JBoss Enterprise Application Platform 5

Cache ユーザーガイド

JBoss Enterprise Application Platform 5 向け

エディッション 5.1.2

Last Updated: 2017-10-26

JBoss Enterprise Application Platform 5 Cache ユーザーガイド

JBoss Enterprise Application Platform 5 向け
エディッション 5.1.2

Manik Surtani
manik@jboss.org

Brian Stansberry
brian.stansberry@jboss.com

Galder Zamarreño
galder.zamarreno@jboss.com

Mircea Markus
mircea.markus@jboss.com

法律上の通知

Copyright © 2011 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

本書は、 JBoss Enterprise Application Platform 5 とそのパッチリリースのキャッシュに関するユーザーガイドです。

目次

パート I. JBOSS CACHE について 6

第1章 概要 7
1.1. JBOSS CACHE とは 7
1.1.1. POJO キャッシュとは 7
1.2. 機能概要 7
1.2.1. オブジェクトのキャッシュ 7
1.2.2. ローカルおよびクラスタ化モード 8
1.2.3. クラスタ化されたキャッシュおよびトランザクション 8
1.2.4. スレッドの安全性 8
1.3. 要件 9
1.4. ライセンス 9

第2章 ユーザー API 10
2.1. API クラス 10
2.2. キャッシュのインスタンス化と起動 10
2.3. データのキャッシングと読み出し 11
2.3.1. データの編成とノード構造の使用 12
2.4. FQN クラス 12
2.5. キャッシュの停止と破棄 13
2.6. キャッシュモード 13
2.7. キャッシュリスナの追加 - キャッシュイベントに対する登録 14
2.7.1. 同期および非同期通知 16
2.8. キャッシュローダーの使用 16
2.9. エビクションポリシーの使用 17

第3章 設定 18
3.1. 設定の概要 18
3.2. CONFIGURATION の設定 18
3.2.1. XML ベース設定ファイルの解析 18
3.2.2. 設定ファイルの検証 18
3.2.3. プログラムを用いた設定 19
3.2.4. IOC フレームワークの使用 19
3.3. CONFIGURATION オブジェクトの構成 19
3.4. 動的な再設定 20
3.4.1. オプション API より設定をオーバーライドする 21

第4章 API のバッチ化 22
4.1. はじめに 22
4.2. バッチ化の設定 22
4.3. API のバッチ化 22

第5章 JBOSS CACHE のデプロイメント 23
5.1. スタンドアロンの使用 / プログラムを用いたデプロイメント 23
5.2. JBOSS MICROCONTAINER (JBOSS AS 5.X) の使用 23
5.3. JBOSS AS における JNDI への自動バインディング 25
5.4. ランタイム管理情報 25
5.4.1. JBoss Cache MBean 25
5.4.2. MBeanServer へ CacheJmxWrapper を登録 25
5.4.2.1. キャッシュインスタンスのプログラムによる登録 25
5.4.2.2. 設定インスタンスのプログラムによる登録 26
5.4.2.3. JBoss AS (JBoss AS 5.x) における JMX ベースのデプロイメント 26
5.4.3. JBoss Cache の統計 28

5.4.4. JMX 通知の受信 28
5.4.5. jconsole ユーティリティを使用してスタンドアロン環境の Cache MBean へアクセス 30

第6章 バージョン互換性および相互運用性 31
6.1. API 互換性 31
6.2. ワイヤレベルの相互運用性 31
6.3. 互換性マトリックス 31

パート II. JBOSS CACHE アーキテクチャ 32

第7章 アーキテクチャ 33
7.1. キャッシュ内のデータ構造 33
7.2. SPI インターフェース 33
7.3. ノード上におけるメソッド呼び出し 34
7.3.1. インターセプタ 35
7.3.1.1. カスタムインターセプタの作成 35
7.3.2. コマンドとビジター 35
7.3.3. InvocationContexts 36
7.4. サブシステムのマネージャ 36
7.4.1. RpcManager 36
7.4.2. BuddyManager 36
7.4.3. CacheLoaderManager 36
7.5. マーシャリングとワイヤ形式 36
7.5.1. Marshaller インターフェース 37
7.5.2. VersionAwareMarshaller 37
7.6. クラスローディングとリージョン 38

第8章 キャッシュモードとクラスタリング 39
8.1. キャッシュレプリケーションモード 39
8.1.1. ローカルモード 39
8.1.2. レプリケートされたキャッシュ 39
8.1.2.1. レプリケートされたキャッシュトランザクション 39
8.1.2.1.1. 単相コミット 39
8.1.2.1.2. 2 相コミット 40
8.1.2.2. バディレプリケーション 40
8.1.2.2.1. バディの選択 40
8.1.2.2.2. BuddyPools 41
8.1.2.2.3. フェイルオーバー 41
8.1.2.2.4. 設定 42
8.2. 無効化 42
8.3. ステート転送 42
8.3.1. ステート転送タイプ 42
8.3.2. バイトアレイおよびストリーミングベースのステート転送 42
8.3.3. 完全および部分的ステート転送 43
8.3.4. 一時 (「インメモリ」) および永続ステート転送 44
8.3.5. ステート転送の設定 45

第9章 キャッシュローダー 46
9.1. CACHELOADER インターフェースとライフサイクル 46
9.2. 設定 47
9.2.1. シングルトンストアの設定 49
9.3. 同梱される実装 50
9.3.1. ファイルシステムベースのキャッシュローダー 50
9.3.2. 他のキャッシュに委譲するキャッシュローダー 51

9.3.3. JDBCCacheLoader 51
9.3.3.1. JDBCCacheLoader の設定 51
9.3.3.1.1. テーブルの設定 51
9.3.3.1.2. DataSource 52
9.3.3.1.3. JDBC ドライバ 52
9.3.3.1.4. c3p0 接続プーリング 52
9.3.3.1.5. 設定例 53
9.3.4. S3CacheLoader 54
9.3.4.1. Amazon S3 ライブラリ 55
9.3.4.2. 設定 55
9.3.5. TcpDelegatingCacheLoader 56
9.3.6. キャッシュローダーの変換 57
9.4. キャッシュパッシベーション 57
9.4.1. パッシベーションを無効にした場合と有効にした場合のキャッシュローダーの挙動 58
9.5. ストラテジ 59
9.5.1. ローカルキャッシュとストア 59
9.5.2. すべてのキャッシュが同じストアを共有するレプリケートされたキャッシュ 59
9.5.3. ストアを持つキャッシュが 1 つしかないレプリケートされたキャッシュ 60
9.5.4. 各キャッシュが独自のストアを持つレプリケートされたキャッシュ 60
9.5.5. 階層的キャッシュ 61
9.5.6. 複数のキャッシュローダー 62

第10章 エビクション ... 64
10.1. 構造 64
10.1.1. 統計の収集 64
10.1.2. エビクトするノードの判定 64
10.1.3. ノードをエビクトする方法 64
10.1.4. エビクションスレッド 65
10.2. エビクションリージョン 65
10.2.1. 常駐ノード 65
10.3. エビクションの設定 66
10.3.1. 基本設定 66
10.3.2. プログラムを用いた設定 66
10.4. 同梱されるエビクションポリシー 67
10.4.1. LRUAlgorithm - 最長時間未使用 67
10.4.2. FIFOAlgorithm - 先入れ先出し 67
10.4.3. MRUAlgorithm - 最も最近使用 67
10.4.4. LFUAlgorithm - 使用頻度が最低 68
10.4.5. ExpirationAlgorithm 68
10.4.6. ElementSizeAlgorithm - ノードにあるキーと値のペアの数を基にしたエビクション 69

第11章 トランザクションと並行性 ... 70
11.1. 同時アクセス 70
11.1.1. MVCC (Multi-Version Concurrency Control) 70
11.1.1.1. MVCC の概念 70
11.1.1.2. MVCC 実装 70
11.1.1.2.1. 分離レベル 71
11.1.1.2.2. 同時書き込みと書き込みのスキュー 71
11.1.1.3. ロッキングの設定 72
11.1.2. 楽観的および悲観的ロッキングスキーム 72
11.2. JTA サポート 72

パート III. JBOSS CACHE の設定に関する参考資料 ... 74

第12章 設定に関する参考資料 ... 75
12.1. XML 設定ファイルの例 75
12.1.1. XML の検証 79
12.2. 設定ファイルのクイックリファレンス 79

第13章 JMX の参照 ... 111
13.1. JBOSS CACHE の統計 111
13.2. JMX MBEAN 通知 113

付録A 改訂履歴 ... 115

パート I. JBOSS CACHE について

本項では、 開発者がプロジェクトで JBoss Cache をすぐに使用するために必要な事柄について取り上げ、 概念や API の概要、設定、 デプロイメント情報について説明します。

第1章 概要

1.1. JBOSS CACHE とは

JBoss Cache はツリー構造のクラスタ化されたトランザクションキャッシュです。 メモリにある頻繁にアクセスされるデータをキャッシュするためスタンドアロンの非クラスタ化環境で使用できるため、 JTA の互換性やエビクション、 永続性などの「エンタープライズ」機能を提供しながらデータの取り出しや計算のボトルネックを排除することができます。

JBoss Cache はクラスタ化されたキャッシュでもあるため、 ステートをレプリケートするためにクラスタで使用すると、 高度なフェイルオーバーを提供できます。 無効化やバディレプリケーションなどを含むさまざまなレプリケーションモードがサポートされ、 ネットワーク通信は同期か非同期になります。

クラスタモードで使用すると、 キャッシュは高可用性やフォールトトレランス、 ロードバランシングをカスタムアプリケーションやフレームワークに構築する効率的なメカニズムとなります。 例えば、 JBoss Application Server や Red Hat Enterprise Application Platform は JBoss Cache を広範囲で使用し、 HTTP や EJB セッションなどのサービスをクラスタ化します。 また、 JPA の分散エンティティキャッシュも提供します。

1.1.1. POJO キャッシュとは

POJO キャッシュとはコアの JBoss Cache API の拡張です。 POJO キャッシュは次のような追加機能を提供します。

- レプリケーションや永続化の後もオブジェクト参照を維持。
- 変更されたオブジェクトフィールドのみがレプリケートされる細かなレプリケーション。
- POJO にはクラスタ化されたとしてアノテーションが付けられる「API なし」のクラスタリングモデル。

ユーザーガイドや FAQ、 チュートリアルなどを含む POJO キャッシュのドキュメントはすべて JBoss Cache のドキュメントウェブサイトで閲覧できます。 そのため、 本書では POJO キャッシュについてこれ以上説明しません。

1.2. 機能概要

1.2.1. オブジェクトのキャッシュ

JBoss Cache は、 簡単な Java オブジェクトであるデータをキャッシュに置くことができる単純明解な API を提供します。 選択された設定オプションを基にすると、 データは以下の 1 つまたはすべてに該当します。

- 効率的でスレッドセーフな読み出し行うためインメモリにキャッシュされます。
- クラスタの一部またはすべてのキャッシュインスタンスへレプリケートされます。
- ディスクやリモートのインメモリキャッシュクラスタ (「ファーキャッシュ」) へ永続化されます。
- メモリの残量が少なくなるとメモリでガベッジコレクションが実行され、 ステートを損失しないようディスクへパッシベートされます。

更に、 JBoss Cache はエンタープライズクラスの機能を複数提供します。

- JTA トランザクションに参加できる機能 (Java EE 対応トランザクションマネージャのほとんどで可能)。
- JMX コンソールに付随し、 キャッシュのステートに関するランタイム統計を提供する機能。
- クライアントコードがリスナを付随できるようにし、 キャッシュイベント上で通知を受け取り可能にする機能。
- レプリケーションを効率化するため、 キャッシュ操作をバッチにグループ化できる機能。

1.2.2. ローカルおよびクラスタ化モード

キャッシュは単一のルートを持つツリーとして構成されます。 ツリーの各ノードには、 キーと値のペアのストアとなるマップが含まれています。 キャッシュされるオブジェクトは、 **`java.io.Serializable`** を実装することのみが要件となります。

JBoss Cache はローカルキャッシュまたはレプリケートされたキャッシュとなります。 ローカルキャッシュは、 キャッシュが作成された JVM の範囲内のみで存在でき、 レプリケートされたキャッシュは、 同じクラスタ内の他のキャッシュすべてまたは一部への変更を伝播します。 クラスタはネットワーク上の異なるホストか、 単一ホスト上の異なる JVM にまたがります。

1.2.3. クラスタ化されたキャッシュおよびトランザクション

キャッシュのオブジェクトに変更が加えられ、 その変更がトランザクションのコンテキストで行われた場合、 変更のレプリケーションはトランザクションが正常に終了するまで延期されます。 すべての変更は、 呼出側のトランザクションに関連付けられたリストに保持されます。 トランザクションがコミットされると、 変更がレプリケートされます。 コミットされない場合は、 ロールバック上で変更をローカルで取り消し、 変更リストを放棄します。 この結果、 ネットワークトラフィックとオーバーヘッドがゼロになります。 例えば、 呼出側が 100 個の変更を行いトランザクションをロールバックすると、 何もレプリケートされずネットワークトラフィックは発生しません。

呼出側に関連付けられたトランザクションやバッチがない場合、 変更は即座にレプリケートされます。 例えば、 前述の例では、 各変更に対して 100 のメッセージがブロードキャストされます。 そのため、 バッチやトランザクションなしでの実行は、 JDBC における自動コミットが有効な状態 (各操作が自動的に即座にコミットされる) での実行と同様であると考えることができます。

JBoss Cache は追加設定なしでほとんどの一般的なトランザクションマネージャと使用することができます。 また、 カスタムのトランザクションマネージャルックアップを作成できる API も提供します。

上記の説明はバッチに対しても同様で、 同じように動作します。

1.2.4. スレッドの安全性

キャッシュは完全にスレッドセーフです。 MVCC (Multi-versioned Concurrency Control) を使用し、 高度な同時実行性を維持しながらリーダーとライター間におけるスレッドの安全性を確保します 。JBoss Cache で使用される特定の MVCC 実装により、 リーダースレッドにはロックや同期化されたブロックが完全になくなるため、 読み取り中心のアプリケーションのパフォーマンスを向上します。 また、 マルチコア CPU アーキテクチャ向けに調整されたライタースレッドの最新の CAS (コンペアアンドスワップ) 技術を使用するパフォーマンスに優れたカスタムのロック実装も使用します。

JBoss Cache 3.x 以降、 MVCC (Multi-versioned Concurrency Control) はデフォルトのロッキングスキームとなっています。これ以前のバージョンの楽観的ロックや悲観的ロックのスキームも使用可能ですが、 MVCC の導入により廃止され、 今後のリリースでは削除される予定です。 そのため、 廃止されたこれらのロックスキームは使用しないようにしてください。

JBoss Cache の MVCC 実装は READ_COMMITTED および REPEATABLE_READ 分離レベルのみをサポートし、 これらレベルのデータベース相当に対応します。 MVCC の詳細は *11章 トランザクションと並行性* の項を参照してください。

1.3. 要件

JBoss Cache には Java 5.0 (または 5.0 以降) 互換の仮想マシンとライブラリのセットが必要です。 JBoss Cache は Sun の JDK 5.0 および JDK 6 上で開発されテストされています。

Java 5.0 以外に、 JBoss Cache は最低でも JGroups と Apache の commons-logging へ依存します。 JBoss Cache には、 追加設定なしで実行するために必要なすべての依存ライブラリやオプション機能に対する複数のオプション jar が同梱されています。

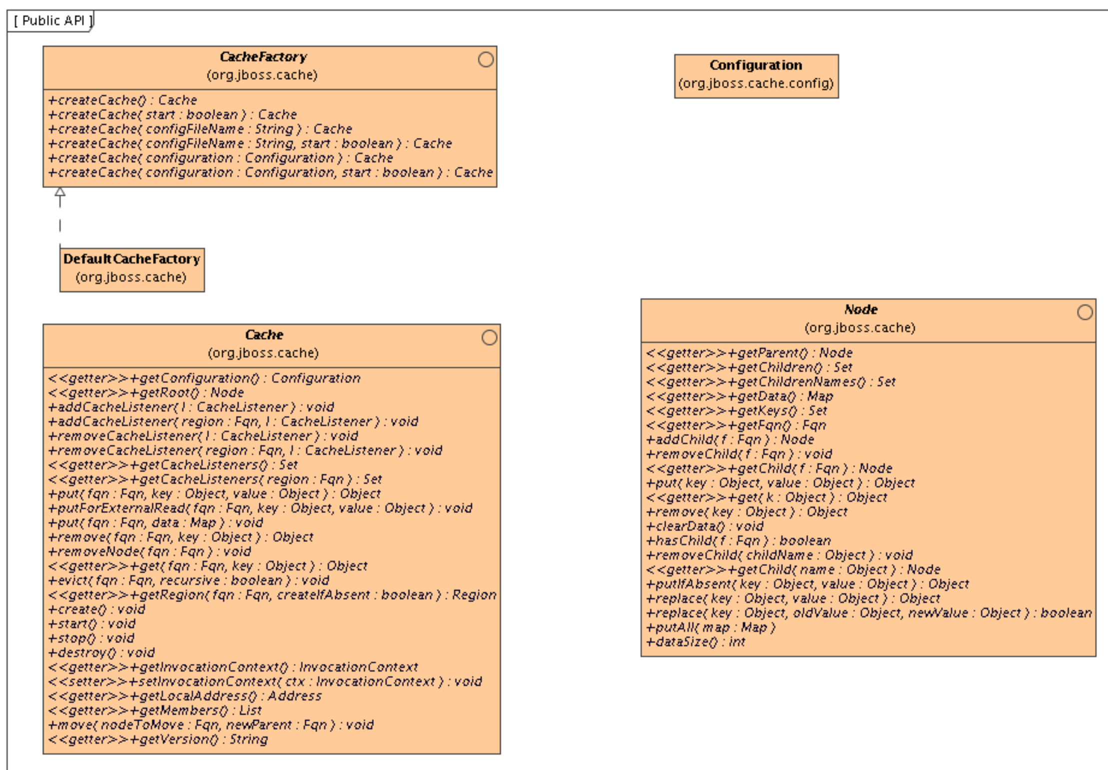
1.4. ライセンス

JBoss Cache は企業や OEM が利用しやすい OSI 認証の LGPL ライセンスを使用したオープンソースプロジェクトです。 商業的な開発サポートや実稼働サポート、 JBoss Cache のトレーニングについては JBoss by Red Hat を参照してください。

第2章 ユーザー API

2.1. API クラス

Cache インターフェースは JBoss Cache と対話する主要なメカニズムです。 **CacheFactory** を使用し構築され、 任意に起動されます。 **CacheFactory** により、 **Configuration** オブジェクトまたは XML ファイルを作成することができます。 キャッシュはデータをノードで構成されるツリー構造に編成します。 **Cache** への参照を取得した後、 参照を使用してツリー構造の **Node** オブジェクトをルックアップし、 データをツリーに保存することができます。



上図は一般的な API メソッドの一部のみを表しています。 上記のインターフェースに関する javadoc を確認することが API について学ぶ最良の方法です。 主なポイントの一部を次に説明します。

2.2. キャッシュのインスタンス化と起動

Cache のインスタンスは **CacheFactory** よりのみ作成可能です。 これが、 古い **TreeCache** クラスのインスタンスを直接インスタンス化できた JBoss Cache 1.x と異なる点になります。

CacheFactory は **Cache** 作成のためのオーバーロードされたメソッドを複数提供しますが、 すべてのメソッドは基本的に同じことを行います。

- メソッドパラメータとしてパスするか、 XML の内容を分析して構築することで **Configuration** へアクセスできるようにします。 XML の内容は、 提供された入力ストリームやクラスパスまたはファイルシステムの場所から取得できます。 **Configuration** の取得に関する詳細は 3章*設定* を参照してください。

- **Cache** をインスタンス化し、 **Configuration** への参照と共に提供します。

- 任意でキャッシュの **create()** メソッドと **start()** メソッドを呼び出します。

デフォルトの設定値を使用してキャッシュを作成起動する最も簡単な例は次の通りです。

```
   CacheFactory factory = new DefaultCacheFactory();
   Cache cache = factory.createCache();
```

この例で、 クラスパス上の設定ファイルを検索し解析するよう **CacheFactory** に伝えます。

```
   CacheFactory factory = new DefaultCacheFactory();
   Cache cache = factory.createCache("cache-configuration.xml");
```

この例で、 ファイルよりキャッシュを設定しますが、 設定要素をプログラムを用いて変更します。 そのため、 キャッシュを起動しないようファクトリに伝えます。

```
   CacheFactory factory = new DefaultCacheFactory();
   Cache cache = factory.createCache("/opt/configurations/cache-
configuration.xml", false);
   Configuration config = cache.getConfiguration();
   config.setClusterName(this.getClusterName());

   // Have to create and start cache before using it
   cache.create();
   cache.start();
```

2.3. データのキャッシングと読み出し

次に、 **Cache** API を使用してキャッシュの **Node** へアクセスし、 そのノードに対して簡単な読み書きを行います。

```
   // Let's get a hold of the root node.
   Node rootNode = cache.getRoot();

   // Remember, JBoss Cache stores data in a tree structure.
   // All nodes in the tree structure are identified by Fqn objects.
   Fqn peterGriffinFqn = Fqn.fromString("/griffin/peter");

   // Create a new Node
   Node peterGriffin = rootNode.addChild(peterGriffinFqn);

   // let's store some data in the node
   peterGriffin.put("isCartoonCharacter", Boolean.TRUE);
   peterGriffin.put("favoriteDrink", new Beer());

   // some tests (just assume this code is in a JUnit test case)
   assertTrue(peterGriffin.get("isCartoonCharacter"));
   assertEquals(peterGriffinFqn, peterGriffin.getFqn());
   assertTrue(rootNode.hasChild(peterGriffinFqn));

   Set keys = new HashSet();
   keys.add("isCartoonCharacter");
   keys.add("favoriteDrink");

   assertEquals(keys, peterGriffin.getKeys());

   // let's remove some data from the node
   peterGriffin.remove("favoriteDrink");
```

```
assertNull(peterGriffin.get("favoriteDrink");

// let's remove the node altogether
rootNode.removeChild(peterGriffinFqn);

assertFalse(rootNode.hasChild(peterGriffinFqn));
```

Cache インスタンスは、便宜上 「**Fqn** クラス」 を引数として取る put、 get、 remove 操作も公開します。

```
Fqn peterGriffinFqn = Fqn.fromString("/griffin/peter");

cache.put(peterGriffinFqn, "isCartoonCharacter", Boolean.TRUE);
cache.put(peterGriffinFqn, "favoriteDrink", new Beer());

assertTrue(peterGriffin.get(peterGriffinFqn, "isCartoonCharacter"));
assertTrue(cache.getRootNode().hasChild(peterGriffinFqn));

cache.remove(peterGriffinFqn, "favoriteDrink");

assertNull(cache.get(peterGriffinFqn, "favoriteDrink");

cache.removeNode(peterGriffinFqn);

assertFalse(cache.getRootNode().hasChild(peterGriffinFqn));
```

2.3.1. データの編成とノード構造の使用

ノードは名前が付けられた論理的にグループされたデータと見なされます。 特定の人物やアカウントの情報など、 単一のデータ記録に対してデータを保存するためにノードを使用します。 キャッシュの全側面 (ロッキング、 キャッシュローディング、 レプリケーション、 エビクション) はノードごとに発生することを覚えておく必要があります。 そのため、 単一ノードに保存してグループ化を行うと、 単一のアトミック単位として処理されます。

2.4. FQN クラス

前項の例では **Fqn** クラスが使用されていました。 このクラスについてより詳しく説明しましょう。

完全修飾名 (Fqn) は、 キャッシュのツリー構造内にある特定場所へのパスを表す名前の一覧をカプセル化します。 一覧の要素は通常**String** ですが、 あらゆる **Object** が可能で異なるタイプを混合することもできます。

パスはキャッシュ内のノードへの絶対パス (ルートノードへ相対的) か相対パスになります。 **Fqn** を使用する各 API 呼び出しのドキュメントを読むと、 API が予期する **Fqn** が相対か絶対であるかが分かります。

Fqn クラスはさまざまなファクトリメソッドを提供します。 提供されるファクトリメソッドについては javadoc を参照してください。 次は Fqn の作成に使用される最も一般的な方法を表しています。

```
// Create an Fqn pointing to node 'Joe' under parent node 'Smith'
// under the 'people' section of the tree

// Parse it from a String
Fqn abc = Fqn.fromString("/people/Smith/Joe/");
```

```
// Here we want to use types other than String
Fqn acctFqn = Fqn.fromElements("accounts", "NY", new Integer(12345));
```

注意:

```
Fqn f = Fqn.fromElements("a", "b", "c");
```

は下記と同じです。

```
Fqn f = Fqn.fromString("/a/b/c");
```

2.5. キャッシュの停止と破棄

特にキャッシュがクラスタ化され、 JGroups チャネルを使用した場合、 キャッシュを使い終えたらキャッシュを停止し、 破棄することが推奨されます。 キャッシュを停止し破棄することで、 ネットワークソケットやメンテナンススレッドなどのリソースを適切にクリーンアップすることができます。

```
cache.stop();
cache.destroy();
```

stop() が呼び出されたキャッシュを再起動するには、 **start()** を新たに呼び出します。 同様に、 **destroy()** が呼び出されたキャッシュを再作成するには、 **create()** を新たに呼び出します (その後、 **start()**を呼び出すと再起動されます)。

2.6. キャッシュモード

技術的には API の一部ではありませんが、 キャッシュを操作するために設定される**モード**は **put** や **remove** 操作のクラスタ全体の動作に影響するため、 ここで簡単に説明します。

JBoss Cache のモードは **org.jboss.cache.config.Configuration.CacheMode** 列挙にて表され、 以下によって構成されます。

- **LOCAL** - ローカルの非クラスタ化キャッシュです。 ローカルキャッシュはクラスタには参加せず、 クラスタ内の他のキャッシュとも通信しません。
- **REPL_SYNC** - 同期レプリケーションです。 レプリケートされたキャッシュは、 クラスタ内の他のキャッシュへすべての変更をレプリケートします。 同期レプリケーションでは変更がレプリケートされ、 レプリケーションの確認が受信されるまで呼出側がブロックします。
- **REPL_ASYNC** - 非同期レプリケーションです。 前述の REPL_SYNC と同様に、 レプリケートされたキャッシュはクラスタ内の他のキャッシュにすべての変更をレプリケートします。 非同期のため、 レプリケーションの確認が受信されるまで呼出側はブロックしません。
- **INVALIDATION_SYNC** - キャッシュがレプリケーションではなく無効化に対して設定されている場合、 キャッシュでデータが変更される度に、 データが陳腐化したためメモリよりエビクトする必要があるというメッセージをクラスタ内の他のキャッシュが受け取ります。 これにより、 レプリケーションのオーバーヘッドを削減しながら、 リモートキャッシュ上の陳腐化データを無効にすることができます。
- **INVALIDATION_ASYNC** - 上記と同じですが、 無効化のメッセージが非同期にブロードキャストされます。

キャッシュモードが動作に与える影響の詳細は 8章キャッシュモードとクラスタリング を参照してください。 キャッシュモードの設定方法は 3章設定 を参照してください。

2.7. キャッシュリスナの追加 - キャッシュイベントに対する登録

JBoss Cache はキャッシュイベントで通知を登録する便利なメカニズムを提供します。

```
Object myListener = new MyCacheListener();
cache.addCacheListener(myListener);
```

登録されたリスナを削除したりクエリする同様のメソッドも存在します。 詳細は **Cache** インターフェースに関する javadoc を参照してください。

基本的に、 **@CacheListener** アノテーションが付けられていれば、 あらゆるパブリッククラスをリスナとして使用できます。 また、 クラスはメソッドレベルアノテーション (**org.jboss.cache.notifications.annotation** パッケージ内) の 1 つが付けられたメソッドを 1 つ以上持たなければなりません。 このようにアノテーション付けされたメソッドはパブリックとなる必要があり、 無効の戻りタイプを持たなければなりません。 また、 タイプ **org.jboss.cache.notifications.event.Event** の単一のパラメータかサブタイプ の 1 つを許可しなければなりません。

- **@CacheStarted** - このアノテーションが付けられたメソッドはキャッシュ起動時に通知を受け取ります。 メソッドは **CacheStartedEvent** より割り当て可能なパラメータタイプを許可する必要があります。
- **@CacheStopped** - このアノテーションが付けられたメソッドはキャッシュ停止時に通知を受け取ります。 メソッドは **CacheStoppedEvent** より割り当て可能なパラメータタイプを許可する必要があります。
- **@NodeCreated** - このアノテーションが付けられたメソッドはノード作成時に通知を受け取ります。 メソッドは **NodeCreatedEvent** より割り当て可能なパラメータタイプを許可する必要があります。
- **@NodeRemoved** - このアノテーションが付けられたメソッドはノードの削除時に通知を受け取ります。 メソッドは **NodeRemovedEvent** より割り当て可能なパラメータタイプを許可する必要があります。
- **@NodeModified** - このアノテーションが付けられたメソッドはノードが変更された時に通知を受け取ります。 メソッドは **NodeModifiedEvent** より割り当て可能なパラメータタイプを許可する必要があります。
- **@NodeMoved** - このアノテーションが付けられたメソッドはノードが移動した時に通知を受け取ります。 メソッドは **NodeMovedEvent** より割り当て可能なパラメータタイプを許可する必要があります。
- **@NodeVisited** - このアノテーションが付けられたメソッドはノードの起動時に通知を受け取ります。 メソッドは **NodeVisitedEvent** より割り当て可能なパラメータタイプを許可する必要があります。
- **@NodeLoaded** - このアノテーションが付けられたメソッドは、 **CacheLoader** よりノードがロードされた時に通知を受け取ります。 メソッドは **NodeLoadedEvent** より割り当て可能なパラメータタイプを許可する必要があります。
- **@NodeEvicted** - このアノテーションが付けられたメソッドはノードがメモリからエビクトされた時に通知を受け取ります。 メソッドは **NodeEvictedEvent** より割り当て可能なパラメー

タタイプを許可する必要があります。

- **@NodeInvalidated** - このアノテーションが付けられたメソッドは、 リモートの無効化イベントによってノードがメモリからエビクトされた時に通知を受け取ります。 メソッドは **NodeInvalidatedEvent** より割り当て可能なパラメータタイプを許可する必要があります。

- **@NodeActivated** - このアノテーションが付けられたメソッドはノードがアクティベートされた時に通知を受け取ります。 メソッドは **NodeActivatedEvent** より割り当て可能なパラメータタイプを許可する必要があります。

- **@NodePassivated** - このアノテーションが付けられたメソッドはノードがパッシベートされた時に通知を受け取ります。 メソッドは **NodePassivatedEvent** より割り当て可能なパラメータタイプを許可する必要があります。

- **@TransactionRegistered** - ノードが登録済みのトランザクションマネージャに **javax.transaction.Synchronization** を登録した時に、 このアノテーションが付けられたメソッドは通知を受け取ります。 メソッドは **TransactionRegisteredEvent** より割り当て可能なパラメータタイプを許可する必要があります。

- **@TransactionCompleted** - ノードが登録済みのトランザクションマネージャからコミットまたはロールバック呼び出しを受け取った時に、 このアノテーションが付けられたメソッドは通知を受け取ります。 メソッドは **TransactionCompletedEvent** より割り当て可能なパラメータタイプを許可する必要があります。

- **@ViewChanged** - このアノテーションが付けられたメソッドはクラスタのグループ構造が変更した時に通知を受け取ります。 メソッドは **ViewChangedEvent** より割り当て可能なパラメータタイプを許可する必要があります。

- **@CacheBlocked** - ステート転送イベントに対してキャッシュ操作をブロックするようクラスタが要求した時に、 このアノテーションが付けられたメソッドは通知を受け取ります。 メソッドは **CacheBlockedEvent** より割り当て可能なパラメータタイプを許可する必要があります。

- **@CacheUnblocked** - ステート転送イベント後にキャッシュ操作をブロックしないようクラスタが要求した時に、 このアノテーションが付けられたメソッドは通知を受け取ります。 メソッドは **CacheUnblockedEvent** より割り当て可能なパラメータタイプを許可する必要があります。

- **@BuddyGroupChanged** - バディグループのクラスタ脱退や、 新しいより密接なバディの参加などが原因でノードがバディグループを変更した時に、 このアノテーションが付けられたメソッドは通知を受け取ります。 メソッドは **BuddyGroupChangedEvent** より割り当て可能なパラメータタイプを許可する必要があります。

メソッドへ渡される対象や渡されるタイミングについての詳細は、 アノテーションや **Event** サブタイプに関する javadoc を参照してください。

例:

```
@CacheListener
public class MyListener
{

   @CacheStarted
   @CacheStopped
   public void cacheStartStopEvent(Event e)
```

```
        {
           switch (e.getType())
           {
              case CACHE_STARTED:
                 System.out.println("Cache has started");
                 break;
              case CACHE_STOPPED:
                 System.out.println("Cache has stopped");
                 break;
           }
        }

        @NodeCreated
        @NodeRemoved
        @NodeVisited
        @NodeModified
        @NodeMoved
        public void logNodeEvent(NodeEvent ne)
        {
           log("An event on node " + ne.getFqn() + " has occured");
        }
     }
```

2.7.1. 同期および非同期通知

デフォルトでは通知はすべて同期で、 イベントを生成した呼出側のスレッド上で発生します。 そのため、 キャッシュリスナの実装が長期実行タスクでスレッドを遅延しないようにしてください。 また、 **CacheListener.sync()** 属性を **false** に設定し、 呼出側のスレッドで通知されないようにすることもできます。 スレッドプールとブロッキングキューサイズの調整については 表12.13「**<listeners />** 要素」 を参照してください。

2.8. キャッシュローダーの使用

キャッシュローダーは JBoss Cache で重要な役割を果たします。 キャッシュローダーは、 ディスクやリモートキャッシュクラスタへノードを永続し、 キャッシュがメモリ不足になるとパッシベーションを実行できるようにします。 また、 キャッシュローダーにより、 JBoss Cache は永続ストレージよりインメモリステートを事前ロードできる「ウォームスタート」を実行できます。 JBoss Cache には複数のキャッシュローダー実装が同梱されています。

- **org.jboss.cache.loader.FileCacheLoader** - ディスクへデータを永続化する基本的なファイルシステムベースのキャッシュローダーです。 非トランザクションで大変パフォーマンスに優れていますが、 大変簡単なソリューションとなります。 主にテスト向けに使用され、 実稼働における使用は推奨されません。

- **org.jboss.cache.loader.JDBCCacheLoader** - JDBC 接続を使用してデータを保存します。 内部プール内 (c3p0 プーリングライブラリを使用) や設定された DataSource より接続を作成し維持することが可能です。 この CacheLoader が接続するデータベースの場所はローカルかリモートになります。

- **org.jboss.cache.loader.BdbjeCacheLoader** - Oracle の BerkeleyDB ファイルベースのトランザクションデータベースを使用してデータを永続化します。 トランザクションで大変パフォーマンス的に優れていますが、 ライセンスが制限される可能性があります。

- **org.jboss.cache.loader.JdbmCacheLoader** - BerkeleyDB の代替となるオープンソースのキャッシュローダーです。

- **org.jboss.cache.loader.tcp.TcpCacheLoader** - TCP ソケットを使用し、 「ファーキャッシュ」パターンを用いて、 リモートクラスタへデータを「永続化」します。

- **org.jboss.cache.loader.ClusteredCacheLoader** - クラスタの他のノードがステートをクエリされる「読み取り専用」のキャッシュローダーとして使用されます。 完全なステート転送は高負荷になるため、 ステートをレイジーにロードしたい場合に便利です。

これらキャッシュローダーの説明や詳細設定、 調整については 9章 *キャッシュローダー* を参照してください。

2.9. エビクションポリシーの使用

エビクションポリシーはキャッシュローダーに対応します。 キャッシュがメモリ不足にならないようにし、 キャッシュが満杯になってきたら個別のスレッドで実行されるエビクションアルゴリズムがインメモリステートをエビクトし、 メモリの空き領域を確保するようにするために必要です。 エビクションポリシーをキャッシュローダーに設定すると、 必要時にステートをキャッシュローダーより読み取ることができます。

エビクションポリシーはリージョンごとに設定できるため、 キャッシュのサブツリーによって異なるエビクションが設定されることがあります。 JBoss Cache には複数のエビクションポリシーが同梱されています。

- **org.jboss.cache.eviction.LRUPolicy** - しきい値に達すると、 最も長い間使用されていないノードをエビクトするエビクションポリシーです。

- **org.jboss.cache.eviction.LFUPolicy** - しきい値に達すると、 最も使用頻度が低いノードをエビクトするエビクションポリシーです。

- **org.jboss.cache.eviction.MRUPolicy** - しきい値に達すると、 最も最近使用されたノードをエビクトするエビクションポリシーです。

- **org.jboss.cache.eviction.FIFOPolicy** - しきい値に達すると、 先入れ先出しキューを作成して最も古いノードをエビクトするエビクションポリシーです。

- **org.jboss.cache.eviction.ExpirationPolicy** - 各ノードに設定された失効時間を基にエビクトするノードを選択するエビクションポリシーです。

- **org.jboss.cache.eviction.ElementSizePolicy** - ノードに保持されているキーと値のペア数を基にエビクトするノードを選択するエビクションポリシーです。

設定の詳細やカスタムエビクションポリシーの実装については、 10章 *エビクション* を参照してください。

第3章 設定

3.1. 設定の概要

org.jboss.cache.config.Configuration クラス (および 「**Configuration** オブジェクトの構成」) は、 **Cache** の設定やアーキテクチャに関する要素すべて (キャッシュローダー、 エビクションポリシーなど) をカプセル化する Java Bean です。

Configuration は多くのプロパティを公開します。 これらのプロパティについては本書の 「設定ファイルのクイックリファレンス」 の項に要約があり、 今後の章で詳しく説明されています。 本書における設定オプションの説明では、 **Configuration** クラスやそのクラスのコンポーネント部の 1 つが設定オプションの簡単なプロパティセッタ/ゲッタを公開することを前提としています。

3.2. CONFIGURATION の設定

「キャッシュのインスタンス化と起動」 で説明した通り、 **Cache** を作成できるようにするには、 **Configuration** オブジェクトと共に **CacheFactory** を提供するか、 XML より **Configuration** を解析するために使用するファイル名または入力ストリームと共に **CacheFactory** を提供する必要があります。 この方法については以降の項で説明します。

3.2.1. XML ベース設定ファイルの解析

XML ファイルを使用するのが JBoss Cache ファイルを設定する最も便利な方法です。 JBoss Cache ディストリビューションには、 一般的なユースケースに対する多くの設定ファイルが同梱されています。 これらのファイルを最初に使用し、 ニーズに合わせて調整することが推奨されます。

キャッシュ設定が LOCAL モードを実行するのが設定 XML ファイルの最も簡単な例で、 次のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>

<jbosscache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:jboss:jbosscache-core:config:3.1">
</jbosscache>
```

このファイルは分離レベルやロックの取得、 タイムアウトなどに賢明なデフォルトを使用します。 より完全な他のサンプル XML ファイルは本書の 「XML 設定ファイルの例」 項にあります。 また、 「設定ファイルのクイックリファレンス」 にはさまざまなオプションの説明が記載されています。

3.2.2. 設定ファイルの検証

デフォルトでは、 JBoss Cache は XML スキーマに対して XML 設定ファイルを検証し、 設定が無効であると例外をスローします。 この動作をオーバーライドするには **-Djbosscache.config.validate=false** JVM パラメータを使用します。 また、 **-Djbosscache.config.schemaLocation=url** パラメータを使用して検証に使用する独自のスキーマを指定することもできます。

デフォルトでは、**jbosscache-core.jar** または **http://www.jboss.org/jbosscache/jbosscache-config-3.0.xsd** に含まれる JBoss Cache

設定スキーマに対して設定ファイルが検証されます。 作成した設定ファイルが適切で有効であるようにするため、 このスキーマと共にほとんどの XML 編集ツールを使用することが可能です。

3.2.3. プログラムを用いた設定

上記の XML ベース設定の他にも、 **Configuration** やそのコンポーネントによって公開される簡単なプロパティミュテータを使用してプログラムを用いて **Configuration** を構築することができます。 構築されると、 **Configuration** オブジェクトが JBoss Cache のデフォルトとして事前設定され、 そのままクイックスタートとして使用することも可能です。

```
Configuration config = new Configuration();
config.setTransactionManagerLookupClass(
  GenericTransactionManagerLookup.class.getName()
  );
config.setIsolationLevel(IsolationLevel.READ_COMMITTED);
config.setCacheMode(CacheMode.LOCAL);
config.setLockAcquisitionTimeout(15000);

CacheFactory factory = new DefaultCacheFactory();
Cache cache = factory.createCache(config);
```

上記のような比較的簡単な設定でもかなり長いプログラムとなります。 そのため、 XML ベース設定の使用が好まれます。 しかし、 アプリケーションがプログラムを用いた設定を必要とする場合、 プログラムを用いてデフォルトから数ヶ所の変更を行うため **Configuration** オブジェクトへアクセスしたりエビクションリージョンを追加するなど、 ほとんどの属性に対して XML ベース設定を使用しない理由はありません。

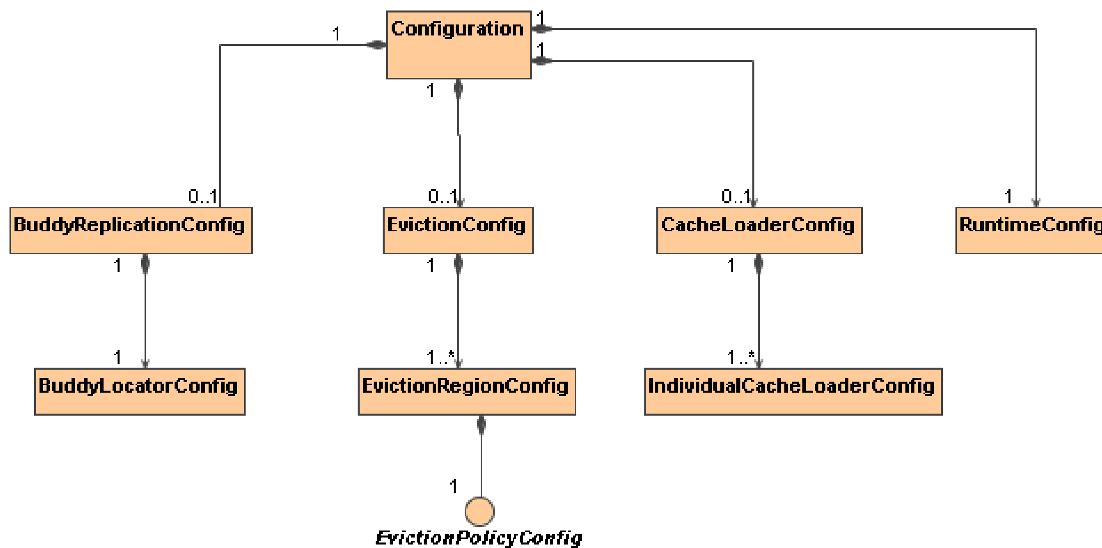
キャッシュが実行されている場合、 **@Dynamic** アノテーションが付けられている設定値以外の設定値はプログラムを用いて変更できないことがあります。 動的なプロパティは動的であると 「設定ファイルのクイックリファレンス」 の表に明記されています。 動的でないプロパティを変更しようとすると、 **ConfigurationException** が発生します。

3.2.4. IOC フレームワークの使用

Configuration クラスとその 「**Configuration** オブジェクトの構成」 は、 簡単なセッタやゲッタよりすべての設定要素を公開する Java Bean です。 そのため、 Sping や Google Guice、 JBoss Microcontainer などの信頼できる IOC フレームワークは、 フレームワーク独自形式の XML ファイルより **Configuration** を構築することができるはずです。 この例については 「JBoss Microcontainer (JBoss AS 5.x) の使用 」 の項を参照してください。

3.3. CONFIGURATION オブジェクトの構成

Configuration は複数のサブオブジェクトで構成されています。



Configuration のコンポーネントの概要は次の通りです。 各コンポーネントに関する設定のより詳細な説明は、 javadoc およびリンクされた章を参照してください。

- **Configuration** : 階層のトップレベルオブジェクトです。 本書の 「設定ファイルのクイックリファレンス」 の項に記載されている設定プロパティを公開します。
- **BuddyReplicationConfig** : 「バディレプリケーション」 が使用されている場合のみ関係します。 一般的なバディレプリケーションの設定オプションです。 次が含まれていなければなりません。
- **BuddyLocatorConfig** : 使用される **BuddyLocator** 実装の実装固有の設定オブジェクトです。 公開される設定要素は **BuddyLocator** 実装の要件によって異なります。
- **EvictionConfig** : 10章 *エビクション* が使用されている場合のみ関係します。 一般的なエビクションの設定オプションです。 最低でも次が 1 つ含まれなければなりません。
- **EvictionRegionConfig** : エビクションリージョンごとに 1 つ割り当てられます。 リージョンの名前付けなどを行います。 次が含まれなければなりません。
- **EvictionAlgorithmConfig** : 使用される **EvictionAlgorithm** 実装の実装固有の設定オブジェクトです。 **EvictionAlgorithm** 実装の要件によって公開される設定要素が異なります。
- **CacheLoaderConfig** : 9章 *キャッシュローダー* が使用される場合のみ関係します。 一般的なキャッシュローダーの設定オプションです。 最低でも次が 1 つ含まれていなければなりません。
- **IndividualCacheLoaderConfig** : 使用される **CacheLoader** 実装の実装固有の設定オブジェクトです。 **CacheLoader** 実装の要件により公開される設定要素が異なります。
- **RuntimeConfig** : キャッシュのランタイム環境に関するキャッシュクライアントの特定情報を公開します(「バディレプリケーション」 が使用される場合、 バディレプリケーショングループのメンバーシップなど)。 また、 JTA **TransactionManager** や JGroups **ChannelFactory** など必要な外部サービスのキャッシュへ直接挿入できるようにします。

3.4. 動的な再設定

キャッシュの実行中に「一部」オプションの設定を動的に変更することは、 プログラムを用いて実行しているキャッシュより **Configuration** オブジェクトを取得し、 値を変更することでサポートされます。 例は次の通りです。

```
Configuration liveConfig = cache.getConfiguration();
liveConfig.setLockAcquisitionTimeout(2000);
```

動的に変更可能なオプションの完全一覧は 「設定ファイルのクイックリファレンス」 の項を参照してください。 動的でない設定を変更しようとすると、 **org.jboss.cache.config.ConfigurationException** がスローされます。

3.4.1. オプション API より設定をオーバーライドする

オプション API は呼び出しごとにキャッシュの特定の動作をオーバーライドできるようにします。 これには、 **org.jboss.cache.config.Option** のインスタンスを作成します。 **Option** オブジェクトでオーバーライドしたいオプションを設定し、 キャッシュでメソッドを呼び出す前に **InvocationContext** へ渡します。

データの読み取り中に書き込みロックを強制する例は次の通りです (トランザクションで使用すると、データベースの SELECT FOR UPDATE に似たセマンティックを提供します)。

```
      // first start a transaction
  cache.getInvocationContext().getOptionOverrides().setForceWriteLock(true);
      Node n = cache.getNode(Fqn.fromString("/a/b/c"));
      // make changes to the node
      // commit transaction
```

REPL_SYNC キャッシュで put 呼び出しのレプリケーションを抑制する例は次の通りです。

```
      Node node = cache.getChild(Fqn.fromString("/a/b/c"));
  cache.getInvocationContext().getOptionOverrides().setLocalOnly(true);
      node.put("localCounter", new Integer(2));
```

使用可能なオプションについては、 **Option** クラスに関する javadoc を参照してください。

第4章 API のバッチ化

4.1. はじめに

JBoss Cache 3.x に導入された API のバッチ化は、 JTA トランザクションから独立して呼び出しがレプリケートされる方法をバッチ化するためのメカニズムを目的としています。

この機能は、進行中の JTA トランザクションよりも細かな範囲内のレプリケーション呼び出しをバッチ化したい場合に便利です。

4.2. バッチ化の設定

バッチ化を使用するには、 キャッシュ設定で呼び出しのバッチ化を有効にする必要があります。 **Configuration** オブジェクト上で行う場合は次のように設定します。

```
Configuration.setInvocationBatchingEnabled(true);
```

XML ファイルで行う場合は次のように設定します。

```
<invocationBatching enabled="true"/>
```

デフォルトでは呼び出しのバッチ化は無効になっています。 バッチ化を使用するようトランザクションマネージャを定義する必要はありません。

4.3. API のバッチ化

キャッシュがバッチ化を使用するよう設定したら、 **Cache** 上で **startBatch()** と **endBatch()** を呼び出してバッチ化を使用します。 例は次の通りです。

```
   Cache cache = getCache();

   // not using a batch
   cache.put("/a", "key", "value"); // will replicate immediately

   // using a batch
   cache.startBatch();
   cache.put("/a", "key", "value");
   cache.put("/b", "key", "value");
   cache.put("/c", "key", "value");
   cache.endBatch(true); // This will now replicate the modifications
since the batch was started.

   cache.startBatch();
   cache.put("/a", "key", "value");
   cache.put("/b", "key", "value");
   cache.put("/c", "key", "value");
   cache.endBatch(false); // This will "discard" changes made in the batch
```

第5章 JBOSS CACHE のデプロイメント

5.1. スタンドアロンの使用 / プログラムを用いたデプロイメント

「キャッシュのインスタンス化と起動」 と 「**Configuration** の設定」 の章の説明通り、 スタンドアロンの Java プログラムで使用する場合は、 **CacheFactory** と **Configuration** インスタンス (または XML ファイル) を使用してキャッシュをインスタンス化することのみが必要となります。

アプリケーションサーバーで実行されているアプリケーションがアプリケーションサーバーのデプロイメント機能に依存せずにプログラムを用いてキャッシュをデプロイしたい場合、 同じ手法を使用することができます。 **javax.servlet.ServletContextListener** よりウェブアプリケーションがキャッシュをデプロイするのが例の 1 つとなります。

作成後、 異なるアプリケーションコンポーネント間でキャッシュインスタンスを共有することができます。 これには、 Sping や JBoss Microcontainer などの IOC コンテナを使用するか、 JNDI へバインドします。 または、 キャッシュへの静的参照を保持するようにします。

キャッシュをデプロイした後に JMX で管理インターフェースをキャッシュに公開したい場合は、「MBeanServer へ CacheJmxWrapper を登録」 を参照してください。

5.2. JBOSS MICROCONTAINER (JBOSS AS 5.X) の使用

JBoss AS は AS 5 よりファイル名が **-jboss-beans.xml** で終わるファイルのデプロイメントによる POJO サービスのデプロイメントをサポートするようになりました。 POJO サービスは実装に Plain Old Java Object を使用します。 Plain Old Java Object とは、 特別なインターフェースの実装や特定のスーパークラスの拡張が必要ない単純な Java Bean のことです。 **Cache** は POJO サービスで、 **Configuration** のすべてのコンポーネントも POJO になります。 そのため、 必然的にこのような方法でキャッシュをデプロイすることになります。

キャッシュのデプロイメントは、 JBoss AS のコアを形成する JBoss Microcontainer を使用して行われます。 JBoss Microcontainer は洗練された IOC フレームワークで、 Spring と似ています。 基本的に、 **-jboss-beans.xml** ファイルは POJO サービスを構成するさまざまな Bean をどのように組み合わせるかを IOC フレームワークに伝えます。

Configuration コンポーネントによって公開される設定可能オプション毎に、 ゲッタ/セッタを設定クラスに定義する必要があります。 これは、 対応するプロパティが設定された時に JBoss Microcontainer が典型的な IOC の方法でこれらのメソッドを呼び出せるようにするため必要となります。

サーバーの **lib** ディレクトリに **jbosscache-core.jar** ライブラリと **jgroups.jar** ライブラリが存在するようしてください。 通常、 JBoss AS を **all** 設定で使用する場合に必要となります。 キャッシュ設定に応じて任意の jar (**jdbm.jar** など) が必要となります。

-beans.xml ファイルの例は次のようになります。 インストールされた JBoss AS 5 の **server/all/deploy** ディレクトリに他の例があります。

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">
```

```
        <!-- Externally injected services -->
        <property name="runtimeConfig">
           <bean class="org.jboss.cache.config.RuntimeConfig">
              <property name="transactionManager">
                 <inject bean="jboss:service=TransactionManager"
                         property="TransactionManager"/>
              </property>
              <property name="muxChannelFactory"><inject
bean="JChannelFactory"/></property>
           </bean>
        </property>

        <property name="multiplexerStack">udp</property>

        <property name="clusterName">Example-EntityCache</property>

        <property name="isolationLevel">REPEATABLE_READ</property>

        <property name="cacheMode">REPL_SYNC</property>

        <property name="stateRetrievalTimeout">15000</property>

        <property name="syncReplTimeout">20000</property>

        <property name="lockAcquisitionTimeout">15000</property>

        <property name="exposeManagementStatistics">true</property>
     </bean>

     <!-- Factory to build the Cache. -->
     <bean name="DefaultCacheFactory"
class="org.jboss.cache.DefaultCacheFactory">
        <constructor factoryClass="org.jboss.cache.DefaultCacheFactory"
                     factoryMethod="getInstance" />
     </bean>

     <!-- The cache itself -->
     <bean name="ExampleCache" class="org.jboss.cache.Cache">

        <constructor factoryMethod="createCache">
            <factory bean="DefaultCacheFactory"/>
            <parameter class="org.jboss.cache.config.Configuration"><inject
bean="ExampleCacheConfig"/></parameter>
            <parameter class="boolean">false</parameter>
        </constructor>

     </bean>

</deployment>
```

上記の構文に関する詳細は JBoss Microcontainer のドキュメント を参照してください。 基本的に、 各 **bean** 要素はオブジェクトを表し、 **`Configuration`** と 「**`Configuration`** オブジェクトの構成」 を作成するために使用されます。 **`DefaultCacheFactory`** Bean はキャッシュを構築し、 概念的に 「キャッシュのインスタンス化と起動」 の章と同じことを実行します。

上記の例で、 **`RuntimeConfig`** オブジェクトの使用を確認してみてください。

TransactionManager や JGroups の **ChannelFactory** などのマイクロコンテナが可視できる外部リソースは、 依存関係が **RuntimeConfig** に挿入されます。 ここでは、 AS における他のデプロイメント記述子の一部では参照された Bean が既に記述されていることが前提となっています。

5.3. JBOSS AS における JNDI への自動バインディング

本書の執筆時点ではこの機能は使用できません。 使用可能になった時点で使用方法を wiki ページに追加します。

5.4. ランタイム管理情報

JBoss Cache にはキャッシュ機能を提供する JMX MBean が含まれ、JBoss Cache はキャッシュ操作を分析するために使用できる統計を提供します。また、JBoss Cache は JMX 管理ツールを使用してキャッシュイベントを処理を行う MBean の通知としてブロードキャストできます。

5.4.1. JBoss Cache MBean

JBoss Cache は使用している環境の JMX サーバーへ登録できる MBean を提供し、 JMX よりキャッシュインスタンスへアクセスできるようにします。 この MBean は StandardMBean の **org.jboss.cache.jmx.CacheJmxWrapper** で、 MBean インターフェースは **org.jboss.cache.jmx.CacheJmxWrapperMBean** になります。 この MBean は次を実行するために使用できます。

- 基礎の **Cache** への参照を取得する。
- 基礎の **Cache** 上で、 create、 start、 stop、 destroy のライフサイクル操作を呼び出す。
- キャッシュの現在のステートに関する詳細を検査する (ノード数やロック情報など)。
- キャッシュの設定に関する詳細を確認し、 キャッシュの起動後に変更可能な設定項目を変更する。

詳細は **CacheJmxWrapperMBean** の javadoc を参照してください。

CacheJmxWrapper が登録されている場合、 JBoss Cache は他の内部コンポーネントやサブシステムの MBean も提供します。 これらの MBean は表示されるサブシステムに関連する統計を取得し、 公開するために使用されます。 また、 階層的に **CacheJmxWrapper** MBean に関連付けられ、 この関係を反映するサービス名が付けられます。 たとえば、 **jboss.cache:service=TomcatClusteringCache** インスタンスのリプリケーションインターセプタ MBean は **jboss.cache:service=TomcatClusteringCache,cache-interceptor=ReplicationInterceptor** という名前のサービスからアクセス可能になります。

5.4.2. MBeanServer へ CacheJmxWrapper を登録

確実に **CacheJmxWrapper** が JMX に登録されるようにする最良の方法は、 キャッシュをデプロイする方法によって異なります。

5.4.2.1. キャッシュインスタンスのプログラムによる登録

Cache を作成し、 **JmxRegistrationManager** コンストラクタへ渡す方法が最も簡単です。

```
CacheFactory factory = new DefaultCacheFactory();
// Build but don't start the cache
```

```
    // (although it would work OK if we started it)
    Cache cache = factory.createCache("cache-configuration.xml");

    MBeanServer server = getMBeanServer(); // however you do it
    ObjectName on = new ObjectName("jboss.cache:service=Cache");

    JmxRegistrationManager jmxManager = new JmxRegistrationManager(server,
cache, on);
    jmxManager.registerAllMBeans();

    ... use the cache

    ... on application shutdown

    jmxManager.unregisterAllMBeans();
    cache.stop();
```

5.4.2.2. 設定インスタンスのプログラムによる登録

Configuration オブジェクトをビルドし、 **CacheJmxWrapper** へ渡すこともできます。 ラッパーが **Cache** を構築します。

```
    Configuration config = buildConfiguration(); // whatever it does

    CacheJmxWrapperMBean wrapper = new CacheJmxWrapper(config);
    MBeanServer server = getMBeanServer(); // however you do it
    ObjectName on = new ObjectName("jboss.cache:service=TreeCache");
    server.registerMBean(wrapper, on);

    // Call to wrapper.create() will build the Cache if one wasn't injected
    wrapper.create();
    wrapper.start();

    // Now that it's built, created and started, get the cache from the
wrapper
    Cache cache = wrapper.getCache();

    ... use the cache

    ... on application shutdown

    wrapper.stop();
    wrapper.destroy();
```

5.4.2.3. JBoss AS (JBoss AS 5.x) における JMX ベースのデプロイメント

CacheJmxWrapper は POJO であるため、 マイクロコンテナによって問題なく作成されます。 **CacheJmxWrapper** が Bean を JMX で登録するようにするのがポイントです。 これには、 **CacheJmxWrapper** Bean 上で **org.jboss.aop.microcontainer.aspects.jmx.JMX** アノテーションを指定します。

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">

      ... build up the Configuration

   </bean>

   <!-- Factory to build the Cache. -->
   <bean name="DefaultCacheFactory"
class="org.jboss.cache.DefaultCacheFactory">
      <constructor factoryClass="org.jboss.cache.DefaultCacheFactory"
                   factoryMethod="getInstance" />
   </bean>

   <!-- The cache itself -->
   <bean name="ExampleCache" class="org.jboss.cache.CacheImpl">

      <constructor factoryMethod="createnewInstance">
          <factory bean="DefaultCacheFactory"/>
          <parameter><inject bean="ExampleCacheConfig"/></parameter>
          <parameter>false</parameter>
      </constructor>

   </bean>

   <!-- JMX Management -->
   <bean name="ExampleCacheJmxWrapper"
class="org.jboss.cache.jmx.CacheJmxWrapper">


<annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.cach
e:service=ExampleTreeCache",

exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
                         registerDirectly=true)</annotation>

      <constructor>
          <parameter><inject bean="ExampleCache"/></parameter>
      </constructor>

   </bean>

</deployment>
```

「MBeanServer へ CacheJmxWrapper を登録」 の項での説明通り、 **Configuration** が提供されれば **CacheJmxWrapper** は **Cache** を構築、 作成、 起動することができます。 **CacheFactory** を作成するために必要なボイラープレート XML を保存するため、 マイクロコンテナを使用する場合はこの方法が推奨されます。

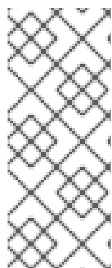
```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">
```

```
    <!-- First we create a Configuration object for the cache -->
    <bean name="ExampleCacheConfig"
       class="org.jboss.cache.config.Configuration">

       ... build up the Configuration

    </bean>

    <bean name="ExampleCache" class="org.jboss.cache.jmx.CacheJmxWrapper">

       <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
         (name="jboss.cache:service=ExampleTreeCache",
         exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
         registerDirectly=true)</annotation>

       <constructor>
           <parameter><inject bean="ExampleCacheConfig"/></parameter>
       </constructor>

    </bean>

</deployment>
```

5.4.3. JBoss Cache の統計

JBoss Cache はインターセプタや他のコンポーネントの統計を収集し、 MBean のセットを通じて公開します。 統計の収集はデフォルトで有効になっています。 特定のキャッシュインスタンスに対して統計の収集を無効にしたい場合は、 **`Configuration.setExposeManagementStatistics()`** セッタより無効にします。 統計のほとんどは **`CacheMgmtInterceptor`** によって提供されるため、 この MBean が最重要になります。 パフォーマンス上の理由からすべての統計を無効にする場合は、 **`Configuration.setExposeManagementStatistics(false)`** を設定し、 キャッシュ起動時にキャッシュのインターセプタスタックに **`CacheMgmtInterceptor`** が含まれないようにします。

`CacheJmxWrapper` が JMX に登録されると、 ラッパーは統計を公開する各インターセプタやコンポーネントに対して MBean がJMX に登録されるようにします。



注記

`CacheJmxWrapper` が JMX に登録されていないと、 インターセプタ MBean も登録されません。 JBoss Cache 1.4 リリースには、 **`MBeanServer`** の「ディスカバリ」を試行し、 自動的にインターセプタ MBean を登録するコードが含まれていました。 JBoss Cache 2.x では、 このような JMX 環境の「ディスカバリ」はライブラリのキャッシングの適正範囲外であると判断されたため、 この機能が削除されました。

この後、 管理ツールが MBean にアクセスし、 統計を調べることができます。 JMX によって使用できる統計については、 「JBoss Cache の統計」 の項を参照してください。

5.4.4. JMX 通知の受信

JBoss Cache ユーザーは、 リスナーを登録して 「キャッシュリスナの追加 - キャッシュイベントに対する登録」 の章で説明したキャッシュイベントを受信できます。 また、 別の方法としてユーザーはキャッシュの管理情報インフラストラクチャを使用して JMX 通知よりこれらのイベントを受信することもできます。 **`CacheJmxWrapper`** に対して **`NotificationListener`** を登録すると、 キャッシュイベントは通知としてアクセス可能になります。

CacheJmxWrapper より受信可能な通知一覧は、「JMX MBean 通知」 の JMX 通知に関する項を参照してください。

下記は、 JBoss AS 環境の実行時にプログラムを用いてキャッシュ通知を受信する方法の例になります。 この例では、 クライアントはフィルタを使用して興味のあるイベントを指定しています。

```
   MyListener listener = new MyListener();
   NotificationFilterSupport filter = null;

   // get reference to MBean server
   Context ic = new InitialContext();
   MBeanServerConnection server =
(MBeanServerConnection)ic.lookup("jmx/invoker/RMIAdaptor");

   // get reference to CacheMgmtInterceptor MBean
   String cache_service = "jboss.cache:service=TomcatClusteringCache";
   ObjectName mgmt_name = new ObjectName(cache_service);

   // configure a filter to only receive node created and removed events
   filter = new NotificationFilterSupport();
   filter.disableAllTypes();
   filter.enableType(CacheNotificationBroadcaster.NOTIF_NODE_CREATED);
   filter.enableType(CacheNotificationBroadcaster.NOTIF_NODE_REMOVED);

   // register the listener with a filter
   // leave the filter null to receive all cache events
   server.addNotificationListener(mgmt_name, listener, filter, null);

   // ...

   // on completion of processing, unregister the listener
   server.removeNotificationListener(mgmt_name, listener, filter, null);
```

前の例で使用された単純な通知リスナー実装を以下に示します。

```
   private class MyListener implements NotificationListener, Serializable
   {
      public void handleNotification(Notification notification, Object
handback)
      {
         String message = notification.getMessage();
         String type = notification.getType();
         Object userData = notification.getUserData();

         System.out.println(type + ": " + message);

         if (userData == null)
         {
            System.out.println("notification data is null");
         }
         else if (userData instanceof String)
         {
            System.out.println("notification data: " + (String) userData);
         }
         else if (userData instanceof Object[])
         {
```

```
                Object[] ud = (Object[]) userData;
                for (Object data : ud)
                {
                   System.out.println("notification data: " +
data.toString());
                }
             }
             else
             {
                System.out.println("notification data class: " +
userData.getClass().getName());
             }
          }
       }
```

JBoss Cache 管理実装は、 クライアントが MBean 通知を受信するよう登録された後でのみキャッシュイベントをリッスンします。 クライアントが通知のために登録されると、 MBean はキャッシュリスナとして MBean 自体を除外します。

5.4.5. `jconsole` ユーティリティを使用してスタンドアロン環境の **Cache MBean** へアクセス

JBoss Cache MBean は、JBoss JMX Console などの MBean サーバーインターフェースを提供するアプリケーションサーバーでキャッシュインスタンスを実行する時に簡単にアクセスされます。 サーバーの MBean コンテナで実行している MBean にアクセスする方法については、 サーバーのドキュメントを参照してください。

また、 非サーバー環境で実行している場合でも、 JDK の **`jconsole`** ツールを使用すると JBoss Cache MBean はアクセス可能です。 アプリケーションサーバー外部でスタンドアロンキャッシュを実行している場合は、 次のようにキャッシュの MBean にアクセスできます。

1. キャッシュが実行される JVM の起動時に、 システムプロパティ **`-Dcom.sun.management.jmxremote`** を設定します。

2. JVM が実行されたら、 JDK の **`/bin`** ディレクトリにある **`jconsole`** ユーティリティを起動します。

3. ユーティリティがロードされたら、 実行している JVM を選択し接続することができます。 JBoss Cache MBean は MBeans パネルにあります。

`jconsole` ユーティリティは、 JBoss Cache インスタンスを実行している JVM への接続時にキャッシュ通知のリスナーとして自動的に登録されます。

第6章 バージョン互換性および相互運用性

6.1. API 互換性

メジャーバージョン内の JBoss Cache リリースは互換性および相互運用性を維持するようになっています。 ここで言う互換性とは、 jar を置き換えるだけであるバージョンから別のバージョンへアプリケーションをアップグレードできることを意味します。 相互運用性とは、 2 つの異なるバージョンの JBoss Cache が同じクラスタで使用される場合に、 バージョン間でレプリケーションやステート転送メッセージを交換できることを意味します。 ただし、 相互運用性を維持するには、 すべてのノードで同じ JGroups のバージョンを使用する必要があります。 ほとんどの場合で、 JBoss Cache のバージョンによって使用される JGroups のバージョンはアップグレードが可能です。

そのため、 JBoss Cache 2.x.x はそれ以前の 1.x.x バージョンとは API やバイナリの互換性を維持しません。 逆に、 JBoss Cache 2.1.x は 2.0.x と API やバイナリの互換性を維持します。

JBoss Cache 3.x が 2.x とバイナリと API 両方の互換性を維持するよう最大の努力を尽くしていますが、 廃止されたメソッドやクラス、 設定ファイルを使用しないようクライアントコードを更新することが推奨されます。

6.2. ワイヤレベルの相互運用性

設定パラメータ **`Configuration.setReplicationVersion()`** を使用し、 キャッシュ間通信のワイヤフォーマットを制御することができます。 古いリリースと通信する際は、 効率の良い新しいプロトコルから「互換性」を持つバージョンへ戻すことができます。 このメカニズムにより、 相互運用性を保持しつつ効率的なワイヤフォーマットを使用して JBoss Cache を向上することができます。

6.3. 互換性マトリックス

JBoss Cache のウェブサイトに掲載されている互換性マトリックスには、 異なるバージョンの JBoss Cache や JGroups、 JBoss Application Server に関する情報が記載されています。

パート II. JBOSS CACHE アーキテクチャ

本項では、 上級キャッシュ機能の使用やキャッシュの拡張または改良、 プラグインの作成などを行いたい開発者や、 JBoss Cache の仕組みについて深く学びたい開発者向けに詳しく JBoss Cache アーキテクチャについて説明します。

第7章 アーキテクチャ

7.1. キャッシュ内のデータ構造

Cache は、 ツリー構造の **Node** インスタンスの集合によって構成されます。 各 **Node** には、 キャッシュされるデータオブジェクトが保持される **Map** が含まれています。 構造はグラフではなく数学的なツリーです。 各 **Node** の親は 1 つのみで、 ルートノードは一定の完全修飾名である **Fqn.ROOT** によって表されます。

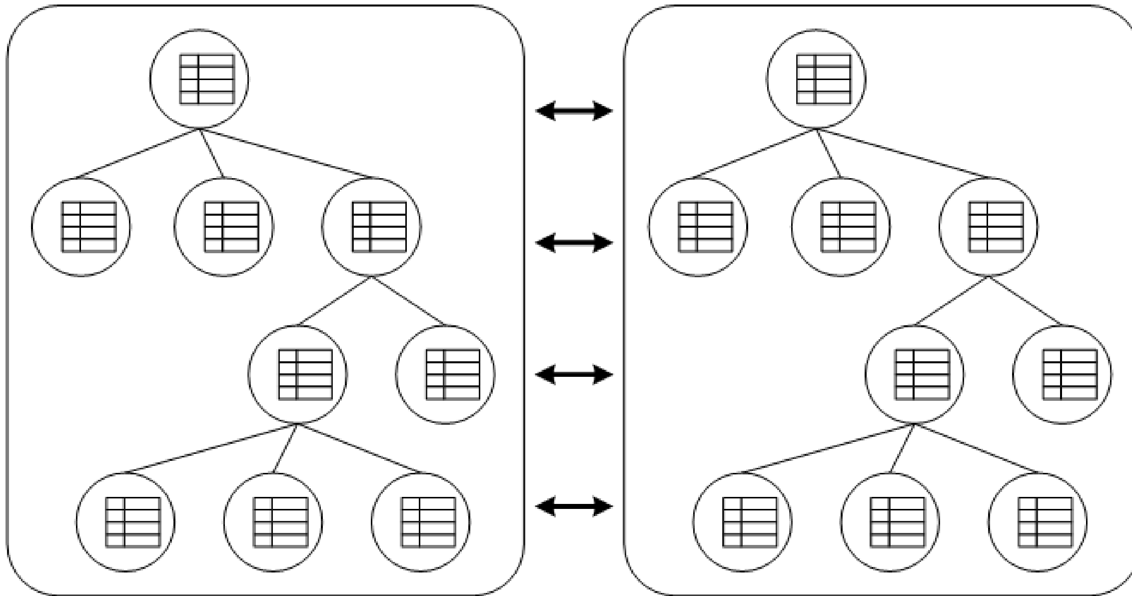


図7.1 ツリー構造のデータ

上図では、 各ボックスは JVM を表しています。 個々の JVM にある 2 つのキャッシュがお互いにデータをレプリケートしているのが分かります。

キャッシュインスタンスでの変更 (2章ユーザー *API* を参照) は別のキャッシュへレプリケートされます。 1 つのクラスタで 3 つ以上のキャッシュを設定できます。 トランザクションの設定に応じて、 このレプリケーションはコミット時の変更後かトランザクションの終了時に実行されます。 新しいキャッシュが作成されると、 開始時に既存キャッシュの 1 つから内容を任意に取得します。

7.2. SPI インターフェース

Cache インターフェースや **Node** インターフェースの他にも、 JBoss Cache は JBoss Cache 内部で更なる制御を提供するより強力な **CacheSPI** および **NodeSPI** インターフェースを公開します。 これらのインターフェースは一般的な使用を目的としたものではありませんが、 JBoss Cache を拡張および強化したいユーザーや、 カスタムの **Interceptor** インスタンスや **CacheLoader** インスタンスを作成したいユーザー向けのインターフェースです。

図7.2 SPI インターフェース

`CacheSPI` インターフェースを作成することはできませんが、 インターフェース上の **`setCache(CacheSPI cache)`** メソッドにより **`Interceptor`** 実装や **`CacheLoader`** 実装へ挿入されます。 **`CacheSPI`** は **`Cache`** を拡張するため、 基本的な API のすべての機能も使用することができます。

同様に、 **`NodeSPI`** インターフェースも作成することはできませんが、 前述の通り取得された **`CacheSPI`** 上で操作を実行すると取得されます。 例えば、 **`Cache.getRoot() : Node`** は **`CacheSPI.getRoot() : NodeSPI`** としてオーバーライドされます。

インターフェースの継承は、 今後維持が保証されるコントラクトではないため、 **`Cache`** や **`Node`** を SPI へ直接キャストすることは推奨されないだけでなく、 不適切であると見なされます。 逆に、 公開された API は維持が保証されます。

7.3. ノード上におけるメソッド呼び出し

キャッシュは基本的にノードの集合であるため、 キャッシュ全体または個別ノード上で操作が呼び出された時にこれらのノードへクラスタリングや永続性、 エビクションなどのアスペクトを適用する必要があります。 これをモジュラーで拡張可能に実行するには、 インターセプタチェーンを使用します。 チェーンは複数のインターセプタで構成され、 各インターセプタがアスペクトや特定機能を追加します。 キャッシュの作成時に使用する設定に応じてチェーンが構築されます。

NodeSPI は、 インターセプタスタックを通過せずに直接ノード上で操作するメソッド (**xxxDirect()** メソッドなど) の一部を提供します。 このようなメソッドを使用すると、 ロッキングやレプリケーションなど適用する必要があるキャッシュのアスペクトに影響することをプラグインの作成者は念頭に置いて作業する必要があります。 完全に理解していない場合は、 このようなメソッドは使用しない方がよいでしょう。

7.3.1. インターセプタ

JBoss Cache は **DataContainer** 実装のコアデータ構造で、 データ構造前のインターセプタを使用して機能が実装されます。 **CommandInterceptor** は抽象クラスで、 インターセプタ実装が拡張します。

CommandInterceptor が **Visitor** インターフェースを実装します。 そのため、 強い型付けにてコマンドを変更することが可能です。 次項でビジターやコマンドの詳細を説明します。

インターセプタのチェーン全体でコマンドを送る **InterceptorChain** クラスでインターセプタ実装はチェーンされます。 特別なインターセプタ **CallInterceptor** は常にこのチェーンの最後に位置し、 コマンドの **process()** メソッドを呼び出してチェーンへ渡されたコマンドを呼び出します。

JBoss Cache には、 異なる動作の側面を表す複数のインターセプタが同梱されています。 その一部は次の通りです。

- **TxInterceptor** - 進行中のトランザクションを検索し、 トランザクションマネージャへ登録して同期イベントへ参加します。
- **ReplicationInterceptor** - RpcManager クラスを使用してクラスタ全体でステートをレプリケートします。
- **CacheLoaderInterceptor** - 要求されたデータがメモリにない場合に永続ストアよりデータをロードします。

キャッシュインスタンスに対して設定されたインターセプタチェーンを取得し検査するには、 **CacheSPI.getInterceptorChain()** を呼び出します。 **CacheSPI.getInterceptorChain()** は、 コマンドが対応する順で順序付けされるインターセプタの **List** を返します。

7.3.1.1. カスタムインターセプタの作成

特定のアスペクトや機能を追加するカスタムインターセプタを作成するには、 阻止したいコマンドに応じて **CommandInterceptor** を拡張し、 関係する **visitXXX()** メソッドをオーバーライドします。 **PrePostProcessingCommandInterceptor** や **SkipCheckChainedInterceptor** など、 代わりに拡張できる他の抽象インターセプタもあります。 提供される追加機能についての詳細は関連する javadoc を参照してください。

Cache.addInterceptor() メソッドを使用してインターセプタチェーンにカスタムインターセプタを追加する必要があります。 メソッドの詳細は javadoc を参照してください。

XML によるカスタムインターセプタの追加もサポートされています。 詳細は 12章*設定に関する参考資料* を参照してください。

7.3.2. コマンドとビジター

JBoss Cache は内部でコマンドとビジターのパターンを使用し、 API 呼び出しを実行します。 キャッシュインターフェース上でメソッドが呼び出される度に **Cache** インターフェースを実装する **CacheInvocationDelegate** が **VisitableCommand** のインスタンスを作成し、 このコマンドをイ

ンターセプタのチェーンへ送ります。 **Visitor** を実装するインターセプタは目的の **VisitableCommand** を処理し、 行動をコマンドに追加することができます。

各コマンドには、 使用されるパラメータや **process()** メソッドにカプセル化される処理動作など、 実行されたコマンドのすべての情報が含まれます。 例えば、 **Cache.removeNode()** が呼び出されると **RemoveNodeCommand** が作成されインターセプタチェーンへ渡され、 データ構造からノードを削除するために必要な情報が **RemoveNodeCommand.process()** に含まれます。

コマンドはビジット可能であるだけでなく、 レプリケートも可能です。 JBoss Cache のマーシャラーは、 コマンドを効率的にマーシャリングし、 JGroups に応じて内部 RPC メカニズムを使用しリモートキャッシュインスタンス上で呼び出す方法を認識します。

7.3.3. InvocationContexts

InvocationContext は単一呼び出しの期間に対する中間ステートを保持し、 インターセプタチェーンの最初に位置する **InvocationContextInterceptor** によって設定および破棄されます。

名前の通り、 **InvocationContext** は単一のキャッシュメソッド呼び出しに関連するコンテキストの情報を保持します。 コンテキストの情報には、 関連する **javax.transaction.Transaction** や **org.jboss.cache.transaction.GlobalTransaction**、 メソッド呼び出し元 (**InvocationContext.isOriginLocal()**)、 「オプション API より設定をオーバーライドする」、 ロックされたノードに関する情報などが含まれます。

InvocationContext を取得するには **Cache.getInvocationContext()** を呼び出します。

7.4. サブシステムのマネージャ

一部のアスペクトや機能は複数のインターセプタによって共有されます。 さまざまなインターセプタが使用できるよう、 このようなアスペクトや機能はマネージャへカプセル化され、 **CacheSPI** インターフェースによって使用できるようになります。

7.4.1. RpcManager

このクラスは、 リモートキャッシュへの全 RPC 呼び出しに対し JGroups チャネルより実行された呼び出しに関与し、 使用される JGroups チャネルをカプセル化します。

7.4.2. BuddyManager

このクラスはバディグループを管理し、 グループ編成リモート呼び出しを実行してキャッシュのクラスタを小さなサブグループに編成します。

7.4.3. CacheLoaderManager

キャッシュローダを設定します。 このクラスは個別の **CacheLoader** インスタンスを **SingletonStoreCacheLoader** や **AsyncCacheLoader** などの委譲クラスにラッピングしたり、 **ChainingCacheLoader** を使用して **CacheLoader** をチェーンに追加します。

7.5. マーシャリングとワイヤ形式

旧バージョンの JBoss Cache は、 レプリケーション中に **ObjectOutputStream** へ書き込みを行い、 キャッシュデータをネットワークに書き込みました。 成熟度が高いマーシャリングフレームワークの導入により、 この方法は JBoss Cache 1.x.x シリーズのリリースで徐々に廃止されました。 JBoss Cache

2.x.xシリーズでは、 この方法がデータストリームにオブジェクトを書き込むメカニズムとして唯一正式サポートされ、 推奨されています。

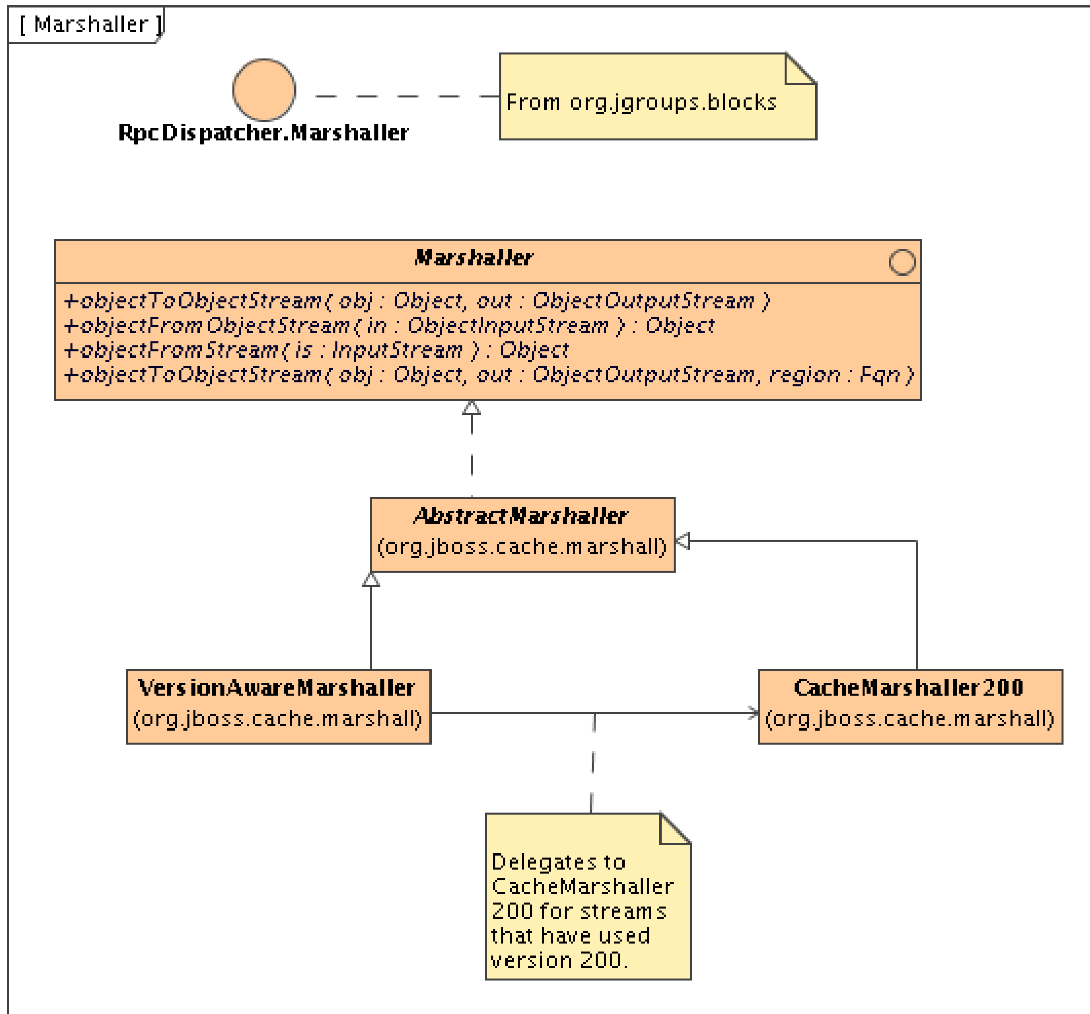


図7.3 Marshaller インターフェース

7.5.1. Marshaller インターフェース

Marshaller インターフェースは JGroups より **RpcDispatcher.Marshaller** を拡張します。 このインターフェースには、 委譲する **VersionAwareMarshaller** と明確な **CacheMarshaller300** の 2 つの主な実装があります。

マーシャラーを取得するには、 **CacheSPI.getMarshaller()** を呼び出し、 **VersionAwareMarshaller** をデフォルトとして設定します。 また、 **Marshaller** インターフェースを実装したり **AbstractMarshaller** クラスを拡張したりして独自のマーシャラーを作成し、 **Configuration.setMarshallerClass()** セッタを使用して設定に追加することもできます。

7.5.2. VersionAwareMarshaller

名前が示す通り、 このマーシャラーは書き込み中にバージョン **short** をストリームの最初に追加し、 同様の **VersionAwareMarshaller** インスタンスがバージョン short を読み取り、 呼び出しを委譲する特定のマーシャラー実装を認識できるようにします。 例えば、 **CacheMarshaller200** は JBoss Cache 2.0.x のマーシャラーになります。 JBoss Cache 3.0.x には向上されたワイヤプロトコルが含ま

れる **CacheMarshaller300** が同梱されています。 **VersionAwareMarshaller** はマイナーリリース間のワイヤプロトコルの互換性を維持するようにしますが、 マイナーおよびマクロリリース間のワイヤプロトコルを調整し改良する柔軟性も持ち合わせています。

7.6. クラスローディングとリージョン

アプリケーションサーバーのステートをクラスタ化するために使用すると、 アプリケーションにデプロイされたアプリケーションは、 固有のオブジェクトのインスタンスをレプリケーションが必要なキャッシュ (または **HttpSession** オブジェクト) に置く傾向にあります。 アプリケーションサーバーが個別の **ClassLoader** インスタンスをデプロイされた各アプリケーションに割り当て、 アプリケーションサーバーの **ClassLoader** によって JBoss Cache ライブラリが参照されるようにすることが一般的です。

このようなクラスローダーよりオブジェクトをマーシャルしたりアンマーシャルできるようにするには、 リージョンという概念を使用します。リージョンとは、 一般的なクラスローダーを共有するキャッシュの部分のことです (リージョンには他の使用方法もあります。 10章*エビクション* を参照してください)。

リージョンは **Cache.getRegion(Fqn fqn, boolean createIfNotExists)** メソッドを使用して作成され、 **Region** インターフェースの実装を返します。 リージョンが取得されると、 リージョンのクラスローダーを設定または未設定でき、 リージョンをアクティベートまたはディアクティベートすることができます。 デフォルトでは、 **InactiveOnStartup** 設定属性が **true** に設定されない限りリージョンは有効になります。

第8章 キャッシュモードとクラスタリング

本章では、 JBoss Cache のクラスタリングについて説明します。

8.1. キャッシュレプリケーションモード

JBoss Cache はローカル (スタンドアロン) またはクラスタとして設定することができます。 クラスタの場合は、 変更をレプリケートまたは無効にするよう設定することができます。 この詳細は後ほど説明します。

8.1.1. ローカルモード

ローカルキャッシュはクラスタに参加せず、 クラスタ内の他のノードと通信しません。 JGroups チャネルは起動されませんが JGroups ライブラリへ依存します。

8.1.2. レプリケートされたキャッシュ

レプリケートされたキャッシュは、 クラスタ内にある他のキャッシュインスタンスの一部またはすべてへの変更をレプリケートします。 レプリケーションは変更後 (トランザクションまたはバッチなし) か、 トランザクションまたはバッチの終了後に行われます。

レプリケーションは同期的または非同期的に行うことができます。 オプションの使用はアプリケーションに依存します。 同期レプリケーションは、 変更がクラスタ内のすべてのノードに正常にレプリケートされるまで呼出元 (**`put()`** 上など) をブロックします。 非同期レプリケーションは、 レプリケーションをバックグラウンドで実行します (**`put()`** は即座に返します)。 また、 JBoss Cache は、 変更が定期的に行われる場合 (間隔ベースの場合など) や、 キューの大きさが要素の数を超えた場合にレプリケーションキューを提供します。 そのため、 レプリケーションキューはバックグラウンドスレッドによって実行される実際のレプリケーションよりも高いパフォーマンスを提供できます。

非同期レプリケーションは、 同期レプリケーションよりも高速です (呼出元のブロックなし)。 これは、 同期レプリケーションはクラスタ内のすべてのノードからノードが変更を正常に受け取り適用したという確認を必要とするためです (ラウンドトリップ時間)。 しかし、 同期レプリケーションが正常に返されると、 呼出元はすべての変更がすべてのノードで適用されたことを確信できますが、 これは非同期レプリケーションでは確信できません。 非同期レプリケーションでは、 エラーは単にログに書き込まれます。 トランザクションを使用している場合でも、 トランザクションに成功してもすべてのキャッシュインスタンスでレプリケーションに成功していないことことがあります。

8.1.2.1. レプリケートされたキャッシュトランザクション

トランザクションを使用している場合、 レプリケーションはトランザクション境界でのみ実行されます (トランザクションのコミット時など)。 この結果、 個々の変更の集まりではなく単一の変更がブロードキャストされるためレプリケーションのトラフィックが最小化され、 トランザクションを使用しない場合よりも非常に効率的になります。 また、 これによりトランザクションをロールバックした時はクラスタ全体に何もブロードキャストされません。

JBoss Cache はクラスタを非同期モードで稼働している場合は単相コミットプロトコル、 同期モードで稼働している場合は 2 相コミットプロトコルを使用します。

8.1.2.1.1. 単相コミット

キャッシュモードが REPL_ASYNC の場合に使用されます。すべての変更は、変更をローカルのインメモリステートに適用し、ローカルでコミットするようリモートキャッシュに指示する単一の呼び出しでレプリケーションされます。通信は非同期であるため、リモートエラー/ロールバックはトランザクションの開始元に通知されません。

8.1.2.1.2. 2 相コミット

キャッシュモードが REPL_SYNC である場合に使用されます。 トランザクションのコミット時に JBoss Cache が、 トランザクションに関連するすべての変更を実行する準備呼び出しをブロードキャストします。 リモートキャッシュはインメモリステートでローカルロックを取得し、 変更を適用します。 すべてのリモートキャッシュが準備呼び出しに応答すると、トランザクションの開始元がコミットをブロードキャストします。 これにより、 すべてのリモートキャッシュがデータをコミットするよう指示されます。 いずれかのキャッシュが準備フェーズの応答に失敗すると、 開始元がロールバックをブロードキャストします。

準備フェーズは同期的ですが、 コミットフェーズとロールバックフェーズは非同期的です。 これは、Sun の JTA 仕様には、 この段階のトランザクション障害のトランザクションリソースによる対処方法が指定されておらず、 トランザクションに参加している他のリソースが中間のステートであることがあるからです。 したがって、 このフェーズのトランザクションに対する同期通信のオーバーヘッドは考慮しません。 **SyncCommitPhase** および **SyncRollbackPhase** 設定属性を使用すると、 強制的に同期にすることができます。

8.1.2.2. バディレプリケーション

バディレプリケーションでは、クラスタ内のすべてのインスタンスにデータをレプリケートすることを回避できます。各インスタンスはクラスタ内の 1 つまたは複数の「バディ (仲間)」を選択し、これらの特定のバディにのみレプリケートします。これにより、他のインスタンスがクラスタに追加された時にメモリとネットワークトラフィックの影響がなくなり、 スケーラビリティが大幅に向上します。

バディレプリケーションの典型的な使用例としては、 HTTP セッションデータを保存するためにサーブレットコンテナによってレプリケートされたキャッシュが使用される場合などがあります。 バディレプリケーションを正常に動作し、 効果的に使用するには「**セッションアフィニティ**」 (HTTP セッションレプリケーションでは「**スティッキーセッション**」とも呼ばれる) を使用する必要があります。 これは、特定のデータが頻繁にアクセスされる場合に、 そのデータがラウンドロビン方式ではなく 1 つのインスタンスで常にアクセスされることが好ましいことを意味します。 これによりキャッシュクラスタがバディを選択する方法やデータの保存場所を最適化し、 レプリケーショントラフィックを最低限に抑えることができます。

これが不可能な場合はバディレプリケーションの利点はなく、 オーバヘッドが増加します。

8.1.2.2.1. バディの選択

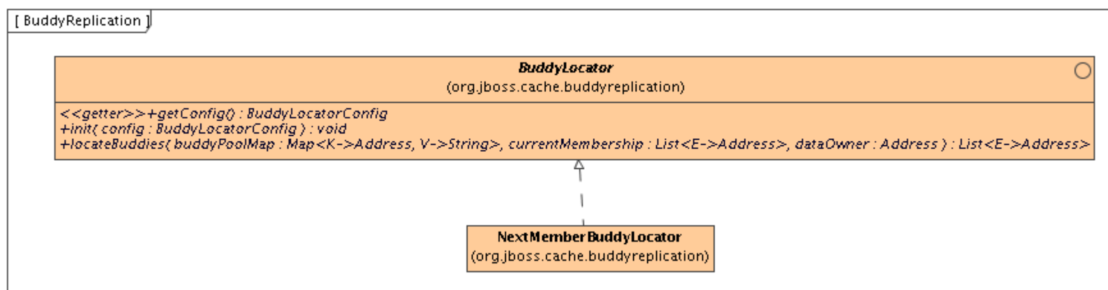


図8.1 BuddyLocator

バディレプリケーションは、 ネットワーク内でのバディ選択に使用する論理が含まれる **BuddyLocator** のインスタンスを使用します。 現在、 JBoss Cache には実装が提供されない場合にデフォルトで使用される単一の実装 **NextMemberBuddyLocator** が同梱されています。 **NextMemberBuddyLocator** は、 名前の通りクラスタ内の次のメンバーを選択し、 各インスタンスに対してバディを均等に分散します。

NextMemberBuddyLocator は 2 つのパラメータを受け取ります (両方ともオプション)。

- **numBuddies** - 各インスタンスがデータをバックアップするバディの数を指定します。このデフォルト値は 1 です。

- **ignoreColocatedBuddies** - 各インスタンスは異なる物理ホスト上でバディの選択を試みます。 選択できない場合は、 共同の場所にあるインスタンスに戻ります。 デフォルトは **true** です。

8.1.2.2.2. BuddyPools

バディプールは「レプリケーショングループ」とも呼ばれるオプションのコンストラクタで、 クラスタ内の各インスタンスをバディプール名で設定できます。 バディを選択する時にバディプールをサポートする **BuddyLocator** が同じバディプール名を共有するバディを選択しようとする「会員制クラブ」のようなものと考えてみてください。 これにより、 システム管理者にある程度の柔軟性を提供し、 バディーの選択方法を制御できるようになります。 例えば、 システム管理者は同じバディプール内の 2 つの異なる物理ラック上にある 2 つの異なる物理サーバーに 2 つのインスタンスを置くことができます。 したがって、 同じラックの異なるホストにあるインスタンスを選択する代わりに、 **BuddyLocator** は異なるラックの同じバディプールのインスタンスを選択するため、 ある程度の冗長性が追加されることがあります。

8.1.2.2.3. フェイルオーバー

不幸にもインスタンスがクラッシュすると、 キャッシュに接続しているクライアント (HTTP セッションレプリケーションなどの他の一部のサービスを使用して直接的または間接的に行う) がクラスタ内の他のキャッシュインスタンスに要求をリダイレクトできるとみなされます。 ここで「データグラビテーション」の概念が重要になります。

データグラビテーションとは、 クラスタ内のキャッシュに対する要求が行われ、 キャッシュにその情報が含まれない場合にクラスタ内の他のインスタンスにデータを要求する概念です。 データはレイジーに転送され、 他のノードが要求する場合のみ移行します。 これにより、 1 つまたは少数のノードのみが犠牲となるため、 ノードの周りに多くのデータが押し込まれるネットワークストームによる影響を回避することができます。

ノードの第 1 セクションにデータが見つからない場合、 他のキャッシュに対して保存したバックアップデータを確認するよう他のインスタンスに要求します (オプション)。 そのため、 セッションが含まれるキャッシュが消失しても、 このデータのバックアップを検索するようクラスタに要求し、 他のインスタンスはこのデータにアクセスすることができます。

見つかったデータは要求したインスタンスへ転送され、 インスタンスのデータツリーに追加されます。 セッションアフィニティが使用された場合にアフィニティがこのデータの「所有権」を取得した新しいキャッシュインスタンスに対して設定されるよう、 このデータは他のインスタンス (およびバックアップ) すべてから削除されます (オプション)。

データグラビテーションはインターセプタとして実装されます。データグラビテーションに関する設定プロパティは以下の通りです (すべてオプション)。

- **dataGravitationRemoveOnFind** - データを所有するかデータのバックアップを保持するすべてのリモートキャッシュがそのデータを削除し、 要求元のキャッシュが新しいデータ所有者になるよう強制します。 新しい所有者がバディへデータをレプリケートした後でのみ削除が実行されます。 **false** に設定すると、 削除ではなくエビクションがブロードキャストされ、 キャッシュローダーに持続されたステートはそのままの状態となります。 共有されたキャッシュローダーが設定されている場合に便利です。 デフォルト値は **true** です。

- **dataGravitationSearchBackupTrees** - バックアップと主要なデータツリーを検索するようリモートインスタンスに要求します。 デフォルト値は **true** です。 **true** に設定されている場合、 バックアップノードはデータ所有者だけでなくデータグラビテーション要求にも応答できます。

- **autoDataGravitation** - キャッシュミスが発生する度にデータグラビテーションを実行するかどうかを設定します。 不必要なネットワーク呼び出しを防ぐため、 デフォルト値は **false** になっています。 ほとんどのユースケースでは、 データのグラビテーションが必要な時や **Option** を渡す時を認識し、 呼び出し毎にデータグラビテーションを有効します。 **autoDataGravitation** が **true** の場合は、この **Option** は必要ありません。

8.1.2.2.4. 設定

バディーレプリケーションの設定に関する詳細は 12章*設定に関する参考資料* を参照してください。

8.2. 無効化

キャッシュがレプリケーションではなく無効化に対して設定されている場合は、 各時間データがキャッシュで変更され、 データが陳腐化しメモリからエビクトする必要があることを示すメッセージをクラスタ内の他のキャッシュが受け取ります。 無効化を共有されたキャッシュローダーと共に使用すると (9章*キャッシュローダー* の章を参照)、 リモートキャッシュが共有キャッシュローダーを参照して変更されたデータを取得します。 この利点は、 更新されたデータのレプリケーションに比べて無効化メッセージは非常に小さいためネットワークトラフィックが最小化されることと、 クラスタ内の他のキャッシュが変更されたデータをレイジーに検索する (必要な場合のみ) ことの 2 つになります。

無効化メッセージは、コミットに成功すると各変更が行われた後 (トランザクションやバッチを使用しない場合) またはトランザクションやバッチの完了時に送信されます。 無効化メッセージを変更ごとではなくトランザクション全体で最適化できるため、 より効率的です。

無効化は同期的または非同期的に設定できます。レプリケーションの場合と同様に同期的な無効化はクラスタ内のすべてのキャッシュが無効化メッセージを受け取り、不整合なデータを除外するまでブロックされます。非同期な無効化は、無効化メッセージがブロードキャストされてもブロックされず、応答を待たない「一方的な」モードで動作します。

8.3. ステート転送

「ステート転送」は、 JBoss Cache インスタンスが別のキャッシュインスタンスから現在のステートを取得し、 そのステートを JBoss Cache インスタンス自体のステートに統合することによりサービスの提供を準備するプロセスを意味します。

8.3.1. ステート転送タイプ

ステート転送に関する観点により、 ステート転送タイプは 3 つに分類されます。 初めに 、特定のステート転送実装である基礎配管 (underlying plumbing) では、 バイトアレイとストリーミングベースステート転送の 2 つの全く異なったステート転送があります。 次に、 転送されるサブツリーにより、 ステート転送は完全または部分的ステート転送になります。 キャッシュツリー全体の転送は完全転送となり、 特定のサブツリーの転送は部分的なステート転送となります。 最後に、 キャッシュの使用方法により、 ステート転送は「インメモリ」や「永続」転送になります。

8.3.2. バイトアレイおよびストリーミングベースのステート転送

バイトアレイベースの転送は、 2.0 までのすべてのリリースでデフォルトとして設定され、 キャッシュの唯一の転送メソッドでした。 バイトアレイベースの転送はバイトアレイへ転送されたステート全体をロードし、 ステートを受信するメンバーへ送信します。 大変大きなステート転送 (>1GB) の場合に OutOfMemoryException が発生することがバイトアレイベース転送の主な制限となります。 ステート転送をストリーミングすることで、 InputStream にステートリーダーを提供し、 OutputStream にステートライターを提供します。 OutputStream 抽象と InputStream 抽象により、 バイトのチャンクによるステート転送が可能になるため、 メモリの要件が軽減されます。 例えば、 集合サイズが 1GB であるツ

リーとしてアプリケーションステートが表現されている場合、 1GB バイトアレイを提供するのではなく、 ストリーミングステート転送はユーザー設定可能な N バイトのチャンクでステートを転送します。

バイトアレイおよびストリーミングベースのステート転送は API に対して完全な透過性と互換性があり、 標準のキャッシュ設定 XML ファイルより静的に設定されます。 転送タイプを変更する方法は、 JGroups のドキュメントを参照してください。

8.3.3. 完全および部分的ステート転送

インメモリまたは永続ステート転送が有効の場合は、キャッシュの使用方法に応じて完全または部分的なステート転送が異なるタイミングで実行されます。「完全な」ステート転送は、ツリー全体 (ルートノードとそれ以下のすべてのノード) に関連するステートの転送を意味します。「部分的な」ステート転送では、ツリーの一部 (ある FQN のノードとそれ以下のすべてのノード) のみが転送されます。

インメモリステート転送または永続ステート転送が有効な場合は、以下にタイミングでステート転送が実行されます。

1. 初期ステート転送。 キャッシュが最初に起動された時に実行される (**`start()`** メソッドの処理の一部として実行) 完全なステート転送になります。 ステートは最も長い期間稼働しているキャッシュインスタンスから取得されます。[1]ステートの受信または統合で問題が発生した場合は、 キャッシュが起動しません。

   以下の場合は初期ステート転送が実行されません。

   1. キャッシュの **`InactiveOnStartup`** プロパティは **`true`** です。 このプロパティはリージョンベースのマーシャリングと併用されます。

   2. バディレプリケーションが使用されます。バディレプリケーションを使用したステート転送の詳細については、以下を参照してください。

2. リージョンアクティベーション後の部分的ステート転送。 リージョンベースマーシャリングが使用されると、 アプリケーションはキャッシュに特定のクラスローダーを登録する必要があります。 このクラスローダーは、 キャッシュの特定リージョン (サブツリー) のステートをアンマーシャルするために使用されます。

   登録後、 関連するサブツリーのステートの部分ステート転送を開始する **`cache.getRegion(fqn, true).activate()`** をアプリケーションが呼び出します。 最初に、 クラスタ内の最も古いキャッシュインスタンスに要求を行います。 このインスタンスがステートを返さない場合は、 あるインスタンスがステートを提供するまで (またはすべてのインスタンスが要求を受けるまで) 各インスタンスへ順番に要求を行います。

   通常、 リージョンベースのマーシャリングが使用された場合、 キャッシュの **`InactiveOnStartup`** プロパティが **`true`** に設定されます。 これにより、 初期ステート転送が抑制され、 転送されたステートを非シリアル化できないため初期ステート転送に失敗します。

3. バディレプリケーション。 バディレプリケーションが使用された場合は、 初期ステート転送が無効になります。 キャッシュインスタンスがクラスタに参加すると、 キャッシュインスタンスは 1 つまたは複数の他のインスタンスのバディになり、 1 つまたは複数の他のインスタンスはそのバディになります。 インスタンスがバックアップを提供する新しいバディを持っていることを確認すると、 インスタンスは現在のステートを新しいバディにプッシュします。 この新しいバディへのステートのプッシュは「プル」のアプローチ (受信側がステートを要求し受信する) に基づいた他形式のステート転送とは若干異なります。 ただし、 ステートを準備および統合するプロセスは同じです。

バディグループ形成時のステートの「プッシュ」は、 **InactiveOnStartup** プロパティが **false** に設定されている場合にのみ実行されます。 **true** に設定されている場合、 グループの複数のメンバーにアプリケーションがリージョンをアクティベートした時のみバディ間のステート転送が実行されます。

リージョンのアクティベート呼び出し後の部分的ステート転送も、 バディレプリケーションの場合と若干異なります。 あるキャッシュインスタンスから部分的なステートを要求し、 インスタンスが応答するまですべてのインスタンスを試すのではなく、 バディレプリケーションではリージョンを有効化するインスタンスはバックアップとして機能する各インスタンスから部分的なステートを要求します。

8.3.4. 一時 (「インメモリ」) および永続ステート転送

取得および統合される基本的なステートは以下の 2 つです。

1. 「一時」または「インメモリ」ステート。 別のキャッシュインスタンスの実際のインメモリステートから構成されます。 ステートを提供しているキャッシュ内のさまざまなインメモリノードの内容がシリアライズされ、 転送されます。 受信側はデータをデシリアライズし、 自身のインメモリツリーに対応するノードを作成して転送されたデータを投入します。

   「インメモリ」ステート転送を有効にするには、キャッシュの **FetchInMemoryState** 設定属性を **true** に設定します。

2. 「永続」ステート。共有されないキャッシュノードが使用されている場合にのみ適用できます。ステートプロバイダキャッシュの永続ストアに保存されたステートは非シリアル化され、転送されます。受信側はそのデータをそれ自体のキャッシュローダーに渡し、データは受信側の永続ストアに永続化されます。

   「永続」ステート転送を有効にするにはキャッシュローダーの **fetchPersistentState** 属性を **true** に設定します。 複数のキャッシュローダーがチェーンで設定されている場合、 このプロパティを true に設定できるキャッシュローダーは 1 つのみです。 複数のキャッシュローダーに true を設定すると、 起動時に例外が発生します。

   共有されたキャッシュローダーで永続ステート転送を行うと、 データを提供する永続ストアがデータを受信することになるため意味がありません。 そのため、 共有されたキャッシュローダーが使用されている場合、 キャッシュローダーの **fetchPersistentState** が **true** に設定されていてもキャッシュは永続ステート転送を許可しません。

どちらのタイプのステート転送が適当であるかはキャッシュの使用方法によって異なります。

1. ライトスルーキャッシュローダーを使用する場合、 現在のキャッシュステートが永続ステートによって完全に表されます。 データはインメモリステートからエビクトされているかもしれませんが、 永続ストアに保存されます。 この場合、キャッシュローダーが共有されていないと、新しいキャッシュが正しいステートを持つように永続ステート転送が使用されます。 「ホット」キャッシュを使用したい場合は、 インメモリステートも転送することができます。 「ホット」キャッシュとはキャッシュがサービスの提供を開始する時にメモリ内に関連するデータがすべてあるキャッシュのことです。 (**<loaders>** 設定要素の **<preload>** 要素を使用して、 インメモリステート転送の必要なく「ウォーム」または「ホット」キャッシュを提供することもできます。 この方法では、 ステートを提供するキャッシュインスタンスの負荷は減少しますが、 受信側の永続ストアの負荷は増加します)

2. キャッシュローダーがパッシベーションで使用された場合、 インメモリ (非パッシベーション) ステートと永続 (パッシベーション) ステートを組み合わせないと完全なステートを取得することはできません。 そのため、 インメモリステート転送が必要となります。 キャッシュローダーが共有されていない場合は永続ステート転送が必要となります。

3. キャッシュローダーが使用されず、 キャッシュがライトアサイド (write-aside) キャッシュ (データベースなどの永続ストアにも存在するデータをキャッシュするために使用されるキャッシュ) である場合、 インメモリステートの転送を行うかどうかは「ホット」キャッシュ使用の有無によって決まります。

8.3.5. ステート転送の設定

ステート転送が予想通り挙動するようにするため、 クラスタのすべてのノードが永続ステートと一時ステートに対して同じ設定であることが重要となります。 これは、 要求されるとバイトアレイベースの転送は要求側の設定に依存しますが、 ストリームベースの転送は要求側と送信側両方の設定に依存し、両方の設定が同じであることを前提としているからです。

---

[1] JGroups では、 最も長い期間稼働しているキャッシュインスタンスは常にコーディネータとなります。

第9章 キャッシュローダー

JBoss Cache は **CacheLoader** を使用してインメモリキャッシュをバックエンドデータストアにバックアップすることができます。 JBoss Cache にキャッシュローダーが設定されている場合は、 以下の機能が提供されます。

- キャッシュ要素がアクセスされ、要素がキャッシュに存在しない場合 (エビクションやサーバー再起動が原因の場合など) は、キャッシュローダーが要素をキャッシュに透過的にロードします (要素がバックエンドストアにある場合)。

- 要素が変更、 追加、 削除されると、 その変更はキャッシュローダーを介してバックエンドストアに保持されます。 トランザクションを使用する場合、 トランザクション内で作成されたすべての変更は保持されます。 この場合、 **CacheLoader** はトランザクションマネージャにより実行された 2 相コミットプロトコルで実行されますが、 明示的には実行されません。

9.1. CACHELOADER インターフェースとライフサイクル

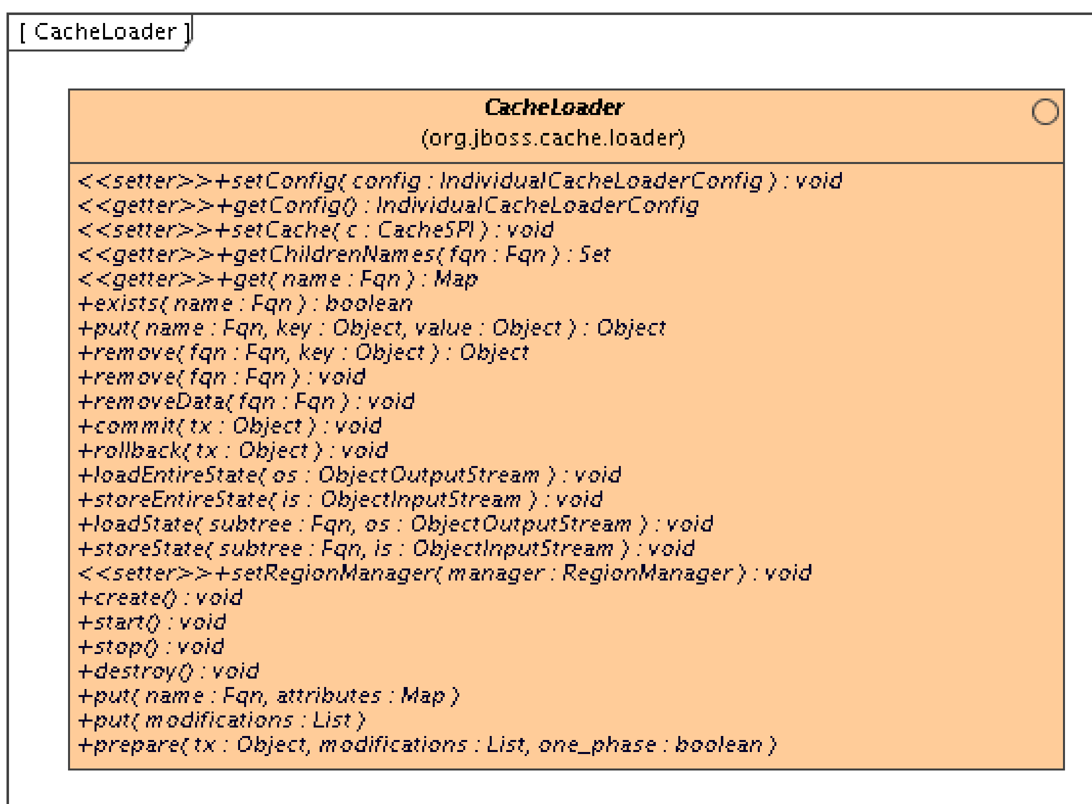


図9.1 CacheLoader インターフェース

JBoss Cache と **CacheLoader** 実装の対話は次のように行われます。 **CacheLoaderConfiguration** (下記参照) は null 以外の値を持ちます。 設定された各 **cacheloader** のインスタンスは、 キャッシュの作成時に作成され、 キャッシュの起動時に起動します。

キャッシュの起動時に **CacheLoader.create()** と **CacheLoader.start()** が呼び出されます。 また、 キャッシュの終了時には **stop()** と **destroy()** が呼び出されます。

次に、 **setConfig()** と **setCache()** が呼び出されます。 **setCache()** は、キャッシュへの参照を保存するため使用され、 **setConfig()** は **CacheLoader** のこのインスタンスを設定するために使用されます。 たとえば、 この例ではキャッシュローダーがデータベースへの接続を確立できます。

CacheLoader インターフェースには、 トランザクションが使用されない場合に呼び出される

get()、 **put()**、 **remove()**、 **removeData()** メソッドがあります。 これらのメソッドは値を即座に取得 (**get()**)、 設定(**put()**)、 削除 (**remove()**、 **removeData()**) します。 これらのメソッドについては、 インターフェースの javadoc コメントで説明されています。

また、 トランザクションで使用されるメソッドには **prepare()**、**commit()**、 **rollback()** の 3 つがあります。 **prepare()** メソッドは、 トランザクションがコミットされる時に呼び出され、 引数としてトランザクションオブジェクトと変更のリストを持ちます。 トランザクションオブジェクトは、値が変更のリストであるトランザクションのハッシュマップへのキーとして使用できます。 各変更リストは、 あるトランザクションに対するキャッシュへの変更を表す複数の **Modification** 要素を持ちます。 **prepare()** が正常に値を返した場合、 キャッシュローダーはトランザクションを正常にコミット (またはロールバック) できるはずです。

JBoss Cache はキャッシュローダー上で prepare() や commit()、 rollback() を適時に呼び出します。

commit() メソッドはキャッシュローダーがトランザクションをコミットするよう指示し、**rollback()** メソッドはキャッシュローダーがそのトランザクションに関連する変更を破棄するよう指示します。

各メソッドや必要なコントラクト実装の詳細は、 このインターフェースに関する javadoc を参照してください。

9.2. 設定

JBoss Cacheの XML ファイルではキャッシュローダーは次のように設定されます。 1 つのチェーンに複数のキャッシュローダーを定義することができます。 キャッシュは、 null 以外の有効なデータ要素を見つけるまで、 設定された順序ですべてのキャッシュローダーを参照します。書き込みを実行すると、 すべてのキャッシュローダーに書き込まれます (特定のキャッシュローダーに対して **ignoreModifications** 要素が **true** に設定されている場合を除きます)。 詳細は次の設定セクションを参照してください。

```
...

<!-- Cache loader config block -->
<!-- if passivation is true, only the first cache loader is used; the rest
are ignored -->
<loaders passivation="false" shared="false">
      <preload>
         <!-- Fqns to preload -->
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
the rest are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
              ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.driver=com.mysql.jdbc.Driver
            cache.jdbc.url=jdbc:mysql://localhost:3306/jbossdb
            cache.jdbc.user=root
            cache.jdbc.password=
         </properties>
      </loader>
  </loaders>
```

class 要素はキャッシュローダー実装のクラスを定義します (JBoss AS のプロパティエディタに存在

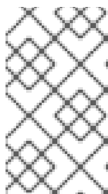
するバグにより、 Windows ファイル名の変数のバックスラッシュが正常に展開されない場合があります。 このような場合、 「replace="false"」 と指定する必要があります)。 キャッシュローダーの実装には空のコンストラクタが存在しなければなりません。

properties 要素は該当する実装に固有する設定を定義します。 たとえば、 ファイルシステムベースの実装は使用するルートディレクトリを定義し、 データベースの実装はデータベース接続を確立するデータベース URL、 名前、 パスワードを定義できます。 この設定は **CacheLoader.setConfig(Properties)** よりキャッシュローダー実装に渡されます。 バックスペースをエスケープ処理しなければならないことがあります。

preload は、 起動時にキャッシュによってビジットされるノードやサブツリー全体のリストを定義し、 これらのノードに関連するデータを事前ロードできるようにします。 デフォルト ("/") では、 バックエンドストアにある全データをキャッシュへロードしますが、 バックエンドストアのデータが大きい場合もあるため、 不適切である場合も多いでしょう。 例えば、 **/a, /product/catalogue** はサブツリー **/a** と **/product/catalogue** をキャッシュにロードしますが、 それ以外はロードしません。 それ以外はアクセスされた時にレイジーにロードされます。 事前ロードは、 あるサブツリー下の要素を頻繁に使用することが予想される場合に適しています。

fetchPersistentState は、 クラスタへ参加した時にキャッシュの永続ステートをフェッチするかどうかを決定します。 このプロパティを true に設定できるキャッシュローダーは 1 つだけです。 複数のキャッシュローダーに true を設定すると、 キャッシュサービスの起動時に設定例外がスローされます。

async は、 キャッシュローダーへの書き込み完了するまでブロックするか、 書き込みが即座に返すよう別のスレッドで実行するかを決定します。 true に設定すると、 **org.jboss.cache.loader.AsyncCacheLoader** のインスタンスが、 使用される実際のキャッシュローダーインスタンスで構築されます。 次に、 **AsyncCacheLoader** は、 必要に応じて別のスレッドを使用してすべての要求を基礎となるキャッシュローダーに委譲します。 詳細については、 **AsyncCacheLoader** に関する Javadoc を参照してください。 指定がない場合、 **async** 要素のデフォルトは **false** になります。



注記

すべての書き込みが非同期的に実行されるため、 常にダーティな読み取りが発生する可能性があります。 したがって、 書き込みの成功は保証できません。 これを念頭に置いて **async** 要素を true に設定するようにしてください。

ignoreModifications は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。 ネットワークの全サーバーによって使用される **JDBCCacheLoader** など、 インメモリキャッシュと同じサーバー上にあるファイルベースのキャッシュローダーのみに一時アプリケーションデータが存在しなければならない場合などが該当します。 この機能により、 共有される **JDBCCacheLoader** へ書き込みせずに 「ローカル」 のファイルキャッシュローダーへ書き込みすることが可能になります。 このプロパティはデフォルトで **false** に設定されているため、 設定された全キャッシュローダーへ書き込みが伝播されます。

purgeOnStatup は、キャッシュローダーが起動した時に指定されたキャッシュローダーを空にします (**ignoreModifications** が **false** に設定されている場合)。

shared は異なるキャッシュインスタンスによってキャッシュローダーが共有されることを表しています。 例えば、 クラスタのすべてのインスタンスが同じ JDBC 設定を使用して同じリモートの共有データベースと通信する場合などがこれに該当します。 **true** を設定すると、 異なるキャッシュインスタンスによって同じデータがキャッシュに繰り返し書き込まれないようにし、 不必要な書き込みが発生しないようにします。 デフォルト値は **false** になります。

9.2.1. シングルトンストアの設定

```
<loaders passivation="false" shared="true">
   <preload>
      <node fqn="/a/b/c"/>
      <node fqn="/f/r/s"/>
   </preload>

   <!-- we can now have multiple cache loaders, which get chained -->
   <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="false"
         ignoreModifications="false" purgeOnStartup="false">
      <properties>
        cache.jdbc.datasource=java:/DefaultDS
      </properties>
      <singletonStore enabled="true"
class="org.jboss.cache.loader.SingletonStoreCacheLoader">
         <properties>
            pushStateWhenCoordinator=true
            pushStateWhenCoordinatorTimeout=20000
         </properties>
      </singletonStore>
   </loader>
</loaders>
```

singletonStore 要素は、 クラスタのノードであるコーディネータのみが変更を保存できるようにします。 データがノードに入力されると、 キャッシュのインメモリステートを同期に保つため、 データがレプリケートされます。 このステートをディスクにプッシュすることのみがコーディネータの役目となります。 すべてのノードに **enabled** サブ要素を設定してこの機能を有効にすることができますが、クラスタのコーディネータのみが **loader** 要素で定義された基礎のキャッシュローダーへ変更を保存することができます。 **singletonStore** が有効になっている状態でキャッシュローダーを **shared** として定義することはできません。 **enabled** のデフォルト値は **false** です。

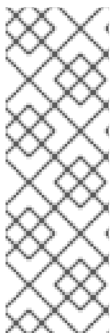
シングルトンストア機能を提供する実装クラスを指定する **class** 要素を **singletonStore** 要素内で定義することもできます。 このクラスは **org.jboss.cache.loader.AbstractDelegatingCacheLoader** を拡張する必要があり、 指定がない場合は **org.jboss.cache.loader.SingletonStoreCacheLoader** がデフォルトとなります。

properties サブ要素は、 シングルトンストア機能を提供するクラスの動作を変更できるプロパティを定義します。 デフォルトでは、 **pushStateWhenCoordinator** プロパティと **pushStateWhenCoordinatorTimeout** プロパティが定義されていますが、 シングルトンストア機能を提供するユーザー定義クラスの必要に応じて追加することができます。

pushStateWhenCoordinator は、 クラスタトポロジの変更によりコーディネータが新たに選出された場合に、 ノードがコーディネータとなった際にインメモリステートをキャッシュストアへプッシュできるようにします。 これは、 コーディネータがクラッシュし、 新しいコーディネータが選出されるまでに時間の隔たりがある場合に大変便利です。 この場合、 プロパティが **false** に設定され、 キャッシュが更新されると、 変更は永続化されません。 各ノードのキャッシュローダーが異なる場所に設定されている場合、 このプロパティを **true** に設定した方がよいでしょう。 デフォルト値は **true** です。

pushStateWhenCoordinatorTimeout は **pushStateWhenCoordinator** が **true** に設定されている場合のみ該当します。 この場合、 インメモリステートを基礎のキャッシュローダーへプッシュする処理の最大時間 (ミリ秒単位) を設定し、 この時間を超過すると **PushStateException** が報告されま

す。 デフォルト値は 20000 です。



注記

キャッシュローダーをシングルトンとして設定し、 キャッシュパッシベーションを使用すると (エビクションより)、 悪影響を与える可能性があります。 クラスタが新しいコーディネータを選出している間に、 エビクションが原因でノードがパッシベートされると、 データが損失されます。 これは、 この時点ではアクティブなコーディネータが存在せず、 パッシベートされたノードを保存するノードがクラスタに存在しないからです。 コーディネータがクラスタから退去したり、 クラッシュしたり応答しなくなった場合に新しいコーディネータがクラスタで選出されます。

9.3. 同梱される実装

JBossCache に同梱される現在利用可能な実装は次の通りです。

9.3.1. ファイルシステムベースのキャッシュローダー

JBoss Cache にはファイルシステムをデータストアとして使用する複数のキャッシュローダーが同梱されています。 全キャッシュローダーの **<loader><properties>** 設定要素に、 永続ストアとして使用されるディレクトリをマップする **location** プロパティが含まれなければなりません (**location=/tmp/myDataStore** など)。 主にテスト向けに使用され、 実稼働での使用は推奨されません。

- **FileCacheLoader** は簡単なファイルシステムベースの実装です。 デフォルトでは、 このキャッシュローダーは無効な文字など場所やツリーノード名に使用される文字の移植性をチェックします。 このチェックを無効にするには、 **check.character.portability** プロパティを **<properties>** 要素に追加し、 **false** に設定します (例: **check.character.portability=false**)。

  FileCacheLoader には限界があるため、 実稼働環境での使用が制限されます。 実稼働環境で使用する場合は相当な注意を払い、 制限を十分に理解する必要があります。

  - FileCacheLoader がディスク (ディレクトリおよびファイル) 上でツリー構造を表す方法により、 深いツリーに対するトラバースは十分ではありません。
  - NFS や Windows の共有など共有ファイルシステムで使用しないようにしてください。 ファイルロッキングを適切に実装しないため、 データが破損することがあります。
  - 分離レベルを NONE として使用すると、 複数のスレッドが同じファイルに書き込みしようとするため、 書き込みに間違いが生じます。
  - ファイルシステムは継承的にトランザクションではないため、 トランザクションコンテキストでキャッシュを使用しようとすると、 ファイルへの書き込み (コミットフェーズで実行される) をリカバリできない場合に障害が発生します。

  FileCacheLoader を非常に並列的なトランザクション環境や負荷が高い環境では使用せず、 テストのみ使用することが推奨されます。

- **BdbjeCacheLoader** は Oracle/Sleepycat の BerkeleyDB Java Edition を基にしたキャッシュローダーの実装です。

- **JdbmCacheLoader** は、 BerkeleyDB の代替となる高速で無料の JDBM エンジン を基にしたキャッシュローダーの実装です。

BerkeleyDB 実装は、 ファイルシステムベースの実装よりも大変効率的で、 トランザクションの保証を提供しますが、 アプリケーションと共に配布される場合は商業ライセンスが必要となります (詳細は http://www.oracle.com/database/berkeley-db/index.html を参照)。

9.3.2. 他のキャッシュに委譲するキャッシュローダー

- **`LocalDelegatingCacheLoader`** は、 別のローカル (同じ JVM) キャッシュからのロードと別のローカルキャッシュへの保存を有効にします。

- **`ClusteredCacheLoader`** はデータをレプリケートするために使用されるクラスタリングプロトコルを用いて、 インメモリデータの同じクラスタで別のキャッシュをクエリできるようにします。 レプリケーションは必要な更新をすべて処理するため、 書き込みは「保存」されません。 **`timeout`** というプロパティを指定する必要があります。 **`timeout`** はキャッシュローダーがクラスタからの応答を待機する時間をミリ秒単位で指定する long 値で、 この時間を経過すると null 値を適用します。 例えば、 **`timeout = 3000`** の場合はタイムアウト値が 3 秒になります。

9.3.3. JDBCCacheLoader

JBossCache は、 ノードのステートをリレーショナルデータベースに保存およびロードする JDBC ベースのキャッシュローダー実装と配布されます。 実装クラスは **`org.jboss.cache.loader.JDBCCacheLoader`** です。

現在の実装は 1 つのテーブルしか使用しません。テーブルの各行は 1 つのノードを表し、 3 つの列を含みます。

- **Fqn** の列 (1 次キー列でもある)
- ノードの内容の列 (属性と値のペア)
- 親 **Fqn** の列

Fqn は文字列として保存されます。 ノードの内容は BLOB として保存されます。

警告

JBoss Cache では **Fqn** で使用されるオブジェクトのタイプに制限がありませんが、 このキャッシュローダー実装では **Fqn** にタイプ **`java.lang.String`** のオブジェクトのみが含まれる必要があります。 **Fqn** の長さも制限があります。 **Fqn** は 1 次キーであるため、 デフォルトの列タイプは、 使用するデータベースによって決められる最大長までのテキスト値を保存できる **`VARCHAR`** になります。

特定のデータベースシステムの設定については、 この wiki ページ を参照してください。

9.3.3.1. JDBCCacheLoader の設定

9.3.3.1.1. テーブルの設定

テーブルと列の名前、および列タイプは以下のプロパティで設定できます。

- **cache.jdbc.table.name** - テーブル名です。 `{schema_name}.{table_name}` のように、テーブルのスキーマ名を先頭に追加することができます。 デフォルト値は「jbosscache」です。

- **cache.jdbc.table.primarykey** - テーブルの 1 次キーの名前。デフォルト値は、「jbosscache_pk」です。

- **cache.jdbc.table.create** - true または false を設定できます。 起動時にテーブルを作成するかどうかを指定します。 true の場合、 テーブルが存在しないと作成されます。デフォルト値は true です。

- **cache.jdbc.table.drop** - true または false を設定できます。シャットダウン時にテーブルを破棄するかどうかを指定します。デフォルト値は true です。

- **cache.jdbc.fqn.column** - FQN の列名。デフォルト値は「fqn」です。

- **cache.jdbc.fqn.type** - FQN の列タイプ。デフォルト値は「varchar(255)」です。

- **cache.jdbc.node.column** - ノード内容の列名。デフォルト値は「node」です。

- **cache.jdbc.node.type** - ノード内容の列タイプ。デフォルト値は「blob」です。このタイプは使用するデータベースに対して有効なバイアンリデータタイプを指定する必要があります。

9.3.3.1.2. DataSource

管理された環境 (アプリケーションサーバーなど) で JBossCache を使用する場合は、使用する DataSource の JNDI 名を指定します。

- **cache.jdbc.datasource** - DataSource の JNDI 名。 デフォルト値は `java:/DefaultDS` です。

9.3.3.1.3. JDBC ドライバ

DataSource を使用**しない場合は**、JDBC ドライバを使用してデータベースアクセスを設定する以下のプロパティを持ちます。

- **cache.jdbc.driver** - 完全修飾 JDBC ドライバ名

- **cache.jdbc.url** - データベースに接続する URL

- **cache.jdbc.user** - データベースに接続するユーザー名

- **cache.jdbc.password** - データベースに接続するパスワード

9.3.3.1.4. c3p0 接続プーリング

JBoss Cache は、 c3p0:JDBC DataSources/Resource Pools ライブラリをスタンドアロンで使用しているアプリケーションサーバー外部で実行している場合に JDBC 接続プーリングを実装します。 有効にするには、 次のプロパティを編集します。

- **cache.jdbc.connection.factory** - 接続ファクトリクラス名。 設定されていない場合、 標準のプールされていない実装がデフォルトとなります。 c3p0 プーリングを有効にするには、 c3p0 の接続ファクトリクラスを設定します。 次の例を参照してください。

同じキャッシュローダープロパティのセクションに任意の c3p0 パラメータを設定することも可能ですが、 必ずプロパティ名の先頭に「c3p0」を付けるようにしてください。 使用できるプロパティの一覧を検索するには、 c3p0:JDBC DataSources/Resource Pools に配布される c3p0 ライブラリバージョン

向けの c3p0 ドキュメントを確認してください。 異なるプーリングパラメータを簡単に試せるようにするため、 **-Dc3p0.maxPoolSize=20** のように JBoss Cache XML 設定ファイルのプロパティにある値を上書きし、 システムプロパティよりこれらのプロパティを設定することができます。 c3p0 プロパティが設定ファイルとシステムプロパティのいずれにも定義されていない場合、 c3p0 ドキュメントに記載されているデフォルト値が適用されます。

9.3.3.1.5. 設定例

以下はデータベースとして Oracle を使用した JDBCCacheLoader の例です。 CacheLoaderConfiguration XML 要素には、 データベース関連の設定を定義する任意のプロパティセットが含まれます。

```
<loaders passivation="false" shared="false">
      <preload>
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
the rest are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
              ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.table.name=jbosscache
            cache.jdbc.table.create=true
            cache.jdbc.table.drop=true
            cache.jdbc.table.primarykey=jbosscache_pk
            cache.jdbc.fqn.column=fqn
            cache.jdbc.fqn.type=VARCHAR(255)
            cache.jdbc.node.column=node
            cache.jdbc.node.type=BLOB
            cache.jdbc.parent.column=parent
            cache.jdbc.driver=oracle.jdbc.OracleDriver
            cache.jdbc.url=jdbc:oracle:thin:@localhost:1521:JBOSSDB
            cache.jdbc.user=SCOTT
            cache.jdbc.password=TIGER
          </properties>
      </loader>
  </loaders>
```

全体の JDBC 接続を設定する代わりに、既存のデータソースの名前を指定できます。

```
 <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
the rest are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
              ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.datasource=java:/DefaultDS
         </properties>
      </loader>
  </loaders>
```

c3p0 JDBC 接続プーリングを使用したキャッシュローダーの設定例:

```
 <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
the rest are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
              ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.table.name=jbosscache
            cache.jdbc.table.create=true
            cache.jdbc.table.drop=true
            cache.jdbc.table.primarykey=jbosscache_pk
            cache.jdbc.fqn.column=fqn
            cache.jdbc.fqn.type=VARCHAR(255)
            cache.jdbc.node.column=node
            cache.jdbc.node.type=BLOB
            cache.jdbc.parent.column=parent
            cache.jdbc.driver=oracle.jdbc.OracleDriver
            cache.jdbc.url=jdbc:oracle:thin:@localhost:1521:JBOSSDB
            cache.jdbc.user=SCOTT
            cache.jdbc.password=TIGER

cache.jdbc.connection.factory=org.jboss.cache.loader.C3p0ConnectionFactory
            c3p0.maxPoolSize=20
            c3p0.checkoutTimeout=5000
         </properties>
      </loader>
   </loaders>
```

9.3.4. S3CacheLoader

`S3CacheLoader` はキャッシュデータの保存に Amazon S3 (Simple Storage Solution) を使用します。 Amazon S3 はリモートネットワークストレージで、 比較的待ち時間が長いため、 メディアやファイルなど大きなデータを保存するキャッシュに最適です。 また、 リモート管理できる信頼のおけるストレージが必要な場合は、 JDBC やファイルシステムベースのキャッシュローダーではなく、 このキャッシュローダーを考慮してください。 Amazon の EC2 (Elastic Compute Cloud) で実行しているアプリケーションに対してもこのキャッシュローダーを使用してください。

ストレージとして Amazon S3 を使用する場合は、 JBoss Cache の使用を考慮してみてください。 JBoss Cache はデータに対してインメモリキャッシングを提供し、 リモートアクセス呼び出しの回数を最小限にするため、 待ち時間や Amazon S3 データをフェッチする負荷を削減することができます。 また、 キャッシュレプリケーションを使用すると、 毎回リモートアクセスする必要なくローカルクラスタよりデータをロードすることができます。

Amazon S3 はトランザクションをサポートしていません。 アプリケーションでトランザクションが使用される場合、 このキャッシュローダーを使用するとステートの不一致が発生する可能性があります。 しかし、 書き込みはアトミックであるため、 書き込みに失敗すると何も書き込まれないため、 データの破損が生じることがありません。

データはノードの Fqn を基にキーに保存され、 ノードデータは **`CacheSPI.getMarshaller()`** インスタンスを使用して java.util.Map としてシリアライズされます。 データの構成や保存については

javadoc を参照してください。 データは Java シリアライゼーションを使用して保存されます。 そのため、 HTTP 上で JBoss Cache 以外のクライアントへデータをアクセスするのは容易ではありません。これに関し、 キャッシュローダーを拡張するためフィードバックやご意見をお寄せください。

このキャッシュローダーでは、 データが単一の Map インスタンスに保存されるため、 **Node.remove(Object)** や **Node.put(Object, Object)** などの単一キー操作は最も遅くなります。 効率性を向上するには、 **Node.replaceAll(Map)** や **Node.clearData()** などのバルク操作を使用してください。 また、 **cache.s3.optimize** オプションも使用してみてください。

9.3.4.1. Amazon S3 ライブラリ

S3 キャッシュローダーはデフォルトのディストリビューションで提供されますが、 ランタイム時にサービスにアクセスするにはライブラリが必要となります。 ランタイムライブラリは Sourceforge Maven Repository より入手できます。 pom.xml ファイルに次のセクションを追加します。

```
<repository>
   <id>e-xml.sourceforge.net</id>
   <url>http://e-xml.sourceforge.net/maven2/repository</url>
</repository>
  ...
<dependency>
   <groupId>net.noderunner</groupId>
   <artifactId>amazon-s3</artifactId>
   <version>1.0.0.0</version>
   <scope>runtime</scope>
</dependency>
```

Maven を使用しなくても、 レポジトリや この URL より amazon-s3 ライブラリをダウンロードすることができます。

9.3.4.2. 設定

最低でも Amazon S3 アクセスキーやシークレットアクセスキーを設定する必要があります。 次の設定キーは一般的な使用順に記載されています。

- **cache.s3.accessKeyId** - アカウントプロフィールより使用できる Amazon S3 アクセスキーです。
- **cache.s3.secretAccessKey** - アカウントプロフィールより使用できる Amazon S3 シークレットアクセスキーです。 パスワードであるため、 配布したりビルドしたソフトウェアに同梱しないでください。
- **cache.s3.secure** - デフォルトは **false** で、 トラフィックは暗号化されずパブリックインターネット上で送信されます。 **true** を設定すると HTTPS を使用します。 暗号化されていないアップロードやダウンロードは CPU の使用を軽減します。
- **cache.s3.bucket** - データを保存するバケットの名前です。 異なるキャッシュが同じアクセスキーを使用する場合は、 異なるバケット名を使用してください。 バケットの定義については S3 のドキュメントを参照してください。 デフォルト値は **jboss-cache** になります。
- **cache.s3.callingFormat** - **PATH**、**SUBDOMAIN**、 **VANITY** のいずれかになります。 呼び出しドメインの使用に関しては、 S3 のドキュメントを参照してください。 デフォルト値は **SUBDOMAIN** になります。

- **cache.s3.optimize** - デフォルトは **false** になります。 true の場合、 フェッチやマージを試行せずに、 **put(Map)** 操作が Fqn に保存されたデータを置き換えます (現在、 このオプションは実験段階です)。

- **cache.s3.parentCache** - デフォルトは **true**です。 別のキャッシュに作成されたノードの親ノードを削除する同じ S3 バケットを共有する複数のキャッシュを使用している場合は、 この値を **false** に設定します (一般的なユースケースではありません)。

  JBoss Cache はノードをツリー形式で保存し、 必要な場合に中間の親ノードを自動的に作成します。 **getChildrenNames** などの操作が正しく動作するようにするため、 S3 キャッシュローダーもこれらの親ノードを作成する必要があります。 各 **put** 操作に対して親ノードがすべて存在するか確認するのは比較的負荷が高くなるため、 デフォルトではキャッシュローダーは親ノードの存在をキャッシュします。

- **cache.s3.location** - データの一次保存場所を選択し、 ロードや読み出しの待ち時間を削減します。 データをヨーロッパに保存する場合は **EU** に設定します。 デフォルトはアメリカにデータを保存する **null** になります。

9.3.5. TcpDelegatingCacheLoader

このキャッシュローダーはロードと保存を JBossCache の別のインスタンスに委譲します。 このインスタンスは、 同じアドレス空間、 同じホストの別プロセス、別ホストの別プロセスのいずれかに存在できます。

TcpDelegatingCacheLoader はリモートの **org.jboss.cache.loader.tcp.TcpCacheServer** と対話します。 **org.jboss.cache.loader.tcp.TcpCacheServer** はコマンドラインで開始されたスタンドアロンプロセスか、 JBoss AS 内に組み込まれた MBean となります。 **TcpCacheServer** は別の JBossCache インスタンスへの参照を持ちます (独自で作成した参照か、 依存関係の挿入により JBoss が作成した参照など提供された参照になります)。

JBoss Cache 2.1.0 より、 TcpCacheServer への接続が切断されると **TcpDelegatingCacheLoader** が透過的に再接続を処理するようになりました。

TcpDelegatingCacheLoader はリモート TcpCacheServer のホストとポートで設定され、 これを使用して通信を行います。 また、 TcpCacheServer への再接続を透過的に制御するため、 新しい任意のパラメータが 2 つ使用されます。 **timeout** プロパティ (デフォルト値は 5000) は、 キャッシュローダーが TcpCacheServer への接続を再試行し続ける時間を指定し、 この時間を超過すると再試行を断念し例外をスローします。 **reconnectWaitTime** (デフォルト値は 500) は、 通信の障害を検出した場合にキャッシュローダーが再接続を実行する前に待機する時間になります。 最後の 2 つのパラメータは、 キャッシュローダーに一定のフォールトトラレンスを追加し、 TcpCacheServer の再起動に対応するために使用することができます。

設定は次のようになります。

```
 <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
the rest are ignored -->
      <loader class="org.jboss.cache.loader.TcpDelegatingCacheLoader">
         <properties>
            host=myRemoteServer
            port=7500
            timeout=10000
```

```
            reconnectWaitTime=250
         </properties>
      </loader>
   </loaders>
```

これは、 JBossCache が **myRemoteServer:7500** で実行されているリモートの TcpCacheServer にすべてのロード要求と保存要求を委譲することを意味します。

典型的なユースケースとしては、 同じクラスタ内でレプリケートされた複数の JBoss Cache インスタンスがすべて同じ TcpCacheServer インスタンスへ委譲する例があります 。TcpCacheServer は JDBCCacheLoader より TcpCacheServer 自体をデータベースへ委譲することがありますが、 ここで重要なのは 5 つのノードがすべて同じデータセットにアクセスしていると、 アンロードされたデータセットごとに 1 つのSQL ステートを実行する TcpCacheServer よりデータをロードすることです。ノードがデータベースに直接アクセスすると、 同じ SQL が複数回実行されます。 したがって、 TcpCacheServer はデータベースの前にある自然なキャッシュとして機能します (ネットワークラウンドトリップがデータベースのアクセスよりも早い場合。 通常データベースのアクセスにはネットワークラウンドトリップも含まれます)。

単一障害点 (Single Point of Failure) の問題を軽減するために複数のキャッシュローダーを設定することができます。 最初のキャッシュローダーは ClusteredCacheLoader、 2 番目は TcpDelegatingCacheLoader、 最後は JDBCacheLoader とし、 低い方から順にキャッシュへアクセスする負荷を効果的に定義します。

9.3.6. キャッシュローダーの変換

FileCacheLoader および **JDBCCacheLoader** ベースのキャッシュストアにキャッシュされたデータが書き込まれる方法は JBoss Cache 2.0 で変更になりました。こ れらのキャッシュローダーは、 ネットワーク全体でデータをレプリケートするために使用される同じマーシャリングフレームワークを使用してデータを読み書きするようになりました。 他のノードがこの形式を認識することのみが必要となるため、 この変更はレプリケーションではほとんど影響がありません。 しかし、 キャッシュストアのデータの形式を変更すると、 他の問題が生じます。 JBoss Cache 1.x.x 形式で保存されたデータをどのように JBoss Cache 2.0 形式へ移行するのでしょうか。

これを考慮し、 JBoss Cache 2.0 には任意の jbosscache-cacheloader-migration.jar ファイルに **org.jboss.cache.loader.TransformingFileCacheLoader** と **org.jboss.cache.loader.TransformingJDBCCacheLoader** の 2 つのキャッシュローダー実装が含まれています。 これらのキャッシュローダーは JBoss Cache 1.x.x 形式のキャッシュストアよりデータを読み取り、 JBoss Cache 2.0 形式のキャッシュストアへデータを書き込む 1 回限りのキャッシュローダーです。

ユーザーが既存のキャッシュ設定ファイルを一瞬に変更し、 これらのキャッシュローダーを使用してキャッシュのインスタンスを作成する小さな Java アプリケーションを作成し、 再帰的にキャッシュ全体を読み取り、 キャッシュにデータを書き戻します。 データが変換されると、 元のキャッシュ設定ファイルに戻ることができます。 ユーザーがこのタスクを簡単に実行できるようにするため、 JBoss Cache ディストリビューション内の **examples/cacheloader-migration** ディレクトリ以下にキャッシュローダーの移行例が格納されています。 **examples.TransformStore** の例は、 キャッシュに保存されている実際のデータとは独立し、 再帰的に読み取られたデータを書き戻します。 データをポートしたいユーザーは最初にこの例を実行することが推奨されます。 この例には例の詳細情報が記載されている **readme.txt** ファイルが含まれ、 独自のアプリケーションの基盤として使用できます。

9.4. キャッシュパッシベーション

キャッシュローダーを使用して、 キャッシュでのエビクションでノードのパッシベーションとアクティベーションを強制することが可能です。

キャッシュのパッシベーションは、 インメモリキャッシュからオブジェクトを削除し、 エビクション時に 2 次データストア (ファイルシステムやデータベースなど) に書き込むプロセスです。 **キャッシュのアクティベーション**は、 オブジェクトを使用する必要がある時にオブジェクトをデータストアからインメモリキャッシュへ復元するプロセスです。 どちらの場合でも、 データストアからの読み取りとデータストアへの書き込みには設定されたキャッシュローダーが使用されます。

有効なエビクションポリシーがキャッシュよりノードをエビクトすると、 パッシベーションが有効になっている場合はノードがパッシベートされたという通知がキャッシュリスナに送られ、 ノードとその子ノードがキャッシュローダーストアに保存されます。 エビクトされたノードをユーザーが読み出そうとすると、 ノードがキャッシュローダーストアよりメモリへロード (レイジーなロード) されます。 ロードされると、 ノードとその子ノードはキャッシュローダーより削除され、 ノードがアクティベートされたという通知がキャッシュリスナに送られます。

キャッシュのパッシベーションやアクティベーションを有効にするには、 **`passivation`** を true に設定します。 デフォルト値は **`false`** です。 パッシベーションが使用される場合は、 最初に設定されたキャッシュローダーのみが使用され、 他のキャッシュローダーは無視されます。

9.4.1. パッシベーションを無効にした場合と有効にした場合のキャッシュローダーの挙動

パッシベーションが無効な場合、 要素が変更、 追加、 削除されるとその変更内容がキャッシュローダーによってバックエンドストアに永続されます。 エビクションとキャッシュローディングに直接的な関係はありません。 エビクションを使用しない場合、 永続ストア内のデータは基本的にメモリ内のデータのコピーとなります。 エビクションを使用する場合、 永続ストア内のデータは基本的にメモリ内のデータのサブセットとなります (メモリからエビクトされたノードが含まれます)。

パッシベーションが有効な場合、 エビクションとキャッシュローダーに直接的な関係が存在します。 エビクションプロセスの一部としてのみキャッシュローダーによる永続ストアへの書き込みが発生します。 アプリケーションがメモリへ読み戻すとデータが永続ストアより削除されます。 この場合、 メモリ内のデータと永続ストア内のデータは全体的な情報セットの 2 つのサブセットとなり、 サブセットは交差しません。

次の簡単な例では、 6 つの手順のうち、 各手順が終了した後に RAM と永続ストアに存在するステートを表しています。

1. /A を挿入
2. /B を挿入
3. エビクションスレッドを実行、 /A をエビクト
4. /A を読み取り
5. エビクションスレッドを実行、 /B をエビクト
6. /B を削除

パッシベーションが無効の場合は次のようになります。

```
1) Memory: /A Disk: /A
2) Memory: /A, /B Disk: /A, /B
3) Memory: /B Disk: /A, /B
4) Memory: /A, /B Disk: /A, /B
5) Memory: /A Disk: /A, /B
6) Memory: /A Disk: /A
```

パッシベーションが有効な場合は次のようになります。

```
1) Memory: /A Disk:
2) Memory: /A, /B Disk:
3) Memory: /B Disk: /A
4) Memory: /A, /B Disk:
5) Memory: /A Disk: /B
6) Memory: /A Disk:
```

9.5. ストラテジ

本項では、 特定の成果を得るための異なるキャッシュローダータイプと設定オプションの組み合わせパターンについて説明します。

9.5.1. ローカルキャッシュとストア

これは最も単純なケースです。 キャッシュモードが **LOCAL** の JBoss Cache インスタンスを使用するため、 レプリケーションは行われません。 キャッシュローダーはストアから存在しない要素をロードし、 変更をストアに保存します。 キャッシュが起動されると、 **preload** 要素に応じて特定のデータが事前ロードされ、 キャッシュが部分的にウォームアップされます。

9.5.2. すべてのキャッシュが同じストアを共有するレプリケートされたキャッシュ

下図は同じバックエンドストアを共有する 2 つの JBoss Cache インスタンスを表しています。

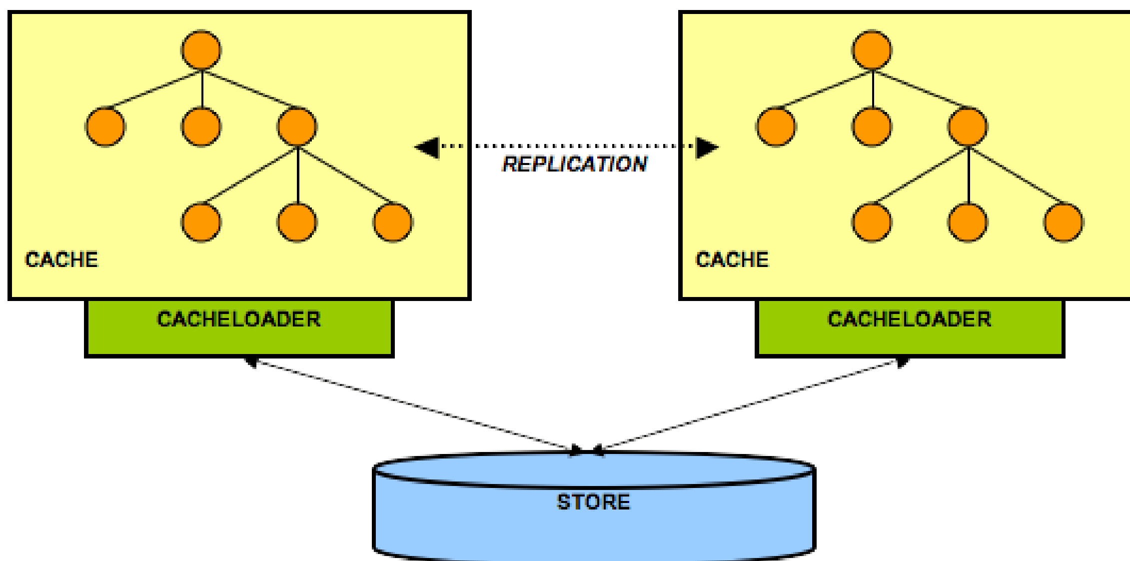
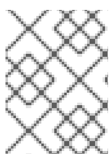


図9.2 バックエンドストアを共有する 2 つのノード

両方のノードは共有されたバックエンドストアにアクセスするキャッシュローダーを持っています。 例えば、 共有ファイルシステム (FileCacheLoader を使用) や共有データベースなどがこれに該当します。 両方のノードは同じストアにアクセスするため、 必ずしも起動時にステート転送を行う必要はありません。



注記

起動後にウォームキャッシュやホットキャッシュが必要な場合は、 ステート転送を有効にすることが可能です。

また、 **FetchInMemoryState** 属性を false に設定し、 要素が初めてアクセスされロードされたときに段階的にウォームアップする「コールド」キャッシュを使用することができます。 これにより、 クラスタ内の各キャッシュは異なるインメモリステートを持つことになります (事前ロードやエビクションストラテジに大きく依存します)。

値を保存する場合は、 書き込み側がバックエンドストアの変更の保存を担当します。 例えば、 node1 が変更 C1 を行い、 node2 が変更 C2 を行う場合、 node1 がキャッシュローダーにC1 を格納するよう指示し、 node2 がキャッシュローダーに C2 を格納するよう指示します。

9.5.3. ストアを持つキャッシュが 1 つしかないレプリケートされたキャッシュ

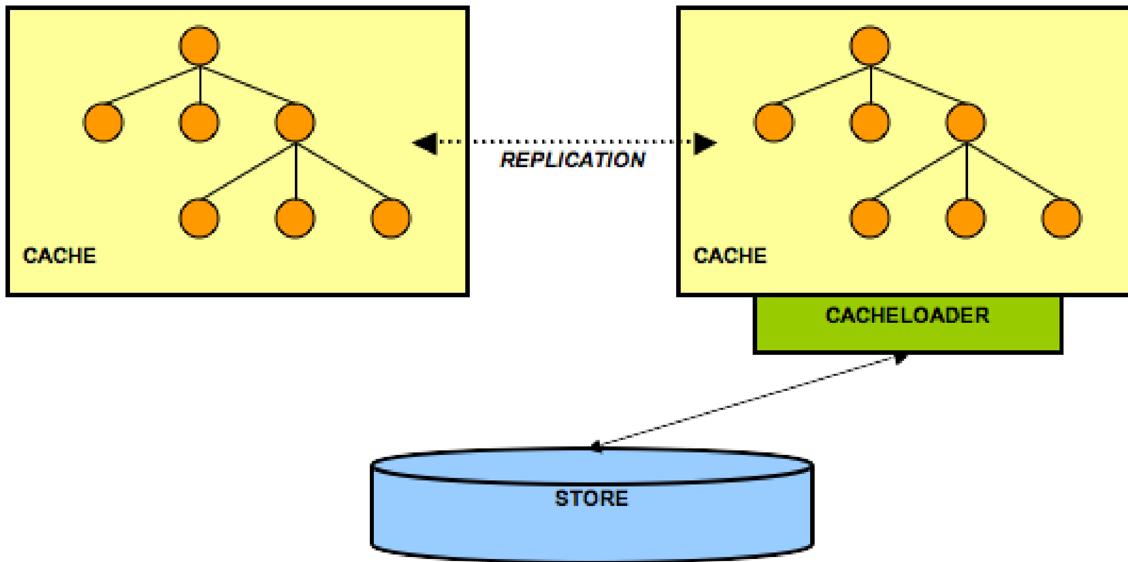


図9.3 2 つのノードのうち 1 つのみがバックエンドストアにアクセス

これは前のケースに似ていますが、 クラスタ内の 1 つのノードのみがキャッシュローダーよりバックエンドストアと対話します。 他のノードはすべてインメモリレプリケーションを実行します。 すべてのアプリケーションステートが各ノードのメモリに保持され、 複数のキャッシュが存在するため、データの高可用性を実現できます (データが必要なクライアントは 1 つのキャッシュから別のキャッシュへフェールオーバーできることが前提となります)。 クラスタ内のすべてのキャッシュが障害を起こしたり再起動が必要になった場合に、 単一の永続バックエンドストアがデータのバックアップコピーを提供します。

この場合、 データベースへのアクセスなどによってクラスタのパフォーマンスを低下させないために、キャッシュローダーが呼出側のスレッド上にない変更を非同期に保存するのが適切であると言えるでしょう。 非同期のレプリケーションを使用する場合は問題ありません。

このアーキテクチャの欠点は、 キャッシュローダーにアクセスするキャッシュが単一障害点になることです。 また、 クラスタが再起動された場合、 キャッシュローダーを持つキャッシュを最初に起動する必要があります (忘れがちな点です)。 単一障害点の問題に対応するには、 各ノードにキャッシュローダーを設定し、 **singletonStore** 設定を **true** に設定します。 この設定により、 1 つのノードのみが常に永続ストアへ書き込みすることになります。 しかし、 再起動する前にシャットダウンや障害が発生する前に書き込みを行っていたキャッシュを判断し、 そのキャッシュを最初に起動する必要があるため、 再起動の問題が複雑になります。

9.5.4. 各キャッシュが独自のストアを持つレプリケートされたキャッシュ

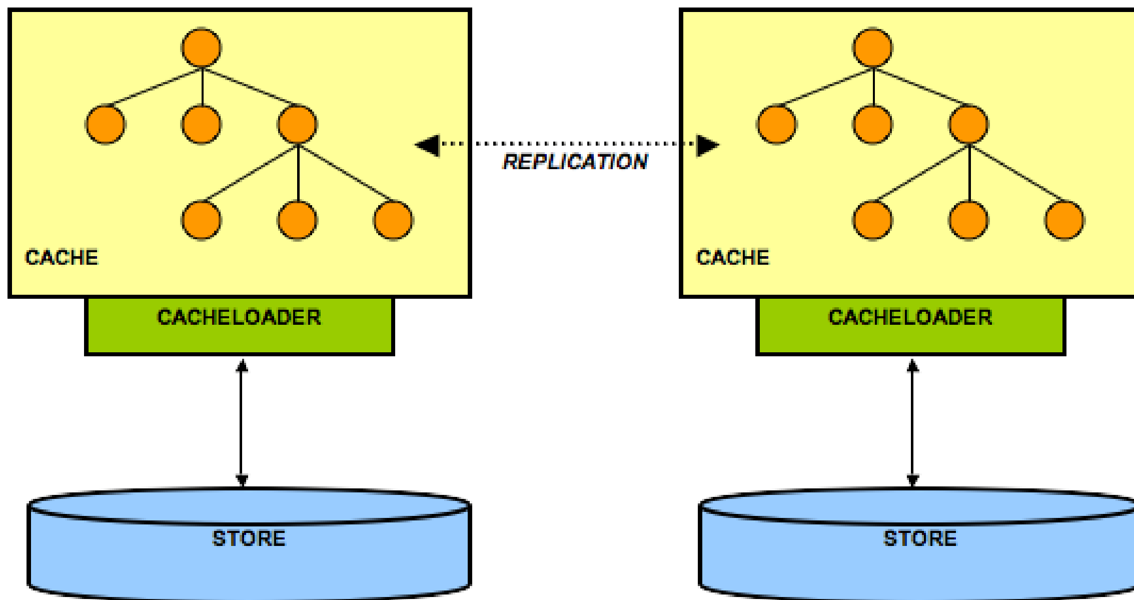


図9.4 各ノードが独自のバックエンドストアを持つ 2 つのノード

このケースでは、 各ノードが独自のデータストアを持ちます。 キャッシュへの変更は クラスタ全体でレプリケートされ、 キャッシュローダーを使用して永続化されます。 つまり、 すべてのデータストアが全く同じステートを持つことになります。 変更をトランザクションで同期的にレプリケートする場合、 2 相コミットプロトコルにより、 すべての変更がレプリケートされ各データストアに永続化されるか、 何もレプリケートされず永続化も行われません (アトミック更新)。

JBoss Cache は XAResource ではありません。 そのため、 復元を実装しません。 復元をサポートするトランザクションマネージャを使用する場合は、 この機能を使用できません。

ここでの問題はステート転送です。 新しいノードが起動した時に以下を実行する必要があります。

1. コーディネータ (クラスタ内の最も古いノード) にステートを送るよう指示します。 これは常に完全ステート転送となり、 既存のステートをすべて上書きします。
2. コーディネータは、未完了トランザクションが完了するまで待機する必要があります。この待機中は新しいトランザクションを開始できません。
3. 次に、 コーディネータは **`loadEntireState()`** を使用してキャッシュローダーに対して全体のステートを要求します。 キャッシュローダーは新しいノードにステートを送ります。
4. 新しいノードは、 古いステートを上書きして新しいステートをストアに保存するようキャッシュローダーに指示します。 これは **`CacheLoader.storeEntireState()`** メソッドです。
5. オプションとして、ステート転送中に一時 (インメモリ) ステートを転送することもできます。
6. これで、 新しいノードのバックエンドストアのステートが、 クラスタ内の他のノードと同じになります。 また、 他のノードから受け取った変更はローカルのキャッシュローダーを使用して永続化されるようになります。

9.5.5. 階層的キャッシュ

単一の JVM 内で階層を設定する必要がある場合は、 **`LocalDelegatingCacheLoader`** を使用します。 現在、 このタイプの階層はプログラムを使用してのみ設定が可能です。

また、 **`TcpDelegatingCacheLoader`** を使用すると、 複数の JVM やサーバーにまたがり階層キャッシュを設定することができます。

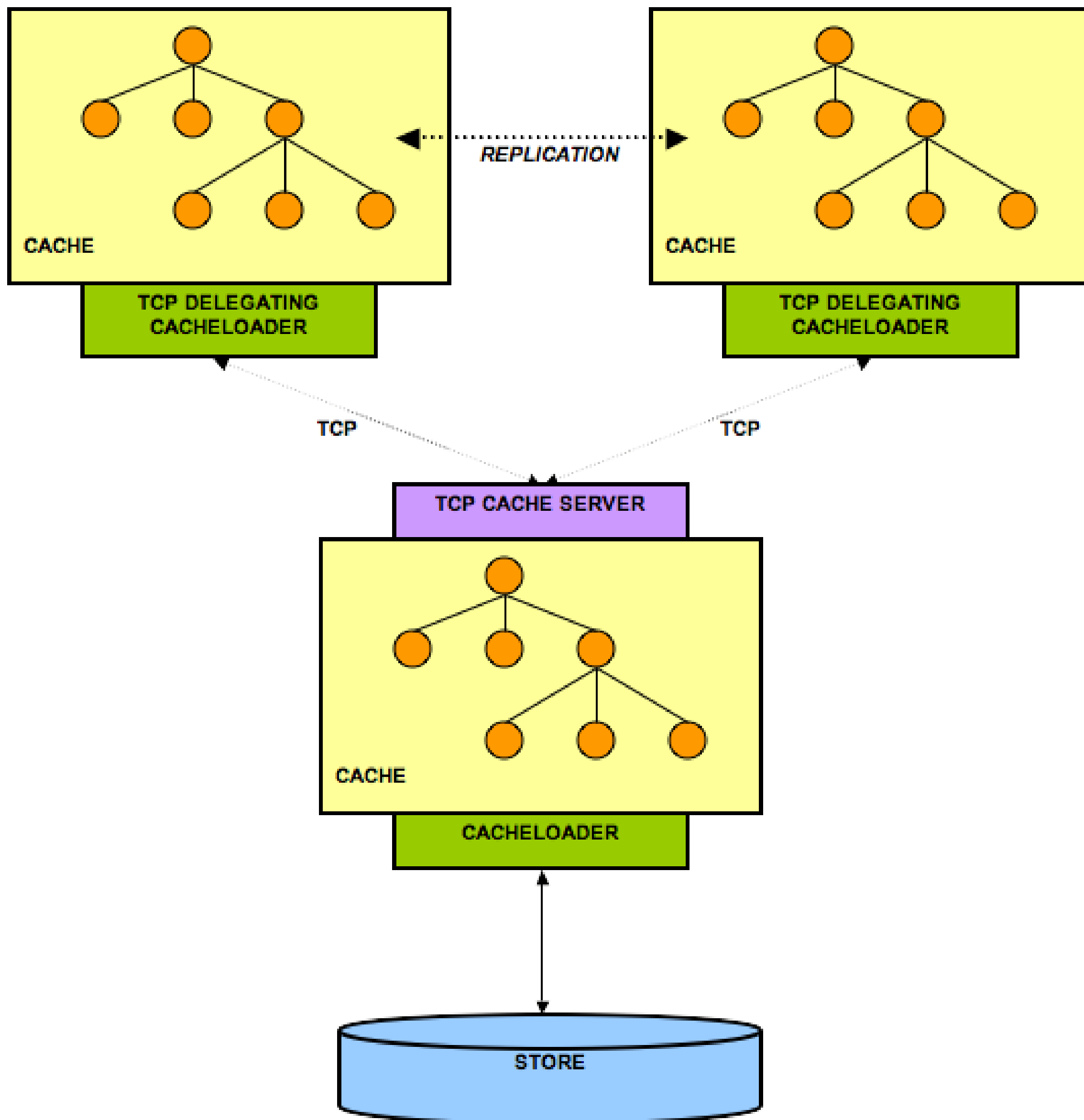


図9.5 TCP が委譲するキャッシュローダー

9.5.6. 複数のキャッシュローダー

1 つのチェーンに複数のキャッシュローダーを設定することができます。 内部的に、 設定した各キャッシュローダーへの参照を持つ、 委譲する **ChainingCacheLoader** が使用されます。 JVM と同じホスト上に存在するファイルシステムベースのキャッシュローダーをメモリのオーバーフローとして使用する例があります。 これにより、 低負荷で比較的簡単にデータを使用できるようにします。 **TcpDelegatingCacheLoader** などの追加のリモートキャッシュローダーは、 サーバー再起動の間の回復機能を提供します。

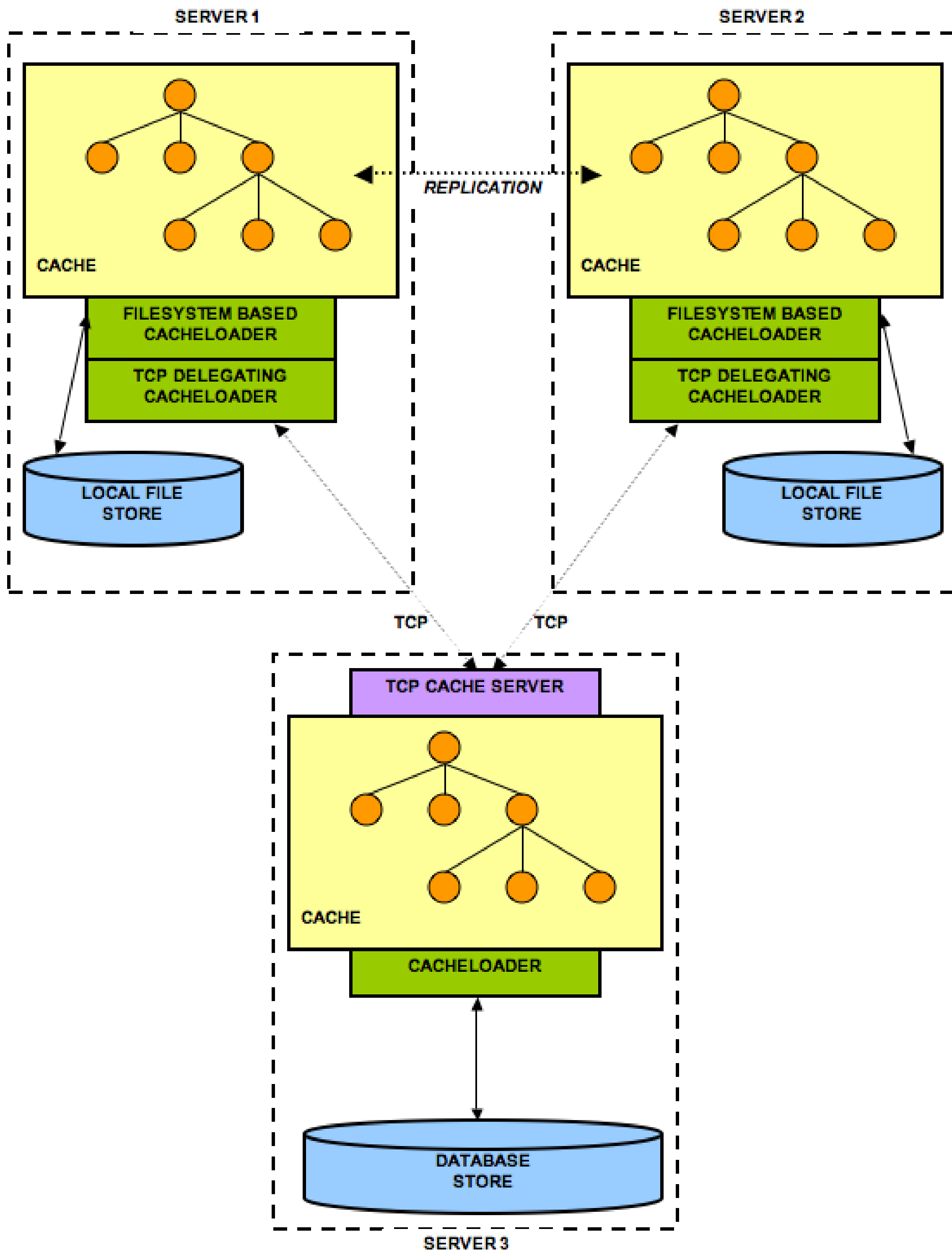


図9.6 チェーンにある複数のキャッシュローダー

第10章 エビクション

エビクションはメモリに保存されるノード数やノードがメモリに保存される期間を制限して JBoss Cache のメモリ管理を制御します。 サーバー上のメモリを制約することにより、 キャッシュが無制限に増加できなくなるため、 メモリ不足エラーが発生しないよるエビクションが必要となります。 エビクションは通常 9章キャッシュローダー と共に使用されます。

10.1. 構造

JBoss Cache のエビクションは 4 つの概念に基づいています。

- 1. 統計の収集
- 2. エビクトするノードの判定
- 3. ノードをエビクトする方法
- 4. エビクションスレッド

また、 キャッシュのサブツリーによってエビクションの特性が異なるよう、 エビクションは常にリージョンごとに設定されるため、 リージョンはエビクションで重要な役割を担います。

10.1.1. 統計の収集

キャッシュと対話が行われる度に呼出側のスレッド上で統計が収集されます。 エビクションが有効になっていると、 **`EvictionInterceptor`** がインターセプタチェーンに追加され、 イベントキュー上でイベントが記録されます。 イベントは **`EvictionEvent`** クラスで表されます。 イベントキューは特定のリージョンで実行されるため、 各リージョンは独自のイベントキューを持っています。

エビクションが有効になっているかに応じて **`EvictionInterceptor`** をインターセプタチェーンに追加するかを設定できる以外、 エビクションのこの側面は設定可能ではありません。

10.1.2. エビクトするノードの判定

`EvictionAlgorithm` 実装はエビクションキューを処理し、 エビクトするキューを決定します。 JBoss Cache には **`FIFOAlgorithm`** や**`LRUAlgorithm`**、 **`LFUAlgorithm`** などを含む複数の実装が同梱されています。 各実装には、ア ルゴリズムの設定詳細と共に対応する **`EvictionAlgorithmConfig`** 実装が含まれています。

インターフェースを実装するか提供されている実装の 1 つを拡張すると、 カスタムの **`EvictionAlgorithm`** 実装を提供することができます。

`process()` メソッドを呼び出し、 処理するためイベントキューに渡すことでアルゴリズムが実行されます。 通常、 リージョンに割り当てられたアルゴリズムを見つける **`Region.processEvictionQueues()`** を呼び出します。

10.1.3. ノードをエビクトする方法

`EvictionAlgorithm` がエビクトするノードを決定すると、 **`EvictionActionPolicy`** の実装を使用してノードをエビクトする方法を判断します。 これはリージョンごとに設定可能で、 エビクトする必要がある各ノードの **`Cache.evict()`** を呼び出す **`DefaultEvictionActionPolicy`** がデフォルトになります。

Cache.evict() の代わりに、 エビクトする必要のある各ノードの **Cache.removeNode()** を呼び出す **RemoveOnEvictActionPolicy** も JBoss Cache に同梱されます。

カスタムの **EvictionActionPolicy** 実装も使用することができます。

10.1.4. エビクションスレッド

デフォルトでは、 定期的に登録されたリージョンを反復し、 各リージョンで **Region.processEvictionQueues()** を呼び出すため、 単一のキャッシュ全体のエビクションスレッドが使用されます。 このスレッドが実行される頻度は、 **eviction** 設定要素の **wakeUpInterval** 属性を使用して設定可能で、 指定がない場合は 5000 ミリ秒がデフォルトとなります。

エビクションスレッドを無効にするには、 **wakeUpInterval** を **0** に設定します。 これは、 独自の定期メンテナンススレッドを実行し、 リージョンを通じて反復を行い **Region.processEvictionQueues()** を呼び出す場合に便利です。

10.2. エビクションリージョン

マーシャリングでは、 リージョンの概念と **Region** クラスは 「クラスローディングとリージョン」 でした。 リージョン内のノードのエビクション動作を定義するためにもリージョンは使用されます。 リージョン固有の設定を使用するだけでなく、 特定のリージョンを定義しなくない場合や、 事前定義されたリージョンに属さないノードに対してデフォルトのキャッシュ全体のエビクション動作を設定することもできます。 設定 XML ファイルを使用してリージョンを定義する場合、 リージョンを定義する **Fqn** の全要素は **String** オブジェクトになります。

各リージョンに対してエビクションパラメータを定義することができます。

重複するリージョンを定義することも可能です。 例えば、 1 つのリージョンを **/a/b/c** に定義し 、別のリージョンを **/a/b/c/d** (**/a/b/c** サブツリーの **d** サブツリー) に定義することが可能です。 アルゴリズムはこのような例を一貫して処理するため、 常に最初に遭遇したリージョンを選択します。 そのため、 ノード **/a/b/c/d/e** の処理方法をアルゴリズムが決定する必要がある場合 、定義されたリージョンを最初に見つけるまでツリーを上がります (この例の場合は **/a/b/c/d**)。

10.2.1. 常駐ノード

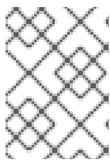
エビクションをトリガするか確認する時と実際にノードのエビクションを開始する時、 常駐と示されているノード (**Node.setResident()** API を使用) はエビクションポリシーによって無視されます。 例えば、 リージョンの最大ノード数が 10 に設定されている場合、 そのリージョンでノードをエビクトするか判定する際に常駐ノードは数に含まれません。 また、 リージョンのエビクションがしきい値に達した時、 常駐ノードはエビクションの対象として考慮されません。

ノードを常駐とするには、 **Node.setResident()** API を使用する必要があります。 デフォルトでは、 新規作成されたノードは常駐ノードにはなりません。 ノードの **resident** 属性はレプリケートや永続化できず、 トランザクションを意識しません。

次の常駐ノードのユースケースは、 「パス」 ノードがエビクションポリシーに「雑音」を追加しないようにします。

```
...
   Map lotsOfData = generateData();
   cache.put("/a/b/c", lotsOfData);
   cache.getRoot().getChild("/a").setResident(true);
   cache.getRoot().getChild("/a/b").setResident(true);
...
```

この例では、 **/a** ノードと **/a/b** ノードはパスで、 **/a/b/c** ノードの存在をサポートするためのみに存在し、 データは保有しません。 そのため、 常駐ノードとするのに最適です。 **/a** や **/a/b** へアクセスする際にエビクションイベントが生成されないため、 メモリ管理が向上されます。



注記

常駐ノードへ属性を追加する場合 (上記の例では **cache.put("/a", "k", "v")**)、 常駐ノードを非常駐ノードに戻し、 エビクションの対象となるようにしてください。

10.3. エビクションの設定

10.3.1. 基本設定

基本的なエビクション設定要素は次のようになります。

```
...
<eviction wakeUpInterval="500" eventQueueSize="100000">
   <default algorithmClass="org.jboss.cache.eviction.LRUAlgorithm">
      <property name="maxNodes" value="5000" />
      <property name="timeToLive" value="1000" />
   </default>
</eviction>
...
```

- **wakeUpInterval** - 必須のパラメータで、 エビクションスレッドが実行される頻度をミリ秒単位で定義します。
- **eventQueueSize** - 任意のパラメータで、 エビクションイベントを保持するバインドされたキューのサイズを定義します。 エビクションスレッドが十分な頻度で実行されないと、 イベントキューが満杯になることがあります。 この場合、 エビクションスレッドが実行される頻度を増やすか、 イベントキューのサイズを大きくします。 この設定は**デフォルト**のイベントキューのサイズとなりますが、 特定のエビクションリージョンでオーバーライドすることが可能です。 指定がない場合、 **200000** がデフォルトとなります。
- **algorithmClass** - すべてのリージョンへ個別に **algorithmClass** 属性を設定しない限り、 必須のパラメータとなります。 このパラメータがリージョンに対して定義されていない場合に、 デフォルトで使用されるエビクションアルゴリズムを定義します。
- アルゴリズム設定属性 - **algorithmClass** で指定されるアルゴリズムに固有の属性です。 詳細は、 特定のアルゴリズムの項を参照してください。

10.3.2. プログラムを用いた設定

Configuration オブジェクトを使用してエビクションを設定すると、 **Configuration.setEvictionConfig()** へ渡される **org.jboss.cache.config.EvictionConfig** Bean が使用されます。 プログラムを用いて **Configuration** をビルドする方法の詳細は 3章*設定* を参照してください。

簡単な POJO Bean を使用して、 キャッシュの設定ですべての要素を表すと、 比較的簡単にキャッシュの起動後にプログラムを用いてエビクションリージョンを追加することができます。 例えば、 上記の EvictionConfig 要素で XML より設定したキャッシュが存在するとします。 ランタイム時に、 異なる **maxNodes** の数で **LRUAlgorithm** を使用して "/org/jboss/fifo" という名前の新しいエビクションリージョンを追加する場合は次のようになります。

```
Fqn fqn = Fqn.fromString("/org/jboss/fifo");

// Create a configuration for an LRUPolicy
LRUAlgorithmConfig lruc = new LRUAlgorithmConfig();
lruc.setMaxNodes(10000);

// Create an eviction region config
EvictionRegionConfig erc = new EvictionRegionConfig(fqn, lruc);

// Create the region and set the config
Region region = cache.getRegion(fqn, true);
region.setEvictionRegionConfig(erc);
```

10.4. 同梱されるエビクションポリシー

本項では JBoss Cache に同梱されるアルゴリズムや各アルゴリズムに使用される設定パラメータについて説明します。

10.4.1. LRUAlgorithm - 最長時間未使用

org.jboss.cache.eviction.LRUAlgorithm はノードの存続時間と経過時間の両方を制御します。 このポリシーによって、 追加、 削除、 検索 (ビジット) に対し一定順序 (**O (1)**) が保証されます。 このポリシーが持つ設定パラメータは次の通りです。

- **maxNodes** - このリージョンで許可されるノードの最大数になります。 0 は即時失効、 -1 は無制限を表します。
- **timeToLive** - この期間 (ミリ秒単位) ノードが読み書きされないとノードが削除されます。 0 は即時失効、 -1 は無制限を表します。
- **maxAge** - ノードが削除されるまでのアイドル時間が関係しないノードの寿命 (ミリ秒単位) になります。 0 は即時失効、 -1 は無制限を表します。
- **minTimeToLive** - エビクションの対象となる前に、 アクセスされた後ノードが存在しなければならない最短期間になります。 0 はこの機能の無効を表し、 デフォルト値となります。

10.4.2. FIFOAlgorithm - 先入れ先出し

org.jboss.cache.eviction.FIFOAlgorithm は適切な先入れ先出しの順序でエビクションを制御します。 このポリシーによって、 追加、 削除、 検索 (ビジット) に対し一定順序 (**O (1)**) が保証されます。 このポリシーが持つ設定パラメータは次の通りです。

- **maxNodes** - このリージョンで許可されるノードの最大数になります。 0 は即時失効、 -1 は無制限を表します。
- **minTimeToLive** - エビクションの対象となる前に、 アクセスされた後ノードが存在しなければならない最短期間になります。 0 はこの機能の無効を表し、 デフォルト値となります。

10.4.3. MRUAlgorithm - 最も最近使用

org.jboss.cache.eviction.MRUAlgorithm は最も最近使用されたアルゴリズムを基にエビクションを制御します。 このポリシーでは、 最も最近使用されたノードが最初にエビクトされます。 このポリシーによって、 追加、 削除、 検索 (ビジット) に対し一定順序 (**O (1)**) が保証されます。 この

ポリシーが持つ設定パラメータは次の通りです。

- **maxNodes** - このリージョンで許可されるノードの最大数になります。 0 は即時失効、 -1 は無制限を表します。

- **minTimeToLive** - エビクションの対象となる前に、 アクセスされた後ノードが存在しなければならない最短期間になります。 0 はこの機能の無効を表し、 デフォルト値となります。

10.4.4. LFUAlgorithm - 使用頻度が最低

org.jboss.cache.eviction.LFUAlgorithm は使用頻度が最も低いアルゴリズムを基にエビクションを制御します。 このポリシーでは、 使用頻度が最も低いノードが最初にエビクトされます。 ノードが最初に追加されると、 ノードの使用は 1 から始まります。 ノードにアクセスするたびに、 ノードの使用カウンタが 1 つ増えます。 この数は、 使用頻度が最も低いノードを決定するために使用されます。 また、 LFU はソートされたエビクションアルゴリズムです。 基礎となる EvictionQueue 実装とアルゴリズムは、 ノードビジットカウンタの昇順でソートされます。 このポリシーによって、 追加、 削除、 検索 (ビジット) に対し一定順序 (**O (1)**) が保証されます。 ただし、 任意数のノードが該当する処理パスのキューに追加またはビジットされる場合、 適切な LFU の順序でキューを再ソートするために単一の準線形 (**O (n * log n)**) 操作が使用されます。 同様に、 ノードが削除またはエビクトされる場合は、 EvictionQueue をクリーンアップするために単一の線形 (**O (n)**) 切り捨て処理が必要です。 LFU は次のパラメータを持ちます。

- **maxNodes** - このリージョンで許可されるノードの最大数になります。 0 は即時失効、 -1 は無制限を表します。

- **minNodes** - このリージョンで許可されるノードの最小数になります。 この値は、 エビクションキューがパスごとに切り捨てるノードの数を決定します。 例えば、 minNodes が 10 で、 キャッシュが 100 個のノードに増加すると、 エビクションタイマーがエビクションアルゴリズムよりパスした時に最も使用頻度が高い 10 個のノードにキャッシュが切り捨てられます。

- **minTimeToLive** - エビクションの対象となる前に、 アクセスされた後ノードが存在しなければならない最短期間になります。 0 はこの機能の無効を表し、 デフォルト値となります。

10.4.5. ExpirationAlgorithm

org.jboss.cache.eviction.ExpirationAlgorithm は絶対有効期限を基にノードをエビクトするポリシーです。 有効期限は **org.jboss.cache.Node.put()** メソッドを使用して表示されます。 **org.jboss.cache.Node.put()** メソッドは文字列キー **expiration** と絶対時間を **java.lang.Long** オブジェクトとして使用し、 値は UTC (協定世界時) 1970 年 1 月 1 日午前 0 時から経過した時間 (ミリ秒単位) で表示されます (**java.lang.System.currentTimeMillis()** によって提供される相対時間と同じです)。

このポリシーによって、 追加と削除に対し一定順序 (**O (1)**) が保証されます。 内部的に、 ノードの有効期限と FQN が含まれるソートされたセット (TreeSet) が保存され、 基本的にヒープとして機能します。

このポリシーが持つ設定パラメータは次の通りです。

- **expirationKeyName** - エビクションアルゴリズムで使用されるノードキー名です。 設定のデフォルトは **expiration** になります。

- **maxNodes** - このリージョンで許可されるノードの最大数になります。 0 は即時失効、 -1 は無制限を表します。

次の一覧は、 有効期限の表示方法とポリシーの適用方法を表しています。

```
    Cache cache = DefaultCacheFactory.createCache();
    Fqn fqn1 = Fqn.fromString("/node/1");
    Long future = new Long(System.currentTimeMillis() + 2000);

    // sets the expiry time for a node

 cache.getRoot().addChild(fqn1).put(ExpirationConfiguration.EXPIRATION_KEY,
 future);

    assertTrue(cache.getRoot().hasChild(fqn1));
    Thread.sleep(5000);

    // after 5 seconds, expiration completes
    assertFalse(cache.getRoot().hasChild(fqn1));
```

ノードの有効期限はリージョンマネージャが **wakeUpIntervalSeconds** ごとにウェイクアップする時のみチェックされるため、 エビクションは表示されている時間の数秒後に発生する可能性があります。

10.4.6. ElementSizeAlgorithm - ノードにあるキーと値のペアの数を基にしたエビクション

org.jboss.cache.eviction.ElementSizeAlgorithm はノードにあるキーと値のペアの数を基にエビクションを制御します。 このポリシーでは、 最も最近使用されたノードが最初にエビクトされます。 このポリシーが持つ設定パラメータは次の通りです。

- **maxNodes** - このリージョンで許可されるノードの最大数になります。 0 は即時失効、 -1 は無制限を表します。

- **maxElementsPerNode** - エビクション対象として選択されるノードに対するノード毎の属性のトリガ数です。 0 は即時失効、 -1 は無制限を表します。

- **minTimeToLive** - エビクションの対象となる前に、 アクセスされた後ノードが存在しなければならない最短期間になります。 0 はこの機能の無効を表し、 デフォルト値となります。

第11章 トランザクションと並行性

11.1. 同時アクセス

JBoss Cache はスレッドセーフのキャッシング API で、 独自の効率的なメカニズムを使用して同時アクセスを制御します。 MVCC (Multi-Version Concurrency Control) の革新的な実装をデフォルトのロックスキームとして使用します。 Boss Cache 3.x 以前のバージョンでは楽観的ロッキングスキームと悲観的ロッキングスキームが提供されていましたが、 MVCC の導入に伴いこれらのスキームは廃止されました。

11.1.1. MVCC (Multi-Version Concurrency Control)

MVCC は、 共有データへの高速で安全な同時アクセスを制御するため最新のデータベース実装によって一般的に使用されるロックスキームです。

11.1.1.1. MVCC の概念

MVCC は同時アクセスに対して次のような機能を提供します。

- ライターをブロックしないリーダー
- フェイルファストなライター

同時ライターにデータのバージョン化とコピーを使用してこれを実行します。 理論としては、 リーダーが共有ステートの読み取りを継続する間にライターが共有ステートをコピーし、 バージョン ID の値を増やしてバージョンが有効であることを検証した後 (別の同時ライターが先にステートを変更していないかなど) 共有ステートを書き戻すことになります。

これにより、 ライターの書き込みを妨害せずにリーダーは読み取りを継続することができます。 また、 リーダーがステートの古いバージョンを読み取れるため、 繰り返し可能な読み取りセマンティックが維持されます。

11.1.1.2. MVCC 実装

MVCC の JBoss Cache 実装は一部の機能が基になっています。

- リーダーはロックを取得しない
- 単一のライターに対し、 共有ステートの 1 つの追加バージョンのみが維持される
- フェイルファストセマンティックを提供するため、 すべてのライターは順次である

スレッドの読み取りに対し、 JBoss Cache の MVCC 実装の大変優れたパフォーマンスを実現するには、 リーダーの同期やロッキングを必要としません。 各リーダースレッドに対して、 **`MVCCLockingInterceptor`** がスレッドの **`InvocationContext`** (トランザクションを実行している場合は **`TransactionContext`**) にあるライトウエイトコンテナオブジェクトのステートをラッピングします。 ステート上の後続操作はすべてコンテナオブジェクトより発生します。 Java の参照をこのように使用することにより、 実際のステートが同時に変更しても読み取りセマンティックを繰り返し可能にすることができます。

ライタースレッドは書き込みを開始する前にロックを取得する必要があります。 現在、 メモリのパフォーマンスを向上するためにロックストライピングを使用しています。 共有ロックプールの大きさは **`locking`** 要素の **`concurrencyLevel`** 属性を使用して調整することが可能です。 詳細は 12章*設定に関する参考資料* を参照してください。 FQN で排他的ロックを取得した後、 リーダースレッドと同様にライタースレッドがコンテナで変更されるステートをラッピングし、 書き込みに対してステートをコピー

します。 コピーする際、 元のバージョンへの参照はコンテナ内で維持されます (ロールバックのため)。 その後、 コピーへ変更が加えられ、 書き込みが終了するとコピーが最終的にデータ構造へ書き込まれます。

これにより、 後続のリーダーは新しいバージョンを認識し、 既存のリーダーはコンテキストに元のバージョンへの参照を保持することができます。

一定期間にライターが書き込みロックを取得できない場合、 `TimeoutException` がスローされます。 ロック取得タイムアウトのデフォルト値は 10000 ミリ秒で、 `locking` 要素の `lockAcquisitionTimeout` 属性を使用して設定することができます。 詳細は 12章*設定に関する参考資料* を参照してください。

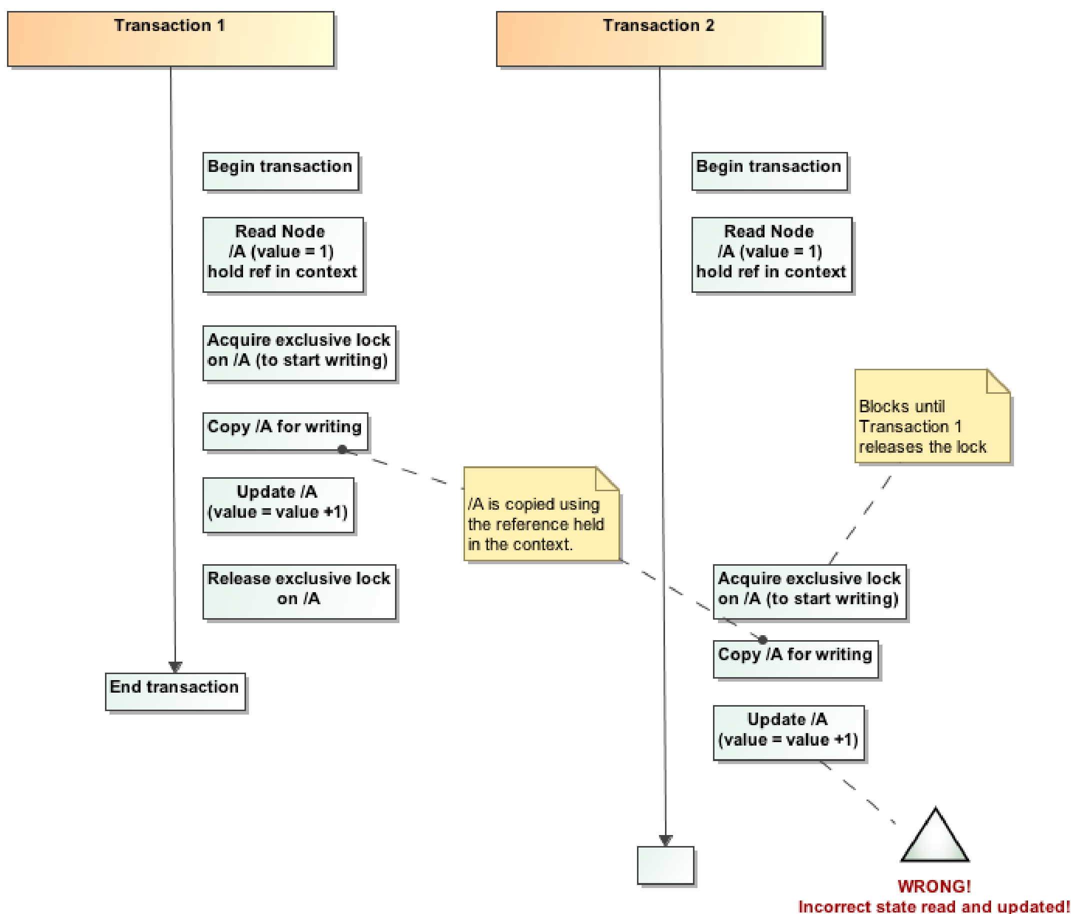
11.1.1.2.1. 分離レベル

JBoss Cache 3.x は REPEATABLE_READ と READ_COMMITTED の 2 つの分離レベルをサポートします。 これらのセマンティックはデータベース分離レベルに相当します。 JBoss Cache の以前のバージョンは 5 つのデータベース分離レベルすべてをサポートしていました。 サポート対象外の分離レベルが設定されると、 最も近いサポート対象レベルにアップグレードまたはダウングレードされます。

以前のバージョンの JBoss Cache との互換性を維持するため、 REPEATABLE_READ がデフォルトの分離レベルとなっています。 READ_COMMITTED の分離は若干弱くなりますが、 REPEATABLE_READ よりもはるかにパフォーマンスが良くなります。

11.1.1.2.2. 同時書き込みと書き込みのスキュー

MVCC はライターに書き込みロックの取得を強制しますが、 REPEATABLE_READ 使用時に書き込みスキューと呼ばれる現象が発生することがあります。



この現象は、 読み取りとなっていた値を基に並行トランザクションが読み取りの後に書き込みを実行すると発生します。 読み取りには、 トランザクションコンテキストにあるステートへの参照の保持が関係するため、 陳腐化した可能性のある元のステート読み取りを使用して後続の読み取りが行われることがあります。

書き込みのためステートをコピーする際に書き込みスキューが検出されると、 デフォルトの動作として **DataVersioningException** がスローされます。 しかし、 ほとんどのアプリケーションでは書き込みスキューは問題となりません (書き込まれたステートが最初に読み取られたステートと関係ない場合など)。 アプリケーションで問題とならない場合は、 **writeSkewCheck** 設定属性を **false** に設定し、書き込みスキューを許可します。 詳細は 12章*設定に関する参考資料* を参照してください。

スレッドはコミット済みステート常に使用するため、 READ_COMMITTED を使用する場合は書き込みスキューは発生しません。

11.1.1.3. ロッキングの設定

MVCC を設定するには、次のように **<locking />** 設定タグを使用する必要があります。

```
<locking
   isolationLevel="REPEATABLE_READ"
   lockAcquisitionTimeout="10234"
   nodeLockingScheme="mvcc"
   writeSkewCheck="false"
   concurrencyLevel="1000" />
```

- **nodeLockingScheme** - 使用されるノードロッキングスキームです。 指定がないとデフォルトとして MVCC が適用されます。 **pessimistic** や **optimistic** などの廃止されたスキームも使用できますが、 推奨されません。
- **isolationLevel** - トランザクションレベルです。 指定がない場合はデフォルトとして REPEATABLE_READ が適用されます。
- **writeSkewCheck** - 指定がない場合は **true** がデフォルトとして適用されます。
- **concurrencyLevel** - 指定がない場合はデフォルトとして 500 が適用されます。
- **lockAcquisitionTimeout** - MVCC を使用する時にライターのみに適用されます。 指定がない場合はデフォルトとして 10000 が適用されます。

11.1.2. 楽観的および悲観的ロッキングスキーム

JBoss Cache 3.x より 「MVCC (Multi-Version Concurrency Control)」 が導入されたため、 悲観的ロッキングスキームと楽観的ロッキングスキームは廃止されました。 将来的にサポートが終了するため、既存のアプリケーションにはレガシーなロッキングスキームを使用しないことが推奨されます。

本ユーザーガイドにはレガシーなロッキングスキームのドキュメントは含まれていません。 必要な場合は、 JBoss Cache ウェブサイトにある本書の旧バージョンを参照してください。

11.2. JTA サポート

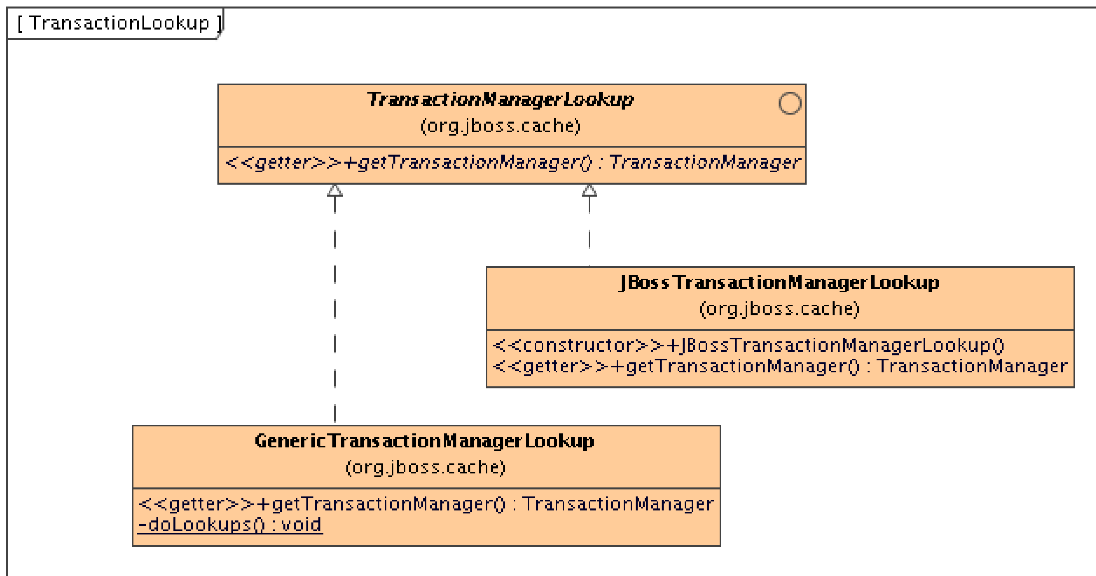
JTA 対応のトランザクションを使用し参加するよう、 JBoss Cache を設定することができます。 また、 トランザクションサポートの無効化は、 JDBC 呼び出しにおける自動コミットの使用に相当します。 この場合、 変更される度に変更がレプリケートされる可能性があります (レプリケーションが有効になっている場合)。

JBoss Cache は着信呼び出し毎に以下を実行します。

1. スレッドに関連する現在の **javax.transaction.Transaction** を読み出します。

2. トランザクションのコミットやロールバック時に通知されるよう、 登録されていない場合は **javax.transaction.Synchronization** をトランザクションマネージャに登録します。

登録するには、 環境の **javax.transaction.TransactionManager** への参照をキャッシュに提供する必要があります。 これには、 通常 **TransactionManagerLookup** インターフェース実装のクラス名でキャッシュを設定します。 キャッシュが起動すると、 このクラスのインスタンスを作成し、 **TransactionManager** への参照を返す **getTransactionManager()** メソッドを呼び出します。



JBoss Cache には **JBossTransactionManagerLookup** と **GenericTransactionManagerLookup** が同梱されます。 **JBossTransactionManagerLookup** は実行されている JBoss AS インスタンスにバインドでき、 **TransactionManager** を取得できます。 **GenericTransactionManagerLookup** は一般的な Java EE アプリケーションサーバーのほとんどにバインドでき、 同様の機能を提供できます。 また、 単体テストのためダミーの実装である **DummyTransactionManagerLookup** も提供されます。 ただし、 ダミーは、 同時トランザクションやリカバリに対して厳しい制限があるため、 本稼働用に使用することは推奨されません。

TransactionManagerLookupを設定する代わりに、 プログラムを用いて **TransactionManager** への参照を **Configuration** オブジェクトの **RuntimeConfig** 要素へ挿入することもできます。

```
TransactionManager tm = getTransactionManager(); // magic method
cache.getConfiguration().getRuntimeConfig().setTransactionManager(tm);
```

TransactionManager への参照を既に持っている IOC コンテナによって **Configuration** が構築されている場合、 **TransactionManager** を挿入する方法が推奨されます。

トランザクションのコミットを実行すると、 単相または 2 相コミットプロトコルが開始されます。 詳細は「レプリケートされたキャッシュトランザクション」を参照してください。

パート III. JBOSS CACHE の設定に関する参考資料

本項は検索を容易にするための、 技術的な参考資料となります。

第12章 設定に関する参考資料

12.1. XML 設定ファイルの例

典型的な XML 設定ファイルは次のようになります。 最初から設定ファイルを作成するのではなく、JBoss Cache ディストリビューションに同梱される設定の 1 つを使用し、 必要に応じて調整することが推奨されます。

```
<?xml version="1.0" encoding="UTF-8"?>

<jbosscache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="urn:jboss:jbosscache-core:config:3.1">



   <!--
      isolation levels supported: READ_COMMITTED and REPEATABLE_READ
      nodeLockingSchemes: mvcc, pessimistic (deprecated), optimistic
(deprecated)
   -->
   <locking
         isolationLevel="REPEATABLE_READ"
         lockParentForChildInsertRemove="false"
         lockAcquisitionTimeout="20000"
         nodeLockingScheme="mvcc"
         writeSkewCheck="false"
         useLockStriping="true"
         concurrencyLevel="500"/>

   <!--
      Used to register a transaction manager and participate in ongoing
transactions.
      -->
   <transaction

transactionManagerLookupClass="org.jboss.cache.transaction.GenericTransact
ionManagerLookup"
         syncRollbackPhase="false"
         syncCommitPhase="false"/>

   <!--
      Used to register JMX statistics in any available MBean server
      -->
   <jmxStatistics
         enabled="false"/>

   <!--
      If region based marshalling is used, defines whether new regions are
inactive on startup.
   -->
   <startup
         regionsInactiveOnStartup="true"/>

   <!--
      Used to register JVM shutdown hooks.
      hookBehavior: DEFAULT, REGISTER, DONT_REGISTER
```

```
   -->
   <shutdown
         hookBehavior="DEFAULT"/>

   <!--
      Used to define async listener notification thread pool size
   -->
   <listeners
         asyncPoolSize="1"
         asyncQueueSize="100000"/>

   <!--
      Used to enable invocation batching and allow the use of
Cache.startBatch()/endBatch() methods.
   -->
   <invocationBatching
         enabled="false"/>

   <!--
      serialization related configuration, used for replication and cache
loading
   -->
   <serialization
         objectInputStreamPoolSize="12"
         objectOutputStreamPoolSize="14"
         version="3.0.0"
         marshallerClass="org.jboss.cache.marshall.VersionAwareMarshaller"
         useLazyDeserialization="false"
         useRegionBasedMarshalling="false"/>

   <!--
      This element specifies that the cache is clustered.
      modes supported: replication (r) or invalidation (i).
   -->
   <clustering mode="replication" clusterName="JBossCache-cluster">

      <!--
         Defines whether to retrieve state on startup
      -->
      <stateRetrieval timeout="20000" fetchInMemoryState="false"/>

      <!--
         Network calls are synchronous.
      -->
      <sync replTimeout="20000"/>
      <!--
         Uncomment this for async replication.
      -->
      <!--<async useReplQueue="true" replQueueInterval="10000"
replQueueMaxElements="500" serializationExecutorPoolSize="20"
serializationExecutorQueueSize="5000000"/>-->

      <!-- Uncomment to use Buddy Replication -->
      <!--
      <buddy enabled="true" poolName="myBuddyPoolReplicationGroup"
communicationTimeout="2000">
```

```
         <dataGravitation auto="true" removeOnFind="true"
searchBackupTrees="true"/>
         <locator
class="org.jboss.cache.buddyreplication.NextMemberBuddyLocator">
            <properties>
               numBuddies = 1
               ignoreColocatedBuddies = true
            </properties>
         </locator>
      </buddy>
      -->

      <!--
         Configures the JGroups channel.  Looks up a JGroups config file
on the classpath or filesystem.  udp.xml
         ships with jgroups.jar and will be picked up by the class loader.
      -->
      <jgroupsConfig configFile="udp.xml">
         <!-- uncomment to define a JGroups stack here

         <PING timeout="2000" num_initial_members="3"/>
         <MERGE2 max_interval="30000" min_interval="10000"/>
         <FD_SOCK/>
         <FD timeout="10000" max_tries="5" shun="true"/>
         <VERIFY_SUSPECT timeout="1500"/>
         <pbcast.NAKACK use_mcast_xmit="false" gc_lag="0"
                        retransmit_timeout="300,600,1200,2400,4800"
                        discard_delivered_msgs="true"/>
         <UNICAST timeout="300,600,1200,2400,3600"/>
         <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
                        max_bytes="400000"/>
         <pbcast.GMS print_local_addr="true" join_timeout="5000"
shun="false"
                     view_bundling="true"
view_ack_collection_timeout="5000"/>
         <FRAG2 frag_size="60000"/>
         <pbcast.STREAMING_STATE_TRANSFER use_reading_thread="true"/>
         <pbcast.FLUSH timeout="0"/>
          -->
      </jgroupsConfig>
   </clustering>

   <!--
      Eviction configuration.  WakeupInterval defines how often the
eviction thread runs, in milliseconds.  0 means
      the eviction thread will never run.
   -->
   <eviction wakeUpInterval="500">
      <default algorithmClass="org.jboss.cache.eviction.LRUAlgorithm"
eventQueueSize="200000">
         <property name="maxNodes" value="5000" />
         <property name="timeToLive" value="1000" />
      </default>
      <region name="/org/jboss/data1">
         <property name="timeToLive" value="2000" />
      </region>
```

```
      <region name="/org/jboss/data2"
algorithmClass="org.jboss.cache.eviction.FIFOAlgorithm"
eventQueueSize="100000">
         <property name="maxNodes" value="3000" />
         <property name="minTimeToLive" value="4000" />
      </region>
   </eviction>

   <!--
      Cache loaders.

      If passivation is enabled, state is offloaded to the cache loaders
ONLY when evicted.  Similarly, when the state
      is accessed again, it is removed from the cache loader and loaded
into memory.

      Otherwise, state is always maintained in the cache loader as well as
in memory.

      Set 'shared' to true if all instances in the cluster use the same
cache loader instance, e.g., are talking to the
      same database.
   -->
   <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/org/jboss"/>
         <node fqn="/org/tempdata"/>
      </preload>

      <!--
         we can have multiple cache loaders, which get chained
      -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="true"
fetchPersistentState="true"
              ignoreModifications="true" purgeOnStartup="true">
         <properties>
            cache.jdbc.table.name=jbosscache
            cache.jdbc.table.create=true
            cache.jdbc.table.drop=true
         </properties>
         <singletonStore enabled="true"
class="org.jboss.cache.loader.SingletonStoreCacheLoader">
            <properties>
               pushStateWhenCoordinator=true
               pushStateWhenCoordinatorTimeout=20000
            </properties>
         </singletonStore>
      </loader>
   </loaders>

   <!--
      Define custom interceptors.  All custom interceptors need to extend
org.jboss.cache.interceptors.base.CommandInterceptor
   -->
   <!--
   <customInterceptors>
```

```
      <interceptor position="first"

class="org.jboss.cache.config.parsing.custominterceptors.AaaCustomIntercep
tor">
         <property name="attrOne" value="value1" />
         <property name="attrTwo" value="value2" />
      </interceptor>
      <interceptor position="last"

class="org.jboss.cache.config.parsing.custominterceptors.BbbCustomIntercep
tor"/>
      <interceptor index="3"

class="org.jboss.cache.config.parsing.custominterceptors.AaaCustomIntercep
tor"/>
      <interceptor before="org.jboss.cache.interceptors.CallInterceptor"

class="org.jboss.cache.config.parsing.custominterceptors.BbbCustomIntercep
tor"/>
      <interceptor after="org.jboss.cache.interceptors.CallInterceptor"

class="org.jboss.cache.config.parsing.custominterceptors.AaaCustomIntercep
tor"/>
   </customInterceptors>
   -->
</jbosscache>
```

12.1.1. XML の検証

設定 XML ファイルは XSD スキーマを使用して検証されます。 このスキーマは **jbosscache-core.jar** に含まれていますが、**http://www.jboss.org/jbosscache/jbosscache-config-3.0.xsd** よりオンラインで入手可能です。 ほとんどの IDE および XML オーサリングツールはこのスキーマを使用でき、 設定ファイルを作成しながら検証することが可能です。

JBoss Cache は起動時にも設定ファイルを検証し、 無効なファイルが見つかると例外をスローします。 この動作を無効にするには、 起動時に **-Djbosscache.config.validate=false** を JVM に渡します。 また、 **-Djbosscache.config.schemaLocation=url** を渡してバリデータを別のスキーマへ示すこともできます。

12.2. 設定ファイルのクイックリファレンス

上記で使用された各 XML 要素属性の定義と、 プログラムを用いた設定での対応 Bean を一覧にしたものです。 属性の説明に**動的**とある場合は、 キャッシュの作成、起動後に変更可能であることを意味します。

表12.1 <jbosscache /> 要素

<jbosscache /> 要素	
説明	JBoss Cache 設定ファイルのルート要素になります。 有効な JBoss Cache 設定ファイルで唯一の必須要素となります。

<jbosscache /> 要素	
親	なし (ルート要素であるため)
子	表12.40「<clustering /> 要素」、表12.37「<customInterceptors /> 要素」、表12.19「<eviction /> 要素」、表12.15「<invocationBatching /> 要素」、表12.7「<jmxStatistics /> 要素」、表12.13「<listeners /> 要素」、表12.27「<loaders /> 要素」、表12.3「<locking /> 要素」、表12.17「<serialization /> 要素」、表12.11「<shutdown /> 要素」、表12.9「<startup /> 要素」、表12.5「<transaction /> 要素」
相当 Bean	**Configuration**

表12.2 <jbosscache /> 属性

<jbosscache /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
xmlns	-	urn:jboss:jbosscache-core:config:3.1	urn:jboss:jbosscache-core:config:3.1	すべての設定エントリに対して XML 名前空間を定義します。
xmlns:xsi	-	http://www.w3.org/2001/XMLSchema-instance	http://www.w3.org/2001/XMLSchema-instance	設定の XML スキーマインスタンスを定義します。

表12.3 <locking /> 要素

<locking /> 要素	
説明	この要素はキャッシュ上のロッキング動作を指定します。
親	表12.1「<jbosscache /> 要素」
子	
相当 Bean	**Configuration**

表12.4 <locking /> 属性

<locking /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
isolationLevel	isolationLevel	READ_COMMITTED、REPEATABLE_READ	REPEATABLE_READ	トランザクションに使用される分離レベル。
lockParentForChildInsertRemove	lockParentForChildInsertRemove	true、 false	false	子を挿入したり削除する際に親ノードをロックするか指定します。 ノードごとに設定可能です (**Node.setLockForChildInsertRemove()** を参照)。
lockAcquisitionTimeout	lockAcquisitionTimeout (**動的**)	正の long 値	10000	スレッドがロックがロックの取得を試みる期間 (ミリ秒単位) です。 この時間内にロックを取得できないと、通 常 **TimeoutException** がスローされます。 **Option.setLockAcquisitionTimeout()** を使用して呼び出しごとにオーバーライド可能です。
nodeLockingScheme (**廃止**)	nodeLockingScheme	mvcc、pessimistic、optimistic	mvcc	使用するノードロッキングスキームを指定します。

`<locking />` 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
writeSkewCheck	writeSkewCheck	true、 false	false	書き込みスキューをチェックするか指定します。 **`nodeLockingScheme`** が **`mvcc`** で **`isolationLevel`** が **REPEATABLE_READ** の場合のみ使用されます。 詳細は 「同時書き込みと書き込みのスキュー」 を参照してください。
useLockStriping	useLockStriping	true、 false	true	ロックストライピングを使用するか指定します。 **`nodeLockingScheme`** が **`mvcc`** の場合のみ使用されます。 ロックストライピングは通常、 パフォーマンスやメモリの使用量を向上しますが、 同じ共有ロックへ複数の Fqn がマップされているとデッドロックが発生することがあります。 この場合、 並行レベルを上げるとデッドロックが軽減されますが、 問題を完全に解決するにはロックストライピングを無効にする必要があります。

<locking /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
concurrencyLevel	concurrencyLevel	正の整数。0 は使用できません。	500	取得した書き込みロックに使用する共有ロックの数を指定します。**nodeLockingScheme** が **mvcc** の場合のみ使用されます。 詳細は「MVCC 実装」 を参照してください。

表12.5 <transaction /> 要素

<transaction /> 要素	
説明	この要素はキャッシュ上のトランザクション動作を指定します。
親	表12.1「**<jbosscache />** 要素」
子	
相当 Bean	**Configuration**

表12.6 <transaction /> 属性

<transaction /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明

`<transaction />` 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
transactionManagerLookupClass	transactionManagerLookupClass	クラスパス上で使用できる有効なクラス	なし	トランザクションマネージャの取得に使用する **`TransactionManagerLookupClass`** 実装です。 指定がなく、**`RuntimeConfig.setTransactionManager()`** を使用して **`TransactionManager`** が挿入されていない場合、キャッシュはすべてのトランザクションに参加できません。
syncCommitPhase	syncCommitPhase (動的)	true、 false	false	有効になっていると、 クラスタ周辺でブロードキャストされるコミットメッセージは同期的にブロードキャストされます。 クラスタの一部のノードは既にコミットされロールバックできない可能性があるため、コミットのブロードキャストに対して障害を検出してもログメッセージに記録する以外の対処法がなく、 あまり意味がありません。

<transaction /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
syncRollbackPhase	syncRollbackPhase (動的)	true、 false	false	有効になっていると、 クラスタ周辺でブロードキャストされるロールバックメッセージは同期的にブロードキャストされます。 クラスタの一部のノードは既にコミットされロールバックできない可能性があるため、 ロールバックのブロードキャストに対して障害を検出してもログメッセージに記録する以外の対処法がなく、 あまり意味がありません。

表12.7 <jmxStatistics /> 要素

<jmxStatistics /> 要素	
説明	この要素は JMX よりキャッシュの統計を収集し、 報告するか指定します。
親	表12.1「<jbosscache /> 要素」
子	
相当 Bean	**Configuration**

表12.8 <jmxStatistics /> 属性

<jmxStatistics /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
enabled	exposeManagementStatistics	true、 false	true	JMX よりキャッシュの統計を収集し公開するかを制御します。

表12.9 <startup /> 要素

<startup /> 要素	
説明	この要素はキャッシュ起動時の挙動を指定します。
親	表12.1「<jbosscache /> 要素」
子	
相当 Bean	Configuration

表12.10 <startup /> 属性

<startup /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
regionsInactiveOnStartup	inactiveOnStartup	true、 false	false	「クラスローディングとリージョン」 が有効になっている場合、 この属性は作成された新規リージョンが起動時に非アクティブになっているかを制御します。

表12.11 <shutdown /> 要素

<shutdown /> 要素	
説明	この要素はキャッシュのシャットダウン時の挙動を指定します。
親	表12.1「<jbosscache /> 要素」
子	
相当 Bean	Configuration

表12.12 <shutdown /> 属性

<shutdown /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
hookBehavior	shutdownHookBehavior	DEFAULT、DONT_REGISTER、REGISTER	DEFAULT	JVM がシャットダウンシグナルを受信した時にリソースをクリーンアップできるよう、キャッシュが JVM シャットダウンフックを登録するかを決定します。デフォルトでは、MBean サーバー (JDK のデフォルト以外) が検出されない場合にシャットダウンフックが登録されます。REGSTER は MBean サーバーが検出されてもシャットダウンフックを登録するようキャッシュに強制します。DONT_REGISTER は MBean サーバーが検出されなくてもシャットダウンフックを登録しないようキャッシュに強制します。

表12.13 `<listeners />` 要素

`<listeners />` 要素	
説明	この要素は登録されたキャッシュリスナの挙動を指定します。
親	表12.1「`<jbosscache />` 要素」
子	
相当 Bean	**`Configuration`**

表12.14 `<listeners />` 属性

<listeners /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
asyncPoolSize	listenerAsyncPoolSize	整数	1	同期リスナとして登録されたキャッシュリスナへイベントを送信するために使用するスレッドプールのサイズです。 値が 1 未満であると、 すべての非同期リスナが同期リスナとして扱われ、 同期的に通知されます。
asyncQueueSize	listenerAsyncQueueSize	正の整数	50000	非同期リスナのスレッドプールによって使用されるバインドされたキューのサイズです。 **asyncPoolSize** が 1 以上である場合に考慮されます。 このキューへイベントを追加しようとするスレッドブロッキングが多い場合はこの値を大きくします。

表12.15 <invocationBatching /> 要素

<invocationBatching /> 要素	
説明	この要素は呼び出しバッチングの挙動を指定します。
親	表12.1「<jbosscache /> 要素」
子	
相当 Bean	`Configuration`

表12.16 <invocationBatching /> 属性

<invocationBatching /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
enabled	invocationBatchingEnabled	true、 false	false	呼び出しバッチングが有効または無効であるか。 詳細は 4章*API のバッチ化* の章を参照してください。

表12.17 <serialization /> 要素

<serialization /> 要素	
説明	この要素は JBoss Cache におけるオブジェクトシリアライゼーションの挙動を指定します。
親	表12.1「<jbosscache /> 要素」
子	
相当 Bean	`Configuration`

表12.18 <serialization /> 属性

<serialization /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
marshallerClass	marshallerClass	クラスパス上で使用できる有効なクラス	VersionAwareMarshaller	レプリケーションや永続化のため、オブジェクトのシリアライズおよびデシリアライズに使用するマーシャラーを指定します。

`<serialization />` 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
useLazyDeserialization	useLazyDeserialization	true、 false	false	必要となり使用されるまでオブジェクトのシリアライズやデシリアライズを延期するメカニズムです。 これにより、 通常デシリアライズを必要とする呼び出しのスレッドコンテキストクラスローダーを使用してデシリアライズが行われます。 クラスローダーの分離を提供する効率的なメカニズムになります。
useRegionBasedMarshalling (廃止)	useRegionBasedMarshalling	true、 false	false	特定リージョンでクラスローダーを登録し、 クラスローダーを分離した古いメカニズムです。
version	replicationVersion	有効な JBoss Cache のバージョン文字列	現バージョン	クラスタで通信を開始する時にデフォルトで使用されるバージョンストリームパーサーを決定するため **`VersionAwareMarshaller`** によって使用されます。 旧バージョンが含まれるクラスタで新しいバージョンの JBoss Cache を実行する必要がある場合に便利です。 ローリングアップグレードを実行するために使用できます。
objectInputStreamPoolSize	objectInputStreamPoolSize	正の整数	50	現在未使用。

<serialization /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
objectOutputStreamPoolSize	objectOutputStreamPoolSize	正の整数	50	現在未使用。

表12.19 `<eviction />` 要素

<eviction /> 要素	
説明	この要素はキャッシュにおけるエビクションの挙動を制御します。
親	表12.1「**<jbosscache />** 要素」
子	表12.21「**<default />** 要素」、 表12.23「**<region />** 要素」
相当 Bean	**EvictionConfig**

表12.20 `<eviction />` 属性

<eviction /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
wakeUpInterval	wakeupInterval	整数	5000	エビクションスレッドが実行されるミリ秒単位の頻度になります。 1 未満の値が設定されると、 エビクションスレッドは実行されず、 エビクションスレッドの実行が無効になります。

表12.21 `<default />` 要素

<default /> 要素	
説明	この要素はデフォルトのエビクションリージョンを定義します。
親	表12.19「**<eviction />** 要素」

<default /> 要素	
子	表12.25「<property /> 要素」
相当 Bean	**EvictionRegionConfig**

表12.22 <default /> 属性

<default /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
algorithmClass	evictionAlgorithmConfig	クラスパス上で使用できる有効なクラス	なし	タグが使用される場合はこの属性を指定する必要があります。 プログラムを用いて設定する場合、 この属性ではなくエビクションアルゴリズムの対応する **EvictionAlgorithmConfig** ファイルを使用する必要があります。 例えば、 XML で**LRUAlgorithm** を使用する場合、 プログラムを用いて **LRUAlgorithmConfig** のインスタンスを使用します。
actionPolicyClass	evictionActionPolicyClassName	クラスパス上で使用できる有効なクラス	DefaultEvictionActionPolicy	ノードのエビクションが必要な場合の対処方法を定義する、 エビクションアクションポリシークラスです。
eventQueueSize	eventQueueSize (動的	整数	200000	バインドされたエビクションイベントキューのサイズ。

表12.23 <region /> 要素

<region /> 要素	
説明	この要素はエビクションリージョンを定義します。 このタグのインスタンスは、 固有の **name** 属性を持つ場合、 複数存在することができます。
親	表12.19「**<eviction />** 要素」
子	表12.25「**<property />** 要素」
相当 Bean	**EvictionRegionConfig**

表12.24 <region /> 属性

<region /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
name	regionFqn	Fqn.fromString() を使用して解析できる文字列	なし	このリージョンを定義する固有の名前でなければなりません。 エビクションリージョンについての詳細は「エビクションリージョン」 を参照してください。
algorithmClass	evictionAlgorithmConfig	クラスパス上で使用できる有効なクラス	なし	タグが使用される場合はこの属性を指定する必要があります。 プログラムを用いて設定する場合、 この属性ではなくエビクションアルゴリズムの対応する **EvictionAlgorithmConfig** ファイルを使用する必要があります。 例えば、 XML で **LRUAlgorithm** を使用する場合、 プログラムを用いて **LRUAlgorithmConfig** のインスタンスを使用します。

<region /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
actionPolicyClass	evictionActionPolicyClassName	クラスパス上で使用できる有効なクラス	DefaultEvictionActionPolicy	ノードのエビクションが必要な場合の対処方法を定義する、 エビクションアクションポリシークラスです。
eventQueueSize	eventQueueSize (動的	整数	200000	バインドされたエビクションイベントキューのサイズ。

表12.25 `<property />` 要素

`<property />` 要素	
説明	名前と値のプロパティをエンクロージング設定要素に渡すメカニズムです。
親	表12.21「`<default />` 要素」、 表12.23「`<region />` 要素」、 表12.38「`<interceptor />` 要素」
子	
相当 Bean	直接セッタまたは `setProperties()` エンクロージング Bean

表12.26 `<property />` 属性

`<property />` 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
name	直接セッタまたは `setProperties()` エンクロージング Bean	文字列	なし	プロパティ名
value	直接セッタまたは `setProperties()` エンクロージング Bean	文字列	なし	プロパティ値

表12.27 <loaders /> 要素

<loaders /> 要素	
説明	キャッシュローダーを定義します。
親	表12.1「<jbosscache /> 要素」
子	表12.29「<preload /> 要素」、 表12.32「<loader /> 要素」
相当 Bean	CacheLoaderConfig

表12.28 <loaders /> 属性

<loaders /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
passivation	passivation	true、 false	false	true の場合、キャッシュローダーはパッシベーションモードで使用されます。 詳細は 9章 *キャッシュローダー* の説明を参照してください。
shared	shared	true、 false	false	true の場合、キャッシュローダーは共有モードで使用されます。詳細は 9章 *キャッシュローダー* の説明を参照してください。

表12.29 <preload /> 要素

<preload /> 要素	
説明	キャッシュ起動時に Fqn サブツリーの事前ロードを定義します。 この要素には属性がありません。
親	表12.27「<loaders /> 要素」
子	表12.30「<node /> 要素」

<preload /> 要素	
相当 Bean	CacheLoaderConfig

表12.30 <node /> 要素

<node /> 要素	
説明	この要素は、 キャッシュ起動時にすべての内容がキャッシュローダーより事前ロードされるサブツリーを定義します。 複数のサブツリーを事前ロードできますが、 サブツリーが重複しない場合は 2 つ以上のサブツリーを定義しないと意味がありません。
親	表12.29「**<preload />** 要素」
子	
相当 Bean	CacheLoaderConfig

表12.31 <node /> 属性

<node /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
fqn	preload	文字列	なし	事前ロードする Fqn。 Fqn.fromString() で解析可能な文字列でなければなりません。 プログラムを用いて実行する場合、 事前ロードしたい Fqn すべてを空白文字で区切った単一の文字列を作成し、 **CacheLoaderConfig.setPreload()** へ渡す必要があります。

表12.32 <loader /> 要素

<loader /> 要素	

`<loader />` 要素	
説明	この要素はキャッシュローダーを定義します。 複数の要素を使用してキャッシュローダーチェーンを作成することができます。
親	表12.27「`<loaders />` 要素」
子	表12.34「`<properties />` 要素」、 表12.35「`<singletonStore />` 要素」
相当 Bean	IndividualCacheLoaderConfig

表12.33 `<loader />` 属性

`<loader />` 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
class	className	クラスパス上で使用できる有効なクラス	なし	使用するキャッシュローダー実装。
async	async	true、 false	false	このキャッシュローダーへの変更はすべて個別のスレッドで非同期に発生します。
fetchPersistentState	fetchPersistentState	true、 false	false	キャッシュが起動すると、 クラスタの他のキャッシュにあるキャッシュローダーより永続ステートを読み出します。 1 つのローダー要素のみを true に設定できます。 また、 表12.40「`<clustering />` 要素」タグが存在する場合のみ有用です。
purgeOnStartup	purgeOnStartup	true、 false	false	起動時にこのキャッシュローダーをパージ (消去) します。

表12.34 <properties /> 要素

<properties /> 要素	
説明	この要素には **java.util.Properties** インスタンスによって読み取り可能なプロパティが複数含まれています。 このタグには属性はなく、 タグの内容は **Properties.load()** によって解析されます。
親	表12.32「**<loader />** 要素」、 表12.35「**<singletonStore />** 要素」、 表12.52「**<locator />** 要素」
子	
相当 Bean	IndividualCacheLoaderConfig.setProperties()

表12.35 <singletonStore /> 要素

<singletonStore /> 要素	
説明	この要素はエンクロージングキャッシュローダーを「シングルトンストアの設定」 として設定します。
親	表12.32「**<loader />** 要素」
子	表12.34「**<properties />** 要素」
相当 Bean	SingletonStoreConfig

表12.36 <singletonStore /> 属性

<singletonStore /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
class	className	クラスパス上で使用できる有効なクラス	SingletonStoreCacheLoader	使用するシングルトンストアのラッパー実装です。
enabled	enabled	true、 false	false	true の場合、 シングルトンストアキャッシュローダーが有効になります。

表12.37 <customInterceptors /> 要素

<customInterceptors /> 要素	
説明	この要素は、 キャッシュのカスタムインターセプタを定義できるようにします。 このタグには属性がありません。
親	表12.1「<jbosscache /> 要素」
子	表12.38「<interceptor /> 要素」
相当 Bean	なし。 ランタイム時にインターセプタをインスタンス化し、 **Cache.addInterceptor()** を使用してキャッシュに渡します。

表12.38 <interceptor /> 要素

<interceptor /> 要素	
説明	この要素はカスタムインターセプタを設定できるようにします。 このタグは複数回表示されることがあります。
親	表12.37「<customInterceptors /> 要素」
子	表12.25「<property /> 要素」
相当 Bean	なし。 ランタイム時にインターセプタをインスタンス化し、 **Cache.addInterceptor()** を使用してキャッシュに渡します。

表12.39 <interceptor /> 属性

<interceptor /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
class	-	クラスパス上で使用できる有効なクラス	なし	**CommandInterceptor** の実装。

`<interceptor />` 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
position	-	first、 last		このインターセプタをチェーンに置く場所となります。 first はキャッシュ上で呼び出しが実行された時に最初に遭遇するインターセプタになります。 last は呼び出しがデータ構造へ渡される前の最後のインターセプタになります。 **この属性は `before`、 `after`、 `index` と相互排他的になります。**
before	-	インターセプタの完全修飾クラス名		名前付きインターセプタのインスタンスの前に新しいインターセプタを直接配置します。 **この属性は `position`、 `after`、 `index` と相互排他的になります。**
after	-	インターセプタの完全修飾クラス名		名前付きインターセプタのインスタンスの後に新しいインターセプタを直接配置します。 **この属性は `position`、 `before`、 `index` と相互排他的になります。**

<interceptor /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
index	-	正の整数		チェーンにこのインターセプタを配置する場所になります。 **この属性は position、before、after と相互排他的になります。**

表12.40 <clustering /> 要素

<clustering /> 要素	
説明	この要素が存在すると、 キャッシュはクラスタモードで起動されます。 属性と子要素はクラスタリングの特性を定義します。
親	表12.1「<jbosscache /> 要素」
子	表12.46「<stateRetrieval /> 要素」、 表12.42「<sync /> 要素」、 表12.44「<async /> 要素」、 表12.48「<buddy /> 要素」、 表12.54「<jgroupsConfig /> 要素」
相当 Bean	設定

表12.41 <clustering /> 属性

<clustering /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明

<clustering /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
mode	cacheMode	replication、invalidation、 r、 i	replication	レプリケーションと無効化の違いについては、 8 章キャッシュモードとクラスタリングを参照してください。 Bean を使用する際、 同期と非同期の通信がクラスタリングモードを組み合わされ、 列挙である **Configuration.CacheMode** が提供されます。
clusterName	clusterName	文字列	JBossCache-cluster	参加するクラスタを識別するために使用されるクラスタ名です。

表12.42 <sync /> 要素

<sync /> 要素	
説明	この要素が存在するとすべての通信が同期で行われ、 スレッドがワイヤ上で送信されたメッセージを送信する度に、 受信側から確認応答を受け取るまでブロックします。 この要素は 表12.44「<async /> 要素」 要素と相互排他的になります。
親	表12.40「<clustering /> 要素」
子	
相当 Bean	Configuration.setCacheMode()

表12.43 <sync /> 属性

<sync /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**

<sync /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
replTimeout	syncReplTimeout (動的)	正の整数	15000	リモート呼び出しを実行する際に、確認応答を待機するタイムアウト値となります。 タイムアウト後に例外がスローされます。

表12.44 <async /> 要素

<async /> 要素	
説明	この要素が存在するとすべての通信が非同期で行われ、 スレッドがワイヤ上で送信されたメッセージを送信する時、 返る前に確認応答を待ちません。 この要素は 表12.42「**<sync />** 要素」 要素と相互排他的になります。
親	表12.40「**<clustering />** 要素」
子	
相当 Bean	Configuration.setCacheMode()

表12.45 <async /> 属性

<async /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明

<async />属性				
属性	Bean フィールド	許可される値	デフォルト	説明
serializationExecutorPoolSize	serializationExecutorPoolSize	正の整数	25	レプリケーションが非同期に行われるだけでなく、 レプリケーション内容のシリアライズも個別のスレッドで実行され、 呼出側ができるだけ早く返せるようにします。 この設定はシリアライザスレッドプールの大きさを制御します。 1 未満の値を設定すると、 シリアライズが非同期に行われなくなります。
serializationExecutorQueueSize	serializationExecutorQueueSize	正の整数	50000	シリアライゼーションエグゼキュータのタスクを保持するバインドされたキューの大きさを定義するため使用されます。 **`serializationExecutorPoolSize`** が 1 未満の場合など、 シリアライゼーションエグゼキュータが使用されない場合は無視されます。
useReplQueue	useReplQueue	true、 false	false	true の場合、 非同期通信をすべてキューに置き、 バッチとして定期的に送信するよう強制します。

<async /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
replQueueInterval	replQueueInterval	正の整数	5000	**useReplQueue** を true に設定すると、 レプリケーションキューの実行をフラッシュするために非同期スレッドを使用する頻度をこの属性が制御します。 スレッドのウェイクアップ時間をミリ秒単位で表す正の整数でなければなりません。
replQueueMaxElements	replQueueMaxElements	正の整数	1000	**useReplQueue** を true に設定した場合、 しきい値に達した時にキューのフラッシュをトリガするようこの属性を使用することができます。

表12.46 `<stateRetrieval />` 要素

<stateRetrieval /> 要素	
説明	このキャッシュインスタンスの起動時に、 隣接キャッシュよりステートが読み出されないよう制御するタグです。
親	表12.40「**<clustering />** 要素」
子	
相当 Bean	設定

表12.47 `<stateRetrieval />` 属性

<stateRetrieval /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
fetchInMemoryState	fetchInMemoryState	true、 false	true	true の場合、キャッシュが「ウォームスタート」するよう、キャッシュ起動時に隣接キャッシュにステートを要求します。
timeout	stateRetrievalTimeout	正の整数	10000	隣接キャッシュからのステートを待つ最大時間 (ミリ秒単位) です。 この時間を過ぎると例外がスローされ、 起動が停止されます。
nonBlocking	useNonBlockingStateTransfer	true、 false	false	この設定スイッチは、 3.1.0 で新たに導入された非ブロッキングステート転送メカニズムを有効にします。 ノードロッキングスキームとして MVCC が必要で、 使用される JGroups スタックに STREAMING_STATE_TRANSFER が存在しなければなりません。

表12.48 <buddy /> 要素

<buddy /> 要素	
説明	このタグが存在するとクラスタ全体でステートがレプリケートされず、 キャッシュインスタンスを選択してバックアップを維持するため、 バディレプリケーションが使用されます。 詳細は 「バディレプリケーション」 を参照してください。 クラスタリングモードが **`replication`** である場合のみ使用され、 **`invalidation`** である場合は使用されません。

<buddy /> 要素	
親	表12.40「<clustering /> 要素」
子	表12.50「<dataGravitation /> 要素」、 表12.52「<locator /> 要素」、
相当 Bean	BuddyReplicationConfig

表12.49 <buddy /> 属性

<buddy /> 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
enabled	enabled	true、 false	false	true の場合、 バディレプリケーションが有効になります。
communicationTimeout	buddyCommunicationTimeout	正の整数	10000	バディキャッシュよりバディグループ編成通信を待つ最大時間 (ミリ秒単位) になります。
poolName	buddyPoolName	文字列		キャッシュインスタンスを識別し、 バディ選択アルゴリズムにヒントを提供する方法として使用されます。 詳細は 「バディレプリケーション」 を参照してください。

表12.50 <dataGravitation /> 要素

<dataGravitation /> 要素	
説明	データグラビテーションが実行される方法を設定するタグです。 詳細は 「バディレプリケーション」 を参照してください。
親	表12.48「<buddy /> 要素」
子	

<dataGravitation /> 要素	
相当 Bean	BuddyReplicationConfig

表12.51 <dataGravitation /> 属性

<dataGravitation /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
auto	autoDataGravitation	true、 false	true	true の場合、 get() がキャッシュ上で実行され、 何も見つからない場合は隣接キャッシュからのグラビテーションが試行されます。 false の場合、 **`Option.setForceDataGravitation()`** オプションが提供される場合のみグラビテーションが実行されます。
removeOnFind	dataGravitationRemoveOnFind	true、 false	true	true の場合、 グラビテーションの発生時にグラビテーションを要求したインスタンスがステートを所有し、他の全インスタンスがグラビテーションが行われたステートをメモリから削除するよう要求します。
searchBackupTrees	dataGravitationSearchBackupTrees	true、 false	true	true の場合、 グラビテーション要求を受け取ると、キャッシュが一次データ構造だけでなく、 バックアップ構造も検索するようになります。

表12.52 <locator /> 要素

<locator /> 要素	
説明	バディの場所のアルゴリズムを提供するプラグ可能なメカニズムを提供するタグです。
親	表12.48「**<buddy />** 要素」
子	表12.34「**<properties />** 要素」
相当 Bean	BuddyLocatorConfig

表12.53 <locator /> 属性

<locator /> 属性				
属性	**Bean フィールド**	**許可される値**	**デフォルト**	**説明**
class	className	クラスパス上で使用できる有効なクラス	NextMemberBuddyLocator	クラスタからバディを選択する時に使用する **BuddyLocator** 実装です。 詳細は **BuddyLocator** の javadoc を参照してください。

表12.54 <jgroupsConfig /> 要素

<jgroupsConfig /> 要素	
説明	このタグは、 ネットワーク通信チャネルを作成するために JGroups と使用される設定を提供します。
親	表12.40「**<clustering />** 要素」
子	JGroups プロトコルを表す複数の要素です (JGroups ドキュメント を参照してください)。 要素属性が使用される場合は子要素はありません。 属性の項を参照してください。
相当 Bean	設定

表12.55 <jgroupsConfig /> 属性

`<jgroupsConfig />` 属性				
属性	Bean フィールド	許可される値	デフォルト	説明
configFile	clusterConfig	クラスパス上の JGroups 設定ファイル	udp.xml	この属性が使用されると、 このタグ内でプロトコルを表す JGroups 要素は無視されます。 JGroups 設定は指定ファイルより読み取られます。 **`multiplexerStack`** 属性と使用することはできません。
multiplexerStack	muxStackName	**`RuntimeConfig`** へ渡されたチャネルファクトリに存在する有効なマルチプレクサスタック名		**`RuntimeConfig.setMuxChannelFactory()`** を使用して JGroups の **`ChannelFactory`** インスタンスを渡す **`RuntimeConfig`** とのみ使用できます。 この属性が使用される場合、 このタグ内でプロトコルを表す JGroups 要素は無視され、 渡されたファクトリを使用して JGroups チャネルが作成されます。 **`configFile`** 属性と使用することはできません。

第13章 JMX の参照

13.1. JBOSS CACHE の統計

JMX には、 キャッシュの監視に関する豊富な情報が収集され公開されます。 一部情報の詳細は次の通りです。

表13.1 JBoss Cache JMX MBean

MBean	属性/操作名	説明
DataContainerImpl	getNumberOfAttributes()	データコンテナの全ノードにある属性の数を返します
	getNumberOfNodes()	データコンテナのノード数を返します
	printDetails()	データコンテナの詳細を出力します
RPCManagerImpl	localAddressString	ローカルアドレスの文字列表現
	membersString	クラスタビューの文字列表現
	statisticsEnabled	RPC 統計が収集されるかどうか
	replicationCount	成功したレプリケーションの数
	replicationFailures	失敗したレプリケーションの数
	successRatio	RPC 呼び出しの成功率
RegionManagerImpl	dumpRegions()	イベントキューの大きさを示すエビクションリージョンを含む登録済みリージョンすべての文字列表現をダンプします
	numRegions	登録済みリージョンの数
BuddyManager	BuddyManager	キャッシュのバディーグループの文字列表現
	buddyGroupsIParticipateIn	キャッシュが参加するすべてのバディーグループの文字列表現
TransactionTable	numberOfRegisteredTransactions	進行中の登録済みトランザクションの数

MBean	属性/操作名	説明
	transactionMap	内部 GlobalTransaction インスタンスへマップされた現在登録済みのトランザクションすべての文字列表現
MVCCLockManager	concurrencyLevel	設定された並行レベル
	numberOfLocksAvailable	共有ロックプールにの未使用のロック数
	numberOfLocksHeld	共有ロックプールの使用中のロック数
	testHashing(String fqn)	FQN 全体でロックの拡散をテストします。 文字列ベースの FQN の場合、 このメソッドはマップするロックアレイのインデックスを返します。
ActivationInterceptor	Activations	有効化されたパッシブノードの数
CacheLoaderInterceptor	CacheLoaderLoads	キャッシュローダーからロードされたノードの数
	CacheLoaderMisses	キャッシュローダーからノードをロードするのに失敗した数
CacheMgmtInterceptor	Hits	属性の取得に成功した数
	Misses	属性の取得に失敗した数
	Stores	属性ストア操作の数
	Evictions	ノード除外の数
	NumberOfAttributes	現在キャッシュされている属性の数
	NumberOfNodes	現在キャッシュされているノードの数
	ElapsedTime	キャッシュが実行されている時間 (秒単位)
	TimeSinceReset	キャッシュ統計がリセットされてから経過した時間 (秒数)

MBean	属性/操作名	説明
	AverageReadTime	キャッシュ属性を取得するのにかかったミリ秒単位の平均時間 (属性の取得の失敗も含む)
	AverageWriteTime	キャッシュ属性を書き込むのにかかる平均時間 (ミリ秒単位)
	HitMissRatio	ヒット数とヒット数/ミス数の比率。ヒットは取得属性操作であり、オブジェクトはクライアントに返されます。エントリがローカルキャッシュに存在しない場合、取得はキャッシュローダーから実行されます。
	ReadWriteRatio	読み取り操作と書き込み操作の比率。これはキャッシュストアに対するキャッシュのヒット数/ミス数の比率です。
CacheStoreInterceptor	CacheLoaderStores	キャッシュローダーに書き込まれるノードの数
InvalidationInterceptor	Invalidations	無効にされたキャッシュノードの数
PassivationInterceptor	Passivations	パッシブ化されたキャッシュ済みノードの数
TxInterceptor	Prepares	このインターセプタによって実行されるトランザクション準備操作の数
	Commits	このインターセプタによって実行されたトランザクションコミット操作の数
	Rollbacks	このインターセプタによって実行されたトランザクションロールバック操作の数
	numberOfSyncsRegistered	トランザクションマネージャに登録されている完了および削除待ちの同期の数。

13.2. JMX MBEAN 通知

以下の表は、 JBoss Cache で利用可能な JMX 通知とそれに対応するキャッシュイベントを表していま

す。 これらの通知は **`CacheJmxWrapper`** MBean より受信することができます。各通知は JBoss Cache によってパブリッシュされた単一のイベントを表し、 イベントのパラメータに対応するユーザーデータを提供します。

表13.2 JBoss Cache MBean 通知

通知タイプ	通知データ	CacheListener イベント
org.jboss.cache.CacheStarted	String: cache service name	@CacheStarted
org.jboss.cache.CacheStopped	String: cache service name	@CacheStopped
org.jboss.cache.NodeCreated	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeCreated
org.jboss.cache.NodeEvicted	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeEvicted
org.jboss.cache.NodeLoaded	String: fqn, boolean: isPre	@NodeLoaded
org.jboss.cache.NodeModifed	String: fqn, boolean: isPre, boolean: isOriginLocal	NodeModifed@NodeModifed
org.jboss.cache.NodeRemoved	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeRemoved
org.jboss.cache.NodeVisited	String: fqn, boolean: isPre	@NodeVisited
org.jboss.cache.ViewChanged	String: view	@ViewChanged
org.jboss.cache.NodeActivated	String: fqn	@NodeActivated
org.jboss.cache.NodeMoved	String: fromFqn, String: toFqn, boolean: isPre	@NodeMoved
org.jboss.cache.NodePassivated	String: fqn	@NodePassivated

付録A 改訂履歴

改訂 5.1.2-2.400	**2013-10-30**	**Rüdiger Landmann**

Rebuild with publican 4.0.0

改訂 5.1.2-2	**2012-07-18**	**Anthony Towns**

Rebuild for Publican 3.0

改訂 5.1.2-100	**Thu Dec 8 2011**	**Jared Morgan**

JBoss Enterprise Application Platform 5.1.2 GAに対する変更を追加。本ガイド文書の変更に関する情報は、『リリースノート 5.1.2』を参照してください。

改訂 5.1.1-100	**Mon Jul 18 2011**	**Jared Morgan**

JBoss Enterprise Application Platform 5.1.1 GAに対する変更を追加。本ガイド文書の変更に関する情報は、『リリースノート 5.1.1』を参照してください。

改訂 5.1.0-100	**Wed Sep 15 2010**	**Laura Bailey**

バージョン番号は新しいバージョン要件に応じて変更されました。
JBoss Enterprise Application Platform 5.1.0.GA 用に改訂。