格納された情報は維持されますが、停電時に実行されていた操作の種類によっては情報が破損することがあります。

ディスクドライブは機械式のヘッドを使用してディスクのシリンダーへ情報を読み書きするため、ディスクドライブから情報を読み出し、あるいは格納するのに時間がかかります。RAM や CPU の記憶領域へは同等の速度でアクセスできますが、ハードディスク上では情報が格納されているディスクブロックへディスクヘッドを移動する必要があります。

ディスクの非断片化やクリーンアップなどの実行は、ファイル読み出しやアプリケーション全体のパフォーマンスの向上を促進します。そのため、データの読み出しや処理を念頭に置きながらディスクストレージを注意して管理することが重要となります。また、できる限り最良のパフォーマンスを得るために、オペレーティングシステムに対して適切なファイルシステムを特定する必要があります。

主な構造の違いや異なるコンピューターハードウェアで発生する問題を理解すると、適切なハードウェアパフォーマンスやニーズに合った災害管理戦略の特定が容易になります。

## 31.3. オペレーティングシステムのパフォーマンスチューニング

最近のオペレーティングシステムの多くには、リアルタイムで CPU、メモリ、ハードディスク、ネットワーク使用率を監視するためのパフォーマンスチューニングツールやプロファイリングツールが含まれています。

Windows では、システムパフォーマンスのボトルネックの特定にタスクマネージャとパフォーマンスモニターを利用できます。Unix ベースのオペレーティングシステムでは、**top** と **ps** が使用されます。Red Hat Enterprise Linux や Fedora などの Linux ディストリビューションは、システムパフォーマンスの監視に便利なグラフィカルユーザーインターフェース **System Monitor** を提供します。

オペレーティングシステムのパフォーマンスチューニングとは個別の要求に対応するリソース管理のことです。オペレーティングシステムのスケーラビリティの管理には、さまざまな量の要求(少量から大容量)でリソース消費の管理が必要となります。

ユーザー要求への応答時間、データベース、ネットワーク、CPU、メモリパフォーマンスなど、ビジネスにとって重要な全体的な操作パフォーマンスメトリックスを特定し、テストする必要があります。また、可能な場合はリアルタイムまたはシステムデプロイメントでログするようにしなければなりません。

クラスター化された環境では、システムの障害を防ぐためにクラスターのパフォーマンスを理解して監視し、オーバーロードを初期に特定することが重要となります。

### 31.3.1. ネットワーキング

ネットワーク設定がパフォーマンスのボトルネックと原因となることがありますが、検出するのが難しい場合があります。例えば、アナログ回線上で Web アプリケーションのロードを試行し、同じページをブロードバンドのインターネット接続でロードするとブラウザにエラーメッセージが表示されることがあります。この例では、問題が帯域幅であってもエラーメッセージには明確に表示されないことがあります。

そのため、ネットワークアーキテクチャとインフラストラクチャーを特定することが、パフォーマンスチューニングとシステムのボトルネック修正に重要となります。

近年のオペレーティングシステムの多くはネットワークハードウェア設定ツールを提供します。ドライバーに拡張ネットワークハードウェア設定ツールを提供するハードウェアのメーカーもあります。

多くのオペレーティングシステムは微調整が可能な異なる通信プロトコルをサポートしています。ネットワークの設計には、TCP バッファメモリ領域、接続バッファ制限、確認オプションなどの要素が考慮されなければなりません。

Web サーバーにおける DNS ルックアップの有効化または無効化の決定は、パフォーマンスにも影響しますが、高セキュリティ環境では有効にする必要があるでしょう。これを考慮し、必要なリソースやハードウェアを割り当てることでシステムのパフォーマンスを向上できます。

## 31.4. JVM のチューニング

Java ベースのアプリケーションでは、JVM (Java 仮想マシン) のチューニングに関する知識を持っていたほうがよいでしょう。JVM で微調整が必要となるのは、メモリ不足例外の対応、Java ヒープ設定、ガベージコレクションなどです。詳細は、<http://java.sun.com/j2se/1.6.0/docs/> の JDK 6 ドキュメントを参照してください。

## 31.5. アプリケーションのチューニング

満足のいくアプリケーションパフォーマンスを得るには、適切なアプリケーション設計と開発の実践が重要となります。リモートサーバーのメモリ割り当てのタイムアウトやネットワーク問題などの要素によって、アプリケーションによるデータの読み書きや処理がパフォーマンスボトルネックの原因となることがあります。そのため、パフォーマンスボトルネックの特定には、各アプリケーションがどのように動作するかを理解することが重要となります。各コード部分が要する予想時間を設定すると、アプリケーションと比較する現実的なベンチマークの作成が容易になります。平均はピーク時によって大きく異なることがあるため、ベンチマークには平均ではなくアプリケーションの高低ピーク使用時を考慮するようにします。

さらに、ベンチマークツールを使用してアプリケーションをテストすると、パフォーマンスボトルネックの原因となることが多いコードの問題を迅速に特定することが可能です。起動などの要素によって発生するキャッシュやハードウェアの問題を特定するには、反復テストを実行するとよいでしょう。

JBoss Enterprise Application Platform の Web コンソール <http://localhost:8080/web-console/> は、デフォルトページの JVM ハードウェア環境統計で始まる監視ツールを提供し、監視ツールやスナップショットへのアクセスを提供します。



## 注記

パフォーマンスモニターは、1 秒毎の要求など全体的なアプリケーションパフォーマンスについて情報を提供します。JBoss Profilerなどのプロファイリングツールを使用すると、アプリケーションが要求への対応に要する時間や特定タイプの要求に対応する頻度などを調べることができます。例えば、メソッドが呼び出された回数や、メソッドで費した平均／最大／最短時間など、通常は個別のメソッドまで分類することができます。

また、アプリケーションのパフォーマンス問題を修正する時に他のアプリケーションのボトルネックを作らないよう注意することが重要となります。

### 31.5.1. インストルメンテーション

パフォーマンス分析にはアプリケーションをインストルメント化する必要があります。ほとんどの場合で、実際の実稼働ワークロードは予想したワークロードと異なるはずです。アプリケーションをインストルメント化しないと、データを明確に追跡することができません。企業規模やビジネスモデル、ビジネス環境は変動するため、アプリケーションのワークロードは時間と共に変更します。

以前は、インストルメント化がアプリケーションに組み込まれていなければなりませんでした。現在では、開発者がコーディングする必要がないインストルメント化の方法が多く存在します。インストルメント化に市販の製品や JBoss AOP フレームワークを使用することができます。コンテナの呼び出し統計を有効にすることも、統計のハイバネートも可能です。詳細は、AOP と Hibernate のプロジェクトページを参照してください。

連続的に取得したスレッドダンプ (各 Java Enterprise Application Platform スレッドの現在のコールスタックを含む) には、アプリケーションで何が起きているかを開発者が予想できる十分な情報が含まれています。アプリケーションがパフォーマンスの限界に達したら、スレッドダンプを取得するのが 1 つの手段です。パフォーマンスの問題が 5 分間継続する場合、スレッドダンプを毎分生成してみます。JVM の「jps -l」コマンドを使用して実行している Java アプリケーションとそのプロセス ID のリストを表示することもできます。この場合、「org.jboss.Main」アプリケーションのプロセス ID を確認し、「jstack ProcessID」コマンド (ProcessID は「org.jboss.Main」のプロセス ID に置き換え) を実行し、スレッドダンプを生成します。jstack コマンドの出力を転送し、出力を保存するようにしてください ("jstack ProcessID > threaddump1.txt")。

## 31.6. JBOSS ENTERPRISE APPLICATION PLATFORM のチューニング

本書の「はじめに」の項に記載されているコンポーネントを理解してから JBoss Enterprise Application Platform をチューニングするようにしてください。また、サーバー上でアプリケーションが使用するサービスについて理解してからそれらのサービスをチューニングしてパフォーマンスを向上するようにしてください。また、アプリケーションが使用する最適化されたデータベース接続を確立し、これらの接続をサーバー上に設定することが重要となります。本項では、JBoss Enterprise Application Platform のパフォーマンスチューニングについて説明します。

### 31.6.1. メモリ使用率

JBoss Enterprise Application Platform を含む Java アプリケーションのメモリ使用率は割り当てられるヒープ領域によって決まります。例えば、現在割り当てられている 1GB のヒープ領域を 800MB に縮小するとメモリフットプリントが縮小されます (十分なヘッドルームがある場合)。

JVM (Java Virtual Machine: Java 仮想マシン) はメモリのセグメント (生成) を管理します。ヒープ領域のセグメントを使い果たしてしまうと、Java OutOfMemoryError (OOM) が表示されます。Java OutOfMemoryError が発生する時、すべてのベットのが無効になります。不良なステートを修正するには



アプリケーションを再起動する必要があります。チューニングの一部として、負荷下におけるメモリヘッドルームの大きさを確認します。使用可能なメモリが少なすぎると、最大 Java メモリサイズを大きくする必要があります (必要な場合は 64 ビット JVM へ変換)。

メモリ不足は例外ではなくエラーであるため、メモリ不足が発生するとエラーが生成されますが java キャッチブロックでマスクされる可能性の低くなっています。ヒープ自体の一部でない永久メモリを使い果たしてしまうと OOME もスローされます。これは、ロードされたクラスの情報が維持される JVM 固有のメモリ領域です。クラスが大量にあると (大量の EJB ページや JSP ページなど) この領域をすぐ使い果す可能性があります。多くの場合、アプリケーションのデプロイや再デプロイに失敗します。下記のように永久メモリ領域を増やし、OOME が発生しないようにします。-**server** スイッチのデフォルトは 64 メガバイトです。

```
-XX:MaxPermSize=256m
```

これは、ヒープへ追加される分となります。この例の場合、ヒープは 512M、永久的な領域は 256M、合計 768 メガバイトになります。JVM 自体がシステムメモリの領域を多く使用し、スレッドごとのスタック領域 (大きさは OS による) もあることを念頭に置いてください。これらが HTTP/S プロセッサに加算されます。

-**XX:MaxPermSize?=256m -Xmx512m** (システムから割り当てられた合計 768 メガバイトです。VM の合計サイズではなく、VM が「C ヒープ」やスタック領域に割り当てる領域は含まれていません。)

HotSpot Java 仮想マシンは、アプリケーションのチューニングに使用するガベージコレクションの情報を収集するため使用できるさまざまなガベージコレクションツールによって構成されています。HotSpot 仮想マシンについての詳細は、<http://java.sun.com/javase/technologies/hotspot/> を参照してください。

Java 6 には、Java アプリケーションを監視できるようにする新しいツールが含まれています。Jmap は、Eclipse メモリアナライザーツール (<http://www.eclipse.org/mat/>) によって簡単に読み取れるヒープダンプファイル (<http://java.sun.com/javase/6/docs/technotes/tools/share/jmap.html>) を生成します。jstat ツール (<http://java.sun.com/javase/6/docs/technotes/tools/share/jstat.html>) は、Java メモリヒープの永続メモリ領域や他のセグメントの正確な状態を確認できるようにします。

### 31.6.1.1. VFS チューニング

VFS (仮想ファイルシステム: Vitrual File System) のさまざまなチューニングオプションを検索する際、存在するほとんどの情報は **VFSUtils** クラスにあります。このストリング定数は、VFS 動作の設定に使用できる異なるシステムプロパティ設定を示します。

- **jboss.vfs.forceCopy**

ネストされた jar の処理方法を定義します。forceCopy が true の場合 (デフォルト)、ネストされた jar の一時コピーを作成し、それに応じて VFS を再配線します。forceCopy が false の場合、メモリ内でネストされた jar を処理します。この場合、一時コピーは作成されませんが、より多くのメモリが消費されます。

- **jboss.vfs.forceNoReaper** は true と false のオプションがあり、デフォルトは false になります。

パフォーマンスを向上するため、別の reaper スレッドより非同期的に JAR ファイルを閉じます。JAR ファイルを同期的に閉じたい場合は、reaper スレッドを使用しないよう強制することができます。URI クエリと **noReaper** クエリセクションを使用して定義することもできます。

- **jboss.vfs.forceCaseSensitive** は true と false のオプションがあり、デフォルトは false になります。

有効にすると、ファイルパスの小文字と大文字を区別するよう強制できます。

- `jboss.vfs.optimizeForMemory` は `true` と `false` のオプションがあり、デフォルトは `false` になります。

有効にすると、メモリ内の JAR 処理を再命令するため、メモリ消費が増加します。

- 既存の一時ファイルを再使用するため (展開やワイヤリングを再実行しないようにするため) **`jboss.vfs.cache (org.jboss.virtual.spi.cache.helpers.NoopVFSCache)`** クラスを定義することができます。VFS レジストリはこのクラスの定義を使用して既存の VFS ルートを維持します。

VFS クラスからの **`VirtualFile`** ルックアップはこの **`singleton`** キャッシュインスタンスを使用して既存の一致するキャッシュエントリを確認します。一致することで、**`VirtualFile`** インスタンスの展開に使用できる既存の「先祖」も考慮します。

#### 31.6.1.1.1. VFS キャッシュのチューニング

前述の通り、VFS キャッシュが VFS ルートを保持し、VFS ルートより `VirtualFile` ルックアップが既存の `VirtualFile` インスタンスにアクセスできます。これは、特に一時ファイル (ネストされた JAR より作成) の場合に便利で、ネストされた JAR ファイルに関連するリソースの解凍を複数回実行する必要がありません。

VFS では **`org.jboss.virtual.spi.cache.helpers.NoopVFSCache`** が使用されるため、デフォルトではキャッシングは関係ありませんが、独自の実装を提供するか、既存の VFS 実装を選択することができます。

**`org.jboss.virtual.plugins.cache`** パッケージのキャッシュ実装は次の通りです。

- **`SoftRefVFSCache`**: マップのエントリ値としてソフト参照を使用します。
- **`WeakRefVFSCache`**: マップのエントリ値として弱参照を使用します。
- **`TimedVFSCache`**: `defaultLifetime` 後にキャッシュエントリをエビクトします。
- **`LRUVFSCache`**: LRU を基にキャッシュエントリをエビクトし 最小エントリと最大エントリを維持します。
- **`CombinedVFSCache`**: 少数の永久ルートを保持し、他の新しいルートは `realCache` プロパティにキャッシュされます。

JBoss Enterprise Application Platform では、監視し維持する永久ルートを認識するため **`CombinedVFSCache`** を使用します。MC の Bean 設定ファイルで設定する方法は次の通りです。

```
<bean name="VFSCache">
  <constructor
factoryClass="org.jboss.virtual.spi.cache.VFSCacheFactory"
factoryMethod="getInstance">
    <!-- Use the CombinedVFSCache implementation -->

<parameter>org.jboss.virtual.plugins.cache.CombinedVFSCache</parameter>
  </constructor>
  <start ignored="true"/>
  <property name="permanentRoots">
    <map keyClass="java.net.URL"
valueClass="org.jboss.virtual.spi.ExceptionHandler">
```

```

    <entry>
      <key>${jboss.lib.url}</key>
      <value><null/></value>
    </entry>
    <entry>
      <key>${jboss.common.lib.url}</key>
      <value><inject bean="VfsNamesExceptionHandler"/></value>
    </entry>
    <entry>
      <key>${jboss.server.lib.url}</key>
      <value><inject bean="VfsNamesExceptionHandler"/></value>
    </entry>
    <entry>
      <key>${jboss.server.home.url}deploy</key>
      <value><inject bean="VfsNamesExceptionHandler"/></value>
    </entry>
  </map>
</property>
<property name="realCache">
  <bean
class="org.jboss.virtual.plugins.cache.IterableTimedVFSCache"/>
</property>
</bean>

```

新しいカスタム VFS ルート (追加の **deploy** ディレクトリなど) はこの設定に追加しなければなりません。

#### 31.6.1.1.2. アノテーションスキャンのチューニング

現在、リソースのスキャンを制限する方法は 3 つあります。

- XML またはプログラムにて **ScanningMetaData** を提供します。
- **deployers/metadata-deployer-jboss-beans** の **GenScanDeployer** Bean に新しいデプロイメントフィルターを追加します。
- **deployers/metadata-deployer-jboss-beans** の **JBossCustomDeployDUFILTER** を変更します。

ScanningMetaData は **META-INF** ディレクトリにある **jboss-scanning.xml** ファイルから取得することもできます。このファイルの簡単な例は次の通りです。

```

<scanning xmlns="urn:jboss:scanning:1.0">
  <path name="myejbs.jar">
    <include name="com.acme.foo"/>
    <exclude name="com.acme.foo.bar"/>
  </path>
  <path name="my.war/WEB-INF/classes">
    <include name="com.acme.foo"/>
  </path>
</scanning>

```

この例では、デプロイメント内にパスをリストし、包含または除外するパッケージをリストします。明示的な包含がない場合は、除外されていないパッケージがすべて包含されます。パス要素がない場合は、次の例のようにすべてのパスが含まれます。

```
<scanning xmlns="urn:jboss:scanning:1.0">
<!-- Purpose: Disable scanning for annotations in contained deployment. --
>
</scanning>
```

**jboss-classloading.xml** ファイルに適切なメタデータを提供しスキャンを制限する方法もあります。詳細は、『JBoss Microcontainer User Guide』のクラスローディング層の章を参照してください。

### 31.6.2. データベース接続

データベースのパフォーマンスチューニングを行うには、初期データベースの概念スキーマを変更し、パフォーマンスを向上します。全体的なデータベース管理システムのパフォーマンスチューニングを行うには、種類に関係なくハードウェア (ハードディスク、CPU、RAM) を効果的かつ効率的に使用し、データベースの読み書きを向上します。

オペレーティングシステムによって設定されたリソースの制限が、データベース管理システムを制限することもあります。データベースの管理者は、データベースを分析して前述の要素を考慮しながらパフォーマンスボトルネックを特定し、ダーティバッファのディスクへの書き込み、チェックポイント、ログファイルロテーションなど必要なデータベース管理システムパラメータを調整します。場合によっては、データベースパフォーマンスの改善にハードウェアのアップグレードが必要となることもあります。

データベース接続の確立や管理はコストがかかります。トランザクションやクエリ毎にデータベースへの新しい接続を作成し、接続を閉じるアプリケーションはオーバーヘッドを大きく増加します。また、JBoss Enterprise Application Platform はデフォルトで、要求をキャンセルし例外をスローする前に要求を 30,000 ミリ秒 (30秒) 間キューに入れるため、接続プールが小さすぎるとアプリケーションを抑制してしまいます。

JBoss Enterprise Application Platform の `deploy` ディレクトリに設定できるデータソース定義に依存し、接続プールの設定を使用することが推奨されます。JBoss Enterprise Application Platform の接続プーリングを利用すると、JMX コンソールより接続使用を簡単に監視し、適切なサイズを判断することができます。データベース管理システムに接続を監視できるツールが同梱されていることもあります。

実装されたデータベースに従い、次のようにご利用中の設定にて `deploy` ディレクトリにデータソースファイルを作成するようにしてください。

```
$JBOSS_HOME/server/$PROFILE/deploy/
```

ファイル名は以下の形式でなければなりません：

```
<yourdatabasename>-ds.xml
```



#### 注記

ファイル名の末尾が **-ds.xml** でないと JBoss Enterprise Application Platform はデータソースファイルとして認識しません。例えば、PostgreSQL データベースのデータソースファイルの名前は **postgres-ds.xml** とします。





## 注記

外部データベースのデータソース定義ファイルの例は、  
**<JBoss\_Home>/docs/examples/jca** ディレクトリにあります。

### 31.6.3. クラスタリングのチューニング

クラスター内でアプリケーションが実行され、特にクラスター周囲で高ボリュームのステートレプリケーションが実行される場合、パフォーマンスを最適化できる方法がいくつかあります。パフォーマンスの最適化を行う場合は、変更により予想通りの結果を得られたか検証するため、常に変更前と変更後にアプリケーションの負荷テストを行うようにしてください。また、どの変更によりどの結果を得られるかを明確にするため、変更は1つずつ行ってください。

#### 31.6.3.1. 十分なネットワークバッファの確保

Enterprise Application Platform は、UDP ベースの IP マルチキャストを利用するため、クラスター間の通信に UDP を使用します。UDP の欠点は、TCP では OS ネットワークレベルで提供される無損失伝送を Java コードに実装する必要があることです。最良のパフォーマンスを得るには、ネットワーク層にドロップされる UDP パケットの頻度を低くすることが重要です。クラスターノードをホストしているマシン上のネットワークバッファのサイズが十分でないことが、パケット損失の原因となることが多くあります。Enterprise Application Platform のクラスタリングコードは、ソケットを開く時に十分な大きさの読み書きバッファを要求しますが、多くのオペレーティングシステムは (Window は例外) 最大サイズまでのバッファを提供するだけです。最大読み書きバッファサイズは OS レベルで設定できますが、デフォルト値は最良のパフォーマンスを実現するには低すぎます。そのため、Enterprise Application Platform のクラスタリングコードが要求するクラスターサイズを OS に設定することが簡単なチューニング手順となります。

最大許可バッファサイズを増やす設定手順は OS に固有しているため、手順は OS の説明書を参照してください。Linux システムでは、次のように **/etc/sysctl.conf** ファイルを編集してバッファサイズの最大値を設定し、マシンの再起動後も設定を維持するようにします。

```
# Allow a 25MB UDP receive buffer for JGroups
net.core.rmem_max = 26214400
# Allow a 1MB UDP send buffer for JGroups
net.core.wmem_max = 1048576
```

#### 31.6.3.2. クラスター間トラフィックの分離

ネットワークのリソースがアプリケーションのボトルネックとなっている場合は、外部の要求トラフィックよりクラスター内のトラフィックを分離すると全体のパフォーマンスを改善することができます。これには、サーバーマシン上に複数の NIC が必要となり、要求トラフィックとクラスター間トラフィックが別の NIC を使用するようにしなければなりません。ハードウェアの設定が終了すれば、Enterprise Application Platform のノードにクラスター間トラフィックには別のインターフェースを使用するよう指示することは簡単です。

```
./run.sh -c all -b 10.0.0.104 -Djgroups.bind_addr=192.168.100.104
```

上記の例では、192.168.100.104 インターフェースでクラスター間の JGroups トラフィックを実行するよう **-Djgroups.bind\_addr** 設定が Enterprise Application Platform を指示しています。**-b** は他のトラフィックすべてが 10.0.0.104 を使用するように指定します。

#### 31.6.3.3. JGroups のメッセージバンドリング

Enterprise Application Platform によって使用される JGroups のグループ通信ライブラリは、「メッ

セージバンドリング」と呼ばれる機能を提供します。メッセージバンドリングは、TCP ソケット上のネーグリング (nagling) と似ています。設定可能なバイト数が累積されるまで送信する前に小さなメッセージをキューに置き (設定可能な最大時間以内)、キューに置かれたメッセージをバンドルして 1 つの大きなメッセージとして送信します。バンドリングを使用すると高ボリュームの非同期セッションレプリケーションのパフォーマンスを大きく向上できます。しかし、バンドリングを使用するとクラスター化された Hibernate/JPA の 2 次キャッシュトラフィックなど他のクラスター間トラフィックの遅延が大幅に大きくなるため、デフォルトでは有効になっていません。

アプリケーションが高ボリュームのセッションレプリケーションを使用する場合 (Web セッションや EJB3 ステートフル Bean)、メッセージバンドリングが有効に設定されている JGroups チャネルを使用して分散キャッシング層を設定するとパフォーマンスを向上できるかもしれません。これを実行するには、**\$JBOSS\_HOME/server/\$PROFILE/deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-jboss-beans.xml** ファイルを編集します。Web セッションにデフォルトで使用するキャッシュの例は次のようになります。

```

. . .

<!-- Standard cache used for web sessions -->
<entry><key>standard-session-cache</key>
<value>
  <bean name="StandardSessionCacheConfig"
class="org.jboss.cache.config.Configuration">

. . .

  <!-- Replace standard 'udp' JGroups stack with
       one that uses message bundling -->
  <property name="multiplexerStack">udp-async</property>

. . .

```

FIELD 粒度の Web セッションでは、**field-granularity-session-cache** キーを用いて同じファイル内で同じ変更をキャッシュ設定に加えることができます。EJB3 ステートフルセッション Bean では、**sfsb-cache** キーを用いて同じファイル内で同じ変更をキャッシュに加えることができます。



#### 注記

セッションキャッシュに **udp-async** JGroups プロトコルスタックを使用すると、追加の JGroups トランスポートプロトコルが使用されます。そのため、標準の Enterprise Application Platform インストールよりも多くのソケットが開かれます。

#### 31.6.3.4. セッションキャッシュのバディレプリケーションを有効化

アプリケーションが 3 ノード以上のクラスターで Web セッションや EJB3 ステートフルセッション Bean の高ボリュームレプリケーションに関与する場合、Web セッションキャッシュとステートフルセッション Bean キャッシュで「バディレプリケーション」を有効にするとパフォーマンスを向上することができます。バディレプリケーションを使用すると、クラスターのすべてのノードにセッションのコピーをレプリケートせずに、コピーは設定可能な数の「バディ」ノードへのみレプリケートされます。

バディレプリケーションを有効にするには、**\$JBOSS\_HOME/server/\$PROFILE/deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-jboss-beans.xml** ファイルを編集します。Web セッションにデフォルトで使用するキャッシュの例は次の通りです。

■

```

. . .

<!-- Standard cache used for web sessions -->
<entry><key>standard-session-cache</key>
<value>
  <bean name="StandardSessionCacheConfig"
class="org.jboss.cache.config.Configuration">

. . .

  <property name="buddyReplicationConfig">
    <bean class="org.jboss.cache.config.BuddyReplicationConfig">

      <!-- Just set to true to turn on buddy replication -->
      <property name="enabled">true</property>

. . .

```

FIELD 粒度のWeb セッションでは、**field-granularity-session-cache** キーを用いて同じファイル内で同じ変更をキャッシュ設定に加えることができます。EJB3 ステートフルセッションBean では、**sfsb-cache** キーを用いて同じファイル内で同じ変更をキャッシュに加えることができます。

### 31.6.3.5. Web セッションレプリケーションのボリュームを低減

Web セッションレプリケーション向けにアプリケーションが設定されている場合、レプリケートされるデータ量を減らすとパフォーマンスを向上できます。これを実現するには、要求が実際にセッションをアップデートしていない場合にレプリケーションを行わないようにし、またレプリケーションを実際に変更されたセッションデータのみに制限します。アプリケーションを設定しレプリケートされるデータ量を制限する方法については、『HTTP コネクターロードバランシングガイド』の**replication-trigger** と **replication-granularity** を参照してください。

### 31.6.3.6. EJB3 ステートフルセッション Bean レプリケーションのボリュームを低減

EJB3 ステートフルセッション Bean レプリケーション向けにアプリケーションが設定されている場合、ステートを変更していない Bean 要求の後にレプリケーションを実行しないようにするとパフォーマンスを向上できます。これは、Bean クラスを **org.jboss.ejb3.cache.Optimized** インターフェースを実装すると制御することができます。詳細は「[EJB 3.0 でのステートフルセッション Bean](#)」を参照してください。

### 31.6.3.7. JPA/Hibernate の 2 次キャッシングの注意

JPA および Hibernate アプリケーションは、アプリケーションサーバーにデータベースのデータをキャッシングするとパフォーマンスが向上する場合があります。しかし、クラスター化されたアプリケーションでは、データキャッシングの有無を判断することはクラスター化されていないアプリケーションよりも複雑です。これは、クラスターの 1 つのノード上でデータベースの書き込みが行われた場合、キャッシュを更新または無効化するよう指示するからです。頻繁に更新されるデータでは、クラスター間メッセージのコストはキャッシングの利益よりも大きくなります。

そのため、Hibernate 2 次キャッシュにデータを保存するか決定する時に、慎重にクラスター化されたアプリケーションの負荷テストを行うようにしてください。すべてのエンティティタイプに対してキャッシングを有効にするのではなく、書き込み頻度の低さや複数のトランザクションがエンティティの読み取る可能性を基にエンティティタイプをランキングするようにします。その後、タイプごとにキャッシングを有効にし、パフォーマンスの影響をテストします。

クエリ結果セットのキャッシングを有効にする場合は十分注意するようにしてください。クエリの

キャッシングを有効にすると、データベースが書き込まれた時にクラスター化されたキャッシュは 2 つのメッセージをクラスターの周囲に送信する必要があります。これらのメッセージを使用して、書き込みによる影響があった可能性のあるクエリ結果がキャッシュより無効化されるようにします。書き込まれたエンティティタイプがキャッシュされたかどうかに関係なく、これらのメッセージは送信されなければなりません。メッセージのコストはクエリ結果キャッシングの利点によって相殺されます。繰り返しになりますが、必ずキャッシングの影響をテストするようにしてください。

### 31.6.3.8. JMX による JGroups の監視

Enterprise Application Platform のクラスタリングサービスがクラスター間通信で使用するため JGroups **Channel** を作成すると、そのチャンネルに関連する複数の MBean (チャンネル自体の MBean と各構成プロトコルの MBean) にて JMX サーバーも登録します。チャンネルのパフォーマンスに関連する動作を監視したいユーザーにとって、複数の MBean 属性は有用となるでしょう。

**jboss.jgroups:cluster=<cluster\_name>,protocol=UDP,type=protocol**

ネットワーク上におけるメッセージ送受信の統計情報や、受信メッセージをチャンネルのプロトコルスタックまで運ぶために使用する 2 つのスレッドプールの動作に関する統計を提供します。

送受信率に直接関係する便利な属性には、**MessagesSent**、**BytesSent**、**MessagesReceived**、**BytesReceived** などがあります。

普通の受信メッセージをプロトコルスタックまで運ぶために使用するスレッドプールの動作に関する便利な属性には、**IncomingPoolSize** や **IncomingQueueSize** などがあります。特別な非順序で帯域外のメッセージをプロトコルスタックまで運ぶために使用するスレッドプールの同等の属性には、**OOBPoolSize** や **OOBQueueSize** などがあります。標準的な JGroups 設定は OOB メッセージのキューを使用しないため、通常 **OOBQueueSize** は **0** となります。

**jboss.jgroups:cluster=<cluster\_name>,protocol=UNICAST,type=protocol**

ユニキャスト (ポイントツーポイント) メッセージを無損失で順番に配信するようにするプロトコルの動作に関する統計情報を提供します。

**NumRetransmissions** と **MessagesSent** との割合を追跡し、ピアによってメッセージが受信されず再送信が必要となる頻度を確認することができます。

**NumberOfMessagesInReceiveWindows** 属性を監視して、前のシーケンス番号を持つメッセージが受信されるまで受信側ノードで待機しているメッセージ数を追跡することができます。このメッセージ数が多いと、メッセージがドロップし、再送信の必要があることを意味します。

**jboss.jgroups:cluster=<cluster\_name>,protocol=NAKACK,type=protocol**

マルチキャスト (ポイントツーマルチポイント) メッセージを無損失で順番に配信するようにするプロトコルの動作に関する統計情報を提供します。

**XmitRequestsReceived** 属性を使用して、ノードが送信したメッセージの再送信を要求される頻度を追跡します。**XmitRequestsSent** を使用して、ノードがメッセージの再送信を要求する必要頻度を追跡します。

**jboss.jgroups:cluster=<cluster\_name>,protocol=FC,type=protocol**

高速のメッセージ送信側によって低速の受信側が圧倒されないようにするプロトコルの動作に関する統計情報を提供します。

受信側からのクレジットを待っている間にメッセージの送信を希望するスレッドがブロックするかを監視する有用な属性には、**Blockings**、**AverageTimeBlocked**、**TotalTimeBlocked** などがあります。

### 31.6.4. その他の主な設定

Enterprise Application Platform のパフォーマンスチューニングに必要なその他の主な設定には、HTTP 要求プールを設定する

`<JBoss_Home>/server/<your_configuration>/deployers/jbossweb.deployer/server.xml` ファイルなどがあります。

JBoss Enterprise Application Platform 5 は適切に大きさが設定される堅牢なスレッドプールを持っています。サーバーには、システムのスレッドプールを定義する

`<JBoss_Home>/server/<your_configuration>/conf` ディレクトリに `jboss-service.xml` ファイルがあります。プールに実行できるスレッドがない場合に動作を定義する設定があります。デフォルトでは、呼び出すスレッドがタスクを実行できるようにします。JMX コンソールよりシステムのキュー深さを監視し、プールを大きくする必要があるか判断することができます。

新しい管理コンソールを使用し、Enterprise Application Platform 環境の異なる側面を設定管理することができます。

**default** 設定は開発環境では適切ですが、実稼働環境では適切でないことがあります。default 設定では、コンソールのロギングが有効になっています。コンソールロギングは、すべてのログメッセージを IDE コンソールビューで表示できるため、特に IDE 内の開発には適しています。実稼働環境では、コンソールロギングは高負荷であるため推奨されません。必要な場合は、ロギングの詳細レベルを下げてください。ログするデータが少ないほど生成される I/O が少なくなるため、全体的なスループットが向上されます。

その他パフォーマンスチューニングにはキャッシング、クラスタリング、レプリケーションなどの側面があり、本書の各章で説明されています。

## パート VI. 添付資料

## 付録A ベンダー固有のデータソース定義

この添付資料では、JBoss Enterprise Application Platform 対応のデータベースに対するデータソース定義が含まれています。

### A.1. デプロイヤーの場所と名前

データベースのデプロイヤーはすべて、**\$JBOSS\_HOME/server/default/deploy/** ディレクトリに保存する必要があります。各デプロイヤーのファイルは**-ds.xml** のプレフィックスを付ける必要があります。例えば、Oracle データソースデプロイヤーは、**oracle-ds.xml** などの名称が可能です。正しくファイル名がつけられない場合、サーバーで検索されません。

### A.2. DB2

#### 例A.1 DB2 Local-XA

**\$db2\_install\_dir/java/db2jcc.jar** and  
**\$db2\_install\_dir/java/db2jcc\_license\_cu.jar** ファイルを  
**\$jboss\_install\_dir/server/default/lib** ディレクトリにコピーします。DB2 バージョン 8.1 以降に含まれている DB2 Universal JDBC ドライバーを使うと、レガシーの CLI ドライバーの一部である **db2java.zip** ファイルは、通常必要ありません。

```
<datasources>

<local-tx-datasource>
  <jndi-name>DB2DS</jndi-name>
  <!-- Use the syntax 'jdbc:db2:yourdatabase' for jdbc type 2
connection -->
  <!-- Use the syntax 'jdbc:db2://serveraddress:port/yourdatabase' for
jdbc type 4 connection -->
  <connection-
url>jdbc:db2://serveraddress:port/yourdatabase</connection-url>
  <driver-class>com.ibm.db2.jcc.DB2Driver</driver-class>
  <user-name>x</user-name>
  <password>y</password>
  <min-pool-size>0</min-pool-size>
  <!-- sql to call when connection is created
<new-connection-sql>some arbitrary sql</new-connection-sql>
  -->

  <!-- sql to call on an existing pooled connection when it is
obtained from pool
  <check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
  -->

  <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml
(optional) -->
  <metadata>
    <type-mapping>DB2</type-mapping>
  </metadata>
</local-tx-datasource>

</datasources>
```

## 例A.2 DB2 XA

`$db2_install_dir/java/db2jcc.jar` と  
`$db2_install_dir/java/db2jcc_license_cu.jar` ファイル  
を `$jboss_install_dir/server/default/lib` ディレクトリにコピーします。

DB2 v8.1 修正パック 14 (および該当のDB2 v8.2 修正パック 7) で XA 向け DB2 Universal JDBC ドライバー (type 4) を使う場合、**db2java.zip** ファイルが必要になります。

```
<datasources>
<!--
  XADatasource for DB2 v8.x (app driver)
-->

<xa-datasource>
  <jndi-name>DB2XADS</jndi-name>

  <xa-datasource-class>com.ibm.db2.jcc.DB2XADataSource</xa-
datasource-class>
  <xa-datasource-property name="ServerName">your_server_address</xa-
datasource-property>
  <xa-datasource-property name="PortNumber">your_server_port</xa-
datasource-property>
  <xa-datasource-property name="DatabaseName">your_database_name</xa-
datasource-property>
  <!-- DriverType can be either 2 or 4, but you most likely want to
use the JDBC type 4 as it doesn't require a DB" client -->
  <xa-datasource-property name="DriverType">4</xa-datasource-
property>
  <!-- If driverType 4 is used, the following two tags are needed -->
  <track-connection-by-tx></track-connection-by-tx>
  <isSameRM-override-value>>false</isSameRM-override-value>

  <xa-datasource-property name="User">your_user</xa-datasource-
property>
  <xa-datasource-property name="Password">your_password</xa-
datasource-property>

  <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml
(optional) -->
  <metadata>
    <type-mapping>DB2</type-mapping>
  </metadata>
</xa-datasource>
</datasources>
```



## 例A.3 AS/400 上の DB2

```

    <?xml version="1.0" encoding="UTF-8"?>

    <!--
    ===== --
    >
    <!--
    -->
    <!--  JBoss Server Configuration
    -->
    <!--
    -->
    <!--
    ===== --
    >

    <!-- $Id: db2-400-ds.xml,v 1.1.4.2 2004/10/27 18:44:10 pilhuhn Exp $ -->

    <!-- You need the jt400.jar that is delivered with IBM iSeries Access or
    the
    OpenSource Project jtopen.

    [systemname] Hostame of the iSeries
    [schema]      Default schema is needed so jboss could use metadat to
    test if the tables exists
    -->

    <datasources>
      <local-tx-datasource>
        <jndi-name>DB2-400</jndi-name>
        <connection-url>jdbc:as400://[systemname]/[schema];extended
dynamic=true;package=jbpkg;package cache=true;package
library=jboss;errors=full</connection-url>
        <driver-class>com.ibm.as400.access.AS400JDBCDriver</driver-class>
        <user-name>[username]</user-name>
        <password>[password]</password>
        <min-pool-size>0</min-pool-size>
        <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>
        -->

        <!-- sql to call on an existing pooled connection when it is
    obtained from pool
        <check-valid-connection-sql>some arbitrary sql</check-valid-
    connection-sql>
        -->
        <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
    (optional) -->
        <metadata>

```

```

        <type-mapping>DB2/400</type-mapping>
    </metadata>

</local-tx-datasource>

</datasources>

```

#### 例A.4 AS/400 「ネイティブ」での DB2

ネイティブのJDBCドライバは、IBM Developer Kit for Java (57xxJV1) の一部として同梱されています。これは、SQL CLI (*Call Level Interface*) にネイティブのメソッド呼出しを行うことで実装されており、i5/OS JVM のみ稼働します。登録クラス名は、**com.ibm.db2.jdbc.app.DB2Driver** となっており、URL サブプロトコルは **db2** です。詳細情報については、<http://www-03.ibm.com/systems/i/software/toolbox/faqjdbc.html#faqA1> のJDBC FAQKS を参照してください。

```

    <?xml version="1.0" encoding="UTF-8"?>
<!--
===== --
>
<!--
-->
<!-- JBoss Server Configuration
-->
<!--
-->
<!--
===== --
>
<!-- $Id: db2-400-ds.xml,v 1.1.4.2 2004/10/27 18:44:10 pilhuhn Exp $
-->
<!-- You need the jt400.jar that is delivered with IBM iSeries Access or
the
OpenSource Project jtopen.
[systemname] Hostame of the iSeries
[schema]      Default schema is needed so jboss could use metadat to
test if the tables exists -->
<datasources>
  <local-tx-datasource>
    <jndi-name>DB2-400</jndi-name>
    <connection-url>jdbc:db2://[systemname]/[schema];extended
dynamic=true;package=jbpkg;package cache=true;package
library=jboss;errors=full</connection-url>
    <driver-class>com.ibm.db2.jdbc.app.DB2Driver</driver-class>
    <user-name>[username]</user-name>
    <password>[password]</password>
    <min-pool-size>0</min-pool-size>
    <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>
  >

```

```

        <!-- sql to call on an existing pooled connection when it is
        obtained from pool
        <check-valid-connection-sql>some arbitrary sql</check-valid-
        connection-sql>        -->
        <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
        (optional) -->
        <metadata>
            <type-mapping>DB2/400</type-mapping>
        </metadata>
    </local-tx-datasource>
</datasources>

```

## ヒント

- このドライバーは、ジョブの CCSID を区別しますが、**CCSID=37** で問題なく機能します。
- **[systemname]** は、**WRKRDBDIRE** のエントリを **\*local** のように定義する必要があります。

## A.3. ORACLE

### 例A.5 Oracle Local-TX Datasource

```

    <?xml version="1.0" encoding="UTF-8"?>

    <!--
    ===== --
    >
    <!--
    -->
    <!--  JBoss Server Configuration
    -->
    <!--
    -->
    <!--
    ===== --
    >

    <!-- $Id: oracle-ds.xml,v 1.6 2004/09/15 14:37:40 loubiansky Exp $ -->
    <!--
    ===== -->
    <!--  Datasource config for Oracle originally from Steven Coy
    -->
    <!--
    ===== -->

    <datasources>

```

```

<local-tx-datasource>
  <jndi-name>OracleDS</jndi-name>
  <connection-
url>jdbc:oracle:thin:@youroraclehost:1521:yoursid</connection-url>
  <!--
    See on WIKI page below how to use Oracle's thin JDBC driver to connect
    with enterprise RAC.
    -->
  <!--
    Here are a couple of the possible OCI configurations.
    For more information, see
    http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/java.9
    20/a96654/toc.htm

    <connection-url>jdbc:oracle:oci:@youroracle-tns-name</connection-url>
    or
    <connection-url>jdbc:oracle:oci:@(description=(address=
    (host=youroraclehost)(protocol=tcp)(port=1521))(connect_data=
    (SERVICE_NAME=yourservicename))</connection-url>

    Clearly, its better to have TNS set up properly.
    -->
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
    <user-name>x</user-name>
    <password>y</password>

    <min-pool-size>5</min-pool-size>
    <max-pool-size>100</max-pool-size>

    <!-- Uses the pingDatabase method to check a connection is still
    valid before handing it out from the pool -->
    <!--valid-connection-checker-class-
    name>org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker
    </valid-connection-checker-class-name-->
    <!-- Checks the Oracle error codes and messages for fatal errors -->
    <exception-sorter-class-
    name>org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</except
    ion-sorter-class-name>
    <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>
    -->

    <!-- sql to call on an existing pooled connection when it is
    obtained from pool - the OracleValidConnectionChecker is preferred
    <check-valid-connection-sql>some arbitrary sql</check-valid-
    connection-sql>
    -->

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
    (optional) -->
    <metadata>
      <type-mapping>Oracle9i</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>

```

## 例A.6 Oracle XA Datasource

```

    <?xml version="1.0" encoding="UTF-8"?>

<!--
===== --
>
<!--
-->
<!--  JBoss Server Configuration
-->
<!--
-->
<!--
===== --
>

<!-- $Id: oracle-xa-ds.xml,v 1.13 2004/09/15 14:37:40 loubyansky Exp $ -
->

<!--
===== --
>
<!-- ATTENTION:  DO NOT FORGET TO SET Pad=true IN transaction-
service.xml -->
<!--
===== --
>

<datasources>
  <xa-datasource>
    <jndi-name>XAOracleDS</jndi-name>
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>false</isSameRM-override-value>
    <xa-datasource-class>oracle.jdbc.xa.client.OracleXADataSource</xa-
datasource-class>
    <xa-datasource-property name="URL">jdbc:oracle:oci8:@tc</xa-
datasource-property>
    <xa-datasource-property name="User">scott</xa-datasource-property>
    <xa-datasource-property name="Password">tiger</xa-datasource-
property>
    <!-- Uses the pingDatabase method to check a connection is still
valid before handing it out from the pool -->
    <!--valid-connection-checker-class-
name>org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker
</valid-connection-checker-class-name-->
    <!-- Checks the Oracle error codes and messages for fatal errors -->
    <exception-sorter-class-

```

```

name>org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</exception-sorter-class-name>
  <!-- Oracles XA datasource cannot reuse a connection outside a transaction once enlisted in a global transaction and vice-versa -->
  <no-tx-separate-pools></no-tx-separate-pools>

  <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml (optional) -->
  <metadata>
    <type-mapping>Oracle9i</type-mapping>
  </metadata>
</xa-datasource>

<mbean
code="org.jboss.resource.adapter.jdbc.vendor.OracleXAExceptionFormatter"
  name="jboss.jca:service=OracleXAExceptionFormatter">
  <depends optional-attribute-name="TransactionManagerService">jboss:service=TransactionManager</depends>
</mbean>

</datasources>

```

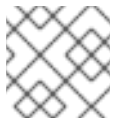
### 例A.7 Enterprise RAC と Oracle の Thin JDBC Driver

<connection-url> の追加設定で、Oracle の Thin JDBC ドライバーを使って Enterprise RAC に接続します。この二つのホスト名があると、基盤の物理データベースに対する負荷分散やフェイルオーバーが行えるようになります。

```

...
<connection-url>jdbc:oracle:thin:@(description=(address_list=
(load_balance=on)(failover=on)(address=(protocol=tcp)(host=xxxxhost1)
(port=1521))(address=(protocol=tcp)(host=xxxxhost2)(port=1521)))
(connect_data=(service_name=xxxxsid)(failover_mode=(type=select)
(method=basic))))</connection-url>
...

```



#### 注記

この例は、Oracle 10g に対してのみテストされています。

### A.3.1. Oracle 10g JDBC ドライバーの変更

`jboss-service.xml` ファイルで **Pad** オプションを有効にする必要がなくなりました。さらに、`<no-tx-separate-pool/>` も必要なくなっています。

### A.3.2. Oracle 10g に対するタイプマッピング

Oracle 10g データソース設定には、Oracle 9i のタイプマッピングを指定する必要があります。

#### 例A.8 Oracle9i タイプマッピング

```
.....  
<metadata>  
  <type-mapping>Oracle9i</type-mapping>  
</metadata>  
.....
```

### A.3.3. 基盤の Oracle Connection オブジェクトをリトリーブ

#### 例A.9 Oracle Connection オブジェクト

```
Connection conn = myJBossDatasource.getConnection();  
WrappedConnection wrappedConn = (WrappedConnection)conn;  
Connection underlyingConn = wrappedConn.getUnderlyingConnection();  
OracleConnection oracleConn = (OracleConnection)underlyingConn;
```

### A.3.4. Oracle 11g の制限

Oracle 11g R2 (RAC およびスタンドアローンの両方)では、`LockMode.UPGRADE` (更新など)を使った複雑なクエリを行うと "No more data to read from socket" のエラーを引き起こしてしまう可能性があります。回避策は、このようなクエリで`LockMode.UPGRADE`を使わないようにすることです。詳細は Oracle のバグ番号 9219636 を参照してください。

## A.4. SYBASE

#### 例A.10 Sybase Datasource

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/SybaseDB</jndi-name>
    <!-- Sybase jConnect URL for the database.
NOTE: The hostname and port are made up values. The optional
database name is provided, as well as some additinal Driver
parameters.
-->
    <connection-url>jdbc:sybase:Tds:host.at.some.domain:5000/db_name?
JCONNECT_VERSION=6</connection-url>
    <driver-class>com.sybase.jdbc2.jdbc.SybDataSource</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.SybaseExceptionSorter</excep
tion-sorter-class-name>
    <!-- sql to call when connection is created
<new-connection-sql>some arbitrary sql</new-connection-sql>
-->

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
<check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
-->

    <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>Sybase</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

[1]

#### A.4.1. Sybase の制限

Sybase では、認識しておくべき例外設定がいくつか存在します。

##### トランザクションにおけるDDLステートメント

Enterprise Platform の中心部分であるHibernateがあると、データベースがトランザクション内でDDLステートメントに対応するか否かをSQLダイアレクトにより決定することができます。Sybaseはこれをオーバーライドしません。デフォルトはJDBCメタデータをクエリしDDLがトランザクション内で許可されているか参照します。しかし、Sybaseはこのオプションを使うかのレポートを修正しません。



Sybaseは、ロック関連の問題があるため、トランザクションにおけるDDLステートメントの利用を推奨していません。**ddl in tran**オプションの有効化あるいは無効化の方法については、Sybase文書を参照してください。

**Sybase は、値が基盤となるカラムの制限を越えても例外をスローしません。**

Sybase ASE は、Parameterized SQLを利用している場合例外をスローしません。デフォルトでは、**jconn3.jar** はParameterized SQLを挿入に使っており、値が基盤カラムの制限を越えても例外はスローされません。例外がスローされないため、Hibernateは挿入が失敗したことを通知できなくなっています。Parameterized SQLの代わりにDynamic Prepare を使うことで、ASEは例外をスローします。Hibernateはこの例外をとらえ、それに従い動作します。

このような理由からHibernateの設定ファイルでは、**Dynamic prepare** パラメーターを**true**に設定してください。

```
<property name="connection.url">jdbc:sybase:Tds:aurum:1503/masterDb?
DYNAMIC_PREPARE=true</property>
```

**jconn4.jar** はデフォルトではDynamic Prepareを利用します。

**SchemaExportは、連鎖トランザクションモードにてストアプロシージャを作成できません。**

Sybaseでは、SchemaExport を使い連鎖トランザクションモードにてストアプロシージャを作成することができません。この問題の回避策は、以下のコードを新規のストアプロシージャの直後に追加します。

```

                <database-object>
    <create>
        sp_procxmode paramHandling, 'chained'
    </create>
    <drop/>
</database-object>
```

## A.5. MICROSOFT SQL SERVER

これらのドライバーを評価するには単純な JSP ページを使って Microsoft SQL Server に同梱されている **pubs** データベースにクエリを行うことができます。

[files/mssql-test.zip](#) にある WAR アーカイブを **/deploy** に移動し、ご利用中の Web ブラウザーで<http://localhost:8080/test/test.jsp> に移動します。

### 例A.11 DataDirect ドライバーを使った Local-TX Datasource

この例は、<http://www.datadirect.com> のDataDirect Connect for JDBC ドライバーを使っています。

```

    <datasources>
    <local-tx-datasource>
        <jndi-name>MerliaDS</jndi-name>
```

```

    <connection-
url>jdbc:datadirect:sqlserver://localhost:1433;DatabaseName=jboss</conne
ction-url>
    <driver-class>com.ddtek.jdbc.sqlserver.SQLServerDriver</driver-
class>
    <user-name>sa</user-name>
    <password>sa</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
        <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
</local-tx-datasource>

</datasources>

```

#### 例A.12 Merlia ドライバーを使った Local-TX Datasource

この例では、<http://www.inetsoftware.de> からのMerlia JDBC Driver ドライバーを使っています。

```

    <datasources>
<local-tx-datasource>
    <jndi-name>MerliaDS</jndi-name>
    <connection-url>jdbc:inetdae7:localhost:1433?
database=pubs</connection-url>
    <driver-class>com.inet.tds.TdsDataSource</driver-class>
    <user-name>sa</user-name>
    <password>sa</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
        <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
</local-tx-datasource>

</datasources>

```

#### 例A.13 Merlia ドライバーを使った XA Datasource

この例では、<http://www.inetsoftware.de> からのMerlia JDBC Driver ドライバーを使っています。

```

        <datasources>
    <xa-datasource>
        <jndi-name>MerliaXADS</jndi-name>
        <track-connection-by-tx></track-connection-by-tx>
        <isSameRM-override-value>false</isSameRM-override-value>
        <xa-datasource-class>com.inet.tds.DTCDatasource</xa-datasource-
class>
        <xa-datasource-property name="ServerName">localhost</xa-datasource-
property>
        <xa-datasource-property name="DatabaseName">pubs</xa-datasource-
property>
        <user-name>sa</user-name>
        <password>sa</password>

        <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
        <metadata>
            <type-mapping>MS SQLSERVER2000</type-mapping>
        </metadata>
    </xa-datasource>

</datasources>

```

### A.5.1. Microsoft JDBC ドライバー

MS SQL Server 向けの Microsoft JDBC ドライバーには 2 種類あります。

- SQL Server 2000 で利用可能な SQL Server 2000 Driver for JDBC Service Pack 3
- SQL 2000 あるいは 2005 のいずれかで利用可能な Microsoft SQL Server 2005 JDBC Driver。このバージョンには様々な修正が含まれており、JBoss Hibernate で認定されています。このドライバーは JDK 5 で動作します。

ドライバーのディストリビューションに含まれる **release.txt** を読み、特に、2005 で導入された新規パッケージ名、同じアプリケーションサーバーで両方のドライバーを使った場合に起こり得るコンフリクトなど、これらのドライバーの違いを理解するようにしてください。

#### 例A.14 Microsoft SQL Server 2000 での Local-TX Datasource

```

    <?xml version="1.0" encoding="UTF-8"?>

    <datasources>
        <local-tx-datasource>
            <jndi-name>MSSQL2000DS</jndi-name>
            <connection-
url>jdbc:microsoft:sqlserver://localhost:1433;SelectMethod=cursor;Databa
seName=pubs</connection-url>
            <driver-class>com.microsoft.jdbc.sqlserver.SQLServerDriver</driver-
class>

```

```
<user-name>sa</user-name>
<password>jboss</password>

<!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
<metadata>
  <type-mapping>MS SQLSERVER2000</type-mapping>
</metadata>
</local-tx-datasource>

</datasources>
```

#### 例A.15 Microsoft SQL Server 2005 での Local-TX Datasource

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>MSSQL2005DS</jndi-name>
    <connection-
url>jdbc:sqlserver://localhost:1433;DatabaseName=pubs</connection-url>
    <driver-class>com.microsoft.sqlserver.jdbc.SQLServerDriver</driver-
class>
    <user-name>sa</user-name>
    <password>jboss</password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

#### 例A.16 Microsoft SQL Server 2005 での XA Datasource

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <xa-datasource>
```

```

    <jndi-name>MSSQL2005XADS</jndi-name>
    <track-connection-by-tx></track-connection-by-tx>
    <isSameRM-override-value>>false</isSameRM-override-value>
    <xa-datasource-
class>com.microsoft.sqlserver.jdbc.SQLServerXADataSource</xa-datasource-
class>
    <xa-datasource-property name="ServerName">localhost</xa-datasource-
property>
    <xa-datasource-property name="DatabaseName">pubs</xa-datasource-
property>
    <xa-datasource-property name="SelectMethod">cursor</xa-datasource-
property>
    <xa-datasource-property name="User">sa</xa-datasource-property>
    <xa-datasource-property name="Password">jboss</xa-datasource-
property>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
        <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
</xa-datasource>

</datasources>

```

### A.5.2. JSQL ドライバー

#### 例A.17 JSQL ドライバー

```

    <?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>JSQLDS</jndi-name>
    <connection-
url>jdbc:JSQLConnect://localhost:1433/databaseName=testdb</connection-
url>
    <driver-class>com.jnetdirect.jsql.JSQLDriver</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <!-- sql to call when connection is created
<new-connection-sql>some arbitrary sql</new-connection-sql>
-->

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
<check-valid-connection-sql>some arbitrary sql</check-valid-

```

```

connection-sql>
-->

</local-tx-datasource>

</datasources>

```

### A.5.3. jTDS JDBC ドライバー

jTDS は、オープンソースで 100% Pure Java (type 4) の Microsoft SQL Server (6.5、7、2000、2005) および Sybase (10、11、12、15) 向け JDBC 3.0 ドライバーです。jTDS は FreeTDS をベースとしており、現在、Microsoft SQL Server と Sybase に対しすぐに本番利用できる JDBC ドライバーの中で最速となっています。jTDS は 100% JDBC 3.0 互換があり、forward-onlyかつスクロール可能／更新可能な **ResultSet**、同時並行ステートメント (完全に独立)、**DatabaseMetaData** および **ResultSetMetaData** メソッドすべての実装に対応しています。

<http://jtds.sourceforge.net/> から jTDS をダウンロードします。

#### 例A.18 jTDS Local-TX Datasource

```

<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>jtdsDS</jndi-name>
    <connection-
url>jdbc:jtds:sqlserver://localhost:1433;databaseName=pubs</connection-
url>
    <driver-class>net.sourceforge.jtds.jdbc.Driver</driver-class>
    <user-name>sa</user-name>
    <password>jboss</password>

    <!-- optional parameters -->
    <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-
isolation>
    <min-pool-size>10</min-pool-size>
    <max-pool-size>30</max-pool-size>
    <idle-timeout-minutes>15</idle-timeout-minutes>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <new-connection-sql>select 1</new-connection-sql>
    <check-valid-connection-sql>select 1</check-valid-connection-sql>
    <set-tx-query-timeout></set-tx-query-timeout>
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

**例A.19 jTDS XA Datasource**

```

    <?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <xa-datasource>
    <jndi-name>jtdsXADS</jndi-name>
    <xa-datasource-class>net.sourceforge.jtds.jdbcx.JtdsDataSource</xa-
datasource-class>
    <xa-datasource-property name="ServerName">localhost</xa-datasource-
property>
    <xa-datasource-property name="DatabaseName">pubs</xa-datasource-
property>
    <xa-datasource-property name="User">sa</xa-datasource-property>
    <xa-datasource-property name="Password">jboss</xa-datasource-
property>

    <!--
      When set to true, emulate XA distributed transaction support. Set to
      false to use experimental
      true distributed transaction support. True distributed transaction
      support is only available for
      SQL Server 2000 and requires the installation of an external stored
      procedure in the target server
      (see the README.XA file in the distribution for details).
    -->
    <xa-datasource-property name="XaEmulation">true</xa-datasource-
property>

    <track-connection-by-tx></track-connection-by-tx>

    <!-- optional parameters -->
    <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-
isolation>
    <min-pool-size>10</min-pool-size>
    <max-pool-size>30</max-pool-size>
    <idle-timeout-minutes>15</idle-timeout-minutes>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <new-connection-sql>select 1</new-connection-sql>
    <check-valid-connection-sql>select 1</check-valid-connection-sql>
    <set-tx-query-timeout></set-tx-query-timeout>
    <metadata>
      <type-mapping>MS SQLSERVER2000</type-mapping>
    </metadata>
  </xa-datasource>
</datasources>

```

#### A.5.4. "Invalid object name 'JMS\_SUBSCRIPTIONS' Exception

起動時に [例A.20「JMS\\_SUBSCRIPTIONS Exception」](#) にあるような例外を受け取る場合、[例A.21「SelectMethod の指定」](#) に記載の接続 URL にて **SelectMethod** を指定します。

##### 例A.20 JMS\_SUBSCRIPTIONS Exception

```
17:17:57,167 WARN [ServiceController] Problem starting
service jboss.mq.destination:name=testTopic,service=Topic
org.jboss.mq.SpyJMSEException: Error getting durable
subscriptions for topic TOPIC.testTopic; - nested throwable:
(java.sql.SQLException: [Microsoft][SQLServer 2000 Driver for JDBC]
[SQLServer]Invalid object name 'JMS_SUBSCRIPTIONS'.)
    at
org.jboss.mq.sm.jdbc.JDBCStateManager.getDurableSubscriptionIdsForTopic(
JDBCStateManager.java:290)
    at
org.jboss.mq.server.JMSDestinationManager.addDestination(JMSDestinationM
anager.java:656)
```

##### 例A.21 SelectMethod の指定

```
<connection-
url>jdbc:microsoft:sqlserver://localhost:1433;SelectMethod=cursor;Databa
seName=jboss</connection-url>
```

## A.6. MYSQL DATASOURCE

### A.6.1. ドライバーのインストール

#### 手順A.1 ドライバーのインストール

1. <http://www.mysql.com/products/connector/j/> からドライバーをダウンロードします。MySQL のご利用のバージョンに応じてドライバーを選択するようにしてください。
2. ドライバーの zip あるいは TAR ファイルを展開し、**.jar**を探します。
3. **.jar** ファイルを **\$JBOSS\_HOME/server/config\_name/lib** に移動します。
4. **\$JBOSS\_HOME/docs/examples/jca/mysql-ds.xml** というサンプルのデータソースデプロイヤーファイルを **\$JBOSS\_HOME/server/config\_name/deploy/** にコピーし、テンプレートとして利用します。

#### MySQL の制限



## ミリ秒およびマイクロ秒の測定

現在MySQL は、**TIME** および**TIMESTAMP**などの値をデータベースに返す際ミリ秒およびマイクロ秒の測定に対応していません。これらの測定に依存するテストは失敗します。

### A.6.2. MySQL Local-TX Datasource

#### 例A.22 MySQL Local-TX Datasource

この例では、**autoReconnect** を有効にし、ポート 3306 で **localhost** にてホストされるデータベースを使います。Transactions サポートが必要でない限り、これは推奨される設定ではありません。

```
<datasources>
<local-tx-datasource>

  <jndi-name>MySQLDS</jndi-name>

  <connection-url>jdbc:mysql://localhost:3306/database</connection-
url>
  <driver-class>com.mysql.jdbc.Driver</driver-class>

  <user-name>username</user-name>
  <password>secret</password>

  <connection-property name="autoReconnect">true</connection-
property>

  <!-- Typemapping for JBoss 4.0 -->
  <metadata>
    <type-mapping>mySQL</type-mapping>
  </metadata>

</local-tx-datasource>
</datasources>
```

### A.6.3. 名前付きパイプを利用した MySQL

#### 例A.23 名前付きパイプを利用した MySQL

この例は、ローカルでホスティングされているデータベースを使いますが、TCP/IP の代わりに名前つきパイプを使います。

```
<datasources>
<local-tx-datasource>

  <jndi-name>MySQLDS</jndi-name>
  <connection-url>jdbc:mysql://./database</connection-url>
```

```

<driver-class>com.mysql.jdbc.Driver</driver-class>

<user-name>username</user-name>
<password>secret</password>

<connection-property
name="socketFactory">com.mysql.jdbc.NamedPipeSocketFactory</connection-
property>

<metadata>
  <type-mapping>mySQL</type-mapping>
</metadata>

</local-tx-datasource>
</datasources>

```

## A.7. POSTGRESQL

### 例A.24 PostgreSQL Local-TX Datasource

```

<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>PostgresDS</jndi-name>
    <connection-url>jdbc:postgresql://[servername]:[port]/[database
name]</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>
  -->

    <!-- sql to call on an existing pooled connection when it is
obtained from pool
  <check-valid-connection-sql>some arbitrary sql</check-valid-
connection-sql>
  -->

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
      <type-mapping>PostgreSQL 8.0</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

## 重要

XA Transactionsは、PostgreSQL v8.4 および v8.2で`max_prepared_transactions`がデフォルト値 (0) を使う場合、却下されます。

PostgreSQLユーザーの文書では、全セッションに準備済みのトランザクションが持てるように`max_connections`の値と同等以上の`max_prepared_transactions` 値を設定するように推奨しています。

詳細情報は、<http://www.postgresql.org/docs/8.4/interactive/runtime-config-resource.html#GUC-MAX-PREPARED-TRANSACTIONS>にあるPostgreSQL v8.4 ユーザー文書を参照してください。

### 例A.25 PostgreSQL XA Datasource

この設定は、PostgreSQL 8.x 以降で機能します。

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <xa-datasource>
    <jndi-name>PostgresDS</jndi-name>

    <xa-datasource-class>org.postgresql.xa.PGXADatasource</xa-
datasource-class>
    <xa-datasource-property name="ServerName">[servername]</xa-
datasource-property>
    <xa-datasource-property name="PortNumber">5432</xa-datasource-
property>

    <xa-datasource-property name="DatabaseName">[database name]</xa-
datasource-property>
    <xa-datasource-property name="User">[username]</xa-datasource-
property>
    <xa-datasource-property name="Password">[password]</xa-datasource-
property>

    <track-connection-by-tx></track-connection-by-tx>
  </xa-datasource>
</datasources>
```

## A.8. INGRES

### 例A.26 Ingres Datasource

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>IngresDS</jndi-name>
    <use-java-context>false</use-java-context>
    <driver-class>com.ingres.jdbc.IngresDriver</driver-class>
    <connection-url>jdbc:ingres://localhost:II7/testdb</connection-url>
    <datasource-class>com.ingres.jdbc.IngresDataSource</datasource-
class>
    <datasource-property name="ServerName">localhost</datasource-
property>
    <datasource-property name="PortName">II7</datasource-property>
    <datasource-property name="DatabaseName">testdb</datasource-
property>
    <datasource-property name="User">testuser</datasource-property>
    <datasource-property name="Password">testpassword</datasource-
property>
    <new-connection-sql>select count(*) from iitables</new-connection-
sql>

    <check-valid-connection-sql>select count(*) from iitables</check-
valid-connection-sql>
    <metadata>
      <type-mapping>Ingres</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

[2]

---

[1] 出典: 『<http://community.jboss.org/wiki/SetUpASybaseDatasource>』

[2] 出典: 『<http://community.ingres.com>』

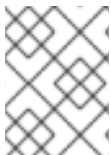
## 付録B ログイン情報とレシピ

### B.1. ログレベルの説明

表B.1「**log4j**におけるログレベルの定義」では、**log4j**の様々なログレベルが一般的にどのような意味合いを持つか一覧で表示しています。ご利用中のアプリケーションによってこれらのレベルの解釈が代わってくる場合もあります。

表B.1 **log4j**におけるログレベルの定義

log4j レベル	JDK レベル	詳細
FATAL		アプリケーションサービスがクラッシュする可能性が高い
ERROR	SEVERE	確実に問題が存在する
WARN	WARNING	問題発生の可能性は高いが、回復する可能性もある
INFO	INFO	サイズの小さい詳細ロギング。若干影響はあるが問題ではない
DEBUG	FINE	サイズの小さい詳細ロギング。興味を引く情報でない
	FINER	中サイズの詳細ロギング
TRACE	FINEST	サイズの大きい詳細ロギング



#### 注記

生成する出力が多いため、ロギングレベルが詳細になると本番システムには適していません。

#### 例B.1 特定のログレベルにログされる情報を制限

```

        <!-- Show the evolution of the DataSource pool in the logs
[inUse/Available/Max]-->
<category
name="org.jboss.resource.connectionmanager.JBossManagedConnectionPool">
  <priority value="TRACE" class="org.jboss.logging.XLevel"></priority>
</category>

```

## B.2. アプリケーション別にログファイルを分類

アプリケーション別にロギング出力を分類するには、**log4j**のカテゴリを特定の appender に割り当てます。通常、**conf/log4j.xml** の配備記述子で行われます。

### 例B.2 App 1 のログ出力を別ファイルにフィルタリング

```
<appender name="App1Log" class="org.apache.log4j.FileAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"></errorHandler>
  <param name="Append" value="false"/>
  <param name="File" value="${jboss.server.home.dir}/log/app1.log"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}]
%m%n"/>
  </layout>
</appender>

...

<category name="com.app1">
  <appender-ref ref="App1Log"></appender-ref>
</category>
<category name="com.util">
  <appender-ref ref="App1Log"></appender-ref>
</category>
```

### 例B.3 TCLMCFilter を利用

Enterprise Platform 5.1 には、新クラス **jboss.logging.filter.TCLMCFilter** が含まれており、デプロイメント URL をもとにフィルタリングが可能になります。

```
<appender name="App1Log" class="org.apache.log4j.FileAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"></errorHandler>
  <param name="Append" value="false"/>
  <param name="File" value="${jboss.server.home.dir}/log/app1.log"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}]
%m%n"/>
  </layout>
  <filter class="org.jboss.logging.filter.TCLMCFilter">
    <param name="AcceptOnMatch" value="true"/>
    <param name="DeployURL" value="app1.ear"/>
  </filter>

<!-- end the filter chain here -->
```

```
<filter class="org.apache.log4j.varia.DenyAllFilter"></filter>

</appender>
```

### B.3. カテゴリ出力のリダイレクト

1 つ以上のカテゴリのログレベルを上げる場合、出力を別のファイルにリダイレクトし検証を簡素化すると便利なが多くなっています。これを実行するには、**appender-ref** をカテゴリに追加します。

#### 例B.4 appender-ref の追加

```
    <appender name="JSR77" class="org.apache.log4j.FileAppender">
      <param name="File" value="${jboss.server.home.dir}/log/jsr77.log"/>
      ...
    </appender>

    <!-- Limit the JSR77 categories -->
    <category name="org.jboss.management" additivity="false">
      <priority value="DEBUG"></priority>
      <appender-ref ref="JSR77"></appender-ref>
    </category>
```

**org.jboss.management** の出力はすべて、**jsr77.log** ファイルに移動します。**additivity**属性は、出力はルートのカテゴリアペンドまで移動し続けるかどうかを制御します。**false** の場合、出力はカテゴリが参照するアペンドまでしか行かないようになっています。

## 付録C 改訂履歴

<b>改訂 5.1.2-2.400</b> Rebuild with publican 4.0.0	<b>2013-10-30</b>	<b>Rüdiger Landmann</b>
<b>改訂 5.1.2-2</b> Rebuild for Publican 3.0	<b>2012-07-18</b>	<b>Anthony Towns</b>
<b>改訂 5.1.2-100</b> JBoss Enterprise Application Platform 5.1.2 GAに対する変更を追加。本ガイド文書の変更に関する情報は、『リリースノート 5.1.2』を参照してください。	<b>Thu Dec 8 2011</b>	<b>Jared Morgan</b>
<b>改訂 5.1.1-100</b> JBoss Enterprise Application Platform 5.1.1 GAに対する変更を追加。本ガイド文書の変更に関する情報は、『リリースノート 5.1.1』を参照してください。	<b>Mon Jul 18 2011</b>	<b>Jared Morgan</b>
<b>改訂 5.1.0-100</b> 新たなバージョンニング要件に従いバージョン番号を変更 JBoss Enterprise Application Platform 5.1.0.GA 向けに以下を改訂： JBPAPP-4586 JBPAPP-4155 JBPAPP-4957 - HTTP Services の章に Solaris 上の NSAPI の項を追加	<b>Thu Sep 23 2010</b>	<b>Rebecca Newton</b>