Red Hat JBoss Data Grid 6

開発者ガイド

JBoss Data Grid 6 を使用した開発に関するガイド

エディッション 1

Last Updated: 2017-11-21

Red Hat JBoss Data Grid 6 開発者ガイド

JBoss Data Grid 6 を使用した開発に関するガイド
エディッション 1

Misha Husnain Ali
Red Hat Engineering Content Services
mhusnain@redhat.com

Gemma Sheldon
Red Hat Engineering Content Services
gsheldon@redhat.com

法律上の通知

概要

JBoss Data Grid 6 を使用している開発者向けの高度なガイド。

目次

前書き

第1章 JBOSS DATA GRID

1.1. JBOSS DATA GRID について

JBoss Data Grid は分散インメモリーデータグリッドで、次の機能を提供します。

- スキーマレスなキーバリューストア – Red Hat JBoss Data Grid は、固定のデータモデルを用いずに異なるオブジェクトを格納するフレキシブルな NoSQL データベースです。
- グリッドベースのデータストレージ – Red Hat JBoss Data Grid は、複数のノードにまたがったデータのレプリケートが簡単に行えるよう設計されています。
- エラスティックスケーリング – シンプルに混乱を起こさずノードの追加と削除を行えます。
- 複数のアクセスプロトコル – REST、Memcached、Hot Rod または簡単なマップのような API を使用して簡単にデータグリッドへアクセスできます。

バグを報告する

1.2. JBOSS DATA GRID の使用モード

JBoss Data Grid には 2 つの使用モードがあります。

リモートクライアントサーバーモード

リモートクライアントサーバーモードは、管理分散されたクラスター化可能なデータグリッドサーバーを提供します。アプリケーションは Hot Rod、Memcached、または REST クライアント API を使用して、データグリッドにリモートアクセスできます。

ライブラリーモード

ライブラリーモードは、カスタムランタイム環境の構築やデプロイに必要なすべてのバイナリを提供します。ライブラリー使用モードでは、分散クラスターの 1 つのノードへローカルアクセスでできます。この使用モードでは、使用されるコンテナの仮想マシン内でアプリケーションがデータグリッドの機能にアクセスできます。サポート対象のコンテナには、Tomcat 7 や JBoss Enterprise Application Platform 6 などがあります。

バグを報告する

1.3. JBOSS DATA GRID の利点

JBoss Data Grid の利点

巨大なヒープと高可用性

JBoss Data Grid では、パフォーマンス向上のためにアプリケーションがデータのルックアップ処理のほとんどを大型のサーバーデータベースへ委譲する必要がありません。JBoss Data Grid は、現在のエンタープライズアプリケーションの大部分に存在するボトルネックを完全に取り除きます。

> **例1.1 巨大なヒープと高可用性の例**
>
> 100 個のブレードサーバーを持つグリッドの例として、各ノードがレプリケートされたキャッシュ専用の 2 GB のストレージ領域を持っているとします。この場合、グリッドの全データが 2 GB のデータのコピーとなります。逆に、分散グリッド (データ項目ごとに 1 つのコピーが必要

> であることを仮定した場合) では、メモリーがサポートする仮想ヒープに 100 GB のデータが含まれます。グリッドのどこからでも効果的にこのデータへアクセスできるようになります。サーバーに障害が発生した場合、グリッドによって損失データの新しいコピーが即座に作成され、グリッドの運用サーバーに置かれます。

スケーラビリティー

JBoss Data Grid ではデータが均一に分散されるため、グリッドサイズの唯一の上限はネットワーク上でのグループ通信になります。ネットワークのグループ通信は最小限で、新規ノードの発見のみに制限されています。すべてのデータアクセスパターンは、ノードによるピアツーピア接続を介した直接通信を許可するため、スケーラビリティーの更なる向上を容易にします。JBoss Data Grid のクラスターは、リアルタイムでスケールアップまたはスケールダウンすることが可能ですが、インフラストラクチャーの再起動を必要としません。スケーリングポリシーへの変更をリアルタイムで適用できるため、非常にフレキシブルな環境を実現することができます。

データ分散

JBoss Data Grid は一貫したハッシュアルゴリズムを使用して、クラスターにおけるキーの場所を判断します。一貫したハッシュに関する利点には次のようなものがあります。

- コスト効果
- 速度
- 更なるメタデータやネットワークトラフィックの必要がない、決定論的なキーの場所

データ分散は、永続性とフォールトトラレンスを提供するため、余分のない十分なコピーがクラスター内に確実に存在するようにします。余分なコピーは環境のスケーラビリティーを低下させます。

永続性

JBoss Data Grid は、**`CacheStore`** インターフェースと複数の高パフォーマンス実装 (JDBC キャッシュストアやファイルシステムベースのキャッシュストアなど) を公開します。キャッシュストアはキャッシュをシードし、関連データを安全に保持し、破損しないようにします。また、キャッシュストアはプロセスがメモリー不足になると、データをディスクへオーバーフローします。

言語のバインディング

JBoss Data Grid は、既に多くの一般的なプログラミング言語向けに使用されている Memcached プロトコルと、Hot Rod と呼ばれる最適化された JBoss Data Grid 固有のプロトコルの両方をサポートします。そのため、Java Data Grid は Java に限定されず、すべての主要な Web サイトやアプリケーションに使用できます。

管理

数百個またはそれ以上のサーバーが存在するグリッド環境では、管理の実行は重要な機能になります。エンタープライズネットワーク管理ソフトウェアである JBoss Operations Network は、複数の JBoss Data Grid インスタンスを管理するのに最適なツールです。JBoss Operations Network の機能を使用すると、キャッシュマネージャーやキャッシュインスタンスを簡単かつ効率的に監視できます。

バグを報告する

1.4. JBOSS DATA GRID の前提条件

JBoss Data Grid の実行には、Java 6.0 およびそれ以降のバージョンと互換性を持つ Java 仮想マシン (JVM) のみが必要となります。JBoss Data Grid にはアプリケーションサーバーは必要ありません。

バグを報告する

1.5. JBOSS DATA GRID のバージョン情報

JBoss Data Grid は、データグリッドソフトウェアのオープンソースコミュニティー版である Infinispan が基盤となっています。Infinispan は、高ストレス環境で試行やテストを行い証明された JBoss Cache のコード、設計、およびアイデアを使用します。このようなデプロイメントの前歴があるため、JBoss Data Grid の初版リリースはバージョン 6.0 になっています。

バグを報告する

1.6. JBOSS DATA GRID のキャッシュアーキテクチャー

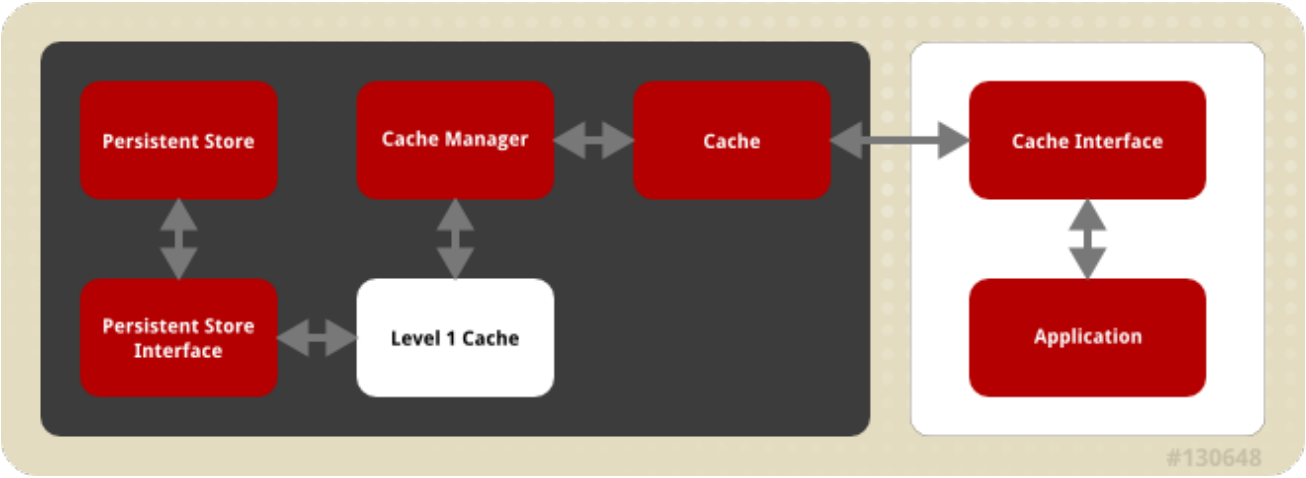


図1.1 JBoss Data Grid のキャッシュアーキテクチャー

JBoss Data Grid のキャッシュアーキテクチャーは個々の要素と要素同士の対話を表現します。分かりやすくするため、キャッシュアーキテクチャーの図は以下の 2 つの部分に分割されています。

- ユーザーが直接対話できない要素 (背景が灰色の部分)。これにはキャッシュ、キャッシュマネージャー、1 次キャッシュ、永続ストアインターフェース、永続ストアなどが含まれます。
- ユーザーが直接対話できる要素 (背景が白の部分)。これにはキャッシュインターフェースやアプリケーションなどが含まれます。

キャッシュアーキテクチャーの要素

JBoss Data Grid のキャッシュアーキテクチャーには次の要素が含まれます。

1. キャッシュインスタンスやエントリーを永久的に格納する永続ストア。
2. JBoss Data Grid は、永続ストアのアクセスに 2 つの永続ストアインターフェースを提供します。永続ストアインターフェースは以下のいずれかになります。
    - キャッシュローダーは永続データストアへの接続を提供する読み取り専用インターフェースです。キャッシュローダーは、キャッシュインスタンスおよび永続ストアよりデータの場所を見つけ、読み出すことができます。
    - キャッシュストアは、キャッシュローダーによるステートのロードおよび格納を許可するメソッドを公開し、キャッシュローダーの機能に書き込み機能が含まれるようにします。

3. 1 次キャッシュ (L1 キャッシュ) は、リモートキャッシュエントリーが最初にアクセスされた後にそれらのエントリーを格納し、同じエントリーがその後使用される度に不必要なリモートフェッチ操作が行われないようにします。

4. キャッシュマネージャーは JBoss Data Grid のキャッシュインスタンスを読み出すために使用される主なメカニズムで、キャッシュを使用する時の開始点とすることができます。

5. キャッシュはキャッシュマネージャーによって読み出されたキャッシュインスタンスを格納します。

6. キャッシュインターフェースは Memcached や Hot Rod などのプロトコルを使用します。キャッシュを持つインターフェースには REST を使用することもできます。リモートインターフェースに関する詳細は『開発者ガイド』を参照してください。

    - Memcached はメモリー内にキーバリューを格納するために使用される分散メモリーオブジェクトキャッシングシステムです。Memcached キャッシングシステムは、Memached プロトコルと呼ばれるテキストベースのクライアントサーバーキャッシングプロトコルを定義します。

    - Hot Rod は JBoss Data Grid で使用されるバイナリ TCP クライアントサーバープロトコルです。Memchached などの他のクライアントサーバープロトコルに不足している機能を補うために作成されました。Hot Rod は、パーティション化または分散化された JBoss Data Grid サーバークラスターの要求を、クライアントがスマートルーティングできるようにします。

    - REST プロトコルは、密結合のクライアントライブラリおよびバインディングを不要にします。REST API ではオーバーヘッドが発生し、REST 呼び出しの理解と作成に REST クライアントまたはカスタムコードを必要とします。

7. アプリケーションは、キャッシュインターフェースを介してユーザーがキャッシュとやりとりできるようにします。このようなエンドユーザーアプリケーションの一般的な例がブラウザーです。

バグを報告する

1.7. JBOSS DATA GRID の API

JBoss Data Grid は次の API を提供します。

- キャッシュ
- バッチ処理
- グループ化
- CacheStore
- 外部化

JBoss Data Grid のリモートクライアントサーバーモードでは、データグリッドとの対話に次の API のみを使用できます。

- 非同期 API (リモートクライアントサーバーモードで Hot Rod クライアントを併用する場合のみ使用可能)
- REST インターフェース

- Memcached インターフェース
- Hot Rod インターフェース
  - RemoteCache API

バグを報告する

パート I. プログラミング可能な API

第2章 キャッシュ API

2.1. キャッシュ API について

キャッシュインターフェースは、エントリーの追加、呼び出し、および削除に対する簡単なメソッドを提供します。これには JDK の **ConcurrentMap** インターフェースによって公開されるアトミックメカニズムが含まれます。エントリーが格納される方法は、使用されるキャッシュモードによって異なります。例えば、エントリーがリモートノードへレプリケートされたり、キャッシュストアでルックアップされたりすることもあります。

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

注記

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

バグを報告する

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

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

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

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

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

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

設定の説明:

提供された設定の各行の説明は以下のとおりです。

1.
```
Configuration c = new
ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC).b
uild();
```

   設定の最初の行で、新しいキャッシュ設定オブジェクト (**c** と命名済み) は、**ConfigurationBuilder** を使用して作成されます。設定 **c** には、キャッシュモードを除くすべてのキャッシュ設定オプションのデフォルト値が割り当てられ、この値は上書きされ、同期レプリケーション (**REPL_SYNC**) に設定されます。

2. ```
   String newCacheName = "repl";
   ```

   設定の 2 行目で、新しい変数 (タイプが **String**) が作成され、値 **repl** が割り当てられます。

3. ```
   manager.defineConfiguration(newCacheName, c);
   ```

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

4. ```
   Cache<String, String> cache = manager.getCache(newCacheName);
   ```

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

バグを報告する

2.3. 呼び出しごとのフラグ

2.3.1. 呼び出しごとのフラグについて

JBoss Data Grid 6 のデータグリッドでは、呼び出しごとのフラグを使用して各キャッシュコールの動作を指定できます。呼び出しごとのフラグを使用すると、時間を節約できる最適化の実装が促進されます。

バグを報告する

2.3.2. 呼び出しごとのフラグ機能

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

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

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

バグを報告する
   ```

2.3.3. 呼び出しごとのフラグの設定

JBoss Data Grid で呼び出しごとのフラグを使用するには、**withFlags()** メソッドコールを介して必要なフラグを高度なキャッシュに追加します。例は以下のとおりです。

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

注記

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

バグを報告する

2.3.4. 呼び出しごとのフラグの例

put() などの書き込み操作が以前の値を返さない必要がある JBoss Data Grid のケースでは、2 つのフラグが使用されます。この 2 つのフラグにより分散環境での (以前の値を修得するための) リモートルックアップが回避され、次に、不必要な以前の値の取得が回避されます。また、キャッシュにキャッシュローダーが設定されている場合は、この 2 つのフラグにより以前の値がキャッシュストアからロードされなくなります。

この 2 つのフラグの使用例は次のとおりです。

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

バグを報告する

2.4. ADVANCEDCACHE インターフェース

2.4.1. AdvancedCache インターフェースについて

JBoss Data Grid は **AdvancedCache** インターフェースを提供し、単純なキャッシュインターフェース以外に JBoss Data Grid を拡張します。 **AdvancedCache** インターフェースは以下のことを行えます。

- カスタムインターセプターを挿入します。
- 特定の内部コンポーネントにアクセスします。
- フラグを適用して特定のキャッシュメソッドの動作を変更します。

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

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

バグを報告する

2.4.2. AdvancedCache インターフェースでのフラグ使用

フラグは、JBoss Data Grid で特定のキャッシュメソッドに適用された場合に、ターゲットメソッドの動作を変更します。たとえば、以下のように **`AdvancedCache.withFlags()`** を使用してキャッシュ呼び出しに任意の数のフラグを適用します。

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

バグを報告する

2.4.3. カスタムインターセプターと AdvancedCache インターフェース

`AdvancedCache` インターフェースは、上級開発者がカスタムインターセプターを使用できるようにするメカニズムを提供します。カスタムインターセプターは、キャッシュ API メソッドの動作を変更でき、**`AdvacedCache`** インターフェースは、実行時にこのようなインターセプターをプログラムにより割り当てるために使用できます。

バグを報告する

2.4.4. カスタムインターセプター

2.4.4.1. カスタムインターセプターについて

カスタムインターセプターは、宣言したり、プログラミングを行ったりして JBoss Data Grid に追加できます。カスタムインターセプターは、キャッシュの変更に影響を与えたり、キャッシュの変更に応答したりすることにより JBoss Data Grid を拡張します。要素またはトランザクションの追加、削除、または更新がこのようなキャッシュ変更の例になります。

バグを報告する

2.4.4.2. カスタムインターセプター設計

JBoss Data Grid でカスタムインターセプターを設計するには、次のガイドラインに従います。

- カスタムインターセプターは **`CommandInterceptor`** を拡張しなければなりません。
- カスタムインターセプターはインスタンス化を可能にするために、パブリックの空のコンストラクターを宣言しなければなりません。
- カスタムインターセプターでは、**`property`** 要素を介して定義されたプロパティーに対して JavaBean スタイルセッターを定義する必要があります。

バグを報告する

2.4.4.3. カスタムインターセプターの追加

2.4.4.3.1. カスタムインターセプターを宣言して追加

JBoss Data Grid の各名前付きキャッシュは独自のインターセプタースタックを持ちます。結果として、カスタムインターセプターは名前付きキャッシュごとに追加できます。

カスタムインターセプターは、XML を使用して追加できます。たとえば、以下のようになります。

```
<namedCache name="cacheWithCustomInterceptors">
   <!--
   Define custom interceptors.  All custom interceptors need to extend
org.jboss.cache.interceptors.base.CommandInterceptor
   -->
   <customInterceptors>
      <interceptor position="FIRST"
class="com.mycompany.CustomInterceptor1">
         <properties>
            <property name="attributeOne" value="value1" />
            <property name="attributeTwo" value="value2" />
         </properties>
      </interceptor>
      <interceptor position="LAST"
class="com.mycompany.CustomInterceptor2"/>
      <interceptor index="3" class="com.mycompany.CustomInterceptor1"/>
      <interceptor before="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor2"/>
      <interceptor after="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor1"/>
   </customInterceptors>
</namedCache>
```

注記

この設定は、JBoss Data Grid のライブラリーモードに対してのみ有効です。

バグを報告する

2.4.4.3.2. カスタムインターセプターをプログラミングにより追加

JBoss Data Grid でカスタムインターセプターをプログラミングにより追加するには、最初に **AdvancedCache** への参照を取得します。

例:

```
CacheManager cm = getCacheManager();
Cache aCache = cm.getCache("aName");
AdvancedCache advCache = aCache.getAdvancedCache();
```

次に、***addInterceptor()*** メソッドを使用してインターセプターを追加します。

例:

```
advCache.addInterceptor(new MyInterceptor(), 0);
```

バグを報告する

第3章 バッチ化 API

3.1. バッチ化 API について

JBoss Data Grid クラスターがトランザクションの唯一の参加者である場合にバッチ化 API が使用されます。トランザクションに複数のシステムが参加している場合は、トランザクション API (JTA) のトランザクション (トランザクションマネージャーを使用) が使用されるはずです。

注記

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

バグを報告する

3.2. JAVA トランザクション API のトランザクションについて

JBoss Data Grid は、JTA 準拠のトランザクションの設定、使用および参加をサポートします。しかし、トランザクションサポートを無効にすることは JDBC 呼び出しの自動コミット機能を使用することと同等になります。JDBC 呼び出しでは、レプリケーションが有効になっている場合、各変更の後に変更内容が潜在的にレプリケートされます。

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

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

バグを報告する

3.3. バッチ化と JAVA トランザクション API (JTA)

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

1. 呼び出し中に取得されたロックは、トランザクションがコミットまたはロールバックするまで保持されます。
2. すべての変更は、クラスター内のすべてのノード上にあるバッチでトランザクションコミットプロセスの一部としてレプリケートされます。複数の変更が単一のトランザクション内で行われるため、レプリケーショントラフィックは小さいままになり、パフォーマンスが向上します。
3. 同期レプリケーションまたは無効化を使用する場合、レプリケーションまたは無効化の失敗により、トランザクションはロールバックされます。
4. JTA リソースと互換性がある **`CacheLoader`** (JTA**`DataSource`**) がトランザクションに使用された場合、JTA リソースはトランザクションに参加できます。
5. トランザクションに関連するすべての設定はバッチ化にも適用されます。

バッチ化に適用できるトランザクションに関連する設定の例は次のとおりです。

```
<transaction syncRollbackPhase="false"
      syncCommitPhase="false"
      useEagerLocking="true"
      eagerLockSingleNode="true" />
```

設定属性は、異なる値を使用してトランザクションおよびバッチ化の両方に使用できます。

注記

バッチ化機能と JTA トランザクションはライブラリーモードでのみサポートされます。

バグを報告する

3.4. バッチ化 API の使用

3.4.1. バッチ化 API の有効化

JBoss Data Grid のバッチ化 API は JBoss Enterprise Application Platform 構文を使用してキャッシュ設定で呼び出しバッチ化を有効にします。この例は以下のとおりです。

```
<distributed-cache name="default" batching="true">
...
</distributed-cache>
```

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

バグを報告する

3.4.2. バッチ化 API の設定

バッチ化 API を使用するには、キャッシュ設定で呼び出しバッチ化を有効にします。

XML 設定

バッチ化 API を XML ファイルで設定する場合:

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

プログラミングによる設定

バッチ化 API を プログラミングにより設定する場合:

```
Configuration c = new
ConfigurationBuilder().invocationBatching().enable().build();
```

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

バグを報告する

3.4.3. バッチ化 API の使用

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

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

例3.1 バッチを使用しない場合

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

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

例3.2 バッチの使用

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

行 **cache.endBatch(true);** が実行された場合、バッチが開始された以降に行われたすべての変更がレプリケートされます。

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

バグを報告する

3.4.4. バッチ化 API の使用例

バッチ化 API の単純な使用例は、2 つの銀行口座間で送金する場合です。

例3.3 バッチ化 API の使用例

JBoss Data Grid は、ある銀行口座から別の銀行口座に送金するトランザクションに使用されます。送金元と送金先の銀行口座両方が JBoss Data Grid 内に存在する場合は、このトランザクションにバッチ化 API が使用されます。ただし、ある口座が JBoss Data Grid 内にあり、別の口座がデータベース内にある場合は、トランザクションに分散トランザクションが必要です。

バグを報告する

3.5. トランザクション

3.5.1. 複数のキャッシュインスタンスにわたるトランザクション

各キャッシュは個別のスタンドアロン Java Transaction API (JTA) リソースとして動作します。ただし、コンポーネントは最適化のために JBoss Data Grid で内部的に共有できますが、この共有は、キャッシュが Java Transaction API (JTA) Manager とどのように対話するかに影響を与えません。

バグを報告する

3.5.2. トランザクション / バッチ処理および無効化メッセージ

バッチ処理またはトランザクションを使用せずにインバリデーションモードで変更を行う場合、各変更が行われた後に無効化メッセージが送信されます。トランザクションまたはバッチ処理が使用される場合、無効化メッセージはコミットが正常実行された後に送信されます。

トランザクションまたはバッチ処理と共に無効化を使用すると、メッセージが一括送信され、ネットワークトラフィックが軽減されるため、効率が良くなります。

バグを報告する

3.5.3. トランザクションマネージャー

3.5.3.1. JTA トランザクションマネージャーのルックアップクラスについて

キャッシュ操作を実行するため、キャッシュは環境のトランザクションマネージャーへの参照が必要となります。**`TransactionManagerLookup`** インターフェースの実装に属するクラス名を用いて、キャッシュを設定します。キャッシュが初期化されると、指定クラスのインスタンスが作成されます。そして **`getTransactionManager()`** メソッドを呼び出してトランザクションマネージャーへの参照を見つけ、返します。

JBoss Data Grid には次のトランザクションマネージャーのルックアップクラスが含まれています。

- **`DummyTransactionManagerLookup`** はテスト目的でトランザクションマネージャーを提供します。このテスト向けのトランザクションマネージャーは実稼働環境では使用されず、特に並列トランザクションやリカバリーなどの機能は厳しく制限されます。
- **`JBossStandaloneJTAManagerLookup`** は、JBoss Data Grid がスタンドアロン環境を実行する場合のデフォルトのトランザクションマネージャーです。これにより、JBoss Transactions ベースの完全に機能するトランザクションマネージャーで、**`DummyTransactionManagerLookup`** の機能制限が解消されます。
- **`GenericTransactionManagaerLookup`** は、ほとんどの Java EE アプリケーションサーバーでトランザクションマネージャーを見つけるために使用されるルックアップクラスです。デフォルトは **`DummyTransactionManagerLookup`** です。
- **`JBossTransactionManagerLookup`** は、JBoss Application Server インスタンス内でトランザクションマネージャーを見つけるルックアップクラスです。

注記

リモートクライアントサーバーモードでは、すべての JBoss Data Grid 操作が非トランザクションとなります。そのため、前述の JTA トランザクションマネージャーのルックアップクラスは、JBoss Data Grid のライブラリモードでのみ使用できます。

バグを報告する

3.5.3.2. DummyTransactionManagerLookup について

`DummyTransactionManagerLookup` はテスト目的でトランザクションマネージャーを提供します。このテスト向けのトランザクションマネージャーは実稼働環境では使用されず、特に並列トランザクションやリカバリーなどの機能は厳しく制限されます。

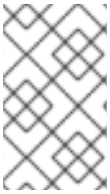
注記

JBoss Data Grid は、ライブラリーモードのJTA トランザクションマネージャールックアップクラスのみを使用します。リモートクライアントサーバーモードでは、すべての JBoss Data Grid 操作はトランザクション対応ではありません。

バグを報告する

3.5.3.3. JBossStandaloneJTAManagerLookup について

`JBossStandaloneJTAManagerLookup` は、JBoss Data Grid がスタンドアロン環境で実行される場合のデフォルトのトランザクションマネージャーです。これにより、JBoss Transactions ベースの完全に機能するトランザクションマネージャーで、**`DummyTransactionManagerLookup`** の機能制限が解消されます。



注記

JBoss Data Grid は、ライブラリーモードのJTA トランザクションマネージャールックアップクラスのみを使用します。リモートクライアントサーバーモードでは、すべての JBoss Data Grid 操作はトランザクション対応ではありません。

バグを報告する

3.5.3.4. GenericTransactionManagerLookup について

`GenericTransactionManagerLookup` は、ほとんどの Java EE アプリケーションサーバーでトランザクションマネージャーを見つけるために使用されるルックアップクラスです。デフォルトは **`DummyTransactionManagerLookup`** です。

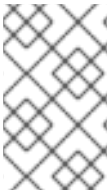
注記

JBoss Data Grid は、ライブラリーモードの JTA トランザクションマネージャールックアップクラスのみを使用します。リモートクライアントサーバーモードでは、すべての JBoss Data Grid 操作はトランザクション対応ではありません。

バグを報告する

3.5.3.5. JBossTransactionManagerLookup について

`JBossTransactionManagerLookup` は、JBoss Application Server インスタンス内でトランザクションマネージャーを見つけるルックアップクラスです。



注記

JBoss Data Grid は、ライブラリーモードのJTA トランザクションマネージャールックアップクラスのみを使用します。リモートクライアントサーバーモードでは、すべての JBoss Data Grid 操作はトランザクション対応ではありません。

バグを報告する

3.5.3.6. トランザクションマネージャーの使用

3.5.3.6.1. キャッシュからのトランザクションマネージャーの取得

初期化後に次の設定を使用して、キャッシュより TransactionManager を取得します。

手順3.1 キャッシュからのトランザクションマネージャーの取得

1. **`BasicCacheContainer`** の設定場所プロパティーに次のプロパティーを追加して、**`transactionManagerLookupClass`** を定義します。

```
Configuration config = new ConfigurationBuilder()
...
  .transaction().transactionManagerLookup(new
GenericTransactionManagerLookup())
```

2. **`TransactionManagerLookup.getTransactionManager`** を次のように呼び出します。

```
TransactionManager tm =
cache.getAdvancedCache().getTransactionManager();
```

バグを報告する

3.5.3.6.2. トランザクションの設定

JBoss Data Grid のトランザクションは、キャッシュレベルで設定されます。トランザクションの設定方法の例は次の通りです。

```
<cache>
..
 <transaction mode="NONE"
              stop-timeout="30000"
              locking="OPTIMISTIC" />
...
</cache>
```

- ***`mode`*** 属性はキャッシュトランザクションモードを設定します。この属性の有効な値は **`NONE`** (デフォルト)、 **`NON_XA`**、 **`NON_DURABLE_XA`**、 **`FULL_XA`** です。
- ***`stop-timeout`*** 属性は、キャッシュが停止した時に JBoss Data Grid が継続するリモートおよびローカルトランザクションを待機する期間 (ミリ秒数) を示します。
- ***`locking`*** 属性はキャッシュに対して使用されるロックモードを指定します。この属性に有効な値は **`OPTIMISTIC`** (デフォルト) と **`PESSIMISTIC`** です。

バグを報告する

3.5.3.6.3. トランザクションマネージャーと XAResources

トランザクションリカバリー処理はトランザクションマネージャーに固有するにも関わらず、**XAResource** 実装への参照を提供し、**XAResource.recover** を実行する必要があります。

バグを報告する

3.5.3.6.4. XAResource の参照の取得

JBoss Data Grid の **XAResource** への参照を取得するには、次の API を使用します。

```
XAResource xar = cache.getAdvancedCache().getXAResource();
```

バグを報告する

3.5.3.6.5. デフォルトの分散トランザクションの挙動

JBoss Data Grid では、**XAResource** より分散トランザクションで JBoss Data Grid を最初のクラス参加者として登録するのがデフォルトの挙動になります。JBoss Data Grid がトランザクションの参加者になる必要がない場合、同期化を介してトランザクションのライフサイクル状態 (準備、完了など) に関して通知することができます。

バグを報告する

3.5.4. トランザクションの同期化

3.5.4.1. トランザクション (JTA) 同期化について

JTA の同期化は、トランザクションのライフサイクルイベントに関する情報を JBoss Data Grid に提供します。また、同期化を登録すると **XAResource** として登録しなくても、JBoss Data Grid がトランザクションイベントへ応答できるようになります。その結果、トランザクションマネージャーがトランザクション操作を最適化し、2 相コミット (2PC) アルゴリズムではなく、1 相コミット (1PC) アルゴリズムが必要となります。

JBoss Data Grid では、JTA 同期化はライブラリーモードでのみ使用できます。

バグを報告する

3.5.4.2. 同期化を有効にする

JBoss Data Grid では、モードに非 XA オプションが設定されている場合にトランザクションに対して同期化が自動的に使用されます。

同期化を有効にするには、トランザクション要素のモードパラメーターを以下のいずれかに設定します。

- **NONE** (同期)
- **NO_XA** (同期)

注記

JBoss Data Grid では、ライブラリーモードにおける同期化を介したトランザクション参加のみが許可されます。

バグを報告する

3.5.5. ステートの調整

3.5.5.1. ステートの調整について

JBoss Data Grid でステートを調整するため、トランザクションマネージャーが、手作業による介入が必要なトランザクションに関する情報をシステム管理者に渡します。

システム管理者は、トランザクションリカバリー作業の要件として、関連するトランザクションの XID を認識している必要があります。トランザクションの XID はバイトアレイとして保存されます。

バグを報告する

3.5.5.2. トランザクションリカバリーについて

トランザクションマネージャーはリカバリー処理を調整し、操作の完了に手作業の介入が必要となるトランザクションを判断するため JBoss Data Grid と共に動作します。この処理はトランザクションリカバリーと呼ばれます。

バグを報告する

3.5.5.3. トランザクションリカバリーを有効にする

JBoss Data Grid では、トランザクションリカバリーがデフォルトで無効になっています。無効にすると、トランザクションマネージャーは手作業による介入が必要なトランザクションを判断できなくなります。

XML の使用

次のように XML 設定を使用し、トランザクションリカバリーを有効にします。

```
<transaction useEagerLocking="true" eagerLockSingleNode="true">
    <recovery enabled="true" recoveryInfoCacheName="noRecovery"/>
</transaction>
```

recoveryInfoCacheName 属性は任意です。

プログラミングによる設定

次のように Fluent Configuration API を介してトランザクションリカバリーを有効にすることもできます。

手順3.2 プログラミングによるトランザクションリカバリーの設定

1. JBoss Data Grid のトランザクションリカバリーを有効にするには **.recovery** を呼び出します。

```
configuration.fluent().transaction().recovery();
```

2. トランザクションリカバリーの状態を確認するには、次のプログラミング設定を使用します。

```
boolean isRecoveryEnabled =
configuration.isTransactionRecoveryEnabled();
```

3. JBoss Data Grid のトランザクションリカバリーを無効にするには、次のプログラミング設定を使用します。

```
configuration.fluent().transaction().recovery().disable();
```

トランザクションリカバリーはキャッシュごとに有効または無効にすることが可能です。例えば、トランザクションはトランザクションリカバリーが有効な状態で 1 つのキャッシュにわたり、トランザクションリカバリーが無効な状態で他のキャッシュにわたることが可能です。

バグを報告する

3.5.5.4. トランザクションリカバリーのプロセス

JBoss Data Grid におけるトランザクションリカバリープロセスの概要は次の通りです。

手順3.3 トランザクションリカバリーのプロセス

1. トランザクションマネージャーは介入が必要なトランザクションの一覧を作成します。
2. 電子メールまたはログを使用して、JMX を使用して JBoss Data Grid に接続するシステム管理者へトランザクション (トランザクション ID を含む) の一覧を提供します。各トランザクションの状態は **`COMMITTED`** か **`PREPARED`** になります。**`COMMITTED`** と **`PREPARED`** の両方の状態であるトランザクションがある場合、ノード上で準備状態である間にトランザクションが他のノードでコミットされたことを意味します。
3. システム管理者は、トランザクションマネージャーより受信した XID を JBoss Data Grid の内部 ID へ視覚的にマッピングします。XID (バイトアレイ) を JMX ツールに渡し、このマッピングがない状態で JBoss Data Grid によって再アセンブルすることはできないため、この手順が必要となります。
4. マッピングされた内部 ID を基に、システム管理者はトランザクションに対してコミットまたはロールバックプロセスを強制します。

バグを報告する

3.5.5.5. トランザクションリカバリーの例

以下の例は、お金がデータベースに格納された口座から JBoss Data Grid に格納された口座に転送される状況でどのようにトランザクションが使用されるかを示しています。

例3.4 データベースに格納された口座から JBoss Data Grid 内の口座への送金

1. 送信元 (データベース) と送信先 (JBoss Data Grid) リソース間の 2 フェーズコミットプロトコルを実行するために、**`TransactionManager.commit()`** メソッドが呼び出されます。
2. **`TransactionManager`** が、準備フェーズ (2 フェーズコミットの最初のフェーズ) を開始するようデータベースと JBoss Data Grid に指示します。

> コミットフェーズ中、データベースは変更を適用しますが、JBoss Data Grid はトランザクションマネージャーのコミット要求を受け取る前に失敗します。結果として、不完全なトランザクションのため、システムは不整合な状態になります。特に、送金される金額はデータベースから引かれますが、準備された変更を適用できないため、JBoss Data Grid に表示されません。
>
> ここでは、トランザクションリカバリーは、データベースと JBoss Data Grid エントリー間の不整合を調整するために使用されます。

バグを報告する

3.5.5.6. トランザクションメモリーおよび JMX サポート

JMX を使用してトランザクションリカバリーを管理するには、JMX サポートを明示的に有効にする必要があります。

バグを報告する

3.5.5.7. 強制コミットおよびロールバック操作

JBoss Data Grid は、トランザクションのコミットまたはロールバックを明示的に強制する操作に対して JMX を使用します。これらのメソッドは関連するトランザクションに関連付けられた数の代わりに XID を定義するバイトアレイを受け取ります。

システム管理者はこのような JMX 操作を使用して、手動介入が必要なトランザクションに対して自動的にジョブを完了できます。このプロセスでは、トランザクションマネージャーのトランザクションリカバリープロセスを使用し、トランザクションマネージャーの XID オブジェクトにアクセスできます。

バグを報告する

3.5.5.8. トランザクションおよび例外

キャッシュメソッドが JTA トランザクションのスコープ内で **`CacheException`** (または、**`CacheException`** のサブクラス) を返すと、トランザクションはロールバックするよう自動的にマークされます。

バグを報告する

3.5.6. デッドロックの検出

3.5.6.1. デッドロックの検出について

デッドロックは、複数のプロセスまたはタスクが他のプロセスまたはタスクが相互に必要なリソースを解放するまで待つときに発生します。デッドロックにより、特に複数のトランザクションが 1 つのキーセットに対して動作する場合にシステムのスループットが大幅に短縮されることがあります。

JBoss Data Grid は、このようなデッドロックを識別するデッドロック検出を提供します。デッドロック検出は、デフォルトで **`disabled`** に設定されます。

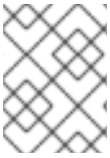
バグを報告する

3.5.6.2. デッドロック検出を有効にする

JBoss Data Grid のデッドロック検出はデフォルトでは **disabled** に設定されていますが、以下を追加して ***namedCache*** 設定要素を使用すると、デッドロック検出を有効にし、各キャッシュに対して設定を行うことが可能です。

```
<deadlockDetection enabled="true" spinDuration="1000"/>
```

デッドロック検出は個別のキャッシュへのみ適用できます。JBoss Data Grid は複数のキャッシュに適用されたデッドロックを検出できません。



注記

JBoss Data Grid では、ライブラリモードでのみデッドロック検出を設定できます。この設定はリモートクライアントサーバーモードでは使用できません。

バグを報告する

第4章 グループ化 API

4.1. グループ化 API について

グループ化 API を使用する場合、JBoss Data Grid では、エントリーを含めるノードを決定するために、キーのハッシュの代わりにグループのハッシュが作成および使用されます。

グループ化 API では、各ノードがアルゴリズムを使用して各キーの所有者を決定できることが重要です。このために、グループを手動で指定することはできず、グループは (キークラスにより生成された) エントリーに組み込まれているか、(外部の関数により生成された) エントリーに対して外部的である必要があります。

バグを報告する

4.2. API 操作のグループ化

JBoss Data Grid は、通常特定のキーのハッシュを使用してエントリーを格納する宛先ノードを決定します。グループ化 API を使用する場合、JBoss Data Grid はキーのハッシュの代わりにグループのハッシュを使用して宛先ノードを決定します。ただし、ノードにエントリーを実際に格納する場合は、キーのハッシュが引き続き使用されます。

各ノードはノード間でエントリーの場所に関するメタデータを渡すのではなく、アルゴリズムを使用して各キーの所有者を決定できることが重要です。この要件の結果として、グループは手動で指定できず、以下のいずれかである必要があります。

- エントリーに組み込まれています。つまり、キークラスにより生成されたことを意味します。
- エントリーに対して外部的です。つまり、外部機能により生成されたことを意味します。

バグを報告する

4.3. グループ化 API の設定

JBoss Data Grid でグループ化 API を使用するには、最初にグループを有効にします。

プログラミングによる設定

プログラミング可能な API を使用して JBoss Data Grid を設定するには、以下を呼び出します。

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

XML 設定

XML を使用して JBoss Data Grid を設定するには、以下を使用します。

```
<clustering>
  <hash>
     <groups enabled="true" />
  </hash>
</clustering>
```

キーのクラス定義が変更可能であり、グループの決定がキークラスに対して直交した関心事でない場合は、組み込みグループを使用します。メソッド内で **@Group** アノテーションを使用して組み込みグループを指定する場合は、以下のようになります。

```
class User {

   ...
   String office;
   ...

   int hashCode() {
      // Defines the hash for the key, normally used to determine location
      ...
   }

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

}
```

注記

グループは **String** である必要があります。

キークラスコントロールがない場合や、グループの決定がキークラスに対して直交した関心事である場合は、外部グループを使用します。外部グループは、**Grouper** インターフェースを実装して指定します。**Grouper** インターフェース内の **computeGroup** メソッドはグループを返します。

Grouper はインターセプターとして動作し、以前に計算された値を **computeGroup()** メソッドに渡します。定義された場合、**@Group** は、最初の **Grouper** に渡されるグループを決定し、組み込みグループを使用したときに改善されたグループコントロールを提供します。

grouper を使用してキーのグループを決定するには、ターゲットキーからその ***keyType*** を割り当てることができる必要があります。

以下は **Grouper** の例です。

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

   // A pattern that can extract from a "kX" (e.g. k1, k2) style key
   // The pattern requires a String key, of length 2, where the first
character is
   // "k" and the second character is a digit. We take that digit, and
perform
   // modular arithmetic on it to assign it to group "1" or group "2".
   private static Pattern kPattern = Pattern.compile("(^k)(<a>\\d</a>)$");

    public String computeGroup(String key, String group) {
        Matcher matcher = kPattern.matcher(key);
        if (matcher.matches()) {
```

```
            String g = Integer.parseInt(matcher.group(2)) % 2 + "";
            return g;
        } else
            return null;
    }

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

}
```

この例では、単純な **grouper** がクラスキーを使用して、パターンでキーからグループを抽出します。キークラスで指定された情報は無視されます。各 **grouper** は登録して使用する必要があります。

プログラミングによる設定

プログラミングにより JBoss Data Grid を設定する場合:

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

XML 設定

または、XML を使用して JBoss Data Grid を設定する場合:

```
<clustering>
  <hash>
     <groups enabled="true">
        <grouper class="com.acme.KXGrouper" />
     </groups>
  </hash>
</clustering>
```

バグを報告する

第5章 CACHESTORE API

5.1. CACHESTORE API について

CacheStore インターフェースの実装は、キャッシュのリードスルーとライトスルーの動作を定義します。キャッシュストアメソッドは、セカンダリーストレージで、エントリーの追加や削除などの操作を実行するために呼び出されます。

バグを報告する

5.2. CONFIGURATIONBUILDER API

5.2.1. ConfigurationBuilder API について

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

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

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

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

バグを報告する

5.2.2. ConfigurationBuilder API の使用

5.2.2.1. プログラムにより CacheManager とレプリケートされたキャッシュを作成

JBoss Data Grid のプログラムによる設定には、ほとんど ConfigurationBuilder API と CacheManager のみが関係します。

手順5.1 JBoss Data Grid でのプログラムによる設定に関する手順

1. XML ファイルで初めに CacheManager を作成します。必要な場合は、この CacheManager は使用ケースの要件を満たす仕様に基づいて実行時にプログラミングできます。以下は、CacheManager の作成方法の例です。

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

2. 同期的にレプリケートされた新しいキャッシュをプログラムにより作成します。

   a. ConfigurationBuilder ヘルパーオブジェクトを使用して新しい設定オブジェクトインスタンスを作成します。

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

b. キャッシュモードを同期レプリケーションに設定します。

```
String newCacheName = "repl";
```

c. マネージャーで設定を定義または登録します。

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

バグを報告する

5.2.2.2. デフォルト名前付きキャッシュを使用したカスタマイズ済みキャッシュの作成

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

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

上記の例の要件を満たすようデフォルトキャッシュをカスタマイズするには、以下の手順を使用します。

手順5.2 デフォルトキャッシュのカスタマイズ

1. デフォルトの Configuration オブジェクトのインスタンスを読み取り、デフォルトの設定を取得します。

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-
config-file.xml");
Configuration dcc = cacheManager.getDefaultCacheConfiguration();
```

2. ConfigurationBuilder を使用してキャッシュモードと新しい設定オブジェクトの L1 キャッシュライフスパンを構築および変更します。

```
Configuration c = new
ConfigurationBuilder().read(dcc).clustering().cacheMode(CacheMode.DI
ST_SYNC).l1().lifespan(60000L).build();
```

3. キャッシュマネージャーでキャッシュ設定を登録または定義します。

```
Cache<String, String> cache = manager.getCache(newCacheName);
```

バグを報告する

5.2.2.3. 非デフォルト名前付きキャッシュを使用したカスタマイズ済みキャッシュの作成

デフォルトでない名前付きキャッシュを使用して新しいカスタマイズ済みキャッシュを作成する必要がある状況が発生することがあります。これを達成する手順は、このためにデフォルトの名前付きキャッシュを使用するときに実行する手順に似ています。

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

手順5.3 非デフォルト名前付きキャッシュを使用したカスタマイズ済みキャッシュの作成

1. **replicatedCache** を読み取りデフォルト設定を取得します。

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration rc =
cacheManager.getCacheConfiguration("replicatedCache");
```

2. ConfigurationBuilder を使用して新しい設定オブジェクトで必要な設定を構築および変更します。

```
Configuration c = new
ConfigurationBuilder().read(rc).clustering().cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).build();
```

3. キャッシュマネージャーでキャッシュ設定を登録または定義します。

```
Cache<String, String> cache = manager.getCache(newCacheName);
```

バグを報告する

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

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

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

バグを報告する

5.2.2.5. グローバル設定の例

5.2.2.5.1. トランスポートレイヤーのグローバル設定

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

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
.globalJmxStatistics()
.build();
```

バグを報告する

5.2.2.5.2. キャッシュマネージャー名のグローバル設定

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

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

バグを報告する

5.2.2.5.3. スレッドプールエクスキューターのグローバルカスタマイズ

一部の JBoss Data Grid 機能は、スレッドプールエクスキューターのグループに基づきます。これらのエクスキューターは、以下のようにグローバルレベルでカスタマイズできます。

```
  GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .replicationQueueScheduledExecutor()
    .factory(DefaultScheduledExecutorFactory.class)
    .addProperty("threadNamePrefix", "RQThread")
  .build();
```

バグを報告する

5.2.2.6. キャッシュレベル設定の例

5.2.2.6.1. クラスターモードに対するキャッシュレベル設定

以下の設定により、ユーザーはキャッシュのクラスターモードなどのオプションをグローバルではなくキャッシュレベルで使用できます。

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

バグを報告する

5.2.2.6.2. キャッシュレベルエビクションおよび期限切れ設定

以下の設定により、ユーザーはキャッシュの期限切れまたはエビクションオプションを設定できます。

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

.maxEntries(20000).strategy(EvictionStrategy.LIRS).expiration()
             .wakeUpInterval(5000L)
```

```
            .maxIdle(120000L)
          .build();
```

バグを報告する

5.2.2.6.3. JTA トランザクションに対するキャッシュレベル設定

JTA トランザクション設定のキャッシュと対話するには、トランザクションレイヤーを設定し、オプションでロック設定を行う必要があります。トランザクション対応キャッシュの場合は、トランザクションリカバリー未完了トランザクションを処理できるようにすることが推奨されます。また、JMX 管理と統計収集も有効にすることが推奨されます。

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

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

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

バグを報告する

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

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

```
  Configuration config = new ConfigurationBuilder()
      .loaders()
        .shared(false).passivation(false).preload(false)

.addFileCacheStore().location("/tmp").streamBufferSize(1800).async().enabl
e().threadPoolSize(20).build();
```

バグを報告する

5.2.2.6.5. 高度なエクスターナライザーに対するキャッシュレベル設定

高度なエクスターナライザーに対するキャッシュレベル設定などの高度なオプションは、以下のようにプログラミングにより設定することもできます。

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

バグを報告する

5.2.2.6.6. カスタムインターセプターに対するキャッシュレベル設定

カスタムインターセプターに対するキャッシュレベル設定などの高度なオプションは、以下のようにプログラミングにより設定することもできます。

```
Configuration config = new ConfigurationBuilder()
.customInterceptors().interceptors()
  .add(new FirstInterceptor()).first()
  .add(new LastInterceptor()).last()
  .add(new FixPositionInterceptor()).atIndex(8)
  .add(new AfterInterceptor()).after(LockingInterceptor.class)
  .add(new BeforeInterceptor()).before(CallInterceptor.class)
.build();
```

バグを報告する

第6章 EXTERNALIZABLE API

6.1. エクスターナライザーについて

`Externalizer` クラスは、以下のことを行えるクラスです。

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

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

バグを報告する

6.2. EXTERNALIZABLE API について

Externalizable インターフェースはシリアライゼーションを使用および拡張します。このインターフェースは、JBoss Data Grid でシリアライゼーションとシリアライゼーション解除を制御するために使用されます。

バグを報告する

6.3. EXTERNALIZABLE API の使用

6.3.1. Externalizable API の使用

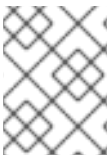
JBoss Data Grid は JBoss Marshalling を使用してオブジェクトをシリアライズします。

シリアライズされたクラスには、以下のことを行うよう設定された対応するエクスターナライザーインターフェース実装が必要です。

- オブジェクトクラスをシリアライズされたクラスに変換します。
- 出力からオブジェクトクラスを読み取ります。

個別のクラスを使用してエクスターナライザーインターフェースを実装します。エクスターナライザー実装は、ストリームからオブジェクトを読み取るときにクラスのオブジェクトがどのように作成されるかを制御します。

`readObject()` 実装はターゲットクラスのオブジェクトインスタンスを作成します。これにより、インスタンスを柔軟に作成し、ターゲットクラスを永続的に変更しないことが可能になります。



注記

エクスターナライザー実装は内部スタティックパブリッククラスとして外部化したクラス内に格納することが推奨されます。

バグを報告する

6.3.2. Externalizable API 設定例

JBoss Data Grid の Externalizable API を設定するには、次の手順を実行します。

- マーシャリングまたはマーシャリング解除するオブジェクトのタイプの ***externalizer*** 実装を提供します。
- {@link SerializeWith} を使用してマーシャリングされたタイプクラスをアノテートし、***externalizer*** クラスを指定します。

例:

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

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

   final String name;
   final int age;

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

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

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

このようにエクスターナライザーを設定するのには複数の欠点があります。

- モデル内の制約により、この方法で生成されたペイロードサイズが非効率になることがあります。
- ソースコードが利用不可、またはソースコードを変更できないクラスにエクスターナライザーが必要になることがあります。
- アノテーションを使用すると、マーシャリングレイヤーなどの下位レベル詳細を抽象化しようとするフレームワーク開発者またはサービスプロバイダーが制約を受けることがあります。

バグを報告する

6.3.3. エクスターナライザーとマーシャラークラスのリンク

エクスターナライザーの **readObject()** および **writeObject()** メソッドは、**getTypeClasses()** 実装を提供することにより外部化するよう設定されたタイプクラスとリンクします。

例:

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

この例では、ReplicableCommandExternalizer は複数のコマンドを外部化できることを示しています。

ソースコードが利用できない、またはソースコードを変更できないため、外部化するクラスインスタンスを参照できない場合があります。この場合は、ユーザーが完全修飾クラス名を使用してクラスをルックアップできます。例:

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

バグを報告する

6.4. ADVANCEDEXTERNALIZER

6.4.1. AdvancedExternalizer について

JBoss Data Grid の AdvancedExternalizer は、ユーザー定義されたクラスのマーシャリングまたはマーシャリング解除のためにエクスターナライザーを提供します。

AdvancedExternalizer により、AdvancedExternalizer でユーザークラスをアノテートする必要がない場合に基本的な Externalizable API 設定を使用するときに認識される欠点が修正されます。

バグを報告する

6.4.2. AdvancedExternalizer 設定例

以下の設定を使用して AdvancedExternalizer を使用します。

-

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

public class Person {

   final String name;
   final int age;

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

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

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

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

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

バグを報告する

6.4.3. エクスターナライザー ID

JBoss Data Grid の AdvancedExternalizers には、次のいずれかを使用して ID を提供するエクスターナライザー実装が必要です。

- **`getId()`** 実装。
- ペイロードのマーシャリング解除時にエクスターナライザーを識別する宣言またはプログラミングによる設定。

宣言またはプログラミングによる設定を使用してエクスターナライザーを登録する場合は、キャッシュマネージャーが作成されたときに登録を行う必要があります。

`GetId()` は、正の整数または null 値のいずれかを返します。

- 正の整数の場合は、読み取りを行い、内容を読み取ることができる正しいエクスターナライザーに割り当てるときに、エクスターナライザーを識別できます。
- null 値の場合は、AdvancedExternalizer の ID が宣言またはプログラミングによる設定で定義されるようになります。

任意の正の整数を使用できます (システムの他の ID で使用されていない場合)。

JBoss Data Grid は起動時に重複する ID をチェックし、重複がある場合に起動するプロセスを停止します。

バグを報告する

6.4.4. 高度なエクスターナライザーの登録

JBoss Data Grid では、AdvancedExternalizer 実装は、XML 、またはプログラミングによる設定、あるいはアノテーションにより登録できます。

静的な内部クラスとして格納された Person オブジェクトの高度なエクスターナライザー実装は、プログラミングまたは宣言により設定できます。

宣言による設定:

以下は、高度なエクスターナライザー実装の宣言による設定の例です。

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer
externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

プログラミングによる設定:

以下は、高度なエクスターナライザー実装のプログラミングによる設定の例です。

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

AdvancedExternalizer 実装は、使用される設定方法に関係なく、ID に関連付ける必要があります。

AdvancedExternalizer 実装の ID が XML/Programmatic 設定とアノテーションを使用して定義される場合は、XML/Programmatic で定義された値が使用されます。

以下の例は、登録時に ID が定義されるエクスターナライザーを示しています。

宣言による設定:

以下は、登録時の ID 定義の場所に対する宣言による設定です。

```
<infinispan>
```

```
  <global>
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer id="123"

externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

プログラミングによる設定:

以下は、登録時の ID 定義の場所に対するプログラミングによる設定です。

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
   .addAdvancedExternalizer(123, new Person.PersonExternalizer());
```

バグを報告する

6.4.5. 複数のエクスターナライザーをプログラミングにより登録

複数のエクスターナライザーは、JBoss Data Grid のプログラミングによる設定を使用して同時に登録できます。この機能を使用するには、**@Marshalls** アノテーションを使用して該当する ID がすでに定義されている必要があります。

以下は、複数のエクスターナライザーの登録例です。

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

バグを報告する

6.5. 内部エクスターナライザー実装アクセス

6.5.1. 内部エクスターナライザー実装アクセス

Externalizable オブジェクトは JBoss Data Grid エクスターナライザー実装にアクセスしないようにする必要があります。このための間違ったメソッドの例を以下に示します。

```
public static class ABCMarshallingExternalizer implements
AdvancedExternalizer<ABCMarshalling> {
   @Override
   public void writeObject(ObjectOutput output, ABCMarshalling object)
throws IOException {
      MapExternalizer ma = new MapExternalizer();
      ma.writeObject(output, object.getMap());
   }

   @Override
   public ABCMarshalling readObject(ObjectInput input) throws IOException,
```

```
ClassNotFoundException {
      ABCMarshalling hi = new ABCMarshalling();
      MapExternalizer ma = new MapExternalizer();
      hi.setMap((ConcurrentHashMap<Long, Long>) ma.readObject(input));
      return hi;
   }

   ...
```

エンドユーザーエクスターナライザーは内部のエクスターナライザークラスと対話する必要がありません。この状況に対処する間違ったメソッドの例を以下に示します。

```
public static class ABCMarshallingExternalizer implements
AdvancedExternalizer<ABCMarshalling> {
   @Override
   public void writeObject(ObjectOutput output, ABCMarshalling object)
throws IOException {
      output.writeObject(object.getMap());
   }

   @Override
   public ABCMarshalling readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      ABCMarshalling hi = new ABCMarshalling();
      hi.setMap((ConcurrentHashMap<Long, Long>) input.readObject());
      return hi;
   }

   ...
}
```

バグを報告する

パート II. リモートクライアントサーバーモードインターフェース

第7章 非同期 API

7.1. 非同期 API について

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

非同期メソッドの命名規則は、同期メソッドの命名規則と似ていますが、各メソッド名に **Async** を追加します。非同期メソッドは、操作の結果が含まれる Future を返します。

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

バグを報告する

7.2. 同期 API の利点

非同期 API はブロックしないため、以下のような複数の利点があります。

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

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

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

たとえば、以下の行は実行時にスレッドをブロックしません。

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

これら 3 つの put 操作からのリモートコールは同時に実行されます。これは、ディストリビューションモードで実行する場合に特に役に立ちます。これら 3 つのキーはクラスターの異なる 3 つのノードにプッシュされます。

バグを報告する

7.3. 非同期プロセスについて

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

- ネットワークコール
- マーシャリング

- キャッシュストアへの書き込み (オプション)
- ロック

JBoss Data Grid では、非同期メソッドを使用すると、クリティカルパスからネットワークコールとマーシャリングが削除されます。技術的な理由により、キャッシュストアへの書き込みおよびロック取得プロセスは、呼び出し側のスレッドで引き続き行われます。

バグを報告する

7.4. 戻り値と非同期 API

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

注記

NotifyingFutures は JBoss Data Grid のライブラリーモードのみで利用可能です。

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

```
Future.get()
```

バグを報告する

第8章 REST インターフェース

8.1. JBOSS DATA GRID の REST インターフェースについて

JBoss Data Grid は RESTful サービスを使用します。この主な利点は、密結合のクライアントライブラリおよびバインディングが不要になることです。REST API ではオーバーヘッドが発生し、REST クライアントまたはカスタムコードが REST 呼び出しを理解し、作成する必要があります。

JBoss Data Grid の唯一の要件は、HTTP クライアントライブラリーです。Java の場合は、Apache HTTP Commons Client が推奨されます。または、java.net API を使用できます。

バグを報告する

8.2. RUBY クライアントコード

以下のコードは ruby を使用して JBoss Data Grid REST API と対話する例です。提供されたコードは特別なライブラリーを必要とせず、標準的な net/HTTP ライブラリーで十分です。

```
require 'net/http'

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

#An example of how to create a new entry

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

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

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

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

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

#An example of Removing the remote copy of the key

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

#An example of creating binary data

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

バグを報告する

8.3. RUBY のサンプルで JSON を使用

前提条件

ruby で JavaScript Object Notation (JSON) を使用して JBoss Data Grid の REST インターフェースと対話するために、以下のコードを使用して要件を宣言します。

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

Ruby で JSON を使用

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

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

バグを報告する

8.4. PYTHON クライアントコード

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

```
import httplib

#How to insert data

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

#How to retrieve data

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

バグを報告する

8.5. JAVA クライアントコード

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

インポート

以下のようにインポートを定義します。

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

文字列値をキャッシュに追加

以下は、Java を使用して文字列値をキャッシュに追加する例です。

```
public class RestExample {

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

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

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

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

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

      connection.disconnect();
   }
```

キャッシュから文字列値を取得

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

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

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

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

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

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

      connection.connect();

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

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

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

      connection.disconnect();

      return stringBuilder.toString();
   }
```

以下のコードは java main メソッドの例です。

```
/**
     * Main method example.
     * @param args
     * @throws IOException
     */
public static void main(String\[\] args) throws IOException {
//Note that the cache name is "cacheX"
RestExample restExample = new RestExample();
restExample.putMethod("http://localhost:8080/rest/cacheX/1", "Infinispan
REST Test");
restExample.getMethod("http://localhost:8080/rest/cacheX/1");
   }
}
}
```

バグを報告する

8.6. REST インターフェースの設定

8.6.1. REST コネクターの設定

以下は、JBoss Data Grid のリモートクライアントサーバーモードの **rest-connector** 要素に対する設定要素です。

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
   <rest-connector virtual-server="default-host"
                   cache-container="default"
                   context-path="$CONTEXT_PATH"
                   security-domain="other"
                   auth-method="BASIC"
                   security-mode="WRITE" />
</subsystem>
```

バグを報告する

8.6.2. REST コネクター属性

以下は、JBoss Data Grid のリモートクライアントサーバーモードの REST コネクターを定義するために使用する属性のリストです。

- **rest-connector** 要素は、REST コネクターの設定情報を指定します。
  - ***virtual-server*** パラメーターは、REST コネクターで使用される仮想サーバーを指定します。このパラメーターのデフォルト値は **default-host** です。これはオプションパラメーターです。
  - ***cache-container*** パラメーターは、REST コネクターで使用されるキャッシュコンテナーを指定します。これは必須パラメーターです。
  - ***context-path*** パラメーターは、REST コネクターのコンテキストパスを指定します。このパラメーターのデフォルト値は空の文字列 (**""**) です。これはオプションパラメーターです。
  - ***security-domain*** パラメーターは、REST エンドポイントへのアクセスを認証するためにセキュリティーサブシステムで宣言された指定済みドメインを使用することを指定します。これはオプションパラメーターです。このパラメーターが省略されると、認証は実行されません。
  - ***auth-method*** パラメーターは、エンドポイントのクレデンシャルを取得するために使用するメソッドを指定します。このパラメーターのデフォルト値は **BASIC** です。サポートされる別の値には **DIGEST**、**CLIENT-CERT**、および **SPNEGO** があります。これはオプションパラメーターです。
  - ***security-mode*** パラメーターは、書き込み操作 (PUT、POST、DELETE など) または読み取り操作 (GET や HEAD など) に対してのみ認証が必要かどうかを指定します。このパラメーターの有効な値は **WRITE** (書き込み操作のみを認証する場合) または **READ_WRITE** (読み書き操作を認証する場合) です。

バグを報告する

8.7. REST インターフェースの使用

8.7.1. REST インターフェース操作

REST インターフェースは以下の操作を実行するために JBoss Data Grid のリモートクライアントサーバーモードで使用できます。

- データの追加。
- データの取得。
- データの削除。

バグを報告する

8.7.2. データの追加

8.7.2.1. REST インターフェースを使用したデータの追加

JBoss Data Grid の REST インターフェースで、次のメソッドを使用してデータをキャッシュに追加します。

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

PUT メソッドと **POST** メソッドが使用される場合、要求の本文には、ユーザーにより追加された情報を含むこのデータが含まれます。

PUT メソッドと **POST** メソッドの両方には、Content-Type ヘッダーが必要です。

バグを報告する

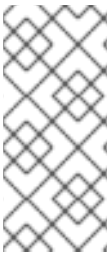
8.7.2.2. PUT /{cacheName}/{cacheKey} について

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

例として、以下の URL では、値 **hr** がキャッシュ名であり、**payRoll%2F3** がキーです。値 **%2F** は、**/** がキーで使用されたことを示します。

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

既存のデータは置き換えられ、更新が必要な場合は ***Time-To-Live*** 値と ***Last-Modified*** 値が更新されます。



注記

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

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

バグを報告する

8.7.2.3. POST /{cacheName}/{cacheKey} について

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

バグを報告する

8.7.3. データの取得

8.7.3.1. REST インターフェースを使用したデータの取得

JBoss Data Grid の REST インターフェースで、次のメソッドを使用してキャッシュからデータを取得します。

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

バグを報告する

8.7.3.2. GET /{cacheName}/{cacheKey} について

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

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

格納されたコンテンツのタイプは、返されたタイプです。例として、文字列が格納された場合は、文字列が返されます。シリアライズされた形式で格納されたオブジェクトは、手動でシリアライズ解除する必要があります。

バグを報告する

8.7.3.3. HEAD /{cacheName}/{cacheKey} について

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

バグを報告する

8.7.4. データの削除

8.7.4.1. REST インターフェースを使用したデータの削除

REST インターフェースを使用して JBoss Data Grid からデータを削除するには、HTTP **DELETE** メソッドを使用してキャッシュからデータを取得します。**DELETE** メソッドは以下のことを行えます。

- キャッシュエントリー/値を削除します。(**DELETE /{cacheName}/{cacheKey}**)
- キャッシュを削除します。(**DELETE /{cacheName}**)

バグを報告する

8.7.4.2. DELETE /{cacheName}/{cacheKey} について

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

バグを報告する

8.7.4.3. DELETE /{cacheName} について

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

バグを報告する

8.7.4.4. バックグラウンド削除操作

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

バグを報告する

8.7.5. REST インターフェース操作ヘッダー

8.7.5.1. ヘッダー

以下の表は、JBoss Data Grid REST インターフェースに含まれるヘッダーを示しています。

表8.1 ヘッダータイプ

ヘッダー	必須/オプション	値	デフォルト値	詳細
Content-Type	必須	-	-	Content-Type が **`application/x-java-serialized-object`** に設定された場合は、Java オブジェクトとして格納されます。

ヘッダー	必須/オプション	値	デフォルト値	詳細
performAsync	オプション	true/false	-	**true** に設定された場合は、すぐに返され、独自にクラスターにデータがレプリケートされます。この機能は、大量のデータ挿入と大きいクラスターを取り扱う場合に役に立ちます。
timeToLiveSeconds	オプション	数値 (正の値および負の値)	**-1** (この値により、timeToLiveSeconds の直接的な結果としてエクスパレーションが回避されます。このデフォルト値よりも、他の場所で設定されたエクスパレーションの値が優先されます。)	該当するエントリーが自動的に削除されるまでの秒数を反映します。timeToLiveSeconds に負の値を設定すると、デフォルト値と同じ結果が提供されます。
maxIdleTimeSeconds	オプション	数値 (正の値および負の値)	**-1** (この値により、maxIdleTimeSeconds の直接的な結果としてエクスパレーションが回避されます。このデフォルト値よりも、他の場所で設定されたエクスパレーションの値が優先されます。)	エントリーが自動的に削除される場合の、最後の使用時以降の秒数を含みます。負の値を渡すと、デフォルト値と同じ結果が提供されます。

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

- ***timeToLiveSeconds*** ヘッダーと ***maxIdleTimeSeconds*** ヘッダーに値 **0** が割り当てられた場合、キャッシュは、 XML を使用するか、またはプログラミングにより設定されたデフォルトの ***timeToLiveSeconds*** 値と ***maxIdleTimeSeconds*** 値を使用します。
- ***maxIdleTimeSeconds*** ヘッダー値のみが **0** に設定された場合は、***timeToLiveSeconds*** 値をパラメーター (または、デフォルトの **-1** (パラメーターが存在しない場合)) として渡す必要があります。また、***maxIdleTimeSeconds*** パラメーター値は、 XML を使用するか、プログラミングにより、設定された値にデフォルトで設定されます。

- ***timeToLiveSeconds*** ヘッダー値のみが **0** に設定された場合は、エクスパレーションが即座に発生し、***maxIdleTimeSeconds*** 値がパラメーターとして渡された値に設定されます (パラメーターが提供されなかった場合はデフォルトの **-1**)。

ETag ベースのヘッダー

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

表8.2 エンティティータグ関連ヘッダー

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

ヘッダー	アルゴリズム	例	詳細
If-Unmodified-Since	If-Unmodified-Since = "If-Unmodified-Since" ":" HTTP-date	If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT	要求されたバリアントの最終変更日時と、提供された時間および日付の値とを比較します。指定された日時以降に要求されたリソースが変更されなかった場合は、指定された操作が実行されます。指定された日時以降に要求されたリソースが変更された場合は、操作が実行されず、エンティティーの代わりに **412** (事前条件失敗) 応答が返されます。

バグを報告する

8.8. REST インターフェースセキュリティー

8.8.1. REST エンドポイントをパブリックインターフェースとして公開

JBoss Data Grid の REST サーバーはデフォルトで管理インターフェースとして動作します。操作をパブリックインターフェースに拡張するには、以下のように、**management** の **socket-binding** 要素の ***interface*** パラメーターの値を **public** に変更します。

```
<socket-binding name="http" interface="public" port="8080"/>
```

バグを報告する

8.8.2. REST エンドポイントのセキュリティーの有効化

前提条件

JBoss Data Grid では、**/docs/examples/configs**) にある JBoss Data Grid ディレクトリー内に **standalone-rest-auth.xml** サンプルファイルがあります。

このファイルを **$JDG_HOME/standalone/configuration** ディレクトリーにコピーして設定を使用します。**$JDG_HOME** から、以下のコマンドを入力して適切な場所に **standalone-rest-auth.xml** を作成します。

```
$ cp docs/examples/configs/standalone-rest-auth.xml
standalone/configuration/standalone.xml
```

必要な場合は、新しい設定テンプレートで使用するサンプル **standalone-rest-auth.xml** の新しいコピーを作成します。

手順8.1 REST エンドポイントのセキュリティーの有効化

REST インターフェースを使用する場合に JBoss Data Grid のセキュリティーを有効にするには、**standalone.xml** に以下の変更を行います。

1. **セキュリティーパラメーターの指定**
   rest エンドポイントで ***security-domain*** パラメーターおよび ***auth-method*** パラメーターの有効な値を指定するようにします。これらのパラメーターの推奨設定は以下のとおりです。

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
            <rest-connector virtual-server="default-host"
                            cache-container="local"
                            security-domain="other"
                            auth-method="BASIC"/>
</subsystem>
```

2. **セキュリティードメイン宣言のチェック**
   セキュリティーサブシステムに、対応するセキュリティードメイン宣言が含まれるようにします。セキュリティードメイン宣言の設定の詳細については、JBoss Application Server 7 または JBoss Enterprise Application Platform 6 ドキュメンテーションを参照してください。

3. **アプリケーションユーザーの追加**
   該当するスクリプトを実行し、アプリケーションユーザーを追加する設定を入力します。

   a. **adduser.sh** スクリプト (**$JDG_HOME/bin** に存在) を実行します。

      - Windows システムでは、**adduser.bat** ファイル (**$JDG_HOME/bin** に存在) を代わりに実行します。

   b. 追加するユーザーのタイプについて尋ねられたら、**b** を入力して **Application User (application-users.properties)** を選択します。

   c. リターンキーを押して、レルム (**ApplicationRealm**) のデフォルト値を使用します。

   d. ユーザー名とパスワードを指定します。

   e. 作成されたユーザーのロールを尋ねられたら、**REST** と入力します。

   f. プロンプトが表示されたら、ユーザー名とアプリケーションレルム情報が正しいことを確認し、"yes" と入力して作業を続行します。

4. **作成されたアプリケーションユーザーの確認**
   作成されたアプリケーションユーザーが正しく設定されていることを確認します。

   a. **application-users.properties** ファイル (**$JDG_HOME/standalone/configuration/** に存在) にリストされた設定を確認します。以下は、このファイルの正しい設定の例です。

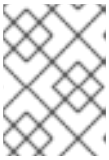
```
user1=2dc3eacfed8cf95a4a31159167b936fc
```

   b. **application-roles.properties** ファイル (**$JDG_HOME/standalone/configuration/** に存在) にリストされた設定を確認します。以下は、このファイルの正しい設定の例です。

```
user1=REST
```

5. **サーバーのテスト**
   サーバーを起動し、ブラウザーウィンドウに以下のリンクを入力して REST エンドポイントにアクセスします。

```
http://localhost:8080/rest/namedCache
```



注記

GET 要求の使用をテストする場合は、**405** 応答コードが期待され、サーバーが正常に認証されたことが示されます。

バグを報告する

第9章 MEMCACHED インターフェース

9.1. MEMCACHED プロトコルについて

Memcached は、データベース駆動 Web サイトの応答時間と操作時間を改善するために使用されるインメモリーキャッシングシステムです。Memcached キャッシングシステムは、Memached プロトコルと呼ばれるテキストベースのクライアントサーバーキャッシングプロトコルを定義します。
Memcached プロトコルはインメモリーオブジェクトを使用するか、(最後の手段として) 特殊な memcached データベースなどの永続ストアに渡されます。

JBoss Data Grid は、Memcached プロトコルを使用するサーバーを提供し、JBoss Data Grid と別に Memcached を使用する必要はありません。また、JBoss Data Grid のクラスタリング機能により、データフェールオーバー機能は Memcached で提供されるものよりも優れています。

バグを報告する

9.2. JBOSS DATA GRID 内の MEMCACHED サーバーについて

JBoss Data Grid には、memcached プロトコルを実装するサーバーモジュールが含まれます。これにより、memcached クライアントは1 つまたは複数の JBoss Data Grid ベース memcached サーバーと対話できるようになります。

サーバーは以下のいずれかになります。

- スタンドアロン。各サーバーは、他の memcached サーバーと通信せずに独立して動作します。
- クラスター。サーバーはデータを他の memcached サーバーにレプリケートおよび分散します。

バグを報告する

9.3. MEMCACHED インターフェースの使用

9.3.1. memcached 統計

以下の表には、JBoss Data Grid 内の memcached プロトコルを使用して利用可能な有効な統計のリストが含まれます。

表9.1 memcached 統計

統計	データタイプ	詳細
uptime	32 ビット符号なし整数。	memcached インスタンスが利用可能であり、実行されている時間 (秒数単位) を含みます。
time	32 ビット符号なし整数。	現在の時間を含みます。
version	文字列	現在のバージョンを含みます。

統計	データタイプ	詳細
curr_items	32 ビット符号なし整数。	インスタンスが現在格納しているアイテムの数を含みます。
total_items	32 ビット符号なし整数。	存続期間中にインスタンスにより格納されたアイテムの合計数を含みます。
cmd_get	64 ビット符号なし整数	get 操作要求 (データ取得要求) の合計数を含みます。
cmd_set	64 ビット符号なし整数	設定された操作要求 (データ格納要求) の合計数を含みます。
get_hits	64 ビット符号なし整数	要求されたキーにあるキーの数を含みます。
get_misses	64 ビット符号なし整数	要求されたキーにないキーの数を含みます。
delete_hits	64 ビット符号なし整数	削除するキー (特定され正常に削除されたキー) の数を含みます。
delete_misses	64 ビット符号なし整数	削除するキー (特定されず、削除できなかったキー) の数を含みます。
incr_hits	64 ビット符号なし整数	増分するキー (特定され正常に増分されたキー) の数を含みます。
incr_misses	64 ビット符号なし整数	増分するキー (特定されず、増分できなかったキー) の数を含みます。
decr_hits	64 ビット符号なし整数	減分するキー (特定され正常に減分されたキー) の数を含みます。
decr_misses	64 ビット符号なし整数	減分するキー (特定されず、減分できなかったキー) の数を含みます。
cas_hits	64 ビット符号なし整数	比較し、スワップするキー (特定され正常に比較およびスワップされたキー) の数を含みます。
cas_misses	64 ビット符号なし整数	比較し、スワップするキー (特定されず、比較およびスワップされなかったキー) の数を含みます。

統計	データタイプ	詳細
cas_badvalue	64 ビット符号なし整数	比較およびスワップが行われたが、元の値が提供された値に一致しなかったキーの数を含みます。
evictions	64 ビット符号なし整数	実行されたエビクションコールの数を含みます。
bytes_read	64 ビット符号なし整数	ネットワークからサーバーが読み取ったバイトの合計数を含みます。
bytes_written	64 ビット符号なし整数	ネットワークからサーバーが書き込んだバイトの合計数を含みます。

バグを報告する

9.4. MEMCACHED インターフェースの設定

9.4.1. JBoss Data Grid コネクターについて

JBoss Data Grid は、次の 3 つのコネクタータイプをサポートします。

- Hot Rod ベースコネクターの設定を定義する **hotrod-connector** 要素。
- memcached ベースコネクターの設定を定義する **memcached-connector** 要素。
- REST インターフェースベースのコネクターの設定を定義する **rest-connector** 要素。

バグを報告する

9.4.2. Memcached コネクターの設定

以下は、JBoss Data Grid のリモートクライアントサーバーモードの **memcached-connector** 要素に対する設定要素です。

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
<memcached-connector socket-binding="memcached"
                     cache-container="default"
                     worker-threads="4"
                     idle-timeout="-1"
                     tcp-nodelay="true"
                     send-buffer-size="0"
                     receive-buffer-size="0" />
</subsystem>
```

バグを報告する

9.4.3. Memcached コネクター属性

- 以下は、JBoss Data Grid のリモートクライアントサーバーモードの **connectors** 要素内にある memcached コネクターを設定するために使用する属性のリストです。
  - **memcached-connector** 要素は、memcached で使用する設定要素を定義します。
    - ***socket-binding*** パラメーターは、memcached コネクターで使用されるソケットバインディングポートを指定します。これは必須パラメーターです。
    - ***cache-container*** パラメーターは、memcached コネクターで使用されるキャッシュコンテナーを指定します。これは必須パラメーターです。
    - ***worker-threads*** パラメーターは、memcached コネクターで利用可能なワーカースレッドの数を指定します。このパラメーターのデフォルト値は、コアを 2 で乗算した数です。これはオプションパラメーターです。
    - ***idle-timeout*** パラメーターは、接続がタイムアウトするまでコネクターがアイドル状態のままになる時間 (ミリ秒単位) を指定します。このパラメーターのデフォルト値は **-1** です (タイムアウト期間が設定されません)。これは、オプションパラメーターです。
    - ***tcp-no-delay*** パラメーターは、TCP パケットが遅延され一括して送信されるかを指定します。このパラメーターの有効な値は **true** と **false** になります。このパラメーターのデフォルト値は、**true** です。これはオプションパラメーターです。
    - ***send-buffer-size*** パラメーターは、memcached コネクターの送信バッファーのサイズを指定します。このパラメーターのデフォルト値は TCP スタックバッファーのサイズです。これはオプションパラメーターです。
    - ***receive-buffer-size*** パラメーターは、memcached コネクターの受信バッファーのサイズを指定します。このパラメーターのデフォルト値は TCP スタックバッファーのサイズです。これはオプションパラメーターです。

バグを報告する

9.5. MEMCACHED インターフェースセキュリティー

9.5.1. Memcached エンドポイントをパブリックインターフェースとして公開

JBoss Data Grid の memcached サーバーはデフォルトで管理インターフェースとして動作します。操作をパブリックインターフェースに拡張するには、以下のように、**management** の **socket-binding** 要素の ***interface*** パラメーターの値を **public** に変更します。

```
<socket-binding name="memcached" interface="public" port="11211" />
```

バグを報告する

第10章 HOT ROD インターフェース

10.1. HOT ROD について

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

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

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

バグを報告する

10.2. JBOSS DATA GRID 内の HOT ROD サーバーについて

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

バグを報告する

10.3. HOT ROD ハッシュ機能

JBoss Data Grid は一貫性のあるハッシュ機能を使用してノードを配置し、ハッシュホイール上の対応するキーを使用してエントリーが存在する場所を決定します。

ハッシュ空間は、***Integer.MAX_INT*** に格納されます。この値は、Hot Rod クライアントが特定のハッシュ空間をデフォルト値と見なすことを防ぐために、ハッシュトポロジーの変更が検出されるたびに Hot Rod プロトコルを使用してクライアントに返されます。このハッシュ空間には **0** から ***Integer.MAX_INT*** までの正数のみを含めることができます。

JBoss Data Grid と対話するために Hot Rod プロトコルが使用された場合、プラットフォームに依存しない動作を実行するためにキー (およびその値) はバイトアレイである必要があります。スマートクライアント (バックグラウンドでハッシュ配信を認識します) はこのプラットフォームに依存しない方法でこのようなバイトアレイキーのハッシュコードを再計算できる必要があります。このために、必要な場合は、Java 以外のクライアントによる実装に対して、JBoss Data Grid で使用されたハッシュ機能のバージョン情報が保存されます。

バグを報告する

10.4. HOT ROD サーバーノード

10.4.1. サーバーノードハッシュ計算について

仮想ノードのサポートのため、たくさんの帯域幅を使用せずにすべての仮想ノード (場合によっては数千) がハッシュコードを返すことは現実的ではありません。結果として、クライアントは各サーバーに対してベース***hash ID*** または ***hash code*** を受け取ります。この情報は、各サーバーの実際のハッ

シュ位置を計算するために使用されます (仮想ノードが設定されているかどうかは関係ありません)。

バグを報告する

10.4.2. 一貫性のあるハッシュアルゴリズムについて

一貫性のあるハッシュアルゴリズムは、ハッシュ空間を円として配置します。ハッシュ空間のセグメントには所有者が割り当てられます。キーが所有者に割り当てられた場合は、キーを格納するセグメントを決定するために、キーのハッシュが使用されます。所有者が削除されると、そのセグメントが円内のネイバーに割り当てられます。つまり、所有者が削除されると、ハッシュ空間のほとんどが安定し、オーバーヘッドが小さくなります。

バグを報告する

10.4.3. クライアント用ハッシュコード計算ルール

サーバー用ハッシュコードを計算する場合、クライアントは特定のルールに従う必要があります。

仮想ノードが無効な場合:

クライアントがサーバーのベースハッシュコードを受け取る場合、ハッシュホイールの正確な位置を見つけるために、そのコードを正規化する必要があります。この正規化プロセスには以下のものが含まれます。

- ベースハッシュコードをハッシュ機能に渡す。
- マイナスの値を回避する計算。

結果はハッシュホイールのノードの位置を示す数です。

以下のコードは、正規化プロセスがどのように行われるかを示しています。

```
public static int getNormalizedHash(int nodeBaseHashCode, Hash hashFct) {
   return hashFct.hash(nodeBaseHashCode) & Integer.MAX_VALUE;
}
```

仮想ノードが有効な場合:

各ノードは *N* 個の異なる仮想ノードを表します。したがって、各仮想ノードのハッシュコードを計算するには、**`0`**〜**`N-1`** の数を使用し、以下のプロセスを適用します。

手順10.1 仮想ノードでのハッシュコード計算

1. ID "0" の仮想ノードの場合は、以下のコードを使用してハッシュコードを取得します。

```
public static int getNormalizedHash(int nodeBaseHashCode, Hash
hashFct) {
   return hashFct.hash(nodeBaseHashCode) & Integer.MAX_VALUE;
}
```

2. 以降のすべての仮想ノード ("1" 〜 "N-1" の ID を持つ) の場合は、以下のコードを実行します。

```
public static int virtualNodeHashCode(int nodeBaseHashCode, int id,
Hash hashFct) {
   int virtualNodeBaseHashCode = id;
```

```
      virtualNodeBaseHashCode = 31 * virtualNodeBaseHashCode +
nodeBaseHashCode;
      return getNormalizedHash(virtualNodeBaseHashCode, hashFct);
   }
```

バグを報告する

10.4.4. hotrod.properties ファイル

リモートキャッシュストアの設定を使用するには、hotrod.properties ファイルを作成し、リモートキャッシュストアの設定に関連するクラスパスに含まれるようにする必要があります。

hotrod.properties ファイルにはプロパティーが 1 つ以上含まれます。最も単純な hotrod.properties ファイルには以下が含まれます。

```
infinispan.client.hotrod.server_list=remote-server:11222
```

hotrod.properties に含めることができるプロパティーは次の通りです。

`infinispan.client.hotrod.request_balancing_strategy`

レプリケートされた Hot Rod サーバークラスターでは、このストラテジーに従ってクライアントがサーバーへの要求のバランスを取ります。

このプロパティーのデフォルト値は **`org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy`** です。

`infinispan.client.hotrod.server_list`

接続する Hot Rod サーバーの初期リストです。「host1:port1;host2:port2」 のような形式で指定されます。最低でも 1 つの host:port を指定する必要があります。

このプロパティーのデフォルト値は **`127.0.0.1:11222`** です。

`infinispan.client.hotrod.force_return_values`

すべての呼び出しに対して暗黙的に Flag.FORCE_RETURN_VALUE を行うかどうか。

このプロパティーのデフォルト値は **`false`** です。

`infinispan.client.hotrod.tcp_no_delay`

TCP スタックの TCP NODELAY に影響します。

このプロパティーのデフォルト値は **`true`** です。

`infinispan.client.hotrod.ping_on_startup`

true の場合、クラスターのトポロジーを取り込むため、ping リクエストがバックエンドサーバーへ送信されます。

このプロパティーのデフォルト値は **`true`** です。

`infinispan.client.hotrod.transport_factory`

使用されるトランスポートを制御します。現在、TcpTransport のみがサポートされます。

このプロパティーのデフォルト値は **org.infinispan.client.hotrod.impl.transport.tcp.TcpTransportFactory** です。

infinispan.client.hotrod.marshaller

ユーザーオブジェクトをシリアライズおよびデシリアライズするため、カスタムマーシャラーの実装を指定できるようにします。

このプロパティーのデフォルト値は **org.infinispan.marshall.jboss.GenericJBossMarshaller** です。

infinispan.client.hotrod.async_executor_factory

非同期呼び出しのカスタム非同期エクゼキューターを指定できるようにします。

このプロパティーのデフォルト値は **org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory** です。

infinispan.client.hotrod.default_executor_factory.pool_size

デフォルトのエクゼキューターが使用される場合、エクゼキューターを初期化するスレッド数を設定します。

このプロパティーのデフォルト値は **10** です。

infinispan.client.hotrod.default_executor_factory.queue_size

デフォルトのエクゼキューターが使用される場合、エクゼキューターを初期化するキューサイズを設定します。

このプロパティーのデフォルト値は **100000** です。

infinispan.client.hotrod.hash_function_impl.1

ハッシュ関数のバージョンと使用中の一貫したハッシュアルゴリズムを指定します。使用される Hot Rod サーバーのバージョンと密接に関係しています。

このプロパティーのデフォルト値は **Hash function specified by the server in the responses as indicated in ConsistentHashFactory** です。

infinispan.client.hotrod.key_size_estimate

このヒントは、キーをシリアライズおよびデシリアライズする時にアレイのサイズ変更を最小限にするため、バイトバッファーのサイズを設定できるようにします。

このプロパティーのデフォルト値は **64** です。

infinispan.client.hotrod.value_size_estimate

このヒントは、値をシリアライズおよびデシリアライズする時にアレイのリサイズを最小限にするため、バイトバッファーをサイジングできるようにします。

このプロパティーのデフォルト値は **512** です。

infinispan.client.hotrod.socket_timeout

このプロパティーは、サーバーからのバイト待ちを断念する前の最大ソケット読み取りタイムアウトを定義します。

The default value for this property is **`60000 (equals 60 seconds)`**.

`infinispan.client.hotrod.protocol_version`

このプロパティーは、クライアントが使用する必要があるプロトコルのバージョンを定義します。他の有効値には 1.0 が含まれます。

このプロパティーのデフォルト値は **`1.1`** です。

`infinispan.client.hotrod.connect_timeout`

このプロパティーは、サーバーへの接続を断念する前の最大ソケット接続タイムアウトを定義します。

The default value for this property is **`60000 (equals 60 seconds)`**.

関連トピック:

- 「RemoteCache インターフェースについて」

バグを報告する

10.5. HOT ROD ヘッダー

10.5.1. Hot Rod ヘッダーデータタイプ

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

表10.1 1ヘッダーデータタイプ

データタイプ	サイズ	詳細
vInt	1〜5 バイト。	符号なし可変長整数値。
vLong	1〜9 バイト。	符号なし可変長ロング値。
文字列	-	文字列は常に UTF-8 エンコーディングを使用して表されます。

バグを報告する

10.5.2. 要求ヘッダー

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

表10.2 要求ヘッダーフィールド

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

フィールド名	データタイプ/サイズ	詳細
Topology ID	vInt	クライアントの最後の既知なビュー ID を含みます。基本的なクライアントはこのフィールドに値 `0` を提供します。トポロジーまたはハッシュ情報をサポートするクライアントは、サーバーが現在のビュー ID に応答するまで値 `0` (新しいビュー ID が現在のビュー ID を置き換えるためにサーバーにより返されるまで使用されます) を提供します。
Transaction Type	1 バイト	2 つの既知のトランザクションタイプのいずれかを表す値を含みます。現時点でサポートされている値は `0` のみです。
Transaction ID	バイトアレイ	呼び出しに関連するトランザクションを一意に識別するバイトアレイを含みます。トランザクションタイプはこのバイトアレイの長さを決定します。***`Transaction Type`*** の値が `0` に設定された場合、トランザクション ID は存在しません。

バグを報告する

10.5.3. 応答ヘッダー

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

表10.3 応答ヘッダーフィールド

フィールド名	データタイプ	詳細
Magic	1 バイト	ヘッダーが要求または応答ヘッダーであるかどうかを示します。
Message ID	vLong	メッセージ ID を含みます。この一意の ID は、応答を元の要求とペアにするために使用されます。これにより、Hot Rod クライアントは非同期でプロトコルを実装できるようになります。

フィールド名	データタイプ	詳細
opcode	1 バイト	関連する操作コードを含みます。応答ヘッダー内でopcode には応答操作コードのみを含めることができます。
Status	1 バイト	応答のステータスを表すコードを含みます。
Topology Change Marker	1 バイト	応答がトポロジー変更情報に含まれるかどうかを示すマーカーバイトを含みます。

バグを報告する

10.5.4. トポロジー変更ヘッダー

10.5.4.1. トポロジー変更ヘッダーについて

Hot Rod を使用して JBoss Data Grid にアクセスする場合は、応答ヘッダーが、異なるトポロジーまたはハッシュ配布を区別できるクライアントを探すことによりクラスターまたはビューフォーメーションの変更に応答します。Hot Rod サーバーは現在の ***`topology ID`*** と、クライアントにより送信された ***`topology ID`*** を比較し、2 つの値が異なる場合は、新しい ***`topology ID`*** を返します。

バグを報告する

10.5.4.2. トポロジー変更マーカー値

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

表10.4 Topology Change Marker フィールド値

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

バグを報告する

10.5.4.3. トポロジー認識クライアントのトポロジー変更ヘッダー

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

表10.5 トポロジー変更ヘッダーフィールド

応答ヘッダーフィールド	データタイプ/サイズ	詳細
Response Header with Topology Change Marker	-	-
Topology ID	vInt	-
Num Servers in Topology	vInt	クラスターで稼働している Hot Rod サーバーの数を含みます。一部のノードのみが Hot Rod サーバーを稼働している場合に、この値は、クラスター全体のサブセットになることがあります。
mX: Host/IP Length	vInt	個別クラスターメンバーのホスト名または IP アドレスの長さを含みます。可変長により、この要素にはホスト名、IPv4、および IPv アドレスを含めることができます。
mX: Host/IP Address	文字列	個別クラスターメンバーのホスト名または IP アドレスを含みます。Hot Rod クライアントはこの情報を使用して個別クラスターメンバーにアクセスします。
mX: Port	符号なしショート。2 バイト	クラスターメンバーと通信するために Hot Rod クライアントが使用するポートを含みます。

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

バグを報告する

10.5.4.4. ハッシュ配布認識クライアントのトポロジー変更ヘッダー

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

表10.6 トポロジー変更ヘッダーフィールド

フィールド	データタイプ/サイズ	詳細
Response Header with Topology Change Marker	-	-
Topology ID	vInt	-

フィールド	データタイプ/サイズ	詳細
Number Key Owners	符号なしショート。2 バイト	配布された各キーに対してグローバルに設定されたコピーの数を含みます。配布がキャッシュで設定されていない場合は、値 **0** を含みます。
Hash Function Version	1 バイト	使用中のハッシュ機能へのポインターを含みます。配布がキャッシュで設定されていない場合は、値 **0** を含みます。
Hash Space Size	vInt	ハッシュコード生成に関連するすべてのモジュール計算のために JBoss Data Grid により使用されるモジュールを含みます。クライアントはこの情報を使用して正しいハッシュ計算をキーに適用します。配布がキャッシュで設定されていない場合は、値 **0** を含みます。
Number servers in topology	vInt	クラスター内で稼働している **Hot Rod** サーバーの数を含みます。一部のノードのみが **Hot Rod** サーバーを稼働している場合に、この値は、クラスター全体のサブセットになることがあります。また、この値はヘッダーに含まれるホストとポートのペアの数を表します。
Number Virtual Nodes Owners	vInt	設定された仮想ノードの数を含みます。仮想ノードが設定されていない場合、または配布がキャッシュで設定されていない場合は、値 **0** を含みます。
mX: Host/IP Length	vInt	個別クラスターメンバーのホスト名または **IP** アドレスの長さを含みます。可変長により、この要素にはホスト名、 **IPv4** 、および **IPv6** アドレスを含めることができます。

フィールド	データタイプ/サイズ	詳細
mX: Host/IP Address	文字列	個別クラスターメンバーのホスト名または **IP** アドレスを含みます。 **Hot Rod** クライアントはこの情報を使用して個別クラスターメンバーにアクセスします。
mX: Port	符号なしショート。2 バイト	クラスターメンバーと通信するために **Hot Rod** クライアントが使用するポートを含みます。
mX: Hashcode	4 バイト。	

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

バグを報告する

10.6. HOT ROD 操作

10.6.1. Hot Rod 操作

以下は、Hot Rod を使用して JBoss Data Grid と対話する場合に有効な操作です。

- Get
- BulkGet
- GetWithVersion
- Put
- PutIfAbsent
- Remove
- RemoveIfUnmodified
- Replace
- ReplaceIfUnmodified
- Clear
- ContainsKey
- Ping
- Stats

バグを報告する

10.6.2. Hot Rod Get 操作

Hot Rod **Get** 操作は、以下の要求形式を使用します。

表10.7 Get 操作要求形式

フィールド	データタイプ	詳細
Header	-	-
Key Length	vInt	キーの長さを含みます。***Integer.MAX_VALUE*** のサイズよりも大きいサイズ (最大 **6** バイト) のため、**vInt** データタイプが使用されます。ただし、Java では、単一アレイサイズを ***Integer.MAX_VALUE*** のサイズよりも大きくすることはできません。結果として、この vInt は ***Integer.MAX_VALUE*** の最大サイズに限定されます。
Key	バイトアレイ	キーを含みます (このキーの対応する値が要求されます)。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表10.8 Get 操作応答形式

応答ステータス	詳細
0x00	操作が成功。
0x02	キーが存在しない。

キーが見つかった場合の **get** 操作の応答の形式は以下のとおりです。

表10.9 Get 操作応答形式

フィールド	データタイプ	詳細
Header	-	-
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

バグを報告する

10.6.3. Hot Rod BulkGet 操作

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

表10.10 BulkGet 操作要求形式

フィールド	データタイプ	詳細
Header	-	-
Entry Count	vInt	サーバーが返す JBoss Data Grid エントリーの最大数を含みます。エントリー数の値はキーおよび関連する値の合計と等しくなります。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表10.11 BulkGet 操作応答形式

フィールド	データタイプ	詳細
Header	-	-
More	vInt	ストリームからエントリーをさらに読み取る必要があるかどうかを示します。***`More`*** は **`1`** に設定される一方で、More の値が **`0`** (ストリームの最後を示します) に設定されるまで追加のエントリーが続きます。
Key Size	-	キーのサイズを含みます。
Key	-	キーの値を含みます。
Value Size	-	値のサイズを含みます。
Value	-	値を含みます。

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

バグを報告する

10.6.4. Hot Rod GetWithVersion 操作

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

表10.12 GetWithVersion 操作要求形式

フィールド	データタイプ	詳細
Header	-	-
Key Length	vInt	キーの長さを含みます。***Integer.MAX_VALUE*** のサイズよりも大きいサイズ (最大 **6** バイト) のため、**vInt** データタイプが使用されます。ただし、Java では、単一アレイサイズを ***Integer.MAX_VALUE*** のサイズよりも大きくすることはできません。結果として、この **vInt** は **Integer.MAX_VALUE** の最大サイズに限定されます。
キー	バイトアレイ	キーを含みます (このキーの対応する値が要求されます)。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表10.13 GetWithVersion 操作応答形式

応答ステータス	詳細
0x00	操作が成功。
0x02	キーが存在しない。

この操作の応答には以下のものが含まれます。

表10.14

フィールド	データタイプ/サイズ	詳細
Entry Version	8 バイト	既存のエントリーの変更の一意の値を含みます。
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

バグを報告する

10.6.5. Hot Rod Put 操作

put 操作要求形式には、以下のものが含まれます。

表10.15

フィールド	データタイプ	詳細詳細
Header	-	-
Key Length	-	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。
Lifespan	vInt	エントリーが期限切れになるまでの秒数を含みます。秒数が 30 日を超える場合、その値はエントリーライフスパンの UNIX 時間 (つまり、日付 **`1/1/1970`** 以降の秒数) として処理されます。値が **`0`** に設定された場合、エントリーは期限切れになりません。
Max Idle	vInt	キャッシュからエビクトされるまでエントリーがアイドル状態のままになることが許可される秒数を含みます。このエントリーが **`0`** に設定された場合、エントリーは無期限でアイドル状態のままになることが許可され、***`max idle`*** 値のため、エビクトされません。
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値。

以下は、この操作から返された応答値です。

表10.16

応答ステータス	詳細詳細
0x00	値が正常に格納されました。

この操作では空の応答がデフォルト応答になります。ただし、***`ForceReturnPreviousValue`*** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **`0`** が含まれます。

バグを報告する

10.6.6. Hot Rod PutIfAbsent 操作

`putIfAbsent` 操作要求形式には、以下のものが含まれます。

表10.17 PutIfAbsent 操作要求フィールド

フィールド	データタイプ	詳細
Header	-	-
Key Length	vInt	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。
Lifespan	vInt	エントリーが期限切れになるまでの秒数を含みます。秒数が 30 日を超える場合、その値はエントリーライフスパンの UNIX 時間 (つまり、日付 **`1/1/1970`** 以降の秒数) として処理されます。値が **`0`** に設定された場合、エントリーは期限切れになりません。
Max Idle	vInt	キャッシュからエビクトされるまでエントリーがアイドル状態のままになることが許可される秒数を含みます。このエントリーが **`0`** に設定された場合、エントリーは無期限でアイドル状態のままになることが許可され、max idle 値のため、エビクトされません。
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

以下は、この操作から返された応答値です。

表10.18

応答ステータス	詳細
0x00	値が正常に格納されました。
0x01	キーが存在しないため、値が格納されませんでした。

この操作では空の応答がデフォルト応答になります。ただし、***`ForceReturnPreviousValue`*** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **`0`** が含まれます。

バグを報告する

10.6.7. Hot Rod Remove 操作

`Hot RodRemove` 操作は、以下の要求形式を使用します。

表10.19 Remove 操作要求形式

フィールド	データタイプ	詳細
Header	-	-
Key Length	vInt	キーの長さを含みます。***`Integer.MAX_VALUE`*** のサイズよりも大きいサイズ (最大 **6** バイト) のため、**`vInt`** データタイプが使用されます。ただし、Java では、単一アレイサイズを ***`Integer.MAX_VALUE`*** のサイズよりも大きくすることはできません。結果として、この **`vInt`** は **`Integer.MAX_VALUE`** の最大サイズに限定されます。
Key	バイトアレイ	キーを含みます (このキーの対応する値が要求されます)。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表10.20 Remove 操作応答形式

応答ステータス	詳細
0x00	操作が成功。
0x02	キーが存在しない。

通常、この操作の応答ヘッダーは空白です。ただし、***`ForceReturnPreviousValue`*** が渡された場合は、応答ヘッダーに以下のいずれかが含まれます。

- 以前のキーの値および長さ。
- キーが存在しないことを示す、値の長さ **`0`** と応答ステータス **`0x02`**。

remove 操作の応答ヘッダーには、提供されたキーの以前の値と、以前の値の長さが含まれます (***`ForceReturnPreviousValue`*** が渡された場合)。キーが存在しない場合、または以前の値が null の場合、値の長さは **`0`** です。

バグを報告する

10.6.8. Hot Rod RemoveIfUnmodified 操作

`RemoveIfUnmodified` 操作要求形式には、以下のものが含まれます。

表10.21 RemoveIfUnmodified 操作要求フィールド

フィールド	データタイプ	詳細
Header	-	-
Key Length	vInt	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。
Entry Version	8 バイト	**`GetWithVersion`** 操作により返された値を使用します。

以下は、この操作から返された応答値です。

表10.22 RemoveIfUnmodified 操作応答

応答ステータス	詳細
0x00	エントリーが置換または削除された場合に返されたステータス。
0x01	キーが変更されたため、エントリーの置換または削除が失敗した場合に、ステータスを返します。
0x02	キーが存在しない場合に、ステータスを返します。

この操作では空の応答がデフォルト応答になります。ただし、***`ForceReturnPreviousValue`*** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **`0`** が含まれます。

バグを報告する

10.6.9. Hot Rod replace 操作

`replace` 操作要求形式には、以下のものが含まれます。

表10.23 replace 操作要求フィールド

フィールド	データタイプ	詳細
Header	-	-
Key Length	vInt	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。

フィールド	データタイプ	詳細
Lifespan	vInt	エントリーが期限切れになるまでの秒数を含みます。秒数が 30 日を超える場合、その値はエントリーライフスパンの UNIX 時間 (つまり、日付 **`1/1/1970`** 以降の秒数) として処理されます。値が **`0`** に設定された場合、エントリーは期限切れになりません。
Max Idle	vInt	キャッシュからエビクトされるまでエントリーがアイドル状態のままになることが許可される秒数を含みます。このエントリーが **`0`** に設定された場合、エントリーは無期限でアイドル状態のままになることが許可され、max idle 値のため、エビクトされません。
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

以下は、この操作から返された応答値です。

表10.24 replace 操作応答

応答ステータス	詳細
0x00	値が正常に格納されました。
0x01	キーが存在しないため、値が格納されませんでした。

この操作では空の応答がデフォルト応答になります。ただし、***`ForceReturnPreviousValue`*** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **`0`** が含まれます。

バグを報告する

10.6.10. Hot Rod ReplaceIfUnmodified 操作

`ReplaceIfUnmodified` 操作要求形式には、以下のものが含まれます。

表10.25 ReplaceIfUnmodified 操作要求フィールド

フィールド	データタイプ	詳細
Header	-	-

フィールド	データタイプ	詳細
Key Length	vInt	キーの長さを含みます。
Key	バイトアレイ	キーの値を含みます。
Lifespan	vInt	エントリーが期限切れになるまでの秒数を含みます。秒数が 30 日を超える場合、その値はエントリーライフスパンの UNIX 時間 (つまり、日付 **`1/1/1970`** 以降の秒数) として処理されます。値が **`0`** に設定された場合、エントリーは期限切れになりません。
Max Idle	vInt	キャッシュからエビクトされるまでエントリーがアイドル状態のままになることが許可される秒数を含みます。このエントリーが **`0`** に設定された場合、エントリーは無期限でアイドル状態のままになることが許可され、max idle 値のため、エビクトされません。
Entry Version	8 バイト	**`GetWithVersion`** 操作により返された値を使用します。
Value Length	vInt	値の長さを含みます。
Value	バイトアレイ	要求された値を含みます。

以下は、この操作から返された応答値です。

表10.26 ReplaceIfUnmodified 操作応答

応答ステータス	詳細
0x00	エントリーが置換または削除された場合に返されたステータス。
0x01	キーが変更されたため、エントリーの置換または削除が失敗した場合に、ステータスを返します。
0x02	キーが存在しない場合に、ステータスを返します。

この操作では空の応答がデフォルト応答になります。ただし、***`ForceReturnPreviousValue`*** が渡された場合は、以前の値とキーが返されます。以前のキーと値が存在しない場合は、値の長さに値 **`0`** が含まれます。

バグを報告する

10.6.11. Hot Rod clear 操作

clear 操作形式には、ヘッダーのみ含まれます。

この操作に有効な応答ステータスは以下のとおりです。

表10.27 clear 操作応答

応答ステータス	詳細
0x00	JBoss Data Grid が正常に消去されました。

バグを報告する

10.6.12. Hot Rod ContainsKey 操作

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

表10.28 ContainsKey 操作要求形式

フィールド	データタイプ	詳細
Header	-	-
Key Length	vInt	キーの長さを含みます。***Integer.MAX_VALUE*** のサイズよりも大きいサイズ (最大 **6** バイト) のため、**vInt** データタイプが使用されます。ただし、Java では、単一アレイサイズを ***Integer.MAX_VALUE*** のサイズよりも大きくすることはできません。結果として、この **vInt** は **Integer.MAX_VALUE** の最大サイズに限定されます。
Key	バイトアレイ	キーを含みます (このキーの対応する値が要求されます)。

この操作の応答ヘッダーには、以下のいずれかの応答ステータスが含まれます。

表10.29 ContainsKey 操作応答形式

応答ステータス	詳細
0x00	操作が成功。
0x02	キーが存在しない。

この操作の応答は空白です。

バグを報告する

10.6.13. Hot Rod ping 操作

`ping` は、サーバーの可用性を確認するアプリケーションレベルの要求です。

この操作に有効な応答ステータスは以下のとおりです。

表10.30 ping 操作応答

応答ステータス	詳細
0x00	エラーなしの正常な ping。

バグを報告する

10.6.14. Hot Rod 統計操作

この操作は、利用可能なすべての統計の概要を返します。返された各統計に対して、名前と値が文字列形式と UTF-8 形式の両方で返されます。

この操作では、以下の統計がサポートされます。

表10.31 統計操作要求フィールド

名前	詳細
timeSinceStart	Hot Rod が起動した以降の秒数を含みます。
currentNumberOfEntries	Hot Rod サーバーに現在存在するエントリーの数を含みます。
totalNumberOfEntries	Hot Rod サーバーに格納されたエントリーの合計数を含みます。
stores	put 操作の試行回数を含みます。
retrievals	get 操作の試行回数を含みます。
hits	get ヒット数を含みます。
misses	get 失敗数を含みます。
removeHits	remove ヒット数を含みます。
removeMisses	removal 失敗数を含みます。

この操作の応答ヘッダーには以下のものが含まれます。

表10.32 統計操作応答

名前	データタイプ	詳細
Header	-	-
Number of Stats	vInt	返された個別統計の数を含みます。
Name Length	vInt	名前付き統計の長さを含みます。
名前	文字列	統計の名前を含みます。
Value Length	vInt	値の長さを含みます。
Value	文字列	統計値を含みます。

要求された各統計に対して、値 ***Name Length***、***Name***、***Value Length***、および ***Value*** が繰り返されます。

バグを報告する

10.6.15. Hot Rod 操作の値

10.6.15.1. opcode 要求および応答の値

以下は、要求ヘッダーと対応する応答ヘッダー値の有効な ***opcode*** 値のリストです。

表10.33 opcode 要求および応答ヘッダー値

操作	要求操作コード	応答操作コード
put	0x01	0x02
get	0x03	0x04
putIfAbsent	0x05	0x06
replace	0x07	0x08
replaceIfUnmodified	0x09	0x0A
remove	0x0B	0x0C
removeIfUnmodified	0x0D	0x0E
containsKey	0x0F	0x10
getWithVersion	0x11	0x12

操作	要求操作コード	応答操作コード
clear	0x13	0x14
stats	0x15	0x16
ping	0x17	0x18
bulkGet	0x19	0x1A

また、応答ヘッダーの ***opcode*** 値が **`0x50`** の場合は、エラー応答を示します。

バグを報告する

10.6.15.2. Magic 値

以下は要求および応答ヘッダー内の ***`Magic`*** フィールドの有効な値のリストです。

表10.34 Magic フィールド値

値	詳細
0xA0	キャッシュ要求マーカー。
0xA1	キャッシュ応答マーカー。

バグを報告する

10.6.15.3. ステータス値

以下は、応答ヘッダー内の ***`Status`*** フィールドに対するすべての有効な値を含む表です。

表10.35 ステータス値

値	詳細
0x00	エラーなし。
0x01	配置、削除、置換なし。
0x02	キーは存在しない。
0x81	無効なマジック値またはメッセージ ID。
0x82	不明なコマンド。
0x83	不明なバージョン。

値	詳細
0x84	要求解析エラー。
0x85	サーバーエラー。
0x86	コマンドタイムアウト。

バグを報告する

10.6.15.4. トランザクションタイプ値

以下は、要求ヘッダー内の ***`Transaction Type`*** の有効な値のリストです。

表10.36 Transaction Type フィールド値

値	詳細
0	非トランザクション呼び出し、またはクライアントがトランザクションをサポートしないことを示します。使用された場合は、***`TX_ID`*** フィールドが省略されます。
1	X/Open XA トランザクション ID (XID) を示します。この値は現在サポートされていません。

バグを報告する

10.6.15.5. Client Intelligence 値

以下は、要求ヘッダー内の ***`Client Intelligence`*** の有効な値のリストです。

表10.37 Client Intelligence フィールド値

値	詳細
0x01	クラスターまたはハッシュ情報が必要でない基本的なクライアントを示します。
0x02	トポロジーを認識し、クラスター情報が必要なクラスターを示します。
0x03	ハッシュと配布を認識し、クラスターおよびハッシュ情報が必要なクライアントを示します。

バグを報告する

10.6.15.6. フラグ値

以下は、要求ヘッダー内の有効な ***flag*** 値のリストです。

表10.38 フラグフィールド値

値	詳細
0x0001	ForceReturnPreviousValue

バグを報告する

10.6.15.7. Hot Rod エラー処理

表10.39 応答ヘッダーフィールドを使用した Hot Rod エラー処理

フィールド	データタイプ	詳細
Error Opcode	-	エラー操作コードを含みます。
Error Status Number	-	***error opcode*** に対応するステータス番号を含みます。
Error Message Length	vInt	エラーメッセージの長さを含みます。
Error Message	文字列	実際のエラーメッセージを含みます。要求の解析エラーが存在することを示す **0x84** エラーコードが返された場合、このフィールドには、 **Hot Rod** サーバーでサポートされた最新バージョンが含まれます。

バグを報告する

10.7. 例

10.7.1. put 要求の例

以下は、Hot Rod を使用した **put** 要求例からのコーディングされた要求です。

表10.40 put 要求の例

バイト	0	1	2	3	4	5	6	7
8	0xA0	0x09	0x41	0x01	0x07	0x4D ('M')	0x79 ('y')	0x43 ('C')
16	0x61 ('a')	0x63 ('c')	0x68 ('h')	0x65 ('e')	0x00	0x03	0x00	0x00

バイト	0	1	2	3	4	5	6	7
24	0x00	0x05	0x48 ('H')	0x65 ('e')	0x6C ('l')	0x6C ('l')	0x6F ('o')	0x00
32	0x00	0x05	0x57 ('W')	0x6F ('o')	0x72 ('r')	0x6C ('l')	0x64 ('d')	-

以下の表には、要求の例に対するすべてのヘッダーフィールドと値が含まれます。

表10.41 要求例のフィールド名と値

フィールド名	バイト	Value
Magic	0	0xA0
Version	2	0x41
Cache Name Length	4	0x07
Flag	12	0x00
Topology ID	14	0x00
Transaction ID	16	0x00
Key	18-22	'Hello'
Max Idle	24	0x00
Value	26-30	'World'
Message ID	1	0x09
Opcode	3	0x01
Cache Name	5-11	'MyCache'
Client Intelligence	13	0x03
Transaction Type	15	0x00
Key Field Length	17	0x05
Lifespan	23	0x00
Value Field Length	25	0x05

以下は、**put** 要求の例に対するコーディングされた応答です。

表10.42 put 要求の例のコーディングされた応答

バイト	0	1	2	3	4	5	6	7
8	0xA1	0x09	0x01	0x00	0x00	-	-	-

以下の表には、応答の例に対するすべてのヘッダーフィールドと値が含まれます。

表10.43 応答例のフィールド名および値

フィールド名	バイト	Value
Magic	0	0xA1
Opcode	2	0x01
Topology Change Marker	4	0x00
Message ID	1	0x09
Status	3	0x00

バグを報告する

10.8. HOT ROD インターフェースの設定

10.8.1. JBoss Data Grid コネクターについて

JBoss Data Grid は、次の 3 つのコネクタータイプをサポートします。

- Hot Rod ベースコネクターの設定を定義する **hotrod-connector** 要素。
- memcached ベースコネクターの設定を定義する **memcached-connector** 要素。
- REST インターフェースベースのコネクターの設定を定義する **rest-connector** 要素。

バグを報告する

10.8.2. Hot Rod コネクターの設定

以下は、JBoss Data Grid のリモートクライアントサーバーモードの **hotrod-connector** 要素に対する設定要素です。

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
      <hotrod-connector socket-binding="hotrod"
                        cache-container="default"
                        worker-threads="4"
                        idle-timeout="-1"
```

```
                        tcp-nodelay="true"
                        send-buffer-size="0"
                        receive-buffer-size="0"   />
        <topology-state-transfer lock-timeout="10000"
                                 replication-timeout="10000"
                                 update-timeout="30000"
                                 external-host="192.168.0.1"
                                 external-port="11222"
                                 lazy-retrieval="true" />
    </subsystem>
```

バグを報告する

10.8.3. Hot Rod コネクター属性

以下は、JBoss Data Grid のリモートクライアントサーバーモードの Hot Rod コネクターを定義するために使用する属性のリストです。

Hotrod-Connector 要素

hotrod-connector 要素は、Hot Rod で使用する設定要素を定義します。

- ***socket-binding*** パラメーターは、Hot Rod コネクターで使用されるソケットバインディングポートを指定します。これは必須パラメーターです。
- ***cache-container*** パラメーターは、Hot Rod コネクターで使用されるキャッシュコンテナーを指定します。これは必須パラメーターです。
- ***worker-threads*** パラメーターは、Hot Rod コネクターで利用可能なワーカースレッドの数を指定します。このパラメーターのデフォルト値は、コアを 2 で乗算した数です。これはオプションパラメーターです。
- ***idle-timeout*** パラメーターは、接続がタイムアウトするまでコネクターがアイドル状態のままになる時間 (ミリ秒単位) を指定します。このパラメーターのデフォルト値は **-1** です (タイムアウト期間が設定されません)。これは、オプションパラメーターです。
- ***tcp-no-delay*** パラメーターは、TCP パケットが遅延され一括して送信されるかを指定します。このパラメーターの有効な値は **true** と **false** になります。このパラメーターのデフォルト値は、**true** です。これはオプションパラメーターです。
- ***send-buffer-size*** パラメーターは、Hot Rod コネクターの送信バッファーのサイズを指定します。このパラメーターのデフォルト値は TCP スタックバッファーのサイズです。これはオプションパラメーターです。
- ***receive-buffer-size*** パラメーターは、Hot Rod コネクターの受信バッファーのサイズを指定します。このパラメーターのデフォルト値は TCP スタックバッファーのサイズです。これはオプションパラメーターです。

Topology-State-Transfer 要素

topology-state-transfer 要素は、Hot Rod コネクターのトポロジー状態転送設定を指定します。この要素は **hotrod-connector** 要素内でのみ使用できます。

- ***lock-timeout*** パラメーターは、ロックを取得しようとする操作がタイムアウトする時間を指定します。このパラメーターのデフォルト値は **10** 秒です。これはオプションパラメーターです。

- ***replication-timeout*** パラメーターは、レプリケーション操作がタイムアウトする時間 (ミリ秒単位) を指定します。このパラメーターのデフォルト値は **10** 秒です。これはオプションパラメーターです。
- ***update-timeout*** パラメーターは、更新操作がタイムアウトする時間 (ミリ秒単位) を指定します。このパラメーターのデフォルト値は **30** 秒です。これはオプションパラメーターです。
- ***external-host*** パラメーターは、トポロジー情報にリストされたクライアントに Hot Rod サーバーが送信するホスト名を指定します。このパラメーターのデフォルト値は、ホストアドレスです。これはオプションパラメーターです。
- ***external-port*** パラメーターは、トポロジー情報にリストされたクライアントに Hot Rod サーバーが送信するポートを指定します。このパラメーターのデフォルト値は、設定されたポートです。これはオプションパラメーターです。
- ***lazy-retrieval*** パラメーターは、Hot Rod コネクターが取得操作をレイジーに実行するかどうかを指定します。このパラメーターのデフォルト値は **true** です。これはオプションパラメーターです。

バグを報告する

10.9. HOT ROD インターフェースセキュリティー

10.9.1. Hot Rod エンドポイントをパブリックインターフェースとして公開

JBoss Data Grid の Hot Rod サーバーはデフォルトで管理インターフェースとして動作します。操作をパブリックインターフェースに拡張するには、以下のように、**management** の **socket-binding** 要素の ***interface*** パラメーターの値を **public** に変更します。

```
<socket-binding name="hotrod" interface="public" port="11222" />
```

バグを報告する

第11章 REMOTECACHE インターフェース

11.1. REMOTECACHE インターフェースについて

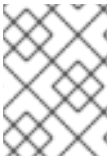
RemoteCache インターフェースは、JBoss Data Grid 外部のクライアントが JBoss Data Grid 内の Hot Rod サーバーモジュールにアクセスできるようにします。RemoteCache インターフェースは、ディストリビューションやエビクションなどのオプション機能を提供します。

バグを報告する

11.2. 新しい REMOTECACHEMANAGER の作成

次の設定を用いて、新しい **RemoteCacheManager** を宣言により設定します。

```
Properties props = new Properties();
props.put("infinispan.client.hotrod.server_list", "127.0.0.1:11222");
RemoteCacheManager manager = new RemoteCacheManager(props);
RemoteCache defaultCache = manager.getCache();
```



注記

JBoss Data Grid で **Hot Rod** を使用するための詳細については、『開発者ガイド』の Hot Rod の章を参照してください。

バグを報告する

付録A 改訂履歴

改訂 1.0.0-13 **Fri Jan 10 2014** **Misha Husnain Ali**

Built from Content Specification: 11622, Revision: 507873 by mhusnain