



Red Hat JBoss Data Grid 7.2

開発者ガイド

Red Hat JBoss Data Grid 7.2 向け

Red Hat JBoss Data Grid 7.2 開発者ガイド

Red Hat JBoss Data Grid 7.2 向け

法律上の通知

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

JBoss Data Grid 7.2 を使用している開発者向けの高度なガイド

目次

パート I. プログラミング可能な API	16
第1章 プログラミング可能な API	17
第2章 CACHE API	18
2.1. CACHE API	18
2.2. CONFIGURATIONBUILDER API を使用したキャッシュ API の設定	18
2.3. 呼び出しごとのフラグ	19
2.3.1. 呼び出しごとのフラグ	19
2.3.2. 呼び出しごとのフラグ機能	19
2.3.3. 呼び出しごとのフラグの設定	19
2.3.4. 呼び出しごとのフラグの例	20
2.4. ADVANCEDCACHE インターフェース	20
2.4.1. AdvancedCache インターフェース	20
2.4.2. AdvancedCache インターフェースでのフラグの使用	20
2.4.3. ディストリビューションでの GET および PUT の使用	21
2.4.3.1. ディストリビューションでの GET および PUT の使用	21
2.4.3.2. 分散された GET および PUT 操作のリソース使用	21
2.4.4. Map メソッドの制限	21
第3章 マルチマップキャッシュ	23
3.1. マルチマップキャッシュ	23
3.2. MAVEN を使用した MULTIMAPCACHE のインストール	23
3.3. マルチマップキャッシュの作成	23
3.4. MULTIMAPCACHE の使用例	23
第4章 非同期 API	25
4.1. 非同期 API	25
4.2. 非同期 API の利点	25
4.3. 非同期プロセス	25
4.4. 戻り値と非同期 API	26
第5章 バッチ化 API	27
5.1. バッチ化 API	27
5.2. JAVA トランザクション API	27
5.3. バッチ化および JAVA トランザクション API (JTA)	27
5.4. バッチ化 API の使用	27
5.4.1. バッチ化 API の設定	27
5.4.2. バッチ化 API の使用	28
第6章 グループ化 API	29
6.1. グループ化 API	29
6.2. グループ化 API の操作	29
6.3. グループ化 API のユースケース	29
6.4. グループ化 API の設定	30
6.4.1. グループ化 API の設定	30
6.4.2. グループの有効化	30
6.4.3. 埋め込みグループの指定	30
6.4.4. 組み込みグループの指定	31
6.4.5. Grouper の登録	31
第7章 永続性 SPI	33
7.1. 永続性 SPI	33

7.2. 永続性 SPI の利点	33
7.3. 永続性 SPI のプログラムを使用した設定	33
7.4. 永続性の例	34
7.4.1. 永続性の例	34
7.4.2. プログラムを使用したキャッシュストアの設定	34
7.4.3. LevelDB キャッシュストアのプログラムを使用した設定	35
7.4.4. JdbcBinaryStore のプログラムを用いた設定	36
7.4.5. JdbcStringBasedStore のプログラムを使用した設定	37
7.4.6. JdbcMixedStore のプログラムを使用した設定	39
7.4.7. JPA キャッシュストアのプログラムを使用した設定例	41
7.4.8. Cassandra キャッシュストアのプログラムを使用した設定例	41
第8章 CONFIGURATIONBUILDER API	43
8.1. CONFIGURATIONBUILDER API	43
8.2. CONFIGURATIONBUILDER API の使用	43
8.2.1. CacheManager およびレプリケートされたキャッシュのプログラムによる作成	43
8.2.2. クラスター全体の動的キャッシュ作成	44
8.2.3. デフォルトの名前付きキャッシュを使用したカスタマイズされたキャッシュの作成	44
8.2.4. デフォルトでない名前付きキャッシュを使用したカスタマイズされたキャッシュの作成	45
8.2.5. Configuration Builder を使用したプログラムによるキャッシュの作成	45
8.2.6. グローバル設定の例	46
8.2.6.1. トランスポート層のグローバル設定	46
8.2.6.2. キャッシュマネージャー名のグローバル設定	46
8.2.6.3. JGroups のグローバル設定	46
8.2.7. キャッシュレベル設定の例	46
8.2.7.1. クラスターモードのキャッシュレベル設定	46
8.2.7.2. キャッシュレベルのエビクションおよびエクスパレーションの設定	47
8.2.7.3. JTA トランザクションのキャッシュレベルの設定	47
8.2.7.4. チェーンされた永続ストアを使用したキャッシュレベルでの設定	47
8.2.7.5. 高度なエクスターナライザーのキャッシュレベルでの設定	48
8.2.7.6. パーティション処理のキャッシュレベルでの設定 (ライブラリーノード)	48
第9章 EXTERNALIZABLE API	49
9.1. EXTERNALIZABLE API	49
9.2. エクスターナライザーのカスタマイズ	49
9.3. @SERIALIZEWITH を使用したマーシャリングのオブジェクトのアノテーション付け	49
9.4. 高度なエクスターナライザーの使用	50
9.4.1. 高度なエクスターナライザーの使用	50
9.4.2. メソッドの実装	51
9.4.3. エクスターナライザーとマーシャラークラスのリンク	52
9.4.4. 高度なエクスターナライザーの登録 (プログラムを使用)	52
9.4.5. 複数のエクスターナライザーの登録	52
9.5. エクスターナライザー ID 値のカスタマイズ	53
9.5.1. エクスターナライザー ID 値のカスタマイズ	53
9.5.2. エクスターナライザー ID のカスタマイズ (プログラムを使用)	53
第10章 通知/リスナー API	55
10.1. 通知/リスナー API	55
10.2. リスナーの例	55
10.3. リスナー通知	55
10.3.1. リスナー通知	55
10.3.2. キャッシュレベルの通知	55
10.3.3. キャッシュマネージャーレベルの通知	55
10.3.4. 同期および非同期通知	57

10.4. キャッシュエントリーの変更	57
10.4.1. キャッシュエントリーの変更	57
10.4.2. キャッシュエントリーが変更されたリスナーの設定	57
10.4.3. キャッシュエントリーが変更されたリスナーの例	57
10.5. クラスター化リスナー	58
10.5.1. クラスター化リスナー	58
10.5.2. クラスター化リスナーの設定	58
10.5.3. キャッシュリスナー API	59
10.5.4. クラスター化リスナーの例	60
10.5.5. 最適化されたキャッシュフィルターコンバーター	62
10.6. リモートイベントリスナー (HOT ROD)	63
10.6.1. リモートイベントリスナー (Hot Rod)	63
10.6.2. イベントリスナーの追加および削除	64
10.6.3. リモートイベントクライアントリスナーの例	64
10.6.4. リモートイベントのフィルター	66
10.6.4.1. リモートイベントのフィルター	66
10.6.4.2. リモートイベントのカスタムフィルター	67
10.6.4.3. 強化されたフィルターファクトリー	69
10.6.5. リモートイベントのカスタマイズ	70
10.6.5.1. リモートイベントのカスタマイズ	70
10.6.5.2. コンバーターの追加	72
10.6.5.3. ライトウェイトイベント	73
10.6.5.4. 動的なコンバーターインスタンス	73
10.6.5.5. カスタムイベントのリモートクライアントリスナーの追加	74
10.6.6. イベントマーシャリング	75
10.6.7. リモートイベントクラスタリングおよびフェイルオーバー	76
第11章 JSR-107 (JCACHE) API	79
11.1. JSR-107 (JCACHE) API	79
11.2. 依存関係	79
11.2.1. オプション 1: Maven	79
11.2.2. オプション 2: 必要なファイルのクラスパスへの追加	79
11.3. ローカルキャッシュの作成	80
11.3.1. ライブラリーモード	81
11.3.2. クライアントサーバーモード	81
11.4. データの格納および読み出し	81
11.5. JAVA.UTIL.CONCURRENT.CONCURRENTMAP API と JAVAX.CACHE.CACHE API の比較	82
11.6. JCACHE インスタンスのクラスター化	84
11.7. 複数のキャッシュプロバイダー	85
第12章 ヘルスチェック API	86
12.1. ヘルスチェック API	86
12.2. プログラムを使用したヘルスチェック API へのアクセス	86
第13章 REST API	88
13.1. REST インターフェース	88
13.2. RUBY クライアントコード	88
13.3. RUBY の例で JSON を使用	89
13.4. PYTHON クライアントコード	89
13.5. JAVA クライアントコード	89
13.6. REST インターフェースの使用	92
13.6.1. REST インターフェース操作	92
13.6.1.1. データ形式	92
13.6.1.2. デフォルトのデータ形式	92

13.6.1.3. サポートされるデータ形式	92
13.6.1.4. ヘッダー	93
13.6.1.5. Accept ヘッダー	93
13.6.1.6. Key-Content-Type ヘッダー	94
13.6.2. REST API を介したデータの追加	94
13.6.2.1. データをキャッシュに追加	94
13.6.2.2. PUT /{cacheName}/{cacheKey}	94
13.6.2.3. POST /{cacheName}/{cacheKey}	95
13.6.2.4. PUT および POST メソッドのヘッダー	95
13.6.3. REST API を介したデータの取得	96
13.6.3.1. キャッシュからのデータの取得	96
13.6.3.2. GET /{cacheName}/{cacheKey}	96
13.6.3.3. HEAD /{cacheName}/{cacheKey}	97
13.6.3.4. GET /{cacheName}	97
13.6.3.5. GET および HEAD メソッドのヘッダー	97
13.6.4. REST API を介したデータの削除	98
13.6.4.1. キャッシュからのデータの削除	98
13.6.4.2. DELETE /{cacheName}/{cacheKey}	98
13.6.4.3. DELETE /{cacheName}	98
13.6.4.4. バックグラウンド削除操作	98
13.6.5. ETag ベースのヘッダー	98
13.6.6. REST インターフェースを介したデータのクエリー	99
13.6.6.1. JSON から Protostream への変換	100
13.6.6.2. Protobuf スキーマの登録	100
13.6.6.3. JSON ドキュメントの Protobuf メッセージへのマッピング	100
13.6.6.4. キャッシュへの内容の追加	101
13.6.6.5. REST エンドポイントのクエリー	101
13.6.6.5.1. 任意のリクエストパラメーター	102
13.6.6.5.2. クエリー結果	102
第14章 クラスター化カウンター	104
14.1. COUNTER API	104
14.2. MAVEN 依存関係の追加	104
14.3. COUNTERMANAGER インターフェースの読み出し	105
14.4. クラスター化カウンターの使用	105
14.4.1. クラスター化カウンターの XML 設定	105
14.4.1.1. XML 定義	106
14.4.2. クラスター化カウンターのランタイム設定	106
14.4.3. クラスター化カウンターのプログラムによる設定	107
14.4.3.1. クラスター化カウンターの使用	108
第15章 クラスター化ロック	110
15.1. LOCK API	110
15.2. サポートされる設定	110
15.3. MAVEN 依存関係の追加	110
15.4. クラスター化ロックの使用	110
第16章 HOT ROD インターフェース	113
16.1. HOT ROD について	113
16.2. HOT ROD ヘッダー	113
16.2.1. Hot Rod ヘッダーデータタイプ	113
16.2.2. 要求ヘッダー	113
16.2.3. 応答ヘッダー	115
16.2.4. トポロジ変更ヘッダー	115

16.2.4.1. トポロジー変更ヘッダー	115
16.2.4.2. トポロジー変更マーカー値	116
16.2.4.3. トポロジー認識クライアントのトポロジー変更ヘッダー	116
16.2.4.4. ハッシュ配布認識クライアントのトポロジー変更ヘッダー	117
16.3. HOT ROD 操作	119
16.3.1. Hot Rod 操作	119
16.3.2. Hot Rod の Authenticate 操作	120
16.3.3. Hot Rod の AuthMechList 操作	121
16.3.4. Hot Rod の BulkGet 操作	121
16.3.5. Hot Rod の BulkKeysGet 操作	122
16.3.6. Hot Rod の Clear 操作	124
16.3.7. Hot Rod の ContainsKey 操作	124
16.3.8. Hot Rod の Exec 操作	125
16.3.9. Hot Rod の Get 操作	126
16.3.10. Hot Rod の GetAll 操作	127
16.3.11. Hot Rod の GetWithMetadata 操作	128
16.3.12. Hot Rod の GetWithVersion 操作	130
16.3.13. Hot Rod の IterationEnd 操作	131
16.3.14. Hot Rod の IterationNext 操作	131
16.3.15. Hot Rod の IterationStart 操作	133
16.3.16. Hot Rod の Ping 操作	135
16.3.17. Hot Rod の Put 操作	135
16.3.18. Hot Rod の PutAll 操作	136
16.3.19. Hot Rod の PutIfAbsent 操作	138
16.3.20. Hot Rod の Query 操作	139
16.3.21. Hot Rod の Remove 操作	140
16.3.22. Hot Rod の RemoveIfUnmodified 操作	141
16.3.23. Hot Rod の Replace 操作	142
16.3.24. Hot Rod の ReplaceIfUnmodified 操作	143
16.3.25. Hot Rod の ReplaceWithVersion 操作	145
16.3.26. Hot Rod の Stats 操作	147
16.3.27. Hot Rod の Size 操作	148
16.4. HOT ROD 操作の値	149
16.4.1. Hot Rod 操作の値	149
16.4.2. Magic 値	150
16.4.3. Status 値	150
16.4.4. Client Intelligence 値	151
16.4.5. フラグ値	151
16.4.6. Hot Rod のエラー処理	152
16.5. HOT ROD のリモートイベント	152
16.5.1. Hot Rod のリモートイベント	152
16.5.2. Hot Rod におけるリモートイベントのクライアントリスナーの追加	152
16.5.3. リモートイベントの Hot Rod リモートクライアントリスナー	155
16.5.4. Hot Rod イベントヘッダー	155
16.5.5. Hot Rod の CacheEntryCreated イベント	156
16.5.6. Hot Rod の CacheEntryModified イベント	156
16.5.7. Hot Rod の CacheEntryRemoved イベント	157
16.5.8. Hot Rod の Custom イベント	157
16.6. PUT 要求の例	158
16.7. HOT ROD JAVA クライアント	160
16.7.1. Hot Rod Java クライアント	160
16.7.2. Hot Rod Java クライアントのダウンロード	160
16.7.3. Hot Rod Java クライアントの設定	160

16.7.4. Hot Rod Java クライアントベーシック API	162
16.7.5. Hot Rod Java クライアントバージョン API	163
16.7.6. Hot Rod Java クライアントでのクラスター全体の動的キャッシュ作成	163
16.8. HOT ROD C++ クライアント	164
16.8.1. Hot Rod C++ クライアント	164
16.8.2. Hot Rod C++ クライアント形式	164
16.8.3. Hot Rod C++ クライアントの前提条件	165
16.8.4. Hot Rod C++ クライアントのインストール	165
16.8.4.1. Hot Rod C++ クライアントのダウンロードおよびインストール	165
16.8.4.2. RHEL における Hot Rod C++ クライアントのダウンロードおよびインストール	165
16.8.4.3. Windows における Hot Rod C++ クライアントのダウンロードおよびインストール	166
16.8.5. Hot Rod C++ クライアントでの Protobuf コンパイラーの使用	166
16.8.5.1. RHEL 7 における Protobuf コンパイラーの使用	166
16.8.5.2. Windows における Protobuf コンパイラーの使用	167
16.8.6. Hot Rod C++ クライアントの設定	167
16.8.7. Hot Rod C++ クライアント API	168
16.8.8. Hot Rod C++ クライアント非同期 API	168
16.8.9. Hot Rod C++ クライアントのリモートイベントリスナー	171
16.8.10. サイトと動作する Hot Rod C++ クライアント	172
16.8.10.1. 手動クラスター切り替え	172
16.8.11. Hot Rod C++ クライアントを用いたリモートクエリーの実行	173
16.8.12. Hot Rod C++ クライアントでのニアキャッシュの使用	176
16.8.13. Hot Rod C++ クライアントを使用したスクリプトの実行	176
16.9. HOT ROD C# クライアント	178
16.9.1. Hot Rod C# クライアント	179
16.9.2. Hot Rod C# クライアントのダウンロードとインストール	179
16.9.3. Hot Rod C# .NET プロジェクトの作成	183
16.9.4. Hot Rod C# クライアントの設定	184
16.9.5. Hot Rod C# クライアント API	184
16.9.6. Hot Rod C# クライアント非同期 API	185
16.9.7. Hot Rod C# クライアントのリモートイベントリスナー	186
16.9.8. サイトと動作する Hot Rod C# クライアント	187
16.9.8.1. 手動クラスター切り替え	188
16.9.9. Hot Rod C# クライアントを用いたリモートクエリーの実行	188
16.9.10. Hot Rod C# クライアントでのニアキャッシュの使用	190
16.9.11. Hot Rod C# クライアントを使用したスクリプトの実行	191
16.9.12. 相互運用性を維持するための文字列マーシャラー	193
16.10. HOT ROD NODE.JS クライアント	193
16.10.1. Hot Rod Node.js クライアント	193
16.10.2. Hot Rod Node.js クライアントのインストール	193
16.10.3. Hot Rod Node.js の要件	194
16.10.4. Hot Rod Node.js の基本機能	194
16.10.5. Hot Rod Node.js の条件付き操作	196
16.10.6. Hot Rod Node.js のデータセット	197
16.10.7. Hot Rod Node.js のリモートイベント	198
16.10.8. クラスターと動作する Hot Rod Node.js	199
16.10.9. サイトと動作する Hot Rod Node.js	200
16.10.9.1. 手動クラスター切り替え	201
16.11. HOT ROD C++ と HOT ROD JAVA クライアント間の相互運用性	201
16.12. サーバーと HOT ROD クライアントバージョン間の互換性	202

パート II. RED HAT JBOSS DATA GRID での INFINISPAN クエリーの作成および使用	204
--	-----

第17章 INFINISPAN クエリーの使用	205
17.1. はじめに	205
17.2. RED HAT JBOSS DATA GRID のクエリーのインストール	205
17.3. RED HAT JBOSS DATA GRID でのクエリー	206
17.3.1. Hibernate Search およびクエリーモジュール	206
17.3.2. Apache Lucene およびクエリーモジュール	206
17.4. インデックス化	206
17.4.1. インデックス化	206
17.4.2. トランザクションおよび非トランザクションキャッシュによるインデックス化	207
17.4.3. プログラムを使用したインデックス化設定	207
17.4.4. インデックスの再構築	208
17.5. 検索	208
第18章 オブジェクトのアノテーション付けおよびクエリー	209
18.1. オブジェクトのアノテーション付けおよびクエリー	209
18.2. アノテーションによるトランスフォーマーの登録	209
18.3. クエリーの例	210
第19章 ドメインオブジェクトのインデックス構造へのマッピング	212
19.1. 基本のマッピング	212
19.1.1. 基本のマッピング	212
19.1.2. @Indexed	212
19.1.3. @Field	212
19.1.4. @NumericField	214
19.2. プロパティーを複数回マッピング	215
19.3. 埋め込みオブジェクトおよび関連するオブジェクト	216
19.3.1. 埋め込みオブジェクトおよび関連するオブジェクト	216
19.3.2. 関連するオブジェクトのインデックス化	216
19.3.3. @IndexedEmbedded	216
19.3.4. targetElement プロパティー	218
19.4. ブースティング	219
19.4.1. ブースティング	219
19.4.2. 静的なインデックス時ブースティング	219
19.4.3. 動的なインデックス時ブースティング	219
19.5. 分析	220
19.5.1. 分析	220
19.5.2. デフォルトのアナライザーおよびクラス別のアナライザー	220
19.5.3. 名前付きのアナライザー	221
19.5.4. アナライザーの定義	222
19.5.5. Solr の @AnalyzerDef	222
19.5.6. アナライザーリソースのロード	224
19.5.7. 動的アナライザーの選択	224
19.5.8. アナライザーの読み出し	226
19.5.9. 使用可能なアナライザー	227
19.6. ブリッジ	229
19.6.1. ブリッジ	229
19.6.2. ビルトインブリッジ	229
19.6.3. カスタムブリッジ	231
19.6.3.1. カスタムブリッジ	231
19.6.3.2. FieldBridge	231
19.6.3.3. StringBridge	232
19.6.3.4. 双方向ブリッジ	232
19.6.3.5. パラメーター化されたブリッジ	233

19.6.3.6. 型対応ブリッジ	234
19.6.3.7. ClassBridge	234
第20章 クエリー	236
20.1. クエリー	236
20.2. クエリーの構築	236
20.2.1. クエリーの構築	236
20.2.2. Lucene ベースのクエリー API を使用した Lucene クエリーの構築	236
20.2.3. Lucene クエリーの構築	236
20.2.3.1. Lucene クエリーの構築	236
20.2.3.2. キーワードクエリー	237
20.2.3.3. ファジークエリー	240
20.2.3.4. ワイルドカードクエリー	240
20.2.3.5. フレーズクエリー	241
20.2.3.6. 範囲クエリー	241
20.2.3.7. クエリーの組み合わせ	241
20.2.3.8. クエリーオプション	242
20.2.4. Infinispan Query でのクエリーの構築	243
20.2.4.1. 一般論	243
20.2.4.2. ページネーション	243
20.2.4.3. ソート	244
20.2.4.4. 射影	244
20.2.4.5. クエリー時間の制限	246
20.2.4.6. 時間制限での例外の発生	246
20.3. 結果の読み出し	247
20.3.1. 結果の読み出し	247
20.3.2. パフォーマンスに関する注意点	247
20.3.3. 結果サイズ	247
20.3.4. 結果の理解	247
20.4. フィルター	248
20.4.1. フィルター	248
20.4.2. フィルターの定義および実装	248
20.4.3. @Factory フィルター	249
20.4.4. キーオブジェクト	250
20.4.5. フルテキストフィルター	251
20.4.6. シャード化された環境におけるフィルターの使用	252
20.5. 継続的クエリー	253
20.5.1. 継続的クエリー	253
20.5.2. 継続的クエリーの評価	254
20.5.3. 継続的クエリーの使用	254
20.5.4. C++ および C# の継続的クエリー	256
20.5.4.1. C++ 継続的クエリー	256
20.5.4.2. C# 継続的クエリー	257
20.5.5. 継続的クエリーでのパフォーマンスに関する注意点	257
20.6. ブロードキャストクエリー	257
20.6.1. ブロードキャストクエリー	258
20.6.1.1. ブロードキャストクエリーの使用	258
第21章 INFINISPAN QUERY DSL	259
21.1. INFINISPAN QUERY DSL	259
21.2. INFINISPAN QUERY DSL を用いたクエリーの作成	259
21.3. INFINISPAN QUERY DSL ベースのクエリーの有効化	259
21.4. INFINISPAN QUERY DSL ベースのクエリーの実行	260

21.5. 射影クエリー	261
21.6. グループ化および集約操作	261
21.7. 名前付きパラメーターの使用	263
第22章 ICKLE クエリー言語を使用したクエリーの構築	265
22.1. ICKLE クエリー言語を使用したクエリーの構築	265
22.2. LUCENE クエリーパーサー構文との違い	265
22.3. ファジークエリー	266
22.4. 範囲クエリー	266
22.5. フレーズクエリー	266
22.6. 近接クエリー	266
22.7. ワイルドカードクエリー	266
22.8. 正規表現クエリー	267
22.9. ブーストクエリー	267
第23章 リモートクエリー	268
23.1. リモートクエリー	268
23.2. クエリーの比較	268
23.3. HOT ROD JAVA クライアント経由のリモートクエリーの実行	269
23.4. HOT ROD C++ クライアントでのリモートクエリー	272
23.5. HOT ROD C# クライアントでのリモートクエリー	272
23.6. PROTOBUF エンコーディング	272
23.6.1. Protobuf エンコーディング	272
23.6.2. Protobuf エンコードされたエンティティの格納	272
23.6.3. Protobuf メッセージ	273
23.6.4. Hot Rod での Protobuf の使用	273
23.6.5. エンティティごとのマーシャラーの登録	274
23.6.6. Protobuf エンコードされたエンティティのインデックス化	275
23.6.7. Protobuf によるカスタムフィールドのインデックス化	276
23.6.8. Java アノテーションでの Protocol Buffers スキーマの定義	278
パート III. RED HAT JBOSS DATA GRID におけるデータのセキュア化	283
第24章 RED HAT JBOSS DATA GRID におけるデータのセキュア化	284
第25章 RED HAT JBOSS DATA GRID セキュリティー: 認証および承認	285
25.1. RED HAT JBOSS DATA GRID セキュリティー: 認証および承認	285
25.2. パーミッション	285
25.3. ロールマッピング	287
25.4. ログインモジュールを使用した認証およびロールマッピングの設定	288
25.5. RED HAT JBOSS DATA GRID の承認の設定	289
25.6. ライブラリーモードのデータセキュリティ	290
25.6.1. サブジェクトおよびプリンシパルクラス	290
25.6.2. サブジェクトの取得	290
25.6.3. サブジェクトの認証	291
25.7. インターフェースのセキュア化	294
25.7.1. インターフェースのセキュア化	294
25.7.2. Hot Rod インターフェースセキュリティ	295
25.7.2.1. Hot Rod サーバーと Hot Rod クライアント間の通信の暗号化	295
25.7.2.2. SSL を使用した LDAP サーバーに対する Hot Rod のセキュア化	296
25.7.2.3. SASL を使用した Hot Rod でのユーザー認証	298
25.7.2.3.1. SASL を使用した Hot Rod でのユーザー認証	298
25.7.2.3.2. Hot Rod 認証の設定 (GSSAPI/Kerberos)	298
25.7.2.3.3. Hot Rod 認証 (MD5) の設定	300

25.7.2.3.4. Hot Rod C++ 認証の設定 (GSSAPI/Kerberos)	301
25.7.2.3.5. Hot Rod C++ 認証の設定 (MD5)	303
25.7.2.3.6. Hot Rod C++ 認証の設定 (PLAIN)	305
25.7.2.3.7. Hot Rod C# 認証の設定 (EXTERNAL)	307
25.7.2.3.8. Hot Rod C# 認証の設定 (MD5)	308
25.7.3. Hot Rod C++ クライアントの暗号化	309
25.7.4. Hot Rod C# クライアントの暗号化	310
25.7.5. Hot Rod Node.js の暗号化	312
25.8. セキュリティー監査ロガー	313
25.8.1. セキュリティー監査ロガー	313
25.8.2. セキュリティー監査ロガーの設定 (ライブラリーモード)	313
25.8.3. カスタム監査ロガー	314
第26章 クラスタートラフィックのセキュリティー	315
26.1. ノードセキュリティーの設定 (ライブラリーモード)	315
26.2. ライブラリーモードのノード承認	316
パート IV. RED HAT JBOSS DATA GRID の高度な機能	317
第27章 RED HAT JBOSS DATA GRID の高度な機能	318
第28章 監視	319
28.1. 監視	319
28.2. JAVA MANAGEMENT EXTENSIONS (JMX)	319
28.2.1. Java Management Extensions (JMX)	319
28.2.2. Red Hat JBoss Data Grid における JMX の使用	319
28.2.3. キャッシュインスタンスに対して JMX を有効にする	319
28.2.4. CacheManager に対して JMX を有効にする	319
28.2.5. 複数の JMX ドメイン	319
28.2.6. デフォルトでない MBean サーバーでの MBean の登録	320
28.3. STATISTICSINFOMBEAN	320
第29章 LUCENE DIRECTORY としての RED HAT JBOSS DATA GRID	321
29.1. LUCENE DIRECTORY としての RED HAT JBOSS DATA GRID	321
29.2. 設定	321
29.3. RED HAT JBOSS DATA GRID モジュール	322
29.4. レプリケートされたインデックスの LUCENE ディレクトリー設定	322
29.5. JMS マスターおよびスレーブのバックエンド設定	323
第30章 トランザクション	324
30.1. JAVA トランザクション API	324
30.2. トランザクションの設定 (ライブラリーモード)	324
30.3. 複数のキャッシュインスタンス間でのトランザクション	326
30.4. トランザクションマネージャー	326
第31章 マーシャリング	328
31.1. マーシャリング	328
31.2. JBOSS MARSHALLING FRAMEWORK	328
31.3. シリアライズ不可能なオブジェクトのサポート	328
31.4. HOT ROD およびマーシャリング	328
31.5. REMOTECACHEMANAGER を使用したマーシャラーの設定	329
31.6. デシリアライズを特定 JAVA クラスに制限	330
31.7. トラブルシューティング	330
31.7.1. マーシャリングのトラブルシューティング	330
31.7.2. その他のマーシャリング関連の問題	333

第32章 INFINISPAN CDI モジュール	336
32.1. INFINISPAN CDI モジュール	336
32.2. INFINISPAN CDI の使用	336
32.2.1. Infinispan CDI の要件	336
32.2.2. CDI Maven 依存関係の設定	336
32.3. INFINISPAN CDI モジュールの使用	336
32.3.1. Infinispan CDI モジュールの使用	337
32.3.2. Infinispan キャッシュの設定およびインジェクション	337
32.3.2.1. Infinispan キャッシュのインジェクション	337
32.3.2.2. リモート Infinispan キャッシュのインジェクション	337
32.3.2.3. インジェクションのターゲットキャッシュの設定	337
32.3.2.3.1. インジェクションのターゲットキャッシュの設定	337
32.3.2.3.2. 修飾子アノテーションの作成	337
32.3.2.3.3. プロデューサークラスの追加	338
32.3.2.3.4. 希望のクラスのインジェクト	338
32.3.3. CDI を用いたキャッシュマネージャーの設定	338
32.3.3.1. CDI を用いたキャッシュマネージャーの設定	339
32.3.3.2. デフォルト設定の指定	339
32.3.3.3. 組み込みキャッシュマネージャーの作成のオーバーライド	339
32.3.3.4. リモートキャッシュマネージャーの設定	340
32.3.3.5. 単一クラスでの複数のキャッシュマネージャーの設定	341
32.4. CDI アノテーションを使用した格納および取得	342
32.4.1. キャッシュアノテーションの設定	342
32.4.2. キャッシュアノテーションの有効化	342
32.4.3. メソッド呼び出しの結果をキャッシュ	343
32.4.3.1. メソッド呼び出しの結果をキャッシュ	343
32.4.3.2. 使用するキャッシュの指定	344
32.4.3.3. キャッシュされた結果のキャッシュキー	344
32.4.3.4. カスタムキーの生成	345
32.4.4. キャッシュ操作	345
32.4.4.1. キャッシュエントリーの更新	345
32.4.4.2. キャッシュからのエントリーの削除	346
32.4.4.3. キャッシュの消去	346
第33章 SPRING FRAMEWORK との統合	347
33.1. SPRING FRAMEWORK との統合	347
33.2. SPRING MAVEN 依存関係の定義	347
33.3. プログラミングによる SPRING キャッシュサポートの有効化 (ライブラリーモード)	347
33.4. プログラミングによる SPRING キャッシュサポートの有効化 (リモートクライアントサーバーモード)	348
33.5. アプリケーションコードへのキャッシングの追加	349
第34章 APACHE SPARK との統合	350
34.1. JBOSS DATA GRID の APACHE SPARK コネクター	350
34.2. SPARK の依存関係	350
34.3. SPARK コネクターの設定	351
34.3.1. バージョン 1.6 コネクターを設定するプロパティ	351
34.3.2. バージョン 2 コネクターを設定するメソッド	351
34.3.3. セキュアな JDG クラスターへの接続	352
34.4. SPARK 1.6 のコード例	353
34.4.1. Spark 1.6 のコード例	353
34.4.2. RDD の作成および使用	353
34.4.3. RDD の作成	353
34.4.4. RDD のクエリー	354

34.4.5. RDD のキャッシュへの書き込み	355
34.4.5.1. DStreams の作成および使用	356
34.4.6. Spark での Infinispan Query DSL の使用	357
34.4.7. クエリーによるフィルリング	357
34.4.8. 射影を使用したフィルタリング	358
34.4.9. デプロイされたフィルターを使用したフィルタリング	358
34.5. SPARK 2 のコード例	358
34.5.1. Spark 2 のコード例	358
34.5.2. RDD の作成および使用	358
34.5.3. RDD の作成	359
34.5.4. RDD のクエリー	359
34.5.4.1. SparkSQL クエリー	359
34.5.5. RDD のキャッシュへの書き込み	360
34.5.6. DStreams の作成	361
34.5.7. Apache Spark Dataset API の使用	362
34.5.8. Spark での Infinispan Query DSL の使用	364
34.5.9. 事前ビルドされたクエリーオブジェクトを使用したフィルタリング	364
34.5.10. Ickle クエリーを使用したフィルタリング	365
34.5.11. サーバー上でのフィルタリング	365
34.6. SPARK のパフォーマンスに関する注意点	366
第35章 APACHE HADOOP との統合	367
35.1. APACHE HADOOP との統合	367
35.2. HADOOP の依存関係	367
35.3. サポートされる HADOOP 設定パラメーター	367
35.4. HADOOP コネクタの使用	368
第36章 EAP との統合	370
36.1. EAP との統合	370
36.2. EAP モジュールのインストール	370
36.3. EAP の依存関係	370
36.4. 特定の JDG コンポーネントの依存関係	371
36.4.1. コア依存関係	371
36.4.2. リモート/Hot Rod 依存関係	371
36.4.3. 埋め込みクエリーの依存関係	371
36.4.4. Lucene ディレクトリーの依存関係	371
36.4.5. Hibernate Search ディレクトリープロバイダーの依存関係	372
36.4.6. EAP の内部 Hibernate Search モジュールの使用	372
36.4.7. その他の Hibernate Search モジュールとの用途	372
36.5. EAP モジュールの使用	372
36.5.1. ライブラリーモード	372
36.5.2. EAP サブシステムモード	372
36.6. EAP サブシステムモードの設定	373
36.7. コンテナおよびキャッシュへのリモートアクセス	376
36.8. EAP サブシステムモードでの EAP および JDG のトラブルシューティング	376
36.8.1. ロギングの有効化	376
36.8.2. 依存関係ツリーの出力	376
第37章 サーバーヒンティングを用いた高可用性	377
37.1. サーバーヒンティングを用いた高可用性	377
37.2. CONSISTENTHASHFACTORIES	377
37.2.1. ConsistentHashFactories	377
37.2.2. ConsistentHashFactory の実装	378
37.3. キーアフィニティーサービス	378

37.3.1. キーアフィニティーサービス	378
37.3.2. ライフサイクル	379
37.3.3. トポロジの変更	379
第38章 分散実行	381
38.1. 分散実行	381
38.2. 分散エグゼキューターサービス	381
38.3. DISTRIBUTEDCALLABLE API	382
38.4. CALLABLE および CDI	382
38.5. 分散タスクのフェイルオーバー	383
38.6. 分散タスク実行ポリシー	384
38.7. 分散実行とローカリティー	385
38.7.1. 分散実行の例	385
第39章 ストリーム	388
39.1. ストリーム	388
39.2. ローカル/インバリデーション/レプリケーションキャッシュでのストリームの使用	388
39.3. 分散キャッシュでのストリームの使用	388
39.4. タイムアウトの設定	388
39.5. 分散ストリーム	389
39.5.1. 分散ストリーム	389
39.5.2. マーシャルの可能性	389
39.5.3. 並列処理	390
39.5.4. 分散演算子	390
39.5.4.1. 終端演算子の分散結果削減	390
39.5.4.2. キーベースのリハッシュ対応演算子	391
39.5.4.3. 中間操作の例外	391
39.5.5. 分散ストリームの例	392
第40章 スクリプト	394
40.1. スクリプト	394
40.2. スクリプトキャッシュへのアクセス	394
40.3. スクリプトのインストール	395
40.4. メタデータのスクリプト	396
40.5. スクリプトバインディング	397
40.6. スクリプトパラメーター	397
40.7. HOT ROD JAVA クライアントを使用したスクリプトの実行	397
40.8. スクリプトの例	398
40.9. 格納されたスクリプト実行時の制限	398
第41章 リモートタスクの実行	399
41.1. リモートタスクの実行	399
41.2. リモートタスクの作成	399
41.3. リモートタスクの例	399
41.4. リモートタスクのインストール	400
41.5. リモートタスクの削除	400
41.6. リモートタスクの実行	401
第42章 データの相互運用性	402
42.1. プロトコルの相互運用性	402
42.1.1. プロトコルの相互運用性	402
42.1.2. ユースケースおよび要件	402
42.1.3. REST 上のプロトコル相互運用性	403
第43章 データセンター間のレプリケーションのセットアップ	404

43.1. データセンター間レプリケーション	404
43.2. データセンター間レプリケーションの操作	404
43.3. プログラミングによるデータセンター間レプリケーションの設定	406
43.4. サイトをオフラインにする	408
43.5. HOT ROD サイト間クラスターフェイルオーバー	408
第44章 ニアキャッシュ	410
44.1. ニアキャッシュ	410
44.2. ニアキャッシュの設定	411
44.3. クラスター環境でのニアキャッシュ	411
第45章 競合マネージャーの使用法	412
45.1. キャッシュの競合の検索および解決	412
付録A 参考資料	413
A.1. エクスターナライザー	413
A.1.1. エクスターナライザーについて	413
A.1.2. 内部エクスターナライザー実装アクセス	413
A.2. ハッシュ領域の割り当て	414
A.2.1. ハッシュ領域の割り当てについて	414
A.2.2. ハッシュ領域におけるキーの検索	414

パート I. プログラミング可能な **API**

第1章 プログラミング可能な API

Red Hat JBoss Data Grid は以下のプログラム可能な API を提供します。

- Cache
- AdvancedCache
- MultimapCache
- Asynchronous
- Batching
- Grouping
- Persistence (以前は CacheStore)
- ConfigurationBuilder
- Externalizable
- Notification (通知とリスナーを処理するため、Listener API とも呼ばれます)
- JSR-107 (JCache)
- Health Check
- REST

第2章 CACHE API

2.1. CACHE API

Cache インターフェースは、エントリーを追加、読み出し、および削除するために簡単なメソッドを提供します。これには JDK の **ConcurrentMap** インターフェースによって公開されるアトミックメカニズムが含まれます。エントリーが格納される方法は、使用されるキャッシュモードによって異なります。たとえば、エントリーがリモートノードへレプリケートされたり、キャッシュストアで検索されたりします。

基本的な作業では、キャッシュ API は JDK マップ API と同様に使用されます。そのため、マップベースの簡単なインメモリーキャッシュを Red Hat JBoss Data Grid のキャッシュへ移行する処理が容易になります。



注記

この API は JBoss Data Grid のリモートクライアントサーバーモードでは使用できません。

2.2. CONFIGURATIONBUILDER API を使用したキャッシュ API の設定

Red Hat JBoss Data Grid は **ConfigurationBuilder API** を使用してキャッシュを設定します。

キャッシュは **ConfigurationBuilder** ヘルパーオブジェクトを使用してプログラミングによって設定されます。

以下は、**ConfigurationBuilder API** を使用してプログラミングにより設定され、同期的にレプリケートされたキャッシュの例になります。

プログラミングによるキャッシュの設定

```
Configuration c = new
ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC).build()
;

String newCacheName = "repl";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

1. 設定の最初の行で、**ConfigurationBuilder** を使用して新しいキャッシュ設定オブジェクト (**c**) が作成されます。設定 **c** には、キャッシュモードを除くすべてのキャッシュ設定オプションのデフォルト値が割り当てられ、この値は上書きされ、同期レプリケーション (**REPL_SYNC**) に設定されます。
2. 設定の 2 行目で、新しい変数 (タイプが **String**) が作成され、値 **repl** が割り当てられます。
3. 設定の 3 行目で、キャッシュマネージャーは名前付きキャッシュ設定を定義するために使用されます。この名前付きキャッシュ設定は **repl** と呼ばれ、設定は最初の行のキャッシュ設定 **c** に提供された設定に基づきます。
4. 設定の 4 行目で、キャッシュマネージャーで保持された **repl** の一意のインスタンスに対する参照を取得するために、キャッシュマネージャーが使用されます。このキャッシュインスタンスはデータを格納および取得する操作を実行するために使用できます。

注記

JBoss EAP には独自の基礎となる JMX が含まれています。そのため、JBoss EAP でサンプルコードを使用するときに競合が発生し、**org.infinispan.jmx.JmxDomainConflictException: Domain already registered org.infinispan** などのエラーが表示されることがあります。

この問題を回避するには、以下のようにグローバル設定を指定します。

```
GlobalConfiguration glob = new GlobalConfigurationBuilder()
    .clusteredDefault()
    .globalJmxStatistics()
    .allowDuplicateDomains(true)
    .enable()
    .build();
```

2.3. 呼び出しごとのフラグ

2.3.1. 呼び出しごとのフラグ

Red Hat JBoss Data Grid では呼び出しごとのフラグを使用して各キャッシュコールの動作を指定できます。呼び出しごとのフラグは、時間を節約できる最適化の実装を容易にします。

2.3.2. 呼び出しごとのフラグ機能

Red Hat JBoss Data Grid のキャッシュ **API** の **putForExternalRead()** メソッドは内部的にフラグを使用します。このメソッドは、外部リソースからロードされたデータが含まれる JBoss Data Grid キャッシュをロードできます。この呼び出しの効率性を改善するために、JBoss Data Grid は通常の **put** 操作を呼び出して以下のフラグを渡します。

- **ZERO_LOCK_ACQUISITION_TIMEOUT** フラグ: 外部ソースからキャッシュへデータをロードするときに JBoss Data Grid のロック取得時間がほぼゼロになります。
- **FAIL_SILENTLY** フラグ: ロックを取得できない場合に、JBoss Data Grid はロック取得例外が発生せずに失敗します。
- **FORCE_ASYNCHRONOUS** フラグ: クラスター化された場合に、設定されたキャッシュモードに関係なくキャッシュが非同期にレプリケートされます。結果として、他のノードからの応答は必要ありません。

上記のフラグを組み合わせると、操作の効率性が大幅に向上します。この効率性の基礎として、データがメモリーにない場合にクライアントは永続ストアから必要なデータを取得できるため、このタイプの **putForExternalRead** コールが使用されます。クライアントはキャッシュミスを検出すると操作を再試行します。

JBoss Data Grid で使用できる全フラグの詳細リストは、JBoss Data Grid API ドキュメントの [Flag](#) クラスを参照してください。

2.3.3. 呼び出しごとのフラグの設定

Red Hat JBoss Data Grid で呼び出しごとのフラグを使用するには、**withFlags()** メソッド呼び出しを使用して必要なフラグを高度なキャッシュに追加します。

呼び出しごとのフラグの設定

```
Cache cache = ...
cache.getAdvancedCache()
    .withFlags(Flag.SKIP_CACHE_STORE, Flag.CACHE_MODE_LOCAL)
    .put("local", "only");
```



注記

呼び出されたフラグは、キャッシュ操作の間のみアクティブになります。同じトランザクション内の複数の呼び出しで同じフラグを使用するには、各呼び出しに **withFlags()** メソッドを使用します。キャッシュ操作を別のノードにレプリケートする必要がある場合は、フラグがリモートノードにも使用されます。

2.3.4. 呼び出しごとのフラグの例

put() などの書き込み操作が以前の値を返してはならない JBoss Data Grid のユースケースでは、**IGNORE_RETURN_VALUES** フラグが使用されます。このフラグにより分散環境で (以前の値を取得するための) リモート検索が実行されないようにし、不必要な以前の値が取得されないようにします。さらに、キャッシュがキャッシュローダーで設定された場合、このフラグによって以前の値がキャッシュストアからロードされないようにします。

IGNORE_RETURN_VALUES フラグの使用

```
Cache cache = ...
cache.getAdvancedCache()
    .withFlags(Flag.IGNORE_RETURN_VALUES)
    .put("local", "only")
```

2.4. ADVANCEDCACHE インターフェース

2.4.1. AdvancedCache インターフェース

Red Hat JBoss Data Grid は **AdvancedCache** インターフェースを提供し、単純な Cache インターフェース以外に JBoss Data Grid を拡張します。 **AdvancedCache** インターフェースは以下を実行できます。

- カスタマーインターセプターのインジェクト
- 特定の内部コンポーネントへのアクセス
- フラグを適用して特定のキャッシュメソッドの動作を変更

以下のコード例は **AdvancedCache** の取得方法の例を示しています。

```
AdvancedCache advancedCache = cache.getAdvancedCache();
```

2.4.2. AdvancedCache インターフェースでのフラグの使用

Red Hat JBoss Data Grid の特定のキャッシュメソッドにフラグが適用されると、ターゲットメソッドの動作が変更されます。キャッシュ呼び出しに任意の数のフラグを適用するには、**AdvancedCache.withFlags()** を使用します。

フラグのキャッシュ呼び出しへの適用


```
advancedCache.withFlags(Flag.CACHE_MODE_LOCAL, Flag.SKIP_LOCKING)
    .withFlags(Flag.FORCE_SYNCHRONOUS)
    .put("hello", "world");
```

2.4.3. ディストリビューションでの GET および PUT の使用

2.4.3.1. ディストリビューションでの GET および PUT の使用

ディストリビューションモードでは、キャッシュは write コマンドの前にリモートで **GET** コマンドを実行します。これは、**Cache.put()** などの一部のメソッドは、**java.util.Map** コントラクトにしたがって指定されたキーに関連する以前の値を返すためです。これがキーを所有しないインスタンスで実行され、エントリーが L1 キャッシュにない場合、**PUT** の前にリモートで **GET** を実行することが唯一信頼できる戻り値の取得方法になります。

Red Hat JBoss Data Grid は戻り値を待たなければならないため、キャッシュが同期または非同期であるかに関わらず、**PUT** 操作の前に発生する **GET** 操作は常に同期になります。

2.4.3.2. 分散された GET および PUT 操作のリソース使用

ディストリビューションモードでは、キャッシュが **GET** 操作を実行してから **PUT** 操作を実行することがあります。

この操作は、リソースの面では非常にコストのかかる操作になります。リモートの **GET** 操作は同期であるにもかかわらず、すべての応答を待たないため、無駄になるリソースが発生します。**GET** 処理は最初に受信する有効な応答を許可するため、パフォーマンスとクラスターの大きさとの関連性はありません。

実装に戻り値が必要でない場合、**Flag.SKIP_REMOTE_LOOKUP** フラグを呼び出しごとの設定に使用します。

このような動作は、キャッシュの操作やパブリックメソッドの正確な機能に悪影響を与えるものではありませんが、**java.util.Map** インターフェースコントラクトに違反します。これは、信頼できない不正確な戻り値が特定のメソッドに提供されるためコントラクトに違反します。そのため、必ずこれらの戻り値が設定上重要な目的に使用されないようにしてください。

2.4.4. Map メソッドの制限

size()、**values()**、**keySet()**、**entrySet()** などの特定の Map メソッドは不安定であるため、Red Hat JBoss Data Grid では一定の制限付きで 사용할 ことができます。これらのメソッドはロック (グローバルまたはローカル) を取得せず、同時編集、追加、および削除はこれらの呼び出しでは考慮されません。

一覧表示されるメソッドはパフォーマンスに大きく影響します。そのため、情報収集やデバッグの目的でのみこれらのメソッドを使用することが推奨されます。

パフォーマンスの問題

JBoss Data Grid 7.2 では、map メソッド **size()**、**values()**、**keySet()**、および **entrySet()** にデフォルトでキャッシュローダーのエントリーが含まれます。これらのコマンドのパフォーマンスは、使用するキャッシュローダーによって決まります。たとえば、データベースを使用している場合、これらのメソッドはデータが格納されるテーブルの完全なスキャンを実行し、処理が遅くなることがあります。キャッシュローダーからエントリーをロードしないようにし、パフォーマンスヒットを避けるには、必要なメソッドを実行する前に

Cache.getAdvancedCache().withFlags(Flag.SKIP_CACHE_LOAD) を使用します。

size() メソッドの概要 (埋め込みキャッシュ)

JBoss Data Grid 7.2 では、**Cache.size()** メソッドは、クラスター全体で、このキャッシュとキャッシュローダーの両方にあるすべての要素の数を提示します。ローダーまたはリモートエントリーを使用している場合、メモリー関連の問題の発生を防げるようにエントリーのサブセットのみが指定時にメモリーに保持されます。すべてのエントリーをロードする場合、その速度が遅くなる場合があります。

この操作モードでは、**size()** メソッドで返される結果は、ローカルノードにあるエントリー数を返すよう強制実行する **org.infinispan.context.Flag#CACHE_MODE_LOCAL** フラグと、パッシベートされたエントリーを無視する **org.infinispan.context.Flag#SKIP_CACHE_LOAD** フラグの影響を受けます。これらのフラグのいずれかを使用すると、クラスター全体ですべての要素の数を返さない代わりにこのメソッドのパフォーマンスを上げることができます。

size() メソッドの概要 (リモートキャッシュ)

JBoss Data Grid 7.1 では、Hot Rod プロトコルには専用の **SIZE** 操作が含まれ、クライアントはこの操作を使用してすべてのエントリーのサイズを計算します。

第3章 マルチマップキャッシュ

3.1. マルチマップキャッシュ

MultimapCache は、キーを値にマップするキャッシュで、各キーには複数の値を含めることができます。現在、ライブラリーモードでのみ機能します。

3.2. MAVEN を使用した MULTIMAPCACHE のインストール

Maven プロジェクトで **MultimapCache** を利用できるようにするには、**pom.xml** を以下のように設定します。

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-multimap</artifactId>
  <version>...</version> <!-- 7.2.0 or later -->
</dependency>
```

3.3. マルチマップキャッシュの作成

以下のようにコードを使用して、**MultimapCache** を作成します。

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager cm = ... ;

// create or obtain a MultimapCacheManager passing the EmbeddedCacheManager
MultimapCacheManager multimapCacheManager =
EmbeddedMultimapCacheManagerFactory.from(cm);

// define the configuration for the multimap cache
multimapCacheManager.defineConfiguration(multimapCacheName, c.build());

// get the multimap cache
multimapCache = multimapCacheManager.get(multimapCacheName);
```

3.4. MULTIMAPCACHE の使用例

MultimapCache の使用方法を表すコードは次のとおりです。

```
MultimapCache<String, String> multimapCache = ...;

multimapCache.put("girlNames", "marie")
    .thenCompose(r1 -> multimapCache.put("girlNames", "oihana"))
    .thenCompose(r3 -> multimapCache.get("girlNames"))
    .thenAccept(names -> {
        if(names.contains("marie"))
            System.out.println("Marie is a girl
name");

        if(names.contains("oihana"))
```

```
name");  
        System.out.println("Oihana is a girl  
    });
```

第4章 非同期 API

4.1. 非同期 API

Red Hat JBoss Data Grid は同期 API メソッドの他に、非ブロッキング方式で同じ機能を実現する非同期 API も提供します。

非同期メソッドの命名規則は、同期メソッドの命名規則と似ていますが、各メソッド名の初めに **Async** が追加されます。非同期メソッドは、操作の結果が含まれる **Future** を返します。

たとえば、**Cache<String, String>** とパラメーター化されたキャッシュでは、**Cache.put(String key, String value)** は **String** を返します。また、**Cache.putAsync(String key, String value)** は **FutureString** を返します。

4.2. 非同期 API の利点

非同期 API はブロックしないため、以下のような複数の利点があります。

- 同期通信が保証される (エラーと例外を処理する機能が追加される)。
- 呼び出しが完了するまでスレッドの操作をブロックする必要がない。

これらの利点により、以下のようにシステムで並列処理を向上させることができます。

非同期 API の使用

```
Set<Future<?>> futures = new HashSet<Future<?>>();
futures.add(cache.putAsync("key1", "value1"));
futures.add(cache.putAsync("key2", "value2"));
futures.add(cache.putAsync("key3", "value3"));
```

たとえば、以下の行は実行時にスレッドをブロックしません。

- **futures.add(cache.putAsync(key1, value1));**
- **futures.add(cache.putAsync(key2, value2));**
- **futures.add(cache.putAsync(key3, value3));**

これら 3 つの put 操作からのリモートコールは同時に実行されます。これは、分散モードで実行する場合に特に役に立ちます。

4.3. 非同期プロセス

Red Hat JBoss Data Grid の一般的な書き込み操作では、以下のプロセスがクリティカルパスで失敗し、リソースが最も必要なものから必要でないものに順序付けされます。

- ネットワークコール
- マーシャリング
- キャッシュストアへの書き込み (オプション)
- ロック

Red Hat JBoss Data Grid では、非同期メソッドを使用すると、クリティカルパスからネットワークコールとマーシャリングが削除されます。

4.4. 戻り値と非同期 API

Red Hat JBoss Data Grid で非同期 **API** が使用された場合、クライアントコードでは以前の値を問い合わせるために非同期操作が **Future** または **CompletableFuture** を返す必要があります。

非同期操作の結果を取得するには、次の操作を呼び出します。この操作は呼び出されたときにスレッドをブロックします。

```
Future.get()
```

第5章 バッチ化 API

5.1. バッチ化 API

Red Hat JBoss Data Grid クラスターがトランザクションの唯一の参加者である場合、バッチ化 **API** が使用されます。複数のシステムがトランザクションの参加者である場合は、トランザクションマネージャーを使用する Java トランザクション API (**JTA**) のトランザクションが使用されます。



注記

バッチ化 API は Red Hat JBoss Data Grid のライブラリーモードでのみ使用できます。

5.2. JAVA トランザクション API

Red Hat JBoss Data Grid では、Java トランザクション API (JTA) に対応するトランザクションの設定、使用、および参加がサポートされます。

JBoss Data Grid は各キャッシュ操作に対して以下を実行します。

1. 最初に、現在スレッドに関連付けされているトランザクションを読み出します。
2. **XAResource** が登録されていない場合は、トランザクションマネージャーに登録し、トランザクションがコミットまたはロールバックされたときに通知を受け取るようにします。

5.3. バッチ化および JAVA トランザクション API (JTA)

Red Hat JBoss Data Grid では、バッチ化機能により、**JTA** トランザクションがバックエンドで開始され、スコープ内のすべての呼び出しがそれに関連付けられます。このため、バッチ化機能は単純なトランザクションマネージャー実装をバックエンドで使用します。結果として、次の動作が行われます。

1. 呼び出し中に取得されたロックは、トランザクションがコミットまたはロールバックするまで保持されます。
2. すべての変更は、クラスター内のすべてのノード上にあるバッチでトランザクションコミットプロセスの一部としてレプリケートされます。複数の変更が単一のトランザクション内で確実に行われるようにするため、レプリケーショントラフィックがより少ない状態になり、パフォーマンスが向上します。
3. 同期のレプリケーションまたはインバリデーションを使用する場合、レプリケーションまたはインバリデーション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。
4. キャッシュがトランザクションで、キャッシュローダーが存在する場合、キャッシュローダーはキャッシュのトランザクションに登録されません。そのため、トランザクションがインメモリー状態を適用し、変更をストアに適用できないと (部分的に)、キャッシュローダーレベルで不整合が発生する可能性があります。
5. トランザクションに関連するすべての設定はバッチ化にも適用されます。

5.4. バッチ化 API の使用

5.4.1. バッチ化 API の設定

バッチ化 API を使用するには、以下の例のようにキャッシュ設定で呼び出しのバッチ化を有効にします。

```
Configuration c = new
ConfigurationBuilder().transaction().transactionMode(TransactionMode.TRANS
ACTIONAL).invocationBatching().enable().build();
```

Red Hat JBoss Data Grid では、呼び出しバッチ化はデフォルトで無効になり、バッチ化は定義されたトランザクションマネージャーなしで使用できます。

5.4.2. バッチ化 API の使用

キャッシュがバッチ化を使用するよう設定された後に、キャッシュで次のように **startBatch()** および **endBatch()** を呼び出して、バッチ化を使用します。

```
Cache cache = cacheManager.getCache();
```

バッチを使用しない場合

```
cache.put("key", "value");
```

cache.put(key, value); 行が実行されると、値はすぐに置き換えられます。

バッチを使用する場合

```
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(true);
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false);
```

行 **cache.endBatch(true);** が実行されると、バッチの開始後に行われたすべての変更が適用されます。

行 **cache.endBatch(false);** が実行されると、バッチで行われた変更は破棄されます。

第6章 グループ化 API

6.1. グループ化 API

グループ化 API は、エントリーのグループを指定のノードやグループのハッシュを使用して選択されたノードに移動できます。

6.2. グループ化 API の操作

通常、Red Hat JBoss Data Grid は特定のキーのハッシュを使用してエントリーの宛先ノードを決定します。しかし、グループ化 API を使用する場合、キーのハッシュではなくキーに関連するグループのハッシュを使用して宛先ノードを決定します。

各ノードはアルゴリズムを使用して各キーの所有者を決定します。これにより、ノード間のエントリーの場所に関するメタデータ (およびメタデータの更新) を渡す必要がなくなります。これは、以下の理由で有用です。

- コストの高いノード全体のメタデータ更新を行わずに、すべてのノードは特定のキーを所有するノードを判断できます。
- ノードの障害時に所有権情報をレプリケートする必要がないため、冗長性が改善されます。

グループ化 API を使用する場合、各ノードはエントリーの所有者を判断できる必要があります。そのため、グループを手作業で指定することはできず、以下のいずれかである必要があります。

- エントリーへの組み込み。キークラスによって生成されたことを意味します。
- エントリーに対して外部的。外部機能によって生成されたことを意味します。

6.3. グループ化 API のユースケース

この機能を使用すると、論理的に関連するデータを 1 つのノードで格納できます。たとえば、キャッシュにユーザー情報が含まれる場合、1 つの場所に存在するすべてのユーザーの情報を 1 つのノードで格納できます。

この方法の利点は、特定のデータ (論理的に関連する) が必要な場合に、分散エグゼキューター (Distributed Executor) のタスクがクラスターのすべてのノード全体ではなく関連するノードでのみ実行されるよう指示されることです。このように操作が指示されるため、パフォーマンスが最適化されます。

グループ化 API の例

Acme 社は世界中に 100 以上の支店を持つ家電販売会社です。支店によって多くの部門の社員が勤務する支店と1、2部門の社員のみが勤務する支店があります。人事部所属の社員はバンコク、ロンドン、シカゴ、ニース、およびベネチアに勤務しています。

Acme 社は人事部用の社員記録をすべてキャッシュの単一のノード (ノード AB) に移動するため、Red Hat JBoss Data Grid のグループ化 API を使用します。そのため、人事部所属の社員が記録を読み出ししようとする、**DistributedExecutor** はノード AB のみをチェックし、必要な社員記録を迅速かつ簡単に呼び出します。

説明どおりに 1 つのノードで関連するエントリーを格納した場合、クラスターのすべてのノードではなく 1 つのノード (またはノードの小さなサブセット) で情報を検索することで、データのアクセスを最適化し、無駄な時間やリソースの発生を防ぐことができます。

6.4. グループ化 API の設定

6.4.1. グループ化 API の設定

以下の手順にしたがってグループ化 API を設定します。

1. 宣言的またはプログラムのメソッドを使用してグループを有効にします。
2. 組み込みグループまたは非組み込みグループのいずれかを指定します。これらのグループのタイプに関する詳細は、「[組み込みグループの指定](#)」および「[非組み込みグループの指定](#)」を参照してください。
3. 指定した grouper をすべて登録します。

6.4.2. グループの有効化

グループ化 API を設定するための最初のステップは、グループの有効化です。以下の例は、グループを有効にする方法を示しています。

```
Configuration c = new
ConfigurationBuilder().clustering().hash().groups().enabled().build();
```

6.4.3. 埋め込みグループの指定

次の場合は、組み込みグループをグループ化 API で使用します。

- キークラスの定義を変更できる場合 (変更不可能なライブラリーの一部でない)。
- キークラスがキーバリュースペアグループの判断を考慮しない場合。

関連するメソッドで **@Group** アノテーションを使用して組み込みグループを指定します。以下の例のように、グループは常に String である必要があります。

組み込みグループの指定例

```
class User {

    <!-- Additional configuration information here -->
    String office;
    <!-- Additional configuration information here -->

    public int hashCode() {
        // Defines the hash for the key, normally used to determine location
        <!-- Additional configuration information here -->
    }

    // Override the location by specifying a group, all keys in the same
    // group end up with the same owner
    @Group
    String getOffice() {
        return office;
    }
}
```

6.4.4. 組み込みグループの指定

次の場合は、非組み込みグループをグループ化 API で使用します。

- キークラスの定義を変更できない場合 (変更不可能なライブラリーの一部である)。
- キークラスがキーバリューペアグループの判断を考慮する場合。

組み込みグループは、**Grouper** インターフェースの実装を使用して指定されます。このインターフェースは **computeGroup** メソッドを使用してグループを返します。

組み込みグループを指定するとき、**Grouper** インターフェースは算出した値を **computeGroup** に渡してインターセプターとして動作します。**@Group** アノテーションが使用される場合、このアノテーションを使用するグループが最初の **Grouper** に渡されます。そのため、組み込みグループを使用するよりも優れた制御が提供されます。

非組み込みグループの指定例

以下の例は、パターンを使用してキーからグループを抽出するためにキークラスを使用する簡単な **Grouper** で構成されます。このような場合、キークラスで指定されたグループ情報は無視されます。

```
public class KXGrouper implements Grouper<String> {

    // A pattern that can extract from a "kX" (e.g. k1, k2) style key
    // The pattern requires a String key, of length 2, where the first
    // character is
    // "k" and the second character is a digit. We take that digit, and
    // perform
    // modular arithmetic on it to assign it to group "1" or group "2".

    private static Pattern kPattern = Pattern.compile("(^k)(\\d)$");

    public String computeGroup(String key, String group) {
        Matcher matcher = kPattern.matcher(key);
        if (matcher.matches()) {
            String g = Integer.parseInt(matcher.group(2)) % 2 + "";
            return g;
        } else {
            return null;
        }
    }

    public Class<String> getKeyType() {
        return String.class;
    }
}
```

6.4.5. Grouper の登録

作成後、各 grouper を使用するには登録する必要があります。

grouper のプログラムを使った登録

```
Configuration c = new  
ConfigurationBuilder().clustering().hash().groups().addGrouper(new  
KXGrouper()).enabled().build();
```

第7章 永続性 SPI

7.1. 永続性 SPI

Red Hat JBoss Data Grid では、永続性によって外部 (永続) ストレージエンジンを設定できます。これらのストレージエンジンは Red Hat JBoss Data Grid のデフォルトのインメモリーストレージを補完します。

永続外部ストレージには以下のような利点があります。

- メモリーは揮発性で、キャッシュストアによってキャッシュの情報の存続期間を延長することが可能です。これにより、持続性が向上します。
- 永続外部ストアをアプリケーションとカスタムストレージエンジン間のキャッシングレイヤーとして使用すると、ライトスルー機能が向上します。
- エビクションとパッシベーションの組み合わせを使用すると、頻繁に必要な情報のみがメモリー内に保存され、他のデータは外部ストレージに保存されます。



注記

Red Hat JBoss Data Grid のライブラリーモードでのみプログラムを使用して永続性を設定できます。

7.2. 永続性 SPI の利点

永続性 SPI の Red Hat JBoss Data Grid 実装には以下の利点があります。

- JSR-107 (<http://jcp.org/en/jsr/detail?id=107>) とのアライメント。JBoss Data Grid の **CacheWriter** および **CacheLoader** インターフェースはJSR-107 ライターおよびリーダーと似ています。そのため、JSR-107 とのアライメントにより、JCache 対応ベンダー全体でストアの移植性が向上します。
- 簡易化されたトランザクション統合。JBoss Data Grid はロックを自動的に処理するため、実装はストアへの同時アクセスを調整する必要はありません。ロックモードによっては同じキーで同時書き込みされないことがあります。しかし、インプリメンターはストアの操作が複数のスレッドから開始され、実装コードが追加されることを想定します。
- シリアライズの減少により CPU の使用率が低下します。新しい SPI は保存されたエントリーをシリアライズされた形式で公開します。リモートで送信するためにエントリーが永続ストレージから取得された場合、デシリアライズ (ストアからの読み取り時) した後に再度シリアライズ (送信の書き込み時) する必要はありません。この代わりにエントリーはストレージから取得されるとシリアライズされた形式で書き込まれます。

7.3. 永続性 SPI のプログラムを使用した設定

以下は、永続性 SPI を使用した単一ファイルストアのプログラムを使用した設定の例になります。

永続性 SPI を使用した単一ファイルストアの設定

```
ConfigurationBuilder builder = new ConfigurationBuilder();
    builder.persistence()
        .passivation(false)
        .addSingleFileStore()
```

```

        .preload(true)
        .shared(false)
        .fetchPersistentState(true)
        .ignoreModifications(false)
        .purgeOnStartup(false)
        .location(System.getProperty("java.io.tmpdir"))
        .async()
            .enabled(true)
            .threadPoolSize(5)
        .singleton()
            .enabled(true)
            .pushStateWhenCoordinator(true)
            .pushStateTimeout(20000);

```

7.4. 永続性の例

7.4.1. 永続性の例

以下は、プログラムを使用してキャッシュストア実装を設定する方法を示す例になります。これらのストアの比較と追加情報については『[Administration and Configuration Guide](#)』を参照してください。

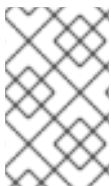
7.4.2. プログラムを使用したキャッシュストアの設定

以下の例では、プログラムを使用してキャッシュストアを設定する方法を示します。

```

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
    .passivation(false)
    .addSingleFileStore()
        .shared(false)
        .preload(true)
        .fetchPersistentState(true)
        .purgeOnStartup(false)
        .location(System.getProperty("java.io.tmpdir"))
    .async()
        .enabled(true)
        .threadPoolSize(5)
    .singleton()
        .enabled(true)
        .pushStateWhenCoordinator(true)
        .pushStateTimeout(20000);

```



注記

この設定は単一ファイルキャッシュストア用です。**location** などの一部の属性は単一ファイルキャッシュストアに固有であり、他の種類のキャッシュストアには使用されません。

プログラムを使用したキャッシュストアの設定

1. **ConfigurationBuilder** を使用して、新規の設定オブジェクトを作成します。
2. **passivation** 要素は Red Hat JBoss Data Grid がストアと通信する方法に影響を与えます。

パッシベーションは、インメモリーキャッシュからオブジェクトを削除し、システムやデータベースなどの2次データストアに書き込みます。2次データストアがない場合、オブジェクトはインメモリーキャッシュから削除されるのみです。パッシベーションはデフォルトで **false** です。

3. **addSingleFileStore()** 要素は、この設定用のキャッシュストアとして **SingleFileStore** を追加します。**addStore** メソッドを使用して追加できる、JDBC キャッシュストアなどの他のストアを作成することができます。
4. **shared** パラメーターは、キャッシュストアが異なるキャッシュインスタンスによって共有されていることを示します。たとえば、クラスター内のすべてのインスタンスが、同じリモートの共有データベースと通信するために同じ JDBC 設定を使用する場合があります。**shared** は、デフォルトで **false** になります。**true** に設定すると、異なるキャッシュインスタンスによって重複データがキャッシュストアに書き込まれないようにすることができます。
5. **preload** 要素はデフォルトでは **false** に設定されます。**true** に設定されると、キャッシュストアに保存されたデータは、キャッシュの起動時にメモリーにプリロードされます。これにより、キャッシュストアのデータが起動後すぐに利用できるようになり、データをレイジーにロードしたことによるキャッシュ操作の遅延を防ぐことができます。プリロードされたデータは、ノード上でローカルのみに保存され、プリロードされたデータのレプリケーションや分散は行われません。JBoss Data Grid は、エビクションのエントリーの最大設定数までの数をプリロードします。
6. **fetchPersistentState** 要素は、キャッシュの永続状態をフェッチするかどうかを決定し、クラスターに参加する際にこれをローカルキャッシュストアに適用します。キャッシュストアが共有される場合、キャッシュが同じキャッシュストアにアクセスするためフェッチ永続状態は無視されます。複数のキャッシュストアでこのプロパティーが **true** に設定された場合にキャッシュサービスを起動すると、設定の例外が発生します。**fetchPersistentState** プロパティーはデフォルトでは **false** です。
7. **purgeOnStartup** 要素は、キャッシュストアの起動時にキャッシュストアをパージするかどうかを制御し、デフォルトでは **false** になります。
8. **location** 設定要素は、ストアが書き込みできるディスクの場所を設定します。
9. これらの属性は、それぞれのキャッシュストアに固有の内容を設定します。たとえば、**location** 属性は、**SingleFileStore** がデータが含まれるファイルを維持する場所を指します。他のストアには、さらに複雑な設定が必要な場合があります。
10. **singleton** 要素を使用すると、クラスター内の1つのノードのみで変更を保存できます。このノードはコーディネーターと呼ばれます。コーディネーターは、インメモリー状態のキャッシュをディスクにプッシュします。この機能は、すべてのノードの **enabled** 属性を **true** に設定することによりアクティベートされます。**shared** パラメーターは、**singleton** を同時に有効にした状態で定義することはできません。**enabled** 属性はデフォルトでは **false** です。
11. **pushStateWhenCoordinator** 要素はデフォルトでは **true** に設定されます。**true** の場合、このプロパティーにより、コーディネーターになったノードがインメモリー状態を基礎となるキャッシュストアに転送します。このパラメーターは、コーディネーターがクラッシュし、新規のコーディネーターが選択される場合に役に立ちます。

7.4.3. LevelDB キャッシュストアのプログラムを使用した設定

以下は、LevelDB キャッシュストアの、プログラムを使用した設定例です。

```
Configuration cacheConfig = new ConfigurationBuilder().persistence()
```



```
.addStore(LevelDBStoreConfigurationBuilder.class)
.location("/tmp/leveldb/data")
.expiredLocation("/tmp/leveldb/expired").build();
```

LevelDB キャッシュストアのプログラムを使用した設定

1. **ConfigurationBuilder** を使用して、新規の設定オブジェクトを作成します。
2. **LevelDBCachedStoreConfigurationBuilder** クラスを使用してストアを追加し、その設定を構築します。
3. LevelDB キャッシュストアのロケーションパスを設定します。指定したパスは、主なキャッシュストアデータを保存します。ディレクトリがない場合は自動的に作成されます。
4. LevelDB スタアの **expiredLocation** パラメーターを使用して、期限切れデータの場所を指定します。指定されたパスは、ページされる前に期限切れデータを保存します。ディレクトリがない場合は自動的に作成されます。

7.4.4. JdbcBinaryStore のプログラムを用いた設定

JdbcBinaryStore は、同じテーブル行/blob の同じハッシュ値 (キー上の **hashCode** メソッド) ですべてのキーを格納し、すべてのキータイプをサポートします。



重要

バイナリー JDBC スタアは JBoss Data Grid 7.2 では非推奨となったため、実稼働での使用は推奨されません。代わりに String ベースのストアを使用することが推奨されます。

以下は、**JdbcBinaryStore** の設定例になります。

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
    .addStore(JdbcBinaryStoreConfigurationBuilder.class)
    .fetchPersistentState(false)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .table()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_BUCKET_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")

    .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .connectionPool()

    .connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
        .username("sa")
        .driverClass("org.h2.Driver");
```

JdbcBinaryStore のプログラムを使用した設定 (ライブラリーモード)

1. **ConfigurationBuilder** を使用して、新規の設定オブジェクトを作成します。

2. **JdbcBinaryStore** 設定ビルダーを追加して、このストアに関連する固有の設定を構築します。
3. **fetchPersistentState** 要素は、キャッシュの永続状態をフェッチするかどうかを決定し、クラスターに参加する際にこれをローカルキャッシュストアに適用します。キャッシュストアが共有される場合、キャッシュが同じキャッシュストアにアクセスするためフェッチ永続状態は無視されます。複数のキャッシュローダーでこのプロパティが **true** に設定された場合にキャッシュサービスを起動すると、設定の例外が発生します。**fetchPersistentState** プロパティはデフォルトでは **false** です。
4. **ignoreModifications** 要素は、書き込み操作を共有キャッシュローダーではなく、ローカルファイルキャッシュローダーに許可することで、書き込みメソッドを特定のキャッシュローダーにプッシュするかどうかを決定します。場合によっては、一時的なアプリケーションデータが、インメモリーキャッシュと同じサーバー上のファイルベースのキャッシュローダーにのみ存在する必要があります。たとえば、これはネットワーク内のすべてのサーバーによって使用される追加の JDBC ベースのキャッシュローダーで適用されます。**ignoreModifications** はデフォルトでは **false** になります。
5. **purgeOnStartup** 要素は、初回起動時にキャッシュがパージされるかどうかを指定します。
6. テーブルを以下のように設定します。
 - a. **dropOnExit** は、キャッシュストアが停止している際にテーブルを破棄するかどうかを決定します。これは、デフォルトでは **false** に設定されます。
 - b. **createOnStart** は、現在テーブルが存在しない場合にキャッシュストアの起動時にテーブルを作成します。このメソッドはデフォルトでは **true** です。
 - c. **tableNamePrefix** は、データが保存されるテーブルの名前に接頭辞を設定します。
 - d. **idColumnName** プロパティは、キャッシュキーまたはバケット ID が保存される列を定義します。
 - e. **dataColumnName** プロパティは、キャッシュエントリまたはバケットが保存される列を指定します。
 - f. **timestampColumnName** 要素は、キャッシュエントリまたはバケットのタイムスタンプが保存される列を指定します。
7. The **connectionPool** 要素は、次のパラメーターを使用して JDBC ドライバーの接続プールを指定します。
 - a. **connectionUrl** パラメーターは、JDBC ドライバー固有の接続 URL を指定します。
 - b. **username** パラメーターには、**connectionUrl** 経由で接続するために使用されるユーザー名が含まれます。
 - c. **driverClass** パラメーターは、データベースへの接続に使用されるドライバーのクラス名を指定します。

7.4.5. JdbcStringBasedStore のプログラムを使用した設定

JdbcStringBasedStore は複数のエントリを各行にグループ化せずに、各エントリをテーブルの独自の行に格納するため、同時に負荷がかかる状態でスループットが増加します。

以下は、**JdbcStringBasedStore** の設定例になります。

```

ConfigurationBuilder builder = new ConfigurationBuilder();

builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
    .fetchPersistentState(false)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .table()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_STRING_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")

    .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .dataSource()
        .jndiUrl("java:jboss/datasources/JdbcDS");

```

JdbcStringBasedStore のプログラムを使用した設定

1. **ConfigurationBuilder** を使用して、新規の設定オブジェクトを作成します。
2. **JdbcStringBasedStore** 設定ビルダーを追加して、このストアに関連する固有の設定を構築します。
3. **fetchPersistentState** パラメーターは、キャッシュの永続状態をフェッチするかどうかを決定し、クラスターに参加する際にこれをローカルキャッシュストアに適用します。キャッシュストアが共有される場合、キャッシュが同じキャッシュストアにアクセスするためフェッチ永続状態は無視されます。複数のキャッシュローダーでこのプロパティが **true** に設定された場合にキャッシュサービスを起動すると、設定の例外が発生します。**fetchPersistentState** プロパティはデフォルトでは **false** です。
4. **ignoreModifications** パラメーターは、書き込み操作を共有キャッシュローダーではなく、ローカルファイルキャッシュローダーに許可することで、書き込みメソッドを特定のキャッシュローダーにプッシュするかどうかを決定します。場合によっては、一時的なアプリケーションデータが、インメモリーキャッシュと同じサーバー上のファイルベースのキャッシュローダーにのみ存在する必要があります。たとえば、これはネットワーク内のすべてのサーバーによって使用される追加の JDBC ベースのキャッシュローダーで適用されません。**ignoreModifications** はデフォルトでは **false** になります。
5. **purgeOnStartup** パラメーターは、初回起動時にキャッシュがパージされるかどうかを指定します。
6. テーブルの設定
 - a. **dropOnExit** は、キャッシュストアが停止している際にテーブルを破棄するかどうかを決定します。これは、デフォルトでは **false** に設定されます。
 - b. **createOnStart** は、現在テーブルが存在しない場合にキャッシュストアの起動時にテーブルを作成します。このメソッドはデフォルトでは **true** です。
 - c. **tableNamePrefix** は、データが保存されるテーブルの名前に接頭辞を設定します。
 - d. **idColumnName** プロパティは、キャッシュキーまたはバケット ID が保存される列を定義します。

- e. **dataColumnName** プロパティは、キャッシュエントリまたはバケットが保存される列を指定します。
 - f. **timestampColumnName** 要素は、キャッシュエントリまたはバケットのタイムスタンプが保存される列を指定します。
7. **dataSource** 要素は、以下のパラメーターを使用してデータソースを指定します。
- **jndiUrl** は、既存の JDBC への JNDI URL を指定します。



注記

JdbcStringBasedStore 使用時に IO 例外である `Unsupported protocol version 48` エラーが発生した場合、データ列タイプが適切な **BLOB** や **VARBINARY** ではなく、**VARCHAR** や **CLOB** などに設定されていることを示します。
JdbcStringBasedStore の値はどのデータタイプでもよく、キーが文字列であることのみ必要となります。そのため、バイナリー列に保存することができます。

7.4.6. JdbcMixedStore のプログラムを使用した設定

JdbcMixedStore は、キーのタイプを基にキーを **JdbcBinaryStore** または **JdbcStringBasedStore** に委譲するハイブリッド実装です。



重要

混合 JDBC ストアは JBoss Data Grid 7.2 では非推奨となったため、実稼働での使用は推奨されません。代わりに String ベースのストアを使用することが推奨されます。

以下は、**JdbcMixedStore** の設定例になります。

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcMixedStoreConfigurationBuilder.class)
    .fetchPersistentState(false)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .stringTable()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_MIXED_STR_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")

    .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .binaryTable()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_MIXED_BINARY_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")

    .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .connectionPool()
```

```
.connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
    .username("sa")
    .driverClass("org.h2.Driver");
```

JdbcMixedStore のプログラムを使用した設定

1. **ConfigurationBuilder** を使用して、新規の設定オブジェクトを作成します。
2. **JdbcMixedStore** 設定ビルダーを追加して、このストアに関連する固有の設定を構築します。
3. **fetchPersistentState** パラメーターは、キャッシュの永続状態をフェッチするかどうかを決定し、クラスターに参加する際にこれをローカルキャッシュストアに適用します。キャッシュストアが共有される場合、キャッシュが同じキャッシュストアにアクセスするためフェッチ永続状態は無視されます。複数のキャッシュローダーでこのプロパティが **true** に設定された場合にキャッシュサービスを起動すると、設定の例外が発生します。**fetchPersistentState** プロパティはデフォルトでは **false** です。
4. **ignoreModifications** パラメーターは、書き込み操作を共有キャッシュローダーではなく、ローカルファイルキャッシュローダーに許可することで、書き込みメソッドを特定のキャッシュローダーにプッシュするかどうかを決定します。場合によっては、一時的なアプリケーションデータが、インメモリーキャッシュと同じサーバー上のファイルベースのキャッシュローダーにのみ存在する必要があります。たとえば、これはネットワーク内のすべてのサーバーによって使用される追加の JDBC ベースのキャッシュローダーで適用されます。**ignoreModifications** はデフォルトでは **false** になります。
5. **purgeOnStartup** パラメーターは、初回起動時にキャッシュがパージされるかどうかを指定します。
6. テーブルを以下のように設定します。
 - a. **dropOnExit** は、キャッシュストアが停止している際にテーブルを破棄するかどうかを決定します。これは、デフォルトでは **false** に設定されます。
 - b. **createOnStart** は、現在テーブルが存在しない場合にキャッシュストアの起動時にテーブルを作成します。このメソッドはデフォルトでは **true** です。
 - c. **tableNamePrefix** は、データが保存されるテーブルの名前に接頭辞を設定します。
 - d. **idColumnName** プロパティは、キャッシュキーまたはバケット ID が保存される列を定義します。
 - e. **dataColumnName** プロパティは、キャッシュエントリまたはバケットが保存される列を指定します。
 - f. **timestampColumnName** 要素は、キャッシュエントリまたはバケットのタイムスタンプが保存される列を指定します。
7. The **connectionPool** 要素は、次のパラメーターを使用して JDBC ドライバーの接続プールを指定します。
 - a. **connectionUrl** パラメーターは、JDBC ドライバー固有の接続 URL を指定します。
 - b. **username** パラメーターには、**connectionUrl** 経由で接続するために使用されるユーザー名が含まれます。

- c. **driverClass** パラメーターは、データベースへの接続に使用されるドライバーのクラス名を指定します。

7.4.7. JPA キャッシュストアのプログラムを使用した設定例

Red Hat JBoss Data Grid で JPA キャッシュストアをプログラムを使用して設定するには、以下を使用します。

```
Configuration cacheConfig = new ConfigurationBuilder().persistence()
    .addStore(JpaStoreConfigurationBuilder.class)

    .persistenceUnitName("org.infinispan.loaders.jpa.configurationTest")
    .entityClass(User.class)
    .build();
```

このコード例で使用されるパラメーターは以下のとおりです。

- **persistenceUnitName** パラメーターは、JPA エンティティークラスが含まれる設定ファイル (**persistence.xml**) の JPA キャッシュストアの名前を指定します。
- **entityClass** パラメーターは、このキャッシュに格納された JPA エンティティークラスを指定します。設定ごとに1つのクラスのみを指定できます。

7.4.8. Cassandra キャッシュストアのプログラムを使用した設定例

Cassandra キャッシュストアは、Red Hat JBoss Data Grid のコアライブラリーの一部ではなく、クラスパスに追加する必要があります。Maven プロジェクトでは、以下を **pom.xml** に追加するとクラスパスに追加できます。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cachestore-cassandra</artifactId>
  <version>...</version> <!-- 7.2.0 or later -->
</dependency>
```

以下の設定スニペットは、プログラムを使用して Cassandra キャッシュストアを定義する方法の例を示しています。

```
Configuration cacheConfig = new ConfigurationBuilder()
    .persistence()
    .addStore(CassandraStoreConfigurationBuilder.class)
    .addServer()
        .host("127.0.0.1")
        .port(9042)
    .addServer()
        .host("127.0.0.1")
        .port(9041)
    .autoCreateKeyspace(true)
    .keyspace("TestKeyspace")
    .entryTable("TestEntryTable")
    .consistencyLevel(ConsistencyLevel.LOCAL_ONE)
    .serialConsistencyLevel(ConsistencyLevel.SERIAL)
    .connectionPool()
        .heartbeatIntervalSeconds(30)
```

```
        .idleTimeoutSeconds(120)
        .poolTimeoutMillis(5)
        .build();
```

第8章 CONFIGURATIONBUILDER API

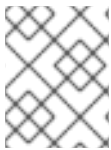
8.1. CONFIGURATIONBUILDER API

ConfigurationBuilder API は Red Hat JBoss Data Grid のプログラミング可能な設定 API です。

ConfigurationBuilder **API** は、以下のことが行えるよう設計されています。

- コーディングプロセスをより効率的にするための設定オプションのチェーンコーディング
- 設定の可読性の向上

Red Hat JBoss Data Grid では、ConfigurationBuilder API は、CacheLoaders を有効にし、グローバルおよびキャッシュレベルの操作を設定するためにも使用されます。



注記

プログラムを使用した設定は、Red Hat JBoss Data Grid のライブラリーモードでのみ実行できます。

8.2. CONFIGURATIONBUILDER API の使用

8.2.1. CacheManager およびレプリケートされたキャッシュのプログラムによる作成

Red Hat JBoss Data Grid のプログラムによる設定は、ほぼ ConfigurationBuilder API および CacheManager のみが関係します。以下はプログラムによる CacheManager 設定の例になります。

CacheManager のプログラムを使用した設定

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-
file.xml");
Cache defaultCache = manager.getCache();
Configuration c = new
ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC)
.build();

String newCacheName = "repl";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

1. XML ファイルで初めに CacheManager を作成します。必要な場合は、ユースケースの要件を満たす仕様に基いて実行時にこの CacheManager をプログラミングできます。
2. プログラムを使用して、同期的にレプリケートされたキャッシュを新規作成します。
 - ConfigurationBuilder ヘルパーオブジェクトを使用して新しい設定オブジェクトインスタンスを作成します。
設定の最初の行で、**ConfigurationBuilder** を使用して新しいキャッシュ設定オブジェクト (**c**) が作成されます。設定 **c** には、キャッシュモードを除くすべてのキャッシュ設定オプションのデフォルト値が割り当てられ、この値は上書きされ、同期レプリケーション (**REPL_SYNC**) に設定されます。
 - マネージャーで設定を定義または登録します。

設定の 3 行目で、キャッシュマネージャーは名前付きキャッシュ設定を定義するために使用されます。この名前付きキャッシュ設定は **repl** と呼ばれ、設定は最初の行のキャッシュ設定 **c** に提供された設定に基づきます。

- 設定の 4 行目で、キャッシュマネージャーで保持された **repl** の一意のインスタンスに対する参照を取得するために、キャッシュマネージャーが使用されます。このキャッシュインスタンスはデータを格納および取得する操作を実行するために使用できます。

8.2.2. クラスター全体の動的キャッシュ作成

上記の例のように、**getCache()** メソッドを使用する場合、キャッシュは 1 つのノードでのみ作成されます。キャッシュをクラスターに参加する新しいノードで動的に作成する必要がある場合は、代わりに **createCache()** メソッドを使用します。

```
Cache<String, String> cache =
    manager.administration().createCache("newCacheName", "newTemplate");
```

このように作成されたキャッシュはクラスターのすべてのノードで利用可能になりますが、それは一時的です。クラスター全体をシャットダウンし、再起動しても、キャッシュは自動的に再作成されません。キャッシュを永続化するには、**PERMANENT** フラグを以下のように使用します。

```
Cache<String, String> cache =
    manager.administration().withFlags(AdminFlag.PERMANENT).createCache("newCacheName", "newTemplate");
```

上記は、グローバル状態を有効にし、適切な設定ストレージを選択しないと動作しません。選択可能な設定ストレージは以下のとおりです。

- **VOLATILE**: 名前が意味するとおり、この設定ストレージは **PERMANENT** キャッシュをサポートしません。
- **OVERLAY**: **caches.xml** という名前のファイルのグローバルな共有状態永続パスに設定を保存します。
- **MANAGED**: サーバーデプロイメントでのみサポートされ、**PERMANENT** キャッシュをサーバーモデルに保存します。
- **CUSTOM**: カスタムの設定ストア。

8.2.3. デフォルトの名前付きキャッシュを使用したカスタマイズされたキャッシュの作成

デフォルトキャッシュ設定 (またはカスタマイズされた設定) は新しいキャッシュを作成する土台として使用できます。

例として、**infinispan-config-file.xml** で、レプリケートされたキャッシュの設定がデフォルト値として指定され、カスタマイズされたライフスパン値を持つ分散キャッシュが必要であるとして。必要な分散キャッシュは **infinispan-config-file.xml** ファイルで指定されたデフォルトキャッシュのすべての内容 (言及された内容を除く) を保持する必要があります。

デフォルトキャッシュのカスタマイズ

```
String newCacheName = "newCache";
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-
```



```
file.xml");
Configuration dcc = manager.getDefaultCacheConfiguration();
Configuration c = new ConfigurationBuilder().read(dcc).clustering()
    .cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).enable()
    .build();
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

1. デフォルトの **Configuration** オブジェクトのインスタンスを読み取り、デフォルトの設定を取得します。
2. **ConfigurationBuilder** を使用して、新しい設定オブジェクトでキャッシュモードと L1 キャッシュライフスパンを構築および変更します。
3. キャッシュマネージャーでキャッシュ設定を登録または定義します。
4. 指定された設定が含まれる **newCache** への参照を取得します。

8.2.4. デフォルトでない名前付きキャッシュを使用したカスタマイズされたキャッシュの作成

デフォルトでない名前付きキャッシュを使用して新しいカスタマイズされたキャッシュを作成する必要があります。この手順は、デフォルトの名前付きキャッシュを使用する場合の手順に似ています。

違いは、デフォルトのキャッシュの代わりに **replicatedCache** という名前のキャッシュを取得することです。

デフォルトでない名前付きキャッシュを使用したカスタマイズされたキャッシュの作成

```
String newCacheName = "newCache";
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-
file.xml");
Configuration rc = manager.getCacheConfiguration("replicatedCache");
Configuration c = new ConfigurationBuilder().read(rc).clustering()
    .cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).enable()
    .build();
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

1. **replicatedCache** を読み取り、デフォルト設定を取得します。
2. **ConfigurationBuilder** を使用して、新しい設定オブジェクトで必要な設定を構築および変更します。
3. キャッシュマネージャーでキャッシュ設定を登録または定義します。
4. 指定された設定が含まれる **newCache** への参照を取得します。

8.2.5. Configuration Builder を使用したプログラムによるキャッシュの作成

デフォルトキャッシュ値で xml ファイルを使用して新しいキャッシュを作成する代わりに、ConfigurationBuilder API を使用して XML ファイルなしで新しいキャッシュを作成します。ConfigurationBuilder API は設定オプションに対してチェーンされたコードを作成するときに使いやすさを提供することを目的としています。

以下の新しい設定は、グローバルおよびキャッシュレベル設定に対して有効です。GlobalConfiguration オブジェクトは、GlobalConfigurationBuilder を使用して構築され、Configuration オブジェクトは ConfigurationBuilder を使用して構築されます。

8.2.6. グローバル設定の例

8.2.6.1. トランスポート層のグローバル設定

通常使用される設定オプションは、トランスポート層を設定します。これにより、どのようにノードが他のノードを検出するかが Red Hat JBoss Data Grid に通知されます。

トランスポート層の設定

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .transport().defaultTransport()
    .build();
```

8.2.6.2. キャッシュマネージャー名のグローバル設定

以下のサンプル設定では、グローバル JMX 統計レベルからオプションを使用してキャッシュマネージャーの名前を設定できます。この名前は、特定のキャッシュマネージャーと同じシステムの他のキャッシュマネージャーを区別します。

キャッシュマネージャー名の設定

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .globalJmxStatistics()
    .cacheManagerName("SalesCacheManager")
    .mBeanServerLookup(new JBossMBeanServerLookup())
    .enable()
    .build();
```

8.2.6.3. JGroups のグローバル設定

Red Hat JBoss Data Grid がクラスターモードで動作するには、適切な JGroups 設定が必要になります。以下の設定例は、事前定義された JGroups 設定ファイルを設定に渡す方法を示しています。

JGroups のプログラムを使用した設定

```
GlobalConfiguration gc = new GlobalConfigurationBuilder()
    .transport()
    .defaultTransport()
    .addProperty("configurationFile", "jgroups.xml")
    .build();
```

Red Hat JBoss Data Grid は最初にクラスパスにある **jgroups.xml** を検索します。クラスパスでインスタンスが見つからない場合は、絶対パス名を検索します。

8.2.7. キャッシュレベル設定の例

8.2.7.1. クラスターモードのキャッシュレベル設定

以下の設定により、グローバルでないキャッシュレベルでのキャッシュのクラスターモードなどのオプションを使用できます。

キャッシュレベルでのクラスターモードの設定

```
Configuration config = new ConfigurationBuilder()
    .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .sync()
    .l1().lifespan(25000L).enable()
    .hash().numOwners(3)
    .build();
```

8.2.7.2. キャッシュレベルのエビクションおよびエクスパレーションの設定

以下の設定を使用して、キャッシュレベルでキャッシュのエクスパレーションまたはエビクションオプションを設定します。

エクスパレーションおよびエビクションのキャッシュレベルでの設定

```
Configuration config = new ConfigurationBuilder()
    .memory()
    .size(20000)
    .expiration()
    .wakeUpInterval(5000L)
    .maxIdle(120000L)
    .build();
```

8.2.7.3. JTA トランザクションのキャッシュレベルの設定

JTA トランザクション設定のキャッシュと対話するには、トランザクション層を設定し、任意でロック設定をカスタマイズします。トランザクションキャッシュでは、トランザクションリカバリーを有効にして未完了のトランザクションに対応することが推奨されます。さらに、JMX 管理および統計収集も有効にすることが推奨されます。

JTA トランザクションのキャッシュレベルでの設定

```
Configuration config = new ConfigurationBuilder()
    .locking()

    .concurrencyLevel(10000).isolationLevel(IsolationLevel.REPEATABLE_READ)

    .lockAcquisitionTimeout(12000L).useLockStriping(false).writeSkewCheck(true)
)
    .transaction()
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .recovery().enable()
    .jmxStatistics().enable()
    .build();
```

8.2.7.4. チェーンされた永続ストアを使用したキャッシュレベルでの設定

以下の設定は、キャッシュレベルで1つまたは複数のチェーンされた永続ストアを設定するために使用できます。

チェーンされた永続ストアのキャッシュレベルでの設定

```
Configuration conf = new ConfigurationBuilder()
    .persistence()
    .passivation(false)
    .addSingleFileStore()
    .location("/tmp/firstDir")
    .persistence()
    .passivation(false)
    .addSingleFileStore()
    .location("/tmp/secondDir")
    .build();
```

8.2.7.5. 高度なエクスターナライザーのキャッシュレベルでの設定

高度なエクスターナライザーに対するキャッシュレベル設定などの高度なオプションは、以下のようにプログラムを使用して設定することもできます。

高度なエクスターナライザーのキャッシュレベルでの設定

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .serialization()
    .addAdvancedExternalizer(new PersonExternalizer())
    .addAdvancedExternalizer(999, new AddressExternalizer())
    .build();
```

8.2.7.6. パーティション処理のキャッシュレベルでの設定 (ライブラリーノード)

スプリットブレインが発生した場合、パーティション処理ストラテジーを選択して、データの一貫性または可用性を提供できます。可用性を選択し、データの一貫性が失われる場合、マージポリシーを選択して、ノードが再参加時にデータをマージする方法を定義することもできます。設定例を以下に示します。

```
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
    .whenSplit(PartitionHandling.DENY_READ_WRITES)
    .mergePolicy(MergePolicies.REMOVE_ALL);
```

パーティション処理に関する詳細は、『[Administration and Configuration Guide](#)』を参照してください。



注記

クライアントサーバーモードでパーティション処理を設定するには、『[Administration and Configuration Guide](#)』の説明にしたがって宣言的に有効にする必要があります。

第9章 EXTERNALIZABLE API

9.1. EXTERNALIZABLE API

Externalizer は以下を実行できるクラスです。

- 該当するオブジェクトタイプをバイトアレイにマーシャリングします。
- バイトアレイの内容のオブジェクトタイプのインスタンスに対するマーシャリングを解除します。

エクスターナライザーは Red Hat JBoss Data Grid によって使用され、ユーザーはオブジェクトタイプをどのようにシリアル化するかを指定できます。Red Hat JBoss Data Grid で使用されるマーシャリングインフラストラクチャーは、JBoss Marshalling に基づいて構築され、効率的なペイロード配信を提供し、ストリームのキャッシュを可能にします。ストリームキャッシングを使用すると、データに複数回アクセスできますが、通常はストリームは 1 度だけ読み取ることができます。

Externalizable インターフェースはシリアライゼーションを使用および拡張します。このインターフェースは、Red Hat JBoss Data Grid でシリアル化とデシリアル化を制御するために使用されます。

9.2. エクスターナライザーのカスタマイズ

Red Hat JBoss Data Grid のデフォルト設定では、分散またはレプリケートされたキャッシュで使われるすべてのオブジェクトはシリアル化可能である必要があります。デフォルトの Java シリアル化メカニズムでは、ネットワークやパフォーマンスの効率が悪化する可能性があります。また、シリアル化のバージョン管理や後方互換性などに懸念があります。

スループットやパフォーマンスを向上し、特定のオブジェクトの互換性を強制するには、カスタマイズされたエクスターナライザーを使用します。Red Hat JBoss Data Grid のカスタマイズされたエクスターナライザーは、以下の 2 つ方法のいずれかで使用できます。

- Externalizable インターフェースを使用します。
- 高度なエクスターナライザーを使用します。

9.3. @SERIALIZEWITH を使用したマーシャリングのオブジェクトのアノテーション付け

オブジェクトをマーシャルするには、マーシャルまたはアンマーシャルする必要があるタイプにエクスターナライザー実装を提供し、マーシャルされたタイプクラスに **@SerializeWith** アノテーションを付けて使用するエクスターナライザークラスを示します。

@SerializeWith アノテーションの使用

```
import org.infinispan.commons.marshall.Externalizer;
import org.infinispan.commons.marshall.SerializeWith;

@SerializeWith(Person.PersonExternalizer.class)
public class Person {

    final String name;
    final int age;
```

```

public Person(String name, int age) {
    this.name = name;
    this.age = age;
}

public static class PersonExternalizer implements Externalizer<Person>
{
    @Override
    public void writeObject(ObjectOutput output, Person person)
        throws IOException {
        output.writeObject(person.name);
        output.writeInt(person.age);
    }

    @Override
    public Person readObject(ObjectInput input)
        throws IOException, ClassNotFoundException {
        return new Person((String) input.readObject(), input.readInt());
    }
}

```

この例では、**@SerializeWith** アノテーションによってオブジェクトがマーシャル可能であると定義されています。よって、JBoss Marshalling は渡されたエクスターナライザークラスを使用してオブジェクトをマーシャルします。

この方法はエクスターナライザーを簡単に定義できますが、以下の欠点があります。

- このやり方で生成されたペイロードサイズは最も効率的ではありません。これは、モデルの制約の一部 (同じクラスの異なるバージョンへのサポートなど) や、エクスターナライザークラスをマーシャルする必要性が原因です。
- このモデルでは、マーシャルされるクラスに **@SerializeWith** アノテーションを付ける必要があります。しかし、ソースコードが利用できないクラスにエクスターナライザーを提供する必要がある可能性があり、他の制約では変更できません。
- このモデルで使用するアノテーションは、マーシャリング層などの下位レベルの詳細を抽象化しようとするフレームワーク開発者やサービスプロバイダーが制限されることがあります。

このような欠点に影響されるユーザーは高度なエクスターナライザーを使用できます。



注記

エクスターナライザー実装のコーディングを容易にし、よりタイプセーフにするには、タイプ `<t>` をマーシャルまたはアンマーシャルされたオブジェクトのタイプとして定義します。

9.4. 高度なエクスターナライザーの使用

9.4.1. 高度なエクスターナライザーの使用

カスタマイズされた高度なエクスターナライザーを使用すると、Red Hat JBoss Data Grid のパフォーマンスを最適化できます。

1. **readObject()** および **writeObject()** メソッドを定義および実装します。

2. エクスターナライザーをマーシャラークラスとリンクします。
3. 高度なエクスターナライザーを登録します。

9.4.2. メソッドの実装

高度なエクスターナライザーを使用するには、**readObject()** および **writeObject()** メソッドを定義および実装します。定義の例は以下のとおりです。

メソッドの定義および実装

```
import org.infinispan.commons.marshall.AdvancedExternalizer;

public class Person {

    final String name;
    final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public static class PersonExternalizer implements
AdvancedExternalizer<Person> {
        @Override
        public void writeObject(ObjectOutput output, Person person)
            throws IOException {
            output.writeObject(person.name);
            output.writeInt(person.age);
        }

        @Override
        public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
            return new Person((String) input.readObject(), input.readInt());
        }

        @Override
        public Set<Class<? extends Person>> getTypeClasses() {
            return Util.<Class<? extends Person>>asSet(Person.class);
        }

        @Override
        public Integer getId() {
            return 2345;
        }
    }
}
```



注記

このメソッドにはアノテーションが付けられたユーザークラスは必要ありません。そのため、このメソッドはソースコードの使用や変更が不可能なクラスに有効です。

9.4.3. エクスターナライザーとマーシャラークラスのリンク

`getTypeClasses()` の実装を使用してこのエクスターナライザーによるマーシャリングが可能なクラスを検索し、`readObject()` および `writeObject()` クラスへリンクします。

実装例を以下に示します。

```
import org.infinispan.util.Util;
<!-- Additional configuration information here -->
@Override
public Set<Class<? extends ReplicableCommand>> getTypeClasses() {
    return Util.asSet(LockControlCommand.class, GetKeyValueCommand.class,
        ClusteredGetCommand.class, MultipleRpcCommand.class,
        SingleRpcCommand.class, CommitCommand.class,
        PrepareCommand.class, RollbackCommand.class,
        ClearCommand.class, EvictCommand.class,
        InvalidateCommand.class, InvalidateL1Command.class,
        PutKeyValueCommand.class, PutMapCommand.class,
        RemoveCommand.class, ReplaceCommand.class);
}
```

この例では、**ReplicableCommandExternalizer** は複数のコマンドタイプを外部化できることを示しています。この例では **ReplicableCommand** インターフェースを拡張するすべてのコマンドをマーシャルしますが、フレームワークはクラスの等価比較のみをサポートするため、マーシャルされたクラスがすべて特定のクラスまたはインターフェースの子であることを示すことはできません。

場合によっては、外部化するクラスがプライベートであるためクラスインスタンスにアクセスできないことがあります。このような場合、提供された完全修飾クラス名でクラスを検索し、戻します。例は次のとおりです。

```
@Override
public Set<Class<? extends List>> getTypeClasses() {
    return Util.<Class<? extends List>>asSet(
        Util.<List>loadClass("java.util.Collections$SingletonList", null));
}
```

9.4.4. 高度なエクスターナライザーの登録 (プログラムを使用)

高度なエクスターナライザーの設定後、Red Hat JBoss Data Grid で使用するために登録します。以下のようにプログラムを使用して登録します。

高度なエクスターナライザーのプログラムを使用した登録

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
    .addAdvancedExternalizer(new Person.PersonExternalizer());
```

最初の行に `GlobalConfigurationBuilder` の必要な情報を入力します。

9.4.5. 複数のエクスターナライザーの登録

`GlobalConfiguration.addExternalizer()` は `varargs` を許可するため、高度なエクスターナライザーを複数登録することもできます。新しいエクスターナライザーを登録する前に、`@Marshall` アノテーションを使用して ID が定義されていることを確認します。

エクスターナライザーの複数登録

```
builder.serialization()
    .addAdvancedExternalizer(new Person.PersonExternalizer(),
                             new Address.AddressExternalizer());
```

9.5. エクスターナライザー ID 値のカスタマイズ

9.5.1. エクスターナライザー ID 値のカスタマイズ

高度なエクスターナライザーには必要な場合にカスタム ID を割り当てることができます。一部の ID 範囲は他のモジュールやフレームワーク用に予約されるため、その使用を避ける必要があります。

表9.1 エクスターナライザーの予約される ID 範囲

ID 範囲	対象
1000-1099	Infinispan Tree モジュール
1100-1199	Red Hat JBoss Data Grid Server モジュール
1200-1299	Hibernate Infinispan 2 次キャッシュ
1300-1399	JBoss Data Grid Lucene Directory
1400-1499	Hibernate OGM
1500-1599	Hibernate Search
1600-1699	Infinispan Query モジュール
1700-1799	Infinispan Remote Query モジュール
1800-1849	JBoss Data Grid Scripting モジュール
1850-1899	JBoss Data Grid Server Event Logger モジュール
1900-1999	JBoss Data Grid リモートストア

9.5.2. エクスターナライザー ID のカスタマイズ (プログラムを使用)

以下の設定を使用して、プログラムを使用して特定の ID をエクスターナライザーに割り当てます。

エクスターナライザーへの ID の割り当て

```
GlobalConfiguration globalConfiguration = new GlobalConfigurationBuilder()
    .serialization()
    .addAdvancedExternalizer($ID, new
```

```
Person.PersonExternalizer()  
    .build();
```

\$ID を指定する ID に置き換えます。

第10章 通知/リスナー API

10.1. 通知/リスナー API

Red Hat JBoss Data Grid は、発生時にイベントの通知を行うリスナー **API** を提供します。クライアントは、関係する通知に対してリスナー **API** の登録を選択できます。この **API** はアノテーション駆動型で、キャッシュレベルのイベントとキャッシュマネージャーレベルのイベント上で操作します。

10.2. リスナーの例

次の例は、新しいエントリーがキャッシュに追加されるたびに情報を出力する Red Hat JBoss Data Grid のリスナーを定義します。

リスナーの設定

```
@Listener
public class PrintWhenAdded {
    @CacheEntryCreated
    public void print(CacheEntryCreatedEvent event) {
        System.out.println("New entry " + event.getKey() + " created in the
cache");
    }
}
```

10.3. リスナー通知

10.3.1. リスナー通知

キャッシュイベントが発生するたびに通知がリスナーに送信されます。リスナーは、**@Listener** アノテーションが付いた簡単な **POJO** です。Listenable は、実装がアタッチするリスナーを持つことを意味するインターフェースです。各リスナーは Listenable で定義されたメソッドを使用して登録されます。

キャッシュレベルまたはキャッシュマネージャーレベルの通知を受信するため、リスナーはキャッシュおよびキャッシュマネージャーの両方にアタッチすることが可能です。

10.3.2. キャッシュレベルの通知

Red Hat JBoss Data Grid では、キャッシュレベルのイベントはキャッシュごとに発生します。キャッシュレベルイベントの例には、関係するキャッシュで登録されたリスナーへの通知を引き起こすエントリーの追加、削除、および変更などが含まれます。

10.3.3. キャッシュマネージャーレベルの通知

Red Hat JBoss Data Grid のキャッシュマネージャーレベルで発生するイベントの例は次のとおりです。

- キャッシュの開始および停止
- クラスターに参加するノードまたはクラスターから離脱するノード

キャッシュマネージャーレベルのイベントはグローバルに位置し、クラスター全体で使用されますが、単一のキャッシュマネージャーによって作成されたキャッシュ内のイベントに制限されます。

最初の 2 つのイベントである **CacheStarted** および **CacheStopped** は大変似ています。以下の例は、開始または停止されたキャッシュの名前を出力します。

```
@CacheStarted
public void cacheStarted(CacheStartedEvent event){
    // Print the name of the Cache that started
    log.info("Cache Started: " + event.getCacheName());
}

@CacheStopped
public void cacheStopped(CacheStoppedEvent event){
    // Print the name of the Cache that stopped
    log.info("Cache Stopped: " + event.getCacheName());
}
```

ViewChangedEvent または **MergeEvent** を受信するとき、新旧メンバーのリストはイベントが生成されたノードから送信されることに注意してください。例として、以下のシナリオについて考えてみましょう。

- 現在、JDG クラスターはノード A、B、および C で構成されます。
- ノード D がクラスターに参加します。
- ノード A、B、および C は、[A,B,C] を旧メンバーのリストとし、[A,B,C,D] を新メンバーのリストとする **ViewChangedEvent** を受信します。
- ノード D は、[D] を旧メンバーのリストとし、[A,B,C,D] を新メンバーのリストとする **ViewChangedEvent** を受信します。

よって、ノードのクラスターへの参加またはクラスターからの離脱を判断するために積集合が使用されることがあります。以下のように **getOldMembers()** と **getNewMembers()** を使用すると、クラスターに参加したノードや離脱したノードのセットを判断できます。

```
@ViewChanged
public void viewChanged(ViewChangedEvent event){
    HashSet<Address> oldMembers = new HashSet(event.getOldMembers());
    HashSet<Address> newMembers = new HashSet(event.getNewMembers());
    HashSet<Address> oldCopy = (HashSet<Address>)oldMembers.clone();

    // Remove all new nodes from the old view.
    // The resulting set indicates nodes that have left the cluster.
    oldCopy.removeAll(newMembers);
    if(oldCopy.size() > 0){
        for (Address oldAdd : oldCopy){
            log.info("Node left:" + oldAdd.toString());
        }
    }

    // Remove all old nodes from the new view.
    // The resulting set indicates nodes that have joined the cluster.
    newMembers.removeAll(oldMembers);
    if(newMembers.size() > 0){
        for(Address newAdd : newMembers){
            log.info("Node joined: " + newAdd.toString());
        }
    }
}
```

```
}

```

MergeEvent の実行中に同様の論理を使用して、クラスターの新しいメンバーセットを判断することもできます。

10.3.4. 同期および非同期通知

デフォルトでは、Red Hat JBoss Data Grid の通知はイベントが生成された同じスレッドで送信されます。そのため、スレッドの進行を妨げないようにリスナーを書く必要があります。

この代わりに、別のスレッドで通知を送信し、元のスレッドの操作を妨げないようにするために、リスナーを非同期としてアノテーション付けすることもできます。

以下を使用してリスナーにアノテーションを付けます。

```
@Listener (sync = false)
public class MyAsyncListener { .... }
```

XML 設定ファイルで **asyncListenerExecutor** 要素を使用し、非同期通知の送信に使用されるスレッドプールを調整します。



重要

CacheEntryExpiredEvent を処理する非クラスターの同期リスナーを使用する場合、非クラスター環境ではエクスパーションリパー (reaper) も同期であるため、リスナーが実行を妨害しないようにしてください。

10.4. キャッシュエントリーの変更

10.4.1. キャッシュエントリーの変更

キャッシュエントリーの作成後、プログラムを使用してキャッシュエントリーを変更できます。

10.4.2. キャッシュエントリーが変更されたリスナーの設定

キャッシュエントリーが変更されたリスナーイベントでは、**getValue()** メソッドの動作は、実際の操作の実行前または実行後にコールバックがトリガーされたかどうかによって特定されます。たとえば、**event.isPre()** が true の場合、**event.getValue()** は変更の前に古い値を返します。**event.isPre()** が false の場合、**event.getValue()** は新しい値を返します。イベントによって新しいエントリーが作成および挿入される場合、古い値は null になります。**isPre()** の詳細は、Red Hat JBoss Data Grid の『[API Documentation](#)』に記載されている

org.infinispan.notifications.cachelistener.event パッケージのリストを参照してください。

Listenable および **FilteringListenable** インターフェース (Cache オブジェクトによって実装される) によって公開されるメソッドを使用する場合のみ、プログラムを使用してリスナーを設定できます。

10.4.3. キャッシュエントリーが変更されたリスナーの例

以下の例は、キャッシュエントリーが変更されるたびに情報を出力する Red Hat JBoss Data Grid のリスナーを定義します。

変更されたリスナー

```
@Listener
public class PrintWhenModified {

    @CacheEntryModified
    public void print(CacheEntryModifiedEvent event) {
        System.out.println("Cache entry modified. Details = " + event);
    }

}
```

10.5. クラスター化リスナー

10.5.1. クラスター化リスナー

クラスター化リスナーは、リスナーを分散キャッシュ設定でできるようにします。分散キャッシュ環境では、イベントが発生したノードのローカルイベントのみが登録されたローカルリスナーに通知されます。クラスター化リスナーでは、イベントが発生したノードに関係なくクラスターで発生する書き込み通知を1つのリスナーが受信できるようにし、この問題に対応します。そのため、クラスター化リスナーのパフォーマンスは、イベントが発生するノードのイベント通知のみを提供する非クラスター化リスナーよりも遅くなります。

クラスター化リスナーを使用する場合、特定のキャッシュでエントリーの追加、更新、期限切れ、または削除が行われるとクライアントアプリケーションに通知が送られます。イベントはクラスター全体であるため、アプリケーションが存在するノードや接続するノードに関係なく、クライアントアプリケーションはイベントにアクセスできます。

イベントは、常にリスナーが登録されたノード上で発生し、キャッシュの更新が発生した場所に関係しません。

10.5.2. クラスター化リスナーの設定

以下のユースケースではリスナーは受け取ったイベントを保存します。

手順: クラスター化リスナーの設定

```
@Listener(clustered = true)
protected static class ClusterListener {
    List<CacheEntryEvent> events = Collections.synchronizedList(new
ArrayList<CacheEntryEvent>());

    @CacheEntryCreated
    @CacheEntryModified
    @CacheEntryExpired
    @CacheEntryRemoved
    public void onCacheEvent(CacheEntryEvent event) {
        log.debugf("Adding new cluster event %s", event);
        events.add(event);
    }
}

public void addClusterListener(Cache<?, ?> cache) {
```

```

ClusterListener clusterListener = new ClusterListener();
cache.addListener(clusterListener);
}

```

1. クラスター化リスナーを有効にするには、**@Listener** クラスに **clustered=true** を付けます。
2. エントリーの追加、変更、期限切れ、または削除時にクライアントアプリケーションが通知を受けるようにするため、以下のメソッドにはアノテーションが付けられます。

- **@CacheEntryCreated**
- **@CacheEntryModified**
- **@CacheEntryExpired**
- **@CacheEntryRemoved**

3. リスナーはキャッシュで登録され、オプションでフィルターまたはコンバーターを渡すことができます。

クラスター化リスナーを使用すると、非クラスターリスナーでは適用されない以下の制限が適用されます。

- クラスターリスナーは作成されたエントリー、変更されたエントリー、期限切れのエントリー、または削除されたエントリーのみをリッスンできます。他のイベントはクラスター化リスナーによってリッスンされません。
- ポストイベントのみがクラスター化リスナーに送信され、プレイベントは無視されます。

10.5.3. キャッシュリスナー API

addListener メソッドを使用すると、クラスター化されたリスナーを既存の **@CacheListener** API の上に追加できます。

キャッシュリスナー API

```
cache.addListener(Object listener, Filter filter, Converter converter);
```

```

public @interface Listener {
    boolean clustered() default false;
    boolean includeCurrentState() default false;
    boolean sync() default true;
}

```

```

interface CacheEventFilter<K,V> {
    public boolean accept(K key, V oldValue, Metadata oldMetadata, V
newValue, Metadata newMetadata, EventType eventType);
}

```

```

interface CacheEventConverter<K,V,C> {
    public C convert(K key, V oldValue, Metadata oldMetadata, V newValue,
Metadata newMetadata, EventType eventType);
}

```

Cache API

ローカルまたはクラスター化リスナーは `cache.addListener` メソッドで登録でき、以下のイベントの 1 つが発生するまでアクティブな状態になります。

- `cache.removeListener` を呼び出してリスナーが明示的に登録解除される。
- リスナーが登録されたノードがクラッシュする。

リスナーアノテーション

リスナーアノテーションは以下の 3 つの属性で強化されます。

- `clustered()`: この属性は、アノテーションが付けられたリスナーがクラスター化されているかどうかを定義します。クラスター化されたリスナーは `@CacheEntryRemoved`、`@CacheEntryCreated`、`@CacheEntryExpired`、および `@CacheEntryModified` イベントのみの通知を受けることができます。この属性はデフォルトで `false` に指定されています。
- `includeCurrentState()`: この属性はクラスター化リスナーのみに適用され、デフォルトで `false` に指定されています。`true` に設定すると、クラスター内で既存の状態全体が評価されます。登録されると、キャッシュの各エントリーの `CacheCreatedEvent` が即座にリスナーに送信されます。
- `sync()` に関する詳細は、「[同期および非同期の通知](#)」を参照してください。

oldValue および oldMetadata

`oldValue` および `oldMetadata` の値は、`CacheEventFilter` および `CacheEventConverter` クラスの許可メソッド上の追加メソッドです。これらの値は、ローカルリスナーを含むすべてのリスナーに提供されます。これらの値の詳細は、[JBoss Data Grid の API ドキュメント](#) を参照してください。

EventType

`EventType` には、イベントのタイプ、再試行であったかどうか、およびプレまたはポストイベントであったかどうかが含まれます。

クラスター化リスナーを使用する場合、キャッシュが更新される順序は通知を受け取る順序に反映されます。

クラスター化リスナーは、イベントが 1 度だけ送信されることを保証しません。同じイベントが複数回送信されないようにするため、リスナー実装はべき等である必要があります。安定したクラスターや、`includeCurrentState` の結果として合成イベントが生成されるタイムスパン外部では、単一性が受け入れられることをインプリメンターは想定することができます。

10.5.4. クラスター化リスナーの例

次のユースケースは、ニューヨーク州のニューヨーク市宛の注文がいつ生成されるか知りたいリスナーを表しています。リスナーには、ニューヨークから出入りする注文をフィルターする `Filter` が必要です。さらに、注文全体は必要ではなく、配達される日付のみが必要であるため、`Converter` も必要になります。

ユースケース: ニューヨーク宛の注文のフィルターおよび変換

```
class CityStateFilter implements CacheEventFilter<String, Order> {
    private String state;
    private String city;
```



```

public boolean accept(String orderId, Order oldOrder,
                      Metadata oldMetadata, Order newOrder,
                      Metadata newMetadata, EventType eventType) {
    switch (eventType.getType()) {
        // Only send update if the order is going to our city
        case CACHE_ENTRY_CREATED:
            return city.equals(newOrder.getCity()) &&
                   state.equals(newOrder.getState());
        // Only send update if our order has changed from our city to
        // elsewhere or if is now going to our city
        case CACHE_ENTRY_MODIFIED:
            if (city.equals(oldOrder.getCity()) &&
                state.equals(oldOrder.getState())) {
                // If old city matches then we have to compare if new
                // order is no longer going to our city
                return !city.equals(newOrder.getCity()) ||
                       !state.equals(newOrder.getState());
            } else {
                // If the old city doesn't match ours then only send
                // update if new update does match ours
                return city.equals(newOrder.getCity()) &&
                       state.equals(newOrder.getState());
            }
        // On remove we have to send update if our order was
        // originally going to city
        case CACHE_ENTRY_REMOVED:
            return city.equals(oldOrder.getCity()) &&
                   state.equals(oldOrder.getState());
    }
    return false;
}

```

```

class OrderDateConverter implements CacheEventConverter<String, Order,
Date> {
    private String state;
    private String city;

    public Date convert(String orderId, Order oldValue,
                        Metadata oldMetadata, Order newValue,
                        Metadata newMetadata, EventType eventType) {
        // If remove we do not care about date - this tells listener to
        // remove its data
        if (eventType.isRemove()) {
            return null;
        } else if (eventType.isModified()) {
            if (state.equals(newValue.getState()) &&
                city.equals(newValue.getCity())) {
                // If it is a modification meaning the destination has
                // changed to ours then we allow it
                return newValue.getDate();
            } else {
                // If destination is no longer our city it means it was
                // changed from us so send null
                return null;
            }
        }
    }
}

```

```

    }
  } else {
    // This was a create so we always send date
    return newValue.getDate();
  }
}
}

```

10.5.5. 最適化されたキャッシュフィルターコンバーター

「[クラスター化リスナーの例](#)」では、結果のフィルターおよび変換を一度に実行するために、最適化された **CacheEventFilterConverter** を使用できます。

CacheEventFilterConverter は、イベントのフィルターと変換を一度に実行できるようにする最適化です。これは、イベントフィルターとコンバーターが同じオブジェクトとして最も効率的に使用され、同じメソッドでフィルターと変換が行われる場合に使用できます。戻り値が null の場合は値がフィルターを通過しなかったことを意味するため、変換が null 値を返さない場合のみ使用できます。null 値を変換するには、**CacheEventFilter** および **CacheEventConverter** インターフェースを独立して使用します。

以下は、**CacheEventFilterConverter** を使用したニューヨーク宛の注文のユースケース例になります。

CacheEventFilterConverter

```

class OrderDateFilterConverter extends
AbstractCacheEventFilterConverter<String, Order, Date> {
    private final String state;
    private final String city;

    public Date filterAndConvert(String orderId, Order oldValue,
                                Metadata oldMetadata, Order newValue,
                                Metadata newMetadata, EventType
eventType) {
        // Remove if the date is not required - this tells listener to
        // remove its data
        if (eventType.isRemove()) {
            return null;
        } else if (eventType.isModified()) {
            if (state.equals(newValue.getState()) &&
                city.equals(newValue.getCity())) {
                // If it is a modification meaning the destination has
                // changed to ours then we allow it
                return newValue.getDate();
            } else {
                // If destination is no longer our city it means it was
                // changed from us so send null
                return null;
            }
        } else {
            // This was a create so we always send date
            return newValue.getDate();
        }
    }
}

```

リスナーの登録時、フィルターとコンバーターの両方の引数として **FilterConverter** を提供します。

```
OrderDateFilterConverter filterConverter = new
OrderDateFilterConverter("NY", "New York");
cache.addListener(listener, filterConveter, filterConverter);
```

10.6. リモートイベントリスナー (HOT ROD)

10.6.1. リモートイベントリスナー (Hot Rod)

イベントリスナーは、Red Hat JBoss Data Grid Hot Rod サーバーが **CacheEntryCreated**、**CacheEntryModified**、**CacheEntryExpired**、**CacheEntryRemoved** などのイベントのリモートクライアントを通知できるようにします。接続されたクライアントが殺到しないようにするため、クライアントはこれらのイベントをリッスンするかどうかを選択することができます。クライアントがサーバーへの永続接続を維持することが前提となります。

リモートイベントのクライアントリスナーは、ライブラリーモードのクラスター化リスナーと同様に追加できます。以下は、受け取った各イベントを出力するリモートクライアントリスナーの例になります。

イベント出力リスナー

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class EventLogListener {

    @ClientCacheEntryCreated
    public void handleCreatedEvent(ClientCacheEntryCreatedEvent e) {
        System.out.println(e);
    }

    @ClientCacheEntryModified
    public void handleModifiedEvent(ClientCacheEntryModifiedEvent e) {
        System.out.println(e);
    }

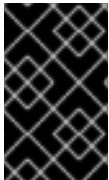
    @ClientCacheEntryExpired
    public void handleExpiredEvent(ClientCacheEntryExpiredEvent e) {
        System.out.println(e);
    }

    @ClientCacheEntryRemoved
    public void handleRemovedEvent(ClientCacheEntryRemovedEvent e) {
        System.out.println(e);
    }
}
```

- **ClientCacheEntryCreatedEvent** および **ClientCacheEntryModifiedEvent** インスタンスは、キーおよびエントリーのバージョンに関する情報を提供します。このバージョンは、**replaceWithVersion** や **removeWithVersion** などのサーバー上の条件付き操作を呼

び出すために使用されます。

- 期限切れのエントリーで **get()** が呼び出された場合、またはエクスパレーションリーパーがエントリーの期限切れを検出した場合、**ClientCacheEntryExpiredEvent** イベントが送信されます。エントリーの期限が切れると、キャッシュがエントリーを null にし、サイズを適切に調整します。しかし、イベントは 2 つのシナリオでのみ生成されます。
- **ClientCacheEntryRemovedEvent** イベントは、削除操作が成功した場合のみ送信されます。削除操作が呼び出された場合にエントリーが見つからなかったり、削除するエントリーがないと、イベントは生成されません。イベント削除が成功するかどうかに関わらずイベントの削除が必要な場合は、カスタマイズされたイベント論理を作成します。
- すべてのクライアントキャッシュエントリーの作成、変更、および削除されたイベントは、トポロジーの変更により書き込みコマンドを再試行しなければならない場合に **true** を返す **boolean isCommandRetried()** メソッドを提供します。これは、イベントが複製されたか、別のイベントが破棄または置換された (作成されたイベントが変更されたイベントに置き換えられるなど) ことを意味します。



重要

想定されるワークロードで書き込みが読み取りよりも優先される場合、送信するイベントをフィルタし、クライアントやネットワークで問題となる可能性がある過剰なトラフィックが大量に生成されないようにします。

10.6.2. イベントリスナーの追加および削除

サーバーでのイベントリスナーの登録

以下の例は、サーバーでイベント出力リスナーを登録します。「[イベント出力リスナー](#)」を参照してください。

イベントリスナーの追加

```
RemoteCache<Integer, String> cache = rcm.getCache();
cache.addClientListener(new EventLogListener());
```

クライアントイベントリスナーの削除

クライアントイベントリスナーは以下のように削除できます。

```
EventLogListener listener = ...
cache.removeClientListener(listener);
```

10.6.3. リモートイベントクライアントリスナーの例

以下は、Hot Rod 経由でリモートキャッシュと対話するようにリモートクライアントリスナーを設定するために必要な手順になります。

リモートイベントリスナーの設定

1. Red Hat カスタマーポータルから Red Hat JBoss Data Grid ディストリビューションをダウンロードします。
最新の JBoss Data Grid ディストリビューションには、クライアントが通信する Hot Rod サーバーが含まれています。

2. サーバーの起動

JBoss Data Grid サーバーのルートから以下のコマンドを実行し、サーバーを起動します。

```
$ ./bin/standalone.sh
```

3. Hot Rod サーバーと対話するアプリケーションの作成

a. Maven ユーザー

以下の依存関係でアプリケーションを作成します。バージョンは **8.5.0.Final-redhat-9** 以上に変更します。

```
<properties>
  <infinispan.version>8.5.0.Final-redhat-9</infinispan.version>
</properties>
[...]
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-remote</artifactId>
  <version>${infinispan.version}</version>
</dependency>
```

b. Maven 以外のユーザーは、使用するビルドツールに合わせて調整するか、すべての JBoss Data Grid jar が含まれるディストリビューションをダウンロードします。

4. クライアントアプリケーションの作成

以下は、受け取ったイベントをすべてログに記録する簡単なリモートイベントリスナーを示しています。

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class EventLogListener {

    @ClientCacheEntryCreated
    @ClientCacheEntryModified
    @ClientCacheEntryRemoved
    public void handleRemoteEvent(ClientEvent event) {
        System.out.println(event);
    }

}
```

5. リモートイベントリスナーを使用したリモートキャッシュに対する操作の実行

以下は、リモートイベントリスナーを追加し、リモートキャッシュに対して一部の操作を実行する簡単なメイン java クラスの例になります。

```
RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache<Integer, String> cache = rcm.getCache();
EventLogListener listener = new EventLogListener();
try {
    cache.addClientListener(listener);
    cache.put(1, "one");
    cache.put(1, "new-one");
}
```

```

        cache.remove(1);
    } finally {
        cache.removeClientListener(listener);
    }
}

```

結果

実行すると、以下と似たコンソール出力が表示されます。

```

ClientCacheEntryCreatedEvent(key=1,dataVersion=1)
ClientCacheEntryModifiedEvent(key=1,dataVersion=2)
ClientCacheEntryRemovedEvent(key=1)

```

出力は、デフォルトでイベントにはキーと現在の値に関連するデータバージョンが含まれることを示しています。実際の値はパフォーマンス上の理由でクライアントへ返送されません。リモートイベントの受け取りはパフォーマンスに影響します。キャッシュサイズが大きくなるとより多くの操作が実行されるため、パフォーマンスへの影響も大きくなります。Hot Rod クライアントにイベントが大量に送られないようにするには、サーバー側でリモートイベントをフィルターするか、イベントコンテンツをカスタマイズします。

10.6.4. リモートイベントのフィルター

10.6.4.1. リモートイベントのフィルター

クライアントに大量のイベントが送信されないようにするため、Red Hat JBoss Data Grid の Hot Rod リモートイベントをフィルターすることができます。フィルターするには、クライアントに送信されるイベントやクライアント提供の情報でこれらのフィルターが動作する方法をフィルターするインスタンスを作成するキーバリューのフィルターファクトリーを提供します。

リモートクライアントへのイベント送信はパフォーマンスに影響します。リモートリスナーが登録されたクライアントの数が増えると、パフォーマンスのコストも増加します。また、キャッシュに対して実行された変更の数が増えると、パフォーマンスへの影響も大きくなります。

サーバー側で送信されるイベントをフィルターするとパフォーマンスのコストを削減できます。カスタムコードを使用して、一部のイベントがリモートクライアントへブロードキャストされないようにしてパフォーマンスを向上することができます。

キーや値の情報またはキャッシュエントリーメタデータの情報を基にしてフィルターを行うことができます。フィルターを有効にするには、フィルターインスタンスを作成するキャッシュイベントフィルターファクトリーを作成する必要があります。以下は、クライアントに送信されたイベントから キー「2」をフィルターする簡単な実装になります。

KeyValueFilter

```

package sample;

import java.io.Serializable;
import org.infinispan.notifications.cachelistener.filter.*;
import org.infinispan.metadata.*;

@NamedFactory(name = "basic-filter-factory")
public class BasicKeyValueFilterFactory implements CacheEventFilterFactory
{
    @Override public CacheEventFilter<Integer, String> getFilter(final
Object[] params) {

```

```

    return new BasicKeyValueFilter();
}

static class BasicKeyValueFilter implements CacheEventFilter<Integer,
String>, Serializable {
    @Override public boolean accept(Integer key, String oldValue,
Metadata oldMetadata, String newValue, Metadata newMetadata, EventType
eventType) {
        return !"2".equals(key);
    }
}
}

```

このキーバリューフィルターファクトリーでリスナーを登録するには、ファクトリーに一意的な名前を付け、Hot Rod サーバーをその名前とキャッシュイベントフィルターファクトリーインスタンスでプラグする必要があります。

10.6.4.2. リモートイベントのカスタムフィルター

カスタムフィルターは、一部のイベント情報がリモートクライアントへブロードキャストされないようにしてパフォーマンスを向上できます。

カスタムフィルターで JBoss Data Grid サーバーをプラグするには、以下の手順を使用します。

カスタムフィルターの使用

1. 内部にフィルター実装が含まれる **JAR** ファイルを作成します。**org.infinispan.filter.NamedFactory** アノテーションを使用して、各ファクトリーに名前を割り当てる必要があります。例は **KeyValueFilterFactory** を使用します。
2. **JAR** ファイル内に **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory** ファイルを作成し、その内部にフィルタークラス実装の完全修飾クラス名を記述します。
3. 以下のオプションの 1 つを実行し、JBoss Data Grid サーバーに **JAR** ファイルをデプロイします。

- オプション 1: デプロイメントスキャナーを用いた JAR のデプロイ
 - **JAR** を **\$JDG_HOME/standalone/deployments/** ディレクトリーにコピーします。デプロイメントスキャナーはこのディレクトリーをアクティブに監視し、新たに配置されたファイルをデプロイします。

- オプション 2: CLI を用いた JAR のデプロイ
 - CLI で目的のインスタンスに接続します。

```
[ $JDG_HOME ] $ bin/cli.sh --connect=$IP:$PORT
```

- 接続後、**deploy** コマンドを実行します。

```
deploy /path/to/artifact.jar
```

- オプション 3: JAR をカスタムモジュールとしてデプロイ

- 以下のコマンドを実行して JDG サーバーに接続します。

```
[ $JDG_HOME ] $ bin/cli.sh --connect=$IP:$PORT
```

- カスタムフィルターが含まれる jar はサーバーのモジュールとして定義する必要があります。追加するには、以下のコマンドのモジュール名と .jar 名を置き換え、カスタムフィルターに追加の依存関係が必要な場合は追加します。

```
module add --name=$MODULE-NAME --resources=$JAR-NAME.jar --
dependencies=org.infinispan
```

- 別のウインドウ

で、`$JDG_HOME/modules/system/layers/base/org/infinispan/main/module.xml` を編集して、新規追加したモジュールを依存関係として `org.infinispan` モジュールに追加します。このファイルに以下のエントリーを追加します。

```
<dependencies>
  [...]
  <module name="$MODULE-NAME">
</dependencies>
```

- JDGを サーバーを再起動します。

サーバーがフィルターでプラグされたら、そのフィルターを使用するリモートクライアントリスナーを追加します。以下の例は、リモートイベントクライアントリスナーの例 ([「リモートイベントクライアントリスナーの例」](#)を参照) を拡張し、`@ClientListener` アノテーションをオーバーライドしてリスナーと使用するフィルターファクトリーを示します。

フィルターファクトリーのリスナーへの追加

```
@org.infinispan.client.hotrod.annotation.ClientListener(filterFactoryName
= "basic-filter-factory")
public class BasicFilteredEventLogListener extends EventLogListener {}
```

リスナーは `RemoteCacheAPI` を使用して追加できるようになりました。以下の例はこれを実証し、一部の操作をリモートキャッシュに対して実行します。

サーバーでのリスナーの登録

```
import org.infinispan.client.hotrod.*;

RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache<Integer, String> cache = rcm.getCache();
BasicFilteredEventLogListener listener = new
BasicFilteredEventLogListener();
try {
  cache.addClientListener(listener);
  cache.putIfAbsent(1, "one");
  cache.replace(1, "new-one");
  cache.putIfAbsent(2, "two");
  cache.replace(2, "new-two");
  cache.putIfAbsent(3, "three");
  cache.replace(3, "new-three");
  cache.remove(1);
}
```



```

    cache.remove(2);
    cache.remove(3);
} finally {
    cache.removeClientListener(listener);
}

```

システム出力は、フィルターされたものを除くすべてのキーのイベントをクライアントが受け取ること
を示しています。

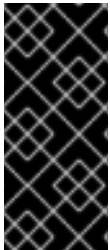
結果

以下は、例の結果となるシステム出力を示しています。

```

ClientCacheEntryCreatedEvent(key=1,dataVersion=1)
ClientCacheEntryModifiedEvent(key=1,dataVersion=2)
ClientCacheEntryCreatedEvent(key=3,dataVersion=5)
ClientCacheEntryModifiedEvent(key=3,dataVersion=6)
ClientCacheEntryRemovedEvent(key=1)
ClientCacheEntryRemovedEvent(key=3)

```



重要

イベントがリスナーが登録されたノード以外で生成された場合でも、イベントが生成された場所でフィルターが実行されるようにするため、フィルターインスタンスはクラスターへのデプロイ時にマーシャル可能である必要があります。フィルターインスタンスをマーシャル可能にするには、フィルターインスタンスが `Serializable` や `Externalizable` を拡張するようにするか、カスタムエクスターナライザーを提供します。

10.6.4.3. 強化されたフィルターファクトリー

クライアントリスナーの追加時、パラメーターをフィルターファクトリーに提供して、クライアント側の情報を基に単一のフィルターファクトリーから挙動の異なるさまざまなフィルターを生成することができます。

以下の設定は、静的に提供されたキーでフィルターを行う代わりに、リスナーの追加時に提供されたキーを基に動的にフィルターを実行できるようにするためのフィルターファクトリーの強化方法を表しています。

強化されたフィルターファクトリーの設定

```

package sample;

import java.io.Serializable;
import org.infinispan.notifications.cachelistener.filter.*;
import org.infinispan.metadata.*;

@NamedFactory(name = "basic-filter-factory")
public class BasicKeyValueFilterFactory implements CacheEventFilterFactory
{
    @Override public CacheEventFilter<Integer, String> getFilter(final
Object[] params) {
        return new BasicKeyValueFilter(params);
    }

    static class BasicKeyValueFilter implements CacheEventFilter<Integer,

```

```
String>, Serializable {
    private final Object[] params;
    public BasicKeyValueFilter(Object[] params) { this.params = params; }
    @Override public boolean accept(Integer key, String oldValue, Metadata
oldMetadata, String newValue, Metadata newMetadata, EventType eventType) {
        return !params[0].equals(key);
    }
}
}
```

このフィルターは「2」ではなく「3」でフィルターが実行されるようになりました。

強化されたフィルターファクトリーの実行

```
import org.infinispan.client.hotrod.*;

RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache<Integer, String> cache = rcm.getCache();
BasicFilteredEventLogListener listener = new
BasicFilteredEventLogListener();
try {
    cache.addClientListener(listener, new Object[]{3}, null); // <- Filter
parameter passed
    cache.putIfAbsent(1, "one");
    cache.replace(1, "new-one");
    cache.putIfAbsent(2, "two");
    cache.replace(2, "new-two");
    cache.putIfAbsent(3, "three");
    cache.replace(3, "new-three");
    cache.remove(1);
    cache.remove(2);
    cache.remove(3);
} finally {
    cache.removeClientListener(listener);
}
```

結果

例の結果として、以下が出力されます。

```
ClientCacheEntryCreatedEvent(key=1,dataVersion=1)
ClientCacheEntryModifiedEvent(key=1,dataVersion=2)
ClientCacheEntryCreatedEvent(key=2,dataVersion=3)
ClientCacheEntryModifiedEvent(key=2,dataVersion=4)
ClientCacheEntryRemovedEvent(key=1)
ClientCacheEntryRemovedEvent(key=2)
```

リモートイベントをカスタマイズすると、クライアントに送信する情報量をさらに削減することができます。

10.6.5. リモートイベントのカスタマイズ

10.6.5.1. リモートイベントのカスタマイズ

Red Hat JBoss Data Grid では、Hot Rod リモートイベントをカスタマイズしてクライアントへ送信する必要がある情報が含まれるようにすることができます。デフォルトでは、クライアントが過負荷にならないようにし、情報送信のコストを削減するため、イベントにはキーやイベントタイプなどの基本的な情報のみが含まれます。

これらのイベントに含まれる情報をカスタマイズし、値などの追加情報が含まれるようにしたり、より少ない情報が含まれるようにすることができます。カスタマイズするには、**CacheEventConverterFactory** クラスを実装して作成される **CacheEventConverter** インスタンスを使用します。各ファクトリーには、**@NamedFactory** アノテーションを使用したこれに関連する名前が必要です。

Red Hat JBoss Data Grid サーバーをイベントコンバーターでプラグするには、以下の手順を使用します。

コンバーターの使用

1. コンバーター実装が含まれる **JAR** ファイルを作成します。各ファクトリーには、**org.infinispan.filter.NamedFactory** アノテーションを使用したこれに関連する名前が必要です。
2. **JAR** ファイル内に **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory** ファイルを作成し、その内部にコンバータークラス実装の完全修飾クラス名を記述します。
3. 以下のオプションの 1 つを実行し、Red Hat JBoss Data Grid サーバーに **JAR** ファイルをデプロイします。

- オプション 1: デプロイメントスキャナーを用いた JAR のデプロイ

- **JAR** を **\$JDG_HOME/standalone/deployments/** ディレクトリーにコピーします。デプロイメントスキャナーはこのディレクトリーをアクティブに監視し、新たに配置されたファイルをデプロイします。

- オプション 2: CLI を用いた JAR のデプロイ

- CLI で目的のインスタンスに接続します。

```
[ $JDG_HOME ] $ bin/cli.sh --connect=$IP:$PORT
```

- 接続後、**deploy** コマンドを実行します。

```
deploy /path/to/artifact.jar
```

- オプション 3: JAR をカスタムモジュールとしてデプロイ

- 以下のコマンドを実行して JDG サーバーに接続します。

```
[ $JDG_HOME ] $ bin/cli.sh --connect=$IP:$PORT
```

- カスタムコンバーターが含まれる jar はサーバーのモジュールとして定義する必要があります。追加するには、以下のコマンドのモジュール名と .jar 名を置き換え、カスタムコンバーターに追加の依存関係が必要な場合は追加します。

```
module add --name=$MODULE-NAME --resources=$JAR-NAME.jar --dependencies=org.infinispan
```

- 別のウィンドウ
で、`$JDG_HOME/modules/system/layers/base/org/infinispan/main/module.xml` を編集して、新規追加したモジュールを依存関係として `org.infinispan` モジュールに追加します。このファイルに以下のエントリーを追加します。

```
<dependencies>
  [...]
  <module name="$MODULE-NAME">
</dependencies>
```

- JDGを サーバーを再起動します。

コンバーターはクライアントが提供する情報にも対応でき、コンバーターインスタンスはリスナーの追加時に提供された情報を基にイベントをカスタマイズできます。API は、リスナーの追加時にコンバーターパラメーターが渡されるようにします。

10.6.5.2. コンバーターの追加

リスナーの追加時、リスナーと使用するコンバーターファクトリーの名前が提供されます。リスナーが追加されると、サーバーはファクトリーを検索し、サーバー側のイベントをカスタマイズするために `getConverter` メソッドを呼び出して `org.infinispan.filter.Converter` クラスインスタンスを取得します。

以下の例は、Integer (整数) と String (文字列) のキャッシュに対して値の情報が含まれるカスタムイベントをリモートクライアントに送信します。コンバーターは、値とイベントのキーが含まれる新しいカスタムイベントを生成します。カスタムイベントのイベントペイロードはデフォルトのイベントよりも大きくなりますが、フィルターを組み合わせると帯域幅のコストを削減できます。

カスタムイベントの送信

```
import org.infinispan.notifications.cachelistener.filter.*;

@NamedFactory(name = "value-added-converter-factory")
class ValueAddedConverterFactory implements CacheEventConverterFactory {
    // The following types correspond to the Key, Value, and the returned
    // Event, respectively.
    public CacheEventConverter<Integer, String, ValueAddedEvent>
    getConverter(final Object[] params) {
        return new ValueAddedConverter();
    }

    static class ValueAddedConverter implements CacheEventConverter<Integer,
    String, ValueAddedEvent> {
        public ValueAddedEvent convert(Integer key, String oldValue,
        Metadata oldMetadata, String
        newValue,
        Metadata newMetadata, EventType
        eventType) {
            return new ValueAddedEvent(key, newValue);
        }
    }
}

// Must be Serializable or Externalizable.
class ValueAddedEvent implements Serializable {
```

```

    final Integer key;
    final String value;
    ValueAddedEvent(Integer key, String value) {
        this.key = key;
        this.value = value;
    }
}

```

10.6.5.3. ライトウェイトイベント

他のコンバーター実装は、キーやイベントタイプ情報が含まれないイベントを返送できます。そのため、イベントが大幅に軽量化されますが、イベントによって詳細情報は提供されません。

このコンバーターでサーバーをプラグするには、コンバーターファクトリーと関連するコンバータークラスを **JAR** ファイル内にデプロイし、以下のように **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory** ファイル内のサービス定義が含まれるようにします。

```
sample.ValueAddedConverterFactory
```

この後、ファクトリー名を **@ClientListener** アノテーションに追加して、クライアントリスナーをコンバーターファクトリーとリンクする必要があります。

```

@ClientListener(converterFactoryName = "value-added-converter-factory")
public class CustomEventLogListener { ... }

```

10.6.5.4. 動的なコンバーターインスタンス

動的なコンバーターインスタンスは、リスナーの登録時に提供されたパラメーターを基にして変換を行います。コンバーターはコンバーターファクトリーによって受信されたパラメーターを使用してこのオプションを有効にします。例は以下のとおりです。

動的なコンバーター

```

import
org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import
org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

class DynamicCacheEventConverterFactory implements
CacheEventConverterFactory {
    // The following types correspond to the Key, Value, and the returned
    // Event, respectively.
    public CacheEventConverter<Integer, String, CustomEvent>
getConverter(final Object[] params) {
    return new DynamicCacheEventConverter(params);
}
}

// Serializable, Externalizable or marshallable with Infinispan
// Externalizers needed when running in a cluster
class DynamicCacheEventConverter implements CacheEventConverter<Integer,
String, CustomEvent>, Serializable {

```

```

    final Object[] params;

    DynamicCacheEventConverter(Object[] params) {
        this.params = params;
    }

    public CustomEvent convert(Integer key, String oldValue, Metadata
metadata, String newValue, Metadata prevMetadata, EventType eventType) {
        // If the key matches a key given via parameter, only send the key
information
        if (params[0].equals(key))
            return new ValueAddedEvent(key, null);

        return new ValueAddedEvent(key, newValue);
    }
}

```

変換の実行に必要な動的パラメーターはリスナーの登録時に提供されます。

```

RemoteCache<Integer, String> cache = rcm.getCache();
cache.addClientListener(new EventLogListener(), null, new Object[]{1});

```

10.6.5.5. カスタムイベントのリモートクライアントリスナーの追加

カスタムイベントのリスナーの実装は、デフォルトでないイベントが関係するため、他のリモートイベントの場合とは若干異なります。他のリモートクライアントリスナー実装と同じアノテーションが使用されますが、コールバックは **ClientCacheEntryCustomEvent<T>** のインスタンスを受信します (T はサーバーから送信するカスタムイベントのタイプになります)。例を以下に示します。

カスタムイベントリスナー実装

```

import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener(converterFactoryName = "value-added-converter-factory")
public class CustomEventLogListener {

    @ClientCacheEntryCreated
    @ClientCacheEntryModified
    @ClientCacheEntryRemoved
    public void
handleRemoteEvent(ClientCacheEntryCustomEvent<ValueAddedEvent> event)
    {
        System.out.println(event);
    }
}

```

リモートイベントリスナーを使用してリモートキャッシュに対して操作を実行するには、リモートイベントリスナーを追加してリモートキャッシュに対して操作を実行する簡単なメイン Java クラスを作成します。例を以下に示します。

リモートキャッシュに対する操作の実行

```

import org.infinispan.client.hotrod.*;

```

```

RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache<Integer, String> cache = rcm.getCache();
CustomEventLogListener listener = new CustomEventLogListener();
try {
    cache.addClientListener(listener);
    cache.put(1, "one");
    cache.put(1, "new-one");
    cache.remove(1);
} finally {
    cache.removeClientListener(listener);
}

```

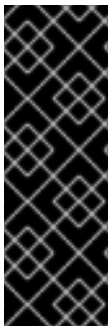
結果

実行すると、以下と似たコンソール出力が表示されます。

```

ClientCacheEntryCustomEvent(eventData=ValueAddedEvent{key=1, value='one'},
eventType=CLIENT_CACHE_ENTRY_CREATED)
ClientCacheEntryCustomEvent(eventData=ValueAddedEvent{key=1, value='new-
one'}, eventType=CLIENT_CACHE_ENTRY_MODIFIED)
ClientCacheEntryCustomEvent(eventData=ValueAddedEvent{key=1,
value='null'}, eventType=CLIENT_CACHE_ENTRY_REMOVED)

```



重要

リスナーが登録されたノードではない別のノードでイベントが生成された場合でも、イベントが生成された場所に変換が発生するようにするには、コンバーターインスタンスがクラスターにデプロイされたときにマーシャル可能である必要があります。コンバーターインスタンスをマーシャル可能にするには、`Serializable` または `Externalizable` を拡張するようにするか、カスタムの `Externalizer` を提供します。タイプセーフな API に対して記述するサーバーとクライアントの両方を円滑にするため、クライアントとサーバーの両方がカスタムイベントタイプを認識し、マーシャルできる必要があります。

10.6.6. イベントマーシャリング

イベントをフィルターまたはカスタマイズする際、**KeyValueFilter** および **Converter** インスタンスはマーシャル可能でなければなりません。クライアントリスナーがクラスターにインストールされると、イベントの送信元でフィルターと変換が行われるようにするため、フィルターインスタンスとコンバーターインスタンスはクラスターの他のノードに送信されます。これにより効率がよくなります。これらのクラスをマーシャル可能にするには、これらのクラスが `Serializable` を拡張するようにするか、カスタム `Externalizer` を提供および登録します。

Marshaller インスタンスをサーバー側にデプロイするには、フィルターおよびカスタマイズされたイベントに使用されるメソッドに似たメソッドを使用します。

マーシャラーのデプロイ

1. コンバーター実装が含まれる **JAR** ファイルを作成します。各ファクトリーには、**org.infinispan.filter.NamedFactory** アノテーションを使用したこれに関連する名前が必要です。
2. **JAR** ファイル内に **META-INF/services/org.infinispan.commons.marshall.Marshaller** ファイルを作成し、その内部にマーシャラークラス実装の完全修飾クラス名を記述します。

3. 以下のオプションの 1 つを実行し、Red Hat JBoss Data Grid に **JAR** ファイルをデプロイします。

- オプション 1: デプロイメントスキャナーを用いた JAR のデプロイ
 - **JAR** を **\$JDG_HOME/standalone/deployments/** ディレクトリーにコピーします。デプロイメントスキャナーはこのディレクトリーをアクティブに監視し、新たに配置されたファイルをデプロイします。

- オプション 2: CLI を用いた JAR のデプロイ
 - CLI で目的のインスタンスに接続します。

```
[ $JDG_HOME ] $ bin/cli.sh --connect=$IP:$PORT
```

- 接続後、**deploy** コマンドを実行します。

```
deploy /path/to/artifact.jar
```

- オプション 3: JAR をカスタムモジュールとしてデプロイ

- 以下のコマンドを実行して JDG サーバーに接続します。

```
[ $JDG_HOME ] $ bin/cli.sh --connect=$IP:$PORT
```

- カスタムマーシャラーが含まれる jar はサーバーのモジュールとして定義する必要があります。追加するには、以下のコマンドのモジュール名と .jar 名を置き換え、カスタムマーシャラーに追加の依存関係が必要な場合は追加します。

```
module add --name=$MODULE-NAME --resources=$JAR-NAME.jar --
dependencies=org.infinispan
```

- 別のウィンドウで、**\$JDG_HOME/modules/system/layers/base/org/infinispan/main/module.xml** を編集して、新規追加したモジュールを依存関係として **org.infinispan** モジュールに追加します。このファイルに以下のエントリーを追加します。

```
<dependencies>
  [...]
  <module name="$MODULE-NAME">
</dependencies>
```

- JDGを サーバーを再起動します。

マーシャラーは、個別の jar または CacheEventConverter や CacheEventFilter インスタンスと同じ jar にデプロイすることができます。



注記

1 つの Marshaller インスタンスのデプロイメントのみがサポートされます。複数の Marshaller インスタンスがデプロイされると、警告メッセージが表示され、使用される Marshaller が示されます。

10.6.7. リモートイベントクラスタリングおよびフェイルオーバー

クライアントがリモートリスナーを追加するとき、クラスターの単一のノードにインストールされます。このノードは、クラスター全体で発生する影響を受ける操作すべてに対してイベントをクライアントに返送します。

クラスター化された環境では、リスナーが含まれるノードがダウンした場合、Hot Rod クライアント実装が透過的にクライアントリスナーの登録を別のノードにフェイルオーバーします。これにより、イベントの消費にギャップが生じることがありますが、以下の方法の 1 つを使用すると解決できます。

状態の配信

@ClientListener アノテーションにはオプションの **includeCurrentState** パラメーターがあります。このパラメーターを有効にすると、サーバーは既存のキャッシュエントリすべての **CacheEntryCreatedEvent** イベントインスタンスをクライアントに送信します。この動作は、リスナーが登録されたノードがオフラインになったときに検出されるクライアントによって決定され、自動的にリスナーをクラスターの別のノードに登録します。**includeCurrentState** を有効にすると、Hot Rod クライアントが登録されたリスナーを透過的にフェイルオーバーする場合にクライアントの状態や計算を再算出できます。**includeCurrentState** パラメーターのパフォーマンスは、キャッシュの大きさに影響されるため、デフォルトでは無効になっています。

@ClientCacheFailover

受信状態に依存する代わりに、**@ClientCacheFailover** アノテーションを使用して、クライアントリスナー実装内で **ClientCacheFailoverEvent** パラメーターを受信するメソッドを定義できます。Hot Rod クライアントが登録されたノードでクライアントリスナーに障害が発生すると、Hot Rod クライアントによって障害が透過的に検出され、障害が発生したノードに登録されたすべてのリスナーは別のノードにフェイルオーバーされます。

フェイルオーバーの間にクライアントが一部のイベントを見逃すことがあります。イベントを見逃さないようにするには、**includeCurrentState** パラメーターを **true** に設定します。これを有効にすると、クライアントはデータを消去でき、**CacheEntryCreatedEvent** インスタンスすべてを受信し、すべてのキーでこれらのイベントをキャッシュできます。この代わりに、コールバックハンドラーを追加して、Hot Rod クライアントがフェイルオーバーイベントを認識できるようにすることができます。このコールバックメソッドは、クライアントリスナーに影響するクラスターポロジの変更に対応する効率的な方法で、クライアントリスナーはフェイルオーバーでどのように動作するかを判断することができます。ニアキャッシュはこの方法を利用し、**ClientCacheFailoverEvent** の受信後にニアキャッシュは消去されます。

@ClientCacheFailover

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class EventLogListener {
    // ...

    @ClientCacheFailover
    public void handleFailover(ClientCacheFailoverEvent e) {
        // Deal with client failover, e.g. clear a near cache.
    }
}
```



注記

ClientCacheFailoverEvent は、クライアントリスナーがインストールされたノードに障害が発生した場合のみ発生します。

第11章 JSR-107 (JCACHE) API

11.1. JSR-107 (JCACHE) API

Red Hat JBoss Data Grid 7.2 より、JCache 1.1.0 API の実装 ([JSR-107](#)) が含まれるようになりました。JCache は、一時的な Java オブジェクトをメモリーにキャッシュするために標準の Java API を選択しました。Java オブジェクトをキャッシュすると、読み出しのコストが高いデータ (DB または Web サービス) または計算が困難なデータの使用により発生したボトルネックに対応できます。このようなオブジェクトをメモリーでキャッシュすると、コストの高いラウンドトリップや再計算を行う代わりに、メモリーから直接データを読み出しすることでアプリケーションを高速化することができます。本書では、新しい仕様の Red Hat JBoss Data Grid で JCache を使用方法と API の主な側面を説明します。

11.2. 依存関係

JCache の依存関係を定義するには、Maven で定義するか、クラスパスに追加します。両方の方法について以下に説明します。

11.2.1. オプション 1: Maven

JCache 実装を使用するには、使用方法に応じて以下の依存関係を Maven の **pom.xml** に追加する必要があります。

- **embedded:**

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-embedded</artifactId>
  <version>${infinispan.version}</version>
</dependency>

<dependency>
  <groupId>javax.cache</groupId>
  <artifactId>cache-api</artifactId>
  <version>1.1.0.redhat-1</version>
</dependency>
```

- **remote:**

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-remote</artifactId>
  <version>${infinispan.version}</version>
</dependency>

<dependency>
  <groupId>javax.cache</groupId>
  <artifactId>cache-api</artifactId>
  <version>1.1.0.redhat-1</version>
</dependency>
```

11.2.2. オプション 2: 必要なファイルのクラスパスへの追加

Maven を使用しない場合、必要な jar ファイルが実行時にクラスパスに存在する必要があります。実行時にこれらのファイルが利用できるようにするには、jar ファイルの直接埋め込み、実行時の指定、またはアプリケーションのデプロイに使用されるコンテナへの追加のいずれかを行います。

埋め込みモード

1. Red Hat カスタマーポータルから **Red Hat JBoss Data Grid 7.2.1 Library** をダウンロードします。
2. ダウンロードしたアーカイブをローカルディレクトリーで展開します。
3. 以下のファイルを見つけます。
 - **jboss-datagrid-7.2.1-library/infinispan-embedded-8.5.0.Final-redhat-9.jar**
 - **jboss-datagrid-7.2.1-library/lib/cache-api-1.1.0.redhat-1.jar**
4. 実行時に両方の jar ファイルがクラスパス上にあるようにします。

リモートモード

1. Red Hat カスタマーポータルから **Red Hat JBoss Data Grid 7.2.1 Hot Rod Java Client** をダウンロードします。
2. ダウンロードしたアーカイブをローカルディレクトリーで展開します。
3. 以下のファイルを見つけます。
 - **jboss-datagrid-7.2.1-remote-java-client/infinispan-remote-8.5.0.Final-redhat-9.jar**
 - **jboss-datagrid-7.2.1-remote-java-client/lib/cache-api-1.1.0.redhat-1.jar**
4. 実行時に両方の jar ファイルがクラスパス上にあるようにします。

11.3. ローカルキャッシュの作成

以下の手順にしたがうと、JCache API 仕様の定義どおりにデフォルトの設定オプションを使用して簡単にローカルキャッシュを作成できます。

```
import javax.cache.*;
import javax.cache.configuration.*;

// Retrieve the system wide cache manager
CacheManager cacheManager =
    Caching.getCachingProvider().getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
```



警告

デフォルトでは、データは **storeByValue** として保存されるよう JCache API によって指定されるため、キャッシュへの操作の外部となるオブジェクト状態の変異はキャッシュに保存されたオブジェクトには影響しません。JBoss Data Grid では、シリアライズやマーシャリングを使用してキャッシュに保存するコピーを作成し、これにより仕様に準拠します。そのため、デフォルトの JCache 設定を Infinispan で使用する場合、保存するデータはマーシャル可能である必要があります。

この代わりに、参照でデータを格納するよう JCache を設定することもできます。これには、以下を呼び出します。

```
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>().setStoreByValue(false));
```

11.3.1. ライブラリーモード

ライブラリーモードでは、**CachingProvider.getCacheManager** の **URL** パラメーターで設定ファイルの場所を指定して **CacheManager** を設定することができます。これにより、設定ファイルでクラスター化されたキャッシュを定義する機会が与えられ、キャッシュの名前を **CacheManager.getCache** メソッドに渡して事前設定されたメソッドへの参照を取得できます。これ以外の場合は、**CacheManager.createCache** から作成されたローカルキャッシュのみを使用できます。

11.3.2. クライアントサーバーモード

リモート **CacheManager** のクライアントサーバーモードに固有する設定は、**CachingProvider.getCacheManager** の **properties** パラメーターを使用して標準の Hot Rod クライアントプロパティを渡して実行されます。参照されるリモートサーバーは稼働中である必要があります、リクエストを受信できなければなりません。

指定がない場合は、デフォルトのアドレスとポートが使用されます (127.0.0.1:11222)。さらに、ライブラリーモードとは異なり、キャッシュの参照が初めて取得されたときにキャッシュを内部で登録するために **CacheManager.createCache** を使用する必要があります。後続のクエリーは、**CacheManager.getCache** を使用して実行できます。

11.4. データの格納および読み出し

JCache API は [java.util.Map](#) または [java.util.concurrent.ConcurrentMap](#) を拡張しませんが、キーバリュー API を提供し、データを保存および読み出します。

```
import javax.cache.*;
import javax.cache.configuration.*;

CacheManager cacheManager =
    Caching.getCachingProvider().getCacheManager();
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
```

```
cache.put("hello", "world"); // Notice that javax.cache.Cache.put(K)
                             // returns void!
String value = cache.get("hello"); // Returns "world"
```

標準の [java.util.Map](#) とは異なり、[javax.cache.Cache](#) には **put** および **getAndPut** と呼ばれる 2 つの基本的な put メソッドが含まれます。put は **void** を返し、getAndPut はキーに関連する以前の値を返します。JCache では [javax.cache.Cache.getAndPut\(K\)](#) が [java.util.Map.put\(K\)](#) と同等になります。

ヒント

JCache API はスタンドアロンのキャッシングのみに対応しますが、永続ストアでプラグ可能で、クラスターや分散を考慮して設計されています。[javax.cache.Cache](#) が 2 つの put メソッドを提供する理由は、標準の [java.util.Map.put](#) コールはインプリメンターに以前の値の計算を強制するためです。永続ストアが使用された場合またはキャッシュが分散された場合、以前の値を戻す操作はコストが高くなる可能性があり、ユーザーは多くの場合で戻り値を使用せずに標準の [java.util.Map.put\(K\)](#) を呼び出します。そのため、JCache ユーザーは戻り値が関係するかどうかを考慮する必要があります。以前の値を返す場合は、[javax.cache.Cache.getAndPut\(K\)](#) を呼び出す必要があります。その他の場合は [java.util.Map.put\(K\)](#) を呼び出して、コストが高くなる可能性がある以前の値を戻す操作を行わないようにします。

11.5. JAVA.UTIL.CONCURRENT.CONCURRENTMAP API と JAVAX.CACHE.CACHE API の比較

以下は、[java.util.concurrent.ConcurrentMap](#) API および [javax.cache.Cache](#) API によって提供されるデータ操作 API の簡単な比較になります。

表11.1 [java.util.concurrent.ConcurrentMap](#) と [javax.cache.Cache](#) の比較

操作	java.util.concurrent.ConcurrentMap<K,V>	javax.cache.Cache<K,V>
格納し、戻り値はなし。	該当なし	void put(K key)
格納し、以前の値を返す。	V put(K key)	V getAndPut(K key)
存在しない場合は格納。	V putIfAbsent(K key, V Value)	boolean putIfAbsent(K key, V value)
読み出し	V get(Object key)	V get(K key)
存在する場合は削除。	V remove(Object key)	boolean remove(K key)

操作	java.util.concurrent.Concurrent Map<K,V>	javax.cache.Cache<K,V>
削除し、以前の値を返す。	<code>V remove(Object key)</code>	<code>V getAndRemove(K key)</code>
条件を削除。	<code>boolean remove(Object key, Object value)</code>	<code>boolean remove(K key, V oldValue)</code>
存在する場合は置き換え。	<code>V replace(K key, V value)</code>	<code>boolean replace(K key, V value)</code>
置き換え、以前の値を返す。	<code>V replace(K key, V value)</code>	<code>V getAndReplace(K key, V value)</code>
条件を置き換え。	<code>boolean replace(K key, V oldValue, V newValue)</code>	<code>boolean replace(K key, V oldValue, V newValue)</code>

2つのAPIを比較すると、JCacheはコストの高いネットワークの操作やIO操作を避けるために以前の値を返さないようにします。これは、JCache APIの設計で最も重要な原則です。実際に、分散キャッシュでは計算のコストが高いため、[java.util.concurrent.ConcurrentMap](#)に存在し、[javax.cache.Cache](#)には存在しない操作があります。唯一の例外が、キャッシュのコンテンツでの繰り返しです。

表11.2 戻り値を回避する javax.cache.Cache

操作	java.util.concurrent.Concurrent Map<K,V>	javax.cache.Cache<K,V>
キャッシュサイズの計算。	<code>int size()</code>	該当なし
キャッシュのすべてのキーを返す。	<code>Set<K> keySet()</code>	該当なし
キャッシュのすべての値を返す。	<code>Collection<V> values()</code>	該当なし

操作	java.util.concurrent.Concurrent Map<K,V>	javax.cache.Cache<K,V>
キャッシュのすべてのエントリーを返す。	<code>Set<Map.Entry<K, V>> entrySet()</code>	該当なし
キャッシュでの繰り返し。	keySet、values、または entrySet での use iterator() メソッドの使用。	<code>Iterator<Cache.Entry<K, V>> iterator()</code>

11.6. JCACHE インスタンスのクラスター化

標準の API を使用したクラスターキャッシュの可能性を提供するために、Red Hat JBoss Data Grid 実装は仕様を超えた機能を提供します。キャッシュをレプリケートするために次のような設定ファイルがあるとします。

```
<namedCache name="namedCache">
  <clustering mode="replication"/>
</namedCache>
```

次のコードを使用するとキャッシュのクラスターを作成することができます。

```
import javax.cache.*;
import java.net.URI;

// For multiple cache managers to be constructed with the standard JCache
// API
// and live in the same JVM, either their names, or their classloaders,
// must
// be different.
// This example shows how to force their classloaders to be different.
// An alternative method would have been to duplicate the XML file and
// give
// it a different name, but this results in unnecessary file duplication.
ClassLoader tccl = Thread.currentThread().getContextClassLoader();
CacheManager cacheManager1 = Caching.getCachingProvider().getCacheManager(
    URI.create("infinispan-jcache-cluster.xml"), new
    TestClassLoader(tccl));
CacheManager cacheManager2 = Caching.getCachingProvider().getCacheManager(
    URI.create("infinispan-jcache-cluster.xml"), new
    TestClassLoader(tccl));

Cache<String, String> cache1 = cacheManager1.getCache("namedCache");
Cache<String, String> cache2 = cacheManager2.getCache("namedCache");

cache1.put("hello", "world");
String value = cache2.get("hello"); // Returns "world" if clustering is
// working

// --
```



```
public static class TestClassLoader extends ClassLoader {
    public TestClassLoader(ClassLoader parent) {
        super(parent);
    }
}
```

11.7. 複数のキャッシュプロバイダー

キャッシュプロバイダーはオーバーロードされた **getCachingProvider()** メソッドを使用して **javax.cache.Caching** から取得されます。デフォルトでは、このメソッドはクラスパスで見つかった **META-INF/services/javax.cache.spi.CachingProvider** ファイルをロードしようとします。1つのファイルが見つかった場合、そのファイルが使用中のキャッシュプロバイダーを判断します。

複数のキャッシュプロバイダーがある場合、以下のメソッドの1つを使用して特定のプロバイダーを選択します。

- **getCachingProvider(ClassLoader classLoader)**
- **getCachingProvider(String fullyQualifiedClassName)**

別のキャッシュプロバイダーに変更する場合、デフォルトのクラスパスにそのプロバイダーが存在することを確認してください。または、上記のメソッドの1つを使用してそのプロバイダーを選択してください。

検出されたまたは **Caching** クラスによってロードされた **javax.cache.spi.CachingProviders** は内部レジストリーで維持されます。同じキャッシュプロバイダーの後続リクエストは、キャッシュプロバイダー実装をリロードまたは再インスタンス化する代わりに、このレジストリーから返されます。現在のキャッシュプロバイダーを表示するには、以下のいずれかのメソッドを使用します。

- **getCachingProviders()** - デフォルトクラスローダーのキャッシュプロバイダーのリストを提供します。
- **getCachingProviders(ClassLoader classLoader)** - 指定されたクラスローダーのキャッシュプロバイダーのリストを提供します。

第12章 ヘルスチェック API

12.1. ヘルスチェック API

ヘルスチェック API は、クラスターとその中に含まれるキャッシュの正常性を監視できるようにします。クラスターやキャッシュの状態を報告するクエリーのメソッドを説明するため、この情報は特にクラウド環境で作業を行う場合に重要になります。

この API は以下の情報を公開します。

- クラスターの名前。
- クラスターにあるマシンの数。
- 以下の 3 つの値の 1 つで表される、クラスターまたはキャッシュの総合的な状態。
 - Healthy - エンティティは正常です。
 - Unhealthy - エンティティは正常ではありません。この値は、1 つ以上のキャッシュがデグレード状態であることを表しています。
 - Rebalancing - このエンティティは運用可能ですが、再調整中です。この値が報告されたらクラスターノードを調整してはなりません。
- 各キャッシュの状態。
- サーバーログの末尾。

プログラミングを使用しないでヘルスチェック API を使用方法については、JBoss Data Grid の『**Administration and Configuration Guide**』を参照してください。

12.2. プログラムを使用したヘルスチェック API へのアクセス

ライブラリーモードでは、プログラムを使用した場合のみヘルスチェック API にアクセス可能で、`embeddedCacheManager.getHealth()` メソッドを呼び出してアクセスすることができます。

このメソッドは、以下のメソッドにアクセスできる `org.infinispan.health.Health` オブジェクトを返します。

- `getClusterHealth()` - `ClusterHealth` オブジェクトを返し、以下のメソッドにアクセスできます。
 - `getNumberOfNodes()` - クラスターの全ノード数を表す `int` 型を返します。
 - `getNodeNames()` - クラスターにあるすべてのノードの名前が含まれる `List<String>` を返します。
 - `getClusterName()` - クラスターの名前が含まれる `String` を返します。
 - `getHealthStatus()` - クラスターの正常性が含まれる `HealthStatus` が返され、正常性は `HEALTHY`、`UNHEALTHY`、または `REBALANCING` で報告されます。
- `getHostInfo()` - `HostInfo` オブジェクトを返し、以下のメソッドにアクセスできます。

- **getNumberOfCpus()** - ホストにインストールされた CPU の数が含まれる **int** 型を返します。
- **getTotalMemoryKb()** - KB 単位のメモリーの合計が含まれる **long** 型を返します。
- **getFreeMemoryInKb()** - KB 単位の空きメモリーが含まれる **long** 型を返します。
- **getCacheHealth()** - **List<CacheHealth>** を返します。各 **CacheHealth** オブジェクトは以下のメソッドにアクセスできます。
 - **getCacheName()** - キャッシュの名前が含まれる **String** を返します。
 - **getStatus()** - キャッシュの正常性が含まれる **HealthStatus** が返され、正常性は **HEALTHY**、**UNHEALTHY**、または **REBALANCING** で報告されます。

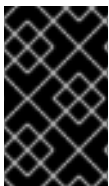
第13章 REST API

13.1. REST インターフェース

Red Hat JBoss Data Grid は REST インターフェースを提供し、クライアントとサーバー間の疎結合を可能にします。REST インターフェースの主な利点は既存の HTTP クライアントとの相互運用性や、php クライアントへの接続の提供です。さらに、特定のバージョンのクライアントライブラリーやバインディングが必要ありません。

REST API によってオーバーヘッドが発生します。REST コールの理解や作成に REST クライアントまたはカスタムコードが必要になります。パフォーマンスに懸念がある場合は Hot Rod クライアントの使用が推奨されます。

Red Hat JBoss Data Grid の REST API と対話するには、HTTP クライアントライブラリーのみが必要です。Java の場合は、Apache HTTP Commons Client や java.net API になります。



重要

以下の例は、REST セキュリティーが REST コネクターで無効になっていることを前提としています。REST セキュリティーを削除するには、コネクターから **authentication** および **encryption** 要素を削除します。

13.2. RUBY クライアントコード

以下のコードは ruby を使用して Red Hat JBoss Data Grid の REST API と対話する例です。提供されたコードは特別なライブラリーを必要とせず、標準的な net/HTTP ライブラリーを使用できます。

Ruby での REST API の使用

```
require 'net/http'

http = Net::HTTP.new('localhost', 8080)

#An example of how to create a new entry

http.post('/rest/MyData/MyKey', 'DATA_HERE', {"Content-Type" =>
"text/plain"})

#An example of using a GET operation to retrieve the key

puts http.get('/rest/MyData/MyKey').body

#An Example of using a PUT operation to overwrite the key

http.put('/rest/MyData/MyKey', 'MORE DATA', {"Content-Type" =>
"text/plain"})

#An example of Removing the remote copy of the key

http.delete('/rest/MyData/MyKey')

#An example of creating binary data
```

```
http.put('/rest/MyImages/Image.png',
File.read('/Users/michaelneale/logo.png'), {"Content-Type" =>
"image/png"})
```

13.3. RUBY の例で JSON を使用

前提条件

ruby で JavaScript Object Notation (**JSON**) を使用して Red Hat JBoss Data Grid の REST インターフェースと対話するには、JSON Ruby ライブラリーをインストールし (プラットフォームのパッケージマネージャーまたは Ruby ドキュメンテーションを参照)、以下のコードを使用して要件を宣言します。

```
require 'json'
```

Ruby での JSON の使用

以下のコードは、Ruby で JavaScript Object Notation (**JSON**) と **PUT** 関数を使用して特定のデータ (この場合は、個人の名前と年齢) を送信する例です。

```
data = {:name => "michael", :age => 42 }
http.put('/rest/Users/data/0', data.to_json, {"Content-Type" =>
"application/json"})
```

13.4. PYTHON クライアントコード

以下のコードは Python を使用して Red Hat JBoss Data Grid の REST API と対話する例です。提供されたコードは、標準的な HTTP ライブラリーのみを必要とします。

Python での REST API の使用

```
import httpplib

#How to insert data

conn = httpplib.HTTPConnection("localhost:8080")
data = "SOME DATA HERE \!" #could be string, or a file...
conn.request("POST", "/rest/default/0", data, {"Content-Type":
"text/plain"})
response = conn.getresponse()
print response.status

#How to retrieve data

import httpplib
conn = httpplib.HTTPConnection("localhost:8080")
conn.request("GET", "/rest/default/0")
response = conn.getresponse()
print response.status
print response.read()
```

13.5. JAVA クライアントコード

以下のコードは Java を使用して Red Hat JBoss Data Grid の REST API と対話する例です。

インポートの定義

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.URL;
```

String 値をキャッシュに追加

```
// Using the imports in the previous example
public class RestExample {

    /**
     * Method that puts a String value in cache.
     * @param urlServerAddress
     * @param value
     * @throws IOException
     */

    public void putMethod(String urlServerAddress, String value) throws
IOException {
        System.out.println("-----");
        System.out.println("Executing PUT");
        System.out.println("-----");
        URL address = new URL(urlServerAddress);
        System.out.println("executing request " + urlServerAddress);
        HttpURLConnection connection = (HttpURLConnection)
address.openConnection();
        System.out.println("Executing put method of value: " + value);
        connection.setRequestMethod("PUT");
        connection.setRequestProperty("Content-Type", "text/plain");
        connection.setDoOutput(true);

        OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(connection.getOutputStream());
        outputStreamWriter.write(value);

        connection.connect();
        outputStreamWriter.flush();

        System.out.println("-----");
        System.out.println(connection.getResponseCode() + " " +
connection.getResponseMessage());
        System.out.println("-----");

        connection.disconnect();
    }
}
```

以下のコードは、Red Hat JBoss Data Grid REST インターフェースと対話するために Java を使用して URL に指定された値を読み取るメソッドの例です。

キャッシュから String 値を取得

```

    // Continuation of RestExample defined in previous example
    /**
     * Method that gets an value by a key in url as param value.
     * @param urlServerAddress
     * @return String value
     * @throws IOException
     */
    public String getMethod(String urlServerAddress) throws IOException {
        String line = new String();
        StringBuilder stringBuilder = new StringBuilder();

        System.out.println("-----");
        System.out.println("Executing GET");
        System.out.println("-----");

        URL address = new URL(urlServerAddress);
        System.out.println("executing request " + urlServerAddress);

        HttpURLConnection connection = (HttpURLConnection)
address.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Content-Type", "text/plain");
        connection.setDoOutput(true);

        BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));

        connection.connect();

        while ((line = bufferedReader.readLine()) != null) {
            stringBuilder.append(line + '\n');
        }

        System.out.println("Executing get method of value: " +
stringBuilder.toString());

        System.out.println("-----");
        System.out.println(connection.getResponseCode() + " " +
connection.getResponseMessage());
        System.out.println("-----");

        connection.disconnect();

        return stringBuilder.toString();
    }

```

Java Main メソッドの使用

```

    // Continuation of RestExample defined in previous example
    /**
     * Main method example.
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {

```

```
//Note that the cache name is "cacheX"
RestExample restExample = new RestExample();
restExample.putMethod("http://localhost:8080/rest/cacheX/1",
    "Infinispan REST Test");
restExample.getMethod("http://localhost:8080/rest/cacheX/1");
    }
}
```

13.6. REST インターフェースの使用

13.6.1. REST インターフェース操作

Red Hat JBoss Data Grid のリモートクライアントサーバーモードでは、クライアントが以下を実行できるようにする REST インターフェースが提供されます。

- データの追加
- データの取得
- データの削除
- データのクエリー

13.6.1.1. データ形式

REST API は、設定可能なメディアタイプによって定義される形式にデータを保存するキャッシュを公開します。

以下の XML スニペットは、キーと値のメディアタイプを定義する設定例になります。

```
<cache>
  <encoding>
    <key media-type="application/x-java-object;
type=java.lang.Integer"/>
    <value media-type="application/xml; charset=UTF-8"/>
  </encoding>
</cache>
```

13.6.1.2. デフォルトのデータ形式

デフォルトのメディアタイプはキーと値の両方で **application/octet-stream** であり、以下の例外があります。

- インデックス化されたキャッシュのデフォルトのメディアタイプは **application/x-protostream** です。
- 互換性モードを使用するキャッシュのデフォルトのメディアタイプは **application/x-java-object** です。

13.6.1.3. サポートされるデータ形式

Red Hat JBoss Data Grid では、クライアントは異なる形式のデータを読み書きでき、自動的に形式を変換できます。

JBoss Data Grid は、相互に交換可能なデータ形式を複数サポートします。

- **application/x-java-object**
- **application/octet-stream**
- **application/x-www-form-urlencoded**
- **text/plain**

JBoss Data Grid は、前述のリストにあるデータ形式が変換前および変換後となるデータ形式もサポートします。

- **application/xml**
- **application/json**
- **application/x-jboss-marshalling**
- **application/x-java-serialized**
- **application/x-protostream**

さらに、JBoss Data Grid は **application/x-protostream** と **application/json** 間の変換をサポートします。

13.6.1.4. ヘッダー

Red Hat JBoss Data Grid REST API への呼び出しは、以下を記述するヘッダーを提供できます。

- キャッシュに書き込まれたコンテンツ。
- キャッシュから読み取るときに必要なコンテンツのデータ形式。

JBoss Data Grid は、値に適用される HTTP/1.1 **Content-Type** および **Accept** ヘッダーと、キーの **Key-Content-Type** ヘッダーをサポートします。

13.6.1.5. Accept ヘッダー

Red Hat JBoss Data Grid の REST サーバーは、**Accept** ヘッダーの RFC-2616 仕様に準拠し、サポートされる変換を基にして適切なメディアタイプをネゴシエートします。

たとえば、キャッシュからデータを読み取るため、クライアントが呼び出しで以下のヘッダーを送信するとします。

```
Accept: text/plain;q=0.7, application/json;q=0.8, */*;q=0.6
```

この場合、ネゴシエーション中に JBoss Data Grid はメディアタイプの優先度が最も高い (0.8) **JSON** 形式を優先します。サーバーが JSON 形式をサポートしない場合、次に優先度が高い (0.7) **text/plain** が優先されます。

サーバーが **JSON** および **text/plain** メディアタイプをサポートしない場合は、**/** が優先されます。これは、キャッシュ設定を基にした適切なメディアタイプを意味します。

ネゴシエーションの終了後、サーバーは選択したメディアタイプを継続して操作に使用します。操作中にエラーが発生した場合、サーバーは他のメディアタイプの使用を試みません。

13.6.1.6. Key-Content-Type ヘッダー

REST API へのほとんどの呼び出しは URL にキーが含まれています。これらの呼び出しを処理するとき、Red Hat JBoss Data Grid はデフォルトで `java.lang.String` をキーのコンテンツタイプとして使用します。しかし、**Key-Content-Type** ヘッダーを使用して別のコンテンツタイプをキーに指定することもできます。

表13.1 Key-Content-Type ヘッダーの例

用途	API 呼び出し	Header
<code>byte[]</code> キーを Base64 文字列として指定	PUT <code>/my-cache/AQIDBDM=</code>	Key-Content-Type: <code>application/octet-stream</code>
<code>byte[]`</code> キーを 16 進文字列として指定	GET <code>/my-cache/0x01CA03042F</code>	Key-Content-Type: <code>application/octet-stream; encoding=hex</code>
ダブルキーを指定	POST <code>/my-cache/3.141456</code>	Key-Content-Type: <code>application/x-java-object; type=java.lang.Double</code>



注記

`application/x-java-object` の `type` パラメーターはプリミティブラッパー型と `java.lang.String` に制限されます。さらにこのパラメーターはバイトに制限され、結果として `application/x-java-object; type=Bytes` が `application/octet-stream; encoding=hex` と同等になります。

13.6.2. REST API を介したデータの追加

13.6.2.1. データをキャッシュに追加

以下のメソッドでデータをキャッシュに追加します。

- HTTP **PUT** メソッド
- HTTP **POST** メソッド

PUT および **POST** メソッドで REST API を呼び出すとき、リクエストのボディーにはデータが含まれます。

13.6.2.2. PUT `/[{cacheName}]/[{cacheKey}]`

提供された URL フォームからの **PUT** 要求により、提供されたキーを使用して要求本文からのペイロードがターゲットキャッシュに配置されます。このタスクが正常に完了するには、ターゲットキャッシュがサーバーに存在する必要があります。

例として、以下の URL では値 **hr** がキャッシュ名で **payRoll%2F3** がキーになります。値 **%2F** は、/ 文字がキーで使用されたことを示しています。

```
http://someserver/rest/hr/payRoll%2F3
```

既存のデータが置き換えられ、必要な場合は **Time-To-Live** および **Last-Modified** の値が更新されます。



注記

以下の引数を使用してサーバーが起動された場合は、キーの / を表す値 **%2F** を含むキャッシュキー (提供された例を参照) を正常に実行できます。

```
-Dorg.apache.tomcat.util.buf.UDecoder.ALLOW_ENCODED_SLASH=true
```

13.6.2.3. POST /{cacheName}/{cacheKey}

提供された URL フォームからの **POST** メソッドにより、提供されたキーを使用して (要求本文からの) ペイロードがターゲットキャッシュに配置されます。ただし、**POST** メソッドでは、値がキャッシュ/キーに存在する場合に、**HTTP CONFLICT** ステータスが返され、内容が更新されません。

13.6.2.4. PUT および POST メソッドのヘッダー

表13.2 PUT および POST メソッドのヘッダー

Header	任意または必須	説明
Key-Content-Type	任意	URL のキーのコンテンツタイプを指定します。
Content-Type	任意	REST API に送信された値のメディアタイプを指定します。
performAsync	任意	ブール値を指定します。値が true の場合はすぐに返され、独自にデータをクラスターにレプリケートします。これは、一括してデータを挿入する場合や大型のクラスターにデータを挿入する場合に便利です。
timeToLiveSeconds	任意	エントリーが自動的に削除されるまでの期間を秒数で指定します。負の値を指定すると、永久に削除されないエントリーが作成されます。

Header	任意または必須	説明
maxIdleTimeSeconds	任意	エントリーがアイドル状態でいられる期間を秒数で指定します。この期間を過ぎるとエントリーは削除されます。負の値を指定すると、永久に削除されないエントリーが作成されます。

timeToLiveSeconds ヘッダーおよび **maxIdleTimeSeconds** ヘッダーには以下の組み合わせを設定できます。

- **timeToLiveSeconds** ヘッダーと **maxIdleTimeSeconds** ヘッダーの両方に値 **0** が割り当てられた場合、キャッシュは XML またはプログラムを使用して設定されたデフォルトの **timeToLiveSeconds** 値と **maxIdleTimeSeconds** 値を使用します。
- **maxIdleTimeSeconds** ヘッダー値のみが **0** に設定された場合は、**timeToLiveSeconds** 値をパラメーターとして渡す必要があります (パラメーターが存在しない場合はデフォルトの **-1**)。さらに **maxIdleTimeSeconds** パラメーター値は、デフォルトで XML またはプログラムを使用して設定された値に設定されます。
- **timeToLiveSeconds** ヘッダー値のみが **0** に設定された場合は、エクスパレーションが即座に発生し、**maxIdleTimeSeconds** 値がパラメーターとして渡された値に設定されます (パラメーターが提供されなかった場合はデフォルトの **-1**)。

13.6.3. REST API を介したデータの取得

13.6.3.1. キャッシュからのデータの取得

以下のメソッドでキャッシュからデータを取得します。

- HTTP **GET** メソッド
- HTTP **HEAD** メソッド

13.6.3.2. GET /{cacheName}/{cacheKey}

GET メソッドは、指定の **cacheName** に存在し、関連するキーに一致するデータを応答の本文として返します。Content-Type ヘッダーは、データのタイプを提供します。ブラウザーはキャッシュに直接アクセスできます。

各エントリーに対して、要求された **URL** でデータの状態を示す Last-Modified ヘッダーとともに一意のエンティティタグ (ETag) が返されます。ETag により、ブラウザー (および他のクライアント) は、(帯域幅を節約するために) データが変更された場合のみデータを要求できます。ETag は、HTTP 標準の一部であり、Red Hat JBoss Data Grid によりサポートされます。

格納されたコンテンツのタイプは、返されたタイプです。例として、String が格納された場合は、文字列が返されます。シリアル化された形式で格納されたオブジェクトは、手動でデシリアル化する必要があります。

クエリーに **extended** パラメーターを追加すると、追加の情報が返されます。例を以下に示します。

```
GET /{cacheName}/{cacheKey}?extended
```

以下のカスタムヘッダーが返されます。

- **Cluster-Primary-Owner**: キーのプライマリーオーナーであるノードを特定します。
- **Cluster-Node-Name**: リクエストを処理したサーバーの JGroups ノード名を指定します。
- **Cluster-Physical-Address**: リクエストを処理したサーバーの物理 JGroups アドレスを指定します。

13.6.3.3. HEAD /{cacheName}/{cacheKey}

HEAD メソッドは、**GET** メソッドと同様に動作しますが、コンテンツを返しません (ヘッダーフィールドが返されます)。



注記

HEAD メソッドは、追加情報を返す **extended** パラメーターもサポートします。

13.6.3.4. GET /{cacheName}

GET メソッドは、キャッシュに存在するキーのリストを返すことができます。返されるキーのリストは応答のボディにあります。

Accept ヘッダーは次のように応答をフォーマットできます。

- **application/xml** はキーのリストを XML 形式で返します。
- **application/json** はキーのリストを JSON 形式で返します。
- **text/plain** はキーのリストを行ごとに 1 つのキーで示したプレーンテキストで返します。

キャッシュが分散されている場合、リクエストを処理するノードが所有するキーのみが返されます。すべてのキーを返すには、次のようにクエリーに **global** パラメータを追加します。

```
GET /{cacheName}?global
```

13.6.3.5. GET および HEAD メソッドのヘッダー

表13.3 GET および HEAD メソッドのヘッダー

Header	任意または必須	説明
--------	---------	----

Header	任意または必須	説明
Key-Content-Type	任意	URL のキーのコンテンツタイプを指定します。指定のない場合、デフォルトは application/x-java-object; type=java.lang.String になります。
Accept	任意	GET メソッドで呼び出しのコンテンツを返す形式を指定します。

13.6.4. REST API を介したデータの削除

13.6.4.1. キャッシュからのデータの削除

HTTP **DELETE** メソッドで Red Hat JBoss Data Grid からデータを削除します。

DELETE メソッドは以下を行うことができます。

- キャッシュエントリ/値を削除します (**DELETE /{cacheName}/{cacheKey}**)
- キャッシュからすべてのエントリを削除します (**DELETE /{cacheName}**)

13.6.4.2. **DELETE /{cacheName}/{cacheKey}**

DELETE メソッドがこのコンテキスト (**DELETE /{cacheName}/{cacheKey}**) で使用された場合、指定のキーのキャッシュからキー/値を削除します。

13.6.4.3. **DELETE /{cacheName}**

このコンテキスト (**DELETE /{cacheName}**) では、**DELETE** メソッドが名前付きキャッシュ内のすべてのエントリを削除します。正常な **DELETE** 操作後に、HTTP ステータスコード **200** が返されます。

13.6.4.4. バックグラウンド削除操作

performAsync ヘッダーの値を **true** に設定して、削除操作がバックグラウンドで続行される状態で値がすぐに返されるようにします。

13.6.5. ETag ベースのヘッダー

ETag ベースのヘッダー

各 REST インターフェースエントリに対して、提供された **URL** でデータの状態を示す **Last-Modified** ヘッダーとともに、ETag (エンティティータグ) が返されます。ETag は、帯域幅を節約するためにデータが変更された場合にのみ、データを要求する HTTP 操作で使用されます。以下のヘッダーは、ETag (エンティティータグ) ベースの楽観的ロックをサポートします。

表13.4 エンティティータグ関連ヘッダー

Header	アルゴリズム	例	説明
If-Match	If-Match = "If-Match" ":" ("*" 1#entity-tag)	-	(リソースから以前に取得された) 指定されたエンティティが最新であることを確認するために、関連するエンティティタグのリストとともに使用されます。
If-None-Match		-	(リソースから以前に取得された) 指定されたエンティティが最新でないことを確認するために、関連するエンティティタグのリストとともに使用されます。この機能により、必要なときに、最小のトランザクションオーバーヘッドで、キャッシュされた情報が効率的に更新されます。
If-Modified-Since	If-Modified-Since = "If-Modified-Since" ":" HTTP-date	If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT	要求されたバリエーションの最終変更日時と、提供された時間および日付の値とを比較します。指定された日時以降に要求されたバリエーションが変更されなかった場合は、エンティティの代わりに 304 (未変更) 応答がメッセージ本文なしで返されます。
If-Unmodified-Since	If-Unmodified-Since = "If-Unmodified-Since" ":" HTTP-date	If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT	要求されたバリエーションの最終変更日時と、提供された時間および日付の値とを比較します。指定された日時以降に要求されたリソースが変更されなかった場合は、指定された操作が実行されます。指定された日時以降に要求されたリソースが変更された場合は、操作が実行されず、エンティティの代わりに 412 (事前条件失敗) 応答が返されます。

13.6.6. REST インターフェースを介したデータのクエリー

Red Hat JBoss Data Grid では、Ickle クエリーを JSON 形式で使用して、REST インターフェースを介してデータをクエリーできます。



重要

JBoss Data Grid 7.2 では、REST インターフェースを介したデータのクエリーはテクノロジープレビューの機能となります。

13.6.6.1. JSON から Protostream への変換

JBoss Data Grid はプロトコルバッファを使用して、データを効率的にバイナリー形式でキャッシュに保存し、クエリーを公開してコンテンツを JSON 形式で読み書きできるようにします。

Protobuf でエンコードされたエントリーを保存するには、キャッシュを **application/x-protostream** メディアタイプで設定する必要があります。設定後、JBoss Data Grid は自動的に JSON を Protobuf に変換します。

キャッシュがインデックス化されている場合、設定を何も実行する必要はありません。デフォルトでは、インデックス化されたキャッシュはエントリーを **application/x-protostream** メディアタイプで保存します。

しかし、キャッシュがインデックス化されていない場合、以下の例のようにキーと値を **application/x-protostream** メディアタイプに設定する必要があります。

```
<cache>
  <encoding>
    <key media-type="application/x-protostream"/>
    <value media-type="application/x-protostream"/>
  </encoding>
</cache>
```

13.6.6.2. Protobuf スキーマの登録

Protobuf スキーマを登録するには、以下の例のように HTTP **POST** メソッドを使用して、スキーマを **__protobuf_metadata** キャッシュに挿入します。

```
curl -u user:password -X POST --data-binary @./schema.proto
http://127.0.0.1:8080/rest/__protobuf_metadata/schema.proto
```

Protobuf のエンコーディングと Protobuf スキーマの登録についての詳細は、「[Protobuf エンコーディング](#)」を参照してください。

13.6.6.3. JSON ドキュメントの Protobuf メッセージへのマッピング

_type フィールドが JSON ドキュメントに含まれるようにする必要があります。このフィールドは、JSON ドキュメントが対応する Protobuf メッセージを特定します。

たとえば、以下は **Person** として定義された Protobuf メッセージになります。

```
message Person {
  required string name = 1;
  required int32 age = 2;
}
```


以下は対応する JSON ドキュメントになります。

Person.json

```
{
  "_type": "Person",
  "name": "user1",
  "age": 32
}
```

13.6.6.4. キャッシュへの内容の追加

以下のように、JSON 形式でキャッシュにコンテンツを書き込むことができます。

```
curl -u user:user -XPOST --data-binary @./Person.json -H "Content-Type:
application/json; charset=UTF-8"
http://127.0.0.1:8080/rest/{cacheName}/{key}
```

- **{cacheName}** はクエリーを行うキャッシュの名前を指定します。
- **{key}** はデータをキャッシュに保存するキーの名前を指定します。

コンテンツをキャッシュに書き込んだ後、以下のように JSON 形式で読み取ることができます。

```
curl -u user:user http://127.0.0.1:8080/rest/{cacheName}/{key}
```

13.6.6.5. REST エンドポイントのクエリー

HTTP **GET** メソッドまたは **POST** メソッドを使用して、REST インターフェース経由でデータのクエリーを行います。

GET メソッドのクエリーリクエストの構造は次のようになります。

```
{cacheName}?action=search&query={ickle query}
```

- **{cacheName}** はクエリーを行うキャッシュの名前を指定します。
- **{ickle query}** は実行する lckle クエリーを指定します。

以下はクエリーの例になります。

- **Person** という名前のエンティティーからすべてのデータを返す:
http://localhost:8080/rest/mycache?action=search&query=from Person
- **select** 句でクエリーを絞り込む:**http://localhost:8080/rest/mycache?action=search&query=Select name, age from Person**
- クエリーの結果をグループ化する:**http://localhost:8080/rest/mycache?action=search&query=from Person group by age**

POST メソッドのクエリーリクエストの構造は次のようになります。

```
{cacheName}?action=search
```

リクエストのボディは、クエリーと JSON 形式のパラメーターを指定します。

以下は、**Entity** という名前のエンティティからデータを返し、**where** 句を使用して結果をフィルターするクエリーの例になります。

```
{
  "query": "from Entity where name:\\"user1\\\"",
  "max_results": 20,
  "offset": 10
}
```

13.6.6.5.1. 任意のリクエストパラメーター

以下の任意のパラメーターをクエリーリクエストに適用できます。

パラメーター	説明
max_results	クエリー結果の数をこの最大数に制限します。デフォルトの値は 10 です。
offset	返す最初の結果のインデックスを指定します。デフォルトの値は 0 です。
query_mode	<p>サーバーがどのようにクエリーを実行するかを、以下の値で指定します。</p> <p>BROADCAST: クラスターの各ノードにクエリーをブロードキャストし、クエリーの結果を取得および組み合わせた後に返します。この実行モードは、各ノードのインデックスにデータのサブセットが含まれる、非共有インデックスに適しています。</p> <p>FETCH: クエリーが呼び出すノードのクエリーを実行します。この実行モードは、クラスター全体でデータのインデックスすべてをローカルで利用できる場合に適しています。これはデフォルトの値です。</p>

13.6.6.5.2. クエリー結果

lckle クエリーの結果は、以下の例のように JSON形式で返されます。

```
{
  "total_results" : 150,
  "hits" : [ {
    "hit" : {
      "name" : "user1",
      "age" : 35
    }
  }, {
    "hit" : {
      "name" : "user2",
      "age" : 42
    }
  }
]
```

```
    }  
  }, {  
    "hit" : {  
      "name" : "user3",  
      "age" : 25  
    }  
  } ]  
}
```

- **total_results** はクエリーが返した結果の数になります。
- **hits** はクエリーと一致するすべての結果をリストします。
- **hit** にはクエリーの各結果のフィールドが含まれます。

lckle クエリーの詳細は、「[lckle クエリー言語を使用したクエリーの構築](#)」を参照してください。

第14章 クラスター化カウンター

クラスター化カウンターは、Red Hat JBoss Data Grid クラスターのノードすべてで分散および共有されます。クラスター化カウンターを使用するとオブジェクトの数を記録することができます。

クラスター化カウンターは名前で識別され、値 (デフォルトは 0) で初期化されます。クラスター化カウンターを永続化して、クラスターの再起動後に値を保持することもできます。

クラスター化カウンターには次の 2 種類があります。

- **Strong** はカウンターの値を単一のキーに保存して一貫性を保ちます。カウンターの更新中でも値は認識できます。カウンター値の更新は、キーロック下で実行されます。しかし、カウンターの現在値を読み取るのに必要なロックはありません。Strong カウンターでは、カウンター値をバインドでき、**compareAndSet** や **compareAndSwap** などのアトミック操作を提供します。
- **Weak** はカウンター値を複数のキーに格納します。各キーはカウンター値の部分的な状態を保存します。キーは同時に更新することが可能です。カウンターの更新中、値は認識できません。カウンター値を取得しても、常に最新の値を返すとは限りません。

Strong および Weak クラスター化カウンターの両方は、カウンター値の更新をサポートし、カウンターの現在の値を返し、カウンター値の更新時にイベントを提供します。

14.1. COUNTER API

counter API は以下で構成されます。

- **EmbeddedCounterManagerFactory** は埋め込みのキャッシュマネージャーからカウンターマネージャーを初期化します。
- **RemoteCounterManagerFactory** はリモートキャッシュマネージャーからカウンターマネージャーを初期化します。
- **CounterManager** は、カウンターの作成、定義および返却を行うメソッドを提供します。
- **StrongCounter** は strong カウンターを実装します。このインターフェースはカウンターのアトミックアップデートを提供します。操作はすべて非同期で実行され、完了ロジックに **CompletableFuture** クラスを使用します。
- **SyncStrongCounter** は同期の strong カウンターを実装します。
- **WeakCounter** は weak カウンターを実装します。すべての操作は非同期に実行され、完了ロジックに **CompletableFuture** クラスが使用されます。
- **SyncWeakCounter** は同期の weak カウンターを実装します。
- **CounterListener** は、strong カウンターへの変更をリスンします。
- **CounterEvent** は strong カウンターへの変更が発生したときにイベントを返します。
- **Handle** は **CounterListener** インターフェースを拡張します。

14.2. MAVEN 依存関係の追加

クラスター化カウンターの使用を開始するには、以下の依存関係を **pom.xml** に追加します。

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-counter</artifactId>
  <version>...</version> <!-- 7.2.0 or later -->
</dependency>
```

14.3. COUNTERMANAGER インターフェースの読み出し

Red Hat JBoss Data Grid 埋め込みモードでクラスター化カウンターを使用するには、以下を行います。

```
// Create or obtain an EmbeddedCacheManager.
EmbeddedCacheManager manager = ...;

// Retrieve the CounterManager interface.
CounterManager counterManager =
EmbeddedCounterManagerFactory.asCounterManager(manager);
```

Red Hat JBoss Data Grid リモートサーバーと対話する Hot Rod クライアントとクラスター化カウンターを使用するには、以下を行います。

```
// Create or obtain a RemoteCacheManager.
RemoteCacheManager manager = ...;

// Retrieve the CounterManager interface.
CounterManager counterManager =
RemoteCounterManagerFactory.asCounterManager(manager);
```

14.4. クラスター化カウンターの使用

クラスター化カウンターの定義および設定は、**cache-container** XML 設定またはプログラムで行います。

14.4.1. クラスター化カウンターの XML 設定

以下の XML スニペットは、クラスター化カウンターの設定例を表しています。

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan>
  <cache-container>
    <!-- cache container configuration goes here -->
    <!-- cache configuration goes here -->
    <counters>
      <strong-counter name="counter-1" initial-value="1">
        <upper-bound value="10"/>
      </strong-counter>
      <strong-counter name="counter-2" initial-value="2"/>
      <weak-counter name="counter-3" initial-value="3"/>
    </counters>
  </cache-container>
</infinispan>
```

14.4.1.1. XML 定義

counters 要素はクラスターのカウンターを設定し、以下の属性があります。

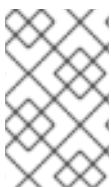
- **num-owners** は、クラスター全体で保存する各カウンターのコピー数を設定します。値が小さいと更新操作が速くなりますが、サポートするサーバーのクラッシュ数が少なくなります。値は正の整数である必要があります。デフォルトは **2** です。
- **reliability** はネットワークパーティションでのカウンター更新動作を設定し、以下の値を取ります。
 - **AVAILABLE**: すべてのパーティションがカウンターの値を読み取りおよび更新できます。これはデフォルト値です。
 - **CONSISTENT**: プライマリーパーティションがカウンターの値を読み取りおよび更新できます。残りのパーティションはカウンターの値の読み取りのみが可能です。

strong-counter 要素は、strong クラスター化カウンターを作成および定義します。**weak-counter** 要素は weak クラスター化カウンターを作成および定義します。以下は、両方の要素に共通する属性です。

- **initial-value** はカウンターの初期値を設定します。デフォルトの値は **0** です。
- **storage** はカウンター値の保存方法を設定します。この属性は、クラスターのシャットダウン後および再起動後にカウンター値が保存されるかどうかを決定します。この属性は以下の値を取ります。
 - **VOLATILE**: カウンターの値をメモリーに保存します。カウンターの値はクラスターのシャットダウン時に破棄されます。これがデフォルトの値になります。
 - **PERSISTENT**: カウンターの値はプライベートなローカル永続ストアに保存されます。カウンターの値は、クラスターのシャットダウン時および起動時に保存されます。

strong-counter 要素に固有する属性は次のとおりです。

- **lower-bound** は strong カウンターの下限を設定します。デフォルトの値は **Long.MIN_VALUE** です。
- **upper-bound** は strong カウンターの上限を設定します。デフォルトの値は **Long.MAX_VALUE** です。



注記

initial-value 属性の値は、**lower-bound** の値と **upper-bound** の値の間である必要があります。strong カウンターの上限と下限を指定しないと、カウンターはバインドされません。

weak-counter 要素に固有する属性は次のとおりです。

- **concurrency-level** はカウンターの値の最大同時更新数を設定します。値は正の整数である必要があります。デフォルトの値は **16** です。

14.4.2. クラスター化カウンターのランタイム設定

以下の例のように、**EmbeddedCacheManager** の初期化後、クラスター化カウンターを起動時にオンデマンドで設定することができます。

```
CounterManager manager = ...;

// Create three counters.
// The first counter is a strong counter bounded to 10.
manager.defineCounter("counter-1",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(1).upperBound(10).build());

// The second counter is an unbounded strong counter.
manager.defineCounter("counter-2",
CounterConfiguration.builder(CounterType.UNBOUNDED_STRONG).initialValue(2).build());

// The third counter is a weak counter.
manager.defineCounter("counter-3",
CounterConfiguration.builder(CounterType.WEAK).initialValue(3).build());
```

カウンターが正常に定義された場合、**defineCounter()** メソッドは **true** を返し、そうでない場合は **false** を返します。カウンターの設定が有効でない場合は **CounterConfigurationException** 例外が発生します。

ヒント

以下の例のように **isDefined()** メソッドを使用して、カウンターがすでに定義されているかどうかを判断します。

```
CounterManager manager = ...
if (!manager.isDefined("someCounter")) {
    manager.define("someCounter", ...);
}
```

14.4.3. クラスター化カウンターのプログラムによる設定

以下のコードサンプルは、**GlobalConfigurationBuilder** を使用してクラスター化カウンターをプログラミングで設定する方法を表しています。

```
// Set up a clustered cache manager.
GlobalConfigurationBuilder global =
GlobalConfigurationBuilder.defaultClusteredBuilder();

// Create a counter configuration builder.
CounterManagerConfigurationBuilder builder =
global.addModule(CounterManagerConfigurationBuilder.class);

// Create three counters.
// The first counter is a strong counter bounded to 10.
builder.addStrongCounter().name("counter-1").upperBound(10).initialValue(1);

// The second counter is an unbounded strong counter.
```

```
builder.addStrongCounter().name("counter-2").initialValue(2);

// The third counter is a weak counter.
builder.addWeakCounter().name("counter-3").initialValue(3);

// Initialize a new default cache manager.
DefaultCacheManager cacheManager = new
DefaultCacheManager(global.build());
```

14.4.3.1. クラスター化カウンターの使用

以下のコードサンプルは、プログラムで作成および定義したクラスター化カウンターを使用する方法を表しています。

```
// Retrieve the CounterManager interface from the cache manager.
CounterManager counterManager =
EmbeddedCounterManagerFactory.asCounterManager(cacheManager);

// Strong counters provide greater consistency than weak counters.
// The value of a strong counter is known during an increment or decrement
operation.
// The value of a strong counter can also be bounded in cases where a
limit is required.
StrongCounter counter1 = counterManager.getStrongCounter("counter-1");

// All methods are asynchronous and return CompletableFuture objects so
that you can perform other operations while the counter value is computed.
counter1.getValue().thenAccept(value -> System.out.println("Counter-1
initial value is " + value)).get();

// Attempt to add a value that exceeds the upper-bound value.
counter1.addAndGet(10).handle((value, throwable) -> {
    // Value is null since the counter is bounded to a maximum of 10.
    System.out.println("Counter-1 Exception is " +
throwable.getMessage());
    return 0;
}).get();

// Check the counter value. The value should be 10.
counter1.getValue().thenAccept(value -> System.out.println("Counter-1
value is " + value)).get();

//Decrement the counter value. The new value should be 9.
counter1.decrementAndGet().handle((value, throwable) -> {
    // No exception is thrown.
    System.out.println("Counter-1 new value is " + value);
    return value;
}).get();

// The second counter, counter2, is a strong counter that is unbounded. It
never throws the CounterOutOfBoundsException.
StrongCounter counter2 = counterManager.getStrongCounter("counter-2");

// All counters allow a listener to be registered.
// The handle interface can remove the listener.
```



```
counter2.addListener(event -> System.out
    .println("Counter-2 event: oldValue=" + event.getOldValue() + "
newValue=" + event.getNewValue()));

// Adding MAX_VALUE does not throw an exception.
// No increments take effect if the value exceeds the MAX_VALUE.
counter2.addAndGet(Long.MAX_VALUE).thenAccept(aLong ->
System.out.println("Counter-2 value is " + aLong)).get();

// Conditional operations are allowed in strong counters.
counter2.compareAndSet(Long.MAX_VALUE, 0)
    .thenAccept(aBoolean -> System.out.println("Counter-2 CAS result is "
+ aBoolean)).get();
counter2.getValue().thenAccept(value -> System.out.println("Counter-2
value is " + value)).get();

// Reset the value of the second counter to its initial value.
counter2.reset().get();
counter2.getValue().thenAccept(value -> System.out.println("Counter-2
initial value is " + value)).get();

// Retrieve the third counter, counter3.
WeakCounter counter3 = counterManager.getWeakCounter("counter-3");

// The value of weak counters is not available during update operations.
// As a result these counters can increment faster than strong counters.
// The counter value is computed lazily and stored locally.
counter3.add(5).thenAccept(aVoid -> System.out.println("Adding 5 to
counter-3 completed!")).get();

// Check the counter value.
System.out.println("Counter-3 value is " + counter3.getValue());

// Stop the cache manager and release all resources.
cacheManager.stop();
```

第15章 クラスター化ロック

クラスター化ロックは、Red Hat JBoss Data Grid クラスターのノードすべてで分散および共有されるデータ構造です。クラスター化ロックを使用すると、クラスターのノード間で同期されるコードを実行できます。

15.1. LOCK API

lock API は以下で構成されます。

- **EmbeddedClusteredLockManagerFactory** は、埋め込みキャッシュマネージャーからクラスター化ロックマネージャーを初期化します。
- **ClusteredLockManager** は、クラスター化ロックを定義、設定、取得、および削除するメソッドを提供します。
- **ClusteredLock** はクラスター化ロックを実装するメソッドを提供します。



注記

クラスター化ロックは、Red Hat JBoss Data Grid 埋め込みモードでのみ利用可能です。

15.2. サポートされる設定

本リリースより、Red Hat JBoss Data Grid は **NODE** オーナーシップと再入可能でないロックをサポートするようになりました。

NODE オーナーシップを使用すると、Red Hat JBoss Data Grid クラスターのすべてのノードがロックを使用できます。

再入可能ロックの場合、そのロックを所有するノードはロックを所有する限りロックを再取得することができます。再入可能でないロックの場合、すべてのノードがロックを取得できます。そのため、2つの連続したロック呼び出しが同じ所有者に送信されると、最初の呼び出しがロックを取得し (ロックがある場合)、2番目の呼び出しはブロックされます。

15.3. MAVEN 依存関係の追加

クラスター化ロックの使用を開始するには、以下の依存関係を **pom.xml** に追加します。

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-lock</artifactId>
  <version>...</version> <!-- 7.2.0 or later -->
</dependency>
```

15.4. クラスター化ロックの使用

以下のコードサンプルは、クラスター化ロックの使用方法を表しています。

```
// Set up a clustered cache manager.
```

```

GlobalConfigurationBuilder global =
GlobalConfigurationBuilder.defaultClusteredBuilder();

// Configure the cache mode as distributed and synchronous.
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC);

// Initialize a new default cache manager.
DefaultCacheManager cm = new DefaultCacheManager(global.build(),
builder.build());

// Initialize a clustered lock manager from the cache manager.
ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);

// Define a clustered lock named 'lock' with the default configuration.
clm1.defineLock("lock");

// Get a lock from each node in the cluster.
ClusteredLock lock = clm1.get("lock");

AtomicInteger counter = new AtomicInteger(0);

// Acquire the lock as follows.
// Each 'lock.tryLock(1, TimeUnit.SECONDS)' method attempts to acquire
the lock.
// If the lock is not available, the method waits for the timeout period
to elapse. When the lock is acquired, other calls to acquire the lock are
blocked until the lock is released.
CompletableFuture<Boolean> call1 = lock.tryLock(1,
TimeUnit.SECONDS).whenComplete((r, ex) -> {
    if (r) {
        System.out.println("lock is acquired by the call 1");
        lock.unlock().whenComplete((nil, ex2) -> {
            System.out.println("lock is released by the call 1");
            counter.incrementAndGet();
        });
    }
});

CompletableFuture<Boolean> call2 = lock.tryLock(1,
TimeUnit.SECONDS).whenComplete((r, ex) -> {
    if (r) {
        System.out.println("lock is acquired by the call 2");
        lock.unlock().whenComplete((nil, ex2) -> {
            System.out.println("lock is released by the call 2");
            counter.incrementAndGet();
        });
    }
});

CompletableFuture<Boolean> call3 = lock.tryLock(1,
TimeUnit.SECONDS).whenComplete((r, ex) -> {
    if (r) {
        System.out.println("lock is acquired by the call 3");
        lock.unlock().whenComplete((nil, ex2) -> {
            System.out.println("lock is released by the call 3");

```

```
        counter.incrementAndGet();
    });
}
});

CompletableFuture.allOf(call1, call2, call3).whenComplete((r, ex) -> {
    // Print the value of the counter.
    System.out.println("Value of the counter is " + counter.get());

    // Stop the cache manager.
    cm.stop();
});
```

第16章 HOT ROD インターフェース

16.1. HOT ROD について

Hot Rod は、Red Hat JBoss Data Grid で使用されるバイナリー TCP クライアントサーバープロトコルであり、Memcached などの他のクライアントサーバープロトコルの欠点を解消するために作成されました。

Hot Rod はサーバークラスターでフェイルオーバーを行い、トポロジが変更されます。Hot Rod は、クラスターポロジに関する更新をクライアントに定期的に提供することによりこれを行います。

Hot Rod では、クライアントはパーティション化されたまたは分散された Red Hat JBoss Data Grid サーバークラスターで要求をスマートにルーティングできます。これを行うために、Hot Rod ではクライアントはキーを格納するパーティションを決定し、キーがあるサーバーと直接通信できます。この機能は、クライアントでクラスターポロジを更新する Hot Rod に依存し、クライアントはサーバーと同じ一貫性のあるハッシュアルゴリズムを使用します。

Red Hat JBoss Data Grid には、Hot Rod プロトコルを実装するサーバーモジュールが含まれます。Hot Rod プロトコルを使用すると、他のテキストベースのプロトコルに比べて、クライアントとサーバーの対話が高速になり、クライアントが負荷分散、フェイルオーバー、およびデータ場所運用に関する決定を行えるようになります。

16.2. HOT ROD ヘッダー

16.2.1. Hot Rod ヘッダーデータタイプ

Red Hat JBoss Data Grid の Hot Rod で使用されたすべてのキーおよび値はバイトアレイとして格納されます。ただし、特定のヘッダー値 (REST や Memcached の値) は以下のデータタイプを使用して格納されます。

表16.1 ヘッダーデータタイプ

データタイプ	Size	説明
vInt	1 - 5 バイト。	符号なし可変長整数値。
vLong	1-9 バイト。	符号なし可変長 long 値。
string	-	文字列は常に UTF-8 エンコーディングを使用して表されます。

16.2.2. 要求ヘッダー

Hot Rod を使用して Red Hat JBoss Data Grid にアクセスする場合、要求ヘッダーの内容は以下のものから構成されます。

表16.2 要求ヘッダーフィールド

フィールド名	データタイプ/サイズ	説明
--------	------------	----

フィールド名	データタイプ/サイズ	説明
Magic	1 バイト	ヘッダーが要求ヘッダーまたは応答ヘッダーであるかどうかを示します。
Message ID	vLong	メッセージ ID を含みます。この一意の ID は、要求に応答するときに使用されます。これにより、Hot Rod クライアントは非同期でプロトコルを実装できるようになります。
Version	1 バイト	Hot Rod サーバーバージョンを含みます。
Opcode	1 バイト	関連する操作コードを含みます。要求ヘッダー内でopcode には要求操作コードのみを含めることができます。
Cache Name Length	vInt	キャッシュ名の長さを格納します。キャッシュ名の長さが 0 に設定され、キャッシュ名に値が提供されない場合、操作はデフォルトのキャッシュと対話します。
Cache Name	string	指定された操作のターゲットキャッシュの名前を格納します。この名前は、キャッシュ設定ファイルの事前定義済みキャッシュの名前に一致する必要があります。
Flags	vInt	システムに渡されるフラグを表す可変長の数値を含みます。さらに多くのバイトを読み取る必要があるかどうかを決定するために使用される最大ビットを除き、各ビットはフラグを表します。各フラグを表すためにビットを使用すると、フラグの組み合わせが連結された状態で表されます。
Client Intelligence	1 バイト	サーバーに対するクライアント機能を示す値を含みます。

フィールド名	データタイプ/サイズ	説明
Topology ID	vInt	クライアントの最後の既知なビュー ID を含みます。基本的なクライアントはこのフィールドに値 0 を提供します。トポロジーまたはハッシュ情報をサポートするクライアントは、サーバーが現在のビュー ID に応答するまで値 0 (新しいビュー ID が現在のビュー ID を置き換えるためにサーバーにより返されるまで使用されます) を提供します。

16.2.3. 応答ヘッダー

Hot Rod を使用して Red Hat JBoss Data Grid にアクセスする場合、応答ヘッダーの内容は以下のものから構成されます。

表16.3 応答ヘッダーフィールド

フィールド名	データタイプ	説明
Magic	1 バイト	ヘッダーが要求または応答ヘッダーであるかどうかを示します。
Message ID	vLong	メッセージ ID を含みます。この一意の ID は、応答を元の要求とペアにするために使用されます。これにより、Hot Rod クライアントは非同期でプロトコルを実装できるようになります。
Opcode	1 バイト	関連する操作コードを含みます。応答ヘッダー内で opcode には応答操作コードのみを含めることができます。
Status	1 バイト	応答のステータスを表すコードを含みます。
Topology Change Marker	1 バイト	応答がトポロジー変更情報に含まれるかどうかを示すマーカーバイトを含みます。

16.2.4. トポロジー変更ヘッダー

16.2.4.1. トポロジー変更ヘッダー

Hot Rod を使用して Red Hat JBoss Data Grid にアクセスする場合は、応答ヘッダーが、異なるトポロジーまたはハッシュ分散を区別できるクライアントを探すことによりクラスターまたはビューフォー

メーシヨンの変更に応答します。Hot Rod サーバーは現在の **topology ID** と、クライアントにより送信された **topology ID** を比較し、2つの値が異なる場合は、新しい **topology ID** を返します。

16.2.4.2. トポロジー変更マーカ値

以下は、応答ヘッダー内の **Topology Change Marker** フィールドの有効な値のリストです。

表16.4 Topology Change Marker フィールド値

Value	説明
0	トポロジーの変更情報は追加されません。
1	トポロジーの変更情報が追加されます。

16.2.4.3. トポロジー認識クライアントのトポロジー変更ヘッダー

トポロジーの変更がサーバーにより返された場合、トポロジー認識クライアントに送信された応答ヘッダーには以下の要素が含まれます。

表16.5 トポロジー変更ヘッダーフィールド

応答ヘッダーフィールド	データタイプ/サイズ	説明
Response Header with Topology Change Marker	変数	「 応答ヘッダー 」を参照してください。
Topology ID	vInt	トポロジー ID。
Num Servers in Topology	vInt	クラスターで稼働している Hot Rod サーバーの数を含みます。一部のノードのみが Hot Rod サーバーを稼働している場合に、この値は、クラスター全体のサブセットになることがあります。
mX: Host/IP Length	vInt	個別クラスターメンバーのホスト名または IP アドレスの長さを含みます。可変長により、この要素にはホスト名、IPv4、および IPv6 アドレスを含めることができます。
mX: Host/IP Address	string	個別クラスターメンバーのホスト名または IP アドレスを含みます。Hot Rod クライアントはこの情報を使用して個別クラスターメンバーにアクセスします。

応答ヘッダーフィールド	データタイプ/サイズ	説明
mX: Port	符号なしショート。2 バイト	クラスターメンバーと通信するために Hot Rod クライアントが使用するポートを含みます。

トポロジ内の各サーバーに対して、接頭辞 **mX** の 3 つのエントリが繰り返されます。トポロジの情報フィールド内の最初のサーバーには接頭辞 **m1** が付けられ、**X** の値が **num servers in topology** フィールドで指定されたサーバーの数と等しくなるまで、各追加サーバーに対して数値が 1 つずつ増分されます。

16.2.4.4. ハッシュ配布認識クライアントのトポロジ変更ヘッダー

トポロジの変更がサーバーにより返された場合、クライアントに送信された応答ヘッダーには以下の要素が含まれます。

表16.6 トポロジ変更ヘッダーフィールド

フィールド	データタイプ/サイズ	説明
Response Header with Topology Change Marker	変数	「 応答ヘッダー 」を参照してください。
Topology ID	vInt	トポロジ ID。
Number Key Owners	符号なしショート、2 バイト	配布された各キーに対してグローバルに設定されたコピーの数を含みます。配布がキャッシュで設定されていない場合は、値 0 を含みます。
Hash Function Version	1 バイト	使用中のハッシュ機能へのポインターを含みます。配布がキャッシュで設定されていない場合は、値 0 を含みます。
Hash Space Size	vInt	ハッシュコード生成に関連するすべてのモジュール計算のために JBoss Data Grid により使用されるモジュールを含みます。クライアントはこの情報を使用して正しいハッシュ計算をキーに適用します。配布がキャッシュで設定されていない場合は、値 0 を含みます。

フィールド	データタイプ/サイズ	説明
Number servers in topology	vInt	クラスター内で稼働している [path]_ Hot Rod_ サーバーの数を含みます。一部のノードのみが [path]_ Hot Rod_ サーバーを稼働している場合に、この値は、クラスター全体のサブセットになることがあります。また、この値はヘッダーに含まれるホストとポートのペアの数を表します。
Number Virtual Nodes Owners	vInt	設定された仮想ノードの数を含みます。仮想ノードが設定されていない場合、または配布がキャッシュで設定されていない場合は、値 0 を含みます。
mX: Host/IP Length	vInt	個別クラスターメンバーのホスト名または [path]_ IP_ アドレスの長さを含みます。可変長により、この要素にはホスト名、[path]_ IPv4_ および [path]_ IPv6_ アドレスを含めることができます。
mX: Host/IP Address	string	個別クラスターメンバーのホスト名または [path]_ IP_ アドレスを含みます。[path]_ Hot Rod_ クライアントはこの情報を使用して個別クラスターメンバーにアクセスします。
mX: Port	符号なしショート、2 バイト	クラスターメンバーと通信するために [path]_ Hot Rod_ クライアントが使用するポートを含みます。
Hash Function Version	1 バイト	0x03
Number of Segments in Topology	vInt	トポロジーの合計セグメント数。
Number of Owners in Segment	1 バイト	所有者の数は 0、1、または 2 のいずれか。
First Wwner's Index	vInt	全ノードのリストにおけるこの所有者の位置。このセグメントの所有者の数が 1 または 2 である場合のみ含まれます。

フィールド	データタイプ/サイズ	説明
Second Owner's Index	vInt	全ノードのリストにおけるこの所有者の位置。このセグメントの所有者の数が2である場合のみ含まれます。



注記

セグメントごとに所有者の数を3以上にすることもできますが、Hot Rod プロトコルは効率性の理由で送信する所有者の数を制限します。

トポロジー内の各サーバーに対して、接頭辞 **mX** の3つのエントリーが繰り返されます。トポロジーの情報フィールド内の最初のサーバーには接頭辞 **m1** が付けられ、**X** の値が **num servers in topology** フィールドで指定されたサーバーの数と等しくなるまで、各追加サーバーに対して数値が1ずつ増分されます。

16.3. HOT ROD 操作

16.3.1. Hot Rod 操作

以下は、Hot Rod プロトコル 1.3 を使用して Red Hat JBoss Data Grid と対話する場合に有効な操作です。

- Authenticate
- AuthMechList
- BulkGet
- BulkKeysGet
- Clear
- ContainsKey
- Exec
- Get
- GetAll
- GetWithMetadata
- GetWithVersion
- IterationEnd
- IterationNext
- IterationStart
- Ping

- Put
- PutAll
- PutIfAbsent
- Query
- Remove
- RemoveIfUnmodified
- Replace
- ReplaceIfUnmodified
- Stats
- Size



重要

RemoteCache API を使用して Hot Rod クライアントの **Put** 操作、**PutIfAbsent** 操作、**Replace** 操作、および **ReplaceWithVersion** 操作を呼び出す場合に、ライフスパンが 30 日より大きい値に設定されると、その値は UNIX 時間として処理され、1970 年 1 月 1 日以降の秒数を表します。

16.3.2. Hot Rod の **Authenticate** 操作

この操作の目的は、SASL を使用してクライアントをサーバーに対して認証することです。選択した mech によっては、認証プロセスが複数のステップの操作になることがあります。この操作が完了すると、接続が認証されます。

Authenticate 操作の要求形式には以下が含まれます。

表16.7 **Authenticate** 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Mech	String	認証用にクライアントによって選択された mech の名前が含まれる String。後続の呼び出しでは空になります。
Response length	vInt	SASL クライアント応答の長さ
Response data	バイトアレイ	SASL クライアント応答。

この操作の応答ヘッダーには以下のものが含まれます。

表16.8 **Authenticate** 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー。
Completed	バイト	さらに処理が必要な場合は 0、認証が完了した場合は 1。
Challenge length	vInt	SASL サーバーチャレンジの長さ
Challenge data	バイトアレイ	SASL サーバーチャレンジ。

16.3.3. Hot Rod の AuthMechList 操作

この操作の目的は、サーバーによってサポートされる 有効な SASL 認証 mech のリストを取得することです。クライアントは優先される mech で **Authenticate** 要求を発行する必要があります。

AuthMechList 操作の要求形式には以下が含まれます。

表16.9 AuthMechList 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー

この操作の応答ヘッダーには以下のものが含まれます。

表16.10 AuthMechList 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー
Mech count	vInt	mech の数。
Mech	String	IANA で登録された形式 (GSSAPI、CRAM-MD5 など) で SASL mech の名前が含まれる String

Mech の値はサポートされる mech ごとに繰り返されます。

16.3.4. Hot Rod の BulkGet 操作

Hot Rod の **BulkGet** 操作は、以下の要求形式を使用します。

表16.11 BulkGet 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Entry Count	vInt	サーバーによって返される Red Hat JBoss Data Grid エントリーの最大数が含まれます。エントリーはキーと値のペアです。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表16.12 BulkGet 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー
More	vInt	ストリームからエントリーをさらに読み取る必要があるかどうかを示します。 More は 1 に設定される一方で、More の値が 0 (ストリームの最後を示します) に設定されるまで追加のエントリーが続きます。
Key Length	vInt	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	値を含みます。

要求された各エントリーに対して、**More**、**Key Size**、**Key**、**Value Size**、および **Value** エントリーが応答に追加されます。

16.3.5. Hot Rod の BulkKeysGet 操作

Hot Rod の **BulkKeysGet** 操作は、以下の要求形式を使用します。

表16.13 BulkKeysGet 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー

フィールド	データタイプ	説明
Scope	vInt	<ul style="list-style-type: none"> ● 0 = デフォルトのスコープ - このスコープは RemoteCache.keySet() メソッドにより使用されます。リモートキャッシュが分散キャッシュである場合は、サーバーによりマップ/削減操作が実行され、すべてのノードからすべてのキーが取得されます (トポロジー認識 Hot Rod クライアントはクラスター内の任意のノードへの要求を負荷分散できます)。それ以外の場合は、要求を受信するサーバーに対してローカルのキャッシュインスタンスからキーを取得します。キーはレプリケートされたキャッシュのすべてのノードで同じである必要があります。 ● 1 = グローバルスコープ - このスコープはデフォルトスコープと同じように動作します。 ● 2 = ローカルスコープ - リモートキャッシュが分散キャッシュである状況では、サーバーはマップ/削減操作を開始してすべてのノードからキーを取得しません。代わりに、要求を受け取るサーバーに対してローカルなキャッシュインスタンスからローカルなキーのみを取得します。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表16.14 BulkKeysGet 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー。
Response Status	1 バイト	0x00 = 成功 (データが続きます)。

フィールド	データタイプ	説明
More	1 バイト	ストリームからより多くのキーを読み取る必要があるかを表す 1 つのバイト。 1 に設定されている場合はエントリーが続き、 0 に設定されている場合はストリームの最後であり、読み取るエントリーが残っていません。
Key Length	vInt	キーの長さ
Key	バイトアレイ	取得されたキー
More	1 バイト	ストリームからより多くのエントリーを読み取る必要があるかどうかを表す 1 つのバイト。1 に設定された場合はエントリーが続きます。0 に設定された場合はストリームの最後であり、読み取るエントリーが残っていません。

Key Length および **Key** の値はキーごとに繰り返されます。

16.3.6. Hot Rod の Clear 操作

clear 操作の形式にはヘッダーのみが含まれます。

この操作に有効な応答ステータスは以下のとおりです。

表16.15 clear 操作の応答

Response Status	説明
0x00	Red Hat JBoss Data Grid が正常に消去されました。

16.3.7. Hot Rod の ContainsKey 操作

Hot Rod の **ContainsKey** 操作は、以下の要求形式を使用します。

表16.16 ContainsKey 操作の要求形式

フィールド	データタイプ	説明
Header	-	-

フィールド	データタイプ	説明
Key Length	vInt	キーの長さを含みます。 Integer.MAX_VALUE のサイズよりも大きいサイズ (最大 5 バイト) であるため、 vInt データタイプが使用されます。ただし、Java では単一のアレイサイズを Integer.MAX_VALUE よりも大きくすることはできません。結果として、この vInt も Integer.MAX_VALUE の最大サイズに制限されます。
Key	バイトアレイ	キーを含みます (このキーの対応する値が要求されます)。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表16.17 ContainsKey 操作の応答形式

Response Status	説明
0x00	操作が成功。
0x02	キーが存在しない。

この操作の応答は空白です。

16.3.8. Hot Rod の Exec 操作

Exec 操作の要求形式には以下が含まれます。

表16.18 Exec 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Script	String	実行するスクリプトの名前。
Parameter Count	vInt	パラメーターの数。
Parameter Name (per parameter)	String	パラメーターの名前。
Parameter Length (per parameter)	vInt	パラメーターの長さ。
Parameter Value (per parameter)	バイトアレイ	パラメーターの値。

この操作の応答ヘッダーには以下のものが含まれます。

表16.19 Exec 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー。
Response status	1 バイト	正常に実行が完了した場合は 0x00、サーバーにエラーが発生した場合は 0x85。
Value Length	vInt	成功した場合は、戻り値の長さ。
Value	バイトアレイ	成功した場合は、実行の結果。

16.3.9. Hot Rod の Get 操作

Hot Rod の **Get** 操作は、以下の要求形式を使用します。

表16.20 Get 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Key Length	vInt	キーの長さを含みます。 Integer.MAX_VALUE のサイズよりも大きいサイズ (最大 5 バイト) であるため、 vInt データタイプが使用されます。ただし、Java では単一のアレイサイズを Integer.MAX_VALUE よりも大きくすることはできません。結果として、この vInt も Integer.MAX_VALUE の最大サイズに制限されます。
Key	バイトアレイ	キーを含みます (このキーの対応する値が要求されます)。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表16.21 Get 操作の応答形式

Response Status	説明
0x00	操作が成功。
0x02	キーが存在しない。

キーが見つかった場合の **get** 操作の応答形式は以下のとおりです。

表16.22 Get 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

16.3.10. Hot Rod の GetAll 操作

指定のキーセットへマップするすべてのエントリーを取得する一括操作。

Hot Rod の **GetAll** 操作は、以下の要求形式を使用します。

表16.23 GetAll 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Key Count	vInt	エンティティーを見つけるためのキーの数。
Key Length	vInt	キーの長さ。
Key	バイトアレイ	取得されたキー

Key Length および **Key** の値はキーごとに繰り返されます。

この操作の応答ヘッダーには以下のものが含まれます。

表16.24 GetAll 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー
Entry count	vInt	返されたエントリーの数。
Key Length	vInt	キーの長さ。
Key	バイトアレイ	取得されたキー
Value Length	vInt	値の長さ。

フィールド	データタイプ	説明
Value	バイトアレイ	取得された値。

Key Length、**Key**、**Value Length**、および **Value** エントリはキーおよび値ごとに繰り返されます。

16.3.11. Hot Rod の **GetWithMetadata** 操作

Hot Rod の **GetWithMetadata** 操作は、以下の要求形式を使用します。

表16.25 **GetWithMetadata** 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Key Length	vInt	キーの長さ。vInt のサイズは最大 5 バイトであり、理論的には Integer.MAX_VALUE よりも大きい数を生成できます。ただし、Java では Integer.MAX_VALUE よりも大きい単一アレイを作成できず、プロトコルにより vInt アレイの長さが Integer.MAX_VALUE に制限されます。
Key	バイトアレイ	値が要求されるキーを含むバイトアレイ。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表16.26 **GetWithMetadata** 操作の応答要求

フィールド	データタイプ	説明
Header	変数	応答ヘッダー。
Response status	1 バイト	0x00 = 成功 (キーが取得された場合)。 0x02 = キーが存在しない場合。

フィールド	データタイプ	説明
Flag	1 バイト	<p>応答に期限切れに関する情報が含まれるかどうかを示すフラグ。フラグの値は、 INFINITE_LIFESPAN (0x01) と INFINITE_MAXIDLE (0x02) 間のビットごとの OR 演算として取得されます。</p>
Created	Long	<p>(任意) サーバーでエントリーが作成されたときのタイムスタンプを表す Long。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合にのみ返されます。</p>
Lifespan	vInt	<p>(任意) 秒単位でエントリーのライフスパンを表す vInt。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合にのみ返されます。</p>
LastUsed	Long	<p>(任意) サーバーでエントリーが最後にアクセスされたときのタイムスタンプを表す Long。この値は、フラグの INFINITE_MAXIDLE ビットが設定されていない場合にのみ返されます。</p>
MaxIdle	vInt	<p>(任意) 秒単位でエントリーの maxIdle を表す vInt。この値は、フラグの INFINITE_MAXIDLE ビットが設定されていない場合にのみ返されます。</p>
Entry Version	8 バイト	<p>既存のエントリー変更の一意な値。プロトコルでは entry_version の値はシーケンシャルであることが保証されず、キーレベルで更新ごとに一意である必要があります。</p>
Value Length	vInt	<p>成功した場合は、値の長さ。</p>
Value	バイトアレイ	<p>成功した場合は、要求された値。</p>

16.3.12. Hot Rod の GetWithVersion 操作

Hot Rod の **GetWithVersion** 操作は、以下の要求形式を使用します。

表16.27 GetWithVersion 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Key Length	vInt	キーの長さを含みます。 Integer.MAX_VALUE のサイズよりも大きいサイズ (最大 5 バイト) であるため、 vInt データタイプが使用されます。ただし、Java では単一のアレイサイズを Integer.MAX_VALUE よりも大きくすることはできません。結果として、この vInt も Integer.MAX_VALUE の最大サイズに制限されます。
Key	バイトアレイ	キーを含みます (このキーの対応する値が要求されます)。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表16.28 GetWithVersion 操作の応答形式

Response Status	説明
0x00	操作が成功。
0x02	キーが存在しない。

キーが見つかった場合の **GetWithVersion** 操作の応答形式は以下のとおりです。

表16.29 GetWithVersion 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー
Entry Version	8 バイト	既存のエントリー変更の一意な値。プロトコルでは entry_version の値はシーケンシャルであることが保証されません。キーレベルで更新ごとに一意である必要があります。

フィールド	データタイプ	説明
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

16.3.13. Hot Rod の IterationEnd 操作

IterationEnd 操作の要求形式には以下が含まれます。

表16.30 IterationEnd 操作の要求形式

フィールド	データタイプ	説明
iterationId	String	イテレーションの一意な ID。

以下は、この操作から返される有効な応答値です。

表16.31 IterationEnd 操作の応答形式

Response Status	説明
0x00	操作が成功。
0x05	存在しない iterationId に対するエラー。

16.3.14. Hot Rod の IterationNext 操作

IterationNext 操作の要求形式には以下が含まれます。

表16.32 IterationNext 操作の要求形式

フィールド	データタイプ	説明
IterationId	String	イテレーションの一意な ID。

この操作の応答ヘッダーには以下のものが含まれます。

表16.33 IterationNext 操作の応答形式

フィールド	データタイプ	説明
Finished segments size	vInt	イテレーションを終了したセグメントを表すビットセットのサイズ。

フィールド	データタイプ	説明
Finished segments	バイトアレイ	イテレーションを終了したセグメントのビットセットエンコーディング。
Entry count	vInt	返されたエントリーの数。
Number of value projections	vInt	値の射影の数。
Metadata	1 バイト	設定された場合、エントリーにメタデータが関連付けされます。
Expiration	1 バイト	応答に期限切れに関する情報が含まれるかどうかを示すフラグ。フラグの値は、 INFINITE_LIFESPAN (0x01) と INFINITE_MAXIDLE (0x02) 間のビットごとの OR 演算として取得されます。上記の Metadata フラグが設定されている場合のみ含まれます。
Created	Long	(任意) サーバーでエントリーが作成されたときのタイムスタンプを表す Long。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合にのみ返されます。
Lifespan	vInt	(任意) 秒単位でエントリーのライフスパンを表す vInt。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合にのみ返されます。
LastUsed	Long	(任意) サーバーでエントリーが最後にアクセスされたときのタイムスタンプを表す Long。この値は、フラグの INFINITE_MAXIDLE ビットが設定されていない場合にのみ返されます。

フィールド	データタイプ	説明
MaxIdle	vInt	(任意) 秒単位でエントリーの maxIdle を表す vInt。この値は、フラグの INFINITE_MAXIDLE ビットが設定されていない場合にのみ返されます。
Entry Version	8 バイト	既存のエントリー変更の一意な値。Metadata フラグが設定されている場合のみ含まれます。
Key Length	vInt	キーの長さ。
Key	バイトアレイ	取得されたキー
Value Length	vInt	値の長さ。
Value	バイトアレイ	取得された値。

エントリーごとに **Metadata**、**Expiration**、**Created**、**Lifespan**、**LastUsed**、**MaxIdle**、**Entry Version**、**Key Length**、**Key**、**Value Length**、および **Value** フィールドが繰り返されます。

16.3.15. Hot Rod の IterationStart 操作

IterationStart 操作の要求形式には以下が含まれます。

表16.34 IterationStart 操作の要求形式

フィールド	データタイプ	説明
Segments size	signed vInt	イテレーションを行うセグメント ID のビットセットエンコーディングのサイズ。サイズは、最も近い 8 の倍数に切り上げまたは切り捨てされた最大セグメント ID になります。-1 の値はセグメントのフィルターが実行されないことを示します。

フィールド	データタイプ	説明
Segments	バイトアレイ	<p>(任意) ビットセットがエンコードされた セグメント ID が含まれ、1 が値の各ビットはセットのセグメントを表します。バイトの順番はリトルエンディアン (little endian) です。たとえば、セグメント [1,3,12,13] は以下のエンコーディングになります。</p> <pre> 00001010 00110000 size: 16 bits first byte: represents segments from 0 to 7, from which 1 and 3 are set second byte: represents segments from 8 to 15, from which 12 and 13 are set </pre> <p>詳細は java.util.BitSet 実装を参照してください。前のフィールドが負の値でない場合、セグメントは送信されます。</p>
FilterConverter size	signed vInt	<p>サーバーにデプロイされた KeyValueFilterConverter ファクトリー名を表す String のサイズ、またはフィルターが使用されない場合は -1。</p>
FilterConverter	UTF-8 バイトアレイ	<p>(任意) サーバーにデプロイされた KeyValueFilterConverter ファクトリー名。前のフィールドが負の値でない場合に含まれます。</p>
Parameters size	バイト	<p>フィルターのパラメーター数。 FilterConverter が提供される場合のみ含まれます。</p>
Parameters	byte[][]	<p>パラメーターのアレイ。各パラメーターがバイトアレイになります。 Parameters サイズが 0 よりも大きい場合のみ含まれます。</p>

フィールド	データタイプ	説明
BatchSize	vInt	一度にサーバーから転送するエントリーの数。
Metadata	1 バイト	各エントリーに対してメタデータを返す場合は 1。その他の場合は 0。

この操作の応答ヘッダーには以下のものが含まれます。

表16.35 IterationEnd 操作の応答形式

フィールド	データタイプ	説明
IterationId	String	イテレーションの一意な ID。

16.3.16. Hot Rod の Ping 操作

ping は、サーバーの可用性を確認するアプリケーションレベルの要求です。

この操作に有効な応答ステータスは以下のとおりです。

表16.36 ping 操作の応答

Response Status	説明
0x00	エラーなしの正常な ping。

16.3.17. Hot Rod の Put 操作

put 操作の要求形式には以下が含まれます。

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Key Length	-	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。

フィールド	データタイプ	説明
TimeUnits	バイト	<p>lifespan (最初の 4 ビット) および maxIdle (最後の 4 ビット) の時間単位。デフォルトのサーバー期限切れには DEFAULT、期限切れのない場合は INFINITE を使用できます。可能な値は次のとおりです。</p> <pre> 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANOSECONDS 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE </pre>
Lifespan	vInt	エントリーが存続できる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。
Max Idle	vInt	各エントリーがキャッシュからエビクトされる前に、アイドル状態でいられる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値。

以下は、この操作から返される有効な応答値です。

Response Status	説明
0x00	値が正常に格納されました。
0x03	値が正常に格納され、以前の値が続きます。

この操作では空の応答がデフォルト応答になります。ただし、**ForceReturnPreviousValue** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **0** が含まれます。

16.3.18. Hot Rod の PutAll 操作

すべてのキーバリュエントリーを同時にキャッシュに置く一括操作。

PutAll 操作の要求形式には以下が含まれます。

表16.37 PutAll 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
TimeUnits	バイト	<p>lifespan (最初の 4 ビット) および maxIdle (最後の 4 ビット) の時間単位。デフォルトのサーバー期限切れには DEFAULT、期限切れのない場合は INFINITE を使用できます。可能な値は次のとおりです。</p> <pre> 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANoseconds 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE </pre>
Lifespan	vInt	エントリーが存続できる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。
Max Idle	vInt	各エントリーがキャッシュからエビクトされる前に、アイドル状態でいられる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。
Entry count	vInt	挿入されたエントリーの数。
Key Length	vInt	キーの長さ。
Key	バイトアレイ	取得されたキー
Value Length	vInt	値の長さ。
Value	バイトアレイ	取得された値。

Key Length、**Key**、**Value Length**、および **Value** フィールドは、置かれたエントリーごとに繰り返されます。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表16.38 PutAll 操作の応答形式

Response Status	説明
0x00	操作に成功し、すべてのキーが正常に置かれたことを示します。

16.3.19. Hot Rod の PutIfAbsent 操作

PutIfAbsent 操作の要求形式には以下が含まれます。

表16.39 PutIfAbsent 操作の要求フィールド

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Key Length	vInt	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。
TimeUnits	バイト	<p>lifespan (最初の 4 ビット) および maxIdle (最後の 4 ビット) の時間単位。デフォルトのサーバー期限切れには DEFAULT、期限切れのない場合は INFINITE を使用できます。可能な値は次のとおりです。</p> <pre> 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANoseconds 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE </pre>
Lifespan	vInt	エントリーが存続できる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。
Max Idle	vInt	各エントリーがキャッシュからエビクトされる前に、アイドル状態でいられる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。

フィールド	データタイプ	説明
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

以下は、この操作から返される有効な応答値です。

Response Status	説明
0x00	値が正常に格納されました。
0x01	キーが存在したため値は保存されませんでした。 キーの現在の値が返されます。
0x04	キーが存在したため操作に失敗し、応答でその値が続きます。

この操作では空の応答がデフォルト応答になります。ただし、**ForceReturnPreviousValue** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **0** が含まれます。

16.3.20. Hot Rod の Query 操作

Query 操作の要求形式には以下が含まれます。

表16.40 Query 操作の要求フィールド

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Query Length	vInt	Protobuf エンコードされたクエリーオブジェクトの長さ。
Query	バイトアレイ	Protobuf エンコードされたクエリーオブジェクトを含むバイトアレイ。長さは前のフィールドにより指定されます。

以下は、この操作から返される有効な応答値です。

表16.41 Query 操作の応答

Response Status	データ	説明
Header	変数	応答ヘッダー。

Response Status	データ	説明
Response payload Length	vInt	Protobuf エンコードされた応答オブジェクトの長さ。
Response payload	バイトアレイ	Protobuf エンコードされた応答オブジェクトを含むバイトアレイ。長さは前のフィールドにより指定されます。

Hot Rod の **Query** 操作の要求および応答タイプは、**infinispan-remote-query-client.jar** 内にある **org/infinispan/query/remote/client/query.proto** リソースファイルで定義されます。

16.3.21. Hot Rod の Remove 操作

Hot Rod の **Remove** 操作は、以下の要求形式を使用します。

表16.42 Remove 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Key Length	vInt	キーの長さを含みます。 Integer.MAX_VALUE のサイズよりも大きいサイズ (最大 5 バイト) であるため、 vInt データタイプが使用されます。ただし、Java では単一のアレイサイズを Integer.MAX_VALUE よりも大きくすることはできません。結果として、この vInt も Integer.MAX_VALUE の最大サイズに制限されます。
Key	バイトアレイ	キーを含みます (このキーの対応する値が要求されます)。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表16.43 Remove 操作の応答形式

Response Status	説明
0x00	操作が成功。
0x02	キーが存在しない。

Response Status	説明
0x03	キーが削除され、応答で以前の値または削除された値が続きます。

通常、この操作の応答ヘッダーは空白です。ただし、**ForceReturnPreviousValue** が渡された場合は、応答ヘッダーに以下のいずれかが含まれます。

- 以前のキーの値および長さ。
- キーが存在しないことを示す、値の長さ **0** と応答ステータス **0x02**。

remove 操作の応答ヘッダーには、提供されたキーの以前の値と、以前の値の長さが含まれます (**ForceReturnPreviousValue** が渡された場合)。キーが存在しない場合、または以前の値が null の場合、値の長さは **0** です。

16.3.22. Hot Rod の RemoveIfUnmodified 操作

RemoveIfUnmodified 操作の要求形式には以下が含まれます。

表16.44 **RemoveIfUnmodified** 操作の要求フィールド

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Key Length	vInt	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。
Entry Version	8 バイト	エントリーのバージョン番号。

以下は、この操作から返される有効な応答値です。

表16.45 **RemoveIfUnmodified** 操作の応答

Response Status	説明
0x00	エントリーは置換または削除されました。
0x01	キーが変更されたため、エントリーの置換または削除に失敗しました。
0x02	キーが存在しない。
0x03	キーが削除され、応答で以前の値または置き換えられた値が続きます。

Response Status	説明
0x04	キーが変更されたためエントリーの削除に失敗し、応答で変更された値が続きます。

この操作では空の応答がデフォルト応答になります。ただし、**ForceReturnPreviousValue** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **0** が含まれます。

16.3.23. Hot Rod の Replace 操作

replace 操作の要求形式には以下が含まれます。

表16.46 Replace 操作の要求フィールド

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Key Length	vInt	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。
TimeUnits	バイト	<p>lifespan (最初の 4 ビット) および maxIdle (最後の 4 ビット) の時間単位。デフォルトのサーバー期限切れには DEFAULT、期限切れのない場合は INFINITE を使用できます。可能な値は次のとおりです。</p> <pre> 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANoseconds 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE </pre>
Lifespan	vInt	エントリーが存続できる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。

フィールド	データタイプ	説明
Max Idle	vInt	各エントリーがキャッシュからエビクトされる前に、アイドル状態でいられる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

以下は、この操作から返される有効な応答値です。

表16.47 Replace 操作の応答

Response Status	説明
0x00	値が正常に格納されました。
0x01	キーが存在しないため、値が格納されませんでした。
0x03	値が正常に置換され、応答で以前の値または置き換えられた値が続きます。

この操作では空の応答がデフォルト応答になります。ただし、**ForceReturnPreviousValue** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **0** が含まれます。

16.3.24. Hot Rod の ReplaceIfUnmodified 操作

ReplaceIfUnmodified 操作の要求形式には以下が含まれます。

表16.48 ReplaceIfUnmodified 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー

フィールド	データタイプ	説明
Key Length	vInt	キーの長さ。vInt のサイズは最大 5 バイトであり、理論的には Integer.MAX_VALUE よりも大きい数を生成できます。ただし、Java では Integer.MAX_VALUE よりも大きい単一アレイを作成できず、プロトコルにより vInt アレイの長さが Integer.MAX_VALUE に制限されます。
Key	バイトアレイ	値が要求されるキーを含むバイトアレイ。
TimeUnits	バイト	<p>lifespan (最初の 4 ビット) および maxIdle (最後の 4 ビット) の時間単位。デフォルトのサーバー期限切れには DEFAULT、期限切れのない場合は INFINITE を使用できます。可能な値は次のとおりです。</p> <pre> 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANoseconds 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE </pre>
Lifespan	vInt	エントリーが存続できる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。
Max Idle	vInt	各エントリーがキャッシュからエビクトされる前に、アイドル状態でいられる期間。TimeUnits が DEFAULT または INFINITE でない場合のみ送信されます。
Entry Version	8 バイト	GetWithVersion 操作により返された値を使用します。
Value Length	vInt	値の長さ。
Value	バイトアレイ	格納する値。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表16.49 **ReplaceIfUnmodified** 操作の応答ステータス

Response Status	説明
0x00	値が正常に格納されました。
0x01	キーが変更されたため、置換が実行されませんでした。
0x02	キーが存在しないため、置換が実行されませんでした。
0x03	キーは置換され、応答で以前の値または置き換えられた値が続きます。
0x04	キーが変更されたためエントリーの置き換えに失敗し、応答で変更された値が続きます。

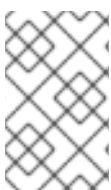
以下は、この操作から返される有効な応答値です。

表16.50 **ReplaceIfUnmodified** 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー。
Previous value length	vInt	以前の値を強制的に戻すフラグが要求で送信された場合、以前の値の長さが返されます。キーが存在しない場合、値の長さは 0 になります。送信されたフラグがない場合、値の長さは含まれません。
Previous value	バイトアレイ	以前の値を強制的に戻すフラグが要求で送信され、キーが置き換えられた場合、前の値になります。

16.3.25. Hot Rod の **ReplaceWithVersion** 操作

ReplaceWithVersion 操作の要求形式には以下が含まれます。



注記

RemoteCache API では、Hot Rod の **ReplaceWithVersion** 操作は **ReplaceIfUnmodified** 操作を使用します。結果として、これらの 2 つの操作は JBoss Data Grid ではまったく同じになります。

表16.51 **ReplaceWithVersion** 操作の要求フィールド

フィールド	データタイプ	説明
Header	-	-
Key Length	vInt	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。
Lifespan	vInt	エントリーが期限切れになるまでの秒数を含みます。秒数が 30 日を超える場合、その値はエントリーライフスパンの UNIX 時間 (つまり、日付 1/1/1970 以降の秒数) として処理されます。値が 0 に設定された場合、エントリーは期限切れになりません。
Max Idle	vInt	キャッシュからエビクトされるまでエントリーがアイドル状態のままになることが許可される秒数を含みます。このエントリーが 0 に設定された場合、エントリーは無期限でアイドル状態のままになることが許可され、max idle 値のため、エビクトされません。
Entry Version	8 バイト	エントリーのバージョン番号。
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

以下は、この操作から返される有効な応答値です。

表16.52 ReplaceWithVersion 操作の応答

Response Status	説明
0x00	エントリーが置換または削除された場合に返されたステータス。
0x01	キーが変更されたため、エントリーの置換または削除が失敗した場合に、ステータスを返します。
0x02	キーが存在しない場合に、ステータスを返します。

この操作では空の応答がデフォルト応答になります。ただし、**ForceReturnPreviousValue** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **0** が含まれます。

16.3.26. Hot Rod の Stats 操作

この操作は、利用可能なすべての統計の概要を返します。返された各統計に対して、名前と値が文字列形式と UTF-8 形式の両方で返されます。

この操作では、以下の統計がサポートされます。

表16.53 Stats 操作の要求フィールド

Name	説明
timeSinceStart	Hot Rod が起動した以降の秒数を含みます。
currentNumberOfEntries	Hot Rod サーバーに現在存在するエントリーの数を含みます。
totalNumberOfEntries	Hot Rod サーバーに格納されたエントリーの合計数を含みます。
stores	put 操作の試行回数を含みます。
retrievals	get 操作の試行回数を含みます。
hits	get ヒット数を含みます。
misses	get 失敗数を含みます。
removeHits	remove ヒット数を含みます。
removeMisses	removal 失敗数を含みます。
globalCurrentNumberOfEntries	Hot Rod クラスター全体における現在のエントリー数。
globalStores	Hot Rod クラスター全体における put 操作の合計数
globalRetrievals	Hot Rod クラスター全体における get 操作の合計数
globalHits	Hot Rod クラスター全体における get ヒット数の合計。
globalMisses	Hot Rod クラスター全体における get 失敗数の合計。
globalRemoveHits	Hot Rod クラスター全体における removal ヒット数の合計。
globalRemoveMisses	Hot Rod クラスター全体における removal 失敗数の合計。



注記

Hot Rod がローカルモードで実行されている場合、**global** で始まる統計は利用できません。

この操作の応答ヘッダーには以下のものが含まれます。

表16.54 Stats 操作の応答

Name	データタイプ	説明
Header	変数	応答ヘッダー。
Number of Stats	vInt	返された個別統計の数を含みます。
Name Length	vInt	名前付き統計の長さを含みます。
Name	string	統計の名前を含みます。
Value Length	vInt	値の長さを含みます。
Value	string	統計値を含みます。

要求された各統計に対して、値 **Name Length**、**Name**、**Value Length**、および **Value** が繰り返されます。

16.3.27. Hot Rod の Size 操作

Size 操作の要求形式には以下が含まれます。

表16.55 Size 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー

この操作の応答ヘッダーには以下のものが含まれます。

表16.56 Size 操作の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー。

フィールド	データタイプ	説明
Size	vInt	クラスター化された構成でグローバルに計算されるリモートキャッシュのサイズ。存在する場合はキャッシュストアの内容も考慮されます。

16.4. HOT ROD 操作の値

16.4.1. Hot Rod 操作の値

以下は、要求ヘッダーと対応する応答ヘッダー値の有効な **opcode** 値のリストです。

表16.57 opcode 要求および応答ヘッダー値

操作	要求操作コード	応答操作コード
put	0x01	0x02
get	0x03	0x04
putIfAbsent	0x05	0x06
replace	0x07	0x08
replaceIfUnmodified	0x09	0x0A
remove	0x0B	0x0C
removeIfUnmodified	0x0D	0x0E
containsKey	0x0F	0x10
clear	0x13	0x14
stats	0x15	0x16
ping	0x17	0x18
bulkGet	0x19	0x1A
getWithMetadata	0x1B	0x1C
bulkKeysGet	0x1D	0x1E
query	0x1F	0x20

操作	要求操作コード	応答操作コード
authMechList	0x21	0x22
auth	0x23	0x24
addClientListener	0x25	0x26
removeClientListener	0x27	0x28
size	0x29	0x2A
exec	0x2B	0x2C
putAll	0x2D	0x2E
getAll	0x2F	0x30
iterationStart	0x31	0x32
iterationNext	0x33	0x34
iterationEnd	0x35	0x36

また、応答ヘッダーの **opcode** 値が **0x50** の場合は、エラー応答を示します。

16.4.2. Magic 値

以下は要求および応答ヘッダー内の **Magic** フィールドの有効な値のリストです。

表16.58 Magic フィールド値

Value	説明
0xA0	キャッシュ要求マーカー。
0xA1	キャッシュ応答マーカー。

16.4.3. Status 値

以下は、応答ヘッダー内の **Status** フィールドに有効なすべての値を含む表です。

表16.59 Status 値

Value	説明
0x00	エラーなし。
0x01	配置、削除、置換なし。
0x02	キーは存在しません。
0x06	成功ステータス、互換性モードが有効です。
0x07	互換性モードの成功ステータスおよび前の値の返信が有効です。
0x08	互換性モードの非実行および前の値の返信が有効です。
0x81	無効なマジック値またはメッセージ ID。
0x82	不明なコマンド。
0x83	不明なバージョン。
0x84	要求解析エラー。
0x85	サーバーエラー。
0x86	コマンドタイムアウト。

16.4.4. Client Intelligence 値

以下は、要求ヘッダー内の **Client Intelligence** の有効な値のリストです。

表16.60 Client Intelligence フィールド値

Value	説明
0x01	クラスターまたはハッシュ情報が必要でない基本的なクライアントを示します。
0x02	トポロジを認識し、クラスター情報が必要なクラスターを示します。
0x03	ハッシュと配布を認識し、クラスターおよびハッシュ情報が必要なクライアントを示します。

16.4.5. フラグ値

以下は、要求ヘッダー内の有効な **flag** 値のリストです。

表16.61 フラグフィールド値

Value	説明
0x0001	ForceReturnPreviousValue

16.4.6. Hot Rod のエラー処理

表16.62 応答ヘッダーフィールドを使用した Hot Rod エラー処理

フィールド	データタイプ	説明
Error Opcode	-	エラー操作コードを含みます。
Error Status Number	-	error opcode に対応するステータス番号を含みます。
Error Message Length	vInt	エラーメッセージの長さを含みます。
Error Message	string	実際のエラーメッセージを含みます。要求の解析エラーが存在したことを示す 0x84 エラーコードが返された場合、このフィールドには、[path]_ Hot Rod_ サーバーでサポートされた最新バージョンが含まれます。

16.5. HOT ROD のリモートイベント

16.5.1. Hot Rod のリモートイベント

クライアントはリモートイベントリスナーを登録して、サーバーで発生するイベントの更新を受信することができます。クライアントリスナーが追加された直後にイベントが生成および送信されるため、リスナーの追加後に発生したすべてのイベントを受け取ることができます。

16.5.2. Hot Rod におけるリモートイベントのクライアントリスナーの追加

リモートイベントのクライアントリスナーを追加するには、以下の要求形式を使用します。

表16.63 Add Client Listener 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Listener ID	バイトアレイ	リスナーの識別子。

フィールド	データタイプ	説明
Include state	バイト	このバイトが 1 に設定された場合、初めてキャッシュリスナーを追加するとき、またはクラスター化環境でリモートリスナーが登録されたノードに変更があったときに、キャッシュされた状態がリモートクライアントへ返信されます。有効にすると、状態はクライアントへイベントを作成したキャッシュエントリとして返信されます。0 に設定された場合、リスナーの追加時に状態はクライアントへ返信されず、リスナーが登録されたノードに変更があっても状態を受信しません。
Key/value filter factory name	String	このリスナーと使用されるキーバリューストリックファクトリーの任意の名前。Hot Rod サーバーで直接イベントがフィルターされるようにし、クライアントに関係ないイベントが送信されないようにするキーバリューストリックインスタンスを作成するために、このファクトリーが使用されます。使用されるファクトリーがない場合、文字列の長さは 0 になります。
Key/value filter factory parameter count	バイト	フィルターインスタンスの作成時、キーバリューストリックファクトリーに任意の数のパラメータを使用できるため、このファクトリーを使用して異なるフィルターインスタンスを動的に作成できます。このカウントフィールドは、ファクトリーへ渡されるパラメータの数を示します。ファクトリー名を指定しないと、このフィールドはリクエストに含まれません。
Key/value filter factory parameter (per parameter)	バイトアレイ	キーバリューストリックファクトリーのパラメータ。

フィールド	データタイプ	説明
Converter factory name	String	このリスナーと使用するコンバーターファクトリーの任意の名前。このファクトリーは、クライアントへ送信されるイベントの内容を変換するために使用されます。使用されるコンバーターがないと、。デフォルトでは生成されたイベントタイプに応じてイベントが明確に定義されます。しかし、イベントに情報を追加する必要がある場合や、イベントのサイズを縮小する必要がある場合があります。このような場合、コンバーターを使用してイベントの内容を変換することができます。コンバーターファクトリー名を指定すると、この処理を行うコンバーターインスタンスが作成されます。ファクトリーが使用されない場合は、文字列の長さが 0 になります。
Converter factory parameter count	バイト	コンバーターインスタンスの作成時、コンバーターファクトリーに任意の数のパラメーターを使用できるため、このファクトリーを使用して異なるコンバーターインスタンスを動的に作成できます。このカウントフィールドは、ファクトリーへ渡されるパラメーターの数を示します。ファクトリー名を指定しないと、このフィールドはリクエストに含まれません。
Converter factory parameter (per parameter)	バイトアレイ	コンバーターファクトリーのパラメーター。
Use raw data	バイト	フィルターまたはコンバーターパラメーターを raw バイナリーにする必要がある場合は 1、その他の場合は 0。

操作の応答形式は次のとおりです。

表16.64 Add Client Listener の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー。

16.5.3. リモートイベントの **Hot Rod** リモートクライアントリスナー

以前追加したクライアントリスナーを削除するには、以下の要求形式を使用します。

表16.65 Remove Client Listener 操作の要求形式

フィールド	データタイプ	説明
Header	変数	要求ヘッダー
Listener ID	バイトアレイ	リスナーの識別子。

操作の応答形式は次のとおりです。

表16.66 Add Client Listener の応答形式

フィールド	データタイプ	説明
Header	変数	応答ヘッダー。

16.5.4. Hot Rod イベントヘッダー

各リモートイベントは以下の形式に準拠するヘッダーを使用します。

表16.67 リモートイベントヘッダー

フィールド名	Size	Value
Magic	1 バイト	0xA1 = response
Message ID	vLong	イベントの ID
Opcode	1 バイト	イベントタイプに응答するコード。 <div> 0x60 = cache entry created event 0x61 = cache entry modified event 0x62 = cache entry removed event 0x50 = error </div>
Status	1 バイト	응答の状態。可能な値は次のとおりです。 <div> 0x00 = No error </div>

フィールド名	Size	Value
Topology Change Marker	1 バイト	新しいトポロジーの送信が必要であるかどうかを決定できるようにするため、イベントは特定の受信トポロジー ID に関連付けられていません。そのため、新しいトポロジーはイベントとともに送信されません。よって、このマーカはイベントに対して常に 0 の値を持ちます。

16.5.5. Hot Rod の CacheEntryCreated イベント

CacheEntryCreated イベントには以下が含まれます。

表16.68 CacheEntryCreated イベント

フィールド名	Size	Value
Header	変数	0x60 操作コードが含まれるイベントヘッダー。
Listener ID	バイトアレイ	このイベントが転送されるリスナー。
Custom Marker	バイト	カスタムイベントマーカ。作成されたイベントは 0 になります。
Command Retried	バイト	再試行されたコマンドによって発生したイベントのマーカ。コマンドが再試行された場合は 1、その他の場合は 0。
Key	バイトアレイ	作成されたキー。
Version	long	作成されたエントリーのバージョン。このバージョン情報を使用して、このキャッシュエントリーで条件付き操作を作成できます。

16.5.6. Hot Rod の CacheEntryModified イベント

CacheEntryModified イベントには以下が含まれます。

表16.69 CacheEntryModified イベント

フィールド名	Size	Value
Header	変数	0x61 操作コードが含まれるイベントヘッダー。
Listener ID	バイトアレイ	このイベントが転送されるリスナー。
Custom Marker	バイト	カスタムイベントマーカ。作成されたイベントは 0 になります。
Command Retried	バイト	再試行されたコマンドによって発生したイベントのマーカ。コマンドが再試行された場合は 1、その他の場合は 0。
Key	バイトアレイ	変更されたキー。
Version	long	変更されたエントリーのバージョン。このバージョン情報を使用して、このキャッシュエントリーで条件付き操作を作成できます。

16.5.7. Hot Rod の **CacheEntryRemoved** イベント

CacheEntryRemoved イベントには以下が含まれます。

表16.70 **CacheEntryRemoved** イベント

フィールド名	Size	Value
Header	変数	0x62 操作コードが含まれるイベントヘッダー。
Listener ID	バイトアレイ	このイベントが転送されるリスナー。
Custom Marker	バイト	カスタムイベントマーカ。作成されたイベントは 0 になります。
Command Retried	バイト	再試行されたコマンドによって発生したイベントのマーカ。コマンドが再試行された場合は 1、その他の場合は 0。
Key	バイトアレイ	削除されたキー。

16.5.8. Hot Rod の **Custom** イベント

Custom イベントには以下が含まれます。

表16.71 Custom イベント

フィールド名	Size	Value
Header	変数	イベント固有の操作コードが含まれるイベントヘッダー
Listener ID	バイトアレイ	このイベントが転送されるリスナー。
Custom Marker	バイト	カスタムイベントマーカ。ユーザーに返す前にイベントデータをアンマーシャルする必要があるカスタムイベントの場合、値は 1 になります。イベントデータをそのままユーザーに返す必要があるカスタムイベントの場合、値は 2 になります。
Event Data	バイトアレイ	カスタムイベントデータ。カスタムマーカが 1 の場合、バイトはコンバーターによって返されたインスタンスのマーシャルされたバージョンを表します。カスタムマーカが 2 の場合、コンバーターによって返されたバイトアレイを表します。

16.6. PUT 要求の例

以下は、Hot Rod を使用した **put** 要求例からのコーディングされた要求です。

表16.72 Put 要求の例

バイト	0	1	2	3	4	5	6	7
8	0xA0	0x09	0x41	0x01	0x07	0x4D ('M')	0x79 ('y')	0x43 ('C')
16	0x61 ('a')	0x63 ('c')	0x68 ('h')	0x65 ('e')	0x00	0x03	0x00	0x00
24	0x00	0x05	0x48 ('H')	0x65 ('e')	0x6C ('l')	0x6C ('l')	0x6F ('o')	0x00
32	0x00	0x05	0x57 ('W')	0x6F ('o')	0x72 ('r')	0x6C ('l')	0x64 ('d')	-

以下の表には、要求の例に対するすべてのヘッダーフィールドと値が含まれます。

表16.73 要求例のフィールド名と値

フィールド名	バイト	Value
Magic	0	0xA0
Version	2	0x41
Cache Name Length	4	0x07
Flag	12	0x00
Topology ID	14	0x00
Transaction ID	16	0x00
Key	18-22	'Hello'
Max Idle	24	0x00
Value	26-30	'World'
Message ID	1	0x09
Opcode	3	0x01
Cache Name	5-11	'MyCache'
Client Intelligence	13	0x03
Transaction Type	15	0x00
Key Field Length	17	0x05
Lifespan	23	0x00
Value Field Length	25	0x05

以下は、**put** 要求の例に対するコーディングされた応答です。

表16.74 put 要求の例のコーディングされた応答

バイト	0	1	2	3	4	5	6	7
8	0xA1	0x09	0x01	0x00	0x00	-	-	-

以下の表には、応答の例のヘッダーフィールドと値がすべて含まれます。

表16.75 応答例のフィールド名および値

フィールド名	バイト	Value
Magic	0	0xA1
Opcode	2	0x01
Topology Change Marker	4	0x00
Message ID	1	0x09
Status	3	0x00

16.7. HOT ROD JAVA クライアント

16.7.1. Hot Rod Java クライアント

Hot Rod はバイナリーの言語非依存プロトコルです。Java クライアントは、Hot Rod Java Client API を使用して Hot Rod プロトコルを介してサーバーと対話できます。

16.7.2. Hot Rod Java クライアントのダウンロード

JBoss Data Grid Hot Rod Java クライアントをダウンロードするには、次の手順に従ってください。

手順: Hot Rod Java クライアントのダウンロード

1. <https://access.redhat.com> のカスタマーポータルにログインします。
2. ページの上部にある **ダウンロード** ボタンをクリックします。
3. **製品のダウンロード** ページで **Red Hat JBoss Data Grid** をクリックします。
4. **Version:** ドロップダウンメニューで適切な JBoss Data Grid のバージョンを選択します。
5. **Red Hat JBoss Data Grid 7.2 Hot Rod Java Client** エントリーを見つけ、その **Download** リンクをクリックします。

16.7.3. Hot Rod Java クライアントの設定

Hot Rod Java クライアントは、プログラムを使用したり、設定ファイルまたはプロパティファイルを外部的に使用したりして設定されます。次の例は、利用可能な Java 対応 API を使用したクライアントインスタンスの作成を示しています。

クライアントインスタンスの作成

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
= new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.tcpNoDelay(true)
  .connectionPool()
    .numTestsPerEvictionRun(3)
    .testOnBorrow(false)
```

```

        .testOnReturn(false)
        .testWhileIdle(true)
    .addServer()
        .host("localhost")
        .port(11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());

```

プロパティファイルを使用した Hot Rod Java クライアントの設定

Hot Rod Java クライアントを設定するには、クラスパス上の **hotrod-client.properties** ファイルを編集します。

次の例は、**hotrod-client.properties** ファイルの内容を示しています。

設定

```

infinispan.client.hotrod.transport_factory =
org.infinispan.client.hotrod.impl.transport.tcp.TcpTransportFactory

infinispan.client.hotrod.server_list = 127.0.0.1:11222

infinispan.client.hotrod.marshaller =
org.infinispan.commons.marshall.jboss.GenericJBossMarshaller

infinispan.client.hotrod.async_executor_factory =
org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory

infinispan.client.hotrod.default_executor_factory.pool_size = 1

infinispan.client.hotrod.default_executor_factory.queue_size = 10000

infinispan.client.hotrod.hash_function_impl.1 =
org.infinispan.client.hotrod.impl.consistenthash.ConsistentHashV1

infinispan.client.hotrod.tcp_no_delay = true

infinispan.client.hotrod.ping_on_startup = true

infinispan.client.hotrod.request_balancing_strategy =
org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy

infinispan.client.hotrod.key_size_estimate = 64

infinispan.client.hotrod.value_size_estimate = 512

infinispan.client.hotrod.force_return_values = false

infinispan.client.hotrod.tcp_keep_alive = true

## below is connection pooling config

maxActive=-1

maxTotal = -1

```

```

maxIdle = -1

whenExhaustedAction = 1

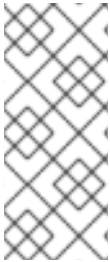
timeBetweenEvictionRunsMillis=120000

minEvictableIdleTimeMillis=300000

testWhileIdle = true

minIdle = 1

```



注記

TCPKEEPALIVE 設定は、例で示された設定プロパティー (`infinispan.client.hotrod.tcp_keep_alive = true/false`) または `org.infinispan.client.hotrod.ConfigurationBuilder.tcpKeepAlive()` メソッドを使用したプログラムによって Hot Rod Java クライアントで有効または無効になります。

Red Hat JBoss Data Grid でプロパティーファイルを使用するには、次の 2 つのコンストラクターのいずれかを使用する必要があります。

1. `new RemoteCacheManager(boolean start)`
2. `new RemoteCacheManager()`

16.7.4. Hot Rod Java クライアントベーシック API

以下のコードは、クライアント API を使用して Hot Rod Java クライアントで Hot Rod サーバーから情報を保存または取得する方法を示しています。この例では、Hot Rod サーバーがデフォルトの場所 `localhost:11222` にバインドするよう起動されていることを前提とします。

ベーシック API

```

//API entry point, by default it connects to localhost:11222
BasicCacheContainer cacheContainer = new RemoteCacheManager();
//obtain a handle to the remote default cache
BasicCache<String, String> cache = cacheContainer.getCache();
//now add something to the cache and ensure it is there
cache.put("car", "ferrari");
assert cache.get("car").equals("ferrari");
//remove the data
cache.remove("car");
assert !cache.containsKey("car") : "Value must have been
removed!";

```

RemoteCacheManager は **DefaultCacheManager** に対応し、両方が **BasicCacheContainer** を実装します。

この API は、Hot Rod を介したローカルコールからリモートコールへの移行を実現します。これは、**DefaultCacheManager** と **RemoteCacheManager** を切り替えることによって行うことができ、共通の **BasicCacheContainer** インターフェースによって単純化されます。

すべてのキーは、**keySet()** メソッドを使用してリモートキャッシュから取得できます。リモートキャッシュが分散キャッシュである場合は、サーバーにより Map/Reduce ジョブが開始され、クラスター化されたノードからすべてのキーが取得され、すべてのキーがクライアントに返されます。

キーの数が多い場合は、このメソッドを注意して使用してください。

```
Set keys = remoteCache.keySet();
```

16.7.5. Hot Rod Java クライアントバージョン API

データの整合性を確保するために、Hot Rod は各変更を一意に識別するバージョン番号を保存します。**getVersioned** を使用して、クライアントはキーと現在のバージョンに関連付けられた値を取得できます。

Hot Rod Java クライアントを使用する場合、**RemoteCacheManager** は、リモートクラスター上の名前付きキャッシュまたはデフォルトのキャッシュにアクセスする **RemoteCache** インターフェースのインスタンスを提供します。これにより、バージョン API を含む、新しいメソッドを追加する **Cache** インターフェースが拡張されます。

バージョンメソッドの使用

```
// To use the versioned API, remote classes are specifically needed
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
RemoteCache<String, String> remoteCache = remoteCacheManager.getCache();
remoteCache.put("car", "ferrari");
VersionedValue valueBinary = remoteCache.getWithMetadata("car");
// removal only takes place only if the version has not been changed
// in between. (a new version is associated with 'car' key on each change)
assert remoteCache.removeWithVersion("car", valueBinary.getVersion());
assert !remoteCache.containsKey("car");
```

置換の使用

```
remoteCache.put("car", "ferrari");
VersionedValue valueBinary = remoteCache.getWithMetadata("car");
assert remoteCache.replaceWithVersion("car", "lamborghini",
valueBinary.getVersion());
```

16.7.6. Hot Rod Java クライアントでのクラスター全体の動的キャッシュ作成

クライアントからキャッシュを動的に作成する必要がある場合は、次のように **createCache()** メソッドを使用します。

```
BasicCache<String, String> cache =
client(0).administration().createCache("newCacheName", "newTemplate");
```

このように作成されたキャッシュはクラスターのすべてのノードで利用可能になりますが、それは一時的です。クラスター全体をシャットダウンし、再起動しても、キャッシュは自動的に再作成されません。キャッシュを永続化するには、**PERMANENT** フラグを以下のように使用します。

```
BasicCache<String, String> cache =
client(0).administration().withFlags(AdminFlag.PERMANENT).createCache("new
CacheName", "newTemplate");
```

■

上記は、グローバル状態を有効にし、適切な設定ストレージを選択しないと動作しません。選択可能な設定ストレージは以下のとおりです。

- **VOLATILE**: 名前が意味するとおり、この設定ストレージは **PERMANENT** キャッシュをサポートしません。
- **OVERLAY**: **cache.xml** という名前のファイルのグローバルな共有状態永続パスに設定を保存します。
- **MANAGED**: サーバーデプロイメントでのみサポートされ、**PERMANENT** キャッシュをサーバーモデルに保存します。
- **CUSTOM**: カスタムの設定ストア。

16.8. HOT ROD C++ クライアント

16.8.1. Hot Rod C++ クライアント

Hot Rod C++ クライアントを使用すると、C++ ランタイムアプリケーションによる Red Hat JBoss Data Grid リモートサーバーへの接続や対話が可能になり、リモートキャッシュヘータを読み書きできます。Hot Rod C++ クライアントは 3 つのレベルすべてでクライアントインテリジェンスをサポートし、以下のプラットフォームでサポートされます。

- Red Hat Enterprise Linux 6、64 ビット
- Red Hat Enterprise Linux 7、64 ビット

Hot Rod C++ クライアントは、Visual Studio 2015 がインストールされた 64 ビット Windows ではテクノロジープレビューとして利用できます。

16.8.2. Hot Rod C++ クライアント形式

Hot Rod C++ クライアントは、以下の 2 つのライブラリー形式で利用可能です。

- 静的ライブラリー
- 共有/動的ライブラリー

静的ライブラリー

静的ライブラリーはアプリケーションに静的にリンクされます。これにより、最終的な実行可能ファイルのサイズは増加します。アプリケーションは自己完結型であり、別のライブラリーを提供する必要はありません。

共有/動的ライブラリー

共有/動的ライブラリーは、実行時にアプリケーションに動的にリンクされます。ライブラリーは別のファイルに格納され、アプリケーションを再コンパイルせずアプリケーションとは別にアップグレードできます。



注記

これは、ライブラリーのメジャーバージョンがコンパイル時にアプリケーションがリンクされたものと同じである (バイナリー互換性がある) 場合にのみ可能です。

16.8.3. Hot Rod C++ クライアントの前提条件

以下の表は、基盤の OS に応じた Hot Rod C++ クライアントの使用要件の詳細を示しています。

表16.76 Hot Rod C++ クライアントの OS 別の前提条件

オペレーティングシステム	Hot Rod C++ クライアントの前提条件
RHEL 6、64 ビット	shared_ptr TR1 (GCC 4.0+) をサポートする C++ 03 コンパイラー
RHEL 7、64 ビット	C++ 11 コンパイラー (GCC 4.8.1)
Windows 7 x64	C 11 コンパイラー (Visual Studio 2015、x64 プラットフォーム用 Microsoft Visual C 2013 再頒布可能パッケージ)

16.8.4. Hot Rod C++ クライアントのインストール

16.8.4.1. Hot Rod C++ クライアントのダウンロードおよびインストール

Hot Rod C++ クライアントは、クライアントが使用されるオペレーティングシステムを基に 2 つのファイルタイプで配布されます。

- RPM ディストリビューションによる RHEL サーバーのインストール
- zip による Windows サーバーのインストール

16.8.4.2. RHEL における Hot Rod C++ クライアントのダウンロードおよびインストール

以下の手順にしたがってクライアントをインストールします。

1. Red Hat Subscription Manager を使用して Red Hat Enterprise Linux (RHEL) システムがアカウントに登録されている必要があります。詳細は [Red Hat Subscription Management のドキュメント](#) を参照してください。
2. Red Hat Subscription Manager を使用して、RHEL のバージョンに応じて適切なリポジトリを有効にします。

表16.77 RHSM リポジトリ

RHEL バージョン	リポジトリ名
RHEL 6	jboss-datagrid-7.2-for-rhel-6-server-rpms
RHEL 7	jboss-datagrid-7.2-for-rhel-7-server-rpms

たとえば、RHEL 7 リポジトリを有効にするには、以下のコマンドを使用できます。

```
subscription-manager repos --enable=jboss-datagrid-7.2-for-rhel-7-server-rpms
```

RHEL 7 では、**rhel-7-server-optional-rpms** リポジトリも有効にする必要があります。このリポジトリは必要な **protobuf-devel** および **protobuf-static** RPM を提供します。

```
subscription-manager repos --enable=rhel-7-server-optional-rpms
```

3. 適切なりポジトリが追加されたら、以下を実行して C++ クライアント RPM をインストールできます。

```
yum install jdgc-cpp-client
```

16.8.4.3. Windows における Hot Rod C++ クライアントのダウンロードおよびインストール

Windows の Hot Rod C++ は、Red Hat カスタマーポータル (<https://access.redhat.com>) の Red Hat JBoss Data Grid バイナリーにある **jboss-datagrid-<version>-hotrod-cpp-WIN-x86_64.zip** という個別の zip ファイルに含まれています。

この zip ファイルをダウンロードした後、システム上の任意の場所に展開すると C++ クライアントをインストールできます。

16.8.5. Hot Rod C++ クライアントでの Protobuf コンパイラーの使用

16.8.5.1. RHEL 7 における Protobuf コンパイラーの使用

RHEL 7 の C++ Hot Rod クライアントチャンネルには Protobuf コンパイラーが含まれています。このコンパイラーを使用するには、以下の手順にしたがいます。

1. 「[RHEL における Hot Rod C++ クライアントのダウンロードおよびインストール](#)」の説明どおりに、C++ チャンネルが RHEL システムに追加されている必要があります。
2. **protobuf** rpm をインストールします。

```
yum install protobuf
```

3. 含まれている protobuf ライブラリーをライブラリーのパスに追加します。これらのライブラリーはデフォルトでは **/opt/lib64** に含まれています。

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/lib64
```

4. 必要な protobuf ファイルを C++ ヘッダーおよびソースファイルにコンパイルします。

```
/bin/protoc --cpp_out  
dllexport_decl=HR_PROTO_EXPORT:/path/to/output/ $FILE
```



注記

HR_PROTO_EXPORT は Hot Rod クライアントコード内でマクロ定義され、ファイルが続いてコンパイルされたときに拡張されます。

5. 結果となるヘッダーとソースファイルは指定の出力ディレクトリーに生成され、特定のアプリケーションコードで通常どおりに参照およびコンパイルされます。

Protobuf の詳細は「[Protobuf エンコーディング](#)」を参照してください。

16.8.5.2. Windows における Protobuf コンパイラーの使用

Windows 用の C++ Hot Rod クライアントには、事前コンパイルされた Hot Rod コンポーネントと Protobuf コンパイラーが含まれます。多くの場合で、含まれるコンポーネントは追加のコンパイルなしで使用されますが、.proto ファイルにコンパイルが必要な場合は以下の手順にしたがいます。

1. **jboss-datagrid-<version>-hotrod-cpp-client-WIN-x86_64.zip** をローカルのファイルシステムに展開します。
2. コマンドプロンプトを開き、展開されたディレクトリーに移動します。
3. 必要な protobuf ファイルを C++ ヘッダーおよびソースファイルにコンパイルします。

```
bin\protoc --cpp_out dllexport_decl=HR_PROTO_EXPORT:path\to\output\
$FILE
```



注記

HR_PROTO_EXPORT は Hot Rod クライアントコード内でマクロ定義され、ファイルが続いてコンパイルされたときに拡張されます。

4. 結果となるヘッダーとソースファイルは指定の出力ディレクトリーに生成され、特定のアプリケーションコードで通常どおりに参照およびコンパイルされます。

Protobuf の詳細は「[Protobuf エンコーディング](#)」を参照してください。

16.8.6. Hot Rod C++ クライアントの設定

Hot Rod C++ クライアントは RemoteCache API を使用してリモートの Hot Rod サーバーと対話します。特定の Hot Rod サーバーとの通信を開始するために、RemoteCache を設定し、Hot Rod サーバーの特定のキャッシュを選択します。

ConfigurationBuilder API を使用して以下を設定します。

- 最初に接続するサーバーのセット。
- 接続プール属性。
- 接続/ソケットタイムアウトおよび TCP nodelay。
- Hot Rod プロトコルバージョン。

C++ の主な実行可能ファイルの設定例

以下の例は、ConfigurationBuilder を使用して **RemoteCacheManager** を設定する方法とデフォルトのリモートキャッシュを取得する方法を示しています。

SimpleMain.cpp

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include <stdlib.h>
```

```
using namespace infinispn::hotrod;
int main(int argc, char** argv) {
    ConfigurationBuilder b;
    b.addServer().host("127.0.0.1").port(11222);
    RemoteCacheManager cm(builder.build());
    RemoteCache<std::string, std::string> cache = cm.getCache<std::string,
std::string>();
    return 0;
}
```

16.8.7. Hot Rod C++ クライアント API

RemoteCacheManager は、RemoteCache への参照を取得する開始点です。RemoteCache API は、リモート Hot Rod サーバーとサーバー上の特定のキャッシュと対話できます。

前の例で取得した RemoteCache 参照を使用すると、リモートキャッシュで値を挿入、取得、置換、および削除できます。また、すべてのキーの取得やキャッシュのクリアなどの一括操作を実行することもできます。

RemoteCacheManager が停止されると、使用中のすべてのリソースが解放されます。

SimpleMain.cpp

```
RemoteCache<std::string, std::string> rc = cm.getCache<std::string,
std::string>();
    std::string k1("key13");
    std::string v1("boron");
    // put
    rc.put(k1, v1);
    std::auto_ptr<std::string> rv(rc.get(k1));
    rc.putIfAbsent(k1, v1);
    std::auto_ptr<std::string> rv2(rc.get(k1));
    std::map<HR_SHARED_PTR<std::string>, HR_SHARED_PTR<std::string> > map =
rc.getBulk(0);
    std::cout << "getBulk size" << map.size() << std::endl;
    ..
    .
    cm.stop();
```

16.8.8. Hot Rod C++ クライアント非同期 API

Hot Rod C++ クライアントは多くの同期メソッドの非同期のバージョンを提供し、リモートキャッシュと対話するための非ブロッキングメソッドを実現します。

非同期メソッドは同期メソッドと同じ命名規則にしたがいますが、各メソッドの末尾に **Async** が追加されます。非同期メソッドは操作の結果が含まれる **std::future** を返します。メソッドが **std::string** を返す場合は、代わりに **std::future < std::string* >** を返します。

以下に非同期メソッドのリストを示します。

- **clearAsync**
- **getAsync**
- **putAsync**

- putAllAsync
- putIfAbsentAsync
- removeAsync
- removeWithVersionAsync
- replaceAsync
- replaceWithVersionAsync

Hot Rod C++ 非同期 API の例

以下の例ではこれらのメソッドが使用されます。

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <iostream>
#include <thread>
#include <future>

using namespace infinispan::hotrod;

int main(int argc, char** argv) {
    ConfigurationBuilder builder;
    builder.addServer().host(argc > 1 ? argv[1] : "127.0.0.1").port(argc >
2 ? atoi(argv[2]) :
11222).protocolVersion(Configuration::PROTOCOL_VERSION_24);
    RemoteCacheManager cacheManager(builder.build(), false);
    auto *km = new BasicMarshaller<std::string>();
    auto *vm = new BasicMarshaller<std::string>();
    auto cache = cacheManager.getCache<std::string, std::string>(km,
&Marshaller<std::string>::destroy, vm, &Marshaller<std::string>::destroy
);
    cacheManager.start();
    std::string ak1("asyncK1");
    std::string av1("asyncV1");
    std::string ak2("asyncK2");
    std::string av2("asyncV2");
    cache.clear();

    // Put ak1,av1 in async thread
    std::future<std::string> future_put= cache.putAsync(ak1,av1);
    // Get the value in this thread
    std::string* arv1= cache.get(ak1);

    // Now wait for put completion
    future_put.wait();

    // All is synch now
    std::string* arv11= cache.get(ak1);
```

```

    if (!arv11 || arv11->compare(av1))
    {
        std::cout << "fail: expected " << av1 << "got " << (arv11 ? *arv11
: "null") << std::endl;
        return 1;
    }

    // Read ak1 again, but in async way and test that the result is the
same
    std::future<std::string*> future_ga= cache.getAsync(ak1);
    std::string* arv2= future_ga.get();
    if (!arv2 || arv2->compare(av1))
    {
        std::cerr << "fail: expected " << av1 << " got " << (arv2 ? *arv2
: "null") << std::endl;
        return 1;
    }

    // Now user pass a simple lambda func that set a flag to true when the
put completes
    bool flag=false;
    std::future<std::string*> future_put1= cache.putAsync(ak2,av2,0,0,[&
(std::string *v){flag=true; return v;});
    // The put is not completed here so flag must be false
    if (flag)
    {
        std::cerr << "fail: expected false got true" << std::endl;
        return 1;
    }
    // Now wait for put completion
    future_put1.wait();
    // The user lambda must be executed so flag must be true
    if (!flag)
    {
        std::cerr << "fail: expected true got false" << std::endl;
        return 1;
    }

    // Same test for get
    flag=false;
    // Now user pass a simple lambda func that set a flag to true when the
put completes
    std::future<std::string*> future_get1= cache.getAsync(ak2,[&
(std::string *v){flag=true; return v;});
    // The get is not completed here so flag must be false
    if (flag)
    {
        std::cerr << "fail: expected false got true" << std::endl;
        return 1;
    }
    // Now wait for get completion
    future_get1.wait();
    if (!flag)
    {
        std::cerr << "fail: expected true got false" << std::endl;
        return 1;
    }

```

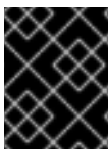
```

    }
    std::string* arv3= future_get1.get();
    if (!arv3 || arv3->compare(av2))
    {
        std::cerr << "fail: expected " << av2 << " got " << (arv3 ? *arv3
: "null") << std::endl;
        return 1;
    }
    cacheManager.stop();
}

```

16.8.9. Hot Rod C++ クライアントのリモートイベントリスナー

The Hot Rod C++ クライアントはリモートキャッシュリスナーをサポートします。これらのリスナーを追加するには **ClientCacheListener** で **add_listener** 関数を使用します。



重要

リモートイベントリスナーは、Red Hat JBoss Data Grid 7.2 の Hot Rod C++ クライアントではテクノロジープレビューの機能となります。

この関数は各イベントタイプ (**create**、**modify**、**remove**、**expire**、または **custom**) のリスナーを取ります。リモートイベントリスナーの詳細は「[リモートイベントリスナー \(Hot Rod\)](#)」を参照してください。例を以下に示します。

```

ConfigurationBuilder builder;
    builder.balancingStrategyProducer(nullptr);
builder.addServer().host("127.0.0.1").port(11222);
builder.protocolVersion(Configuration::PROTOCOL_VERSION_24);
RemoteCacheManager cacheManager(builder.build(), false);
cacheManager.start();
JBasicMarshaller<int> *km = new JBasicMarshaller<int>();
JBasicMarshaller<std::string> *vm = new JBasicMarshaller<std::string>();
RemoteCache<int, std::string> cache = cacheManager.getCache<int,
std::string>(km,
    &Marshaller<int>::destroy,
    vm,
    &Marshaller<std::string>::destroy);
cache.clear();
std::vector<std::vector<char> > filterFactoryParams;
std::vector<std::vector<char> > converterFactoryParams;
CacheClientListener<int, std::string> cl(cache);
int createdCount=0, modifiedCount=0, removedCount=0, expiredCount=0;

// We're using future and promise to have a basic listeners/main thread
synch
int setFutureEventKey=0;
std::promise<void> promise;
std::function<void(ClientCacheEntryCreatedEvent<int>)> listenerCreated =
[&createdCount, &setFutureEventKey, &promise]
(ClientCacheEntryCreatedEvent<int> e) { createdCount++; if
(setFutureEventKey==e.getKey()) promise.set_value(); };
std::function<void(ClientCacheEntryModifiedEvent<int>)> listenerModified =
[&modifiedCount, &setFutureEventKey, &promise]
(ClientCacheEntryModifiedEvent <int> e) { modifiedCount++; if

```



```
(setFutureEventKey==e.getKey()) promise.set_value(); };
std::function<void(ClientCacheEntryRemovedEvent<int>)> listenerRemoved =
[&removedCount, &setFutureEventKey, &promise](ClientCacheEntryRemovedEvent
<int> e) { removedCount++; if (setFutureEventKey==e.getKey())
promise.set_value(); };
std::function<void(ClientCacheEntryExpiredEvent<int>)> listenerExpired =
[&expiredCount, &setFutureEventKey, &promise](ClientCacheEntryExpiredEvent
<int> e) { expiredCount++; if (setFutureEventKey==e.getKey())
promise.set_value(); };

cl.add_listener(listenerCreated);
cl.add_listener(listenerModified);
cl.add_listener(listenerRemoved);
cl.add_listener(listenerExpired);

cache.addClientListener(cl, filterFactoryParams, converterFactoryParams);
```

16.8.10. サイトと動作する Hot Rod C++ クライアント

複数の Red Hat JBoss Data Grid サーバークラスターをデプロイし、各クラスターが異なるサイトに属するようにすることができます。このようなデプロイメントは、あるクラスターから別のクラスター(地理的に異なる場所にある可能性がある)にデータをバックアップできるようにするために行われます。C++ クライアント実装はクラスター内のノードの間でフェイルオーバーでき、元のクラスターが応答しなくなった場合は全体を他のクラスターにフェイルオーバーできます。クラスター間でのフェイルオーバーを可能にするには、すべての Red Hat JBoss Data Grid サーバークロスデータセンターレプリケーションで設定されている必要があります。この手順は、Red Hat JBoss Data Grid の『[Administration and Configuration Guide](#)』を参照してください。

フェイルオーバーが発生した場合、クライアントの代替クラスターへの接続は、例外が発生して代替クラスターが利用できなくなるまで維持されます。元のクラスターが操作可能になっても、クライアントは接続を自動的に切り替えません。元のクラスターに接続を切り替える場合は、以下の `switchToDefaultCluster()` メソッドを使用します。

クロスデータセンターレプリケーションがサーバ上で設定されたら、クライアントは代替クラスターの設定を提供し、設定した各クラスターに対して 1 つ以上のホスト/ポートペアの詳細情報を指定する必要があります。以下に例を示します。

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include <stdlib.h>
using namespace infinispan::hotrod;
int main(int argc, char** argv) {
    ConfigurationBuilder b;
    b.addServer().host("127.0.0.1").port(11222);
    b.addCluster("nyc").addClusterNode("127.0.0.1", 11322);

    RemoteCacheManager cm(builder.build());
    RemoteCache<std::string, std::string> cache = cm.getCache<std::string,
std::string>();
    return 0;
}
```

16.8.10.1. 手動クラスター切り替え

自動サイトフェイルオーバーの他にも、C++ クライアントは以下のメソッドのいずれかを呼び出してクラスターの切り替えを行うことができます。

- **switchToCluster(clusterName)** - クライアントを事前定義されたクラスター名に強制的に切り替えます。
- **switchToDefaultCluster** - クライアントをクライアント設定で定義された最初のサーバーに強制的に切り替えます。

16.8.11. Hot Rod C++ クライアントを用いたリモートクエリーの実行

Protobuf マーシャラーで **RemoteCacheManager** が設定された後、Hot Rod C++ クライアントでは Google の Protocol Buffers を使用してリモートクエリーを実行できます。



重要

リモートクエリーの実行は、Red Hat JBoss Data Grid 7.2 の Hot Rod C++ クライアントではテクノロジープレビューの機能となります。

Hot Rod C++ クライアントでのリモートクエリーの有効化

1. リモートの Red Hat JBoss Data Grid サーバーへの接続を取得します。

```
#include "addressbook.pb.h"
#include "bank.pb.h"
#include <infinispan/hotrod/BasicTypesProtoStreamMarshaller.h>
#include <infinispan/hotrod/ProtoStreamMarshaller.h>
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"
#include "infinispan/hotrod/query.pb.h"
#include "infinispan/hotrod/QueryUtils.h"
#include <vector>
#include <tuple>

#define PROTOBUF_METADATA_CACHE_NAME "__protobuf_metadata"
#define ERRORS_KEY_SUFFIX ".errors"

using namespace infinispan::hotrod;
using namespace org::infinispan::query::remote::client;

std::string read(std::string file)
{
    std::ifstream t(file);
    std::stringstream buffer;
    buffer << t.rdbuf();
    return buffer.str();
}

int main(int argc, char** argv) {
    std::cout << "Tests for Query" << std::endl;
    ConfigurationBuilder builder;
    builder.addServer().host(argc > 1 ? argv[1] :
"127.0.0.1").port(argc > 2 ? atoi(argv[2]) :
```

```
11222).protocolVersion(Configuration::PROTOCOL_VERSION_24);
RemoteCacheManager cacheManager(builder.build(), false);
cacheManager.start();
```

2. Protobuf マーシャラーで Protobuf メタデータキャッシュを作成します。

```
// This example continues the previous codeblock
// Create the Protobuf Metadata cache peer with a Protobuf
marshaller
auto *km = new BasicTypesProtoStreamMarshaller<std::string>();
auto *vm = new BasicTypesProtoStreamMarshaller<std::string>();
auto metadataCache = cacheManager.getCache<std::string,
std::string>(
    km, &Marshaller<std::string>::destroy,
    vm,
    &Marshaller<std::string>::destroy, PROTOBUF_METADATA_CACHE_NAME,
    false);
```

3. データモデルを Protobuf メタデータキャッシュにインストールします。

```
// This example continues the previous codeblock
// Install the data model into the Protobuf metadata cache
metadataCache.put("sample_bank_account/bank.proto",
read("proto/bank.proto"));
if (metadataCache.containsKey(ERRORS_KEY_SUFFIX))
{
    std::cerr << "fail: error in registering .proto model" <<
std::endl;
    return -1;
}
```

4. このステップは、この実証の目的でデータをキャッシュに追加します。リモートキャッシュを単にクエリーする場合、これは無視されることがあります。

```
// This example continues the previous codeblock
// Fill the cache with the application data: two users Tom and
Jerry
testCache.clear();
sample_bank_account::User_Address a;
sample_bank_account::User user1;
user1.set_id(3);
user1.set_name("Tom");
user1.set_surname("Cat");
user1.set_gender(sample_bank_account::User_Gender_MALE);
sample_bank_account::User_Address * addr= user1.add_addresses();
addr->set_street("Via Roma");
addr->set_number(3);
addr->set_postcode("202020");
testCache.put(3, user1);
user1.set_id(4);
user1.set_name("Jerry");
user1.set_surname("Mouse");
addr->set_street("Via Milano");
user1.set_gender(sample_bank_account::User_Gender_MALE);
testCache.put(4, user1);
```

5. リモートキャッシュをクエリーします。

```

// This example continues the previous codeblock
// Simple query to get User objects
{
    QueryRequest qr;
    std::cout << "Query: from sample_bank_account.User" <<
std::endl;
    qr.set_jpqlstring("from sample_bank_account.User");
    QueryResponse resp = testCache.query(qr);
    std::vector<sample_bank_account::User> res;
    unwrapResults(resp, res);
    for (auto i : res) {
        std::cout << "User(id=" << i.id() << ",name=" <<
i.name()
        << ",surname=" << i.surname() << ")" << std::endl;
    }
    cacheManager.stop();
    return 0;
}

```

その他のクエリー例

以下の例は、より複雑なクエリーを示すために含まれています。上記の手順の同じデータセットで使用することができます。

条件を用いたクエリーの使用

```

// Simple query to get User objects with where condition
{
    QueryRequest qr;
    std::cout << "from sample_bank_account.User u where
u.addresses.street=\"Via Milano\"" << std::endl;
    qr.set_jpqlstring("from sample_bank_account.User u where
u.addresses.street=\"Via Milano\"");
    QueryResponse resp = testCache.query(qr);
    std::vector<sample_bank_account::User> res;
    unwrapResults(resp, res);
    for (auto i : res) {
        std::cout << "User(id=" << i.id() << ",name=" << i.name()
        << ",surname=" << i.surname() << ")" << std::endl;
    }
}

```

射影を用いたクエリーの使用

```

// Simple query to get projection (name, surname)
{
    QueryRequest qr;
    std::cout << "Query: select u.name, u.surname from
sample_bank_account.User u" << std::endl;
    qr.set_jpqlstring(
        "select u.name, u.surname from sample_bank_account.User u");
    QueryResponse resp = testCache.query(qr);
}

```

```

//Typed resultset
std::vector<std::tuple<std::string, std::string> > prjRes;
unwrapProjection(resp, prjRes);
for (auto i : prjRes) {
    std::cout << "Name: " << std::get<0> (i)
    << " Surname: " << std::get<1> (i) << std::endl;
}
}

```

16.8.12. Hot Rod C++ クライアントでのニアキャッシュの使用

ニアキャッシュは Hot Rod C++ クライアントの任意のキャッシュで、ユーザーの近くに最近アクセスされたデータを保持することで、頻繁に使用されるデータへのアクセスをより迅速にします。このキャッシュは、バックグラウンドでリモートサーバーに同期されたローカルの Hot Rod クライアントキャッシュとして動作します。

ニアキャッシュをプログラムを使用して **ConfigurationBuilder** で有効にするには、以下の例のように **nearCache()** メソッドを使用します。

```

int main(int argc, char** argv) {
    ConfigurationBuilder confBuilder;
    confBuilder.addServer().host("127.0.0.1").port(11222);
    confBuilder.protocolVersion(Configuration::PROTOCOL_VERSION_24);
    confBuilder.balancingStrategyProducer(nullptr);

    // Enable the near cache support

    confBuilder.nearCache().mode(NearCacheMode::INVALIDATED).maxEntries(4);
}

```

ニアキャッシュの動作の設定には、以下のメソッドが使用されます。

- **nearCache()** - さらに変更を追加できる **NearCacheConfigurationBuilder** を定義します。
- **mode(NearCacheMode mode)** - **NearCacheMode** を渡す必要があります。デフォルトは **DISABLED** で、ニアキャッシュはすべて無効になります。
- **maxEntries(int maxEntries)** - ニアキャッシュに含まれるエントリーの最大数を示します。ニアキャッシュが満杯になると、最も古いエントリーがエビクトされます。この値を **0** に設定すると、バインドされないニアキャッシュが定義されます。

ニアキャッシュのエントリーは、イベントを介してリモートキャッシュとのアライメントを維持します。サーバーで変更が発生すると、適切なイベントがクライアントへ送信され、ニアキャッシュを更新します。

16.8.13. Hot Rod C++ クライアントを使用したスクリプトの実行

Hot Rod C++ クライアントを使用すると、リモート実行を介してタスクを直接 JBoss Data Grid サーバーで実行できます。この機能はデータに近いロジックを実行し、クラスターのノードすべてのリソースを利用します。タスクはサーバーインスタンスにデプロイでき、デプロイ後にプログラムを使用して実行できます。



重要

リモート実行は、Red Hat JBoss Data Grid 7.2 の Hot Rod C++ クライアントではテクノロジープレビューの機能となります。

タスクのインストール

`__script_cache` の `put(std::string name, std::string script)` メソッドが使用されると、タスクはサーバーにインストールされます。スクリプト名の拡張はスクリプトの実行に使用されるエンジンを判断しますが、これはスクリプト自体のメタデータによって上書きされます。

以下の例はスクリプトをインストールします。

C++ クライアントでのタスクのインストール

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"
#include "infinispan/hotrod/JBasicMarshaller.h"
using namespace infinispan::hotrod;
int main(int argc, char** argv) {
    // Configure the client
    ConfigurationBuilder builder;
    builder.addServer().host("127.0.0.1").port(11222).protocolVersion(
        Configuration::PROTOCOL_VERSION_24);
    RemoteCacheManager cacheManager(builder.build(), false);
    try {
        // Create the cache with the given marshallers
        auto *km = new JBasicMarshaller<std::string>();
        auto *vm = new JBasicMarshaller<std::string>();
        RemoteCache<std::string, std::string> cache = cacheManager.getCache<
            std::string, std::string>(km, &Marshaller<std::string>::destroy,
            vm, &Marshaller<std::string>::destroy,
            std::string("namedCache"));
        cacheManager.start();

        // Obtain a reference to the __script_cache
        RemoteCache<std::string, std::string> scriptCache =
            cacheManager.getCache<std::string, std::string>(
                "__script_cache", false);
        // Install on the server the getValue script
        std::string getValueScript(
            "// mode=local,language=javascript\n "
            "var cache = cacheManager.getCache(\"namedCache\");\n "
            "var ct = cache.get(\"accessCounter\");\n "
            "var c = ct==null ? 0 : parseInt(ct);\n "
            "cache.put(\"accessCounter\",(++c).toString());\n "
            "cache.get(\"privateValue\") ");
        std::string getValueScriptName("getValue.js");
        std::string pGetValueScriptName =
            JBasicMarshaller<std::string>::addPreamble(getValueScriptName);
        std::string pGetValueScript =
            JBasicMarshaller<std::string>::addPreamble(getValueScript);
        scriptCache.put(pGetValueScriptName, pGetValueScript);
        // Install on the server the get access counter script
```

```

std::string getAccessScript(
    "// mode=local,language=javascript\n "
    "var cache = cacheManager.getCache(\"namedCache\");\n "
    "cache.get(\"accessCounter\");");
std::string getAccessScriptName("getAccessCounter.js");
std::string pGetAccessScriptName =
    JBasicMarshaller<std::string>::addPreamble(getAccessScriptName);
std::string pGetAccessScript =
    JBasicMarshaller<std::string>::addPreamble(getAccessScript);
scriptCache.put(pGetAccessScriptName, pGetAccessScript);

```

タスクの実行

インストール後、タスクは **execute(std::string name, std::map<std::string, std::string> args)** メソッドを使用して実行されます。このメソッドには、実行するスクリプトの名前と実行に必要な引数を指定します。

以下の例はスクリプトを実行します。

C++ クライアントでのスクリプトの実行

```

// The following is a continuation of the above example
cache.put("privateValue", "Counted Access Value");
std::map<std::string, std::string> s;
// Execute the getValue script
std::vector<unsigned char> execValueResult = cache.execute(
    getValueScriptName, s);
// Execute the getAccess script
std::vector<unsigned char> execAccessResult = cache.execute(
    getAccessScriptName, s);

std::string value(
    JBasicMarshallerHelper::unmarshall<std::string>(
        (char*) execValueResult.data()));
std::string access(
    JBasicMarshallerHelper::unmarshall<std::string>(
        (char*) execAccessResult.data()));

std::cout << "Returned value is '" << value
    << "' and has been accessed: " << access << " times."
    << std::endl;

} catch (const Exception& e) {
std::cout << "is: " << typeid(e).name() << '\n';
std::cerr << "fail unexpected exception: " << e.what() << std::endl;
return 1;
}

cacheManager.stop();
return 0;
}

```

16.9. HOT ROD C# クライアント

16.9.1. Hot Rod C# クライアント

Hot Rod C# クライアントでは、.NET ランタイムアプリケーションは Red Hat JBoss Data Grid サーバーへ接続し、対話できます。Hot Rod C# クライアントは、クラスタトポロジーとハッシュスキームを認識し、Hot Rod Java クライアントと Hot Rod C++ クライアントに類似した単一のホップでサーバー上のエンタリにアクセスできます。

Hot Rod C# クライアントは、.NET Framework が Microsoft Visual Studio 2015 によりサポートされる 64 ビットのオペレーティングシステムと互換性があります。NET 4.6.2 は Hot Rod C# クライアントに必須です。

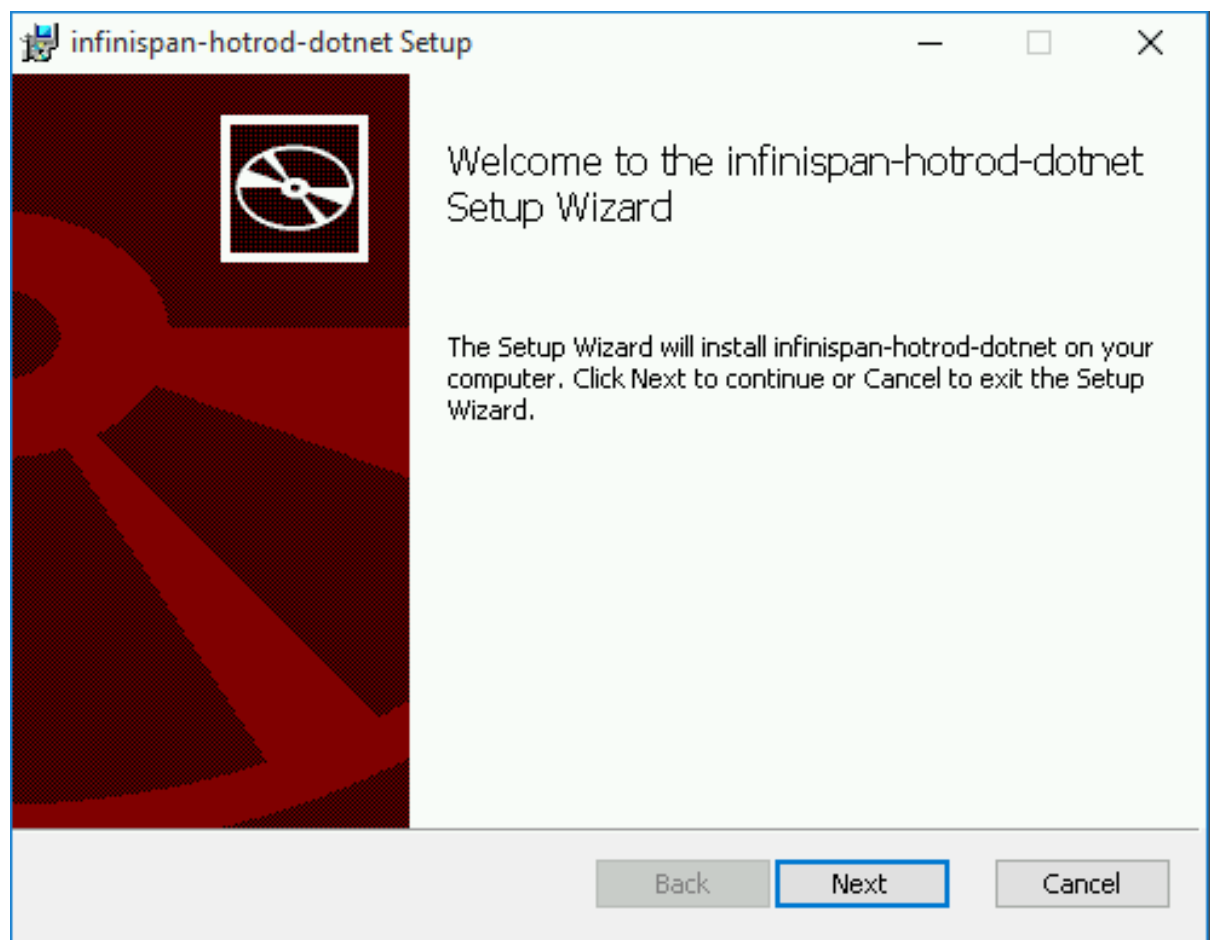
16.9.2. Hot Rod C# クライアントのダウンロードとインストール

Hot Rod C# クライアントは、Red Hat JBoss Data Grid でダウンロード用にパッケージされた .msi ファイル `jboss-datagrid-<version>-hotrod-dotnet-client.msi` に含まれます。Hot Rod C# クライアントをインストールするには、以下の手順を実行してください。

Hot Rod C# クライアントのインストール

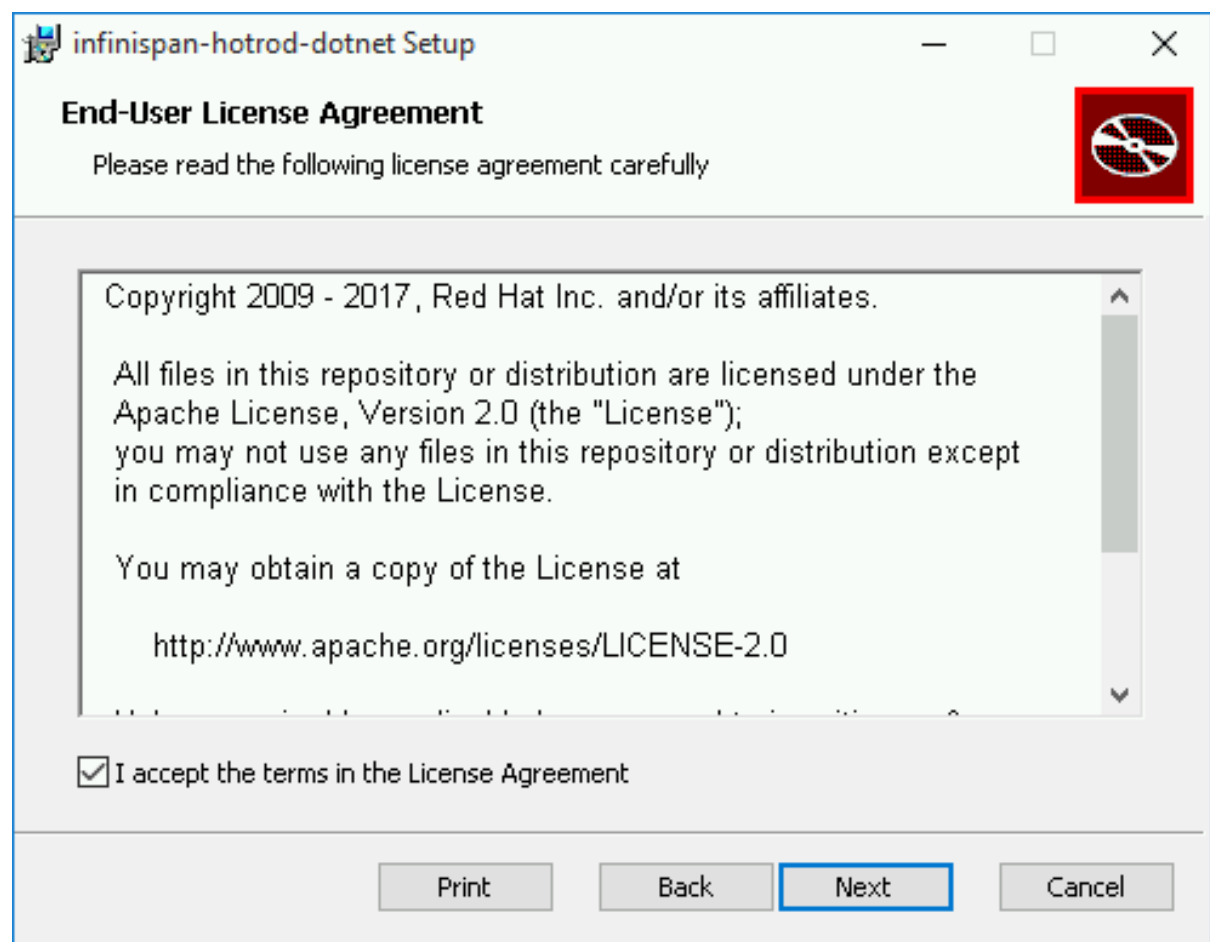
1. 管理者として、Hot Rod C# .msi ファイルがダウンロードされた場所に移動します。.msi ファイルを実行してウィンドウインストーラーを起動し、**Next** をクリックします。

図16.1 Hot Rod C# クライアントのセットアップの開始



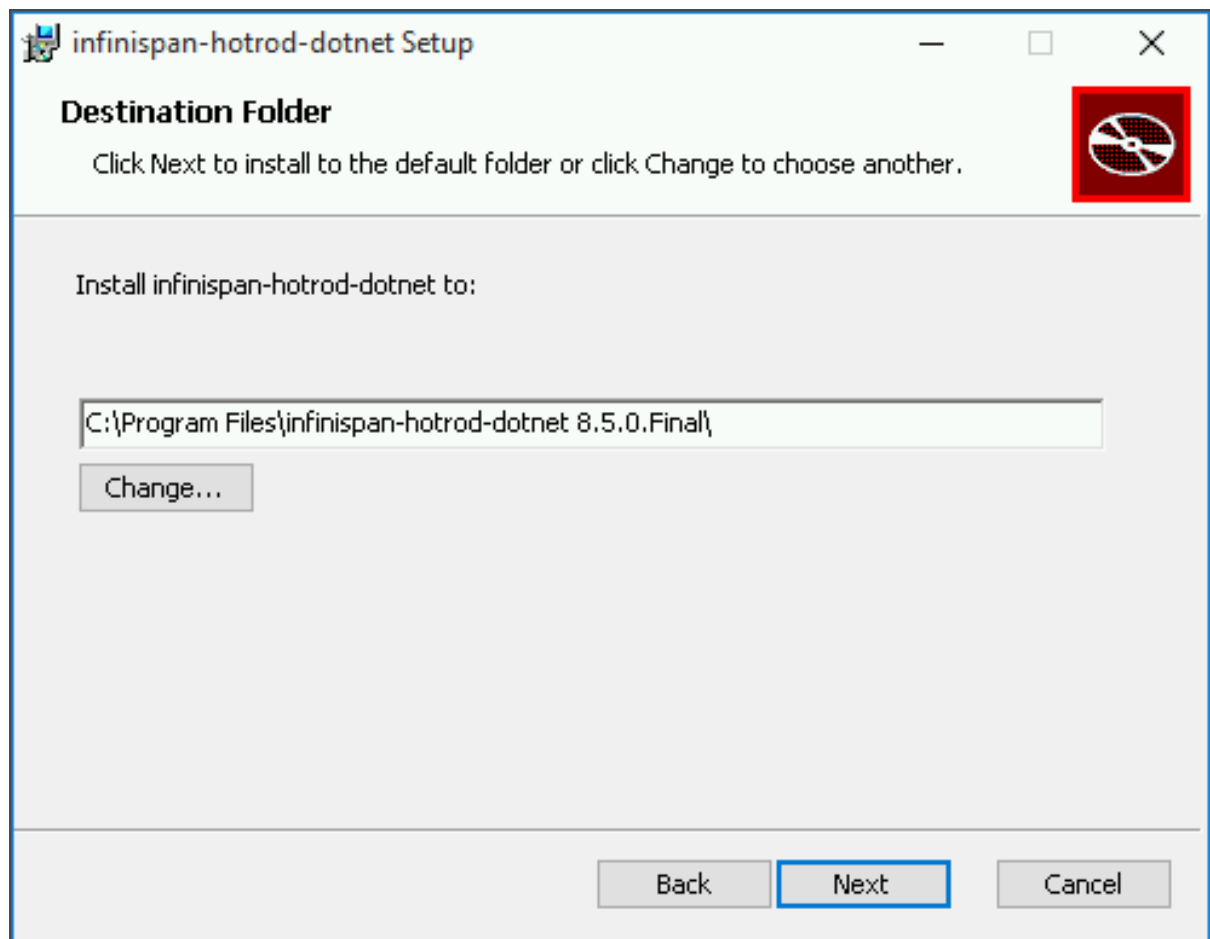
2. 使用許諾契約書の内容を確認します。**I accept the terms in the License Agreement** (使用許諾契約に同意します) チェックボックスを選択し、**Next** をクリックします。

図16.2 Hot Rod C# クライアントの使用許諾契約



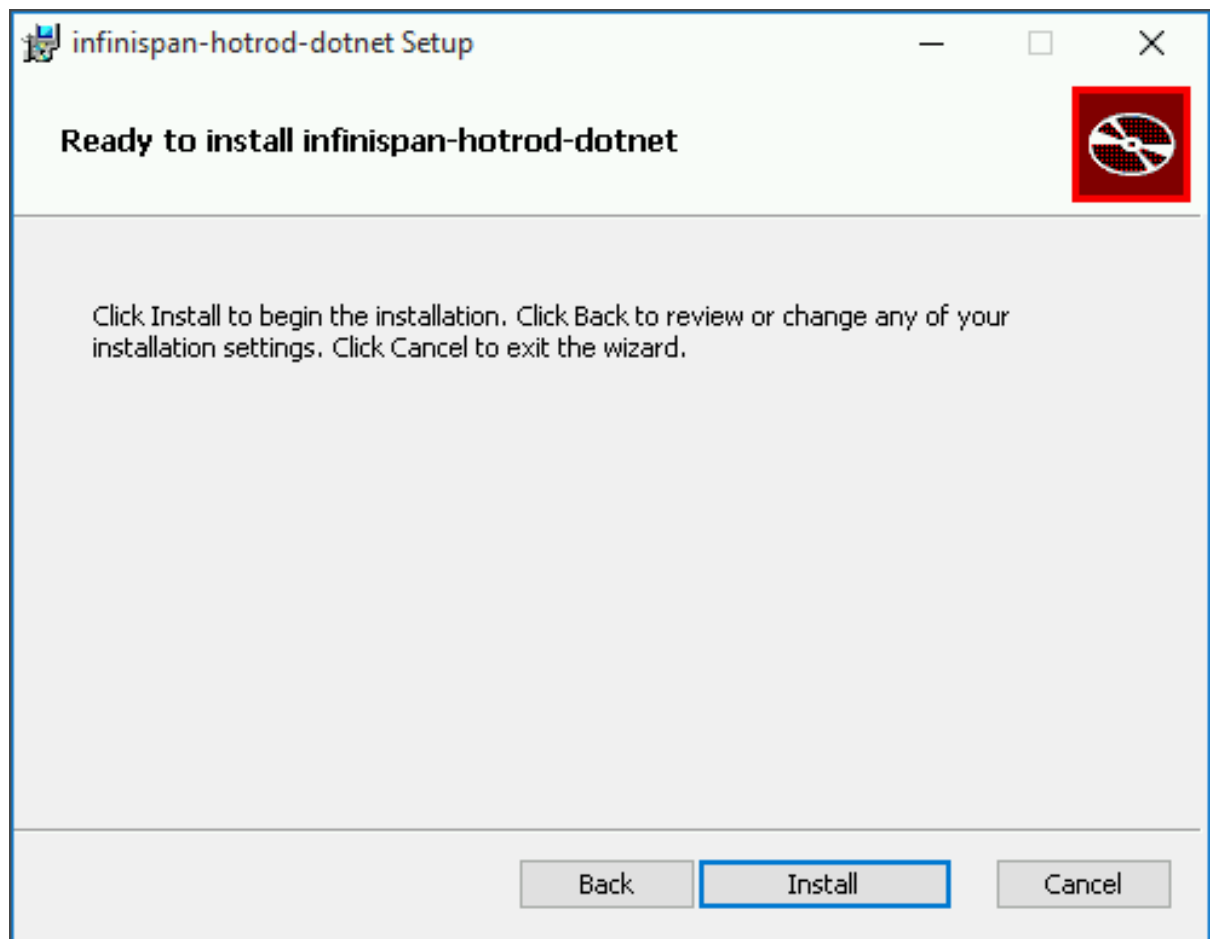
3. デフォルトのディレクトリーを変更するには、**Change...** または **Next** をクリックしてデフォルトのディレクトリーをインストールします。

図16.3 Hot Rod C# クライアントの宛先フォルダー



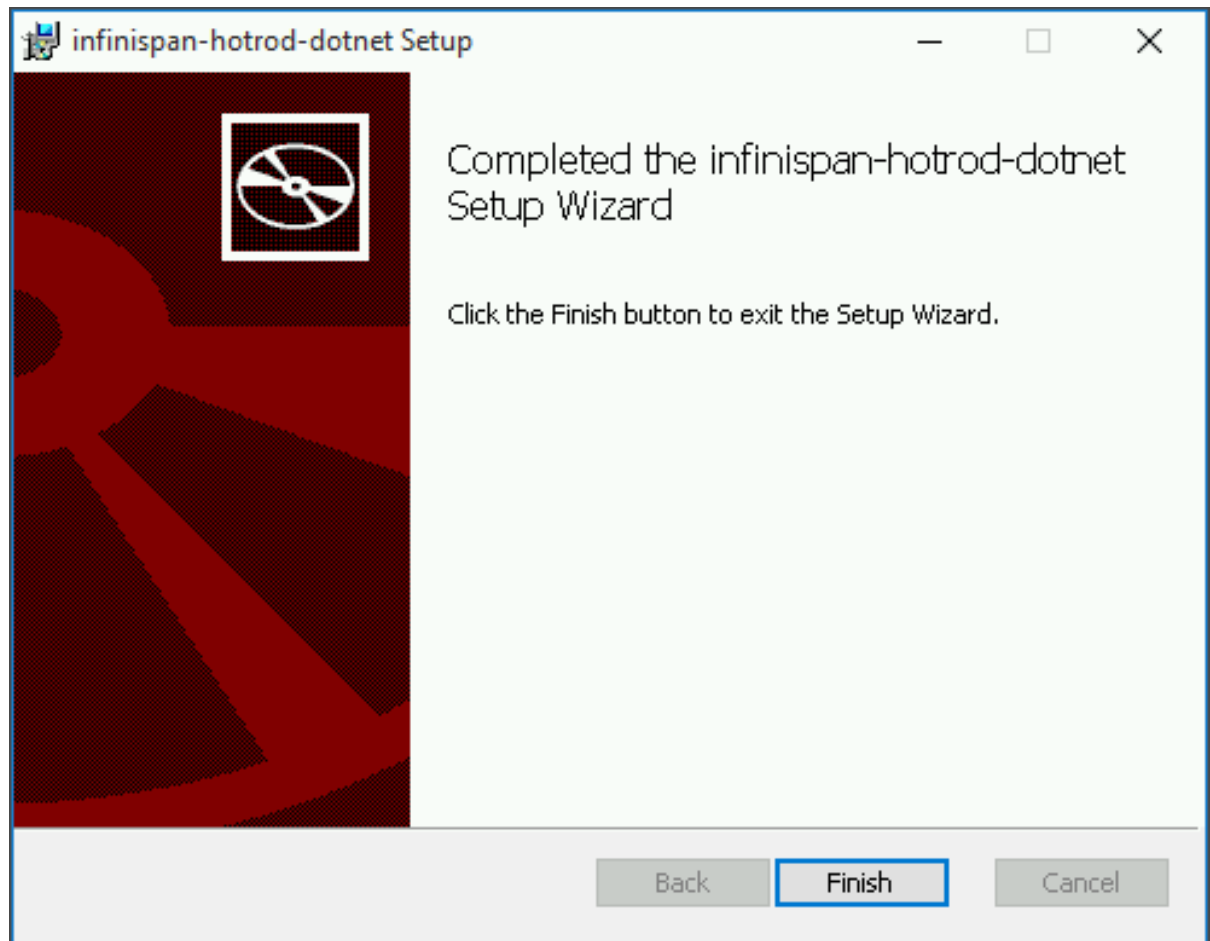
4. **Install** をクリックして、Hot Rod C# クライアントのインストールを開始します。

図16.4 Hot Rod C# クライアントのインストールを開始



5. **Finish** をクリックして Hot Rod C# クライアントのインストールを完了します。

図16.5 Hot Rod C# クライアントのセットアップの完了



16.9.3. Hot Rod C# .NET プロジェクトの作成

.NET プロジェクトで Hot Rod C# クライアントを使用するには、以下の手順を実行する必要があります。

Hot Rod C# プロジェクトの設定

1. Path 環境変数の追加

PATH 環境変数に以下のフォルダーを追加する必要があります。

```
C:\path\to\infinispan-hotrod-dotnet 8.5.0.Final\bin  
C:\path\to\infinispan-hotrod-dotnet 8.5.0.Final\lib
```

2. **Prefer 32 bit** の削除

Build タブの Project プロパティにある **Prefer 32 bit** にチェックマークが入っていないようにしてください。

3. Hot Rod C# dll の追加

- a. **Solution Explorer** ビューで **Project** を選択します。
- b. **References** を選択します。
- c. 参照を右クリックし、**Add Reference** を選択します。

- d. 表示されたウィンドウで **Browse** をクリックし、**C:\path\to\infinispan-hotrod-dotnet 8.5.0.Final\lib\hotrodcs.dll** ファイルに移動します。
- e. **OK** をクリックします。

Hot Rod C# API が .NET プロジェクトで使用できるようになりました。

16.9.4. Hot Rod C# クライアントの設定

Hot Rod C# クライアントは `ConfigurationBuilder` を使用してプログラミングにより設定されます。クライアントが接続する必要があるホストとポートを設定します。

C# ファイルの設定例

以下の例は、`ConfigurationBuilder` を使用して **RemoteCacheManager** を設定する方法を示しています。

C# の設定

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
namespace simpleapp
{
    class Program
    {
        static void Main(string[] args)
        {
            ConfigurationBuilder builder = new ConfigurationBuilder();
            builder.AddServer()
                .Host(args.Length > 1 ? args[0] : "127.0.0.1")
                .Port(args.Length > 2 ? int.Parse(args[1]) : 11222);
            Configuration config = builder.Build();
            RemoteCacheManager cacheManager = new
RemoteCacheManager(config);
            [...]
        }
    }
}
```

16.9.5. Hot Rod C# クライアント API

RemoteCacheManager は、`RemoteCache` への参照を取得する開始点です。

以下の例は、サーバーからのデフォルトキャッシュの取得と基本的な複数の操作を示しています。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```

using Infinispan.HotRod;
using Infinispan.HotRod.Config;
namespace simpleapp
{
    class Program
    {
        static void Main(string[] args)
        {
            ConfigurationBuilder builder = new ConfigurationBuilder();
            builder.AddServer()
                .Host(args.Length > 1 ? args[0] : "127.0.0.1")
                .Port(args.Length > 2 ? int.Parse(args[1]) : 11222);
            Configuration config = builder.Build();
            RemoteCacheManager cacheManager = new
RemoteCacheManager(config);
            cacheManager.Start();
            // Retrieve a reference to the default cache.
            IRemoteCache<String, String> cache =
cacheManager.GetCache<String, String>();
            // Add entries.
            cache.Put("key1", "value1");
            cache.PutIfAbsent("key1", "anotherValue1");
            cache.PutIfAbsent("key2", "value2");
            cache.PutIfAbsent("key3", "value3");
            // Retrive entries.
            Console.WriteLine("key1 -> " + cache.Get("key1"));
            // Bulk retrieve key/value pairs.
            int limit = 10;
            IDictionary<String, String> result = cache.GetBulk(limit);
            foreach (KeyValuePair<String, String> kv in result)
            {
                Console.WriteLine(kv.Key + " -> " + kv.Value);
            }
            // Remove entries.
            cache.Remove("key2");
            Console.WriteLine("key2 -> " + cache.Get("key2"));
            cacheManager.Stop();
        }
    }
}

```

16.9.6. Hot Rod C# クライアント非同期 API

Hot Rod C# クライアントは多くの同期メソッドの非同期のバージョンを提供し、リモートキャッシュと対話するための非ブロッキングメソッドを実現します。

非同期メソッドは同期メソッドと同じ命名規則にしたがいますが、各メソッドの末尾に `Async` が追加されます。非同期メソッドは操作の結果が含まれる **Task** を返します。メソッドが **String** を返す場合は、代わりに **Task<String>** を返します。

以下に非同期メソッドのリストを示します。

- **ClearAsync**
- **GetAsync**

- PutAsync
- PutAllAsync
- PutIfAbsentAsync
- RemoveAsync
- RemoveWithVersionAsync
- ReplaceAsync
- ReplaceWithVersionAsync

Hot Rod C# 非同期 API の例

以下の例ではこれらのメソッドが使用されます。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
namespace simpleapp
{
    class Program
    {
        static void Main(string[] args)
        {
            ConfigurationBuilder builder = new ConfigurationBuilder();
            builder.AddServer()
                .Host(args.Length > 1 ? args[0] : "127.0.0.1")
                .Port(args.Length > 2 ? int.Parse(args[1]) : 11222);
            Configuration config = builder.Build();
            RemoteCacheManager cacheManager = new
RemoteCacheManager(config);
            IRemoteCache<String,String> cache =
cacheManager.GetCache<String,String>();

            // Add Entries Async
            cache.PutAsync("key1", "value1");
            cache.PutAsync("key2", "value2");

            // Retrieve Entries Async
            Task<string> futureExec = cache.GetAsync("key1");

            string result = futureExec.Result;
        }
    }
}
```

16.9.7. Hot Rod C# クライアントのリモートイベントリスナー

The Hot Rod C++ クライアントはリモートキャッシュリスナーをサポートします。これらのリスナーを追加するには **ClientListener** で **addListener** 関数を使用します。



重要

リモートイベントリスナーは、Red Hat JBoss Data Grid 7.2 の Hot Rod C++ クライアントではテクノロジープレビューの機能となります。

このメソッドは各イベントタイプ (**create**、**modify**、**remove**、**expire**、または **custom**) のリスナーを取ります。リモートイベントリスナーの詳細は「[リモートイベントリスナー \(Hot Rod\)](#)」を参照してください。modifiedEvent の例は次のとおりです。

```
[...]
private static void
modifiedEventAction(Event.ClientCacheEntryModifiedEvent<string> e)
{
    ++modifiedEventCounter;
    modifiedSemaphore.Release();
}
[...]
public void ModifiedEventTest()
{
    IRemoteCache<string, string> cache = remoteManager.GetCache<string,
string>();
    cache.Clear();
    Event.ClientListener<string, string> cl = new
Event.ClientListener<string, string>();
    cl.filterFactoryName = "";
    cl.converterFactoryName = "";
    cl.addListener(modifiedEventAction);
    cache.addClientListener(cl, new string[] { }, new string[] { }, null);
}
```

16.9.8. サイトと動作する Hot Rod C# クライアント

複数の Red Hat JBoss Data Grid サーバークラスターをデプロイし、各クラスターが異なるサイトに属するようにすることができます。このようなデプロイメントは、あるクラスターから別のクラスター (地理的に異なる場所にある可能性がある) にデータをバックアップできるようにするために行われます。C# クライアント実装はクラスター内のノードの間でフェイルオーバーでき、元のクラスターが応答しなくなった場合は全体を他のクラスターにフェイルオーバーできます。クラスター間でのフェイルオーバーを可能にするには、すべての Red Hat JBoss Data Grid サーバーがクロスデータセンターレプリケーションで設定されている必要があります。この手順は、Red Hat JBoss Data Grid [『Administration and Configuration Guide』](#) を参照してください。

フェイルオーバーが発生した場合、クライアントの代替クラスターへの接続は、例外が発生して代替クラスターが利用できなくなるまで維持されます。元のクラスターが操作可能になっても、クライアントは接続を自動的に切り替えません。元のクラスターに接続を切り替える場合は、以下の **SwitchToDefaultCluster()** メソッドを使用します。

クロスデータセンターレプリケーションがサーバー上で設定されたら、クライアントは代替クラスターの設定を提供し、設定した各クラスターに対して 1 つ以上のホスト/ポートペアの詳細情報を指定する必要があります。以下に例を示します。

```
ConfigurationBuilder conf1 = new ConfigurationBuilder();
```

```

conf1.AddServer().Host("127.0.0.1").Port(11222);
conf1.AddCluster("nyc").AddClusterNode("127.0.0.1", 11322);
RemoteCacheManager manager1 = new RemoteCacheManager(conf1.Build(), true);

ConfigurationBuilder conf2 = new ConfigurationBuilder();
conf2.AddServer().Host("127.0.0.1").Port(11322);
conf2.AddCluster("lon").AddClusterNode("127.0.0.1", 11222);
RemoteCacheManager remoteManager = new RemoteCacheManager(conf2.Build(),
true);

```

16.9.8.1. 手動クラスター切り替え

自動サイトフェイルオーバーの他にも、C++ クライアントは以下のメソッドのいずれかを呼び出してクラスターの切り替えを行うことができます。

- **SwitchToCluster(clusterName)** - クライアントを事前定義されたクラスター名に強制的に切り替えます。
- **SwitchToDefaultCluster()** - クライアントをクライアント設定で定義された最初のサーバーに強制的に切り替えます。

16.9.9. Hot Rod C# クライアントを用いたリモートクエリーの実行

Protobuf マーシャラーで **RemoteCacheManager** が設定された後、Hot Rod C# クライアントでは Google の Protocol Buffers を使用してリモートクエリーを実行できます。



重要

リモートクエリーの実行は、Red Hat JBoss Data Grid 7.2 の Hot Rod C# クライアントではテクノロジープレビューの機能となります。

Hot Rod C# クライアントでのリモートクエリーの有効化

1. Protobuf マーシャラーを設定に渡し、リモートの Red Hat JBoss Data Grid サーバーへの接続を取得します。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
using Google.Protobuf;
using Org.Infinispan.Protostream;
using Org.Infinispan.Query.Remote.Client;
using QueryExampleBankAccount;
using System.IO;

namespace Query
{
    /// <summary>
    /// This sample code shows how to perform Infinispan queries
    using the C# client
    /// </summary>

```



```

class Query
{
    static void Main(string[] args)
    {
        // Cache manager setup
        RemoteCacheManager remoteManager;
        const string ERRORS_KEY_SUFFIX = ".errors";
        const string PROTOBUF_METADATA_CACHE_NAME =
            "__protobuf_metadata";
        ConfigurationBuilder conf = new ConfigurationBuilder();

        conf.AddServer().Host("127.0.0.1").Port(11222).ConnectionTimeout(900
00).SocketTimeout(6000);
        conf.Marshaller(new BasicTypesProtoStreamMarshaller());
        remoteManager = new RemoteCacheManager(conf.Build(),
            true);

        IRemoteCache<String, String> metadataCache =
            remoteManager.GetCache<String, String>
                (PROTOBUF_METADATA_CACHE_NAME);
        IRemoteCache<int, User> testCache =
            remoteManager.GetCache<int, User>("namedCache");
    }
}

```

2. Protobuf エンティティモデルをインストールします。

```

        // This example continues the previous codeblock
        // Installing the entities model into the Infinispan
        __protobuf_metadata cache
        metadataCache.Put("sample_bank_account/bank.proto",
            File.ReadAllText("resources/proto2/bank.proto"));
        if (metadataCache.ContainsKey(ERRORS_KEY_SUFFIX))
        {
            Console.WriteLine("fail: error in registering
                .proto model");
            Environment.Exit(-1);
        }
    }
}

```

3. このステップは、この実証の目的でデータをキャッシュに追加します。リモートキャッシュを単にクエリーする場合、これは無視されることがあります。

```

        // This example continues the previous codeblock
        // The application cache must contain entities only
        testCache.Clear();
        // Fill the application cache
        User user1 = new User();
        user1.Id = 4;
        user1.Name = "Jerry";
        user1.Surname = "Mouse";
        User ret = testCache.Put(4, user1);
    }
}

```

4. リモートキャッシュをクエリーします。

```

        // This example continues the previous codeblock
        // Run a query
        QueryRequest qr = new QueryRequest();
        qr.JpqlString = "from sample_bank_account.User";
    }
}

```

```

        QueryResponse result = testCache.Query(qr);
        List<User> listOfUsers = new List<User>();
        unwrapResults(result, listOfUsers);
    }

```

5. 結果を処理するには、protobuf を C# オブジェクトに変換します。以下の例はこの変換を示しています。

```

// Convert Protobuf matter into C# objects
private static bool unwrapResults<T>(QueryResponse resp,
List<T> res) where T : IMessage<T>
{
    if (resp.ProjectionSize > 0)
    { // Query has select
        return false;
    }
    for (int i = 0; i < resp.NumResults; i++)
    {
        WrappedMessage wm = resp.Results.ElementAt(i);

        if (wm.WrappedBytes != null)
        {
            WrappedMessage wmr =
WrappedMessage.Parser.ParseFrom(wm.WrappedBytes);
            if (wmr.WrappedMessageBytes != null)
            {
                System.Reflection.PropertyInfo pi =
typeof(T).GetProperty("Parser");

                MessageParser<T> p =
(MessageParser<T>)pi.GetValue(null);
                T u = p.ParseFrom(wmr.WrappedMessageBytes);
                res.Add(u);
            }
        }
    }
    return true;
}
}

```

16.9.10. Hot Rod C# クライアントでのニアキャッシュの使用

ニアキャッシュは Hot Rod C# クライアントの任意のキャッシュで、ユーザーの近くに最近アクセスされたデータを保持することで、頻繁に使用されるデータへのアクセスをより迅速にします。このキャッシュは、バックグラウンドでリモートサーバーに同期されたローカルの Hot Rod クライアントキャッシュとして動作します。

ニアキャッシュをプログラムを使用して **ConfigurationBuilder** で有効にするには、以下の例のように **NearCache()** メソッドを使用します。

```

ConfigurationBuilder conf = new ConfigurationBuilder();
conf.AddServer().Host("127.0.0.1").Port(11222)

```

```
// Define a Near Cache that contains up to 10 entries
.NearCache().Mode(NearCacheMode.INVALIDATED).MaxEntries(10);
```

ニアキャッシュの動作の設定には、以下のメソッドが使用されます。

- **NearCache()** - さらに変更を追加できる **NearCacheConfigurationBuilder** を定義します。
- **Mode(NearCacheMode mode)** - **NearCacheMode** を渡す必要があります。デフォルトは **DISABLED** で、ニアキャッシュはすべて無効になります。
- **MaxEntries(int maxEntries)** - ニアキャッシュに含まれるエントリーの最大数を示します。ニアキャッシュが満杯になると、最も古いエントリーがエビクトされます。この値を **0** に設定すると、バインドされないニアキャッシュが定義されます。

ニアキャッシュのエントリーは、イベントを介してリモートキャッシュとのアライメントを維持します。サーバーで変更が発生すると、適切なイベントがクライアントへ送信され、ニアキャッシュを更新します。

16.9.11. Hot Rod C# クライアントを使用したスクリプトの実行

Hot Rod C# クライアントを使用すると、リモート実行を介してタスクを直接 Red Hat JBoss Data Grid サーバーで実行できます。この機能はデータに近いロジックを実行し、クラスターのノードすべてのリソースを利用します。タスクはサーバーインスタンスにデプロイでき、デプロイ後にプログラムを使用して実行できます。

タスクのインストール

`__script_cache` の **Put(string name, string script)** メソッドが使用されると、タスクはサーバーにインストールされます。スクリプト名の拡張はスクリプトの実行に使用されるエンジンを判断しますが、これはスクリプト自体のメタデータによって上書きされます。

以下の例はスクリプトをインストールします。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;

namespace RemoteExec
{
    /// <summary>
    /// This sample code shows how to perform a server remote execution
    using the C# client
    /// </summary>
    class RemoteExec
    {
        static void Main(string[] args)
        {
            // Cache manager setup
            RemoteCacheManager remoteManager;
            IMarshaller marshaller;
            ConfigurationBuilder conf = new ConfigurationBuilder();
```

```

conf.AddServer().Host("127.0.0.1").Port(11222).ConnectionTimeout(90000).SocketTimeout(6000);
marshaller = new JBasicMarshaller();
conf.Marshaller(marshaller);
remoteManager = new RemoteCacheManager(conf.Build(), true);

// Install the .js code into the Infinispan __script_cache
const string SCRIPT_CACHE_NAME = "__script_cache";
string valueScriptName = "getValue.js";
string valueScript = "// mode=local,language=javascript\n "
    + "var cache = cacheManager.getCache(\"namedCache\");\n "
    + "var ct = cache.get(\"accessCounter\");\n "
    + "var c = ct==null ? 0 : parseInt(ct);\n "
    + "cache.put(\"accessCounter\",(++c).toString());\n "
    + "cache.get(\"privateValue\") ";
string accessScriptName = "getAccess.js";
string accessScript = "// mode=local,language=javascript\n "
    + "var cache = cacheManager.getCache(\"namedCache\");\n "
    + "cache.get(\"accessCounter\");";
IRemoteCache<string, string> scriptCache =
remoteManager.GetCache<string, string>(SCRIPT_CACHE_NAME);
IRemoteCache<string, string> testCache =
remoteManager.GetCache<string, string>("namedCache");
scriptCache.Put(valueScriptName, valueScript);
scriptCache.Put(accessScriptName, accessScript);

```

タスクの実行

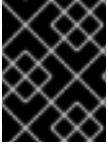
インストール後、タスクは **Execute(string name, Dictionary<string, string> scriptArgs)** メソッドを使用して実行されます。このメソッドには、実行するスクリプトの名前と実行に必要な引数を指定します。

以下の例はスクリプトを実行します。

```

// This example continues the previous codeblock
testCache.Put("privateValue", "Counted Access Value");
Dictionary<string, string> scriptArgs = new
Dictionary<string, string>();
byte[] ret1 = testCache.Execute(valueScriptName, scriptArgs);
string value = (string)marshaller.ObjectFromByteBuffer(ret1);
byte[] ret2 = testCache.Execute(accessScriptName, scriptArgs);
string accessCount =
(string)marshaller.ObjectFromByteBuffer(ret2);
Console.WriteLine("Return value is '" + value + "' and has been
accessed '" + accessCount + "' times.");
    }
}
}

```

**重要**

Hot Rod C# クライアントを使用したスクリプトの実行は、JBoss Data Grid 7.2 ではテクノロジープレビューの機能となります。

16.9.12. 相互運用性を維持するための文字列マーシャラー

文字列互換性マーシャラーを使用するには、以下の例のように **CompatibilityMarshaller** のインスタンスを **ConfigurationBuilder** オブジェクトの **Marshaller()** メソッドに渡します。

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.Marshaller(new CompatibilityMarshaller());
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build(),
true);
IRemoteCache<String, String> cache = cacheManager.GetCache<String, String>
();
[....]
cache.Put("key", "value");
[... ]
cache.Get("key");
[... ]
```

**注記**

非文字列キー/値を格納または取得しようとする、**HotRodClientException** が発生します。

16.10. HOT ROD NODE.JS クライアント**16.10.1. Hot Rod Node.js クライアント**

Hot Rod Node.js クライアントは非同期型のイベント駆動クライアントで、Node.js ユーザーは Red Hat JBoss Data Grid サーバーとの通信が可能です。このクライアントは、スクリプトの実行および格納、キャッシュリスナーの利用、完全なクラスタポートロジーの受信など、Java クライアントの多くの機能をサポートします。

非同期操作の結果は **Promise** インスタンスで提供されるため、クライアントは簡単に複数の呼び出しをチェーンでき、エラーの処理を一元化できます。

16.10.2. Hot Rod Node.js クライアントのインストール

Hot Rod Node.js クライアントは別のディストリビューションに含まれているため、Red Hat JBoss Data Grid とは別にダウンロードする必要があります。以下の手順にしたがってこのクライアントをインストールします。

手順: Hot Rod Node.js クライアントのインストール

1. Red Hat カスタマーポータルから **jboss-datagrid-7.2.1-nodejs-client.zip** をダウンロードします。
2. ダウンロードしたアーカイブを展開します。
3. 以下のコマンドのように、**npm** を使用して tarball をインストールします。

```
npm install /path/to/jboss-datagrid-7.2.0-nodejs-client/infinispan-7.2.0-Final-redhat-9.tgz
```

16.10.3. Hot Rod Node.js の要件

Hot Rod Node.js を使用するには、以下の要件を満たす必要があります。

- Node.js バージョン 0.10 以上。
- Red Hat JBoss Data Grid サーバーインスタンス 7.0.0 以上。

16.10.4. Hot Rod Node.js の基本機能

以下の例は Red Hat JBoss Data Grid サーバーに接続し、データの配置や取得などの基本操作を実行する方法を示しています。以下の例では、Red Hat JBoss Data Grid サーバーがデフォルトの **localhost:11222** で利用できることを前提としています。

```
var infinispan = require('infinispan');

// Obtain a connection to the JBoss Data Grid server
// As no cache is specified all operations will occur on the 'default'
// cache
var connected = infinispan.client({port: 11222, host: '127.0.0.1'});

connected.then(function (client) {

    // Attempt to put a value in the cache.
    var clientPut = client.put('key', 'value');

    // Retrieve the value just placed
    var clientGet = clientPut.then(
        function() { return client.get('key'); });

    // Print out the value that was retrieved
    var showGet = clientGet.then(
        function(value) { console.log('get(key)= ' + value); });

    // Disconnect from the server
    return showGet.finally(
        function() { return client.disconnect(); });
}).catch(function(error) {

    // Log any errors received
    console.log("Got error: " + error.message);

});
```

名前付きキャッシュへの接続

特定のキャッシュに接続するには、以下の例のように Red Hat JBoss Data Grid サーバーインスタンスの場所を指定するときに **cacheName** 属性を定義します。

```
var infinispan = require('infinispan');
```

```
// Obtain a connection to the JBoss Data Grid server
// and connect to namedCache
var connected = infinispn.client(
  {port: 11222, host: '127.0.0.1'}, {cacheName: 'namedCache'});

connected.then(function (client) {

  // Log the result of the connection
  console.log('Connected to `namedCache`');

  // Disconnect from the server
  return client.disconnect();

}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});
```

データセットの使用

putAll および **getAll** メソッドを使用すると、単一のエントリーだけでなく、データセットを配置または取得できます。この操作の例を以下に示します。

```
var infinispn = require('infinispn');

// Obtain a connection to the JBoss Data Grid server
// As no cache is specified all operations will occur on the 'default'
// cache
var connected = infinispn.client({port: 11222, host: '127.0.0.1'});

connected.then(function (client) {
  var data = [
    {key: 'multi1', value: 'v1'},
    {key: 'multi2', value: 'v2'},
    {key: 'multi3', value: 'v3'}];

  // Place all of the key/value pairs in the cache
  var clientPutAll = client.putAll(data);

  // Obtain the values for two of the keys
  var clientGetAll = clientPutAll.then(
    function() { return client.getAll(['multi2', 'multi3']); });

  // Print out the values obtained.
  var showGetAll = clientGetAll.then(
    function(entries) {
      console.log('getAll(multi2, multi3)=%s', JSON.stringify(entries));
    }
  );

  // Obtain an iterator for the cache
  var clientIterator = showGetAll.then(
    function() { return client.iterator(1); });
```

```

// Iterate over the entries in the cache, printing the values
var showIterated = clientIterator.then(
  function(it) {
    function loop(promise, fn) {
      // Simple recursive loop over iterator's next() call
      return promise.then(fn).then(function (entry) {
        return !entry.done ? loop(it.next(), fn) : entry.value;
      });
    }

    return loop(it.next(), function (entry) {
      console.log('iterator.next()=' + JSON.stringify(entry));
      return entry;
    });
  }
);

// Clear the cache of all values
var clientClear = showIterated.then(
  function() { return client.clear(); });

// Disconnect from the server
return clientClear.finally(
  function() { return client.disconnect(); });

}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});

```

16.10.5. Hot Rod Node.js の条件付き操作

Hot Rod プロトコルは、キーに関連する各値だけでなく、メタデータも格納します。**getWithMetadata** は値とキーに関連するメタデータを取得します。

以下はメタデータを利用する例になります。

```

var infinispn = require('infinispn');

// Obtain a connection to the JBoss Data Grid server
// As no cache is specified all operations will occur on the 'default'
// cache
var connected = infinispn.client({port: 11222, host: '127.0.0.1'});

connected.then(function (client) {

  // Attempt to put a value in the cache if it does not exist
  var clientPut = client.putIfAbsent('cond', 'v0');

  // Print out the result of the put operation
  var showPut = clientPut.then(
    function(success) { console.log(':putIfAbsent(cond)= ' + success);
  });
});

```



```

// Replace the value in the cache
var clientReplace = showPut.then(
  function() { return client.replace('cond', 'v1'); } );

// Print out the result of the replace
var showReplace = clientReplace.then(
  function(success) { console.log('replace(cond)= ' + success); });

// Obtain the value and metadata
var clientGetMetaForReplace = showReplace.then(
  function() { return client.getWithMetadata('cond'); });

// Replace the value only if the version matches
var clientReplaceWithVersion = clientGetMetaForReplace.then(
  function(entry) {
    console.log('getWithMetadata(cond)= ' + JSON.stringify(entry));
    return client.replaceWithVersion('cond', 'v2', entry.version);
  }
);

// Print out the result of the previous replace
var showReplaceWithVersion = clientReplaceWithVersion.then(
  function(success) { console.log('replaceWithVersion(cond)= ' +
success); });

// Obtain the value and metadata
var clientGetMetaForRemove = showReplaceWithVersion.then(
  function() { return client.getWithMetadata('cond'); });

// Remove the value only if the version matches
var clientRemoveWithVersion = clientGetMetaForRemove.then(
  function(entry) {
    console.log('getWithMetadata(cond)= ' + JSON.stringify(entry));
    return client.removeWithVersion('cond', entry.version);
  }
);

// Print out the result of the previous remove
var showRemoveWithVersion = clientRemoveWithVersion.then(
  function(success) { console.log('removeWithVersion(cond)= ' +
success)); });

// Disconnect from the server
return showRemoveWithVersion.finally(
  function() { return client.disconnect(); });

}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});

```

16.10.6. Hot Rod Node.js のデータセット

クライアントは接続を定義するときに複数のサーバーアドレスを指定できます。複数のサーバーが定義されると、正常にノードへの接続が確立されるまで各サーバーをループします。この設定の例を以下に示します。

```
var infinispn = require('infinispn');

// Accepts multiple addresses and fails over if connection not possible
var connected = infinispn.client(
  [{port: 99999, host: '127.0.0.1'}, {port: 11222, host: '127.0.0.1'}]);

connected.then(function (client) {

  // Obtain a list of all members in the cluster
  var members = client.getTopologyInfo().getMembers();

  // Print out the list of members
  console.log('Connected to: ' + JSON.stringify(members));

  // Disconnect from the server
  return client.disconnect();

}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});
```

16.10.7. Hot Rod Node.js のリモートイベント

Hot Rod Node.js クライアントはリモートキャッシュリスナーをサポートします。これらのリスナーは **addListener** メソッドを使用して追加できます。このメソッドはイベントタイプ (**create**、**modify**、**remove**、および **expiry**) と関数コールバックをパラメーターとして取ります。リモートイベントリスナーの詳細は、「[リモートイベントリスナー \(Hot Rod\)](#)」を参照してください。例を以下に示します。

```
var infinispn = require('infinispn');
var Promise = require('promise');

var connected = infinispn.client({port: 11222, host: '127.0.0.1'});

connected.then(function (client) {

  var clientAddListenerCreate = client.addListener(
    'create', function(key) { console.log('[Event] Created key: ' + key);
  });

  var clientAddListeners = clientAddListenerCreate.then(
    function(listenerId) {
      // Multiple callbacks can be associated with a single client-side
      // listener.
      // This is achieved by registering listeners with the same listener
      // id
      // as shown in the example below.
      var clientAddListenerModify = client.addListener(
```

```

        'modify', function(key) { console.log('[Event] Modified key: ' +
key); },
        {listenerId: listenerId});

    var clientAddListenerRemove = client.addListener(
        'remove', function(key) { console.log('[Event] Removed key: ' +
key); },
        {listenerId: listenerId});

    return Promise.all([clientAddListenerModify,
clientAddListenerRemove]);
    });

    var clientCreate = clientAddListeners.then(
        function() { return client.putIfAbsent('eventful', 'v0'); });

    var clientModify = clientCreate.then(
        function() { return client.replace('eventful', 'v1'); });

    var clientRemove = clientModify.then(
        function() { return client.remove('eventful'); });

    var clientRemoveListener =
        Promise.all([clientAddListenerCreate, clientRemove]).then(
            function(values) {
                var listenerId = values[0];
                return client.removeListener(listenerId);
            });

    return clientRemoveListener.finally(
        function() { return client.disconnect(); });

}).catch(function(error) {

    console.log("Got error: " + error.message);

});

```

16.10.8. クラスターと動作する Hot Rod Node.js

複数の Red Hat JBoss Data Grid サーバーインスタンスをクラスター化してフェイルオーバーおよびスケールアップの機能を提供できます。クラスターとの動作は単一インスタンスを使用した場合と大変似ていますが、以下を考慮する必要があります。

- クライアントは、クラスターの大きさに関わらず、サーバークラスター全体の情報を受け取るために単一のサーバーのアドレスのみを認識する必要があります。
- 分散キャッシュでは、サーバーによって使用される一貫した同じハッシュアルゴリズムを使用してキーベースの操作がクラスターでルーティングされます。そのため、追加のネットワークホップを必要とせずに、クライアントはキーの場所を特定できます。
- 分散キャッシュでは、複数キー操作またはキーなし操作はラウンドロビン形式でルーティングされます。

- レプリケートされたキャッシュおよびインバリデートされたキャッシュでは、キーベース、複数キー、またはキーなしであるかに関わらず、すべての操作はラウンドロビン形式でルーティングされます。

すべてのルーティングとフェイルオーバーはクライアントに透過的であるため、クラスターに対して実行された操作は上記で実行されたコードサンプルと同じように見えます。

以下の例を使用するとクラスターポロジータを取得できます。

```
var infinispn = require('infinispn');

var connected = infinispn.client({port: 11322, host: '127.0.0.1'});

connected.then(function (client) {

    var members = client.getTopologyInfo().getMembers();

    // Should show all expected cluster members
    console.log('Connected to: ' + JSON.stringify(members));

    // Add your own operations here...

    return client.disconnect();

}).catch(function(error) {

    // Log any errors received
    console.log("Got error: " + error.message);

});
```

16.10.9. サイトと動作する Hot Rod Node.js

複数の Red Hat JBoss Data Grid サーバークラスターをデプロイし、各クラスターが異なるサイトに属するようにすることができます。このようなデプロイメントは、あるクラスターから別のクラスター(地理的に異なる場所にある可能性がある)にデータをバックアップできるようにするために行われます。Node.js クライアント実装はクラスター内のノードの間でフェイルオーバーでき、元のクラスターが応答しなくなった場合は全体を他のクラスターにフェイルオーバーできます。クラスター間でのフェイルオーバーを可能にするには、すべての Red Hat JBoss Data Grid サーバーがクロスデータセンターレプリケーションで設定されている必要があります。この手順は、Red Hat JBoss Data Grid の『[Administration and Configuration Guide](#)』を参照してください。

フェイルオーバーが発生した場合、クライアントの代替クラスターへの接続は、例外が発生して代替クラスターが利用できなくなるまで維持されます。利用できなくなると、元のサーバー設定を含む定義された他のクラスターを試行します。

クロスデータセンターレプリケーションがサーバー上で設定されたら、クライアントは代替クラスターの設定を提供し、設定した各クラスターに対して 1 つ以上のホスト/ポートペアの詳細情報を指定する必要があります。以下に例を示します。

```
var connected = infinispn.client({port: 11322, host: '127.0.0.1'},
{
  clusters: [
    {
      name: 'LON',
```

```

        servers: [{port: 1234, host: 'LONA1'}]
    },
    {
        name: 'NYC',
        servers: [{port: 2345, host: 'NYCB1'}, {port: 3456, host:
'NYCB2'}]
    }
]
});

```

16.10.9.1. 手動クラスター切り替え

自動サイトフェイルオーバーの他にも、Node.js クライアントは以下のメソッドのいずれかを呼び出してサイトクラスターの手動で切り替えることができます。

- **switchToCluster(clusterName)** - クライアントを事前定義されたクラスター名に強制的に切り替えます。
- **switchToDefaultCluster()** - クライアントをクライアント設定で定義された最初のサーバーに強制的に切り替えます。

たとえば、手動で NYC クラスターへ切り替える場合は以下を使用できます。

```

var connected = infinispn.client({port: 11322, host: '127.0.0.1'},
{
    clusters: [
        {
            name: 'LON',
            servers: [{port: 1234, host: 'LONA1'}]
        },
        {
            name: 'NYC',
            servers: [{port: 2345, host: 'NYCB1'}, {port: 3456, host:
'NYCB2'}]
        }
    ]
});

connected.then(function (client) {

    var switchToB = client.getTopologyInfo().switchToCluster('NYC');
    [...]
});

```

16.11. HOT ROD C++ と HOT ROD JAVA クライアント間の相互運用性

Red Hat JBoss Data Grid は、構造化データにアクセスするために Hot Rod Java と Hot Rod C++ クライアント間の相互運用性を提供します。これは、Google の Protobuf 形式を使用してデータを構造化およびシリアルライズすることにより可能になります。

たとえば、言語間の相互運用性を使用すると、Hot Rod C++ クライアントが Protobuf を使用して構造化およびシリアルライズされた次の **Person** オブジェクトを記述し、Hot Rod Java クライアントが Protobuf として構造化された同じ **Person** オブジェクトを読み取ることができます。

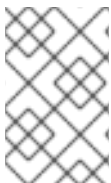
言語間の相互運用性の使用

```
package sample;
message Person {
    required int32 age = 1;
    required string name = 2;
}
```

C++ と Hot Rod Java クライアント間の相互運用性は基本データタイプ、文字列、バイトアレイにおいて完全にサポートされています。Protobuf と Protostream はこれらのタイプの相互運用性のために必要ありません。

16.12. サーバーと HOT ROD クライアントバージョン間の互換性

Hot Rod Java、Hot Rod C++、および Hot Rod C# などの Hot Rod クライアントは異なるバージョンの Red Hat JBoss Data Grid サーバーと互換性があります。このサーバーのバージョンは、複数の異なる Hot Rod クライアントと共に実行するために最新バージョンである必要があります。



注記

既知の問題を防ぐために、移行またはアップグレードの場合を除き、同じバージョンの Hot Rod クライアントおよび Red Hat JBoss Data Grid サーバーを使用することをお勧めします。

以下のシナリオを見てみましょう。

シナリオ 1: サーバーが Hot Rod クライアントよりも新しいバージョンで実行される。

以下の影響がクライアント側に及ぶことが考えられます。

- クライアントには、プロトコルの最新改良によるメリットはない。
- クライアントはサーバー側のバージョンで修正されている既知の問題に直面する可能性がある。
- クライアントは現在のバージョンと以前のバージョンで利用可能な機能のみを使用できる。

シナリオ 2: Hot Rod クライアントがサーバーよりも新しいバージョンで実行される。

Hot Rod クライアントが Red Hat JBoss Data Grid サーバーに接続される場合、その接続は例外エラーを出して拒否されます。クライアント側のプロパティ `infinispan.client.hotrod.protocol_version` を設定するか、`ConfigurationBuilder` の `protocolVersion(String version)` を使用すると、クライアントを既知のプロトコルバージョンにダウングレードできます。どちらかの方法でクライアントのバージョンをダウングレードすると、適切なバージョンが含まれる `String` が渡されるはずですが、この場合、クライアントはサーバーに接続できますが、そのバージョンの機能に制限されます。このプロトコルのバージョンでサポートされないコマンドは機能せず、例外が発生します。さらに、この場合のトポロジー情報は効率的でない可能性があります。

クライアントの Hot Rod プロトコルバージョンのダウングレード

以下のコード例は、`protocolVersion(String version)` メソッドを使用してバージョンをダウングレードする方法を示しています。

```
Configuration config = new ConfigurationBuilder()
```

```
[...]
.protocolVersion("2.2")
.build();
```



注記
Red Hat サポートの指示を受けずにこの方法を使用することは推奨されません。

以下の表は、異なる Hot Rod クライアントとサーバーのバージョン間の互換性についての詳細情報です。

表16.78 Hot Rod プロトコルとサーバーの互換性

Red Hat JBoss Data Grid サーバーのバージョン	Hot Rod プロトコルのバージョン
Red Hat JBoss Data Grid 7.2.0	Hot Rod 2.5 以上
Red Hat JBoss Data Grid 7.1.0	Hot Rod 2.5 以上
Red Hat JBoss Data Grid 7.0.0	Hot Rod 2.5 以上

パート II. RED HAT JBOSS DATA GRID での INFINISPAN クエ リーの作成および使用

第17章 INFINISPAN クエリーの使用

17.1. はじめに

Red Hat JBoss Data Grid のライブラリーモードで Querying API を使用すると、キーではなく値のプロパティを使用してグリッドでエントリーを検索できます。以下のような機能が提供されます。

- キーワード、範囲、ファジー、ワイルドカード、およびフレーズのクエリー
- クエリーの組み合わせ
- クエリー結果のソート、フィルター、およびページ編集

Apache Lucene および Hibernate Search をベースとしたこの API は Red Hat JBoss Data Grid でサポートされます。さらに、Red Hat JBoss Data Grid はインデックスを使用しない検索とインデックスを使用する検索の両方を許可する代替のメカニズムを提供します。詳細は「[Infinispan Query DSL](#)」を参照してください。

クエリーの有効化

リモートクライアントサーバーモードでは、Querying API はデフォルトで有効になっています。ライブラリーモードで有効にする方法は Red Hat JBoss Data Grid の『[Administration and Configuration Guide](#)』を参照してください。

17.2. RED HAT JBOSS DATA GRID のクエリーのインストール

Red Hat JBoss Data Grid では、クエリーの実行に必要な JAR ファイルは Red Hat JBoss Data Grid のライブラリーモードおよびリモートクライアントサーバーモードのダウンロード内にパッケージ化されています。

Red Hat JBoss Data Grid の ダウンロードおよびインストールに関する詳細は、『[Getting Started Guide](#)』の「[Download and Install JBoss Data Grid](#)」の章を参照してください。

さらに、以下の Maven 依存関係を定義する必要があります。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-embedded-query</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```



警告

Infinispan Query API は Hibernate Search と Lucene API を直接公開し、**infinispan-embedded-query.jar** ファイル内に埋め込むことはできません。他のバージョンの Hibernate Search と Lucene が **infinispan-embedded-query** と同じデプロイメントに含まれないようにしてください。これらが含まれると、クラスパスの競合が発生する原因となり、予期せぬ動作が実行されます。

17.3. RED HAT JBOSS DATA GRID でのクエリー

17.3.1. Hibernate Search およびクエリーモジュール

Red Hat JBoss Data Grid では、ユーザーは特定の項目を保存されたデータセット全体でクエリーできます。アプリケーションは常に特定のキーを認識できるわけではありませんが、クエリーモジュールを使用すると値の異なる部分をクエリーできます。

プロパティの一部を基にしてオブジェクトを検索することができます。例を以下に示します。

- 赤い車をすべて読み出し (メタデータの完全一致)。
- 特定トピックに関するすべての本を検索 (完全テキスト検索および関連度スコア)

データの完全一致は、MapReduce 関数で実装することもできますが、完全テキストおよび関連度ベースのスコアはクエリーモジュールを介してのみ実行できます。



警告

現在、クエリー機能はリッチドメインオブジェクトを対象にしており、プリミティブ値は現在クエリーではサポートされていません。'

17.3.2. Apache Lucene およびクエリーモジュール

分散されたグリッドに保存されたデータセット全体でクエリーを実行するため、Red Hat JBoss Data Grid は Apache Lucene のインデックス化ツールと Hibernate Search の機能を利用します。

- Apache Lucene はドキュメントインデックス化ツールおよび検索エンジンです。JBoss Data Grid は Apache Lucene 5.5.1 を使用します。
- JBoss Data Grid のクエリーモジュールは、Hibernate Search をベースとしたツールキットです。Java オブジェクトを、Apache Lucene によってインデックス化およびクエリー可能なドキュメントと似た形式に縮小します。

JBoss Data Grid では、クエリーモジュールは Hibernate Search のインデックス化アノテーションが付けられた値をインデックス化し、Apache Lucene を基にインデックスを更新します。

Hibernate Search は、データに保存されたエントリーの変更を阻止し、対応するインデックス化操作を生成します。

17.4. インデックス化

17.4.1. インデックス化

インデックス化が設定されると、クエリーモジュールは追加、更新、または削除されたキャッシュエントリーを透過的にインデックス化します。インデックスはクエリーのパフォーマンスを向上しますが、更新中は追加のオーバーヘッドが発生します。インデックスを使用しないクエリーに関する詳細は、「[Infinispan Query DSL](#)」を参照してください。

グリッドにすでに存在するデータに対しては、最初の Lucene インデックスを作成します。関連するプロパティとアノテーションが追加された後、「[インデックスの再構築](#)」に示された最初のバッチインデックスをトリガーします。

17.4.2. トランザクションおよび非トランザクションキャッシュによるインデックス化

Red Hat JBoss Data Grid では、トランザクションとインデックス化の関係は次のようになります。

- トランザクションキャッシュである場合、コミット処理の後にリスナーを使用して (コミット後のリスナー) インデックスの更新が適用されます。インデックスの更新に失敗しても書き込みに失敗しません。
- 非トランザクションキャッシュである場合、イベント完了後に動作するリスナー (イベント後のリスナー) を使用してインデックスの更新が適用されます。インデックスの更新に失敗しても書き込みに失敗しません。

17.4.3. プログラムを使用したインデックス化設定

インデックス化は、**XML** 設定ファイルを使用せずにプログラムを使用して設定できます。

この例では、Red Hat JBoss Data Grid はプログラムによって起動されます。また、グリッドに格納され、クラスにアノテーションを付けずに 2 つのプロパティを使用して検索可能な **Author** オブジェクトもマップします。

プログラムを使用したインデックス化設定

```
SearchMapping mapping = new SearchMapping();
mapping.entity(Author.class).indexed().providedId()
    .property("name", ElementType.METHOD).field()
    .property("surname", ElementType.METHOD).field();

Properties properties = new Properties();
properties.put(org.hibernate.search.cfg.Environment.MODEL_MAPPING,
mapping);
properties.put("[other.options]", "[...]");

Configuration infinispanConfiguration = new ConfigurationBuilder()
    .indexing()
    .index(Index.LOCAL)
    .withProperties(properties)
    .build();

DefaultCacheManager cacheManager = new
DefaultCacheManager(infinispanConfiguration);

Cache<Long, Author> cache = cacheManager.getCache();
SearchManager sm = Search.getSearchManager(cache);

Author author = new Author(1, "FirstName", "Surname");
cache.put(author.getId(), author);

QueryBuilder qb = sm.buildQueryBuilderForClass(Author.class).get();
Query q =
```

```
qb.keyword().onField("name").matching("FirstName").createQuery();
CacheQuery cq = sm.getQuery(q, Author.class);
Assert.assertEquals(cq.getResultSize(), 1);
```

17.4.4. インデックスの再構築

Lucene インデックスは、キャッシュ内のデータストアから再構築することによって、必要な場合に再構築できます。

インデックスは以下の場合に再構築する必要があります。

- タイプでインデックス化されている内容の定義が変更されている。
- **Analyser** などのインデックスの定義方法に影響を与えるパラメーターが変更されている。
- インデックスがシステム管理者のエラーにより、破壊または破損している。

インデックスを再構築するには、以下のように **MassIndexer** への参照を取得し、開始します。

```
SearchManager searchManager = Search.getSearchManager(cache);
searchManager.getMassIndexer().start();
```

この操作はグリッド内のすべてのデータを再処理するため、時間がかかる場合があります。

17.5. 検索

検索を実行するには、Lucene クエリーを作成します (「[Lucene ベースのクエリー API を使用した Lucene クエリーの構築](#)」を参照)。クエリーを **org.infinispan.query.CacheQuery** でラップして Lucene ベースの API から必要な機能を取得します。以下のコードはインデックス化されたフィールドに対してクエリーを準備します。このコードを実行すると、**Book** のリストが返されます。

Infinispan クエリーを使用した検索の作成および実行

```
QueryBuilder qb =
Search.getSearchManager(cache).buildQueryBuilderForClass(Book.class).get()
;

org.apache.lucene.search.Query query = qb
    .keyword()
    .onFields("title", "author")
    .matching("Java rocks!")
    .createQuery();

// wrap Lucene query in a org.infinispan.query.CacheQuery
CacheQuery cacheQuery = Search.getSearchManager(cache).getQuery(query);

List list = cacheQuery.list();
```

第18章 オブジェクトのアノテーション付けおよびクエリー

18.1. オブジェクトのアノテーション付けおよびクエリー

インデックス化が有効になったら、Red Hat JBoss Data Grid に保存されたカスタムオブジェクトに適切なアノテーションを割り当てる必要があります。

基本的な要件として、インデックス化されるすべてのオブジェクトに以下のアノテーションを付ける必要があります。

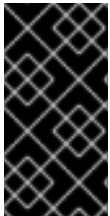
- **@Indexed**

さらに、検索されるオブジェクト内のすべてのフィールドに **@Field** アノテーションを付ける必要があります。

@Field アノテーションをオブジェクトに追加

```
@Indexed
public class Person implements Serializable {
    @Field(store = Store.YES)
    private String name;
    @Field(store = Store.YES)
    private String description;
    @Field(store = Store.YES)
    private int age;
}
```

その他のアノテーションおよびオプションについては、「[ドメインオブジェクトのインデックス構造へのマッピング](#)」を参照してください。



重要

JBoss EAP モジュールを JBoss Data Grid と使用し、ドメインモデルをモジュールとして使用する場合、スロット 7.2 の `org.infinispan.query` 依存関係を **module.xml** ファイルに追加します。`org.infinispan.query` 依存関係がないと、カスタムアノテーションはクエリーによって取得されず、エラーが発生します。

18.2. アノテーションによるトランスフォーマーの登録

各値のキーもインデックス化する必要があり、キーインスタンスを文字列に変換する必要があります。

Red Hat JBoss Data Grid には共通のプリミティブをエンコードするためにデフォルトのトランスメーションルーティングが一部含まれていますが、カスタムキーを使用するには **org.infinispan.query.Transformer** の実装を提供する必要があります。

以下の例は、**org.infinispan.query.Transformer** を使用してキータイプにアノテーションを付ける方法を示しています。

キータイプのアノテーション付け

```
@Transformable(transformer = CustomTransformer.class)
public class CustomKey {
```

```

    }

    public class CustomTransformer implements Transformer {
        @Override
        public Object fromString(String s) {
            return new CustomKey(...);
        }

        @Override
        public String toString(Object customType) {
            CustomKey ck = (CustomKey) customType;
            return ck.toString();
        }
    }
}

```

2つのメソッドは1対1の対応関係 (Biunique correspondence) を実装する必要があります。

たとえば、オブジェクト A は以下が true である必要があります。

1 対 1 の対応関係 (Biunique correspondence)

```
A.equals(transformer.fromString(transformer.toString(A)));
```

これは、トランスフォーマーはタイプ A のオブジェクトの適切なトランスフォーマー実装であることを前提とします。

18.3. クエリーの例

以下は、Red Hat JBoss Data Grid でクエリーを設定および実行する方法の例になります。

この例では、**Person** オブジェクトは以下を使用してアノテーションが付けられています。

Person オブジェクトのアノテーション付け

```

@Indexed
public class Person implements Serializable {
    @Field(store = Store.YES)
    private String name;
    @Field
    private String description;
    @Field(store = Store.YES)
    private int age;
}

```

複数の **Person** オブジェクトが JBoss Data Grid に保存されていることを前提とした場合、クエリーを使用してこれらのオブジェクトを検索できます。以下のコードは **SearchManager** および **QueryBuilder** インスタンスを作成します。

SearchManager および QueryBuilder の作成

```

SearchManager manager = Search.getSearchManager(cache);
QueryBuilder builder = manager.buildQueryBuilderForClass(Person.class)
    .get();
Query luceneQuery = builder.keyword()

```

```
.onField("name")  
.matching("FirstName")  
.createQuery();
```

The **SearchManager** および **QueryBuilder** は **Lucene** クエリーの構築に使用されます。**Lucene** クエリーは **CacheQuery** インスタンスを取得するために **SearchManager** に渡されます。

クエリーの実行

```
CacheQuery query = manager.getQuery(luceneQuery);  
List<Object> results = query.list();  
for (Object result : results) {  
    System.out.println("Found " + result);  
}
```

この **CacheQuery** インスタンスにはクエリーの結果が含まれ、リストの作成やクエリーの繰り返しに使用できます。

第19章 ドメインオブジェクトのインデックス構造へのマッピング

19.1. 基本のマッピング

19.1.1. 基本のマッピング

Red Hat JBoss Data Grid では、すべての **@Indexed** オブジェクトの識別子は値の保存に使用されるキーになります。キーがインデックス化される方法は、**@Transformable**、**@ProvidedId**、カスタムタイプ、およびカスタム **FieldBridge** 実装の組み合わせを使用するとカスタマイズできます。

@DocumentId 識別子は JBoss Data Grid の値には適用されません。

Lucene ベースの Query API は、以下の一般的なアノテーションを使用してエンティティをマップします。

- **@Indexed**
- **@Field**
- **@NumericField**

19.1.2. @Indexed

@Indexed アノテーションはキャッシュされたエントリーをインデックス可能であると宣言します。**@Indexed** アノテーションが付いてないエントリーはすべて無視されます。

@Indexed でクラスをインデックス可能にする

```
@Indexed
public class Essay {
}
```

任意で **@Indexed** アノテーションの **index** 属性を指定して、インデックスのデフォルト名を変更することもできます。

19.1.3. @Field

エンティティの各プロパティまたは属性をインデックス化できます。プロパティや属性はデフォルトではアノテーションが付けられていないため、インデックス化の処理では無視されます。**@Field** アノテーションはプロパティをインデックス化されたプロパティとして宣言し、以下の属性を1つ以上設定するとインデックス化処理の内容を設定できます。

name

この名前プロパティが Lucene Document に保存されます。デフォルトでは JavaBeans の慣例に準拠し、この属性はプロパティ名と同じになります。

store

プロパティが Lucene インデックスに保存されるかどうかを指定します。プロパティが保存されると、Lucene Document から元の値で読み出すことができます。これは、要素のインデックス化の有無には関係ありません。有効なオプションは次のとおりです。

- **Store.YES**: より多くのインデックス領域を消費しますが、射影を許可します。「[射影](#)」を参照してください。

- **Store.COMPRESS**: プロパティを圧縮して格納します。この属性は CPU をより消費します。
- **Store.NO**: ストレージはありません。これは store 属性のデフォルト設定になります。

index

プロパティがインデックス化されたかどうかを示します。以下の値を適用できます。

- **Index.NO**: インデックス化は適用されません。クエリーで見つけることができません。この設定は検索可能である必要がなく、射影可能なプロパティに使用されます。
- **Index.YES**: 要素はインデックス化され検索可能です。これは index 属性のデフォルト設定になります。

analyze

プロパティが分析されるかどうかを判断します。analyze 属性は、そのコンテンツでプロパティを検索できるようにします。たとえば、テキストフィールドを分析する必要がある場合でも日付フィールドは分析する必要がない場合があります。以下を使用して analyze 属性を有効または無効にします。

- **Analyze.YES**
- **Analyze.NO**

analyze 属性はデフォルトで有効になっています。**Analyze.YES** 設定では、**Index.YES** 属性でプロパティがインデックス化される必要があります。

以下の属性はソートに使用され、分析してはいけません。

norms

インデックス時間のブースティング情報を格納するかどうかを判断します。有効な設定は次のとおりです。

- **Norms.YES**
- **Norms.NO**

この属性のデフォルトは **Norms.YES** です。norms を無効にするとメモリーを節約できますが、インデックス時のブースティング情報は利用できません。

termVector

単語 (term) と出現頻度 (frequency) のペアを示します。この属性を使用すると、インデックス化中にドキュメント内で term vector を保存できます。デフォルト値は **TermVector.NO** です。この属性に使用できる設定は次のとおりです。

- **TermVector.YES**: 各ドキュメントに term vector を保存します。これにより、同期された 2 つの配列が生成されます (1 つの配列には document term が含まれ、もう 1 つの配列には term の頻度が含まれます)。
- **TermVector.NO**: term vector を保存しません。

- **TermVector.WITH_OFFSETS**: term vector およびトークンオフセット情報を保存します。これは、**TermVector.YES** と同じですが、言葉の開始および終了オフセット位置情報が含まれます。
- **TermVector.WITH_POSITIONS**: term vector およびトークン位置情報を保存します。これは、**TermVector.YES** と同じですが、ドキュメントで言葉が発生する順序位置 (ordinal position) が含まれます。
- **TermVector.WITH_POSITION_OFFSETS**: term vector、トークン位置、およびオフセット情報を保存します。これは **YES**、**WITH_OFFSETS**、および **WITH_POSITIONS** の組み合わせです。

indexNullAs

この属性は null プロパティの置換値を提供します。この値は以下の形式要件に準拠する必要があります。

- 文字列の値の形式要件はありません。
- 数値にはフィールドタイプに応じて **Double.parseDouble()**、**Integer.parseInt()**、およびその他のプリミティブ解析メソッドが受け入れる形式を使用する必要があります。
- ブール値は **true** または **false** のいずれかである必要があります。
- **java.util.Calendar**、**java.util.Date**、および **java.time.*** は ISO-8601 形式を使用する必要があります。

19.1.4. @NumericField

@NumericField アノテーションは **@Field** と同じ範囲で指定できます。

@NumericField アノテーションは、Integer、Long、Float、および Double プロパティに対して指定できます。インデックス化するときにはトライ木の構造を使用して値がインデックス化されます。プロパティが数値のフィールドとしてインデックス化されると、効率的な範囲のクエリーおよびソートが有効になり、標準の **@Field** プロパティで同じクエリーを実行するよりも高速に順序付けできます。**@NumericField** アノテーションは以下の任意のパラメーターを許可します。

- **forField**: 数値としてインデックス化される関連する **@Field** の名前を指定します。プロパティに 1 つの **@Field** 宣言以外が含まれる場合は必須になります。
- **precisionStep**: トライ木構造がインデックスに保存される方法を変更します。**precisionSteps** を小さくするとディスク領域の使用量が増え、範囲およびソートクエリーが高速になります。値を大きくすると使用される領域が少なくなり、範囲クエリーのパフォーマンスが通常の **@Fields** の範囲クエリーに近くなります。**precisionStep** のデフォルト値は 4 です。

@NumericField は、**Double**、**Long**、**Integer**、および **Float** のみをサポートします。他の数値型に対して Lucene の同様の機能を使用できないため、他の型はデフォルトまたはカスタムの **TwoWayFieldBridge** を用いた文字列のエンコーディングを使用する必要があります。

カスタムの **NumericFieldBridge** を使用することもできます。カスタム設定は、型の変換中に近似が必要になります。以下は、カスタム **NumericFieldBridge** を定義する例になります。

カスタム NumericFieldBridge の定義

```

public class BigDecimalNumericFieldBridge extends NumericFieldBridge {
    private static final BigDecimal storeFactor = BigDecimal.valueOf(100);

    @Override
    public void set(String name,
                    Object value,
                    Document document,
                    LuceneOptions luceneOptions) {
        if (value != null) {
            BigDecimal decimalValue = (BigDecimal) value;
            Long indexedValue = Long.valueOf(
                decimalValue
                .multiply(storeFactor)
                .longValue());
            luceneOptions.addNumericFieldToDocument(name, indexedValue,
document);
        }
    }

    @Override
    public Object get(String name, Document document) {
        String fromLucene = document.get(name);
        BigDecimal storedBigDecimal = new BigDecimal(fromLucene);
        return storedBigDecimal.divide(storeFactor);
    }
}

```

19.2. プロパティを複数回マッピング

異なるインデックス化ストラテジーを使用して、プロパティをインデックスごとに複数回マッピングする必要があることがあります。たとえば、フィールドでクエリーをソートするには、フィールドが分析されていない必要があります。このプロパティで単語を基に検索し、ソートを行うには、このプロパティを2回インデックス化する必要があります (分析する回と分析しない回)。@Fields を使用してこの検索を実行できます。例を以下に示します。

@Fields を使用してプロパティを複数回マッピング

```

@Indexed(index = "Book")
public class Book {
    @Fields( {
        @Field,
        @Field(name = "summary_forSort", analyze = Analyze.NO, store =
Store.YES)
    })
    public String getSummary() {
        return summary;
    }
}

```

上記の例では、**summary** が2回インデックス化されます。**summary** としてトークン化される方法で1度インデックス化され、**summary_forSort** として非トークン化される方法でもう1度インデックス化されます。@Field が使用される場合、@Fields は2つの属性をサポートします。

- analyzer: プロパティごとではなくフィールドごとに `@Analyzer` アノテーションを定義します。
- bridge: プロパティごとではなくフィールドごとに `@FieldBridge` アノテーションを定義します。

19.3. 埋め込みオブジェクトおよび関連するオブジェクト

19.3.1. 埋め込みオブジェクトおよび関連するオブジェクト

関連するオブジェクトと埋め込みオブジェクトはルートエンティティインデックスの一部としてインデックス化できます。これにより、関連するオブジェクトのプロパティを基にしたエンティティの検索が可能になります。

19.3.2. 関連するオブジェクトのインデックス化

以下の例の目的は、Lucene クエリー **`address.city:Atlanta`** を使用して、関連する市がアトランタである場所を返すことです。場所フィールドは **`Place`** インデックスでインデックス化されます。**`Place`** インデックスドキュメントには以下のフィールドも含まれます。

- **`address.street`**
- **`address.city`**

これらのフィールドはクエリーも可能です。

関連付けのインデックス化

```
@Indexed
public class Place {

    @Field
    private String name;

    @IndexedEmbedded
    @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private Address address;
}

public class Address {

    @Field
    private String street;

    @Field
    private String city;

    @ContainedIn
    @OneToMany(mappedBy = "address")
    private Set<Place> places;
}
```

19.3.3. @IndexedEmbedded

@IndexedEmbedded を使用する場合、データは Lucene インデックスで非正規化されます。そのため、**Place** および **Address** オブジェクトの変更を反映するために Lucene ベースの Query API を更新し、インデックスを最新の状態にする必要があります。**Address** の変更時に Lucene ドキュメントが確実に更新されるようにするため、双方向の関係の逆側に **@ContainedIn** を付けます。**@ContainedIn** は、エンティティーを示す関連付けと埋め込みオブジェクトを示す関連付けの両方に使用できます。

@IndexedEmbedded アノテーションは入れ子にすることができます。属性には **@IndexedEmbedded** アノテーションを付けることができます。関連するクラスの属性はメインエントリーインデックスに追加されます。下記の例では、インデックスに以下のフィールドが含まれます。

- name
- address.street
- address.city
- address.ownedBy_name

入れ子の **@IndexedEmbedded** および **@ContainedIn** の使用法

```
@Indexed
public class Place {
    @Field
    private String name;

    @IndexedEmbedded
    @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private Address address;
}

public class Address {
    @Field
    private String street;

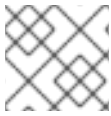
    @Field
    private String city;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_")
    private Owner ownedBy;

    @ContainedIn
    @OneToMany(mappedBy = "address")
    private Set<Place> places;
}

public class Owner {
    @Field
    private String name;
}
```

デフォルトの接頭辞は **propertyName.** で、従来のオブジェクトナビゲーションの慣例に従います。これをオーバーライドするには **ownedBy** プロパティーで示されたように **prefix** 属性を使用します。



注記

接頭辞を空の文字列に設定することはできません。

オブジェクトグラフにクラスの循環依存関係が含まれる場合、**depth** プロパティーが使用されます。たとえば、**Owner** が **Place** を示す場合がこれに該当します。予期された深さまたはオブジェクトグラフの境界に達した後、クエリーモジュールは属性の追加を停止します。自己参照クラスは循環依存関係の例の 1 つになります。この例では、深さが 1 に設定されているため、**Owner** の **@IndexedEmbedded** 属性はすべて無視されます。

オブジェクトの関連付けに **@IndexedEmbedded** を使用すると、Lucene のクエリー構文を使用してクエリーを表現できます。例を以下に示します。

- 名前に JBoss が含まれ、住所の市が Atlanta である場所を返す場合、Lucene クエリーでは次のようになります。

```
+name:jboss +address.city:atlanta
```

- 名前に JBoss が含まれ、所有者の名前に Joe が含まれる場所を返す場合、Lucene クエリーでは次のようになります。

```
+name:jboss +address.ownedBy_name:joe
```

この操作は関係結合操作と似ていますが、データが重複されません。そのままの状態では、Lucene インデックスには関連付けの観念がなく、結合操作は存在しません。完全テキストのインデックス速度や機能充実の利点を活かしながら正規化されたリレーショナルモデルを維持すると有用であることがあります。

関連したオブジェクトは **@Indexed** であることもあります。**@IndexedEmbedded** がエンティティーを示す場合、関連付けは指向性を持つ必要があり、逆側に **@ContainedIn** アノテーションを付ける必要があります。このアノテーションを付けないと、関連したエンティティーが更新されたときに Lucene ベースのクエリー API はルートインデックスを更新できません。この例では、関連する Address インスタンスが更新されると **Place** インデックスドキュメントが更新されます。

19.3.4. targetElement プロパティー

targetElement パラメーターを使用すると目的のオブジェクト型をオーバーライドできます。このメソッドは、**@IndexedEmbedded** アノテーションが付けられたオブジェクト型がデータグリッドおよび Lucene ベースの Query API が目的とするオブジェクト型ではない場合に使用されます。これは、実装の代わりにインターフェースが使用されると発生します。

@IndexedEmbedded の targetElement プロパティーの使用

```
@Indexed
public class Address {

    @Field
    private String street;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_", targetElement =
Owner.class)
    private Person ownedBy;

    ...
}
```

```

}

public class Owner implements Person { ... }

```

19.4. ブースティング

19.4.1. ブースティング

Lucene はブースティングを使用して特定のフィールドやドキュメントの重要度を上げます。Lucene はインデックス時のブースティングと検索時のブースティングを区別します。

19.4.2. 静的なインデックス時ブースティング

インデックス化されたクラスまたはプロパティの静的なブースト値を定義するには、**@Boost** アノテーションを使用します。このアノテーションは **@Field** 内で使用でき、メソッドやクラスレベルで直接指定することもできます。

下記の例は以下を意味します。

- Essay が検索リストの最上部に達する可能性が 1.7 倍になります。
- プロパティの **@Field.boost** および **@Boost** は累積的であるため、summary フィールドは 3.0 倍 (2 x 1.5) になり、ISBN フィールドよりも重要度が高くなります。
- text フィールドの重要度は ISBN フィールドの 1.2 倍になります。

@Boost の異なる使用方法

```

@Indexed
@Boost(1.7f)
public class Essay {

    @Field(name = "Abstract", store=Store.YES, boost = @Boost(2f))
    @Boost(1.5f)
    public String getSummary() { return summary; }

    @Field(boost = @Boost(1.2f))
    public String getText() { return text; }

    @Field
    public String getISBN() { return isbn; }

}

```

19.4.3. 動的なインデックス時ブースティング

@Boost アノテーションは静的ブースト係数を定義します。この係数は、実行時のインデックス化されたエンティティの状態とは関係ありません。ただし、場合によってはブースト係数がエンティティの実際の状態に依存することがあります。この場合、**@DynamicBoost** アノテーションをカスタム **BoostStrategy** とともに使用します。

@Boost および **@DynamicBoost** アノテーションはエンティティとの関連で使用することができ、定義されたブースト係数はすべて累積的です。**@DynamicBoost** をクラスまたはフィールドレベルで置くことができます。

以下の例では、インデックス化するとき使用される **BoostStrategy** インターフェースの実装として **VIPBoostStrategy** を指定することにより、動的なブーストがクラスレベルで定義されます。エンティティ全体が **defineBoost** メソッドに渡されるか、またはアノテーションが付けられたフィールド/プロパティの値のみが渡されるかはアノテーションの配置によります。渡されたオブジェクトを正しい型にキャストする必要があります。

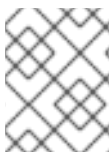
動的なブーストの例

```
public enum PersonType {
    NORMAL,
    VIP
}

@Indexed
@DynamicBoost(impl = VIPBoostStrategy.class)
public class Person {
    private PersonType type;
}

public class VIPBoostStrategy implements BoostStrategy {
    public float defineBoost(Object value) {
        Person person = (Person) value;
        if (person.getType().equals(PersonType.VIP)) {
            return 2.0f;
        }
        else {
            return 1.0f;
        }
    }
}
```

この例では、VIP のインデックス化された値すべての重要度は VIP でない値の 2 倍になります。



注記

指定の **BoostStrategy** 実装は、パブリックの引数のないコンストラクターを定義する必要があります。

19.5. 分析

19.5.1. 分析

19.5.2. デフォルトのアナライザーおよびクラス別のアナライザー

デフォルトのアナライザークラスはトークン化されたフィールドをインデックス化するために使用され、**default.analyzer** プロパティを用いて設定が可能です。このプロパティのデフォルト値は **org.apache.lucene.analysis.standard.StandardAnalyzer** です。

アナライザークラスはエンティティ、プロパティ、および **@Field** ごとに定義できます。これは、1つのプロパティから複数のフィールドがインデックス化されるときに便利です。

以下の例では、**EntityAnalyzer** を使用して、name などのトークン化されたプロパティがすべてインデックス化されます。例外は summary と body で、これらはそれぞれ **PropertyAnalyzer** と **FieldAnalyzer** によってインデックス化されます。

@Analyzer の異なる使用方法

```
@Indexed
@Analyzer(impl = EntityAnalyzer.class)
public class MyEntity {

    @Field
    private String name;

    @Field
    @Analyzer(impl = PropertyAnalyzer.class)
    private String summary;

    @Field(analyzer = @Analyzer(impl = FieldAnalyzer.class))
    private String body;
}
```



注記

1つのエンティティで複数のアナライザを使用しないようにしてください。特に **QueryParser** を使用する場合はクエリーの構築が複雑になり、結果の予測が難しくなります。フィールドのインデックス化とクエリーに同じアナライザを使用してください。

19.5.3. 名前付きのアナライザ

クエリーモジュールはアナライザ定義を使用して Analyzer 関数の複雑さに対応します。アナライザ定義は、複数の **@Analyzer** 宣言による再使用が可能で、以下が含まれます。

- 名前: 定義の参照に使用される一意の文字列。
- **CharFilters** のリスト: 各 **CharFilter** はトークン化の前に入力文字を事前処理します。**CharFilters** は文字を追加、変更、および削除できます。一般的な使用例の1つが文字の正規化です。
- **Tokenizer**: 入力ストリームを個別の単語にトークン化します。
- フィルターのリスト: 各フィルターは単語を削除または変更します。また、**Tokenizer** によって提供されたストリームに単語を追加することもあります。

Analyzer はこれらのコンポーネントを複数のタスクに分類し、以下の手順を使用して各コンポーネントの再使用やコンポーネントの柔軟な構築を実現します。

アナライザの処理

1. **CharFilters** は文字入力を処理します。

2. **Tokenizer** は文字入力をトークンに変換します。
3. トークンは **TokenFilters** によって処理されます。

Lucene ベースの Query API は、Solr アナライザーフレームワークを利用してこのインフラストラクチャーをサポートします。

19.5.4. アナライザーの定義

アナライザー定義は定義後に **@Analyzer** アノテーションによって再使用されます。

名前によるアナライザーの参照

```
@Indexed
@AnalyzerDef(name = "customanalyzer")
public class Team {

    @Field
    private String name;

    @Field
    private String location;

    @Field
    @Analyzer(definition = "customanalyzer")
    private String description;
}
```

@AnalyzerDef によって宣言されたアナライザーインスタンスは、**SearchFactory** の名前でも利用できます。これは、クエリーの構築時に役立ちます。

```
Analyzer analyzer =
    Search.getSearchManager(cache).getAnalyzer("customanalyzer")
```

クエリーの実行時、フィールドはフィールドのインデックス化に使用されたアナライザーを使用する必要があります。クエリーとインデックス化処理で同じトークンが再使用されます。

19.5.5. Solr の @AnalyzerDef

Maven を使用する場合、必要なすべての Apache Solr 依存関係はアーティファクト **org.hibernate:hibernate-search-analyzers** の依存関係として定義されるようになりました。以下の依存関係を追加します。

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search-analyzers</artifactId>
  <version>${version.hibernate.search}</version>
</dependency>
```

以下の例では、**CharFilter** はそのファクトリーによって定義されます。この例では、マッピング文字フィルターが使用され、マッピングファイルに指定されたルールを基にして入力のある文字列を置き換えます。この例では、専用の単語プロパティーファイルを読み取って **StopFilter** フィルターが構築されます。フィルターは大文字と小文字を区別しません。

@AnalyzerDef および Solr フレームワーク

1. CharFilter の設定

ファクトリーで **CharFilter** を定義します。この例では、マッピング **CharFilter** が使用され、マッピングファイルに指定されたルールを基に入力の文字を置き換えます。

```

@AnalyzerDef(name = "customanalyzer",
    charFilters = {
        @CharFilterDef(factory = MappingCharFilterFactory.class,
            params = {
                @Parameter(name = "mapping",
                    value =

"org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
            })
    },

```

2. Tokenizer の定義

Tokenizer は **StandardTokenizerFactory.class** を使用して定義されます。

```

@AnalyzerDef(name = "customanalyzer",
    charFilters = {
        @CharFilterDef(factory = MappingCharFilterFactory.class,
            params = {
                @Parameter(name = "mapping",
                    value =

"org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
            })
    },

    tokenizer = @TokenizerDef(factory =
StandardTokenizerFactory.class)

```

3. フィルターのリスト

ファクトリーでフィルターのリストを定義します。この例では、専用の単語プロパティファイルを読み取って **StopFilter** フィルターが構築されます。フィルターは大文字と小文字を区別しません。

```

@AnalyzerDef(name = "customanalyzer",
    charFilters = {
        @CharFilterDef(factory = MappingCharFilterFactory.class,
            params = {
                @Parameter(name = "mapping",
                    value =

"org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
            })
    },

    tokenizer = @TokenizerDef(factory =
StandardTokenizerFactory.class),
    filters = {

        @TokenFilterDef(factory =

```

```

        ISOLatin1AccentFilterFactory.class),
        @TokenFilterDef(factory = LowerCaseFilterFactory.class),
        @TokenFilterDef(factory = StopFilterFactory.class, params =
    {
        @Parameter(name = "words",
            value=
"org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
        @Parameter(name = "ignoreCase", value = "true")
    })
    })
    public class Team {
    }

```



注記

フィルターと **CharFilters** は、**@AnalyzerDef** アノテーションで定義された順に適用されます。

19.5.6. アナライザーリソースのロード

Tokenizers、**TokenFilters**、および **CharFilters** は **StopFilterFactory.class** または類義語フィルターを使用して、設定ファイルやメタデータファイルなどのリソースをロードできます。仮想マシンのデフォルトを明示的に指定するには、**resource_charset** パラメーターを追加します。

特定の文字セットを使用したプロパティファイルのロード

```

@AnalyzerDef(name = "customanalyzer",
    charFilters = {
        @CharFilterDef(factory = MappingCharFilterFactory.class, params =
    {
        @Parameter(name = "mapping",
            value =
"org/hibernate/search/test/analyzer/solr/mapping-
chars.properties")
    })
    },
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
        @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
        @TokenFilterDef(factory = LowerCaseFilterFactory.class),
        @TokenFilterDef(factory = StopFilterFactory.class, params = {
            @Parameter(name="words",
                value=
"org/hibernate/search/test/analyzer/solr/stoplist.properties"),
            @Parameter(name = "resource_charset", value = "UTF-16BE"),
            @Parameter(name = "ignoreCase", value = "true")
        })
    })
    })
    public class Team {
    }

```

19.5.7. 動的アナライザーの選択

クエリーモジュールは **@AnalyzerDiscriminator** アノテーションを使用して動的なアナライザーの選択を有効にします。

インデックス化するエンティティの現在の状態を基にアナライザーを選択できます。これは、特に多言語のアプリケーションで役に立ちます。たとえば、**BlogEntry** クラスを使用する場合、アナライザーはエントリの言語プロパティに依存することができます。このプロパティに応じて、テキストのインデックス化に適切な言語固有のステマーが選択されます。

Discriminator インターフェースの実装は既存のアナライザー定義の名前を返す必要がありますが、デフォルトのアナライザーがオーバーライドされない場合は `null` を返す必要があります。

下記の例は、言語パラメーターが **de** または **en** のいずれかで、**@AnalyzerDefs** で指定されることを前提とします。

@AnalyzerDiscriminator の設定

1. 動的なアナライザーの事前定義

@AnalyzerDiscriminator には、動的に使用されるすべてのアナライザーを **@AnalyzerDef** を使用して事前定義する必要があります。動的にアナライザーを選択するために、**@AnalyzerDiscriminator** アノテーションをクラスまたはエンティティの特定のプロパティに配置できます。**Discriminator** インターフェースの実装は、**@AnalyzerDiscriminatorimpl** パラメーターを使用して指定できます。

```
@Indexed
@AnalyzerDefs({
    @AnalyzerDef(name = "en",
        tokenizer = @TokenizerDef(factory =
StandardTokenizerFactory.class),
        filters = {
            @TokenFilterDef(factory = LowerCaseFilterFactory.class),
            @TokenFilterDef(factory =
EnglishPorterFilterFactory.class)
        }),
    @AnalyzerDef(name = "de",
        tokenizer = @TokenizerDef(factory =
StandardTokenizerFactory.class),
        filters = {
            @TokenFilterDef(factory = LowerCaseFilterFactory.class),
            @TokenFilterDef(factory = GermanStemFilterFactory.class)
        })
})
public class BlogEntry {

    @Field
    @AnalyzerDiscriminator(impl = LanguageDiscriminator.class)
    private String language;

    @Field
    private String text;

    private Set<BlogEntry> references;

    // standard getter/setter
}
```

2. Discriminator インターフェースの実装

Lucene ドキュメントにフィールドを追加するたびに呼び出される

getAnalyzerDefinitionName() メソッドを実装します。インデックス化されたエンティティもインターフェースメソッドに渡されます。

@AnalyzerDiscriminator がクラスレベルではなく、プロパティレベルに配置されると **value** パラメーターが設定されます。この例では、値はこのプロパティの現在の値を表します。

```
public class LanguageDiscriminator implements Discriminator {
    public String getAnalyzerDefinitionName(Object value, Object
entity, String field) {
        if (value == null || !(entity instanceof Article)) {
            return null;
        }
        return (String) value;
    }
}
```

19.5.8. アナライザーの読み出し

ステミングや音声的近似などを活用するため、ドメインモデルで複数のアナライザーが使用された場合にアナライザーを読み出すことができます。この場合、同じアナライザーを使用してクエリーを構築します。この代わりに、正しいアナライザーを自動的に選択する Lucene ベースの Query API を使用することもできます。詳細は「[Lucene クエリーの構築](#)」を参照してください。

エンティティのスコープ指定されたアナライザーは、Lucene プログラム API または Lucene クエリーパーサーのどちらかを使用して読み出すことができます。スコープ指定されたアナライザーは、インデックス化されたフィールドに応じて正しいアナライザーを適用します。個々のフィールドで動作する複数のアナライザーは 1 つのエンティティで定義できます。スコープ指定されたアナライザーは、これらのアナライザーをコンテキストを認識するアナライザーに統合します。

以下の例では、曲名は 2 つのフィールドでインデックス化されます。

- 標準のアナライザー: **title** フィールドで使用されます。
- ステミングアナライザー: **title_stemmed** フィールドで使用されます。

クエリーは、検索ファクトリーによって提供されるアナライザーを使用して、目的のフィールドに応じて適切なアナライザーを使用します。

フルテキストクエリーの構築時にスコープ指定されたアナライザーを使用

```
SearchManager manager = Search.getSearchManager(cache);

org.apache.lucene.queryparser.classic.QueryParser parser = new
QueryParser(
    org.apache.lucene.util.Version.LUCENE_5_5_1,
    "title",
    manager.getAnalyzer(Song.class)
);

org.apache.lucene.search.Query luceneQuery =
    parser.parse("title:sky Or title_stemmed:diamond");
```

```
// wrap Lucene query in a org.infinispan.query.CacheQuery
CacheQuery cacheQuery = manager.getQuery(luceneQuery, Song.class);

List result = cacheQuery.list();
//return the list of matching objects
```



注記

@AnalyzerDef によって定義されたアナライザーは、`searchManager.getAnalyzer(String)` を使用して定義名で取得することもできます。

19.5.9. 使用可能なアナライザー

Apache Solr と Lucene には、デフォルトの **CharFilters**、**tokenizers**、および **filters** が複数含まれています。**CharFilter**、**tokenizer**、および **filter** ファクトリーの完全リストは <http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters> を参照してください。以下の表は、**CharFilters**、**tokenizers**、および **filters** の一部を示しています。

表19.1 利用可能な CharFilters の例

ファクトリー	説明	Parameters	追加の依存関係
MappingCharFilterFactory	リソースファイルに指定されたマッピングを基に、1 つ以上の文字を置き換えます。	mapping: 以下の形式を使用したマッピングが含まれるリソースファイルを示します。 <pre>"á" => "a" "ñ" => "n" "ø" => "o"</pre>	なし
HTMLStripCharFilterFactory	標準の HTML タグを削除し、テキストは保持します。	なし	なし

表19.2 利用可能なトークナイザーの例

ファクトリー	説明	Parameters	追加の依存関係
StandardTokenizerFactory	Lucene の標準トークナイザーを使用します。	なし	なし
HTMLStripCharFilterFactory	標準の HTML タグを削除してテキストは保持し、StandardTokenizer へ渡します。	なし	solr-core

ファクトリー	説明	Parameters	追加の依存関係
PatternTokenizerFactory	指定の正規表現パターンでテキストを改行します。	pattern: トークン化に使用する正規表現。 group: トークンに抽出するパターングループを示します。	solr-core

表19.3 利用可能なフィルターの例

ファクトリー	説明	Parameters	追加の依存関係
StandardFilterFactory	頭字語からピリオドを削除し、単語からアポストロフィー (') を削除します。	なし	solr-core
LowerCaseFilterFactory	すべての言葉を小文字にします。	なし	solr-core
StopFilterFactory	ストップワードのリストと一致する言葉 (トークン) を削除します。	words: ストップワードが含まれるリソースファイルを示します。 ignoreCase: ストップワードを比較するときに case を無視する場合は true 、無視しない場合は false 。	solr-core
SnowballPorterFilterFactory	単語を指定言語の語根にします。たとえば、 <code>protect</code> 、 <code>protects</code> 、および <code>protection</code> はすべて同じ語根を持ちます。このようなフィルターを使用すると、関連する単語に一致する検索を実行できます。	language: デンマーク語、オランダ語、英語、フィンランド語、フランス語、ドイツ語、イタリア語、ノルウェー語、ポルトガル語、ロシア語、スペイン語、スウェーデン語など。	solr-core
ISOLatin1AccentFilterFactory	フランス語などの言語に使用されるアクセント記号を削除します。	なし	solr-core

ファクトリー	説明	Parameters	追加の依存関係
PhoneticFilterFactory	音声的に似ているトークンをトークンストリームに挿入します。	encoder: DoubleMetaphone、Metaphone、Soundex、または RefinedSoundex のいずれか。 inject: true を設定するとトークンをストリームに追加し、 false を設定すると既存のトークンを置き換えます。 maxCodeLength: 生成されるコードの最大長を設定します。 Metaphone および DoubleMetaphone エンコーディングのみに対してサポートされます。	solr-core および commons-codec
CollationKeyFilterFactory	各トークンを java.text.CollationKey に変換し、 CollationKey を IndexableBinaryStringTools でエンコードして索引語 (index term) として保存されるようにします。	custom 、 language 、 country 、 variant 、 strength 、 decomposition 。詳細は Lucene の CollationKeyFilter javadocs を参照してください。	solr-core および commons-io

19.6. ブリッジ

19.6.1. ブリッジ

エンティティをマッピングするとき、Lucene はすべてのインデックスフィールドを文字列として表します。**@Field** アノテーションが付けられたエンティティプロパティは、すべて文字列に変換され、インデックス化されます。ビルトインブリッジは Lucene ベースの Query API に対してプロパティを自動的に変換します。ブリッジをカスタマイズすると変換プロセスを制御することができます。

19.6.2. ビルトインブリッジ

Lucene ベースの Query API には、Java プロパティ型とそのフルテキスト表現との間のビルドインブリッジが含まれています。

null

デフォルトでは **null** 要素はインデックス化されません。Lucene は null 要素をサポートしませんが、場合によっては **null** の値を表すカスタムトークンを挿入すると便利ことがあります。詳細は「[@Field](#)」を参照してください。

java.lang.String

文字列は以下としてインデックス化されます。

- **short**、**Short**
- **integer**、**Integer**
- **long**、**Long**
- **float**、**Float**
- **double**、**Double**
- **BigInteger**
- **BigDecimal**

数字は文字列の表現に変換されます。Lucene では数字を比較できず、そのままの状態では範囲指定のクエリーに使用できないため、パディングを行う必要があります。



注記

Range クエリーの使用には欠点があるため、代わりに結果クエリーを適切な範囲にフィルターする Filter クエリーを使用することができます。

クエリーモジュールは、カスタム **StringBridge** の使用をサポートします。「[カスタムブリッジ](#)」を参照してください。

java.util.Date

日付はグリニッジ標準時 (GMT) で **yyyyMMddHHmmssSSS** の形式で保存されます。たとえば、アメリカ東部標準時 (EST) の 2006 年 11 月 7 日午後 4 時 3 分 12 秒は 200611072203012 になります。**TermRangeQuery** を使用する場合、日付は GMT で表示されます。

@DateBridge はインデックスへの保存に適切な精度 (resolution) を定義します (例:

@DateBridge(resolution=Resolution.DAY))。日付パターンはこの定義に従って省略されます。

```
@Indexed
public class Meeting {
    @Field(analyze=Analyze.NO)
    @DateBridge(resolution=Resolution.MINUTE)
    private Date date;
```

デフォルトの **Date**ブリッジは Lucene の **DateTools** を使用して **String** からの変換と **String** への変換を行います。日付を固定のタイムゾーンで保存するには、カスタムの date ブリッジを実装します。

java.net.URI, java.net.URL

URI および URL は文字列の表現に変換されます。

java.lang.Class

クラスは完全修飾クラス名に変換されます。クラスがリハイドレートされるとスレッドコンテキストトクラスローダーが使用されます。

19.6.3. カスタムブリッジ

19.6.3.1. カスタムブリッジ

カスタムブリッジは、ビルトインブリッジやブリッジの文字列表現が、必要なプロパティ型に十分に対応できない場合に使用することができます。

19.6.3.2. FieldBridge

柔軟性を向上するため、ブリッジを **FieldBridge** として実装できます。**FieldBridge** インターフェースは、**Lucene Document** でマップできるプロパティ値を提供します。たとえば、1つのプロパティを2つのドキュメントフィールドに保存できます。

FieldBridge インターフェースの実装

```
public class DateSplitBridge implements FieldBridge {
    private final static TimeZone GMT = TimeZone.getTimeZone("GMT");

    public void set(String name,
                    Object value,
                    Document document,
                    LuceneOptions luceneOptions) {
        Date date = (Date) value;
        Calendar cal = GregorianCalendar.getInstance(GMT);
        cal.setTime(date);
        int year = cal.get(Calendar.YEAR);
        int month = cal.get(Calendar.MONTH) + 1;
        int day = cal.get(Calendar.DAY_OF_MONTH);

        // set year
        luceneOptions.addFieldToDocument(
            name + ".year",
            String.valueOf(year),
            document);

        // set month and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".month",
            month < 10 ? "0" : "" + String.valueOf(month),
            document);

        // set day and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".day",
            day < 10 ? "0" : "" + String.valueOf(day),
            document);
    }
}

//property
@FieldBridge(impl = DateSplitBridge.class)
private Date date;
```

以下の例では、フィールドは直接 **Lucene Document** に追加されません。代わりに、追加は **LuceneOptions** ヘルパーに委譲されます。このヘルパーは、**Store** や **TermVector** などの **@Field** で選択されたオプションを適用したり、選択された **@Boost** の値を適用したりします。

LuceneOptions を委譲してフィールドを **Document** に追加することが推奨されますが、**LuceneOptions** を無視して直接 **Document** を編集することも可能です。



注記

LuceneOptions は、**Lucene API** の変更がアプリケーションに影響しないようにし、コードを簡易化します。

19.6.3.3. StringBridge

org.infinispan.query.bridge.StringBridge インターフェースを使用して、想定される **Object** の実装で Lucene ベースの Query API を **String** ブリッジまたは **StringBridge** に提供します。すべての実装は同時に使用されるため、スレッドセーフである必要があります。

カスタム StringBridge 実装

```
/**
 * Padding Integer bridge.
 * All numbers will be padded with 0 to match 5 digits
 *
 * @author Emmanuel Bernard
 */
public class PaddedIntegerBridge implements StringBridge {

    private int PADDING = 5;

    public String objectToString(Object object) {
        String rawInteger = ((Integer) object).toString();
        if (rawInteger.length() > PADDING)
            throw new IllegalArgumentException("Try to pad on a number too big");
        StringBuilder paddedInteger = new StringBuilder();
        for (int padIndex = rawInteger.length() ; padIndex < PADDING ; padIndex++) {
            paddedInteger.append('0');
        }
        return paddedInteger.append(rawInteger).toString();
    }
}
```

@FieldBridge アノテーションを使用すると、この例のプロパティまたはフィールドはブリッジを使用できます。

```
@FieldBridge(impl = PaddedIntegerBridge.class)
private Integer length;
```

19.6.3.4. 双方向ブリッジ

TwoWayStringBridge は **StringBridge** の拡張バージョンで、ブリッジ実装が ID プロパティで使われる場合に使用できます。Lucene ベースの Query API は識別子の文字列表現を読み取り、オブジェクトの生成に使用します。@**FieldBridge** アノテーションは同様に使用されます。

ID プロパティの **TwoWayStringBridge** の実装

```
public class PaddedIntegerBridge implements TwoWayStringBridge,
ParameterizedBridge {

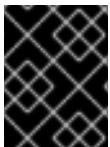
    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map parameters) {
        Object padding = parameters.get(PADDING_PROPERTY);
        if (padding != null) this.padding = (Integer) padding;
    }

    public String objectToString(Object object) {
        String rawInteger = ((Integer) object).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException("Try to pad on a number too
big");
        StringBuilder paddedInteger = new StringBuilder();
        for (int padIndex = rawInteger.length(); padIndex < padding;
padIndex++) {
            paddedInteger.append('0');
        }
        return paddedInteger.append(rawInteger).toString();
    }

    public Object stringToObject(String stringValue) {
        return new Integer(stringValue);
    }
}

@FieldBridge(impl = PaddedIntegerBridge.class,
            params = @Parameter(name = "padding", value = "10"))
private Integer id;
```



重要

object = stringToObject(objectToString(object)) のように、双方向処理はべき等である必要があります。

19.6.3.5. パラメーター化されたブリッジ

ParameterizedBridge インターフェースはパラメーターをブリッジ実装に渡し、柔軟性を向上します。**ParameterizedBridge** は **StringBridge**、**TwoWayStringBridge**、および **FieldBridge** 実装による実行が可能です。すべての実装がスレッドセーフである必要があります。

以下の例は、**ParameterizedBridge** インターフェースを実装し、パラメーターは @**FieldBridge** アノテーションによって渡されます。

ParameterizedBridge インターフェースの設定

```

public class PaddedIntegerBridge implements StringBridge,
ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map <String,String> parameters) {
        String padding = parameters.get(PADDING_PROPERTY);
        if (padding != null) this.padding = Integer.parseInt(padding);
    }

    public String objectToString(Object object) {
        String rawInteger = ((Integer) object).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException("Try to pad on a number too
big");
        StringBuilder paddedInteger = new StringBuilder();
        for (int padIndex = rawInteger.length() ; padIndex < padding ;
padIndex++) {
            paddedInteger.append('0');
        }
        return paddedInteger.append(rawInteger).toString();
    }
}

//property
@FieldBridge(impl = PaddedIntegerBridge.class,
            params = @Parameter(name = "padding", value = "10")
            )
private Integer length;

```

19.6.3.6. 型対応ブリッジ

AppliedOnTypeAwareBridge を実装するすべてのブリッジは、適用またはインジェクトされるブリッジの型を取得します。

- フィールド/ゲッターレベルのブリッジに対するプロパティの戻り値の型。
- クラスレベルのブリッジに対するクラス型。

インジェクトされた型に特有のスレッドセーフ要件はありません。

19.6.3.7. ClassBridge

@ClassBridge アノテーションを使用すると、1つのエンティティの1つ以上のプロパティを Lucene インデックスに固有の方法で組み合わせたリインデックス化することができます。**@ClassBridge** はクラスレベルで定義でき、**termVector** 属性をサポートします。

以下の例では、カスタムの **FieldBridge** 実装はエンティティインスタンスを値パラメーターとして受け取ります。特定の **CatFieldsClassBridge** が department インスタンスに適用されます。**FieldBridge** はブランチとネットワークの両方を連結し、その連結をインデックス化します。

ClassBridge の実装

```
@Indexed
```

```
@ClassBridge(name = "branchnetwork",
              store = Store.YES,
              impl = CatFieldsClassBridge.class,
              params = @Parameter(name = "sepChar", value = ""))
public class Department {
    private int id;
    private String network;
    private String branchHead;
    private String branch;
    private Integer maxEmployees;
}

public class CatFieldsClassBridge implements FieldBridge,
ParameterizedBridge {
    private String sepChar;

    public void setParameterValues(Map parameters) {
        this.sepChar = (String) parameters.get("sepChar");
    }

    public void set(String name,
                    Object value,
                    Document document,
                    LuceneOptions luceneOptions) {

        Department dep = (Department) value;
        String fieldValue1 = dep.getBranch();
        if (fieldValue1 == null) {
            fieldValue1 = "";
        }
        String fieldValue2 = dep.getNetwork();
        if (fieldValue2 == null) {
            fieldValue2 = "";
        }
        String fieldValue = fieldValue1 + sepChar + fieldValue2;
        Field field = new Field(name, fieldValue,
luceneOptions.getStore(),
        luceneOptions.getIndex(), luceneOptions.getTermVector());
        field.setBoost(luceneOptions.getBoost());
        document.add(field);
    }
}
```

第20章 クエリー

20.1. クエリー

Infinispan Query は Lucene クエリーを実行し、ドメインオブジェクトを Red Hat JBoss Data Grid キャッシュから取得できます。

クエリーの準備および実行

1. 以下のように、インデックス化が有効になっているキャッシュの **SearchManager** を取得します。

```
SearchManager manager = Search.getSearchManager(cache);
```

2. 以下のように、**QueryBuilder** を作成して **Myth.class** のクエリーを構築します。

```
final org.hibernate.search.query.dsl.QueryBuilder queryBuilder =  
    manager.buildQueryBuilderForClass(Myth.class).get();
```

3. 以下のように、**Myth.class** クラスの属性をクエリーする Apache Lucene クエリーを作成します。

```
org.apache.lucene.search.Query query = queryBuilder.keyword()  
    .onField("history").boostedTo(3)  
    .matching("storm")  
    .createQuery();  
  
// wrap Lucene query in a org.infinispan.query.CacheQuery  
CacheQuery cacheQuery = manager.getQuery(query);  
  
// Get query result  
List<Object> result = cacheQuery.list();
```

20.2. クエリーの構築

20.2.1. クエリーの構築

クエリーモジュールは Lucene クエリー上で構築されるため、ユーザーはすべての Lucene クエリータイプを使用できます。クエリーが構築されると、Infinispan Query は `org.infinispan.query.CacheQuery` をクエリー操作 API として使用して、さらにクエリー処理を続行します。

20.2.2. Lucene ベースのクエリー API を使用した Lucene クエリーの構築

Lucene API では、クエリーパーサー (簡単なクエリー) または Lucene プログラム API (複雑なクエリー) を使用します。詳細は、Lucene のオンラインドキュメントや「**Lucene in Action**」または「**Hibernate Search in Action**」を参照してください。

20.2.3. Lucene クエリーの構築

20.2.3.1. Lucene クエリーの構築

Lucene プログラム API を使用すると、フルテキストクエリーを書くことができます。しかし、Lucene プログラム API を使用する場合はパラメーターを同等の文字列に変換し、さらに正しいアナライザーを適切なフィールドに適用する必要があります。たとえば、N-gram アナライザーは複数の N-gram を指定の言葉のトークンとして使用し、そのように検索する必要があります。この作業には **QueryBuilder** の使用が推奨されます。

Lucene ベースのクエリー API は流動的です。この API には以下の特徴があります。

- メソッド名は英語になります。そのため、API 操作は一連の英語のフレーズや指示として読み取りおよび理解されます。
- 入力した接頭辞の補完を可能にし、ユーザーが適切なオプションを選択できるようにする IDE 自動補完を使用します。
- チェイニングメソッドパターンを頻繁に使用します。
- API 操作の使用および読み取りは簡単です。

API を使用するには、最初に指定のインデックスタイプにアタッチされるクエリービルダーを作成します。この **QueryBuilder** は、使用するアナライザーと適用するフィールドブリッジを認識します。複数の **QueryBuilder** を作成できます (クエリーのルートに関係する書くタイプごとに 1 つ)。**QueryBuilder** は **SearchManager** から派生します。

```
Search.getSearchManager(cache).buildQueryBuilderForClass(Myth.class).get();
```

指定のフィールドに使用されるアナライザーをオーバーライドすることもできます。

```
SearchManager searchManager = Search.getSearchManager(cache);
QueryBuilder mythQB =
searchManager.buildQueryBuilderForClass(Myth.class)
    .overridesForField("history", "stem_analyzer_definition")
    .get();
```

Lucene クエリーの構築にクエリービルダーが使用されるようになります。

20.2.3.2. キーワードクエリー

以下の例は特定の単語を検索する方法を示しています。

キーワード検索

```
Query luceneQuery =
mythQB.keyword().onField("history").matching("storm").createQuery();
```

表20.1 キーワードクエリーパラメーター

パラメーター	説明
keyword()	このパラメーターを使用して特定の単語を見つけます。

パラメーター	説明
onField()	このパラメーターを使用して単語を検索する Lucene フィールドを指定します。
matching()	このパラメーターを使用して検索する文字列の一致を指定します。
createQuery()	Lucene クエリーオブジェクトを作成します。

- 値「storm」は「history」**FieldBridge** から渡されます。これは、数字や日付が関係する場合に便利です。
- その後、フィールドブリッジの値はフィールド「history」をインデックス化するために使用されるアナライザーへ渡されます。これにより、クエリーがインデックス化と同じ用語変換を使用するようにします (小文字、N-gram、ステミングなど)。分析プロセスが指定の単語に対して複数の用語を生成する場合、ブール値クエリーは **SHOULD** ロジック (おおよそ **OR** ロジックと同様) とともに使用されます。

文字列型でないプロパティを検索します。

```
@Indexed
public class Myth {
    @Field(analyze = Analyze.NO)
    @DateBridge(resolution = Resolution.YEAR)
    public Date getCreationDate() { return creationDate; }
    public void setCreationDate(Date creationDate) { this.creationDate =
creationDate; }
    private Date creationDate;
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword()
    .onField("creationDate")
    .matching(birthdate)
    .createQuery();
```



注記

プレーンな Lucene では、**Date** オブジェクトは文字列の表現に変換する必要がありました (この例では年)。

この変換は、**FieldBridge** に **objectToString** メソッドがあれば (組み込みの **FieldBridge** 実装にはすべてこのメソッドがあります) どのオブジェクトに対しても実行できます。

次の例は、N-gram アナライザーを使用するフィールドを検索します。N-gram アナライザーは単語の N-gram の連続をインデックス化します。これは、ユーザーによる誤字を防ぐのに役立ちます。たとえば、単語 hibernate の 3-gram は hib、ibe、ber、rna、nat、ate になります。

N-gram アナライザーを使用した検索

```

@AnalyzerDef(name = "ngram",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
        @TokenFilterDef(factory = StandardFilterFactory.class),
        @TokenFilterDef(factory = LowerCaseFilterFactory.class),
        @TokenFilterDef(factory = StopFilterFactory.class),
        @TokenFilterDef(factory = NGramFilterFactory.class,
            params = {
                @Parameter(name = "minGramSize", value = "3"),
                @Parameter(name = "maxGramSize", value = "3")}
        })
    })
public class Myth {
    @Field(analyzer = @Analyzer(definition = "ngram"))
    public String getName() { return name; }
    public String setName(String name) { this.name = name; }
    private String name;
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword()
    .onField("name")
    .matching("Sisiphus")
    .createQuery();

```

一致する単語「Sisiphus」は小文字に変換され、3-gram (sis、isi、sip、phu、hus) に分割されます。各 N-gram はクエリーの一部になります。その後、ユーザーは Sysiphus (i ではなく **y**) myth (シーシュポスの神話) を検索できます。ユーザーに対してすべてが透過的行われます。



注記

特定のフィールドがフィールドブリッジまたはアナライザーを使用しないようにするには、**ignoreAnalyzer()** または **ignoreFieldBridge()** 関数を呼び出すことができます。

同じフィールドで可能な単語を複数検索し、すべてを一致する句に追加します。

複数の単語の検索

```

//search document with storm or lightning in their history
Query luceneQuery =
    mythQB.keyword().onField("history").matching("storm
    lightning").createQuery();

```

複数のフィールドで同じ単語を検索するには、**onFields** メソッドを使用します。

複数のフィールドでの検索

```

Query luceneQuery = mythQB
    .keyword()
    .onFields("history", "description", "name")
    .matching("storm")
    .createQuery();

```

同じ用語を検索する場合でも、あるフィールドに必要な処理が他のフィールドとは異なることがあります。このような場合は **andField()** メソッドを使用します。

andField メソッドの使用

```
Query luceneQuery = mythQB.keyword()  
    .onField("history")  
    .andField("name")  
    .boostedTo(5)  
    .andField("description")  
    .matching("storm")  
    .createQuery();
```

前述の例では、フィールド名のみが 5 にブーストされます。

20.2.3.3. ファジークエリー

レーベンシュタイン距離 (Levenshtein Distance) アルゴリズムを基にしてファジークエリーを実行するには、**keyword** クエリーと同様に起動して fuzzy フラグを追加します。

ファジークエリー

```
Query luceneQuery = mythQB.keyword()  
    .fuzzy()  
    .withEditDistanceUpTo(1)  
    .withPrefixLength(1)  
    .onField("history")  
    .matching("starm")  
    .createQuery();
```

withEditDistanceUpTo は、2 つの用語の一致を考慮するための編集距離 (レーベンシュタイン距離) の最大値です。この値は 0 から 2 までの整数値で、デフォルト値は 2 になります。**prefixLength** は「ファジーの度合い」によって無視される接頭辞の長さになります。デフォルト値は 0 ですが、異なる用語が大量に含まれるインデックスではゼロ以外の値を指定することが推奨されます。

20.2.3.4. ワイルドカードクエリー

ワイルドカードクエリーを実行することもできます (単語の一部が不明のクエリー)。? は単一の文字を表し、* は連続する文字を表します。パフォーマンス上の理由で、クエリーの最初に ? または * を使用しないことが推奨されます。

ワイルドカードクエリー

```
Query luceneQuery = mythQB.keyword()  
    .wildcard()  
    .onField("history")  
    .matching("sto*")  
    .createQuery();
```



注記

ワイルドカードクエリーは、アナライザーを一致する用語に適用しません。これは、* または ? が適切に処理されないリスクが大変高くなるためです。

20.2.3.5. フレーズクエリー

これまでは、単語または単語のセットを検索しましたが、完全一致の文または近似の文を検索することも可能です。文の検索には **phrase()** を使用します。

フレーズクエリー

```
Query luceneQuery = mythQB.phrase()
    .onField("history")
    .sentence("Thou shalt not kill")
    .createQuery();
```

おおよその文はスロップ (slop) 係数を追加すると検索可能になります。スロップ係数は、その文で許可される別の単語の数を表します。これは、within または near 演算子と同様に動作します。

スロップ係数の追加

```
Query luceneQuery = mythQB.phrase()
    .withSlop(3)
    .onField("history")
    .sentence("Thou kill")
    .createQuery();
```

20.2.3.6. 範囲クエリー

範囲クエリーは指定の境界の間、上、または下で値を検索します。

範囲クエリー

```
//look for 0 <= starred < 3
Query luceneQuery = mythQB.range()
    .onField("starred")
    .from(0).to(3).excludeLimit()
    .createQuery();

//look for myths strictly BC
Date beforeChrist = ...;
Query luceneQuery = mythQB.range()
    .onField("creationDate")
    .below(beforeChrist).excludeLimit()
    .createQuery();
```

20.2.3.7. クエリーの組み合わせ

クエリーを集合 (組み合わせ) するとさらに複雑なクエリーを作成できます。以下の集合演算子を使用できます。

- **SHOULD**: クエリーにはサブクエリーの一致要素が含まれるはずですが。
- **MUST**: クエリーにはサブクエリーの一致要素が含まれていなければなりません。
- **MUST NOT**: クエリーにサブクエリーの一致要素が含まれてはなりません。

サブクエリーはブール値クエリー自体を含む Lucene クエリーです。例を以下に示します。

サブクエリーの組み合わせ

```
//look for popular modern myths that are not urban
Date twentiethCentury = ...;
Query luceneQuery = mythQB.bool()

.must(mythQB.keyword().onField("description").matching("urban").createQuery())
    .not()
    .must(mythQB.range().onField("starred").above(4).createQuery())
    .must(mythQB.range()
        .onField("creationDate")
        .above(twentiethCentury)
        .createQuery())
    .createQuery();

//look for popular myths that are preferably urban
Query luceneQuery = mythQB
    .bool()
    .should(mythQB.keyword()
        .onField("description")
        .matching("urban")
        .createQuery())
    .must(mythQB.range().onField("starred").above(4).createQuery())
    .createQuery();

//look for all myths except religious ones
Query luceneQuery = mythQB.all()
    .except(mythQB.keyword()
        .onField("description_stem")
        .matching("religion")
        .createQuery())
    .createQuery();
```

20.2.3.8. クエリーオプション

クエリー型およびフィールドのクエリーオプションの概要は次のとおりです。

- **boostedTo** (クエリー型およびフィールド上) は、クエリーまたはフィールドを指定の係数にブーストします。
- **withConstantScore** (クエリー上) は、クエリーと一致し、ブーストと等しくなる定数スコアを持つすべての結果を返します。
- **filteredBy(Filter)** (クエリー上) は **Filter** インスタンスを使用してクエリー結果をフィルターします。
- **ignoreAnalyzer** (フィールド上) は、このフィールドの処理時にアナライザーを無視します。
- **ignoreFieldBridge** (フィールド上) は、このフィールドの処理時にフィールドブリッジを無視します。

以下の例は、これらのオプションの使用方法を表しています。

クエリーオプション

```

Query luceneQuery = mythQB
    .bool()

    .should(mythQB.keyword().onField("description").matching("urban").createQuery())
    .should(mythQB
        .keyword()
        .onField("name")
        .boostedTo(3)
        .ignoreAnalyzer()
        .matching("urban").createQuery())
    .must(mythQB
        .range()
        .boostedTo(5)
        .withConstantScore()
        .onField("starred")
        .above(4).createQuery())
    .createQuery();

```

20.2.4. Infinispan Query でのクエリーの構築

20.2.4.1. 一般論

Lucene クエリーを構築したら、Infinispan CacheQuery クエリー内でラップします。クエリーは、インデックス化されたエンティティをすべて検索し、インデックス化されたクラスのすべての型を返します。この挙動を変更するには、明示的に設定する必要があります。

Infinispan CacheQuery での Lucene クエリーのラッピング

```

CacheQuery cacheQuery =
    Search.getSearchManager(cache).getQuery(luceneQuery);

```

パフォーマンスを向上するには、以下のように戻り値の型を制限します。

エンティティ型での検索結果のフィルター

```

CacheQuery cacheQuery =
    Search.getSearchManager(cache).getQuery(luceneQuery, Customer.class);
// or
CacheQuery cacheQuery =
    Search.getSearchManager(cache).getQuery(luceneQuery, Item.class,
    Actor.class);

```

2 つ目の例の最初の部分は、一致する **Customer** インスタンスのみを返します。同じ例の 2 番目の部分は、一致する **Actor** および **Item** インスタンスを返します。型制限は多形です。このため、ベースクラス **Person** の 2 つのサブクラスである **Salesman** および **Customer** が返される場合は、**Person.class** を指定して結果の型に基づいてフィルターします。

20.2.4.2. ページネーション

パフォーマンスの劣化を防ぐため、クエリーごとに返されるオブジェクトの数を制限することが推奨されます。ユーザーがあるページから別のページへ移動するユースケースは大変一般的です。ページネーションを定義する方法は、プレーン HQL または Criteria クエリーでページネーションを定義する方法

に似ています。

検索クエリーに対するページネーションの定義

```
CacheQuery cacheQuery = Search.getSearchManager(cache)
    .getQuery(luceneQuery, Customer.class);
cacheQuery.firstResult(15); //start from the 15th element
cacheQuery.maxResults(10); //return 10 elements
```



注記

ページネーションに関係なく、一致する要素の合計数は `cacheQuery.getResultSize()` で取得できます。

20.2.4.3. ソート

Apache Lucene には、柔軟で強力な結果ソートメカニズムが含まれています。デフォルトでは関連性でソートされます。他のプロパティーでソートするようソートメカニズムを変更するには、Lucene Sort オブジェクトを使用して Lucene ソートストラテジーを適用します。

Lucene Sort の指定

```
org.infinispan.query.CacheQuery cacheQuery =
Search.getSearchManager(cache).getQuery(luceneQuery, Book.class);
org.apache.lucene.search.Sort sort = new Sort(
    new SortField("title", SortField.STRING_FIRST));
cacheQuery.sort(sort);
List results = cacheQuery.list();
```



注記

ソートに使用されるフィールドはトークン化しないでください。トークン化に関する詳細は「[@Field](#)」を参照してください。

20.2.4.4. 射影

場合によってはプロパティーの小さなサブセットのみが必要になることがあります。以下のように、Infinispan Query を使用してプロパティーのサブセットを返します。

完全なドメインオブジェクトを返す代わりに射影を使用

```
SearchManager searchManager = Search.getSearchManager(cache);
CacheQuery cacheQuery = searchManager.getQuery(luceneQuery, Book.class);
cacheQuery.projection("id", "summary", "body", "mainAuthor.name");
List results = cacheQuery.list();
Object[] firstResult = (Object[]) results.get(0);
Integer id = (Integer) firstResult[0];
String summary = (String) firstResult[1];
String body = (String) firstResult[2];
String authorName = (String) firstResult[3];
```


クエリーモジュールは Lucene インデックスからプロパティを抽出して、オブジェクト表現に変換してから **Object[]** のリストを返します。射影により、時間がかかるデータベースのラウンドトリップは回避されますが、以下の制約があります。

- 射影されたプロパティはインデックス (**@Field(store=Store.YES)**) に保存される必要があります。これにより、インデックスのサイズが大きくなります。
- 射影されたプロパティは **org.infinispan.query.bridge.TwoWayFieldBridge** または **org.infinispan.query.bridge.TwoWayStringBridge** を実装する **FieldBridge** を使用する必要があります。**org.infinispan.query.bridge.TwoWayStringBridge** はより簡単なバージョンです。



注記

Lucene ベースのクエリー API の組み込み型はすべて双方向です。

- インデックス化されたエンティティのシンプルなプロパティまたは埋め込みされた関連のみを射影できます。埋め込みエンティティ全体は射影できません。
- 射影は、**@IndexedEmbedded** を用いてインデックス化されたコレクションまたはマップでは動作しません。

Lucene はクエリー結果のメタデータ情報を提供します。射影定数を使用してメタデータを読み出します。

射影を使用したメタデータの読み出し

```
SearchManager searchManager = Search.getSearchManager(cache);
CacheQuery cacheQuery = searchManager.getQuery(luceneQuery, Book.class);
cacheQuery.projection("mainAuthor.name");
List results = cacheQuery.list();
Object[] firstResult = (Object[]) results.get(0);
float score = (Float) firstResult[0];
Book book = (Book) firstResult[1];
String authorName = (String) firstResult[2];
```

フィールドは、以下の射影定数と組み合わせることができます。

- **FullTextQuery.THIS** は、射影されていないクエリーのように、初期化および管理されたエンティティを返します。
- **FullTextQuery.DOCUMENT** は、射影されたオブジェクトに関連する Lucene Document を返します。
- **FullTextQuery.OBJECT_CLASS** は、インデックス化されたエンティティのクラスを返します。
- **FullTextQuery.SCORE** は、クエリーのドキュメントスコアを返します。スコアを使用して、指定のクエリーの結果の 1 つを他の結果と比較します。スコアは異なる 2 つのクエリーの結果を比較するためのものではありません。
- **FullTextQuery.ID** は、射影されたオブジェクトの ID プロパティ値です。
- **FullTextQuery.DOCUMENT_ID** は Lucene ドキュメント ID です。Lucene ドキュメント ID は開かれる 2 つの IndexReader の間で変更されます。

- **FullTextQuery.EXPLANATION** は、クエリーの一一致するオブジェクト/ドキュメントに対して Lucene Explanation オブジェクトを返します。これは、大量のデータの取得には適していません。**FullTextQuery.EXPLANATION** の実行コストは、一致する各要素に対して Lucene クエリーを実行するのに匹敵します。そのため、射影が推奨されます。

20.2.4.5. クエリー時間の制限

次のように、Infinispan Query でクエリーが要する時間を制限します。

- 制限に達したら例外を発生します。
- 時間制限に達したら取得された結果の数を制限します。

20.2.4.6. 時間制限での例外の発生

定義された時間を超えてクエリーが実行される場合に発生する、カスタム例外を定義できます。

CacheQuery API の使用時に制限を定義するには、以下の方法を使用します。

クエリー実行でのタイムアウトの定義

```
SearchManagerImplementor searchManager = (SearchManagerImplementor)
Search.getSearchManager(cache);
searchManager.setTimeoutExceptionFactory(new MyTimeoutExceptionFactory());
CacheQuery cacheQuery = searchManager.getQuery(luceneQuery, Book.class);

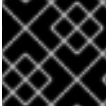
//define the timeout in seconds
cacheQuery.timeout(2, TimeUnit.SECONDS);

try {
    cacheQuery.list();
}
catch (MyTimeoutException e) {
    //do something, too slow
}

private static class MyTimeoutExceptionFactory implements
TimeoutExceptionFactory {
    @Override
    public RuntimeException createTimeoutException(String message, String
queryDescription) {
        return new MyTimeoutException();
    }
}

public static class MyTimeoutException extends RuntimeException {
}
```

getResultSize()、**iterate()**、および **scroll()** はメソッド呼び出しが終了するまでタイムアウトを考慮します。そのため、**Iterable** または **ScrollableResults** はタイムアウトを無視します。さらに、**explain()** はこのタイムアウト期間を考慮しません。このメソッドは、デバッグやクエリーのパフォーマンスが遅い理由をチェックするために使用されます。



重要

サンプルコードは、クエリーが指定された結果の値で停止することを保証しません。

20.3. 結果の読み出し

20.3.1. 結果の読み出し

Infinispan Query は構築後に HQL や Criteria クエリーと同様に実行できます。同じパラダイムとオブジェクトセマンティックが Lucene クエリーおよび `list()` などの一般的な操作すべてに適用されます。

20.3.2. パフォーマンスに関する注意点

`list()` を使用すると妥当な数の結果を受信し (たとえば、ページネーションを使用する場合など)、すべてに対応することができます。`list()` は `batch-size` エンティティが適切に設定されている場合に最適に動作します。`list()` が使用されると、クエリーモジュールはページネーション内ですべての Lucene Hits 要素を処理します。

20.3.3. 結果サイズ

ユースケースによっては、一致するドキュメントの合計数に関する情報が必要になります。以下の例について考えてみましょう。

一致するドキュメントをすべて読み出すには大量のリソースを消費します。Lucene ベースのクエリー API は、ページネーションのパラメーターに関係なく、一致するドキュメントをすべて読み出します。一致するドキュメントをすべて読み出すには大量のリソースが必要であるため、Lucene ベースのクエリー API はページネーションのパラメーターに関係なく、一致するドキュメントの合計数を読み出します。一致するすべての要素は、オブジェクトのロードを発生せずに読み出されます。

クエリーの結果サイズの決定

```
CacheQuery cacheQuery =
    Search.getSearchManager(cache).getQuery(luceneQuery,
        Book.class);
//return the number of matching books without loading a single one
assert 3245 == cacheQuery.getResultSize();

CacheQuery cacheQueryLimited =
    Search.getSearchManager(cache).getQuery(luceneQuery, Book.class);
cacheQuery.maxResults(10);
List results = cacheQuery.list();
assert 10 == results.size();
//return the total number of matching books regardless of pagination
assert 3245 == cacheQuery.getResultSize();
```

インデックスが適切にデータベースと同期されていない場合、結果の数は近似値になります。この場合の例の 1 つとして非同期クラスターが挙げられます。

20.3.4. 結果の理解

Luke を使用すると、想定されるクエリー結果で結果が表示される (または表示されない) 理由を判断できます。また、クエリーモジュールは指定の結果 (指定のクエリーの) に対する Lucene **Explanation**

オブジェクトも提供します。これは上級クラスです。次のように **Explanation** オブジェクトにアクセスします。

cacheQuery.explain(int) メソッド

このメソッドでは、パラメーターとするドキュメント ID が必要で、**Explanation** オブジェクトを返します。



注記

explanation オブジェクトの構築は、Lucene クエリーの実行と同様にリソースを大量に消費します。実装で必要になる場合のみ、explanation オブジェクトを構築してください。

20.4. フィルター

20.4.1. フィルター

Apache Lucene は、カスタムのフィルター処理に従ってクエリーの結果をフィルターすることができます。これは、フィルターをキャッシュおよび再使用できるため、データの制限を追加で適用する強力な方法です。該当するユースケースには以下が含まれます。

- セキュリティー
- 一時データ (例: 閲覧専用の先月のデータ)
- 入力 (population) フィルター (例: 指定のカテゴリに限定される検索)
- その他多くのユースケース

20.4.2. フィルターの定義および実装

Lucene ベースのクエリー API には、パラメーターが含まれる filters という名前の透過キャッシュが含まれます。この API は Hibernate Core フィルターと似ています。

クエリーに対するフルテキストフィルターの有効化

```
cacheQuery = Search.getSearchManager(cache).getQuery(query, Driver.class);
cacheQuery.enableFullTextFilter("bestDriver");
cacheQuery.enableFullTextFilter("security").setParameter("login",
"andre");
cacheQuery.list(); //returns only best drivers where andre has credentials
```

この例では、クエリーで 2 つのフィルターが有効になっています。フィルターを有効または無効にしてクエリーをカスタマイズします。

@FullTextFilterDef アノテーションを使用してフィルターを宣言します。このアノテーションは、フィルターのクエリーに関係なく **@Indexed** エンティティに適用されます。フィルター定義はグローバルであるため、各フィルターに一意な名前を付ける必要があります。同じ名前を持つ 2 つの **@FullTextFilterDef** アノテーションが定義された場合、**SearchException** が発生します。名前の付いた各フィルターにはそのフィルター実装を指定する必要があります。

フィルターの定義および実装

```

@FullTextFilterDefs({
    @FullTextFilterDef(name = "bestDriver", impl =
BestDriversFilter.class),
    @FullTextFilterDef(name = "security", impl =
SecurityFilterFactory.class)
})
public class Driver { ... }

public class BestDriversFilter extends org.apache.lucene.search.Filter {

    public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
        OpenBitSet bitSet = new OpenBitSet(reader.maxDoc());
        TermDocs termDocs = reader.termDocs(new Term("score", "5"));
        while (termDocs.next()) {
            bitSet.set(termDocs.doc());
        }
        return bitSet;
    }
}

```

BestDriversFilter はドライバーへの結果セットを減らす Lucene フィルターで、スコアは **5** になります。この例では、フィルターは直接 **org.apache.lucene.search.Filter** を実装し、引数がないコンストラクターが含まれます。

20.4.3. @Factory フィルター

フィルターの作成に追加の手順が必要であったり、フィルターに引数がないコンストラクターがない場合は、以下のファクトリーパターンを使用します。

ファクトリーパターンを使用したフィルターの作成

```

@FullTextFilterDef(name = "bestDriver", impl =
BestDriversFilterFactory.class)
public class Driver { ... }

public class BestDriversFilterFactory {

    @Factory
    public Filter getFilter() {
        //some additional steps to cache the filter results per
IndexReader
        Filter bestDriversFilter = new BestDriversFilter();
        return new CachingWrapperFilter(bestDriversFilter);
    }
}

```

Lucene ベースのクエリー API は **@Factory** アノテーションが付いたメソッドを使用してフィルターインスタンスを構築します。ファクトリーには引数がないコンストラクターが含まれる必要があります。

名前付きフィルターは、パラメーターをフィルターに渡す必要がある場合に便利です。たとえば、セキュリティフィルターが、適用するセキュリティレベルを認識する場合を考えてみます。

パラメーターを定義されたフィルターに渡す

```
cacheQuery = Search.getSearchManager(cache).getQuery(query, Driver.class);
cacheQuery.enableFullTextFilter("security").setParameter("level", 5);
```

対象となる名前付きフィルター定義のフィルターまたはフィルターファクトリーのいずれかで、関連付けられたセッターが各パラメーターに含まれる必要があります。

実際のフィルター実装でのパラメーターの使用

```
public class SecurityFilterFactory {
    private Integer level;

    /**
     * injected parameter
     */
    public void setLevel(Integer level) {
        this.level = level;
    }

    @Key
    public FilterKey getKey() {
        StandardFilterKey key = new StandardFilterKey();
        key.addParameter(level);
        return key;
    }

    @Factory
    public Filter getFilter() {
        Query query = new TermQuery(new Term("level", level.toString()));
        return new CachingWrapperFilter(new QueryWrapperFilter(query));
    }
}
```

@Key アノテーションの付いたメソッドは **FilterKey** オブジェクトを返します。返されたオブジェクトには特別なコントラクトがあります。キーオブジェクトは **equals()** / **hashCode()** を実装して、特定の **Filter** タイプが同じでパラメーターのセットが同じ場合でのみ 2 つのキーが等しくなるようにします。つまり、キーが生成されるフィルターが交換可能な場合のみ 2 つのフィルターキーは等しくなります。キーオブジェクトはキャッシュメカニズムでキーとして使用されます。

20.4.4. キーオブジェクト

@Key メソッドは以下の場合のみ必要です。

- フィルターキャッシュシステムが有効である (デフォルトで有効)
- フィルターにパラメーターが含まれる

StandardFilterKey は **equals()** / **hashCode()** 実装をパラメーター **equals** および **hashCode** メソッドに委譲します。

定義されたフィルターはデフォルトでキャッシュされ、キャッシュはハード参照とソフト参照の組み合わせを使用して必要な場合にメモリーの破棄を許可します。ハード参照キャッシュは最後に使用されたフィルターを追跡し、使用頻度が最も低いフィルターを必要に応じて **SoftReferences** に変換します。ハード参照キャッシュの制限に達すると、追加のフィルターは **SoftReferences** としてキャッシュされます。ハード参照キャッシュのサイズを調整するには、

`default.filter.cache_strategy.size` (デフォルト値は 128) を使用します。フィルターキャッシュの高度な使用については、独自の **FilterCachingStrategy** を実装してください。クラス名は `default.filter.cache_strategy` によって定義されます。

このフィルターキャッシュメカニズムを実際のフィルター結果と混同しないでください。Lucene では、**CachingWrapperFilter** に **IndexReader** を使用してフィルターをラップすることが一般的です。このラッパーは、コストがかかる再計算を回避するために `getDocIdSet (IndexReader reader)` メソッドから返された **DocIdSet** をキャッシュします。リーダーは開いたときのインデックスの状態を表すため、計算される **DocIdSet** は同じ **IndexReader** インスタンスに対してのみキャッシュできることに注意してください。ドキュメントリストは開いた **IndexReader** 内で変更できません。ただし、別または新しい **IndexReader** インスタンスが **Document** の別のセット (別のインデックスのもの、またはインデックスが変更されたため) で動作することがあります。この場合、キャッシュされた **DocIdSet** は再計算する必要があります。

20.4.5. フルテキストフィルター

Lucene ベースのクエリー API は `@FullTextFilterDef` の `cache` フラグを使用し、**FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS** に設定します。これは、フィルターインスタンスを自動的にキャッシュし、**CachingWrapperFilter** の Hibernate 固有の実装にフィルターをラッピングします。Lucene バージョンのこのクラスとは異なり、**SoftReference** はハード参照数 (フィルターキャッシュに関する説明を参照) とともに使用されます。ハード参照数は、`default.filter.cache_docidresults.size` を使用して調整できます (デフォルト値は 5)。ラッピングは `@FullTextFilterDef.cache` パラメーターを使用して調整されます。このパラメーターには以下の 3 つの値があります。

Value	定義
<code>FilterCacheModeType.NONE</code>	フィルターインスタンスなしと結果なしは Hibernate Search によってキャッシュされます。フィルターの呼び出しごとに、新しいフィルターインスタンスが作成されます。この設定は、頻繁に変更するデータやメモリの制約が大きい環境に役に立つことがあります。
<code>FilterCacheModeType.INSTANCE_ONLY</code>	フィルターインスタンスはキャッシュされ、同時 Filter.getDocIdSet() 呼び出しで再使用されます。 DocIdSet の結果はキャッシュされません。この設定は、アプリケーション固有のイベントにより DocIdSet のキャッシュが不必要になったことが原因で、フィルターが独自のキャッシュメカニズムを使用する場合やフィルター結果が動的に変更される場合に役に立ちます。
<code>FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS</code>	フィルターインスタンスの結果と DocIdSet の結果の両方がキャッシュされます。これはデフォルト値です。

フィルターは次の状況でキャッシュされる必要があります。

- システムが対象となるエンティティインデックスを頻繁に更新しない (つまり、**IndexReader** が頻繁に再利用される)

- フィルターの DocIdSet の計算のコストが高い (クエリーを実行するのにかかる時間と比較して)

20.4.6. シャード化された環境におけるフィルターの使用

シャード化された環境にて、利用可能なシャードのサブセットでクエリーを実行するには、以下の手順にしたがいます。

1. フィルター設定に応じて **IndexManager** のサブセットを選択するため、シャードストラテジーを作成します。
2. クエリーの実行時にフィルターをアクティベートします。

以下は、customer フィルターがアクティベートされた場合に特定のシャードをクエリーするシャードストラテジーの例になります。

特定シャードのクエリー

```
public class CustomerShardingStrategy implements IndexShardingStrategy {

    // stored IndexManagers in a array indexed by customerID
    private IndexManager[] indexManagers;

    public void initialize(Properties properties, IndexManager[]
indexManagers) {
        this.indexManagers = indexManagers;
    }

    public IndexManager[] getIndexManagersForAllShards() {
        return indexManagers;
    }

    public IndexManager getIndexManagerForAddition(
        Class<?> entity, Serializable id, String idInString, Document
document) {
        Integer customerID =
Integer.parseInt(document.getFieldable("customerID")
.stringValue());

        return indexManagers[customerID];
    }

    public IndexManager[] getIndexManagersForDeletion(
        Class<?> entity, Serializable id, String idInString) {
        return getIndexManagersForAllShards();
    }

    /**
     * Optimization; don't search ALL shards and union the results; in
this case, we
     * can be certain that all the data for a particular customer Filter
is in a single
     * shard; return that shard by customerID.
     */
    public IndexManager[] getIndexManagersForQuery(
        FullTextFilterImplementor[] filters) {
        FullTextFilter filter = getCustomerFilter(filters, "customer");
        if (filter == null) {
```



```

        return getIndexManagersForAllShards();
    }
    else {
        return new IndexManager[] { indexManagers[Integer.parseInt(
            filter.getParameter("customerID").toString())] };
    }
}

private FullTextFilter getCustomerFilter(FullTextFilterImplementor[]
filters,
                                         String name) {
    for (FullTextFilterImplementor filter: filters) {
        if (filter.getName().equals(name)) return filter;
    }
    return null;
}
}

```

この例では、**customer** フィルターが存在する場合、クエリーはカスタマー専用のシャードのみを使用します。**customer** フィルターが見つからない場合は、クエリーはすべてのシャードを返します。シャードストラテジーは提供されたパラメーターに応じて各フィルターに反応します。

クエリーの実行が必要なときに、フィルターをアクティベートします。フィルターは、クエリーの後に Lucene の結果をフィルターする通常のフィルターです (「[フィルター](#)」にて定義)。この代わりに、シャードストラテジーに渡され、クエリーの実行中は無視される特別なフィルターを使用できます。**ShardSensitiveOnlyFilter** クラスを使用してそのフィルターを宣言します。

ShardSensitiveOnlyFilter クラスの使用

```

@Indexed
@FullTextFilterDef(name = "customer", impl =
ShardSensitiveOnlyFilter.class)
public class Customer {
    ...
}

CacheQuery cacheQuery = Search.getSearchManager(cache).getQuery(query,
    Customer.class);
cacheQuery.enableFullTextFilter("customer").setParameter("CustomerID", 5);
@SuppressWarnings("unchecked")
List results = cacheQuery.list();

```

ShardSensitiveOnlyFilter フィルターが使用される場合、Lucene フィルターを実装する必要はありません。これらのフィルターに反応するフィルターとシャードストラテジーを使用して、シャード化された環境でクエリーを迅速に実行します。

20.5. 継続的クエリー

20.5.1. 継続的クエリー

継続的クエリーは、アプリケーションが現在クエリーと一致するエントリーを受信できるようにし、クエリーされたデータセットへの変更が継続して通知されるようにします。これには、追加のキャッシュ操作による、受信の一致 (セットに参加した値に対する) と送信の一致 (セットから離脱した値に対する)

の両方が含まれます。継続的なクエリーを使用すると、アプリケーションは安定してイベントを受信するため、変更を発見するために同じクエリーを繰り返し実行しません。そのため、リソースをより効率的に使用できます。

たとえば、以下のユースケースはすべて継続的なクエリーを利用できます。

1. 年齢が 18 歳から 25 歳までの人を全員返します (**Person** エンティティに **age** プロパティがあり、ユーザーアプリケーションによって更新されることを前提とします)。
2. \$2000 を超える取引をすべて返します。
3. 1:45.00s 未満である F1 レーサーのラップタイムをすべて返します (キャッシュに **Lap** エントリーが含まれ、レース中にラップタイムがリアルタイムで入力されることを前提とします)。

20.5.2. 継続的クエリーの評価

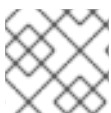
継続的クエリーは以下の場合に通知を受け取るリスナーを使用します。

- エントリーが指定クエリーの一致を開始したとき。**Join** イベントによって表されます。
- エントリーが指定クエリーの一致を停止したとき。**Leave** イベントによって表されます。

クライアントが継続的クエリーリスナーを登録した直後、現在クエリーと一致する結果の受信が開始されます。前述のとおり、これは **Join** イベントとして受信されます。さらに、通常は **creation**、**modification**、**removal**、または **expiration** イベントを生成するキャッシュ操作の結果として、他のエントリーがクエリーに一致すると、**Join** イベントとして後続の通知も受信され、クエリーの一致が停止されると **Leave** イベントとして受信されます。

リスナーが **Join** または **Leave** イベントを受信するかを判断するため、以下の論理が使用されます。

1. 新旧両方の値でクエリーが **false** と評価された場合、イベントは抑制されます。
2. 新旧両方の値でクエリーが **true** と評価された場合、イベントは抑制されます。
3. 古い値でクエリーが **false** と評価され、新しい値でクエリーが **true** と評価された場合、**Join** イベントが送信されます。
4. 古い値でクエリーが **true** と評価され、新しい値でクエリーが **false** と評価された場合、**Leave** イベントが送信されます。
5. 古い値でクエリーが **true** と評価され、エントリーが削除された場合、**Leave** イベントが送信されます。



注記

継続的クエリーはグループ化、集計、およびソート操作を使用できません。

20.5.3. 継続的クエリーの使用

以下の手順は、ライブラリーモードとリモートクライアントサーバーモードの両方に適用されます。

継続的クエリーの追加

継続的クエリーを作成するため、他のクエリーメソッドと同様に **Query** オブジェクトが作成されますが、**Query** が **org.infinispan.query.api.continuous.ContinuousQuery** と登録されるようにし、**org.infinispan.query.api.continuous.ContinuousQueryListener** が使用されてい

るようにしてください。

キャッシュに関連付けられた **ContinuousQuery** オブジェクトを取得するには、クライアントサーバーモードで実行している場合は静的メソッド

org.infinispan.client.hotrod.Search.getContinuousQuery(RemoteCache<K, V> cache) を呼び出し、ライブラリーモードで実行している場合は

org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache) を呼び出します。

ContinuousQueryListener が定義されたら、**ContinuousQuery** の **addContinuousQueryListener** メソッドを使用して追加できます。

```
continuousQuery.addContinuousQueryListener(query, listener)
```

以下の例は、ライブラリーモードで継続的クエリーを実装および追加する簡単なメソッドを表しています。

継続的クエリーの定義および追加

```
import org.infinispan.query.api.continuous.ContinuousQuery;
import org.infinispan.query.api.continuous.ContinuousQueryListener;
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

[...]
```

// To begin we create a ContinuousQuery instance on the cache
ContinuousQuery<Integer, Person> continuousQuery =
Search.getContinuousQuery(cache);

*// Define our query. In this case we will be looking for any
// Person instances under 21 years of age.*
QueryFactory queryFactory = Search.getQueryFactory(cache);
Query query = queryFactory.from(Person.class)
 .having("age").lt(21)
 .build();

final Map<Integer, Person> matches = new ConcurrentHashMap<Integer,
Person>();

// Define the ContinuousQueryListener
ContinuousQueryListener<Integer, Person> listener = new
ContinuousQueryListener<Integer, Person>() {
 @Override
 public void resultJoining(Integer key, Person value) {
 matches.put(key, value);
 }

 @Override
 public void resultLeaving(Integer key) {
 matches.remove(key);
 }
}

```

    }
};

// Add the listener and generated query
continuousQuery.addContinuousQueryListener(query, listener);

[...]

// Remove the listener to stop receiving notifications
continuousQuery.removeContinuousQueryListener(listener);

```

Person インスタンスが 21 未満の **Age** が含まれるキャッシュへ追加されると **matches** に配置されます。これらのエントリーがキャッシュから削除されると、**matches** から削除されます。

継続的クエリーの削除

クエリーの実行を停止するには、リスナーを削除します。

```
continuousQuery.removeContinuousQueryListener(listener);
```

20.5.4. C++ および C# の継続的クエリー

JBoss Data Grid はネイティブの Java ベースの継続的クエリーの他に、C++ および C# ベースの継続的クエリーもサポートします。

20.5.4.1. C++ 継続的クエリー

C++ 継続的クエリーは以下のコードを使用して設定できます。

C++ 継続的クエリーのセットアップ

```

ContinuousQueryListener<int, sample_bank_account::User>
cql(testCache, "select id from sample_bank_account.User");
std::function<void(int, sample_bank_account::User)> join = [](int k,
sample_bank_account::User u) {
    std::cout << "JOINING: key=" << u.id() << " value=" << u.name() <<
std::endl;
};
std::function<void(int, sample_bank_account::User)> leave = [](int k,
sample_bank_account::User u) {
    std::cout << "LEAVING: key=" << u.id() << " value=" << u.name() <<
std::endl;
};
std::function<void(int, sample_bank_account::User)> change = [](int k,
sample_bank_account::User u) {
    std::cout << "CHANGING: key=" << u.id() << " value=" << u.name() <<
std::endl;
};

cql.setJoiningListener(join);
cql.setLeavingListener(leave);
cql.setUpdatedListener(change);
testCache.addContinuousQueryListener(cql);

```

C++ 継続的クエリーは以下のコードを使用して削除できます。

C++ 継続的クエリーの削除

```
testCache.AddContinuousQueryListener(cql);

[...]
```

// Remove the listener to stop receiving notifications
testCache.RemoveContinuousQueryListener(cql);

20.5.4.2. C# 継続的クエリー

C# 継続的クエリーは以下のコードを使用して設定できます。

C# 継続的クエリーのセットアップ

```
qr.QueryString = "from sample_bank_account.User";

Event.ContinuousQueryListener<int, User> cql = new
Event.ContinuousQueryListener<int, User>(qr.QueryString);
cql.JoiningCallback = (int k, User v) => { Console.WriteLine("JOINING: " +
k + ", " + v); s.Release(); };
cql.LeavingCallback = (int k, User v) => { Console.WriteLine("LEAVING: " +
k + ", " + v); };
cql.UpdatedCallback = (int k, User v) => { Console.WriteLine("UPDATED: " +
k + ", " + v); };
userCache.AddContinuousQueryListener(cql);
```

C# 継続的クエリーは以下のコードを使用して削除できます。

C# 継続的クエリーの削除

```
userCache.AddContinuousQueryListener(cql);

[...]
```

// Remove the listener to stop receiving notifications
userCache.RemoveContinuousQueryListener(cql);

20.5.5. 継続的クエリーでのパフォーマンスに関する注意点

継続的クエリーは、アプリケーションが常に最新の状態を保持するように設計されているため、特に広範囲のクエリーに対して生成されたイベントの数が増える可能性があります。さらに、各イベントに対して新たにメモリーが割り当てられます。注意してクエリーを作成しないと、この挙動が原因でメモリーの負荷が発生し、エラーなどを引き起こす可能性があります。

この問題を防ぐため、各クエリーには必要な情報のみを指定し、各 **ContinuousQueryListener** が受信したすべてのイベントを迅速に処理できるようにすることが強く推奨されます。

20.6. ブロードキャストクエリー

20.6.1. ブロードキャストクエリー

ブロードキャストクエリー機能は、各ノードが書き込み中に独自のデータをインデックス化し、クエリー時に各ノードにクエリーを送信 (ブロードキャスト) できるようにします。各ノードの結果は、呼び出し元に返される前に組み合わせられます。これは、転送されるデータの量はクエリー自体と結果であるため、大きなインデックスを持つ **DIST** キャッシュに適しています。

20.6.1.1. ブロードキャストクエリーの使用

ブロードキャストクエリーを使用するには、**IndexedQueryMode.BROADCAST** が引数としてクエリーに含まれるようにします。この例を以下に示します。

```
CacheQuery<Person> broadcastQuery =  
    Search.getSearchManager(cache).getQuery(new MatchAllDocsQuery(),  
        IndexedQueryMode.BROADCAST);  
  
List<Person> result = broadcastQuery.list();
```

第21章 INFINISPAN QUERY DSL

21.1. INFINISPAN QUERY DSL

Infinispan Query DSL はキャッシュをクエリーするための統一された方法を提供します。これは、ライブラリーモードでインデックス化されたクエリーとインデックスのないクエリーの両方に使用でき、リモートクエリー (Hot Rod Java クライアントを経由) にも使用できます。Infinispan Query DSL では、Lucene ネイティブクエリー API や Hibernate Search クエリー API に依存せずにクエリーを実行できます。

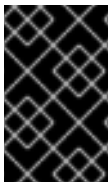
Infinispan Query DSL では、インデックスのないクエリーは JBoss Data Grid リモートおよび埋め込みモードでのみ利用できます。インデックスのないクエリーには設定されたインデックス (「[Infinispan Query DSL ベースのクエリー](#)」を参照) は必要ありません。Hibernate Search や Lucene ベースの API はインデックスのないクエリーを使用できません。

21.2. INFINISPAN QUERY DSL を用いたクエリーの作成

新しいクエリー API は `org.infinispan.query.dsl` パッケージにあります。クエリーは、**`Search.getQueryFactory()`** を使用して取得される **`QueryFactory`** インスタンスを利用して作成されます。各 **`QueryFactory`** インスタンスは 1 つのキャッシュインスタンスにバインドされ、複数の並列クエリーを作成する場合に使用できるステートレスでスレッドセーフなオブジェクトです。

Infinispan Query DSL は以下の手順にしたがってクエリーを実行します。

1. クエリーは **`from(Class entityType)`** メソッドを呼び出して作成されます。これは、特定のキャッシュから指定のエンティティークラスのクエリーを作成する **`QueryBuilder`** オブジェクトを返します。
2. **`QueryBuilder`** は、DSL メソッドを呼び出して指定される検索基準と設定を累積します。また、構築を完了する **`QueryBuilder.build()`** メソッドを呼び出して **`Query`** オブジェクトを構築するために使用されます。**`QueryBuilder`** オブジェクトを使用して、入れ子のクエリー以外の複数のクエリーを同時に構築することはできませんが、後で再使用することができます。
3. **`Query`** オブジェクトの **`list()`** メソッドを呼び出してクエリーを実行し、結果を取得します。実行すると **`Query`** オブジェクトは再使用できなくなります。新しい結果を取得する必要がある場合は、**`QueryBuilder.build()`** を呼び出して新しいインスタンスを取得します。



重要

クエリーは単一のエンティティ型を対象とし、単一キャッシュの内容全体で評価されます。複数キャッシュでのクエリーの実行や、複数のエンティティ型を対象とするクエリーの作成はサポートされません。

21.3. INFINISPAN QUERY DSL ベースのクエリーの有効化

ライブラリーモードでは、Infinispan Query DSL ベースのクエリーの実行は Lucene ベースの API クエリーの実行とほぼ同じです。前提条件は次のとおりです。

- Infinispan Query に必要なすべてのライブラリーがクラスパスにある。詳細は『[Administration and Configuration Guide](#)』を参照してください。
- インデックス化が有効で、キャッシュに対して設定されている (任意)。詳細は、『[Administration and Configuration Guide](#)』を参照してください。

- アノテーションが付いた POJO キャッシュ値 (任意)。インデックス化が有効になっていない場合は POJO アノテーションは必要なく、設定されている場合は無視されます。インデックス化が有効になっていない場合、Hibernate Search のアノテーションが付いたフィールドのみではなく、JavaBeans の慣例にしたがうフィールドすべてが検索可能になります。

21.4. INFINISPAN QUERY DSL ベースのクエリーの実行

Infinispan Query DSL ベースのクエリーが有効になったら、DSL ベースのクエリーを実行するために **Search** から **QueryFactory** を取得します。

キャッシュの QueryFactory の取得

ライブラリーモードで、以下のように **QueryFactory** を取得します。

```
QueryFactory qf = org.infinispan.query.Search.getQueryFactory(cache)
```

DSL ベースのクエリーの構築

```
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

QueryFactory qf = Search.getQueryFactory(cache);
Query q = qf.from(User.class)
    .having("name").eq("John")
    .build();
List list = q.list();
assertEquals(1, list.size());
assertEquals("John", list.get(0).getName());
assertEquals("Doe", list.get(0).getSurname());
```

リモートクエリーをリモートクライアントサーバーモードで使用する場合は、**Search** は **org.infinispan.client.hotrod** パッケージにあります。詳細は「[Hot Rod Java クライアント経由のリモートクエリーの実行](#)」の例を参照してください。

複数の条件 (サブ条件を含む) をブール値演算子と組み合わせることも可能です。

複数条件の組み合わせ

```
Query q = qf.from(User.class)
    .having("name").eq("John")
    .and().having("surname").eq("Doe")
    .and().not(qf.having("address.street").like("%Tanzania%"))
    .or().having("address.postCode").in("TZ13", "TZ22"))
    .build();
```

このクエリー API は、ユーザーに Lucene クエリーオブジェクト構築の詳細内容を公開しないことで、クエリーの作成方法を簡素化します。また、リモート Hot Rod クライアントが利用できる利点もあります。

以下の例は、**Book** エンティティのクエリーの書き方を示しています。

Book エンティティのクエリー


```
import org.infinispan.query.Search;
import org.infinispan.query.dsl.*;

// get the DSL query factory, to be used for constructing the Query
// object:
QueryFactory qf = Search.getQueryFactory(cache);
// create a query for all the books that have a title which contains the
// word "engine":
Query query = qf.from(Book.class)
    .having("title").like("%engine%")
    .build();
// get the results
List<Book> list = query.list();
```

21.5. 射影クエリー

多くの場合で、完全なドメインオブジェクトを返す必要はなく、アプリケーションには属性の小さなサブセットのみが必要になります。射影クエリーでは、属性 (または属性のパス) の特定のサブセットを返すことができます。射影クエリーが使用されると、**Query.list()** はドメインエンティティー全体 (**List<Object>**) を返さずに、アレイの各エントリーが射影された属性に対応する **List<Object[]>** を返します。

射影クエリーを定義するには、以下の例のようにクエリーの構築時に **select(...)** メソッドを使用します。

書名と発行年の取得

```
// Match all books that have the word "engine" in their title or
// description
// and return only their title and publication year.
Query query = queryFactory.from(Book.class)
    .select(Expression.property("title"),
    Expression.property("publicationYear"))
    .having("title").like("%engine%")
    .or().having("description").like("%engine%")
    .build();

// results.get(0)[0] contains the first matching entry's title
// results.get(0)[1] contains the first matching entry's publication year
List<Object[]> results = query.list();
```

21.6. グループ化および集約操作

Infinispan Query DSL には、グループ化フィールドのセットを基にしてクエリー結果をグループ化する機能や、値のセットに集約関数を適用して各グループからの結果の集約を構築する機能があります。グループ化と集約は、射影クエリーとのみ使用できます。

グループ化フィールドのセットは、**groupBy(field)** メソッドを複数回呼び出して指定されます。グループ化フィールドの順番は関係ありません。

射影で選択されたすべての非グループ化フィールドは、以下に説明するグループ化関数の 1 つを使用して集約する必要があります。

著者による本のグループ化とその冊数

```
Query query = queryFactory.from(Book.class)
    .select(Expression.property("author"), Expression.count("title"))
    .having("title").like("%engine%")
    .groupBy("author")
    .build();

// results.get(0)[0] will contain the first matching entry's author
// results.get(0)[1] will contain the first matching entry's title
List<Object[]> results = query.list();
```

集約操作

以下の集約操作を指定のフィールドで実行できます。

- **avg()** - **Number** のセットの平均を算出し、**Double** として表します。null 以外の値がない場合、結果は null になります。
- **count()** - null でない行の数を **Long** として返します。null 以外の値がない場合、結果は 0 になります。
- **max()** - 指定のフィールドで見つかった最も大きな値と、適用されたフィールドと同じ戻り値の型を返します。null 以外の値がない場合は、結果は null になります。



注記

指定のフィールドの値は、**Comparable** 型である必要があります。そうでないと **IllegalStateException** が発生します。

- **min()** - 指定のフィールドで見つかった最も小さな値と、適用されたフィールドと同じ戻り値の型を返します。null 以外の値がない場合は、結果は null になります。



注記

指定のフィールドの値は、**Comparable** 型である必要があります。そうでないと **IllegalStateException** が発生します。

- **sum()** - **Number** のセットの合計を算出し、返します。戻り値の型は指定のフィールド型によります。null で以外の値がない場合、結果は null になります。
以下の表は、指定のフィールドを基にした戻り値の型を表しています。

表21.1 戻り値の型の合計

フィールド型	戻り値の型
Integral (BigInteger 以外)	Long
Floating Point	Double
BigInteger	BigInteger
BigDecimal	BigDecimal

射影クエリーの特別なケース

射影クエリーでは、以下のケースは特別なユースケースとなります。

- 選択されたフィールドがすべて集約され、グループ化に何も使用されないことが適切である射影クエリー。この場合、集約はグループごとに算出されず、グローバルに算出されます。
- フィールドのグループ化を集約で利用できる。これは、集約が単一のデータポイント上で算出され、値が現在のグループに属する退化したケースになります。
- グループ化フィールドのみを選択し、集約フィールドは選択しないことが適切であるクエリー。

グループ化および集約クエリーの評価

通常のクエリーのように、集約クエリーにフィルター条件を含めることができます。これは、任意でグループ化操作の前後に実行できます。

groupBy メソッドの呼び出し前に指定されたすべてのフィルター条件は、グループ化操作の実行前に直接キャッシュエントリに適用されます。これらのフィルター条件は、クエリーされたエンティティ型のプロパティーを参照することがあり、後でグループ化に使用されるデータセットを制限するためのものです。

groupBy メソッドの呼び出し後に指定されたすべてのフィルター条件は、グループ化操作によって生じる射影に適用されます。これらのフィルター条件は、**groupBy** によって指定されたフィールドまたは集約フィールドを参照できます。**select** 句に指定されていない集約フィールドの参照は許可されますが、非集約フィールドや非グループ化フィールドの参照は禁止されます。このフェーズをフィルターすると、プロパティーを基にしてグループの数が削減されます。

順序付けも通常のクエリーと同様に指定できます。順序付け操作はグループ化操作の後に実行され、前述のとおりグループ化後のフィルターによって許可されるフィールドを参照できます。

21.7. 名前付きパラメーターの使用

各リクエストに新しいクエリーを作成する代わりに、各実行で置き換えられるパラメーターを含むことが可能です。これにより、クエリーを1度に定義でき、必要時にクエリーの変数を調整できます。

having(...) の比較演算子の右側に **Expression.param(...)** 演算子を使用すると、クエリーの作成時パラメーターが定義されます。

名前付きパラメーターの定義

```
import org.infinispan.query.Search;
import org.infinispan.query.dsl.*;
[...]
```

```
QueryFactory queryFactory = Search.getQueryFactory(cache);
// Defining a query to search for various authors
Query query = queryFactory.from(Book.class)
    .select("title")
    .having("author").eq(Expression.param("authorName"))
    .build()
[...]
```

名前付きパラメーターの値設定

デフォルトでは、宣言されたパラメーターはすべて null となり、クエリーの実行前に定義されたすべてのパラメーターを null でない値に更新する必要があります。パラメーターの宣言後、クエリーで新しい値を指定して **setParameter(parameterName, value)** または **setParameters(parameterMap)** を呼び出すと値を更新することができます。また、クエリーを再構築する必要はありません。新しいパラメーターが定義された後に再度実行することができます。

パラメーターを個別に更新

```
[...]
query.setParameter("authorName", "Smith");

// Rerun the query and update the results
resultList = query.list();
[...]
```

パラメーターのマップとして更新

```
[...]
parameterMap.put("authorName", "Smith");

query.setParameters(parameterMap);

// Rerun the query and update the results
resultList = query.list();
[...]
```

第22章 ICKLE クエリー言語を使用したクエリーの構築

22.1. ICKLE クエリー言語を使用したクエリーの構築

フルテキスト拡張を持つ JP-QL の軽量小型のサブセットである Ickle を使用すると、ライブラリーモードとリモートクライアントサーバーモードの両方でリレーショナルおよびフルテキストクエリーを作成することができます。Ickleは文字列ベースのクエリー言語で、以下の特徴があります。

- Java クラスをクエリーし、Protocol Buffers をサポートします。
- クエリーは単一のエンティティー型を対象にすることができます。
- クエリーはコレクションを含む、埋め込みオブジェクトのプロパティーでフィルターできます。
- 射影、集約、ソート、名前付きパラメーターをサポートします。
- インデックス化された実行とインデックス化されていない実行をサポートします。
- 複雑なブール式をサポートします。
- フルテキストクエリーをサポートします。
- `user.age > sqrt(user.shoeSize+3)` など、式の中での計算はサポートしません。
- 結合をサポートしません。
- サブクエリーをサポートしません。
- 多くの JBoss Data Grid API でサポートされます。継続的クエリーやリスナーのイベントフィルターなど、**QueryBuilder** によって作成されたクエリーは許可されます。

API を使用するには、最初に **QueryFactory** をキャッシュで取得し、**.create()** メソッドを呼び出してクエリーで使用する文字列を渡します。以下に例を示します。

```
QueryFactory qf = Search.getQueryFactory(remoteCache);
Query q = qf.create("from sample_bank_account.Transaction where amount > 20");
```

Ickle を使用する場合、フルテキスト演算子と使用されるすべてのフィールドはインデックス化され、分析される必要があります。

22.2. LUCENE クエリーパーサー構文との違い

Ickle は JP-QL のサブセットですが、クエリー構文に以下の違いがあります。

- 空白は重要ではありません。
- フィールド名のワイルドカードをサポートしません。
- デフォルトのフィールドがないため、フィールド名またはパスを常に指定する必要があります。
- フルテキストおよび JPA 述語では、**AND** と **OR** の代わりに **&&** と **||** が許可されます。

- **NOT** の代わりに **!** を使用できます。
- 見つからないブール演算子は **OR** として解釈されます。
- 文字列の単語を一重引用符または二重引用符のいずれかで囲む必要があります。
- ファジーとブーストは任意の順番では許可されません。常にファジーが最初になります。
- **<>** の代わりに **!=** が許可されます。
- ブーストは **>**、**>=**、**<**、および **≠** 演算子には適用できません。範囲を使用すると同じ結果を得られます

22.3. ファジークエリー

ファジークエリーを実行するには、単語の後に **~** と、単語からの距離を表す整数を追加します。例を以下に示します。

```
Query fuzzyQuery = qf.create("from sample_bank_account.Transaction where
description : 'cofee'~2");
```

22.4. 範囲クエリー

範囲クエリーを実行するには、以下の例のように境界を角括弧で囲んで定義します。

```
Query rangeQuery = qf.create("from sample_bank_account.Transaction where
amount : [20 to 50]");
```

22.5. フレーズクエリー

単語の組み合わせを検索するには、以下の例のように引用符で囲みます。

```
Query q = qf.create("from sample_bank_account.Transaction where
description : 'bus fare'");
```

22.6. 近接クエリー

近接クエリーを実行し、特定の距離内にある 2 つの単語を見つけるには、それらの言葉の後に **~** とその距離を追加します。たとえば以下の例では、間に 3 つ以下の言葉が存在する **canceling** および **fee** を見つけます。

```
Query proximityQuery = qf.create("from sample_bank_account.Transaction
where description : 'canceling fee'~3 ");
```

22.7. ワイルドカードクエリー

単一文字および複数文字のワイルドカード検索を実行できます。

- 単一文字のワイルドカード検索には **?** 文字を使用できます。
- 複数文字のワイルドカード検索には ***** 文字を使用できます。

text または **test** を検索するには、以下のように単一文字のワイルドカード検索を使用できます。

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction  
where description : 'te?t'");
```

test、**tests**、または **tester** を検索するには、以下のように複数文字のワイルドカード検索を使用できます。

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction  
where description : 'test*'");
```

22.8. 正規表現クエリー

正規表現クエリーを実行するには、/ の間にパターンを指定します。Ickle は Lucene の正規表現構文を使用するため、以下を使用して **moat** または **boat** を検索できます。

```
Query regexpQuery = qf.create("from sample_library.Book where title :  
/[mb]oat/");
```

22.9. ブーストクエリー

単語の後に ^ を追加し、クエリーでの関連度を高めると、単語をブーストすることができます。ブースト係数が高いほど単語の関連度も高くなります。たとえば、**beer** と **wine** が含まれる書名を検索し、**beer** の係数が 3 大きい場合、以下を使用できます。

```
Query boostedQuery = qf.create("from sample_library.Book where title :  
beer^3 OR wine");
```

第23章 リモートクエリー

23.1. リモートクエリー

Red Hat JBoss Data Grid の Hot Rod プロトコルは、Infinispan Query Domain-specific Language (DSL) または Ickle (JP-QL のサブセット) のいずれかを使用して、リモートの言語中立のクエリーを実現します。どちらを使用した場合でも、リモートの言語中立のクエリーを実行でき、Hot Rod クライアントで現在使用可能なすべての言語に実装できます。

Infinispan Query Domain-specific Language (DSL)

JBoss Data Grid は、内部の DSL を基に独自のクエリー言語を使用します。Infinispan DSL は簡単にクエリーを作成できる方法を提供し、基盤のクエリーメカニズムには依存しません。Infinispan Query DSL の詳細情報は「[Infinispan Query DSL](#)」を参照してください。

Ickle

Ickle は、フルテキストおよびリレーショナル検索を実行できる文字別ベースのクエリー言語です。Ickleの詳細は、「[Building a Query using Ickle, the JP-QL API](#)」を参照してください。

Protobuf エンコーディング

Google の Protocol Buffers は、データの保存およびクエリーのエンコーディング形式として使用されます。Infinispan Query DSL は、Protobuf マーシャラーを使用するように設定された Hot Rod クライアント経由でリモートで使用できます。Protocol Buffers は、キャッシュエントリーの保存とマーシャリングの共通形式を採用するために使用されます。保存されたエンティティのインデックス化およびクエリーが必要なリモートクライアントは、Protobuf エンコーディング形式を使用する必要があります。また、インデックス化が必要でない場合はインデックス化を有効にせずにプラットフォームの独立性を確保して Protobuf エンティティを保存することも可能です。

23.2. クエリーの比較

ライブラリーモードでは、Lucene クエリーベースのクエリーと DSL クエリーの両方を利用できます。リモートクライアントサーバーモードでは、DSL を使用したリモートクエリーのみを使用できます。以下の表は、Lucene クエリーベースのクエリー、Infinispan Query DSL、およびリモートクエリーを比較しています。

表23.1 埋め込みクエリーおよびリモートクエリー

機能	ライブラリー モード/Lucene クエリー	ライブラリー モード/DSL ク エリー	リモートクラ イアントサー バーモー ド/DSL クエ リー	ライブラリー モード/Ickle ク エリー	リモートクラ イアントサー バーモー ド/Ickle クエ リー
インデックス 化	必須	任意、しかし 強く推奨	任意、しかし 強く推奨	任意、しかし 強く推奨	任意、しかし 強く推奨
インデックス の内容	選択された フィールド	選択された フィールド	選択された フィールド	選択された フィールド	選択された フィールド
データスト レージ形式	Java オブジェ クト	Java オブジェ クト	Protocol Buffers	Java オブジェ クト	Protocol Buffers

機能	ライブラリー モード/Lucene クエリー	ライブラリー モード/DSL ク エリー	リモートクラ イアントサー バーモー ド/DSL クエ リー	ライブラリー モード/lckle ク エリー	リモートクラ イアントサー バーモー ド/lckle クエ リー
キーワードク エリー	可	不可	不可	可	可
範囲クエリー	可	可	可	可	可
ファジークエ リー	可	不可	不可	可	可
ワイルドカード	可	like クエリーに 限定 (JPA ルー ルに従うワイ ルドカードパ ターンと一致)	like クエリーに 限定 (JPA ルー ルに従うワイ ルドカードパ ターンと一致)	可	可
フレーズクエ リー	可	不可	不可	可	可
クエリーの組 み合わせ	AND、OR、 NOT、 SHOULD	AND、OR、 NOT	AND、OR、 NOT	AND、OR、 NOT	AND、OR、 NOT
結果のソート	可	可	可	可	可
結果のフィル ター	可 (クエリー内 および先頭に 追加された演 算子)	クエリー内	クエリー内	クエリー内	クエリー内
結果のページ ネーション	可	可	可	可	可
継続的クエ リー	不可	可	可	不可	不可
クエリーの集 約操作	不可	可	可	可	可

23.3. HOT ROD JAVA クライアント経由のリモートクエリーの実行

RemoteCacheManager が Protobuf マーシャラーと設定された後、Hot Rod 上のリモートクエリーを有効にできます。

以下の手順では、キャッシュでのリモートクエリーを有効にする方法を説明します。

前提条件

Protobuf マーシャラーを使用するよう、**RemoteCacheManager** を設定する必要があります。

Hot Rod 経由のリモートクエリーの有効化

1. infinispan-remote.jar の追加

infinispan-remote.jar は uberjar であるため、この機能に必要な他の依存関係はありません。

2. キャッシュ設定でのインデックス化の有効化

リモートクエリーでは、インデックス化は必須ではありませんが、大量のデータが含まれるキャッシュでの検索が大幅に高速化されるため、強く推奨されます。インデックス化はいつでも設定できます。インデックス化の有効化や設定は、Library モードと同じです。

Infinispan サブシステム要素内にある **cache-container** 要素内に以下の設定を追加します。

```
<!-- A basic example of an indexed local cache
      that uses the RAM Lucene directory provider -->
<local-cache name="an-indexed-cache">
  <!-- Enable indexing using the RAM Lucene directory provider -->
  <indexing index="ALL">
    <property name="default.directory_provider">ram</property>
  </indexing>
</local-cache>
```

3. Protobuf スキーマ定義ファイルの登録

Protobuf スキーマ定義ファイルを登録するには、これらのファイルを **__protobuf_metadata** システムキャッシュに追加します。キャッシュキーはファイル名を意味する文字列で、値は文字列としての **.proto** ファイルです。この代わりに、サーバーの **ProtobufMetadataManager** MBean の **registerProtofile** メソッドを呼び出して Protobuf スキーマを登録することもできます。キャッシュコンテナごとにこの MBean の 1 つのインスタンスがあり、**__protobuf_metadata** によってサポートされるため、この 2 つの方法は同等です。

__protobuf_metadata システムキャッシュ経由で Protobuf スキーマを提供する例は、「[Protocol Buffers スキーマファイルの登録](#)」を参照してください。



注記

__protobuf_metadata キャッシュへの書き込みには、書き込みを実行するユーザーに **__schema_manager** ロールを追加する必要があります。

以下の例は、**ProtobufMetadataManager** MBean の **registerProtofile** メソッドを呼び出す方法を表しています。

JMX での Protobuf スキーマ定義ファイルの登録

```
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXServiceURL;

...
```

```

String serverHost = ...           // The address of your JDG server
int serverJmxPort = ...           // The JMX port of your server
String cacheContainerName = ...   // The name of your cache container
String schemaFileName = ...       // The name of the schema file
String schemaFileContents = ...   // The Protobuf schema file contents

JMXConnector jmxConnector = JMXConnectorFactory.connect(new
JMXServiceURL(
    "service:jmx:remoting-jmx://" + serverHost + ":" +
serverJmxPort));
MBeanServerConnection jmxConnection =
jmxConnector.getMBeanServerConnection();

ObjectName protobufMetadataManagerObjName =
    new ObjectName("jboss.infinispan:type=RemoteQuery,name=" +
ObjectName.quote(cacheContainerName) +
    ",component=ProtobufMetadataManager");

jmxConnection.invoke(protobufMetadataManagerObjName,
    "registerProtofile",
    new Object[]{schemaFileName,
schemaFileContents},
    new String[]{String.class.getName(),
String.class.getName()});
jmxConnector.close();

```

結果

キャッシュに置かれたすべてのデータは、インデックス化が使用されているかどうかに関わらず、即座に検索可能になります。埋め込みクエリーのように、エントリーにアノテーションを付ける必要はありません。エンティティークラスは Java クラスのみに意味があり、サーバー側には存在しません。

リモートのキューが有効になったら、以下を使用して **QueryFactory** を取得できます。

QueryFactory の取得

```

import org.infinispan.client.hotrod.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;
import org.infinispan.query.dsl.SortOrder;
...
remoteCache.put(2, new User("John", 33));
remoteCache.put(3, new User("Alfred", 40));
remoteCache.put(4, new User("Jack", 56));
remoteCache.put(4, new User("Jerry", 20));

QueryFactory qf = Search.getQueryFactory(remoteCache);
Query query = qf.from(User.class)
    .orderBy("age", SortOrder.ASC)
    .having("name").like("J%")
    .and().having("age").gte(33)
    .build();

List<User> list = query.list();
assertEquals(2, list.size());
assertEquals("John", list.get(0).getName());

```

```
assertEquals(33, list.get(0).getAge());
assertEquals("Jack", list.get(1).getName());
assertEquals(56, list.get(1).getAge());
```

これで、ライブラリーモードと同様にクエリーを Hot Rod 上で実行できるようになります。

23.4. HOT ROD C++ クライアントでのリモートクエリー

Hot Rod C++ クライアントでリモートクエリーを使用する手順は、「[Hot Rod C++ クライアントを用いたリモートクエリーの実行](#)」を参照してください。

23.5. HOT ROD C# クライアントでのリモートクエリー

Hot Rod C# クライアントでリモートクエリーを使用する手順は、「[Hot Rod C# クライアントを用いたリモートクエリーの実行](#)」を参照してください。

23.6. PROTOBUF エンコーディング

23.6.1. Protobuf エンコーディング

Infinispan Query DSL は Hot Rod 経由で使用できます。これには、Protocol Buffers を使用してキャッシュエントリーの保存およびマーシャリングに共通の形式を採用します。

詳細は、<https://developers.google.com/protocol-buffers/docs/overview> を参照してください。

23.6.2. Protobuf エンコードされたエンティティの格納

Protobuf ではデータを構造化する必要があります。これには、**.proto** ファイルで Protocol Buffer の型を宣言します。

例を以下に示します。

.library.proto

```
package book_sample;
message Book {
    required string title = 1;
    required string description = 2;
    required int32 publicationYear = 3; // no native Date type available
in Protobuf

    repeated Author authors = 4;
}
message Author {
    required string name = 1;
    required string surname = 2;
}
```

この例では以下が実行されます。

1. **Book** という名前のエンティティが **book_sample** という名前のパッケージに配置されます。

```
package book_sample;
message Book {
```

- エンティティは、プリミティブ型の複数のフィールドと **authors** という名前の繰り返し可能なフィールドを宣言します。

```
    required string title = 1;
    required string description = 2;
    required int32 publicationYear = 3; // no native Date type
    available in Protobuf

    repeated Author authors = 4;
}
```

- Author** メッセージインスタンスは **Book** メッセージインスタンスに埋め込みされます。

```
message Author {
    required string name = 1;
    required string surname = 2;
}
```

23.6.3. Protobuf メッセージ

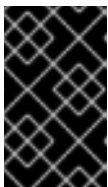
Protobuf メッセージに関する重要事項は次のとおりです。

- メッセージを入れ子にすることは可能ですが、結果として構造は常にツリーになり、グラフにはなりません。
- 型の継承はありません。
- コレクションはサポートされませんが、繰り返されるフィールドを使用してアレイを簡単にエミュレートできます。

23.6.4. Hot Rod での Protobuf の使用

Protobuf を JBoss Data Grid の Hot Rod と使用するには、以下の 2 つのステップを使用します。

- 専用のマーシャラー (ここでは **ProtoStreamMarshaller**) を使用するようクライアントを設定します。このマーシャラーは **ProtoStream** ライブラリーを使用して、オブジェクトのエンコーディングを手助けします。



重要

infinispan-remote jar が使用されていない場合、**infinispan-remote-query-client** Maven 依存関係を追加して **ProtoStreamMarshaller** を使用する必要があります。

- エンティティごとのマーシャラーを登録し、**ProtoStream** ライブラリーにメッセージ型のマーシャル方法を指示します。

ProtoStreamMarshaller を使用したメッセージのエンコードおよびマーシャル

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
```

```
import org.infinispan.client.hotrod.marshall.ProtoStreamMarshaller;
import org.infinispan.protostream.FileDescriptorSource;
import org.infinispan.protostream.SerializationContext;
...
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1").port(11234)
    .marshaller(new ProtoStreamMarshaller());

RemoteCacheManager remoteCacheManager = new
RemoteCacheManager(clientBuilder.build());
SerializationContext serCtx =
    ProtoStreamMarshaller.getSerializationContext(remoteCacheManager);
serCtx.registerProtoFiles(FileDescriptorSource.fromResources("/library.proto"));
serCtx.registerMarshaller(new BookMarshaller());
serCtx.registerMarshaller(new AuthorMarshaller());
// Book and Author classes omitted for brevity
```

この例では以下が実行されます。

- **SerializationContext** が **ProtoStream** ライブラリーによって提供されます。
- **SerializationContext.registerProtofile** メソッドは、メッセージ型の定義が含まれる **.proto** クラスパスリソースファイルの名前を受信します。
- **RemoteCacheManager** に関連する **SerializationContext** が取得され、**ProtoStream** は **protobuf** 型のマーシャルを指示されます。



注記

ProtoStreamMarshaller を使用するよう設定されていないと、**RemoteCacheManager** に関連する **SerializationContext** はありません。

23.6.5. エンティティーごとのマーシャラーの登録

リモートクエリーの目的で **ProtoStreamMarshaller** を使用する場合、各ドメインモデル型に対してエンティティーごとにマーシャラーを登録しないと、マーシャリングに失敗します。マーシャラーの1つのインスタンスが使用されるため、ステートレスでスレッドセーフなマーシャラーを作成する必要があります。

以下の例はマーシャラーの書き方を示しています。

BookMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;
...
public class BookMarshaller implements MessageMarshaller<Book> {
    @Override
    public String getTypeName() {
        return "book_sample.Book";
    }
    @Override
    public Class<? extends Book> getJavaClass() {
        return Book.class;
    }
}
```

```

    }
    @Override
    public void writeTo(ProtoStreamWriter writer, Book book) throws
IOException {
        writer.writeString("title", book.getTitle());
        writer.writeString("description", book.getDescription());
        writer.writeCollection("authors", book.getAuthors(),
Author.class);
    }
    @Override
    public Book readFrom(ProtoStreamReader reader) throws IOException {
        String title = reader.readString("title");
        String description = reader.readString("description");
        int publicationYear = reader.readInt("publicationYear");
        Set<Author> authors = reader.readCollection("authors",
            new HashSet<Author>(), Author.class);
        return new Book(title, description, publicationYear, authors);
    }
}

```

クライアントが設定されたら、リモートキャッシュへの Java オブジェクトの読み書きにはエンティティマーシャラーが使用されます。関係するすべての型のリモートクライアントでマーシャラーが登録された場合、キャッシュに格納される実際のデータは protobuf でエンコードされます。この例では、**Book** と **Author** になります。

protobuf 形式で格納されたオブジェクトは、異なる言語で書かれた互換性のあるクライアントと使用することができます。

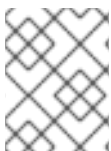
23.6.6. Protobuf エンコードされたエンティティのインデックス化

クライアントが Protobuf を使用するよう設定したら、JBoss Data Grid でキャッシュのインデックス化を設定できます。

キャッシュでエントリーをインデックス化するには、JBoss Data Grid は Protobuf スキーマに定義されたメッセージ型にアクセスする必要があります。Protobuf スキーマは **.proto** 拡張子が付いているファイルです。

put、**putAll**、**putIfAbsent**、または **replace** 操作で Protobuf スキーマを **__protobuf_metadata** キャッシュに置き、JBoss Data Grid に Protobuf スキーマを提供します。この代わりに、JMX 経由で **ProtobufMetadataManager** MBean を呼び出すこともできます。

__protobuf_metadata のキーと値は両方 String です。キーはファイル名で、値はスキーマファイルの内容になります。



注記

__protobuf_metadata キャッシュに書き込み操作を実行するユーザーは **__schema_manager** ロールが必要です。

Protocol Buffers スキーマファイルの登録

```

import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import

```



```
org.infinispan.query.remote.client.ProtobufMetadataManagerConstants;

RemoteCacheManager remoteCacheManager = ... // obtain a RemoteCacheManager

// obtain the '__protobuf_metadata' cache
RemoteCache<String, String> metadataCache =
    remoteCacheManager.getCache(
        ProtobufMetadataManagerConstants.PROTOBUF_METADATA_CACHE_NAME);

String schemaFileContents = ... // this is the contents of the schema file
metadataCache.put("my_protobuf_schema.proto", schemaFileContents);
```

ProtobufMetadataManager は、Protobuf スキーマ定義または [path].**proto** ファイルのクラスター全体でレプリケートされたりポジトリです。実行中のキャッシュマネージャーごとに個別の **ProtobufMetadataManager** MBean インスタンスが存在し、**__protobuf_metadata** キャッシュによってサポートされます。**ProtobufMetadataManager** ObjectName は以下のパターンを使用します。

```
<jmx domain>:type=RemoteQuery,
    name=<cache manager<methodname>putAllname>,
    component=ProtobufMetadataManager
```

以下の署名は、Protobuf スキーマファイルを登録するメソッドによって使用されます。

```
void registerProtofile(String name, String contents)
```

キャッシュのインデックス化が有効になっている場合、Protobuf エンコードされたエントリーのすべてのフィールドがインデックス化されます。Protobuf エンコードされたすべてのエントリーは、インデックス化が有効になっているかどうかに関わらず検索可能です。



注記

インデックス化はパフォーマンスの向上のために推奨されますが、リモートクエリーの使用時には必須ではありません。インデックス化を使用すると検索速度が向上しますが、インデックスの維持に必要なオーバーヘッドにより挿入および更新の速度が低下します。

23.6.7. Protobuf によるカスタムフィールドのインデックス化

デフォルトでは、キャッシュのインデックス化を有効にした後、すべての Protobuf 型フィールドはインデックス化および格納されます。ほとんどの場合でこのインデックス化は適していますが、多くのフィールドまたは非常に大きなフィールドを持つ Protobuf メッセージ型を処理する場合は、パフォーマンスが劣化したり、十分でない可能性があります。

そのため、**@Indexed** および **@Field** アノテーションを直接コメント定義内の Protobuf スキーマで使用して、どのフィールドがインデックス化されるかを制御する必要があります。

インデックス化されるフィールドの指定

```
/*
    This type is indexed, but not all its fields are.
    @Indexed
*/
message Note {
```



```

/*
   This field is indexed but not stored. Since these are the default
values no annotation attributes are defined.
   It can be used for querying but not for projections.
   @Field
*/
optional string text = 1;

/*
   A field that is both indexed and stored.
   @Field(store=Store.YES)
*/
optional string author = 2;

/* @Field(index=Index.NO, store=Store.YES) */
optional bool isRead = 3;

/* This field is not annotated, so it is neither indexed nor stored. */
optional int32 priority;
}

```

アノテーションは、アノテーションが付けられる要素 (メッセージまたはフィールドのいずれか) の前にあるコメントの最終行に付けられます。

@Indexed アノテーションに関する説明は次のとおりです。

- メッセージ型のみに適用され、デフォルトが **true** であるブール値を持ちます。そのため、**@Indexed** の使用は **@Indexed(true)** と同等になります。
- このアノテーションは、インデックス化されたメッセージ型のフィールドの指定を可能にします。**@Indexed(false)** の使用はインデックス化されるフィールドがないことを示します。そのため、**@Field** アノテーションは無視されます。

@Field アノテーションに関する説明は次のとおりです。

- フィールドのみに適用され、**index**、**store**、および **analyze** の 3 つの属性を持ちます。
- デフォルトは **@Field(index=Index.YES, store=Store.NO, analyze=Analyze.NO)** です。これらの属性は以下の定義を持ちます。
 - **index** 属性はフィールドがインデックスされるべきであるかどうかを示し、インデックス化されたクエリーでの使用を許可します。
 - **store** 属性はフィールドがインデックスに格納されるべきかどうかを示し、プロジェクションでの使用を許可します。
 - **analyze** 属性は、フルテキスト検索にフィールドを使用できるかどうかを示します。

すべての Protobuf メッセージ型のインデックス化を無効化

アノテーションが付けられていないすべての Protobuf メッセージ型のインデックス化を無効にすることができます。各スキーマファイルの最初にある **indexed_by_default** Protobuf スキーマオプションの値を次のように **false** に設定します。

```
option indexed_by_default = false; //Disable indexing of all types that
are not annotated for indexing.
```

23.6.8. Java アノテーションでの Protocol Buffers スキーマの定義

Java アノテーションを使用して Protobuf メタデータを宣言できます。**MessageMarshaller** 実装と **.proto** スキーマファイルを提供する代わりに、最小限のアノテーションを Java クラスとそのフィールドに追加できます。

この方法の目的は、**ProtoStream** ライブラリーを使用して Java obyekuto を protobuf にマーシャルすることです。**ProtoStream** ライブラリーは内部でマーシャラーを生成し、手動で実装されたマーシャラーは必要ありません。Java アノテーションには、Protobuf タグ番号などの最低限の情報が必要になります。その他は常識的なデフォルトを基に推定され (Protobuf 型、Java コレクション型、およびコレクション要素型)、オーバーライドすることができます。

自動生成されたスキーマは **SerializationContext** で登録され、他の言語のドメインモデルクラスおよびマーシャラーを実装するための参照として使用することもできます。

Java アノテーションの例を以下に示します。

User.Java

```
package sample;

import org.infinispan.protostream.annotations.ProtoEnum;
import org.infinispan.protostream.annotations.ProtoEnumValue;
import org.infinispan.protostream.annotations.ProtoField;
import org.infinispan.protostream.annotations.ProtoMessage;

@ProtoMessage(name = "ApplicationUser")
public class User {

    @ProtoEnum(name = "Gender")
    public enum Gender {
        @ProtoEnumValue(number = 1, name = "M")
        MALE,

        @ProtoEnumValue(number = 2, name = "F")
        FEMALE
    }

    @ProtoField(number = 1, required = true)
    public String name;

    @ProtoField(number = 2)
    public Gender gender;
}
```

Note.Java

```
package sample;

import org.infinispan.protostream.annotations.ProtoDoc;
import org.infinispan.protostream.annotations.ProtoField;
```

```

@ProtoDoc("@Indexed")
public class Note {

    private String text;

    private User author;

    @ProtoDoc("@Field")
    @ProtoField(number = 1)
    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    @ProtoDoc("@Field(store = Store.YES)")
    @ProtoField(number = 2)
    public User getAuthor() {
        return author;
    }

    public void setAuthor(User author) {
        this.author = author;
    }
}

```

ProtoSchemaBuilderDemo.Java

```

import org.infinispan.protostream.SerializationContext;
import org.infinispan.protostream.annotations.ProtoSchemaBuilder;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.marshall.ProtoStreamMarshaller;

...

RemoteCacheManager remoteCacheManager = ... // we have a RemoteCacheManager
SerializationContext serCtx =
    ProtoStreamMarshaller.getSerializationContext(remoteCacheManager);

// generate and register a Protobuf schema and marshallers based
// on Note class and the referenced classes (User class)
ProtoSchemaBuilder protoSchemaBuilder = new ProtoSchemaBuilder();
String generatedSchema = protoSchemaBuilder
    .fileName("sample_schema.proto")
    .packageName("sample_package")
    .addClass(Note.class)
    .build(serCtx);

// the types can be marshalled now
assertTrue(serCtx.canMarshall(User.class));
assertTrue(serCtx.canMarshall(Note.class));
assertTrue(serCtx.canMarshall(User.Gender.class));

```

```
// display the schema file
System.out.println(generatedSchema);
```

以下は、**ProtoSchemaBuilderDemo.java** の例によって生成される **.proto** ファイルになります。

Sample_Schema.Proto

```
package sample_package;

/* @Indexed */
message Note {

    /* @Field */
    optional string text = 1;

    /* @Field(store = Store.YES) */
    optional ApplicationUser author = 2;
}

message ApplicationUser {

    enum Gender {
        M = 1;
        F = 2;
    }

    required string name = 1;
    optional Gender gender = 2;
}
```

以下の表は、サポートされる Java アノテーションとそのアプリケーションおよびパラメーターを示しています。

表23.2 Java アノテーション

アノテーション	適用対象	目的	要件	Parameters
@ProtoDoc	クラス/フィールド/列挙型/列挙型メンバー	生成された Protobuf スキーマ要素にアタッチされるドキュメンテーションコメントを指定します (メッセージ型、フィールド定義、列挙型、列挙値定義)。	任意	単一の String パラメーター、ドキュメンテーションテキスト。

アノテーション	適用対象	目的	要件	Parameters
@ProtoMessage	クラス	生成されたメッセージ型の名前を指定します。指定がない場合はクラス名が使用されます。	任意	名前 (String)、生成されたメッセージ型の名前。指定がない場合はデフォルトで Java クラス名が使用されます。
@ProtoField	フィールド、ゲッターまたはセッター	Protobuf フィールド番号とその Protobuf 型を指定します。また、フィールドが繰り返し返されるかどうか、任意または必須であるかどうかを示し、任意でデフォルト値を示します。Java フィールド型がインターフェースまたは抽象クラスである場合、その実際の型を示す必要があります。フィールドが繰り返し可能で、宣言されたコレクション型が抽象である場合、実際のコレクション実装型を指定する必要があります。このアノテーションがない場合、マーシャリングでこのフィールドが無視されます (一時的です)。Protobuf がマーシャル可能と見なされるには、クラスに @ProtoField アノテーションが付けられたフィールドが 1 つ以上必要です。	必須	数 (int、必須)、Protobuf 番号型 (org.infinispan.protostream.descriptors.Type、任意)、Protobuf 型: 通常は必須と推測されます (ブール値、任意)、名前 (String、任意)、Protobuf namejava 型 (Class、任意)、実際の型: 宣言された型が抽象 collectionImplementation の場合のみ必要 (Class、任意)、実際のコレクション型: 宣言された型が抽象 default Value の場合のみ必要 (String、任意)、文字列は Java フィールド型に基づいた適切な形式である必要があります。

アノテーション	適用対象	目的	要件	Parameters
@ProtoEnum	列挙型	生成された列挙型の名前を指定します。指定がない場合は Java 列挙名が使用されます。	任意	名前 (String)、生成された列挙型の名前。指定がない場合、デフォルトで Java 列挙名が使用されます。
@ProtoEnumValue	列挙型メンバー	対応する Protobuf 列挙値の数を指定します。	必須	数 (int、必須)、Protobuf 番号名 (String)、Protobuf 名。指定のない場合、Java メンバーの名前が使用されます。



注記

@ProtoDoc アノテーションを使用すると、生成されたスキーマでドキュメンテーションコメントを提供できます。また、@Indexed および @Field アノテーションが必要な場所にインジェクトすることもできます。詳細は「[Protobuf でのカスタムフィールドのインデックス化](#)」を参照してください。

パート III. RED HAT JBOSS DATA GRID におけるデータのセキュア化

第24章 RED HAT JBOSS DATA GRID におけるデータのセキュア化

Red Hat JBoss Data Grid では、データセキュリティを以下の方法で実装できます。

ロールベースアクセス制御

JBoss Data Grid には、指定のセキュア化されたキャッシュ上の操作に対応するロールベースのアクセス制御機能が含まれます。ロールは、キャッシュおよびキャッシュマネージャー操作のパーミッションにマップされた状態で、アプリケーションにアクセスするユーザーに割り当てられます。認証されたユーザーのみがこれらのロールで許可されている操作を実行できます。

ライブラリースタイルモードでは、データは、認証がコンテナまたはアプリケーションに委譲された状態で CacheManager および Cache 用のロールベースのアクセス制御によってセキュア化されます。リモートクライアントサーバーモードでは、JBoss Data Grid は、Identity トークンを Hot Rod クライアントから、Cache および CacheManager のサーバーおよびロールベースのアクセス制御に渡すことによってセキュア化されます。

ノードの認証および承認

ノードレベルのセキュリティでは、新規ノードやマージされるパーティションがクラスターに参加する前に認証される必要があります。これを実行できるのは、クラスターへの参加が承認された認証済みのノードのみです。これは、不正なサーバーによるデータの保存を防ぐため、データが保護されます。

クラスター内の暗号化通信

JBoss Data Grid では、JCA (Java Cryptography Architecture、Java 暗号化アーキテクチャー) によってサポートされるユーザー指定の暗号化アルゴリズムを使用して、ノード間での通信の暗号化がサポートされるようになり、データのセキュリティが強化されました。

また、JBoss Data Grid は操作の監査ログ、および Transport Layer Security (TLS/SSL) を使用した Hot Rod クライアントとサーバー間の通信を暗号化する機能も提供します。

第25章 RED HAT JBOSS DATA GRID セキュリティー: 認証および承認

25.1. RED HAT JBOSS DATA GRID セキュリティー: 認証および承認

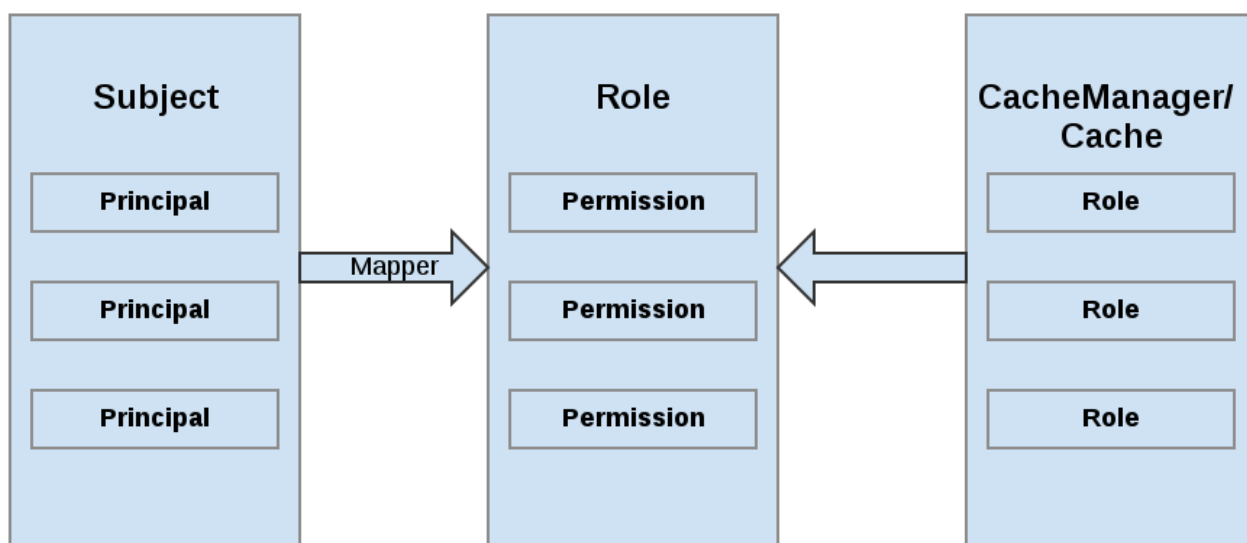
Red Hat JBoss Data Grid は、CacheManager および Cache での承認を実行できます。JBoss Data Grid 承認は、JAAS および SecurityManager などの JDK で利用できる標準的なセキュリティ機能に基づいています。

アプリケーションがセキュアな CacheManager および Cache と対話する場合、JBoss Data Grid のセキュリティレイヤーが必須ロールおよびパーミッションのセットに対して検証できるアイデンティティを指定する必要があります。検証されると、クライアントに後続操作のトークンが発行されます。アクセスが拒否されると、セキュリティ違反を示す例外が発生します。

キャッシュの承認が設定される場合、キャッシュが取得されると **SecureCache** のインスタンスが返されます。**SecureCache** はキャッシュの単純なラッパーであり、「現行ユーザー」が操作の実行に必要なパーミッションを持っているかどうかを確認します。「現行ユーザー」は **AccessControlContext** に関連する Subject です。

JBoss Data Grid はプリンシパル名を、1 つ以上のパーミッションを表すロールにマップします。以下の図はこれらの関係を示しています。

図25.1 ロールとパーミッションのマッピング



25.2. パーミッション

CacheManager または Cache へのアクセスは必要なパーミッションセットを使用して制御されます。パーミッションは、操作されているデータのタイプではなく、CacheManager または Cache で実行されるアクションのタイプを制御します。これらのパーミッションの一部は、名前付きキャッシュなどの名前エンティティーにとくに適用されます。エンティティーによって使用できるタイプのパーミッションが異なります。

表25.1 CacheManager パーミッション

パーミッション	関数	説明
CONFIGURATION	defineConfiguration	新規キャッシュ設定を定義できるかどうか。
LISTEN	addListener	リスナーをキャッシュマネージャーに対して登録できるかどうか。
LIFECYCLE	stop, start	キャッシュマネージャーを停止または開始できるかどうか。
ALL		上記すべてを含む便利なパーミッション。

表25.2 キャッシュパーミッション

パーミッション	関数	説明
READ	get, contains	エントリーをキャッシュから取得できるかどうか。
WRITE	put, putIfAbsent, replace, remove, evict	データをキャッシュから書き込み、置き換え、エビクトできるかどうか。
EXEC	distexec, mapreduce	コード実行をキャッシュに対して実行できるかどうか。
LISTEN	addListener	リスナーをキャッシュに対して登録できるかどうか。
BULK_READ	keySet, values, entrySet, query	一括取得操作を実行できるかどうか。
BULK_WRITE	g	一括書き込み操作を実行できるかどうか。
LIFECYCLE	start, stop	キャッシュを開始または停止できるかどうか。

パーミッション	関数	説明
ADMIN	getVersion, addInterceptor*, removeInterceptor, getInterceptorChain, getEvictionManager, getComponentRegistry, getDistributionManager, getAuthorizationManager, evict, getRpcManager, getCacheConfiguration, getCacheManager, getInvocationContextContainer, setAvailability, getDataContainer, getStats, getXAResource	基礎となるコンポーネントまたは内部構造へのアクセスが可能かどうか。
ALL		上記すべてを含む便利なパーミッション。
ALL_READ		READ と BULK_READ の組み合わせ。
ALL_WRITE		WRITE と BULK_WRITE の組み合わせ。



注記

使いやすさを強化するために一部のパーミッションを他のパーミッションに組み合わせることが必要になる場合があります。たとえば、EXEC を READ または WRITE と組み合わせることがあります。

25.3. ロールマッピング

Subject のプリンシパルを承認で使用される一連のロールに変換するには、**PrincipalRoleMapper** をグローバル設定に指定する必要があります。Red Hat JBoss Data Grid には 3 つのマッパーが含まれ、さらにカスタムマッパーを使用することもできます。

表25.3 マッパー

マッパー名	Java	XML	説明
IdentityRoleMapper	org.infinispan.security.impl.IdentityRoleMapper	<identity-role-mapper />	ロール名としてプリンシパル名を使用します。

マッパー名	Java	XML	説明
CommonNameRoleMapper	org.infinispan.security.impl.CommonRoleMapper	<common-name-role-mapper />	プリンシパル名が識別名 (DN) の場合、このマッパーは共通名 (CN) を抽出し、これをロール名として使用します。たとえば、DN cn=managers, ou=people, dc=example, dc=com はロールの managers にマップされます。
ClusterRoleMapper	org.infinispan.security.impl.ClusterRoleMapper	<cluster-role-mapper />	ClusterRegistry を使用してプリンシパルをロールマッピングに保存します。これにより、CLI の GRANT および DENY コマンドを使用してロールのプリンシパルへの追加または削除を実行できます。
Custom Role Mapper		<custom-role-mapper class="a.b.c" />	org.infinispan.security.impl.PrincipalRoleMapper の実装の完全修飾クラス名を指定します。

25.4. ログインモジュールを使用した認証およびロールマッピングの設定

LDAP からロールクエリー用の認証 **login-module** を使用する場合、カスタムクラスは使用中であるため、プリンシパルからロールへの独自のマッピングを実装する必要があります。以下の例は、**login-module** から取得したプリンシパルをロールにマップする方法を示しています。これは、ユーザープリンシパル名をロールにマッピングし、同様のアクションを **IdentityRoleMapper** に対して実行します。

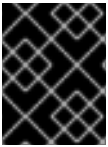
プリンシパルのマッピング

```
public class SimplePrincipalGroupRoleMapper implements PrincipalRoleMapper
{
    @Override
    public Set<String> principalToRoles(Principal principal) {
        if (principal instanceof SimpleGroup) {
            Enumeration<Principal> members = ((SimpleGroup)
principal).members();
            if (members.hasMoreElements()) {
                Set<String> roles = new HashSet<String>();
                while (members.hasMoreElements()) {
                    Principal innerPrincipal = members.nextElement();
                    if (innerPrincipal instanceof SimplePrincipal) {
```

```

        SimplePrincipal sp = (SimplePrincipal) innerPrincipal;
        roles.add(sp.getName());
    }
}
return roles;
}
}
return null;
}
}

```



重要

LDAP サーバーの設定や LDAP サーバーでのユーザーおよびロールの指定についての詳細は、Red Hat Directory Server の『**Administration Guide**』を参照してください。

25.5. RED HAT JBOSS DATA GRID の承認の設定

承認はキャッシュコンテナ (CacheManager) と単一キャッシュの 2 つのレベルで設定されます。

各キャッシュコンテナは以下を決定します。

- 承認を使用するかどうか。
- プリンシパルをルールセットにマップするクラス。
- 名前付きロールのセットとそれらが表すパーミッション。

コンテナレベルで定義されたロールのサブセットのみを使用することを選択できます。

ロール

ロールは、以下のようにキャッシュコンテナのレベルで定義されたロールを使用して、キャッシュごとに適用できます。



重要

認証が必要とされるキャッシュには、ロールの一覧が定義されている必要があります。そうでない場合、キャッシュの承認では非匿名ポリシーは定義されないため、認証は強制されません。

プログラムを使用した CacheManager 承認 (ライブラリーモード)

以下の例は、プログラムによる設定を使用して同じ承認パラメーターをライブラリーモードに対して設定する方法を示しています。

プログラムを使用した CacheManager 承認の設定

```

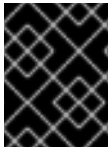
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
global
    .security()
    .authorization()
    .principalRoleMapper(new IdentityRoleMapper())
    .role("admin")
    .permission(CachePermission.ALL)

```

```

        .role("supervisor")
        .permission(CachePermission.EXEC)
        .permission(CachePermission.READ)
        .permission(CachePermission.WRITE)
        .role("reader")
        .permission(CachePermission.READ);
ConfigurationBuilder config = new ConfigurationBuilder();
config
    .security()
    .enable()
    .authorization()
    .role("admin")
    .role("supervisor")
    .role("reader");

```



重要

REST プロトコルの承認での使用はサポートされておらず、承認を有効にした状態でキャッシュにアクセスしようとするとき **SecurityException** が発生します。

25.6. ライブラリーモードのデータセキュリティ

25.6.1. サブジェクトおよびプリンシパルクラス

リソースへのアクセスを承認するには、アプリケーションが最初に要求元を認証する必要があります。JAAS フレームワークは、要求元を表す用語サブジェクトを定義します。**Subject** クラスは JAAS の重要なクラスです。**Subject** は人やサービスなどの、単一エンティティの情報を表します。これには、エンティティのプリンシパル、パブリックのクレデンシャル、プライベートのクレデンシャルなどが含まれます。JAAS API は既存の Java 2 **java.security.Principal** インターフェースを使用して、型指定された名前であるプリンシパルを表します。

認証プロセス中、サブジェクトには関連するアイデンティティまたはプリンシパルが入力されます。サブジェクトに複数のプリンシパルが含まれるようにすることができます。たとえば、一個人は名前のプリンシパル (John Doe)、ソーシャルセキュリティ番号のプリンシパル (123-45-6789)、ユーザー名のプリンシパル (johnd) を持つことができ、これらすべてのプリンシパルはサブジェクトを他のサブジェクトから区別するのに役立ちます。1 つのサブジェクトに関連するプリンシパルを取得する方法は 2 つあります。

```

public Set getPrincipals() {...}
public Set getPrincipals(Class c) {...}

```

getPrincipals() はサブジェクトに含まれるすべてのプリンシパルを返します。**getPrincipals(Class c)** は、クラス **c** のインスタンスであるプリンシパルまたはそのサブクラスの 1 つのみを返します。サブジェクトに一致するプリンシパルがない場合は、空のセットが返されます。



注記

java.security.acl.Group インターフェースは **java.security.Principal** のサブインターフェースであるため、プリンシパルのセットのインスタンスは、他のプリンシパルやプリンシパルのグループの論理グループを表すことがあります。

25.6.2. サブジェクトの取得

ライブラリーモードでセキュア化されたキャッシュを使用するには、**javax.security.auth.Subject** を取得する必要があります。Subject は、人やサービスなどの単一のキャッシュエンティティの情報を表します。

Red Hat JBoss Data Grid では、コンテナの機能またはサードパーティーライブラリーのどちらかをを使用して JAAS Subject を取得できます。

JBoss コンテナの場合は以下を使用して取得します。

```
Subject subject = SecurityContextAssociation.getSubject();
```

Subject には、LDAP や Active Directory などのセキュリティドメインで属するユーザーおよびグループを表すプリンシパルのセットを指定する必要があります。

Java EE API では、以下のメソッドを用いてコンテナセットのプリンシパルを取得できます。

- サーブレット: **ServletRequest.getUserPrincipal()**
- EJB: **EJBContext.getCallerPrincipal()**
- MessageDrivenBeans: **MessageDrivenContext.getCallerPrincipal()**

mapper は、Subject と関連するプリンシパルを識別し、コンテナレベルで定義したものに対応するロールへの変換に使用されます。

プリンシパルは Subject のコンポーネントの 1 つにすぎず、**java.security.AccessControlContext** から取得されます。コンテナが **AccessControlContext** の Subject を設定するか、**Security.doAs()** メソッドを使用して呼び出しを JBoss Data Grid API にラッピングする前にユーザーがプリンシパルを適切なサブジェクトへマップする必要があります。

Subject が取得されると、キャッシュは **PrivilegedAction** のコンテキスト内で対話できます。

サブジェクトの取得

```
import org.infinispan.security.Security;

Security.doAs(subject, new PrivilegedExceptionAction<Void>() {
    public Void run() throws Exception {
        cache.put("key", "value");
    }
});
```

Security.doAs() メソッドは、通常の **Subject.doAs()** メソッドの代わりになります。アプリケーションのセキュリティモデルに固有する理由で **AccessControlContext** を変更しなければならない場合以外は、**Security.doAs()** を使用するとパフォーマンスが向上します。

現在の Subject を取得するには、Subject を JBoss Data Grid コンテキストまたは **AccessControlContext** から取得する **Security.getSubject();** を使用します。

25.6.3. サブジェクトの認証

サブジェクトの認証には JAAS ログインが必要です。ログイン手順は次のようになります。

1. アプリケーションは **LoginContext** をインスタンス化し、ログイン設定の名前と **CallbackHandler** を渡して、設定 **LoginModule** が必要とする **Callback** オブジェクトを追加します。
2. **LoginContext** は、名前付きログイン設定に含まれるすべての **LoginModules** をロードするため **Configuration** を確認します。このような名前付き設定が存在しない場合は、**other** 設定がデフォルトで使用されます。
3. アプリケーションは **LoginContext.login** メソッドを呼び出します。
4. ログインメソッドはロードされたすべての **LoginModule** を呼び出します。各 **LoginModule** はサブジェクトを認証するため、関連する **LoginModule** でハンドルメソッドを呼び出し、認証プロセスに必要な情報を取得します。必要な情報は、**Callback** オブジェクトの配列の形式でハンドルメソッドに渡されます。認証に成功すると、**LoginModule** は関連のプリンシパルとクレデンシャルをサブジェクトに関連付けします。
5. **LoginContext** は認証ステータスをアプリケーションに返します。ログインメソッドから返されると認証が成功したことになります。ログインメソッドによって **LoginException** が発生すると認証に失敗したことになります。
6. 認証に成功すると、アプリケーションは **LoginContext.getSubject** メソッドを使用して認証されたサブジェクトを取得します。
7. サブジェクトの認証が完了した後に **LoginContext.logout** メソッドを呼び出すと、**login** メソッドによりサブジェクトに関連付けられたすべてのプリンシパルおよび関連情報を削除できます。

LoginContext クラスは、サブジェクト認証の基本メソッドを提供し、基礎となる認証技術に依存しないアプリケーションの開発方法を提供します。**LoginContext** は、特定のアプリケーション向けに設定された認証サービスを判断するため **Configuration** を確認します。**LoginModule** クラスは認証サービスを表します。そのため、アプリケーション自体を変更しなくても、異なるログインモジュールをアプリケーションにプラグ可能です。次のコードは、アプリケーションによるサブジェクトの認証で必要になる手順を示しています。

```
CallbackHandler handler = new MyHandler();
LoginContext lc = new LoginContext("some-config", handler);

try {
    lc.login();
    Subject subject = lc.getSubject();
} catch(LoginException e) {
    System.out.println("authentication failed");
    e.printStackTrace();
}

// Perform work as authenticated Subject
// ...

// Scope of work complete, logout to remove authentication info
try {
    lc.logout();
} catch(LoginException e) {
    System.out.println("logout failed");
    e.printStackTrace();
}
```



```
// A sample MyHandler class
class MyHandler
    implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws
        IOException, UnsupportedCallbackException
    {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof NameCallback) {
                NameCallback nc = (NameCallback)callbacks[i];
                nc.setName(username);
            } else if (callbacks[i] instanceof PasswordCallback) {
                PasswordCallback pc = (PasswordCallback)callbacks[i];
                pc.setPassword(password);
            } else {
                throw new UnsupportedCallbackException(callbacks[i],
                    "Unrecognized
Callback");
            }
        }
    }
}
```

開発者は、**LoginModule** インターフェースの実装を作成することで、認証技術を統合します。これにより、管理者は異なる認証技術を 1 つのアプリケーションにプラグできます。複数の **LoginModule** をチェーン化し、複数の認証技術を認証プロセスに加えることが可能です。たとえば、1 つの **LoginModule** がユーザー名およびパスワードベースの認証を行い、別の **LoginModule** をスマートカードリーダーや生体認証などのハードウェアデバイスへ接続するインターフェースとすることが可能です。

LoginModule のライフサイクルは、クライアントがログインメソッドを作成し公開する **LoginContext** オブジェクトによって決定されます。このプロセスには 2 つのフェーズがあり、プロセスの手順は次のようになります。

- **LoginContext** は引数のないパブリックコンストラクターを使用して、設定された **LoginModule** を作成します。
- 各 **LoginModule** は、初期化メソッドへの呼び出しによって初期化されます。**Subject** 引数は null 以外になることが保証されます。初期化メソッドのシグネチャーは **public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)** です。
- **login** メソッドは、認証プロセスを開始するために呼び出されます。たとえば、あるメソッド実装はユーザーにユーザー名とパスワードの入力を求め、NIS や LDAP などのネーミングサービスに保存されたデータと照合してこの情報を検証することがあります。別の実装は、スマートカードや生体認証デバイスにインターフェースとして接続したり、基礎となるオペレーティングシステムからユーザー情報を抽出したりすることがあります。各 **LoginModule** によるユーザーアイデンティティの検証は、JAAS 認証のフェーズ 1 とみなされます。**login** メソッドのシグネチャーは **boolean login() throws LoginException** です。**LoginException** は失敗を意味します。true の戻り値はメソッドが成功したことを示し、false の戻り値はログインモジュールが無視されることを示します。
- **LoginContext** の全体的な認証が成功すると、各 **LoginModule** で **commit** が呼び出されます。フェーズ 1 が **LoginModule** に対して成功すると、コミットメソッドはフェーズ 2 を続行

し、関連するプリンシパル、パブリッククレデンシャル、プライベートクレデンシャルをサブジェクトに関連付けます。フェーズ 1 が **LoginModule** に対して失敗すると、**commit** はユーザー名やパスワードなどの以前保存した認証状態をすべて削除します。**commit** メソッドのシグネチャーは **boolean commit() throws LoginException** です。**LoginException** の発生は、コミットフェーズの完了に失敗したことを示します。true が返されるとメソッドが成功したことを示し、false が返されるとログインモジュールが無視されることを示します。

- **LoginContext** の全体的な認証が失敗すると、各 **LoginModule** で **abort** メソッドが呼び出されます。**abort** メソッドはログインまたは初期化メソッドによって作成されたすべての認証状態を削除または破棄します。**abort** メソッドのシグネチャーは **boolean abort() throws LoginException** です。**LoginException** の発生は、**abort** フェーズの完了に失敗したことを示します。true が返されるとメソッドが成功したことを示し、false が返されるとログインモジュールが無視されることを示します
- ログイン成功後に認証状態を削除するため、アプリケーションは **LoginContext** で **logout** を呼び出します。これにより、各 **LoginModule** で **logout** メソッドが呼び出されます。**logout** メソッドは、**commit** 操作中に当初サブジェクトに関連付けられていたプリンシパルとクレデンシャルを削除します。クレデンシャルは削除時に破棄されるはずですが、**logout** メソッドのシグネチャーは **boolean logout() throws LoginException** です。**LoginException** の発生は、ログアウトプロセスの完了に失敗したことを示します。true が返されるとメソッドが成功したことを示し、false が返されるとログインモジュールが無視されることを示します。

LoginModule がユーザーと通信して認証情報を取得する必要がある場合、**CallbackHandler** オブジェクトを使用します。アプリケーションはインターフェースを実装して **LoginContext** に渡し、基礎となるログインモジュールに直接認証情報を送信します。

ログインモジュールは、**CallbackHandler** を使用して、パスワードやスマートカード PIN などのユーザー入力による情報を取得したり、ステータスなどの情報をユーザーに提供したりします。アプリケーションによる **CallbackHandler** の指定を可能にすることで、基礎となる **LoginModule** がアプリケーションとユーザーが対話するさまざまな方法に依存しないようにします。たとえば、GUI アプリケーションの **CallbackHandler** の実装は、ウィンドウを表示してユーザーの入力を求めることがあります。一方でアプリケーションサーバーなどの GUI でない環境の **CallbackHandler** 実装は、アプリケーションサーバー API を使用してクレデンシャル情報を取得することがあります。インターフェースには実装するメソッドが 1 つあります。

```
void handle(Callback[] callbacks)
    throws java.io.IOException,
           UnsupportedOperationException;
```

最後に説明する認証クラスは **Callback** インターフェースです。これは複数のデフォルト実装が提供されているタグ付けインターフェースで、前述の例で使用した **NameCallback** と **PasswordCallback** が含まれます。**LoginModule** は **Callback** を使用し、認証メカニズムで必要となる情報を要求します。**LoginModule** は認証のログインフェーズの間に **Callback** のアレイを直接 **CallbackHandler.handle** メソッドに渡します。**CallbackHandler** がハンドルメソッドに渡された **Callback** オブジェクトの使用方法を理解できない場合は **UnsupportedCallbackException** が発生し、ログイン呼び出しが中止されます。

25.7. インターフェースのセキュア化

25.7.1. インターフェースのセキュア化

Hot Rod インターフェースはプログラムを使ってセキュア化できますが、memcached および REST インターフェースは宣言的にセキュア化する必要があります。これらのインターフェースをセキュア化する手順は、JBoss Data Grid の『**Administration and Configuration Guide**』を参照してください。

25.7.2. Hot Rod インターフェースセキュリティー

25.7.2.1. Hot Rod サーバーと Hot Rod クライアント間の通信の暗号化

Hot Rod は TLS/SSL を使用して暗号化でき、証明書ベースのクライアント認証を必要とするオプションを指定できます。

以下の手順にしたがって、SSL を使用して Hot Rod コネクターをセキュア化します。

SSL/TLS を使用した Hot Rod のセキュア化

```
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.impl.ConfigurationProperties;

[...]

public class SslConfiguration {

    public static final String ISPN_IP = "127.0.0.1";
    public static final String SERVER_NAME = "node0";
    public static final String SASL_MECH = "EXTERNAL";

    private static final String KEYSTORE_PATH = "./keystore_client.jks";
    private static final String KEYSTORE_PASSWORD = "secret";
    private static final String TRUSTSTORE_PATH =
"./truststore_client.jks";
    private static final String TRUSTSTORE_PASSWORD = "secret";

    SslConfiguration(boolean enabled,
        String keyStoreFileName,
        char[] keyStorePassword,
        SSLContext sslContext,
        String trustStoreFileName,
        char[] trustStorePassword) {
        ConfigurationBuilder builder = new ConfigurationBuilder();
        builder.addServer()
            .host(ISPN_IP)
            .port(ConfigurationProperties.DEFAULT_HOTROD_PORT);
        //setup auth
        builder.security()
            .authentication()
            .serverName(SERVER_NAME)
            .saslMechanism(SASL_MECH)
            .enable()
            .callbackHandler(new VoidCallbackHandler());
        //setup encrypt
```

```

        builder.security()
            .ssl()
            .enable()
            .keyStoreFileName(KEYSTORE_PATH)
            .keyStorePassword(KEYSTORE_PASSWORD.toCharArray())
            .trustStoreFileName(TRUSTSTORE_PATH)
            .trustStorePassword(TRUSTSTORE_PASSWORD.toCharArray());

        RemoteCacheManager cacheManager = new
RemoteCacheManager(builder.build());
        RemoteCache<Object, Object> cache =
cacheManager.getCache(RemoteCacheManager.DEFAULT_CACHE_NAME);
    }

    private static class VoidCallbackHandler implements CallbackHandler {
        @Override
        public void handle(Callback[] clbcks) throws IOException,
UnsupportedCallbackException {
        }
    }
}

```

重要

プレーンテキストのパスワードが設定またはソースコードに表示されないようにするには、プレーンテキストのパスワードを Vault パスワードに変更する必要があります。Vault パスワードのセットアップ方法についての詳細は、JBoss Enterprise Application Platform 『How to Configure Server Security』の「[Password Vault](#)」を参照してください。

25.7.2.2. SSL を使用した LDAP サーバーに対する Hot Rod のセキュア化

SSL を有効にして LDAP に接続する際に、適切な証明書が含まれるトラストストアまたはキーストアを指定する必要がある場合があります。

LDAP サーバーに対する Hot Rod クライアントの認証には、SSL 上の **PLAIN** 認証を使用できます。Hot Rod クライアントはプレーンテキストのクレデンシャルを SSL 上で JBoss Data Grid に送信し、サーバーは提供されたクレデンシャルを指定の LDAP サーバーに対して検証します。さらに、JBoss Data Grid サーバーと LDAP サーバー間にセキュアな接続を設定する必要があります。サーバーが LDAP バックエンドと通信するための設定方法の詳細は、JBoss Data Grid の『**Administration and Configuration Guide**』を参照してください。以下は、Hot Rod のクライアント側で SSL 上の **PLAIN** 認証を設定する例になります。

Hot Rod クライアントの LDAP サーバーに対する認証

```

import static org.infinispan.demo.util.CacheOps.dumpCache;
import static org.infinispan.demo.util.CacheOps.onCache;
import static org.infinispan.demo.util.CacheOps.putTestKV;
import static org.infinispan.demo.util.CmdArgs.LOGIN_KEY;
import static org.infinispan.demo.util.CmdArgs.PASS_KEY;
import static org.infinispan.demo.util.CmdArgs.getCredentials;

import java.util.Map;

import javax.net.ssl.SSLContext;

```

```

import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.impl.ConfigurationProperties;
import org.infinispan.commons.util.SslContextFactory;
import org.infinispan.demo.util.SaslUtils.SimpleLoginHandler;

public class HotRodPlainAuthOverSSL {

    public static final String ISPN_IP = "127.0.0.1";
    public static final String SERVER_NAME = "node0";
    public static final String SASL_MECH = "PLAIN";
    private static final String SECURITY_REALM = "ApplicationRealm";

    private static final String TRUSTSTORE_PATH =
"./truststore_client.jks";
    private static final String TRUSTSTORE_PASSWORD = "secret";

    public static void main(String[] args) {
        Map<String, String> userArgs = null;
        try {
            userArgs = getCredentials(args);
        } catch (IllegalArgumentException e) {
            System.err.println(e.getMessage());
            System.err.println(
                "Invalid credentials format, please provide
credentials (and optionally cache name) with --cache=<cache> --user=<user>
--password=<password>");
            System.exit(1);
        }

        ConfigurationBuilder builder = new ConfigurationBuilder();

        builder.addServer().host(ISPN_IP).port(ConfigurationProperties.DEFAULT_HOT
ROD_PORT);

        //set up PLAIN auth

        builder.security().authentication().serverName(SERVER_NAME).saslMechanism(
SASL_MECH).enable().callbackHandler(
            new SimpleLoginHandler(userArgs.get(LOGIN_KEY),
userArgs.get(PASS_KEY), SECURITY_REALM));

        //set up SSL
        SSLContext cont = SslContextFactory.getContext(null, null,
TRUSTSTORE_PATH, TRUSTSTORE_PASSWORD.toCharArray());
        builder.security().ssl().sslContext(cont).enable();

        RemoteCacheManager cacheManager = new
RemoteCacheManager(builder.build());
        RemoteCache<Object, Object> cache =
cacheManager.getCache(RemoteCacheManager.DEFAULT_CACHE_NAME);

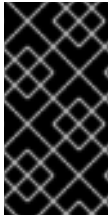
        onCache(cache, putTestKV.andThen(dumpCache));
    }
}

```

```

        cacheManager.stop();
        System.exit(0);
    }
}

```



重要

プレーンテキストのパスワードが設定またはソースコードに表示されないようにするには、プレーンテキストのパスワードを Vault パスワードに変更する必要があります。Vault パスワードのセットアップ方法についての詳細は、『**Red Hat Enterprise Application Platform Security Guide**』を参照してください。

25.7.2.3. SASL を使用した Hot Rod でのユーザー認証

25.7.2.3.1. SASL を使用した Hot Rod でのユーザー認証

Hot Rod でのユーザー認証は、以下の SASL (Simple Authentication and Security Layer) メカニズムを使用して実装できます。

- **PLAIN** は、クレデンシャルがプレーンテキスト形式でトランスポートされるため、最も安全性の低いメカニズムになります。ただし、実装が最も単純なメカニズムでもあります。このメカニズムは、セキュリティを強化するために暗号化 (**SSL**) と併用できます。
- **DIGEST-MD5** は、クレデンシャルをトランスポートする前にハッシュ化するメカニズムです。その結果、**PLAIN** メカニズムよりも安全性が高くなります。
- **GSSAPI** は、Kerberos チケットを使用するメカニズムです。その結果、正しく設定された Kerberos Domain Controller (例: Microsoft Active Directory) が必要になります。
- **EXTERNAL** は、基礎となるトランスポート (例: **X.509** クライアント証明書) から必要なクレデンシャルを取得するメカニズムであるため、正常に機能するにはクライアント証明書の暗号化が必要です。

25.7.2.3.2. Hot Rod 認証の設定 (GSSAPI/Kerberos)

以下の手順を使用し、SASL GSSAPI/Kerberos メカニズムを使用して Hot Rod 認証をセットアップします。

SASL GSSAPI/Kerberos 認証の設定: クライアント側の設定

1. サーバー側の設定が完了していることを確認してください。この設定は宣言的に行われ、JBoss Data Grid の『**Administration and Configuration Guide**』に記載されています。
2. クライアント側のログイン設定ファイル (**gss.conf**) にログインモジュールを定義します。
[source],options="nowrap"

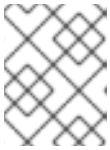
```

GssExample {
    com.sun.security.auth.module.Krb5LoginModule required client=TRUE;
};

```

1. 以下のシステムプロパティを設定します。


```
java.security.auth.login.config=gss.conf
java.security.krb5.conf=/etc/krb5.conf
```



注記

krb5.conf ファイルは環境に依存し、Keberos の鍵配布センター (Key Distribution Center) を示している必要があります。

2. CallbackHandler を実装します。

```
public class MyCallbackHandler implements CallbackHandler {
    final private String username;
    final private char[] password;
    final private String realm;

    public MyCallbackHandler() { }

    public MyCallbackHandler (String username, String realm, char[]
password) {
        this.username = username;
        this.password = password;
        this.realm = realm;
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            if (callback instanceof NameCallback) {
                NameCallback nameCallback = (NameCallback)
callback;
                nameCallback.setName(username);
            } else if (callback instanceof PasswordCallback) {
                PasswordCallback passwordCallback =
(PasswordCallback) callback;
                passwordCallback.setPassword(password);
            } else if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback authorizeCallback =
(AuthorizeCallback) callback;
                authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationI
D().equals(
                    authorizeCallback.getAuthorizationID()));
            } else if (callback instanceof RealmCallback) {
                RealmCallback realmCallback = (RealmCallback)
callback;
                realmCallback.setText(realm);
            } else {
                throw new UnsupportedCallbackException(callback);
            }
        }
    }
}
```

3. 以下のように Hot Rod クライアントを設定します。

```

LoginContext lc = new LoginContext("GssExample", new
MyCallbackHandler("krb_user", "krb_password".toCharArray()));
lc.login();
Subject clientSubject = lc.getSubject();

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .socketTimeout(1200000)
    .security()
        .authentication()
            .enable()
            .serverName("infinispan-server")
            .saslmMechanism("GSSAPI")
            .clientSubject(clientSubject)
            .callbackHandler(new MyCallbackHandler());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache =
remoteCacheManager.getCache("secured");

```

25.7.2.3.3. Hot Rod 認証 (MD5) の設定

以下の手順にしたがって、SASL MD5 メカニズムを使用して Hot Rod 認証をセットアップします。

1. サーバーに MD5 認証が設定されていることを確認してください。この設定を行う手順は、JBoss Data Grid の『**Administration and Configuration Guide**』に記載されています。
2. **CallbackHandler** を実装します。

```

public class MyCallbackHandler implements CallbackHandler {
    final private String username;
    final private char[] password;
    final private String realm;

    public MyCallbackHandler (String username, String realm, char[]
password) {
        this.username = username;
        this.password = password;
        this.realm = realm;
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            if (callback instanceof NameCallback) {
                NameCallback nameCallback = (NameCallback) callback;
                nameCallback.setName(username);
            } else if (callback instanceof PasswordCallback) {
                PasswordCallback passwordCallback = (PasswordCallback)
callback;
                passwordCallback.setPassword(password);
            } else if (callback instanceof AuthorizeCallback) {

```



```

        AuthorizeCallback authorizeCallback =
        (AuthorizeCallback) callback;

authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationI
D().equals(
        authorizeCallback.getAuthorizationID()));
    } else if (callback instanceof RealmCallback) {
        RealmCallback realmCallback = (RealmCallback) callback;
        realmCallback.setText(realm);
    } else {
        throw new UnsupportedCallbackException(callback);
    }
}
}
}
}

```

3. 以下のように、クライアントを設定された Hot Rod コネクターに接続します。

```

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .socketTimeout(1200000)
    .security()
    .authentication()
    .enable()
    .serverName("myhotrodserver")
    .saslmMechanism("DIGEST-MD5")
    .callbackHandler(new MyCallbackHandler("myuser",
"ApplicationRealm", "qwer1234!".toCharArray()));
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache =
remoteCacheManager.getCache("secured");

```

25.7.2.3.4. Hot Rod C++ 認証の設定 (GSSAPI/Kerberos)

以下の手順にしたがって、SASL GSSAPI/Kerberos メカニズムを使用して Hot Rod C++ クライアント認証をセットアップします。

SASL GSSAPI/Kerberos 認証の設定: クライアント側の設定

1. サーバー側の設定が完了していることを確認してください。この設定は宣言的に行われ、JBoss Data Grid の『**Administration and Configuration Guide**』に記載されています。

以下は、Hot Rod C++ クライアントに Kerberos を使用した完全な例になります。

```

#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <sasl/saslplug.h>
#include <krb5.h>
#include <err.h>

```

```

#include <stdlib.h>

using namespace infinispn::hotrod;

int kinit();
void kdestroy();

/* Hotrod SASL is based on Cyrus Sasl libraries.
 * Check cyrus docs for more info on how to setup callbacks
 * https://www.cyrusimap.org/sasl/
 */
static int simple(void* context , int id, const char **result, unsigned
*len) {
    *result = *(char**)context;
    if (len)
        *len = strlen(*result);
    return SASL_OK;
}

static int getsecret(void* /* conn */, void* context, int id,
sasl_secret_t **psecret) {
    char *secret_data=*(char**)context;
    size_t len = strlen(secret_data);
    static sasl_secret_t *x;
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    x->len = len;
    strcpy((char *) x->data, secret_data);
    *psecret = x;
    return SASL_OK;
}

char *pusername;
char *psecret;

static std::vector<sasl_callback_t> callbackHandler {
    { SASL_CB_USER, (sasl_callback_ft) &simple, &pusername },
    { SASL_CB_PASS, (sasl_callback_ft) &getsecret, &psecret },
    { SASL_CB_LIST_END, NULL, NULL } };

int kinit();
void kdestroy();

int main(int argc, char** argv) {
    int result = 0;
    {
        kinit();
        ConfigurationBuilder builder;
        char username[]="supervisor@INFINISPAN.ORG";
        char secret_data[]="lessStrongPassword";
        pusername=username;
        psecret=secret_data;
        builder.addServer().host(argc > 1 ? argv[1] :
"127.0.0.1").port(argc > 2 ? atoi(argv[2]) : 11222);
        builder.protocolVersion(Configuration::PROTOCOL_VERSION_24);

        builder.security().authentication().saslMechanism("GSSAPI").serverFQDN(

```

```

        "node0").callbackHandler(callbackHandler).enable();
builder.balancingStrategyProducer(nullptr);
RemoteCacheManager cacheManager(builder.build(), false);
BasicMarshaller<std::string> *km = new
BasicMarshaller<std::string>();
BasicMarshaller<std::string> *vm = new
BasicMarshaller<std::string>();
RemoteCache<std::string, std::string> cache =
cacheManager.getCache<std::string, std::string>(km,
        &Marshaller<std::string>::destroy, vm,
&Marshaller<std::string>::destroy, std::string("authCache"));
cacheManager.start();
try {
    cache.put("key", "value");
    std::shared_ptr<std::string> ret(cache.get("key"));
    result = 0;
} catch (Exception& ex) {
    std::cerr << "FAIL: 'supervisor' should read and write" <<
std::endl;
    result = -1;
}
cacheManager.stop();
std::cout << "PASS: 'GSSAPI' sasl authorization" << std::endl;
kdestroy();
}
return result;
}

krb5_context context;
krb5_creds creds;
krb5_principal client_princ = NULL;

int kinit() {
    // Delegate Kerberos setup to the system
    setenv("KRB5CCNAME", "krb5cc_hotrod", 1);
    setenv("KRB5_CONFIG", "krb5.conf", 1);
    std::system("echo lessStrongPassword | kinit -c krb5cc_hotrod
supervisor@INFINISPAN.ORG");
}
void kdestroy() {
    std::system("kdestroy");
}

```

25.7.2.3.5. Hot Rod C++ 認証の設定 (MD5)

以下の手順にしたがって、SASL MD5 メカニズムを使用して Hot Rod C++ クライアント認証をセットアップします。

SASL MD5 認証の設定 - クライアント側の設定

1. サーバー側の設定が完了していることを確認してください。この設定は宣言的に行われ、JBoss Data Grid の『**Administration and Configuration Guide**』に記載されています。

以下は、Hot Rod C++ クライアントに SASL MD5 を使用した完全な例になります。

```

#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <sasl/saslplug.h>
#include <krb5.h>

using namespace infinispan::hotrod;

/* Hotrod SASL is based on Cyrus Sasl libraries.
 * Check cyrus docs for more info on how to setup callbacks
 * https://www.cyrusimap.org/sasl/
 */
static int simple(void* context , int id, const char **result, unsigned
*len) {
    *result = *(char**)context;
    if (len)
        *len = strlen(*result);
    return SASL_OK;
}

static int getsecret(void* /* conn */, void* context, int id,
sasl_secret_t **psecret) {
    char *secret_data=*(char**)context;
    size_t len = strlen(secret_data);
    static sasl_secret_t *x;
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    x->len = len;
    strcpy((char *) x->data, secret_data);
    *psecret = x;
    return SASL_OK;
}

char *pusername;
char *psecret;

static std::vector<sasl_callback_t> callbackHandler {
    { SASL_CB_AUTHNAME, (sasl_callback_ft) &simple, &pusername },
    {SASL_CB_PASS, (sasl_callback_ft) &getsecret, &psecret },
    {SASL_CB_LIST_END, NULL, NULL } };

/* This sample authenticates the client with
 * user=reader
 * password=password
 * credential, which is an account that can only do WRITE
 * on the server.
 */
int main(int argc, char** argv) {
    int result = 0;
    {
        ConfigurationBuilder builder;
        char username[]="reader";
        char secret_data[]="password";
    }
}

```

```

        pusername=username;
        psecret=secret_data;
        builder.addServer().host("127.0.0.1").port(11222);
        builder.protocolVersion(Configuration::PROTOCOL_VERSION_24);
        builder.security().authentication().saslMechanism("DIGEST-
MD5").serverFQDN("node0").callbackHandler(callbackHandler).enable();
        RemoteCacheManager cacheManager(builder.build(), false);
        BasicMarshaller<std::string> *km = new
BasicMarshaller<std::string>();
        BasicMarshaller<std::string> *vm = new
BasicMarshaller<std::string>();
        auto cache = cacheManager.getCache<std::string, std::string>(km,
&Marshaller<std::string>::destroy, vm, &Marshaller<std::string>::destroy,
std::string("authCache"));
        cacheManager.start();
        std::shared_ptr<std::string> ret(cache.get("key"));
        try {
            cache.put("key", "value");
            std::cerr << "FAIL: 'reader' should not write" << std::endl;
            return -1;
        } catch (Exception& ex) {

        }
        std::cout << "PASS: 'DIGEST-MD5' sasl authorization" << std::endl;
        cacheManager.stop();
    }
    return result;
}

```

25.7.2.3.6. Hot Rod C++ 認証の設定 (PLAIN)

以下の手順にしたがって、SASL PLAIN メカニズムを使用して Hot Rod C++ クライアント認証をセットアップします。

SASL PLAIN 認証の設定 - クライアント側の設定

1. サーバー側の設定が完了していることを確認してください。この設定は宣言的に行われ、JBoss Data Grid の『**Administration and Configuration Guide**』に記載されています。

以下は、Hot Rod C++ クライアントに SASL PLAIN を使用した完全な例になります。

```

#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <sasl/saslplug.h>
#include <krb5.h>

using namespace infinispan::hotrod;

/* Hotrod SASL is based on Cyrus Sasl libraries.
 * Check cyrus docs for more info on how to setup callbacks
 * https://www.cyrusimap.org/sasl/

```

```

    */
static int simple(void* context , int id, const char **result, unsigned
*len) {
    *result = *(char**)context;
    if (len)
        *len = strlen(*result);
    return SASL_OK;
}

static int getsecret(void* /* conn */, void* context, int id,
sasl_secret_t **psecret) {
    char *secret_data=*(char**)context;
    size_t len = strlen(secret_data);
    static sasl_secret_t *x;
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    x->len = len;
    strcpy((char *) x->data, secret_data);
    *psecret = x;
    return SASL_OK;
}

char *pusername;
char *psecret;

static std::vector<sasl_callback_t> callbackHandler {
    { SASL_CB_AUTHNAME, (sasl_callback_ft) &simple, &pusername },
    {SASL_CB_PASS, (sasl_callback_ft) &getsecret, &psecret },
    {SASL_CB_LIST_END, NULL, NULL } };

/* This sample authenticates the client with
 * user=writer
 * password=somePassword
 * credential, which is an account that can only do WRITE
 * on the server.
 */
int main(int argc, char** argv) {
    int result = 0;
    {
        ConfigurationBuilder builder;
        char username[]="writer";
        char secret_data[]="somePassword";
        pusername=username;
        psecret=secret_data;
        builder.addServer().host("127.0.0.1").port(11222);
        builder.protocolVersion(Configuration::PROTOCOL_VERSION_24);

        builder.security().authentication().saslMechanism("PLAIN").serverFQDN("nod
e0").callbackHandler(callbackHandler).enable();
        RemoteCacheManager cacheManager(builder.build(), false);
        BasicMarshaller<std::string> *km = new
BasicMarshaller<std::string>();
        BasicMarshaller<std::string> *vm = new
BasicMarshaller<std::string>();
        auto cache = cacheManager.getCache<std::string, std::string>(km,
&Marshaller<std::string>::destroy, vm, &Marshaller<std::string>::destroy,
std::string("authCache"));
    }
}

```

```

        cacheManager.start();
        cache.put("key", "value");
        try {
            std::shared_ptr<std::string> ret(cache.get("key"));
            std::cerr << "FAIL: 'writer' should not read" << std::endl;
            return -1;
        } catch (Exception& ex) {

        }
        std::cout << "PASS: 'PLAIN' sasl authorization" << std::endl;
        cacheManager.stop();
    }
    return result;
}

```

25.7.2.3.7. Hot Rod C# 認証の設定 (EXTERNAL)

以下の手順にしたがって、SASL EXTERNAL メカニズムを使用して Hot Rod C# クライアント認証をセットアップします。

SASL EXTERNAL 認証の設定 - クライアント側の設定

1. サーバー側の設定が完了していることを確認してください。この設定は宣言的に行われ、JBoss Data Grid の『**Administration and Configuration Guide**』に記載されています。

以下は、Hot Rod C++ クライアントに SASL EXTERNAL を使用した完全な例になります。

```

using Infinispan.HotRod;
using Infinispan.HotRod.Config;
using System;
using System.Text;

namespace Authentication
{
    class Program
    {

        static void Main(string[] args)
        {
            ConfigurationBuilder conf = new ConfigurationBuilder();
            conf.AddServer()
                .Host("127.0.0.1")
                .Port(11222)
                .ConnectionTimeout(90000)
                .SocketTimeout(900);
            // Enable EXTERNAL mechanism for SASL
            conf.Security().Authentication()
                .Enable()
                .SaslMechanism("EXTERNAL")
                .ServerFQDN("node0");
            // Enable SSL (EXTERNAL is based on the client certificate)
            conf.Ssl().Enable()
                .ServerCAFile("infinispan-ca.pem")
                .ClientCertificateFile("keystore_client.p12");
            // end of SASL configuration

```

```
// The subject specified in the truststore_client.p12 cert will be used to
// identify the client
IMarshaller marshaller = new JBasicMarshaller();
conf.Marshaller(marshaller);
Configuration c = conf.Build();
RemoteCacheManager remoteManager = new RemoteCacheManager(c,
true);

IRemoteCache<string, string> authCache =
remoteManager.GetCache<string, string>("authCache");
authCache.Put("K1", "V1");
authCache.Get("K1");
authCache.Clear();

    }
}
}
```

25.7.2.3.8. Hot Rod C# 認証の設定 (MD5)

以下の手順にしたがって、SASL MD5 メカニズムを使用して Hot Rod C# クライアント認証をセットアップします。

SASL MD5 認証の設定 - クライアント側の設定

1. サーバー側の設定が完了していることを確認してください。この設定は宣言的に行われ、JBoss Data Grid の『**Administration and Configuration Guide**』に記載されています。

以下は、Hot Rod C# クライアントに SASL MD5 を使用した完全な例になります。

```
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
using System;
using System.Text;

namespace Authentication
{
    class Program
    {
        static void Main(string[] args)
        {
            ConfigurationBuilder conf = new ConfigurationBuilder();
            conf.AddServer()
                .Host("127.0.0.1")
                .Port(11222)
                .ConnectionTimeout(90000)
                .SocketTimeout(900);

            // Enable authentication use PLAIN as mechanism (DIGEST-MD5 can be used
            // the same way)
            // and setup user password and realm
            conf.Security().Authentication()
                .Enable()
                .SaslMechanism("DIGEST-MD5")
                .ServerFQDN("node0")
                .SetCallback("writer", "somePassword",
"ApplicationRealm");
        }
    }
}
```



```
// end of SASL configuration
    IMarshaller marshaller = new JBasicMarshaller();
    conf.Marshaller(marshaller);
    Configuration c = conf.Build();
    RemoteCacheManager remoteManager = new RemoteCacheManager(c,
true);
    IRemoteCache<string, string> authCache =
remoteManager.GetCache<string, string>("authCache");
    authCache.Put("K1", "V1");
    authCache.Get("K1");
    authCache.Clear();
    }
}
}
```

25.7.3. Hot Rod C++ クライアントの暗号化

デフォルトでは、リモートサーバーとの通信はすべて暗号化されていませんが、**SslConfigurationBuilder** で **serverCAFile** メソッドを使用してサーバーのキーを定義すると、TLS による暗号化を有効にすることができます。さらに、**clientCertificateFile** でクライアントの証明書を定義すると、クライアント認証を有効にできます。

以下は、任意のクライアント証明書でサーバーキーを定義する例になります。

Hot Rod C++ での TLS の例

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <stdlib.h>
#include <iostream>
#include <memory>
#include <typeinfo>

using namespace infinispan::hotrod;

int main(int argc, char** argv) {
    std::cout << "TLS Test" << std::endl;
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " server_ca_file
[client_ca_file]" << std::endl;
        return 1;
    }
    {
        ConfigurationBuilder builder;

        builder.addServer().host("127.0.0.1").port(11222).protocolVersion(Configuration::PROTOCOL_VERSION_24);
        // Enable the TLS layer and install the server public key
        // this ensure that the server is authenticated
        builder.ssl().enable().serverCAFile(argv[1]);
        if (argc > 2) {
```

```

        // Send a client certificate for authentication (optional)
        // without this the socket will only be encrypted
        std::cout << "Using supplied client certificate for
authentication against the server" << std::endl;
        builder.ssl().clientCertificateFile(argv[2]);
    }
    // That's all. Now do business as usual
    RemoteCacheManager cacheManager(builder.build(), false);
    BasicMarshaller<std::string> *km = new BasicMarshaller<std::string>
();
    BasicMarshaller<std::string> *vm = new BasicMarshaller<std::string>
();
    RemoteCache<std::string, std::string> cache =
cacheManager.getCache<std::string, std::string>(km,
        &Marshaller<std::string>::destroy, vm,
&Marshaller<std::string>::destroy );
    cacheManager.start();
    cache.clear();
    std::string k1("key13");
    std::string v1("boron");

    cache.put(k1, v1);
    std::unique_ptr<std::string> rv(cache.get(k1));
    if (rv->compare(v1)) {
        std::cerr << "get/put fail for " << k1 << " got " << *rv << "
expected " << v1 << std::endl;
        return 1;
    }
    cacheManager.stop();
}
return 0;
}

```

クライアントは、**sniHostName** 関数に値を提供して TLS/SNI ハンドシェイク処理の開始時に接続を試みるホスト名を示すこともできます。たとえば、以下を使用できます。

```

[... ]
    builder.ssl().enable().serverCAFile(argv[1]).sniHostName("sni");
[... ]

```

25.7.4. Hot Rod C# クライアントの暗号化

デフォルトでは、リモートサーバーとの通信はすべて暗号化されていませんが、**SslConfigurationBuilder** で **ServerCAFile** メソッドを使用してサーバーのキーを定義すると、TLS による暗号化を有効にすることができます。さらに、**ClientCertificateFile** でクライアントの証明書を定義すると、クライアント認証を有効にできます。

以下は、任意のクライアント証明書でサーバーキーを定義する例になります。

Hot Rod C# での TLS の例

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

```

```

using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;

namespace TLS
{
    /// <summary>
    /// This sample code shows how to perform operations over TLS using
    the C# client
    /// </summary>

    class TLS
    {
        static void Main(string[] args)
        {
            // Cache manager setup
            RemoteCacheManager remoteManager;
            ConfigurationBuilder conf = new ConfigurationBuilder();

            conf.AddServer().Host("127.0.0.1").Port(11222).ConnectionTimeout(90000).SocketTimeout(900);
            SslConfigurationBuilder sslConfB = conf.Ssl();
            // Retrieve the server public certificate, needed to do server
            authentication. Mandatory
            if (!System.IO.File.Exists("resources/infinispan-ca.pem"))
            {
                Console.WriteLine("File not found: resources/infinispan-
ca.pem.");
                Environment.Exit(-1);
            }
            sslConfB.Enable().ServerCAFile("resources/infinispan-ca.pem");
            // Retrieve the client public certificate, needed if the
            server requires client authentication. Optional
            if (!System.IO.File.Exists("resources/keystore_client.p12"))
            {
                Console.WriteLine("File not found:
resources/keystore_client.p12.");
                Environment.Exit(-1);
            }

            sslConfB.ClientCertificateFile("resources/keystore_client.p12");

            // Usual business now
            conf.Marshaller(new JBasicMarshaller());
            remoteManager = new RemoteCacheManager(conf.Build(), true);
            IRemoteCache<string, string> testCache =
remoteManager.GetCache<string, string>();
            testCache.Clear();
            string k1 = "key13";
            string v1 = "boron";
            testCache.Put(k1, v1);
        }
    }
}

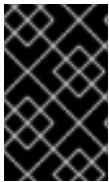
```

クライアントは、**SniHostName** に値を提供して TLS/SNI ハンドシェイク処理の開始時に接続を試みるホスト名を示すこともできます。たとえば、**ServerCAFile** を定義した直後に以下の呼び出しを含めることができます。

```
[...]
sslConfB.ServerCAFile("resources/infinispan-ca.pem").SniHostName("sni");
[...]
```

25.7.5. Hot Rod Node.js の暗号化

The Node.js クライアントは SSL/TLS 経由の暗号化と任意の TLS/SNI をサポートします。クライアントで設定を行うには、JDK に含まれる **keytool** アプリケーションを使用して Java KeyStore (JKS) を作成する必要があります。作成されたキーストアには、JBoss Data Grid サーバーが接続を承認するために必要なキーと証明書が含まれる必要があり、暗号化に対して JBoss Data Grid サーバーを設定する必要があります。暗号化に対してサーバーを設定するための詳細は、JBoss Data Grid の『**Administration and Configuration Guide**』を参照してください。



重要

TLS/SSL の Node.js クライアント実装は自己署名証明書を許可しません。以前証明書が自己署名されていた場合は、ローカルの証明機関を設定して証明書に署名するか、オープンな証明機関を使用することが推奨されます。

信頼できる証明書の場所を定義すると、クライアントの接続がサーバーによって承認されます。

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
    ssl: {
      enabled: true,
      trustCerts: ['my-root-ca.crt.pem']
    }
  }
);
```

また、クライアントは **PKCS#12** または **PFX** 形式のキーストアから信頼できる証明書を読み取ることもできます。

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
    ssl: {
      enabled: true,
      cryptoStore: {
        path: 'my-truststore.p12',
        passphrase: 'secret'
      }
    }
  }
);
```

さらに、クライアントを暗号化された認証と設定することもできます。認証を設定するには、クライアントのプライベートキー、パスフレーズ、および証明書キーを提供する必要があります。

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
```

```
{
  ssl: {
    enabled: true,
    trustCerts: ['my-root-ca.crt.pem'],
    clientAuth: {
      key: 'privkey.pem',
      passphrase: 'secret',
      cert: 'cert.pem'
    }
  }
}
);
```

クライアントは、**sniHostName** ディレクティブを含めることで、TLS/SNI ハンドシェイク処理の開始時に接続を試みるホスト名を示すこともできます

```
var connected = infinispn.client({port: 11222, host: '127.0.0.1'},
{
  ssl: {
    enabled: true,
    trustCerts: ['my-root-ca.crt.pem']
    sniHostName: 'example.com'
  }
});
```



注記

sniHostName が提供されない場合、クライアントは **localhost** を SNI パラメーターとして送信します。サーバーのデフォルトレルムが **localhost** と一致しない場合、エラーが発生します。

25.8. セキュリティー監査ロガー

25.8.1. セキュリティー監査ロガー

Red Hat JBoss Data Grid には、キャッシュまたはキャッシュマネージャーの操作が各種操作で許可または拒否されたかどうかを確認するために、キャッシュのセキュリティログを監査するロガーが含まれます。

デフォルトの監査ロガーは **org.infinispan.security.impl.DefaultAuditLogger** です。このロガーは、利用可能なロギングフレームワーク (JBoss Logging など) を使用して監査ログを出力し、**TRACE** レベルおよび **AUDIT** カテゴリで結果を提供します。

AUDIT カテゴリを、ログファイル、JMS キュー、またはデータベースのいずれかに送信するには、適切なログアペンダーを使用します。

25.8.2. セキュリティー監査ロガーの設定 (ライブラリーモード)

以下のように、Red Hat JBoss Data Grid の監査ロガーを設定します。

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
global.security()
```

```
.authorization()  
    .auditLogger(new DefaultAuditLogger());
```

25.8.3. カスタム監査ロガー

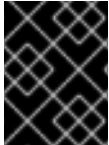
ユーザーは、カスタム監査ロガーを Red Hat JBoss Data Grid のライブラリーおよびリモートクライアントサーバーモードで実装できます。カスタムロガーは **org.infinispan.security.AuditLogger** インターフェースを実装する必要があります。カスタムロガーが指定されない場合、デフォルトのロガー (**DefaultAuditLogger**) が使用されます。

第26章 クラスタートラフィックのセキュリティ

26.1. ノードセキュリティの設定 (ライブラリーモード)

ライブラリーモードでは、ノード認証は JGroups 設定で直接設定されます。JGroups を設定してノードがクラスターへの参加またはマージの際に相互に認証できるようにします。認証は SASL を使用し、**SASL** プロトコルを JGroups XML 設定に追加して有効にします。

SASL は、**CallbackHandlers** などの JAAS の概念に基づいて認証ハンドシェイクに必要な特定の情報を取得します。ユーザーはクライアント側とサーバー側の両方に **CallbackHandlers** を指定する必要があります。



重要

JAAS API はユーザー認証および承認の設定時にのみ利用でき、ノードのセキュリティには利用できません。

以下の例は **CallbackHandler** クラスの実装方法を示しています。この例では、ログインとパスワードは、JBoss Data Grid の起動時に Java プロパティ経由で提供される値と照合してチェックされます。承認は、クラス ("**test_user**") で定義された **role** と照合してチェックされます。

コールバックハンドラークラス

```
public class SaslPropAuthUserCallbackHandler implements CallbackHandler {

    private static final String APPROVED_USER = "test_user";

    private final String name;
    private final char[] password;
    private final String realm;

    public SaslPropAuthUserCallbackHandler() {
        this.name = System.getProperty("sas1.username");
        this.password = System.getProperty("sas1.password").toCharArray();
        this.realm = System.getProperty("sas1.realm");
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            if (callback instanceof PasswordCallback) {
                ((PasswordCallback) callback).setPassword(password);
            } else if (callback instanceof NameCallback) {
                ((NameCallback) callback).setName(name);
            } else if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback authorizeCallback = (AuthorizeCallback)
callback;
                if
(APPROVED_USER.equals(authorizeCallback.getAuthorizationID())) {
                    authorizeCallback.setAuthorized(true);
                } else {
                    authorizeCallback.setAuthorized(false);
                }
            }
        }
    }
}
```

```

        } else if (callback instanceof RealmCallback) {
            RealmCallback realmCallback = (RealmCallback) callback;
            realmCallback.setText(realm);
        } else {
            throw new UnsupportedOperationException(callback);
        }
    }
}

```

認証では、`javax.security.auth.callback.NameCallback` および `javax.security.auth.callback.PasswordCallback` コールバックを指定します。

承認では、認証に必要なコールバックと `javax.security.sasl.AuthorizeCallback` コールバックを指定します。

26.2. ライブラリーモードのノード承認

J Groups の **SASL** は認証プロセスでのみ関係します。ノードの承認を実装するには、例外を発生させ、サーバーコールバックハンドラー内で行います。

以下の例を示します。

ノード承認の実装

```

public class AuthorizingServerCallbackHandler implements CallbackHandler {

    @Override
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedOperationException {
        for (Callback callback : callbacks) {
            <!-- Additional configuration information here -->
            if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback acb = (AuthorizeCallback) callback;
                if (!"myclusterrole".equals(acb.getAuthenticationID()))
            {
                throw new SecurityException("Unauthorized node "
+user);
            }
            <!-- Additional configuration information here -->
        }
    }
}

```


パート IV. RED HAT JBOSS DATA GRID の高度な機能

第27章 RED HAT JBOSS DATA GRID の高度な機能

Red Hat JBoss Data Grid には高度な機能が含まれています。これらの機能の一部と基本設定の手順は、『[Administration and Configuration Guide](#)』に記載されています。『[開発者ガイド](#)』では、これらの機能とその他の機能を詳細に説明し、より高度な使用例の手順を取り上げます。

本ガイドで取り上げる高度な機能は次のとおりです。

- 監視
- Red Hat JBoss Data Grid の Lucene Directory としての使用
- トランザクション
- マーシャリング
- リスナーと通知
- Infinispan CDI モジュール
- Spring Framework との統合
- Apache Spark との統合
- Apache Hadoop との統合
- EAP との統合
- サーバーヒンティングを用いた高可用性
- 分散実行
- ストリーム
- スクリプト
- リモートタスクの実行
- 相互運用性および互換性モード
- データセンター間レプリケーション
- ニアキャッシュ

第28章 監視

28.1. 監視

28.2. JAVA MANAGEMENT EXTENSIONS (JMX)

28.2.1. Java Management Extensions (JMX)

Java Management Extensions (JMX) は、アプリケーション、デバイス、システムオブジェクト、およびサービス指向ネットワークを管理および監視するツールを提供する Java ベースの技術です。これらの各オブジェクトは **MBeans** によって管理および監視されます。

JMX はミドルウェア管理の事実上の業界標準です。そのため、Red Hat JBoss Data Grid では管理および統計情報の公開に **JMX** が使用されます。

28.2.2. Red Hat JBoss Data Grid における JMX の使用

Red Hat JBoss Data Grid インスタンスの管理は、関連する統計情報をできるだけ多く公開することを目的としています。この統計情報により、管理者は各インスタンスの状態を確認することができます。1 つのインストールは数十または数百のインスタンスを構成することがあるため、各インスタンスの統計情報を明確かつ簡潔に公開し、表示する必要があります。

このような統計情報を公開し、管理者に対して適切な情報を見やすく表示するため、JBoss Data Grid では、JMX は JBoss Operations Network (JON) と共に使用されます。

28.2.3. キャッシュインスタンスに対して JMX を有効にする

キャッシュレベルでは、宣言的に、またはプログラムを用いて、次のように JMX 統計を有効にすることができます。

キャッシュレベルでプログラムを用いて JMX を有効にする

プログラムを用いて JMX をキャッシュレベルで有効にするには、以下のコードを追加します。

```
Configuration configuration = new  
ConfigurationBuilder().jmxStatistics().enable().build();
```

28.2.4. CacheManager に対して JMX を有効にする

CacheManager レベルでは、宣言的に、またはプログラムを用いて、次のように JMX 統計を有効にすることができます。

CacheManager レベルでプログラムを用いて JMX を有効にする

プログラムを用いて JMX を **CacheManager** レベルで有効にするには、以下のコードを追加します。

```
GlobalConfiguration globalConfiguration = new  
GlobalConfigurationBuilder().globalJmxStatistics().enable().build();
```

28.2.5. 複数の JMX ドメイン

複数の **CacheManager** インスタンスが 1 つの仮想マシンに存在したり、異なる **CacheManager** のキャッシュインスタンスの名前が競合する場合に、複数の JMX ドメインが使用されます。

この問題を解決するには、JMX や JBoss Operations Network などの監視ツールが簡単に識別および使用できるような名前を各 **CacheManager** に付けるようにします。

CacheManager の名前をプログラムを用いて設定する

次のコードを追加し、プログラムを用いて **CacheManager** の名前を設定します。

```
GlobalConfiguration globalConfiguration = new
GlobalConfigurationBuilder().globalJmxStatistics().enable().
cacheManagerName("Hibernate2LC").build();
```

28.2.6. デフォルトでない MBean サーバーでの MBean の登録

使用されるすべての MBean がデフォルトで登録される場所は、標準の JVM MBeanServer プラットフォームです。ユーザーは代替の MBeanServer インスタンスを設定することもできます。

MBeanServerLookup インターフェースを実装して、確実に **getMBeanServer()** メソッドが必要な (デフォルト以外の) MBeanServer を返すようにします。

デフォルト以外の場所を設定して MBean を登録するには、実装を作成し、クラスの完全修飾名を用いて Red Hat JBoss Data Grid を設定します。例は次のとおりです。

完全修飾ドメイン名をプログラムを用いて追加する

以下のコードを追加します。

```
GlobalConfiguration globalConfiguration = new
GlobalConfigurationBuilder().globalJmxStatistics().enable().
mBeanServerLookup("com.acme.MyMBeanServerLookup").build();
```

28.3. STATISTICSINFOMBEAN

StatisticsInfoMBean MBean は、前セクションの説明どおりに **Statistics** オブジェクトにアクセスします。

第29章 LUCENE DIRECTORY としての RED HAT JBOSS DATA GRID

29.1. LUCENE DIRECTORY としての RED HAT JBOSS DATA GRID

Red Hat JBoss Data Grid は、リレーショナルデータベースの Hibernate Search クエリー用の共有インメモリーインデックス (Infinispan Directory) として使用することができます。デフォルトでは、Hibernate Search はローカルファイルシステムを使用して Lucene インデックスを保存しますが、JBoss Data Grid をストレージとして使用するように設定すると、複数のサーバーノード全体でリアルタイムのレプリケーションを実現できます。

Infinispan Directory では、インデックスはメモリーに保存され、複数のノード全体で共有されます。Infinispan Directory は、参加するすべてのノードに分散される 1 つのディレクトリーとして動作します。1 つのノードでインデックスが更新されると、すべてのノードのインデックスが更新されます。クラスター全体でノードの更新が行われた直後にインデックスを検索できます。デフォルトの Hibernate Search 設定は、すべてのノードでインデックスを定義するデータをレプリケートします。

大型のインデックスのデータ分散を有効にするとメモリーの消費を少なくすることができますが、ローカルでのクエリー操作の効率が悪くなります。また、インデックス化されたデータを各ノードに設定された CacheStore にオフロードしたり、各ノードが共有する単一の集中型 CacheStore を設定したりすることもできます。



注記

レプリケーションの代わりに分散を有効にするとメモリーを節約できる可能性があります。ただし、クエリーの速度は遅くなります。CacheStore を有効にすると、さらにメモリーを節約できる可能性があります。パッシベーションの使用時にパフォーマンスが低下します。

29.2. 設定

ディレクトリープロバイダーは、インデックスごとに指定して有効にします。**default** インデックスが指定されると、指定がある場合を除き、すべてのインデックスがディレクトリープロバイダーを使用します。

```
hibernate.search.[default|<indexname>].directory_provider = infinispan
```

上記は、クラスターがレプリケートされたインデックスを提供しますが、デフォルト設定はインデックスの永続化を永久にする方法を有効にしません。このような機能を有効にするには、Infinispan 設定ファイルを提供します。

Hibernate Search では、**CacheManager** が Infinispan を使用する必要があります。JNDI 経由で既存の **CacheManager** を検索および再使用でき、新しい **CacheManager** を開始または管理できます。既存の **CacheManager** を検索するとき、登録された Infinispan サブシステムから提供されます。たとえば、JBoss EAP 経由で登録された場合、JBoss EAP の Infinispan サブシステムが **CacheManager** を提供します。



注記

JNDI を使用して **CacheManager** を登録する場合、Red Hat JBoss Data Grid の設定ファイルのみを使用して登録する必要があります。

JNDI 経由で既存の CacheManager を使用する (任意のパラメーター):

```
hibernate.search.infinispan.cachemanager_jndiname = [jndiname]
```

設定ファイルから新規の **CacheManager** を開始する (任意のパラメーター):

```
hibernate.search.infinispan.configuration_resourceName = [infinispan
configuration filename]
```

両方のパラメーターが定義されると、JNDI が優先されます。どちらも定義されないと、Hibernate Search は永続キャッシュストアにインデックスを格納しないデフォルトの Infinispan 設定を使用します。

29.3. RED HAT JBOSS DATA GRID モジュール

Hibernate Search の Red Hat JBoss Data Grid ディレクトリープロバイダーは、JBoss EAP の JBoss Data Grid ライブラリーモジュールの一部として配布されます。[Red Hat カスタマーポータル](#) からファイルをダウンロードします。

目的の JBoss Enterprise Application Platform フォルダの modules/ ディレクトリーにアーカイブを展開します。

以下のエントリーをプロジェクトアーカイブの **MANIFEST.MF** ファイルに追加します。

```
Dependencies: org.hibernate.search.orm services
```

詳細は、Red Hat JBoss EAP 『[開発ガイド](#)』の「[Maven を使用した MANIFEST.MF エントリーの生成](#)」を参照してください。

29.4. レプリケートされたインデックスの LUCENE ディレクトリー設定

標準の JPA を使用する場合、以下のプロパティーを Hibernate 設定および永続ユニット設定ファイルで定義します。たとえば、デフォルトのストレージインデックスをすべて変更する場合は、以下のプロパティーを設定できます。

```
hibernate.search.default.directory_provider=infinispan
```

これを一意なインデックスで実行することもできます。以下の例では、**tickets** と **actors** がインデックス名になります。

```
hibernate.search.tickets.directory_provider=infinispan
hibernate.search.actors.directory_provider=filesystem
```

Lucene の **DirectoryProvider** は以下のオプションを使用してキャッシュ名を設定します。

- **locking_cachename** - Lucene のロックが保存されるキャッシュの名前。デフォルトは **LuceneIndexesLocking** です。
- **data_cachename** - 最大のデータチャンクや最大のオブジェクトを含む Lucene のデータが保存されるキャッシュの名前。デフォルトは **LuceneIndexesData** です。
- **metadata_cachename** - Lucene のメタデータが保存されるキャッシュの名前。デフォルトは **LuceneIndexesMetadata** です。

ロッキングキャッシュの名前を **CustomLockingCache** に変更するには、以下を使用します。

-

```
hibernate.search.default.directory_provider.locking_cachename="CustomLockingCache"
```

また、インデックスの大型のファイルは設定可能な小型のチャンクに分割されます。通常、インデックスの **chunk_size** をネットワークが効率的に処理できる最大値に設定することが推奨されます。

Hibernate Search には、レプリケートされたキャッシュを使用してインデックスを処理するデフォルト値がすでに含まれています。

複数のノードが同時にインデックスに書き込みを行う場合は、JMS バックエンドを設定することが重要です。設定の詳細は、Hibernate Search のドキュメントを参照してください。



重要

分散モードを設定する必要がある設定では、**LuceneIndexesMetadata** と **LuceneIndexesLocking** キャッシュがすべての場合で常にレプリケーションモードを使用する必要があります。

29.5. JMS マスターおよびスレーブのバックエンド設定

Infinispan ディレクトリーを使用する場合、JMS マスター/スレーブバックエンドの使用が推奨されます。Infinispan では、すべてのノードが同じインデックスを共有します。また、異なるノード上で **IndexWriter** がアクティブであるため、同じインデックスでロックを取得します。そのため、インデックスへ直接更新を送信する代わりに、JMS へ送信し、1 つのノードが他のノードの代わりにすべての変更を適用するようにします。



警告

JMS ベースのバックエンドを有効にしないと、複数のノードがインデックスへの書き込みを試行するときにタイムアウト例外が発生する原因となります。

JMS スレーブを設定するには、バックエンドのみを置き換え、ディレクトリープロバイダーを Infinispan に設定します。同じディレクトリープロバイダーをマスターで設定すると接続され、ノード全体でコピージョブを設定する必要はありません。

マスターおよびスレーブの設定例は、Red Hat JBoss EAP 『**Developing Hibernate Applications**』の「**Back End Setup and Operations**」を参照してください。

第30章 トランザクション

30.1. JAVA トランザクション API

Red Hat JBoss Data Grid では、Java トランザクション API (JTA) に対応するトランザクションの設定、使用、および参加がサポートされます。

JBoss Data Grid は各キャッシュ操作に対して以下を実行します。

1. 最初に、現在スレッドに関連付けされているトランザクションを読み出します。
2. **XAResource** が登録されていない場合は、トランザクションマネージャーに登録し、トランザクションがコミットまたはロールバックされたときに通知を受け取るようにします。

30.2. トランザクションの設定 (ライブラリーモード)

Red Hat JBoss Data Grid では、ライブラリーモードのトランザクションは、同期およびトランザクションリカバリーと共に設定できます。トランザクションは全体として (同期およびトランザクションリカバリーを含む)、リモートクライアントサーバーモードでは使用できません。

キャッシュ操作を実行するには、キャッシュに環境のトランザクションマネージャーへの参照が必要となります。**TransactionManagerLookup** インターフェースの実装に属するクラス名を用いて、キャッシュを設定します。キャッシュが初期化されると、キャッシュは指定クラスのインスタンスを作成し、**getTransactionManager()** メソッドを呼び出してトランザクションマネージャーへの参照を見つけ、これを返します。

ライブラリーモードでは、トランザクションは以下のように設定されます。

ライブラリーモードでのトランザクションの設定 (プログラムによる設定)

1. トランザクションの有効化

```
Configuration config = new ConfigurationBuilder()/* ...
*/.transaction()
    .transactionMode(TransactionMode.TRANSACTIONAL)
    .transactionManagerLookup(new
GenericTransactionManagerLookup())
    .lockingMode(LockingMode.OPTIMISTIC)
    .useSynchronization(true)
    .recovery()

    .recoveryInfoCacheName("anotherRecoveryCacheName").build();
```

- a. トランザクションモードを設定します。
- b. ルックアップクラスを選択し、設定します。利用可能なルックアップクラスのリストについては、この手順の下にある表を参照してください。
- c. **lockingMode** 値は、楽観的または悲観的ロックを使用するかどうかを決定します。キャッシュが非トランザクションの場合、ロックモードは無視されます。デフォルト値は **OPTIMISTIC** です。

- d. **useSynchronization** 値は、トランザクションマネージャーを使って同期化を登録するようにキャッシュを設定するか、またはキャッシュ自体を XA リソースとして登録するようにキャッシュを設定します。デフォルト値は **true** (同期の使用) です。
- e. **recovery** パラメーターを **true** に設定すると、キャッシュのリカバリーが有効になります。
recoveryInfoCacheName は、リカバリー情報が保持されるキャッシュの名前を設定します。キャッシュのデフォルト名は **RecoveryConfiguration.DEFAULT_RECOVERY_INFO_CACHE** によって指定されます。

2. 書き込みスキューチェックの設定

writeSkewチェックは、異なるトランザクションからのエントリーへの変更によってトランザクションがロールバックされるべきかどうかを判別します。書き込みスキューが **true** に設定された場合、**isolation_level** を **REPEATABLE_READ** に設定する必要があります。**writeSkew** および **isolation_level** のデフォルト値はそれぞれ **false** と **READ_COMMITTED** です。

```
Configuration config = new ConfigurationBuilder()/* ... */.locking()
    .isolationLevel(IsolationLevel.REPEATABLE_READ).writeSkewCheck(true)
    ;
```

3. エントリーのバージョン管理の設定

クラスター化されたキャッシュについては、エントリーのバージョン管理を有効にし、その値を **SIMPLE** に設定することにより書き込みスキューのチェックを有効にします。

```
Configuration config = new ConfigurationBuilder()/* ...
    */.versioning()
        .enable()
        .scheme(VersioningScheme.SIMPLE);
```

表30.1 トランザクションマネージャーのルックアップクラス

クラス名	説明
org.infinispan.transaction.lookup.DummyTransactionManagerLookup	テスト環境で主に使用されます。このテスト向けのトランザクションマネージャーは実稼働環境では使用されず、特に並列トランザクションやリカバリーなどの機能は厳しく制限されます。
org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup	Red Hat JBoss Data Grid がスタンドアロン環境で実行される場合のデフォルトのトランザクションマネージャーです。これは、JBoss Transactions ベースの完全に機能するトランザクションマネージャーで、 DummyTransactionManager の機能上の制限が解消されます。

クラス名	説明
org.infinispan.transaction.lookup.GenericTransactionManagerLookup	GenericTransactionManagerLookup は、トランザクションルックアップクラスが指定されていない場合にデフォルトで使用されます。このルックアップクラスは、JBoss Data Grid を、TransactionManager インターフェースを提供する Java EE 互換環境で使用する場合に推奨されます。このルックアップクラスは、ほとんどの Java EE アプリケーションサーバーでトランザクションマネージャーを見つけることができます。トランザクションマネージャーが見つからない場合、デフォルトは DummyTransactionManager になります。
org.infinispan.transaction.lookup.JBossTransactionManagerLookup	JbossTransactionManagerLookup は、アプリケーションサーバーで実行中の標準的なトランザクションマネージャーを見つけます。このルックアップクラスは JNDI を使用して TransactionManager インスタンスを検索します。これは、カスタムキャッシュが JTA トランザクションで使用されている場合に推奨されます。



注記

Red Hat JBoss Data Grid を Tomcat または通常の Java Virtual Machine (JVM) と使用する場合、推奨される Transaction Manager Lookup クラスは JBoss Transactions を使用する **JBossStandaloneJTAManagerLookup** です。

30.3. 複数のキャッシュインスタンス間でのトランザクション

各キャッシュは個別のスタンドアロン Java Transaction API (JTA) リソースとして動作します。ただし、コンポーネントは最適化のために Red Hat JBoss Data Grid の内部で共有できますが、この共有は、キャッシュが Java Transaction API (JTA) Manager とどのように対話するかに影響を与えません。

30.4. トランザクションマネージャー

以下を使用して、キャッシュから TransactionManager を取得します。

```
TransactionManager tm = cache.getAdvancedCache().getTransactionManager();
```

トランザクション内で操作のシーケンスを実行するには、TransactionManager 上で begin() および commit() メソッド (または rollback() メソッド) への呼び出しを使ってこれらの操作をラップします。

操作の実行

```
tm.begin();
Object value = cache.get("A");
cache.remove("A");
Object prev = cache.put("B", value);
```

```
if (prev == null)
    tm.commit();
else
    tm.rollback();
```



注記

キャッシュメソッドが JTA トランザクションのスコープ内で `CacheException` (または、`CacheException` のサブクラス) を返すと、トランザクションはロールバックするよう自動的にマークされます。

Red Hat JBoss Data Grid `XAResource` への参照を取得するには、以下の API を使用します。

```
XAResource xar = cache.getAdvancedCache().getXAResource();
```

第31章 マーシャリング

31.1. マーシャリング

マーシャリングとは、Java オブジェクトを回線上で転送可能な形式に変換するプロセスのことです。アンマーシャリングはこれとは逆のプロセスで、回線上で転送可能な形式から読み取られたデータを Java オブジェクトに変換することを言います。

Red Hat JBoss Data Grid はマーシャリングおよびアンマーシャリングを使用して以下を行います。

- クラスター内の他の JBoss Data Grid ノードへリレーするためにデータを変換します。
- 基盤のキャッシュストアに保存するためにデータを変換します。

31.2. JOSS MARSHALLING FRAMEWORK

Red Hat JBoss Data Grid は JBoss Marshalling Framework を使用して Java **POJO** をマーシャリングおよびアンマーシャリングします。JBoss Marshalling Framework は優れたパフォーマンスを提供するため、Java のシリアライゼーションの代わりに使用されます。さらに、JBoss Marshalling Framework は Java クラスを含む Java **POJO** を効率的にマーシャリングできます。

Java Marshalling Framework は、標準の `java.io.ObjectOutputStream` や `java.io.ObjectInputStream` よりも高パフォーマンスな `java.io.ObjectOutput` および `java.io.ObjectInput` 実装を使用します。

31.3. シリアライズ不可能なオブジェクトのサポート

Red Hat JBoss Data Grid がシリアライズ不可能なオブジェクトのストレージをサポートするかどうかはユーザーに共通する懸念事項です。JBoss Data Grid では、シリアライズ不可能なキーと値のオブジェクトに対してマーシャリングはサポートされます。ユーザーはシリアライズ不可能なオブジェクトにエクスターナライザー実装を提供できます。

Serializable または **Externalizable** サポートをクラスに導入できない場合、一例として `XStream` を使用してシリアライズ不可能なオブジェクトを JBoss Data Grid に格納できる String に変換し、対応することもできます。



注記

必要となる XML トランスフォーメーションが原因で、キーと値のオブジェクトを格納するプロセスが遅くなります。

31.4. HOT ROD およびマーシャリング

リモートクライアントサーバーモードでは、Red Hat JBoss Data Grid サーバーとクライアントの両方のレベルでマーシャリングが発生しますが、詳細は異なります。

- JBoss Data Grid サーバーのクライアントによって保存されるすべてのデータは、バイトアレイまたは JBoss Data Grid のマーシャリングに対応するプリミティブな形式のいずれかとして提供されます。
JBoss Data Grid のサーバー側では、プリミティブな形式で保存されたデータがバイトアレイに変換され、クラスター周囲でのレプリケートまたはキャッシュストアへの保存が行われるとマーシャリングが発生します。JBoss Data Grid のサーバー側ではマーシャリングの設定は必要

ありません。

- POJO をシリアライズまたはデシリアライズするには、クライアントレベルでマーシャリングの **Marshaller** 設定要素が RemoteCacheManager 設定に指定されている必要があります。Hot Rod のバイナリーの性質により、マーシャリングに依存して POJO (キーまたは値) をバイトアレイに変換します。

31.5. REMOTECACHEMANAGER を使用したマーシャラーの設定

マーシャラーは RemoteCacheManager の **marshaller** 設定要素を使用して指定されます。その値は、Marshaller インターフェースを実装するクラスの名前である必要があります。このプロパティのデフォルト値は **org.infinispan.commons.marshall.jboss.GenericJBossMarshaller** です。



警告

独自のカスタムマーシャラーを開発する場合は、インスタンス化する前に読み取ったクラス名が想定されるまたは許可されるクラス名であることを検証して、潜在的なインジェクション攻撃から保護してください。

以下の手順は、RemoteCacheManager と使用するマーシャラーの定義方法を示しています。

マーシャラーの定義

1. 設定ビルダーの作成

ConfigurationBuilder を作成し、必要な設定を指定します。

```
ConfigurationBuilder builder = new ConfigurationBuilder();
//... (other configuration)
```

2. マーシャラークラスの追加

Marshaller メソッド内に Marshaller クラス仕様を追加します。

```
builder.marshaller(GenericJBossMarshaller.class);
```

a. この代わりに、カスタム Marshaller インスタンスを指定することもできます。

```
builder.marshaller(new GenericJBossMarshaller());
```

3. RemoteCacheManager の起動

Marshaller が含まれる設定を構築し、その設定で新しい RemoteCacheManager を起動します。

```
Configuration configuration = builder.build();
RemoteCacheManager manager = new RemoteCacheManager(configuration);
```

クライアントレベルでは、POJO は Serializable、Externalizable、またはプリミティブ型のいずれかである必要があります。



注記

Hot Rod Java クライアントは、Externalizer インスタンスの提供による POJO のシリアライズをサポートしません。これは、JBoss Data Grid のライブラリーモードでのみ使用できます。

31.6. デシリアライズを特定 **JAVA** クラスに制限

Red Hat JBoss Data Grid サーバーでは、ホワイトリストに指定した Java クラス以外に、標準の Java クラスとプリミティブのみデシリアライズできます。

その一方で、クライアントは、特定クラスのデシリアライズを制限しなければあらゆる Java クラスに属するオブジェクトをデシリアライズできます。これを行うには、**org.infinispan.client.hotrod.configuration.ConfigurationBuilder** クラスで **addJavaSerialWhiteList** メソッドを使用します。

たとえば、Person または Employee のいずれかが含まれる完全修飾名を持つ Java クラスのみにデシリアライズを制限するには、以下の設定を指定します。

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;

...
ConfigurationBuilder configBuilder = ...
configBuilder.addJavaSerialWhiteList(".*Person.*", ".*Employee.*");
```

JBoss Data Grid サーバーでデシリアライゼーションホワイトリストを設定するための詳細は、『Administration and Configuration Guide』の「[Configuring the Deserialization Whitelist](#)」を参照してください。

31.7. トラブルシューティング

31.7.1. マーシャリングのトラブルシューティング

Red Hat JBoss Data Grid では、ユーザーオブジェクトをマーシャリングまたはアンマーシャリングする時にマーシャリングレイヤーと JBoss Marshalling によってエラーが発生することがあります。問題をデバッグできるようにするため、例外スタックトレースに詳細情報が含まれます。

例外スタックトレース

```
java.io.NotSerializableException: java.lang.Object
at
org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.
java:857)
at
org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.ja
va:407)
at
org.infinispan.marshall.exts.ReplicableCommandExternalizer.writeObject(Rep
licableCommandExternalizer.java:54)
at
org.infinispan.marshall.jboss.ConstantObjectTable$ExternalizerAdapter.writ
eObject(ConstantObjectTable.java:267)
at
org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.
```

```

java:143)
at
org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectToObjectStream(JBossMarshaller.java:167)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToBuffer(VersionAwareMarshaller.java:92)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToByteBuffer(VersionAwareMarshaller.java:170)
at
org.infinispan.marshall.VersionAwareMarshallerTest.testNestedNonSerializable(VersionAwareMarshallerTest.java:415)
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
... Removed 22 stack frames

```

in object で始まるメッセージとスタックトレースは同様に読み取られます。一番上の **in object** メッセージは最も内側のもので、最も外側にある **in object** メッセージは一番下になります。

この例は、**java.lang.Object@b40ec4** はシリアライズ可能でないため、**org.infinispan.commands.write.PutKeyValueCommand** インスタンス内の **java.lang.Object** はシリアライズできないことを示しています。

しかし、**DEBUG** または **TRACE** ログレベルが有効である場合、スタックトレースにあるオブジェクトの **toString()** 表現がマーシャリング例外に含まれます。このような状況を表す例は次のとおりです。

ログレベルが有効である場合の例外

```

java.io.NotSerializableException: java.lang.Object
...
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
-> toString = java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
-> toString = PutKeyValueCommand{key=k, value=java.lang.Object@b40ec4,
putIfAbsent=false, lifespanMillis=0, maxIdleTimeMillis=0}

```

アンマーシャリング例外でこのレベルの情報を表示するとリソース的に高価になります。しかし、JBoss Data Grid は可能な場合はクラス型の情報を表示します。以下は、このようなレベルの情報が表示された例を示しています。

アンマーシャリングの例外

```

java.io.IOException: Injected failue!
at
org.infinispan.marshall.VersionAwareMarshallerTest$1.readExternal(VersionAwareMarshallerTest.java:426)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadNewObject(RiverUnmarsh

```

```

aller.java:1172)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshall
er.java:273)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshall
er.java:210)
at
org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller
.java:85)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectFromObjectStream(JBoss
Marshaller.java:210)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(Versio
nAwareMarshaller.java:104)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(Versio
nAwareMarshaller.java:177)
at
org.infinispan.marshall.VersionAwareMarshallerTest.testErrorUnmarshalling(
VersionAwareMarshallerTest.java:431)
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.VersionAwareMarshallerTest$1

```

上記の例では、内部クラス **org.infinispan.marshall.VersionAwareMarshallerTest\$1** のインスタンスがアンマーシャルされたときに **IOException** が発生しました。

DEBUG または **TRACE** ログレベルが有効な場合、マーシャリング例外と似た方法で、クラスタイプのクラスローダー情報が提供されます。このクラスローダー情報の例は次のとおりです。

クラスローダー情報

```

java.io.IOException: Injected failue!
...
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.VersionAwareMarshallerTest$1
-> classloader hierarchy:
-> type classloader = sun.misc.Launcher$AppClassLoader@198dfaf
-
>...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/ec
lipse-testng.jar
-
>...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/lib
/testng-jdk15.jar
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/test-
classes/
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/classes/
->...file:/home/galder/.m2/repository/org/testng/testng/5.9/testng-5.9-
jdk15.jar
->...file:/home/galder/.m2/repository/net/jcip/jcip-annotations/1.0/jcip-
annotations-1.0.jar
-
>...file:/home/galder/.m2/repository/org/easymock/easymockclassexension/2
.4/easymockclassexension-2.4.jar
->...file:/home/galder/.m2/repository/org/easymock/easymock/2.4/easymock-

```



```

2.4.jar
->...file:/home/galder/.m2/repository/cglib/cglib-nodep/2.1_3/cglib-nodep-
2.1_3.jar
->...file:/home/galder/.m2/repository/javax/xml/bind/jaxb-api/2.1/jaxb-
api-2.1.jar
->...file:/home/galder/.m2/repository/javax/xml/stream/stax-api/1.0-
2/stax-api-1.0-2.jar
-
>...file:/home/galder/.m2/repository/javax/activation/activation/1.1/activ
ation-1.1.jar
->...file:/home/galder/.m2/repository/jgroups/jgroups/2.8.0.CR1/jgroups-
2.8.0.CR1.jar
->...file:/home/galder/.m2/repository/org/jboss/javaee/jboss-transaction-
api/1.0.1.GA/jboss-transaction-api-1.0.1.GA.jar
-
>...file:/home/galder/.m2/repository/org/jboss/marshalling/river/1.2.0.CR4
-SNAPSHOT/river-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/marshalling/marshalling-
api/1.2.0.CR4-SNAPSHOT/marshalling-api-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/jboss-common-
core/2.2.14.GA/jboss-common-core-2.2.14.GA.jar
->...file:/home/galder/.m2/repository/org/jboss/logging/jboss-logging-
spi/2.0.5.GA/jboss-logging-spi-2.0.5.GA.jar
->...file:/home/galder/.m2/repository/log4j/log4j/1.2.14/log4j-1.2.14.jar
-
>...file:/home/galder/.m2/repository/com/thoughtworks/xstream/xstream/1.2/
xstream-1.2.jar
->...file:/home/galder/.m2/repository/xpp3/xpp3_min/1.1.3.4.0/xpp3_min-
1.1.3.4.0.jar
->...file:/home/galder/.m2/repository/com/sun/xml/bind/jaxb-
impl/2.1.3/jaxb-impl-2.1.3.jar
-> parent classloader = sun.misc.Launcher$ExtClassLoader@1858610
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/localedata.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunpkcs11.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunjce_provider.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/dnsns.jar
... Removed 22 stack frames

```

31.7.2. その他のマーシャリング関連の問題

マーシャリングに関連する問題および例外は、**EOFException** による状態遷移の間などの異なる状況で発生することもあります。状態遷移の間、状態レシーバーが Read past end of file であるという内容の **EOFException** がログに記録された場合、状態の生成時に状態プロバイダーにエラーが発生するかどうかに応じて対応することができます。たとえば、現在状態プロバイダーが状態をノードに提供している場合、他のノードが状態を要求すると、状態ジェネレーターのログには以下が含まれる可能性があります。

状態ジェネレーターログ

```

2010-12-09 10:26:21,533 20267 ERROR
[org.infinispan.remoting.transport.jgroups.JGroupsTransport]
(STREAMING_STATE_TRANSFER-sender-1,Infinispan-Cluster,NodeJ-2368:) Caught
while responding to state transfer request
org.infinispan.statetransfer.StateTransferException:
java.util.concurrent.TimeoutException: Could not obtain exclusive

```

```

processing lock
    at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateT
ransferManagerImpl.java:175)
    at
org.infinispan.remoting.InboundInvocationHandlerImpl.generateState(Inbound
InvocationHandlerImpl.java:119)
    at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.getState(JGroup
sTransport.java:586)
    at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.handleUpEvent(Message
Dispatcher.java:691)
    at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.up(MessageDispatcher.
java:772)
    at org.jgroups.JChannel.up(JChannel.java:1465)
    at org.jgroups.stack.ProtocolStack.up(ProtocolStack.java:954)
    at org.jgroups.protocols.pbcast.FLUSH.up(FLUSH.java:478)
    at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderHandler
.process(STREAMING_STATE_TRANSFER.java:653)
    at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderThreadS
pawner$1.run(STREAMING_STATE_TRANSFER.java:582)
    at
java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.
java:886)
    at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java
:908)
    at java.lang.Thread.run(Thread.java:680)
Caused by: java.util.concurrent.TimeoutException: Could not obtain
exclusive processing lock
    at
org.infinispan.remoting.transport.jgroups.JGroupsDistSync.acquireProcessin
gLock(JGroupsDistSync.java:71)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.generateTransactionL
og(StateTransferManagerImpl.java:202)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateT
ransferManagerImpl.java:165)
    ... 12 more

```

ログでもマーシャリングに関連するように見受けられる例外を発見することもできますが、例外の根本的な理由は異なることがあります。この例外は、状態ジェネレーターがトランザクションログを生成できなかったため、書き込み中の出力が閉じられたことを意味しています。このような状況で、送信側によって書き込まれなかったトランザクションログの読み取りに失敗した場合、状態レシーバーは通常次のような **EOFException** をログに表示します。

EOFException

```

2010-12-09 10:26:21,535 20269 TRACE
[org.infinispan.marshall.VersionAwareMarshaller] (Incoming-2,Infinispan-
Cluster,NodeI-38030:) Log exception reported

```

```
java.io.EOFException: Read past end of file
    at
org.jboss.marshalling.AbstractUnmarshaller.eofOnRead(AbstractUnmarshaller.
java:184)
    at
org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByteDirect(Abstract
Unmarshaller.java:319)
    at
org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByte(AbstractUnmars
haller.java:280)
    at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshall
er.java:207)
    at
org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller
.java:85)
    at
org.infinispan.marshall.jboss.GenericJBossMarshaller.objectFromObjectStrea
m(GenericJBossMarshaller.java:175)
    at
org.infinispan.marshall.VersionAwareMarshaller.objectFromObjectStream(Vers
ionAwareMarshaller.java:184)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.processCommitLog(Sta
teTransferManagerImpl.java:228)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.applyTransactionLog(
StateTransferManagerImpl.java:250)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.applyState(StateTran
sferManagerImpl.java:320)
    at
org.infinispan.remoting.InboundInvocationHandlerImpl.applyState(InboundInv
ocationHandlerImpl.java:102)
    at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.setState(JGroup
sTransport.java:603)
    ...
```

このエラーが発生すると、状態レシーバーは正常に完了するまで毎数秒ごとにこの操作を試行します。ほとんどの場合、最初の試行後に状態ジェネレーターは2つ目のノードの処理を完了し、想定どおりに状態を完全に受け入れます。

第32章 INFINISPAN CDI モジュール

32.1. INFINISPAN CDI モジュール

Infinispan の **infinispan-cdi** モジュールには Context and Dependency Injection (CDI) が含まれています。 **infinispan-cdi** モジュールは以下を提供します。

- Cache API を使用した設定およびインジェクション。
- キャッシュリスナーと CDI イベントシステム間のブリッジ。
- JCACHE をキャッシュするアノテーションの部分サポート。

32.2. INFINISPAN CDI の使用

32.2.1. Infinispan CDI の要件

以下は Infinispan CDI モジュールを Red Hat JBoss Data Grid と使用するための要件になります。

- 最新バージョンの **infinispan-cdi** モジュールを使用してください。
- 依存関係の情報を正しく設定してください。

32.2.2. CDI Maven 依存関係の設定

CDI モジュールは各デプロイメント型の Infinispan jar に含まれています。他に必要な依存関係はありません。

ライブラリーモード

ライブラリーモードでは、**infinispan-embedded** アーティファクトに CDI モジュールが含まれています。以下の例のように依存関係として追加する必要があります。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-embedded</artifactId>
  <version>${infinispan.version}</version>
</dependency>
```

リモートクライアントサーバーモード

リモートクライアントサーバーモードでは、**infinispan-remote** アーティファクトに CDI モジュールが含まれています。以下の例のように依存関係として追加する必要があります。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-remote</artifactId>
  <version>${infinispan.version}</version>
</dependency>
```

32.3. INFINISPAN CDI モジュールの使用

32.3.1. Infinispan CDI モジュールの使用

Infinispan CDI モジュールは以下の目的に使用できます。

- Infinispan キャッシュを CDI Bean および Java EE コンポーネントに設定およびインジェクトする。
- キャッシュマネージャーの設定。
- CDI アノテーションを使用した格納および取得の制御。

32.3.2. Infinispan キャッシュの設定およびインジェクション

32.3.2.1. Infinispan キャッシュのインジェクション

Infinispan キャッシュは、プロジェクトの CDI Bean にインジェクトできる複数のコンポーネントの 1 つです。

以下のコードスニペットはキャッシュインスタンスを CDI Bean にインジェクトする方法を示しています。

```
public class MyCDIBean {
    @Inject
    Cache<String, String> cache;
}
```

32.3.2.2. リモート Infinispan キャッシュのインジェクション

リモート Infinispan キャッシュをインジェクトするには、普通のキャッシュをインジェクトするコードスニペットを次のように若干変更します。

```
public class MyCDIBean {
    @Inject
    RemoteCache<String, String> remoteCache;
}
```

32.3.2.3. インジェクションのターゲットキャッシュの設定

32.3.2.3.1. インジェクションのターゲットキャッシュの設定

インジェクションのターゲットキャッシュを設定するには、次の 3 つのステップに従います。

1. 修飾子アノテーションを作成します。
2. プロデューサークラスを追加します。
3. 希望のクラスをインジェクトします。

32.3.2.3.2. 修飾子アノテーションの作成

CDI を使用して特定のキャッシュを返すには、次のようにカスタムのキャッシュ修飾子アノテーションを作成します。

カスタムキャッシュ修飾子

```
@javax.inject.Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface SmallCache {}
```

作成した **@SmallCache** 修飾子を使用して、特定のキャッシュの作成方法を指定します。

32.3.2.3.3. プロデューサークラスの追加

以下のコードスニペットは、**@SmallCache** 修飾子 (前のステップで作成) がどのようにキャッシュの作成方法を指定するかを示しています。

@SmallCache 修飾子の使用

```
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.cdi.ConfigureCache;
import javax.enterprise.inject.Produces;

public class CacheCreator {
    @ConfigureCache("smallcache")
    @SmallCache
    @Produces
    public Configuration specialCacheCfg() {
        return new ConfigurationBuilder()
            .memory()
            .size(10)
            .build();
    }
}
```

コードスニペットの要素は次のとおりです。

- **@ConfigureCache** はキャッシュの名前を指定します。
- **@SmallCache** はキャッシュ修飾子です。

32.3.2.3.4. 希望のクラスのインジェクト

以下のように **@SmallCache** 修飾子と新しいプロデューサークラスを使用して、特定のキャッシュを CDI Bean にインジェクトします。

```
public class MyCDIBean {
    @Inject @SmallCache
    Cache<String, String> mySmallCache;
}
```

32.3.3. CDI を用いたキャッシュマネージャーの設定

32.3.3.1. CDI を用いたキャッシュマネージャーの設定

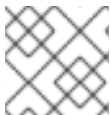
Red Hat JBoss Data Grid のキャッシュマネージャー (組み込みとリモートの両方) は CDI を使用して設定できます。設定するキャッシュマネージャーが組み込みかリモートであるかに関わらず、最初の手順として、アノテーションを付けてプロデューサーとして動作するデフォルト設定を指定します。

32.3.3.2. デフォルト設定の指定

Red Hat JBoss Data Grid 設定に対してプロデューサーとしてアノテーション付けされたメソッドを指定し、デフォルトの Infinispan 設定を置き換えます。以下の設定例はこの手順を示しています。

デフォルト設定の指定

```
public class Config {
    @Produces
    public Configuration defaultEmbeddedConfiguration () {
        return new ConfigurationBuilder()
            .memory()
            .size(100)
            .build();
    }
}
```



注記

他の修飾子が提供されない場合、CDI は **@Default** 修飾子を追加します。

@Produces アノテーションが設定インスタンスを返すメソッドに配置された場合、Configuration オブジェクトが必要なときにメソッドが呼び出されます。

この設定例では、作成後に設定され返される新しい Configuration オブジェクトが作成されます。

32.3.3.3. 組み込みキャッシュマネージャーの作成のオーバーライド

前提条件

「[デフォルト設定の指定](#)」参照してください。

非クラスターキャッシュの作成

プロデューサーメソッドにアノテーションが付けられた後、以下のように **EmbeddedCacheManager** の作成時にこのメソッドが呼び出されます。

```
public class Config {

    @Produces
    @ApplicationScoped
    public EmbeddedCacheManager defaultEmbeddedCacheManager() {
        Configuration cfg = new ConfigurationBuilder()
            .memory()
            .size(150)
            .build();

        return new DefaultCacheManager(cfg);
    }
}
```

■

@ApplicationScoped アノテーションはメソッドが 1 度だけ呼び出されることを指定します。

クラスターキャッシュの作成

以下の設定は、クラスターキャッシュの作成が可能な **EmbeddedCacheManager** の作成に使用できます。

```
public class Config {

    @Produces
    @ApplicationScoped
    public EmbeddedCacheManager defaultClusteredCacheManager() {
        GlobalConfiguration g = new GlobalConfigurationBuilder()
            .clusteredDefault()
            .transport()
            .clusterName("InfinispanCluster")
            .build();
        Configuration cfg = new ConfigurationBuilder()
            .memory()
            .size(150)
            .build();
        return new DefaultCacheManager(g, cfg);
    }
}
```

EmbeddedCacheManager を生成するメソッドの呼び出し

非クラスターメソッドの **@Produces** アノテーションが付けられたメソッドは **Configuration** オブジェクトを生成します。クラスターキャッシュの例で **@Produces** アノテーションが付けられたメソッドは **EmbeddedCacheManager** オブジェクトを生成します。

以下のように CDI Bean にインジェクションを追加し、適切にアノテーションが付けられたメソッドを呼び出します。これにより **EmbeddedCacheManager** が生成され、起動時にコードにインジェクトされます。

EmbeddedCacheManager の生成

```
...
@Inject
EmbeddedCacheManager cacheManager;
...
```

32.3.3.4. リモートキャッシュマネージャーの設定

RemoteCacheManager は、以下のように **EmbeddedCacheManagers** と似た方法で設定されます。

リモートキャッシュマネージャーの設定

```
public class Config {
    @Produces
    @ApplicationScoped
    public RemoteCacheManager defaultRemoteCacheManager() {
        Configuration conf = new
        ConfigurationBuilder().addServer().host(ADDRESS).port(PORT).build();
```



```

        return new RemoteCacheManager(conf);
    }
}

```

32.3.3.5. 単一クラスでの複数のキャッシュマネージャーの設定

単一のクラスを使用して、作成された修飾子を基に複数のキャッシュマネージャーおよびリモートキャッシュマネージャーを設定できます。以下に例を示します。

複数のキャッシュマネージャーの設定

```

public class Config {
    @Produces
    @ApplicationScoped
    public org.infinispan.manager.EmbeddedCacheManager
    defaultEmbeddedCacheManager() {
        Configuration cfg = new ConfigurationBuilder()
            .memory()
            .size(150)
            .build();
        return new DefaultCacheManager(cfg);
    }

    @Produces
    @ApplicationScoped
    @DefaultClustered
    public org.infinispan.manager.EmbeddedCacheManager
    defaultClusteredCacheManager() {
        GlobalConfiguration g = new GlobalConfigurationBuilder()
            .clusteredDefault()
            .transport()
            .clusterName("InfinispanCluster")
            .build();
        Configuration cfg = new ConfigurationBuilder()
            .memory()
            .size(150)
            .build();
        return new DefaultCacheManager(g, cfg);
    }

    @Produces
    @ApplicationScoped
    @DefaultRemote
    public RemoteCacheManager
    defaultRemoteCacheManager() {
        org.infinispan.client.hotrod.configuration.Configuration conf =
new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder().addServe
r().host(ADDRESS).port(PORT).build();
        return new RemoteCacheManager(conf);
    }

    @Produces
    @ApplicationScoped
    @RemoteCacheInDifferentDataCentre

```

```

    public RemoteCacheManager newRemoteCacheManager() {
        org.infinispan.client.hotrod.configuration.Configuration confid =
new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder().addServe
r().host(ADDRESS_FAR_AWAY).port(PORT).build();
        return new RemoteCacheManager(confid);
    }
}

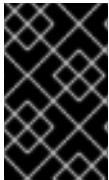
```

32.4. CDI アノテーションを使用した格納および取得

32.4.1. キャッシュアノテーションの設定

特定の CDI アノテーションは JCache (JSR-107) 仕様で受け入れられます。含まれるアノテーションはすべて **javax.cache** パッケージにあります。

アノテーションインターセプトメソッドは CDI Bean を呼び出し、インターセプトの結果として格納および取得タスクを実行します。



重要

CDI はリモートクライアントサーバーモードとライブラリーモードの両方でサポートされますが、`@CachePut`、`@CacheRemove`、`@CacheRemoveAll`、`@CacheResult` などのアノテーションはリモートクライアントサーバーモードでは使用できません。

32.4.2. キャッシュアノテーションの有効化

JBoss Data Grid には、使用方法に応じてインターセプターが 2 セット含まれています。**beans.xml** ファイルを使用するとインターセプターを CDI Bean アーカイブに追加できます。

オプション 1: CDI インターセプター

以下のコードを追加する

と、**InjectedCacheResultInterceptor**、**InjectedCachePutInterceptor**、**InjectedCacheRemoveEntryInterceptor**、および **InjectedCacheRemoveAllInterceptor** などのインターセプターが追加されます。

CDI インターセプターの追加

```

<beans 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/beans_1_0.xsd" >
    <interceptors>

<class>org.infinispan.jcache.annotation.InjectedCacheResultInterceptor</cl
ass>

<class>org.infinispan.jcache.annotation.InjectedCachePutInterceptor</class
>

<class>org.infinispan.jcache.annotation.InjectedCacheRemoveEntryIntercepto
r</class>

```

```
<class>org.infinispan.jcache.annotation.InjectedExceptionHandler<
/class>
</interceptors>
</beans>
```

オプション 2: JCache インターセプター

以下のコードを追加すると

CacheResultInterceptor、**CachePutInterceptor**、**CacheRemoveEntryInterceptor**、および **CacheRemoveAllInterceptor** などのインターセプターが追加されます。

JCache インターセプターの追加

```
<beans 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/beans_1_0.xsd">

    <interceptors>

<class>org.infinispan.jcache.annotation.CacheResultInterceptor</class>
        <class>org.infinispan.jcache.annotation.CachePutInterceptor</class>

<class>org.infinispan.jcache.annotation.CacheRemoveEntryInterceptor</class>
    >

<class>org.infinispan.jcache.annotation.CacheRemoveAllInterceptor</class>
    </interceptors>

</beans>
```



注記

Red Hat JBoss Data Grid が javax.cache アノテーションを使用するには、記載されているインターセプターが **beans.xml** ファイルに存在する必要があります。

32.4.3. メソッド呼び出しの結果をキャッシュ

32.4.3.1. メソッド呼び出しの結果をキャッシュ

今後のアクセスのために結果をキャッシュに保存することは、時間またはリソース集約型の操作で一般的に行われます。以下のコードはこのような操作の例になります。

```
public String toCelsiusFormatted(float fahrenheit) {
    return
        NumberFormat.getInstance()
            .format((fahrenheit * 5 / 9) - 32)
            + " degrees Celsius";
}
```

一般的には、メソッド呼び出しの結果をキャッシュし、次に結果が必要になるときにキャッシュをチェックします。以下は、このような操作の結果をキャッシュで検索するコードスニペットの例になります。結果が見つからない場合、コードスニペットは **toCelsiusFormatted** を再実行し、結果をキャッシュに格納します。

```
float f = getTemperatureInFahrenheit();
Cache<Float, String>
    fahrenheitToCelsiusCache = getCache();
String celsius =
    fahrenheitToCelsiusCache.get(f);
    if (celsius == null) {
        celsius = toCelsiusFormatted(f);
        fahrenheitToCelsiusCache.put(f, celsius);
    }
```

このような場合、Infinispan CDI モジュールを使用すると、関連する例で余分なコードを削除できます。以下のように、メソッドに **@CacheResult** アノテーションを付けます。

```
@javax.cache.annotation.CacheResult
public String toCelsiusFormatted(float fahrenheit) {
    return NumberFormat.getInstance()
        .format((fahrenheit * 5 / 9) - 32)
        + " degrees Celsius";
}
```

アノテーションにより、Infinispan はキャッシュをチェックし、結果が見つからなかった場合は **toCelsiusFormatted()** メソッドを呼び出します。



注記

Infinispan CDI モジュールでは、保存した結果のキャッシュをチェックできますが、これは適用前に注意して考慮する必要があります。呼び出しの結果が常に最新のデータである必要がある場合やキャッシュの読み取りにリモートネットワークの検索やキャッシュローダーからのデシリアライズが必要である場合、コールメソッド呼び出し前のキャッシュのチェックは逆効果になる可能性があります。

32.4.3.2. 使用するキャッシュの指定

以下のオプションの属性 (**cacheName**) を **@CacheResult** アノテーションに追加し、メソッドコールの結果をチェックするキャッシュを指定します。

```
@CacheResult(cacheName = "mySpecialCache")
public String doSomething(String parameter) {
    <!-- Additional configuration information here -->
}
```

32.4.3.3. キャッシュされた結果のキャッシュキー

デフォルトでは、**@CacheResult** アノテーションはキャッシュから取得された結果のキーを作成します。このキーは、関連するメソッドのすべてのパラメーターの組み合わせで構成されます。

次のように **@CacheKey** アノテーションを使用してカスタムキーを作成します。

カスタムキーの作成

```
@CacheResult
public String doSomething
    (@CacheKey String p1,
```

```
@CacheKey String p2,
String dontCare) {
<!-- Additional configuration information here -->
}
```

この例では、キャッシュキーの作成に **p1** と **p2** のみが値として使用されます。キャッシュキーの判断には **dontCare** の値は使用されません。

32.4.3.4. カスタムキーの生成

次のようにカスタムキーを生成します。

```
import javax.cache.annotation.CacheKey;
import javax.cache.annotation.CacheKeyGenerator;
import javax.cache.annotation.CacheKeyInvocationContext;
import java.lang.annotation.Annotation;

public class MyCacheKeyGenerator implements CacheKeyGenerator {

    @Override
    public CacheKey generateCacheKey(CacheKeyInvocationContext<? extends
Annotation> ctx) {

        return new MyCacheKey(
            ctx.getAllParameters()[0].getValue()
        );
    }
}
```

このメソッドはカスタムキーを構築します。このキーは呼び出しコンテキストの最初のパラメーターによって生成された値の一部として渡されます。

カスタムのキー生成スキームを指定するには、以下のように任意のパラメーター **cacheKeyGenerator** を **@CacheResult** アノテーションに追加します。

```
@CacheResult(cacheKeyGenerator = MyCacheKeyGenerator.class)
public void doSomething(String p1, String p2) {
<!-- Additional configuration information here -->
}
```

提供されたメソッドを使用し、**p1** にカスタムキーが含まれます。

32.4.4. キャッシュ操作

32.4.4.1. キャッシュエントリーの更新

@CachePut アノテーションが含まれるメソッドが呼び出されると、パラメーター (通常は **@CacheValue** アノテーションが付けられたメソッドに渡されます) はキャッシュに格納されます。

@CachePut アノテーションが付けられたメソッドの例

```
import javax.cache.annotation.CachePut;
import javax.cache.annotation.CacheKey;
```

```
import javax.cache.annotation.CacheValue;

@CachePut (cacheName = "personCache")
public void updatePerson
    (@CacheKey long personId,
    @CacheValue Person newPerson) {
    <!-- Additional configuration information here -->
}
```

@CachePut メソッドの **cacheName** および **cacheKeyGenerator** を使用するとさらにカスタマイズすることができます。また、呼び出されたメソッドの一部のパラメーターに **@CacheKey** アノテーションを付けるとキーの生成を制御できます。

「[キャッシュされた結果のキャッシュキー](#)」も参照してください。

32.4.4.2. キャッシュからのエントリーの削除

以下は、キャッシュからエントリーを削除するために使用される **@CacheRemoveEntry** アノテーションが付けられたメソッドの例になります。

キャッシュからエントリーを削除

```
import javax.cache.annotation.CacheRemoveEntry;
import javax.cache.annotation.CacheKey;

@CacheRemoveEntry (cacheName = "cacheOfPeople")
public void changePersonName
    (@CacheKey long personId,
    string newName) {
    <!-- Additional configuration information here -->
}
```

このアノテーションは、任意の **cacheName** および **cacheKeyGenerator** 属性を受け入れます。

32.4.4.3. キャッシュの消去

@CacheRemoveAll メソッドを呼び出して、キャッシュからすべてのエントリーを消去します。

@CacheRemoveAll を使用してキャッシュからすべてのエントリーを消去

```
import javax.cache.annotation.CacheRemoveAll;

@CacheRemoveAll (cacheName = "statisticsCache")
public void resetStatistics() {
    <!-- Additional configuration information here -->
}
```

例のとおり、このアノテーションは任意の **cacheName** 属性を受け入れます。

第33章 SPRING FRAMEWORK との統合

33.1. SPRING FRAMEWORK との統合

JBoss Data Grid では、ユーザーは Spring キャッシュプロバイダーを定義できるため、アプリケーションにキャッシュサポートを簡単に追加する方法を提供し、Spring のプログラミングモデルに精通したユーザーは JBoss Data Grid によるキャッシングを実現できます。

33.2. SPRING MAVEN 依存関係の定義

Spring モジュールは、ライブラリーの依存関係およびリモートクライアントサーバーの依存関係とは別にバンドルされます。JBoss Data Grid の使用方法に応じて、以下の Maven 設定を使用する必要があります。

ライブラリーモードの Spring 4 用の pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-spring4-embedded</artifactId>
  <version>${infinispan.version}</version>
</dependency>
```

リモートクライアントサーバーモードの Spring 4 用の pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-spring4-remote</artifactId>
  <version>${infinispan.version}</version>
</dependency>
```

33.3. プログラミングによる **SPRING** キャッシュサポートの有効化 (ライブラリーモード)

Spring のキャッシュサポートは、アプリケーションでプログラミングを使用して有効にできます。Spring のサポートを有効にするには、以下の手順を実行します。

1. 使用している Spring 設定クラスに **@EnableCaching** アノテーションを追加します。
2. **@Bean** アノテーションが付けられた **SpringEmbeddedCacheManager** を返すメソッドを定義します。

以下のコードスニペットにはこれらの変更が反映されています。

プログラミングによる設定の例

```
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.spring.provider.SpringEmbeddedCacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Bean;
[...]
```

```

@org.springframework.context.annotation.Configuration
@EnableCaching
public class Config {

    [...]

    @Bean
    public SpringEmbeddedCacheManager cacheManager() throws Exception {
        Configuration config = new ConfigurationBuilder()
            .memory()
            .size(150)
            .build();

        return SpringEmbeddedCacheManager(new
DefaultCacheManager(config));
    }
    [...]

```

33.4. プログラミングによる **SPRING** キャッシュサポートの有効化 (リモートクライアントサーバーモード)

Spring のキャッシュサポートは、以下の手順を実行し、アプリケーションでプログラミングを使用すると有効にできます。

1. 使用している Spring 設定クラスに **@EnableCaching** アノテーションを追加します。
2. **@Bean** アノテーションが付けられた **SpringRemoteCacheManager** を返すメソッドを定義します。

以下のコードスニペットにはこれらの変更が反映されています。

プログラミングによる設定の例

```

import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.configuration.Configuration;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.spring.provider.SpringRemoteCacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Bean;
[... ]

@org.springframework.context.annotation.Configuration
@EnableCaching
public class Config {

    [...]

    @Bean
    public SpringRemoteCacheManager cacheManager() throws Exception {
        Configuration config = new ConfigurationBuilder()
            .addServer()
            .host(ADDRESS)
            .port(PORT)
            .build();

```



```

        return new SpringRemoteCacheManager(new
RemoteCacheManager(config));
    }
    [...]

```

33.5. アプリケーションコードへのキャッシングの追加

『[Spring Cache Abstraction](#)』に記載されている Spring アノテーションを使用すると、キャッシングを各アプリケーションに追加できます。

キャッシュエントリーの追加

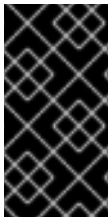
キャッシュにエントリーを追加するには、**@Cacheable** アノテーションをメソッドに追加します。このアノテーションは戻り値を指定のキャッシュに追加します。たとえば、特定のキーを基に **Book** を返すメソッドの場合、次のように **@Cacheable** アノテーションを追加します。

```

@Cacheable(value = "books", key = "#bookId")
public Book findBook(Integer bookId) {...}

```

findBook(Integer bookId) から返されたすべての **Book** インスタンスは、**bookId** を値のキーとして使用し、**books** という名前のキャッシュに置かれます。



重要

key 属性が指定されていない場合、Spring は提供された引数からハッシュを生成し、生成されたこの値をキャッシュキーとして使用します。アプリケーションが直接エントリーを参照する必要がある場合、エントリーを簡単に取得できるようにするため、key 属性を含めることが推奨されます。

キャッシュエントリーの削除

キャッシュからのこのエントリーを削除するには、メソッドに **@CacheEvict** アノテーションを付けます。このアノテーションを設定すると、キャッシュのすべてのエントリーをエビクトしたり、指定のキーを持つエントリーのみを対象としたりすることができます。以下の例を見てください。

```

// Evict all entries in the "books" cache
@CacheEvict (value="books", key = "#bookId", allEntries = true)
public void deleteBookAllEntries() {...}

// Evict any entries in the "books" cache that match the passed in bookId
@CacheEvict (value="books", key = "#bookId")
public void deleteBook(Integer bookId) {...}}

```

第34章 APACHE SPARK との統合

34.1. JBOSS DATA GRID の APACHE SPARK コネクタ

JBoss Data Grid には Spark コネクタが含まれ、Apache Spark との密な統合を提供します。また、Java または Scala で書かれたアプリケーションが JBoss Data Grid をバックエンドデータストアとして使用できるようにします。

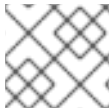
実際には、Apache Spark 1.6.x をサポートするコネクタと、Apache Spark 2.x をサポートするコネクタの 2 つがあり、それぞれ Scala 2.10.x と 2.11.x をサポートします。両方のコネクタは、メインディストリビューションとは別に提供されます。

Apache Spark 1.6 コネクタには以下のサポートが含まれています。

- キャッシュからの RDD の作成
- キーバリュー RDD のキャッシュへの書き込み
- キャッシュレベルイベントからの DStream の作成
- キーバリュー DStream のキャッシュへの書き込み

上記の機能の他に、Apache Spark 2 コネクタは以下の機能もサポートします。

- JDG サーバー側フィルターを使用したキャッシュベースの RDD の作成
- JBoss Marshalling を基にした Spark シリアライザー
- 述語のプッシュダウンをサポートする Dataset API



注記

Apache Spark のサポートは、リモートクライアントサーバーモードでのみ有効です。

34.2. SPARK の依存関係

Apache Spark のバージョンに応じて、以下の Maven 設定を使用する必要があります。

Spark 1.6.x の pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-spark</artifactId>
  <version>0.3.0.Final-redhat-2</version>
</dependency>
```

Spark 2.x の pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-spark</artifactId>
  <version>0.6.0-redhat-9</version>
</dependency>
```

34.3. SPARK コネクターの設定

Apache Spark のバージョン 1.6 コネクターとバージョン 2 コネクターは、設定に同じインターフェースを使用しません。バージョン 1.6 コネクターはプロパティを使用し、バージョン 2 コネクターはメソッドを使用します。

34.3.1. バージョン 1.6 コネクターを設定するプロパティ

表34.1 バージョン 1.6 コネクターを設定するプロパティ

プロパティ名	説明	デフォルト値
<code>infinispan.client.hotrod.server_list</code>	JBoss Data Grid ノードのリスト	localhost:11222
<code>infinispan.rdd.cacheName</code>	RDD をバックするキャッシュの名前	デフォルトキャッシュ
<code>infinispan.rdd.read_batch_size</code>	キャッシュから読み取りするときのバッチサイズ (エントリー数)	10000
<code>infinispan.rdd.write_batch_size</code>	キャッシュへ書き込みするときのバッチサイズ (エントリー数)	500
<code>infinispan.rdd.number_server_partitions</code>	JBoss Data Grid サーバーごとに作成されるパーティションの数	2
<code>infinispan.rdd.query.proto.protofiles</code>	protobuf ファイル名およびコンテンツを持つマップ	エントリーに protobuf エンコーディング情報のアノテーションが付けられている場合は省略できます。protobuf エンコーディングは、クエリーによる RDD のフィルターに必要です。
<code>infinispan.rdd.query.proto.marshallers</code>	キャッシュにあるオブジェクトの protostream マーシャラークラスのリスト	エントリーに protobuf エンコーディング情報のアノテーションが付けられている場合は省略できます。protobuf エンコーディングは、クエリーによる RDD のフィルターに必要です。

34.3.2. バージョン 2 コネクターを設定するメソッド

以下のメソッドは、バージョン 2 コネクターの設定に使用できます。これらのメソッドは `org.infinispan.spark.config.ConnectorConfiguration` クラスによって提供されます。

表34.2 バージョン 2 コネクターを設定するメソッド

メソッド名	説明	デフォルト値
<code>setServerList(String)</code>	JBoss Data Grid ノードのリスト	localhost:11222

メソッド名	説明	デフォルト値
setCacheName(String)	RDD をバックするキャッシュの名前	デフォルトキャッシュ
setReadBatchSize(Integer)	キャッシュから読み取りするときのバッチサイズ (エントリー数)	10000
setWriteBatchSize(Integer)	キャッシュへ書き込みするときのバッチサイズ (エントリー数)	500
setPartitions(Integer)	JBoss Data Grid サーバーごとに作成されるパーティションの数	2
addProtoFile(String name, String contents)	protobuf ファイル名およびコンテンツを持つマップ	エントリーに protobuf エンコーディング情報のアノテーションが付けられている場合は省略できます。protobuf エンコーディングは、クエリーによる RDD のフィルターに必要です。
addMessageMarshaller(Class)	キャッシュにあるオブジェクトの protostream マーシャラークラスのリスト	エントリーに protobuf エンコーディング情報のアノテーションが付けられている場合は省略できます。protobuf エンコーディングは、クエリーによる RDD のフィルターに必要です。
addProtoAnnotatedClass(Class)	protobuf アノテーションが含まれるクラスの登録	addProtoFile および addMessageMarshaller メソッドはアノテーションを基に自動生成されるため、これらの代わりに使用します。
setAutoRegisterProto()	protobuf スキーマをサーバーで自動登録	なし
addHotRodClientProperty(key, value)	JBoss Data Grid サーバーとの通信時に追加の Hot Rod クライアントプロパティを設定	なし
setTargetEntity(Class)	クエリーターゲットを指定するために Dataset API とともに使用	省略した場合、protobuf のアノテーションが付けられた 1 つのクラスのみが設定されていれば、そのクラスが選択されます。

34.3.3. セキュアな JDG クラスターへの接続

JDG クラスターがセキュアな場合、Hot Rod にセキュリティーを設定しないと、Spark コネクターは動作しません。Hot Rod セキュリティーを設定するためにセットアップできる Hot Rod プロパティは複数あります。これらのプロパティの説明は、『Administration and Configuration Guide』の付録にあ

る「[Hot Rod Properties table](#)」の `infinispan.client.hotrod.use_ssl` 以降に記載されています。



重要

これらのプロパティは、バージョン 2 Apache Spark コネクタのみに限定されます。

34.4. SPARK 1.6 のコード例

34.4.1. Spark 1.6 のコード例

Apache Spark 1.6 のコネクタは、バージョン 2 コネクタとは異なる設定メカニズムを使用し、バージョン 2 コネクタは 1.6 がサポートしない機能を一部サポートするため、各バージョンのコード例は独自のセクションに分割されています。以下のコード例は、バージョン 1.6 の Spark コネクタと動作します。[Spark 2 のコード例はこちら](#)を参照してください。

34.4.2. RDD の作成および使用

Apache Spark 1.6 コネクタを使用する場合、RDD は「[バージョン 1.6 コネクタを設定するプロパティ](#)」に記載されている設定に **Properties** インスタンスを指定して作成されます。その後、Spark コンテキストとともに使用して、通常の Spark 操作と使用される **InfinispanRDD** を作成します。

以下に Java および Scala での例を示します。

34.4.3. RDD の作成

RDD の作成 (Java)

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.spark.rdd.InfinispanJavaRDD;
import java.util.Properties;
[...]
```

// Begin by defining a new Spark configuration and creating a Spark context from this.

```
SparkConf conf = new SparkConf().setAppName("example-RDD");
JavaSparkContext jsc = new JavaSparkContext(conf);

// Create the Properties instance, containing the JBoss Data Grid node and cache name.
Properties properties = new Properties();
properties.put("infinispan.client.hotrod.server_list", "server:11222");
properties.put("infinispan.rdd.cacheName", "exampleCache");

// Create the RDD
JavaPairRDD<Integer, Book> exampleRDD =
    InfinispanJavaRDD.createInfinispanRDD(jsc, properties);

JavaRDD<Book> booksRDD = exampleRDD.values();
```

RDD の作成 (Scala)

```
import java.util.Properties

import org.apache.spark.{SparkConf, SparkContext}
import org.infinispan.spark.rdd.InfinispanRDD
import org.infinispan.spark._

// Begin by defining a new Spark configuration and creating a Spark
// context from this.
val conf = new SparkConf().setAppName("example-RDD-scala")
val sc = new SparkContext(conf)

// Create the Properties instance, containing the JBoss Data Grid node and
// cache name.
val properties = new Properties
properties.put("infinispan.client.hotrod.server_list", "server:11222")
properties.put("infinispan.rdd.cacheName", "exampleCache")

// Create an RDD from the DataGrid cache
val exampleRDD = new InfinispanRDD[Integer, Book](sc, properties)

val booksRDD = exampleRDD.values
```

34.4.4. RDD のクエリー

RDD が利用可能になると、Spark RDD 操作または Spark の SQL サポートのいずれかを使用してバッキングキャッシュを取得できます。上記の例を拡張して author (著者) ごとにエントリーをカウントすると、結果となる RDD と SQL クエリーは次のようになります。

RDD のクエリー (Java)

```
// The following imports should be added to the list from the previous
// example
import org.apache.spark.sql.DataFrame;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SQLContext;
[...]
// Continuing the previous example

// Create a SQLContext, registering the data frame and table
SQLContext sqlContext = new SQLContext(jsc);
DataFrame dataframe = sqlContext.createDataFrame(booksRDD, Book.class);
dataframe.registerTempTable("books");

// Run the Query and collect results
List<Row> rows = sqlContext.sql("SELECT author, count(*) as a from books
WHERE author != 'N/A' GROUP BY author ORDER BY a desc").collectAsList();
```

RDD のクエリー (Scala)

```
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext
[...]
// Create a SQLContext, register a data frame and table
val sqlContext = new SQLContext(sc)
```

```
val dataframe = sqlContext.createDataFrame(booksRDD, classOf[Book])
dataframe.registerTempTable("books")

// Run the Query and collect the results
val rows = sqlContext.sql("SELECT author, count(*) as a from books WHERE
author != 'N/A' GROUP BY author ORDER BY a desc").collect()
```

34.4.5. RDD のキャッシュへの書き込み

静的 `InfinispanJavaRDD.write()` メソッドを使用すると、あらゆるキーバリュースペースの RDD を Data Grid キャッシュに書き込みできます。これにより、RDD の内容がキャッシュにコピーされます。

RDD の書き込み (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.spark.domain.Address;
import org.infinispan.spark.domain.Person;
import org.infinispan.spark.rdd.InfinispanJavaRDD;
import scala.Tuple2;
import java.util.List;
import java.util.Properties;
[...]
```

// Define the location of the JBoss Data Grid node

```
Properties properties = new Properties();
properties.put("infinispan.client.hotrod.server_list", "localhost:11222");
properties.put("infinispan.rdd.cacheName", "exampleCache");

// Create the JavaSparkContext
SparkConf conf = new SparkConf().setAppName("write-example-RDD");
JavaSparkContext jsc = new JavaSparkContext(conf);

// Defining two entries to be stored in a RDD
// Each Book will contain the title, author, and publicationYear
Book bookOne = new Book("Linux Bible", "Negus, Chris", "2015");
Book bookTwo = new Book("Java 8 in Action", "Urma, Raoul-Gabriel",
"2014");

List

```

RDD の書き込み (Scala)

```
import java.util.Properties
import org.infinispan.spark._
```

```
import org.infinispan.spark.rdd.InfinispanRDD
[...]
// Define the location of the JBoss Data Grid node
val properties = new Properties
properties.put("infinispan.client.hotrod.server_list", "localhost:11222")
properties.put("infinispan.rdd.cacheName", "exampleCache")

// Create the SparkContext
val conf = new SparkConf().setAppName("write-example-RDD-scala")
val sc = new SparkContext(conf)

// Create an RDD of Books
val bookOne = new Book("Linux Bible", "Negus, Chris", "2015")
val bookTwo = new Book("Java 8 in Action", "Urma, Raoul-Gabriel", "2014")

val sampleBookRDD = sc.parallelize(Seq(bookOne, bookTwo))
val pairsRDD = sampleBookRDD.zipWithIndex().map(_._2.swap)

// Write the Key/Value RDD to the Data Grid
pairsRDD.writeToInfinispan(properties)
```

34.4.5.1. DStreams の作成および使用

DStream は継続するデータのストリームを表し、継続的な一連の RDD によって内部で表されます。各 RDD には特定の時間間隔からのデータが含まれます。

DStream の作成には、以下の例のように **StreamingContext** が **StorageLevel** および JBoss Data Grid RDD 設定とともに渡されます。

DStream の作成 (Scala)

```
import org.infinispan.spark.stream._
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.storage.StorageLevel
import java.util.Properties

// Spark context
val sc = ...
// java.util.Properties with Infinispan RDD configuration
val props = ...
val ssc = new StreamingContext(sc, Seconds(1))

val stream = new InfinispanInputDStream[String, Book](ssc,
StorageLevel.MEMORY_ONLY, props)
```

InfinispanInputDStream は多くの Spark の DStream 操作を使用して変換でき、**StreamingContext** で「start」を呼び出した後に処理が発生します。たとえば、最後の 30 秒でキャッシュに挿入された本の数を 10 秒ごとに表示する場合は以下のようになります。

DStream の処理 (Scala)

```
import org.infinispan.spark.stream._

val stream = ... // From previous sample
```



```
// Filter only created entries
val createdBooksRDD = stream.filter { case (_, _, t) => t ==
Type.CLIENT_CACHE_ENTRY_CREATED }

// Reduce last 30 seconds of data, every 10 seconds
val windowedRDD: DStream[Long] = createdBooksRDD.count().reduceByWindow(_
+ _, Seconds(30), Seconds(10))

// Prints the results, counting the number of occurrences in each individual
RDD
windowedRDD.foreachRDD { rdd => println(rdd.reduce(_ + _)) }

// Start the processing
ssc.start()
ssc.awaitTermination()
```

DStreams での JBoss Data Grid への書き込み

キーバリュー型のすべての DStream は、Java の場合は

InfinispanJavaDStream.writeToInfinispan() Java メソッド経由で JBoss Data Grid へ書き込みでき、Scala の場合は暗黙的な **writeToInfinispan(properties)** メソッドを直接 DStream インスタンスで使用して書き込みできます。両方のメソッドは JBoss Data Grid RDD 設定を入力として取り、DStream 内に含まれる各 RDD を書き込みます。

34.4.6. Spark での Infinispan Query DSL の使用

Infinispan Query DSL は InfinispanRDD のフィルターとして使用でき、RDD レベルではなくソースでデータを事前にフィルターできます。



重要

クエリーする DSL が適切に動作するには、キャッシュのデータを protobuf でエンコードする必要があります。protobuf エンコーディングの使用手順は「[Protobuf エンコーディング](#)」を参照してください。

著者の名前に **Doe** 含まれる本のリストを取得する以下の例を見てください。

34.4.7. クエリーによるフィルリング

クエリーによるフィルリング (Scala)

```
import org.infinispan.client.hotrod.impl.query.RemoteQuery
import org.infinispan.client.hotrod.{RemoteCacheManager, Search}
import org.infinispan.spark.domain._
[...]
```

```
val query =
Search.getQueryFactory(remoteCacheManager.getCache(getCacheName))
  .from(classOf[Book])
  .having("author").like("Doe")
  .toBuilder[RemoteQuery].build()

val rdd = createInfinispanRDD[Int, Book]
  .filterByQuery[Book]](query, classOf[Book])
```

射影も完全サポートされます。たとえば、上記の例を調整して著書名と出版年のみを取得し、出版年のフィールドでソートすることができます。

34.4.8. 射影を使用したフィルタリング

射影を使用したフィルタリング (Scala)

```
import org.infinispan.client.hotrod.impl.query.RemoteQuery
import org.infinispan.client.hotrod.{RemoteCacheManager, Search}
import org.infinispan.spark.domain._
[...]
```

```
val query =
  Search.getQueryFactory(remoteCacheManager.getCache(getCacheName))
    .select("title", "publicationYear")
    .from(classOf[Book])
    .having("author").like("Doe")
    .groupBy("publicationYear")
    .toBuilder[RemoteQuery].build()

val rdd = createInfinispanRDD[Int, Book]
  .filterByQuery[Array[AnyRef]](query, classOf[Book])
```

さらに、フィルターがすでに JBoss Data Grid サーバーにデプロイされている場合は、以下のように名前で参照することもできます。

34.4.9. デプロイされたフィルターを使用したフィルタリング

デプロイされたフィルターを使用したフィルタリング (Scala)

```
val rdd = InfinispanRDD[String, Book] = ....
// "my-filter-factory" filter and converts Book to a String, and has two
// parameters
val filteredRDD = rdd.filterByCustom[String]("my-filter-factory",
  "param1", "param2")
```

34.5. SPARK 2 のコード例

34.5.1. Spark 2 のコード例

Apache Spark 2 のコネクタは 1.6 コネクタとは異なる設定メカニズムを使用し、バージョン 2 コネクタは 1.6 がサポートしない機能を一部サポートするため、各バージョンのコード例は独自のセクションに分割されています。以下のコード例は、バージョン 2 の Spark コネクタと動作します。[Spark 1.6 のコード例はこちら](#)を参照してください。

34.5.2. RDD の作成および使用

Apache Spark 2 では、RDD (Resilient Distributed Datasets) は「[バージョン 2 コネクタを設定するメソッド](#)」に記載されている設定に **ConnectorConfiguration** インスタンスを指定して作成されます。

以下に Java および Scala での例を示します。

34.5.3. RDD の作成

RDD の作成 (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration config = new ConnectorConfiguration()
    .setCacheName("exampleCache").setServerList("server:11222");

JavaPairRDD<String, MyEntity> infinispanRDD =
    InfinispanJavaRDD.createInfinispanRDD(jsc, config);

JavaRDD<MyEntity> entitiesRDD = infinispanRDD.values();
```

RDD の作成 (Scala)

```
import org.apache.spark.SparkContext
import org.infinispan.spark.config.ConnectorConfiguration
import org.infinispan.spark.rdd.InfinispanRDD

val sc: SparkContext = new SparkContext()

val config = new ConnectorConfiguration().setCacheName("my-
cache").setServerList("10.9.0.8:11222")

val infinispanRDD = new InfinispanRDD[String, MyEntity](sc, config)

val entitiesRDD = infinispanRDD.values
```

34.5.4. RDD のクエリー

34.5.4.1. SparkSQL クエリー

SparkSQL クエリーの使用 (Java)

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration conf = new ConnectorConfiguration();
```

```
// Obtain the values from an InfinispanRDD
JavaPairRDD<Long, MyEntity> infinispanRDD =
InfinispanJavaRDD.createInfinispanRDD(jsc, conf);

JavaRDD<MyEntity> valuesRDD = infinispanRDD.values();

// Create a DataFrame from a SparkSession
SparkSession sparkSession = SparkSession.builder().config(new
SparkConf().setMaster("masterHost")).getOrCreate();
Dataset<Row> dataFrame = sparkSession.createDataFrame(valuesRDD,
MyEntity.class);

// Create a view
dataFrame.createOrReplaceTempView("myEntities");

// Create and run the Query
Dataset<Row> rows = sparkSession.sql("SELECT field1, count(*) as c from
myEntities WHERE field1 != 'N/A' GROUP BY field1 ORDER BY c desc");
```

SparkSQL クエリーの使用 (Scala)

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.{SparkConf, SparkContext}
import org.infinispan.spark.config.ConnectorConfiguration
import org.infinispan.spark.rdd._

val sc: SparkContext = // Initialize your SparkContext here

val config = new
ConnectorConfiguration().setServerList("myserver1:port,myserver2:port")

// Obtain the values from an InfinispanRDD
val infinispanRDD = new InfinispanRDD[Long, MyEntity](sc, config)
val valuesRDD = infinispanRDD.values

// Create a DataFrame from a SparkSession
val sparkSession = SparkSession.builder().config(new
SparkConf().setMaster("masterHost")).getOrCreate()
val dataFrame = sparkSession.createDataFrame(valuesRDD, classOf[MyEntity])

// Create a view
dataFrame.createOrReplaceTempView("myEntities")

// Create and run the Query, collect and print results
sparkSession.sql("SELECT field1, count(*) as c from myEntities WHERE
field1 != 'N/A' GROUP BY field1 ORDER BY c desc")
.collect().take(20).foreach(println)
```

34.5.5. RDD のキャッシュへの書き込み

静的 `InfinispanJavaRDD.write()` メソッドを使用すると、あらゆるキーバリュースペースの RDD を Data Grid キャッシュに書き込みできます。これにより、RDD の内容がキャッシュにコピーされます。

RDD の書き込み (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration connectorConfiguration = new
ConnectorConfiguration();

List<Integer> numbers = IntStream.rangeClosed(1,
1000).boxed().collect(Collectors.toList());
JavaPairRDD<Integer, Long> numbersRDD =
jsc.parallelize(numbers).zipWithIndex();

InfinispanJavaRDD.write(numbersRDD, connectorConfiguration);
```

RDD の書き込み (Scala)

```
import org.apache.spark.SparkContext
import org.infinispan.spark._
import org.infinispan.spark.config.ConnectorConfiguration

val config: ConnectorConfiguration = // Initialize your
ConnectorConfiguration here
val sc: SparkContext = // Initialize your SparkContext here

val simpleRdd = sc.parallelize(1 to 1000).zipWithIndex()
simpleRdd.writeToInfinispan(config)
```

34.5.6. DStreams の作成

DStream は継続するデータのストリームを表し、継続的な一連の RDD によって内部で表されます。各 RDD には特定の時間間隔からのデータが含まれます。

DStream の作成には、以下の例のように **StreamingContext** が **StorageLevel** および JBoss Data Grid RDD 設定とともに渡されます。

DStream の作成 (Java)

```
import org.apache.spark.SparkConf;
import org.apache.spark.streaming.Seconds;
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.stream.InfinispanJavaDStream;
import static org.apache.spark.storage.StorageLevel.MEMORY_ONLY;

SparkConf conf = new SparkConf().setAppName("my-stream-app");

ConnectorConfiguration configuration = new ConnectorConfiguration();
```

```

JavaStreamingContext jsc = new JavaStreamingContext(conf,
Seconds.apply(1));

InfinispanJavaDStream.createInfinispanInputDStream(jsc, MEMORY_ONLY(),
configuration);

```

DStream の作成 (Scala)

```

import org.apache.spark.SparkContext
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.infinispan.spark.config.ConnectorConfiguration
import org.infinispan.spark.stream._

val sc = new SparkContext()
val config = new ConnectorConfiguration()
val ssc = new StreamingContext(sc, Seconds(1))
val stream = new InfinispanInputDStream[String, MyEntity](ssc,
StorageLevel.MEMORY_ONLY, config)

```

DStreams での JBoss Data Grid への書き込み

キーバリュ型すべての DStream は、Java の場合は

InfinispanJavaDStream.writeToInfinispan() Java メソッド経由で JBoss Data Grid へ書き込みでき、Scala の場合は暗黙的な **writeToInfinispan(properties)** メソッドを直接 DStream インスタンスで使用して書き込みできます。両方のメソッドは JBoss Data Grid RDD 設定を入力として取り、DStream 内に含まれる各 RDD を書き込みます。

34.5.7. Apache Spark Dataset API の使用

JBoss Data Grid は、RDD (Resilient Distributed Dataset) プログラミングインターフェースの他に、Apache Spark Dataset API が含まれています。この API には、**rdd.filterByQuery** と似た述語のプッシュダウンのサポートが含まれます。

Dataset API の例 (Java)

```

import org.apache.spark.SparkConf;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;
import org.infinispan.spark.config.ConnectorConfiguration;

import java.util.List;

// Configure the connector using the ConnectorConfiguration: register
// entities annotated with Protobuf,
// and turn on automatic registration of schemas
ConnectorConfiguration connectorConfig = new ConnectorConfiguration()
    .setServerList("server1:11222,server2:11222")
    .addProtoAnnotatedClass(User.class)
    .setAutoRegisterProto();

// Create the SparkSession
SparkSession sparkSession = SparkSession.builder().config(new
SparkConf().setMaster("masterHost")).getOrCreate();

```

```
// Load the "infinispan" datasource into a DataFame, using the infinispan
config
Dataset<Row> df =
sparkSession.read().format("infinispan").options(connectorConfig.toStrings
Map()).load();

// From here it's possible to query using the DatasetSample API...
List<Row> rows =
df.filter(df.col("age").gt(30)).filter(df.col("age").lt(40)).collectAsList
();

// ... or execute SQL queries
df.createOrReplaceTempView("user");
String query = "SELECT first(r.name) as name, first(r.age) as age FROM
user u GROUP BY r.age";
List<Row> results = sparkSession.sql(query).collectAsList();
```

Dataset API の例 (Scala)

```
import org.apache.spark._
import org.apache.spark.sql._
import org.infinispan.protostream.annotations.{ProtoField, ProtoMessage}
import org.infinispan.spark.config.ConnectorConfiguration

import scala.annotation.meta.beanGetter
import scala.beans.BeanProperty

// Entities can be annotated in order to automatically generate protobuf
schemas.
// Also, they should be valid java beans. From Scala this can be
achieved as:

@ProtoMessage(name = "user")
class User(@ProtoField@beanGetter)(number = 1, required = true)
@BeanProperty var name: String,
              @ProtoField@beanGetter)(number = 2, required = true)
@BeanProperty var age: Int) {
  def this() = {
    this(name = "", age = -1)
  }
}

// Configure the connector using the ConnectorConfiguration: register
entities annotated with Protobuf,
// and turn on automatic registration of schemas
val infinispanConfig: ConnectorConfiguration = new
ConnectorConfiguration()
  .setServerList("server1:11222,server2:11222")
  .addProtoAnnotatedClass(classOf[User])
  .setAutoRegisterProto()

// Create the SparkSession
val sparkSession = SparkSession.builder().config(new
SparkConf().setMaster("masterHost")).getOrCreate()
```



```
// Load the "infinispan" datasource into a DataFame, using the infinispan
config
val df =
sparkSession.read.format("infinispan").options(infinispanConfig.toStringsM
ap).load()

// From here it's possible to query using the DatasetSample API...
val rows: Array[Row] =
df.filter(df("age").gt(30)).filter(df("age").lt(40)).collect()

// ... or execute SQL queries
df.createOrReplaceTempView("user")
val query = "SELECT first(r.name) as name, first(r.age) as age FROM user u
GROUP BY r.age"
val rowsFromSQL: Array[Row] = sparkSession.sql(query).collect()
```

34.5.8. Spark での Infinispan Query DSL の使用

Infinispan Query DSL は InfinispanRDD のフィルターとして使用でき、RDD レベルではなくソースでデータを事前にフィルターできます。



重要

クエリーする DSL が適切に動作するには、キャッシュのデータを protobuf でエンコードする必要があります。protobuf エンコーディングの使用手順は「[Protobuf エンコーディング](#)」を参照してください。

34.5.9. 事前ビルドされたクエリーオブジェクトを使用したフィルタリング

事前ビルドされたクエリーオブジェクトを使用したフィルタリング (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.Search;
import org.infinispan.query.dsl.Query;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration conf = new ConnectorConfiguration();

InfinispanJavaRDD<String, MyEntity> infinispanRDD =
InfinispanJavaRDD.createInfinispanRDD(jsc, conf);

RemoteCache<String, MyEntity> remoteCache = new
RemoteCacheManager().getCache();

// Assuming MyEntity is already stored in the cache with protobuf
encoding, and has protobuf annotations.
Query query =
```



```
Search.getQueryFactory(remoteCache).from(MyEntity.class).having("field").equal("value").build();
```

```
JavaPairRDD<String, MyEntity> filtered =  
infinispanRDD.filterByQuery(query);
```

事前ビルドされたクエリーオブジェクトを使用したフィルタリング (Scala)

```
import org.infinispan.client.hotrod.{RemoteCache, Search}  
import org.infinispan.spark.rdd.InfinispanRDD  
  
val rdd: InfinispanRDD[String, MyEntity] = // Initialize your InfinispanRDD here  
val cache: RemoteCache[String, MyEntity] = // Initialize your RemoteCache here  
  
// Assuming MyEntity is already stored in the cache with protobuf encoding, and has protobuf annotations.  
val query =  
Search.getQueryFactory(cache).from(classOf[MyEntity]).having("field").equal("value").build()  
  
val filteredRDD = rdd.filterByQuery(query)
```

34.5.10. Ickle クエリーを使用したフィルタリング

Ickle クエリーを使用したフィルタリング (Java)

```
import org.apache.spark.api.java.JavaPairRDD;  
import org.apache.spark.api.java.JavaSparkContext;  
import org.infinispan.spark.config.ConnectorConfiguration;  
import org.infinispan.spark.rdd.InfinispanJavaRDD;  
  
JavaSparkContext jsc = new JavaSparkContext();  
ConnectorConfiguration conf = new ConnectorConfiguration();  
  
InfinispanJavaRDD<String, MyEntity> infinispanRDD =  
InfinispanJavaRDD.createInfinispanRDD(jsc, conf);  
  
JavaPairRDD<String, MyEntity> filtered = infinispanRDD.filterByQuery("From  
myEntity where field = 'value'");
```

Ickle クエリーを使用したフィルタリング (Scala)

```
import org.infinispan.spark.rdd.InfinispanRDD  
  
val rdd: InfinispanRDD[String, MyEntity] = // Initialize your InfinispanRDD here  
val filteredRDD = rdd.filterByQuery("FROM MyEntity e where e.field BETWEEN  
10 AND 20")
```

34.5.11. サーバー上でのフィルタリング

サーバー上でのフィルタリング (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration conf = new ConnectorConfiguration();

InfinispanJavaRDD<String, MyEntity> infinispanRDD =
InfinispanJavaRDD.createInfinispanRDD(jsc, conf);

JavaPairRDD<String, MyEntity> filtered = infinispanRDD.filterByCustom("my-
filter", "param1", "param2");
```

サーバー上でのフィルタリング (Scala)

```
import org.infinispan.spark.rdd.InfinispanRDD

val rdd: InfinispanRDD[String, MyEntity] = // Initialize your InfinispanRDD
here
// "my-filter-factory" filter and converts MyEntity to a Double, and has
two parameters
val filteredRDD = rdd.filterByCustom[Double]("my-filter-factory",
"param1", "param2")
```

34.6. SPARK のパフォーマンスに関する注意点

Data Grid の Spark コネクタは、デフォルトでは Data Grid ノードごとに 2 つのパーティションを作成し、各パーティションはそのノードのデータのサブセットを指定します。

これらのパーティションは Spark ワーカーへ送信され、並列処理されます。Spark ワーカーの数が Data Grid ノードよりも少ない場合、各ワーカーにはタスクを並列実行できる最大限度があるため、遅延が発生することがあります。そのため、Spark ワーカーの数を最低でも Data Grid ノードの数と同じにしてこの並行処理を活用することが推奨されます。

また、Spark ワーカーが Data Grid ノードと同じノードに位置する場合、各ワーカーがローカルの Data Grid ノードにあるデータのみを処理するようコネクタによってタスクが分散されます。

第35章 APACHE HADOOP との統合

35.1. APACHE HADOOP との統合

JBoss Data Grid コネクタは、JBoss Data Grid を Hadoop 対応のデータソースにします。この統合は、Hadoop の **InputFormat** および **OutputFormat** の実装を提供して実現され、アプリケーションが最適な場所の JBoss Data Grid サーバーに対してデータを読み書きできるようにします。JBoss Data Grid の **InputFormat** および **OutputFormat** 実装は、従来の Hadoop Map/Reduce ジョブの実行を可能にしますが、Hadoop の **InputFormat** データソースをサポートするツールやユーティリティーと使用することもできます。

35.2. HADOOP の依存関係

Hadoop の形式の JBoss Data Grid 実装は以下の Maven 依存関係にあります。

```
<dependency>
  <groupId>org.infinispan.hadoop</groupId>
  <artifactId>infinispan-hadoop-core</artifactId>
  <version>0.2.2.Final-redhat-1</version>
</dependency>
```

35.3. サポートされる HADOOP 設定パラメーター

以下のパラメーターがサポートされます。

表35.1 サポートされる Hadoop 設定パラメーター

パラメーター名	説明	デフォルト値
hadoop.ispn.input.filter.factory	読み取りの前にデータを事前にフィルターするためにサーバーにデプロイされるフィルターファクトリーの名前。	null (有効なフィルタリングなし)
hadoop.ispn.input.cache.name	データが読み取られるキャッシュの名前。	___defaultcache
hadoop.ispn.input.remote.cache.servers	以下の形式を使用する、入力キャッシュのサーバーのリスト。 host1:port;host2:port2	localhost:11222
hadoop.ispn.output.cache.name	データが書き込まれるキャッシュの名前。	default

パラメーター名	説明	デフォルト値
hadoop.ispn.output.remote.cache.servers	以下の形式を使用する、出力キャッシュのサーバーのリスト。 host1:port;host2:port2	null (出力キャッシュなし)
hadoop.ispn.input.read.batch	キャッシュから読み取りするときのバッチサイズ。	5000
hadoop.ispn.output.write.batch	キャッシュに書き込みするときのバッチサイズ。	500
hadoop.ispn.input.converter	キャッシュから読み取りした後に適用される、 org.infinispan.hadoop.p.KeyValueConverter の実装とのクラス名。	null (有効な変換なし)。
hadoop.ispn.output.converter	書き込みの前に適用される、 org.infinispan.hadoop.K eyValueConverter の実装とのクラス名。	null (有効な変換なし)。

35.4. HADOOP コネクターの使用

InfinispanInputFormat および InfinispanOutputFormat

Hadoop では、**InputFormat** インターフェースは、特定のデータソースがパーティション化される方法と、各パーティションからのデータの読み取り方法を示します。**OutputFormat** インターフェースはデータの書き込み方法を指定します。

InputFormat インターフェースには、重要なメソッドが2つ定義されています。

1. **getSplits** メソッドは、データの特定セクションに関する情報が含まれる1つ以上の **InputSplit** インスタンスを返すデータパーティショナーを定義します。

```
List<InputSplit> getSplits(JobContext context);
```

2. **InputSplit** を使用すると、結果となるデータセットで反復処理を行うために使用される **RecordReader** を取得できます。

```
RecordReader<K,V> createRecordReader(InputSplit  
split, TaskAttemptContext context);
```

これらの2つの操作は、複数のノード全体でデータの並列処理を可能にするため、大型のデータセットで Hadoop のスループットが大きくなります。

JBoss Data Grid では、パーティションはセグメントの所有権を基に生成されるため、各パーティションは特定のサーバーのセグメントのセットになります。デフォルトでは、パーティションの数はクラス

ターのサーバーの数と同じになり、各パーティションにはその特定のサーバーに関連するすべてのセグメントが含まれます。

JBoss Data Grid における Hadoop Map Reduce ジョブの実行

以下に JBoss Data Grid クラスターをターゲットとする Map Reduce ジョブの設定例を示します。

```
import org.infinispan.hadoop.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapreduce.Job;

[...]
Configuration configuration = new Configuration();
configuration.set(InfinispanConfiguration.INPUT_REMOTE_CACHE_SERVER_LIST,
"localhost:11222");
configuration.set(InfinispanConfiguration.INPUT_REMOTE_CACHE_NAME, "map-
reduce-in");
configuration.set(InfinispanConfiguration.OUTPUT_REMOTE_CACHE_SERVER_LIST,
"localhost:11222");
configuration.set(InfinispanConfiguration.OUTPUT_REMOTE_CACHE_NAME, "map-
reduce-out");

Job job= Job.getInstance(configuration, "Infinispan Integration");
[...]
```

JBoss Data Grid をターゲットとするには、ジョブを **InfinispanInputFormat** および **InfinispanOutputFormat** クラスと設定する必要があります。

```
[...]
// Define the Map and Reduce classes
job.setMapperClass(MapClass.class);
job.setReducerClass(ReduceClass.class);

// Define the JBoss Data Grid implementations
job.setInputFormatClass(InfinispanInputFormat.class);
job.setOutputFormatClass(InfinispanOutputFormat.class);
[...]
```

第36章 EAP との統合

36.1. EAP との統合

EAP には Infinispan モジュールが含まれていますが、内部の EAP で使用することを目的とし、JBoss Data Grid との使用はサポートされていません。EAP 内で JDG を使用するには、JDG が提供する EAP モジュールを使用します。これらのモジュールを使用すると、スロットが異なるため、EAP の内部モジュールとの競合を防ぐことができます。さらに、JDG をデプロイメント内 (WAR、EAR など) にパッケージ化せずにアプリケーションをデプロイできるため、サイズが最小限になります。

36.2. EAP モジュールのインストール

EAP のモジュールは、Red Hat カスタマーポータルからダウンロードできます。

手順: EAP モジュールのダウンロード

1. <https://access.redhat.com> のカスタマーポータルにログインします。
2. ページの上部にある **ダウンロード** ボタンをクリックします。
3. **製品のダウンロード** ページで **Red Hat JBoss Data Grid** をクリックします。
4. **Version:** ドロップダウンメニューで適切な JBoss Data Grid のバージョンを選択します。
5. **Red Hat JBoss Data Grid 7.2 Library Module for JBoss EAP** を見つけ、対応する **Download** リンクをクリックします。

zip ファイルを **EAP_HOME/modules** に展開する必要があります。ファイルが適切に展開されると、Infinispan コアモジュールの場所は **EAP_HOME/modules/org/infinispan/core** になります。

36.3. EAP の依存関係

Maven を使用してモジュールを設定するには、JDG の依存関係を**提供済み**としてマークし、アーティファクトアーカイバーを設定して、以下の **pom.xml** ファイルを使用して適切な **MANIFEST.MF** で WAR ファイルを生成します。

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-core</artifactId>
    <version>${infinispan.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-cachestore-jdbc</artifactId>
    <version>${infinispan.version}</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
```

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <archive>
      <manifestEntries>
        <Dependencies>org.infinispan.core:jdg-7.2 services,
org.infinispan.cachestore.jdbc:jdg-7.2 services</Dependencies>
      </manifestEntries>
    </archive>
  </configuration>
</plugin>
</plugins>
</build>

```

36.4. 特定の JDG コンポーネントの依存関係

以下は、JDG の特定の機能を有効にする **MANIFEST.MF** 設定ファイルの例になります。

36.4.1. コア依存関係

JDG のコア依存関係のみをアプリケーションに公開するには、次のマニフェストを追加します。

MANIFEST.MF

```

Manifest-Version: 1.0
Dependencies: org.infinispan:jdg-7.2 services

```

36.4.2. リモート/Hot Rod 依存関係

リモートクエリー実行のために含まれる Hot Rod を介してリモート JDG サーバーに接続するには、**org.infinispan.remote** モジュールを使用します。このモジュールは、必要な依存関係をすべて自動的に公開します。

MANIFEST.MF

```

Manifest-Version: 1.0
Dependencies: org.infinispan.remote:jdg-7.2 services

```

36.4.3. 埋め込みクエリーの依存関係

Infinispan Query DSL、Lucene、および Hibernate Search Queries を含む埋め込みクエリーの場合、以下をマニフェストに追加します。

MANIFEST.MF

```

Manifest-Version: 1.0
Dependencies: org.infinispan:jdg-7.2 services, org.infinispan.query:jdg-7.2 services

```

36.4.4. Lucene ディレクトリーの依存関係

org.apache.lucene.store.Directory を使用して JDG を Lucene のディレクトリーとして使用する場合はクエリーモジュールは必要なく、以下で対応できます。

MANIFEST.MF

Manifest-Version: 1.0

Dependencies: org.infinispan.lucene-directory:jdg-7.2 services

36.4.5. Hibernate Search ディレクトリープロバイダーの依存関係

JDG の Hibernate Search ディレクトリープロバイダーも JBoss EAP zip ファイルの JBoss Data Grid 7.2 ライブラリーモジュール内に含まれています。Hibernate Search モジュールにはすでに任意の依存関係があるため、マニフェストファイルにエントリーを追加する必要はありません。どの JDG モジュール zip を使用するかを決める場合、どの Hibernate Search が使用されているかをチェックしてから決めてください。

36.4.6. EAP の内部 Hibernate Search モジュールの使用

EAP 7.1 の Hibernate Search モジュールにはバージョン 5.5.x があり、スロットが **for-hibernatesearch-5.5** の **org.infinispan.hibernate-search.directory-provider** モジュールへの任意の依存関係があります。この依存関係は、Infinispan モジュールが [インストール](#) されると利用可能になります。

36.4.7. その他の Hibernate Search モジュールとの用途

JDG と配布されるモジュール **org.hibernate.search:jdg-7.2** は、Infinispan Query のみと併用し (キャッシュからデータをクエリー)、Hibernate ORM アプリケーションによって使用されないようにします。Hibernate Search を EAP に存在する他のバージョンと使用する場合は、[Hibernate Search のドキュメント](#)を参照してください。

org.infinispan.hibernate-search.directory-provider に選択した Hibernate Search のオプションスロットが JBoss Data Grid と配布されるものと一致するようにしてください。

36.5. EAP モジュールの使用

アプリケーションは、ライブラリー (埋め込み) モードまたは EAP サブシステムモードにて EAP 内で JDG を使用できます。

36.5.1. ライブラリーモード

ライブラリーモードにて EAP 内で JDG を使用する場合、すべての **CacheManager** およびキャッシュインスタンスはアプリケーションロジックで作成されます。よって、**EmbeddedCacheManager** のライフサイクルはアプリケーションのライフサイクルと密に結合しているため、アプリケーションが破棄されると、そのアプリケーションによって作成されたマネージャーインスタンスが破棄されます。

36.5.2. EAP サブシステムモード

JDG が EAP のサブシステムである EAP サブシステムモードでは、EAP の **domain/configuration/domain.xml** 設定の一部としてランタイム前にキャッシュコンテナーおよびキャッシュが作成されるようにすることが可能です。これにより、キャッシュインスタンスを複数のアプリケーション全体で共有することができ、基盤のキャッシュコンテナーのライフサイクルはデプロイされたアプリケーションとは無関係になります。

36.6. EAP サブシステムモードの設定

EAP サブシステムモードを有効にするには、以下を **domain/configuration/domain.xml** の EAP 設定に追加します。



注記

ローカルキャッシュインスタンスには最初の 2 つの手順のみが必要になります。

1. Infinispan 拡張を **<extensions>** セクションに追加します。

```
<extensions>
  <extension module="org.infinispan.extension:jdg-7.2"/>
  <extension module="org.jgroups.extension:jdg-7.2"/>

  <!--Other EAP extensions-->
</extensions>
```

2. Infinispan を必要とするサーバープロファイルに、必要なすべてのコンテナーおよびキャッシュとともに Infinispan サブシステムを設定します。



注記

必ずモジュール属性が定義されているようにしてください。そうでないと、正しい Infinispan クラスがロードされません。

```
<subsystem xmlns="urn:infinispan:server:core:8.5">
  <cache-container name="jdg-container" default-cache="default"
module="org.infinispan.extension:jdg-7.2">
    <transport channel="jdg-cluster"/>
    <global-state/>
    <distributed-cache-configuration name="default"/>
  <distributed-cache name="default"/>
  </cache-container>
</subsystem>
```

3. JGroups サブシステムが必要な EAP インターフェースおよびソケットバインディングを定義します。

インターフェースの定義:

```
<interfaces>
  <interface name="jdg">
    <inet-address value="{jdg.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```

ソケットバインディングの定義:

```
<socket-binding-group name="full-sockets" default-interface="public">
  <socket-binding name="jdg-jgroups-udp" interface="jdg" port="55200"
multicast-address="{jdg.default.multicast.address:230.0.0.4}" multicast-
```

```
port="45688"/>
<socket-binding name="jdg-jgroups-udp-fd" interface="jdg" port="54200"/>
</socket-binding-group>
```

EAP インターフェースとソケットバインディングの詳細は、JBoss EAP 『設定ガイド』の「[ネットワークおよびポート設定](#)」を参照してください。

4. 指定されたすべてのプロトコルのモデル属性が定義されるよう、JGroups トランスポートを定義します。

```
<subsystem xmlns="urn:infinispan:server:jgroups:8.0">
  <channels>
    <channel name="jdg-cluster" stack="udp"/>
  </channels>
  <stacks>
    <stack name="udp">
      <transport type="UDP" socket-binding="jdg-jgroups-udp"
module="org.jgroups:jdg-7.2"/>
      <protocol type="PING" module="org.jgroups:jdg-7.2"/>
      <protocol type="MERGE3" module="org.jgroups:jdg-7.2"/>
      <protocol type="FD SOCK" socket-binding="jdg-jgroups-udp-fd"
module="org.jgroups:jdg-7.2"/>
      <protocol type="FD_ALL" module="org.jgroups:jdg-7.2"/>
      <protocol type="VERIFY_SUSPECT" module="org.jgroups:jdg-7.2"/>
      <protocol type="pbcast.NAKACK2" module="org.jgroups:jdg-7.2"/>
      <protocol type="UNICAST3" module="org.jgroups:jdg-7.2"/>
      <protocol type="pbcast.STABLE" module="org.jgroups:jdg-7.2"/>
      <protocol type="pbcast.GMS" module="org.jgroups:jdg-7.2"/>
      <protocol type="UFC" module="org.jgroups:jdg-7.2"/>
      <protocol type="MFC" module="org.jgroups:jdg-7.2"/>
      <protocol type="FRAG3" module="org.jgroups:jdg-7.2"/>
    </stack>
  </stacks>
</subsystem>
```

サーバーモードの設定には、コマンドラインスクリプトも使用できます。

```
# adding the necessary modules to the EAP configuration
# remember to add the datagrid library modules of JDG 7.2 before !
/extension=org.infinispan.extension\:jdg-7.2:add
/extension=org.jgroups.extension\:jdg-7.2:add

batch
/profile=full/subsystem=datagrid-infinispan:add
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-
container:add(module="org.infinispan.extension:jdg-7.2", default-
cache="default")
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-
container/transport=TRANSPORT:add(channel=jdg-cluster)
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-
container/global-state=GLOBAL_STATE:add

# add an interface for JDG cluster communication, can be skipped if the
same as JGroups or public is used
/interface=jdg:add(inet-address="$ {jdg.bind.address:127.0.0.1}")
```

```
# add the port numbers for JDG JGroups
/socket-binding-group=full-sockets/socket-binding=jdg-jgroups-
udp:add(interface="jdg", port=55200, multicast-
address="${jdg.default.multicast.address:230.0.0.4}", multicast-
port="45688"
/socket-binding-group=full-sockets/socket-binding=jdg-jgroups-udp-
fd:add(port=54200, interface="jdg")

# adding the datagrid JGroups subsystem with UDP stack
/profile=full/subsystem=datagrid-jgroups:add(default-channel=jdg-cluster)
/profile=full/subsystem=datagrid-jgroups/channel=jdg-
cluster:add(stack=udp)
/profile=full/subsystem=datagrid-jgroups/stack=udp:add( )
/profile=full/subsystem=datagrid-
jgroups/stack=udp/transport=UDP:add(socket-binding=jdg-jgroups-udp,
module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=PING:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=MERGE3:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=FD_SOCKET:add(module="org.jgroups:jdg-7.2",
socket-binding=jdg-jgroups-udp-fd)
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=FD_ALL:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=VERIFY_SUSPECT:add(module="org.jgroups:jdg-
7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=pbcst.NAKACK2:add(module="org.jgroups:jdg-
7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=UNICAST3:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=pbcst.STABLE:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=pbcst.GMS:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=UFC:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=MFC:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-
jgroups/stack=udp/protocol=FRAG3:add(module="org.jgroups:jdg-7.2")

# add a configuration as this is needed if the CLI is used to add a cache
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-
container/configurations=CONFIGURATIONS:add
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-
container/configurations=CONFIGURATIONS/distributed-cache-
configuration=default:add

run-batch
```

キャッシュを追加するには、次のコマンドを使用します。

```
# add a simple cache
/profile=full/subsystem=datagrid-infinispan/cache-container=jdgc-
container/distributed-cache=default:add(configuration=default)
```

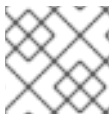
36.7. コンテナおよびキャッシュへのリモートアクセス

コンテナがサーバーの設定に定義されたら、**@Resource** JNDI ルックアップを使用して **CacheContainer** または **Cache** のインスタンスをアプリケーションにインジェクトできます。コンテナは文字列 `java:jboss/datagrid-infinispan/container/<container_name>` を使用してアクセスされ、キャッシュも同様に `java:jboss/datagrid-infinispan/container/<container_name>/cache/<cache_name>` を使用してアクセスされます。

以下の例は、「jdgc-container」と呼ばれる **CacheContainer** と分散キャッシュ「default」をアプリケーションにインジェクトする方法を示しています。

```
public class ExampleApplication {
    @Resource(lookup = "java:jboss/datagrid-infinispan/container/jdgc-
container")
    CacheContainer container;

    @Resource(lookup = "java:jboss/datagrid-infinispan/container/jdgc-
container/cache/default")
    Cache cache;
}
```



注記

このコード例では、**jdgc-7.2** モジュールに依存関係があります。

36.8. EAP サブシステムモードでの EAP および JDG のトラブルシューティング

36.8.1. ロギングの有効化

org.jboss.modules でトレースを有効にすると、**LinkageError** や **ClassNotFoundException** などの問題をデバッグするのに便利です。起動時にトレースロギングを有効にするには、EAP CLI を使用します。

```
bin/jboss-cli.sh -c '/subsystem=logging/logger=org.jboss.modules:add'
bin/jboss-cli.sh -c '/subsystem=logging/logger=org.jboss.modules:write-
attribute(name=level,value=TRACE)'
```

36.8.2. 依存関係ツリーの出力

以下のコマンドを使用すると、特定モジュールのすべての依存関係を出力できます。たとえば、モジュール **org.infinispan:jdgc-7.2** のツリーを取得するには、以下を **EAP_HOME** から実行します。

```
java -jar jboss-modules.jar -deptree -mp modules/ "org.infinispan:jdgc-7.2"
```

第37章 サーバーヒンティングを用いた高可用性

37.1. サーバーヒンティングを用いた高可用性

Red Hat JBoss Data Grid では、サーバーヒンティングによってデータのバックアップコピーが元データと同じ物理サーバー、ラック、またはデータセンター上に保存されないようにします。トータルレプリケーションはすべてのサーバー、ラックおよびデータセンター上で完全なレプリカの作成を強制するため、サーバーヒンティングはトータルレプリケーションには適用されません。

複数のノードにまたがるデータ分散は一貫したハッシュメカニズムによって制御されます。JBoss Data Grid は、一貫したハッシュアルゴリズムを指定するためのプラグ可能なポリシーを提供します。詳細は「[ConsistentHashFactories](#)」を参照してください。

トランスポート設定で **machineId**、**rackId**、または **siteId** を設定すると、**TopologyAwareConsistentHashFactory** が使用されます。これは、サーバーヒンティングが有効な **DefaultConsistentHashFactory** と同じです。

サーバーヒンティングは、JBoss Data Grid 実装の高可用性を確実に実現する場合に特に重要になります。

37.2. CONSISTENTHASHFACTORIES

37.2.1. ConsistentHashFactories

Red Hat JBoss Data Grid は、一貫したハッシュアルゴリズムを選択するためのプラグ可能なメカニズムを提供します。これは 4 つの実装に同梱されていますが、カスタム実装を使用することもできます。

JBoss Data Grid には 4 つの ConsistentHashFactory 実装が同梱されています。

- **DefaultConsistentHashFactory** - すべてのノード間でセグメントが均等に分散されるようにします。ただし、キーマッピングは、各キャッシュの履歴に依存するため、複数のキャッシュ間で同じであることは保証されません。consistentHashFactory が指定されない場合は、このクラスが使用されます。
- **SyncConsistentHashFactory** - 現在のメンバーシップが同じである場合、各キャッシュでキーマッピングが同じになることを保証します。ただし、この欠点として、キャッシュに参加するノードによって既存ノードもセグメントを交換する可能性があり、その結果、状態遷移のトラフィックが追加で発生したり、データの分散がより不均等になります。
- **TopologyAwareConsistentHashFactory** - **DefaultConsistentHashFactory** と同等ですが、設定にサーバーヒンティングが含まれる場合に自動的に選択されます。
- **TopologyAwareSyncConsistentHashFactory** - **SyncConsistentHashFactory** と同等ですが、設定にサーバーヒンティングが含まれる場合に自動的に選択されます。

一貫したハッシュ実装をハッシュ設定で選択できます。

```
<distributed-cache consistent-hash-  
factory="org.infinispan.distribution.ch.impl.SyncConsistentHashFactory">
```

この設定は、同じメンバーを持つキャッシュに同一の一貫したハッシュがあることを保証し、さらに **machineId**、**rackId**、または **siteId** 属性がトランスポート設定で指定される場合に、バックアップコピーを複数の物理マシン、ラック、およびデータセンターにまたがって分散します。

想定される欠点として、この設定により、バランスの再調整時に必要とする数以上のセグメントが移動する可能性があります。ただし、この影響は、多数のセグメントを使用することによって軽減できます。

もう 1 つの想定される欠点として、セグメントが均等に分散されないことや、さらには多数のセグメントを実際に使用することによりセグメントの分散状況が悪化することなどがあります。

上述の欠点がありますが、多くの場合、**SyncConsistentHashFactory** および **TopologyAwareSyncConsistentHashFactory** を使用すると、ノードがクラスターに参加した順序に基いてハッシュが計算されないため、クラスター化環境でオーバーヘッドが減少します。また、これらのクラスは通常、各ノードに割り当てられるセグメントの数に大きな違いがあっても許容されるため、デフォルトのアルゴリズムよりも高速になります。

37.2.2. ConsistentHashFactory の実装

カスタム **ConsistentHashFactory** は、以下のメソッド (これらすべては **org.infinispan.distribution.ch.ConsistentHash** の実装を返します) を使って、**org.infinispan.distribution.ch.ConsistentHashFactory** インターフェースを実装する必要があります。

ConsistentHashFactory メソッド

```
create(Hash hashFunction, int numOwners, int numSegments, List<Address>
members, Map<Address, Float> capacityFactors)
updateMembers(ConsistentHash baseCH, List<Address> newMembers,
Map<Address,
Float> capacityFactors)
rebalance(ConsistentHash baseCH)
union(ConsistentHash ch1, ConsistentHash ch2)
```

37.3. キーアフィニティーサービス

37.3.1. キーアフィニティーサービス

キーアフィニティーサービスを使用すると、分散された Red Hat JBoss Data Grid クラスターの特定のノードに値を配置できます。サービスは、識別する指定済みクラスターアドレスに基いて、ハッシュ化されたキーを特定のノードに返します。

キーアフィニティーサービスにより返されたキーは、ユーザー名などの意味を持つことができません。これらは、このレコードのためにアプリケーション全体で使用される無作為の ID にすぎません。提供されたキージェネレーターは、このサービスにより返されるキーが一意であることを保証しません。カスタムキー形式について、**KeyGenerator** の独自の実装を渡すことができます。

このサービスの参照を取得し、使用方法の例を以下に示します。

キーアフィニティーサービス

```
EmbeddedCacheManager cacheManager = getCacheManager();
Cache cache = cacheManager.getCache();
KeyAffinityService keyAffinityService =
    KeyAffinityServiceFactory.newLocalKeyAffinityService(
        cache,
        new RndKeyGenerator(),
        Executors.newSingleThreadExecutor(),
```



```

        100);
Object localKey =
keyAffinityService.getKeyForAddress(cacheManager.getAddress());
cache.put(localKey, "yourValue");

```

以下の手順は、提供された例の説明です。

キーアフィニティーサービスの使用

1. キャッシュマネージャーおよびキャッシュの参照を取得します。
2. これにより、サービスが起動されます。その後、提供された **Executor** を使用してキーが生成され、キューに置かれます。
3. ローカルノードにマップされるサービスからキーを取得します (**cacheManager.getAddress()** はローカルアドレスを返します)。
4. **KeyAffinityService** から取得されたキーを持つエントリは常に、提供されたアドレスを持つノードに格納されます。この場合は、ローカルノードになります。

37.3.2. ライフサイクル

KeyAffinityService は **Lifecycle** を拡張します。これにより、キーアフィニティーサービスを停止、起動、および再起動することが可能になります。

キーアフィニティーサービスライフサイクルパラメーター

```

public interface Lifecycle {
    void start();
    void stop();
}

```

サービスは、**KeyAffinityServiceFactory** を介してインスタンス化されます。すべてのファクトリーメソッドには非同期キー生成に使用される **Executor** があるため、呼び出し元のスレッドでは行われません。ユーザーはこの **Executor** のシャットダウンを制御します。

KeyAffinityService は、不必要になった時点で明示的に停止する必要があります。これにより、バックグラウンドキーの生成が停止され、保持された他のリソースが解放されます。**KeyAffinityService** は、登録されているキャッシュマネージャーがシャットダウンした場合のみ停止します。

37.3.3. トポロジーの変更

KeyAffinityService キーの所有権は、トポロジーが変更されると、変わることがあります。キーアフィニティーサービスは、トポロジーの変更と更新を監視し、古いキーまたは指定されたものと異なるノードにマップされるキーを返さないようにします。ただし、キーが使用されたときにノードアフィニティーが変更されないことは保証されません。以下に例を示します。

1. スレッド (**T1**) は、ノード (**A**) にマップされるキー (**K1**) を読み取ります。
2. トポロジーが変更され、**K1** がノード **B** にマップされます。
3. **T1** は **K1** を使用してキャッシュにデータを追加します。この時点では、**K1** はリクエストされたノードとは異なる **B** にマップします。

上記のシナリオは理想的ではありませんが、クラスターの変更中にすでに使用中のキーを移動できるため、アプリケーションのサポートされた動作です。**KeyAffinityService** は安定したクラスターに対してアクセス近接の最適化を提供します。トポロジーの変更が安定的でないときには適用されません。

第38章 分散実行

38.1. 分散実行

Red Hat JBoss Data Grid は、標準の JDK **ExecutorService** インターフェースより分散実行を提供します。実行のために提出されたタスクは、ローカル JVM ではなく、JBoss Data Grid ノードのクラスター全体で実行されます。

JBoss Data Grid の分散タスクエグゼキューターは、JBoss Data Grid キャッシュノードからのデータを実行タスクの入力として使用できます。そのため、中間または最終結果のキャッシュストアを設定する必要はありません。JBoss Data Grid の入力データはすでに負荷分散されているため、タスクも自動的に分散されます。そのため、明示的にタスクを特定のノードに割り当てる必要はありません。

JBoss Data Grid の分散実行フレームワークでは、以下が行われます。

- 各 **DistributedExecutorService** は単一のキャッシュにバインドされます。提出されたタスクが **DistributedCallable** のインスタンスである場合、そのタスクは特定のキャッシュからキーバリュペアにアクセスできます。
- タスクの 1 つが実行されるときにタスクが他のノードに移行されないようにするため、提出された各 **Callable**、**Runnable**、**DistributedCallable** は、**Serializable** または **Externalizable** である必要があります。**Callable** から返された値も **Serializable** または **Externalizable** である必要があります。

38.2. 分散エグゼキューターサービス

DistributedExecutorService は **DistributedCallable** と、クラスターの他の **Callable** および **Runnable** クラスの実行を制御します。これらのインスタンスは、インストール時に渡される特定キャッシュに関係します。

```
DistributedExecutorService des = new DefaultExecutorService(cache);
```

「[DistributedCallableAPI](#)」の説明にあるとおり、**DistributedCallable** が拡張されると、**DistributedTask** はキーのサブセットに対してのみ実行できます。タスクがこのように単一ノードへ提出されると、JBoss Data Grid は指定のキーが含まれるノードを見つけます。さらに、**DistributedCallable** をこのノードに移行し、**CompletableFuture** を返します。タスクがこのように利用可能なノードすべてに提出された場合、指定のキーが含まれるノードのみがタスクを受け取ります。

DistributedTask が作成されたら、以下のメソッドの 1 つを使ってクラスターに提出することができます。

- **submitEverywhere** メソッドを使用すると、クラスターの利用可能なノードすべてとキーバリュペアすべてにタスクを提出できます。

```
des.submitEverywhere(task)
```

- **submitEverywhere** メソッドは、キーのセットを引数として取ることもできます。このようにキーを渡すと、指定のキーが含まれる利用可能なノードのみにタスクが提出されます。

```
des.submitEverywhere(task, $KEY)
```

- キーが指定された場合、タスクは指定されたキーが 1 つ以上含まれる単一のノードで実行されます。ローカルにないキーは、クラスターから取得されます。このバージョンの **submit** メソッドは、以下の例のように操作する 1 つ以上のキーを受け入れます。

```
des.submit(task, $KEY)
des.submit(task, $KEY1, $KEY2, $KEY3)
```

- ノードの **Address** を **submit** メソッドに渡すと、特定ノードの実行を指示できます。以下は、クラスターの **Coordinator** でのみ実行されます。

```
des.submit(cache.getCacheManager().getCoordinator(), task)
```



注記

デフォルトではタスクは自動的に分散されるため、通常は、実行の対象となる特定のノードを指定する必要はありません。

38.3. DISTRIBUTEDCALLABLE API

DistributedCallable インターフェースは、`java.util.concurrent.package` からの既存の **Callable** のサブタイプです。リモート JVM で実行でき、Red Hat JBoss Data Grid からの入力を受け取ることができます。**DistributedCallable** インターフェースは、JBoss Data Grid のキャッシュデータにアクセスする必要があるタスクのために使用されます。

DistributedCallable API を使用してタスクを実行する場合、タスクのメインアルゴリズムに変更はありませんが、入力ソースは変更になります。

キャッシュへのアクセスや渡したキーセットへのアクセスが必要な場合、**Callable** インターフェースをすでに実装済みのユーザーは **DistributedCallable** を拡張する必要があります。

DistributedCallable API の使用

```
public interface DistributedCallable<K, V, T> extends Callable<T> {

    /**
     * Invoked by execution environment after DistributedCallable
     * has been migrated for execution to a specific Infinispan node.
     *
     * @param cache
     *         cache whose keys are used as input data for this
     *         DistributedCallable task
     * @param inputKeys
     *         keys used as input for this DistributedCallable task
     */
    public void setEnvironment(Cache<K, V> cache, Set<K> inputKeys);
}
```

38.4. CALLABLE および CDI

DistributedCallable を実装できない場合または **DistributedCallable** が適切でない場合に **DistributedExecutorService** で使用される入力キャッシュへの参照が必要であれば、CDI メカニズムによって入力キャッシュをインジェクトするオプションを使用できます。

Callable タスクが Red Hat JBoss Data Grid の実行ノードに到達すると、JBoss Data Grid の CDI メカニズムは適切なキャッシュ参照を提供し、実行している **Callable** にインジェクトします。

Callable を用いて JBoss Data Grid CDI を使用するには、以下の手順に従います。

1. **Callable** で **Cache** フィールドを宣言し、**org.infinispan.cdi.Input** でアノテーションを付けます。
2. 必須の **@Inject** アノテーションを含めます。

callable および CDI の使用

```
public class CallableWithInjectedCache implements Callable<Integer>,
    Serializable {

    @Inject
    @Input
    private Cache<String, String> cache;

    @Override
    public Integer call() throws Exception {
        //use injected cache reference
        return 1;
    }
}
```

38.5. 分散タスクのフェイルオーバー

Red Hat JBoss Data Grid の分散実行フレームワークは、以下の場合でタスクのフェイルオーバーをサポートします。

- タスクが実行されている場所でノードの障害によるフェイルオーバーが発生した場合。
- タスクの障害によるフェイルオーバー。たとえば、**Callable** タスクによって例外が発生した場合。

フェイルオーバーポリシーはデフォルトで無効になっており、**Runnable**、**Callable**、および **DistributedCallable** タスクはフェイルオーバーメカニズムを呼び出しせずに失敗します。

JBoss Data Grid は、ランダムノードフェイルオーバーポリシーを提供し、利用可能な場合に別のランダムなノードで **Distributed** タスクの一部を実行します。

たとえば、以下を使用してランダムフェイルオーバー実行ポリシーを指定できます。

ランダムフェイルオーバー実行ポリシー

```
DistributedExecutorService des = new DefaultExecutorService(cache);
DistributedTaskBuilder<Boolean> taskBuilder =
des.createDistributedTaskBuilder(new SomeCallable());
taskBuilder.failoverPolicy(DefaultExecutorService.RANDOM_NODE_FAILOVER);
```

```
DistributedTask<Boolean> distributedTask = taskBuilder.build();
Future<Boolean> future = des.submit(distributedTask);
Boolean r = future.get();
```

DistributedTaskFailoverPolicy インターフェースを実行してフェイルオーバー管理を提供することもできます。

分散タスクのフェイルオーバーポリシーインターフェース

```
/**
 * DistributedTaskFailoverPolicy allows pluggable fail over target
 * selection for a failed remotely
 * executed distributed task.
 *
 */
public interface DistributedTaskFailoverPolicy {

    /**
     * As parts of distributively executed task can fail due to the task
     * itself throwing an exception
     * or it can be an Infinispan system caused failure (e.g node failed or
     * left cluster during task
     * execution etc).
     *
     * @param failoverContext
     *         the FailoverContext of the failed execution
     * @return result the Address of the Infinispan node selected for fail
     * over execution
     */
    Address failover(FailoverContext context);

    /**
     * Maximum number of fail over attempts permitted by this
     * DistributedTaskFailoverPolicy
     *
     * @return max number of fail over attempts
     */
    int maxFailoverAttempts();
}
```

38.6. 分散タスク実行ポリシー

DistributedTaskExecutionPolicy は、ノードのサブセットへのタスクの実行をスコープ指定することで、タスクが Red Hat JBoss Data Grid クラスター全体でカスタム実行ポリシーを指定できるようにします。

たとえば、**DistributedTaskExecutionPolicy** を使用すると、以下の場合でタスクの実行を管理できます。

- バックアップリモートネットワークセンターではなく、ローカルネットワークの場所でのみタスクが実行される場合。
- タスクの実行に特定の JBoss Data Grid ラックノードの専用サブセットのみが必要な場合。

ラックノードを使用した特定タスクの実行

```
DistributedExecutorService des = new DefaultExecutorService(cache);
DistributedTaskBuilder<Boolean> taskBuilder =
des.createDistributedTaskBuilder(new SomeCallable());
taskBuilder.executionPolicy(DistributedTaskExecutionPolicy.SAME_RACK);
DistributedTask<Boolean> distributedTask = taskBuilder.build();
Future<Boolean> future = des.submit(distributedTask);
Boolean r = future.get();
```

38.7. 分散実行とローカリティ

分散環境の所有権では、**DistributionManager** と **ConsistentHash** は理論的で、これらのクラスはデータがキャッシュでアクティブであるかを認識しません。代わりに、これらのクラスは指定のキーを格納するノードを判断するために使用されます。

指定のキーのローカリティを確認するには、以下のオプションの 1 つを使用します。

- **オプション 1:** 以下の例のように、キーがキャッシュにあり、そのキーがローカルであることを **DistributionManager** が示すことを確認します。

```
(cache.getAdvancedCache().withFlags(SKIP_REMOTE_LOOKUP).containsKey(
key)
&&
cache.getAdvancedCache().getDistributionManager().getLocality(key).isLocal())
```

- **オプション 2:** 直接 **DataContainer** をクエリーします。

```
cache.getAdvancedCache().getDataContainer().containsKey(key)
```



注記

エントリーがパッシブな場合、キーの存在に関わらず **DataContainer** は **False** を返します。

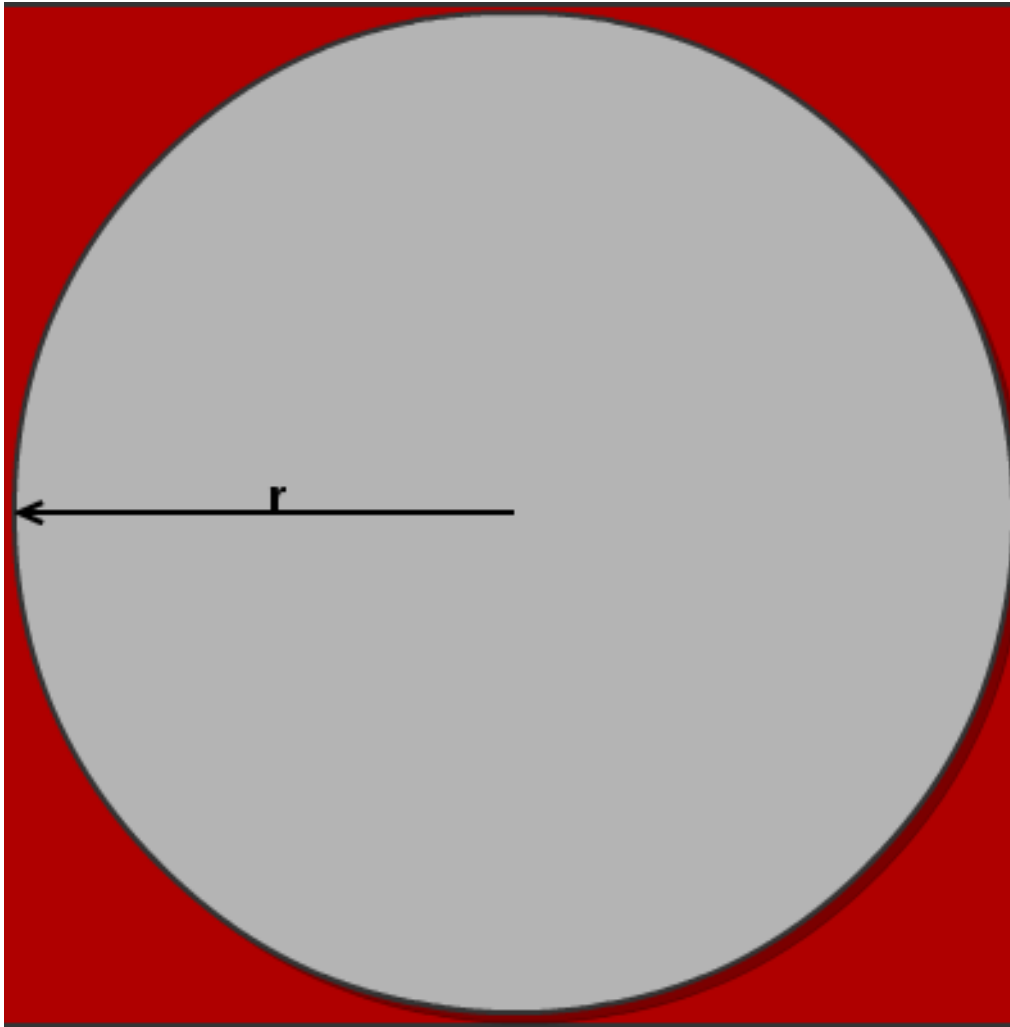
38.7.1. 分散実行の例

この例では、 $\pi()$ の近似値の算出に並列分散実行が使用されます。

1. 以下に示された正方形の面積:
正方形の面積 (S) = $4r^2$
2. 円の面積の方程式:
円の面積 (C) = $\pi \times r^2$
3. 最初の式の r について解く:
 $r^2 = S/4$
4. この r の値を 2 番目の式に挿入し、 π の値を算出:
 $C = S\pi/4$
5. 方程式を解く:
 $C = S\pi/4$
 $4C = S\pi$

$$4C/S = \pi$$

図38.1 分散実行の例



正方形に大量のダーツを投げ、その正方形の中に円を描き、円の内部に突き刺さったすべてのダーツを破棄すると、 C/S の近似値を算出できます。

以前、 π の値は $4C/S$ と算出されました。この値を使用して π の近似値を求めることができます。投げるダーツの数を最大限にすると近似値がより正確になります。

以下の例では、クラスター全体で並列処理して 1 千万本のダーツを投げます。

分散実行の例

```
public class PiAppx {

    public static void main (String [] arg){
        List<Cache> caches = ...;
        Cache cache = ...;

        int numPoints = 100000000;
        int numServers = caches.size();
        int numberPerWorker = numPoints / numServers;

        DistributedExecutorService des = new DefaultExecutorService(cache);
        long start = System.currentTimeMillis();
        CircleTest ct = new CircleTest(numberPerWorker);
```

```

        List<CompletableFuture<Integer>> results = des.submitEverywhere(ct);
        int countCircle = 0;
        for (Future<Integer> f : results) {
            countCircle += f.get();
        }
        double appxPi = 4.0 * countCircle / numPoints;

        System.out.println("Distributed PI appx is " + appxPi +
            " completed in " + (System.currentTimeMillis() - start) + " ms");
    }

    private static class CircleTest implements Callable<Integer>,
        Serializable {

        /** The serialVersionUID */
        private static final long serialVersionUID = 3496135215525904755L;

        private final int loopCount;

        public CircleTest(int loopCount) {
            this.loopCount = loopCount;
        }

        @Override
        public Integer call() throws Exception {
            int insideCircleCount = 0;
            for (int i = 0; i < loopCount; i++) {
                double x = Math.random();
                double y = Math.random();
                if (insideCircle(x, y))
                    insideCircleCount++;
            }
            return insideCircleCount;
        }

        private boolean insideCircle(double x, double y) {
            return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))
                <= Math.pow(0.5, 2);
        }
    }
}

```


第39章 ストリーム

39.1. ストリーム

ストリームは Java 8 で導入され、キャッシュ全体などの変な大きなデータセットで操作を効率的に行うことができます。これらの操作は、データセット全体で手順通りに反復操作を行うのではなく、コレクションで実行されます。

さらに、キャッシュが分散されている場合は、クラスター全体で同時に操作を実行できるため、操作の実行がより効率的になります。

Stream を取得するには、単一スレッドのストリームの場合は **stream()** メソッド、複数スレッドのストリームの場合は **parallelStream()** メソッドを指定の **Map** 上で呼び出します。並列ストリームの詳細は [並列処理](#) で説明します。

39.2. ローカル/インバリデーション/レプリケーションキャッシュでのストリームの使用

ローカル、インバリデーション、またはレプリケーションキャッシュと使用するストリームは、通常のコレクション上のストリームと同じように使用できます。

たとえば複数の Book が含まれるキャッシュで、書名に「JBoss」が含まれるすべてのエントリが含まれるマップを作成するには、以下を使用できます。

```
Map<Object, String> jbossBooks = cache.entrySet().stream()
    .filter(e -> e.getValue().getTitle().contains("JBoss"))
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

39.3. 分散キャッシュでのストリームの使用

ストリーム操作が分散キャッシュで実行されると、中間および終端操作を各ノードに送信し、結果となるデータは元のノードに返信されます。この動作により、操作をリモートノードで実行でき、最終結果のみが返されます。中間値は返されないため、最良のパフォーマンスが実現されます。

リハッシュ対応

ストリームが作成されたら、データはセグメント化され、各ノードはプライマリー所有者として所有するデータでのみ操作を実行します。セグメントの細かさが、ノードごとにデータを均一に分散できる細かさである場合、セグメント全体でデータを均一に処理できます。

データはノード間で再分散されるため、新しいノードが追加されたり、古いノードがクラスターから削除されると、この処理は不安定になります。これにより、データを 2 度処理できる問題が発生することがありますが、[分散ストリーム](#) は自動的にデータの再分散を処理するため、ノードを手動で監視する必要はありません。

39.4. タイムアウトの設定

操作リクエストのタイムアウト値を設定することは可能です。これは、リモートリクエストのみで使用され、リクエストごとに設定されます。そのため、ローカルリクエストは常にタイムアウトせず、フェイルオーバーが発生すると後続の各リクエストには新しいタイムアウトが指定されます。

タイムアウトの指定がない場合、デフォルトではレプリケーションタイムアウトが使用されます。以下の例のようにストリームの `timeout(long timeout, TimeUnit unit)` メソッドを使用すると、手作業で設定できます。

```
CacheStream<Map.Entry<Object, String>> stream = cache.entrySet().stream();
stream.timeout(1, TimeUnit.MINUTES);
```

39.5. 分散ストリーム

39.5.1. 分散ストリーム

分散ストリームは、map reduce と同様に動作します。しかし、分散ストリームでは、ゼロから多数の中間操作があり、その後にはワークが実行される各ノードへ送信される単一の終端操作が続きます。この動作には以下の手順が使用されます。

1. どのノードが各指定セグメントのプライマリ所有者であるかによって、セグメントはグループ化されます。
2. リクエストは各リモートノードに対して生成されます。このリクエストには中間と終端操作が処理するセグメントとともに含まれます。
 - 終端操作が開始されたスレッドは直接ローカル操作を実行します。
 - 各リモートノードは生成されたリクエストを受信し、リモートスレッド上で操作を実行した後、応答を返信します。
3. すべてのリクエストが完了したら、ユーザースレッドはすべての応答を収集し、操作によって指定された削減を実行します。
4. 最終応答がユーザーに返されます。

39.5.2. マーシャルの可能性

分散キャッシュまたはレプリケートされたキャッシュを使用する場合、キーと値がマーシャル可能である必要があります。また、分散ストリームで実行される操作はクラスターの他のノードに送信されるため、これらの操作もマーシャル可能である必要があります。これらをマーシャル可能にするには、一般的に **Serializable** である新しいクラスを使用するか、**Externalizer** が登録済みの新しいクラスを使用します。しかし、**FunctionalInterface** は **Serializable** を実装し、すべてのラムダは即座にシリアルライズされるため、追加のキャストは必要ありません。



注記

分散ストリームの中間値はマーシャル可能である必要はありません。返信される最終値 (通常は終端操作) はマーシャル可能でなければなりません。

ラムダ関数を使用されている場合、パラメーターを **Serializable** のインスタンスとしてキャストするとシリアルライズできます。たとえば、Book エントリーを格納するキャッシュの場合、以下は特定の著者と一致する Book インスタンスのコレクションを作成します。

```
List<Book> books = cache.keySet().stream()
    .filter(e -> e.getAuthor().equals("authorname"))
    .collect(toList());
```

さらに、デフォルトでは作成されたすべての **Collectors** がマーシャル可能であるとは限りません。JBoss Data Grid には、マーシャリングが必要なときに適切に機能する **Collectors** の組み合わせを容易に使用できる **org.infinispan.stream.CacheCollectors** が含まれています。

39.5.3. 並列処理

ストリームを並列処理するメソッドは 2 つあります。

- 並列ストリーム (Parallel Streams) - 単一ノードで各操作が並列して実行されます。
- 並列分散 (Parallel Distribution) - 複数のノードが関係するようにリクエストを並列処理します。

デフォルトでは、分散ストリームは並列分散を有効にしますが、さらに並列 **Stream** に結合できます。これにより、各ノードで複数のスレッドを使用して、複数のノード全体で同時操作を実行できます。

Stream を並列としてマーク付けするには、**parallelStream()** で取得するか、**parallel()** を呼び出して **Stream** の取得後に有効にします。以下の例は両方の方法を示しています。

```
// Obtain a parallel Stream initially
List<Book> books = cache.keySet().parallelStream()
    [...]

// Create the initial stream and then invoke parallel
List<Book> books = cache.keySet().stream()
    .parallel()
    [...]
```



注記

リハッシュ対応イテレーターや `forEach` 操作などの一部の操作では、順次ストリームがローカルで強制されるものがあります。現在、このような操作で並列ストリームを使用することはできません。

39.5.4. 分散演算子

39.5.4.1. 終端演算子の分散結果削減

以下で各終端演算子について説明し、さらに終端演算子に対して分散削減がどのように動作するかも説明します。

- **allMatch**
この演算子は各ノードで実行され、すべての結果はローカルの論理 **AND** 操作を使用して組み合わせられ、最終値が取得されます。通常のストリーム操作が早く返された場合、これらのメソッドも早期に完了します。
- **noneMatch anyMatch**
これらの演算子は各ノードで実行され、すべての結果は論理 **OR** 操作を使用して組み合わせられ、最終値が取得されます。通常のストリーム操作が早く返された場合、これらのメソッドも早期に完了します。
- **collect**
`collect` メソッドはいくつかの追加ステップを実行できます。他のメソッドと同様に、リモートノードはすべて想定どおりに実行します。しかし、最後の `finisher` 演算子を実行する代わりに、完全に組み合わせられた結果を返信します。ローカルスレッドはすべてのローカルおよびリ

モートの結果を組み合わせる値にし、finisher 演算子を実行します。さらに、最終値はシリアル化可能である必要はありませんが、supplier および combiner メソッドによって作成された値はシリアル化する必要があります。

- **count**

count メソッドは各キャッシュから受信した数字を加算します。

- **findAny findFirst**

findAny メソッドは、値がリモートまたはローカルノードからであるかに関わらず、最初に見つかった値を返します。この操作は早期終了をサポートし、最初の値が見つかったら他の値は処理されません。findFirst メソッドの動作と似ていますが、findFirst メソッドには「[中間操作の例外](#)」に説明のあるソートされた中間操作が必要になります。

- **max min**

max および min メソッドは、最終削減の実行前に各ノードでそれぞれの値を見つけ、すべてのノード全体の最大値または最小値を判断します。

- **reduce**

さまざまな reduce メソッドは、ローカルでローカルの結果とリモートの結果を累積する (有効な場合は組み合わせを行う) 前に、できるだけ多く結果をシリアル化します。この動作により、combiner から返された値はシリアル化可能である必要はありません。

39.5.4.2. キーベースのリハッシュ対応演算子

以下の演算子は、他の終端演算子とは異なり、セグメントごとに処理されたキーを追跡するために特別なリハッシュ対応を必要とします。これにより、クラスターのメンバーシップが変更になっても、**iterator** および **spliterator** 演算子では各キーが 1 度のみ処理されることが保証され、**forEach** 演算子では最低でも 1 度処理されることが保証されます。

- **iterator spliterator**

これらの演算子はリモートノードで実行されると、エントリーのバッチを返し、以前のバッチが完全に消費された後でのみ次のバッチが送信されます。この動作は、1 度にメモリーに保持されるエントリーの数を制限します。ユーザーノードは、どのキーが処理されたかを追跡し、セグメントが完了するとこれらのキーはメモリーから解放されます。この動作により、順次処理の使用が推奨され、すべてのノードからのキーを保持する代わりにセグメントキーのサブセットのみをメモリーに保持するようにします。

- **forEach**

forEach はバッチを返しますが、最低でもバッチに値するキーを処理した後でのみバッチを返します。これにより、元のノードは処理済みのキーを認識でき、同じエントリーを再処理する可能性を低減します。しかし、ノードが予期せずダウンした場合に同じセットが繰り返し処理される可能性があります。この場合、ノードがダウンしたときに未完了のバッチの処理中であった可能性があり、リハッシュ障害の操作が発生するときに同じバッチが再度実行される可能性があります。リハッシュのフェイルオーバーはすべての応答が受信されるまで発生しないため、ノードを追加してもこの問題の原因にはなりません。

操作のバッチサイズは、**CacheStream** の **distributedBatchSize** によって制御されます。値が設定されていない場合は、状態の遷移で設定された **chunkSize** がデフォルトとして使用されます。値が大きいと大きなバッチが許可されますが、返信回数が少なくなり、メモリーの使用量が増加します。テストを実行して各アプリケーションに適切なサイズを判断してください。

39.5.4.3. 中間操作の例外

以下の中間操作には特別な例外があります。これらすべてのメソッドのストリーム処理には、適切な処理を保証する人工的なイテレーターのようなものが埋め込まれています。そのため、以下の中間操作を使用すると、パフォーマンスが大幅に劣化する可能性があります。

- **Skip**

人工的なイテレーターは skip 操作まで埋め込まれ、結果はローカルに取り込まれるため、適切な要素の数がスキップされる可能性があります。

- **Peek**

人工的なイテレーターは peek 操作まで埋め込まれます。特定数の peek 処理された要素のみがリモートノードに返され、結果はローカルに取り込まれるため、指定の数のみに peek が行われる可能性があります。

- **Sorted**

人工的なイテレーターは sorted 操作まで埋め込まれ、すべての結果はローカルでソートされます。



警告

この操作では、すべてのエントリーをローカルノードのメモリーに格納する必要があります。

- **Distinct**

Distinct は各リモートノードで実行され、すべての結果に distinct 操作が実行された後に人工的なイテレーターは distinct の値を返します。



警告

この操作では、すべてのエントリーをローカルノードのメモリーに格納する必要があります。

39.5.5. 分散ストリームの例

典型的な Map/Reduce の例が単語数のカウントです。キーと値に対して **String** を持つキャッシュがあり、文すべての単語数を数える必要がある場合、以下を使用して実装できます。

```
Map<String, Long> wordCountMap = cache.entrySet().parallelStream()
    .map(e -> e.getValue().split("\\s"))
    .flatMap(Arrays::stream)
    .collect(CacheCollectors.serializableCollector(() ->
        Collectors.groupingBy(Function.identity(), Collectors.counting())));
```

例を変更して最も頻繁に使用される単語を見つける場合、すべての単語が必要になり、最初にローカルで数える必要があります。以下のスニペットは前述の例を拡張し、この検索を実行します。

```

String mostFrequentWord = cache.entrySet().parallelStream()
    .map((Serializable & Function<Map.Entry<String, String>, String[]>) e
-> e.getValue().split("\\s"))
    .flatMap((Function<String[], Stream<String>>) Arrays::stream)
    .collect(CacheCollectors.serializableCollector(() ->
Collectors.collectingAndThen(
    Collectors.groupingBy(Function.identity(), Collectors.counting()),
    wordCountMap -> {
        String mostFrequent = null;
        long maxCount = 0;
        for (Map.Entry<String, Long> e : wordCountMap.entrySet())
        {
            int count = e.getValue().intValue();
            if (count > maxCount) {
                maxCount = count;
                mostFrequent = e.getKey();
            }
        }
        return mostFrequent;
    }
    )));

```

この時点で、最後のステップが単一のスレッドで実行されます。ローカルで並列ストリームを使用して最終の操作を実行すると、この操作をさらに最適化できます。

```

Map<String, Long> wordCount = cache.entrySet().parallelStream()
    .map((Function<Map.Entry<String, String>, String[]>) e ->
e.getValue().split("\\s"))
    .flatMap((Function<String[], Stream<String>>) Arrays::stream)
    .collect(CacheCollectors.serializableCollector(() ->
Collectors.groupingBy(Function.identity(), Collectors.counting())));
Optional<Map.Entry<String, Long>> mostFrequent =
wordCount.entrySet().parallelStream()
    .reduce((e1, e2) -> e1.getValue() > e2.getValue() ? e1 : e2);

```

第40章 スクリプト

40.1. スクリプト

データグリッドにはサーバーにスクリプトを格納するメソッドが含まれています。リモートクライアントは、JDK の `javax.script.ScriptEngines` を使用してリモートクライアントがローカルでスクリプトを実行できるようにします。デフォルトでは、JDK には JavaScript を実行できる Nashorn が含まれていますが、これを拡張すると独自の **ScriptEngine** を提供する JVM 言語を実行できます。

40.2. スクリプトキャッシュへのアクセス

スクリプトは `__script_cache` という特別な保護されたキャッシュに格納されます。これは保護されたキャッシュであるため、ループバックリクエストや承認が有効になっている接続のみがキャッシュにアクセスできます。

`__script_cache` へリモートで接続するには、以下の要件を満たす必要があります。

- ユーザーが `__script_manager` ロールで定義されている必要があります。
- クライアントはサーバーへセキュアに接続できる必要があります。これを実現するには、「[インターフェースのセキュア化](#)」の手順にしたがってください。
- キャッシュコンテナで承認が有効になっている必要があります。

スクリプトキャッシュへのアクセスをサーバーで設定

以下の例は、Hot Rod コネクターをセキュア化する **DIGEST-MD5** メソッドを使用して、スクリプトキャッシュへのアクセスをサーバーで設定する方法を示しています。

1. 以下のように、ユーザーをサーバーに追加します。

- a. `$JDG_HOME/bin/add-user.sh` スクリプト (Linux の場合) または `$JDG_HOME/bin/add-user.bat` スクリプト (Windows の場合) を実行します。

- b. 最初のプロンプトで **b** を作成し、**ApplicationRealm** ユーザーを作成します。

```
What type of user do you wish to add?
a) Management User (mgmt-users.properties)
b) Application User (application-users.properties)
(a): b
```

- c. プロンプトにしたがって、ユーザーのユーザー名とパスワードを定義します。

- d. グループの入力を要求されたら、このユーザーに対して `__script_manager` を入力します。

```
What groups do you want this user to belong to? (Please enter a
comma separated list, or leave blank for none)[ ]:
__script_manager
```

2. クライアントサーバー間の通信をセキュア化します。この例では **DIGEST-MD5** を使用するため、「[Hot Rod 認証 \(MD5\) の設定](#)」の手順にしたがいます。以下のスニペットは必要な xml 設定を示しています。

```

<cache-container name="local" default-cache="default"
statistics="true">
  <security>
    <authorization>
      <identity-role-mapper />
      <role name="admin" permissions="ALL" />
      <role name="reader" permissions="READ" />
      <role name="writer" permissions="WRITE" />
      <role name="supervisor" permissions="READ WRITE EXEC" />
    </authorization>
  </security>
  [...]
</cache-container>
[...]
<hotrod-connector socket-binding="hotrod" cache-container="local">
  <authentication security-realm="ApplicationRealm">
    <sasl server-name="scriptserver" mechanisms="DIGEST-MD5"
qop="auth" />
  </authentication>
</hotrod-connector>

```

3. 以下のコードスニペットのように、セキュアな接続を使用してキャッシュマネージャーを作成します。

```

Configuration config = new ConfigurationBuilder()
    .addServer()
        .host("localhost")
        .port(11222)
    .security()
        .authentication()
        .enable()
        .saslMechanism("DIGEST-MD5")
        .serverName("scriptserver")
        .callbackHandler(new MyCallbackHandler("user",
"ApplicationRealm", "password".toCharArray()))
    .build();

cacheManager = new RemoteCacheManager(config);

```

40.3. スクリプトのインストール

スクリプトを `__script_cache` に追加するには、スクリプトの名前をキーとし、スクリプトの内容を値としてキャッシュ自体をスクリプトに追加します。スクリプトの名前に **sample.js** などのファイル拡張子が含まれている場合、スクリプトを実行するエンジンは拡張子によって決定されます。この動作をオーバーライドするには、スクリプト内部のメタデータを指定します。

スクリプトの内容は `__script_cache` の値に格納する必要があり、既存のファイルからロードするか手作業で入力することができます。以下の例はこれらのオプションを示しています。

ファイルからスクリプトをロード

ファイル内にスクリプトが格納されていること前提とした場合、以下のコードサンプルを使用するとファイルの内容を読み出し、スクリプティングキャッシュへ格納することができます。

```
private static final String SCRIPT_CACHE = "__script_cache";
```



```
private RemoteCache<String, String> scriptingCache;
[...]
    scriptingCache = cacheManager.getCache(SCRIPT_CACHE);
[...]

    private void loadScript(String filename) throws IOException{
        StringBuilder sb = new StringBuilder();
        BufferedReader reader = new BufferedReader(new
FileReader(filename));
        for (String line = reader.readLine(); line != null; line =
reader.readLine()) {
            sb.append(line);
            sb.append("\n");
        }
        System.out.println(sb.toString());
        scriptingCache.put(filename, sb.toString());
    }
}
```

スクリプトの内容を定義

ファイルからスクリプトをロードする代わりに、手作業でスクリプトを定義し、スクリプティングキャッシュに置くことができます。

```
RemoteCache<String, String> scriptCache =
cacheManager.getCache("__script_cache");
scriptCache.put("multiplication.js",
    "// mode=local,language=javascript\n" +
    "// parameters=[multiplicand,multiplier]" +
    "multiplicand * multiplier\n");
```

40.4. メタデータのスクリプト

メタデータをスクリプトに格納し、スクリプトの実行方法に関する追加情報をサーバーに提供することができます。このメタデータは、スクリプトの最初の行にある特別書式のコメントに含まれます。

プロパティは、カンマ区切りのキーバリューペアとして定義され、コメントのスタイルは `//`、`;;`、`\#` など、使用されるスクリプト言語によって異なります。必要な場合はこの情報を複数の行に分散でき、1 重または 2 重引用符を使用して値が区切られます。

以下は有効なメタデータコメントの例になります。

```
// name=test, language=javascript
// mode=local, parameters=[a,b,c]
```

メタデータプロパティ

以下のメタデータプロパティを使用できます。

- **mode**: スクリプト実行モードを定義します。以下の値の 1 つになります。
 - **local**: スクリプトはリクエストを処理するノードのみによって実行されます。スクリプト自体はクラスター化された操作を呼び出しできます。
 - **distributed**: 分散エグゼキューターサービスを使用してスクリプトを実行します。

- **language**: Javascript など、スクリプトの実行に使用されるスクリプトエンジンを定義します。
- **extension**: js などのスクリプトの実行に使用されるスクリプトエンジンを指定する代替メソッド。
- **role**: スクリプトの実行に必要な特定のロール。
- **parameters**: このスクリプトの有効なパラメーター名のアレイ。パラメーター名を指定する呼び出しがこのリストに含まれていないと、例外が発生します。

実行モードはスクリプトの特徴であるため、異なるモードでスクリプトを呼び出しするためにクライアント側で追加の設定を行う必要はありません。

40.5. スクリプトバインディング

スクリプトエンジンはスクリプトの実行時に複数の内部オブジェクトを事前定義のバインディングとして公開します。これらの内部オブジェクトは次のとおりです。

- **cache**: このキャッシュに対してスクリプトが実行されます。
- **cacheManager**: キャッシュの cacheManager。
- **marshaller**: キャッシュへデータをマーシャル/アンマーシャルするために使用されるマーシャラー。
- **scriptingManager**: スクリプトの実行に使用されているスクリプトマネージャーのインスタンス。これを使用してスクリプトから別のスクリプトを実行できます。

40.6. スクリプトパラメーター

スクリプトには、標準のバインディングの他に、バインディングのようにも見える名前付きパラメーターのセットを渡すことができます。パラメーターは、名前と値のペアのマップとして渡され、名前は文字列で、値は使用中のマーシャラーが理解できる値になります。

multiplicand と **multiplier** の 2 つのパラメーターを取る以下のスクリプトを見てみましょう。

```
// mode=local, language=javascript
// parameters=[multiplicand, multiplier]
multiplicand * multiplier
```

最後の操作は評価であるため、その結果はスクリプトインボーカーへ返されます。渡された値はスクリプトの実行方法に応じて変更され、各実行メソッドによって対応されます。

40.7. HOT ROD JAVA クライアントを使用したスクリプトの実行

サーバー側で承認が無効になっている場合、スクリプトがインストールされると誰でも実行することができます。その他の場合では、**EXEC** パーミッションを持つユーザーのみがインストール済みのスクリプトを実行することができます。

Hot Rod でスクリプトを実行するにはスクリプトを実行するキャッシュ上で **execute(scriptName, parameters)** を呼び出します。この場合、**scriptName** は **__script_cache** に格納されたスクリプトの名前で、**parameters** は名前付きパラメーターの **Map<String, Object>** になります。

以下は、Hot Rod 経由で上記の **multiplication.js** スクリプトを実行する例になります。

```
RemoteCache<String, Integer> cache = cacheManager.getCache();
// Create the parameters for script execution
Map<String, Object> params = new HashMap<>();
params.put("multiplicand", 10);
params.put("multiplier", 20);
// Run the script on the server, passing in the parameters
Object result = cache.execute("multiplication.js", params);
```

40.8. スクリプトの例

以下の例は、ユーザーが構文のスクリプト化に関する理解を深め、各環境でスクリプトに適切なタスクを考慮するためのさまざまなタスクを示しています。

分散実行

以下は、分散エグゼキューター内で実行されるスクリプトです。各ノードはそのアドレスを返し、すべてのノードはクライアントに返す **List** に収集されます。

```
// mode:distributed,language=javascript
cacheManager.getAddress().toString();
```

単語カウントストリーム

以下は、ローカルキャッシュで実行され、結果セットで各単語の発生回数を数え、単語と発生回数をキーバリューペアで返すスクリプトになります。

```
// mode=local,language=javascript
var Function = Java.type("java.util.function.Function")
var Collectors = Java.type("java.util.stream.Collectors")
var Arrays = Java.type("org.infinispan.scripting.utils.JSArrays")
cache
    .entrySet().stream()
    .map(function(e) e.getValue())
    .map(function(v) v.toLowerCase())
    .map(function(v) v.split(/[\\W]+/))
    .flatMap(function(f) Arrays.stream(f))
    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()));
```

40.9. 格納されたスクリプト実行時の制限

Java ストリームを DIST モードのクラスターと使用するとエラーが発生します。

クラスターが **DIST** モードである場合、JavaScript で **Stream** を作成するスクリプトを使用することはできません。このようなスクリプトを実行しようとする、シリアル化の際にラムダが失敗し、**NotSerializableException** が発生します。この問題を回避するため、**Iterator** を使用してデータ上で手動で繰り返し処理を行うか、データがスクリプトから発信元ノードへ遷移された後にラムダを実行することが推奨されます。

他のモードのクラスターでストリームを使用した場合は問題はありません。

第41章 リモートタスクの実行

41.1. リモートタスクの実行

タスクまたはビジネスロジックは直接 JBoss Data Grid サーバーで実行できるため、タスクの実行はデータに近く、クラスターのすべてのノードにあるリソースを使用します。

タスクを Java 実行可能ファイルにバンドルし、プログラミングで実行可能ファイルを実行できるサーバーインスタンスにデプロイすることができます。

41.2. リモートタスクの作成

リモート実行のタスクを作成するには、**org.infinispan.tasks.ServerTask** インターフェースを実装するクラスが含まれる **.jar** ファイルを作成する必要があります。

実装には以下のメソッドが必要になります。

- **void setTaskContext(TaskContext taskContext)**: タスクコンテキストを設定します。このメソッドを使用して、キャッシュや必要なその他のリソースにアクセスします。
- **String getName()**: タスクに一意的な名前を提供します。この名前は **TaskManager** による実行に使用されます。

以下は実装の任意のメソッドになります。

- **TaskExecutionMethod getExecutionMode()**: **TaskExecutionMode.ONE_NODE** のようにタスクが1つのノードで実行されるか、または **TaskExecutionMode.ALL_NODES** のようにすべてのノードで実行されるかを決定します。デフォルトでは1つのノードで実行されます。
- **Optional<String> getAllowedRole()**: ユーザーがタスクの実行に必要なロールを設定します。デフォルトでは追加のユーザーロールは設定されません。詳細は「[リモートタスクの実行](#)」を参照してください。
- **Set<String> getParameters()**: タスクに使用する名前付きパラメーターを指定します。

41.3. リモートタスクの例

以下には、**org.infinispan.tasks.ServerTask** インターフェースを実装するクラスの例が含まれています。

```
public class HelloTask implements ServerTask<String> {

    private TaskContext ctx;

    //Set the task context.
    @Override
    public void setTaskContext(TaskContext ctx) {
        this.ctx = ctx;
    }

    //Take the name of a person as a parameter.
    //Return a greeting with that person's name.
    @Override
```

```

    public String call() throws Exception {
        String name = (String) ctx.getParameters().get().get("name");
        return "Hello " + name;
    }

    //Return a unique name that clients can use to invoke the task.
    @Override
    public String getName() {
        return "hello-task";
    }
}

```

41.4. リモートタスクのインストール

リモートタスクを作成し、**.jar** ファイルにバンドルしたら、以下のオプションの 1 つを使用して JBoss Data Grid サーバーインスタンスにデプロイできます。

オプション 1: **deployments** ディレクトリーへコピーする

1. **.jar** ファイルを **deployments/** ディレクトリーにコピーします。

```
$] cp /path/to/sample_task.jar $JDG_HOME/standalone/deployments/
```

オプション 2: CLI でコピーする

1. JBoss Data Grid サーバーに接続します。

```
[$JDG_HOME] $ bin/cli.sh --connect --controller=$IP:$PORT
```

2. **.jar** ファイルをデプロイします。

```
deploy /path/to/sample_task.jar
```



注記

JBoss Data Grid がドメインモードである場合、**--all-server-groups** または **--server-groups** パラメーターを使用してサーバーグループを市営する必要があります。

41.5. リモートタスクの削除

JBoss Data Grid の実行中のインスタンスからリモートタスクを削除するには、以下を行います。

1. JBoss Data Grid サーバーに接続します。

```
[$JDG_HOME] $ bin/cli.sh --connect --controller=$IP:$PORT
```

2. **undeploy** コマンドを実行して **.jar** ファイルを削除します。

```
undeploy /path/to/sample_task.jar
```



注記

JBoss Data Grid がドメインモードである場合、**--all-server-groups** または **--server-groups** パラメーターを使用してサーバーグループを市営する必要があります。

41.6. リモートタスクの実行

JBoss Data Grid サーバーで承認が有効になっている場合、**EXEC** パーミッションを持つユーザーのみがリモートタスクを実行できます。承認が有効になっていない場合は、すべてのユーザーがリモートタスクを実行できます。

リモートタスクには、**getAllowedRole** メソッドで指定された追加のユーザーロールを付与することができます。この場合、ユーザーはそのロールに属しないとリモートタスクを実行できません。

以前デプロイしたタスクを実行するには、対象のキャッシュで **execute(String taskName, Map parameters)** を呼び出します。

以下の例は、**sampleTask** という名前のタスクを実行する方法を示しています。

```
import org.infinispan.client.hotrod.*;
import java.util.*;
[...]
String TASK_NAME = "sampleTask";

RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache remoteCache = rcm.getCache();

// Assume the task takes a single parameter, and will return a result
Map<String, String> params = new HashMap<>();
params.put("name", "James");

String result = (String) remoteCache.execute(TASK_NAME, params);
```

第42章 データの相互運用性

42.1. プロトコルの相互運用性

42.1.1. プロトコルの相互運用性

Red Hat JBoss Data Grid のプロトコルの相互運用性により、C++ または Java などの各種プログラミング言語で書かれた REST、Memcached、および Hot Rod などの各種プロトコルによる、raw バイト形式のデータへの読み書きアクセスを可能にします。

デフォルトで、各プロトコルはそのプロトコルの最も効率的な形式でデータを保存し、エントリーの取得時に変換が必要とならないようにします。このデータを複数のプロトコルからアクセスする必要がある場合、互換性モードは、共有されるキャッシュで有効にされる必要があります。

互換性モードの有効化

互換性モードを有効にする手順は、『[Administration and Configuration Guide](#)』を参照してください。

42.1.2. ユースケースおよび要件

以下の表は、Red Hat JBoss Data Grid におけるデータ相互運用性の一般的なユースケースを示しています。

表42.1 互換性モードのユースケース

ユースケース	クライアント A (リーダーまたはライター)	クライアント B (クライアント A に対する書き込み/読み取りクライアント)
1	Memcached	Hot Rod Java
2	REST	Hot Rod Java
3	Memcached	REST
4	Hot Rod Java	Hot Rod C++
5	Hot Rod Java	Hot Rod C#
6	Memcached	Hot Rod C++

これらのユースケースでは、マーシャルはユーザーの判断に委ねられます。JBoss Data Grid は byte[] を格納し、ユーザーはこれを意味のあるデータにマーシャルまたはアンマーシャルします。

たとえばユースケース 1 の場合、相互運用性は Memcached クライアント (A) と Hot Rod Java クライアント (B) の間になります。クライアント A が **Person** インスタンスなどのアプリケーション固有のオブジェクトをシリアル化する場合、String をキーとして使用できます。

以下の手順はすべてのユースケースに適用されます。

クライアント A 側

1. A は Protobuf や Avro などのサードパーティーマーシャラーを使用して **Person** の値を `byte[]` にシリアル化します。UTF-8 でエンコードされた文字列をキーとして使用する必要があります (Memcached プロトコルの要件のため)。
2. A はキーバリューペアをサーバーに書き込みます (キーを UTF-8 文字列、値をバイトアレイとして)。

クライアント B 側

1. B は特定キー (String) の **Person** を読み取る必要があります。
2. B は同じ UTF-8 キーに対応する `byte[]` にシリアル化します。
3. B は **get(byte[])** を呼び出します。
4. B はシリアル化されたオブジェクトを表す `byte[]` を取得します。
5. B は A と同じマーシャラーを使用して `byte[]` に対応する **Person** オブジェクトにアンマーシャルします。



注記

- ユースケース 4 では、Hot Rod Java クライアントに含まれる Protostream Marshaller の使用が推奨されます。Hot Rod C++ クライアントでは、Google の Protobuf Marshaller (<https://developers.google.com/protocol-buffers/docs/overview>) が推奨されます。
- ユースケース 5 では、デフォルトの Hot Rod マーシャラーを使用できます。

42.1.3. REST 上のプロトコル相互運用性

JBoss Data Grid の REST インターフェースは、データを格納および読み出しするために Hot Rod および Memcached クライアントがアクセスできるキャッシュを公開します。

各キャッシュは、メディアタイプによって定義された設定可能な形式でデータを格納します。JBoss Data Grid では、クライアントは異なる形式でデータを読み書きでき、必要時に形式を自動的に変換できます。

デフォルトのメディアタイプやメディアタイプの変換などが含まれる REST API の詳細は、『[The REST API](#)』を参照してください。

第43章 データセンター間のレプリケーションのセットアップ

43.1. データセンター間レプリケーション

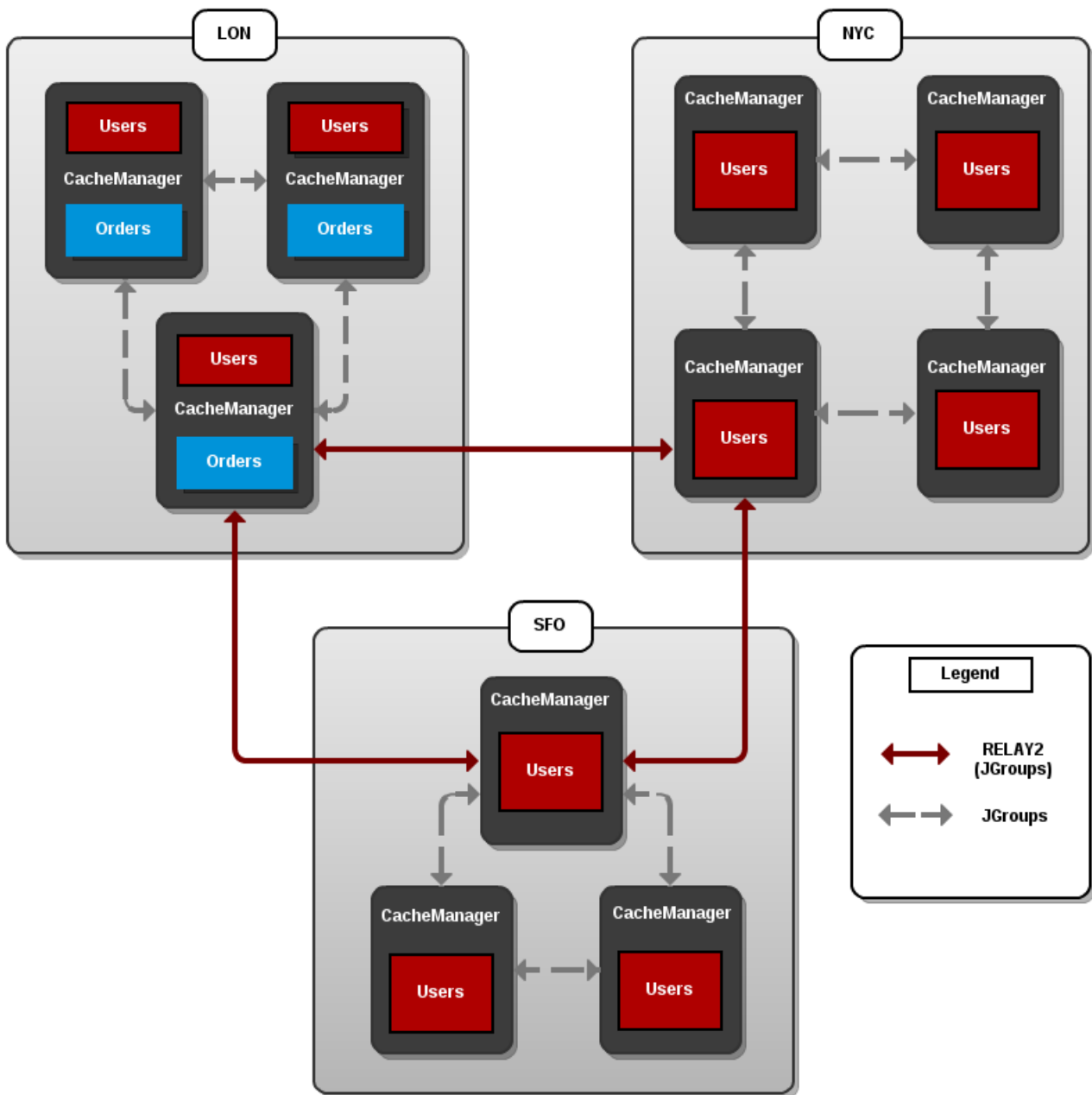
Red Hat JBoss Data Grid では、データセンター間レプリケーションにより、管理者は複数のクラスターでデータのバックアップを作成することができます。これらのクラスターは物理的に同じ場所または異なる場所に置くことができます。JBoss Data Grid のサイト間レプリケーションの実装は JGroups の **RELAY2** プロトコルをベースとします。

データセンター間レプリケーションにより、クラスター全体におけるデータの冗長性が保証されます。データを復元するためのバックアップを作成する他に、これらのデータセットをアクティブ-アクティブモードで使用することもできます。このように設定されると、あるクラスターに障害が発生したときに別の環境にあるシステムがセッションを処理することができます。各クラスターを他のクラスターとは物理的に異なる場所に置くことが理想的です。

43.2. データセンター間レプリケーションの操作

Red Hat JBoss Data Grid のデータセンター間レプリケーションの操作は、以下のような例を使用して説明できます。

図43.1 データセンター間レプリケーションの例



この例では、**LON**、**NYC** および **SFO** の3つのサイトが設定されます。それぞれのサイトは、3つから4つの物理ノードから構成される実行中の JBoss Data Grid クラスターをホストします。

Users キャッシュは、**LON**、**NYC** および **SFO** の3つのすべてのサイトでアクティブです。これらのサイトのいずれかの **Users** キャッシュへの変更は、キャッシュが設定で他の2つのサイトをバックアップとして定義している限り、それらの他の2つのサイトにレプリケートされます。ただし、**Orders** キャッシュは、他のサイトにレプリケートされないため **LON** サイトでのみ利用できます。

Users キャッシュは各サイトで異なるレプリケーションメカニズムを使用できます。たとえば、データのバックアップを **SFO** に同期的に、**NYC** と **LON** に非同期的に行います。

さらに、**Users** キャッシュの設定をサイトごとに変えることもできます。たとえば、**LON** サイトでは **owners** を 2 に設定した分散キャッシュとして設定し、**NYC** サイトではレプリケートされたキャッシュとして、さらに **SFO** サイトでは **owners** を 1 に設定した分散キャッシュとして設定することができます。

JGroups は各サイト内の通信やサイト間の通信に使用されます。**RELAY2** という JGroups プロトコルは、サイト間の通信を容易にします。詳細は、JBoss Data Grid『**Administration Guide**』の「**RELAY2**」を参照してください。

43.3. プログラミングによるデータセンター間レプリケーションの設定

Red Hat JBoss Data Grid でデータセンター間のレプリケーションを設定するプログラムを用いた方法を以下に示します。

プログラミングによるデータセンター間レプリケーションの設定

1. ノードの場所を特定します。
ノードが所属するサイトを宣言します。

```
globalConfiguration.site().localSite("LON");
```

2. JGroups を設定します。
RELAY プロトコルを使用するように JGroups を設定します。

```
globalConfiguration.transport().addProperty("configurationFile", "jgroups-with-relay.xml");
```

3. リモートサイトをセットアップします。
リモートサイトにレプリケートするために JBoss Data Grid キャッシュをセットアップします。

```
ConfigurationBuilder lon = new ConfigurationBuilder();
lon.sites().addBackup()
    .site("NYC")
    .backupFailurePolicy(BackupFailurePolicy.WARN)
    .strategy(BackupConfiguration.BackupStrategy.SYNC)
    .replicationTimeout(12000)
    .sites().addInUseBackupSite("NYC")
    .sites().addBackup()
    .site("SFO")
    .backupFailurePolicy(BackupFailurePolicy.IGNORE)
    .strategy(BackupConfiguration.BackupStrategy.ASYNC)
    .sites().addInUseBackupSite("SFO")
```

4. オプション: バックアップクラッシュを設定します。
JBoss Data Grid は、リモートサイトと同じ名前でデータをキャッシュに暗黙的にレプリケートします。リモートサイトのバックアップキャッシュの名前が異なる場合、データが確実に正しいキャッシュにレプリケートされるようにするため **backupFor** キャッシュを指定する必要があります。



注記

この手順は任意であり、リモートサイトのキャッシュの名前が元のキャッシュとは異なる場合にのみ必要になります。

- a. **LON** からのバックアップデータを受信できるようにサイト **NYC** のキャッシュを設定します。

■

```
ConfigurationBuilder NYCbackupOfLon = new ConfigurationBuilder();
NYCbackupOfLon.sites().backupFor().remoteCache("lon").remoteSite(
    "LON");
```

- b. **LON** からバックアップデータを受信できるようにサイト **SFO** のキャッシュを設定します。

```
ConfigurationBuilder SFObackupOfLon = new ConfigurationBuilder();
SFObackupOfLon.sites().backupFor().remoteCache("lon").remoteSite(
    "LON");
```

5. 設定ファイルの内容を追加します。

デフォルトでは、Red Hat JBoss Data Grid の **infinispan-embedded-{VERSION}.jar** パッケージに **default-configs/default-jgroups-tcp.xml** や **default-configs/default-jgroups-udp.xml** などのJGroups 設定ファイルが含まれています。

JGroups 設定ファイルを新規ファイル (この例ではファイル名は **jgroups-with-relay.xml**) にコピーし、指定の設定情報をこのファイルに追加します。**relay.RELAY2** プロトコル設定は設定スタックの最新のプロトコルである必要があります。

```
<config>
  <!-- Additional configuration information here -->
  <relay.RELAY2 site="LON"
    config="relay.xml"
    relay_multicasts="false" />
</config>
```

6. relay.xml ファイルを設定します。

relay.xml ファイルで **relay.RELAY2** 設定をセットアップします。このファイルにはグローバルクラスター設定が記述されています。

```
<RelayConfiguration>
  <sites>
    <site name="LON"
      id="0">
      <bridges>
        <bridge config="jgroups-global.xml"
          name="global"/>
      </bridges>
    </site>
    <site name="NYC"
      id="1">
      <bridges>
        <bridge config="jgroups-global.xml"
          name="global"/>
      </bridges>
    </site>
    <site name="SFO"
      id="2">
      <bridges>
        <bridge config="jgroups-global.xml"
          name="global"/>
      </bridges>
    </site>
  </sites>
</RelayConfiguration>
```

```

        </site>
    </sites>
</RelayConfiguration>

```

7. グローバルクラスターを設定します。

relay.xml で参照される **jgroups-global.xml** ファイルには、グローバルクラスター、つまりサイト間の通信で使用される別の JGroups 設定が含まれます。

グローバルクラスターは、通常 **TCP** ベースで、**TCPPING** プロトコル (**PING** や **MPING** ではなく) を使用してメンバーを検索します。**default-configs/default-jgroups-tcp.xml** の内容を **jgroups-global.xml** にコピーし、**TCPPING** を設定するために以下の設定を追加します。

```

<config>
    <TCP bind_port="7800" <!-- Additional configuration information
here --> />
    <TCPPING
initial_hosts="lon.hostname[7800],nyc.hostname[7800],sfo.hostname[78
00]"
        ergonomics="false" />
    <!-- Rest of the protocols -->
</config>

```

TCPPING.initial_hosts のホスト名 (または IP アドレス) をサイトマスターに使用されるものに置き換えます。ポート (この例では **7800**) は **TCP.bind_port** と一致する必要があります。

TCPPING プロトコルの詳細は、JBoss Data Grid の『**Administration and Configuration Guide**』を参照してください。

43.4. サイトをオフラインにする

Red Hat JBoss Data Grid のデータセンター間のレプリケーション設定では、一定の時間間隔に 1 つのサイトへのバックアップが複数回失敗する場合、そのサイトを自動的にオフラインとすることができま。この機能により、サイトをオフラインとする管理者による手動の介入が不要になります。

プログラムを用いて Red Hat JBoss Data Grid でデータセンター間のレプリケーションサイトを自動的にオフラインにするよう設定するには、以下を行います。

サイトをオフラインにする (プログラムを使用)

```

lon.sites().addBackup()
    .site("NYC")
    .backupFailurePolicy(BackupFailurePolicy.FAIL)
    .strategy(BackupConfiguration.BackupStrategy.SYNC)
    .takeOffline()
        .afterFailures(500)
        .minTimeToWait(10000);

```

43.5. HOT ROD サイト間クラスターフェイルオーバー

クラスター内のフェイルオーバーに加えて、Hot Rod クライアントは、それぞれが独立したサイトを表す異なるクラスターにフェイルオーバーできます。Hot Rod サイト間クラスターフェイルオーバーは自動および手動モードの両方で利用できます。

自動サイト間フェイルオーバー

メイン/プライマリークラスターノードが利用できない場合、クライアントアプリケーションは、代替方法で定義されたクラスターをチェックし、それらへのフェイルオーバーを試行します。フェイルオーバーが正常に実行されると、クライアントはその代替クラスターが利用不可になるまでそのクラスターに接続されたままになります。その後、クライアントは他の定義済みクラスターにフェイルオーバーし、接続が復元される場合は最終的に元のサーバー設定を持つメイン/プライマリークラスターに切り替わります。

Hot Rod クライアントに代替クラスターを設定するには、以下の例が示す設定済みクラスターのそれぞれに対して 1 つ以上のホスト/ポートのペアの詳細情報を指定します。

代替クラスターの設定

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
    = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.addCluster("remote-cluster").addClusterNode("remote-cluster-host",
11222);
RemoteCacheManager rcm = new RemoteCacheManager(cb.build());
```



注記

クラスター定義を問わず、デフォルトのサーバーホストおよびポートの詳細を使用して初期サーバーが解決されない場合は初期サーバー設定を指定する必要があります。

手動によるサイト間クラスターフェイルオーバー

手動によるサイトクラスターのスイッチオーバーでは、RemoteCacheManager の **switchToCluster(clusterName)** または **switchToDefaultCluster()** を呼び出します。

switchToCluster(clusterName) を使用すると、ユーザーは Hot Rod クライアント設定で事前定義されたクラスターの 1 つへの切り替えをクライアントに対して強制できます。デフォルトのクラスターへ切り替える場合は、代わりに **switchToDefaultCluster()** を使用します。

第44章 ニアキャッシュ

44.1. ニアキャッシュ

ニアキャッシュは Rod Rod Java クライアント実装の任意のキャッシュで、ユーザーの近くに最近アクセスされたデータを保持し、頻繁に使用されるデータへのアクセスをより迅速にします。このキャッシュは、**get** または **getVersioned** 操作によってリモートエントリーが読み出されると更新されるローカルの Hot Rod クライアントキャッシュとして動作します。



重要

ライブラリーモードのニアキャッシュまたは Hot Rod 以外のインターフェースは、1 次キャッシュを設定して実現します。1 次キャッシュの設定方法は、JBoss Data Grid の『**Administration and Configuration Guide**』を参照してください。

Red Hat JBoss Data Grid では、エントリーの変更時または削除時にクライアントへ通知を送信するリモートイベントを使用して、ニアキャッシュの整合性を保持します (「[リモートイベントリスナー \(Hot Rod\)](#)」を参照してください)。ニアキャッシュでは、ローカルキャッシュはリモートキャッシュと整合性をとります。サーバーのリモートエントリーが更新または削除されるたびにローカルエントリーは更新またはインバリデートされます。クライアントレベルでは、ニアキャッシュは以下のいずれかとして設定されます。

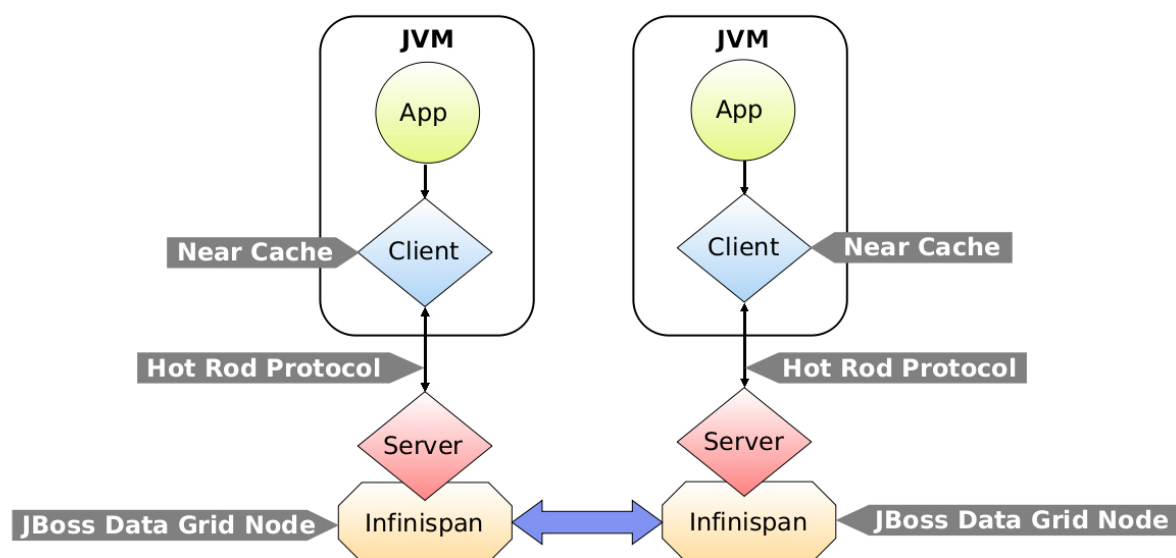
- **DISABLED**: デフォルトのモード。ニアキャッシュが有効になっていないことを示します。
- **INVALIDATED**: ニアキャッシュを有効にし、インバリデーションメッセージ経由でリモートキャッシュとの同期を維持します。



注記

デフォルトでは、Hot Rod クライアントに対してニアキャッシュは無効になっています。

図44.1 ニアキャッシュアーキテクチャー



44.2. ニアキャッシュの設定

ニアキャッシュは、Hot Rod クライアントに変更を加えずに、設定から有効または無効にすることができます。ニアキャッシュを有効にするには、ニアキャッシュモードをクライアント側で **INVALIDATED** として設定し、任意でキャッシュに保持するエントリーの数を指定します。

ニアキャッシュのモードは、**NearCacheMode** 列挙を使用して設定されます。

以下の例はニアキャッシュの設定方法を表しています。

ニアキャッシュの有効化

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
...

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(100);
```

maxEntries(int maxEntries) メソッドを使用してニアキャッシュの最大サイズを提供する必要があります。上記の設定では 100 に定義されています。最大サイズに到達した場合、ニアキャッシュのエントリーは LRU (leaset-recently-used、最近最も使われていない) アルゴリズムを使用してエビクトされます。無制限のニアキャッシュを定義するには、ゼロまたは負の値を渡します。

44.3. クラスター環境でのニアキャッシュ

ニアキャッシュは Hot Rod リモートイベントを使用して実装され、クラスター化リスナーを利用してクラスター全体からのイベントを受信します。クラスター化リスナーはクラスター内の単一ノード上にインストールされ、残りのノードはリスナーがインストールされるノードにイベントを送信します。そのため、ニアキャッシュがバックアップするクラスター化リスナーを実行しているノードに障害が発生する可能性があります。このような場合、別のノードがクラスター化リスナーを引き継ぎます。

クラスター化リスナーを実行しているノードに障害が発生した場合、クライアントフェイルオーバーイベントコールバックを定義および呼び出しすることができます。ニアキャッシュでは、フェイルオーバー中にイベントを見逃す可能性があるため、このコールバックとその実装がニアキャッシュを消去します。

詳細は、「[クラスター化リスナー](#)」を参照してください。

第45章 競合マネージャーの使用方法

45.1. キャッシュの競合の検索および解決

競合マネージャー (Conflict Manager) は通常 [パーティション処理 \(Partition Handling\)](#) とともに使用されます。スプリットブレインは、クラスターのノードがお互いに通信できない複数のグループ (パーティション) に分割されると発生します。場合によっては、ノードに異なるデータを書き込むことが可能になります。このような場合、JBoss Data Grid のパーティション処理を競合マネージャーとともに使用すると、ノード全体で同じ **CacheEntries** の違いを自動的に解決することができます。競合マネージャーは、手動による競合の検索および解決にも使用できます。

以下のコードは、**EmbeddedCacheManager** の **ConflictManager** を取得する方法、指定キーのバージョンをすべて取得する方法、および指定キャッシュ全体で競合をチェックする方法を示しています。

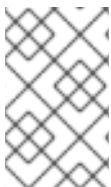
```
EmbeddedCacheManager manager = new DefaultCacheManager("example-
config.xml");
Cache<Integer, String> cache = manager.getCache("testCache");
ConflictManager<Integer, String> crm =
ConflictManagerFactory.get(cache.getAdvancedCache());

// Get All Versions of Key
Map<Address, InternalCacheValue<String>> versions = crm.getAllVersions(1);

// Process conflicts stream and perform some operation on the cache
Stream<Map<Address, InternalCacheEntry<Integer, String>>> stream =
crm.getConflicts();
stream.forEach(map -> {
    CacheEntry<Object, Object> entry = map.values().iterator().next();
    Object conflictKey = entry.getKey();
    cache.remove(conflictKey);
});

// Detect and then resolve conflicts using the configured EntryMergePolicy
crm.resolveConflicts();

// Detect and then resolve conflicts using the passed EntryMergePolicy
instance
crm.resolveConflicts((preferredEntry, otherEntries) -> preferredEntry);
```



注記

ConflictManager::getConflicts ストリームはエントリーごとに処理されますが、基盤のスプリッター (spliterator) はセグメントごとにキャッシュエントリーをレイジーにロードします。

付録A 参考資料

A.1. エクスターナライザー

A.1.1. エクスターナライザーについて

Externalizer は以下を実行できるクラスです。

- 該当するオブジェクトタイプをバイトアレイにマーシャリングします。
- バイトアレイの内容のオブジェクトタイプのインスタンスに対するマーシャリングを解除します。

エクスターナライザーは Red Hat JBoss Data Grid により使用され、ユーザーはオブジェクトタイプをどのようにシリアル化するかを指定できます。JBoss Data Grid で使用されるマーシャリングインフラストラクチャーは、JBoss Marshalling に基づいて構築され、効率的なペイロード配信を提供し、ストリームをキャッシュすることを可能にします。ストリームキャッシングを使用すると、データに複数回アクセスできますが、通常はストリームは 1 度だけ読み取ることができます。

A.1.2. 内部エクスターナライザー実装アクセス

Externalizable オブジェクトは Red Hat JBoss Data Grid エクスターナライザー実装にアクセスしないようにする必要があります。間違った使用法の例を以下に示します。

```
public static class ABCMarshallingExternalizer implements
AdvancedExternalizer<ABCMarshalling> {
    @Override
    public void writeObject(ObjectOutput output, ABCMarshalling object)
throws IOException {
        MapExternalizer ma = new MapExternalizer();
        ma.writeObject(output, object.getMap());
    }

    @Override
    public ABCMarshalling readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
        ABCMarshalling hi = new ABCMarshalling();
        MapExternalizer ma = new MapExternalizer();
        hi.setMap((ConcurrentHashMap<Long, Long>) ma.readObject(input));
        return hi;
    }
    <!-- Additional configuration information here -->
}
```

エンドユーザーエクスターナライザーは内部のエクスターナライザークラスと対話する必要がありません。正しい使用法の例を以下に示します。

```
public static class ABCMarshallingExternalizer implements
AdvancedExternalizer<ABCMarshalling> {
    @Override
    public void writeObject(ObjectOutput output, ABCMarshalling object)
throws IOException {
        output.writeObject(object.getMap());
    }
}
```

```
@Override
public ABCMarshalling readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
    ABCMarshalling hi = new ABCMarshalling();
    hi.setMap((ConcurrentHashMap<Long, Long>) input.readObject());
    return hi;
}

<!-- Additional configuration information here -->
}
```

A.2. ハッシュ領域の割り当て

A.2.1. ハッシュ領域の割り当てについて

Red Hat JBoss Data Grid は、使用可能なハッシュ領域全体の一部を各ノードに割り当てる役割があります。エントリーを格納する必要がある後続の操作中、JBoss Data Grid は関連するキーのハッシュを作成し、その部分のハッシュ領域を所有するノード上にエントリーを格納します。

A.2.2. ハッシュ領域におけるキーの検索

Red Hat JBoss Data Grid は常にアルゴリズムを使用してハッシュ領域のキーを見つけます。そのため、キーを格納するノードを手動で指定することはありません。このスキームにより、キーの所有者情報を配信しなくても、すべてのノードは特定のキーを所有するノードを判断することができます。このスキームによりオーバーヘッドの量が削減されます。さらに重要なことに、ノードの障害時に所有者情報をレプリケートする必要がないため、冗長性が向上されます。