Red Hat Data Grid 7.3

Red Hat Data Grid User Guide

Red Hat Data Grid 7.3 向け

Last Updated: 2023-03-21

Red Hat Data Grid 7.3 Red Hat Data Grid User Guide

Red Hat Data Grid 7.3 向け

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Red_Hat_Data_Grid_User_Guide.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

本ガイドでは、Red Hat Data Grid 7.3 の管理および設定を説明します。

目次

第1章 はじめに

Red Hat Data Grid User Guide にようこそ。この包括的なドキュメントでは、Red Hat Data Grid の最終詳細をすべて説明します。

1.1. RED HAT DATA GRID とは

Red Hat Data Grid は、オプションのスキーマを備えた分散型インメモリーキー/値のデータストアです。埋め込み型 Java ライブラリーとして、およびさまざまなプロトコル（Hotgitops、REST、および Memcached）でリモートでアクセスされる言語に依存しないサービスの両方を使用できます。トランザクション、イベント、分散処理などの高度な機能や、JCache API 標準、CDI、Hibernate、WildFly、Spring Cache、Spring Session、Lucene、Spark、および Hadoop などのフレームワークとの多くのインテグレーションを提供します。

1.2. RED HAT DATA GRID を使用する理由

1.2.1. ローカルキャッシュとして

Red Hat Data Grid の主な使用方法は、頻繁にアクセスされるデータの高速インメモリーキャッシュを提供することです。低速なデータソース（データベース、Web サービス、テキストファイルなど）があるとします。コードからメモリーアクセスがなくなるように、メモリー内に一部またはすべてのデータを読み込むことができます。Red Hat Data Grid の使用は、有効期限やエビクションなどの便利な機能があるため、単純な ConcurrentHashMap を使用するよりも適しています。

1.2.2. クラスター化されたキャッシュ

データが 1 つのノードに収まらない場合や、アプリケーションの複数のインスタンスでエントリーを無効にする必要がある場合、Red Hat Data Grid は 100 未満のノードに水平スケーリングできます。

1.2.3. アプリケーションのクラスタリングビルディングブロック

アプリケーションのクラスター対応が必要な場合は、Red Hat Data Grid を統合して、トポロジー変更通知、クラスターの通信、クラスター実行などの機能にアクセスできます。

1.2.4. リモートキャッシュとして

アプリケーションとキャッシュ層を別個にスケーリングできるようにする場合や、異なる言語/プラットフォームを使用する場合でも、異なるアプリケーションでデータを利用できるようにする必要がある場合は、Red Hat Data Grid Server とそのさまざまなクライアントを使用します。

1.2.5. グリッドとして使用

Red Hat Data Grid に配置したデータは一時的なものである必要はありません。Red Hat Data Grid をプライマリーストアとして使用し、トランザクション、通知、クエリー、分散実行、分散ストリーム、解析などの強力な機能を使用してデータを迅速に処理できます。

1.2.6. データの地理的なバックアップ

Red Hat Data Grid はクラスター間のレプリケーションをサポートし、地理的なリモートサイト全体でデータをバックアップできます。

第2章 埋め込み CACHEMANAGER

CacheManager は Red Hat Data Grid の主なエントリーポイントです。CacheManager を使用して

- キャッシュの設定および取得
- ノードの管理およびモニタリング
- クラスター全体でコードを実行する
- その他

アプリケーションに Red Hat Data Grid を埋め込むか、リモートに使用しているかに応じて、**EmbeddedCacheManager** または **RemoteCacheManager** のいずれかを使用します。これらは一部のメソッドとプロパティーを共有しますが、セマンティックの違いがあることに注意してください。以下の章は、主に **埋め込み** の実装に焦点を当てています。**リモート** 実装の詳細は、「 Hoting Java Client 」を参照してください。

CacheManager は負荷の高いオブジェクトであり、JVM ごとに複数の CacheManager が使用されないようにします（特定のセットアップには複数の CacheManager が必要ない場合）。ただし、いずれの場合も、インスタンスの最小数および致命的になります。

CacheManager を作成する最も簡単な方法は、以下のとおりです。

```
EmbeddedCacheManager manager = new DefaultCacheManager();
```

最も基本的なローカルモードである、キャッシュがない非クラスターキャッシュマネージャーを起動します。CacheManagers にはライフサイクルがあり、デフォルトのコンストラクターは start（）とも呼ばれ ます。CacheManager を起動しないコンストラクターのオーバーロードバージョンを利用できます。ただし、CacheManager を使用して Cache インスタンスの作成を使用する前に、CacheManager を起動する必要があることに注意してください。

構築後、JNDI、ServletContext、または IoC コンテナーなどの他のメカニズムを介して、アプリケーション全体のスコープを介して CacheManager をコンポーネントで利用できるようにする必要があります。

CacheManager で作業が終了したら、リソースを解放することができるように、これを停止する必要があります。

```
manager.stop();
```

これにより、その範囲内内のすべてのキャッシュが適切に停止され、スレッドプールがシャットダウンされます。CacheManager がクラスター化されている場合は、クラスターも正常に残ります。

2.1. 設定

Red Hat Data Grid は、宣言的およびプログラムによる設定の両方を提供します。

2.1.1. キャッシュの宣言的設定

宣言型の設定は、提供される Red Hat Data Grid 設定 XML スキーマ に準拠する XML ドキュメントの形式で提供されます。

宣言的に設定できる Red Hat Data Grid のすべての側面は、プログラム的に設定することもできます。実際、宣言型の設定では、背後で XML 設定ファイルが処理される際にプログラムによる設定 API を呼

び出します。これらのアプローチの組み合わせを使用することもできます。たとえば、静的な XML 設定ファイルを読み取り、ランタイム時に同じ設定をプログラム的に調整できます。または、開始点またはテンプレートとして XML で定義された特定の静的設定を使用して、ランタイムで追加の設定を定義できます。

Red Hat Data Grid には、**global** と **cache** の 2 つの主な設定抽象化があります。

グローバル設定

グローバル設定は、単一の EmbeddedCacheManager が作成したすべてのキャッシュインスタンス間で共有されるグローバル設定を定義します。スレッドプール、シリアライズ/マーシャリングの設定、トランスポートとネットワーク設定などの共有リソースは、JMX ドメインはすべてグローバル設定の一部です。

キャッシュ設定

キャッシュ設定は実際のキャッシュドメイン自体に固有のものです。エビクション、ロック、トランザクション、クラスタリングなどを指定します。名前付きのキャッシュ設定を必要な数だけ指定できます。これらのキャッシュの 1 つは、**CacheManager.getCache（）API によって返されるキャッシュである デフォルト キャッシュと指定できます** が、他の名前付きキャッシュは **CacheManager.getCache(String name)** API 経由で取得されます。

指定されている場合、名前付きキャッシュは追加の動作を指定または上書きできる間にデフォルトキャッシュから設定を継承します。Red Hat Data Grid は、設定テンプレートの階層を定義し、複数のキャッシュが同じ設定を共有したり、必要に応じて特定のパラメーターを上書きしたりできる非常に柔軟な継承メカニズムも提供します。

注記

埋め込みおよびサーバー設定は異なるスキーマを使用しますが、これら 2 つのスキーマ間で簡単に移行できるように、できるだけ互換性を維持しています。

Red Hat Data Grid の主な目的は、ゼロ設定の目的です。開始するには、1 つ以上の infinispan 要素が含まれる単純な XML 設定ファイルがあれば十分です。以下の設定ファイルは適切なデフォルト値であり、完全に有効です。

infinispan.xml

```
<infinispan />
```

ただし、これは最も基本的なローカルモードで、キャッシュのない非クラスターキャッシュマネージャーのみを提供します。基本以外の設定は、カスタマイズされたグローバルおよびデフォルトのキャッシュ要素を使用する可能性が非常に高くなります。

宣言型の設定は、Red Hat Data Grid キャッシュインスタンスを設定する最も一般的な方法です。XML 設定ファイルを読み取るため、通常は Red Hat Data Grid 設定を含む XML ファイルを参照して DefaultCacheManager のインスタンスを構築します。設定ファイルが読み取られると、デフォルトのキャッシュインスタンスへの参照を取得できます。

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

my-config-file.xml で指定されたその他の名前付きインスタンス。

```
Cache someNamedCache = manager.getCache("someNamedCache");
```

デフォルトのキャッシュの名前は XML 設定ファイルの < **cache-container** > 要素に定義され、< **local-cache>、<distributed-cache >、または <invalidation-cache** > 要素を使用して追加の **キャッシュ** を設定できます。

以下の例は、Red Hat Data Grid でサポートされる各キャッシュタイプで最も簡単な設定を示しています。

```
<infinispan>
  <cache-container default-cache="local">
    <transport cluster="mycluster"/>
    <local-cache name="local"/>
    <invalidation-cache name="invalidation" mode="SYNC"/>
    <replicated-cache name="repl-sync" mode="SYNC"/>
    <distributed-cache name="dist-sync" mode="SYNC"/>
  </cache-container>
</infinispan>
```

2.1.1.1. キャッシュ設定テンプレート

上記のように、Red Hat Data Grid は **設定テンプレート** の概念をサポートします。これらは、複数のキャッシュ間で共有できる完全または部分的な設定宣言です。

以下の例は、**local** -template という名前のキャッシュを定義するのに **local-template** という設定を使用する方法を示しています。

```
<infinispan>
  <cache-container default-cache="local">
    <!-- template configurations -->
    <local-cache-configuration name="local-template">
      <expiration interval="10000" lifespan="10" max-idle="10"/>
    </local-cache-configuration>

    <!-- cache definitions -->
    <local-cache name="local" configuration="local-template" />
  </cache-container>
</infinispan>
```

テンプレートは、以前に定義したテンプレートから継承でき、一部の設定要素またはすべての設定要素をオーバーライドすることができます。

```
<infinispan>
  <cache-container default-cache="local">
    <!-- template configurations -->
    <local-cache-configuration name="base-template">
      <expiration interval="10000" lifespan="10" max-idle="10"/>
    </local-cache-configuration>

    <local-cache-configuration name="extended-template" configuration="base-template">
      <expiration lifespan="20"/>
      <memory>
        <object size="2000"/>
      </memory>
    </local-cache-configuration>

    <!-- cache definitions -->
```

```
    <local-cache name="local" configuration="base-template" />
    <local-cache name="local-bounded" configuration="extended-template" />
  </cache-container>
</infinispan>
```

上記の例では、**base-template** は特定の **有効期限** 設定でローカルキャッシュを定義します。**拡張テンプレート設定は base-template** から継承され、**expiration** 要素の単一のパラメーターのみを上書きします（他の属性はすべて継承されます）。また、**メモリー** 要素を追加します。最後に、2 つのキャッシュが定義され、**local** は **base-template** 設定を使用し、**local- template 設定を使用するローカルにバインド** されています。

警告

継承は多値要素（ **プロパティー**）では追加されています。つまり、子設定は親のプロパティーと独自のプロパティーをマージする結果になります。

2.1.1.2. キャッシュ設定ワイルドカード

テンプレートをキャッシュに適用する代替方法として、テンプレート名（例： **basecache***）でワイルドカードを使用することができます。テンプレートワイルドカードに名前が一致するキャッシュは、その設定を継承します。

```
<infinispan>
  <cache-container>
    <local-cache-configuration name="basecache*">
      <expiration interval="10500" lifespan="11" max-idle="11"/>
    </local-cache-configuration>
    <local-cache name="basecache-1"/>
    <local-cache name="basecache-2"/>
  </cache-container>
</infinispan>
```

上記の例では、**cache-1** および **basecache-2** をキャッシュすると、**basecache*** 設定が使用されます。また、未定義のキャッシュをプログラムで取得する際にも設定が適用されます。

注記

キャッシュ名が複数のワイルドカードと一致する場合（つまりあいまいである場合）、例外が発生します。

2.1.1.3. 宣言的設定リファレンス

宣言型設定スキーマの詳細は、「設定参照」を 参照 してください。設定の記述に XML 編集ツールを使用する場合は、提供された Red Hat Data Grid スキーマ を使用して役に立ちます。

2.1.2. プログラムによるキャッシュの設定

プログラムによる Red Hat Data Grid 設定は、CacheManager および ConfigurationBuilder API をベースとしています。Red Hat Data Grid 設定のすべての側面はプログラム的に設定できますが、最も一般

的なアプローチは、必要に応じて XML 設定ファイル形式で開始点を作成し、ランタイムで必要に応じて特定の設定をプログラム的に調整してユースケースに最も適宜調整することです。

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

新しい同期的に複製されたキャッシュをプログラム的に設定すると仮定してみましょう。まず、Configuration オブジェクトの新しいインスタンスは ConfigurationBuilder ヘルパーオブジェクトを使用して作成され、キャッシュモードは同期レプリケーションに設定されます。最後に、設定はマネージャーで定義/登録されます。

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

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

デフォルトのキャッシュ設定（またはその他のキャッシュ設定）は、新規キャッシュの作成の開始点として使用できます。たとえば、**infinispan-config-file.xml** がレプリケートされたキャッシュをデフォルトとして指定し、分散キャッシュが特定の L1 ライフスパンで必要である一方で、デフォルトキャッシュの他のすべての側面を保持する場合を考えてみましょう。したがって、開始点は、デフォルトの設定オブジェクトのインスタンスを読み取り、 **Configuration Builder** を使用して新しい設定オブジェクトでキャッシュモードと L1 ライフスパンを構築して変更します。最後のステップとして、設定はマネージャーで定義/登録されます。

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration dcc = manager.getDefaultCacheConfiguration();
Configuration c = new
ConfigurationBuilder().read(dcc).clustering().cacheMode(CacheMode.DIST_SYNC).l1().lifespan(6000
0L).build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

ベース設定がデフォルトの名前のキャッシュである限り、以前のコードは問題なく機能します。ただし、ベース設定が別の名前のキャッシュになる場合もあります。そのため、他の定義されたキャッシュに基づいて新しい設定を定義する方法。前述の例を考えて、デフォルトのキャッシュをベースとして使用する代わりに、「replicatedCache」という名前の名前付きキャッシュが base として使用されます。コードは以下のようになります。

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration rc = manager.getCacheConfiguration("replicatedCache");
Configuration c = new
ConfigurationBuilder().read(rc).clustering().cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000
L).build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

詳細は、CacheManager、ConfigurationBuilder、Configuration、および GlobalConfiguration javadocs を参照してください。

2.1.2.1. ConfigurationBuilder Programmatic Configuration API

上記の段落では、宣言的およびプログラムによる設定を組み合わせる方法を示していますが、XML 設定から始まります。ConfigurationBuilder Fluent インターフェーススタイルを使用すると、簡単に記述でき、より読みやすいプログラムによる設定を作成できます。この方法は、グローバルレベルとキャッシュレベル設定の両方に使用できます。GlobalConfiguration オブジェクトは GlobalConfigurationBuilder を使用して構築されますが、Configuration オブジェクトは ConfigurationBuilder を使用してビルドされます。この API でグローバルレベルとキャッシュレベルのオプションの両方を設定する例を見てみましょう。

2.1.2.2. トランスポートの設定

最も一般的なグローバルオプションの 1 つにトランスポート層があります。ここでは、Red Hat Data Grid ノードが他方を検出する方法を示します。

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("qa-cluster")
        .addProperty("configurationFile", "jgroups-tcp.xml")
        .machineId("qa-machine").rackId("qa-rack")
      .build();
```

2.1.2.3. カスタム JChannel の使用

JGroups JChannel を自分で構築したい場合は、それを行うことができます。

注記

JChannel はまだ接続されていない。

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
JChannel jchannel = new JChannel();
// Configure the jchannel to your needs.
JGroupsTransport transport = new JGroupsTransport(jchannel);
global.transport().transport(transport);
new DefaultCacheManager(global.build());
```

2.1.2.4. JMX MBean および統計の有効化

キャッシュマネージャーレベルで グローバル JMX 統計 のコレクションを有効にしたり、トランスポートに関する情報を取得したりする場合もあります。グローバル JMX 統計を有効にするには、以下を行います。

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

グローバル JMX 統計を有効にしないこと（または明示的に無効）しないようにして、統計の収集をオフにするだけです。対応する MBean は引き続き登録され、キャッシュマネージャーを一般的に管理するために使用できますが、統計属性は意味のある値を返しません。

クローバル JMX 統計レベルで追加オプションを使用すると、複数のキャッシュマネージャーが同じシステムで実行している場合や、JMX MBean サーバーを見つける方法に便利なキャッシュマネージャー名を設定できます。

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

2.1.2.5. スレッドプールの設定

Red Hat Data Grid 機能の一部は、スレッドプールエグゼキューターのグループによってサポートされており、このグローバルレベルで調整することもできます。以下に例を示します。

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
   .replicationQueueThreadPool()
     .threadPoolFactory(ScheduledThreadPoolExecutorFactory.create())
  .build();
```

グローバル、キャッシュマネージャーレベル、オプションだけでなく、クラスターモードなどのキャッシュレベルのオプションを設定することもできます。

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

または 、エビクションおよび有効期限の設定を設定 できます。

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

2.1.2.6. トランザクションとロックの設定

アプリケーションは JTA の境界内で Red Hat Data Grid キャッシュと対話し、トランザクションレイヤーを設定してオプションでロック設定を調整する必要がある場合もあります。トランザクションキャッシュと対話する場合は、リカバリーを有効にしてヒューリスティックな結果となったトランザクションを処理したい場合があります。その場合、JMX 管理と統計収集を有効にすることが多くあります。

```
Configuration config = new ConfigurationBuilder()
  .locking()
    .concurrencyLevel(10000).isolationLevel(IsolationLevel.REPEATABLE_READ)
    .lockAcquisitionTimeout(12000L).useLockStriping(false).writeSkewCheck(true)
```

```
    .versioning().enable().scheme(VersioningScheme.SIMPLE)
  .transaction()
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .recovery()
  .jmxStatistics()
  .build();
```

2.1.2.7. キャッシュストアの設定

チェーンキャッシュストアでの Red Hat Data Grid の設定もシンプルです。

```
Configuration config = new ConfigurationBuilder()
  .persistence().passivation(false)
  .addSingleFileStore().location("/tmp").async().enable()
  .preload(false).shared(false).threadPoolSize(20).build();
```

2.1.2.8. 高度なプログラムによる設定

Fluent（流れるような）設定は、高度な外部ャーなどのより高度なオプションまたは具体化オプションの設定にも使用できます。

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .serialization()
    .addAdvancedExternalizer(998, new PersonExternalizer())
    .addAdvancedExternalizer(999, new AddressExternalizer())
  .build();
```

または、カスタムインターセプターを追加します。

```
Configuration config = new ConfigurationBuilder()
  .customInterceptors().addInterceptor()
    .interceptor(new FirstInterceptor()).position(InterceptorConfiguration.Position.FIRST)
    .interceptor(new LastInterceptor()).position(InterceptorConfiguration.Position.LAST)
    .interceptor(new FixPositionInterceptor()).index(8)
    .interceptor(new AfterInterceptor()).after(NonTransactionalLockingInterceptor.class)
    .interceptor(new BeforeInterceptor()).before(CallInterceptor.class)
  .build();
```

個別の設定オプションの詳細は、『設定ガイド』 を参照してください。

2.1.3. 設定移行ツール

埋め込みスキーマをサーバーで使用されるものに合わせるために、Red Hat Data Grid の設定形式はスキーマバージョン 6.0 に変更されました。このため、スキーマ 7.x 以降にアップグレードする場合は、**すべて** のディストリビューションに含まれる設定コンバーターを使用する必要があります。単に、古い設定ファイルを最初のパラメーターとして渡し、変換されたファイルの名前を 2 番目のパラメーターとして渡すだけです。

Unix/Linux/macOS で変換するには、次のコマンドを実行します。

```
bin/config-converter.sh oldconfig.xml newconfig.xml
```

Windows の場合：

```
bin\config-converter.bat oldconfig.xml newconfig.xml
```

ヒント

他のキャッシュシステムから変換ツールを書き込む場合は infinispan-dev にお問い合わせください。

2.1.4. クラスター化の設定

Red Hat Data Grid は、クラスター化モードの場合に、ネットワーク通信に JGroups を使用します。Red Hat Data Grid には **事前設定され** た JGroups スタックが含まれており、クラスター化された設定を簡単に開始できます。

2.1.4.1. 外部 JGroups ファイルの使用

キャッシュをプログラムで設定する場合は、以下のことを行う必要があります。

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

また、XML ファイルを使用して Red Hat Data Grid を設定する場合は、以下を使用します。

```
<infinispan>
  <jgroups>
    <stack-file name="external-file" path="jgroups.xml"/>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <transport stack="external-file" />
    <replicated-cache name="replicatedCache"/>
  </cache-container>

  ...

</infinispan>
```

いずれの場合も、Red Hat Data Grid は最初にクラスパスで **jgroups.xml** を検索し、クラスパスにない場合の絶対パス名を検索します。

2.1.4.2. 事前設定された JGroups ファイルの 1 つを使用します。

Red Hat Data Grid には異なる JGroups ファイル（infinispan-core.jar にパッケージ）が含まれており、デフォルトではクラスパスにすでに存在します。上記の **jgroups.xml** の代わりにファイル名を指定するだけで結構です。たとえば、**/default-configs/default-jgroups-tcp.xml** を指定します。

利用可能な設定は以下のとおりです。

- default-jgroups-udp.xml - UDP をトランスポートとして使用し、UDP マルチキャストを検出に使用します。通常は、大規模な（100 ノードを超える）クラスター **や、レプリケーション また** は無効化を使用している場合に適して い ます。ソケットを過剰に開きます。
- default-jgroups-tcp.xml: TCP をトランスポートとして使用し、UDP マルチキャストを検出に使用します。TCP がポイントツーポイントプロトコルとしてより効率的であるため、ディストリビューション を使用して **いる場合** に限り、小規模なクラスター（100 ノード未満）に適し

ています。

- default-jgroups-ec2.xml: TCP をトランスポートとして使用し、S3_PING を 検出に使用します。UDP マルチキャストが利用できない Amazon EC2 ノードに適しています。
- default-jgroups-kubernetes.xml: TCP をトランスポートとして使用し、検出に KUBE_PING を使用します。UDP マルチキャストが常に利用可能ではない Kubernetes および OpenShift ノードに適しています。

2.1.4.2.1. JGroups 設定のチューニング

上記の設定は、XML ファイルを編集することなくさらに調整できます。起動時に特定のシステムプロパティーを JVM に渡すと、これらの設定の一部の動作に影響を及ぼす可能性があります。以下の表は、この方法で設定できる設定を示しています。例:

```
$ java -cp ... -Djgroups.tcp.port=1234 -Djgroups.tcp.address=10.11.12.13
```

表2.1 default-jgroups-udp.xml

システムプロパティー	説明	デフォルト	必須?
jgroups.udp.mcast_addr	マルチキャストに使用する IP アドレス（通信および検出の両方）IP マルチキャストに適した有効な Class D IP アドレスである必要があります。	228.6.7.8	▮
jgroups.udp.mcast_port	マルチキャストソケットに使用するポート	46655	▮
jgroups.udp.ip_ttl	IP マルチキャストパケットの Time-to-live(TTL)を指定します。ここでの値は、パケットが破棄される前に実行できるネットワークホップの数を指します。	2	▮

表2.2 default-jgroups-tcp.xml

システムプロパティー	説明	デフォルト	必須?
jgroups.tcp.address	TCP トランスポートに使用する IP アドレス。	127.0.0.1	▮
jgroups.tcp.port	TCP ソケットに使用するポート	7800	▮

jgroups.udp.mcast_addr	マルチキャストに使用する IP アドレス（検出用）。IP マルチキャストに適した有効な Class D IP アドレスである必要があります。	228.6.7.8	▮
jgroups.udp.mcast_port	マルチキャストソケットに使用するポート	46655	▮
jgroups.udp.ip_ttl	IP マルチキャストパケットの Time-to-live(TTL)を指定します。ここでの値は、パケットが破棄される前に実行できるネットワークホップの数を指します。	2	▮

表2.3 default-jgroups-ec2.xml

システムプロパティー	説明	デフォルト	必須?
jgroups.tcp.address	TCP トランスポートに使用する IP アドレス。	127.0.0.1	▮
jgroups.tcp.port	TCP ソケットに使用するポート	7800	▮
jgroups.s3.access_key	S3 バケットのアクセスに使用される Amazon S3 アクセスキー		▮
jgroups.s3.secret_access_key	S3 バケットのアクセスに使用される Amazon S3 シークレットキー		▮
jgroups.s3.bucket	使用する Amazon S3 バケットの名前。固有で、存在している必要があります		▮

表2.4 default-jgroups-kubernetes.xml

システムプロパティー	説明	デフォルト	必須?
jgroups.tcp.address	TCP トランスポートに使用する IP アドレス。	eth0	▮

jgroups.tcp.port	TCP ソケットに使用するポート	7800	▮

2.1.4.3. 関連資料

JGroups は、より多くのシステムプロパティーのオーバーライドをサポートします。詳細は、SystemPropsを参照してください。

さらに、Red Hat Data Grid に同梱された JGroups 設定ファイルも、何かを稼働させるためのジャンプポイントとして設計されています。しかし、あまりよく対応しない場合は、JGroups スタックをさらに微調整して、ネットワーク機器からすべてのパフォーマンスを抽出します。そのため、次の停止は JGroups マニュアルになります。JGroups 設定ファイルに表示する各プロトコルの設定に 関する詳しいセクション があります。

2.2. キャッシュの取得

CacheManagerを設定した後、キャッシュを取得および制御できます。

getCache（） メソッドを呼び出して、以下のようにキャッシュを取得します。

```
Cache<String, String> myCache = manager.getCache("myCache");
```

上記の操作は、**myCache**という名前のキャッシュがまだ存在しない場合は作成し、それを返します。

getCache()メソッドを使用すると、メソッドを呼び出すノードにのみキャッシュが作成されます。つまり、クラスター全体の各ノードで呼び出す必要のあるローカル操作を実行します。通常、複数のノードにまたがってデプロイされたアプリケーションは、初期化中にキャッシュを取得して、キャッシュが**対称**であり、各ノードに存在することを確認します。

createCache（） メソッドを呼び出して、以下のようにクラスター全体でキャッシュを動的に作成します。

```
Cache<String, String> myCache = manager.administration().createCache("myCache",
"myTemplate");
```

上記の操作では、後でクラスターに参加するすべてのノードにキャッシュが自動的に作成されます。

createCache()メソッドを使用して作成するキャッシュは、デフォルトでは一時的です。クラスタ全体がシャットダウンした場合、再起動時にキャッシュが自動的に再作成されることはありません。

PERMANENT フラグを使用して、以下のようにキャッシュが再起動後も存続できるようにします。

```
Cache<String, String> myCache =
manager.administration().withFlags(AdminFlag.PERMANENT).createCache("myCache",
"myTemplate");
```

PERMANENT フラグを有効にするには、グローバルの状態を有効にし、設定ストレージプロバイダーを設定する必要があります。

設定ストレージプロバイダーの詳細は、GlobalStateConfigurationBuilder#configurationStorage（）を参照してください。

2.3. クラスタリング情報

EmbeddedCacheManagerには、クラスターの動作に関する情報を提供するためのメソッドが多数あります。以下のメソッドは、クラスター環境で使用される場合（Transport が設定されている場合）にのみ意味があります。

2.3.1. メンバー情報

クラスターを使用している場合、クラスターの所有者が誰であるかなど、クラスターのメンバーシップに関する情報を見つけられることが非常に重要となります。

getMembers()

getMembers()メソッドは、現在のクラスター内のすべてのノードを返します。

getCoordinator()

getCoordinator()メソッドは、どのメンバーがクラスターのコーディネーターであるかを指示します。ほとんどの場合、コーディネーターが誰であるかを気にする必要はありません。isCoordinator（） メソッドを直接使用して、ローカルノードもコーディネーターであるかどうかを確認できます。

2.3.2. その他の方法

getTransport()

このメソッドは、他のノードにメッセージを送信するために使用される基礎となるトランスポートへのアクセスを提供します。ほとんどの場合、ユーザーはこのレベルに移動する必要はありませんが、Transport 固有の情報（この場合は JGroups）を取得する場合は、このメカニズムを使用できます。

getStats()

ここで提供される統計は、このマネージャーのすべてのアクティブなキャッシュから結合されます。これらの統計は、クラスター全体で何らかの問題が発生しているかを確認するのに役に立つ場合があります。

第3章 キャッシュ API

3.1. キャッシュインターフェース

Red Hat Data Grid のキャッシュは、キャッシュインターフェースを介して操作されます。https://access.redhat.com/webassets/avalon/d/red-hat-data-grid/7.3/api/org/infinispan/Cache.html

キャッシュは、JDK の ConcurrentMap インターフェースに公開されるアトミックメカニズムなど、エントリーを追加、取得、および削除するための簡単な方法を公開します。使用されるキャッシュモードに基づいて、これらのメソッドを呼び出すと、リモートノードにエントリを複製したり、リモートノードからエントリを検索することやキャッシュストアからエントリを検索することなど、数多くのことが発生します。

注記

単純な使用方法では、Cache API を使用するのと JDK Map API を使用することはできないため、Map から Red Hat Data Grid の Cache への移行は簡単なはずです。

3.1.1. 特定のマップメソッドのパフォーマンスに関する懸念

Map で公開されるメソッドによっては、size( )、values （）、keySet（）、および entrySet （）などの Red Hat Data Grid で使用すると、特定のパフォーマンス結果が得 られます。**keySet**の特定のメソッドである**values**および**entrySet**を使用できます。詳細については、Javadocを参照してください。

これらの操作をグローバルに実行しようとすると、パフォーマンスに大きな影響を及ぼし、スケーラビリティーのボトルネックにもなります。そのため、これらの方法は情報またはデバッグの目的でのみ使用してください。

withFlags メソッドで特定のフラグを使用すると、このような懸念を軽減することができることに注意してください。詳細は、各メソッドのドキュメントを確認してください。

3.1.2. Mortal および Immortal データ

単にエントリーを格納するだけで、Red Hat Data Grid のキャッシュ API では、可変情報をデータに割り当てることができます。たとえば、単にput(key, value)を使用すると、**immortal** エントリーが作成されます。このエントリーは削除されるまで（またはメモリー不足にならないようにメモリーからエビクトされるまで）、いつまでもキャッシュに存在します。ただし、place (key, value, lifespan, timeunit)を使用してキャッシュにデータを配置すると、 **mortal** エントリーが作成されます。つまり、有効期限が固定されたエントリーとなり、そのライフスパンの後に有効期限が切れます。

Red Hat Data Grid は、有効期限を決定するための追加のメトリクスとして **maxIdle** もサポートしています。*lifespans または maxIdles の任意の組み合わせを使用できます。*

3.1.3. 有効期限および変動データ

Red Hat Data Grid でmortal データを使用する方法は、「Expiration 」を参照してください。

3.1.4. putForExternalRead 操作

Red Hat Data Grid の Cache クラスには、putForExternalRead と呼ばれる別の 'put' 操作が含まれています。この操作は、Red Hat Data Grid が別の場所で永続化されるデータの一時的なキャッシュとして使用する場合に特に便利です。読み取りが非常に多い場合、キャッシュは単に最適化のために行われ、妨

害するものではないため、キャッシュの競合によって実際のトランザクションが遅延してはなりません。

これには、PlaceForExternalRead はキーがキャッシュに存在しない場合にのみ動作する put 呼び出しとして機能し、別のスレッドが同時に同じ鍵を保存するかどうかを警告なしで失敗します。このようなシナリオでは、データのキャッシュはシステムを最適化する方法であり、キャッシュ内の障害が進行中のトランザクションに影響を及ぼすことが望ましいと言えます。そのため、失敗は異なる理由が処理されます。PlasForExternalRead は成功するかどうかに関わらず、ロックを待たず、呼び出し元のプロンプトに戻るため、高速操作と見なされます。

この操作の使用方法を理解するために、基本的な例を見てみましょう。PersonIdによって入力されるPersonインスタンスのキャッシュを想像してください。このデータは個別のデータストアから入力されます。以下のコードは、この例のコンテキスト内で putForExternalRead を使用する最も一般的なパターンを示しています。

```
// Id of the person to look up, provided by the application
PersonId id = ...;

// Get a reference to the cache where person instances will be stored
Cache<PersonId, Person> cache = ...;

// First, check whether the cache contains the person instance
// associated with with the given id
Person cachedPerson = cache.get(id);

if (cachedPerson == null) {
  // The person is not cached yet, so query the data store with the id
  Person person = dataStore.lookup(id);

  // Cache the person along with the id so that future requests can
  // retrieve it from memory rather than going to the data store
  cache.putForExternalRead(id, person);
} else {
  // The person was found in the cache, so return it to the application
  return cachedPerson;
}
```

putForExternalRead は、アプリケーション実行から発信された新しい Person インスタンスでキャッシュを更新するメカニズムとして使用しないでください（Person のアドレスを変更するトランザクションから）。キャッシュされた値を更新する場合は、標準のput操作を使用してください。使用しないと、破損したデータをキャッシュする可能性が高くなります。

3.2. ADVANCEDCACHE インターフェース

Red Hat Data Grid は、単純なキャッシュインターフェースのほかに、拡張作成者向けに調整された AdvancedCache インターフェースを提供します。AdvancedCache は、カスタムインターセプターを挿入したり、特定の内部コンポーネントにアクセスし、フラグを適用して特定のキャッシュメソッドのデフォルト動作を変更する機能を提供します。次のコードスニペットは、AdvancedCacheを取得する方法を示しています。

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

3.2.1. Flags

フラグは通常のキャッシュメソッドに適用され、特定のメソッドの動作を変更します。利用可能なフラグの一覧とその効果は、「フラグの列挙」を参照 し てください。フラグは、AdvancedCache.withFlags（） を使用して適用されます。このビルダーメソッドを使用して、キャッシュ呼び出しに任意の数のフラグを適用できます。次に例を示します。

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

3.2.2. カスタムインターセプター

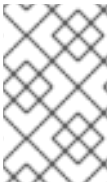
AdvancedCache インターフェースは、高度な開発者はカスタムインターセプターをアタッチするメカニズムも提供します。カスタムインターセプターを使用すると、開発者はキャッシュ API メソッドの動作を変更でき、AdvancedCache インターフェースを使用すると、開発者は実行時にこれらのインターセプターをプログラムを使用してアタッチすることができます。詳細は、AdvancedCache Javadocs を参照してください。

カスタムインターセプターの作成に関する詳細は、「カスタムインターセプター」を参照 し てください。

3.3. リスナーおよび通知

Red Hat Data Grid は、イベントの発生時にクライアントが登録し、通知できるリスナー API を提供します。このアノテーション駆動型APIは、キャッシュレベルイベントとキャッシュマネージャーレベルイベントの2つの異なるレベルに適用されます。

イベントは、リスナーにディスパッチされる通知をトリガーします。リスナーは、@Listener アノテーションが付けられた単純な POJO であり、Listenable インターフェースで定義されたメソッドを使用して登録されます。



注記

CacheとCacheManagerはどちらもListenableを実装しています。つまり、リスナーをキャッシュまたはキャッシュマネージャーのいずれかにアタッチして、キャッシュレベルまたはキャッシュマネージャーレベルのいずれかの通知を受信できます。

たとえば、以下のクラスは、新しいエントリーがキャッシュに追加されるたびに情報を出力するリスナーを定義します。

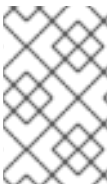
```
@Listener
public class PrintWhenAdded {

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

}
```

より包括的な例は、@Listener の Java ドキュメント を参照 してください。

3.3.1. キャッシュレベルの通知

キャッシュレベルのイベントはキャッシュごとに発生し、デフォルトでは、イベントが発生したノードでのみ発生します。分散キャッシュでは、これらのイベントは影響を受けるデータの所有者に対してのみ発生することに注意してください。キャッシュレベルのイベントの例としては、エントリの追加、削除、変更などがあります。これらのイベントは、特定のキャッシュに登録されているリスナーへの通知をトリガーします。

すべての キャッシュレベルの通知の包括的な一覧とメソッドレベルのアノテーションについては、org.infinispan.notifications.cachelistener.annotation パッケージの Javadocs を参照してください。



注記

Red Hat Data Grid で利用可能な キャッシュレベルの通知の一覧は、org.infinispan.notifications.cachelistener.annotation パッケージの Javadocs を参照してください。

3.3.1.1. クラスターリスナー

単一ノードでキャッシュイベントをリッスンすることが望ましい場合は、クラスターリスナーを使用する必要があります。

そのために必要なのは、リスナーがクラスター化されているというアノテーションを付けるよう設定することだけです。

```
@Listener (clustered = true)
public class MyClusterListener { .... }
```

クラスター化されていないリスナーからのクラスターリスナーには、いくつかの制限があります。

1. クラスターリスナーは、**@CacheEntryModified**、**@CacheEntryCreated**、**@CacheEntryRemoved**、および **@CacheEntryExpired** イベントのみをリッスンできます。これは、他のタイプのイベントは、このリスナーに対してリッスンされないことを意味することに注意してください。

2. ポストイベントのみがクラスターリスナーに送信され、プレイベントは無視されます。

3.3.1.2. イベントのフィルタリングおよび変換

リスナーがインストールされているノードで適用可能なすべてのイベントがリスナーに発生します。KeyFilter （キーのフィルタリングのみ）または CacheEventFilter （キーのフィルタリングのみを許可する）を使用して、発生したイベントを動的にフィルターすることができます（キー、古いメタデータ、新しい値、新しいメタデータのフィルターに使用され、イベントがイベントの前にある場合 (ie. isPre)、コマンドタイプなど、新しいメタデータのフィルターに使用されます。

この例で、イベントがキー**Only Me**のエントリーを変更したときにイベントのみを発生させる単純な**KeyFilter**を示しています。

```
public class SpecificKeyFilter implements KeyFilter<String> {
   private final String keyToAccept;

   public SpecificKeyFilter(String keyToAccept) {
     if (keyToAccept == null) {
       throw new NullPointerException();
     }
     this.keyToAccept = keyToAccept;
   }
```

```
    boolean accept(String key) {
      return keyToAccept.equals(key);
    }
}

...
cache.addListener(listener, new SpecificKeyFilter("Only Me"));
...
```

これは、より効率的な方法で受信するイベントを制限したい場合に便利です。

また、イベントを発生させする前に別の値に値を変換できる CacheEventConverter を指定することもできます。これは、値の変換を行うコードをモジュール化するのに適しています。

注記

上記のフィルターとコンバーターは、クラスターリスナーと組み合わせて使用すると特に効果的です。これは、イベントがリッスンされているノードではなく、イベントが発生したノードでフィルタリングと変換が行われるためです。これにより、クラスター全体でイベントを複製する必要がない（フィルター）、またはペイロードを減らす（コンバーター）という利点があります。

3.3.1.3. 初期状態のイベント

リスナーがインストールされると、完全にインストールされた後にのみイベントが通知されます。

リスナーの初回登録時にキャッシュコンテンツの現在の状態を取得することが望ましい場合があります。この場合、キャッシュ内の各要素の **@CacheEntryCreated** タイプのイベントが生成されます。この最初のフェーズで追加で生成されたイベントは、適切なイベントが発生するまでキューに置かれます。

注記

現時点では、これはクラスター化されたリスナーに対してのみ機能します。ISPN-4608 では、クラスター化されていないリスナーへの追加を説明しています。

3.3.1.4. 重複イベント

トランザクションではないキャッシュで、重複したイベントを受け取ることが可能です。これは、put などの書き込み操作の実行中に、キーのプライマリー所有者がダウンした場合に可能になります。

Red Hat Data Grid は、内部で put 操作を指定のキーの新しいプライマリー所有者に送信することで認識します。ただし、書き込みが最初にバックアップに複製されたかどうかについては保証はありません。そのため、**CacheEntryCreatedEvent**、**CacheEntryModifiedEvent**、および**CacheEntryRemovedEvent**の書き込みイベントの1 つ以上が、1 つの操作で送信される可能性があります。

複数のイベントが生成されると、Red Hat Data Grid はretried コマンドで生成されたイベントを示し、変更の表示に注意せずにこの状況を把握できるようにします。

```
@Listener
public class MyRetryListener {
  @CacheEntryModified
  public void entryModified(CacheEntryModifiedEvent event) {
```

```
    if (event.isCommandRetried()) {
      // Do something
    }
  }
}
```

また、**CacheEventFilter** または **CacheEventConverter** を使用する場合は、EventType にメソッド **isRetry** が含まれ、再試行のためにイベントが生成されたかどうかを指示します。

3.3.2. キャッシュマネージャーレベルの通知

キャッシュマネージャーレベルのイベントは、キャッシュマネージャーで行われます。これらはグローバルでクラスター全体でもありますが、単一のキャッシュマネージャーによって作成されたすべてのキャッシュに影響するイベントが関係します。キャッシュマネージャーレベルのイベントの例として、クラスターに参加または退出するノード、または開始または停止するキャッシュがあります。

すべての キャッシュマネージャーレベルの通知の包括的な一覧とメソッドレベルのアノテーションについては、org.infinispan.notifications.cachemanagerlistener.annotation パッケージの Javadocs を参照してください。

3.3.3. イベントの同期

デフォルトでは、すべての通知はイベントを生成する同じスレッドでディスパッチされます。つまり、スレッドの進捗を妨げるため、リスナーを書き込んでいたり、時間がかかりすぎたりしないようにする **必要** があります。または、リスナーに **非同期** のアノテーションを付けることもできます。この場合、個別のスレッドプールを使用して通知をディスパッチし、イベント送信元スレッドをブロックしないようにします。これには、以下のようにリスナーにアノテーションを付けます。

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

3.3.3.1. 非同期スレッドプール

このような **非同期通知のディスパッチに使用されるスレッドプールを調整するには、設定ファイルの <listener-executor /> XML 要素を使用します。**

3.4. ASYNCHRONOUS API

Cache. put（）、Cache. remove （）などの同期 API メソッドに加えて、Red Hat Data Grid には非同期でノンブロッキングの API があり、これを使用するとブロック以外の方法で同じ結果が得られます。

これらのメソッドの名前は、ブロックメソッドと同様の名前が付けられ、"Async"が追加されます。（例： Cache.putAsync（）、Cache.removeAsync（） など）。 これらの非同期の対応は、操作の実際の結果を含む Future を返します。

たとえば、cache< **String, String> , Cache.put(String key, String value)** として化されたキャッシュパラメーターでは、**String** を返します。**cache.putAsync(String key, String value)** は **Future<String> を返します。**

3.4.1. このような API を使用する理由

ノンブロッキングAPIは、通信の失敗や例外を処理する機能を備えており、同期通信の保証をすべて提供するという点で強力なもので、呼び出しが完了するまでブロックする必要がありません。 これにより、システムで並列処理をより有効に活用できます。 以下に例を示します。

```
Set<Future<?>> futures = new HashSet<Future<?>>();
futures.add(cache.putAsync(key1, value1)); // does not block
futures.add(cache.putAsync(key2, value2)); // does not block
futures.add(cache.putAsync(key3, value3)); // does not block

// the remote calls for the 3 puts will effectively be executed
// in parallel, particularly useful if running in distributed mode
// and the 3 keys would typically be pushed to 3 different nodes
// in the cluster

// check that the puts completed successfully
for (Future<?> f: futures) f.get();
```

3.4.2. 実際に非同期で発生するプロセス

Red Hat Data Grid には、通常の書き込み操作の重要なパスに考慮できる 4 つの点があります。これらの項目をコスト順に示します。

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

Red Hat Data Grid 4.0 の時点では、async メソッドを使用すると、ネットワークの呼び出しを取得し、重要なパスをマーシャリングします。 ただし、さまざまな技術的な理由により、キャッシュストアへの書き込みとロックの取得は、呼び出し元のスレッドで引き続き発生します。 今後は、これらをオフラインにする予定です。 このトピックについては、この開発者メールリストスレッド を参照してください。

3.4.3. future の通知

厳密には、これらのメソッドは JDK Futures を返さず、NotifyingFuture として知られるサブインターフェースです。 主な違いは、今後の完了時に通知できるように、リスナーを NotifyingFuture にアタッチできることです。 以下は、通知の future を使用する例です。

```
FutureListener futureListener = new FutureListener() {

  public void futureDone(Future future) {
    try {
      future.get();
    } catch (Exception e) {
      // Future did not complete successfully
      System.out.println("Help!");
    }
  }
};

cache.putAsync("key", "value").attachListener(futureListener);
```

3.4.4. 関連資料

Cache インターフェースの Javadocs には非同期 API を使用する例がいくつかあります。これは、Mande Surtani の API の導入によって この記事 を行います。

3.5. 呼び出しフラグ

Red Hat Data Grid の大半を取得する重要なことは、特定のキャッシュ呼び出しに特定の動作を提供するために、invocation フラグを使用する方法です。これを行うことで、重要な最適化が実装され、非常に多くの時間とネットワークリソースを節約できます。フラグの最も一般的な使用方法の 1 つは、Cache API で右側にあります。これは、外部リソースから読み取られたデータで Red Hat Data Grid キャッシュをロードするのに使用される putForExternalRead（） メソッド下にあります。この呼び出しを効率的に実行できるように、Red Hat Data Grid は基本的に FAIL_SILENTLY、FORCE_ASYNCHRONOUS、ZERO_LOCK_ACQUISITION_TIMEOUTのフラグを渡す通常の put 操作を呼び出します。

ここで行う Red Hat Data Grid とは、外部読み取りからデータを読み込んだときに、ほぼゼロのロック通信時間が使用され、ロックを取得できない場合には、ロックアップに関連する例外をスローせずに警告なしで失敗します。また、キャッシュモードに関係なく、キャッシュがクラスター化されている場合に、非同期的に複製されるため、他のノードからの応答は待機しないように指定します。これらのフラグの組み合わせにより、この種の操作は非常に効率的で、効率は、このタイプの **putForExternalRead** 呼び出しを、クライアントがメモリーに保存される必要のあるデータを取得するために常に永続的なストアに戻るナレッジで使用されます。そのため、データの格納を試みることはベストエフォートで、ことができない場合は、キャッシュミスがある場合はクライアントが再試行する必要があります。

3.5.1. 例

「フラグの 列挙 」で詳細に説明しているように、利用可能な他のフラグまたはその他のフラグを使用する場合は、高度なキャッシュを取得し、withFlags（） メソッド呼び出しで必要なフラグを追加するだけです。以下に例を示します。

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

これらのフラグはキャッシュ操作の期間のみアクティブであることに留意してください。同じトランザクションであっても、同じ呼び出しで同じフラグを使用する必要がある場合は、Flags（）を 繰り返し呼び出す必要があります。キャッシュ操作が別のノードでレプリケートされる場合、フラグもリモートノードに引き継がれます。

3.5.1.1. put（）または remove（）からの戻り値の抑制

もう 1 つの重要なユースケースは、put（）などの書き込み操作を以前の値を返さ**ない** 場合です。これを実行するには、分散環境で 2 つのフラグを使用する必要があります。これは以前の値を取得する可能性があるため、キャッシュストアから以前の値を読み込まないようにキャッシュローダーでキャッシュが設定されている場合にリモートルックアップは行われません。以下の例のように、これらの 2 つのフラグが action に表示されます。

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

詳細は、「フラグの 列挙 Java」 を 参照してください。

3.6. ツリー API モジュール

Red Hat Data Grid のツリー API モジュールは、API などのツリー構造を使用してデータを保存することができます。この API は JBoss Cache が提供する API と似て います。そのため、移行の一環としてコードベースを変更したいユーザーは、ツリーモジュールは JBoss Cache から Red Hat Data Grid に移行するのに便利です。さらに、Red Hat Data Grid はこのツリー API を JBoss Cache よりも効率的に提供する点を理解することが重要です。そのため、JBoss Cache にツリー API のユーザーであれば Red Hat Data Grid への移行を検討する必要があります。

3.6.1. ツリー API とは

この API の目的は、情報を階層的に保存することです。階層は、Fqn または完全修飾名 で表されるパス（例： **/this/is/a/fqn/path** または **/another/path** ）を使用して定義されます。階層には、すべてのパスの開始点を表す root と呼ばれる特別なパスがあり、これは **/** として表されます。

各 FQN パスは、キー/値のペアスタイル API（マップなど）を使用してデータを保存できるノードとして表されます。たとえば、**/persons/john** では、surname=Smith、birthdate=05/02/1980...etc など、John に属する情報を保存できます。

ユーザーは、データを格納する場所として root を使用するべきではないことに注意してください。代わりに、ユーザーは独自のパスを定義し、そこでデータを保存する必要があります。以下のセクションでは、この API の実用的な側面について説明します。

3.6.2. ツリー API の使用

3.6.2.1. 依存関係

アプリケーションがツリー API を使用するには、Red Hat Data Grid バイナリーディストリビューションにある infinispan-tree.jar をインポートするか、pom.xml のこのモジュールに依存関係を追加するだけです。

pom.xml

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

${version.infinispan} を Red Hat Data Grid の適切なバージョンに置き換えます。

3.6.3. ツリーキャッシュの作成

ツリー API を使用する最初の手順は、実際にツリーキャッシュを作成することです。これを実行するには、通常実行するように Red Hat Data Grid Cache を作成し、TreeCacheFactory を使用して TreeCache のインスタンスを作成します。ここでは、ファクトリーに渡される Cache インスタンスは呼び出しバッチで設定する必要があります。以下に例を示します。

```
import org.infinispan.config.Configuration;
import org.infinispan.tree.TreeCacheFactory;
```

```
import org.infinispan.tree.TreeCache;
...
Configuration config = new Configuration();
config.setInvocationBatchingEnabled(true);
Cache cache = new DefaultCacheManager(config).getCache();
TreeCache treeCache = TreeCacheFactory.createTreeCache(cache);
```

3.6.4. ツリーキャッシュでのデータを操作する

ツリー API はデータと対話するための 2 つの方法を提供します。

TreeCache 利便性メソッド - これらのメソッドは TreeCache インターフェース内にあり、ユーザーは Fqn 形式、String 形式または Fqn 形式で、1 回の呼び出しを使用して、取得、移動、削除、呼び出しに関与するデータを 保存、取得できるようにします。以下に例を示します。

```
treeCache.put("/persons/john", "surname", "Smith");
```

または、以下を実行します。

```
import org.infinispan.tree.Fqn;
...
Fqn johnFqn = Fqn.fromString("persons/john");
Calendar calendar = Calendar.getInstance();
calendar.set(1980, 5, 2);
treeCache.put(johnFqn, "birthdate", calendar.getTime()));
```

ノード API 経由：FQN を構成する個々のノードを細かく制御し、特定のノードに関連するノードの操作を可能にします。以下に例を示します。

```
import org.infinispan.tree.Node;
...
TreeCache treeCache = ...
Fqn johnFqn = Fqn.fromElements("persons", "john");
Node<String, Object> john = treeCache.getRoot().addChild(johnFqn);
john.put("surname", "Smith");
```

または、以下を実行します。

```
Node persons = treeCache.getRoot().addChild(Fqn.fromString("persons"));
Node<String, Object> john = persons.addChild(Fqn.fromString("john"));
john.put("surname", "Smith");
```

または、以下のようになります。

```
Fqn personsFqn = Fqn.fromString("persons");
Fqn johnFqn = Fqn.fromRelative(personsFqn, Fqn.fromString("john"));
Node<String, Object> john = treeCache.getRoot().addChild(johnFqn);
john.put("surname", "Smith");
```

ノードは、親 または 子 にアクセスする機能も提供します。以下に例を示します。

```
Node<String, Object> john = ...
Node persons = john.getParent();
```

または、以下を実行します。

```
Set<Node<String, Object>> personsChildren = persons.getChildren();
```

3.6.5. 一般的な操作

前のセクションでは、追加や取得など、最も使用されている操作の一部が表示されています。ただし、remove など、注目すべき重要な操作もあります。

たとえば、以下を使用して、ノード全体（**/persons/john** など）を削除できます。

```
treeCache.removeNode("/persons/john");
```

または、以下を実行して子ノードを削除します。

```
treeCache.getRoot().removeChild(Fqn.fromString("persons"));
```

ノードの特定のキーと値のペアを削除することもできます。

```
Node john = treeCache.getRoot().getChild(Fqn.fromElements("persons", "john"));
john.remove("surname");
```

または、以下を使用してノードのすべてのデータを削除できます。

```
Node john = treeCache.getRoot().getChild(Fqn.fromElements("persons", "john"));
john.clearData();
```

ツリー API でサポートされるもう 1 つの重要な操作は、ツリー内のノードを移動できることです。ルートノードの下にあるノード「john」があるとします。以下の例では、「john」ノードを「persons」ノード下に移動する方法を説明します。

現在のツリー構造：

```
/persons
/john
```

ツリーをある FQN から別の FQN に移動します。

```
Node john = treeCache.getRoot().addChild(Fqn.fromString("john"));
Node persons = treeCache.getRoot().getChild(Fqn.fromString("persons"));
treeCache.move(john.getFqn(), persons.getFqn());
```

最終的なツリー構造：

```
/persons/john
```

3.6.6. ツリー API のロック

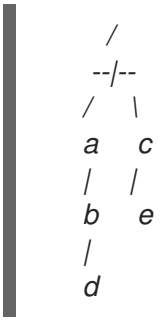
ツリー構造を操作するときにロックを取得するタイミングと方法を理解することは、ツリーに対して対話するクライアントアプリケーションのパフォーマンスを最大化し、同時に一貫性を維持する上で重要です。

ツリーAPI でのロックはノードごとに行われます。したがって、特定のノードでキー/値を置くか、または更新すると、そのノードの書き込みロックが取得されます。この場合、変更されるノードの親ノードへの書き込みロックは取得されず、子ノードのロックは取得されません。

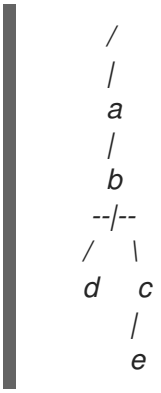
ノードを追加または削除しても、親は書き込みにロックされません。JBoss Cache では、この動作はデフォルトでは挿入や削除に対してロックされていなかった状態で設定されました。

最後に、ノードが移動されると、移動したノードとその子のいずれかがロックされますが、ターゲットノードとその子の新しい場所もロックされます。この点をよりよく理解するには、例を見てみましょう。

このような階層があり、c/ を下方向の b/ に移動させたいとします。



最終的な結果は、以下のようになります。



これを移動するには、ロックが取得されました。

- **/a/b** - データが置かれる親の下にあるためです。
- **/c** および **/c/e** - 移動中のノードであるため
- **/a/b/c** および **/a/b/c/e**: 移動中のノードの新しいターゲットの場所であるためです。

3.6.7. ツリーキャッシュイベントのリスナー

現在の Red Hat Data Grid リスナーは、キー/値のストア通知を念頭に置いて設計されているため、ツリーキャッシュイベントに正しくマッピングされません。ツリーキャッシュイベントに直接マップするツリーキャッシュ固有のリスナー（つまり子... の追加）は適切ですが、これらはまだ利用できません。このタイプのリスナーに興味がある場合は、この問題に従って、この エリアでの進捗を確認してください。

3.7. エンコーディング

3.7.1. 概要

encoding（エンコーディング）は、Red Hat Data Grid キャッシュが実行するデータ変換操作で、データの保存やストレージからの読み取り時に行われるデータ変換操作です。

これにより、API 呼び出し中（マップ、リスナー、ストリームなど）時に特定のデータ形式を扱うことができますが、実際に保存される形式は異なります。

*データ変換は、**org.infinispan.commons.dataconversion.Encoder** のインスタンスによって処理されます。*

```
public interface Encoder {

   /**
    * Convert data in the read/write format to the storage format.
    *
    * @param content data to be converted, never null.
    * @return Object in the storage format.
    */
   Object toStorage(Object content);

   /**
    * Convert from storage format to the read/write format.
    *
    * @param content data as stored in the cache, never null.
    * @return data in the read/write format
    */
   Object fromStorage(Object content);

   /**
     * Returns the {@link MediaType} produced by this encoder or null if the storage format is not known.
     */
   MediaType getStorageFormat();
}
```

3.7.2. デフォルトのエンコーダー

Red Hat Data Grid は、キャッシュ設定に応じて、Encoder を自動的に選択します。以下の表は、複数の設定に使用される内部エンコーダーを示しています。

Mode	設定	エンコーダー	説明
組み込み/サーバー	デフォルト	IdentityEncoder	パススルーエンコーダ、変換なし
組み込み	StorageType.OFF_HEAP	GlobalMarshallerEncoder	Red Hat Data Grid 内部マーシャラーを使用して byte[] に変換します。キャッシュマネージャーで設定されたマーシャラーに委譲できます。

Mode	設定	エンコーダー	説明
組み込み	StorageType.BINARY	BinaryEncoder	プリミティブおよび文字列を除き、Red Hat Data Grid 内部マーシャラーを使用して byte[] に変換します。
サーバー	StorageType.OFF_HEAP	IdentityEncoder	リモートクライアントが受信したように byte[] を直接保存

3.7.3. プログラムでの上書き

AdvancedCache から **withEncoding()** メソッドバリアントを呼び出すことで、キーと値の両方に使用されるエンコーディングをプログラムで上書きできます。

たとえば、OFF_HEAP として設定された以下のキャッシュについて考えてみましょう。

```
// Read and write POJO, storage will be byte[] since for
// OFF_HEAP the GlobalMarshallerEncoder is used internally:
cache.put(1, new Pojo())
Pojo value = cache.get(1)

// Get the content in its stored format by overriding
// the internal encoder with a no-op encoder (IdentityEncoder)
Cache<?,?> rawContent = cache.getAdvancedCache().withValueEncoding(IdentityEncoder.class)
byte[] marshalled = rawContent.get(1)
```

オーバーライドは、エントリー数のカウントや OFF_HEAP キャッシュの byte [] のサイズの計算など、キャッシュ内の操作にデコードが不要な場合に役立ちます。

3.7.4. カスタムエンコーダーの定義

カスタムエンコーダーは **EncoderRegistry** に登録できます。

注意

キャッシュを起動する前に、クラスターの各ノードで登録が行われていることを確認してください。

gzip で圧縮/解凍するために使用されるカスタムエンコーダを考えてみましょう。

```
public class GzipEncoder implements Encoder {

   @Override
   public Object toStorage(Object content) {
      assert content instanceof String;
      return compress(content.toString());
   }

   @Override
```

```
  public Object fromStorage(Object content) {
    assert content instanceof byte[];
    return decompress((byte[]) content);
  }

  private byte[] compress(String str) {
    try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
         GZIPOutputStream gis = new GZIPOutputStream(baos)) {
      gis.write(str.getBytes("UTF-8"));
      gis.close();
      return baos.toByteArray();
    } catch (IOException e) {
      throw new RuntimeException("Unabled to compress", e);
    }
  }

  private String decompress(byte[] compressed) {
    try (GZIPInputStream gis = new GZIPInputStream(new ByteArrayInputStream(compressed));
         BufferedReader bf = new BufferedReader(new InputStreamReader(gis, "UTF-8"))) {
      StringBuilder result = new StringBuilder();
      String line;
      while ((line = bf.readLine()) != null) {
        result.append(line);
      }
      return result.toString();
    } catch (IOException e) {
      throw new RuntimeException("Unable to decompress", e);
    }
  }

  @Override
  public MediaType getStorageFormat() {
    return MediaType.parse("application/gzip");
  }

  @Override
  public boolean isStorageFormatFilterable() {
    return false;
  }

  @Override
  public short id() {
    return 10000;
  }
}
```

以下で登録できます。

```
GlobalComponentRegistry registry = cacheManager.getGlobalComponentRegistry();
EncoderRegistry encoderRegistry = registry.getComponent(EncoderRegistry.class);
encoderRegistry.registerEncoder(new GzipEncoder());
```

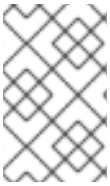
次に、キャッシュからのデータの読み書きに使用します。

```
AdvancedCache<String, String> cache = ...
```

```
// Decorate cache with the newly registered encoder, without encoding keys (IdentityEncoder)
// but compressing values
AdvancedCache<String, String> compressingCache = (AdvancedCache<String, String>)
cache.withEncoding(IdentityEncoder.class, GzipEncoder.class);

// All values will be stored compressed...
compressingCache.put("297931749", "0412c789a37f5086f743255cfa693dd5");

// ... but API calls deals with String
String value = compressingCache.get("297931749");

// Bypassing the value encoder to obtain the value as it is stored
Object value = compressingCache.withEncoding(IdentityEncoder.class).get("297931749");

// value is a byte[] which is the compressed value
```

3.7.5. MediaType

キーと値について、オプションで **org.infinispan.commons.dataconversion.MediaType** でキャッシュを設定できます。キャッシュのデータ形式を記述することで、Red Hat Data Grid はキャッシュ操作時にすぐにデータを変換できます。



注記

MediaType 設定は、バイナリーデータを保存する場合により適しています。サーバーモードを使用する場合、MediaType が設定されていて、REST や Hot336 などのクライアントが異なる形式で読み取り/書き込みを行うことが一般的です。

MediaType 形式のデータ変換は、**org.infinispan.commons.dataconversion.Transcoder**のインスタンスによって処理されます。

```
public interface Transcoder {

   /**
    * Transcodes content between two different {@link MediaType}.
    *
    * @param content         Content to transcode.
    * @param contentType     The {@link MediaType} of the content.
    * @param destinationType The target {@link MediaType} to convert.
    * @return the transcoded content.
    */
   Object transcode(Object content, MediaType contentType, MediaType destinationType);

   /**
    * @return all the {@link MediaType} handled by this Transcoder.
    */
   Set<MediaType> getSupportedMediaTypes();
}
```

3.7.5.1. 設定

宣言設定：

-

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

プログラムによる：

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg.encoding().key().mediaType("text/plain");
cfg.encoding().value().mediaType("application/json");
```

3.7.5.2. MediaType プログラムによるオーバーライド

異なる MediaType でキャッシュをデコードし、キャッシュ操作を実行し、異なるデータフォーマットの送受信を行うことができます。

例:

```
DefaultCacheManager cacheManager = new DefaultCacheManager();

// The cache will store POJO for keys and values
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-java-object");
cfg.encoding().value().mediaType("application/x-java-object");

cacheManager.defineConfiguration("mycache", cfg.build());

Cache<Integer, Person> cache = cacheManager.getCache("mycache");

cache.put(1, new Person("John","Doe"));

// Wraps cache using 'application/x-java-object' for keys but JSON for values
Cache<Integer, byte[]> jsonValuesCache = (Cache<Integer, byte[]>)
cache.getAdvancedCache().withMediaType("application/x-java-object", "application/json");

byte[] json = jsonValuesCache.get(1);
```

JSON 形式で値を返します。

```
{
   "_type":"org.infinispan.sample.Person",
   "name":"John",
   "surname":"Doe"
}
```

注意

ほとんどのトランスフォーマーはサーバーモードが使用される場合にインストールされます。ライブラリーモードを使用する場合は、追加の依存関係 **org.infinispan:infinispan-server-core** をプロジェクトに追加する必要があります。

3.7.5.3. トランスコードおよびエンコーダー

通常、キャッシュ操作に関係するデータ変換はないか、1 つだけです。

- 組み込みモードまたはサーバーモードを使用したキャッシュでは、デフォルトでは変換されません。
- MediaType は設定されず、OFF_HEAP または BINARY を使用する組み込みキャッシュの**エンコーダ** ベースの変換。
- 複数の REST クライアントと Hot Rod クライアントが異なる形式でデータを送受信するサーバーモードで使用されるキャッシュの **トランスコーダー** ベースの変換。これらのキャッシュには、ストレージを記述する MediaType が設定されています。

ただし、高度なユースケースでは、エンコーダーとトランスコーダーの両方を同時に使用できます。

たとえば、マーシャリングされるオブジェクト（jboss marshaller を使用して）コンテンツを格納するキャッシュについて考えてみましょう。セキュリティー上の理由から、「プレーン」データが外部ストアに格納されないように、透過的な暗号化レイヤーを追加する必要があります。クライアントは、複数の形式でデータを読み書きできる必要があります。

これは、エンコーディング層に関係なく、ストレージを記述する MediaType を使用してキャッシュを設定することで実現できます。

```
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
```

透過的な暗号化は、たとえば次のように、保存/取得で暗号化/復号化する特別な **エンコーダー** で キャッシュをデコレートすることで追加できます。

```
public class Scrambler implements Encoder {

  Object toStorage(Object content) {
    // Encrypt data
  }

  Object fromStorage(Object content) {
    // Decrypt data
  }

  MediaType getStorageFormat() {
    return "application/scrambled";
  }

}
```

キャッシュに書き込まれるすべてのデータが暗号化され保存されるようにするには、上記のエンコーダーでキャッシュをデコレートし、このデコレートされたキャッシュですべてのキャッシュ操作を実行する必要があります。

```
Cache<?,?> secureStorageCache =
cache.getAdvancedCache().withEncoding(Scrambler.class).put(k,v);
```

キャッシュを目的の MediaType でデコレートすることにより、複数の形式でデータを読み取る機能を追加できます。

```
// Obtain a stream of values in XML format from the secure cache
secureStorageCache.getAdvancedCache().withMediaType("application/xml","application/xml").values().stream();
```

内部的には、Red Hat Data Grid は最初にエンコーダーを **Storage 操作から** 適用し、"application/x-jboss-marshalling" 形式のエントリーを取得し、適切なトランスフォーマーを使用して「application/xml」に連続した変換を適用します。

第4章 エビクションおよびデータコンテナー

Red Hat Data Grid はエントリーのエビクションをサポートし、メモリーが不足しないようにします。通常、エビクションはキャッシュストアと併用されるため、エビクションはキャッシュストアまたは残りのクラスターからではなくメモリーからエントリーのみを削除し、エントリーがエビクトされると永続的に失われないようにします。

Red Hat Data Grid は、複数の異なる形式でデータの保存をサポートします。データはオブジェクト iself、バイナリーを byte[] として保存し、バイト[] をネイティブメモリーに保存する off-heap として保存できます。

ヒント

パッシベーションはエビクションを使用する場合も一般的なオプションであり、メモリーまたはキャッシュストアのいずれかではなく、エントリーの単一コピーのみが維持されるようにします。通常のキャッシュストアでパッシベーションを使用する主な利点は、更新もキャッシュストアに対して行う必要がないため、メモリーに存在するエントリーへの更新が少なくなることです。

重要

エビクションは **ローカル** ベースで行われ、クラスター全体には含まれません。各ノードはエビクションスレッドを実行してインメモリーコンテナーの内容を分析し、エビクトするかを決定します。エビクションは、エントリーのエビクトを開始するしきい値として JVM の空きメモリー容量を考慮しません。エビクションを有効にするには、エビクション要素の **サイズ** 属性をゼロよりも大きい値に設定する必要があります。サイズが大きすぎる場合は、メモリー不足になる可能性があります。各ユースケースで **サイズ** 属性を調整する必要がある場合があります。

4.1. エビクションの有効化

エビクションは、< **memory** /> 要素を <* **-cache** /> 設定セクションに追加するか、または MemoryConfigurationBuilder API プログラムによるアプローチを使用して設定します。

すべてのキャッシュエントリーは、キャッシュにヒットされるユーザースレッドのペクリーバックによってエビクトされます。

4.1.1. エビクションストラテジー

ストラテジーはエビクションの処理方法を制御します。

可能な選択肢は、

NONE

エビクションは有効ではなく、ユーザーがキャッシュで直接エビクトを呼び出しないことが想定されます。パッシベーションが有効な場合には、警告メッセージが出力されます。これはデフォルトのストラテジーです。

MANUAL

このストラテジーは NONE と同様になりますが、ユーザーが直接エビクトを呼び出すことを想定しています。パッシベーションが有効な場合は、警告メッセージがログに記録されません。

REMOVE

このストラテジーは実際には「古い」エントリーをエビクトし、着信するエントリーを収容します。

エビクションは、追加の受付ウィンドウで TinyLFU アルゴリズムを利用する Caffeine によって処理されます。これは、ヒット率が高くなり、メモリーオーバーヘッドが低いことが求められるため選択されました。これにより、LRU よりもヒット比率が改善されますが、LIRS よりも少ないメモリーも要求されます。

EXCEPTION

このストラテジーにより、**ContainerFullException** を発生させて新規エントリーが作成されるのを防ぎます。このストラテジーは、1 フェーズコミットまたは同期の最適化が許可されない 2 フェーズコミットで常に実行されるトランザクションキャッシュでのみ機能します。

4.1.2. エビクションタイプ

エビクションタイプは、サイズが 0 を超えるものに設定されている場合にのみ適用されます。以下のエビクションタイプは、コンテナーがエントリーの削除を決定するタイミングを決定します。

カウント

このタイプのエビクションは、キャッシュにあるものの数に基づいてエントリーを削除します。エントリーの数が **サイズ** よりも大きいと、部屋を作成するためにエントリーが削除されます。

MEMORY

このタイプのエビクションは、各エントリーがメモリーにかかる量を予測し、すべてのエントリーの合計サイズが設定 **サイズ** よりも大きい場合にエントリーを削除します。このタイプは、以下の **OBJECT** ストレージタイプとは機能しません。

4.1.3. ストレージタイプ

Red Hat Data Grid では、データを格納するフォームを設定できます。各フォームは Red Hat Data Grid と同じ機能をサポートしますが、エビクションはフォームによっては制限できます。現在、Red Hat Data Grid が提供するストレージフォーマットは 3 つあります。

オブジェクト

キーと値をオブジェクトとして Java heap Only **COUNT** エビクションタイプに保存します。

BINARY

キーと値を byte[] として Java ヒープに保存します。キャッシュ用に設定されたマーシャラーを使用します（存在する場合）。**COUNT** および **MEMORY** エビクションタイプの両方がサポートされます。

OFF-HEAP

キーと値をバイトとして Java ヒープ外のネイティブメモリーに保存します。キャッシュに 1 つある場合、設定されたマーシャラーが使用されます。**COUNT** および **MEMORY** エビクションタイプの両方がサポートされます。

警告

***BINARY** および **OFF-HEAP** はいずれも、オブジェクトインスタンスの代わりに生成する結果として生成される byte[] によって指定される等価および hashCode に違反する。*

4.1.4. その他のデフォルト値

*デフォルトでは、< **memory** /> 要素が指定されていない場合、エビクションは実行されません。**OBJECT** ストレージタイプが使用され、**NONE** のストラテジーが想定されます。*

メモリー要素がある場合、以下の表は、xml 設定（Supplied size または Supplied strategy 列の「-」）で提供される情報に基づいてエビクションの動作を記述します。

指定されたサイズ	例	エビクションの動作
-	**<memory />**	オブジェクトとしてエビクションがない
-	**<memory> <object strategy="MANUAL" /> </memory>**	パッシベーションが有効な場合にはエビクションをオブジェクトとして記録せず、警告をログに記録しません。
> 0	**<memory> <object size="100" /> </memory>**	エビクションは実行され、オブジェクトとして保存されます。
> 0	**<memory> <binary size="100" eviction="MEMORY"/> </memory>**	エビクションが行われ、バイナリー削除として保存され、メモリーオフェンスが 100 を超えないようにします。
> 0	**<memory> <off-heap size="100" /> </memory>**	エビクションが行われ、オフヒープに保存される
> 0	**<memory> <off-heap size="100" strategy="EXCEPTION" /> </memory>**	エントリーはオフヒープに保存され、追加で 100 個のエントリーがコンテナー例外にあるとスローされます。
0	**<memory> <object size="0" /> </memory>**	エビクションなし
< 0	**<memory> <object size="-1" /> </memory>**	エビクションなし

4.2. 有効期限

の場合と同様ですが、エビクションとは異なり、エビクションは有効期限です。有効期限により、有効期限や最大アイドル時間をエントリーに割り当てることができます。これらの時間を超えるエントリーは無効として扱われ、削除されます。期限切れのエントリーがエビクトされたエントリーと同様にパッシベートされない場合（パッシベーションがオンの場合）。

ヒント

エビクションとは異なり、期限切れのエントリーはメモリー、キャッシュストア、およびクラスター全体で削除されます。

デフォルトでは、作成されたエントリーは偽装で、ライフスパンや最大アイドル時間はありません。キャッシュ API を使用すると、mortal エントリーは、有効期限または最大アイドル時間で作成できます。さらに、<* **-cache /> 設定セクションに要素を追加<expiration />** **http://docs.jboss.org/infinispan/9.4.0/configdocs/infinispan-config-9.4.0.html することで、デフォルトのライフスパンや最大アイドル時間を設定** できます。

エントリーが期限切れになると、ユーザー要求によって再度アクセスされるまで、データコンテナーまたはキャッシュストアに置かれます。期限切れリーパーも、期限切れのエントリーを確認し、設定可能なミリ秒単位で削除します。

reaper-interval 属性または **ExpirationConfigurationBuilder** クラスの **enableReaper** メソッドを使用して、有効期限を宣言的に有効にできます。

注記

- キャッシュストアが存在する場合、有効期限リーパーを無効にすることはできません。
- クラスター化されたキャッシュで最大アイドル時間を使用する場合、期限切れリーパーを常に有効にする必要があります。詳細は、「Clustered Max Idle 」を参照してください。

4.2.1. エビクションと有効期限の違い

エビクションと有効期限はいずれも未使用のエントリーのキャッシュをクリーンアップすることを目的としているため、**OutOfMemory** 例外からヒープを保護するため、違いを簡単に説明できるようになりました。

エビクション では、キャッシュに保持する **エントリーの最大数** を設定し、この制限を超えると、選択した **エビクションストラテジー** （LRU、LIRS など）に基づいて一部の候補が削除される可能性があります。エビクションは、キャッシュストアへのエビクションであるパッシベーションと連携するように設定できます。

有効期限 では、エントリーの **時間基準** を設定し **て、キャッシュに保持する期間** を指定します。

有効期間

エントリーが期限切れになる前にキャッシュに残っている期間を指定します。デフォルト値は **-1** で、無制限の時間です。

最大アイドル時間

エントリーが期限切れになる前にアイドル状態でいられる期間を指定します。キーで操作が実行されない場合には、キャッシュ内のエントリーはアイドル状態になります。デフォルト値は **-1** で、無制限の時間です。

4.3. 有効期限の詳細

1. **有効期限** は、設定およびキャッシュ API において示されるトップレベルのコンストラクトです。

2. エビクションは **各キャッシュインスタンスに対してローカル** ですが、有効期限は **クラスター全体** になります。expiration ***lifespan*** および ***maxIdle*** の値は、キャッシュエントリーとともにレプリケートされます。

3. キャッシュエントリーの最大アイドル時間には、クラスター環境で追加のネットワークメッセージが必要になります。このため、クラスター化されたキャッシュに ***maxIdle*** を設定すると、操作が遅くなる可能性があります。

4. 有効期限のライフサイクルおよび ***maxIdle*** は CacheStore でも永続化されるため、この情報はエビクション/パッシベーション後も維持されます。

4.3.1. 最大アイドルの有効期限

最大アイドル有効期限は、ローカルおよびクラスターキャッシュ環境での動作が異なります。

重要

アイドルの最大有効期限(***max-idle***)は、現在オフヒープメモリーに保存されるエントリーとは機能しません。同様に、キャッシュストアを永続レイヤーとして使用する場合、***max-idle*** は動作しません。

4.3.1.1. Local Max Idle

ローカルキャッシュモードでは、Red Hat Data Grid は以下の場合に ***maxIdle*** 設定でエントリーを失効させます。

- 直接アクセス（***cache.get（）***）
- 繰り返し処理された（***cache.size（）***）。
- 有効期限のリーパースレッドが実行されます。

4.3.1.2. クラスター化された Max Idle

クラスター化されたキャッシュモードでは、クライアントが ***max-idle*** 有効期限の値を持つエントリーを読み取ると、Red Hat Data Grid は touch コマンドをすべての所有者に送信します。これにより、エントリーの相対アクセス時間がクラスター全体で同じになります。

ノードがエントリーの最大アイドル時間に到達することを検出すると、Red Hat Data Grid はキャッシュから削除され、要求したクライアントにエントリーを返しません。

クラスター化されたキャッシュモードで ***max-idle*** を使用する前に、以下の点を確認する必要があります。

- ***cache.get（）*** は、タッチコマンドが完了するまで返されません。この同期動作により、クライアント要求のレイテンシーが長くなります。
- クラスター化された ***max-idle*** は、Red Hat Data Grid がエビクションに使用するすべての所有者のキャッシュエントリーの「最近アクセスした」メタデータも更新します。

- クラスター化されたキャッシュでの反復を行うと、最大アイドル時間で有効期限が切れる可能性があるエントリーが返されます。この動作は、反復中にリモート呼び出しが実行されないため、パフォーマンスが向上します。ただし、期限切れリーパーによって削除されるエントリーや直接アクセスする場合（**Cache.get（）**）**は期限切れのエントリーは更新されません。**

重要

- クラスター化されたキャッシュは、常に **maxIdle** 設定で expiration reaper を使用する必要があります。
- 例外ベースのエビクションで **maxIdle** 有効期限を使用する場合、期限切れだが、データコンテナーのサイズに対してキャッシュ数から削除されないエントリー。

4.3.2. 設定

エビクションおよび有効期限は、プログラムによる XML 設定を使用して設定できます。この設定はキャッシュごとに行われます。有効なエビクション/有効期限関連の設定要素は以下のとおりです。

```
<!-- Eviction -->
<memory>
   <object size="2000"/>
</memory>
<!-- Expiration -->
<expiration lifespan="1000" max-idle="500" interval="1000" />
```

プログラムを用いて、以下を使用して同様のものを定義します。

```
Configuration c = new ConfigurationBuilder()
            .memory().size(2000)
            .expiration().wakeUpInterval(5000l).lifespan(1000l).maxIdle(500l)
            .build();
```

4.3.3. メモリーベースのエビクション設定

Red Hat Data Grid は通常使用されるため、メモリーベースのエビクションでは、独自のカスタムタイプを使用している場合に、追加の設定オプションが必要になる場合があります。この場合、Red Hat Data Grid はクラスのメモリー使用量を予測できないため、メモリーベースのエビクションが使用される場合は **storeAsBinary** を使用する必要があります。

```
<!-- Enable memory based eviction with 1 GB/>
<memory>
   <binary size="1000000000" eviction="MEMORY"/>
</memory>
```

```
Configuration c = new ConfigurationBuilder()
            .memory()
            .storageType(StorageType.BINARY)
            .evictionType(EvictionType.MEMORY)
            .size(1_000_000_000)
            .build();
```

4.3.4. デフォルト値

エビクションはデフォルトで無効にされています。デフォルト値が使用されます。

- size: 指定されていない場合は -1 が使用されます。これは、無制限のエントリーを意味します。
- 0 はエントリーがなく、エビクションスレッドはキャッシュを空に保ちます。

有効期限と maxIdle の両方は、デフォルトで -1 に設定されます。これは、デフォルトでエントリーが作成されることを意味します。これは、API でエントリーごとに上書きできます。

4.3.5. 有効期限の使用

有効期限により、キャッシュに保存されているキー/値のペアに有効期限または最大アイドル時間を設定できます。これは、上記のように設定を使用してキャッシュ全体を設定するか、Cache インターフェースを使用してキー/値のペアごとに定義できます。キー/値のペアに定義した値は、対象の特定のエントリーのキャッシュ全体のデフォルトを上書きします。

たとえば、以下の設定を前提とします。

```
<expiration lifespan="1000" />
```

```
// this entry will expire in 1000 millis
cache.put("pinot noir", pinotNoirPrice);

// this entry will expire in 2000 millis
cache.put("chardonnay", chardonnayPrice, 2, TimeUnit.SECONDS);

// this entry will expire 1000 millis after it is last accessed
cache.put("pinot grigio", pinotGrigioPrice, -1,
        TimeUnit.SECONDS, 1, TimeUnit.SECONDS);

// this entry will expire 1000 millis after it is last accessed, or
// in 5000 millis, which ever triggers first
cache.put("riesling", rieslingPrice, 5,
        TimeUnit.SECONDS, 1, TimeUnit.SECONDS);
```

4.4. 有効期限の設計

有効期限の中央は ExpirationManager です。

ExpirationManager の目的は、DataContainer からアイテムを定期的にパージする有効期限スレッドを駆動することです。有効期限スレッドが無効である（wakeupInterval が -1）である場合は、アプリケーションで定期的に実行される可能性のある別のメンテナンススレッドなどを使用して、ExpprationManager.processExpiration（）を使用して有効期限を手動で起動できます。

有効期限マネージャーは以下の方法で有効期限を処理します。

1. データコンテナーが期限切れのエントリーをパージする
2. キャッシュストア（ある場合）が期限切れのエントリーをパージする

第5章 永続性

永続性により、外部（永続）ストレージエンジンは、Red Hat Data Grid が提供するメモリーストレージのデフォルトに補完できます。外部の永続ストレージは、以下のような理由で役に立ちます。

- 永続性の向上メモリーは揮発性なので、キャッシュストアは情報ストアのライフサイクルをキャッシュに増やすことができます。
- ライトスルー。アプリケーションと（カスタム）外部ストレージエンジンとの間のキャッシング層として Red Hat Data Grid を干渉します。
- オーバーフローデータ。エビクションおよびパッシベーションを使用すると、メモリーに「ホット」データのみを保存し、ディスクに頻繁に使用されるデータをオーバーフローさせることができます。

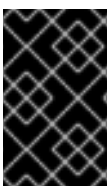
永続ストアとの統合は、CacheLoader、CacheWriter、AdvancedCacheLoader、および AdvancedCacheWriter（以下のセクションで説明）の SPI により行われます。

これらの SPI は、以下の機能を許可します。

- JSR-107 と連携します。CacheWriter および CacheLoader インターフェースは、JSR 107 のローダーとライターに似ています。これは、JCache 準拠のベンダー全体でポータブルストアを作成するのに非常に役立ちます。
- 簡略化されたトランザクション統合。必要なロックはすべて Red Hat Data Grid により自動的に処理され、実装はストアへの同時アクセスを調整する必要はありません。同じキーで同時書き込みは行われませんが（使用中のロックモードによる）、実装側はストアでの操作が複数のスレッドから行われることを期待し、それに応じて実装のコーディングを行う必要があります。
- 並列イタリケーション。複数のスレッドを持つストアのエントリーを繰り返し処理できるようになりました。
- シリアライゼーションを削減します。これにより、CPU 使用率が少なくなります。新しい API は、保存されたエントリーをシリアライズ形式で公開します。エントリーがリモートから送信される唯一の目的のために永続ストレージから取得される場合、デシリアライズ（ストアからの読み取り時）をシリアライズし、（ワイヤへの書き込み時）にシリアライズする必要がなくなりました。これで、ストレージから直接読み取ってシリアライズされた形式に書き込みを行うことができます。

5.1. 設定

チェーンには（reader や writers）を保存できます。キャッシュの読み取り操作は、有効かつ null 以外の要素が見つかるまで、設定される順序で指定されたすべての **CacheLoader** を参照します。書き込みを実行すると、特定 **のキャッシュ** ライターに対して **ignoreModifications** 要素が true に設定されている場合を除き、すべてのキャッシュキャッシュライターが書き込まれます。



CACHEWRITER と CACHELOADER の両方の実装

ストアプロバイダーは **CacheWriter** インターフェースと **CacheLoader** インターフェースの両方を実装する必要があります。これを実行するストアは、(**read-only=false**) データの読み取りと書き込みの両方について考慮されます。

```
This is the configuration of a custom(not shipped with infinispan) store:
  <local-cache name="myCustomStore">
    <persistence passivation="false">
```

```
        <store
          class="org.acme.CustomStore"
          fetch-state="false" preload="true" shared="false"
          purge="true" read-only="false" singleton="false">

          <write-behind modification-queue-size="123" thread-pool-size="23" />

          <property name="myProp">${system.property}</property>
        </store>
      </persistence>
    </local-cache>
```

永続性の設定に使用できるパラメーターは、以下のとおりです。

connection-attempts

設定済みの各 CacheWriter/CacheLoader の起動を試行する最大回数を設定します。開始試行に成功しなかった場合は、例外が発生し、キャッシュは起動しません。

connection-interval

起動時に接続試行間の待機時間をミリ秒単位で指定します。負の値またはゼロの値は、接続試行間の待ち時間がないことを意味します。

availability-interval

PersistenceManager が利用可能かどうかを判断するために可用性チェックの間隔（ミリ秒単位）を指定します。つまり、この間隔は **org.infinispan.persistence.spi.CacheWriter#isAvailable** または **org.infinispan.persistence.spi.CacheLoader#isAvailable** 実装を介してストア/ローダーをポーリングする頻度を設定します。単一のストア/ローダーが利用できない場合、キャッシュ操作中に例外が発生します。

passivation

パッシベーションを有効にします。デフォルト値は **false** （ブール値）です。
このプロパティーは、ローダーとの Red Hat Data Grid の対話に大きな影響があります。詳細は、「キャッシュパッシベーション」を参照してください。

class

ストアのクラスを定義し、**CacheLoader**、**CacheWriter**、またはその両方を実装する必要があります。

fetch-state

クラスターに参加する際にキャッシュの永続状態を取得します。デフォルト値は **false** （ブール値）です。
このプロパティーの目的は、キャッシュの永続的な状態を取得し、クラスターに参加する際にノードのローカルキャッシュストアに適用することです。永続状態の取得は、キャッシュストアが他のストアと同じデータにアクセスするため、共有されている場合は適用されません。

このプロパティーは設定済みの 1 つのキャッシュローダーにのみ **該当** します。複数のキャッシュローダーが永続状態を取得すると、キャッシュサービスの起動時に設定例外が発生します。

preload

キャッシュの起動時にデータをキャッシュローダーからメモリーに事前読み込みます。デフォルト値は **false** （ブール値）です。
このプロパティーは、データが遅延的に読み込まれると、起動直後にキャッシュローダー内のデータが必要な場合に、キャッシュ操作の遅延を防ぐ場合に便利です。このプロパティーは起動時に「ウォームキャッシュ」を提供できますが、開始時間に影響するためパフォーマンスに影響します。

事前にロードされたデータはローカルで実行されるため、読み込まれたデータはローカルにのみ保存されます。事前に読み込んだデータはレプリケートまたは分散されません。同様に、Red Hat Data Grid は、エビクション で設定される最大エントリー数だけを事前読み込みます。

shared

キャッシュローダーがキャッシュインスタンス間で共有されるかどうかを判断します。デフォルト値は **false** （ブール値）です。
このプロパティーにより、異なるキャッシュインスタンスによるキャッシュローダーへのデータの複製書き込みが回避されます。たとえば、クラスターのすべてのキャッシュインスタンスは、同じリモート共有データベースに同じ JDBC 設定を使用します。

read-only

新規データがキャッシュストアに永続化されないようにします。デフォルト値は **false** （ブール値）です。

purge

起動時に指定されたキャッシュローダーを除算します。デフォルト値は **false** （ブール値）です。このプロパティーは、**読み取り専用** のプロパティーが **false** に設定されている場合に限り有効です。

max-batch-size

キャッシュストアからの削除を挿入するバッチの最大サイズを設定します。デフォルト値は 100 です。
値が **1** 未満の場合、上限はバッチの操作数には適用されません。

write-behind

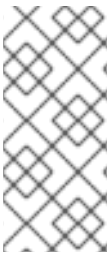
非同期的には、キャッシュストアにデータを永続化します。デフォルト値は **false** （ブール値）です。詳細は「非同期書き込み /書き込み/

singleton

クラスターの 1 つのノード（コーディネーター）を有効にし、変更を保存します。デフォルト値は **false** （ブール値）です。
データがノードに入るたびに、キャッシュのメモリー内の状態を維持するために複製または分散されます。コーディネーターは状態をディスクにプッシュします。

true の場合、クラスター内のすべてのノードにこのプロパティーを設定する必要があります。クラスターのコーディネーターのみがデータを永続化します。ただし、他のノードがデータを永続化しないように、すべてのノードがこのプロパティーを有効にする必要があります。

キャッシュストアが共有されている場合は **singleton** プロパティーを有効にすることはできません。



注記

properties セクションで追加の属性を定義して、前の例の **myProp** 属性など、各キャッシュローダーの特定側面を設定できます。

より複雑な設定を含む他のキャッシュローダーには、追加のプロパティーも含まれます。例については、以下の JDBC キャッシュストア設定を参照してください。

上記の設定では、一般的なキャッシュストアの実装が適用されます。ただし、デフォルトの Red Hat Data Grid ストアの実装には、さらに複雑な設定スキーマがあり、**properties** セクションは XML 属性に置き換えられます。

```
<persistence passivation="false">
  <!-- note that class is missing and is induced by the fileStore element name -->
  <file-store
        shared="false" preload="true"
        fetch-state="true"
        read-only="false"
        purge="false"
        path="${java.io.tmpdir}">
    <write-behind thread-pool-size="5" />
  </file-store>
</persistence>
```

同じ設定をプログラムで実行できます。

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

5.2. キャッシュパッシベーション

CacheWriter を使用すると、キャッシュでエントリーパッシベーションおよびエビクションでアクティベートを実行できます。キャッシュパッシベーションとは、インメモリーキャッシュからオブジェクトを削除し、これをエビクションのセカンダリーデータストア（ファイルシステム、データベースなど）に書き込むプロセスのことです。キャッシュのアクティベーションは、使用する必要がある場合には、データストアからインメモリーキャッシュにオブジェクトを復元するプロセスです。パッシベーションを完全にサポートするには、ストアを CacheWriter と CacheLoader の両方にする必要があります。いずれの場合も、設定されたキャッシュストアはローダーから読み込み、データライターへの書き込みに使用されます。

effect のエビクションポリシーがキャッシュからエントリーをエビクトすると、パッシベーションが有効にされている場合、エントリーがパッシベートされたことを示す通知がキャッシュリスナーに出力され、エントリーはキャッシュリスナーに保存されます。ユーザーが以前にエビクトされたエントリーの取得を試みると、エントリーはキャッシュローダーからメモリーに読み込まれた（レイジー）になります。エントリーがロードされると、エントリーがアクティベートされたキャッシュリスナーに通知が生成されます。パッシベーションを有効にするには、パッシベーションを true に設定します（デフォルトでは false）。パッシベーションが使用される場合、設定された最初のキャッシュローダーのみが使用され、他のすべてのキャッシュローダーは無視されます。

5.2.1. 制限事項

パッシベーションの性質上、他のストア設定では対応していません。

- トランザクションストア： 実際の Infinispan コミット境界の範囲外にあるストアからエントリーを書き込み/削除します。
- 共有ストア： 共有ストアでは、他の所有者のためにエントリーが常にストアになければなりません。そのため、ストアからエントリーを削除できないため、パッシベーションは意味がありません。

5.2.2. パッシベーションが無効になったキャッシュローダー動作と有効化

パッシベーションが無効の場合、要素の変更、追加、または削除のたびに、キャッシュローダーを介してその変更がバックエンドストアで永続化されます。エビクションとキャッシュの読み込みの間に直接関係はありません。エビクションを使用しない場合、永続ストアの内容は基本的にメモリー内の内容のコピーになります。エビクションを実行する場合、永続的ストアには基本的にメモリー内の内容のスーパーセットになります（つまり、メモリーからエビクトされたエントリーが含まれます）。パッシベーションが有効になっていて、共有されていないストアの場合には、エビクションとキャッシュローダーの間に直接関係があります。キャッシュローダーを介して永続ストアへの書き込みは、エビクションプロセスの一部としてのみ実行されます。アプリケーションがメモリーに読み取られると、データは永続ストアから削除されます。この場合、メモリー内の内容と永続ストアは合計情報セットの 2 つのサブセットであり、サブセット間で交差はありません。共有ストアでは、以前にパッシベーションされたエントリーはストアに存在しますが、メモリーで上書きされた場合は古い値がある可能性があります。

以下の例は、6 ステッププロセスの各ステップの後に RAM と永続ストアのステータスを示す簡単な例です。

操作	パッシベーションオフ	パッシベーション、共有オフ	パッシベーション、共有オン
keyOne を挿入	**memory:** keyOne **Disk:** keyOne	**memory:** keyOne **Disk:** （なし）	**memory:** keyOne **Disk:** （なし）
keyTwo の挿入	**memory:** keyOne, keyTwo **Disk:** keyOne, keyTwo	**memory:** keyOne, keyTwo **Disk:** (none)	**memory:** keyOne, keyTwo **Disk:** (none)
エビクションスレッドが実行され、keyOne のエビクト	**memory:** keyTwo **Disk:** keyOne, keyTwo	**memory:** keyTwo **Disk:** keyOne	**memory:** keyTwo **Disk:** keyOne
keyOne の読み取り	**memory:** keyOne, keyTwo **Disk:** keyOne, keyTwo	**memory:** keyOne, keyTwo **Disk:** (none)	**memory:** keyOne, keyTwo **Disk:** keyOne
エビクションスレッドが実行され、keyTwo のエビクト	**memory:** keyOne **Disk:** keyOne, keyTwo	**memory:** keyOne **Disk:** keyTwo	**memory:** keyOne **Disk:** keyOne, keyTwo
keyTwo の削除	**memory:** keyOne **Disk:** keyOne	**memory:** keyOne **Disk:** （なし）	**memory:** keyOne **Disk:** keyOne

5.3. キャッシュロードおよびトランザクションキャッシュ

キャッシュがトランザクションで、キャッシュローダーが存在する場合、キャッシュローダーはキャッシュが含まれるトランザクションに登録されません。つまり、キャッシュローダーレベルで不整合が発生する可能性があります。つまり、トランザクションはメモリー内の状態の適用に成功しますが、（特に）ストアへの変更の適用に失敗します。手動リカバリーはキャッシュストアでは機能しません。

5.4. ライトアンド経由のライト-BEHIND CACHING

任意で1つまたは複数のキャッシュストアでRed Hat Data Gridを設定すると、共有JDBCデータベース、ローカルファイルシステムなどの永続的な場所にデータを保存できます。Red Hat Data Gridはキャッシュストアへの更新を2種類の方法で処理できます。

- ライト経由（同期）
- write-Behind（非同期）

5.4.1. ライト経由（同期）

バージョン4.0でサポートされるこのモードでは、クライアントがキャッシュエントリーを更新する場合（cache.put（）呼び出しを介して）、Red Hat Data Gridが基礎となるキャッシュストアに切り替わり、更新されるまで呼び出しは返されません。通常、キャッシュストアへの更新は、クライアントスレッドの境界内で行われることを意味します。

このモードの主な利点は、キャッシュストアがキャッシュと同時に更新されるため、キャッシュストアはキャッシュの内容と一致していることです。一方、このモードを使用すると、キャッシュストアにアクセス、更新する必要があるレイテンシーがキャッシュ操作の期間に直接影響するため、このモードを使用するとパフォーマンスが低下します。

ライトスルーまたは同期キャッシュストアを設定すると、特定の設定オプションは必要ありません。デフォルトでは、明示的にwrite-behindまたはasynchronousとマークされない限り、すべてのキャッシュストアはライトスルーまたは同期になります。以下の、共有されていないローカルファイルキャッシュストアの設定ファイルのサンプルをご覧ください。

```
<persistence passivation="false">
  <file-store fetch-state="true"
              read-only="false"
              purge="false" path="${java.io.tmpdir}"/>
</persistence>
```

5.4.2. write-Behind（非同期）

このモードでは、キャッシュへの更新は非同期でキャッシュストアに書き込まれます。Red Hat Data Gridは、変更を素早く保存できるように、保留中の変更を変更キューに配置します。

設定されたスレッド数はキューを消費し、基礎となるキャッシュストアに変更を適用します。設定されたスレッド数が変更をすばやく消費できない場合、または基盤のストアが利用できなくなると、変更キューが満杯になります。この場合、キャッシュストアは、キューが新しいエントリーを受け入れるまでライトスルーになります。

このモードでは、キャッシュ操作が基礎となるストアへの更新の影響を受けません。ただし、更新は非同期的に行われるため、キャッシュストアのデータはキャッシュのデータと一貫性のない期間があります。

write-behind ストラテジーは、低レイテンシーおよび小規模な操作コストを持つキャッシュストアに適しています。たとえば、キャッシュ自体にローカルとなる非共有ファイルベースのキャッシュストアなどです。この場合、キャッシュストアとキャッシュ間のデータの一貫性が最小期間に短縮されます。

以下は、write-behind ストラテジーの設定例です。

```
<persistence passivation="false">
  <file-store fetch-state="true"
              read-only="false"
              purge="false" path="${java.io.tmpdir}">
  <!-- start write-behind configuration -->
  <write-behind modification-queue-size="123" thread-pool-size="23" />
  <!-- end write-behind configuration -->
  </file-store>
</persistence>
```

5.5. ファイルシステムベースのキャッシュストア

通常、ファイルシステムベースのキャッシュストアは、メモリーからオーバーフローしたデータを保存し、サイズや時間の制限を超過する、ローカルで利用可能なキャッシュストアを持つキャッシュを取得する場合に使用されます。

警告

NFS、Windows 共有などの共有ファイルシステムでファイルシステムベースのキャッシュストアを使用すると、適切なファイルロックが実装されず、データが破損する可能性があるため、回避する必要があります。ファイルシステムは通常トランザクションではないため、トランザクションコンテキストでキャッシュを使用しようとすると（コミットフェーズ中に発生する）ファイルへの書き込みに失敗します。

5.5.1. 単一ファイルストア

単一ファイルキャッシュストアは、すべてのデータを 1 つのファイルに保持します。データを検索する方法は、キーのインメモリーインデックスと、その値の位置を保持することで、このファイルにあります。これにより、古いファイルキャッシュストアと比較してパフォーマンスが向上します。ただし、注意点があります。1 つのファイルベースのキャッシュストアは鍵をメモリーに保持するため、メモリー消費が増大するため、大きなキーを持つキャッシュを使用することは推奨されません。

特定のユースケースでは、このキャッシュストアは断片化からの影響を受けます。大きな値を保存すると、領域は再利用されず、エントリーがファイルの末尾に追加されます。スペース（現在は空）は、配置できる別のエントリーを書き込む場合にのみ再利用されます。また、キャッシュからすべてのエントリーを削除すると、ファイルは縮小されず、デフラグされません。

以下は、単一のファイルキャッシュストアで使用できる設定オプションです。

- **path** は、Red Hat Data Grid がデータを格納するファイルシステムの場所です。
  埋め込みデプロイメントの場合、Red Hat Data Grid はデータを現在の作業ディレクトリーに保存します。別の場所にデータを保存するには、そのディレクトリーへの絶対パスを値として指定します。

サーバーのデプロイメントの場合には、Red Hat Data Grid はデータを **$RHDG_HOME/data** に保存します。Red Hat Data Grid サーバーのデフォルトの場所は変更しないでください。つまり、サーバーデプロイメントで **path** の値を設定しないでください。

パス宣言を作成し、ファイルストアの場所に **relative-to** 属性を使用する必要があります。**relative-to** をディレクトリーの場所または環境変数に設定すると、Red Hat Data Grid サーバーの起動に失敗します。

- **max-entries** は、このファイルストアに保持するエントリーの最大数を指定します。前述のように、ルックアップを迅速化するために、単一のファイルキャッシュストアはキーとその対応する位置をファイルに保持します。このインデックスでメモリー消費の問題が発生するのを回避するために、このキャッシュストアは、保存するエントリーの最大数によってバインドされます。この制限を超えると、インメモリーインデックスと基礎となるファイルベースのキャッシュストアの両方から LRU アルゴリズムを使用してエントリーが完全に削除されます。そのため、Red Hat Data Grid がキャッシュとして使用される場合にのみ最大制限を設定すると、コンテンツを再計算できるか、権威データストアから取得できます。Red Hat Data Grid が権威データストアとして使用されるときにこの最大制限が設定されている場合には、データが失われる可能性があるため、このユースケースでは推奨されません。デフォルト値は **-1** で、ファイルストアサイズは無制限になります。

5.5.1.1. 宣言型設定

以下の例は、単一のファイルキャッシュストアの宣言設定を示しています。

ライブラリーモード

```
<persistence>
  <file-store path="/tmp/myDataStore" max-entries="5000"/>
</persistence>
```

サーバーモード

```
<persistence>
  <file-store max-entries="5000"/>
</persistence>
```

5.5.1.2. プログラムによる設定

以下の例は、単一のファイルキャッシュストアのプログラムによる設定を示しています。

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
   .addSingleFileStore()
   .location("/tmp/myDataStore")
   .maxEntries(5000);
```

5.5.2. soft-index ファイルストア

Soft-Index ファイルストアは、実験的なローカルのファイルベースのものです。これは、Java のソフト参照を使用してインメモリーにキャッシュされる B+ ツリーのバリアントを実装することで、単一ファイルストアの欠点を確保しようとする純粋な Java 実装です。ここでは、Soft-Index File Store とい

う名前の Soft-Index File Store という名前が使用されます。この B+ ツリー（インデックス と呼ばれる）はファイルシステム上で永続化する必要がない単一ファイルにオフロードされます。キャッシュストアの再起動時に削除され再ビルドされます。その目的はオフロードのみになります。

永続化する必要のあるデータは、追加のみの方法で記述される一連のファイルに保存されます。つまり、従来のマジックディスクにこれを保存する場合は、エントリーが急増する時にシークを行う必要はありません。これは単一ファイルに保存されず、ファイルのセットに保存されます。このファイルの使用が 50% 未満になると（ファイルのエントリーが別のファイルに上書きされる）、そのファイルが収集され、ライブエントリーが異なるファイルに移動し、最後にそのファイルをディスクから削除します。

Soft Index File Store の構造のほとんどはバインドされるため、OOMEs の RAID を取得する必要はありません。たとえば、同時に開かれたファイルに制限を設定することもできます。

5.5.2.1. 設定

以下は、XML による Soft-Index ファイルストアの設定例になります。

```
<persistence>
   <soft-index-file-store xmlns="urn:infinispan:config:store:soft-index:8.0">
      <index path="/tmp/sifs/testCache/index" />
      <data path="/tmp/sifs/testCache/data" />
   </soft-index-file-store>
</persistence>
```

プログラムによる設定は以下のようになります。

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
   .addStore(SoftIndexFileStoreConfigurationBuilder.class)
      .indexLocation("/tmp/sifs/testCache/index");
      .dataLocation("/tmp/sifs/testCache/data")
```

5.5.2.2. 現在の制限

インデックス内のノードのサイズは制限されていますが、デフォルトでは設定可能ですが 4096 バイトです。このサイズにより、キーの長さ（またはシリアライズされた形式の長さ）も制限されます。ノードのサイズよりも長いキーを使用することはできません - 15 バイト。さらに、鍵の長さは「short」として保存され、これを 32767 バイトに制限します。廃止されたキーの使用方法はありません。キーがシリアライズ後になくなった場合に SIFS が例外をスローします。

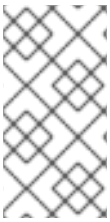
エントリーが有効期限に保存されると、SIFS はこれらのエントリーの一部の有効期限が切れることを検知できません。そのため、このような古いファイルは圧縮されません（AdvancedStore.purgeExpired（）のメソッドは実装されません）。これにより、ファイルシステムの領域が過剰に使用される可能性があります。

5.6. JDBC 文字列ベースのキャッシュストア

提供された JDBC ドライバーに依存して基盤のデータベースに値をロード/保存するキャッシュストア。

キャッシュの各キーはデータベースの独自の行に保存されます。各キーを独自の行に格納するために、このストアは、各キーを String オブジェクトにマップする（プラグ可能な）ボーナクションに依存します。ボットは Key2StringMapper インターフェースで定義されます。Red Hat Data Grids には、プリミ

ティブ型の処理方法を認識するデフォルト実装（主に DefaultTwoWayKey2StringMapper）が同梱されます。



注記

デフォルトでは、Red Hat Data Grid 共有は保存されません。つまり、更新ごとにクラスター内の全ノードが基礎となるストアに書き込まれます。操作をベースとなるデータベースにのみ書き込みする場合は、JDBC ストアを共有するよう設定する必要があります。

5.6.1. 接続管理（プール）

データベースへの接続を取得するために、JDBC キャッシュストアは ConnectionFactory 実装に依存します。接続ファクトリーは、JdbcStringBasedStoreConfigurationBuilder クラスの connectionPool（）メソッド、dataSource（）メソッド、または simpleConnection（）メソッドのいずれかを使用してプログラム的に指定されます。または、< **connectionPool** />、< **dataSource** />、または< **simpleConnection** /> 要素のいずれかを使用して宣言的に指定されます。

Red Hat Data Grid には 3 つの ConnectionFactory 実装が含まれています。

- PooledConnectionFactoryConfigurationBuilder は HikariCP に基づくファクトリーです。HikariCP の追加プロパティーは、クラスパスに **hikari.properties** ファイルを配置するか、接続プールの xml 設定の **PooledConnectionFactoryConfiguration.propertyFile** または **properties-file** でファイルへのパスを指定して、プロパティーファイルにより提供できます。N.B. 設定で指定したプロパティーファイルは、クラスパス上の **hikari.properties** ファイルの代わりにロードされます。また、PooledConnectionFactoryConfiguration で明示的に設定される Connection プール特性は、常にプロパティーファイルからロードされた値を上書きします。

すべての設定プロパティーの詳細については、公式 ドキュメント を参照してください。

- ManagedConnectionFactoryConfigurationBuilder は、アプリケーションサーバーなどの管理環境内で使用できる接続ファクトリーです。これは、特定の場所（設定可能な）で JNDI ツリーを確認し、接続管理を DataSource に委譲する方法を認識します。

- SimpleConnectionFactoryConfigurationBuilder は、呼び出しごとにデータベース接続を作成するファクトリー実装です。実稼働環境では推奨されません。

通常、**pooled** ConnectionFactory はスタンドアロンデプロイメントに推奨されます（AS またはサーブレットコンテナー内では実行されません）。**DataSource** が存在する管理環境で実行する場合は **ManagedConnectionFactory** を使用できます。これにより、接続プールは **DataSource** 内で実行されます。

5.6.2. 設定例

以下は、JdbcStringBasedStore の設定例です。使用されるすべてのパラメーターの詳細な説明は、JdbcStringBasedStore を参照してください。

```
<persistence>
  <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:9.4" dialect="H2">
    <connection-pool connection-url="jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1"
username="sa" driver="org.h2.Driver"/>
    <string-keyed-table drop-on-exit="true" create-on-start="true" prefix="ISPN_STRING_TABLE">
      <id-column name="ID_COLUMN" type="VARCHAR(255)" />
      <data-column name="DATA_COLUMN" type="BINARY" />
      <timestamp-column name="TIMESTAMP_COLUMN" type="BIGINT" />
```

```
    </string-keyed-table>
  </string-keyed-jdbc-store>
</persistence>
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
    .shared(true)
    .table()
      .dropOnExit(true)
      .createOnStart(true)
      .tableNamePrefix("ISPN_STRING_TABLE")
      .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
      .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
      .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .connectionPool()
      .connectionUrl("jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1")
      .username("sa")
      .driverClass("org.h2.Driver");
```

最後に、データソース JNDI の場所を指定して暗黙的に選択される管理対象接続ファクトリーを持つ JDBC キャッシュストアの例は次のとおりです。

```
<persistence>
  <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:9.4" shared="true" dialect="H2">
    <data-source jndi-url="java:/StringStoreWithManagedConnectionTest/DS" />
    <string-keyed-table drop-on-exit="true" create-on-start="true" prefix="ISPN_STRING_TABLE">
      <id-column name="ID_COLUMN" type="VARCHAR(255)" />
      <data-column name="DATA_COLUMN" type="BINARY" />
      <timestamp-column name="TIMESTAMP_COLUMN" type="BIGINT" />
    </string-keyed-table>
  </string-keyed-jdbc-store>
</persistence>
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
    .shared(true)
    .table()
      .dropOnExit(true)
      .createOnStart(true)
      .tableNamePrefix("ISPN_STRING_TABLE")
      .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
      .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
      .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .dataSource()
      .jndiUrl("java:/StringStoreWithManagedConnectionTest/DS");
```

5.7. リモートストア

RemoteStore は、リモートの Red Hat Data Grid クラスターにデータを格納するキャッシュローダーおよびライター実装です。リモートクラスターと通信するために、**RemoteStore** は HotRod クライアント/サーバーアーキテクチャーを使用します。ホットrod は、呼び出しの負荷分散とフォールトトレランスを悪用し、RemoteCacheStore と実際のクラスター間の接続を微調整できます。プロトコル、クライアント、およびサーバー設定の詳細は「Hotke」を参照してください。RemoteStore 設定の一覧は、javadoc を参照してください。例:

5.7.1. 使用例

```
<persistence>
  <remote-store xmlns="urn:infinispan:config:store:remote:8.0" cache="mycache" raw-values="true">
    <remote-server host="one" port="12111" />
    <remote-server host="two" />
    <connection-pool max-active="10" exhausted-action="CREATE_NEW" />
    <write-behind />
  </remote-store>
</persistence>
```

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence().addStore(RemoteStoreConfigurationBuilder.class)
    .fetchPersistentState(false)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .remoteCacheName("mycache")
    .rawValues(true)
.addServer()
    .host("one").port(12111)
    .addServer()
    .host("two")
    .connectionPool()
    .maxActive(10)
    .exhaustedAction(ExhaustedAction.CREATE_NEW)
    .async().enable();
```

この設定例では、リモートキャッシュストアが「one」および「two」のサーバーで「mycache」という名前のリモートキャッシュを使用するように設定されています。また、接続プールを設定し、カスタムトランスポートエグゼキューターを提供します。さらに、キャッシュストアは非同期です。

5.8. クラスターキャッシュローダー

ClusterCacheLoader は、他のクラスターメンバーからデータを取得するキャッシュローダー実装です。

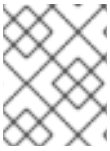
これは、（ストアではない）何も永続化しないため（ストアではない）キャッシュローダーであるため、**fetchPersistentState** などの機能は該当しません。

クラスターキャッシュローダーは、State **Transfer** の代わりにノンブロッキング（部分的）として使用できます。ローカルノードでまだ利用できないキーは、クラスター内の他のノードからオンデマンドでフェッチされます。これは、キャッシュコンテンツの Lazy-loading の種類です。

```
<persistence>
  <cluster-loader remote-timeout="500"/>
</persistence>
```

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
   .addClusterLoader()
   .remoteCallTimeout(500);
```

ClusterCacheLoader 設定の一覧は、javadoc を参照してください。



注記

ClusterCacheLoader は preloading(preload=true) をサポートしません。
fetchPersistentSate=true の場合も状態を提供しません。

5.9. コマンドラインインターフェースキャッシュローダー

コマンドラインインターフェース(CLI)キャッシュローダーは、CLI を使用して別の Red Hat Data Grid ノードからデータを取得するキャッシュローダー実装です。CLI が接続するノードはスタンドアロンノードであったり、クラスターの一部であるノードであったりします。このキャッシュローダーは読み取り専用であるため、データの取得にのみ使用されます。そのため、データの永続化時には使用されません。

CLI キャッシュローダーは、接続する Red Hat Data Grid ノードを参照する接続 URL で設定されます。以下は例です。

注記

URL の形式や、ノードが CLI 経由で呼び出しを受信できるようにする方法は、「コマンドラインインターフェース」の章 を参照してください。

```
<persistence>
  <cli-loader connection="jmx://1.2.3.4:4444/MyCacheManager/myCache" />
</persistence>
```

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
   .addStore(CLInterfaceLoaderConfigurationBuilder.class)
   .connectionString("jmx://1.2.3.4:4444/MyCacheManager/myCache");
```

5.10. ROCKSDB キャッシュストア

Red Hat Data Grid は、RocksDB をキャッシュストアとして使用してサポートします。

5.10.1. はじめに

RocksDB は、Facebook から高速なキーと値のファイルシステムベースのストレージです。Google の LevelDB のフォークとして開始しましたが、特に同時実行シナリオにおいて優れたパフォーマンスおよび信頼性が向上します。

5.10.1.1. 使用例

RocksDB キャッシュストアは、2 つのファイルシステムディレクトリーを設定する必要があります。各ディレクトリーには、RocksDB データベースが含まれます。1 つの場所は、期限切れでないデータを保存するために使用されます。2 番目の場所は、期限切れのキーがパージを保留された鍵を保存するために使用されます。

```
Configuration cacheConfig = new ConfigurationBuilder().persistence()
   .addStore(RocksDBStoreConfigurationBuilder.class)
   .build();
EmbeddedCacheManager cacheManager = new DefaultCacheManager(cacheConfig);
```

```
Cache<String, User> usersCache = cacheManager.getCache("usersCache");
usersCache.put("raytsang", new User(...));
```

5.10.2. 設定

基礎となる rocks db インスタンスを設定することもできます。これは、ストア設定のプロパティーを使用して実行できます。**データベース** の名前が付けられたプロパティーはすべて rocks db データベースを設定します。データは列ファミリーに保存され、name **データ** のプレフィックスが付けられたプロパティーを設定することで、データベースとは独立して設定できます。

プロパティーを指定する必要はありませんが、これは完全に任意であることに注意してください。

5.10.2.1. プログラミングの設定例

```
Properties props = new Properties();
props.put("database.max_background_compactions", "2");
props.put("data.write_buffer_size", "512MB");

Configuration cacheConfig = new ConfigurationBuilder().persistence()
    .addStore(RocksDBStoreConfigurationBuilder.class)
    .location("/tmp/rocksdb/data")
    .expiredLocation("/tmp/rocksdb/expired")
        .properties(props)
    .build();
```

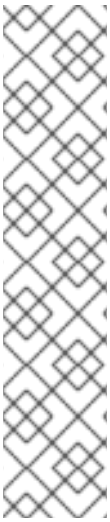
パラメーター	説明
location	RocksDB に使用するディレクトリーで、プライマリーキャッシュストアデータを格納します。ディレクトリーが終了しないと、そのディレクトリーが自動的に作成されます。
expiredLocation	RocksDB に使用するディレクトリーで、期限切れのデータを永続的にパージするのを保留します。ディレクトリーが終了しないと、そのディレクトリーが自動的に作成されます。
expiryQueueSize	期限切れの RocksDB ストアにフラッシュされる前に、期限切れエントリーを保持するインメモリーキューのサイズ
clearThreshold	RocksDB のすべてのエントリーを消去する方法は 2 つあります。1 つの方法は、すべてのエントリーを繰り返し処理し、各エントリーを個別に削除する方法です。もう 1 つの方法は、データベースを削除し、再作成することです。小規模なデータベースの場合は、個別のエントリーを削除する方が、後者による方法よりも速くなります。この設定では、後者の使用前に許可されるエントリーの最大数を設定します。

パラメーター	説明
compressionType	データ圧縮用の RocksDB の設定。オプションは「CompressionType enum」を参照してください。
blockSize	RocksDB の設定 - パフォーマンスチューニングに関する ドキュメント を参照してください。
cacheSize	RocksDB の設定 - パフォーマンスチューニングに関する ドキュメント を参照してください。

5.10.2.2. XML 設定例

infinispan.xml

```
<local-cache name="vehicleCache">
  <persistence>
    <rocksdb-store path="/tmp/rocksdb/data">
      <expiration path="/tmp/rocksdb/expired"/>
      <property name="database.max_background_compactions">2</property>
      <property name="data.write_buffer_size">512MB</property>
    </rocksdb-store>
  </persistence>
</local-cache>
```



注記

Red Hat Data Grid サーバーは、RocksDB に特有の設定プロパティーを追加すると、起動時に警告メッセージをログに記録します。

*たとえば、上記の例のように、設定に **data.write_buffer_size** を設定すると、以下のメッセージがログに書き込まれます。*

```
WARN  [org.infinispan.configuration.parsing.XmlConfigHelper] ISPN000292:
Unrecognized attribute 'data.write_buffer_size'. Please check your
configuration. Ignoring!
```

*Red Hat Data Grid は、ログメッセージで有効ではないことを示す RocksDB プロパティーを適用します。これらの **WARN** メッセージは無視できます。*

5.10.3. 追加の参考資料

動作中のコードサンプルについては、テストケース を参照してください。

設定例については、「テスト設定」を参照してください。

5.11. LEVELDB キャッシュストア

警告

LevelDB Cache Store は非推奨となり、RocksDB Cache Store に置き換えられました。LevelDB Cache Store に既存のデータが保存されると、RocksDB Cache Store はこれを最初の実行時に新しい SST ベースの形式に変換します。

5.12. JPA キャッシュストア

この実装は、エンティティーメタモデルにアクセスする JPA 2.0 仕様によって異なります。

通常のユースケースでは、JPA 秒レベルキャッシュやクエリーキャッシュに Red Hat Data Grid を利用することが推奨されます。ただし、Red Hat Data Grid API のみを使用し、一般的な形式（明確に定義されたスキーマを持つデータベースなど）を使用して Red Hat Data Grid をキャッシュストアに永続化したい場合は、JPA Cache Store が適切になる可能性があります。

留意すべき点

- JPA キャッシュストアを使用する場合、キーはエンティティーの ID で、値はエンティティーオブジェクトである必要があります。
- **@Id** または **@EmbeddedId** アノテーションがついたプロパティーのみが許可されます。
- 自動生成される ID はサポートされません。
- 最後に、すべてのエントリーは immortal エントリーとして保存されます。

5.12.1. 使用例

たとえば、永続ユニット "myPersistenceUnit" と JPA エンティティー User があるとします。

persistence.xml

```
<persistence-unit name="myPersistenceUnit">
 ...
</persistence-unit>
```

ユーザーエンティティークラス

User.java

```
@Entity
public class User implements Serializable {
 @Id
 private String username;
 private String firstName;
 private String lastName;

 ...
}
```

キャッシュ「usersCache」を設定して JPA キャッシュストアを使用するように設定できます。これにより、データをキャッシュに配置すると、JPA 設定を基にデータがデータベースに永続化されます。

```
EmbeddedCacheManager cacheManager = ...;


Configuration cacheConfig = new ConfigurationBuilder().persistence()
        .addStore(JpaStoreConfigurationBuilder.class)
        .persistenceUnitName("org.infinispan.loaders.jpa.configurationTest")
        .entityClass(User.class)
        .build();
cacheManager.defineCache("usersCache", cacheConfig);

Cache<String, User> usersCache = cacheManager.getCache("usersCache");
usersCache.put("raytsang", new User(...));
```

通常、単一の Red Hat Data Grid キャッシュは複数のタイプのキーと値のペアを保存できます。以下に例を示します。

```
Cache<String, User> usersCache = cacheManager.getCache("myCache");
usersCache.put("raytsang", new User());
Cache<Integer, Teacher> teachersCache = cacheManager.getCache("myCache");
teachersCache.put(1, new Teacher());
```

キャッシュが JPA キャッシュストアを使用するように設定されている場合、キャッシュは ONE タイプのデータのみを保存できることに注意してください。

```
Cache<String, User> usersCache = cacheManager.getCache("myJPACache"); // configured for User entity class
usersCache.put("raytsang", new User());
Cache<Integer, Teacher> teachersCache = cacheManager.getCache("myJPACache"); // cannot do this when this cache is configured to use a JPA cache store
teachersCache.put(1, new Teacher());
```

複合キーを使用できるように **@EmbeddedId** の使用がサポートされています。

```
@Entity
public class Vehicle implements Serializable {
 @EmbeddedId
 private VehicleId id;
 private String color; ...
}

@Embeddable
public class VehicleId implements Serializable
{
 private String state;
 private String licensePlate;
 ...
}
```

最後に、自動生成される ID （例：**@GeneratedValue**）はサポートされていません。JPA キャッシュストアを使用してキャッシュに何かを配置する場合、キーは ID の値である必要があります。

5.12.2. 設定

5.12.2.1. プログラミングの設定例

```
Configuration cacheConfig = new ConfigurationBuilder().persistence()
        .addStore(JpaStoreConfigurationBuilder.class)
        .persistenceUnitName("org.infinispan.loaders.jpa.configurationTest")
        .entityClass(User.class)
        .build();
```

パラメーター	説明
persistenceUnitName	JPA エンティティークラスが含まれる JPA 設定 NRPE(persistence.xml)の JPA 永続ユニット名
entityClass	このキャッシュに保存されることが予想される JPA エンティティークラス。1つのクラスのみが許可されます。

5.12.2.2. XML 設定例

```
<local-cache name="vehicleCache">
  <persistence passivation="false">
    <jpa-store xmlns="urn:infinispan:config:store:jpa:7.0"
      persistence-unit="org.infinispan.persistence.jpa.configurationTest"
      entity-class="org.infinispan.persistence.jpa.entity.Vehicle">
 />
  </persistence>
</local-cache>
```

パラメーター	説明
persistence-unit	JPA エンティティークラスが含まれる JPA 設定 NRPE(persistence.xml)の JPA 永続ユニット名
entity-class	このキャッシュに保存されることが予想される完全修飾 JPA エンティティークラス名。1つのクラスのみが許可されます。

5.12.3. 追加の参考資料

動作中のコードサンプルについては、テストケース を参照してください。

設定例については、「テスト設定」を参照してください。

5.13. カスタムキャッシュストア

提供されたキャッシュストアがすべての要件を満たすことができない場合は、独自のストアを実装することができます。独自のストアの作成に必要な手順は次のとおりです。

1. *以下のインターフェースのいずれかを実装して、カスタムストアを作成します。*
   - ***org.infinispan.persistence.spi.AdvancedCacheWriter***
   - ***org.infinispan.persistence.spi.AdvancedCacheLoader***
   - ***org.infinispan.persistence.spi.CacheLoader***
   - ***org.infinispan.persistence.spi.CacheWriter***
   - ***org.infinispan.persistence.spi.ExternalStore***
   - ***org.infinispan.persistence.spi.AdvancedLoadWriteStore***
   - ***org.infinispan.persistence.spi.TransactionalCacheWriter***
2. *ストアクラスに **@Store** アノテーションを付け、ストアに関連するプロパティーを指定します。たとえば、ストアを Replicated または Distributed mode: **@Store(shared = true)** で共有できます。*
3. *カスタムキャッシュストア設定およびビルダーを作成します。これには、**AbstractStoreConfiguration** および **AbstractStoreConfigurationBuilder** を拡張する必要があります。任意の手順として、以下のアノテーションを設定(**@ConfigurationFor,@BuiltBy**)に追加し、**@ConfiguredBy** をストア実装クラスに追加する必要があります。これらの追加のアノテーションにより、カスタム設定ビルダーが xml からストア設定を解析するために使用されます。これらのアノテーションが追加されない場合は、**CustomStoreConfigurationBuilder** を使用して **AbstractStoreConfiguration** Configuration で定義された共通のストア属性を解析し、追加の要素は無視されます。ストアとその設定が **@Store** と **@ConfigurationFor** アノテーションをそれぞれ宣言しない場合、キャッシュの初期時に警告メッセージがログに記録されます。*
4. *カスタムストアをキャッシュの設定に追加します。*
   a. *カスタムストアを ConfigurationBuilder に追加します。以下に例を示します。*

```
Configuration config = new ConfigurationBuilder()
        .persistence()
        .addStore(CustomStoreConfigurationBuilder.class)
        .build();
```

   b. *xml でカスタムストアを定義します。*

```
<local-cache name="customStoreExample">
  <persistence>
    <store class="org.infinispan.persistence.dummy.DummyInMemoryStore" />
  </persistence>
</local-cache>
```

5.13.1. hotrod デプロイメント

カスタムキャッシュストアは、以下の手順を使用して個別の JAR ファイルにパッケージ化でき、HotRod サーバーにデプロイできます。

1. *前のセクションでの カスタムキャッシュ ストア, 手順1-3> に従って、実装を JAR ファイルにパッケージ化します（または、カスタムのキャッシュストア Archetype を使用します）。*

2. Jar で **META-INF/services/** に適切なファイルを作成します。これには、ストア実装の完全修飾クラス名が含まれます。このサービスファイルの名前には、ストアが実装するインターフェースを反映させる必要があります。たとえば、ストアが **AdvancedCacheWriter** インターフェースを実装する場合は、以下のファイルを作成します。

   - **/META-INF/services/org.infinispan.persistence.spi.AdvancedCacheWriter**

3. JAR ファイルを Red Hat Data Grid Server にデプロイします。

5.14. MIGRATOR を保存する

Red Hat Data Grid 7.3 では、以前のバージョンの Red Hat Data Grid と後方互換性のない内部マーシャリング機能への変更が導入されました。そのため、Red Hat Data Grid 7.3.x 以降では、以前のバージョンの Red Hat Data Grid で作成したキャッシュストアを読み取ることができません。また、Red Hat Data Grid は JDBC Mixed や Binary ストアなどの一部のストア実装を提供しなくなりました。

StoreMigrator.java を使用してキャッシュストアを移行できます。この移行ツールは、以前のバージョンでキャッシュストアからデータを読み取り、現在のマーシャリング実装との互換性のためにコンテンツを書き直します。

5.14.1. キャッシュストアの移行

StoreMigrator で移行を実行するには、以下を実行します。

1. JDBC ドライバーなどのソースおよびターゲットデータベースの **infinispan-tools-9.4.0.jar** と依存関係をクラスパスに配置します。

2. ソースおよびターゲットキャッシュストアの設定プロパティーが含まれる **.properties** ファイルを作成します。
   migrator.properties に適用可能な設定オプションをすべて含むプロパティーファイルの例があります。

3. **.properties** ファイルを **StoreMigrator** の引数として指定します。

4. **mvn exec:java** を実行して migrator を実行します。

StoreMigrator の以下の Maven **pom.xml** の例を参照してください。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>org.infinispan.example</groupId>
   <artifactId>jdbc-migrator-example</artifactId>
   <version>1.0-SNAPSHOT</version>

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

```
    <!-- ADD YOUR REQUIRED DEPENDENCIES HERE -->
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.2.1</version>
        <executions>
          <execution>
            <goals>
              <goal>java</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <mainClass>StoreMigrator</mainClass>
          <arguments>
            <argument><!-- PATH TO YOUR MIGRATOR.PROPERTIES FILE --></argument>
          </arguments>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

${version.infinispan} *を Red Hat Data Grid の適切なバージョンに置き換えます。*

5.14.2. 移行プロパティーの保存

migrator プロパティーはすべて、ソースまたはターゲットストアのコンテキストで設定されます。各プロパティーは ***source.*** *または* ***target*** *のいずれかで開始する必要があります。*

以下のセクションのすべてのプロパティーは、バイナリーテーブルに移行することができないため、 ***table.binary.****プロパティーを除き、ソースストアとターゲットストアの両方に適用されます。*

5.14.2.1. 一般的なプロパティー

プロパティー	説明	値の例	必須
type	JDBC_STRING | JDBC_BINARY | JDBC_MIXED | LEVELDB | ROCKSDB | SINGLE_FILE_STORE | SOFT_INDEX_FILE_STORE	JDBC_MIXED	TRUE
cache_name	ストアに関連付けられたキャッシュの名前	persistentMixedCache	TRUE

5.14.2.2. JDBC プロパティー

プロパティー	説明	値の例	必須
dialect	基礎となるデータベースの方言	POSTGRES	TRUE
marshaller.type	ストアに使用するマーシャラー。以下の値が使用できます。 - **LEGACY** Red Hat Data Grid 7.2.x マーシャラー。ソースのストアにのみ有効です。 - **CURRENT** Red Hat Data Grid 7.3.x マーシャラー。 - **CUSTOM** カスタムマーシャラー。	CURRENT	TRUE
marshaller.class	type=CUSTOM の場合のマーシャラーのクラス	org.example.CustomMarshaller	
marshaller.externalizers	ロードするカスタムの AdvancedExternalizer 実装のコンマ区切りリスト **[id]:<Externalizer class>**	**25:Externalizer1,org.example.Externalizer2**	
connection_pool.connection_url	JDBC 接続 URL	**jdbc:postgresql:postgres**	TRUE
connection_pool.driver_class	JDBC ドライバーのクラス	org.postrgesql.Driver	TRUE
connection_pool.username	データベースのユーザー名		TRUE
connection_pool.password	データベースのパスワード		TRUE
db.major_version	データベースのメジャーバージョン	9	
db.minor_version	データベースのマイナーバージョン	5	
db.disable_upsert	db upsert を無効にします。	false	

プロパティー	説明	値の例	必須
db.disable_indexing	テーブルインデックスが作成されない	false	
table.&**lt;binary|string&** gt;.table_name_prefix	テーブル名の追加接頭辞	tablePrefix	
表。&**lt;binary|string>.<id|data|timestamp>**.name	列の名前	id_column	TRUE
表。&**lt;binary|string>.<id|data|timestamp>**.type	列のタイプ	VARCHAR	TRUE
key_to_string_mapper	TwoWayKey2StringMapper クラス	**org.infinispan.persistence.keymappers.DefaultTwoWayKey2StringMapper**	

5.14.2.3. leveldb/RocksDB プロパティー

プロパティー	説明	値の例	必須
location	db ディレクトリーの場所	/some/example/dir	TRUE
圧縮	使用する圧縮タイプ	SNAPPY	

5.14.2.4. SingleFileStore プロパティー

プロパティー	説明	値の例	必須
location	ストアの .dat ファイルを含むディレクトリー	/some/example/dir	TRUE

5.14.2.5. SoftIndexFileStore プロパティー

プロパティー	説明	値の例	必須

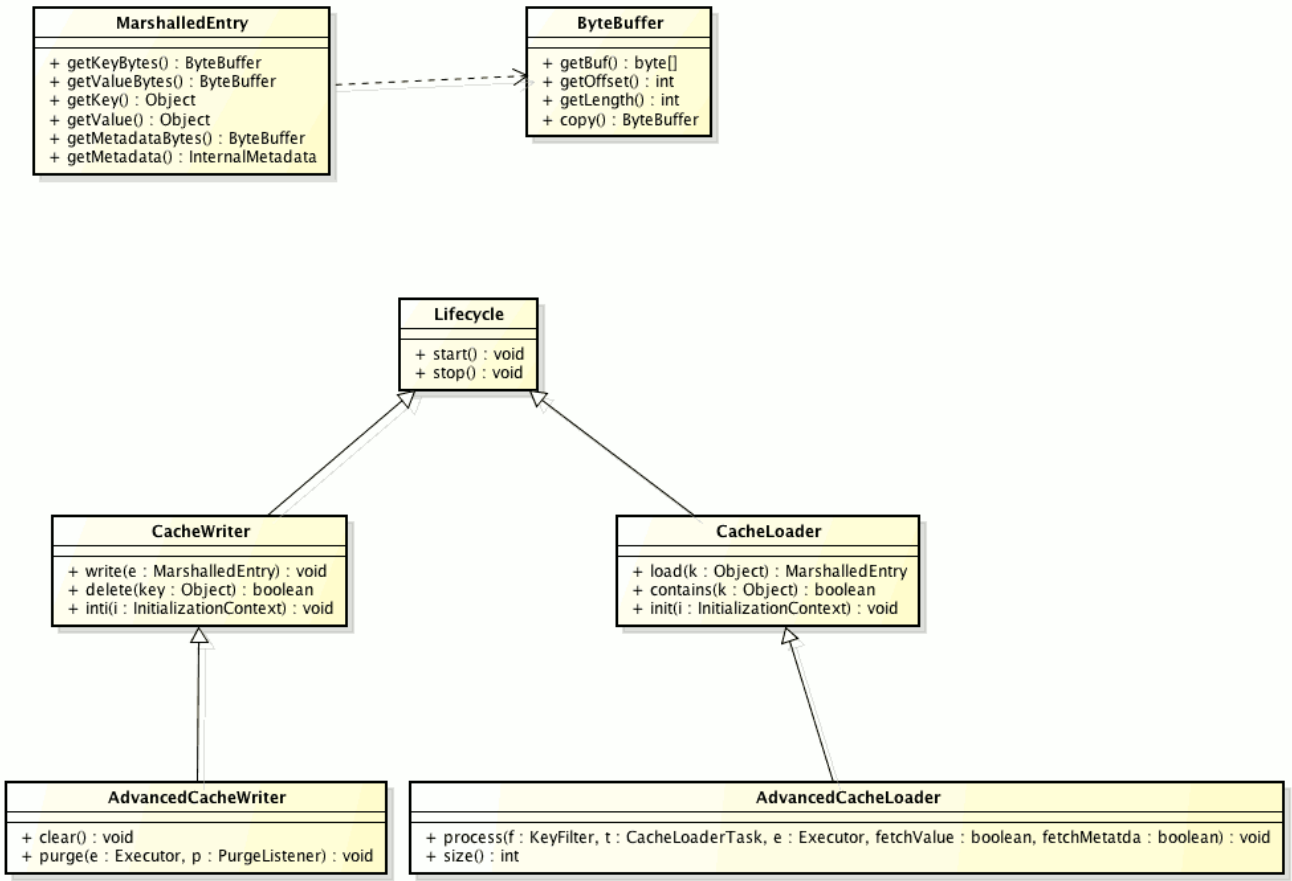
プロパティー	説明	値の例	必須
location	db ディレクトリーの場所	/some/example/dir	TRUE
index_location	db インデックスの場所	/some/example/dir-index	ターゲットのみ

以下のセクションではSPI について説明し、Red Hat Data Grid が追加設定なしで提供されるSPI 実装についても説明します。

5.15. SPI

以下のクラス図は、永続API の主なSPI インターフェースを示しています。

図5.1 永続性 SPI



クラスに関する注意事項

- *ByteBuffer: オブジェクトのシリアライズされた形式を抽象化します。*
- *MarshalledEntry: キャッシュに追加されたキー値に対応する永続ストア内に保持される情報を抽象化します。シリアライズされた（バイトバッファー）およびデシリアライズ(Object)形式の両方でこの情報を読み取るメソッドを提供します。通常、ストアから読み取られたデータはシリアライズされた形式で保持され、MarishalledEntry 実装内にオンデマンドでデシリアライズされます。*

- *CacheWriter および CacheLoader は、ストアの読み書きを行うための基本的な方法を提供します。*
- *AdvancedCacheLoader および AdvancedCacheWriter は、有効期限の切れたエントリーの並行反復とパージ、削除およびサイズなど、ストレージを一括して操作する操作を提供します。*

プロバイダーは、以下のインターフェースのサブセットのみを実装するように選択することができます。

- *AdvancedCacheWriter を実装しないと、指定されたライターが期限切れのエントリーをパージしたり、消去したりすることができません。*
- *ローダーが AdvancedCacheLoader インクラインを実装していない場合は、プリロードやキャッシュの反復に参加しません（ストリーム操作に必要）。*

既存のストアを新しい API に移行したり、新しいストア実装を作成したりする場合は、SingleFileStore が開始点/サンプルになっている可能性があります。

5.16. その他の実装

他にも多くのキャッシュローダーとキャッシュストアの実装が存在します。詳細は、このWeb サイトを参照してください。

第6章 クラスタリング

キャッシュマネージャーは、ローカル（スタンドアロン）またはクラスター化のいずれかに設定できます。クラスター化すると、マネージャーインスタンスは JGroups の検出プロトコルを使用して、同じローカルネットワーク上の近傍インスタンスを自動的に検出し、クラスターを形成します。

ローカルのみのキャッシュマネージャーの作成は簡単なです。no-argument **DefaultCacheManager** コンストラクターを使用するか、以下の XML 設定ファイルを指定するだけです。

```
<infinispan/>
```

クラスターキャッシュマネージャーを起動するには、クラスター構成を作成する必要があります。

```
GlobalConfigurationBuilder gcb = GlobalConfigurationBuilder.defaultClusteredBuilder();
DefaultCacheManager manager = new DefaultCacheManager(gcb.build());
```

```
<infinispan>
  <cache-container>
    <transport/>
  </cache-container>
</infinispan>
```

個別のキャッシュは、異なるモードで設定できます。

- **ローカル**: 変更と読み取りは複製されません。これは、クラスター化されていないキャッシュマネージャーで使用できる唯一のモードです。
- **Invalidation**: 変更はレプリケートされません。代わりに、キーがすべてのノードで無効にされます。読み取りはローカルになります。
- **replicated**: 変更はすべてのノードにレプリケートされ、読み取りは常にローカルになります。
- **Distributed** （分散）: 変更が固定された数のノードに複製され、最低でも 1 つの所有者ノードから値を読み取ります。

6.1. ローカルモード

Red Hat Data Grid はクラスター化モードで特に興味がありますが、非常に多くのローカルモードも提供します。このモードでは、**ConcurrentHashMap** と同様の単純なインメモリーデータキャッシュとして機能します。

ただし、マップではなくローカルキャッシュを使用するのはなぜでしょうか。キャッシュは、単純なマップよりも多くの機能を提供します。これには、永続ストアへのライトスルーおよびライトビハインド、エントリーのエビクション、メモリー不足や有効期限の発生を防ぐことができます。

Red Hat Data Grid の **Cache** インターフェースは、JDK の **ConcurrentMap** ConcurrentMap で Red Hat Data Grid trivial への移行を拡張します。

Red Hat Data Grid キャッシュは、既存のトランザクションマネージャーと統合するか、別のトランザクションマネージャーを実行するか、トランザクションもサポートしています。ローカルキャッシュトランザクションには、2 つの選択肢があります。

1. ロックのタイミング。**悲観的ロック**は、書き込み操作時、またはユーザーが**AdvancedCache.lock(keys)**を明示的に呼び出したときにキーをロックします。**楽観的ロッ**

*クは、トランザクションのコミット中にのみキーをロックし、代わりに、現在のトランザクションがキーを読み取った後に別のトランザクションが同じキーを変更した場合は、コミット時に **WriteSkewCheckException** をスローします。*

2. *分離レベルread-committed および repeatable read をサポートします。*

6.1.1. 簡易キャッシュ

従来のローカルキャッシュは、クラスター化されたキャッシュと同じアーキテクチャーを使用します。つまり、インターセプタースタックを使用します。これにより、多くの実装を再利用できます。ただし、高度な機能が不要でパフォーマンスがより重要な場合は、インターセプタースタックを削除して、単純なキャッシュを使用できます。

そのため、どの機能も削除されますか。設定の観点からは、簡単なキャッシュは以下に対応していません。

- *トランザクションと呼び出しバッチ処理*
- *永続性（キャッシュストアおよびローダー）*
- *カスタムインターセプター（インターセプタースタックなし）*
- *インデックス化*
- *互換性（組み込み/サーバーモード）*
- *バイナリーとして保存（ローカルキャッシュに非常に便利です）*

API パースペクティブから、これらの機能は例外をスローします。

- *カスタムインターセプターの追加*
- *分散済みエグゼキューターフレームワーク*

そして、何が残っていますか。

- *基本的なマップのようなAPI*
- *キャッシュリスナー（ローカルリスナー）*
- *有効期限*
- *eviction*
- *security*
- *JMX アクセス*
- *統計（ただし、最大のパフォーマンスを得るには、statistics-available=false を使用してこれをオフにすることが推奨されます）*

6.1.1.1. 宣言型設定

```
<local-cache name="mySimpleCache" simple-cache="true">
    <!-- expiration, eviction, security... -->
</local-cache>
```

6.1.1.2. プログラムによる設定

```
CacheManager cm = getCacheManager();
ConfigurationBuilder builder = new ConfigurationBuilder().simpleCache(true);
cm.defineConfiguration("mySimpleCache", builder.build());
Cache cache = cm.getCache("mySimpleCache");
```

サポートされていない機能に対する簡単なキャッシュチェック。たとえばトランザクションを使用するよう設定すると、設定検証によって例外が出力されます。

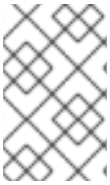
6.2. インバリデーションモード

インバリデーションでは、異なるノードのキャッシュは実際にはどのデータも共有しません。代わりに、キーが書き込まれると、キャッシュは他のノードから古くなったデータを削除するだけです。このキャッシュモードは、データベースなどのデータに対して別の永続ストアがあり、読み込みシステムに Red Hat Data Grid の最適化として使用し、読み取りごとにデータベースに到達しないようにします。キャッシュがインバリデーション用に設定されている場合は、データをキャッシュに変更するたびに、クラスター内の他のキャッシュは、データが古いため、メモリーおよびローカルストアから削除される必要があることを通知するメッセージを受信します。

図6.1 インバリデーションモード

アプリケーションは外部ストアから値を読み取り、他のノードから削除せずにローカルキャッシュに書き込む場合があります。これを実行するには、**Cache.put(key, value)** の代わりに **Cache.putForExternalRead(key, value)** を呼び出す必要があります。

インバリデーションモードは、共有キャッシュストアと使用できます。書き込み操作は、共有ストアを更新し、他のノードメモリーから古い値を削除します。これには2つの利点があります。値全体を複製する場合に比べてインバリデーションメッセージが非常に小さいため、ネットワークトラフィックが最小限に抑えられます。また、クラスター内の他のキャッシュは、必要な場合にのみ、変更されたデータを遅延的に検索します。



注記

ローカル ストアでインバリデーションモードを使用しないでください。インバリデーションメッセージはローカルストアのエントリーを削除せず、一部のノードが古い値を認識します。

インバリデーションキャッシュは、特別なキャッシュローダー(**ClusterLoader**) で設定することもできます。**ClusterLoader**が有効になっている場合、ローカルノードでキーが見つからない読み取り操作は、最初に他のすべてのノードからキーを要求し、ローカルのメモリに保存します。特定の状況では古い値を保存するため、古くなった値の耐性がある場合にのみ使用します。

インバリデーションモードは、同期または非同期です。同期すると、クラスター内のすべてのノードが古い値をエビクトするまで、書き込みがブロックされます。非同期の場合、発信者はインバリデーションメッセージをブロードキャストしますが、応答を待ちません。つまり、発信者で書き込みが完了した後も、他のノードはしばらくの間古い値を確認します。

トランザクションはインバリデーションメッセージをバッチするために使用できます。トランザクションはプライマリ所有者でキーロックを取得します。プライマリー所有者の割り当て方法の詳細は、Key Ownership セクションを参照してください。

- 悲観的ロックでは、各書き込みは、すべてのノードにブロードキャストされるロックメッセージをトリガーします。トランザクションのコミット中に、発信者は、影響を受けるすべてのキーを無効にし、ロックを解放する1フェーズの準備メッセージ（任意で fire-and-forget）をブロードキャストします。

- 楽観的ロックを使用すると、発信者は準備メッセージ、コミットメッセージ、およびロック解除メッセージ（任意）をブロードキャストします。1 フェーズの準備またはロック解除メッセージのいずれかが fire-and-forget であり、最後のメッセージは常にロックを解放します。

6.3. レプリケートモード

任意のノードでレプリケートされたキャッシュに書き込まれたエントリーは、クラスター内の他のすべてのノードに複製され、任意のノードからローカルで取得できます。レプリケートモードは、クラスタ間で状態を共有するための迅速で簡単な方法を提供しますが、書き込みに必要なメッセージの数がクラスタサイズに比例してスケーリングするため、レプリケーションは実際には小さなクラスタ（10ノード未満）でのみ適切に実行します。Red Hat Data Grid は UDP マルチキャストを使用するよう設定できるため、この問題をある程度軽減できます。

各キーにはプライマリ所有者があり、一貫性を提供するためにデータコンテナの更新をシリアル化します。プライマリー所有者の割り当て方法の詳細は、Key Ownership セクションを参照してください。

図6.2 レプリケートモード

レプリケートモードは、同期または非同期にすることができます。

- 同期レプリケーションは、変更がクラスターのすべてのノードに正常に複製されるまで、呼び出し元（**cache.put(key, value)**など）をブロックします。

- 非同期レプリケーションはバックグラウンドでレプリケーションを実行し、書き込み操作が即座に返されます。非同期レプリケーションは推奨されません。これは、通信エラーやリモートノードで発生したエラーは呼び出し元に報告されないためです。

トランザクションが有効になっていると、書き込み操作はプライマリー所有者によって複製されません。

- 悲観的ロックでは、各書き込みは、すべてのノードにブロードキャストされるロックメッセー

ジをトリガーします。トランザクションのコミット時に、送信元は1 フェーズの準備メッセージとロック解除メッセージ（任意）をブロードキャストします。1 フェーズの準備またはロック解除メッセージのいずれかが fire-and-forget になります。

- 楽観的ロックを使用すると、発信者は準備メッセージ、コミットメッセージ、およびロック解除メッセージ（任意）をブロードキャストします。ここで、1 フェーズの準備またはロック解除メッセージが fire-and-forget になります。

6.4. ディストリビューションモード

分散は、**numOwners** として設定された、キャッシュ内の任意のエントリの固定数のコピーを保持しようとします。これにより、キャッシュを線形にスケーリングし、ノードがクラスターに追加されるにつれて、より多くのデータを格納できます。

ノードがクラスターに参加およびクラスターから離脱すると、キーのコピー数が**numOwners**より多い場合と少ない場合があります。特に、**numOwners**ノードがすぐに連続して離れると、一部のエントリが失われるため、分散キャッシュは、**numOwners - 1**ノードの障害を許容すると言われます。

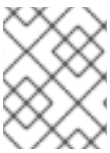
コピー数は、パフォーマンスとデータの持続性を示すトレードオフを表します。維持するコピーが増えると、パフォーマンスは低くなりますが、サーバーやネットワークの障害によるデータ喪失のリスクも低くなります。保持するコピー数にかかわらず、ディストリビューションは直線的に増加します。これは、Red Hat Data Grid のスケーラビリティーに鍵となります。

キーの所有者は、キーへの書き込みを調整する1つの**プライマリ所有者**と、0個以上の**バックアップ所有者**に分割されます。プライマリ所有者とバックアップ所有者の割り当て方法の詳細は、キー所有権セクションを参照してください。

図6.3 分散モード

読み取り操作はプライマリー所有者からの値を要求しますが、妥当な時間内に応答しない場合、バックアップの所有者からも値を要求します。（**infinispan.stagger.delay** システムプロパティー）は、リクエスト間の遅延を制御します。） キーがローカルキャッシュに存在する場合、読み取り操作には**0**メッセージが必要になる場合があり、すべての所有者が遅い場合は最大 **2 * numOwners** メッセージが必要になる場合があります。

また、書き込み操作は、最大 **2 * numOwners** メッセージで、送信元からプライマリー所有者、**numOwners - 1** メッセージ、プライマリーからバックアップへの1メッセージ、対応する ACK メッセージまでです。



注記

キャッシュトポロジの変更により、読み取りと書き込みの両方で再試行と追加メッセージが発生する可能性があります。

レプリケートモードと同様に、分散モードは同期または非同期にすることもできます。レプリケートされたモードでは、更新が失われる可能性があるため、非同期レプリケーションは推奨されません。更新の損失に加えて、非同期の分散キャッシュは、スレッドがキーに書き込むときに古い値を確認し、その後に同じキーをすぐに読み取ることもできます。

トランザクション分散キャッシュは、トランザクション複製キャッシュと同じ種類のメッセージを使用しますが、ロック/準備/コミット/ロック解除メッセージは、クラスター内のすべてのノードにブロードキャストされるのではなく、**影響を受けるノード**（トランザクションの影響を受ける少なくとも1つのキーを所有するすべてのノード）にのみ送信されます。最適化として、トランザクションが単一のキーに書き込み、送信元がキーの主な所有者である場合、ロックメッセージは複製されません。

6.4.1. 読み取りの一貫性

同期レプリケーションを使用しても、分散キャッシュは線形化できません。（トランザクションキャッシュの場合は、シリアル化/スナップショットの分離に対応していないとします。）1つのスレッドで1つの put を実行できます。

```
cache.get(k) -> v1
cache.put(k, v2)
cache.get(k) -> v2
```

ただし、別のスレッドでは、異なる順序で値が表示される場合があります。

```
cache.get(k) -> v2
cache.get(k) -> v1
```

その理由は、プライマリ所有者が応答する速度に応じて、クラスター内のすべてのノードが**任意の** 所有者から値を返す可能性があるためです。書き込みはすべての所有者にわたってアトミックではありません。実際、プライマリはバックアップから確認を受け取った後にのみ更新をコミットします。プライマリがバックアップからの確認メッセージを待機している間、バックアップからの読み取りには新しい値が表示されますが、プライマリからの読み取りには古い値が表示されます。

6.4.2. キーの所有者

分散キャッシュは、エントリーを固定数のセグメントに分割し、各セグメントを所有者ノードの一覧に割り当てます。レプリケートされたキャッシュは同じで、すべてのノードが所有者である場合を除きます。

所有者リストの最初のノードは**プライマリ所有者**です。一覧のその他のノードは**バックアップの所有者**です。キャッシュトポロジが変更すると、ノードがクラスターに参加またはクラスターから離脱するため、セグメント所有権テーブルがすべてのノードにブロードキャストされます。これにより、ノードはマルチキャスト要求を行ったり、各キーのメタデータを維持したりすることなく、キーを見つけることができます。

numSegments プロパティーでは、利用可能なセグメントの数を設定します。ただし、クラスターが再起動しない限り、セグメントの数は変更できません。

同様に、キーからセグメントのマッピングは変更できません。鍵は、クラスタートポロジーの変更に関係なく、常に同じセグメントにマップする必要があります。キーからセグメントのマッピングは、クラスタートポロジーの変更時に移動する必要のあるセグメント数を最小限に抑える一方で、各ノードに割り当てられたセグメント数を均等に分散することが重要になります。

KeyPartitioner を設定するか、Grouping API を使用してキー間のマッピングをカスタマイズできます。

ただし、Red Hat Data Grid は以下の実装を提供します。

SyncConsistentHashFactory

一貫性のあるハッシュ に基づくアルゴリズムを使用します。サーバーヒントを無効にした場合は、デフォルトで選択されています。
この実装では、クラスターが対称である限り、すべてのキャッシュの同じノードに常にキーが割り当てられます。つまり、すべてのキャッシュがすべてのノードで実行します。この実装には、負荷の分散が若干不均等であるため、負のポイントが若干異なります。また、参加または脱退時に厳密に必要な数よりも多くのセグメントを移動します。

TopologyAwareSyncConsistentHashFactory

***SyncConsistentHashFactory**に似ていますが、サーバーヒントに適合しています。サーバーヒントが有効な場合にデフォルトで選択されます。*

DefaultConsistentHashFactory

***SyncConsistentHashFactory** よりも均等に分散を行いますが、1 つの欠点があります。ノードがクラスターに参加する順序によって、どのノードがどのセグメントを所有するかが決まります。その結果、キーは異なるキャッシュ内の異なるノードに割り当てられる可能性があります。*
サーバーヒントを無効にした状態で、バージョン5.2 からバージョン8.1 のデフォルトでした。

TopologyAwareConsistentHashFactory

DefaultConsistentHashFactory*に似ていますが、サーバーヒントに適合しています。*
サーバーヒントが有効なバージョン5.2 から 8.1 へのデフォルトでした。

ReplicatedConsistentHashFactory

レプリケートされたキャッシュの実装に内部で使用されます。このアルゴリズムは分散キャッシュで明示的に選択しないでください。

6.4.2.1. 容量ファクト

設備利用率は、ノードで使用可能なリソースに基づいてセグメントからノードへのマッピングを割り当てます。

容量要因を設定するには、負の数を指定し、Red Hat Data Grid ハッシュアルゴリズムは各ノードに容量係数（プライマリー所有者およびバックアップ所有者の両方）で重みを割り当てます。

*たとえば、nodeA は同じ Red Hat Data Grid クラスターの nodeB よりも 2x のメモリーを持ちます。この場合、**capacityFactor** を **2** に設定すると、Red Hat Data Grid はセグメント数を nodeA に割り当てるように設定します。*

*容量係数を**0**に設定することは可能ですが、ノードが有用なデータ所有者として十分な長さでクラスターに参加していない場合にのみ推奨されます。*

6.4.2.2. ゼロ容量ノード

*各キャッシュ、ユーザー定義キャッシュ、および内部キャッシュに対して容量係数が **0** であるノード全体の設定が必要になる場合があります。ゼロの容量ノードを定義する場合、ノードはデータを保持しません。これは、ゼロ容量ノードを宣言します。*

```
<cache-container zero-capacity-node="true" />
```

```
new GlobalConfigurationBuilder().zeroCapacityNode(true);
```

ただし、これは分散キャッシュにのみ適用されることに注意してください。レプリケートされたキャッシュを使用している場合、ノードは値のコピーを保持します。この機能を最適なものにするには、分散キャッシュのみを使用します。

6.4.2.3. ハッシュ設定

これは、XML を使用して、ハッシュを宣言的に設定する方法です。

```
<distributed-cache name="distributedCache" owners="2" segments="100" capacity-factor="2" />
```

Java では、プログラムを用いてこの方法で設定できます。

```
Configuration c = new ConfigurationBuilder()
  .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .hash()
      .numOwners(2)
      .numSegments(100)
      .capacityFactor(2)
  .build();
```

6.4.3. 初期クラスターサイズ

Red Hat Data Grid は、トポロジーの変更（実行時に追加/削除されたノードなど）を処理するための非常に動的な特性です。つまり、通常ノードは他のノードが存在するのを待たずに開始してください。これは非常に柔軟性がありますが、キャッシュの開始前に、特定の数のノードがクラスターに参加する必要があるアプリケーションには適切ではない場合があります。このため、キャッシュの初期化に進む前に、クラスターに参加するノードの数を指定できます。これには、**initialClusterSize** および **initialClusterTimeout** トランスポートプロパティーを使用します。宣言型 XML 設定:

```
<transport initial-cluster-size="4" initial-cluster-timeout="30000" />
```

プログラムによる Java 設定:

```
GlobalConfiguration global = new GlobalConfigurationBuilder()
  .transport()
     .initialClusterSize(4)
     .initialClusterTimeout(30000)
  .build();
```

上記の設定は、初期化前に 4 つのノードがクラスターに参加するのを待機します。指定されたタイムアウト内に最初のノードが表示されない場合は、キャッシュマネージャーが起動に失敗します。

6.4.4. L1 キャッシュ

L1が有効になっている場合、ノードはリモート読み取りの結果を短期間（設定可能、デフォルトでは10分）ローカルに保持し、ルックアップを繰り返すと、所有者に再度尋ねるのではなく、ローカルのL1値が返されます。

図6.4 L1 キャッシュ

L1 キャッシュは無料ではありません。有効にするとコストがかかり、このコストは、すべてのエントリー更新でインバリデーションメッセージをすべてのノードにブロードキャストする必要があることです。L1 エントリーは、キャッシュが最大サイズで設定されている場合に他のエントリーと同様にエビクトできます。L1を有効にすると、非ローカルキーの繰り返される読み取りのパフォーマンスが向上しますが、書き込みが遅くなり、メモリ消費量がある程度増加します。

L1 キャッシュが正しいか。正しいアプローチとして、L1 を有効にしない状態でアプリケーションをベンチマークし、アクセスパターンに最も適した動作を確認できます。

6.4.5. サーバーヒント

以下のトポロジーヒントを指定できます。

マシン

これは、複数のJVMインスタンスが同じノードで実行されている場合、または複数の仮想マシンが同じ物理マシンで実行している場合でも、おそらく最も便利です。

ラック

大規模なクラスターでは、同じラックにあるノードでハードウェアまたはネットワークの障害が同時に発生する可能性が高くなります。

サイト

一部のクラスターでは、復元力を高めるために、複数の物理的な場所にノードが存在する場合があります。サイト間のレプリケーション は、2 つ以上のデータセンターにまたがるクラスターの代替手段であることに注意してください。

上記のすべては任意です。指定した場合、ディストリビューションアルゴリズムは、できるだけ多くのサイト、ラック、およびマシン全体に各セグメントの所有権を分散しようとします。

6.4.5.1. 設定

ヒントは、トランスポートレベルで設定されます。

```
<transport
   cluster="MyCluster"
   machine="LinuxServer01"
   rack="Rack01"
   site="US-WestCoast" />
```

6.4.6. キーアフィニティーサービス

分散キャッシュでは、不透明なアルゴリズムを使用してノードのリストにキーが割り当てられます。計

算を逆にし、特定のノードにマップする鍵を生成する簡単な方法はありません。ただし、一連の（疑似）ランダムキーを生成し、それらのプライマリ所有者が何であるかを確認し、特定のノードへのキーマッピングが必要なときにアプリケーションに渡すことができます。

6.4.6.1. API

以下のコードスニペットは、このサービスへの参照を取得し、使用する方法を示しています。

```
// 1. Obtain a reference to a cache
Cache cache = ...
Address address = cache.getCacheManager().getAddress();

// 2. Create the affinity service
KeyAffinityService keyAffinityService = KeyAffinityServiceFactory.newLocalKeyAffinityService(
      cache,
      new RndKeyGenerator(),
      Executors.newSingleThreadExecutor(),
      100);

// 3. Obtain a key for which the local node is the primary owner
Object localKey = keyAffinityService.getKeyForAddress(address);

// 4. Insert the key in the cache
cache.put(localKey, "yourValue");
```

サービスはステップ2で開始します。この時点以降、サービスは提供された**エグゼキューター**を使用してキーを生成してキューに入れます。ステップ3では、サービスから鍵を取得し、手順4ではそれを使用します。

6.4.6.2. ライフサイクル

KeyAffinityService は ***ライフサイクル*** を拡張し、停止と(再)起動を可能にします。

```
public interface Lifecycle {
   void start();
   void stop();
}
```

サービスは ***KeyAffinityServiceFactory*** でインスタンス化されます。ファクトリーメソッドはすべて ***Executor*** パラメーターを持ち、これは非同期キー生成に使用されます（呼び出し元のスレッドでは処理されません）。ユーザーは、この ***Executor*** のシャットダウンを処理します。

KeyAffinityService が起動したら、明示的に停止する必要があります。これにより、バックグラウンドキーの生成が停止し、保持されている他のリソースが解放されます。

KeyAffinityService がそれ自体で停止する唯一の状況は、登録済みのキャッシュマネージャーがシャットダウンした時です。

6.4.6.3. トポロジーの変更

キャッシュトポロジが変更すると（つまり、ノードがクラスターに参加またはクラスターから離脱する）、***KeyAffinityService***によって生成されたキーの所有権が変更される可能性があります。主なアフィニティーサービスはこれらのトポロジーの変更を追跡し、現在別のノードにマップされるキーを返しませんが、先に生成したキーに関しては何も実行しません。

そのため、アプリケーションは **KeyAffinityService** を純粋に最適化として処理し、正確性のために生成されたキーの場所に依存しないようにしてください。

特に、アプリケーションは、同じアドレスが常に一緒に配置されるように、**KeyAffinityService**によって生成されたキーに依存するべきではありません。キーのコロケーションは、Grouping API によってのみ提供されます。

6.4.7. Grouping API

Key affinity サービス および AtomicMap と同様に補完される API を使用すると、実際のノードを選択できなくても、同じノード上のエントリーのグループを共存させることができます。

6.4.7.1. 仕組み

デフォルトでは、キーのセグメントはキーの**hashCode()**を使用して計算されます。グループ化 API を使用する場合、Red Hat Data Grid はグループのセグメントを計算し、キーのセグメントとして使用します。セグメントがノードにマップされる方法についての詳細は、キーの所有権セクションを参照してください。

Grouping API を使用している場合は、他のノードに問い合わせることなく、すべてのノードが全キーの所有者を計算できることが重要です。このため、グループは手動で指定できません。グループは、エントリに固有（キークラスによって生成される）または外部（外部関数によって生成される）のいずれかです。

6.4.7.2. Grouping API を使用する方法

まず、グループを有効にする必要があります。Red Hat Data Grid をプログラムで設定する場合は、以下を呼び出します。

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

または、XML を使用している場合は、以下を行います。

```
<distributed-cache>
   <groups enabled="true"/>
</distributed-cache>
```

キークラスを制御できる場合（クラス定義を変更できるが、変更不可能なライブラリの一部ではない）、組み込みグループを使用することをお勧めします。intrinsic グループは、**@Group** アノテーションをメソッドに追加して指定します。以下に例を示します。

```
class User {
   ...
   String office;
   ...

   public 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 owners
   @Group
   public String getOffice() {
      return office;
   }
   }
}
```

注記

group メソッドは**String**を返す必要があります。

キークラスを制御できない場合、またはグループの決定がキークラスと直交する懸念事項である場合は、外部グループを使用することをお勧めします。外部グループは、**Grouper**インターフェイスを実装することによって指定されます。

```
public interface Grouper<T> {
   String computeGroup(T key, String group);

   Class<T> getKeyType();
}
```

同じキータイプに対して複数の**Grouper**クラスが構成されている場合は、それらすべてが呼び出され、前のクラスで計算された値を受け取ります。キークラスにも **@Group** アノテーションがある場合、最初の **Grouper** はアノテーション付きのメソッドによって計算されたグループを受信します。これにより、組み込みグループを使用するときに、グループをさらに細かく制御できます。**Grouper** 実装の例を見てみましょう。

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

   // 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 "0" or group "1".
   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;
   }
}
```

Grouper 実装は、キャッシュ設定で明示的に登録する必要があります。Red Hat Data Grid をプログラムで設定している場合は、以下を行います。

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

または、XML を使用している場合は、以下を行います。

```
<distributed-cache>
   <groups enabled="true">
      <grouper class="com.acme.KXGrouper" />
   </groups>
</distributed-cache>
```

6.4.7.3. 高度なインターフェース

AdvancedCache には、グループ固有のメソッドが2 つあります。

`getGroup(groupName)`
: グループに属するキャッシュ内のすべてのキーを取得します。

`removeGroup(groupName)`
: グループに属するキャッシュ内のすべてのキーを削除します。

どちらのメソッドもデータコンテナー全体とストア（存在する場合）を繰り返し処理するため、キャッシュに多くの小規模なグループが含まれる場合に処理が遅くなる可能性があります。

6.5. 非同期オプション

6.5.1. 非同期通信

クラスター化されたキャッシュモードはすべて、< **replicated-cache/>、< distributed -cache>、または < invalidation-cache**/ > 要素の **mode="ASYNC"** 属性と非同期通信を使用するように設定できます。

非同期通信では、送信元ノードは操作のステータスについて他のノードから確認応答を受け取ることはありません。そのため、他のノードで成功したかどうかを確認する方法はありません。

非同期通信はデータに不整合を引き起こす可能性があり、結果を推論するのが難しいため、一般的に非同期通信はお勧めしません。ただし、速度が一貫性よりも重要であり、このようなケースでオプションが利用できる場合があります。

6.5.2. Asynchronous API

非同期 API を使用すると、ユーザースレッドをブロックしなくても同期通信を使用できます。

注意点が1つあります。非同期操作はプログラムの順序を保持しません。スレッドが **cache.putAsync(k, v1); cache.putAsync(k, v2)** を呼び出す場合の **k** の最終的な値は **v1** または **v2** のいずれかになります。非同期通信を使用する場合の利点は、最終的な値をあるノードで**v1**にして別のノードで**v2**にすることができないことです。

注記

バージョン9.0 よりも前のバージョンでは、非同期API は内部スレッドプールからスレッドを作成し、そのスレッドでブロッキング操作を実行してエミュレートされました。

6.5.3. 戻り値

***Cache** インターフェースは **java.util.Map** を拡張するため、**put(key, value)** や **remove(key)** などの書き込みメソッドはデフォルトで以前の値を返します。*

戻り値が正しくないことがあります。

1. ***Flag.IGNORE_RETURN_VALUE**、**Flag.SKIP_REMOTE_LOOKUP**、または **Flag.SKIP_CACHE_LOAD** で **AdvancedCache.withFlags()** を使用する場合。*
2. *キャッシュに **unreliable-return-values="true"** が設定されている場合。*
3. *非同期通信を使用する場合。*
4. *同じキーへの同時書き込みが複数あり、キャッシュトポロジが変更された場合。トポロジーの変更により、Red Hat Data Grid は書き込み操作を再試行し、再試行された操作の戻り値は信頼性がありません。*

トランザクションキャッシュは、3 と 4 の場合、正しい以前の値を返します。しかし、トランザクションキャッシュには gotcha: in distributed モードもあり、read-committed 分離レベルは、繰り返し可能な読み取りとして実装されます。つまり、この"double-checked locking"例は機能しません。

```
Cache cache = ...
TransactionManager tm = ...

tm.begin();
try {
  Integer v1 = cache.get(k);
  // Increment the value
  Integer v2 = cache.put(k, v1 + 1);
  if (Objects.equals(v1, v2) {
    // success
  } else {
    // retry
  }
} finally {
  tm.commit();
}
```

*これを実装する適切な方法として、**cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(k)** を使用します。*

最適化されたロックを持つキャッシュでは、書き込みは古い以前の値を返すことができます。書き込み skew チェックでは、古い値を回避できます。詳細は、「Write Skews」を参照してください。

6.6. パーティション処理

Red Hat Data Grid クラスターは、データが保存される多数のノードから構築されます。ノード障害の発生時にデータが失われないように、Red Hat Data Grid は Red Hat Data Grid parlance

*PROVISIONING- で複数のノードに同じ data xmvn-gitopscache エントリーをコピーします。このレベルのデータ冗長性は **numOwners** 設定属性で設定され、**numOwners** ノードが同時にクラッシュした限り、Red Hat Data Grid には利用可能なデータのコピーがあります。*

*ただし、**numOwners**を超えるノードがクラスターから消えるという壊滅的な状況が発生する可能性があります。*

スプリットブレイン

たとえば、ルーターのクラッシュにより、クラスターは 2 つ以上のパーティションに分割されるか、または個別に動作するサブクラスターに分割されます。このような場合、異なるパーティションから読み取り/書き込みを行う複数のクライアントが、同じキャッシュエントリーの異なるバージョンを参照し、多くのアプリケーションで問題があります。冗長ネットワークやIP ボンディングなど、スプリットブレインが発生する可能性を軽減する方法があることに注意してください。これらは、問題の発生期間のみを縮小します。

numOwners ノードを順番にクラッシュ

***numOwners** ノードが連続してクラッシュし、Red Hat Data Grid にクラッシュ間の状態を適切にリバランスする時間がない場合、結果は部分的なデータが失われることになります。*

このセクションで説明されているパーティション処理機能により、スプリットブレインが発生したときに、キャッシュで実行できる操作を設定できます。Red Hat Data Grid は複数のパーティション処理ストラテジーを提供します。これは、Brewer のCAP 理論により、パーティションの有無を判断できます。以下は、指定されるストラテジーの一覧です。

ストラテジー	詳細	CAP
DENY_READ_WRITES	パーティションに特定のセグメントのすべての所有者がない場合、そのセグメント内のすべてのキーの読み取りと書き込みの両方が拒否されます。	一貫性
ALLOW_READS	このパーティションに存在する場合は特定のキーの読み取りを許可しますが、このパーティションにセグメントのすべての所有者が含まれている場合にのみ書き込みを許可します。これは、このパーティションで利用可能なものの、クライアントアプリケーションの観点から見えても決定論的ではない場合に一部のエントリーが読み取り可能であるため、一貫したアプローチです。	一貫性
ALLOW_READ_WRITES	各パーティションのエントリが分岐することを許可し、パーティションのマージ時に競合解決を試みます。	可用性

アプリケーションの要件によって、適切なストラテジーを決定する必要があります。たとえば、DENY_READ_WRITES は、高い整合性要件を持つアプリケーションにより適しています。つまり、システムからデータを読み取ったデータを正確にする必要があります。Red Hat Data Grid がベストエ

フォートキャッシュとして使用される場合、パーティションは完全に容認でき、ALLOW_READ_WRITES は一貫性を保つため、より適している可能性があります。

以下のセクションでは、Red Hat Data Grid が各パーティション処理ストラテジーの スプリットブレイン および 連続する失敗 を処理する方法を説明します。これには、マージ ポリシー を使用したパーティションマージ後の Red Hat Data Grid による自動競合解決がどのように使用できるかのセクションがあります。最後に、パーティション処理ストラテジーおよびマージポリシーの設定方法 を説明するセクションを説明します。

6.6.1. スプリットブレイン

スプリットブレインの状況では、各ネットワークパーティションは独自のJGroupsビューをインストールし、他のパーティションからノードを削除します。パーティションが互いに認識していないため、2つ以上のパーティションに分割されているかどうかを直接判断する方法はありません。代わりに、明示的な脱退メッセージを送信せずに1つ以上のノードがJGroupsクラスターから消えたときに、クラスターが分割されたと想定します。

6.6.1.1. 分割ストラテジー

このセクションでは、スプリットブレインが発生したときに各パーティション処理ストラテジーがどのように動作するかを詳細に説明します。

6.6.1.1.1. ALLOW_READ_WRITES

各パーティションは、AVAILABLE モードで残りのすべてのパーティションで、独立したクラスターとして機能します。つまり、各パーティションはデータの一部のみを確認でき、各パーティションはキャッシュに競合する更新を書き込む可能性があることを意味します。これらの競合をマージすると、ConflictManager と設定された EntryMergePolicy を利用することで自動的に解決されます。

6.6.1.1.2. DENY_READ_WRITES

分割が検出されると、各パーティションは即座にリバランスを開始しませんが、まずは代わりに **DEGRADED** モードになるかどうかを確認します。

- 少なくとも1つのセグメントがすべての所有者を失った場合（最後のリバランスが終了してから少なくともnumOwnersノードが残っていることを意味します）、パーティションは DEGRADED モードに入ります。
- **最新の安定したトポロジー** に、パーティションに単純なノード (floor(numNodes/2) + 1) が含まれていない場合は、パーティションも DEGRADED モードになります。
- それ以外の場合、パーティションは正常に機能し、リバランスを開始します。

安定したトポロジーは、リバランス操作が終了するたびに更新され、コーディネーターによって別のリバランスが不要であると判断されます。

これらのルールは、複数のパーティションがAVAILABLE モードになるようにし、他のパーティションがDEGRADED モードになるようにします。

パーティションがDEGRADED モードの場合、完全に所有されているキーへのアクセスのみを許可します。

- このパーティション内のノード上のコピーをすべて持つエントリーのリクエスト（読み取りおよび書き込み）は異常な状態です。

- *消えたノードによって部分的または完全に所有されているエントリの要求は、**AvailabilityException** で拒否されます。*

これにより、パーティションが同じキーに異なる値を書き込めないようにし（キャッシュの一貫性を保つ）、また、1 つのパーティションが他のパーティションで更新されたキーを読み取れないようにすることができます（古いデータはありません）。

*強調を行うには、**numOwners = 2** を使用した分散モードに設定されている初期クラスター **M = {A, B, C, D}** を検討してください。さらに、**owners(k1) = {A,B}**、**owners(k2) = {B,C}**、および**owners(k3) = {C,D}**となる3つのキー**k1**、**k2**、および **k3**（キャッシュに存在するかどうかは不明）について考えてみます。次に、ネットワークは **N1 = {A, B}** と **N2 = {C, D}** の 2 つのパーティションに分割され、DEGRADED モードに入り、以下のように動作します。*

- ***N1**では、**k1**は読み取り/書き込みに使用でき、**k2**（部分的に所有）と**k3**（所有されていない）は使用できず、それらにアクセスすると、**AvailabilityException** が発生します。*
- ***N2** では**k1** および **k2** は読み取り/書き込みには使用できません。**k3** が利用できます。*

*パーティション処理プロセスの関連する要素として、スプリットブレインが発生すると、作成されるパーティションは、キーの所有権を算出するために元のセグメントマッピング（スプリットブレインの前に存在したもの）に依存します。**k1**、**k2**、または **k3** がすでに存在しているかどうかは重要で、それらの可用性は同じです。*

*さらに別の時点でネットワークが回復し、**N1**パーティションと**N2**パーティションがマージして最初のクラスター**M**に戻ると、**M**は劣化モードを終了し、再び完全に使用可能になります。このマージ操作中、**M**は再利用可能になったため、ConflictManager と設定された EntryMergePolicy を使用して、スプリットブレインの発生と検出の間に発生した可能性のある競合をチェックします。*

*別の例として、クラスターは **O1 = {A, B, C}** および **O2 = {D}** の 2 つのパーティションに分割して、パーティション**O1**が完全に使用可能になるようにすることができます（残りのメンバーのキャッシュエントリのバランスを取り直します）。ただし、パーティション **O2** は分割を検出し、パフォーマンスが低下します。キーは完全に所有されていないため、**AvailabilityException**による読み取り操作または書き込み操作は拒否されます。*

*その後、**O1** および **O2** のパーティションが **M** にマージし直すと、ConflictManager は競合を解決しようとし、もう一度 **D** が完全に利用可能になります。*

6.6.1.1.3. ALLOW_READS

パーティションは DENY_READ_WRITES と同じ方法で処理されます。ただし、パーティションが DEGRADED モードの場合に、部分的に所有されたキー上の読み取り操作は、AvailabilityException をスローしません。

6.6.1.2. 現在の制限

2 つのパーティションは、分離して開始できます。マージしない限り、一貫性のないデータの読み込み、書き込みを行うことができます。今後は、カスタムの可用性ストラテジーを許可します（例： 特定のノードがクラスターの一部であるか、または外部のマシンにアクセスできるかどうかを確認など）。

6.6.2. 連続するノードが停止

前のセクションで説明したように、Red Hat Data Grid はプロセス/マシンがクラッシュしたためにノードが JGroups ビューを残すか、ネットワーク障害の原因かを検出することができません。ノードが JGroups クラスターを突然残す場合は、ネットワークの問題が原因であることが想定されます。

*設定したコピー数(**numOwners**)が1よりも大きい場合、クラスターは利用可能なままになり、クラッ*

シュしたノードでデータの新しいレプリカを作成しようとします。ただし、リバランスプロセス中にその他のノードがクラッシュする可能性があります。短期間に **numOwners** ノードがクラッシュすると、一部のキャッシュエントリーがクラスターから完全に消える可能性があります。この場合、DENY_READ_WRITES または ALLOW_READS ストラテジーを有効にすると、スプリットブレインがあると仮定し、スプリットブレインセクションで説明されているように DEGRADED モードを入力します。

管理者は、**numOwners** を超えるノードを急速にシャットダウンして、それらのノードにのみ保存したデータが失われないようにすることもできます。管理者がノードを正常にシャットダウンすると、Red Hat Data Grid はノードが戻らないことを認識します。ただし、クラスターは各ノードの残状況を追跡せず、それらのノードがクラッシュしたかのように、キャッシュが DEGRADED モードに入ります。

この段階では、クラスターが停止し、外部ソースからのデータを使用して再起動時にその状態を回復する方法はありません。クラスターは、**numOwners** の連続するノードの失敗を回避するために、適切な **numOwners** で構成されることが予想されます。したがって、この状況は非常に稀なはずです。アプリケーションがキャッシュ内の一部のデータを失うことが可能な場合、管理者は JMX 経由で可用性モードを AVAILABLE に戻すことができます。

6.6.3. 競合マネージャー

競合マネージャーは、ユーザーが特定のキーに保存されているすべてのレプリカ値を取得できるツールです。保存したレプリカの値が競合しているキャッシュエントリーのストリームをユーザーが処理できるようにする他にもあります。さらに、EntryMergePolicy インターフェースの実装を利用することで、そのような競合を自動的に解決できます。

6.6.3.1. 競合の検出

競合は、指定のキーの保存された各値を取得することによって検出されます。競合マネージャーは、現在の整合性ハッシュによって定義された各キーの書き込み所有者から保存された値を取得します。保存した値の .equals メソッドは、すべての値が等しいかどうかを判断するために使用されます。すべての値が等しい場合は、キーに競合が発生しません。それ以外の場合は競合が発生しています。指定のノードにエントリーが存在しない場合は、null 値が返されることに注意してください。そのため、指定のキーに null 値と非 null 値が存在する場合に競合が発生している点に注意してください。

6.6.3.2. マージポリシー

指定の CacheEntry の 1 つ以上のレプリカ間で競合が発生した場合は、競合解決アルゴリズムを定義する必要があるため、EntryMergePolicy インターフェースを提供します。このインターフェースは、単一のメソッド"merge" で構成され、その返されたCacheEntryは、指定されたキーの"resolved"エントリとして使用されます。null 以外の CacheEntry が返されると、このエントリー値はキャッシュのすべてのレプリカに "put" されます。ただし、マージの実装が null 値を返すと、競合するキーに関連付けられたすべてのレプリカがキャッシュから削除されます。

merge メソッドは、"preferredEntry" および "otherEntries" という 2 つのパラメーターを取ります。パーティションのマージのコンテキストでは、希望のEntry がパーティションに保存されている CacheEntry のプライマリーレプリカで、パーティションには最も大きな topologyId を持つものと等しくなります。パーティションが重複している場合、つまりノード A は両方のパーティション {A}、{A,B,C} のトポロジーにあります。このパーティションでは、他のパーティションのトポロジーの背後に topologId が高いため、{A} を選択します。パーティションのマージが発生しないと、"preferredEntry" は CacheEntry のプライマリーレプリカです。2 番目のパラメーター "otherEntries" は、競合が検出されたキーに関連付けられた他のすべてのエントリーのリストです。

注記

EntryMergePolicy::merge は、競合が検出されたときにのみ呼び出され、すべての CacheEntrys が同じ場合は呼び出されません。

現在、Red Hat Data Grid は以下の EntryMergePolicy の実装を提供します。

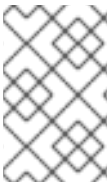
Policy	詳細
MergePolicy.NONE（デフォルト）	競合を解決するための試行はありません。マイノリティパーティションでホストされているエントリは削除され、このパーティションのノードはリバランスが開始されるまでデータを保持しません。この動作は、競合解決をサポートしない Infinispan の以前のバージョンと同等であることに注意してください。この場合、マイナーパーティションでホストされるエントリーに対する変更はすべて失われますが、リバランスが終了するとすべてのエントリーの一貫性が保たれます。
MergePolicy.PREFERRED_ALWAYS	常に "preferredEntry" を使用します。MergePolicy.NONE は、PREFERRED_ALWAYS とほぼ同等で、競合解決の実行によるパフォーマンスへの影響はありません。そのため、以下のシナリオが懸念されない限り、MergePolicy.NONE を選択する必要があります。DENY_READ_WRITES または DENY_READ ストラテジーを使用する場合は、パーティションが DEGRADED モードに入る場合にのみ、書き込み操作が部分的に完了するため、一貫性のない値が含まれるレプリカが作成されます。MergePolicy.PREFERRED_ALWAYS は対象の不整合を検出し、これを解決します。ただし、MergePolicy.NONE の場合、CacheEntry レプリカはクラスターのリバランス後に一貫性がありません。
MergePolicy.PREFERRED_NON_NULL	null 以外の場合は "preferredEntry" を使用します。それ以外の場合は、"otherEntries" の最初のエントリーを利用します。
MergePolicy.REMOVE_ALL	競合が検出されると、必ずキャッシュからキーを削除します。
完全修飾クラス名	マージのカスタム実装は、カスタムマージポリシーを使用します。

6.6.4. 使用法

パーティションが ConflictManager をマージすると、設定された EntryMergePolicy の競合を自動的に解決しようとしますが、アプリケーションが必要とする競合の有無を手動で検索または解決することもできます。

以下のコードは、EmbedCacheManager の ConflictManager を取得する方法、指定されたキーの全バージョンを取得する方法、および特定のキャッシュ全体で競合をチェックする方法を示しています。

```
EmbeddedCacheManager manager = new DefaultCacheManager("example-config.xml");
Cache<Integer, String> cache = manager.getCache("testCache");
ConflictManager<Integer, String> crm = ConflictManagerFactory.get(cache.getAdvancedCache());

// Get All Versions of Key
Map<Address, InternalCacheValue<String>> versions = crm.getAllVersions(1);

// Process conflicts stream and perform some operation on the cache
Stream<Map<Address, InternalCacheEntry<Integer, String>>> stream = crm.getConflicts();
stream.forEach(map -> {
   CacheEntry<Object, Object> entry = map.values().iterator().next();
   Object conflictKey = entry.getKey();
   cache.remove(conflictKey);
});

// Detect and then resolve conflicts using the configured EntryMergePolicy
crm.resolveConflicts();

// Detect and then resolve conflicts using the passed EntryMergePolicy instance
crm.resolveConflicts((preferredEntry, otherEntries) -> preferredEntry);
```



注記

ConflictManager::getConflicts ストリームはエントリーごとに処理されますが、基礎となるスプリットは、セグメントごとにキャッシュエントリーを遅延読み込みしています。

6.6.5. パーティション処理の設定

キャッシュが分散またはレプリケートされない限り、パーティション処理の設定は無視されます。デフォルトのパーティション処理ストラテジーは ALLOW_READ_WRITES で、デフォルトの EntryMergePolicy は MergePolicies::PREFERRED_ALWAYS です。

```
<distributed-cache name="the-default-cache">
   <partition-handling when-split="ALLOW_READ_WRITES" merge-
policy="PREFERRED_NON_NULL"/>
</distributed-cache>
```

プログラムを使用しても同じことができます。

```
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
                .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
                .mergePolicy(MergePolicies.PREFERRED_ALWAYS);
```

6.6.5.1. カスタムマージポリシーの実装

EntryMergePolicy のカスタム実装を提供することもできます。

```
<distributed-cache name="the-default-cache">
```

```
   <partition-handling when-split="ALLOW_READ_WRITES" merge-
policy="org.example.CustomMergePolicy"/>
</distributed-cache>
```

```
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
                .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
                .mergePolicy(new CustomMergePolicy());
```

```
public class CustomMergePolicy implements EntryMergePolicy<String, String> {

   @Override
   public CacheEntry<String, String> merge(CacheEntry<String, String> preferredEntry,
List<CacheEntry<String, String>> otherEntries) {
      // decide which entry should be used

      return the_solved_CacheEntry;
   }
```

6.6.5.2. Infinispan サーバーインスタンスへのカスタムマージポリシーのデプロイ

サーバーでカスタム EntryMergePolicy 実装を使用するには、実装クラスをサーバーにデプロイする必要があります。これは、java service-provider の規則を使用し、EntryMergePolicy 実装の完全修飾クラス名を含む META-INF/services/org.infinispan.conflict.EntryMergePolicy ファイルを持つ jar のクラスファイルをパッケージ化することで実現できます。

```
# list all necessary implementations of EntryMergePolicy with the full qualified name
org.example.CustomMergePolicy
```

サーバー上でカスタムのマージポリシーが使用されるようにするには、ポリシーセマンティクスが保存された Key/Value オブジェクトへのアクセスを必要とする場合は、オブジェクトストレージを有効にする必要があります。これは、サーバーのキャッシュエントリーはマーシャル形式に格納され、ポリシーに返された Key/Value オブジェクトは WrappedByteArray のインスタンスとなる可能性があるためです。ただし、カスタムポリシーがキャッシュエントリーに関連付けられたメタデータのみに依存する場合、オブジェクトストレージは必要ではなく、リクエストごとのマーシャリングデータの追加のパフォーマンスコストが原因で（他の理由で必要としなければ）回避する必要があります。最後に、提供されるマージポリシーのいずれかが使用されている場合は、オブジェクトストレージは必要ありません。

6.6.6. 監視および管理

キャッシュの可用性モードは、キャッシュ MBean の属性として JMX で公開されます。属性は書き込み可能で、管理者はキャッシュを DEGRADED モードから AVAILABLE（一貫性のコスト）に強制的に移行することができます。

可用性モードは、AdvancedCache インターフェースからもアクセスできます。

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

// Read the availability
boolean available = ac.getAvailability() == AvailabilityMode.AVAILABLE;

// Change the availability
```

```
if (!available) {
   ac.setAvailability(AvailabilityMode.AVAILABLE);
}
```

第7章 マーシャリング

マーシャリングは、Java POJO をネットワーク経由で転送できる形式で記述できるプロセスです。アンマーシャリングは、有線形式から読み取られたデータが Java POJO に変換するリバースプロセスです。Red Hat Data Grid は以下の目的でマーシャリング/アンマーシャリングを使用します。

- クラスターの他の Red Hat Data Grid ノードにデータを送信できるように変換します。
- データを変換し、基礎となるキャッシュストアに保存できるようにします。
- データを Red Hat Data Grid をワイヤ形式で保存し、レイジーデシリアライズ機能を提供します。

7.1. JBOSS MARSHALLING のロール

このプロセスではパフォーマンスは非常に重要であるため、Red Hat Data Grid は標準の Java Serialization の代わりに JBoss Marshalling フレームワークを使用して Java POJO をマーシャリング/アンマーシャリングします。また、このフレームワークにより、Red Hat Data Grid は、常に使用される Red Hat Data Grid Java POJO をマーシャリングするための非常に効率的な方法を提供します。内部 Java クラスを含む Java POJO をマーシャリングする効率的な方法を提供する以外に、JBoss Marshalling は標準の **java.io.ObjectOutput** および java.io.ObjectOutput および **java.io.ObjectInput** 実装よりも高性能な **java.io.ObjectOutputStream** および **java.io.** ObjectInput 実装を使用します。

7.2. 非データセンターオブジェクトのサポート

ユーザーの観点では、Red Hat Data Grid が非システムオブジェクトの保存をサポートするかどうかが非常に一般的です。4.0 では、Red Hat Data Grid キャッシュインスタンスは、以下の場合にのみ、非終了キーまたは値オブジェクトのみを保存できます。

- キャッシュは、ローカルキャッシュ **として設定されます。**
- キャッシュはレイジーシリアライゼーションなどで設定されていません **。**
- キャッシュが write-behind キャッシュストアで設定されない

これらのオプションのいずれかが true の場合、キャッシュ内のキー/値のペアをマーシャリングし、現時点では java.io.336 または **java.io. Externalizable** を拡張する必要があります。

ヒント

Red Hat Data Grid 5.0 以降、ユーザーがこれらの非サルカライズ可能なオブジェクトに意味のある Externalizer 実装を提供できる限り、Red Hat Data Grid 5.0 はアンマーシャリングします。

インスタンスが Red Hat Data Grid に保存されているクラスに Rich または Externalizable を再調整できない場合は、XStream のようなものを使用して、Red Hat Data Grid に格納できる文字列に変換することもできます。XStream の使用に関する注意点は、XML 変換が必要な XML 変換により、キー/値オブジェクトを格納するプロセスが遅くなるためです。

7.2.1. バイナリーとして保存

バイナリーとして保存すると、データをシリアライズされた形式で保存できます。これは、Red Hat Data Grid がオブジェクトのシリアライズとデシリアライズによって使用されて必要な時点を遅らせる仕組みで、レイジーデシリアライズを実現すると便利です。これは通常、デシリアライズが必要な呼び

出しのスレッドコンテキストクラスローダーを使用して行われ、クラスローダーの分離を提供する効果的なメカニズムであることを意味します。デフォルトではレイジーデシリアライズは無効になっていますが、有効にする場合は、以下のように実行できます。

- キャッシュレベルでの XML より <* **-cache /> 要素の下にあり、以下のように** なります。

```
<memory>
  <binary />
</memory>
```

- プログラムで行う:

```
ConfigurationBuilder builder = ...
builder.memory().storageType(StorageType.BINARY);
```

7.2.1.1. 等価性に関する考慮事項

Lazy deserialization/storing をバイナリーとして使用する場合、キーと値は WrappedBytes としてラップされます。このラッパークラスは、オンデマンドでシリアル化とデシリアライズを透過的に行い、内部でオブジェクト自体への参照、またはこのオブジェクトのシリアライズされたバイトアレイ表現を持つ場合があります。

これは、等価動作に特に有効です。このクラスの equals（）メソッドは、比較時に両方のインスタンスがシリアライズされているか、またはデシリアライズされた形式であるかどうかに応じて、ラップされたオブジェクトインスタンスの equals（）メソッドに委譲されるか、またはラップされたオブジェクトインスタンスの equals（）メソッドに委譲されます。比較される 2 つのインスタンスの形式が異なる場合、1 つのインスタンスはシリアライズまたはデシリアライズされます。

これは、**storeAsBinary** を使用すると、キャッシュに保存されている鍵が機能する方法に影響します。これは、MarishalledValue でラップされるキーの比較が行われるためです。 キーに equals（）メソッドの実装は、上記のようにキーが MarshalledValue としてラップされる場合は、等価比較の動作を認識する必要があります。

7.2.1.2. 復元コピー経由のストア別値

設定 **ストアAsBinary** は、Deensive コピーを有効にすることができます。これにより、動作のようなストアごとの値が可能になります。

Red Hat Data Grid は、保存時にオブジェクトがマーシャリングするため、ローカルキャッシュだけでなく、オブジェクト参照への変更はキャッシュに保存されません。これにより、動作のような store-by-value が提供されます。**storeAsBinary** の有効化が可能です。

- キャッシュレベルの XML より、<* **-cache /> または < default /> 要素のいずれかで以下を** 行います。

```
<store-as-binary keys="true" values="true"/>
```

- プログラムで行う:

```
ConfigurationBuilder builder = ...
builder.storeAsBinary().enable().storeKeysAsBinary(true).storeValuesAsBinary(true);
```

7.3. 高度な設定

内部的には、Red Hat Data Grid は この Marshaller インターフェース の実装を使用して、Java オブジェクトをマーシャリングまたはアンマーシャリングして、グリッド内の他のノードを送信したり、キャッシュストアに保存したり、レイジーデシリアライズ用のバイトアレイに変換することもできます。

デフォルトでは、Red Hat Data Grid は GlobalMarshaller を使用します。オプションで、Red Hat Data Grid ユーザーは、以下のように独自のマーシャラーを提供できます。

- CacheManager レベルで < **cache-manager /&gt**; 要素で XML を使用します。

```
<serialization marshaller="com.acme.MyMarshaller"/>
```

- プログラムで行う:

```
GlobalConfigurationBuilder builder = ...
builder.serialization().marshaller(myMarshaller); // needs an instance of the marshaller
```

7.3.1. トラブルシューティング

Red Hat Data Grid のマーシャリング層および特定の JBoss Marshalling で、一部のユーザーオブジェクトのマーシャリング/アンマーシャリングで問題が発生する場合があります。Red Hat Data Grid 4.0 では、マーシャリングの例外にはマーシャリングされたオブジェクトの詳細情報が含まれます。例:

```
java.io.NotSerializableException: java.lang.Object
at org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.java:857)
at org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at
org.infinispan.marshall.exts.ReplicableCommandExternalizer.writeObject(ReplicableCommandExternali
er.java:54)
at
org.infinispan.marshall.jboss.ConstantObjectTable$ExternalizerAdapter.writeObject(ConstantObjectTab
e.java:267)
at org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.java:143)
at org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at org.infinispan.marshall.jboss.JBossMarshaller.objectToObjectStream(JBossMarshaller.java:167)
at org.infinispan.marshall.VersionAwareMarshaller.objectToBuffer(VersionAwareMarshaller.java:92)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToByteBuffer(VersionAwareMarshaller.java:170)

at
org.infinispan.marshall.DefaultMarshallerTest.testNestedNonSerializable(VersionAwareMarshallerTest.j
ava:415)
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
... Removed 22 stack frames
```

「in object」メッセージの読み取り方法は、stacktraces が読み取られる方法と同じです。「in object」が最も内部にあり、一番古い「in object」メッセージが最も古いメッセージです。そのため、上記の例は、java.lang.Object@b40ec4 がシリアライズ可能でないため、org.infinispan.commands.write.PutKeyValueCommand のインスタンスに含まれる java.lang.Object.write.PutKeyValueCommand はシリアライズできないことを示しています。

これはすべてではありません。DEBUG または TRACE ロギングレベルを有効にすると、マーシャリングの例外には、スタックトレース内のオブジェクトの toString（）表現が表示されます。以下に例を示します。

```
java.io.NotSerializableException: java.lang.Object
...
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
-> toString = java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
-> toString = PutKeyValueCommand{key=k, value=java.lang.Object@b40ec4, putIfAbsent=false,
lifespanMillis=0, maxIdleTimeMillis=0}
```

例外のアンマーシャリングに関しては、このような情報レベルが非常に複雑であるものの、可能な場合を示しています。Red Hat Data Grid はクラスタイプの情報を提供します。以下に例を示します。

```
java.io.IOException: Injected failure!
at
org.infinispan.marshall.DefaultMarshallerTest$1.readExternal(VersionAwareMarshallerTest.java:426)
at org.jboss.marshalling.river.RiverUnmarshaller.doReadNewObject(RiverUnmarshaller.java:1172)
at org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:273)
at org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:210)
at org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller.java:85)
at org.infinispan.marshall.jboss.JBossMarshaller.objectFromObjectStream(JBossMarshaller.java:210)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarshaller.java:10
4)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarshaller.java:17
7)
at
org.infinispan.marshall.DefaultMarshallerTest.testErrorUnmarshalling(VersionAwareMarshallerTest.java
:431)
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.DefaultMarshallerTest$1
```

この例では、内部クラス org.infinispan.marshall.DefaultMarshallerTest$1 のインスタンスをアンマーシャリングしようとすると IOException がスローされました。例外のマーシャリングと同様に、DEBUG または TRACE ロギングレベルが有効な場合には、クラスタイプのクラスローダー情報が提供されます。以下に例を示します。

```
java.io.IOException: Injected failure!
...
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.DefaultMarshallerTest$1
-> classloader hierarchy:
-> type classloader = sun.misc.Launcher$AppClassLoader@198dfaf
->...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/eclipse-testng.jar
->...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/lib/testng-jdk15.jar
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/test-classes/
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/classes/
->...file:/home/galder/.m2/repository/org/testng/testng/5.9/testng-5.9-jdk15.jar
->...file:/home/galder/.m2/repository/net/jcip/jcip-annotations/1.0/jcip-annotations-1.0.jar
-
>...file:/home/galder/.m2/repository/org/easymock/easymockclassextension/2.4/easymockclassextensio
```

```
2.4.jar
->...file:/home/galder/.m2/repository/org/easymock/easymock/2.4/easymock-2.4.jar
->...file:/home/galder/.m2/repository/cglib/cglib-nodep/2.1_3/cglib-nodep-2.1_3.jar
->...file:/home/galder/.m2/repository/javax/xml/bind/jaxb-api/2.1/jaxb-api-2.1.jar
->...file:/home/galder/.m2/repository/javax/xml/stream/stax-api/1.0-2/stax-api-1.0-2.jar
->...file:/home/galder/.m2/repository/javax/activation/activation/1.1/activation-1.1.jar
->...file:/home/galder/.m2/repository/jgroups/jgroups/2.8.0.CR1/jgroups-2.8.0.CR1.jar
->...file:/home/galder/.m2/repository/org/jboss/javaee/jboss-transaction-api/1.0.1.GA/jboss-
transaction-api-1.0.1.GA.jar
->...file:/home/galder/.m2/repository/org/jboss/marshalling/river/1.2.0.CR4-SNAPSHOT/river-
1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/marshalling/marshalling-api/1.2.0.CR4-
SNAPSHOT/marshalling-api-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/jboss-common-core/2.2.14.GA/jboss-common-core-
2.2.14.GA.jar
->...file:/home/galder/.m2/repository/org/jboss/logging/jboss-logging-spi/2.0.5.GA/jboss-logging-spi-
2.0.5.GA.jar
->...file:/home/galder/.m2/repository/log4j/log4j/1.2.14/log4j-1.2.14.jar
->...file:/home/galder/.m2/repository/com/thoughtworks/xstream/xstream/1.2/xstream-1.2.jar
->...file:/home/galder/.m2/repository/xpp3/xpp3_min/1.1.3.4.O/xpp3_min-1.1.3.4.O.jar
->...file:/home/galder/.m2/repository/com/sun/xml/bind/jaxb-impl/2.1.3/jaxb-impl-2.1.3.jar
-> parent classloader = sun.misc.Launcher$ExtClassLoader@1858610
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/localedata.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunpkcs11.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunjce_provider.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/dnsns.jar
... Removed 22 stack frames
</code>
```

マーシャリング/アンマーシャリング例外の根本原因を見つけることが実際には難しいかもしれませんが、上記の改善により、より速く効率的な方法でこれらを見つけることができます。

7.4. ユーザー定義の外部ナライザー

Red Hat Data Grid の主な機能の1つは、一部の機能を提供するために、多くの場合、オブジェクトをマーシャリング/アンマーシャリングする必要があることです。たとえば、ライトスルーまたはライトベロワードキャッシュストアにオブジェクトを格納する必要がある場合、保存されたオブジェクトにはマーシャリングが必要です。Red Hat Data Grid ノードのクラスターが形成されている場合は、同梱されるオブジェクトマーシャリングが必要です。レイジーデシリアライズを有効にした場合でも、正しいクラスローダーで遅延アンマーシャリングできるようにオブジェクトをマーシャリングする必要があります。

標準の JDK シリアライゼーションの使用は遅くなり、大きすぎるペイロードを生成し、帯域幅の使用量に影響を及ぼす可能性があります。その上では、JDK シリアライゼーションは不変である可能性のあるオブジェクトと適切に動作しません。これらの問題を回避するため、Red Hat Data Grid はマーシャリング/アンマーシャリングオブジェクトに JBoss Marshalling を使用します。JBoss マーシャリングは高速で、非常にスペース効率の高いペイロードを生成し、その上にアンマーシャリング時にユーザーがオブジェクトの作成方法を完全に制御できるため、オブジェクトをイミュータブルに保つことができます。

5.0 からは、Red Hat Data Grid のユーザーはこのマーシャリングフレームワークを活用できるようになり、独自の外部ナライザーの実装も提供できるようになりましたが、外部ファイナライザーの提供方法を見つける前に、そのメリットを見ていきましょう。

7.4.1. Externalizers の利点

JDK では、最もシンプルな形式でオブジェクトをシリアライズする簡単な方法は、java.io. 336 を拡張するだけですが、既知のとおり、これは遅くなり、大きすぎるペイロードを生成します。シリアライズを行う別の方法として、JDK シリアライゼーションに依存してオブジェクトがjava.io.Externalizable を拡張することです。これにより、ユーザーは独自のクラスをマーシャリング/アンマーシャリングすることができますが、低速な JDK シリアライゼーションに依存して、シリアライゼーションするクラスを強制的に拡張するために強制するので、深刻な問題があります。1 つ目は、この 2 つの副作用があります。1 つ目は、ソースを所有せず、そうでないクラスのソースコードを変更するだけで良いからです。2 つ目の点として、外部化可能な実装はオブジェクト作成を制御しないので、状態を復元するためにセットメソッドを追加することが強制されるため、イミュータブルなオブジェクトが変更不能になることがあります。

JDK のシリアライズに依存する代わりに、Red Hat Data Grid は JBoss Marshalling を使用してオブジェクトをシリアライズし、特定のクラスのオブジェクトをシリアライズしたフォームに変換する方法を認識する Externalizer インターフェース実装に関連付けてクラスをシリアライズする必要があります。Red Hat Data Grid は、Externalizer を実装するためにオブジェクトのシリアライズを強制しません。実際、別のクラスを使用して Externalizer インターフェースを実装することが推奨されます。これは、JDK のシリアライゼーションとは異なり、Externalizer 実装がストリームからオブジェクトを読み取ろうとするときに特定クラスのオブジェクトの作成方法を制御するためです。つまり、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するため、ユーザーはこれらのインスタンスを作成する方法（ファクトリーまたはリフレクションによる直接インスタンス化など）をどのように柔軟にするか、またターゲットクラスを不変に保つことができます。この種の外部データベースアーキテクチャーは、単一の責任 の原則など、適切な OOP 設計原則をプロモートします。

これは非常に一般的であり、一般的には、Externalizer 実装を外部化するクラス内の内部静的パブリッククラスとして保存できます。これを行う利点は、関連するコードが一緒に維持され、維持が容易になります。Red Hat Data Grid では、Red Hat Data Grid をユーザー定義の外部ライザーとプラグインする方法が 2 つあります。

7.4.2. User Friendly Externalizers

最も単純な形式では、ユーザーはマーシャリング/アンマーシャリングするタイプに Externalizer 実装を提供し、使用する外部ナライザークラスを示す {@link SerializeWith} アノテーションを使用してマーシャリングされたタイプクラスにアノテーションを付けるだけです。以下に例を示します。

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

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

   final String name;
   final int age;

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

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

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

ランタイム時に JBoss Marshalling はオブジェクトを検査し、それがマーシャリング可能である（アノテーションに想定）であることを検出します。そのため、渡された外部izer クラスを使用してマーシャリングします。外部データベースの実装をコードをより簡単にするために、型< **T** > をマーシャリング/アンマーシャリングするオブジェクトのタイプとして定義するようにしてください。

この外部ライザーの定義方法は非常にユーザーフレンドリーですが、以下のような不利な点があります。

- モデルの制約により、同じクラスの異なるバージョンのサポートや Externalizer クラスをマーシャリングする必要があるなどの理由で、この方法で生成されたペイロードサイズは最も効率的ではありません。
- このモデルでは、マーシャリングされたクラスには link:https://access.redhat.com/webassets/avalon/d/red-hat-data-grid/7.3/api/org/infinispan/commons/marshall/SerializeWith.html を指定する必要があります。ただし、ユーザーはソースコードが利用できないクラス用に Externalizer を指定する必要がある場合や、他の制約では変更できません。
- このモデルによるアノテーションの使用は、ユーザーから離れるマーシャリング層などの低レベルの詳細を抽象化しようとするフレームワーク開発者やサービスプロバイダーに制限される場合があります。

これらの欠点のいずれかの影響を受ける場合、外部担当者を提供する代替方法は、より高度な外部管理者で利用できます。

7.4.3. 高度な外部ナライザー

AdvancedExternalizer は、Externalizer で説明されている、ユーザーフレンドリーな外部データベース定義モデルの辞書性を解決する、マーシャリング/アンマーシャリングユーザー定義のクラスを外部が提供する代替方法を提供します。以下に例を示します。

```
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;
      }
    }
  }
```

最初の顕著な違いは、このメソッドではユーザークラスを anyway でアノテーションする必要がないことです。そのため、ソースコードが利用できないクラスや、変更できないクラスで使用できます。externalizer とマーシャリング/アンマーシャリングされたクラスとの間のバインドは、この外部ャーがマーシャリングできるクラスの一覧を返す getTypeClasses（） の実装を提供することで設定されます。

7.4.3.1. Externalizers と Marshaller クラスをリンク

Externalizer の readObject（）メソッドおよび writeObject（）メソッドを実装したら、それらを外部化するタイプクラスとリンクします。これを実行するには、Externalizer 実装は getTypeClasses（）実装を提供する必要があります。以下に例を示します。

```
import org.infinispan.commons.util.Util;
...
@Override
public Set<Class<? extends ReplicableCommand>> getTypeClasses() {
  return Util.asSet(LockControlCommand.class, RehashControlCommand.class,
      StateTransferControlCommand.class, GetKeyValueCommand.class,
      ClusteredGetCommand.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);
}
```

上記のコードでは、ReplicableCommandExternalizer は複数のタイプのコマンドを外部化できることを示しています。実際、ReplicableCommand インターフェースを拡張するすべてのコマンドをマーシャリングしますが、現在フレームワークはクラスの等価比較のみをサポートしているため、マーシャリングするクラスが特定のクラス/インターフェースのすべての子であることを示すことはできません。

ただし、外部化するクラスがプライベートである場合があるため、実際のクラスインスタンスを参照できない場合があります。このような場合、ユーザーは指定の完全修飾クラス名でクラスを検索し、そのクラスをパスできます。以下に例を示します。

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

7.4.3.2. Externalizer Identifier

2 つ目は、生成されたペイロードで可能な限り最大領域を節約するために、高度なサイライザーは getId（） 実装で識別されたポジュライザーの実装や、ペイロードのアンマーシャリング時に externalizer を識別する XML/programmatic 設定を介して特定される外部ャーの実装を必要とします。ただし、これを機能させるには、高度なサイライザーを使用するには、次のセクションで説明される XML またはプログラムによる設定を介して、外部アーライザーをキャッシュマネージャーの作成時間に登録する必要があります。Externalizer と SerializeWith をベースにした外部ライザーでは、事前登録は必要ありません。内部では、Red Hat Data Grid はこの高度な外部データベースメカニズムを使用して、内部クラスをマーシャリング/アンマーシャリングします。

したがって、getId（）は正の整数を返す必要があります。getId（）が null を返す場合、この高度なサイライザーの ID は、次のセクションで説明される XML/programmatic 設定を介して定義されることを示します。

id のソースに関係なく、正の整数を使用すると、数値の変数長のエンコードが非常に効率的で、外部ナライザー実装クラス情報やクラス名よりも効率が高くなります。Red Hat Data Grid ユーザーは、システム内の他の識別子と競合しない限り、任意の正の整数を使用できます。内部 Red Hat Data Grid Core 外部データベースは特別なもので、ユーザー定義の外部ライザーに対して異なる数値を使用するため、ユーザー定義の外部ナライザーと、Red Hat Data Grid Core プロジェクトにある外部ャーと同じ数値を使用できる点を理解することが重要です。一方、ユーザーは、本記事の最後にある事前に割り当てられた識別子範囲内にある数字を使用しないようにする必要があります。Red Hat Data Grid は起動時に ID の重複の有無を確認し、見つかった場合は起動がエラーを出して停止します。

使用中の ID を維持する場合は、集中的に実行することを強く推奨します。たとえば、getId（）実装は、別のクラスまたはインターフェースで静的に定義された識別子のセットを参照できます。このようなクラス/インターフェースは、使用中の識別子のグローバルビューを提供するので、新しい ID の割り当てが容易になります。

7.4.3.3. 高度な外部データベースの登録

以下の例は、以前に保存した Person オブジェクトの高度な外部サイザー実装を登録するために必要な設定のタイプを示しています。

infinispan.xml

```
<infinispan>
  <cache-container>
    <serialization>
      <advanced-externalizer class="Person$PersonExternalizer"/>
    </serialization>
  </cache-container>
  ...
</infinispan>
```

プログラムで行う:

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

前述のように、これらの外部データベース実装を一覧表示すると、ユーザーは getId（）実装ではなく、XML またはプログラムを使用して外部ナライザーの識別子をオプションで指定できます。ここでも、識別子を維持するための集中的な方法を提供しますが、ルールは明確であることが重要です。XML/programmatic 設定またはアノテーション経由の AdvancedExternalizer の実装は、識別子に関連付ける必要があります。そうでない場合、Red Hat Data Grid はエラーをスローし、起動を中止します。特定の AdvancedExternalizer 実装が XML/programmatic 設定とアノテーションの両方で ID を定義する場合、XML/ プログラムで定義した値は、使用される値になります。id が登録時に定義される外部データベースの例を以下に示します。

infinispan.xml

```
<infinispan>
  <cache-container>
    <serialization>
      <advanced-externalizer id="123"
                             class="Person$PersonExternalizer"/>
    </serialization>
  </cache-container>
  ...
</infinispan>
```

プログラムで行う:

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

最後に、プログラムによる設定に関するいくつかのメモです。GlobalConfiguration.addExternalizer（）は varargs を取ります。したがって、ID が @Marshalls アノテーションですでに定義されていることを仮定して、複数の外部ライザーを 1 度だけ登録できることを意味します。以下に例を示します。

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

7.4.3.4. 事前に割り当てられた Externalizer Id Ranges

これは、Red Hat Data Grid ベースのモジュールまたはフレームワークが使用する Externalizer 識別子の一覧です。Red Hat Data Grid ユーザーは、これらの範囲内の ID を使用しないようにする必要があります。

Red Hat Data Grid ツリーモジュール：	1000 - 1099
Red Hat Data Grid Server モジュール：	1100 - 1199
Hibernate Red Hat Data Grid 2 レベルキャッシュ：	1200 - 1299

Red Hat Data Grid Lucene ディレクトリー：	1300 - 1399
Hibernate OGM:	1400 - 1499
Hibernate Search:	1500 - 1599
Red Hat Data Grid のクエリーモジュール：	1600 - 1699
Red Hat Data Grid のリモートクエリーモジュール：	1700 - 1799
Red Hat Data Grid スクリプトモジュール：	1800 - 1849
Red Hat Data Grid サーバーイベントロガーモジュール：	1850 - 1899
Red Hat Data Grid のリモートストア：	1900 - 1999
Red Hat Data Grid Counters:	2000 - 2049
Red Hat Data Grid のマルチマップ：	2050 - 2099
Red Hat Data Grid のロック：	2100 - 2149

第8章 トランザクション

Red Hat Data Grid は JTA 準拠のトランザクションを使用し、参加するよう設定できます。または、トランザクションのサポートが無効になっている場合は、JDBC 呼び出しで自動コミットを使用する場合と同等になります。ここでは、すべての変更後に変更がレプリケートされる可能性があります (レプリケーションが有効な場合)。

キャッシュ操作ごとに、Red Hat Data Grid は以下を行います。

1. スレッドに関連する現在の トランザクション を取得します。
2. トランザクションのコミットまたはロールバック時に通知されるように、XAResource をトランザクションマネージャーに登録します (登録されていない場合)。

これを実行するには、キャッシュに環境の TransactionManager への参照を提供する必要があります。これは通常、TransactionManagerLookup インターフェースの実装のクラス名でキャッシュを設定することで行われます。キャッシュが起動すると、このクラスのインスタンスを作成し、**TransactionManager** への参照を返す **getTransactionManager()** メソッドを呼び出します。

Red Hat Data Grid には、複数のトランザクションマネージャーの検索クラスが同梱されています。

トランザクションマネージャールックアップの実装

- EmbeddedTransactionManagerLookup: これは、他の実装が利用できない場合にのみ埋め込みモードに使用する必要がある基本的なトランザクションマネージャーを提供します。この実装は、同時トランザクションおよびリカバリーでは、重大な制限があります。
- JBossStandaloneJTAManagerLookup: スタンドアロン環境で Red Hat Data Grid を実行しているか、JBoss AS 7 以前、WildFly 8、9、および 10 で実行している場合、トランザクションマネージャーのデフォルト選択になります。このトランザクションは、**EmbeddedTransactionManager** の不足をすべて解消する JBoss Transactions をベースとした本格的なトランザクションマネージャーです。
- WildflyTransactionManagerLookup: Wildfly 11 以降で Red Hat Data Grid を実行している場合は、トランザクションマネージャーのデフォルト選択になります。
- GenericTransactionManagerLookup: これは、最も一般的な 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ルックアップクラスです。トランザクションマネージャーが見つからない場合は、**EmbeddedTransactionManager** がデフォルトの設定になります。

警告: **DummyTransactionManagerLookup** は 9.0 で非推奨となり、今後削除される予定です。代わりに **EmbeddedTransactionManagerLookup** を使用してください。

初期化すると、**TransactionManager** は **Cache** 自体から取得することもできます。

```
//the cache must have a transactionManagerLookupClass defined
Cache cache = cacheManager.getCache();

//equivalent with calling TransactionManagerLookup.getTransactionManager();
TransactionManager tm = cache.getAdvancedCache().getTransactionManager();
```

8.1. トランザクションの設定

トランザクションはキャッシュレベルで設定されます。以下はトランザクションの動作に影響する設定と、各設定属性の簡単な説明になります。

```
<locking
  isolation="READ_COMMITTED"
  write-skew="false"/>
<transaction
  locking="OPTIMISTIC"
  auto-commit="true"
  complete-timeout="60000"
  mode="NONE"
  notifications="true"
  protocol="DEFAULT"
  reaper-interval="30000"
  recovery-cache="__recoveryInfoCacheName__"
  stop-timeout="30000"
  transaction-manager-
lookup="org.infinispan.transaction.lookup.GenericTransactionManagerLookup"/>
<versioning
  scheme="NONE"/>
```

プログラムを使用する場合

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.locking()
   .isolationLevel(IsolationLevel.READ_COMMITTED)
   .writeSkewCheck(false);
builder.transaction()
   .lockingMode(LockingMode.OPTIMISTIC)
   .autoCommit(true)
   .completedTxTimeout(60000)
   .transactionMode(TransactionMode.NON_TRANSACTIONAL)
   .useSynchronization(false)
   .notifications(true)
   .transactionProtocol(TransactionProtocol.DEFAULT)
   .reaperWakeUpInterval(30000)
   .cacheStopTimeout(30000)
   .transactionManagerLookup(new GenericTransactionManagerLookup())
   .recovery()
   .enabled(false)
   .recoveryInfoCacheName("__recoveryInfoCacheName__");
builder.versioning()
   .enabled(false)
   .scheme(VersioningScheme.NONE);
```

- ***isolation*** - 分離レベルを設定します。詳細は、「分離レベル」を参照してください。デフォルトは ***REPEATABLE_READ*** です。
- ***write-skew***: 書き込みスキューチェックを有効にします（非推奨）。Red Hat Data Grid は、この属性を Library Mode に自動的に設定します。デフォルトは ***READ_COMMITTED*** で ***false*** です。デフォルトは ***REP EATABLE_READ*** です。詳細は、「 Write Skews 」を参照してください。
- ***locking*** - キャッシュが楽観的または悲観的ロックを使用するかどうかを設定します。詳細は、「トランザクションのロック」を参照してください。デフォルトは ***OPTIMISTIC*** です。

- ***auto-commit***: 有効にすると、ユーザーは1回の操作でトランザクションを手動で開始する必要はありません。トランザクションは自動的に起動およびコミットされます。デフォルトは **true** です。
- ***complete-timeout*** - 完了したトランザクションに関する情報を保持する期間(ミリ秒単位)。デフォルトは **60000** です。
- ***mode***: キャッシュがトランザクションかどうかを設定します。デフォルトは **NONE** です。利用可能なオプションは以下のとおりです。
  - ***NONE*** - 非トランザクションキャッシュ
  - ***FULL_XA*** - リカバリーが有効になっている XA トランザクションキャッシュリカバリーの詳細は、「トランザクションリカバリー」を参照してください。
  - ***NON_DURABLE_XA*** - リカバリーが無効になっている XA トランザクションキャッシュ。
  - ***NON_XA*** - XA の代わりに 同期化 を介して統合されたトランザクションキャッシュ。詳細は、「同期の登録」のセクションを参照してください。
  - ***BATCH***- バッチを使用して操作をグループ化するトランザクションキャッシュ。詳細は「バッチ処理」のセクションを参照してください。
- ***notifications*** - キャッシュリスナーのトランザクションイベントを有効/無効にします。デフォルトは **true** です。
- ***protocol***: プロトコルの使用を設定します。デフォルトは **DEFAULT** です。利用可能な値は次のとおりです。
  - ***DEFAULT*** - 従来の Two-Phase-Commit プロトコルを使用します。これについては、以下で説明します。
  - ***TOTAL_ORDER***: **トランスポート** がトランザクションをコミットするために確保された合計順序を使用します。詳細は「 Total Order based commit protocol 」セクションを確認してください。
- ***reaper-interval*** - トランザクション完了情報をクリーンアップするスレッドが開始する間隔(ミリ秒単位)。デフォルトは **30000** です。
- ***recovery-cache*** - リカバリー情報を保存するキャッシュ名を設定します。リカバリーの詳細は、「トランザクションリカバリー」を参照してください。デフォルトは **recoveryInfoCacheName** です。
- ***stop-timeout*** - キャッシュの停止時に進行中のトランザクションを待機する時間(ミリ秒単位)。デフォルトは **30000** です。
- ***transaction-manager-lookup*** - ***javax.transaction.TransactionManager*** への参照を検索するクラスの完全修飾クラス名を設定します。デフォルトは ***org.infinispan.transaction.lookup.GenericTransactionManagerLookup*** です。
- バージョン管理 ***スキーム*** - 書き込みスキューが最適なトランザクションまたは合計注文トランザクションで有効化されたときに使用するバージョンスキームを設定します。詳細は、「 Write Skews 」のセクションを参照してください。デフォルトは **NONE** です。

Two-Phase-Commit(2PC) を Red Hat Data Grid に実装する方法と、ロックを取得する方法の詳細は、以下のセクションを参照してください。構成設定の詳細は、「 設定リファレンス 」を参照してください。

8.2. 分離レベル

Red Hat Data Grid は、READ_COMMITTED と REPEATABLE_READ の 2 つの分離レベルを提供します。

これらの分離レベルは、リーダーが同時書き込みを確認するタイミングを決定し、**MVCCEntry** の異なるサブクラスを使用して内部的に実装されます。MVCCEntry では、状態がデータコンテナーにコミットされる方法が異なります。

以下は、Red Hat Data Grid のコンテキストにおける **READ_COMMITTED** と **REPEATABLE_READ** の相違点を理解するのに役立つ詳細な例になります。**READ_COMMITTED** の場合、同じキーで連続して 2 つの読み取り呼び出しを行うと、キーが別のトランザクションによって更新され、2 つ目の読み取りによって新しい更新値が返されることがあります。

```
Thread1: tx1.begin()
Thread1: cache.get(k) // returns v
Thread2:                          tx2.begin()
Thread2:                          cache.get(k) // returns v
Thread2:                          cache.put(k, v2)
Thread2:                          tx2.commit()
Thread1: cache.get(k) // returns v2!
Thread1: tx1.commit()
```

REPEATABLE_READ では、最終 get は引き続き **v** を返します。そのため、トランザクション内で同じキーを複数回取得する場合は、**REPEATABLE_READ** を使用する必要があります。

ただし、読み取りロックが **REPEATABLE_READ** に対しても取得されないため、この現象が発生する可能性があります。

```
cache.get("A") // returns 1
cache.get("B") // returns 1

Thread1: tx1.begin()
Thread1: cache.put("A", 2)
Thread1: cache.put("B", 2)
Thread2:                          tx2.begin()
Thread2:                          cache.get("A") // returns 1
Thread1: tx1.commit()
Thread2:                          cache.get("B") // returns 2
Thread2:                          tx2.commit()
```

8.3. トランザクションのロック

8.3.1. 悲観的なトランザクションキャッシュ

ロック取得の観点では、悲観的トランザクションはキーの書き込み時にキーのロックを取得します。

1. ロック要求がプライマリー所有者に送信されます (明示的なロック要求または操作のいずれか)。
2. プライマリーの所有者はロックの取得を試みます。
    a. 成功した場合は、正の応答が返されます。

b. *そうでない場合は、負の応答が送信され、トランザクションはロールバックされます。*

たとえば、以下のようになります。

```
transactionManager.begin();
cache.put(k1,v1); //k1 is locked.
cache.remove(k2); //k2 is locked when this returns
transactionManager.commit();
```

***cache.put(k1,v1)** が返されると、**k1** はロックされ、クラスター内のどこかで実行中の他のトランザクションは、これに書き込むことができません。**k1** の読み取りは引き続き可能です。トランザクションの完了時に **k1** のロックが解放されます（コミットまたはロールバック）。*

注記

条件付き操作の場合、検証はオリジネーターで実行されます。

8.3.2. 楽観的トランザクションキャッシュ

楽観的トランザクションロックはトランザクションの準備時に取得され、トランザクションのコミット (またはロールバック) まで保持されます。これは、書き込みでローカルロックを取得し、準備中にクラスターのロックが取得される 5.0 デフォルトロックモデルとは異なります。

1. *準備はすべての所有者に送信されます。*
2. *プライマリーの所有者は、必要なロックの取得を試みます。*
   a. *ロックに成功すると、書き込みのスキューチェックが実行されます。*
   b. *書き込みスキューチェックが成功した場合 (または無効化された場合) は、正の応答を送信します。*
   c. *それ以外の場合は、負の応答が送信され、トランザクションはロールバックされます。*

たとえば、以下のようになります。

```
transactionManager.begin();
cache.put(k1,v1);
cache.remove(k2);
transactionManager.commit(); //at prepare time, K1 and K2 is locked until committed/rolled back.
```

注記

条件付きコマンドの場合、検証は引き続きオリジネーターで実行されます。

8.3.3. 悲観的または楽観的トランザクションのどちらが必要か

*ユースケースの観点からは、同時に実行されている複数のトランザクション間で多くの競合が **ない** 場合は、楽観的トランザクションを使用する必要があります。これは、読み取り時と、コミット時 (書き込みスキューチェックが有効) の間でデータが変更された場合に、楽観的トランザクションがロールバックするためです。*

一方、キーでの競合が多く、トランザクションのロールバックがあまり望ましくない場合は、悲観的トランザクションの方が適している可能性があります。悲観的トランザクションは、その性質上、よりコストがかかります。各書き込み操作ではロックの取得に RPC が関係する可能性があります。

8.4. スキューの書き込み

書き込みスキューは、2 つのトランザクションが独立して同時に同じキーの読み取りと書き込みを行うときに発生します。書き込みスキューの結果、両方のトランザクションは同じキーに対して更新を正常にコミットしますが、値は異なります。

ライブラリーモードでは、Red Hat Data Grid は自動的に書き込みスキューチェックを実行し、最適なトランザクションで **REPEATABLE_READ** 分離レベルのデータの整合性を確保します。これにより、Red Hat Data Grid はトランザクションの1 つを検出し、ロールバックすることができます。

注記

write-skew 属性はライブラリーモードで非推奨となりました。Remote Client/Server Mode では、この属性は有効な宣言ではありません。

LOCAL モードで動作する場合、書き込みスキューの確認は Java オブジェクト参照に依存して違いを比較します。これにより、書き込みスキューをチェックするための信頼性の高い技術が提供されます。

クラスター環境では、データをバージョン管理し、信頼できる書き込みのスキューチェックを行う必要があります。Red Hat Data Grid は、**SIMPLE** バージョン管理と呼ばれる **EntryVersion** インターフェースの実装を提供します。これは、エントリーが更新されるたびに増分される期間でサポートされます。

```
<versioning scheme="SIMPLE|NONE" />
```

または

```
new ConfigurationBuilder().versioning().scheme(SIMPLE);
```

8.4.1. 悲観的トランザクションでのキーへの書き込みロックの強制

pessimistic トランザクションで write-skews を回避するには、**Flag.FORCE_WRITE_LOCK** を指定して読み取り専用でキーをロックします。

注記

- トランザクション以外のキャッシュでは、**Flag.FORCE_WRITE_LOCK** は動作しません。**get()** 呼び出しは、キーの値を読み取りますが、ロックをリモートで取得しません。
- **Flag.FORCE_WRITE_LOCK** は、同じトランザクションでエンティティーが後で更新されるトランザクションと併用する必要があります。

Flag.FORCE_WRITE_LOCK の例については、以下のコードスニペットを比較してください。

```
// begin the transaction
if (!cache.getAdvancedCache().lock(key)) {
   // abort the transaction because the key was not locked
} else {
   cache.get(key);
```

```
   cache.put(key, value);
   // commit the transaction
}
```

```
// begin the transaction
try {
   // throws an exception if the key is not locked.
   cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(key);
   cache.put(key, value);
} catch (CacheException e) {
   // mark the transaction rollback-only
}
// commit or rollback the transaction
```

8.5. 例外への対処

JTA トランザクションの範囲内のキャッシュメソッドによってCacheException （またはそのサブクラス）がスローされた場合、トランザクションは自動的にロールバック用にマークされます。

8.6. 同期の登録

デフォルトでは、Red Hat Data Grid はXAResource を通じて分散トランザクションの最初のクラス参加者として登録します。Red Hat Data Grid がトランザクションの参加者である必要はありませんが、ライフサイクル（prepare、complete）によってのみ通知されます。たとえば、Red Hat Data Grid が Hibernate で 2 番目のレベルキャッシュとして使用される場合などです。

Red Hat Data Grid は、同期 を介してトランザクションエンリストメントを許可します。これを有効にするには、**NON_XA** トランザクションモードを使用します。

Synchronization には、**TransactionManager** が 1PC で 2PC を最適化できるという利点があります。この場合、他の 1 つのリソースのみがそのトランザクションにエンリストされます (last resource commit optimization)。例:Hibernate 2 レベルのキャッシュ： Red Hat Data Grid がコミット時よりも **XAResource** Manager に自らを登録すると、 **TransactionManager** は 2 つの **XAResource** （キャッシュとデータベース）を認識し、この最適化は行いません。2つのリソース間で調整する必要があるため、txログをディスクに書き込む必要があります。一方、Red Hat Data Grid を **Synchronisation** として登録すると、**TransactionManager** はディスクへのログの記述を省略します（パフォーマンス向上）。

8.7. バッチ処理

バッチ処理は、トランザクションの原子性といくつかの特性を許可しますが、本格的なJTAまたはXA機能は許可しません。多くの場合、バッチ処理は本格的なトランザクションよりもはるかに軽量で安価です。

ヒント

一般的には、トランザクションの参加者のみがRed Hat Data Grid クラスターである場合に、バッチ処理 API を使用する必要があります。反対に、トランザクションに複数のシステムが必要な場合に、（**TransactionManager**に関連する）JTA トランザクションを使用する必要があります。たとえば、トランザクションの「Hello world!」を考慮すると、ある銀行口座から別の銀行口座にお金を転送します。両方のアカウントがRed Hat Data Grid に保存されている場合は、バッチ処理を使用できます。1 つのアカウントがデータベースにあり、もう 1 つのアカウントがRed Hat Data Grid の場合は、分散トランザクションが必要になります。

注記

*バッチ処理を使用するためにトランザクションマネージャーを定義する必要は**ありません**。*

8.7.1. API

*バッチ処理を使用するようにキャッシュを設定したら、**Cache** で **startBatch()** と **endBatch()** を呼び出して使用します。例:*

```
Cache cache = cacheManager.getCache();
// not using a batch
cache.put("key", "value"); // will replicate immediately

// using a batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k2", "value");
cache.endBatch(true); // This will now replicate the modifications since the batch was started.

// a new batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false); // This will "discard" changes made in the batch
```

8.7.2. バッチ処理と JTA

*裏では、バッチ機能がJTAトランザクションを開始し、その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。これには、内部 **TransactionManager** 実装が非常に簡単な（例: リカバリーなし）を使用します。バッチ処理では、以下を取得します。*

1. *呼び出し中に取得したロックはバッチが完了するまで保持されます。*
2. *変更はすべて、バッチ完了プロセスの一部として、クラスター内でバッチ内に複製されます。バッチの各更新のレプリケーションチャット数を減らします。*
3. *同期のレプリケーションまたは無効化が使用された場合は、レプリケーション/無効化の失敗により、バッチがロールバックされます。*
4. *すべてのトランザクション関連の設定は、バッチ処理にも適用されます。*

8.8. トランザクションリカバリー

リカバリーはXA トランザクションの機能であり、リソースの不測の事態、場合によってはトランザクションマネージャーの障害を対処し、それに応じてそのような状況から回復します。

8.8.1. リカバリーを使用するタイミング

*外部データベースに保存されているアカウントから Red Hat Data Grid に保存されているアカウントに転送される分散トランザクションについて考えてみましょう。**TransactionManager.commit()** が呼び出されると、両方のリソースが正常に完了します(第1フェーズ)。コミット（2 番目）フェーズ中、トラ*

ンザクションマネージャーからコミットリクエストを受信する前に、データベースは Red Hat Data Grid の失敗時に変更を正常に適用します。この時点で、システムは一貫性のない状態にあります。これは、外部データベースのアカウントから取得されますが、Red Hat Data Grid にはまだ表示されません（2 フェーズコミットプロトコルの 2 番目のフェーズでのみロックがリリースされるため）。リカバリーはこの状況に対応し、データベースと Red Hat Data Grid の両方のデータが一貫した状態で終了します。

8.8.2. 仕組み

リカバリーはトランザクションマネージャーによって調整されます。トランザクションマネージャーは Red Hat Data Grid と連携して、手動の介入が必要なインダウト状態のトランザクションの一覧を決定し、（メール、ログアラートなどで）システム管理者に通知します。このプロセスはトランザクションマネージャーに固有のものですが、通常トランザクションマネージャーで設定が必要になります。

インダウト状態のトランザクション ID を把握すると、システム管理者は Red Hat Data Grid クラスターに接続し、トランザクションのコミットを再生したり、ロールバックを強制的に実行したりできます。Red Hat Data Grid は、この JMX ツールを提供します。これについては、「トランザクションリカバリーおよび調整」セクションで詳細に説明しています。

8.8.3. リカバリーの設定

Red Hat Data Grid では、リカバリーはデフォルトでは有効になっていません。無効にすると、**TransactionManager** は Red Hat Data Grid と連携してインダウトトランザクションを判断できません。トランザクションの設定 セクションでは、その設定を有効にする方法を示しています。

注記: **recovery-cache** 属性は必須ではなく、キャッシュごとに設定されます。

注記

リカバリーが機能するには、完全な XA トランザクションが必要であるため、**mode** を **FULL_XA** に設定する必要があります。

8.8.3.1. JMX サポートの有効化

リカバリー JMX サポートの管理に JMX を使用できるようにするには、明示的に有効にする必要があります。管理の章で JMX の有効化に関する詳細は、「管理」の章で JMX を有効にする手順を参照してください。

8.8.4. リカバリーキャッシュ

インダウト状態のトランザクションを追跡し、応答できるようにするために、Red Hat Data Grid は今後使用するためにすべてのトランザクション状態をキャッシュします。この状態は、未確定のトランザクションに対してのみ保持され、コミット/ロールバックフェーズが完了した後、正常に完了したトランザクションに対しては削除されます。

この未確定のトランザクションデータはローカルキャッシュ内に保持されます。これにより、データが大きくなりすぎた場合に、キャッシュローダーを介してこの情報をディスクにスワップするように構成できます。このキャッシュは、**recovery-cache**設定属性を介して指定できます。指定されていない場合、Red Hat Data Grid はローカルキャッシュを設定します。

（必須ではない）リカバリーが有効になっているすべての Red Hat Data Grid キャッシュ間で、同じリカバリーキャッシュを共有することができます（必須ではありません）。デフォルトのリカバリーキャッシュが上書きされた場合、指定のリカバリーキャッシュは、キャッシュ自体によって使用されるものとは異なるトランザクションマネージャーを返す TransactionManagerLookup を使用する必要があります。

8.8.5. トランザクションマネージャーとの統合

これはトランザクションマネージャーに固有のものですが、通常トランザクションマネージャーは **XAResource.recover()** を呼び出すために **XAResource** 実装への参照が必要になります。Red Hat Data Grid **XAResource** への参照を取得するには、以下の API を使用できます。

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

トランザクションを実行するプロセスとは異なるプロセスで復元を実行することが一般的です。

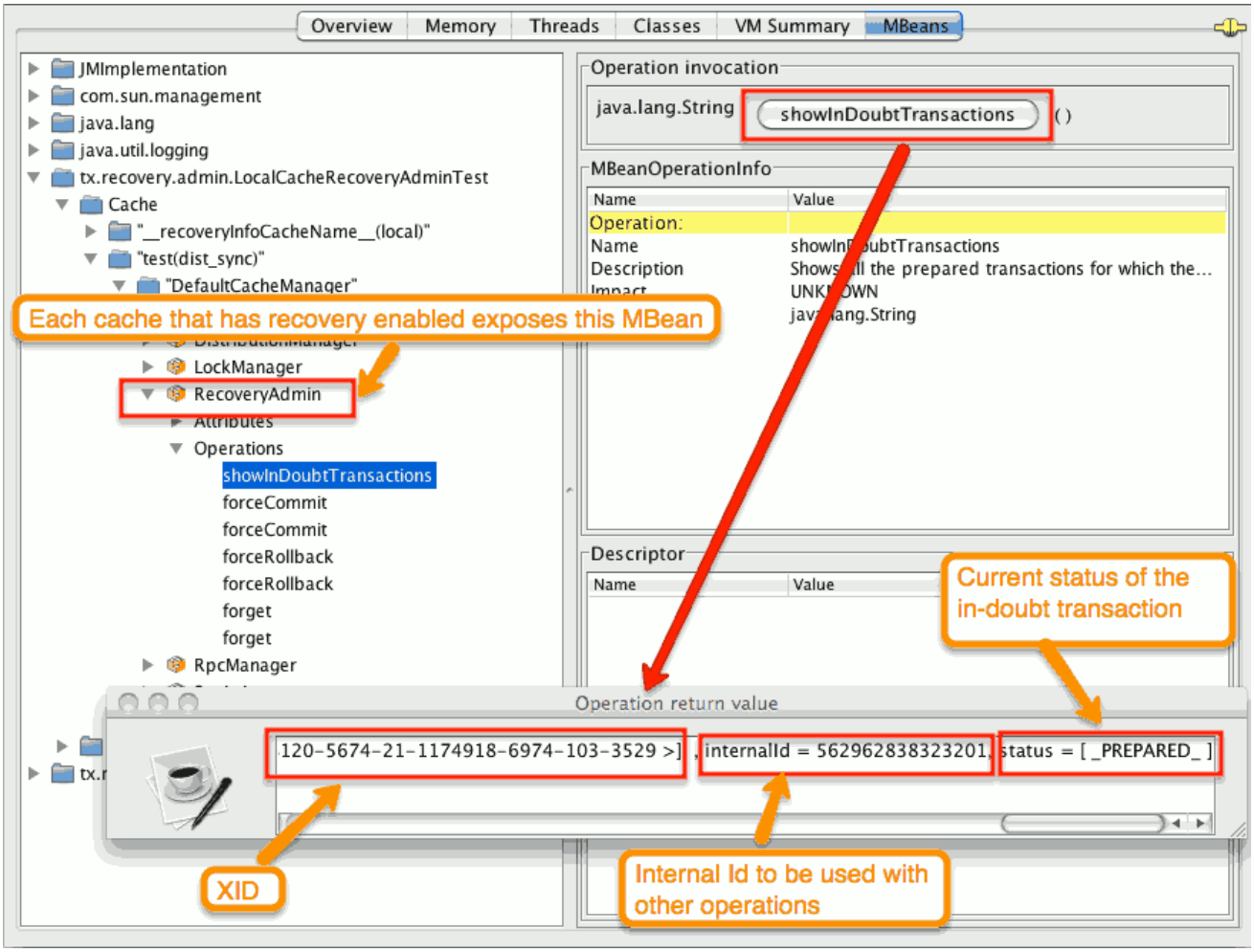
8.8.6. 調整

トランザクションマネージャは、システム管理者に未確定のトランザクションについて独自の方法で通知します。この段階では、システム管理者がトランザクションの XID（バイトアレイ）を把握していることを前提としています。

通常のリカバリーフローは以下のとおりです。

- **STEP 1**: システム管理者は JMX 経由で Red Hat Data Grid サーバーに接続し、インダウト状態のトランザクションを一覧表示します。以下の図は、JConsole がインダウト状態のトランザクションがある Red Hat Data Grid ノードに接続する方法を示しています。

図8.1 未確定のトランザクションの表示



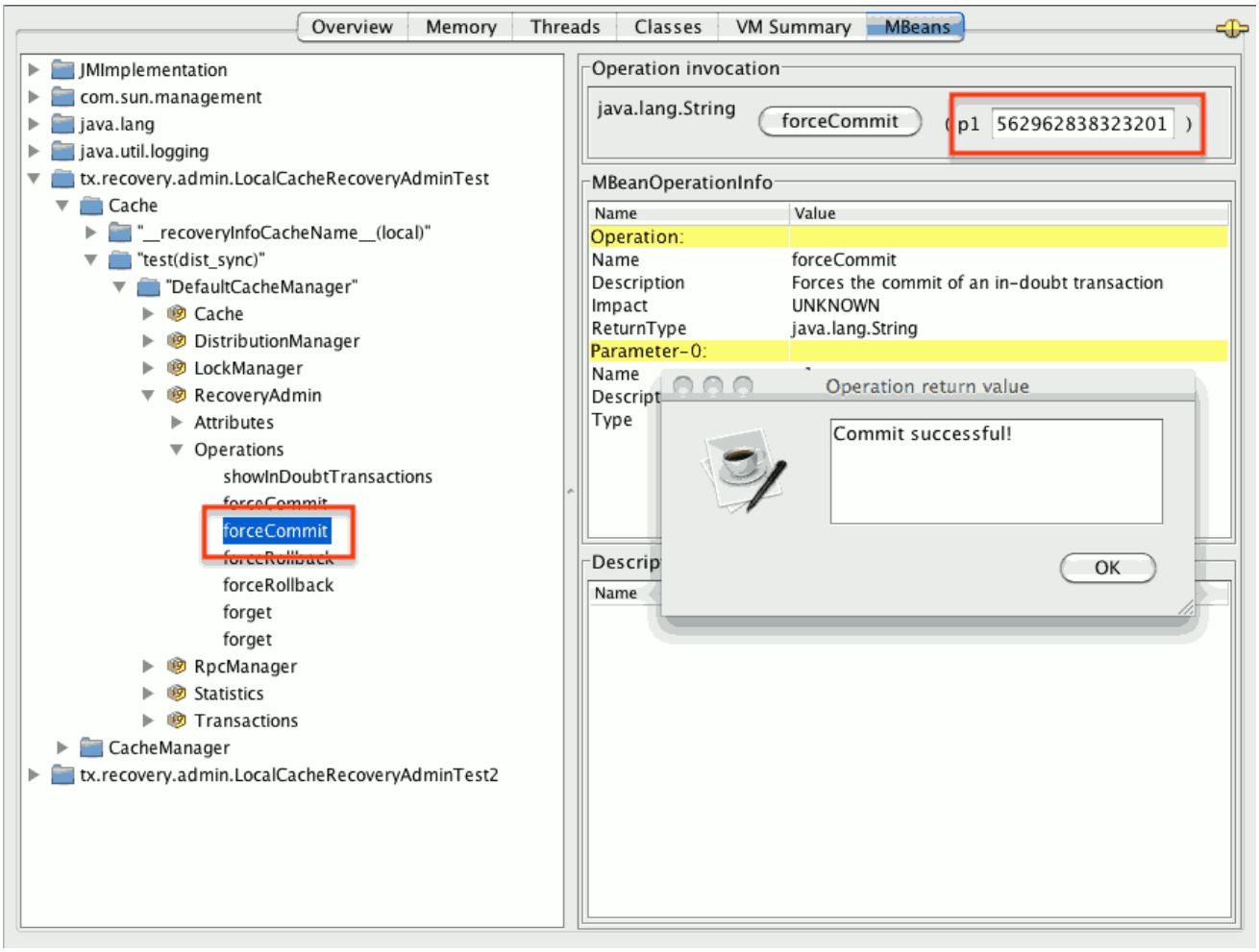
未確定の各トランザクションのステータスが表示されます（この例では **PREPARED** です）。status フィールドに複数の要素が存在する可能性があります。たとえば、トランザクションが特定ノードでコミットされていても、それらのノードでコミットされない場合は「PREPARED」および

「COMMITTED」です。

- *STEP 2: システム管理者は、トランザクションマネージャーから受け取った XID を、数字で表される Red Hat Data Grid 内部 ID に視覚的にマップします。この手順は、XID（バイト配列）が JMX ツール（JConsole など）に渡らず、Red Hat Data Grid 側で再度構築できないため、必要です。*

- *ステップ 3: システム管理者は、内部 ID に基づいて、対応する jmx 操作を介してトランザクションのコミット/ロールバックを強制的に実行します。以下のイメージは、内部 ID に基づいてトランザクションのコミットを強制することで取得します。*

図8.2 コミットの強制



ヒント

上記のすべての JMX 操作は、トランザクションの発信場所に関係なく、任意のノードで実行できます。

8.8.6.1. XID に基づくコミット/ロールバックの強制

未確定のトランザクションのコミット/ロールバックの強制を行う XID ベースの JMX 操作も使用できます。これらのメソッドはトランザクションに関連する番号ではなく、XID を記述する byte[] アレイを受け取ります（前述のステップ 2 で説明）。これらは、たとえば、特定の未確定トランザクションの自動完了ジョブを設定する場合に役立ちます。このプロセスはトランザクションマネージャーのリカバリーにプラグインされ、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。

8.8.7. 詳細

リカバリー設計ドキュメントでは、トランザクションリカバリー実装の内部について詳しく説明しています。

8.9. 順序ベースのコミットプロトコルの合計

Total Order ベースのプロトコルは、マルチマスタースキームです（このコンテキストでは、マルチマスタースキームは、すべてのノードが Red Hat Data Grid に実装される(optimistic/pessimist)ロックモードとしてすべてのデータを更新できることを意味します。このコミットプロトコルは、メッセージを完全に順序付けされた配信の概念に依存します。つまり、メッセージセットを提供する各ノードが同じ順序で配信されることを意味します。

このプロトコルには、この利点があります。

1. トランザクションは受信順に同じ順序で配信されるため、1 つのフェーズでコミットできます。
2. 分散デッドロックを軽減します。

このアプローチの弱い点は、トランザクションと変更を提供するノードごとに単一のスレッドに依存すること、および **Transport** でメッセージの合計順序付けのわずかコストに依存することです。

したがって、このプロトコルは、**競合が高い** シナリオで最適なパフォーマンスを提供します。これにより、1 フェーズコミットの利点が得られます。また、配信スレッドはボトルネックではありません。

現在、Total Order based プロトコルは、**レプリケート** モードおよび **分散** モードの トランザクションキャッシュでのみ利用できます。

8.9.1. 概要

Total Order based commit プロトコルは、Red Hat Data Grid がトランザクションをコミットする方法と、分離レベルと書き込みが動作に影響を与える方法にのみ影響します。

書き込みスキューが無効になっていると、トランザクションは単一フェーズでコミットまたはロールバックできます。データの一貫性は、キーのすべての所有者が同じ順序で設定された同じトランザクションを提供するように保証されます。

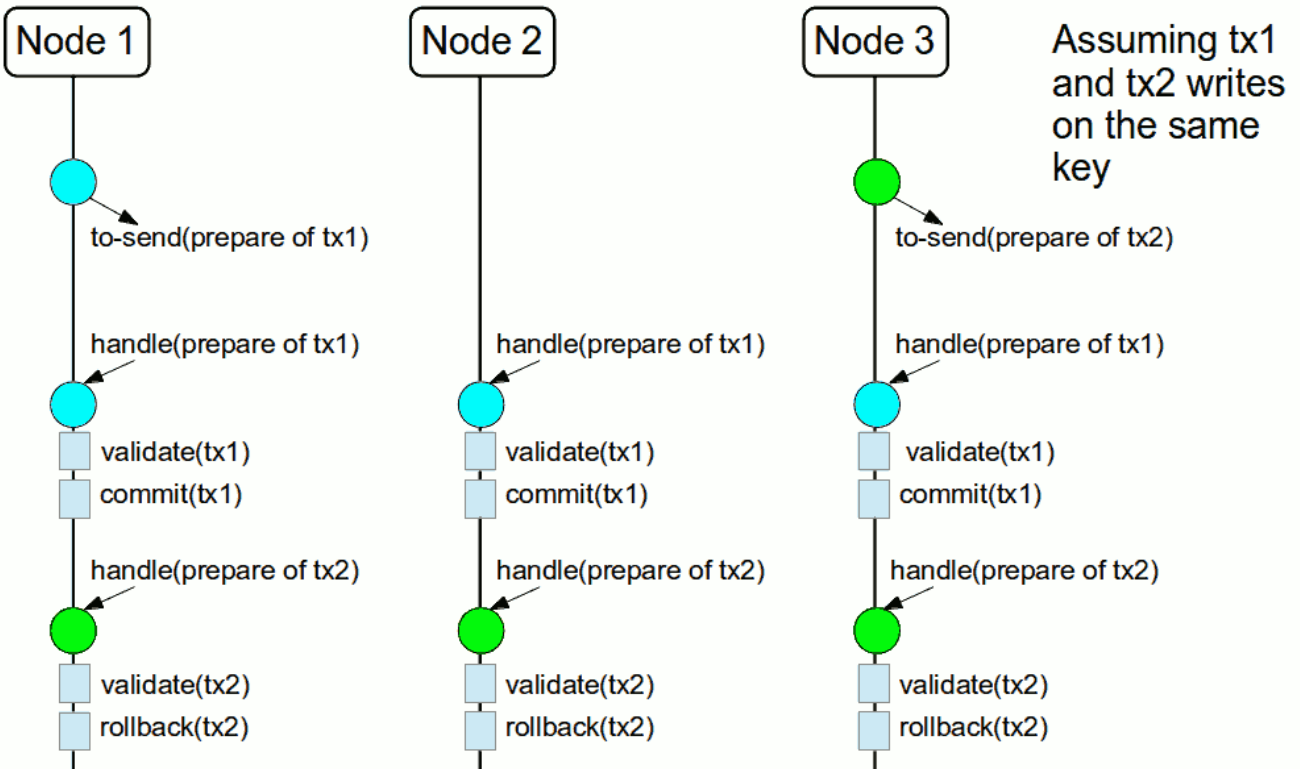
一方、書き込みスキューが有効になっている場合、プロトコルは適応し、安全であれば 1 フェーズコミットを使用します。XaResource エンリストメントでは、TransactionManager が 1 つのフェーズでコミットを要求する（最後のリソースコミット最適化）と Red Hat Data Grid キャッシュがレプリケートモードで設定されている場合は、1 つのフェーズを使用できます。各ノードが異なるキーサブセットで書き込みスキューの検証を実行するため、この最適化は分散モードでは安全ではありません。同期 エンリストメントでは、Red Hat Data Grid がリソース登録された唯一のリソース（最後のリソースコミットの最適化）である場合、TransactionManager は情報を提供しないため、1 つのフェーズでコミットすることはできません。

8.9.1.1. 1 フェーズでのコミット

トランザクションが終了すると、Red Hat Data Grid はトランザクション（およびその変更）を合計順序で送信します。これにより、すべてのトランザクションが関連するすべての Red Hat Data Grid ノードで同じ順序で配信されるようになります。その結果、トランザクションが配信されると、同じ状

態（有効な場合）で決定論的な書き込みスキューチェックを実行し、その結果（トランザクションのコミットまたはロールバック）を行います。

図8.3 1 フェーズコミット

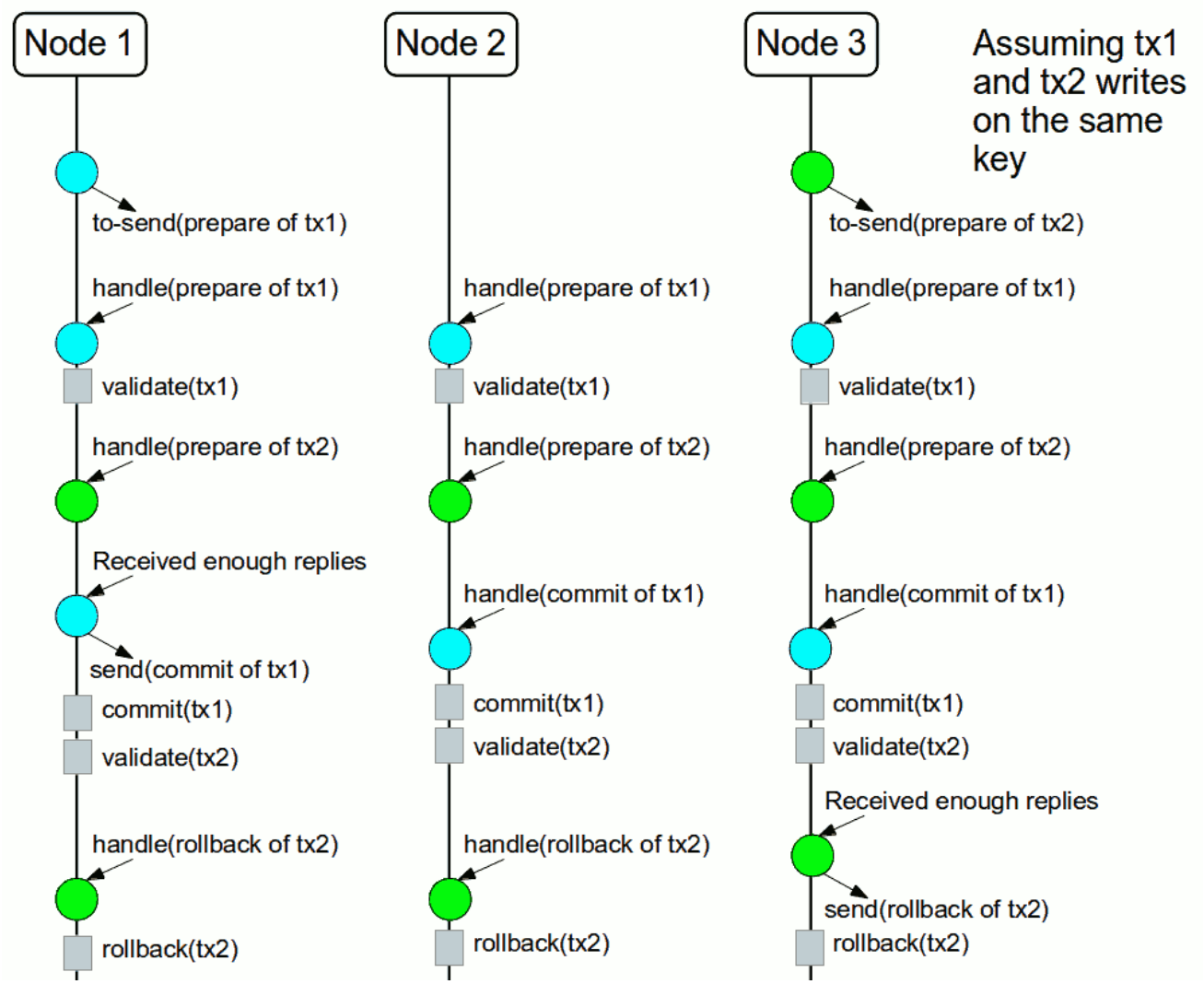


上の図は、3 つのノードで構成される高レベルの例を示しています。Node1 および Node3 は各トランザクションを 1 つ実行しており、両方のトランザクションが同じキーに書き込むことを仮定します。より複雑なものにするために、両方のノードが、最初の色付きの円で表されているように、両方のノードが同時にコミットしようとすることを仮定します。青い 円はトランザクション tx1 とトランザクション tx2 を表します。両方のノードは、トランザクションの変更とともに合計順序（送信）でリモート呼び出しを行います。この時点で、すべてのノードは同じ配信順序で合意します（例： tx1 の後に tx2 が続きます）。次に、各ノードは tx1 を提供し、検証を実行し、変更をコミットします。tx2 に同じ手順が実行されますが、この場合検証に失敗し、トランザクションは関係するすべてのノードでロールバックされます。

8.9.1.2. 2 つのフェーズでのコミット

最初のフェーズでは、変更を合計順序で送信し、書き込みスキューチェックが実行されます。書き込みスキューチェックの結果が起点に返されます。すべてのキーが正常に検証されたことを確認するとすぐに、TransactionManager への適切な応答が提供されます。一方、負の応答を受信すると、TransactionManager への負の応答が返されます。最後に、TransactionManager リクエストに応じて、トランザクションは 2 番目のフェーズでコミットまたは中止されます。

図8.4 2 フェーズコミット



上の図では、1 番目の図で説明されているシナリオを示していますが、2 つのフェーズを使用してトランザクションをコミットするようになりました。tx1 が配信されると、検証を実行し、TransactionManager に返信します。次に、TransactionManager が tx1 の 2 番目のフェーズを要求する前に tx2 が配信されていることを前提としています。この場合、tx2 はエンキューされ、tx1 が完了した場合にのみ検証されます。最終的に、tx1 の TransactionManager は 2 番目のフェーズ（コミット）をリクエストし、すべてのノードが tx2 の検証を自由に実行できます。

8.9.1.3. トランザクションリカバリー

現時点で、トランザクションリカバリー は Total Order ベースのコミットプロトコルでは使用できません。

8.9.1.4. 状態遷移

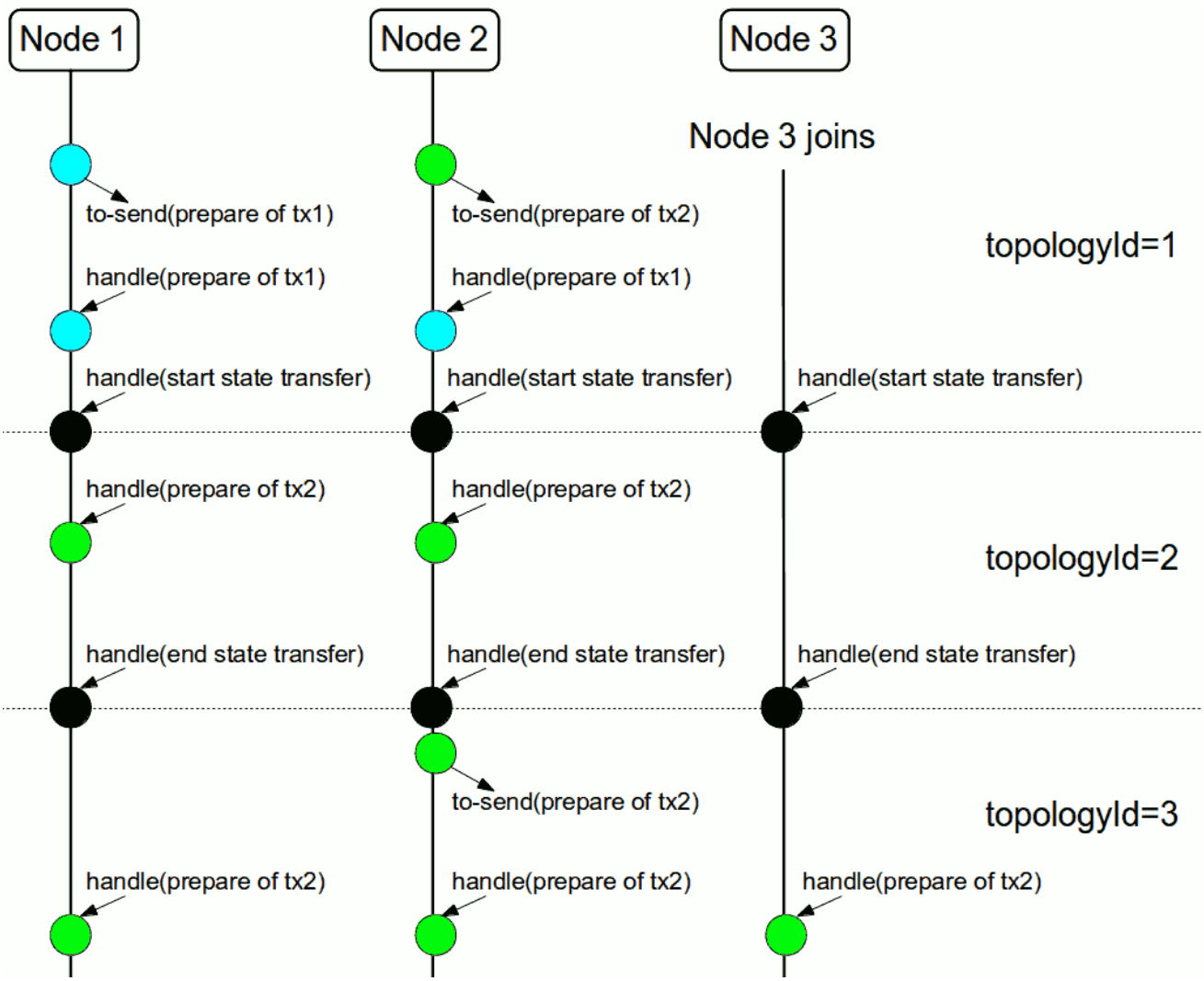
分かりやすくするために、合計順序ベースのコミットプロトコルは、現在の状態遷移のブロッキングバージョンを使用します。主な違いは以下のとおりです。

1. *状態遷移中トランザクションの配信をキューに入れます。*

2. *状態転送制御メッセージ(CacheTopologyControlCommand)は合計順序で送信されます。*

これにより、状態遷移とトランザクション間の同期が可能になり、同じすべてのノードであるトランザクションが提供されます。しかし、トランザクションは状態遷移の途中にあります（状態遷移の開始前に送信され、その後配信されます）は、新しい参加者に関連する新しい合計注文を見つけるために再送する必要があります。

図8.5 トランザクション中のノードの参加



上記の図は、ノードの参加について示しています。このシナリオでは、tx2 は topologyId=1 で送信されますが、受信されると topologyId= 2 になります。そのため、トランザクションは新規ノードが関係します。

8.9.2. 設定

キャッシュで合計順序を使用するには、TOA プロトコルを jgroups.xml 設定ファイルに追加する必要があります。

jgroups.xml

```
<tom.TOA />
```

注記

詳細は、JGroups マニュアル を参照してください。

注記

JGroups による合計順序の保証方法に関心がある場合は、link::http://jgroups.org/manual/index.html#TOA[TOA マニュアル] を確認してください。

また、トランザクション設定 に示されるように、<transaction> 要素に protocol=TOTAL_ORDER を 設定 する必要があります。

8.9.3. 使用のタイミング

合計順序は、書き込み負荷の高いワークロードや、内容の高いワークロードで使用される場合の利点を示しています。ロックキーの競合を回避します。

第9章 ロックおよび同時実行

Red Hat Data Grid では、マルチバージョンコンカレンシー制御(MVCC)が使用されます。これは、リレーショナルデータベースやその他のデータストアで人気のある同時実行スキームです。MVCC には、粗粒度の Java 同期や、共有データにアクセスするための JDK ロックに比べて、次のような多くの利点があります。

- 同時リーダーとライターの許可
- リーダーとライターが互いにブロックしない
- 書き込みスキューを検出して処理できる
- 内部ロックのストライピングが可能

9.1. 実装の詳細のロック

Red Hat Data Grid の MVCC 実装では、最小限のロックと同期が使用され、可能な限り比較やスワップやロックのないデータ構造などのロックのない手法を多用します。これにより、マルチ CPU やマルチコア環境の最適化に役立ちます。

特に、Red Hat Data Grid の MVCC 実装はリーダー向けに大幅に最適化されています。リーダースレッドは、エントリーの明示的なロックを取得せず、代わりに問題のエントリーを直接読み込みます。

一方、ライターは、書き込みロックを取得する必要があります。これにより、エントリーごとに 1 つの同時書き込みのみが保証されるため、同時ライターはキューイングしてエントリーを変更することになります。同時読み取りを可能にするため、ライターはエントリーを MVCCEntry でラップして、変更する予定のエントリのコピーを作成します。このコピーは、同時リーダーが部分的に変更された状態を認識できないようにします。書き込みが完了したら、MVCCEntry.commit() はデータコンテナーへの変更をフラッシュし、後続のリーダーに変更内容が反映されます。

9.1.1. クラスター化されたキャッシュでの仕組み

クラスター化されたキャッシュでは、各キーにキーをロックするノードがあります。このノードはプライマリー所有者と呼ばれます。

9.1.1.1. 非トランザクションキャッシュ

1. 書き込み操作はキーのプライマリー所有者に送信されます。
2. プライマリー所有者はキーをロックしようとします。
   a. 成功すると、操作が他の所有者に転送されます。
   b. そうでない場合は、例外が発生します。

注記

操作が条件付きで、プライマリオーナーで失敗した場合、他のオーナーには転送されません。

注記

操作がプライマリ所有者でローカルに実行される場合、最初のステップはスキップされます。

9.1.2. トランザクションキャッシュ

トランザクションキャッシュは、楽観的および悲観的ロックモードをサポートします。詳細は、「トランザクションロック」のセクションを参照してください。

9.1.3. 分離レベル

分離レベルは、他のトランザクションと同時に実行されているときに読み取ることができるトランザクションに影響します。詳細は、「分離レベル」のセクションを参照してください。

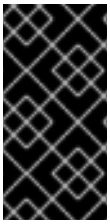
9.1.4. LockManager

LockManager は、書き込み用にエントリーをロックするコンポーネントです。LockManager は、LockContainer を使用して、ロックを検索、保持、作成します。LockContainers には、ロックストライピングをサポートするものと、エントリーごとに 1 つのロックをサポートするものの 2 つの大まかな特徴があります。

9.1.5. ロックストライピング

ロックストライピングでは、固定サイズの共有ロックコレクションをキャッシュ全体に使用する必要があり、ロックはエントリーのキーのハッシュコードに基づいてエントリーに割り当てられます。JDK の ConcurrentHashMap がロックを割り当てる方法と同様に、これにより、関連性のない可能性のあるエントリが同じロックによってブロックされる代わりに、拡張性の高い固定オーバーヘッドのロックメカニズムが可能になります。

別の方法は、ロックストライピングを無効にすることです。これは、エントリーごとに新しい ロックが作成されることを意味します。このアプローチでは、スループットが高くなる 可能性 がありますが、追加のメモリー使用量やガベージコレクションのチャーンなどのコストがかかります。



デフォルトのロックストライピング設定

異なるキーのロックが同じロックストライプになってしまうとデッドロックが発生する可能性があるため、ロックストライピングはデフォルトで無効になっています。

ロックストライピングによって使用される共有ロックコレクションのサイズは、'<locking /> 設定要素の concurrencyLevel 属性を使用して調整できます。

設定例:

```
<locking striping="false|true"/>
```

または

```
new ConfigurationBuilder().locking().useLockStriping(false|true);
```

9.1.6. 同時実行レベル

この同時実行レベルは、ストライプロックコンテナーのサイズを決定する他に、DataContainer の内部など、関連する JDK ConcurrentHashMap ベースのコレクションを調整するためにも使用されます。コンカレンシーレベルの詳細な説明は JDK ConcurrentHashMap Javadocs を参照してください。このパラメーターは Red Hat Data Grid でまったく同じ方法で使用されるためです。

設定例:

```
<locking concurrency-level="32"/>
```

-

または

```
new ConfigurationBuilder().locking().concurrencyLevel(32);
```

9.1.7. ロックタイムアウト

ロックタイムアウトは、競合するロックを待つ時間 (ミリ秒単位) を指定します。

設定例:

```
<locking acquire-timeout="10000"/>
```

または

```
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10000);
//alternatively
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10, TimeUnit.SECONDS);
```

9.1.8. 一貫性

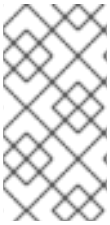
(すべての所有者がロックされているのとは対照的に) 単一の所有者がロックされるという事実により、次の一貫性の保証が失われることはありません。キー K がノード {A, B} に対してハッシュ化され、トランザクション TX1 が、たとえば、A 上の K のロックを取得したとします。別のトランザクション TX2 が B (またはその他のノード) 上で開始され、TX2 が K のロックを試みる場合、ロックがすでに TX1 によって保持されているため、タイムアウトでロックに失敗します。理由は、キー K のロックがトランザクションの発生場所に関係なく、常に、確定的に、クラスターの同じノードで取得されるからです。

9.2. データのバージョン管理

Red Hat Data Grid は、シンプルと外部という 2 つの形式のデータバージョン管理をサポートします。simple バージョン管理は、書き込みスキューチェックのトランザクションキャッシュで使用されます。Write Skews を参照してください。

外部バージョン管理は、Red Hat Data Grid 内でデータソースの外部ソースをカプセル化するために使用されます。

このスキームでは、バージョンに渡すメカニズムが必要になり、オーバーロードされたバージョン put() および putForExternalRead() が、AdvancedCache で提供され、外部データバージョンを取り込みます。その後、これは InvocationContext に保管され、コミット時にエントリーに適用されます。



注記

external バージョン管理の場合、書き込みスキューチェックは実行できず、実行されません。

第10章 グリッドでのコードの実行

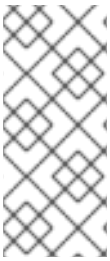
キャッシュの主な利点は、マシン全体でもキーで値を迅速に検索できることです。実際、これはおそらく多くのユーザーが Red Hat Data Grid を使用する理由です。ただし、Red Hat Data Grid はすぐに理解できない多くの利点を提供できます。通常、Red Hat Data Grid はマシンのクラスターで使用されるため、ユーザーに適したワークロードを実行するためのクラスター全体での活用に役立つ機能もあります。

注記

このセクションでは、埋め込みキャッシュを使用したグリッドでのコードの実行のみを説明します。リモートキャッシュを使用している場合は、Remote Grid の Executing コードを 確認する必要があります。

10.1. クラスターエグゼキューター

マシンのグループがあるため、それらすべてでコードを実行するためにそれらの結合された計算能力を活用することは理にかなっています。キャッシュマネージャーには、クラスター内で任意のコードを実行できる優れたユーティリティーが付属しています。この機能にはキャッシュを使用する必要はありません。この Cluster Executor は、EmbeddedCacheManager で executor（）を呼び出すことで取得できます。このエグゼキュータは、クラスター構成と非クラスター構成の両方で取得できます。



注記

ClusterExecutorは、コードがキャッシュ内のデータに依存しないコードを実行するために特別に設計されており、代わりに、ユーザーがクラスター内でコードを簡単に実行できるようにする方法として使用されます。

このマネージャーは、Java 8を使用して特別に構築されており、機能的なAPIを念頭に置いているため、すべてのメソッドは機能的なインターフェイスを引数として取ります。また、これらの引数は他のノードに送信されるため、シリアライズする必要があります。ラムダがすぐに Serializable になるような策を使用しています。つまり、引数に Serializable と実際の引数タイプ（つまり、Runnable または Function）の両方を実装させることです。JRE は、呼び出す方法を決定する際に最も具体的なクラスを選択するため、ラムダは常にシリアライズ可能です。また、Externalizer を使用してメッセージサイズをさらに減らすこともできます。

マネージャーはデフォルトで、指定されたコマンドを、送信元のノードを含むクラスター内のすべてのノードに送信します。セクションで説明されているように、filterTargets メソッドを使用して、タスクが実行するノードを制御できます。

10.1.1. 実行ノードのフィルタリング

コマンドを実行するノードを制限できます。たとえば、同じラック内のマシンでのみ計算を実行したい場合があります。または、ローカルサイトで 1 回、別のサイトで操作を再実行することもできます。クラスターエグゼキューターは、同じマシン、ラック、またはサイトレベルのスコープで要求を送信するノードを制限できます。

SameRack.java

```
EmbeddedCacheManager manager = ...;
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_RACK).submit(...)
```

このトポロジーベースのフィルタリングを使用するには、Server Hinting でトポロジーを認識しているハッシュを有効にする必要があります。

ノードの Address に基づいて述部を使用してフィルタリングすることもできます。これは任意で、以前のコードスニペットでトポロジーベースのフィルタリングと組み合わせることもできます。

また、実行対象と見なすことができるノードを除外する Predicate を使用して、任意の方法でターゲットノードを選択することもできます。これは同時に Topology フィルタリングと組み合わせて、クラスター内でコードを実行する場所をより詳細に制御できるようにすることもできます。

Predicate.java

```
EmbeddedCacheManager manager = ...;
// Just filter
manager.executor().filterTargets(a -> a.equals(..)).submit(...)
// Filter only those in the desired topology
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_SITE, a ->
a.equals(..)).submit(...)
```

10.1.2. タイムアウト

Cluster Executor を使用すると、呼び出しごとにタイムアウトを設定できます。デフォルトは、Transport Configuration で設定された分散同期のタイムアウトになります。このタイムアウトは、ク

ラスター化されたキャッシュマネージャーとクラスター化されていないキャッシュマネージャーの両方で機能します。タイムアウトの期限が切れると、エグゼキューターがタスクを実行しているスレッドを中断する場合と中断しない場合があります。ただし、タイムアウトが発生すると、Consumer または Future はTimeoutException を渡して完了します。この値は、タイムアウト メソッドを公開して必要な期間を指定することで上書きできます。

10.1.3. 単一ノードの提出

Cluster Executorは、すべてのノードにコマンドを送信する代わりに、単一ノード送信モードで実行することもできます。代わりに、通常はコマンドを受信するノードの1つを選択し、1つだけに送信します。それぞれの送信は、別のノードを使用してタスクが実行される可能性があります。これは、ClusterExecutor が実装する java.util.concurrent.Executor として ClusterExecutor を使用するのが非常に便利です。

SingleNode.java

```
EmbeddedCacheManager manager = ...;
manager.executor().singleNodeSubmission().submit(...)
```

10.1.3.1. Failover

シングルノード送信で実行する場合は、コマンドを再試行することにより、特定のコマンドの処理中に例外が発生した場合にClusterExecutorが処理できるようにすることが望ましい場合があります。これが発生すると、Cluster Executor は単一のノードを再度選択し、任意のフェイルオーバー試行までコマンドを再実行します。選択したノードは、トポロジーまたは述部のチェックをパスするノードである可能性があることに注意してください。フェイルオーバーは、上書きされた singleNodeSubmission メソッドを呼び出すことによって有効にされます。指定されたコマンドは、コマンドが例外なく完了するか、送信の合計量が指定されたフェイルオーバーカウントと等しくなるまで、単一のノードに再送信されます。

10.1.4. 例: PI アプローチ

この例は、ClusterExecutor を使用して PI の値を見積もる方法を示しています。

Pi近似は、ClusterExecutorを介した並列分散実行から大きな利点を得ることができます。正方形の面積はSa = 4r2であり、円の面積はCa=pi*r2であることを思い出してください。2 つ目の式からの r2 を置き換えると、pi = 4 * Ca/S になります。ここで、正方形に非常に多くのダーツを射ることができると仮定して、射ったダーツの総数に対して円の中に入ったダーツの割合を取ると、Ca/Saの値が近似します。pi = 4 * Ca/Sa であるため、piの近似値を簡単に導き出すことができます。ダーツを多く撃つほど、より良い近似が得られます。以下の例では、「Shooting」ではなく、Red Hat Data Grid クラス

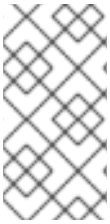
ター全体でのダイアピングの作業を並列化しています。これは 1 のクラスターで正常に機能しますが、遅くなることに注意してください。

```
public class PiAppx {

  public static void main (String [] arg){
    EmbeddedCacheManager cacheManager = ..
    boolean isCluster = ..

    int numPoints = 1_000_000_000;
    int numServers = isCluster ? cacheManager.getMembers().size() : 1;
    int numberPerWorker = numPoints / numServers;

    ClusterExecutor clusterExecutor = cacheManager.executor();
    long start = System.currentTimeMillis();
    // We receive results concurrently - need to handle that
    AtomicLong countCircle = new AtomicLong();
    CompletableFuture<Void> fut = clusterExecutor.submitConsumer(m -> {
      int insideCircleCount = 0;
      for (int i = 0; i < numberPerWorker; i++) {
        double x = Math.random();
        double y = Math.random();
        if (insideCircle(x, y))
          insideCircleCount++;
      }
      return insideCircleCount;
    }, (address, count, throwable) -> {
      if (throwable != null) {
        throwable.printStackTrace();
        System.out.println("Address: " + address + " encountered an error: " + throwable);
      } else {
        countCircle.getAndAdd(count);
      }
    });
    fut.whenComplete((v, t) -> {
      // This is invoked after all nodes have responded with a value or exception
      if (t != null) {
        t.printStackTrace();
        System.out.println("Exception encountered while waiting:" + t);
      } else {
        double appxPi = 4.0 * countCircle.get() / numPoints;

        System.out.println("Distributed PI appx is " + appxPi +
            " using " + numServers + " node(s), completed in " + (System.currentTimeMillis() -
start) + " ms");
      }
    });

    // May have to sleep here to keep alive if no user threads left
  }

  private static boolean insideCircle(double x, double y) {
    return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))
```

```
            <= Math.pow(0.5, 2);
      }
   }
```

10.2. ストリーム

結果を生成するために、キャッシュ内のサブセットまたはすべてのデータを処理したい場合があります。これにより、マップの削減が可能になります。Red Hat Data Grid を使用すると、ユーザーは非常に似ていますが、標準の JRE API を使用してこれを行うことができます。Java 8 では、ユーザーがデータに対して処理を細かく反復するのではなく、コレクションで機能スタイルの操作を可能にする ストリーム の概念が導入されました。ストリーム操作は、MapReduce と似た方法で実装できます。MapReduce と同様、キャッシュ全体で処理を実行できますが、非常に大きなデータセットになりますが、効率的な方法になります。



注記

ストリームは、クラスタートポロジーの変更に自動的に調整されるため、キャッシュに存在するデータを扱う場合に推奨される方法です。

また、エントリの反復方法を制御できるため、クラスター全体ですべての操作を同時に実行する場合は、分散されたキャッシュで操作をより効率的に実行できます。

ストリームは、stream または parallelStream メソッドを呼び出して、Cache から返される entrySet、keySet、または values コレクションから取得されます。

10.2.1. 一般的なストリーム操作

本セクションでは、使用している基礎となるキャッシュの種類に関係なく、さまざまなオプションを説明します。

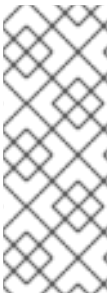
10.2.2. キーのフィルタリング

特定のキーのサブセットでのみ動作するようにストリームをフィルターできます。そのためには、CacheStream で filterKeys メソッドを呼び出します。これは常に述部 フィルター で使用する必要があります。述部がすべてのキーを保持する場合はより高速になります。

AdvancedCache インターフェースに精通している場合は、この keyFilter で getAll を使用する理由を把握している可能性があります。エントリをそのまま必要とし、それらすべてをローカルノードのメモリに必要とする場合、getAllを使用することにはいくつかの小さな利点（ほとんどの場合ペイロー

ドが小さい）があります。ただし、これらの要素で処理を行う必要がある場合は、分散並列処理とスレッド並列処理の両方を無料で取得できるため、ストリームをお勧めします。

10.2.3. セグメントベースのフィルタリング



注記

これは高度な機能であるため、Red Hat Data Grid セグメントおよびハッシュ技術の深い知識でのみ使用する必要があります。これらのセグメントベースのフィルタリングは、データを個別の呼び出しに分割する必要がある場合に便利です。これは、Apache Spark などの他のツールと統合する際に便利です。

このオプションは、レプリケートされたキャッシュと分散されたキャッシュでのみサポートされます。これにより、ユーザーは KeyPartitioner によって決定されるデータのサブセットに対して操作できます。セグメントは、CacheStream で filterKeySegments メソッドを呼び出すことでフィルターできます。これは、キーフィルターの後に、中間操作が実行される前に適用されます。

10.2.4. ローカル/無効化

ローカルキャッシュまたは無効化キャッシュで使用されるストリームは、通常のコレクションでストリームを使用する場合とまったく同じように使用できます。Red Hat Data Grid は、背後で必要時にすべての翻訳を処理し、より興味深いすべてのオプション（storeAsBinary、互換性モード、キャッシュローダー）と連携します。ストリーム操作が実行されるノードにローカルデータのみが使用されます。たとえば、無効化はローカルエントリーのみを使用します。

10.2.5. 例

以下のコードはキャッシュを取得し、値に "JBoss" の文字列が含まれるすべてのキャッシュエントリーを持つマップを返します。

```
Map<Object, String> jbossValues = cache.entrySet().stream()
        .filter(e -> e.getValue().contains("JBoss"))
        .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

10.3. 配布/複製/散在

これは、ストリームがストライドになるところです。ストリーム操作が実行されると、関連データを持つ各ノードにさまざまな中間操作と端末操作が送信されます。これにより、データを所有するノードで中間値を処理し、最終結果を元のノードにのみ送信し、パフォーマンスが向上します。

10.3.1. 再ハッシュ対応

内部的にはデータがセグメント化され、各ノードはプライマリー所有者として所有するデータでのみ操作を実行します。これにより、セグメントが各ノードで等量のデータを提供するのに十分な粒度であると仮定して、データを均等に処理できます。

分散キャッシュを使用する場合には、新規ノードが加わったり、残ったりすると、データをノード間で再シャッフルすることができます。分散ストリームはこのデータの再シャッフルを自動的に処理するため、ノードがクラスターを離れたり、クラスターに参加したりするときの監視について心配する必要はありません。シャッフルされたエントリーは 2 回処理される可能性があり、重複処理の量を制限するために、キーレベルまたはセグメントレベル（端末操作に応じて）で処理されたエントリを追跡します。

ストリームで再ハッシュ認識を無効にすることは可能ですが、推奨されません。これは、再ハッシュが発生したときに、リクエストがデータのサブセットの確認を処理できる場合に限り考慮する必要があります。これは、cache Stream.disableRehashAware（） を呼び出すことで実行できます。rehash が発生しない場合のパフォーマンスは、完全に無視できません。唯一の例外は、処理されたキーを追跡する必要がないため、使用するメモリが少ないiteratorとforEachです。

警告

自分が何をしているかを本当に理解していない限り、再ハッシュ認識を無効にすることを再考してください。

10.3.2. シリアル化

操作は他のノードに送信されるため、Red Hat Data Grid のマーシャリングでシリアライズ可能でなければなりません。これにより、他のノードに操作を送信できます。

最も簡単な方法は、CacheStreamインスタンスを使用し、通常どおりラムダを使用することです。Red Hat Data Grid は、さまざまな Stream の中間メソッドおよび端末メソッドをすべてオーバーライドし、引数の pidFunction(ie. gitopsFunction, 336Predicate...)を取り出します。これらのメソッドは CacheStream で見つけることができます。これは、ここで定義されている最も具体的な方法を選択するための仕様に依存しています。

上記の例では、Collector を使用してすべての結果を Map に収集しました。ただし、Collector クラスは Serializable インスタンスを生成しません。そのため、これらを使用する必要がある場合は、2 つの方法があります。

1 つのオプションとして CacheCollectors クラスを使用します。このインスタンスは、シリアライズされていない Collector を提供するために、Collectors を使用することができます。分散 実行 時の分散方法でコレクターのペフォールメントの詳細を確認できます。

```
Map<Object, String> jbossValues = cache.entrySet().stream()
        .filter(e -> e.getValue().contains("Jboss"))
        .collect(CacheCollectors.serializableCollector(() ->
Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue)));
```

または、CacheCollectors の使用を回避し、その代わりに Supplier<Collector> を取るオーバーロードされた 収集 メソッドを使用できます。オーバーロードされた collect メソッドは CacheStream インターフェースでしか利用できません。

```
Map<Object, String> jbossValues = cache.entrySet().stream()
        .filter(e -> e.getValue().contains("Jboss"))
        .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

ただし、Cache および CacheStream インターフェースを使用できない場合は、Serializable 引数を使用できないため、ラムダを複数インターフェースをキャストすることで、ラムダを Serializable に手動でキャストする必要があります。優れた方法ではありませんが、設定することは可能です。

```
Map<Object, String> jbossValues = map.entrySet().stream()
        .filter((Serializable & Predicate<Map.Entry<Object, String>>) e ->
e.getValue().contains("Jboss"))
        .collect(CacheCollectors.serializableCollector(() ->
Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue)));
```

推奨される最も高性能な方法は、AdvancedExternalizer を使用する方法です。これは最小限のペイロードを提供するためです。残念ながら、これは、高度なエクスターナライザーが事前にクラスを定義する必要があるため、ラムダを使用できないことを意味します。

以下に示すように、高度なエクスターナライザーを使用できます。

```
Map<Object, String> jbossValues = cache.entrySet().stream()
        .filter(new ContainsFilter("Jboss"))
        .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

class ContainsFilter implements Predicate<Map.Entry<Object, String>> {
  private final String target;

  ContainsFilter(String target) {
    this.target = target;
  }

  @Override
```

```
    public boolean test(Map.Entry<Object, String> e) {
      return e.getValue().contains(target);
    }
  }

  class JbossFilterExternalizer implements AdvancedExternalizer<ContainsFilter> {

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

    @Override
    public Integer getId() {
      return CUSTOM_ID;
    }

    @Override
    public void writeObject(ObjectOutput output, ContainsFilter object) throws IOException {
      output.writeUTF(object.target);
    }

    @Override
    public ContainsFilter readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      return new ContainsFilter(input.readUTF());
    }
  }
```

コレクターサプライヤーに高度なエクスターナライザーを使用して、ペイロードサイズをさらに減らすこともできます。

```
  Map<Object, String> jbossValues = cache.entrySet().stream()
          .filter(new ContainsFilter("Jboss"))
          .collect(ToMapCollectorSupplier.INSTANCE);

 class ToMapCollectorSupplier<K, U> implements Supplier<Collector<Map.Entry<K, U>, ?,
Map<K, U>>> {
    static final ToMapCollectorSupplier INSTANCE = new ToMapCollectorSupplier();

    private ToMapCollectorSupplier() { }

    @Override
    public Collector<Map.Entry<K, U>, ?, Map<K, U>> get() {
      return Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue);
    }
  }

  class ToMapCollectorSupplierExternalizer implements
AdvancedExternalizer<ToMapCollectorSupplier> {

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

```
    }

    @Override
    public Integer getId() {
      return CUSTOM_ID;
    }

    @Override
    public void writeObject(ObjectOutput output, ToMapCollectorSupplier object) throws
IOException {
    }

    @Override
    public ToMapCollectorSupplier readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      return ToMapCollectorSupplier.INSTANCE;
    }
  }
```

10.3.3. 並列計算

分散ストリームは、デフォルトではできるだけ並列処理を試みます。エンドユーザーはこれを制御でき、実際にはオプションのいずれかを制御する必要があります。これらのストリームを並列化する方法は 2 つあります。

各ノードにローカル キャッシュコレクションからストリームを作成している場合、エンドユーザーは stream または parallelStream メソッドの呼び出しのいずれかを選択できます。並列ストリームが選択されたかどうかに応じて、各ノードに対してローカルで複数のスレッドが有効になります。再ハッシュ対応のiterator やforEachオペレーションなどの一部のオペレーションは、常にローカルで順次ストリームを使用することに注意してください。これは、並行ストリームをローカルに許可するように、ある時点で強化できます。

ローカルの並列処理を使用する場合は、計算が高速にかかる多数のエントリーや操作が必要になるため注意が必要です。また、ユーザーがforEachで並列ストリームを使用する場合、これは通常は計算オペレーションに予約されている共有プールで実行されるため、アクションをブロックしないようにする必要があることに注意してください。

リモートリクエスト 複数のノードがある場合に、リモート要求をすべて同時に処理するか、一度に1つずつ処理するかを制御することが望ましい場合があります。デフォルトでは、iterator 以外のすべての端末オペレーションは同時リクエストを実行します。iterator は、ローカルノードでのメモリー使用量全体を減らす方法であり、実際に実行する連続要求のみを実行します。

ユーザーがこのデフォルトを変更したい場合、CacheStream で sequentialDistribution または parallelDistribution メソッドを呼び出して実行できます。

10.3.4. タスクのタイムアウト

操作リクエストのタイムアウト値を設定できます。このタイムアウトはリモートリクエストのタイムアウトにのみ使用され、リクエストごとに使用されます。前者はローカル実行がタイムアウトしないことを意味し、後者は上記のようなフェイルオーバーシナリオがある場合、後続のリクエストにはそれぞれ新しいタイムアウトがあることを意味します。タイムアウトを指定しないと、レプリケーションのタイムアウトをデフォルトのタイムアウトとして使用します。以下を実行することで、タスクでタイムアウトを設定できます。

```
CacheStream<Object, String> stream = cache.entrySet().stream();
stream.timeout(1, TimeUnit.MINUTES);
```

詳細は、 timeout javadoc で java doc を確認してください。

10.3.5. 注入

Stream には、 forEach と呼ばれる端末オペレーションがあり、データに副次的な影響を与える操作を実行できます。この場合、このストリームをサポートする Cache への参照を取得することが推奨されます。コンシューマーが CacheAware インターフェースを実装した場合は、 Consumer インターフェースからの accept メソッドの前に injectCache メソッドが呼び出されます。

10.3.6. 分散ストリームの実行

分散ストリームの実行は、マップの削減に非常に似ています。ここでは、ゼロを多数の中間操作（マップ、フィルターなど）に送信し、1 つの端末オペレーションが各種ノードに送信します。オペレーションは、基本的に次のようになります。

1. 必要なセグメントは、どのノードが指定のセグメントのプライマリー所有者であるかによってグループ化されます。
2. リクエストが生成され、処理すべきセグメントを含む中間および端末オペレーションが含まれる各リモートノードに送信されます。
    a. 端末オペレーションは、必要に応じてローカルで実行されます。
    b. 各リモートノードはこの要求を受け取り、オペレーションを実行し、その後に応答を戻します。
3.

その後、ローカルノードが、ローカル応答とリモート応答を収集し、オペレーション自体に必要な削減を実行します。

4. その後、最終的な縮小応答がユーザーに返されます

ほとんどの場合、オペレーションはすべて各リモートノードに完全に適用されるため、すべてのオペレーションは完全に分散されます。通常、複数のノードからの結果を減らすために、最後のオペレーションまたは関連するものだけが再適用される場合があります。重要な点の 1 つは、実際にはシリアライズする必要がないことに注意してください。これは、希望の部分であるものが最後に送信された最後の値になります（さまざまなオペレーションの例外は以下に強調表示されます）。

端末オペレーターの分散結果の縮小 以下の段落では各種の端末オペレーターの分散処理方法を説明します。これらのいくつかは、最終結果の代わりに中間値をシリアル化可能にする必要があるという点で特別です。

allMatch noneMatch anyMatch

allMatch オペレーションは各ノードで実行され、すべての結果が論理的に結合されて適切な値を取得します。noneMatch オペレーションおよび anyMatch オペレーションは、論理的または代わりに使用します。これらのメソッドは早期終了もサポートしており、最終結果が判明するとリモート操作とローカル操作を停止します。

collect

collect メソッドは、いくつかの追加手順を実行できるという点で興味深いものです。リモートノードは、結果に対して最終finisherを実行せず、代わりに完全に結合された結果を送り返すことを除いて、すべてを通常どおり実行します。次に、ローカルスレッドは、リモートとローカルの結果を値に結合し、最終的に終了します。ここで覚えておくべき重要な点は、最終的な値はシリアル化可能である必要はなく、supplierメソッドおよびcombinerメソッドから生成された値である必要があるということです。

count

count メソッドは、各ノードから番号を一緒に追加します。

findAny findFirst

findAny オペレーションは、最初に見つかった値（リモートノードからのものかローカル）を返します。これは、値が見つかると他の値を処理しないという点で、早期終了をサポートすることに注意してください。findFirst メソッドは、ソートされた中間オペレーションが必要になるため特別なものです。これは、例外 セクションで説明されています。

max min

max メソッドおよび min メソッドは、各ノードの各最小値または最大値を見つけ、最終的にノード間の最小値または最大値のみが返されるようにローカルで実行されます。

reduce

さまざまな reduce メソッド 1、2、3 は、アキュムレーターが実行可能な量の結果のシリアライズを最終的に行います。次に、ローカルとリモートの結果をローカルでまとめて累積してから、指定した場合は組み合わせます。これは、組み合わせた値がシリアライズ可能である必要がないことを意味する点に注意してください。

10.3.7. キーベースのリハッシュ対応 Operator

イテレーター、スプリッター、および それぞれの 情報は、セグメントだけではなく、セグメントごとのキーを追跡するために、リハッシュ認識が他の端末オペレーターとは異なります。これは、クラスターメンバーシップが変更された場合でも、1回だけ（iterator と spliterator）または1回以上の（forEach）の動作を保証するためです。

リモートノードで呼び出されると iterator および spliterator オペレーターは、エントリーの再バッチを返します。この場合、次のバッチは最後に使用された後にのみ送信されます。このバッチ処理は、ある時点のメモリー内のエントリー数を制限するために行われます。ユーザーノードは、処理したキーを保持し、特定のセグメントが完了すると、それらのキーをメモリから解放します。そのため、iterator メソッドには順次処理が優先されることがあるため、すべてのノードからではなく、セグメントキーのサブセットのみがメモリーに保持されます。

forEach() メソッドはバッチを返しますが、キーの処理が少なくともバッチ処理された後に、キーのバッチを返します。これにより、送信元ノードはどの鍵がすでに処理されているかを把握して、同じエントリーを再処理する可能性を減らすことができます。ただし、これはノードが予期せずダウンした場合に、少なくとも 1 回の動作を要する可能性があることを意味します。この場合、そのノードはバッチを処理していてまだ完了していない可能性があり、処理されたが完了したバッチに含まれていないエントリは、再ハッシュ失敗オペレーションが発生したときに再度実行されます。ノードを追加しても、すべての応答を受け取るまで、再ハッシュフェイルオーバーが発生しないため、この問題は発生しません。

これらの操作バッチサイズは、どちらも CacheStream で distributedBatchSize メソッドを呼び出すことで設定できる同じ値によって制御されます。この値はデフォルトで、状態遷移で設定された chunkSize に設定されます。残念ながら、この値は、メモリ使用量とパフォーマンスと少なくとも1回のトレードオフであり、マイレージは異なる場合があります。

レプリケートされた分散キャッシュでのiteratorの使用

ノードが分散ストリームのすべての要求されたセグメントのプライマリーまたはバックアップ所有者である場合、Red Hat Data Grid は イテレーター または Spliterator ターミナル操作をローカルで実行します。リモート反復がリソース集約型であるため、パフォーマンスを最適化します。

この最適化は、レプリケートされたキャッシュと分散キャッシュの両方に適用されます。ただし、共有 され、write-behind が有効になっているキャッシュストアを使用する場合、Red Hat Data Grid はリモートで反復を実行します。この場合は、リモートで反復を行うことで一貫性が確保されます。

10.3.8. 中間オペレーションの例外

特別な例外がある中間操作があります。これらの操作は 省略 され、1 2. と並び替えられ ます。これらの方法はすべて、正確さを保証するためにストリーム処理に埋め込まれたある種の人為的な iterator を備えています。これらは以下のように文書化されています。このオペレーションにより、パフォーマンスが低下する可能性があります。

スキップ

中間スキップオペレーションまで人為的な iterator が埋め込まれています。結果はローカルに格納され、適切な要素量をスキップできます。

ソート済み

警告: この操作には、ローカルノード上のメモリーのすべてのエントリーが必要です。人為的な iterator は、中間のソートされたオペレーションまで埋め込まれます。すべての結果がローカルでソートされます。要素のバッチを返す分散ソートを計画することは可能ですが、これはまだ実装されていません。

一意

警告: この操作には、ローカルノード上のメモリーのすべて、またはほぼすべてのエントリーが必要です。各リモートノードで distinct が実行され、人為的な iterator がそれらの distinct 値を返します。そして最後に、これらの結果はすべて、個別のオペレーションが実行されます。

残りの中間オペレーションは、期待通りに完全に配布されます。

10.3.9. 例

単語数

単語数は使いすぎると、map/reducパラダイムの典型的な例になります。Red Hat Data Grid ノードにキー → 文のマッピングが保存されることを前提とします。キーは文字列であり、各文も文字列であり、使用可能なすべての文のすべての単語の出現をカウントする必要があります。このような分散タスクの実装は、以下のように定義できます。

```
public class WordCountExample {
```

```
  /**
   * In this example replace c1 and c2 with
   * real Cache references
   *
   * @param args
   */
  public static void main(String[] args) {
    Cache<String, String> c1 = ...;
    Cache<String, String> c2 = ...;

    c1.put("1", "Hello world here I am");
    c2.put("2", "Infinispan rules the world");
    c1.put("3", "JUDCon is in Boston");
    c2.put("4", "JBoss World is in Boston as well");
    c1.put("12","JBoss Application Server");
    c2.put("15", "Hello world");
    c1.put("14", "Infinispan community");
    c2.put("15", "Hello world");

    c1.put("111", "Infinispan open source");
    c2.put("112", "Boston is close to Toronto");
    c1.put("113", "Toronto is a capital of Ontario");
    c2.put("114", "JUDCon is cool");
    c1.put("211", "JBoss World is awesome");
    c2.put("212", "JBoss rules");
    c1.put("213", "JBoss division of RedHat ");
    c2.put("214", "RedHat community");

    Map<String, Long> wordCountMap = c1.entrySet().parallelStream()
      .map(e -> e.getValue().split("\\s"))
      .flatMap(Arrays::stream)
      .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
  }
}
```

この場合、前述の例から単語数を簡単に実行できます。

ただし、例で最も頻繁に使用される単語を見つけたい場合はどうすればよいでしょうか。このケースについて少し考えてみると、最初にすべての単語をカウントしてローカルで利用できるようにする必要があることに気付くでしょう。そのため、実際にはいくつかのオプションがあります。

コレクターでフィニッシャーを使用できます。これは、すべての結果が収集された後にユーザースレッドで呼び出されます。前の例からいくつかの冗長な行が削除されました。

```
public class WordCountExample {
  public static void main(String[] args) {
    // Lines removed

    String mostFrequentWord = c1.entrySet().parallelStream()
```

```
        .map(e -> e.getValue().split("\\s"))
        .flatMap(Arrays::stream)
        .collect(() -> Collectors.collectingAndThen(
          Collectors.groupingBy(Function.identity(), Collectors.counting()),
            wordCountMap -> {
              String mostFrequent = null;
              long maxCount = 0;
                for (Map.Entry<String, Long> e : wordCountMap.entrySet()) {
                  int count = e.getValue().intValue();
                  if (count > maxCount) {
                    maxCount = count;
                    mostFrequent = e.getKey();
                  }
                }
                return mostFrequent;
            }));

}
```

残念ながら、最後のステップは単一のスレッドでのみ実行されるため、単語が多い場合は非常に遅くなる可能性があります。これを Streams で並列化するもう 1 つの方法があります。

前述したように、処理後にローカルノードに含まれるため、実際にはマップ結果でストリームを使用することができました。そのため、結果に並列ストリームを使用できます。

```
public class WordFrequencyExample {
  public static void main(String[] args) {
    // Lines removed

    Map<String, Long> wordCount = c1.entrySet().parallelStream()
        .map(e -> e.getValue().split("\\s"))
        .flatMap(Arrays::stream)
        .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
    Optional<Map.Entry<String, Long>> mostFrequent =
wordCount.entrySet().parallelStream().reduce(
        (e1, e2) -> e1.getValue() > e2.getValue() ? e1 : e2);
```

これにより、最も頻繁に発生する要素を計算する際に、すべてのコアをローカルで利用できるようになります。

特定のエントリーの削除

分散ストリームは、ライブ先のデータを変更する方法として使用することもできます。たとえば、特定の単語が含まれるキャッシュのエントリーをすべて削除します。

```
public class RemoveBadWords {
```

```
public static void main(String[] args) {
  // Lines removed
  String word = ..

  c1.entrySet().parallelStream()
    .filter(e -> e.getValue().contains(word))
    .forEach((c, e) -> c.remove(e.getKey());
```

シリアル化されているものとそうでないものを注意深く記録すると、ラムダによって取得されるときに、オペレーションとともに単語のみが他のノードにシリアル化されることがわかります。ただし、実際に節約できるのは、キャッシュ操作がプライマリ所有者に対して実行されるため、これらの値をキャッシュから削除するために必要なネットワークトラフィックの量が削減されることです。各ノードで呼び出されたときにキャッシュをBiConsumerに渡す特別なBiConsumerメソッドのオーバーライドを提供するため、キャッシュはラムダによって取得されません。

この方法でforEachコマンドを使用する際に留意すべきことの1つは、基になるストリームがロックを取得しないことです。キャッシュの削除操作は自然にロックを取得しますが、値はストリームが見たものから変更されている可能性があります。つまり、ストリームがエントリを読み取った後にエントリが変更された可能性がありますが、削除によって実際に削除されました。

LockedStreamと呼ばれる新しいバリアントを具体的に追加しました。

他の多くの例

Streams APIはJREツールであり、それを使用するための例がたくさんあります。操作は何らかの方法でシリアル化可能である必要があることを覚えておいてください。

10.4. 分散実行

注記

分散エグゼキューターが Red Hat Data Grid 9.1 で非推奨となりました。Cluster Executor または Distributed Stream のいずれかを使用して、以前の操作を実行する必要があります。

Red Hat Data Grid は、標準の JDK ExecutorService インターフェースを介して分散実行を提供します。ローカル JVM で実行されるのではなく、実行用に送信されたタスクは、Red Hat Data Grid ノードのクラスター全体で実行されます。DistributedExecutorService はすべて特定の 1 つのキャッシュにバインドされます。送信されたタスクは、送信されたタスクが DistributedCallable のインスタンスである場合にのみ、その特定のキャッシュからキーと値のペアにアクセスできます。また、ユーザーが他の ExecutorService と同様に一般的な Runnable または Callable を送信することを妨げるものは何もありません。しかし、DistributedExecutorService の名前が示すように、送信された

Callable または Runnable を Red Hat Data Grid クラスターの別の JVM に移行し、タスクインカーカーに結果を返す可能性があります。Callable、Runnable、DistributedCallable 提出はすべて他のノードへの移行が生じる可能性があるため、この送信は 336 または Externalizable のいずれかである必要があります。また、呼び出し不可能な値も外部化可能である必要があります。返された値がシリアライズ不可能な場合は、Not336Exception がスローされます。

Red Hat Data Grid の分散タスクエグゼキューターは、Red Hat Data Grid キャッシュノードのデータを実行タスクの入力として使用します。他の分散フレームワークの多くは活用されず、ユーザーは既知の場所から分散タスクの入力を指定する必要があります。さらに、Red Hat Data Grid 分散実行フレームワークのユーザーは、中間および最終的な結果にストアを設定する必要がないため、複雑さとメンテナンスの別のレイヤーが取り除かれます。

この分散実行フレームワークは、Red Hat Data Grid の Data Grid のファクト入力データをすでに負荷分散しています（DIST モードの場合）。入力データはすでに分散されている実行タスクも自動的に分散されるため、ユーザーは明示的に作業タスクを特定の Red Hat Data Grid ノードに割り当てる必要はありません。ただし、このフレームワークでは、ユーザーは分散実行タスクの入力として任意のキャッシュキーのサブセットを指定できます。

10.4.1. DistributedCallable API

タスクを実行するために Red Hat Data Grid キャッシュデータにアクセスする必要がある場合は、DistributedCallable インターフェースでタスクをカプセル化することを推奨します。DistributedCallable は、java.util.concurrent パッケージから既存の Callable パッケージのサブタイプで、DistributedCallable はリモート JVM で実行し、Red Hat Data Grid キャッシュから入力を受け取ることができます。タスクのメインアルゴリズムは基本的に変更できず、入力ソースのみが変更されます。DistributedCallable が Red Hat Data Grid キャッシュから入力を取得している間に、既存の呼び出し可能実装は Java オブジェクト/プリミティブ形式で入力を取得する可能性があります。したがって、呼び出し可能なインターフェースがすでに実装されているユーザーは、DistributedCallable を拡張し、Red Hat Data Grid 実行環境からのキーをタスクの入力として使用するだけです。DistributedCallable(Implentation of DistributedCallable)は、分散可能な拡張により、既存の呼び出し可能の実装を引き続きサポートしますが、同時に実行する準備ができています。

```
public interface DistributedCallable<K, V, T> extends Callable<T> {

   /**
    * Invoked by execution environment after DistributedCallable
    * has been migrated for execution to a specific node.
    *
    * @param cache
    *          cache whose keys are used as input data for this
    *          DistributedCallable task
    * @param inputKeys
    *          keys used as input for this DistributedCallable task
    */
   public void setEnvironment(Cache<K, V> cache, Set<K> inputKeys);

}
```

10.4.2. 呼び出し可能および CDI

DistributedCallable で使用する、または実装できないユーザーは、DistributedExecutorService で使用される入力キャッシュへの参照には、CDI メカニズムによってインジェクトされる入力キャッシュのオプションが必要です。Red Hat Data Grid がノードを実行する Red Hat Data Grid に対して呼び出し可能になると、Red Hat Data Grid CDI メカニズムは適切なキャッシュ参照を提供し、それを注入して呼び出し可能です。すべてのフィールドは Callable の Cache フィールドを宣言し、必須の @Inject アノテーションと共に org.infinispan.cdi.Input アノテーションでアノテーションを付ける必要があります。

```
public class CallableWithInjectedCache implements Callable<Integer>, Serializable {

    @Inject
    @Input
    private Cache<String, String> cache;


    @Override
    public Integer call() throws Exception {
      //use injected cache reference
      return 1;
    }
}
```

10.4.3. DistributedExecutorService、DistributedTaskBuilder、および DistributedTask API

DistributedExecutorService は、java.util.concurrent パッケージの使い慣れた ExecutorService のシンプルな拡張です。ただし、DistributedExecutorService の利点を見落とすことはできません。JDK の ExecutorService で実行される代わりに、既存の呼び出し可能なタスクは、Red Hat Data Grid クラスターで実行することもできます。Red Hat Data Grid 実行環境は、タスクを実行ノードに移行し、タスクを実行し、呼び出したノードに結果を返します。当然ながら、すべての呼び出されたタスクが並列分散実行の利点を得られるわけではありません。優れた候補は、長時間実行され、計算されたタスクであり、同時に処理できる入力データを使用して同時に実行する、またはタスクを実行できます。並列実行と並列アルゴリズムの適切な候補については、『Introduction to Parallel Computing』を参照してください。

DistributedExecutorService の 2 つ目の利点は、Red Hat Data Grid キャッシュノードからの入力を取り、特定の計算を実行して、結果を呼び出し元に返すタスクの迅速な実装を可能にすることです。ユーザーは、指定された DistributedCallable の入力として使用するキーを指定し、Red Hat Data Grid クラスターで実行するために呼び出し可能なキーを送信します。Red Hat Data Grid ランタイムは Appriate キーを見つけ、分散可能な移行をターゲット実行ノードに移行し、最後に実行された各呼び出し可能な結果のリストを返します。当然ながら、Red Hat Data Grid が指定のキャッシュのすべてのキーで DistributedCallable を実行する場合は、入力キーの指定を省略できます。

Callable/Runnable タスクがすでに定義されている場合は、DistributedExecutorService を使用する方法を見てみましょう。さらに、実行のために DefaultExecutorService のインスタンスに送信するだけです！

```
ExecutorService des = new DefaultExecutorService(cache);
Future<Boolean> future = des.submit(new SomeCallable());
Boolean r = future.get();
```

タスクタイムアウト、カスタムのフェイルオーバーポリシー、実行ポリシーなど、さらにタスクパラメーターを指定する必要がある場合は、DistributedTaskBuilder および DistributedTask API を使用します。

```
DistributedExecutorService des = new DefaultExecutorService(cache);
DistributedTaskBuilder<Boolean> taskBuilder = des.createDistributedTaskBuilder(new
SomeCallable());
taskBuilder.timeout(10,TimeUnit.SECONDS);
...
...
DistributedTask<Boolean> distributedTask = taskBuilder.build();
Future<Boolean> future = des.submit(distributedTask);
Boolean r = future.get();
```

10.4.4. 分散タスクのフェイルオーバー

分散実行フレームワークは、タスクのフェイルオーバーをサポートします。デフォルトでは、フェイルオーバーポリシーはインストールされず、タスクの Runnable/Callable/DistributedCallable は単に失敗します。フェイルオーバーメカニズムは以下の場合に呼び出されます。

a）Failover（タスクが実行されているノード障害）

B）タスクの失敗によりフェイルオーバーを実行します（呼び出し可能なタスクが例外をスローするなど）。

Red Hat Data Grid は、そのノードが利用可能な場合、別のランダムなノードで分散タスクの一部の実行を試みるランダムなノードフェイルオーバーポリシーを提供します。ただし、より高度なフェイルオーバーポリシーを実装する必要があるユーザーは、DistributedTaskFailoverPolicy インターフェースを実装することができます。たとえば、完了していないタスクのフェイルオーバーに一貫性のあるハッシュ(CH)メカニズムを使用する必要がある場合があります。CH ベースのフェイルオーバーは、失敗したノード F で実行された入力データのバックアップを持つクラスターノードへの失敗したタスク T 等の可能性があります。

```
/**
 * DistributedTaskFailoverPolicy allows pluggable fail over target selection for a failed
remotely
 * executed distributed task.
 *
 */
public interface DistributedTaskFailoverPolicy {
```

```
    /**
     * As parts of distributively executed task can fail due to the task itself throwing an exception
     * or it can be a system caused failure (e.g node failed or left cluster during task
     * execution etc).
     *
     * @param failoverContext
     *          the FailoverContext of the failed execution
     * @return result the Address of the node selected for fail over execution
     */
    Address failover(FailoverContext context);

    /**
     * Maximum number of fail over attempts permitted by this DistributedTaskFailoverPolicy
     *
     * @return max number of fail over attempts
     */
    int maxFailoverAttempts();
}
```

そのため、たとえば、以下によってランダムフェイルオーバーの実行ポリシーを指定するだけです。

```
DistributedExecutorService des = new DefaultExecutorService(cache);
DistributedTaskBuilder<Boolean> taskBuilder = des.createDistributedTaskBuilder(new
SomeCallable());
taskBuilder.failoverPolicy(DefaultExecutorService.RANDOM_NODE_FAILOVER);
DistributedTask<Boolean> distributedTask = taskBuilder.build();
Future<Boolean> future = des.submit(distributedTask);
Boolean r = future.get();
```

10.4.5. 分散タスク実行ポリシー

DistributedTaskExecutionPolicy は、タスクが Red Hat Data Grid クラスター全体でカスタムタスク実行ポリシーを指定できるようにする列挙です。DistributedTaskExecutionPolicy は、タスクの実行をノードのサブセットに効果的にスコープします。たとえば、あるユーザーがバックアップリモートネットワーク centre ではなく、ローカルネットワークサイトでタスクのみを実行したいとします。たとえば、特定のタスク実行には、特定の Red Hat Data Grid ラックノードの専用サブセットのみを使用してください。DistributedTaskExecutionPolicy は DistributedTask のインスタンスごとに設定されます。

```
DistributedExecutorService des = new DefaultExecutorService(cache);
DistributedTaskBuilder<Boolean> taskBuilder = des.createDistributedTaskBuilder(new
SomeCallable());
taskBuilder.executionPolicy(DistributedTaskExecutionPolicy.SAME_RACK);
DistributedTask<Boolean> distributedTask = taskBuilder.build();
Future<Boolean> future = des.submit(distributedTask);
Boolean r = future.get();
```

10.4.6. 例

ピークの概算は、DistributedExecutorService での並行分散実行から大幅に利点を得られます。正方形の面積はSa = 4r2であり、円の面積はCa=pi*r2であることを思い出してください。2 つ目の式からの r2 を置き換えると、pi = 4 * Ca/S になります。ここで、正方形に非常に多くのダーツを射ることができると仮定して、射ったダーツの総数に対して円の中に入ったダーツの割合を取ると、Ca/Saの値が近似します。pi = 4 * Ca/Sa であるため、piの近似値を簡単に導き出すことができます。ダーツを多く撃つほど、より良い近似が得られます。以下の例では、Red Hat Data Grid クラスター全体にわたるダイアの作業は、1,000万人物ではなく、「サイート」の作業を行います。

```
public class PiAppx {

 public static void main (String [] arg){
   List<Cache> caches = ...;
   Cache cache = ...;

   int numPoints = 10000000;
   int numServers = caches.size();
   int numberPerWorker = numPoints / numServers;

   DistributedExecutorService des = new DefaultExecutorService(cache);
   long start = System.currentTimeMillis();
   CircleTest ct = new CircleTest(numberPerWorker);
   List<Future<Integer>> results = des.submitEverywhere(ct);
   int countCircle = 0;
   for (Future<Integer> f : results) {
     countCircle += f.get();
   }
   double appxPi = 4.0 * countCircle / numPoints;

   System.out.println("Distributed PI appx is " + appxPi +
   " completed in " + (System.currentTimeMillis() - start) + " ms");
 }

 private static class CircleTest implements Callable<Integer>, Serializable {

   /** The serialVersionUID */
   private static final long serialVersionUID = 3496135215525904755L;

   private final int loopCount;

   public CircleTest(int loopCount) {
     this.loopCount = loopCount;
   }

   @Override
   public Integer call() throws Exception {
     int insideCircleCount = 0;
     for (int i = 0; i < loopCount; i++) {
       double x = Math.random();
       double y = Math.random();
       if (insideCircle(x, y))
         insideCircleCount++;
     }
     return insideCircleCount;
   }
```

```
    private boolean insideCircle(double x, double y) {
      return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))
      <= Math.pow(0.5, 2);
    }
  }
}
```

第11章 インデックス化およびクエリー

11.1. 概要

Red Hat Data Grid は、メインの Map のような API を補完する強力な検索 API を使用して、グリッドに保存された Protocol Buffers 経由でエンコードされた Java Pojo(s)またはオブジェクトをインデックス化して検索します。

クエリーは ライブラリー と クライアント/サーバーモード （Java、C#、Node.js など）でも可能で、Red Hat Data Grid は Apache Lucene を使用してデータをインデックス化でき、さまざまなデータ取得のユースケースに対応するために、効率的 な全文 の検索エンジンを提供できます。

インデックス設定はスキーマ定義に依存し、Red Hat Data Grid はライブラリーモードの場合はアノテーション付き Java クラスを使用し、他の言語で書かれたリモートクライアントの protobuf スキーマを使用できます。Red Hat Data Grid は、protobuf に基づいて、Java クライアントと非 Java クライアント間の完全な相互運用性を可能にします。

Red Hat Data Grid はインデックス化されたクエリーとは別に、インデックス化されていないデータ（インデックスなしのクエリー）および部分的にインデックス化されたデータ（ハイブリッドクエリー）でクエリーを実行できます。

検索 API の観点では、Red Hat Data Grid には Ickle と呼ばれる独自のクエリー言語があります。これは文字列ベースであり、全文のクエリーをサポートします。Java 組み込みクライアント Red Hat Data Grid では、グリッドでの Lucene クエリーの実行をサポートする Hibernate Search Query API （Faceted 検索や Spatial 検索などの高度な検索機能以外に）は、埋め込み型 Java クライアントとリモート Java クライアントの両方に使用できます。#query_dsl

最後に、Red Hat Data Grid は Continuous Queries をサポートしており、これは他の API と逆の方法で動作します。クエリーを作成し、結果を取得する代わりに、クライアントはクラスターの変更時に継続的に評価されるクエリーを登録し、変更されたデータがクエリーと一致するたびに通知を生成することができます。

11.2. 埋め込みクエリー

Red Hat Data Grid をライブラリーとして使用すると、埋め込みクエリーを利用できます。protobuf マッピングは不要であり、インデックス作成と検索の両方が Java オブジェクト上で実行されます。ライブラリーモードでは、Lucene クエリーを直接実行し、利用可能なすべての Query API を使用し、柔軟なインデックス化設定により、レイテンシーを最小限に保つことができます。

11.2.1. 簡単な例

Book インスタンスを「books」と呼ばれる Red Hat Data Grid キャッシュに保存します。本書 のインスタンスはインデックス化されるため、Red Hat Data Grid ではインデックスを 自動的に設定します。

Red Hat Data Grid の設定：

infinispan.xml

```
<infinispan>
   <cache-container>
      <transport cluster="infinispan-cluster"/>
      <distributed-cache name="books">
         <indexing index="LOCAL" auto-config="true"/>
      </distributed-cache>
   </cache-container>
</infinispan>
```

キャッシュを取得します。

```
import org.infinispan.Cache;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.manager.EmbeddedCacheManager;

EmbeddedCacheManager manager = new DefaultCacheManager("infinispan.xml");
Cache<String, Book> cache = manager.getCache("books");
```

各書籍は 以下の例のように定義されます。インデックス化するプロパティーを選択する必要があります。各プロパティーに対して、オプションで Hibernate Search プロジェクトに定義されたアノテーションを使用して、高度なインデックスオプションを選択できます。

Book.java

```
import org.hibernate.search.annotations.*;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

//Values you want to index need to be annotated with @Indexed, then you pick which fields
```

```
and how they are to be indexed:
@Indexed
public class Book {
  @Field String title;
  @Field String description;
  @Field @DateBridge(resolution=Resolution.YEAR) Date publicationYear;
  @IndexedEmbedded Set<Author> authors = new HashSet<Author>();
}
```

Author.java

```
public class Author {
  @Field String name;
  @Field String surname;
  // hashCode() and equals() omitted
}
```

これで、複数の Book インスタンスを Red Hat Data Grid Cache に保存した場合、以下の例に示すように一致するフィールドを検索できます。

Lucene クエリーの使用：

```
// get the search manager from the cache:
SearchManager searchManager = org.infinispan.query.Search.getSearchManager(cache);

// create any standard Lucene query, via Lucene's QueryParser or any other means:
org.apache.lucene.search.Query fullTextQuery = //any Apache Lucene Query

// convert the Lucene query to a CacheQuery:
CacheQuery cacheQuery = searchManager.getQuery( fullTextQuery );

// get the results:
List<Object> found = cacheQuery.list();
```

Lucene Query は、「title:infinispan AND authors.name:sanne」などのテキスト形式でクエリーを解析するか、Hibernate Search が提供するクエリービルダーを使用して作成されます。

```
// get the search manager from the cache:
```

```
SearchManager searchManager = org.infinispan.query.Search.getSearchManager( cache );

// you could make the queries via Lucene APIs, or use some helpers:
QueryBuilder queryBuilder = searchManager.buildQueryBuilderForClass(Book.class).get();

// the queryBuilder has a nice fluent API which guides you through all options.
// this has some knowledge about your object, for example which Analyzers
// need to be applied, but the output is a fairly standard Lucene Query.
org.apache.lucene.search.Query luceneQuery = queryBuilder.phrase()
            .onField("description")
            .andField("title")
            .sentence("a book on highly scalable query engines")
            .createQuery();

// the query API itself accepts any Lucene Query, and on top of that
// you can restrict the result to selected class types:
CacheQuery query = searchManager.getQuery(luceneQuery, Book.class);

// and there are your results!
List objectList = query.list();

for (Object book : objectList) {
    System.out.println(book);
}
```

list（） の他に、結果をストリーミングするか、ページネーションを使用します。

Lucene または全文機能を必要としない検索や、ほとんど集約および完全一致については、Red Hat Data Grid Query DSL API を使用できます。

```
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;
import org.infinispan.query.Search;

// get the query factory:
QueryFactory queryFactory = Search.getQueryFactory(cache);

Query q = queryFactory.from(Book.class)
        .having("author.surname").eq("King")
        .build();

List<Book> list = q.list();
```

最後に、Ickle クエリーを直接使用でき、1 つ以上の述語で Lucene 構文に許可します。

```
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

// get the query factory:
```

```
QueryFactory queryFactory = Search.getQueryFactory(cache);


Query q = queryFactory.create("from Book b where b.author.name = 'Stephen' and " +
            "b.description : (+'dark' -'tower')");

List<Book> list = q.list();
```

11.2.2. インデックス化

Red Hat Data Grid でのインデックス作成はキャッシュごとに行われ、デフォルトではキャッシュはインデックス化されません。インデックスの有効化は必須ではありませんが、インデックスを使用するクエリーにはパフォーマンスが非常に高くなります。一方、インデックスを有効にすると、クラスターの書き込みスループットに悪影響を及ぼす可能性があるため、キャッシュの種類やユースケースに応じてこの影響を最小限に抑えるために、一部のストラテジーの クエリーパフォーマンスガイド を確認してください。

11.2.2.1. 設定

11.2.2.1.1. 一般的な形式

XML でインデックスを有効にするには、<indexing> 要素と インデックス （インデックス モード）をキャッシュ設定に追加し、必要に応じて追加のプロパティーを指定する必要があります。

```
<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="default">
      <indexing index="ALL">
        <property name="property.name">some value</property>
      </indexing>
    </replicated-cache>
  </cache-container>
</infinispan>
```

プログラムによる：

```
import org.infinispan.configuration.cache.*;

ConfigurationBuilder cacheCfg = ...
cacheCfg.indexing().index(Index.ALL)
    .addProperty("property name", "propery value")
```

11.2.2.1.2. インデックス名

index 要素内の各プロパティーの前にインデックス名が付けられ、org.infinispan.sample.Car という名前 のインデックスは local-heap です。

```
    ...
    <indexing index="ALL">
       <property name="org.infinispan.sample.Car.directory_provider">local-heap</property>
    </indexing>
    ...
</infinispan>
```

```
cacheCfg.indexing()
  .index(Index.ALL)
    .addProperty("org.infinispan.sample.Car.directory_provider", "local-heap")
```

Red Hat Data Grid は、キャッシュに存在する各エンティティーのインデックスを作成し、これらのインデックスを個別に設定できます。@Indexed アノテーションが付けられたクラスの場合、インデックス名はアノテーションの name 引数で上書きされない限り、完全修飾クラス 名 になります。

以下のスニペットでは、すべてのエンティティーのデフォルトストレージは infinispan ですが、Boat インスタンスは boatIndex という名前のインデックスの local-heap に保存されます。空プレーン エンティティーは local-heap にも保存されます。その他のエンティティーのインデックスは、デフォルト で接頭辞が付けられたプロパティーで設定されます。

```
package org.infinispan.sample;

@Indexed(name = "boatIndex")
public class Boat {

}

@Indexed
public class Airplane {

}
```

```
    ...
    <indexing index="ALL">
       <property name="default.directory_provider">infinispan</property>
       <property name="boatIndex.directory_provider">local-heap</property>
       <property name="org.infinispan.sample.Airplane.directory_provider">
          ram
       </property>
    </indexing>
    ...
</infinispan>
```

11.2.2.1.3. インデックス化されたエンティティーの指定

Red Hat Data Grid は、キャッシュ内の異なるエンティティータイプのインデックスを自動的に認識および管理できます。Red Hat Data Grid の今後のバージョンではこの機能が削除されるため、イン

デックス化されるタイプを事前に宣言することが推奨されます（完全修飾クラス名でリストされます）。これは、xml で行うことができます。

```
<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="default">
      <indexing index="ALL">
        <indexed-entities>
          <indexed-entity>com.acme.query.test.Car</indexed-entity>
          <indexed-entity>com.acme.query.test.Truck</indexed-entity>
        </indexed-entities>
      </indexing>
    </replicated-cache>
  </cache-container>
</infinispan>
```

プログラムを使用する場合

```
cacheCfg.indexing()
    .index(Index.ALL)
     .addIndexedEntity(Car.class)
     .addIndexedEntity(Truck.class)
```

サーバーモードでは、'indexed-entities' 要素の下に一覧表示されるクラス名は、JBoss Modules モジュール識別子、スロット名、および完全修飾クラス名で構成される 'extended' クラス名形式を使用する必要があります。これらの 3 つのコンポーネントは、':' 文字によって区切られます（例：''com.acme.my-module-with-entity-classes:my-slot:com.acme.query.test.Car''）。エンティティークラスは、参照されるモジュールに置く必要があります。これは、サーバーの 'modules' フォルダーにデプロイされるユーザー提供モジュールか、'deployments' フォルダーにデプロイされるプレーン jar のいずれかになります。問題のモジュールはキャッシュの自動依存関係となるため、最終的に再デプロイするとキャッシュが再起動されます。

注記

サーバーに対してのみ、'extended' クラス名の使用要件をフォローできず、プレーンクラス名を使用すると、誤った ClassLoader が使用されているためクラスがないために解決に失敗します（Red Hat Data Grid の内部クラスパスが使用されている）。

11.2.2.2. インデックスモード

通常、Red Hat Data Grid ノードは local と remote の 2 つのソースからデータを受け取ります。ローカル変換は、同じ JVM でマップ API を使用してデータを操作します。リモートデータはレプリケーションまたはリバランス中に他の Red Hat Data Grid ノードから提供されます。

インデックスモードの設定は、クラスターポイントのノードから、インデックス化されるデータを

定義します。

以下の値を使用できます。

- ALL: すべてのデータがインデックス化、ローカル、およびリモートされます。
- LOCAL: ローカルデータのみがインデックス化されます。
- PRIMARY_OWNER: ローカルまたはリモートの起点に関係なく、ノードがプライマリー所有者であるキーを含むエントリーのみがインデックス化されます。
- NONE: インデックス化されません。インデックスを全く設定しないと同等です。

11.2.2.3. インデックスマネージャー

インデックスマネージャーは、Lucene の IndexReader や IndexWriter などの複数のクエリーコンポーネントのインデックス設定、ディストリビューション、および内部ライフサイクルを行う Red Hat Data Grid クエリーの中心的なコンポーネントです。各インデックスマネージャーは、インデックスの物理ストレージを定義する Directory Provider に関連付けられます。

インデックスディストリビューションに関して、Red Hat Data Grid は共有インデックスまたは共有されていないインデックスで設定できます。

11.2.2.4. 共有インデックス

共有インデックスは、特定のキャッシュに対して、単一の分散されたクラスター全体のインデックスです。主な利点は、インデックスがすべてのノードに表示され、インデックスがローカルであるかのようにクエリーできることです。すべてのメンバーにクエリーを ブロードキャスト し、結果を集約する必要はありません。マイナス面は、Lucene が、同時にインデックスに書き込む複数のプロセスを許可しないことです。また、適切な共有インデックス対応インデックスマネージャーによってロックの取り組みの調整を行う必要があります。いずれの場合も、書き込みロックが 1 つ必要です。

Red Hat Data Grid は、別のキャッシュセットにインデックスを格納する Red Hat Data Grid Directory Provider を使用する共有インデックスをサポートします。共有インデックスを使用するためには、InfinispanIndexManager と AffinityIndexManager の 2 つのインデックスマネージャーを使用できます。

11.2.2.4.1. インデックスモードの影響

共有インデックスは、冗長インデックスが発生するため、すべて のインデックスモードを使用しないでください。単一のインデックスクラスターが存在するため、エントリーはキャッシュ API を介して挿入されるとインデックス化され、Red Hat Data Grid がこれを別のノードに複製する場合にもインデックス化されます。ALL モードは通常、各ノードで完全な インデックスレプリカを作成するために共有されていない インデックスに関連付けられます。

11.2.2.4.2. InfinispanIndexManager

このインデックスマネージャーは Red Hat Data Grid Directory Provider を使用し、共有インデックスの作成に適しています。この設定では、インデックスモードを LOCAL に設定する必要があります。

設定:

```
<distributed-cache name="default" >
   <indexing index="LOCAL">
      <property name="default.indexmanager">
         org.infinispan.query.indexmanager.InfinispanIndexManager
      </property>
      <!-- optional: tailor each index cache -->
      <property name="default.locking_cachename">LuceneIndexesLocking_custom</property>
      <property name="default.data_cachename">LuceneIndexesData_custom</property>
      <property name="default.metadata_cachename">LuceneIndexesMetadata_custom</property>
   </indexing>
</distributed-cache>

<!-- Optional -->
<replicated-cache name="LuceneIndexesLocking_custom">
   <indexing index="NONE" />
   <-- extra configuration -->
</replicated-cache>

<!-- Optional -->
<replicated-cache name="LuceneIndexesMetadata_custom">
   <indexing index="NONE" />
   <-- extra configuration -->
</replicated-cache>

<!-- Optional -->
<distributed-cache name="LuceneIndexesData_custom">
   <-- extra configuration -->
   <indexing index="NONE" />
</distributed-cache>
```

インデックスは、デフォルトの LuceneIndexesData、LuceneIndexesMetadata、および LuceneIndexesLocking によって呼び出されるクラスター化されたキャッシュのセットに保存されま

す。

LuceneIndexesLocking キャッシュは、Lucene ロックの保存に使用され、非常に小さいキャッシュです。これにはエンティティー（インデックス）ごとにエントリーが 1 つ含まれます。

LuceneIndexesMetadata キャッシュは、名前、チャンク、サイズなど、インデックスの一部である論理ファイルの情報を保存するために使用されます。また、サイズが小さくなります。

LuceneIndexesData キャッシュは、ほとんどのインデックスが置かれます。つまり、他の 2 つのサイズが大きくなりますが、Lucene の効率的な保存技術により、キャッシュ自体のデータよりも小さくする必要があります。

これらの 3 つのケースの設定を再定義する必要はありません。Red Hat Data Grid では適切なデフォルト値が選択されます。再定義を行う理由は、特定のシナリオのパフォーマンスチューニングであるか、たとえばキャッシュストアを設定して永続化するためです。

クラスターの 2 つ以上のノードが同時にインデックスへの書き込みを試みる際にインデックスの破損を回避するために、InfinispanIndexManager は内部でクラスター内（JGroups コーディネーター）のマスターを選択し、すべてのインデックスをこのマスターに転送します。

11.2.2.4.3. AffinityIndexManager

AffinityIndexManager は、Red Hat Data Grid Directory Provider を使用してインデックスも保存する共有インデックスに使用される 実験的 インデックスマネージャーです。InfinispanIndexManager とは異なり、クラスター全体にインデックスを処理する単一ノード（マスター）はありませんが、複数のシャードを使用してインデックスを分割します。各シャードは、1 つ以上の Red Hat Data Grid セグメントに関連付けられたデータをインデックス化します。内部の作業の詳細は、設計ドキュメント を参照してください。

特別な種類の KeyPartitioner と共に、PRIMARY_OWNER インデックスモードが必要です。

XML 設定：

```
<distributed-cache name="default"
                   key-partitioner="org.infinispan.distribution.ch.impl.AffinityPartitioner">
    <indexing index="PRIMARY_OWNER">
        <property name="default.indexmanager">
            org.infinispan.query.affinity.AffinityIndexManager
        </property>
```

```
        <!-- optional: control the number of shards, the default is 4 -->
        <property name="default.sharding_strategy.nbr_of_shards">10</property>
    </indexing>
</distributed-cache>
```

プログラムによる：

```
import org.infinispan.distribution.ch.impl.AffinityPartitioner;
import org.infinispan.query.affinity.AffinityIndexManager;

ConfigurationBuilder cacheCfg = ...
cacheCfg.clustering().hash().keyPartitioner(new AffinityPartitioner());
cacheCfg.indexing()
      .index(Index.PRIMARY_OWNER)
      .addProperty("default.indexmanager", AffinityIndexManager.class.getName())
      .addProperty("default.sharding_strategy.nbr_of_shards", "10")
```

デフォルトでは、AffinityIndexManager には Red Hat Data Grid のセグメントと同じシャードがありますが、この値は上記の例で示すように設定可能です。

シャードの数は、クエリーのパフォーマンスや書き込みスループットに直接影響します。一般的に、シャードの数が多いほど書き込みスループットが向上しますが、クエリーパフォーマンスに悪影響があります。

11.2.2.5. 共有されていないインデックス

共有されていないインデックスは、各ノードで独立したインデックスです。この設定は、各ノードがすべてのクラスターデータを持つレプリケートされたキャッシュに特に利点があります。そのため、クエリー時にネットワークレイテンシーがゼロで最適にクエリーパフォーマンスが向上します。インデックスは各ノードにローカルであるため、各ノードはクラスター全体のインデックスロックに基づいているため、書き込み時に競合は少なくなります。

各ノードは部分的なインデックスを保持する可能性があるため、適切な検索結果を取得するために link#query_clustered_query_api[broadcast] クエリーを link:キャッシュが REPL の場合、ブロードキャストは必要ありません。各ノードはインデックスの完全なローカルコピーを保持でき、クエリーはローカルインデックスを活用し、最適な速度で実行されます。

Red Hat Data Grid には、共有されていないインデックスには ディレクトリーベース と ほぼリアルタイム インデックスの 2 つのインデックスマネージャーがあります。ストレージの場合、共有されていないインデックスは ram、filesystem、または Red Hat Data Grid のローカルキャッシュに配置できます。

11.2.2.5.1. インデックスモードの影響

ディレクトリーベース および ほぼリアルタイム インデックスマネージャーは、異なるインデックス モード に関連付けることができます。その結果、インデックスディストリビューションが異なります。

REPL キャッシュを ALL インデックスモードと組み合わせると、各ノードのクラスター全体のインデックスが完全にコピーされます。このモードでは、ネットワークのレイテンシーなしにクエリーが実質的にローカルになります。これは、REPL キャッシュをインデックス化し、REPL キャッシュが検出されると auto-config によって選択されるモードです。ALL モードは、DIST キャッシュとは使用しないでください。

REPL または DIST キャッシュを LOCAL インデックスモードと組み合わせると、各ノードは同じ JVM から挿入されるデータのみのインデックスをインデックス化し、インデックスの分散が不均等になります。正しいクエリー結果を取得するには、ブロードキャスト クエリーを使用する必要があります。

REPL キャッシュまたは DIST キャッシュを PRIMARY_OWNER と組み合わせると、ブロードキャストクエリーも必要になります。LOCAL モードとは異なり、各ノードのインデックスには、一貫したハッシュに従ってノードで主に所有されるキーを持つインデックス化されたエントリーが含まれます。これにより、ノード間でより均等に分散インデックスが作成されます。

11.2.2.5.2. ディレクトリーベースのインデックスマネージャー

これは、インデックスマネージャーが設定されていない場合に使用されるデフォルトのインデックスマネージャーです。ディレクトリーベース のインデックスマネージャーは、ローカルの lucene ディレクトリーがサポートするインデックスを管理するために使用されます。ram、ファイルシステム、および非クラスター infinispan ストレージをサポートします。

Filesystem storage

これはデフォルトのストレージであり、インデックスマネージャーの設定が省略されたときに使用されます。インデックスは MMapDirectory を使用してファイルシステムに保存されます。これは、ローカルインデックスの推奨ストレージです。インデックスはディスク上の永続的ですが、Lucene がマッピングしたメモリーを取得するため、クエリーのパフォーマンスが悪くなります。

設定:

```
<replicated-cache name="myCache">
  <indexing index="ALL">
    <!-- Optional: define base folder for indexes -->
```

```
    <property name="default.indexBase">${java.io.tmpdir}/baseDir</property>
  </indexing>
</replicated-cache>
```

Red Hat Data Grid は、キャッシュに存在する各エンティティー（インデックス）に default.indexBase の下に異なるフォルダーを作成します。

Ram storage

インデックスは、Lucene RAMDirectory を使用してメモリーに保存されます。大規模なインデックスや、非常に同時の状況には推奨しません。Ram に保存されるインデックスは永続的ではないため、クラスターのシャットダウン後に 再インデックス が必要になります。設定:

```
<replicated-cache name="myCache">
  <indexing index="ALL">
    <property name="default.directory_provider">local-heap</property>
  </indexing>
</replicated-cache>
```

Red Hat Data Grid storage

Red Hat Data Grid ストレージは、インデックスをキャッシュのセットに保存する Red Hat Data Grid Lucene ディレクトリー を利用します。これらのキャッシュは、他の Red Hat Data Grid キャッシュと同様に設定できます。たとえば、キャッシュストアを追加して、インデックスをメモリー以外の場所に永続化します。共有されていないインデックスで Red Hat Data Grid ストレージを使用するには、インデックスに LOCAL キャッシュを使用する必要があります。

```
<replicated-cache name="default">
   <indexing index="ALL">
      <property name="default.locking_cachename">LuceneIndexesLocking_custom</property>
      <property name="default.data_cachename">LuceneIndexesData_custom</property>
      <property name="default.metadata_cachename">LuceneIndexesMetadata_custom</property>
   </indexing>
</replicated-cache>

<local-cache name="LuceneIndexesLocking_custom">
   <indexing index="NONE" />
</local-cache>

<local-cache name="LuceneIndexesMetadata_custom">
   <indexing index="NONE" />
</local-cache>

<local-cache name="LuceneIndexesData_custom">
   <indexing index="NONE" />
</local-cache>
```

11.2.2.5.3. ほぼリアルタイムインデックスマネージャー

ディレクトリーベース のインデックスマネージャーと同様ですが、Lucene の Near-Real-Time 機能を利用します。基礎となるストアにインデックスをフラッシュする頻度は少なくなるため、ディレクトリーベース よりも書き込みパフォーマンスが向上します。欠点は、シャットダウンが適切に行われない場合に、フラッシュされていないインデックスの変更が失われることです。local-heap、filesystem、および local infinispan ストレージと併用できます。異なるストレージタイプごとの設定は、ディレクトリーベース のインデックスマネージャーと同じです。

ram を使用した例：

```
<replicated-cache name="default">
   <indexing index="ALL">
      <property name="default.indexmanager">near-real-time</property>
      <property name="default.directory_provider">local-heap</property>
   </indexing>
</replicated-cache>
```

ファイルシステムの例：

```
<replicated-cache name="default">
   <indexing index="ALL">
      <property name="default.indexmanager">near-real-time</property>
   </indexing>
</replicated-cache>
```

11.2.2.6. 外部インデックス

Red Hat Data Grid 自体によって共有共有および非共有インデックスを管理している以外に、サードパーティー検索エンジンにインデックスをオフロードすることができます。現在、Red Hat Data Grid は Elasticsearch を外部インデックスストレージとしてサポートしています。

11.2.2.6.1. Elasticsearch IndexManager（実験的）

このインデックスマネージャーは、すべてのインデックスを外部 Elasticsearch サーバーに転送します。これは実験的な統合であり、@IndexedEmbedded アノテーションの indexNullAs は 現在サポートされていません。

設定:

```
<indexing index="LOCAL">
```

```
    <property name="default.indexmanager">elasticsearch</property>
    <property name="default.elasticsearch.host">link:http://elasticHost:9200</property>
    <!-- other elasticsearch configurations -->
  </indexing>
```

Red Hat Data Grid は Elasticsearch を単一共有インデックスとみなすため、インデックスモードを LOCAL に設定する必要があります。設定プロパティーの完全な説明を含む Elasticsearch インテグレーションに関する詳細は、Hibernate Search マニュアル を参照してください。

11.2.2.7. 自動設定

auto-config 属性は、キャッシュタイプに基づいてインデックスを設定する簡単な方法を提供します。レプリケートされたキャッシュとローカルキャッシュの場合、インデックスはディスクで永続化するよう設定され、他のプロセスと共有されません。また、オブジェクトがインデックス化される時点と検索に利用できる時間（ほぼリアルタイム）の間に最小遅延が発生するように設定されます。

```
<local-cache name="default">
  <indexing index="LOCAL" auto-config="true">
  </indexing>
</local-cache>
```

注記

auto-config で追加されたプロパティーを再定義し、新しいプロパティーを追加して高度なチューニングが可能になります。

auto 設定は、レプリケートされたキャッシュとローカルキャッシュに以下のプロパティーを追加します。

プロパティー名	value	description
default.directory_provider	Filesystem	ファイルシステムベースのインデックス。詳細は、Hibernate Search のドキュメントを参照してください。
default.exclusive_index_use	true	排他的モードでのインデックス操作（Hibernate Search による書き込みの最適化）

プロパティー名	value	description
default.indexmanager	near-real-time	ほぼリアルタイム機能をお使いください。つまり、インデックス化されたオブジェクトが検索ですぐに利用可能であることを意味します。
default.reader.strategy	shared	複数のクエリー間でインデックスリーダーを再利用するため、再度開かれないようにします。

分散キャッシュの場合、auto-config は Red Hat Data Grid 自体のインデックスをセットアップし、インデックスへの書き込みを行う単一ノードにインデックス操作を送信するマスター/スレーブメカニズムとして処理されます。

分散キャッシュの自動設定プロパティーは次のとおりです。

プロパティー名	value	description
default.directory_provider	infinispan	Red Hat Data Grid に保存されるインデックス。詳細は、Hibernate Search のドキュメントを参照してください。
default.exclusive_index_use	true	排他的モードでのインデックス操作（Hibernate Search による書き込みの最適化）
default.indexmanager	org.infinispan.query.indexmanager.InfinispanIndexManager	Red Hat Data Grid クラスターの単一ノードにインデックスの書き込みを委譲します。
default.reader.strategy	shared	複数のクエリー間でインデックスリーダーを再利用し、再度開かれないようにします。

11.2.2.8. インデックスの再インデックス

キャッシュに保存されているデータから再構築して Lucene インデックスを再構築しないといけない場合があります。タイプでインデックス化される内容の定義を変更する場合や、Analyzer パラメーターなどを変更すると、インデックスの記述方法に影響するため、インデックスを再構築する必要があります。また、一部のシステム管理で破棄された場合は、インデックスを再構築しないといけない場合があります。インデックスを再構築するには、MassIndexer への参照を取得して開始します。グリッド内のすべてのデータを再処理する必要があるので、注意が必要です。

```
// Blocking execution
SearchManager searchManager = Search.getSearchManager(cache);
searchManager.getMassIndexer().start();

// Non blocking execution
CompletableFuture<Void> future = searchManager.getMassIndexer().startAsyc();
```

ヒント

これは、名前 org.infinispan:type=Query,manager="{name-of-cache-manager}",cache="{name-of-cache}",component=MassIndexer に登録されている MassIndexer MBean で start JMX 操作としても利用できます。

11.2.2.9. マッピングエンティティー

Red Hat Data Grid は、エンティティーレベルでインデックスの詳細な設定を定義するために、Hibernate Search の豊富な API に依存します。この設定には、アノテーションが付けられたフィールド、使用するアナライザー、ネストされたオブジェクトのマッピング方法などが含まれます。詳細ドキュメントは、Hibernate Search のマニュアルを参照してください。

11.2.2.9.1. @DocumentId

Hibernate Search とは異なり、@DocumentId を使用してフィールドを識別子としてマークしても Red Hat Data Grid の値には適用されません。Red Hat Data Grid では、@Indexed オブジェクトすべての識別子は値を保存するために使用されるキーになります。@Transformable、カスタムタイプ、およびカスタム FieldBridge 実装の組み合わせを使用して、キーのインデックス化方法をカスタマイズできます。

11.2.2.9.2. @Transformable keys

各値のキーもインデックス化する必要があります。また、キーインスタンスは String で変換する必要があります。Red Hat Data Grid には、共通のプリミティブをエンコードするためのデフォルトの変換ルーチンが含まれていますが、カスタムキーを使用するには、org.infinispan.query.Transformer の実装を提供する必要があります。

Registering a Transformer via annotations

キータイプに org.infinispan.query.Transformable アノテーションを付けます。

```
@Transformable(transformer = CustomTransformer.class)
public class CustomKey {
   ...
```

```
}

public class CustomTransformer implements Transformer {
  @Override
  public Object fromString(String s) {
    ...
    return new CustomKey(...);
  }

  @Override
  public String toString(Object customType) {
    CustomKey ck = (CustomKey) customType;
    return ...
  }
}
```

Registering a Transformer programmatically

この手法を使用して、カスタムキータイプにアノテーションを付ける必要はありません。

```
org.infinispan.query.SearchManager.registerKeyTransformer(Class<?>, Class<? extends
Transformer>)
```

11.2.2.9.3. プログラムによるマッピング

アノテーションを使用してエンティティーをインデックスにマップする代わりに、プログラムで設定することもできます。

以下の例では、グリッドに保管されるオブジェクト Author をマッピングし、クラスにアノテーションを付けずに 2 つのプロパティーで検索できるようにします。

```
import org.apache.lucene.search.Query;
import org.hibernate.search.cfg.Environment;
import org.hibernate.search.cfg.SearchMapping;
import org.hibernate.search.query.dsl.QueryBuilder;
import org.infinispan.Cache;
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.configuration.cache.Index;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.query.CacheQuery;
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;

import java.io.IOException;
import java.lang.annotation.ElementType;
import java.util.Properties;
```

```
SearchMapping mapping = new SearchMapping();
mapping.entity(Author.class).indexed()
      .property("name", ElementType.METHOD).field()
      .property("surname", ElementType.METHOD).field();

Properties properties = new Properties();
properties.put(Environment.MODEL_MAPPING, mapping);
properties.put("hibernate.search.[other options]", "[...]");

Configuration infinispanConfiguration = new ConfigurationBuilder()
      .indexing().index(Index.LOCAL)
      .withProperties(properties)
      .build();

DefaultCacheManager cacheManager = new DefaultCacheManager(infinispanConfiguration);

Cache<Long, Author> cache = cacheManager.getCache();
SearchManager sm = Search.getSearchManager(cache);

Author author = new Author(1, "Manik", "Surtani");
cache.put(author.getId(), author);

QueryBuilder qb = sm.buildQueryBuilderForClass(Author.class).get();
Query q = qb.keyword().onField("name").matching("Manik").createQuery();
CacheQuery cq = sm.getQuery(q, Author.class);
assert cq.getResultSize() == 1;
```

11.2.3. API のクエリー

以下を使用して Red Hat Data Grid にクエリーできます。

- Lucene または Hibernate Search クエリー。Red Hat Data Grid は Hibernate Search DSL を公開し、Lucene クエリーを生成します。1 つのノードで Lucene クエリーを実行したり、Red Hat Data Grid クラスターの複数のノードにクエリーをブロードキャストしたりできます。

- Ickle クエリー。全文の拡張機能を含むカスタムの文字列ベースのクエリー言語です。

11.2.3.1. Hibernate Search

インデックス化を設定するために Hibernate Search アノテーションをサポートする以外に、他の Hibernate Search API を使用してキャッシュをクエリーすることもできます。

11.2.3.1.1. Lucene クエリーの実行

Lucene クエリーを直接実行するには、CacheQuery で作成してラップします。

```
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;
import org.apache.lucene.Query;


SearchManager searchManager = Search.getSearchManager(cache);
Query query = searchManager.buildQueryBuilderForClass(Book.class).get()
        .keyword().wildcard().onField("description").matching("*test*").createQuery();
CacheQuery<Book> cacheQuery = searchManager.getQuery(query);
```

11.2.3.1.2. Hibernate Search DSL の使用

Hibernate Search DSL を使用して Lucene クエリーを作成できます。以下に例を示します。

```
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;
import org.apache.lucene.search.Query;

Cache<String, Book> cache = ...

SearchManager searchManager = Search.getSearchManager(cache);

Query luceneQuery = searchManager
                .buildQueryBuilderForClass(Book.class).get()
                .range().onField("year").from(2005).to(2010)
                .createQuery();

List<Object> results = searchManager.getQuery(luceneQuery).list();
```

この DSL のクエリー機能の詳細は、Hibernate Search マニュアル の関連セクションを参照してください。

11.2.3.1.3. Ffted Search

Red Hat Data Grid は、Hibernate Search FacetManager を使用して Faceted Search をサポートします。

```
// Cache is indexed
Cache<Integer, Book> cache = ...

// Obtain the Search Manager
SearchManager searchManager = Search.getSearchManager(cache);

// Create the query builder
QueryBuilder queryBuilder = searchManager.buildQueryBuilderForClass(Book.class).get();
```

```
// Build any Lucene Query. Here it's using the DSL to do a Lucene term query on a book name
Query luceneQuery =
queryBuilder.keyword().wildcard().onField("name").matching("bitcoin").createQuery();

// Wrap into a cache Query
CacheQuery<Book> query = searchManager.getQuery(luceneQuery);

// Define the Facet characteristics
FacetingRequest request = queryBuilder.facet()
            .name("year_facet")
            .onField("year")
            .discrete()
            .orderedBy(FacetSortOrder.COUNT_ASC)
            .createFacetingRequest();

// Associated the FacetRequest with the query
FacetManager facetManager = query.getFacetManager().enableFaceting(request);

// Obtain the facets
List<Facet> facetList = facetManager.getFacets("year_facet");
```

上記のお気に入り検索は、年ごとにリリースされた「ビットカン」に一致する数字の書籍を返します。以下に例を示します。

```
AbstractFacet{facetingName='year_facet', fieldName='year', value='2008', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2009', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2010', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2011', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2012', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2016', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2015', count=2}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2013', count=3}
```

Faceted Search の詳細は、「Hibernate Search Faceting」を参照してください。

11.2.3.1.4. 空間クエリー

Red Hat Data Grid は Spatial Queries にも対応しているため、距離、ジオメン、地理座標に基づいて全テキストと制限を組み合わせることができます。

たとえば、検索されるエンティティーで @Spatial アノテーションを使用し、@Latitude および @Longitude とともに使用します。

```
@Indexed
@Spatial
public class Restaurant {
```

```
    @Latitude
    private Double latitude;

    @Longitude
    private Double longitude;

    @Field(store = Store.YES)
    String name;

    // Getters, Setters and other members omitted

}
```

空間クエリーを実行するには、Hibernate Search DSL を使用できます。

```
// Cache is configured as indexed
Cache<String, Restaurant> cache = ...

// Obtain the SearchManager
Searchmanager searchManager = Search.getSearchManager(cache);

// Build the Lucene Spatial Query
Query query =
Search.getSearchManager(cache).buildQueryBuilderForClass(Restaurant.class).get()
        .spatial()
          .within( 2, Unit.KM )
            .ofLatitude( centerLatitude )
            .andLongitude( centerLongitude )
          .createQuery();

// Wrap in a cache Query
CacheQuery<Restaurant> cacheQuery = searchManager.getQuery(query);

List<Restaurant> nearBy = cacheQuery.list();
```

Hibernate Search マニュアルの詳細

11.2.3.1.5. IndexedQueryMode

インデックス化されたクエリーのクエリーモードを指定できます。IndexedQueryMode.BROADCAST を使用すると、クラスターの各ノードにクエリーをブロードキャストし、結果を取得し、呼び出し元に戻る前にそれらを組み合わせることができます。各ノードのローカル インデックスにはインデックス化されたデータのサブセットのみがあるため、共有されていない インデックスと併用することが適切です。

IndexedQueryMode.FETCH は呼び出し元でクエリーを実行します。クラスター全体のデータのインデックスすべてがローカルで使用できる場合、パフォーマンスは最適になります。そうでないと、こ

のクエリーモードにはリモートノードからインデックスデータを取得することが必要になる場合があります。

IndexedQueryMode は、現時点で Lucene Queries および Ickle String クエリー（Red Hat Data Grid Query DSL なし）に対してサポートされます。

例:

```
CacheQuery<Person> broadcastQuery = Search.getSearchManager(cache).getQuery(new
MatchAllDocsQuery(), IndexedQueryMode.BROADCAST);

List<Person> result = broadcastQuery.list();
```

11.2.3.2. Red Hat Data Grid Query DSL

Red Hat Data Grid は、Lucene および Hibernate Search からは独立して独自のクエリー DSL を提供します。基礎となるクエリーおよびインデックス化メカニズムからクエリー API を切り離すことで、今後新しい代替エンジンを導入でき、Lucene 以外にも、同じ統一されたクエリー API を使用できます。インデックス化と検索の現在の実装は、Hibernate Search および Lucene をベースとしているため、本章で紹介しているインデックス関連の要素はすべて適用されます。

新しい API は、Lucene クエリーオブジェクトを構築する低レベル詳細にユーザーを公開せずにクエリーの書き込みを簡素化し、remote Hot クライアントに利用可能な利点があります。しかし、より詳細な詳細を解決する前に、まず前の例から Book エンティティーのクエリーを記述する簡単な例を確認してください。

Red Hat Data Grid のクエリー DSL を使用したクエリー例

```
import org.infinispan.query.dsl.*;

// get the DSL query factory from the cache, to be used for constructing the Query object:
QueryFactory qf = org.infinispan.query.Search.getQueryFactory(cache);

// create a query for all the books that have a title which contains "engine":
org.infinispan.query.dsl.Query query = qf.from(Book.class)
    .having("title").like("%engine%")
    .build();

// get the results:
List<Book> list = query.list();
```

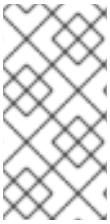
API は **org.infinispan.query.dsl** *パッケージにあります。クエリーは、キャッシュごとの* **SearchManager** *から取得する* **QueryFactory** *インスタンスを使って作成されます。各* **QueryFactory** *インスタンスは* **SearchManager** *と同じ* **Cache** *インスタンスにバインドされますが、複数のクエリーを並行して作成するために使用されるステートレスおよびスレッドセーフオブジェクトです。*

クエリーの作成は、 **from(Class entityType)** *メソッドの呼び出しから始まります。これは、指定のキャッシュから指定のエンティティークラスにターゲットが設定されるクエリーを作成する* **QueryBuilder** *オブジェクトを返します。*

注記

クエリーは常に単一のエンティティータイプをターゲットにし、単一のキャッシュの内容に対して評価されます。複数のキャッシュでクエリを実行したり、複数のエンティティタイプ（結合）を対象とするクエリを作成したりすることは、サポートされていません。

QueryBuilder *は、DSL メソッドの呼び出しを通じて指定された検索条件と設定を累積し、構築を完了する* **Query** *Builder.build（）メソッドの呼び出しによって* **Query** *オブジェクトをビルド するために使用されます。ステートフルなオブジェクトの場合、これは同時に複数のクエリーを構築するのに使用できません（ ネストされたクエリーを除く）。ただし、後で再利用できます。*



注記

この **QueryBuilder** *は Hibernate Search とは異なりますが、同じ名前になります。曖昧さを防ぐために、今後名前を変更することを検討しています。*

クエリーを実行し、結果の取得は、 **Query** *オブジェクトの* **list**（） *メソッドを呼び出すのと同じくらい簡単です。実行すると、* **Query** *オブジェクトは再利用できません。新しい結果を取得するために新しい結果を再実行する必要がある場合は、* **QueryBuilder.build**（） *を呼び出すことで新しいインスタンスを取得する必要があります。*

11.2.3.2.1. Operator のフィルタリング

クエリーの構築は、複数の基準を構成する階層プロセスであり、この階層に従って説明するのが最適です。

クエリー基準の最も単純な形式は、ゼロ以上の引数を許可するフィルター演算子に従ってエンティティー属性の値の制限です。entity 属性は、利用可能なすべての演算子を公開する中間コンテキストオブジェクト(FilterConditionEndContext)を返すクエリービルダーの having(String attributePath) メソッドを呼び出して指定されます。FilterConditionEndContext で定義される各メソッドは、2 つの引数を持ち、引数のない isNull を除く、引数を受け入れる演算子です。引数はクエリーが構築される際に静的に評価されます。そのため、SQL の相関サブクィーストと同様の機能を探す場合、これが現在利用できません。

```
// a single query criterion
QueryBuilder qb = ...
qb.having("title").eq("Hibernate Search in Action");
```

表11.1 FilterConditionEndContext は以下のフィルター Operator を公開します。

フィルター	引数	説明
in	コレクションの値	左のオペランドが引数として指定された値のコレクションからの要素のいずれかと等しいことを確認します。
in	Object... 値	左のオペランドが、引数として指定される値の（固定）一覧と等しくなることを確認します。
contains	オブジェクト値	左側の引数（配列またはコレクション）に指定の要素が含まれることを確認します。
containsAll	コレクションの値	左の引数（配列またはコレクション）に指定のコレクションのすべての要素が任意の順序で含まれていることを確認します。
containsAll	Object... 値	左の引数（配列またはコレクション）に指定の全要素が任意の順序で含まれていることを確認します。
containsAny	コレクションの値	左側の引数（配列またはコレクション）に指定のコレクションの要素が含まれていることを確認します。
containsAny	Object... 値	左側の引数（配列またはコレクション）に指定の要素が含まれることを確認します。
isNull		左側の引数が null であることを確認します。

フィルター	引数	説明
like	文字列パターン	(文字列として想定される) 左側の引数が、JPA ルールに準拠するワイルドカードパターンと一致することを確認します。
eq	オブジェクト値	左側の引数が指定された値と同じであることを確認します。
equal	オブジェクト値	eq のエイリアス。
gt	オブジェクト値	左側の引数が指定の値よりも大きいことを確認します。
gte	オブジェクト値	左側の引数が指定の値以上であることを確認します。
lt	オブジェクト値	左側の引数が指定の値未満であることを確認します。
LTE	オブジェクト値	左側の引数が指定の値以下であることを確認します。
between	オブジェクトからオブジェクト	左側の引数が指定された範囲の制限の間にあることを確認します。

クエリー構築には、適切なシーケンスで実行する必要があるメソッド呼び出しの複数ステップチェーンが必要であり、適切に完了しなければならず、2 回目に実行しないようにしてください。そうでない場合にはエラーが生じます。以下の例は無効であり、それぞれのケースによって基準が無視され（無害なケース）、または例外がスローされます（より深刻なケース）。

```
// Incomplete construction. This query does not have any filter on "title" attribute yet,
// although the author may have intended to add one.
QueryBuilder qb1 = ...
qb1.having("title");
Query q1 = qb1.build(); // consequently, this query matches all Book instances regardless of title!

// Duplicated completion. This results in an exception at run-time.
// Maybe the author intended to connect two conditions with a boolean operator,
// but this does NOT actually happen here.
QueryBuilder qb2 = ...
qb2.having("title").like("%Data Grid%");
qb2.having("description").like("%clustering%");   // will throw java.lang.IllegalStateException: Sentence already started. Cannot use 'having(..)' again.
Query q2 = qb2.build();
```

11.2.3.2.2. 埋め込みエンティティーの属性に基づくフィルタリング

having メソッドは 埋め込みエンティティー 属性を参照するドットで区切られた属性パスも受け入れるため、以下は有効なクエリーになります。

```
// match all books that have an author named "Manik"
Query query = queryFactory.from(Book.class)
    .having("author.name").eq("Manik")
    .build();
```

属性パスの各部分は、対応するエンティティーまたは埋め込みエンティティークラスでそれぞれ既存のインデックス化された属性を参照する必要があります。複数のレベルを埋め込むことができます。

11.2.3.2.3. ブール値の条件

以下の例では、複数の属性条件を論理結合 (and) および非結合 (or) 演算子と組み合わせて、より複雑な条件を作成する方法を示しています。ブール演算子のよく知られている演算子の優先順位ルールはここで適用されるので、構築時の DSL メソッド呼び出しの順序は無関係です。ここで、or が最初に呼び出された場合でも、and Operator の優先順位は or よりも高くなります。

```
// match all books that have "Data Grid" in their title
// or have an author named "Manik" and their description contains "clustering"
Query query = queryFactory.from(Book.class)
  .having("title").like("%Data Grid%")
  .or().having("author.name").eq("Manik")
  .and().having("description").like("%clustering%")
  .build();
```

ブール値の否定は not 演算子で達成されます。

```
// match all books that do not have "Data Grid" in their title and are authored by "Manik"
Query query = queryFactory.from(Book.class)
  .not().having("title").like("%Data Grid%")
  .and().having("author.name").eq("Manik")
  .build();
```

11.2.3.2.4. ネストされた条件

論理演算子の優先順位の変更は、ネスト化されたフィルター条件を使用して実行できます。論理演算子を使用すると、前述のように 2 つの単純な属性条件に接続できますが、同じクエリーファクトリーで作成された後続の複雑な条件で単純な属性条件に接続することもできます。

```
// match all books that have an author named "Manik" and their title contains
// "Data Grid" or their description contains "clustering"
Query query = queryFactory.from(Book.class)
  .having("author.name").eq("Manik")
```

```
  .and(queryFactory.having("title").like("%Data Grid%")
        .or().having("description").like("%clustering%"))
  .build();
```

11.2.3.2.5. Projections

一部のユースケースでは、属性のごく一部のみがアプリケーションによって実際に使用されている場合、特にドメインエンティティにエンティティが埋め込まれている場合、ドメインオブジェクト全体を返すのはやり過ぎです。クエリ言語を使用すると、プロジェクションを返す属性（または属性パス）のサブセットを指定できます。projections を使用すると、Query.list（） はドメインエンティティー全体を返しませんが、Object[] のリストを返します。これは、Projected 属性に対応するアレイの各スロットを返します。

```
// match all books that have "Data Grid" in their title or description
// and return only their title and publication year
Query query = queryFactory.from(Book.class)
  .select("title", "publicationYear")
  .having("title").like("%Data Grid%")
  .or().having("description").like("%Data Grid%"))
  .build();
```

11.2.3.2.6. ソート

1 つ以上の属性または属性パスに基づいて結果を順序付けるには、属性パスとソート方向を受け入れる QueryBuilder.orderBy（） メソッドを使用します。複数の並べ替え基準を指定すると、orderBy メソッドの呼び出しの順序が優先されます。ただし、各属性の個別ソート操作シーケンスではなく、指定された属性のタプルに対して、複数のソート基準を考える必要があります。

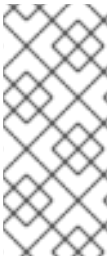
```
// match all books that have "Data Grid" in their title or description
// and return them sorted by the publication year and title
Query query = queryFactory.from(Book.class)
  .orderBy("publicationYear", SortOrder.DESC)
  .orderBy("title", SortOrder.ASC)
  .having("title").like("%Data Grid%")
  .or().having("description").like("%Data Grid%"))
  .build();
```

11.2.3.2.7. ページネーション

QueryBuilder の maxResults プロパティーを設定すると、返される結果の数を制限できます。これは、結果セットのページネーションを実現するために startOffset の設定と併用できます。

```
// match all books that have "clustering" in their title
// sorted by publication year and title
// and return 3'rd page of 10 results
Query query = queryFactory.from(Book.class)
  .orderBy("publicationYear", SortOrder.DESC)
  .orderBy("title", SortOrder.ASC)
```

```
.startOffset(20)
.maxResults(10)
.having("title").like("%clustering%")
.build();
```



注記

フェッチされる結果が maxResults に制限されている場合でも、Query.getResultSize() を呼び出すことで、一致する結果の合計数を見つけることができます。

11.2.3.2.8. グループ化およびアグリゲーション

Red Hat Data Grid には、グループ化フィールドのセットに従ってクエリーの結果をグループ化し、各グループに含まれる値のセットに集約機能を適用して、各グループからの結果の集約を構築する機能があります。グループ化と集約は、プロジェクトクエリーにのみ適用できます。サポートされる集計は avg、sum、count、max、min です。グループ化フィールドのセットは groupBy(field) メソッドで指定され、複数回起動できます。グループ化フィールドの定義に使用される順序は関係ありません。プロジェクションで選択されたすべてのフィールドは、グループ化フィールドであるか、以下で説明するグループ化関数の1つを使用して集約される必要があります。Projection フィールドは集約され、同時にグループ化に使用できます。フィールドをグループ化し、集約フィールドは一切選択しないクエリーです。

例： 作成者およびカウントによる書籍のグループ化

```
Query query = queryFactory.from(Book.class)
   .select(Expression.property("author"), Expression.count("title"))
   .having("title").like("%engine%")
   .groupBy("author")
   .build();
```

注記

選択したすべてのフィールドに集計関数が適用され、グループ化にフィールドが使用されないプロジェクションクエリが許可されます。この場合、集計は、単一のグローバルグループが存在するかのようにグローバルに計算されます。

11.2.3.2.9. 集約

avg、sum、count、max、min などの集約関数をフィールドに適用できます。

- avg()：数字の平均を計算します。許可される値は、java.lang.Number のプリミティブ番号およびインスタンスです。結果は java.lang.Double として表されます。null 以外の値がな

い場合、結果は代わりに null になります。

- count（）: null 以外の行の数をカウントし、java.lang.Long を返します。null 以外の値がない場合、結果は代わりに 0 になります。

- max（）: 見つかった最大値を返します。許可される値は java.lang.Comparable のインスタンスである必要があります。null 以外の値がない場合、結果は代わりに null になります。

- min（）: 見つかった最小値を返します。許可される値は java.lang.Comparable のインスタンスである必要があります。null 以外の値がない場合、結果は代わりに null になります。

- sum（）- 数値の合計を計算します。null 以外の値がない場合、結果は代わりに null になります。以下の表は、指定のフィールドに基づいて返されるタイプを示しています。

表11.2 テーブル合計戻り値のタイプ

フィールドタイプ	戻り値のタイプ
Integral (BigInteger 以外)	Long
Float または Double	double
BigInteger	BigInteger
BigDecimal	BigDecimal

11.2.3.2.10. グループ化および集計を使用したクエリーの評価

集計クエリーには、通常のクエリーのようにフィルター条件を含めることができます。フィルタリングは、グループ化操作の前後の 2 つのステージで実行できます。groupBy メソッドを呼び出す前に定義されるすべてのフィルター条件は、grouped 操作が実行される前に適用され、キャッシュエントリーに直接適用されます（最終的なプロジェクトではありません）。これらのフィルター条件はクエリーされたエンティティータイプのフィールドを参照する可能性があり、グループ化ステージに対する入力となるデータセットを制限することを目的としています。groupBy メソッドを呼び出した後に定義されたすべてのフィルター条件は、projection および grouping 操作から得られるプロジェクトに適用されます。これらのフィルター条件は、groupBy フィールドまたは集約フィールドのいずれかを参照できます。select句で指定されていない集約フィールドを参照することは許可されています。ただし、非集計フィールドと非グループ化フィールドを参照することは禁止されています。このフェーズでフィルタリングすると、プロパティに基づいてグループの数が減ります。並べ替えは通常のクエリーと同様に指定できます。順序付け操作は、grouping 操作の後に実行され、groupBy フィールドまたは集約フィールドのいずれかを参照できます。

11.2.3.2.11. 名前付きクエリーパラメーターの使用

実行ごとに新しいQueryオブジェクトを作成する代わりに、実行前に実際の値に置き換えることができる名前付きパラメーターをクエリに含めることができます。これにより、クエリーを 1 度定義し、複数回効率的に実行できます。パラメーターは Operator の右側でのみ使用でき、通常の定数値ではなく、org.infinispan.query.dsl.Expression.param(String paramName) メソッドによって生成されたオブジェクトを Operator に指定すると、クエリーが作成されたときに定義されます。パラメーターを定義したら、以下の例にあるように Query.setParameter(parameterName, value) または Query.setParameters(parameterMap) を呼び出すことで設定できます。

```
import org.infinispan.query.Search;
import org.infinispan.query.dsl.*;
[...]

QueryFactory queryFactory = Search.getQueryFactory(cache);
// Defining a query to search for various authors and publication years
Query query = queryFactory.from(Book.class)
   .select("title")
   .having("author").eq(Expression.param("authorName"))
   .and()
   .having("publicationYear").eq(Expression.param("publicationYear"))
   .build();

// Set actual parameter values
query.setParameter("authorName", "Doe");
query.setParameter("publicationYear", 2010);

// Execute the query
List<Book> found = query.list();
```

または、実際のパラメーター値のマッピングを指定して、複数のパラメーターを 1 度に設定できます。

複数の名前付きパラメーターを一度に設定する

```
import java.util.Map;
import java.util.HashMap;

[...]

Map<String, Object> parameterMap = new HashMap<>();
parameterMap.put("authorName", "Doe");
parameterMap.put("publicationYear", 2010);

query.setParameters(parameterMap);
```

注記

クエリーの解析、検証、および実行計画の作業の大部分は、パラメーターでのクエリーの最初の実行時に実行されます。この作業は後続の実行時には繰り返し行われないため、クエリーパラメーターではなく定数値を使用した同様のクエリーの場合よりもパフォーマンスが向上します。

11.2.3.2.12. その他のクエリー DSL の例

Query DSL API を使用して確認する最適な方法は、テストスイートを確認することです。QueryDslConditionsTest は詳細な例です。

11.2.3.3. Ickle

Ickle クエリー言語を使用して、ライブラリーおよびリモートクライアント/サーバーモードの両方でリレーショナルおよびフルテキストクエリーを作成します。

Ickle は文字列ベースで、以下の特徴があります。

- *Java クラスをクエリーし、プロトコルバッファーをサポートします。*
- *クエリーは単一のエンティティータイプをターゲットにできます。*
- *クエリーは、コレクションを含む埋め込みオブジェクトのプロパティーでフィルタリングできます。*
- *プロジェクト、集計、ソート、名前付きパラメーターをサポートします。*
- *インデックス付きおよびインデックスなしの実行をサポートします。*
- *複雑なブール式をサポートします。*
- *フルテキストクエリーをサポートします。*

- *user. age > sqrt(user.shoeSize+3) などの式での計算をサポートしません。*
- *参加はサポートしていません。*
- *サブクストリーはサポートしません。*
- *さまざまな Red Hat Data Grid} API でサポートされています。Query が QueryBuilder によって生成されると、継続的なクエリーやリスナーのイベントフィルターなど、Query が受け入れられます。*

API を使用するには、最初にキャッシュに QueryFactory を取得し、クエリーで使用する文字列を渡す .create（）メソッドを呼び出します。たとえば、以下のようになります。

```
QueryFactory qf = Search.getQueryFactory(remoteCache);
Query q = qf.create("from sample_bank_account.Transaction where amount > 20");
```

フルテキスト演算子で使用されるすべてのフィールドを使用する場合は、インデックス付きと Analysed の両方 に する必要があります。

11.2.3.3.1. Ickle クエリー言語パーサー構文

Ickle クエリー言語のパーサー構文には、いくつかの重要なルールがあります。

- *空白は重要ではありません。*
- *フィールド名ではワイルドカードはサポートされません。*
- *デフォルトのフィールドがないため、フィールド名またはパスは必ず指定する必要があります。*
- *&& および || は、フルテキストと JPA 述語の両方で、AND または OR の代わりに使用できます。*

- ! は NOT の代わりに使用できます。
- 足りないブール値OperatorはORとして解釈されます。
- 文字列の用語は、 一重引用符または二重引用符で囲む必要があります。
- ファジー性とブースティングは任意の順序で受け入れられず、常にファジー性が最初になります。
- <> の代わりに != が許可されます。
- ブーディングは、>、>=、<、⇐ Operator には適用できません。同じ結果を達成するために範囲を使用することができます。

11.2.3.3.2. Fuzzy クエリー

ファジークエリー add ~ を整数とともに実行するには、用語の後に使用される用語からの距離を表します。たとえば、以下のようになります。

```
Query fuzzyQuery = qf.create("from sample_bank_account.Transaction where description :
'cofee'~2");
```

11.2.3.3.3. 範囲クエリー

以下の例に示すように、範囲クエリーを実行するには、中括弧のペア内で指定の境界を定義します。

```
Query rangeQuery = qf.create("from sample_bank_account.Transaction where amount : [20
to 50]");
```

11.2.3.3.4. フレーズクエリー

以下の例のように、単語のグループは引用符で囲むことで検索できます。

```
Query q = qf.create("from sample_bank_account.Transaction where description : 'bus fare'");
```

11.2.3.3.5. 近接クエリー

特定の距離内で2つの用語を検索して近接クエリを実行するには、フレーズの後に距離とともに~を追加します。たとえば、以下の例では、キャンセル と fee という単語が 3 個以上ありません。

```
Query proximityQuery = qf.create("from sample_bank_account.Transaction where description
: 'canceling fee'~3 ");
```

11.2.3.3.6. ワイルドカードクエリー

1 文字と複数の文字のワイルドカード検索の両方を実行できます。

- 1 文字のワイルドカード検索は ? 文字で使用できます。
- 複数の文字のワイルドカード検索は * 文字で使用できます。

テキストの検索またはテストには、以下の 1 文字のワイルドカード検索が使用されます。

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where description :
'te?t'");
```

テスト、テスト、またはテストに使用するには、以下の複数文字のワイルドカード検索が使用されます。

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where description :
'test*'");
```

11.2.3.3.7. 正規表現クエリー

正規表現クエリーは、/ 間でパターンを指定することにより実行できます。Ickle は Lucene の正規表現構文を使用して、moat または boat という語を検索し、以下を使用できます。

```
Query regExpQuery = qf.create("from sample_library.Book  where title : /[mb]oat/");
```

11.2.3.3.8. クエリーのブースト

特定のクエリーで関連性を増やす用語の後に ^ を追加することで、用語を強化することができます。たとえば、ビールとビールとの関連性が3倍高いビールとワインを含むタイトルを検索するには、次のように使用できます。

```
Query boostedQuery = qf.create("from sample_library.Book where title : beer^3 OR wine");
```

11.2.3.4. 継続的なクエリー

継続的なクエリーにより、アプリケーションはクエリーフィルターに現在一致したエントリーを受信するリスナーを登録し、さらにキャッシュ操作の結果としてクエリーされたデータセットへの変更を継続的に通知できます。これには、セットに結合された値の着信一致、更新された一致、変更されて引き続き一致する一致値、およびセットを離れた値の発信一致が含まれます。継続的なクエリーを使用することにより、アプリケーションは、変更を検出するために同じクエリーを繰り返し実行する代わりに、イベントの安定したストリームを受信し、リソースがより効率的に使用されるようになります。たとえば、以下のユースケースすべてで、継続的なクエリーを使用できます。

- 18 から 25 までの年齢が指定された全ユーザーを返します（Person エンティティーに age プロパティーがあり、ユーザーアプリケーションによって更新されることを前提とします）。
- $2000 を超えるすべてのトランザクションを返す。
- F1レーサーのラップスピードが 1:45.00 秒未満だったすべての時間を返す (キャッシュにラップエントリーが含まれていて、レース中にラップがライブ入力されていると仮定)。

11.2.3.4.1. 連続クエリー実行

継続的クエリーは、以下の場合に通知されるリスナーを使用します。

- エントリーは、Join イベントで表される指定のクエリーの一致を開始します。
- 一致するエントリーは更新され、Update イベントによって表されるクエリーと引き続き一致します。
- Leave イベントで表されるクエリーの照合を停止します。

クライアントが継続的なクエリーリスナーを登録すると、上記のように Join イベントとして受信される現在クエリーに一致する結果の受信が即座に開始します。さらに、作成、変更、削除、または有効期限のイベントを生成するキャッシュ操作が原因で、他のエントリーがクエリーに一致するか、クエリーの一致を止めると、後続の通知を受け取ります。 更新されたキャッシュエントリーは、エントリーが操作前および後のクエリーフィルターと一致する場合に Update events を生成します。要約すると、リスナーが Join、Update、または Leave イベントを受け取るかどうかを決定するために使用されるロジックが以下になります。

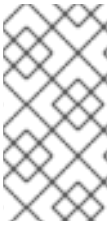
1. 古い値と新しい値の両方に対するクエリーが false と評価された場合、イベントは抑制されます。

2. 古い値のクエリーが false を評価し、新しい値が true を評価すると、Join イベントが送信されます。

3. 古い値と新しい値両方のクエリーが true である場合、Update イベントが送信されます。

4. 古い値のクエリーが true を評価し、新しい値が false を評価すると、Leave イベントが送信されます。

5. 古い値のクエリーが true を評価し、エントリーが削除または期限切れである場合、Sirave イベント が 送信されます。



注記

継続的なクエリーは、グループ化、集計、およびソート操作を除く Query DSL の完全な機能を使用できます。

11.2.3.4.2. 継続的なクエリーの実行

継続的なクエリーを作成するには、最初に Query オブジェクトを作成して開始します。これについては、Query DSL セクションで 説明されています。次に、キャッシュの ContinuousQuery(org.infinispan.query.api.continuous.ContinuousQuery)オブジェクトを取得し、クエリーと継続的なクエリーリスナー (org.infinispan.query.api.continuous.ContinuousQueryListener)を登録する必要があります。リモートモードで実行している場合は、リモートモードまたは org.infinispan.query.Search.getContinuousQuery（Cache<K, V> キャッシュ）で実行している場合は static メソッド org.infinispan.client.hotrod.Search.getContinuousQuery（RemoteCache <K, V> キャッシュ）を呼び出すと、キャッシュ に関連付けられた ContinuousQuery オブジェクトを取得できます。リスナーを作成したら、ContinuousQuery の addContinuousQueryListener メソッドを使用して登録できます。

```
continuousQuery.addContinuousQueryListener(query, listener);
```

以下の例は、埋め込みモードの単純な継続的なクエリーのユースケースを示しています。

継続的クエリーの登録

```
import org.infinispan.query.api.continuous.ContinuousQuery;
import org.infinispan.query.api.continuous.ContinuousQueryListener;
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

[...]

// We have a cache of Persons
Cache<Integer, Person> cache = ...

// We begin by creating a ContinuousQuery instance on the cache
ContinuousQuery<Integer, Person> continuousQuery = Search.getContinuousQuery(cache);

// Define our query. In this case we will be looking for any Person instances under 21 years of
age.
QueryFactory queryFactory = Search.getQueryFactory(cache);
Query query = queryFactory.from(Person.class)
   .having("age").lt(21)
   .build();

final Map<Integer, Person> matches = new ConcurrentHashMap<Integer, Person>();

// Define the ContinuousQueryListener
ContinuousQueryListener<Integer, Person> listener = new ContinuousQueryListener<Integer,
Person>() {
   @Override
   public void resultJoining(Integer key, Person value) {
      matches.put(key, value);
   }

   @Override
   public void resultUpdated(Integer key, Person value) {
      // we do not process this event
   }

   @Override
   public void resultLeaving(Integer key) {
      matches.remove(key);
   }
};
```

```
// Add the listener and the query
continuousQuery.addContinuousQueryListener(query, listener);

[...]

// Remove the listener to stop receiving notifications
continuousQuery.removeContinuousQueryListener(listener);
```

21 未満の年齢が 21 未満の Person インスタンスは、リスナーによって受信され、一致 するマップに配置され、これらのエントリーがキャッシュから削除されたり、年齢が 21 未満になると、一致 するものから削除されます。

11.2.3.4.3. 継続的なクエリーの削除

クエリのそれ以上の実行を停止するには、リスナーを単に削除します。

```
continuousQuery.removeContinuousQueryListener(listener);
```

11.2.3.4.4. 継続的なクエリーのパフォーマンスに関する注意

継続的なクエリーは、アプリケーションに一定の更新ストリームを提供するように設計されており、特に幅広いクエリーに対して非常に多くのイベントが生成される可能性があります。イベントごとに新規の一時的なメモリー割り当てが行われます。この動作により、クエリーが慎重に設計されていない場合に、メモリー不足になる可能性があり、OutOfMemoryErrors （特にリモートモード）が発生する可能性があります。このような問題を防ぐには、各クエリーが、一致するエントリーの数と各一致のサイズの両方で必要な最低限の情報をキャプチャーすることを強く推奨します（プロジェクトを使用して関心のあるプロパティーをキャプチャーすることができます）。また、各 ContinuousQueryListener は、ブロックせずに受信されたすべてのイベントを迅速に処理し、リッスンするキャッシュから新しい一致するイベントを生成しないように設計されていることを確認することが強く推奨されます。

11.3. リモートクエリー

埋め込みクライアントへの Java エンティティーのインデックス化や検索をサポートする以外に、Red Hat Data Grid ではリモート、言語に依存しない、クエリーのサポートが導入されました。

このうるうには、以下の 2 つの重要な変更が必要です。

- Red Hat Data Grid は、Apache Lucene の Java API を直接使用しても、Apache Lucene

の Java API を使用して、独自の新しい クエリー言語 を定義することができないので、Hot クライアントに現在実装されているすべての言語で簡単に実装できる内部 DSL に基づいて、独自の新しいクエリー言語を定義します。

- インデックスを有効にするために、クライアントによってキャッシュに置かれるエンティティーは、クライアントによってのみ認識される不透明なバイナリーブロブではなくなりました。構造はサーバーとクライアントの両方を認識する必要があるため、エンコーディングされたデータを一般的な方法で採用する必要がありました。さらに、複数言語のクライアントがデータにアクセスできるようにするには、言語とプラットフォームに依存しないエンコーディングが必要です。Google の Protocol Buffers は、効率性、堅牢、スキーマの進化に対する優れた多言語のサポートとサポートにより、ネットワーク上の上線とストレージの両方にエンコーディング形式として選択されました。

11.3.1. Protobuf でエンコードされたエンティティーの保存

保存されたエンティティーをインデックス化してクエリーできるリモートクライアントは、Protobuf エンコーディング形式を使用してこれを実行する必要があります。これは、検索機能が機能するための キー です。しかし、Protobuf エンティティーを保存することもできます。これにより、プラットフォーム独立性を活用し、インデックスが必要ない場合にインデックスを有効にしないようにすることもできます。

Protobuf は構造化データに関するものすべてであるため、まずデータの構造を定義します。これは、以下の例のように、.proto ファイルでプロトコルバッファーメッセージタイプを宣言することで実行できます。Protobuf は広範囲にわたり、ここでは説明しません。そのため、詳細な説明は Protobuf 開発者ガイド を参照してください。この例では、book_sample という名前のパッケージに配置される Book という名前のエンティティー（protobuf のメッセージタイプ）を定義します。エンティティーは、プリミティブタイプの複数のフィールドと、作成者 （基本的に配列）という名前の繰り返し可能なフィールドを宣言します。Author メッセージインスタンスは、Book メッセージインスタンスに組み込まれています。

library.proto

```
package book_sample;

message Book {
  required string title = 1;
  required string description = 2;
  required int32 publicationYear = 3; // no native Date type available in Protobuf

  repeated Author authors = 4;
}

message Author {
  required string name = 1;
  required string surname = 2;
}
```

Protobuf メッセージについて以下のような重要な点に留意してください。

- メッセージのネストは可能ですが、作成される構造は厳密にツリーであり、グラフではありません。
- タイプ継承の概念がない
- コレクションはサポート対象外ですが、繰り返しフィールドを使用してアレイを簡単にエミュレートできます。

Java Hotgitops クライアントでの Protobuf の使用は 2 つの手順です。まず、クライアントは専用のマーシャラーである ProtoStreamMarshaller を使用するように設定する必要があります。このマーシャラーは、ProtoStream ライブラリーを使用してオブジェクトのエンコードを支援します。2 番目のステップは、メッセージタイプをマーシャリングする方法について ProtoStream ライブラリーに指示します。以下の例は、このプロセスを強調表示しています。

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.marshall.ProtoStreamMarshaller;
import org.infinispan.protostream.SerializationContext;
...

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
   .host("10.1.2.3").port(11234)
   .marshaller(new ProtoStreamMarshaller());

RemoteCacheManager remoteCacheManager = new
RemoteCacheManager(clientBuilder.build());

SerializationContext serCtx =
ProtoStreamMarshaller.getSerializationContext(remoteCacheManager);

FileDescriptorSource fds = new FileDescriptorSource();
fds.addProtoFiles("/library.proto");
serCtx.registerProtoFiles(fds);
serCtx.registerMarshaller(new BookMarshaller());
serCtx.registerMarshaller(new AuthorMarshaller());

// Book and Author classes omitted for brevity
```

この例における興味深い部分は、RemoteCacheManager に関連付けられた SerializationContext

を取得し、マーシャリングしたい protobuf タイプについて ProtoStream に伝えています。SerializationContext は、この目的のためにライブラリーによって提供されます。SerializationContext.registerProtoFiles メソッドは、タイプ宣言が含まれる protobuf 定義であることが想定される 1 つ以上のクラスパスリソースの名前を受信します。

注記

RemoteCacheManager には、ProtoStreamMarshaller を使用するように設定されていない限り、SerializationContext が関連付けられていません。

次の関連する部分は、ドメインモデルタイプのエンティティーマーシャラーごとに登録されます。各タイプに対してユーザーが提供する必要があります。そうでないと、マーシャリングは失敗します。マーシャラーの作成は単純なプロセスです。BookMarshaller のサンプルは開始する必要があります。考慮すべき最も重要な点は、ステートレスであり、スレッドセーフでなければなりません。

BookMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;
...

public class BookMarshaller implements MessageMarshaller<Book> {

   @Override
   public String getTypeName() {
      return "book_sample.Book";
   }

   @Override
   public Class<? extends Book> getJavaClass() {
      return Book.class;
   }

   @Override
   public void writeTo(ProtoStreamWriter writer, Book book) throws IOException {
      writer.writeString("title", book.getTitle());
      writer.writeString("description", book.getDescription());
      writer.writeInt("publicationYear", book.getPublicationYear());
      writer.writeCollection("authors", book.getAuthors(), Author.class);
   }

   @Override
   public Book readFrom(ProtoStreamReader reader) throws IOException {
      String title = reader.readString("title");
      String description = reader.readString("description");
      int publicationYear = reader.readInt("publicationYear");
      Set<Author> authors = reader.readCollection("authors", new HashSet<>(), Author.class);
```

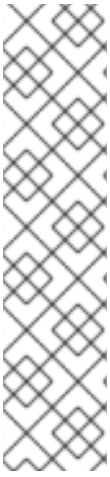
```
      return new Book(title, description, publicationYear, authors);
   }
}
```

以下の手順にしたがってクライアントを設定したら、Java オブジェクトをリモートキャッシュに読み書きし、キャッシュに保存されている実際のデータは、関係するすべてのタイプ（Book および Author ）に対してリモートクライアントに登録されていると protobuf エンコードされます。protobuf 形式で保存されたオブジェクトを維持すると、異なる言語で書かれた互換性のあるクライアントでオブジェクトを使用することができるという利点があります。

11.3.2. Protobuf でエンコードされたエントリーのインデックス化

前項の説明どおりにクライアントを設定したら、サーバー側でキャッシュのインデックス設定を開始できます。インデックス の有効化と各種インデックス固有の設定は埋め込みモードと同じで、Red Hat Data Grid のクエリー に記載されています。

ただし、追加の設定ステップも含まれます。埋め込みモードでは、エントリーのクラスにさまざまな Hibernate Search アノテーションが存在するかどうかを分析することで、インデックスメタデータが Java リフレクションによって取得されますが、エントリーが protobuf エンコードされている場合は明らかに実行できません。サーバーは、クライアントと同じ記述子（.proto ファイル）から関連するメタデータを取得する必要があります。記述子は '___protobuf_metadata' という名前のサーバーの専用のキャッシュに保存されます。このキャッシュのキーと値はどちらもプレーンテキストの文字列です。そのため、スキーマの名前をキーとして使用し、スキーマファイル自体を値として使用して、このキャッシュで put 操作を実行するのと同じくらい簡単です。または、（cache-container=*:register-proto-schemas（）操作経由）CLI、管理コンソール、または JMX 経由で ProtobufMetadataManager MBean を使用できます。セキュリティーが有効になっている場合、リモートプロトコル経由でスキーマキャッシュにアクセスするには、ユーザーが '___schema_manager' ロールに属している必要があります。



注記

インデックス化を有効にすると、インデックス化する必要のあるフィールドを正確に制御するために @Indexed および @Field protobuf スキーマ擬似アノテーションを使用しない限り、Protobuf でエンコードされたエントリーのすべてのフィールドに対してインデックスが有効になります。デフォルトの動作は、多くのフィールドを持つタイプを処理する場合に非常に効率的ではない可能性があるため、デフォルトのインデックス動作に依存する代わりに、インデックス化すべきフィールドを常に指定することが推奨されます。アノテーションのない protobuf メッセージタイプのインデックス動作も、スキーマファイルの最初に protobuf スキーマオプション 'indexed_by_default' を false に設定して変更できます（デフォルト値は true と見なされます）。

```
option indexed_by_default = false;  // This disables indexing of types that are not annotated
for indexing
```

11.3.3. リモートクエリーの例

protobuf と通信できるようにクライアントとサーバーの両方を設定し、インデックスを有効にしている。キャッシュにデータを追加して、検索してみてください。

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

RemoteCacheManager remoteCacheManager = ...;
RemoteCache<Integer, Book> remoteCache = remoteCacheManager.getCache();

Book book1 = new Book();
book1.setTitle("Hibernate in Action");
remoteCache.put(1, book1);

Book book2 = new Book();
book2.setTile("Hibernate Search in Action");
remoteCache.put(2, book2);

QueryFactory qf = Search.getQueryFactory(remoteCache);
Query query = qf.from(Book.class)
        .having("title").like("%Hibernate Search%")
        .build();

List<Book> list = query.list(); // Voila! We have our book back from the cache!
```

クエリーを作成する主な部分は、org.infinispan.client.hotrod.Search.get QueryFactory （）メソッドを使用してリモートキャッシュの QueryFactory を取得します。このクエリーの作成後は、このセクションで説明する埋め込みモードと似ています。

11.3.4. 分析

分析は、入力データを、インデックスを作成してクエリーできる 1 つ以上の用語に変換するプロセスです。

11.3.4.1. デフォルトのアナライザー

Red Hat Data Grid は、以下のようにデフォルトのアナライザーのセットを提供します。

定義	説明
standard	テキストフィールドをトークンに分割し、空白と句読点を区切り文字として扱います。
simple	非文字で区切り、すべての文字を小文字に変換することにより、入力ストリームをトークン化します。空白と非文字は破棄されます。
whitespace	テキストストリームを空白で分割し、空白以外の文字のシーケンスをトークンとして返します。
キーワード	テキストフィールド全体を単一トークンとして扱います。
stemmer	SnowballPorterフィルターを使用して英語の単語を語幹にします。
ngram	デフォルトでサイズ 3 つのグラムである n-gram トークンを生成します。
filename	テキストフィールドを **standard** アナライザーよりも大きなサイズトークンに分割し、空白文字を区切り文字として扱い、すべての文字を小文字に変換します。

これらのアナライザー定義は Apache Lucene をベースとし、「as-is」で提供されます。tokenizers、filters、および CharFilters に関する詳細は、適切な Lucene のドキュメントを参照してください。

11.3.4.2. アナライザー定義の使用

アナライザー定義を使用するには、.proto スキーマファイルで名前でこれらを参照します。

1. *Analyze.YES 属性を追加して、プロパティーが分析されていることを示します。*
2. *@Analyzer アノテーションで アナライザー定義を指定します。*

以下は、参照されたアナライザー定義の例になります。

```
/* @Indexed */
message TestEntity {

   /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer = @Analyzer(definition =
"keyword")) */
   optional string id = 1;

   /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer = @Analyzer(definition =
"simple")) */
   optional string name = 2;
}
```

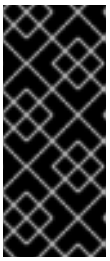
11.3.4.3. カスタムアナライザー定義の作成

カスタムアナライザー定義が必要な場合は、以下を行います。

1. JAR ファイルにパッケージ化された ProgrammaticSearchMappingProvider インターフェースの実装を作成します。

2. JAR の META-INF/services/ ディレクトリーに org.infinispan.query.spi.ProgrammaticSearchMappingProvider という名前のファイルを指定します。このファイルには、実装の完全修飾クラス名が含まれている必要があります。

3. JAR を Red Hat Data Grid インストールの standalone/deployments ディレクトリーにコピーします。



重要

起動時に、デプロイメントは Red Hat Data Grid サーバーで利用可能でなければなりません。サーバーがすでに実行中の場合は、デプロイメントを追加できません。

以下は、ProgrammaticSearchMappingProvider インターフェースの実装例です。

```
import org.apache.lucene.analysis.core.LowerCaseFilterFactory;
import org.apache.lucene.analysis.core.StopFilterFactory;
import org.apache.lucene.analysis.standard.StandardFilterFactory;
import org.apache.lucene.analysis.standard.StandardTokenizerFactory;
import org.hibernate.search.cfg.SearchMapping;
import org.infinispan.Cache;
import org.infinispan.query.spi.ProgrammaticSearchMappingProvider;
```

```
public final class MyAnalyzerProvider implements
ProgrammaticSearchMappingProvider {

  @Override
  public void defineMappings(Cache cache, SearchMapping searchMapping) {
    searchMapping
        .analyzerDef("standard-with-stop", StandardTokenizerFactory.class)
          .filter(StandardFilterFactory.class)
          .filter(LowerCaseFilterFactory.class)
          .filter(StopFilterFactory.class);
  }
}
```

4. JAR をキャッシュコンテナー設定で指定します。以下に例を示します。

```
<cache-container name="mycache" default-cache="default">
  <modules>
    <module name="deployment.analyzers.jar"/>
  </modules>
...
```

11.4. 統計

以下のコードスニペットにあるように、クエリー 統計 は SearchManager から取得できます。

```
SearchManager searchManager = Search.getSearchManager(cache);
org.hibernate.search.stat.Statistics statistics = searchManager.getStatistics();
```

ヒント

このデータは、org.infinispan:type=Query,manager="{name-of-cache-manager}",cache="{name-of-cache}",component=Statistics の名前で登録された Hibernate Search StatisticsInfoMBean を介して JMX 経由でも利用できます。この MBean は常に Red Hat Data Grid によって登録されますが、統計の収集がキャッシュレベルで 有効になっている場合のみ統計が収集されることに注意してください。

警告

Hibernate Search には、ここで説明するように、JMX 統計の独自の設定プロパティーhibernate.search.jmx_enabled および hibernate.search.generate_statistics があります。Red Hat Data Grid Query でこれらを使用すると、MBean の重複と予測不可能な結果につながるため禁止されます。

11.5. パフォーマンスチューニング

11.5.1. SYNC モードでのバッチ書き込み

デフォルトでは、インデックス マネージャー は同期モードで動作します。つまり、データが Red Hat Data Grid に書き込まれると、インデックス操作が同期的に実行されます。この同期性により、インデックスは常にデータと整合性が保たれます（したがって、検索で表示されます）が、インデックスへのコミットも実行されるため、書き込み操作が遅くなる可能性があります。Lucene ではコミットは非常にコストのかかる操作であるため、異なるノードからの複数の書き込みを自動的に1つのコミットにバッチ処理して、影響を軽減できます。

そのため、インデックスが有効な Red Hat Data Grid にデータを読み込む場合は、複数のスレッドを使用してこのバッチを活用してください。

複数のスレッドを使用しても必要なパフォーマンスが得られない場合は、インデックスを一時的に無効にしてデータをロードし、後で 再インデックス化 操作を実行することもできます。これは、SKIP_INDEXING フラグを使用してデータを書き込むことで実行できます。

```
cache.getAdvancedCache().withFlags(Flag.SKIP_INDEXING).put("key","value");
```

11.5.2. 非同期モードを使用した書き込み

データ書き込み間のわずかな遅延が許容可能であり、そのデータがクエリーに表示される場合は、インデックスマネージャーを非同期モードで動作するように設定できます。非同期モードでは、設定可能な間隔でコミットが行われるため、書き込みパフォーマンスが大幅に向上します。

設定:

```
<distributed-cache name="default">
```

```
    <indexing index="LOCAL">
      <property name="default.indexmanager">
          org.infinispan.query.indexmanager.InfinispanIndexManager
      </property>
      <!-- Index data in async mode -->
      <property name="default.worker.execution">async</property>
      <!-- Optional: configure the commit interval, default is 1000ms -->
      <property name="default.index_flush_interval">500</property>
    </indexing>
  </distributed-cache>
```

11.5.3. インデックスリーダーの非同期ストラテジー

Lucene は、内部的にインデックスのスナップショットと連携します。IndexReader が開かれると、開いた時点までのインデックスの変更のみが表示されます。IndexReader が更新されるまでさらにインデックスの変更は表示されません。デフォルトでは、Red Hat Data Grid で使用されるインデックスマネージャーは、クエリーごとにインデックスリーダーの最新の状態を確認し、必要に応じてそれらを更新します。

async として設定された reader.strategy 設定を使用すると、このストラテジーを調整して、この更新チェックを事前設定された間隔に緩和することができます。

```
<distributed-cache name="default"
                   key-partitioner="org.infinispan.distribution.ch.impl.AffinityPartitioner">
    <indexing index="PRIMARY_OWNER">
      <property name="default.indexmanager">
          org.infinispan.query.affinity.AffinityIndexManager
      </property>
      <property name="default.reader.strategy">async</property>
      <!-- refresh reader every 1s, default is 5s -->
      <property name="default.reader.async_refresh_period_ms">1000</property>
    </indexing>
  </distributed-cache>
```

async reader ストラテジーは、AffinityIndexManager などのシャードに依存するインデックスマネージャーに特に便利です。

11.5.4. Lucene オプション

Lucene でチューニングオプションを直接適用することが可能です。詳細は、Hibernate Search マニュアル を参照してください。

第12章 クラスター化カウンター

クラスター化 されたカウンターは、Red Hat Data Grid クラスターのすべてのノード間で分散され、共有されるカウンターです。カウンターは異なる整合性レベル（strong および weak）を持つことができます。

strong/weakと一貫性のあるカウンターには個別のインターフェイスがありますが、どちらもその値の更新をサポートし、現在の値を返し、その値が更新されたときにイベントを提供します。このドキュメントでは、ユースケースに最適なものを選択する上で役立つ詳細を以下に示します。

12.1. インストールおよび設定

カウンターの使用を開始するには、Maven の pom.xml ファイルに依存関係を追加する必要があります。

pom.xml

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

${version.infinispan} を Red Hat Data Grid の適切なバージョンに置き換えます。

カウンターは、本書で後述する CounterManager インターフェースを介して Red Hat Data Grid 設定ファイルまたはオンデマンドを設定できます。EmbeddedCacheManager の起動時に、Red Hat Data Grid 設定ファイルに設定したカウンターが起動時に作成されます。これらのカウンターは Eagerly で開始され、すべてのクラスターのノードで利用できます。

configuration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan>
   <cache-container ...>
```

```
        <!-- if needed to persist counter, global state needs to be configured -->
        <global-state>
           ...
        </global-state>
        <!-- your caches configuration goes here -->
         <counters xmlns="urn:infinispan:config:counters:9.2" num-owners="3"
reliability="CONSISTENT">
            <strong-counter name="c1" initial-value="1" storage="PERSISTENT"/>
            <strong-counter name="c2" initial-value="2" storage="VOLATILE">
               <lower-bound value="0"/>
            </strong-counter>
            <strong-counter name="c3" initial-value="3" storage="PERSISTENT">
               <upper-bound value="5"/>
            </strong-counter>
            <strong-counter name="c4" initial-value="4" storage="VOLATILE">
               <lower-bound value="0"/>
               <upper-bound value="10"/>
            </strong-counter>
            <weak-counter name="c5" initial-value="5" storage="PERSISTENT" concurrency-level="1"/>
         </counters>
      </cache-container>
</infinispan>
```

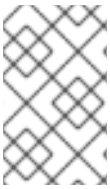
または、プログラムを使用して GlobalConfigurationBuilder で以下を行います。

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...;
CounterManagerConfigurationBuilder builder =
globalConfigurationBuilder.addModule(CounterManagerConfigurationBuilder.class);
builder.numOwner(3).reliability(Reliability.CONSISTENT);
builder.addStrongCounter().name("c1").initialValue(1).storage(Storage.PERSISTENT);
builder.addStrongCounter().name("c2").initialValue(2).lowerBound(0).storage(Storage.VOLATI
LE);
builder.addStrongCounter().name("c3").initialValue(3).upperBound(5).storage(Storage.PERSIS
TENT);
builder.addStrongCounter().name("c4").initialValue(4).lowerBound(0).upperBound(10).storage
(Storage.VOLATILE);
builder.addWeakCounter().name("c5").initialValue(5).concurrencyLevel(1).storage(Storage.PE
RSISTENT);
```

一方、このカウンターは、EmbeddedCacheManager を初期化した後にいつでも設定することができます。

```
CounterManager manager = ...;
manager.defineCounter("c1",
CounterConfiguration.builder(CounterType.UNBOUNDED_STRONG).initialValue(1).storage(St
orage.PERSISTENT)build());
manager.defineCounter("c2",
```

```
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(2).lowerBound(0
).storage(Storage.VOLATILE).build());
manager.defineCounter("c3",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(3).upperBound(5
).storage(Storage.PERSISTENT).build());
manager.defineCounter("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(4).lowerBound(0
).upperBound(10).storage(Storage.VOLATILE).build());
manager.defineCounter("c2",
CounterConfiguration.builder(CounterType.WEAK).initialValue(5).concurrencyLevel(1).storage
(Storage.PERSISTENT).build());
```



注記

CounterConfiguration は変更できず、再利用できます。

カウンターが正常に設定されていると、defineCounter() メソッドは true を返します。そうでない場合は、true を返します。ただし、設定が無効な場合は、メソッドによって CounterConfigurationException が発生します。カウンターがすでに定義されているかを調べるには、isDefined() メソッドを使用します。

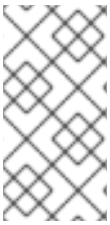
```
CounterManager manager = ...
if (!manager.isDefined("someCounter")) {
   manager.define("someCounter", ...);
}
```

クラスターごとの属性:

- num-owners: クラスター全体で保持するカウンターのコピー数を設定します。数値が小さいほど更新操作は高速になりますが、サポートされるサーバークラッシュの数は少なくなります。正の値である必要があり、デフォルト値は 2 です。
- reliability: ネットワークパーティションでカウンターの更新動作を設定します。デフォルト値は AVAILABLE で、有効な値は次のとおりです。
  - AVAILABLE: すべてのパーティションはカウンター値の読み取りと更新が可能です。
  - CONSISTENT: プライマリーパーティション（ノードの大多数）のみがカウンター値の読み取りと更新が可能です。残りのパーティションは、その値の読み取りのみ可能です。

カウンターごとの属性:

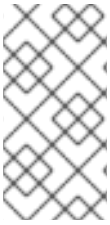
- *initial-value [common]: カウンターの初期値を設定します。デフォルトは 0 (ゼロ) です。*
- *storage [common]: クラスターのシャットダウンおよび再起動時のカウンターの動作を設定します。デフォルト値は VOLATILE で、有効な値は次のとおりです。*
  - *VOLATILE: カウンターの値はメモリーでのみ利用できます。この値は、クラスターがシャットダウンすると失われます。*
  - *PERSISTENT: カウンターの値はプライベートおよびローカル永続ストアに保存されます。この値は、クラスターがシャットダウンされたときに保持され、再起動後に復元されます。*



注記

オンデマンドおよび VOLATILE カウンターは、クラスターのシャットダウン後にその値と設定を失います。再起動後に再度定義する必要があります。

- *lower-bound [strong]: 強力な一貫性のあるカウンターの下限を設定します。デフォルト値は Long.MIN_VALUE です。*
- *upper-bound [strong]: 強力な一貫性のあるカウンターの上限を設定します。デフォルト値は Long.MAX_VALUE です。*



注記

lower-bound も upper-bound も設定されていない場合は、強力なカウンターは無制限として設定されます。

警告

initial-value は、lower-bound 以上 upper-bound 以下である必要があります。

- concurrency-level [weak]: 同時更新の数を設定します。正の値である必要があり、デフォルト値は 16 です。

12.1.1. カウンター名の一覧表示

定義されたすべてのカウンターを一覧表示するには、CounterManager.getCounterNames() メソッドは、クラスター全体で作成されたすべてのカウンター名のコレクションを返します。

12.2. COUNTERMANAGER インターフェース。

CounterManager インターフェースは、カウンターを定義、取得、および削除するエントリーポイントです。EmbeddedCacheManager の作成を自動的にリッスンし、EmbeddedCacheManager ごとのインスタンスの登録を続行します。カウンター状態を保存し、デフォルトのカウンターの設定に必要なキャッシュを開始します。

CounterManager の取得は、以下の例のように EmbeddedCounterManagerFactory.asCounterManager(EmbeddedCacheManager) を呼び出すだけです。

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

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

Hotgitops クライアントでは、CounterManager が RemoteCacheManager に登録され、以下のように取得できます。

```
// create or obtain your RemoteCacheManager
RemoteCacheManager manager = ...;
```

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

注記

hotgitops messages format can be found in Hotgitops Protocol 2.7（ Hotgitops Protocol 2.7 でホットでメッセージ形式が見つかります）

12.2.1. CounterManager を介したカウンターの削除

警告

注意して使用してください。

Strong/WeakCounterと CounterManager でカウンターを削除するのに違いがあります。CounterManager.remove(String) は、クラスターからカウンター値を削除し、ローカルカウンターインスタンスのカウンターに登録されているすべてのリスナーを削除します。さらに、カウンターインスタンスは再利用可能ではなくなり、無効な結果が返される可能性があります。

一方で、Strong/WeakCounter を削除するとカウンター値のみが削除されます。インスタンスは引き続き再利用でき、リスナーは引き続き動作します。

注記

削除後にアクセスされると、カウンターは再作成されます。

12.3. カウンター

カウンターは、strong (StrongCounter)またはweak(WeakCounter)になり、いずれも名前で識別されます。各インターフェースには特定のインターフェースがありますが、ロジック（つまり各操作により CompletableFuture が返される）を共有しているため、更新イベントが返され、初期値にリセットできます。

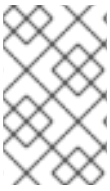
非同期 API を使用しない場合は、sync() メソッドを介して同期カウンターを返すことができます。

API は同じですが、CompletableFuture の戻り値はありません。

以下のメソッドは、両方のインターフェースに共通しています。

```
String getName();
CompletableFuture<Long> getValue();
CompletableFuture<Void> reset();
<T extends CounterListener> Handle<T> addListener(T listener);
CounterConfiguration getConfiguration();
CompletableFuture<Void> remove();
SyncStrongCounter sync(); //SyncWeakCounter for WeakCounter
```

- getName() はカウンター名(identifier)を返します。
- getValue() は現在のカウンターの値を返します。
- reset() により、カウンターの値を初期値にリセットできます。
- reset() はリスナーを登録し、更新イベントを受信します。詳細については、「通知およびイベント」セクションをご覧ください。
- getConfiguration() はカウンターによって使用される設定を返します。
- remove() はクラスターからカウンター値を削除します。インスタンスは引き続き使用でき、リスナーが保持されます。
- sync() は同期カウンターを作成します。



注記

削除後にアクセスされると、カウンターは再作成されます。

12.3.1. StrongCounter インターフェース: 一貫性または境界が明確になります。

強力なカウンターでは、Red Hat Data Grid キャッシュに保存されている単一のキーを使用して、必

要な整合性を提供します。すべての更新は、その値を更新するためにキーロックの下で実行されます。一方、読み取りはロックを取得し、現在の値を読み取ります。さらに、このスキームではカウンター値をバインドでき、比較および設定/スワップなどのアトミック操作を提供できます。

StrongCounter は、getStrongCounter() メソッドを使用して CounterManager から取得することができます。たとえば、以下のようになります。

```
CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getStrongCounter("my-counter);
```

警告

すべての操作は単一のキーに到達するため、StrongCounter は競合レートが高くなります。

StrongCounter インターフェースでは、以下のメソッドを追加します。

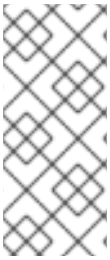
```
default CompletableFuture<Long> incrementAndGet() {
  return addAndGet(1L);
}

default CompletableFuture<Long> decrementAndGet() {
  return addAndGet(-1L);
}

CompletableFuture<Long> addAndGet(long delta);

CompletableFuture<Boolean> compareAndSet(long expect, long update);

CompletableFuture<Long> compareAndSwap(long expect, long update);
```

- incrementAndGet() はカウンターを 1 つずつ増分し、新しい値を返します。
- decrementAndGet() は、1 つずつカウンターをデクリメントし、新しい値を返します。
- addAndGet() は、delta をカウンターの値に追加し、新しい値を返します。

- compareAndSet() および compareAndSwap() は、現在の値が想定される場合にカウンターの値を設定します。

注記

CompletableFuture が完了すると、操作は完了とみなされます。



注記

compare-and-set と compare-and-swap の相違点は、操作に成功した場合に、compare-and-set は true を返しますが、compare-and-swap は前の値をか返すことです。戻り値が期待値と同じ場合は、compare-and-swap が正常になります。

12.3.1.1. バインドされた StrongCounter

バインドされている場合、上記の更新メソッドはすべて、下限または上限に達すると CounterOutOfBoundsException をスローします。例外には、どちら側にバインドが到達したかを確認するための次のメソッドがあります。

```
public boolean isUpperBoundReached();
public boolean isLowerBoundReached();
```

12.3.1.2. ユースケース

強力なカウンターは、次の使用例に適しています。

- 各更新後にカウンターの値が必要な場合（例: クラスター単位のIDジェネレーターまたはシーケンス）

- バインドされたカウンターが必要な場合は（例: レートリミッター）

12.3.1.3. 使用例

```
StrongCounter counter = counterManager.getStrongCounter("unbounded_coutner");

// incrementing the counter
System.out.println("new value is " + counter.incrementAndGet().get());

// decrement the counter's value by 100 using the functional API
```

```
counter.addAndGet(-100).thenApply(v -> {
  System.out.println("new value is " + v);
  return null;
}).get

// alternative, you can do some work while the counter is updated
CompletableFuture<Long> f = counter.addAndGet(10);
// ... do some work ...
System.out.println("new value is " + f.get());

// and then, check the current value
System.out.println("current value is " + counter.getValue().get());

// finally, reset to initial value
counter.reset().get();
System.out.println("current value is " + counter.getValue().get());

// or set to a new value if zero
System.out.println("compare and set succeeded? " + counter.compareAndSet(0, 1));
```

以下に、バインドされたカウンターを使用する別の例を示します。

```
StrongCounter counter = counterManager.getStrongCounter("bounded_counter");

// incrementing the counter
try {
   System.out.println("new value is " + counter.addAndGet(100).get());
} catch (ExecutionException e) {
   Throwable cause = e.getCause();
   if (cause instanceof CounterOutOfBoundsException) {
      if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
         System.out.println("ops, upper bound reached.");
      } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
         System.out.println("ops, lower bound reached.");
      }
   }
}

// now using the functional API
counter.addAndGet(-100).handle((v, throwable) -> {
  if (throwable != null) {
     Throwable cause = throwable.getCause();
     if (cause instanceof CounterOutOfBoundsException) {
        if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
           System.out.println("ops, upper bound reached.");
        } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
           System.out.println("ops, lower bound reached.");
        }
     }
     return null;
  }
  System.out.println("new value is " + v);
  return null;
}).get();
```

–

Compare-and-set と Compare-and-swap の比較例:

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue, newValue;
do {
  oldValue = counter.getValue().get();
  newValue = someLogic(oldValue);
} while (!counter.compareAndSet(oldValue, newValue).get());
```

compare-and-swap では、呼び出しカウンターの呼び出し (counter.getValue()) が 1 つ保存されます。

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue = counter.getValue().get();
long currentValue, newValue;
do {
  currentValue = oldValue;
  newValue = someLogic(oldValue);
} while ((oldValue = counter.compareAndSwap(oldValue, newValue).get()) != currentValue);
```

12.3.2. WeakCounter インターフェース: 速度が必要な場合

WeakCounter は、カウンターの値を Red Hat Data Grid キャッシュの複数のキーに保存します。作成されたキーの数は concurrency-level 属性によって設定されます。各キーはカウンターの値の一部の状態を保存し、同時に更新できます。StrongCounter よりも優れた点は、キャッシュの競合率が低いことです。一方、値の読み取りはよりコストが高く、バインドは許可されません。

警告

リセット操作は注意して行う必要があります。これは アトミックではなく、中間値を生成します。これらの値は、読み取り操作および登録されたリスナーによって確認できます。

WeakCounter は、getWeakCounter() メソッドを使用して CounterManager から取得できます。たとえば、以下のようになります。

```
CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getWeakCounter("my-counter);
```

12.3.2.1. weak カウンターインターフェース

WeakCounter は、以下のメソッドを追加します。

```
default CompletableFuture<Void> increment() {
  return add(1L);
}

default CompletableFuture<Void> decrement() {
  return add(-1L);
}

CompletableFuture<Void> add(long delta);
```

これらは `StrongCounter のメソッドと似ていますが、新しい値は返されません。

12.3.2.2. ユースケース

weak カウンターは、更新操作の結果が必要ない場合やカウンターの値があまり必要でないユースケースに最適です。統計の収集は、このようなユースケースの良い例になります。

12.3.2.3. 例

以下では、弱いカウンターの使用例を示します。

```
WeakCounter counter = counterManager.getWeakCounter("my_counter");

// increment the counter and check its result
counter.increment().get();
System.out.println("current value is " + counter.getValue().get());

CompletableFuture<Void> f = counter.add(-100);
//do some work
f.get(); //wait until finished
System.out.println("current value is " + counter.getValue().get());

//using the functional API
counter.reset().whenComplete((aVoid, throwable) -> System.out.println("Reset done " +
(throwable == null ? "successfully" : "unsuccessfully"))).get();
System.out.println("current value is " + counter.getValue().get());
```

12.4. 通知およびイベント

strong カウンターと weakカウンターの両方が、更新イベントを受信するためにリスナーをサポートします。リスナーは CounterListener を実装する必要があり、以下の方法で登録できます。

```
<T extends CounterListener> Handle<T> addListener(T listener);
```

CounterLister には以下のインターフェースがあります。

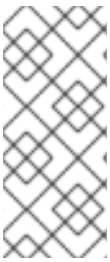
```
public interface CounterListener {
  void onUpdate(CounterEvent entry);
}
```

返される Handle オブジェクトには、CounterListener が必要なくなったときに削除するという主な目的があります。また、処理している CounterListener インスタンスにアクセスできます。これには、以下のインターフェースがあります。

```
public interface Handle<T extends CounterListener> {
  T getCounterListener();
  void remove();
}
```

最後に、CounterEvent には、以前と現在の値と状態があります。これには、以下のインターフェースがあります。

```
public interface CounterEvent {
  long getOldValue();
  State getOldState();
  long getNewValue();
  State getNewState();
}
```



注記

状態は、非有界である strong カウンターおよび weak カウンターでは常に State.VALID になります。State.LOWER_BOUND_REACHED および State.UPPER_BOUND_REACHED は有界である strong カウンターのみに有効です。

警告

weak カウンター reset() 操作は、中間値で複数の通知をトリガーします。

第13章 クラスター化されたロック

クラスター化されたロック は、Red Hat Data Grid クラスターの全ノードに分散および共有されるロックであり、現在、指定のクラスターのノード間で同期されるコードを実行する方法を提供します。

13.1. インストールシステム

クラスター化されたロックの使用を開始するには、Maven の pom.xml ファイルに依存関係を追加する必要があります。

pom.xml

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

${version.infinispan} を Red Hat Data Grid の適切なバージョンに置き換えます。

13.2. CLUSTEREDLOCK の設定

現時点で、ClusteredLock の 1 つのタイプがサポートされています。これは reentrant で、NODE の所有権ロックに対応しています。

13.2.1. 所有権

- NODE: ClusteredLock が定義されている場合、このロックは Red Hat Data Grid クラスターのすべてのノードから使用することができます。所有者が NODE タイプである場合、ロックの所有者は、指定したタイミングでロックを取得した Red Hat Data Grid ノードになります。つまり、ClusteredCacheManager で ClusteredLock インスタンスを取得するたびに、このインスタンスは Red Hat Data Grid ノードごとに同じインスタンスになります。このロックを使用して、Red Hat Data Grid ノード間のコードの同期に使用できます。このロックの利点は、ノードのスレッドが特定の時間にロックを解放できることです。

- INSTANCE - not supported

ClusteredLock が定義されると、Red Hat Data Grid クラスターのすべてのノードからこのロックを使用できます。所有者が INSTANCE タイプである場合、ClusteredLockManager.get("lockName") が呼び出されると、ロックの所有者が取得した実際のインスタンスになります。

つまり、ClusteredCacheManager で ClusteredLock インスタンスを取得するたびに、このインスタンスが新しいインスタンスになります。このロックを使用して、Red Hat Data Grid ノードと各 Red Hat Data Grid ノード内でコードを同期できます。このロックの利点は、「lock」と呼ばれるインスタンスのみがロックを解放できることです。

13.2.2. 再入可能性

ClusteredLock が reentrant に設定されている場合、ロックの所有者はロックを保持しつつ必要な回数だけロックを受け取ることができます。

現在、非依存ロックのみがサポートされています。つまり、連続して同じ所有者に対して 2 回連続した ロック 呼び出しが送信されると、最初の呼び出しは利用可能な場合にロックを取得し、2 回目の呼び出しはブロックされます。

13.3. CLUSTEREDLOCKMANAGER インターフェース

実験的な としてマーク されている ClusteredLockManager インターフェースは、ロックを定義、取得、および削除するエントリーポイントです。EmbeddedCacheManager の作成を自動的にリッスンし、EmbeddedCacheManager ごとのインスタンスの登録を続行します。ロックの状態を保存するのに必要な内部キャッシュを開始します。

ClusteredLockManager の取得は、以下の例のように EmbeddedClusteredLockManagerFactory.from(EmbeddedCacheManager) を呼び出すだけで簡単です。

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

// retrieve the ClusteredLockManager
ClusteredLockManager clusteredLockManager =
EmbeddedClusteredLockManagerFactory.from(manager);
```

```
@Experimental
public interface ClusteredLockManager {
```

```
   boolean defineLock(String name);

   boolean defineLock(String name, ClusteredLockConfiguration configuration);

   ClusteredLock get(String name);

   ClusteredLockConfiguration getConfiguration(String name);

   boolean isDefined(String name);

   CompletableFuture<Boolean> remove(String name);

   CompletableFuture<Boolean> forceRelease(String name);
}
```

- defineLock : 指定の名前とデフォルトの ClusteredLockConfiguration でロックを定義します。既存の設定を上書きしません。

- defineLock(String name, ClusteredLockConfiguration configuration) : 指定された名前と ClusteredLockConfiguration でロックを定義します。既存の設定を上書きしません。

- ClusteredLock get(String name) : 名前で ClusteredLock を取得します。defineLock の呼び出しは、クラスター内で少なくとも 1 回実行する必要があります。get メソッド呼び出しの影響を理解するには、「所有 者レベルのセクション」を参照してください。

現在、サポートされている唯一の所有権レベルは NODE です。

- ClusteredLockConfiguration getConfiguration(String name) :

ClusteredLock の設定を返します（ある場合）。

- boolean isDefined(String name) : ロックがすでに定義されているかどうかを確認します。

- CompletableFuture<Boolean> remove(String name) : ClusteredLock が存在する場合は削除します。

- CompletableFuture<Boolean> forceRelease(String name) : Releases - または unlocks - ClusteredLock - if such exists, if such exists, whether they are keep it at a certain time.こ

のメソッドを呼び出すと、同時実行の問題を引き起こす可能性があり、例外的な状況 で使用する必要があります。

13.4. CLUSTEREDLOCK インターフェース

実験的な としてマーク されている ClusteredLock インターフェースは、クラスター化されたロックを実装するインターフェースです。

```
@Experimental
public interface ClusteredLock {

  CompletableFuture<Void> lock();

  CompletableFuture<Boolean> tryLock();

  CompletableFuture<Boolean> tryLock(long time, TimeUnit unit);

  CompletableFuture<Void> unlock();

  CompletableFuture<Boolean> isLocked();

  CompletableFuture<Boolean> isLockedByMe();
}
```

- lock : ロックを取得します。ロックが利用できない場合は、ロックを取得するまでブロックを呼び出します。現在、ロックリクエストが失敗するまでに最大時間が指定されていない ため、スレッドが不足する可能性があります。

- tryLock は、呼び出し時に空きがある場合にのみロックを取得し、その場合は true を返します。この方法では、ロックの取り組みをブロック（または待機）しません。

- tryLock(long time, TimeUnit unit) ロックが利用可能な場合は、このメソッドが true ですぐに返されます。ロックが使用できない場合、呼び出しは以下のように待機します。

  - ロックが取得されます。

  - 指定された待機時間の経過

時間がゼロ以下である場合、メソッドは全く待機しません。

- *unlock*

ロックを解放します。ロックのホルダーのみがロックを解放できます。

- ロックがロックされ、ロックがリリースされると false の場合は isLocked Returns true になります。

- isLockedByMe Returns true: ロックが呼び出し元によって所有され、ロックが他の人またはリリースされると false になります。

13.4.1. 使用例

```
EmbeddedCache cm = ...;
ClusteredLockManager cclm = EmbeddedClusteredLockManagerFactory.from(cm);

lock.tryLock()
  .thenCompose(result -> {
     if (result) {
      try {
          // manipulate protected state
         } finally {
           return lock.unlock();
         }
     } else {
       // Do something else
     }
  });
}
```

13.5. CLUSTEREDLOCKMANAGER の設定

以下の属性を使用して、宣言的またはプログラムで異なるストラテジーをロックに使用するように ClusteredLockManager を設定できます。

num-owners

クラスター化されたロックの状態を保存する各クラスターのノードの合計数を定義します。デフォルト値は -1 で、値をすべてのノードに複製します。

信頼性

クラスターをパーティションに分割したり、複数のノードに分割すると、クラスター化ロックの動作を制御します。以下の値を設定できます。

- *AVAILABLE: 任意のパーティションのノードが、ロックで同時に操作することができます。*

- *CONSISTENT: 大多数のパーティションに属するノードのみが、ロック上で動作できます。これはデフォルト値です。*

以下は、ClusteredLockManager の宣言的設定の例です。

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:${infinispan.core.schema.version}
http://www.infinispan.org/schemas/infinispan-config-9.4.xsd"
      xmlns="urn:infinispan:config:9.4">
   ...
   <cache-container default-cache="default">
      <transport/>
      <local-cache name="default">
         <locking concurrency-level="100" acquire-timeout="1000"/>
      </local-cache>

      <clustered-locks xmlns="urn:infinispan:config:clustered-locks:9.4"
                       num-owners = "3"
                       reliability="AVAILABLE">
         <clustered-lock name="lock1" />
         <clustered-lock name="lock2" />
      </clustered-locks>
   </cache-container>
   ...
</infinispan>
```

第14章 マルチマップキャッシュ

MutimapCache は、キーを値にマッピングし、各キーに複数の値を含めることができます。

14.1. インストールと設定

pom.xml

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

${version.infinispan} を Red Hat Data Grid の適切なバージョンに置き換えます。

14.2. MULTIMAPCACHE API

MultimapCache API は、Multimap キャッシュと対話する複数のメソッドを公開します。これらのメソッドはすべて、ほとんどのケースでブロックされません。[制限] を参照してください。

```
public interface MultimapCache<K, V> {

   CompletableFuture<Void> put(K key, V value);

   CompletableFuture<Collection<V>> get(K key);

   CompletableFuture<Boolean> remove(K key);

   CompletableFuture<Boolean> remove(K key, V value);

   CompletableFuture<Void> remove(Predicate<? super V> p);

   CompletableFuture<Boolean> containsKey(K key);

   CompletableFuture<Boolean> containsValue(V value);

   CompletableFuture<Boolean> containsEntry(K key, V value);

   CompletableFuture<Long> size();
```

```
  boolean supportsDuplicates();

}
```

14.2.1. CompletableFuture<Void> put(K key, V value)

キーと値のペアをマルチマップキャッシュに配置します。

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

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

                 if(names.contains("oihana"))
                   System.out.println("Oihana is a girl name");
               });
```

このコードの出力は、以下のようになります。

```
Marie is a girl name
Oihana is a girl name
```

14.2.2. CompletableFuture<Collection<V>> get(K key)

存在する場合、このマルチマップキャッシュ内のキーに関連付けられた値のビューコレクションを返す非同期。取得したコレクションへの変更は、このマルチマップキャッシュの値を変更しません。このメソッドは空のコレクションを返すと、キーが見つからないことを意味します。

14.2.3. CompletableFuture<Boolean> remove(K key)

キーに関連付けられたエントリがマルチマップキャッシュに存在する場合は、それを削除する非同期。

14.2.4. CompletableFuture<Boolean> remove(K key, V value)

キーと値のペアが存在する場合は、マルチマップキャッシュから削除する非同期。

14.2.5. CompletableFuture<Void> remove(Predicate<? super V> p)

非同期メソッド。指定の述語に一致するすべての値を削除します。

14.2.6. CompletableFuture<Boolean> containsKey(K key)

このマルチマップにキーが含まれる場合に true を返す非同期。

14.2.7. CompletableFuture<Boolean> containsValue(V value)

このマルチマップに少なくとも1つのキーの値が含まれている場合にtrueを返す非同期。

14.2.8. CompletableFuture<Boolean> containsEntry(K key, V value)

このマルチマップに値を持つキーと値のペアが 1 つ以上含まれている場合、true を返す非同期。

14.2.9. CompletableFuture<Long> size()

マルチマップキャッシュ内のキーと値のペアの数を返す非同期。明確な数のキーは返されません。

14.2.10. boolean supportsDuplicates()

マルチマップキャッシュが重複をサポートする場合は true を返す非同期。これは、マルチマップのコンテンツが'a' → ['1', '1', '2']になる可能性があることを意味します。重複はまだサポートされていないため、今のところ、このメソッドは常にfalseを返します。指定された値の存在は、'equals' および `hashcode' method'のコントラクトによって決定されます。

14.3. マルチマップキャッシュの作成

現在、MultimapCache は通常のキャッシュとして設定されます。これは、コードまたは XML 設定のいずれかで実行できます。[configure a cache]へのセクションリンクで、通常のキャッシュを構成する方法を参照してください。

14.3.1. 組み込みモード

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

// create or obtain a MultimapCacheManager passing the EmbeddedCacheManager
MultimapCacheManager multimapCacheManager =
```

```
EmbeddedMultimapCacheManagerFactory.from(cm);

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

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

14.4. 制限事項

ほとんどの場合、Multimap キャッシュは通常のキャッシュとして機能しますが、現行バージョンにはいくつかの制限があります。

14.4.1. 重複のサポート

重複はまだサポートされていません。これは、マルチマップに重複したキーと値のペアが含まれていないことを意味します。put メソッドが呼び出されるたびに、キーと値のペアがすでに存在する場合、このキーと値のペアは追加されません。Multimap にキーと値のペアがすでに存在しているかどうかを確認するために使用されるメソッドは equals および hashcode です。

14.4.2. エビクション

現時点では、エビクションはキーと値のペアごとではなく、キーごとに機能します。これは、キーがエビクトされるたびに、キーに関連付けられているすべての値も削除されることを意味します。キー値ごとのエビクションは今後サポートされる可能性があります。

14.4.3. トランザクション

暗黙的なトランザクションは、自動コミットによってサポートされ、すべてのメソッドは非ブロッキングです。ほとんどの場合、明示的なトランザクションはブロックせずに機能します。ブロックするメソッドは size、containsEntry、および remove(Predicate<? super V> p) です。

第15章 CDI サポート

Red Hat Data Grid には、Red Hat Data Grid の infinispan-cdi-embedded または infinispan-cdi-remote モジュールを使用した Contexts and Dependency Injection（CDI と呼ばれる記事） との統合が含まれています。CDI は Java EE 仕様 の一部で、コンテナーで Bean のライフサイクルを管理することを目的としています。統合により、Cache インターフェースおよびブリッジ Cache および CacheManager イベントをインジェクトできます。jcache アノテーション(JSR-107)は infinispan-jcache および infinispan-jcache-remote アーティファクトでサポートされます。詳細は、JCACHE 仕様の 第 11 章 を参照してください。

15.1. MAVEN 依存関係

プロジェクトに Red Hat Data Grid の CDI サポートを含めるには、以下のいずれかの依存関係を使用します。

pom.xml for Embedded mode

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

${version.infinispan} を Red Hat Data Grid の適切なバージョンに置き換えます。

リモートモードの pom.xml

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

${version.infinispan} を Red Hat Data Grid の適切なバージョンに置き換えます。

使用する RED HAT DATA GRID のバージョン

Red Hat Data Grid の最終バージョンを使用することが推奨されます。

15.2. 埋め込みキャッシュの統合

15.2.1. 埋め込みキャッシュの挿入

デフォルトでは、デフォルトの Red Hat Data Grid キャッシュを注入できます。以下の例を見てみましょう。

デフォルトのキャッシュ挿入

```
...
import javax.inject.Inject;

public class GreetingService {

   @Inject
   private Cache<String, String> cache;

   public String greet(String user) {
      String cachedValue = cache.get(user);
      if (cachedValue == null) {
         cachedValue = "Hello " + user;
         cache.put(user, cachedValue);
      }
      return cachedValue;
   }
}
```

デフォルトではなく特定のキャッシュを使用する場合は、独自のキャッシュ設定およびキャッシュ修飾子を指定する必要があります。以下の例を参照してください。

修飾子の例

```
...
import javax.inject.Qualifier;

@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GreetingCache {
}
```

修飾子を使用したキャッシュのインジェクト

```
...
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.cdi.ConfigureCache;
import javax.enterprise.inject.Produces;

public class Config {

   @ConfigureCache("greeting-cache") // This is the cache name.
   @GreetingCache // This is the cache qualifier.
   @Produces
   public Configuration greetingCacheConfiguration() {
      return new ConfigurationBuilder()
               .memory()
                  .size(1000)
               .build();
   }

   // The same example without providing a custom configuration.
   // In this case the default cache configuration will be used.
   @ConfigureCache("greeting-cache")
   @GreetingCache
   @Produces
   public Configuration greetingCacheConfiguration;
}
```

GreetingService でこのキャッシュを使用するには、キャッシュインジェクションポイントに @GeetingCache 修飾子を追加します。

15.2.2. デフォルトの埋め込みキャッシュマネージャーおよび設定を上書きします。

デフォルトの EmbeddedCacheManager で使用されるデフォルトのキャッシュ設定を上書きできます。そのため、以下のスニペットに示されるように、デフォルトの修飾子で Configuration プロデューサーを作成する必要があります。

設定の上書き

```
public class Config {

   // By default CDI adds the @Default qualifier if no other qualifier is provided.
   @Produces
   public Configuration defaultEmbeddedCacheConfiguration() {
      return new ConfigurationBuilder()
                .memory()
                   .size(100)
                .build();
   }
}
```

デフォルトの EmbeddedCacheManager を上書きすることもできます。新規作成されたマネージャーには、デフォルトの修飾子とアプリケーションスコープが必要です。

EmbeddedCacheManager の上書き

```
...
import javax.enterprise.context.ApplicationScoped;

public class Config {

   @Produces
   @ApplicationScoped
   public EmbeddedCacheManager defaultEmbeddedCacheManager() {
     return new DefaultCacheManager(new ConfigurationBuilder()
                                    .memory()
                                       .size(100)
                                    .build());
   }
}
```

15.2.3. クラスター化で使用するためにトランスポートを設定する

クラスターモードで Red Hat Data Grid を使用するには、GlobalConfiguration でトランスポートを設定する必要があります。前のセクションで説明したように、デフォルトのキャッシュマネージャーを上書きします。以下のスニペットを参照してください。

デフォルトの EmbeddedCacheManager のオーバーライド

```
...
package org.infinispan.configuration.global.GlobalConfigurationBuilder;

@Produces
@ApplicationScoped
public EmbeddedCacheManager defaultClusteredCacheManager() {
   return new DefaultCacheManager(
      new GlobalConfigurationBuilder().transport().defaultTransport().build(),
      new ConfigurationBuilder().memory().size(7).build()
   );
}
```

15.3. リモートキャッシュの統合

15.3.1. リモートキャッシュの挿入

CDI 統合では、以下のスニペットで説明されているように RemoteCache を使用することもできます。

RemoteCache のインジェクト

```
public class GreetingService {

   @Inject
   private RemoteCache<String, String> cache;

   public String greet(String user) {
      String cachedValue = cache.get(user);
      if (cachedValue == null) {
         cachedValue = "Hello " + user;
         cache.put(user, cachedValue);
      }
```

```
        return cachedValue;
    }
}
```

greeting-cache などの別のキャッシュを使用する場合は、キャッシュ名が含まれるキャッシュインジェクションポイントに @Remote 修飾子を追加します。

修飾子を使用した RemoteCache のインジェクト

```
public class GreetingService {

    @Inject
    @Remote("greeting-cache")
    private RemoteCache<String, String> cache;

    ...
}
```

各インジェクションポイントに @Remote キャッシュ修飾子を追加すると、エラーが発生することがあります。このため、リモートキャッシュ統合は、同じ目的を達成するための別の方法を提供します。これには、@Remote アノテーションが付けられた独自の修飾子を作成する必要があります。

RemoteCache 修飾子

```
@Remote("greeting-cache")
@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RemoteGreetingCache {
}
```

GreetingService でこのキャッシュを使用するには、キャッシュインジェクションに修飾子

@RemoteGreetingCache 修飾子を追加します。

15.3.2. デフォルトのリモートキャッシュマネージャーの上書き

埋め込みキャッシュインテグレーションと同様に、リモートキャッシュ統合にはデフォルトのリモートキャッシュマネージャープロデューサーが含まれます。以下のスニペットにあるように、このデフォルトの RemoteCacheManager は上書きできます。

デフォルトの RemoteCacheManager のオーバーライド

```
public class Config {

   @Produces
   @ApplicationScoped
   public RemoteCacheManager defaultRemoteCacheManager() {
      return new RemoteCacheManager(localhost, 1544);
   }
}
```

15.4. 1 つ以上のキャッシュにカスタムのリモート/組み込みキャッシュマネージャーを使用

1 つ以上のキャッシュにカスタムキャッシュマネージャーを使用できます。キャッシュマネージャープロデューサーにキャッシュ修飾子でアノテーションを付ける必要があります。以下の例を参照してください。

```
public class Config {

  @GreetingCache
  @Produces
  @ApplicationScoped
  public EmbeddedCacheManager specificEmbeddedCacheManager() {
    return new DefaultCacheManager(new ConfigurationBuilder()
                                   .expiration()
                                      .lifespan(60000l)
                                   .build());
  }

  @RemoteGreetingCache
  @Produces
  @ApplicationScoped
  public RemoteCacheManager specificRemoteCacheManager() {
```

```
        return new RemoteCacheManager("localhost", 1544);
    }
}
```

上記のコードでは、GreetingCache または RemoteGreetingCache は生成されたキャッシュマネージャーに関連付けられます。

プロデューサーメソッドのスコープ

プロデューサーが適切に機能するには、スコープ @ApplicationScoped が必要です。そうでないと、キャッシュの注入はそれぞれキャッシュマネージャーの新しいインスタンスに関連付けられます。

15.5. JCACHE キャッシュアノテーションの使用

ヒント

API を含む JSR 107(JCACHE)統合に別のモジュールが追加されました。詳細は、本章 を参照してください。

CDI 統合および JCache アーティファクトがクラスパスに存在する場合は、CDI 管理対象 Bean で JCache アノテーションを使用できます。これらのアノテーションは、一般的なユースケースをより簡単に処理できます。この仕様では、以下のキャッシュアノテーションが定義されます。

- @CacheResult: メソッド呼び出しの結果をキャッシュします。
- @CachePut - メソッドパラメーターをキャッシュします。
- @CacheRemoveEntry - キャッシュからエントリーを削除します。
- @CacheRemoveAll - removes all entries from a cache

アノテーションのターゲットタイプ

これらのアノテーションはメソッドでのみ使用する必要があります。

*これらのアノテーションを使用するには、適切なインターセプターを **beans.xml** ファイルで宣言する必要があります。*

アプリケーションサーバーなどの管理環境のインターセプター

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
   version="1.2" bean-discovery-mode="annotated">

  <interceptors>
    <class>org.infinispan.jcache.annotation.InjectedCacheResultInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedCachePutInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedCacheRemoveEntryInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedCacheRemoveAllInterceptor</class>
  </interceptors>
</beans>
```

スタンドアロンアプリケーションなどの管理対象外の環境のインターセプター

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
   version="1.2" bean-discovery-mode="annotated">

  <interceptors>
    <class>org.infinispan.jcache.annotation.CacheResultInterceptor</class>
    <class>org.infinispan.jcache.annotation.CachePutInterceptor</class>
    <class>org.infinispan.jcache.annotation.CacheRemoveEntryInterceptor</class>
```

```
    <class>org.infinispan.jcache.annotation.CacheRemoveAllInterceptor</class>
  </interceptors>
</beans>
```

以下のコードのスニペットは @CacheResult アノテーションの使用を示しています。ご覧のとおり、Greetingservice#greet メソッドの結果のキャッシュが簡素化されます。

JCache アノテーションの使用

```
import javax.cache.interceptor.CacheResult;

public class GreetingService {

   @CacheResult
   public String greet(String user) {
      return "Hello" + user;
   }
}
```

GreetingService と上記のバージョンの最初のバージョンは同じ動作を持ちます。唯一の違いは、使用されるキャッシュのみです。デフォルトでは、パラメーター型（例：org.infinispan.example.GreetingService.greet(java.lang.String)）を持つアノテーション付きメソッドの完全修飾名になります。

default 以外のキャッシュを使用することはシンプルです。キャッシュアノテーションの cacheName 属性を使用して名前を指定する必要があります。以下に例を示します。

JCache のキャッシュ名の指定

```
@CacheResult(cacheName = "greeting-cache")
```

15.6. キャッシュイベントと CDI の使用

CDI イベントを使用して Cache および Cache Manager レベルのイベントを受信できます。これは、以下のスニペットに示すように @Observes アノテーションを使用して実現できます。

CDI に基づくイベントリスナー

```
import javax.enterprise.event.Observes;
import org.infinispan.notifications.cachemanagerlistener.event.CacheStartedEvent;
import org.infinispan.notifications.cachelistener.event.*;

public class GreetingService {

    // Cache level events
    private void entryRemovedFromCache(@Observes CacheEntryCreatedEvent event) {
        ...
    }

    // Cache Manager level events
    private void cacheStarted(@Observes CacheStartedEvent event) {
        ...
    }
}
```

ヒント

イベントタイプに関する詳細は、「リスナーおよび通知」を参照してください。

第16章 JCACHE(JSR-107)プロバイダー

Red Hat Data Grid は JCache 1.0 API(JSR-107)の実装を提供します。JCacheは、一時Javaオブジェクトをメモリにキャッシュするための標準Java APIを指定します。Javaオブジェクトをキャッシュすると、取得にコストがかかるデータ（DBやWebサービスなど）や計算が難しいデータを使用することで発生するボトルネックを回避するのに役立ちます。これらのタイプのオブジェクトをメモリにキャッシュすると、コストのかかるラウンドトリップや再計算を行う代わりに、メモリから直接データを取得することで、アプリケーションのパフォーマンスを高速化できます。本書では、仕様の Red Hat Data Grid の実装で JCache を使用する方法を指定し、API の主要な側面について説明します。

16.1. 依存関係

Red Hat Data Grid JCache 実装の使用を開始するには、Maven の pom.xml ファイルに単一の依存関係を追加する必要があります。

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-jcache</artifactId>
  <version>${version.infinispan}</version>
  <scope>test</scope>
</dependency>
```

${version.infinispan} を Red Hat Data Grid の適切なバージョンに置き換えます。

16.2. ローカルキャッシュの作成

JCache API 仕様で定義されているデフォルトの設定オプションを使用したローカルキャッシュの作成は、以下のようにシンプルです。

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

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

警告

デフォルトでは、JCache API はデータを storeByValue として保存するように指定しているため、キャッシュへの操作以外のオブジェクト状態の変更は、キャッシュに保存されているオブジェクトに影響を与えません。これまで、Red Hat Data Grid はシリアル化/マーシャリングを使用してキャッシュに保存するコピーを作成し、その方法は仕様に従って実装しました。そのため、Red Hat Data Grid でデフォルトの JCache 設定を使用する場合、保存されたデータはマーシャリングできる必要があります。

JCache は、（Red Hat Data Grid や JDK Collections のように）参照によりデータを保存するように設定できます。これを行うには、次のコマンドを実行します。

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

16.3. リモートキャッシュの作成

JCache API 仕様で定義されているデフォルトの設定オプションを使用したリモートキャッシュ（クライアントサーバーモード）の作成は、以下のように簡単です。

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

// Retrieve the system wide cache manager via org.infinispan.jcache.remote.JCachingProvider
CacheManager cacheManager =
Caching.getCachingProvider("org.infinispan.jcache.remote.JCachingProvider").getCacheMan
ager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("remoteNamedCache",
    new MutableConfiguration<String, String>());
```

注記

org.infinispan.jcache.remote.JCachingProvider を使用するには、infinispan-jcache-remote-<version>.jar とその推移的な依存関係がすべてクラスパスに配置する必要があります。

16.4. データの保管および取得

JCacheのAPIがjava.util.Mapまたはjava.util.concurrent.ConcurrentMapのいずれも拡張していないにもかかわらず、キー/値のAPIを提供してデータを格納および取得します。

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

CacheManager cacheManager = Caching.getCacheManager();
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
cache.put("hello", "world"); // Notice that javax.cache.Cache.put(K) returns void!
String value = cache.get("hello"); // Returns "world"
```

標準の java.util.Map とは異なり、javax.cache.Cache には put と getAndPut と呼ばれる 2 つの基本的な put メソッドが含まれています。前者は void を返しますが、後者はキーに関連付けられた以前の値を返します。そのため、JCache の java.util.Map.put(K) に相当するものは javax.cache.Cache.getAndPut(K) になります。

ヒント

JCache APIはスタンドアロンキャッシングのみを対象としていますが、永続ストアにプラグインすることができ、クラスタリングまたは分散を念頭に置いて設計されています。javax.cache.Cache が 2 つの put メソッドを提供する理由は、標準の java.util.Map put 呼び出しにより以前の値を計算するためです。永続ストアが使用されている場合、またはキャッシュが分散されている場合、前の値を返すことはコストのかかる操作になる可能性があり、多くの場合、ユーザーは戻り値を使用せずに標準の java.util.Map.put(K) を呼び出します。そのため、JCache ユーザーは戻り値が関連するかどうかについて考慮する必要があります。この場合、javax.cache.Cache.getAndPut(K) を呼び出す必要があります。それ以外の場合は、java.util.Map.put(K, V) を呼び出すことができ、以前の値を返すようなコストのかかる操作が返されなくなります。

16.5. JAVA.UTIL.CONCURRENT.CONCURRENTMAP と JAVAX.CACHE.CACHE APIS の比較

ここでは、java.util.concurrent.ConcurrentMap および javax.cache.Cache API によって提供されるデータ操作 API を簡単に比較します。

操作	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
保存して返さない	該当なし	**void put(K key)**
保存して以前の値を返す	**V put(K key)**	**V getAndPut(K key)**
存在しない場合は保存する	**V putIfAbsent(K key, V value)**	**boolean putIfAbsent(K key, V value)**

操作	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
取得	**V get(Object key)**	**V get(K key)**
存在する場合は削除	**V remove(Object key)**	**boolean remove(K key)**
以前の値を削除して返す	**V remove(Object key)**	**V getAndRemove(K key)**
条件の削除	**boolean remove(Object key, Object value)**	**boolean remove(K key, V oldValue)**
存在する場合は置き換え	**V replace(K key, V value)**	**boolean replace(K key, V value)**
以前の値を置き換えて返す	**V replace(K key, V value)**	**V getAndReplace(K key, V value)**
条件の置き換え	**boolean replace(K key, V oldValue, V newValue)**	**boolean replace(K key, V oldValue, V newValue)**

2つのAPIを比較すると、可能であれば、JCacheが以前の値を返さないようにして、コストのかかるネットワークまたはIO操作を実行するオペレーションを回避していることがわかります。これは、JCache API の設計における最も重要な原則です。実際、java.util.concurrent.ConcurrentMapには存在するが、分散キャッシュでの計算にコストがかかる可能性があるため、javax.cache.Cacheには存在しない一連のオペレーションがあります。唯一の例外は、キャッシュの内容を反復処理することです。

操作	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
キャッシュのサイズを計算する	**int size()**	該当なし
キャッシュのすべてのキーを返す	**Set<K> keySet()**	該当なし
キャッシュのすべての値を返す	**Collection<V> values()**	該当なし
キャッシュ内のすべてのエントリーを返す	**Set<Map.Entry<K, V>> entrySet()**	該当なし
キャッシュを繰り返し処理する	keySet、value、または entrySet で **iterator()** メソッドを使用します。	**Iterator<Cache.Entry<K, V>> iterator()**

16.6. JCACHE インスタンスのクラスタリング

Red Hat Data Grid JCache の実装は仕様の範囲を超え、標準 API を使用してクラスターのキャッシュを提供します。以下のようなキャッシュを複製するように設定された Red Hat Data Grid 設定ファイルがあるとします。

infinispan.xml

```
<infinispan>
  <cache-container default-cache="namedCache">
    <transport cluster="jcache-cluster" />
    <replicated-cache name="namedCache" />
  </cache-container>
</infinispan>
```

このコードを使用して、キャッシュのクラスターを作成できます。

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

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

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

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

// --

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

第17章 管理ツール

Red Hat Data Grid インスタンスの管理はすべて関連する統計情報を公開し、管理者が各 Red Hat Data Grid インスタンスの状態を確認できるようにします。1 つのインストールで数十万または数百の Red Hat Data Grid インスタンスを使用でき、効率的な方法で明確かつ簡潔な情報を提供することは必須です。以下のセクションでは、Red Hat Data Grid が提供する管理ツールの範囲を説明します。

17.1. JMX

これまでは、JMX はミドルウェアの管理および管理のためのデファクト標準となっています。その結果、Red Hat Data Grid チームは、管理情報や統計情報を公開するために、この技術を標準としています。

17.1.1. 公開される MBean について

JConsole や VisualVM などの標準の JMX GUI で Red Hat Data Grid が実行されている仮想マシンに接続すると、以下の MBean を見つける必要があります。

- CacheManager レベルの JMX 統計については、CacheManager MBean によって指定されたプロパティーを持つ org.infinispan:type=CacheManager,name="DefaultCacheManager" という MBean が表示されるはずです。
- globalJmxStatistics XML 要素で cacheManagerName 属性を使用するか、対応する GlobalJmxStatisticsConfigurationBuilder.cacheManagerName(String cacheManagerName)呼び出しを使用すると、JMX オブジェクト名の一部として名前が使用されるようにキャッシュマネージャーに名前を付けることができます。そのため、名前が「Hibernate2LC」であった場合、キャッシュマネージャーの JMX 名は org.infinispan:type=CacheManager,name="Hibernate2LC" になります。これにより、JMX のベストプラクティス に従って複数のキャッシュマネージャーがデプロイされる環境を管理する優れたクリーン方法が提供されます。
- キャッシュレベルの JMX 統計では、どの設定オプションが有効になっているかによって、いくつかの異なる MBean が表示されるはずです。たとえば、キャッシュストアの背後で書き込みを設定している場合、キャッシュストアコンポーネントに属する MBean を公開するプロパティーが表示されるはずです。すべてのキャッシュレベル MBean は、org.infinispan:type=Cache,name="${name-of-cache}(${cache-mode})",manager="${name-of-cache-manager}",component=${component-name} と同じ形式に従います。
- ${name-of-cache} は実際のキャッシュ名に置き換えられました。このキャッシュがデフォルトのキャッシュを表す場合は、その名前が defaultCache になります。

- ${cache-mode} はキャッシュのキャッシュモードに置き換えられました。キャッシュモードは、に示される可能な列挙値の小文字バージョンで表されます。https://access.redhat.com/webassets/avalon/d/red-hat-data-grid/7.3/api/org/infinispan/configuration/cache/CacheMode

- ${name-of-cache-manager} が、このキャッシュが属するキャッシュマネージャーの名前に置き換えられました。名前は globalJmxStatistics 要素の cacheManagerName 属性値から取得されます。

- ${component-name} は、JMX 参照ドキュメント の JMX コンポーネント名の 1 つに置き換えられました。

たとえば、同期ディストリビューションで設定されたデフォルトキャッシュのキャッシュストア JMX コンポーネント MBean の名前は、org.infinispan:type=Cache,name="___defaultcache(dist_sync)",manager="DefaultCacheManager",component=CacheStoreになります。

キャッシュおよびキャッシュマネージャー名は、これらのユーザー定義の名前で使用される不正な文字から保護するために引用符で囲まれることに注意してください。

17.1.2. JMX 統計の有効化

前のセクションで説明した MBean は常に MBeanServer に作成および登録されるため、キャッシュを管理できますが、一部の属性では統計のコレクションを有効にする追加の手順がない限り、意味のある値が公開されません。JMX による統計の収集およびレポートは、2 つの異なるレベルで有効にできます。

CacheManager レベル

CacheManager は、そこから作成されたすべてのキャッシュインスタンスを管理するエンティティーです。CacheManager 統計コレクションの有効化は、設定スタイルによって異なります。

- XML で CacheManager を設定する場合は、以下の XML を < cache-container /> 要素の下に追加し てください。

```
<cache-container statistics="true"/>
```

- *CacheManager をプログラム的に設定する場合は、以下のコードを追加するだけです。*

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...
globalConfigurationBuilder.globalJmxStatistics().enable();
```

キャッシュレベル

このレベルでは、各キャッシュインスタンスによって生成された管理情報を受信します。キャッシュ統計コレクションの有効化は、設定スタイルによって異なります。

- *XML で Cache を設定する場合は、< local-cache /> などの最上位のキャッシュ要素の下に以下の XML を追加して ください。*

```
<local-cache statistics="true"/>
```

- *キャッシュをプログラム的に設定する場合は、以下のコードを追加するだけです。*

```
ConfigurationBuilder configurationBuilder = ...
configurationBuilder.jmxStatistics().enable();
```

17.1.3. クラスターの正常性のモニタリング

JMX を使用して Red Hat Data Grid クラスターの正常性を監視することもできます。CacheManager には、CacheContainerHealth と呼ばれる追加のオブジェクトがあります。これには、以下の属性が含まれます。

- *cacheHealth: キャッシュと対応するステータス（HEALTHY、UNHEALTHY、REBALANCING）の一覧*
- *clusterHealth - cluster health*
- *clusterName - クラスター名*
- *freeMemoryKb: KB で測定される JVM ランタイムから取得した空きメモリー*

- numberOfCpus: JVM ランタイムから取得した CPU の数
- numberOfNodes: クラスター内のノード数
- totalMemoryKb: KB で測定された JVM ランタイムから取得した合計メモリー

17.1.4. 複数の JMX ドメイン

単一の仮想マシンに複数の CacheManager インスタンスが作成されるか、同じ仮想マシンのクラッシュ下で異なる CacheManager に属する Cache 名が作成される可能性があります。

マルチキャッシュマネージャー環境に異なる JMX ドメインを使用することは、最後にすべきです。代わりに、キャッシュマネージャーに名前を付けて、監視ツールにより簡単に識別し、使用できるようになりました。以下に例を示します。

- XML 経由：

```
<cache-container statistics="true" name="Hibernate2LC"/>
```

- プログラムで行う:

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...
globalConfigurationBuilder.globalJmxStatistics()
   .enable()
   .cacheManagerName("Hibernate2LC");
```

これらのオプションのいずれかを使用すると、CacheManager MBean 名が org.infinispan:type=CacheManager,name="Hibernate2LC"になります。

その間は、ドメインが重複している場合や、JMX 名を重複させる必要がある場合は、独自の jmxDomain を設定できますが、同じ JVM 内の異なるキャッシュマネージャーの名前が同等になる特殊なケースに限定する必要があります。

17.1.5. 非デフォルト MBean サーバーでの MBean の登録

Red Hat Data Grid がこれらすべての MBean を登録する場所を説明しましょう。デフォルトでは、

Red Hat Data Grid は 標準の JVM MBeanServer プラットフォーム に登録します。ただし、これらの MBean を異なる MBean インスタンスに登録する場合があります。たとえば、アプリケーションサーバーは、デフォルトのプラットフォームとは別の MBeanServer インスタンスと連携する場合があります。このような場合、Red Hat Data Grid が提供する MBeanServerLookup インターフェース を実装する必要があります。これにより、getMBeanServer（）メソッド は管理 MBean を登録する MBeanServer を返します。実装の準備ができたら、このクラスの完全修飾名で Red Hat Data Grid を設定するだけです。以下に例を示します。

- XML 経由：

```
<cache-container statistics="true">
  <jmx mbean-server-lookup="com.acme.MyMBeanServerLookup" />
</cache-container>
```

- プログラムで行う：

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...
globalConfigurationBuilder.globalJmxStatistics()
   .enable()
   .mBeanServerLookup(new com.acme.MyMBeanServerLookup());
```

17.1.6. 利用可能な MBean

利用可能な MBean の完全リストは、JMX リファレンスドキュメントを参照してください。

17.2. コマンドラインインターフェース(CLI)

Red Hat Data Grid は、キャッシュ内のデータや、ほとんどの内部コンポーネント（トランザクション、クロスサイトバックアップ、ローリングアップグレードなど）と相互作用できるシンプルなコマンドラインインターフェース(CLI)を提供します。

CLI は、サーバー側のモジュールとクライアントコマンドツールという 2 つの要素から構築されます。サーバー側のモジュール(infinispan-cli-server-$VERSION.jar)は、コマンドに実際のインタープリターを提供し、アプリケーションと一緒に組み込む必要があります。Red Hat Data Grid Server には、追加設定なしで CLI のサポートが含まれています。

現在、サーバー（およびクライアント）は JMX プロトコルを使用して通信しますが、今後のリリースでは、他の通信プロトコル（特に独自のホットミネート）をサポートする予定です。

CLI は対話モードとバッチモードの両方を提供します。クライアントを呼び出すには、bin/cli.

[sh|bat] スクリプトを実行します。

以下は、CLI の起動方法に影響するコマンドラインスイッチの一覧です。

```
-c, --connect=URL       connects to a running instance of Infinispan.
                        JMX over RMI jmx://[username[:password]]@host:port[/container[/cache]]
                        JMX over JBoss remoting
remoting://[username[:password]]@host:port[/container[/cache]]
-f, --file=FILE         reads input from the specified file instead of using
                        interactive mode. If FILE is '-', then commands will be read
                        from stdin
-h, --help              shows this help page
-v, --version           shows version information
```

- JMX over RMI は、JMX クライアントが MBeanServers に接続する従来の方法です。監視するプロセスの設定方法の詳細は、JDK Monitoring および Management のドキュメントを参照してください。

- Red Hat Data Grid アプリケーションが EAP 内で実行されている場合、JBoss Remoting 上の JMX が好まれるプロトコルになります。

アプリケーションへの接続は、connect コマンドを使用して CLI 内から開始することもできます。

```
[disconnected//]> connect jmx://localhost:12000
[jmx://localhost:12000/MyCacheManager/>
```

CLI プロンプトには、現在選択されている CacheManager を含むアクティブな接続情報が表示されます。最初にキャッシュが選択されていないため、キャッシュ操作を実行する前に選択する必要があります。これには **cache** コマンドを使用します。CLI は、すべてのコマンドおよびオプションと、これが意味のあるほとんどのパラメーターのタブ補完をサポートします。そのため、キャッシュ を入力して TAB を押すと、アクティブなキャッシュの一覧が表示されます。

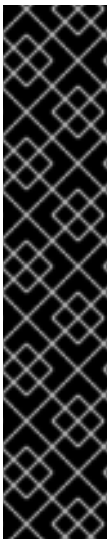
```
[jmx://localhost:12000/MyCacheManager/> cache
___defaultcache  namedCache
[jmx://localhost:12000/MyCacheManager/]> cache ___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]>
```

空のプロンプトで TAB を押すと、利用可能なすべてのコマンドの一覧が表示されます。

```
alias     cache     container  encoding  get       locate    remove    site      upgrade
abort     clearcache create     end       help      put       replace   start     version
begin     commit    disconnect  evict     info      quit      rollback  stats
```

-

CLI は gitops sh をベースとしているため、コマンドの履歴を移動および検索するためのキーボードショートカットが多数あり、操作の Emacs モードと VI モードの両方を含むプロンプトでカーソルを操作することができます。



重要

Red Hat Data Grid CLI セッションは、6 分以上アイドル状態のと有効期限が切れます。セッションの期限が切れた後にコマンドを実行すると、以下のメッセージが表示されます。

ISPN019015: Invalid session id '<session_id>'

新しいセッションを開始するには、CLI を再起動する必要があります。

17.2.1. コマンド

17.2.1.1. 強制終了

abort コマンドは、start コマンドで開始された実行中のバッチを中止するために使用されます。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> abort
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
null
```

17.2.1.2. begin

begin コマンドはトランザクションを開始します。このコマンドが機能するには、後続の操作を呼び出すキャッシュでトランザクションを有効にする必要があります。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> commit
```

17.2.1.3. cache

cache コマンドは、後続のすべての操作でデフォルトとして使用するキャッシュを選択します。パ

ラメーターなしで呼び出される場合は、現在選択されているキャッシュが表示されます。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> cache ___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]> cache
___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]>
```

17.2.1.4. clearcache

clearcache コマンドは、すべてのコンテンツからキャッシュを削除します。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> clearcache
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
null
```

17.2.1.5. commit

commit コマンドは、継続中のトランザクションをコミットします。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> commit
```

17.2.1.6. container

container コマンドはデフォルトのコンテナー（キャッシュマネージャー）を選択します。パラメーターなしで呼び出されると、利用可能なすべてのコンテナーが一覧表示されます。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> container
MyCacheManager OtherCacheManager
[jmx://localhost:12000/MyCacheManager/namedCache]> container OtherCacheManager
[jmx://localhost:12000/OtherCacheManager/]>
```

17.2.1.7. create

create コマンドは、既存のキャッシュ定義の設定に基づいて新しいキャッシュを作成します。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> create newCache like namedCache
[jmx://localhost:12000/MyCacheManager/namedCache]> cache newCache
[jmx://localhost:12000/MyCacheManager/newCache]>
```

17.2.1.8. deny

承認が有効になり、ロールマッパーが ClusterRoleMapper に設定される場合、ロールマッピングへのプリンシパルはクラスターレジストリー内に保存されます（すべてのノードで利用可能な複製キャッシュ）。deny コマンドを使用すると、以前にプリンシパルに割り当てられたロールを拒否できます。

```
[remoting://localhost:9999]> deny supervisor to user1
```

17.2.1.9. disconnect

disconnect コマンドは、現在アクティブな接続を切断して、CLI が別のインスタンスに接続できるようにします。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> disconnect
[disconnected//]
```

17.2.1.10. encoding

encoding コマンドを使用して、キャッシュから/へのエントリーの読み取り/書き込み時に使用するデフォルトのコーデックを設定します。引数なしで呼び出されると、現在選択されているコーデックが表示されます。このコマンドは、HotRod、Memcached などのリモートプロトコルなど、特殊な構造でキーと値をラップするので便利です。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding
none
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding --list
memcached
hotrod
none
rest
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding hotrod
```

17.2.1.11. end

end コマンドは、start コマンドで開始された実行中のバッチを正常に終了するために使用されます。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> end
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
a
```

17.2.1.12. エビクト

evict コマンドは、特定のキーに関連付けられたエントリーのキャッシュからエビクトするために使用されます。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> evict a
```

17.2.1.13. get

get コマンドは、指定されたキーに関連付けられた値を表示するために使用されます。プリミティブ型と文字列の場合、**get** コマンドは単にデフォルトの表現を出力します。他のオブジェクトの場合は、オブジェクトの **JSON** 表現が出力されます。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
a
```

17.2.1.14. Grant

承認が有効になり、ロールマッパーが **ClusterRoleMapper** に設定される場合、ロールマッピングへのプリンシパルはクラスターレジストリー内に保存されます（すべてのノードで利用可能な複製キャッシュ）。**grant** コマンドを使用して、新規ロールをプリンシパルに付与することができます。

```
[remoting://localhost:9999]> grant supervisor to user1
```

17.2.1.15. info

info コマンドは、現在選択されているキャッシュまたはコンテナーの設定を表示するために使用されます。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> info
GlobalConfiguration{asyncListenerExecutor=ExecutorFactoryConfiguration{factory=org.infinispan.execu
ors.DefaultExecutorFactory@98add58},
asyncTransportExecutor=ExecutorFactoryConfiguration{factory=org.infinispan.executors.DefaultExecut
rFactory@7bc9c14c},
evictionScheduledExecutor=ScheduledExecutorFactoryConfiguration{factory=org.infinispan.executors.D
efaultScheduledExecutorFactory@7ab1a411},
replicationQueueScheduledExecutor=ScheduledExecutorFactoryConfiguration{factory=org.infinispan.ex
ecutors.DefaultScheduledExecutorFactory@248a9705},
globalJmxStatistics=GlobalJmxStatisticsConfiguration{allowDuplicateDomains=true, enabled=true,
jmxDomain='jboss.infinispan',
mBeanServerLookup=org.jboss.as.clustering.infinispan.MBeanServerProvider@6c0dc01,
cacheManagerName='local', properties={}}, transport=TransportConfiguration{clusterName='ISPN',
machineId='null', rackId='null', siteId='null', strictPeerToPeer=false, distributedSyncTimeout=240000,
```

```
transport=null, nodeName='null', properties={}},
serialization=SerializationConfiguration{advancedExternalizers=
{1100=org.infinispan.server.core.CacheValue$Externalizer@5fabc91d,
1101=org.infinispan.server.memcached.MemcachedValue$Externalizer@720bffd,
1104=org.infinispan.server.hotrod.ServerAddress$Externalizer@771c7eb2},
marshaller=org.infinispan.marshall.VersionAwareMarshaller@6fc21535, version=52,
classResolver=org.jboss.marshalling.ModularClassResolver@2efe83e5},
shutdown=ShutdownConfiguration{hookBehavior=DONT_REGISTER}, modules={},
site=SiteConfiguration{localSite='null'}}
```

17.2.1.16. locate

locate コマンドは、分散クラスターの指定されたエントリーの物理的な場所を表示します。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> locate a
[host/node1,host/node2]
```

17.2.1.17. put

put コマンドはエントリーをキャッシュに挿入します。キャッシュにキーのマッピングが以前に含まれていた場合、古い値は指定された値に置き換えられます。ユーザーは、CLI がキーと値を保存するために使用するデータのタイプを制御できます。「Data Types」セクションを参照してください。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b 100
[jmx://localhost:12000/MyCacheManager/namedCache]> put c 4139l
[jmx://localhost:12000/MyCacheManager/namedCache]> put d true
[jmx://localhost:12000/MyCacheManager/namedCache]> put e { "package.MyClass": {"i": 5, "x": null,
"b": true } }
```

put コマンドはオプションで、ライフスパンと最大アイドル時間を指定することができます。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a expires 10s
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a expires 10m maxidle 1m
```

17.2.1.18. replace

replace コマンドは、キャッシュ内の既存のエントリーを置き換えます。古い値を指定すると、キャッシュの一貫性が保たれている場合にのみ代替が発生します。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
b
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b c
```

```
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
c
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b d
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
c
```

17.2.1.19. roles

承認が有効になり、ロールマッパーが ClusterRoleMapper に設定される場合、ロールマッピングへのプリンシパルはクラスターレジストリー内に保存されます（すべてのノードで利用可能な複製キャッシュ）。roles コマンドは、特定のユーザーに関連するロールを一覧表示するには、または全ユーザーに割り当てられているロールを一覧表示するには、以下のコマンドを実行します。

```
[remoting://localhost:9999]> roles user1
[supervisor, reader]
```

17.2.1.20. rollback

rollback コマンドは、継続中のトランザクションをロールバックします。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> rollback
```

17.2.1.21. site

site コマンドは、クロスサイトレプリケーションの管理に関連する操作を実行します。サイトのステータスに関連する情報を取得し、ステータスを変更できます(online/offline)。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> site --status NYC
online
[jmx://localhost:12000/MyCacheManager/namedCache]> site --offline NYC
ok
[jmx://localhost:12000/MyCacheManager/namedCache]> site --status NYC
offline
[jmx://localhost:12000/MyCacheManager/namedCache]> site --online NYC
```

17.2.1.22. start

start コマンドは、操作のバッチを開始します。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
```

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> end
```

17.2.1.23. stats

stats コマンドはキャッシュに関する統計を表示します。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> stats
Statistics: {
  averageWriteTime: 143
  evictions: 10
  misses: 5
  hitRatio: 1.0
  readWriteRatio: 10.0
  removeMisses: 0
  timeSinceReset: 2123
  statisticsEnabled: true
  stores: 100
  elapsedTime: 93
  averageReadTime: 14
  removeHits: 0
  numberOfEntries: 100
  hits: 1000
}
LockManager: {
  concurrencyLevel: 1000
  numberOfLocksAvailable: 0
  numberOfLocksHeld: 0
}
```

17.2.2. upgrade

upgrade コマンドは、ローリングアップグレードの手順で使用される操作を実行します。この手順の詳細な説明は、「ローリングアップグレード」を参照 して ください。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> upgrade --synchronize=hotrod --all
[jmx://localhost:12000/MyCacheManager/namedCache]> upgrade --disconnectsource=hotrod --all
```

17.2.3. version

version コマンドは、**CLI** クライアントとサーバー両方に関するバージョン情報を表示します。

```
[jmx://localhost:12000/MyCacheManager/namedCache]> version
Client Version 5.2.1.Final
Server Version 5.2.1.Final
```

17.2.4. データタイプ

CLI は以下のタイプを理解します。

- 文字列文字列は、一重引用符(')または二重引用符(")の間引用符で囲むか、引用符で囲むことができます。この場合、スペースを入れず、句読点を入れず、数字（例：「文字列」、「key001」など）で始めることはできません。
- int an integer は、10 進数のシーケンスで識別されます（例： 256）。
- long は、サフィックスが 'l' が付いた 10 進数のシーケンスで識別されます（例： 1000l）。
- double
  - 二重精度番号は、浮動小数点番号（オプションの指数）およびオプションの「d」接尾辞（3.14 など）で識別されます。
- float
  - 単一の精度番号は、浮動小数点番号（オプションの exponent part）と 'f' の接尾辞で識別されます（例： 10.3f）。
- ブール値はキーワード true および false で表される
- UUID は、正規の形式 XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX で表されます。
- JSON のシリアライズされた Java クラスは JSON 表記を使用して表すことができます（例： {"package.MyClass":{"i":5,"x":null,"b":true}}）。指定のクラスは CacheManager のクラスローダーで使用できる必要があることに注意してください。

17.2.5. 時間の値

時間値は整数で、時間単位接尾辞（日(d)、時間(h)、分(m)、秒(s)、ミリ秒(ms)）です。

17.2.6. Red Hat Data Grid エンドポイントの開始および停止

コマンドラインインターフェース(CLI)を使用して、Red Hat Data Grid エンドポイントコネクターを開始および停止します。

エンドポイントコネクターを開始および停止するコマンド：

- 個々のエンドポイントに適用します。すべてのエンドポイントコネクターを停止または開始するには、各エンドポイントコネクターでコマンドを実行する必要があります。
- 単一ノードでのみ有効になります（クラスター全体ではありません）。

手順

1. CLI を起動し、Red Hat Data Grid に接続します。
2. 以下のように、datagrid-infinispan-endpoint サブシステムのエンドポイントコネクターを一覧表示します。

```
[standalone@localhost:9990 /] ls subsystem=datagrid-infinispan-endpoint
hotrod-connector     memcached-connector  rest-connector       router-connector
```

3. 起動または停止するエンドポイントコネクターに移動します。以下に例を示します。

```
[standalone@localhost:9990 /] cd subsystem=datagrid-infinispan-endpoint

[standalone@localhost:9990 subsystem=datagrid-infinispan-endpoint] cd rest-
connector=rest-connector
```

4. 必要に応じて、:stop-connector および :start-connector コマンドを使用します。

```
[standalone@localhost:9990 rest-connector=rest-connector] :stop-connector
{"outcome" => "success"}
```

```
[standalone@localhost:9990 rest-connector=rest-connector] :start-connector
{"outcome" => "success"}
```

17.3. HAWT.IO

slick、fast、HTML5 ベースのオープンソース管理コンソールである Hawt.io は、Red Hat Data Grid もサポートします。このプラグイン の詳細は、Haswt.io のドキュメント を参照してください。

17.4. 他の管理ツールのプラグインの作成

JMX をサポートする管理ツールは、Red Hat Data Grid の基本サポートです。ただし、使用を容易にするために、カスタムプラグインを記述して JMX 情報を調整することができます。

第18章 カスタムインターセプター

カスタムインターセプターは、宣言的およびプログラムで Red Hat 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、応答したりできるため、Red Hat Data Grid を拡張する方法です。要素が追加/削除/更新されること、またはトランザクションがコミットされることが、このような変更の例としてあげられます。詳細なリストは、「CommandInterceptor API」を参照してください。

18.1. カスタムインターセプターの宣言的追加

カスタムの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。これは、名前の付いた各キャッシュに独自のインターセプタースタックがあるためです。以下の xml スニペットは、カスタムインターセプターを追加する方法を示しています。

```
<local-cache name="cacheWithCustomInterceptors">
  <!--
  Define custom interceptors.  All custom interceptors need to extend
org.jboss.cache.interceptors.base.CommandInterceptor
  -->
  <custom-interceptors>
    <interceptor position="FIRST" class="com.mycompany.CustomInterceptor1">
        <property name="attributeOne">value1</property>
        <property name="attributeTwo">value2</property>
    </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"/>
  </custom-interceptors>
</local-cache>
```

18.2. プログラムによるカスタムインターセプターの追加

これを実行するには、AdvancedCache への参照を取得する必要があります。これは以下のように実行できます。

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

次に、addInterceptor() メソッドの 1 つを使用して、実際のインターセプターを追加する必要があります。詳細は、AdvancedCache javadoc を参照してください。

18.3. カスタムインターセプターの設計

カスタムインターセプターを作成するときは、次のルールに従う必要があります。

+ * カスタムインターセプターは BaseCustomInterceptor を拡張する必要があります。カスタムインターセプターは、構築を有効にするためにパブリック、空のコンストラクターを宣言する必要があります。* カスタムインターセプターには、XML 設定で使用されるプロパティータグで定義されたプロパティーのセッターがあります。

第19章 クラウドサービスでの実行

Red Hat Data Grid ライブラリーモードの Cloud サポートを有効にするには、クラスパスに新しい依存関係を追加する必要があります。

ライブラリーモードでのクラウドサポート

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

${version.infinispan} を Red Hat Data Grid の適切なバージョンに置き換えます。

上記の依存関係は、infinispan-core をクラスパスと一部のデフォルト設定に追加します。

19.1. 一般的な検出プロトコル

プライベート環境で Red Hat Data Grid を稼働する主な違いは、マルチキャストが機能しないため、後者のノード検出が若干複雑になることです。これを回避するには、代替の JGroups PING プロトコルを使用できます。クラウド固有のままにする前に、一般的な検出プロトコルをいくつか試すことができます。

19.1.1. TCPPING

TCPPing アプローチには、JGroups 設定ファイルのクラスターの各メンバーの IP アドレスの静的リストが含まれます。これは機能しますが、クラスターノードがクラスターに動的に追加されると、本当に役立ちます。

TCPPing 設定のサンプル

```
<config>
    <TCP bind_port="7800" />
```

```
    <TCPPING timeout="3000"
        initial_hosts="${jgroups.tcpping.initial_hosts:localhost[7800],localhost[7801]}"
        port_range="1"
        num_initial_members="3"/>
...
...
</config>
```

TCPPing の詳細は、「JGroups TCPPING」を参照してください。

19.1.2. GossipRouter

もう 1 つの方法として、中央サーバー（各ノードが接続するように設定されます）を設定することができます。この中央サーバーは、クラスター内の各ノードに対し、他のノードについて指示します。

Gossip ルーターがリッスンしているアドレス(ip:port)は、Red Hat Data Grid によって使用される JGroups 設定に注入することができます。これを行うには、gossip ルーターアドレスをシステムプロパティーとして JVM に渡します（例： -DGossipRouterAddress="10.10.2.4[12001]" ）、Red Hat Data Grid が使用している JGroups 設定でこのプロパティーを参照します。

TCPGOSSIP の設定例

```
<config>
    <TCP bind_port="7800" />
    <TCPGOSSIP timeout="3000" initial_hosts="${GossipRouterAddress}" num_initial_members="3"
/>
...
...
</config>
```

Gossip ルーター @ http://www.jboss.org/community/wiki/JGroupsGossipRouterの詳細

19.2. AMAZON WEB SERVICES

Amazon Web Service(AWS)プラットフォームおよび同様のクラウドベースの環境で実行する場合

は、検出に S3_PING プロトコルを使用できます。

19.3. NATIVE_S3_PING

共有ストレージを使用してクラスターノードの詳細を交換するように JGroups インスタンスを設定できます。NATIVE_S3_PING により、Amazon S3 を共有ストレージとして使用できます。この方法を使用するには、Amazon S3 および EC2 にサインアップしている。

NATIVE_S3_PING の設定例

```
<config>
   <TCP bind_port="7800" />
   <org.jgroups.aws.s3.NATIVE_S3_PING
         region_name="replace this with your region (e.g. eu-west-1)"
         bucket_name="replace this with your bucket name"
         bucket_prefix="replace this with a prefix to use for entries in the bucket (optional)" />
</config>
```

19.3.1. JDBC_PING

S3_PING と同様のアプローチですが、共有データベースへの JDBC 接続を使用します。EC2 では、Amazon RDS を使用すると非常に簡単です。詳細は JDBC_PING Wiki ページ を参照してください。

19.4. MICROSOFT AZURE

Red Hat Data Grid は Azure プラットフォームで使用できます。TCP_PING または GossipRouter を使用する以外に、Azure 固有の検出プロトコルがあります。

19.4.1. AZURE_PING

AZURE_PING は共有 Azure Blob ストレージ を使用して検出情報を保存します。設定は以下のとおりです。

```
<azure.AZURE_PING
 storage_account_name="replace this with your account name"
 storage_access_key="replace this with your access key"
```

```
 container="replace this with your container name"
/>
```

19.5. GOOGLE COMPUTE ENGINE

Red Hat Data Grid は、Google Compute Engine(GCE)プラットフォームで使用できます。TCP_PING または GossipRouter を使用する以外に、GCE 固有の検出プロトコルがあります。

19.5.1. GOOGLE_PING

GOOGLE_PING は Google Cloud Storage(GCS)を使用してクラスターメンバーについての情報を保存します。

```
<protocol type="GOOGLE_PING">
  <property name="location">The name of the bucket</property>
  <property name="access_key">The access key</property>
  <property name="secret_access_key">The secret access key</property>
</protocol>
```

19.6. KUBERNETES

OKD や OpenShift などの Kubernetes 環境の Red Hat Data Grid は、クラスター検出に Kube_PING または [DNS_PING] を使用できます。

19.6.1. Kube_PING

JGroups Kube_PING プロトコルは以下の設定を使用します。

KUBE_PING の設定例

```
<config>
    <TCP bind_addr="${match-interface:eth.*}" />
    <kubernetes.KUBE_PING />
...
...
</config>
```

最も重要なことは、JGroups を eth0 インターフェースにバインドすることです。これは Docker コンテナーがネットワーク接続に使用されます。

KUBE_PING プロトコルは環境変数によって設定されます（コンテナー内で利用可能である必要があります）。最も重要なのは、KUBERNETES_NAMESPACE を適切な名前空間に設定することです。これはハードコーディングされているか、または Kubernetes の Downward API で設定される場合があります。

KUBE_PING は利用可能な Pod を取得するために Kubernetes API を使用するため、OpenShift では追加の権限を追加する必要があります。oc project -q が現在の namespace を返し、default がサービスアカウント名である場合、以下を実行する必要があります。

追加の OpenShift 権限の追加

```
oc policy add-role-to-user view system:serviceaccount:$(oc project -q):default -n $(oc project -q)
```

上記の手順をすべて実行した後、クラスタリングを有効にし、すべての Pod は単一の namespace 内でクラスターを自動的に形成します。

19.6.2. DNS_PING

JGroups DNS_PING プロトコルは以下の設定を使用します。

DNS_PING の設定例

```
<stack name="dns-ping">
...
    <dns.DNS_PING
      dns_query="myservice.myproject.svc.cluster.local" />
...
</stack>
```

DNS_PING は DNS サーバーに対して指定のクエリーを実行して、クラスターメンバーの一覧を取得します。

クラスター内のノードの DNS エントリーを作成する方法は、「サービスおよび Pod の DNS」を参照してください。

19.6.3. Kubernetes および OpenShift のローリングアップデートの使用

Kubernetes および OpenShift の Pod はイミュータブルであるため、設定を変更する唯一の方法は、新規デプロイメントをロールアウトすることです。これを実行するためのストラテジーがいくつかありますが、ローリングアップデート の使用が推奨されます。

Deployment Configuration(Kubernetes)のサンプルは Deploymentと呼ばれる非常に似た概念を使用します。

ローリングアップデート用の DeploymentConfiguration

```
- apiVersion: v1
  kind: DeploymentConfig
  metadata:
    name: infinispan-cluster
  spec:
    replicas: 3
    strategy:
      type: Rolling
      rollingParams:
        updatePeriodSeconds: 10
        intervalSeconds: 20
        timeoutSeconds: 600
        maxUnavailable: 1
        maxSurge: 1
    template:
      spec:
        containers:
        - args:
          - -Djboss.default.jgroups.stack=kubernetes
          image: jboss/infinispan-server:latest
          name: infinispan-server
          ports:
          - containerPort: 8181
            protocol: TCP
          - containerPort: 9990
            protocol: TCP
          - containerPort: 11211
            protocol: TCP
```

```
      - containerPort: 11222
        protocol: TCP
      - containerPort: 57600
        protocol: TCP
      - containerPort: 7600
        protocol: TCP
      - containerPort: 8080
        protocol: TCP
      env:
      - name: KUBERNETES_NAMESPACE
        valueFrom: {fieldRef: {apiVersion: v1, fieldPath: metadata.namespace}}
      terminationMessagePath: /dev/termination-log
      terminationGracePeriodSeconds: 90
      livenessProbe:
        exec:
          command:
          - /usr/local/bin/is_running.sh
        initialDelaySeconds: 10
        timeoutSeconds: 80
        periodSeconds: 60
        successThreshold: 1
        failureThreshold: 5
      readinessProbe:
         exec:
            command:
            - /usr/local/bin/is_healthy.sh
         initialDelaySeconds: 10
         timeoutSeconds: 40
         periodSeconds: 30
         successThreshold: 2
         failureThreshold: 5
```

また、新しいノード（または残す）をより迅速に検出するために JGroups スタックを調整することが強く推奨されます。少なくとも FD_ALL タイムアウトの値を調整し、最も長い GC Pause に調整する必要があります。

設定パラメーターの調整に関するその他のヒントは以下のとおりです。

- OpenShift は実行中のノードを 1 つずつ置き換える必要があります。これは、rollingParams を調整することで実行できます（maxUnavailable: 1 および maxSurge: 1）。

- クラスターサイズに応じて、updatePeriodSeconds および intervalSeconds を調整する必要があります。クラスターサイズが大きくなると、これらの値は大きくなければなりません。

- Initial State Transfer を使用する場合は、両方のプローブの initialDelaySeconds の値を、より高い値に設定する必要があります。

- 初期状態の譲渡中は、プローブに応答されない場合があります。failureThreshold および successThreshold の値を増やすと、最善の結果が得られます。

19.6.4. Kubernetes および OpenShift を使用したローリングアップグレード

ローリングアップグレードとローリング更新も同様にサウンドできますが、異なることを意味します。ローリング更新 は、古い Pod を新規 Pod に置き換えるプロセスです。つまり、新しいバージョンのアプリケーションをロールアウトするプロセスです。典型的な例は設定変更です。Pod はイミュータブルであるため、更新された設定ビットを使用するために Kubernetes/OpenShift はそれらを 1 つずつ置き換える必要があります。一方、ローリングアップグレード は、ある Red Hat Data Grid クラスターから別のクラスターにデータを移行するプロセスです。典型的な例として、あるバージョンから別のバージョンに移行しています。

Kubernetes と OpenShift の両方の場合、ローリングアップグレード手順はほぼ同じです。これは、小規模な変更を含む標準 のローリングアップグレード手順 に基づいています。

OpenShift/Kubernetes を使用してアップグレードする場合の主な相違点は次のとおりです。

- 設定によっては、OpenShift ルート または Kubernetes Ingress API を使用してサービスをクライアントに公開することが推奨されます。アップグレード時に、クライアントによって使用されるルート（または Ingress）は新規クラスターを参照するように変更できます。

- CLI コマンドの呼び出しは、Kubernetes(kubectl exec)または OpenShift クライアント(oc exec)を使用して実行できます。以下に例を示します。oc exec <POD_NAME>gitops-TEMPLATES'/opt/datagrid/bin/cli.sh' '-c' '--controller=$(hostname -i):9990' '/subsystem=datagrid-infinispan/cache-container=clustered/distributed-cache=default:disconnect-source(migrator-name=hotrod)'

ライブラリーモードを使用したアップグレード時の主な相違点：

- クライアントアプリケーションは JMX を公開する必要があります。通常はアプリケーションと環境タイプによって異なりますが、最も簡単な方法として、以下のスイッチを Java ブースト収集スクリプト -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=<PORT> に追加することです。

- JMX への接続は、ポートを転送することで実行できます。OpenShift では、これは oc port-forward コマンドを使用して実行できますが、kubectl の port-forward コマンドで実行で

きます。

ローリングアップグレード（リモートキャッシュストアを再度削除）の最後の手順は異なる方法で実行する必要があります。Kubernetes/OpenShift のローリングアップデート コマンドを使用し、Pod 設定をリモートキャッシュストアを含まないものに置き換える必要があります。

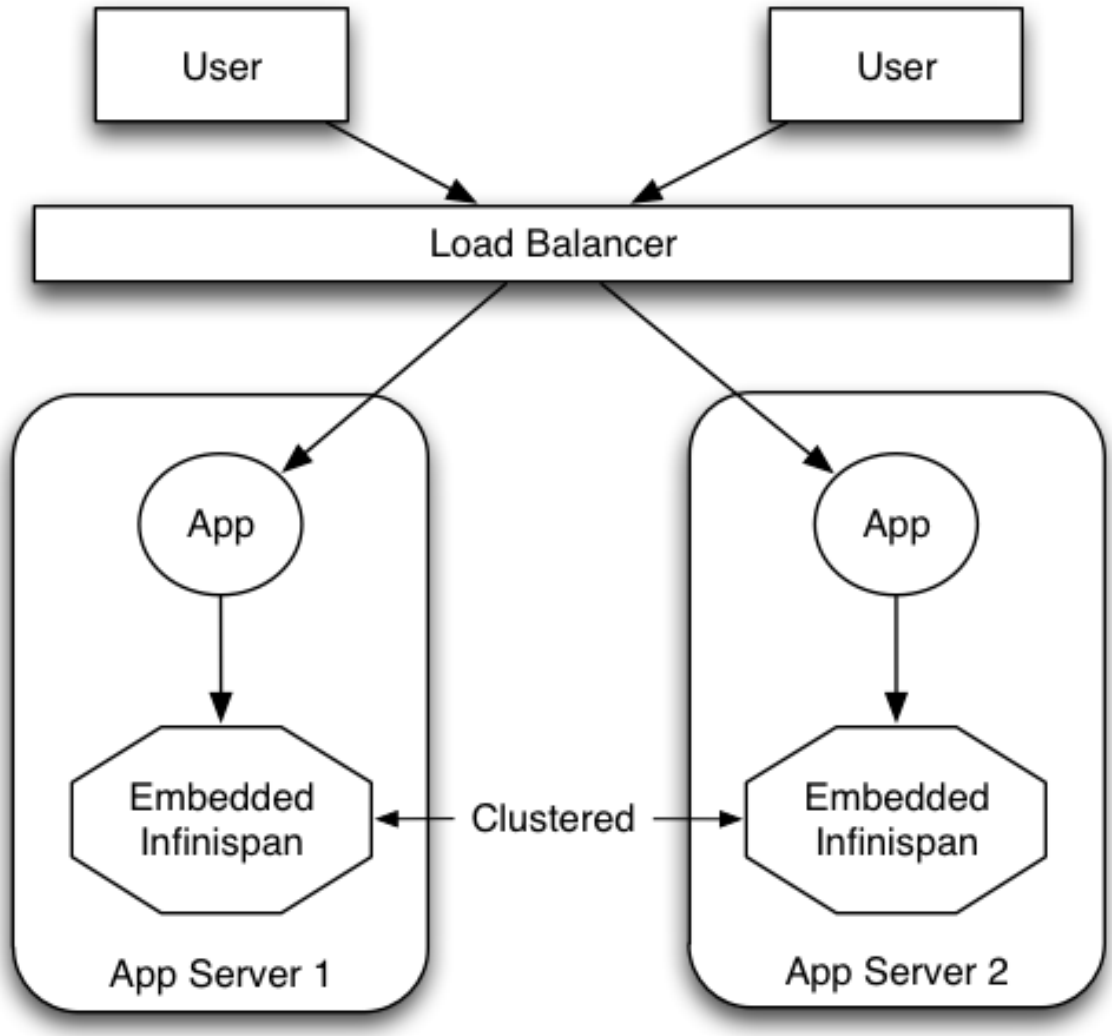
詳細な手順は、ISPN-6673 チケットを参照してください。

第20章 クライアント/サーバー

Red Hat Data Grid は、埋め込みモード と クライアントサーバー モード の 2 つの代替アクセス方法を提供します。

- Embedded モードでは、Red Hat Data Grid ライブラリーは、以下の図に示すように同じ JVM のユーザーアプリケーションと共存します。

図20.1 ピアツーピアアクセス

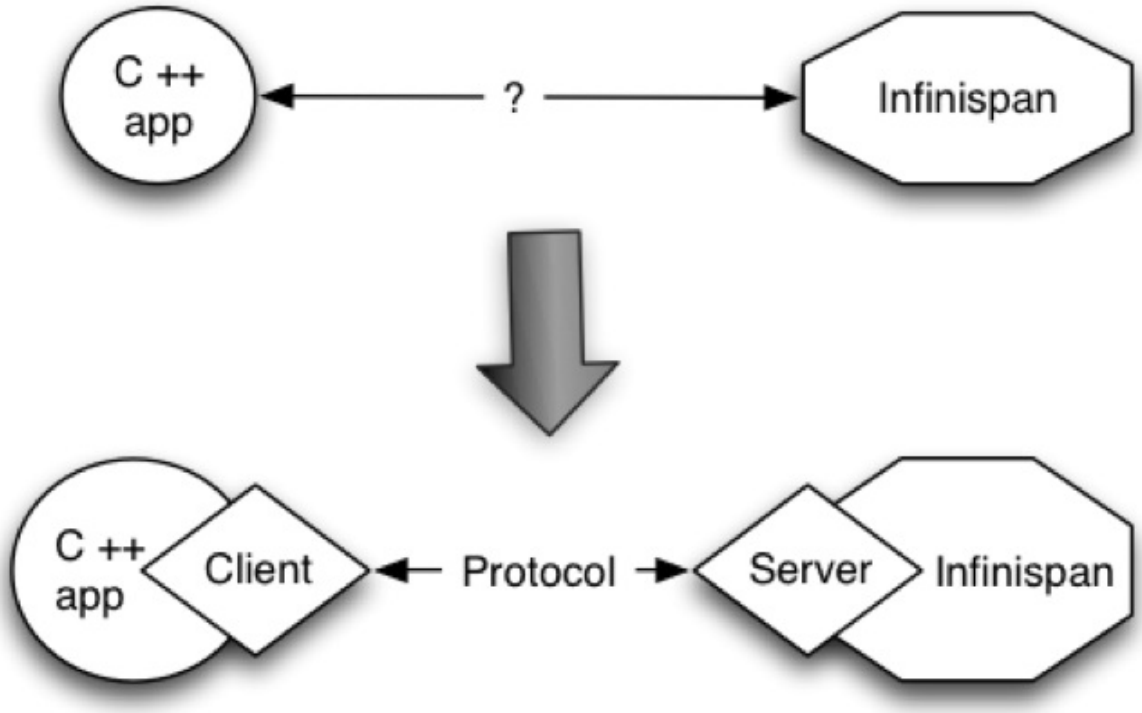


- クライアントサーバーモードは、一部のネットワークプロトコルを使用して、アプリケーションがリモート Red Hat Data Grid サーバーに保存されているデータにアクセスする場合です。

20.1. クライアント/サーバー理由

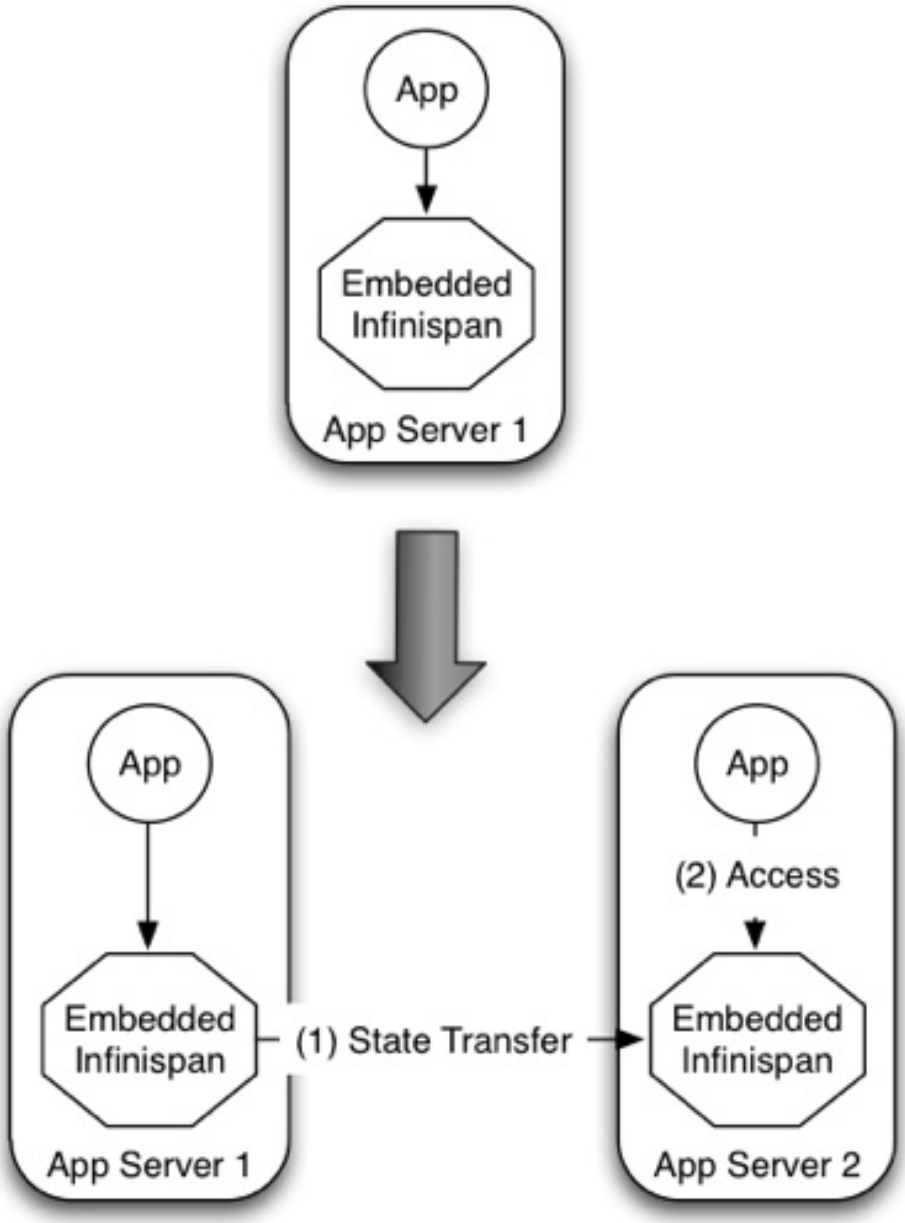
クライアントサーバーモードで Red Hat Data Grid にアクセスすると、たとえば、JVM 以外の環境から Red Hat Data Grid にアクセスしようとすると、アプリケーションに埋め込まれるよりも適切である場合もあります。Red Hat Data Grid は Java で書かれているため、アクセスしたい C\\ アプリケーションがある場合、p2p 方式では実行できません。一方、言語に依存しないプロトコルが使用され、対応するクライアントおよびサーバーの実装が利用可能であることを前提とします。

図20.2 JVM 以外のアクセス



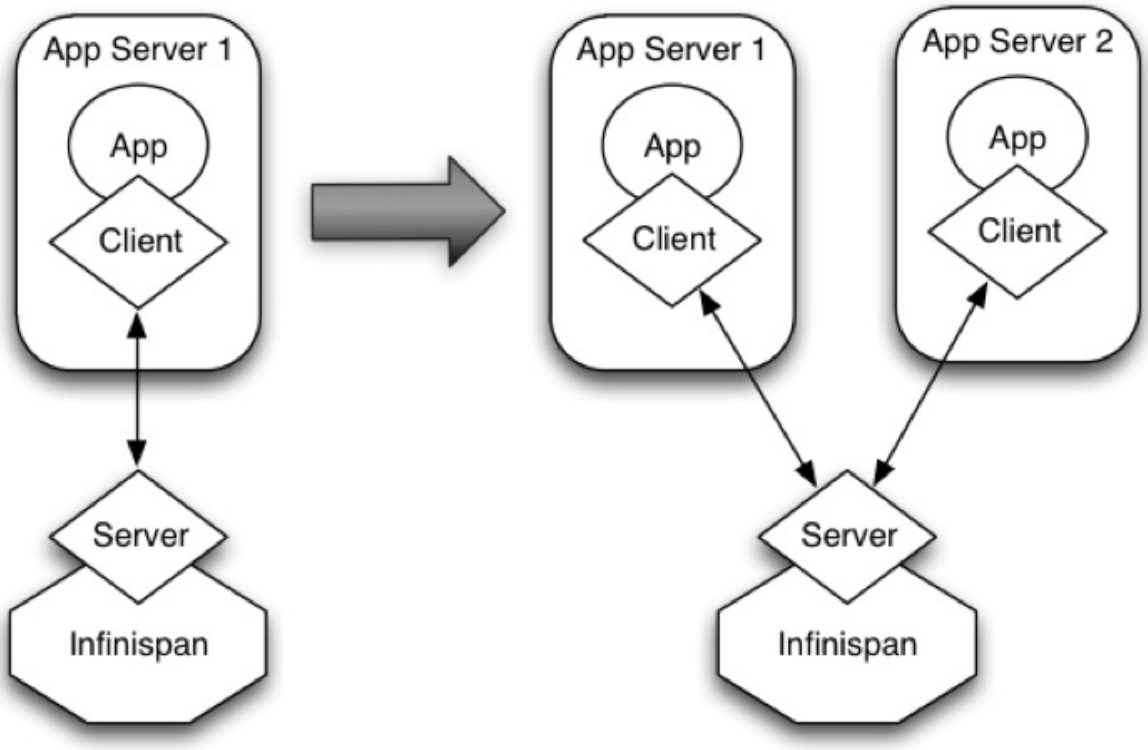
他の状況では、Red Hat Data Grid のユーザーは、ビジネスプロセスサーバーを定期的に起動/停止するエラスティックアプリケーション層が必要です。ユーザーがディストリビューションまたは状態遷移で設定した Red Hat Data Grid をデプロイした場合、このような状況で発生するデータに関するシャッフルに起動時間が大きく影響できるようになりました。以下の図では、Red Hat Data Grid が p2p モードでデプロイされたと、状態遷移が完了するまで 2 番目のサーバーのアプリケーションが Red Hat Data Grid にアクセスできませんでした。

図20.3 P2P でのエラビリティーの問題



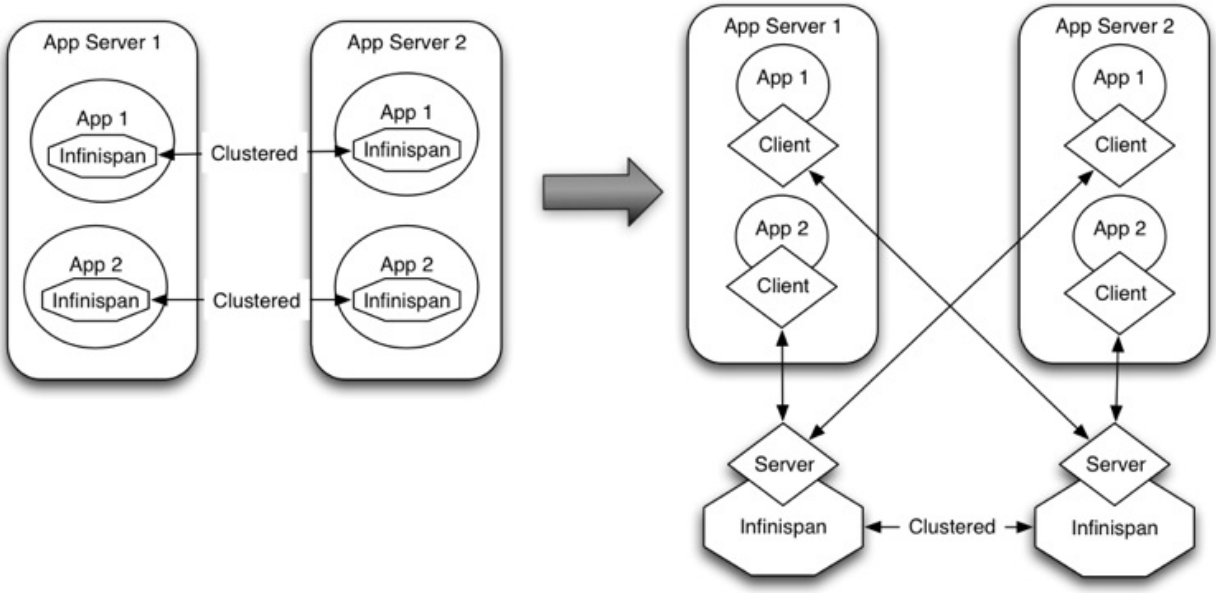
*これは、アプリケーションがこれらのプロセスが終了するまで **Red Hat Data Grid** にアクセスできないため、新規のアプリケーション層サーバーを起動することは、状態遷移による影響を受けません。これは、移動する状態が大きくなると、時間がかかる可能性があります。これは、アプリケーション層のサーバーのターンアラウンドかつ予測可能な起動時間を使用するエラスティック環境で望ましくありません。このような問題は、新しいアプリケーション層サーバーを起動することで、クライアントサーバーモードで **Red Hat Data Grid** にアクセスすることで解決できます。これは、バッキングデータグリッドサーバーに接続できる軽量クライアントを起動するだけです。再ハッシュや状態遷移が発生する必要がないため、サーバーの起動時間はより予測可能で、アプリケーション層のエラビリティーが重要な最新のクラウドベースのデプロイメントで非常に重要になります。*

図20.4 弾力性を実現



また、データストレージにアクセスする必要のある複数のアプリケーションを見つけることが一般的です。このような場合、これらの各アプリケーションごとに Red Hat Data Grid インスタンスをデプロイすることは可能ですが、保守が困難である可能性があります。ここでデータベースについて考えても、各アプリケーションとともにデータベースをデプロイしませんか？そのため、クライアントサーバーモードで Red Hat Data Grid をデプロイする場合は、アプリケーションの共有ストレージ層として機能する Red Hat Data Grid データグリッドノードのプールを維持できます。

図20.5 共有データストレージ



このように Red Hat Data Grid をデプロイすると、各層を個別に管理できます。たとえば、Red Hat Data Grid のグリッドノードをダウンすることなく、アプリケーションやアプリケーションサーバーをアップグレードすることができます。

20.2. 埋め込みモードを使用する理由

個別の Red Hat Data Grid サーバーモジュールについて学ぶ前に、すべての利点があるものの、クライアントサーバー Red Hat Data Grid には p2p よりも短所があります。p2p では、すべてのピアが相互に等しいため、p2p デプロイメントはクライアントサーバーサーバーよりも単純であるため、デプロイメントが簡素化されます。そのため、Red Hat Data Grid を初めて使用する場合には、クライアントサーバーと比較すると、p2p が容易になる可能性が高くなります。

リモート呼び出しのシリアライズおよびネットワークコストにより、クライアントサーバーの Red Hat Data Grid 要求が p2p 要求よりも長くなる可能性が高くなります。そのため、アプリケーションを設計する際に考慮すべき重要な要素です。たとえば、複製された Red Hat Data Grid キャッシュでは、特にデータサイズが大きくなると、クライアントサーバーモードで Red Hat Data Grid にアクセスするサーバー側のアプリケーションに接続する軽量 HTTP クライアントが、p2p モードで Red Hat Data Grid にアクセスするのがより高性能になる可能性があります。分散キャッシュでは、p2p デプロイメントではローカルで利用可能なデータがある保証がないため、大きな違いが大きくなるとは限りません。

アプリケーション層の弾力性が重要ではない場合や、サーバー側のアプリケーションが state-transfer-disabled で複製された Red Hat Data Grid キャッシュインスタンスにアクセスする場所は、Red Hat Data Grid の p2p デプロイメントがクライアントサーバーの場合よりも適しています。

20.3. サーバーモジュール

そこで、Red Hat Data Grid をクライアントサーバーモードでデプロイすると適切であれば、どのようなソリューションを利用できるのでしょうか？すべての Red Hat Data Grid サーバーモジュールは、サーバーバックエンドが埋め込み Red Hat Data Grid インスタンスを作成するのと同じパターンをベースにしています。複数のバックエンドを起動した場合には、クラスターを形成し、設定している場合はクラスターを共有/分散することができます。以下のサーバータイプは、主に受信接続を処理するために使用されるリスナーエンドポイントのタイプによって異なります。

利用可能なサーバーエンドポイントの概要を以下に示します。

- hot 336 サーバーモジュール： このモジュールは、クライアントが動的負荷分散とフェイルオーバー、およびスマートルーティングを実行できるようにする Red Hat Data Grid がサポートする Hotgitops バイナリープロトコル の実装です。
  - このプロトコルには さまざまなクライアント が存在します。
  - クライアントが Java を実行している場合は、動的負荷分散とフェイルオーバーを可能にするため、これは事実上のサーバーモジュールの選択になります。これは、これらがクラスター化されている限り、Hot336 クライアントが Hotgitops サーバーのトポロジーの変更を動的に検出できるため、新規ノードが参加または退出すると、クライアントが Hotgitops サーバートポロジービューを更新します。その上で、Hotfsprogs サーバーがディストリビューションで設定されていると、クライアントは特定のキーの場所を検出できるため、リクエストをスマートにルーティングできます。
  - 負荷分散とフェイルオーバーは、サーバーによって提供される情報を使用して HotTEMPLATES クライアント実装によって動的に提供されます。
- REST Server モジュール - WAR ファイルとして配布される REST サーバーをサーブレットコンテナーにデプロイし、Red Hat Data Grid を RESTful HTTP インターフェース経由でアクセスできるようにします。
  - これに接続するには、そこから任意の HTTP クライアントを使用し、非常に多くの言語やシステムで利用できる異なるクライアント実装を利用できます。
  - このモジュールは、クライアントとサーバー間で許可されるアクセスメソッドが HTTP ポートである環境に対して特に推奨されます。
  -

異なる Red Hat Data Grid REST サーバー間で負荷分散またはフェイルオーバーを行うクライアントは、mod_cluster などの標準の HTTP ロードバランサーを使用して実行できます。これらのロードバランサーがバックエンドでサーバーの静的ビューを維持し、新規サーバーを追加する必要がある場合は、ロードバランサーを手動で更新する必要があります。

- Memcached サーバーモジュール: このモジュールは、Red Hat Data Grid がサポートする Memcached テキストプロトコル の実装です。
  - これに接続するには、非常に多くの 既存の Memcached クライアント を使用できます。
  - Memcached サーバーとは対照的に、Red Hat Data Grid ベースの Memcached サーバーを実際にクラスター化できるため、クラスター全体で一貫性のあるハッシュアルゴリズムを使用してデータの複製や分散が可能です。そのため、このモジュールは、Memcached サーバーに保存されているデータにフェイルオーバー機能を提供するユーザーに特に関係しています。
  - 負荷分散とフェイルオーバーの観点では、サーバーアドレスの静的リスト（perl の Cache::Memcached など）を指定すると負荷分散やフェイルオーバーが可能なクライアントがいくつかありますが、サーバーの追加または削除には手動による介入が必要です。

20.4. 使用するプロトコル

適切なプロトコルの選択は、複数の要因によって異なります。

	Hot Rod	HTTP / REST	Memcached
トポロジー対応	Y	N	N
ハッシュ対応	Y	N	N
暗号化	Y	Y	N
認証	Y	Y	N
条件付き操作	Y	Y	Y
バルク操作	Y	N	N
トランザクション	N	N	N

	Hot Rod	HTTP / REST	Memcached
リスナー	Y	N	N
クエリー	Y	Y	N
実行	Y	N	N
クロスサイトフェイルオーバー	Y	N	N

20.5. 「HOTENT SERVER の使用」

Red Hat Data Grid Server ディストリビューションには、Hot Warehouse と呼ばれる Red Hat Data Grid のカスタムバイナリープロトコルを実装するサーバーモジュールが含まれています。このプロトコルは、他の既存のテキストベースのプロトコルと比較してクライアント/サーバーの対話を高速にし、クライアントが負荷分散、フェイルオーバー、データの場所操作に関してよりインテリジェントな決定を可能にするように設計されています。HotRod サーバーの設定方法および実行方法については、Red Hat Data Grid Server の ドキュメント を参照してください。

この非常に効率的な HotTEMPLATES プロトコル上で Red Hat Data Grid に接続するには、本章で説明したクライアントのいずれかを使用するか、Hibernate OGM などの高レベルのツールを使用できます。

20.6. HOT ROD プロトコル

以下の記事では、カスタム TCP クライアント/サーバーホットーティングプロトコルの各バージョンに関する詳細情報を説明します。

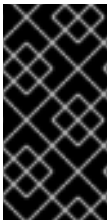
- *hotgitops Protocol 1.0*
- *hotgitops Protocol 1.1*
- *hotgitops Protocol 1.2*
- *hotgitops Protocol 1.3*

- *hotgitops Protocol 2.0*
- *hotgitops Protocol 2.1*
- *hotgitops Protocol 2.2*
- *hotgitops Protocol 2.3*
- *hotgitops Protocol 2.4*
- *hotgitops Protocol 2.5*
- *hotgitops Protocol 2.6*
- *hotgitops Protocol 2.7*
- *hotgitops Protocol 2.8*
- *hotgitops Protocol 2.9*

20.6.1. hotgitops Protocol 1.0

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 4.1.0.Final 以降実装されます。



重要

キーと値はすべて送信され、バイトアレイとして保存されます。hotgitops では、そのタイプに関しては想定されていません。

その他のタイプについての明確化：

- *vInt: Variable-length 整数は圧縮された正の整数として定義され、各バイトの上位ビットがより多くのバイトを読み取る必要があるかどうかを示します。7 つのビットが大きくなると、整数値に大幅なビットが高まるため、デコードが効率的になります。したがって、ゼロから 127 までの値は 1 バイトに保存され、128 から 16,383 の値は 2 バイトに保存されます。*

値	最初のバイト	2 バイト	3 バイト
0	00000000		
1	00000001		
2	00000010		
…			
127	01111111		
128	10000000	00000001	
129	10000001	00000001	
130	10000010	00000001	
…			
16,383	11111111	01111111	
16,384	10000000	10000000	00000001
16,385	10000001	10000000	00000001
…			

- *署名済み vInt: 上記の vInt は負の値をエンコードすることもできますが、終了値を小さくしても常に最大サイズ（5 バイト）を使用します。負の値のペイロードも小さくなるべく、署名された vInt は vInt エンコーディングに加えて ZigZag エンコーディングを使用します。詳細は、こちらを参照してください。*

- *VLong: vInt と同様の未署名の変数長の長い値を参照し、長い値に適用されます。1 バイトから 9 バイトの長さ。*

- ***string : 文字列は常に UTF-8 エンコーディングを使用して表されます。***

20.6.1.1. リクエストヘッダー

リクエストのヘッダーは以下で構成されています。

表20.1 要求ヘッダー

フィールド名	サイズ	値
magic	1バイト	0xA0 = request
メッセージ ID	vLong	応答にコピーされるメッセージの ID。これにより、Hot336 クライアントが非同期的にプロトコルを実装できます。
Version	1バイト	hotgitops サーバーバージョン。この場合、これは 10 です。
opcode	1バイト	Request operation code: 0x01 = put(since 1.0) 0x03 = get(since 1.0) 0x05 = putIfAbsent(since 1.0) 0x07 = replace(since 1.0) 0x09 = replaceIfUnmodified(since 1.0) 0x0B = remove(since 1.0) 0x0 D = removeIfUnmodified(since 1.0) 0x0F = containsKey(since 1.0) 0x11 = getWithVersion(since 1.0) 0x13 = clear(since 1.0) 0x15 = stats(since 1.0) 0x17 = ping(since 1.0) 0x19 = bulkGet(since 1.2) 0x1B = getWithMetadata(since 1.2) 0x1D = bulkGetKeys(since 1.2) 0x1F = query(since 1.3) 0x21 = authMechList(since 2.0) 0x23 = auth(since 2.0) 0x25 = addClientListener(since 2.0) 0x27 = removeClientListener(since 2.0) 0x29 = size(since 2.0) 0x2B = exec(since 2.1) 0x2D = putAll(since 2.1) 0x2F = getAll(since 2.1) 0x31 = iterationStart(since 2.3) 0x33 = iterationNext(since 2.3) 0 x35 = iterationEnd(since 2.3) 0x37 = getStream（以降 2.6） 0x39 = putStream（2.6 以降）

フィールド名	サイズ	値
キャッシュ名の長さ	vInt	キャッシュ名の長さ。渡された長さが 0（キャッシュ名なし）の場合、操作はデフォルトのキャッシュと対話します。
キャッシュ名	string	動作させるキャッシュの名前。この名前は、Red Hat Data Grid 設定ファイルの事前に定義されたキャッシュの名前と一致している必要があります。
Flags	vInt	システムに渡されるフラグを表す変数長番号。各フラグはビットで表されます。このフィールドは変数の長さとして送信されるため、バイトで最も大きなビットを使用して、より多くのバイトを読み取る必要があるかどうかを判断するため、このビットはフラグを表します。このモデルを使用すると、フラグを短いスペースに統合できます。各フラグの現在の値は以下の通りです。 0x0001 = force return previous value
クライアントのインテリジェンス	1 バイト	このバイトヒントは、クライアント機能のサーバーをヒントします。 0x01 = basic client（クラスターまたはハッシュ情報ではない） 0x02 = topology-aware クライアント、クラスター情報に関心のある 0x03 = hash-distribution-aware クライアント 0x03 = hash-distribution-aware クライアント（クラスターおよびハッシュ情報の両方に関心がある）
トポロジー ID	vInt	このフィールドは、クライアントの最後の既知のビューを表します。基本的なクライアントは、このフィールドには 0 のみを送信します。トポロジー対応または hash-distribution 対応のクライアントは、現在のビュー ID でサーバーから応答を受け取るまで 0 を送信します。その後、応答で新しいビュー ID を受け取るまで、ビュー ID を送信する必要があります。
トランザクションタイプ	1 バイト	これは、以下のよく知られたトランザクションタイプの 1 つが含まれる 1 バイトフィールドです（このバージョンのプロトコルでは、サポートされる唯一のトランザクションタイプは 0）。 0 = Non-transactional 呼び出し、またはクライアントはトランザクションをサポートしません。後続の TX_ID フィールドは省略されます。 1 = x/Open XA トランザクション ID(XID)。これはよく知られており、固定サイズの形式です。
トランザクション ID	バイト配列	この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。その長さはトランザクションタイプにより決定されます。トランザクションタイプが 0 の場合は、トランザクション ID は存在しません。

20.6.1.2. レスポンスヘッダー

応答のヘッダーは以下で構成されています。

表20.2 応答ヘッダー

フィールド名	サイズ	値
magic	1バイト	0xA1 = response
メッセージ ID	vLong	メッセージの ID。応答が送信されるリクエストと一致します。
opcode	1バイト	response operation code: 0x02 = put(since 1.0) 0x04 = get(since 1.0) 0x06 = putIfAbsent(since 1.0) 0x08 = replace(since 1.0) 0x0A = replaceIfUnmodified(since 1.0) 0x0C = remove(since 1.0) 0x 0e = removeIfUnmodified(since 1.0) 0x10 = containsKey(since 1.0) 0x12 = getWithVersion(since 1.0) 0x14 = clear(since 1.0) 0x16 = stats(since 1.0) 0x18 = ping(since 1.0) 0x1A = bulkGet(since 1.0) 0x1C = getWithMetadata(since 1.2) 0x1E = bulkGetKeys(since 1.2) 0x20 = query(since 1.3) 0x22 = authMechList(since 2.0) 0x24 = auth(since 2.0) 0x26 = addClientListener(since 2.0) 0x28 = removeClientListener(since 2.0) 0x2A = size(since 2.0) 0x2C = exec(since 2.1) 0x2E = putAll(since 2.1) 0x30 = getAll(since 2.1) 0x32 = iterationStart(since 2.3) 0x34 = iterationNext(since 2.3) 0 x36 = iterationEnd(since 2.3) 0x38 = getStream(since 2.6) 0x3A = putStream(since 2.6) 0x50 = error(since 1.0)
ステータス	1バイト	status of the response, possible values: 0x00 = No error 0x01 = Not put/removed/replaced 0x02 = Key does not exist 0x81 = Invalid magic or message id 0x82 = Unknown command 0x83 = Unknown version 0x84 = Request parsing error 0x85 = Server Error 0x86 = Command timed out
トポロジー変更マーカー	string	これは、応答の前にトポロジーの変更情報が含まれるかどうかを示すマーカーバイトです。トポロジーが変更されないと、このバイトの内容は 0 になります。トポロジーが変更されると、その内容は 1 になります。

注意

0x8 ... で始まる例外のステータス応答は、エラーメッセージの長さ（vInt として）およびエラーメッセージ自体を String として続きます。

20.6.1.3. トポロジー変更ヘッダー

以下のセクションでは、クラスターまたはビューのフォーマットが変更されている場合に、応答ヘッダーがトポロジー対応またはハッシュディストリビューション対応のクライアントを検索する方法を説明します。現在のトポロジー ID とクライアントが送信された 1 つに基づいて新しいトポロジーを送信するかどうかを決定するサーバーであることに注意してください。これが異なる場合は、新しいトポロジーを送り返します。

20.6.1.4. Topology-Aware クライアントトポロジー変更ヘッダー

これは、トポロジーの変更が返信される際に応答ヘッダーとして受信されるトポロジー対応のクライアントです。

フィールド名	サイズ	値
トポロジー変更マーカーのある応答ヘッダー	variable	先のセクションを参照してください。
トポロジー ID	vInt	トポロジー ID
トポロジーの num サーバー	vInt	クラスター内で稼働している Hotでサーバーの数。これは、これらのノードの一部のみが HotTEMPLATES サーバーを実行している場合に、クラスター全体のサブセットになる可能性があります。
M1: ホスト/IP 長	vInt	Hotgitops クライアントがアクセスするために使用できる各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。ここで変数の長さを使用すると、ホスト名、IPv4 アドレス、および IPv6 アドレスに対応できます。
M1: ホスト/IP アドレス	string	Hot336 クライアントがアクセスに使用できる個別のクラスターメンバーのホスト名または IP アドレスが含まれる文字列。
M1: ポート	2 バイト（未署名）	Hotgitops クライアントがこのクラスターメンバーとの通信に使用できるポート。
m2: ホスト/IP の長さ	vInt	

フィールド名	サイズ	値
m2: ホスト/IP アドレス	string	
m2: ポート	2 バイト（未署名）	
...etc		

20.6.1.5. Distribution-Aware クライアントトポロジー変更ヘッダー

これは、トポロジーの変更が返信される際に応答ヘッダーとして受信される hash-distribution 対応のクライアントです。

フィールド名	サイズ	値
トポロジー変更マーカーのある応答ヘッダー	variable	先のセクションを参照してください。
トポロジー ID	vInt	トポロジー ID
num Key Owners	2 バイト（未署名）	各 Red Hat Data Grid 分散キーのコピーをグローバルに設定した数
ハッシュ関数バージョン	1 バイト	使用中の特定のハッシュ関数を参照するハッシュ関数のバージョン。詳細は「 Hotgitops hash functions 」を参照してください。
ハッシュスペースサイズ	vInt	ハッシュコード生成に関連するすべてのモジュールの Red Hat Data Grid によって使用されるモジュール。キーに正しいハッシュ計算を適用するには、クライアントでこの情報が必要になる可能性があります。
トポロジーの num サーバー	vInt	クラスター内で稼働している Red Hat Data Grid Hotgitops サーバーの数。これは、これらのノードの一部のみが HotTEMPLATES サーバーを実行している場合に、クラスター全体のサブセットになる可能性があります。
M1: ホスト/IP 長	vInt	Hotgitops クライアントがアクセスするために使用できる各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。ここで変数の長さを使用すると、ホスト名、IPv4 アドレス、および IPv6 アドレスに対応できます。

フィールド名	サイズ	値
M1: ホスト/IP アドレス	string	Hot336 クライアントがアクセスに使用できる個別のクラスターメンバーのホスト名または IP アドレスが含まれる文字列。
M1: ポート	2 バイト（未署名）	Hotgitops クライアントがこのクラスターメンバーと組み合わせるために使用できるポート。
M1: Hashcode	4 バイト	ユーザーが CSA を適用しているクラスターメンバーがどのクラスターメンバーであるかをインデントできるクラスターメンバーのハッシュコードを表す 32 ビット整数。
m2: ホスト/IP の長さ	vInt	
m2: ホスト/IP アドレス	string	
m2: ポート	2 バイト（未署名）	
m2: Hashcode	4 バイト	
...etc		

ハッシュヘッダーはサーバーで使用される一貫性のあるハッシュアルゴリズムに依存し、キャッシュが対話する要素であるため、hash-distribution 対応ヘッダーは特定のキャッシュをターゲットとする操作にのみ返すことができることに注意してください。現在 ping コマンドはキャッシュをターゲットにしません（ISPN-424に従って変更されます）。ハッシュトポロジー対応のクライアント設定で ping コマンドを呼び出すと、「Num Key Owners」、「Hash Function Version」、「Hash Function Version」、「Hash Function Version」、個々のホストのハッシュコードはすべて 0 に設定されたハッシュディストリビューション対応ヘッダーを返します。また、ディストリビューションで設定されていないキャッシュを対象とする hash-topology 対応のクライアント設定を持つ操作への応答として、このタイプのヘッダーも返されます。

20.6.1.6. 操作

get(0x03)/Remove(0x0B)/ContainsKey(0x0F)/GetWithVersion(0x11)

一般的な要求形式：

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
キーの長さ	vInt	キーの長さ。vint のサイズは最大 5 バイトで、理論では Integer.MAX_VALUE よりも大きな数値を生成することができます。ただし、Java では Integer.MAX_VALUE を超える単一のアレイを作成できないため、プロトコルは vint アレイの長さを Integer.MAX_VALUE に制限します。
キー	バイト配列	値が要求されているキーが含まれるバイトアレイ。

応答を取得します(0x04)。

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success（キーの取得の場合） 0x02 = if key does not exist
値の長さ	vInt	成功した場合、値の長さ
値	バイト配列	成功した場合、要求された値。

応答を削除します(0x0C):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success（キーが削除された場合） 0x02 = if key does not exist
以前の値の長さ	vInt	以前の値フラグが要求で送信され、キーが削除されると、以前の値の長さが返されます。キーが存在しない場合は、値の長さは 0 になります。フラグが送信されていない場合、値の長さはありません。
以前の値	バイト配列	以前の値フラグが要求で送信され、キーが削除された場合、以前の値。

ContainsKey 応答(0x10):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success, if key exists 0x02 = if key does not exist

GetWithVersion response (0x12):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success（キーの取得の場合） 0x02 = if key does not exist
エントリーバージョン	8 バイト	既存のエントリーの変更の一意の値。このプロトコルは、entry_version 値が連続して行われるという義務はありません。更新ごとにキーレベルで一意となる必要があります。
値の長さ	vInt	成功した場合、値の長さ
値	バイト配列	成功した場合、要求された値。

BulkGet

要求(0x19):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
エントリー数	vInt	サーバーが返す Red Hat Data Grid エントリーの最大数（entry == key + 関連付けられた値）。CacheLoader.load(int)をサポートするために必要です。0 の場合、すべてのエントリーが返されます（CacheLoader.loadAll（）に必要）。

応答(0x20):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success, data follows
詳細表示	1バイト	複数のエントリーをストリームから読み取る必要があるかどうかを表す1つのバイト。そのため、1に設定すると、エントリーが後に続くことを意味しますが、0に設定された場合はストリームの最後となり、他のエントリーは読み取られません。BulkGetの詳細は、こちらを参照してください。
キー1長	vInt	キーの長さ
キー1	バイト配列	取得されるキー
値1の長さ	vInt	値の長さ
値1	バイト配列	取得した値
詳細表示	1バイト	
キー2長	vInt	
キー2	バイト配列	
値2の長さ	vInt	
値2	バイト配列	
...		

Put (0x01)/PutIfAbsent (0x05)/Replace (0x07)

一般的な要求形式：

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

フィールド名	サイズ	値
キーの長さ	vInt	キーの長さ。vint のサイズは最大 5 バイトで、理論では Integer.MAX_VALUE よりも大きな数値を生成することができます。ただし、Java では Integer.MAX_VALUE を超える単一のアレイを作成できないため、プロトコルは vint アレイの長さを Integer.MAX_VALUE に制限します。
キー	バイト配列	値が要求されているキーが含まれるバイトアレイ。
有効期間	vInt	エントリーの有効期間が許可される秒数。秒数が 30 日を超える場合、この秒数は UNIX 時間として扱われるため、1/1/1970 からの秒数を表します。0 に設定すると、ライフスパンは無制限になります。
最大 ID	vInt	キャッシュからエビクトされる前に、エントリーをアイドル状態でいられる秒数。0 の場合は、最大アイドル時間がありません。
値の長さ	vInt	値の長さ
値	byte-array	保存する値

応答を追加します(0x02)。

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1 バイト	0x00 = success（保存されている場合）
以前の値の長さ	vInt	以前の値フラグが要求で送信され、キーが配置されている場合、以前の値の長さが返されます。キーが存在しない場合は、値の長さは 0 になります。フラグが送信されていない場合、値の長さはありません。
以前の値	バイト配列	以前の値フラグが要求で送信され、キーが配置された場合、以前の値。

応答(0x08)を置き換えます。

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

フィールド名	サイズ	値
応答のステータス	1バイト	0x00 = success（保存された場合は 0x01 = if store does not exist）
以前の値の長さ	vInt	リクエストで以前の値フラグを強制的に返すと、前の値の長さが返されます。キーが存在しない場合は、値の長さは 0 になります。フラグが送信されていない場合、値の長さはありません。
以前の値	バイト配列	リクエストで以前の値フラグが送信され、キーが置き換えられた場合に、以前の値が返されます。

PutIfAbsent 応答(0x06):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success（保存された場合は 0x01 = if store was exist）
以前の値の長さ	vInt	リクエストで以前の値フラグを強制的に返すと、前の値の長さが返されます。キーが存在しない場合は、値の長さは 0 になります。フラグが送信されていない場合、値の長さはありません。
以前の値	バイト配列	リクエストで以前の値フラグが送信され、キーが置き換えられた場合に、以前の値が返されます。

ReplaceIfUnmodified

要求(0x09):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
キーの長さ	vInt	キーの長さ。vint のサイズは最大 5 バイトで、理論では Integer.MAX_VALUE よりも大きな数値を生成することができます。ただし、Java では Integer.MAX_VALUE を超える単一のアレイを作成できないため、プロトコルは vint アレイの長さを Integer.MAX_VALUE に制限します。

フィールド名	サイズ	値
キー	バイト配列	値が要求されているキーが含まれるバイトアレイ。
有効期間	vInt	エントリーの有効期間が許可される秒数。秒数が 30 日を超える場合、この秒数は UNIX 時間として扱われるため、1/1/1970 からの秒数を表します。0 に設定すると、ライフスパンは無制限になります。
最大 ID	vInt	キャッシュからエビクトされる前に、エントリーをアイドル状態でいられる秒数。0 の場合は、最大アイドル時間がありません。
エントリーバージョン	8 バイト	GetWithVersion 操作によって返される値を使用します。
値の長さ	vInt	値の長さ
値	byte-array	保存する値

応答(0x0A):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1 バイト	0x00 = success, if replaced 0x01 = if replace because key was changed 0x02 = if replaced because because key does not exist because
以前の値の長さ	vInt	リクエストで以前の値フラグを強制的に返すと、前の値の長さが返されます。キーが存在しない場合は、値の長さは 0 になります。フラグが送信されていない場合、値の長さはありません。
以前の値	バイト配列	リクエストで以前の値フラグが送信され、キーが置き換えられた場合に、以前の値が返されます。

RemoveIfUnmodified

要求(0x0D):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

フィールド名	サイズ	値
キーの長さ	vInt	キーの長さ。vint のサイズは最大 5 バイトで、理論では Integer.MAX_VALUE よりも大きな数値を生成することができます。ただし、Java では Integer.MAX_VALUE を超える単一のアレイを作成できないため、プロトコルは vint アレイの長さを Integer.MAX_VALUE に制限します。
キー	バイト配列	値が要求されているキーが含まれるバイトアレイ。
エントリーバージョン	8 バイト	GetWithMetadata 操作によって返される値を使用します。

応答(0x0E):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1 バイト	0x00 = success（削除されている場合） 0x01 = if remove did not occurred because key was changed 0x02 = if not removed because key does not exist
以前の値の長さ	vInt	リクエストで以前の値フラグを強制的に返すと、前の値の長さが返されます。キーが存在しない場合は、値の長さは 0 になります。フラグが送信されていない場合、値の長さはありません。
以前の値	バイト配列	以前の値フラグが要求で送信され、キーが削除された場合、以前の値。

消去

要求(0x13):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

応答(0x14):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success（消去されている場合）

PutAll

すべてのキー値エントリーをキャッシュに同時に配置する一括操作。

要求(0x2D):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
有効期間	vInt	提供されたエントリーが存続できる秒数。秒数が 30 日を超える場合、この秒数は UNIX 時間として扱われるため、1/1/1970 からの秒数を表します。0 に設定すると、ライフスパンは無制限になります。
最大 ID	vInt	キャッシュからエビクトされる前に、各エントリーをアイドル状態でいられる秒数。0 の場合は、最大アイドル時間がありません。
エントリー数	vInt	挿入されるエントリーの数
キー 1 長	vInt	キーの長さ
キー 1	バイト配列	取得されるキー
値 1 の長さ	vInt	値の長さ
値 1	バイト配列	取得した値
キー 2 長	vInt	
キー 2	バイト配列	
値 2 の長さ	vInt	
値 2	バイト配列	

フィールド名	サイズ	値
... エントリー数に達するまで継続する		

応答(0x2E):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success（すべてが配置されている場合）

GetAll

指定されたキーのセットにマップするすべてのエントリーを取得する一括操作。

要求(0x2F):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
キー数	vInt	エントリーを検索するキーの数
キー1長	vInt	キーの長さ
キー1	バイト配列	取得されるキー
キー2長	vInt	
キー2	バイト配列	
... キー数に達するまで継続される		

応答(0x30):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	
エントリー数	vInt	返されるエントリーの数
キー1長	vInt	キーの長さ
キー1	バイト配列	取得されるキー
値1の長さ	vInt	値の長さ
値1	バイト配列	取得した値
キー2長	vInt	
キー2	バイト配列	
値2の長さ	vInt	
値2	バイト配列	
... エントリー数に達するまで継続する		0x00 = success。get returned sucessfully

Stats

利用可能なすべての統計の概要を返します。統計で返される各統計では、名前と値はどちらもString UTF-8形式で返されます。サポートされる統計は以下のとおりです。

名前	説明
timeSinceStart	Hotgitopsの開始からの秒数。
currentNumberOfEntries	「Hoting servers」サーバーに現在含まれるエントリーの数。
totalNumberOfEntries	Hotgitopsサーバーに保存されているエントリーの数。

名前	説明
stores	put 操作の数。
retrievals	get 操作の数。
Hits	ヒット数。
misses	get misses の数。
removeHits	削除ヒット数。
removeMisses	削除のミスの数。

要求(0x15):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

応答(0x16):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success（統計が取得された場合）
統計数	vInt	返された個々の統計の数。
名前1の長さ	vInt	名前付き統計の長さ。
Name 1	string	統計名が含まれる文字列。
値1の長さ	vInt	value フィールドの長さ。
値1	string	統計値が含まれる文字列。
名前2の長さ	vInt	
名前2	string	

フィールド名	サイズ	値
値 2 の長さ	vInt	
値 2	文字列	
...etc		

Ping

サーバーが利用可能かどうかを確認するためのアプリケーションレベルの要求。

要求(0x17):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

応答(0x18):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1 バイト	0x00 = success（エラーがない場合）

エラー処理

エラー応答(0x50)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1 バイト	0x8x = エラー応答コード
エラーメッセージの長さ	vInt	エラーメッセージの長さ

フィールド名	サイズ	値
エラーメッセージ	string	エラーメッセージ0x84 の場合、このエラーフィールドには Hot336 サーバーでサポートされる最新バージョンが含まれます。長さは合計本文の長さで定義されます。

複数取得操作

マルチget操作は、1 つのキーを要求せずにキーのセットを要求する get 操作の形式です。Hotgitops プロトコルにはこのような操作は含まれませんが、remote HotTEMPLATES クライアントは、個別の get リクエストパッシングまたはパイプのいずれかを使用して、このタイプの操作を簡単に実装できます。もう 1 つの可能性として、リモートクライアントが非同期またはノンブロッキング取得リクエストを使用することができます。たとえば、クライアントが N キーが必要な場合は、N async get requests を送信し、すべての応答を待つことができます。最後に、マルチットは一括取得操作と混同しないようにしてください。一括取得では、すべてまたは多数のキーが取得されますが、クライアントはどのキーを取得するかを認識しませんが、マルチget では、クライアントはどのキーを取得するかを定義します。

20.6.1.7. 例： 要求の変更

- ***コード化されたリクエスト***

Byte	0	1	2	3	4	5	6	7
8	0xA0	0x09	0x41	0x01	0x07	0x4D ('M')	0x79 ('y')	0x43 ('C')
16	0x61 ('a')	0x63 ('c')	0x68 ('h')	0x65 ('e')	0x00	0x03	0x00	0x00
24	0x00	0x05	0x48 ('H')	0x65 ('e')	0x6C ('l')	0x6C ('l')	0x6F ('o')	0x00
32	0x00	0x05	0x57 ('W')	0x6F ('o')	0x72 ('r')	0x6C ('l')	0x64 ('d')	

- ***フィールドの説明***

フィールド名	値	フィールド名	値
マジック(0)	0xA0	Message Id(1)	0x09
バージョン(2)	0x41	opcode(3)	0x01

フィールド名	値	フィールド名	値
キャッシュ名の長さ(4)	0x07	Cache name(5-11)	'MyCache'
フラグ(12)	0x00	クライアントの Intelligence(13)	0x03
トポロジー ID(14)	0x00	トランザクションタイプ (15)	0x00
トランザクション ID(16)	0x00	キーフィールド長(17)	0x05
キー(18 - 22)	'hello'	lifespan(23)	0x00
最大アイドル時間(24)	0x00	値フィールドの長さ(25)	0x05
値(26-30)	'World'		

- ***コード化された応答***

Byte	0	1	2	3	4	5	6	7
8	0xA1	0x09	0x01	0x00	0x00			

- ***フィールドのプランニング***

フィールド名	値	フィールド名	値
マジック(0)	0xA1	Message Id(1)	0x09
opcode(2)	0x01	ステータス(3)	0x00
トポロジー変更マーカー (4)	0x00		

20.6.2. hotgitops Protocol 1.1

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 5.1.0.FINAL 以降実装されます。

20.6.2.1. リクエストヘッダー

ヘッダーの version フィールドは 11 に更新されました。

20.6.2.2. Distribution-Aware クライアントトポロジー変更ヘッダー

1.1 向けに更新

このセクションは、仮想ノードを有効にして分散キャッシュと通信する際により効率的になるように変更されました。

これは、トポロジーの変更が返信される際に応答ヘッダーとして受信される hash-distribution 対応のクライアントです。

フィールド名	サイズ	値
トポロジー変更マーカーのある応答ヘッダー	variable	先のセクションを参照してください。
トポロジー ID	vInt	トポロジー ID
num Key Owners	2 バイト（未署名）	各 Red Hat Data Grid 分散キーのコピーをグローバルに設定した数
ハッシュ関数バージョン	1 バイト	使用中の特定のハッシュ関数を参照するハッシュ関数のバージョン。詳細は「 Hotgitops hash functions 」を参照してください。
ハッシュスペースサイズ	vInt	ハッシュコード生成に関連するすべてのモジュールの Red Hat Data Grid によって使用されるモジュール。キーに正しいハッシュ計算を適用するには、クライアントでこの情報が必要になる可能性があります。
トポロジーの num サーバー	vInt	クラスター内で稼働している Hotでサーバーの数。これは、これらのノードの一部のみが HotTEMPLATES サーバーを実行している場合に、クラスター全体のサブセットになる可能性があります。
num Virtual Nodes Owners	vInt	設定された仮想ノード数を表すプロトコルのバージョン 1.1 に追加されます。仮想ノードが設定されていない場合や、キャッシュがディストリビューションで設定されていない場合、このフィールドには 0 が含まれます。

フィールド名	サイズ	値
M1: ホスト/IP 長	vInt	Hotgitops クライアントがアクセスするために使用できる各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。ここで変数の長さを使用すると、ホスト名、IPv4 アドレス、および IPv6 アドレスに対応できます。
M1: ホスト/IP アドレス	string	Hot336 クライアントがアクセスに使用できる個別のクラスターメンバーのホスト名または IP アドレスが含まれる文字列。
M1: ポート	2 バイト（未署名）	Hotgitops クライアントがこのクラスターメンバーと組み合わせるために使用できるポート。
M1: Hashcode	4 バイト	ユーザーが CSA を適用しているクラスターメンバーがどのクラスターメンバーであるかをインデントできるクラスターメンバーのハッシュコードを表す 32 ビット整数。
m2: ホスト/IP の長さ	vInt	
m2: ホスト/IP アドレス	string	
m2: ポート	2 バイト（未署名）	
m2: Hashcode	4 バイト	
...etc		

20.6.2.3. サーバーノードのハッシュコード計算

仮想ノードのサポートを追加すると、帯域幅によりバージョン 1.0 が非現実的になり、クラスター内の全仮想ノードのハッシュコードを返す必要があります（この数は、数百万に簡単になる可能性があります）。そのため、Hotgitops プロトコルのバージョン 1.1 では、クライアントには各サーバーのベースハッシュ ID またはハッシュコードが指定されるため、各サーバーの実際のハッシュ位置を仮想ノードが設定されていない状態で、または両方計算する必要があります。ノードのハッシュコードの計算を試みる際に、ルールクライアントに従う必要があります。

1\.仮想ノードが無効 の場合： クライアントがサーバーのベースハッシュコードを受信したら、ハッシュ wheel の正確な位置を見つけるために、それらを正規化する必要があります。正規化のプロセスには、ベースハッシュコードをハッシュ関数に渡して、負の値を避けるために小さな計算を行う必要があります。生成される数字は、ハッシュ wheel におけるノードの位置になります。

```
public static int getNormalizedHash(int nodeBaseHashCode, Hash hashFct) {
```

```
   return hashFct.hash(nodeBaseHashCode) & Integer.MAX_VALUE; // make sure no negative numbers are involved.
}
```

2\.仮想ノードが有効 な場合： この場合、各ノードは N つの異なる仮想ノードを表し、各仮想ノードのハッシュコードを計算するには、0 から N-1 までの数字を取り、以下のロジックを適用する必要があります。

- 0 が id である仮想ノードの場合は、前のセクションで示したように、ノードのハッシュコードの取得に使用する手法を使用します。
- 1 から N-1 id の仮想ノードの場合は、以下のロジックを実行します。

```
public static int virtualNodeHashCode(int nodeBaseHashCode, int id, Hash hashFct) {
   int virtualNodeBaseHashCode = id;
   virtualNodeBaseHashCode = 31 * virtualNodeBaseHashCode + nodeBaseHashCode;
   return getNormalizedHash(virtualNodeBaseHashCode, hashFct);
}
```

20.6.3. hotgitops Protocol 1.2

INFINISPAN のバージョン

このバージョンのプロトコルは、Red Hat Data Grid 5.2.0.Final 以降実装されます。Red Hat Data Grid 5.3.0 以降、HotRod は SSL による暗号化をサポートします。ただし、これはトランスポートにのみ影響するため、プロトコルのバージョン番号はインクリメントされていません。

20.6.3.1. リクエストヘッダー

ヘッダーの バージョン フィールドは 12 に更新されます。

新しいリクエスト操作コードが追加されました。

- 0x1B = getWithMetadata request
- 0x1D = bulkKeysGet request

新しい 2 つのフラグも追加されました。

- *0x0002 = キャッシュレベルで設定されたデフォルトのライフスパンを使用する*
- *0x0004 = キャッシュレベルに設定されたデフォルトの最大アイドルを使用する*

20.6.3.2. レスポンスヘッダー

新しいレスポンス操作コードが新たに追加されました。

- *0x1C = getWithMetadata response*
- *0x1E = bulkKeysGet response*

20.6.3.3. 操作

GetWithMetadata

要求(0x1B):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
キーの長さ	vInt	キーの長さ。vint のサイズは最大 5 バイトで、理論では Integer.MAX_VALUE よりも大きな数値を生成することができます。ただし、Java では Integer.MAX_VALUE を超える単一のアレイを作成できないため、プロトコルは vint アレイの長さを Integer.MAX_VALUE に制限します。
キー	バイト配列	値が要求されているキーが含まれるバイトアレイ。

応答(0x1C):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success（キーの取得の場合） 0x02 = if key does not exist
フラグ	1バイト	応答に有効期限情報が含まれるかどうかを示すフラグ。フラグの値は、INFINITE_LIFESPAN(0x01)と **INFINITE_MAXIDLE(0x02)の間のビット単位の OR 操作として取得されます**。
Created	Long	（オプション）エントリーがサーバー上で作成された時点のタイムスタンプを表す長期。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合のみ返されます。
有効期間	vInt	（オプション）エントリーの有効期間を表す vInt を秒単位で表示します。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合のみ返されます。
LastUsed	Long	（オプション）エントリーがサーバー上で最後にアクセスされた時点のタイムスタンプを表す長期。この値は、フラグの **INFINITE_MAXIDLE** ビットが設定されていない場合のみ返されます。
MaxIdle	vInt	（オプション）エントリーの maxIdle を表す vInt（秒単位）。この値は、フラグの **INFINITE_MAXIDLE** ビットが設定されていない場合のみ返されます。
エントリーバージョン	8 バイト	既存のエントリーの変更の一意の値。このプロトコルは、entry_version 値が連続して行われるという義務はありません。更新ごとにキーレベルで一意となる必要があります。
値の長さ	vInt	成功した場合、値の長さ
値	バイト配列	成功した場合、要求された値。

BulkKeysGet

要求(0x1D):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

フィールド名	サイズ	値
スコープ	vInt	0 = デフォルトスコープ - このスコープは RemoteCache.keySet（）メソッドによって使用されます。リモートキャッシュが分散キャッシュの場合、サーバーはストリーム操作を起動し、すべてのノードからキーを取得します。（トポロジー対応型クライアントでは、要求をクラスター内のいずれかのノードに負荷分散する可能性があることを覚えておいてください）。それ以外の場合は、リクエストを受信するサーバーに対してローカルにあるキャッシュインスタンスからキーを取得します（キーはレプリケートされたキャッシュ内のすべてのノードで同じである必要があります）。 1 = グローバルスコープ - このスコープは Default Scope と同じように動作します。 2 = local Scope - リモートキャッシュが分散キャッシュである場合、サーバーはすべてのノードからキーを取得するためにストリーム操作を開始しません。代わりに、リクエストを受信するサーバーにローカルにあるキャッシュインスタンスからローカルなキーのみを取得します。

応答(0x1E):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success, data follows
詳細表示	1バイト	ストリームから他のキーを読み取る必要があるかどうかを表す1つのバイト。そのため、1に設定すると、エントリーが後に続くことを意味しますが、0 に設定された場合はストリームの最後となり、他のエントリーは読み取られません。BulkGet の詳細は、こちらを参照してください。
キー1長	vInt	キーの長さ
キー1	バイト配列	取得されるキー
詳細表示	1バイト	
キー2長	vInt	
キー2	バイト配列	
...		

20.6.4. hotgitops Protocol 1.3

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 6.0.0.Final 以降実装されます。

20.6.4.1. リクエストヘッダー

ヘッダーの version フィールドは 13 に更新されました。

新しいリクエスト操作コードが追加されました。

- *0x1F = クエリーリクエスト*

20.6.4.2. レスポンスヘッダー

新しいレスポンス操作コードが追加されました。

- *0x20 = クエリー応答*

20.6.4.3. 操作

クエリー

要求(0x1F):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
クエリーの長さ	vInt	protobuf でエンコードされたクエリーオブジェクトの長さ
クエリー	バイト配列	前のフィールドで指定された長さを持つ protobuf でエンコードされたクエリーオブジェクトを含むバイト配列。

応答(0x20):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答ペイロードの長さ	vInt	protobuf でエンコードされた応答オブジェクトの長さ
応答ペイロード	バイト配列	前のフィールドで指定された長さを持つ protobuf でエンコードされた応答オブジェクトを含むバイト配列。

Infinispan 6.0 では、クエリーおよび応答オブジェクトは、protobuf メッセージタイプ 'org.infinispan.client.hotrod.impl.query.QueryRequest' と 'org.infinispan.client.hotrod.impl.query/remote-query-client/src/main/resources/org/infinispan/query/remote/client/query.proto で指定されます。これらの定義は今後の Infinispan バージョンで変更される可能性がありますが、これらの進化は後方互換性を維持する（ここで定義されるルールに準拠）している限り、これに対応するため、新しい Hot5115 プロトコルバージョンが導入されました。

20.6.5. hotgitops Protocol 2.0

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 7.0.0.Final 以降実装されます。

20.6.5.1. リクエストヘッダー

要求ヘッダーには使用していないため、トランザクションタイプとトランザクション ID 要素が含まれなくなり、ping や stats コマンドなど、意味のない操作がいくつかあります。トランザクションが実装されると、リクエストヘッダーに必要な変更が含まれるプロトコルバージョンが起動します。

ヘッダーの version フィールドは 20 に更新されました。

新しいフラグが 2 つ追加されました。

- *0x0008 = 設定されたキャッシュローダーからの読み込みをスキップします。*
- *0x0010 = 操作はインデックス化をスキップします。クエリーモジュールがキャッシュに対して有効になっている場合にのみ関連します。*

以下の新しいリクエスト操作コードが追加されました。

- 0x21 = auth mech list request
- 0x23 = 認証要求
- 0x25 = クライアントリモートイベントリスナー要求の追加
- 0x27 = クライアントリモートイベントリスナー要求の削除
- 0x29 = size request

20.6.5.2. レスポンスヘッダー

以下の新しいレスポンス操作コードが追加されました。

- 0x22 = auth mech list response
- 0x24 = auth mech response
- 0x26 = クライアントリモートイベントリスナー応答の追加
- 0x28 = クライアントリモートイベントリスナーの応答の削除
- 0x2A = size response

クライアントによりインテリジェントな決定を可能にするために、新しいエラーコードが 2 つ追加されました。特に、フェイルオーバーロジックは次の通りです。

- 0x87 = Node mayed.クライアントが応答としてこのエラーを受信すると、応答したノー

ドが 3 番目のノードに操作を送信するという問題があったことを意味します。通常、このエラーを返す要求は他のノードにフェイルオーバーする必要があります。

- *0x88 = Illegal lifecycle state。クライアントがこのエラーを応答として受け取ると、サーバー側のキャッシュまたはキャッシュマネージャーが停止または同様の状況のいずれかが停止しているため、要求で利用できないことを意味します。通常、このエラーを返す要求は他のノードにフェイルオーバーする必要があります。*

以下のコマンドの応答に対して調整が行われ、送信される情報を追跡する必要なしに応答のデコードをより適切に処理できるようになりました。より正確には、以前の値を解析する方法が変更され、コマンドの応答のステータスは以前の値に従うかどうかについて包含します。具体的には、以下のようになります。

- *成功すると、以前の値の後に応答が 0x03 ステータスコードを返します。*
- *PutIfAbsent 応答は、キーが存在し、応答に値が続くため、putIfAbsent 操作が失敗した場合にのみ 0x04 ステータスコードを返します。putIfAbsent が機能した場合、以前の値はなかったため、追加で何も返されません。*
- *応答を置き換えるのは、置き換えが発生した場合にのみ 0x03 ステータスコードを返し、応答の前の値または置き換えられた値が続きます。置き換えが行われなかった場合は、キャッシュエントリーが存在しないため、返すことができる以前の値はありません。*
- *ReplaceIfUnmodified は、変更が発生した場合にのみ 0x03 ステータスコードを返し、応答の前の値または置き換えられた値に従います。*
- *ReplaceIfUnmodified は、キーが変更された場合に置き換えられなかった場合に限り 0x04 ステータスコードを返し、変更後の値は応答で続きます。*
- *削除の発生時に 0x03 ステータスコードが返され、応答の前または削除された値が続きます。キーが存在しないために削除が発生しなかった場合、以前の値情報を送信する意味はありません。*
- *RemoveIfUnmodified は、削除が発生したときにのみ 0x03 ステータスコードを返し、応答の前の値または置き換えられた値に従います。*
- *RemoveIfUnmodified は、キーの変更が原因で削除が行われなかった場合のみ 0x04 ス*

テータスコードを返し、変更値は応答で続きます。

20.6.5.3. Distribution-Aware クライアントトポロジー変更ヘッダー

Infinispan 5.2 では、一貫性のあるハッシュ機能に基づいた仮想ノードが破棄され、代わりにセグメントベースの一貫性のあるハッシュが実装されました。Hotgitops クライアントがデータを可能な限り確実に見つけ出す機能を満たすために、Red Hat Data Grid は、セグメントベースの一貫したハッシュを、Hoting 1.x プロトコルに適合するように変換しました。バージョン 2.0 以降、完全に新しいディストリビューション対応トポロジー変更ヘッダーが実装されました。このヘッダーは、一貫性のあるハッシュに基づいて一貫したハッシュに基づいて、データの場所の保証を提供するものです。

フィールド名	サイズ	値
トポロジー変更マーカーのある応答ヘッダー	variable	
トポロジー ID	vInt	トポロジー ID
トポロジーの num サーバー	vInt	クラスター内で稼働している Red Hat Data Grid Hotgitops サーバーの数。これは、これらのノードの一部のみが HotTEMPLATES サーバーを実行している場合に、クラスター全体のサブセットになる可能性があります。
M1: ホスト/IP 長	vInt	Hotgitops クライアントがアクセスするために使用できる各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。ここで変数の長さを使用すると、ホスト名、IPv4 アドレス、および IPv6 アドレスに対応できます。
M1: ホスト/IP アドレス	string	Hot336 クライアントがアクセスに使用できる個別のクラスターメンバーのホスト名または IP アドレスが含まれる文字列。
M1: ポート	2 バイト（未署名）	Hotgitops クライアントがこのクラスターメンバーと組み合わせるために使用できるポート。
m2: ホスト/IP の長さ	vInt	
m2: ホスト/IP アドレス	string	
m2: ポート	2 バイト（未署名）	
...	...	

フィールド名	サイズ	値
ハッシュ関数バージョン	1バイト	使用中の特定のハッシュ関数を参照するハッシュ関数のバージョン。詳細は「 Hotgitops hash functions 」を参照してください。
トポロジーの num セグメント	vInt	トポロジー内のセグメントの合計数
セグメントの所有者数	1バイト	これには 0、1または 2 所有者のいずれかを指定できます。
最初の所有者のインデックス	vInt	すべてのノードの一覧を指定すると、この一覧にあるこの所有者の位置になります。これは、このセグメントの所有者の数が 1 または 2 の場合にのみ表示されます。
2 つ目の所有者のインデックス	vInt	すべてのノードの一覧を指定すると、この一覧にあるこの所有者の位置になります。これは、このセグメントの所有者の数が 2 の場合にのみ表示されます。

この情報を指定すると、Hotgitops クライアントはすべてのハッシュセグメントを再計算でき、各セグメントの所有者であるノードを検出できる必要があります。セグメントごとに所有者が 2 つ以上存在する可能性があっても、Hot pressure プロトコルは効率の理由から所有者の数を制限するものです。

20.6.5.4. 操作

認証メックリスト

要求(0x21):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

応答(0x22):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
mech count	vInt	メッションの数

フィールド名	サイズ	値
mech 1	string	IANA-registered 形式で SASL mech の名前を含む文字列（例：GSSAPI、CRAM-MD5 など）
mech 2	string	
...etc		

この操作の目的は、サーバーがサポートする有効な SASL 認証機械リストを取得することです。その後、クライアントは優先される mech で Authenticate 要求を発行する必要があります。

認証

要求(0x23):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
mech	string	認証用にクライアントが選択する mech の名前が含まれる文字列。連続呼び出しで空
応答の長さ	vInt	SASL クライアントの応答の長さ
応答データ	バイト配列	SASL クライアントの応答

応答(0x24):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
完了済み	byte	さらなる処理が必要な場合は 0、認証の完了の場合は 1
チャレンジの長さ	vInt	SASL サーバーのチャレンジの長さ
チャレンジデータ	バイト配列	SASL サーバーのチャレンジ

この操作の目的は、SASL を使用してサーバーに対してクライアントを認証することです。選択した mech に応じて、認証プロセスはマルチステップ操作である可能性があります。完了すると、接続は認証されます。

リモートイベントのクライアントリスナーの追加

要求(0x25):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
リスナー ID	バイト配列	リスナー識別子
状態を含める	byte	このバイトが **1** に設定されると、初めてキャッシュリスナーを追加するとき、またはリモートリスナーが登録されたノードがクラスター環境に変更されると、キャッシュされた状態はリモートクライアントに送信されます。有効にすると、状態はキャッシュエントリーで作成されたイベントとしてクライアントに送信されます。**0** に設定すると、リスナーの追加時に状態はクライアントに返されず、リスナーが登録されたノードが変更されると状態が生じます。
キー/値フィルターファクトリー名	string	このリスナーと使用するキー/値フィルターファクトリーのオプション名。ファクトリーはキー/値のフィルターインスタンスを作成するために使用されます。これにより、イベントが Hotgitops サーバーで直接フィルターされ、クライアントが関係しないイベントの送信を回避することができます。ファクトリーが使用されない場合、文字列の長さは **0** になります。
キー/値フィルターファクトリーパラメーターの数	byte	キー/値のフィルターファクトリーは、フィルターインスタンスの作成時に任意の数のパラメーターを取り、ファクトリーを使用して異なるフィルターインスタンスを動的に作成できるようにします。この count フィールドは、ファクトリーに渡されるパラメーターの数を示します。ファクトリー名が指定されていない場合、このフィールドはリクエストに表示されません。
キー/値フィルターファクトリーパラメーター 1	バイト配列	最初のキー/値フィルターファクトリーパラメーター
キー/値フィルターファクトリーパラメーター 2	バイト配列	2 つ目のキー/値フィルターファクトリーパラメーター
...		

フィールド名	サイズ	値
コンバーターファクトリー名	string	このリスナーと使用するコンバーターファクトリーのオプション名。ファクトリーは、クライアントに送信されたイベントの内容を変換するために使用されます。デフォルトでは、コンバーターが使用されていない場合、生成されるイベントのタイプに応じてイベントが適切に定義されます。ただし、ユーザーがイベントに追加情報を追加したり、イベントのサイズを縮小する必要がある場合もあります。このような場合、コンバーターを使用してイベントの内容を変換できます。指定のコンバーターファクトリー名は、このジョブを実行するコンバーターインスタンスを作成します。ファクトリーが使用されない場合、文字列の長さは **0** になります。
コンバーターファクトリーパラメーター数	byte	コンバーターインスタンスの作成時にコンバーターファクトリーは任意の数のパラメーターを取り、ファクトリーを使用して異なるコンバーターインスタンスを動的に作成できます。この count フィールドは、ファクトリーに渡されるパラメーターの数を示します。ファクトリー名が指定されていない場合、このフィールドはリクエストに表示されません。
コンバーターファクトリーパラメーター1	バイト配列	最初のコンバーターファクトリーパラメーター
コンバーターファクトリーパラメーター2	バイト配列	2番目のコンバーターファクトリーパラメーター
...		

応答(0x26):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

リモートイベントのクライアントリスナーの削除

要求(0x27):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

フィールド名	サイズ	値
リスナー ID	バイト配列	リスナー識別子

応答(0x28):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

サイズ

要求(0x29):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

応答(0x2A):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
サイズ	vInt	クラスター化されたセットアップでグローバルに計算されるリモートキャッシュのサイズ。存在する場合は、キャッシュストアの内容も考慮されます。

exec

要求(0x2B):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
スクリプト	string	実行するタスクの名前

フィールド名	サイズ	値
パラメーター数	vInt	パラメーターの数
パラメーター1名	string	最初のパラメーターの名前
パラメーター1長	vInt	最初のパラメーターの長さ
パラメーター1値	バイト配列	最初のパラメーターの値

応答(0x2C):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success, if execution completed successfully 0x85 = server error
値の長さ	vInt	成功した場合、戻り値の長さ
値	バイト配列	成功すると、実行の結果

20.6.5.5. リモートイベント

Hotgitops 2.0 以降、クライアントはサーバーで発生するリモートイベントのリスナーを登録できます。これらのイベントを送信すると、クライアントがリモートイベントにクライアントリスナーを追加するようになります。

イベントヘッダー：

フィールド名	サイズ	値
magic	1バイト	0xA1 = response
メッセージ ID	vLong	イベントの ID

フィールド名	サイズ	値
opcode	1バイト	event type: 0x60 = cache entry created event 0x61 = cache entry modified event 0x62 = cache entry removed event 0x66 = counter event 0x50 = error
ステータス	1バイト	応答のステータス。可能な値 - 0x00 = No error
トポロジー変更マーカー	1バイト	イベントは特定の受信トポロジー ID に関連付けられておらず、新しいトポロジーを送信する必要があるかどうかを判断できるようにするため、新しいトポロジーはイベントで送信されません。そのため、このマーカーは常にイベントの **0** 値を持ちます。

表20.3 Cache entry created event

フィールド名	サイズ	値
ヘッダー	variable	**0x60** 操作コードのあるイベントヘッダー
リスナー ID	バイト配列	このイベントが転送されるリスナー
カスタムマーカー	byte	カスタムイベントマーカー。作成されたイベントの場合、これは **0** です。
再試行したコマンド	byte	再試行されたコマンドの結果であるイベントのマーカー。コマンドが再試行されると **1** を返します。それ以外の場合は **0** を返します。
キー	バイト配列	作成されたキー
Version	Long	作成されたエントリーのバージョン。このバージョン情報を使用して、このキャッシュエントリーで条件付き操作を行うことができます。

表20.4 キャッシュエントリー変更イベント

フィールド名	サイズ	値
ヘッダー	variable	**0x61** 操作コードのイベントヘッダー
リスナー ID	バイト配列	このイベントが転送されるリスナー

フィールド名	サイズ	値
カスタムマーカー	byte	カスタムイベントマーカー。作成されたイベントの場合、これは **0** です。
再試行したコマンド	byte	再試行されたコマンドの結果であるイベントのマーカー。コマンドが再試行されると **1** を返します。それ以外の場合は **0** を返します。
キー	バイト配列	変更したキー
Version	Long	変更したエントリーのバージョン。このバージョン情報を使用して、このキャッシュエントリーで条件付き操作を行うことができます。

表20.5 キャッシュエントリーの削除イベント

フィールド名	サイズ	値
ヘッダー	variable	**0x62** 操作コードのあるイベントヘッダー
リスナー ID	バイト配列	このイベントが転送されるリスナー
カスタムマーカー	byte	カスタムイベントマーカー。作成されたイベントの場合、これは **0** です。
再試行したコマンド	byte	再試行されたコマンドの結果であるイベントのマーカー。コマンドが再試行されると **1** を返します。それ以外の場合は **0** を返します。
キー	バイト配列	削除されたキー

表20.6 カスタムイベント

フィールド名	サイズ	値
ヘッダー	variable	イベント固有の操作コードを含むイベントヘッダー
リスナー ID	バイト配列	このイベントが転送されるリスナー
カスタムマーカー	byte	カスタムイベントマーカー。カスタムイベントの場合、これは **1** です。
イベントデータ	バイト配列	コンバーター実装ロジックに従ってフォーマットされるカスタムイベントデータ。

20.6.6. hotgitops Protocol 2.1

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 7.1.0.Final 以降実装されます。

20.6.6.1. リクエストヘッダー

ヘッダーの バージョン フィールドは 21 に更新されます。

20.6.6.2. 操作

リモートイベントのクライアントリスナーの追加

追加のバイトパラメーターが最後に追加されます。これは、クライアントがフィルター/変換コールバックの raw バイナリーデータと連携するようにクライアントリスナーを優先するかどうかを示します。raw データを使用する場合、その値は 1 になります。それ以外の場合は 0 です。

要求の形式：

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
リスナー ID	バイト配列	...
状態を含める	byte	...
キー/値フィルターファクトリーパラメーターの数	byte	...
...		
コンバーターファクトリー名	string	...
コンバーターファクトリーパラメーター数	byte	...
...		

フィールド名	サイズ	値
生データの使用	byte	filter/converter パラメーターを raw バイナリーにする必要がある場合は **1**、それ以外は **0** です。

カスタムイベント

Hot336 2.1 以降、カスタムイベントは、ユーザーに渡す前にホットフィックスクライアントがアンマーシャリングを試行しない生データを返すことができます。これは、Hotfaillock クライアントに送信される方法は、カスタムイベントマーカーとして 2 を送信することです。そのため、カスタムイベントの形式は以下のようになります。

フィールド名	サイズ	値
ヘッダー	variable	イベント固有の操作コードを含むイベントヘッダー
リスナー ID	バイト配列	このイベントが転送されるリスナー
カスタムマーカー	byte	カスタムイベントマーカー。ユーザーに戻る前にイベントデータをアンマーシャリングする必要があるカスタムイベントの場合、値は **1** です。イベントデータを初期状態でユーザーに返す必要があるカスタムイベントの場合、値は **2** です。
イベントデータ	バイト配列	カスタムイベントデータ。カスタムマーカーが **1** の場合、バイトはコンバーターによって返されるインスタンスのマーシャリングされたバージョンを表します。カスタムマーカーが **2** の場合、コンバーターによって返されるようにバイト配列を表します。

20.6.7. hotgitops Protocol 2.2

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 8.0 以降実装されます。

異なる時間単位のサポートを追加

20.6.7.1. 操作

Put/PutAll/PutIfAbsent/Replace/ReplaceIfUnmodified

一般的な要求形式：

フィールド名	サイズ	値
TimeUnits	Byte	ライフスパンの時間単位（最初の 4 ビット）および maxIdle（最後 4 ビット）。デフォルトのサーバーの有効期限には、特別なユニット DEFAULT と INFINITE を使用できます。使用できる値： 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANOSECONDS 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE
有効期間	vLong	エントリーの有効期間。時間単位が DEFAULT または INFINITE でない場合にのみ送信されます。
最大 ID	vLong	各エントリーがキャッシュからエビクトされる前にアイドル状態になる期間。時間単位が DEFAULT または INFINITE でない場合にのみ送信されます。

20.6.8. hotgitops Protocol 2.3

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 8.0 以降実装されます。

20.6.8.1. 操作

iteration Start

要求(0x31):

フィールド名	サイズ	値
セグメントサイズ	署名済み vInt	反復するセグメント ID のビットセットエンコーディングのサイズ。このサイズは、8 の倍数に丸められた最大のセグメント ID です。 値が -1 の場合は、セグメントのフィルタリングが行われないことを示します。

フィールド名	サイズ	値
Segments	バイト配列	（オプション）segments ids ビットセットがエンコードされており、各ビットが値 1 のセグメントを表します。バイトの順序はリトルエンディアンです。 例： segments [1,3,12,13] では、エンコーディング 00001010 00110000 サイズ： 16 ビット の最初のバイト： 0 から 7 までのセグメントを表し、1 および 3 秒から 4 番目のバイトにセグメントを表します。12 および 13 は、 java.util.BitSet 実装の の詳細に設定されます。 以前のフィールドが負の値でない場合にセグメントが送信されます。
FilterConverter サイズ	署名済み vInt	サーバーにデプロイされた KeyValueFilterConverter ファクトリー名を表す String のサイズ。フィルターを使用しない場合は -1。
FilterConverter	UTF-8 バイト配列	（オプション）サーバーにデプロイされた KeyValueFilterConverter ファクトリー名。前のフィールドが負の値の場合には指定します。
BatchSize	vInt	サーバーから転送するエントリーの数

応答(0x32):

フィールド名	サイズ	値
IterationId	文字列	反復の一意の ID

反復次へ

要求(0x33):

フィールド名	サイズ	値
IterationId	文字列	反復の一意の ID

応答(0x34):

フィールド名	サイズ	値
セグメントサイズの終了	vInt	反復を完了したセグメントを表すビットセットのサイズ
終了したセグメント	バイト配列	反復したセグメントのビットセットエンコーディング。
エントリー数	vInt	返されるエントリーの数
キー1長	vInt	キーの長さ
キー1	バイト配列	取得されるキー
値1の長さ	vInt	値の長さ
値1	バイト配列	取得した値
キー2長	vInt	
キー2	バイト配列	
値2の長さ	vInt	
値2	バイト配列	
... エントリー数に達するまで継続する		

iteration End

要求(0x35):

フィールド名	サイズ	値
IterationId	文字列	反復の一意の ID

応答(0x36):

ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success, if execution completed successfully 0x05 = for non existent IterationId

20.6.9. hotgitops Protocol 2.4

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 8.1 以降実装されます。

この Hotgitops プロトコルバージョンは、サーバーの互換モードが有効かどうかについてクライアントのヒントを提供する新しいステータスコードを追加します。

- *0x06: Success status and compatibility mode is enabled.*
- *0x07: 成功し、互換性モードが有効になっている以前の値を返します。*
- *0x08: 互換性モードが有効になっているため、実行されず、以前の値を返します。*

Iteration Start 操作は、カスタムフィルターが提供され、パラメーター化された場合にパラメーターをオプションで送信することができます。

20.6.9.1. 操作

iteration Start

要求(0x31):

フィールド名	サイズ	値
セグメントサイズ	署名済み vInt	プロトコルバージョン 2.3 と同じ
Segments	バイト配列	プロトコルバージョン 2.3 と同じ

フィールド名	サイズ	値
FilterConverter サイズ	署名済み vInt	プロトコルバージョン 2.3 と同じ
FilterConverter	UTF-8 バイト配列	プロトコルバージョン 2.3 と同じ
パラメーターサイズ	byte	フィルターのパラメーター数。FilterConverter が提供される場合にのみ表示されます。
パラメーター	byte[][]	パラメーターの配列。各パラメーターはバイト配列です。パラメーターサイズが 0 よりも大きい場合にのみ表示されます。
BatchSize	vInt	プロトコルバージョン 2.3 と同じ

Iteration Next 操作は任意で値にプロジェクトを返すことができます。つまり、同じエントリーに複数の値が含まれます。

反復次へ

応答(0x34):

フィールド名	サイズ	値
セグメントサイズの終了	vInt	プロトコルバージョン 2.3 と同じ
終了したセグメント	バイト配列	プロトコルバージョン 2.3 と同じ
エントリー数	vInt	プロトコルバージョン 2.3 と同じ
値プロジェクト数	vInt	値のプロジェクト数。1 の場合、バージョンプロトコルバージョン 2.3 のように動作します。
Key1 Length	vInt	プロトコルバージョン 2.3 と同じ
Key1	バイト配列	プロトコルバージョン 2.3 と同じ
Value1 projection1 length	vInt	value1 の最初の展開の長さ

フィールド名	サイズ	値
Value1 projection1	バイト配列	取得された value1 の最初の展開
Value1 projection2 length	vInt	value2 second projection の長さ
Value1 projection2	バイト配列	取得された value2 秒の展開
... 取得された値のすべての展開が継続される	Key2 Length	vInt
プロトコルバージョン 2.3 と同じ	Key2	バイト配列
プロトコルバージョン 2.3 と同じ	Value2 projection1 length	vInt
値 2 の最初の展開の長さ	Value2 projection1	バイト配列
取得した最初の展開値 2	Value2 projection 2 length	vInt
値 2 秒の展開の長さ	Value2 projection 2	バイト配列
取得した値 2 秒の展開	... エントリー数に達するまで継続する	

1.

stats:

これまでの HotTEMPLATES プロトコルバージョンによって返される統計は、Hotgitops 操作が呼び出されたノードにローカルでした。2.4 以降、以前返された統計のグローバル数を提供する新しい統計が追加されました。Hotfsprogs がローカルモードで実行されている場合は、以下の統計は返されません。

名前	説明
globalCurrentNumberOfEntries	現在、Hotim cluster クラスター全体にあるエントリーの数。
globalStores	Hotgitops クラスター全体での put 操作の合計数。
globalRetrievals	Hotgitops クラスター全体の get 操作の合計数。
globalHits	Hotgitops クラスター全体の get hits の合計数。
globalMisses	Hotgitops クラスター全体での get misss の合計数。
globalRemoveHits	Hotgitops クラスター全体の削除ヒットの合計数。
globalRemoveMisses	Hotgitops クラスター全体の削除ミスの合計数。

20.6.10. hotgitops Protocol 2.5

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 8.2 以降実装されます。

この Hotgitops プロトコルバージョンは、イテレーターのエントリーとともにメタデータ取得のサポートが追加されました。これには、以下の 2 つの変更が含まれます。

- *iteration Start 要求にはオプションのフラグが含まれます。*
- *IterationNext operation には、上記のフラグが設定されている場合、各エントリーのメタデータ情報を含めることができます。*

iteration Start

要求(0x31):

フィールド名	サイズ	値
セグメントサイズ	署名済み vlnt	プロトコルバージョン 2.4 と同じです。

フィールド名	サイズ	値
Segments	バイト配列	プロトコルバージョン 2.4 と同じです。
FilterConverter サイズ	署名済み vInt	プロトコルバージョン 2.4 と同じです。
FilterConverter	UTF-8 バイト配列	プロトコルバージョン 2.4 と同じです。
パラメーターサイズ	byte	プロトコルバージョン 2.4 と同じです。
パラメーター	byte[][]	プロトコルバージョン 2.4 と同じです。
BatchSize	vInt	プロトコルバージョン 2.4 と同じです。
メタデータ	1バイト	各エントリーに対してメタデータが返される場合は 1、それ以外は 0

反復次へ

応答(0x34):

フィールド名	サイズ	値
セグメントサイズの終了	vInt	プロトコルバージョン 2.4 と同じです。
終了したセグメント	バイト配列	プロトコルバージョン 2.4 と同じです。
エントリー数	vInt	プロトコルバージョン 2.4 と同じです。
値プロジェクト数	vInt	プロトコルバージョン 2.4 と同じです。
メタデータ（エントリー 1）	1バイト	設定されている場合、エントリーにはメタデータが関連付けられます。
有効期限（エントリー 1）	1バイト	応答に有効期限情報が含まれるかどうかを示すフラグ。フラグの値は、INFINITE_LIFESPAN(0x01)と **INFINITE_MAXIDLE(0x02)の間のビット単位の OR 操作として取得されます**。上記のメタデータフラグが設定されている場合にのみ表示されます。

フィールド名	サイズ	値
作成日（エントリー1）	Long	（オプション）エントリーがサーバー上で作成された時点のタイムスタンプを表す長期。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合のみ返されます。
ライフスパン（エントリー1）	vInt	（オプション）エントリーの有効期間を表す vInt を秒単位で表示します。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合のみ返されます。
LastUsed（エントリー1）	Long	（オプション）エントリーがサーバー上で最後にアクセスされた時点のタイムスタンプを表す長期。この値は、フラグの **INFINITE_MAXIDLE** ビットが設定されていない場合のみ返されます。
maxIdle（エントリー1）	vInt	（オプション）エントリーの maxIdle を表す vInt（秒単位）。この値は、フラグの **INFINITE_MAXIDLE** ビットが設定されていない場合のみ返されます。
エントリーバージョン（エントリー1）	8 バイト	既存のエントリーの変更の一意の値。Metadata フラグが設定されている場合にのみ表示されます。
キー1長	vInt	プロトコルバージョン 2.4 と同じです。
キー1	バイト配列	プロトコルバージョン 2.4 と同じです。
値1の長さ	vInt	プロトコルバージョン 2.4 と同じです。
値1	バイト配列	プロトコルバージョン 2.4 と同じです。
メタデータ（エントリー2）	1バイト	エントリー1と同じ
有効期限（エントリー2）	1バイト	エントリー1と同じ
作成日（エントリー2）	Long	エントリー1と同じ
lifespan（エントリー2）	vInt	エントリー1と同じ
LastUsed（エントリー2）	Long	エントリー1と同じ
maxIdle（エントリー2）	vInt	エントリー1と同じ

フィールド名	サイズ	値
エントリーバージョン（エントリー 2）	8 バイト	エントリー 1 と同じ
キー 2 長	vInt	
キー 2	バイト配列	
値 2 の長さ	vInt	
値 2	バイト配列	
... エントリー数に達するまで継続する		

20.6.11. hotgitops Protocol 2.6

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 9.0 以降実装されます。

この Hotfsprogs プロトコルバージョンは、ストリーミングの get 操作および put 操作のサポートが追加されました。これには、2 つの新たな操作が含まれます。

- *任意の初期オフセットを使用して、ストリームとしてデータを取得する getStream*
- *必要に応じてバージョンを指定して、データをストリームとして記述するための PutStream*

GetStream

要求(0x37):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
Offset	vInt	取得を開始するバイト単位のオフセット。最初から取得する場合は 0 に設定します。
キーの長さ	vInt	キーの長さ。vint のサイズは最大 5 バイトで、理論では Integer.MAX_VALUE よりも大きな数値を生成することができます。ただし、Java では Integer.MAX_VALUE を超える単一のアレイを作成できないため、プロトコルは vint アレイの長さを Integer.MAX_VALUE に制限します。
キー	バイト配列	値が要求されているキーが含まれるバイトアレイ。

GetStream

応答(0x38):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答のステータス	1バイト	0x00 = success（キーの取得の場合） 0x02 = if key does not exist
フラグ	1バイト	応答に有効期限情報が含まれるかどうかを示すフラグ。フラグの値は、INFINITE_LIFESPAN(0x01)と **INFINITE_MAXIDLE(0x02)の間のビット単位の OR 操作として取得されます。**
Created	Long	（オプション）エントリーがサーバー上で作成された時点のタイムスタンプを表す長期。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合のみ返されます。
有効期間	vInt	（オプション）エントリーの有効期間を表す vInt を秒単位で表示します。この値は、フラグの INFINITE_LIFESPAN ビットが設定されていない場合のみ返されます。
LastUsed	Long	（オプション）エントリーがサーバー上で最後にアクセスされた時点のタイムスタンプを表す長期。この値は、フラグの **INFINITE_MAXIDLE** ビットが設定されていない場合のみ返されます。
MaxIdle	vInt	（オプション）エントリーの maxIdle を表す vInt（秒単位）。この値は、フラグの **INFINITE_MAXIDLE** ビットが設定されていない場合のみ返されます。

フィールド名	サイズ	値
エントリーバージョン	8 バイト	既存のエントリーの変更の一意の値。このプロトコルは、entry_version 値が連続して行われるという義務はありません。更新ごとにキーレベルで一意となる必要があります。
値の長さ	vInt	成功した場合、値の長さ
値	バイト配列	成功した場合、要求された値。

PutStream

要求(0x39)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
エントリーバージョン	8 バイト	使用できる値 0 = Unconditional put -1 = Put If Absent Other values = pass a version retrieved by **GetWithMetadata** operation to perform a conditional replace.
キーの長さ	vInt	キーの長さ。vint のサイズは最大 5 バイトで、理論では Integer.MAX_VALUE よりも大きな数値を生成することができます。ただし、Java では Integer.MAX_VALUE を超える単一のアレイを作成できないため、プロトコルは vint アレイの長さを Integer.MAX_VALUE に制限します。
キー	バイト配列	値が要求されているキーが含まれるバイトアレイ。
Value Chunk 1 Length	vInt	データの最初のチャンクのサイズ。この値が 0 の場合は、クライアントが値の転送を完了し、操作を実行する必要があります。
Value Chunk 1	バイト配列	データの fist チャンクを形成するバイトの配列。
値が完了するまで継続		

応答(0x3A):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

これらの追加上部で、この HotTEMPLATES プロトコルバージョンは、グローバルレベルで示されているバイトを追加してリモートリスナーの登録が改善されます。これは、クライアントが対象のイベントのタイプを示します。たとえば、クライアントは作成されたイベントのみ、または有効期限および削除イベントのみを示すことができます。キーごとにより詳細なイベント関連の情報は、key/value フィルターパラメーターを使用して定義できます。

新しい add リスナーリクエストは以下のようになります。

リモートイベントのクライアントリスナーの追加

要求(0x25):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
リスナー ID	バイト配列	リスナー識別子
状態を含める	byte	このバイトが **1** に設定されると、初めてキャッシュリスナーを追加するとき、またはリモートリスナーが登録されたノードがクラスター環境に変更されると、キャッシュされた状態はリモートクライアントに送信されます。有効にすると、状態はキャッシュエントリーで作成されたイベントとしてクライアントに送信されます。**0** に設定すると、リスナーの追加時に状態はクライアントに返されず、リスナーが登録されたノードが変更されると状態が生じます。
キー/値フィルターファクトリー名	string	このリスナーと使用するキー/値フィルターファクトリーのオプション名。ファクトリーはキー/値のフィルターインスタンスを作成するために使用されます。これにより、イベントが Hotgitops サーバーで直接フィルターされ、クライアントが関係しないイベントの送信を回避することができます。ファクトリーが使用されない場合、文字列の長さは **0** になります。
キー/値フィルターファクトリーパラメーターの数	byte	キー/値のフィルターファクトリーは、フィルターインスタンスの作成時に任意の数のパラメーターを取り、ファクトリーを使用して異なるフィルターインスタンスを動的に作成できるようにします。この count フィールドは、ファクトリーに渡されるパラメーターの数を示します。ファクトリー名が指定されていない場合、このフィールドはリクエストに表示されません。

フィールド名	サイズ	値
キー/値フィルターファクトリーパラメーター1	バイト配列	最初のキー/値フィルターファクトリーパラメーター
キー/値フィルターファクトリーパラメーター2	バイト配列	2つ目のキー/値フィルターファクトリーパラメーター
...		
コンバーターファクトリー名	string	このリスナーと使用するコンバーターファクトリーのオプション名。ファクトリーは、クライアントに送信されたイベントの内容を変換するために使用されます。デフォルトでは、コンバーターが使用されていない場合、生成されるイベントのタイプに応じてイベントが適切に定義されます。ただし、ユーザーがイベントに追加情報を追加したり、イベントのサイズを縮小する必要がある場合もあります。このような場合、コンバーターを使用してイベントの内容を変換できます。指定のコンバーターファクトリー名は、このジョブを実行するコンバーターインスタンスを作成します。ファクトリーが使用されない場合、文字列の長さは**0**になります。
コンバーターファクトリーパラメーター数	byte	コンバーターインスタンスの作成時にコンバーターファクトリーは任意の数のパラメーターを取り、ファクトリーを使用して異なるコンバーターインスタンスを動的に作成できます。この count フィールドは、ファクトリーに渡されるパラメーターの数を示します。ファクトリー名が指定されていない場合、このフィールドはリクエストに表示されません。
コンバーターファクトリーパラメーター1	バイト配列	最初のコンバーターファクトリーパラメーター
コンバーターファクトリーパラメーター2	バイト配列	2番目のコンバーターファクトリーパラメーター
...		
タイプ関連のあるリスナー	vInt	関係するリスナーイベントタイプを表す変数長番号。各イベントタイプはビットで表されます。各フラグはビットで表されます。このフィールドは変数の長さとして送信されるため、バイトで最も大きなビットを使用して、より多くのバイトを読み取る必要があるかどうかを判断するため、このビットはフラグを表します。このモデルを使用すると、フラグを短いスペースに統合できます。各フラグの現在の値は次のとおりです。 0x01 = cache entry created events 0x02 = cache entry modified events 0x04 = cache entry removed events 0x08 = cache entry expired events

20.6.12. hotgitops Protocol 2.7

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 9.2 以降実装されます。

この Hotfsprogs プロトコルバージョンは、トランザクション操作のサポートを追加しています。これには、3 つの新たな操作が含まれます。

- *トランザクション書き込みセット（変更されたキーなど）を使用して prepare は、サーバーでトランザクションを準備および検証しようとします。*
- *準備済みトランザクションをコミットします。*
- *ロールバックし、準備済みのトランザクションをロールバックします。*

要求の準備

要求(0x3B):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
Xid	XID	トランザクション ID(XID)
OnePhaseCommit	byte	**1** に設定すると、サーバーは利用可能な場合は 1 フェーズコミットを使用します（XA のみ）。
キー数	vInt	キー数
各キーについて（キーは一意でなければなりません）		
キーの長さ	vInt	キーの長さ。vInt のサイズは最大 5 バイトで、理論では **Integer.MAX_VALUE** よりも大きな数値を生成することができる点に注意してください。ただし、Java では **Integer.MAX_VALUE** を超える単一のアレイを作成できないため、プロトコルは vInt アレイの長さを **Integer.MAX_VALUE** に制限します。

フィールド名	サイズ	値
キー	バイト配列	キーを含むバイトアレイ
Control Byte	Byte	以下の意味を持つビットセット。 0x01 = **NOT_READ** 0x02 = **NON_EXISTING** 0x04 = **REMOVE_OPERATION** を同時に設定できないことに注意してください。
バージョン読み取り	Long	読み込まれたバージョン。**NOT_READ** および **NON_EXISTING** が存在しない場合のみ送信されます。
TimeUnits	Byte	ライフスパンの時間単位（最初の 4 ビット）および maxIdle（最後 4 ビット）。デフォルトのサーバーの有効期限には、特別なユニット **DEFAULT** と **INFINITE** を使用できます。使用できる値： 0x00 = **SECONDS** 0x01 = **MILLISECONDS** 0x02 = **NANOSECONDS** 0x03 = **MICROSECONDS** 0x04 = **MINUTES** 0x05 = **HOURS** 0x06 = **DAYS** 0x 07 = **DEFAULT** 0x08 = **INFINITE** のみが送信され、**REMOVE_OPERATION** が設定されていない場合のみ送信されます。
有効期間	vLong	エントリーの有効期間。時間の単位が **DEFAULT** または **INFINITE _OPERATION** が設定されていない場合にのみ送信されます。
最大 ID	vLong	各エントリーがキャッシュからエビクトされる前にアイドル状態になる期間。時間の単位が **DEFAULT** または **INFINITE _OPERATION** が設定されていない場合にのみ送信されます。
値の長さ	vInt	値の長さ。**REMOVE_OPERATION** が設定されていない場合にのみ送信されます。
値	byte-array	保存する値。**REMOVE_OPERATION** が設定されていない場合にのみ送信されます。

コミットとロールバックリクエスト

要求。コミット(0x3D)および Rollback(0x3F):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
Xid	XID	トランザクション ID(XID)

準備、コミット、ロールバックリクエストからの応答。

応答。prepare(0x3C)、Commit(0x3E)、および Rollback(0x40)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
XA 戻りコード	vInt	準備応答を表す XA コード。 **XA_OK(0**)、**XA_RDONLY(3)** またはエラーコードのいずれかを指定できます（**XaException**を参照）。 応答の状態が **Successful** と異なる場合、このフィールドは表示されません。

XID 形式

リクエストの XID の形式は以下のとおりです。

フィールド名	サイズ	値
形式 ID	署名済み vInt	XID 形式。
グローバルトランザクション ID の長さ	byte	グローバルトランザクション ID バイトアレイの長さ。最大値は **64** です。
グローバルトランザクション ID	バイト配列	グローバルトランザクション ID。
ブランチ修飾子の長さ	byte	ブランチ修飾子バイトアレイの長さ。最大値は **64** です。
ブランチ修飾子	バイト配列	ブランチ修飾子。

カウンターの設定エンコーディング形式

CounterConfiguration クラスエンコーディングの形式は以下のとおりです。

注記

カウンター関連の操作では、Request Header の Cache Name フィールドが空にすることができます。

注記

Response Header:
** 0x00: Operation successful.*
** 0x01: Operation failed.*
** 0x02: The counter is't defined.*
** 0x04 : The counter reached a boundary. * 0x04: The counter reached a boundary.*
STRONG カウンターのみ可能です。

フィールド名	サイズ	値
Flags	byte	**CounterType** および **Storage** がエンコードされました。以下として、大きなビットのみが使用されます。 1 - **WEAK** カウンターの場合は **1**、**STRONG** カウンターの場合は **0** です。 2 番目のビット - **BOUNDED** カウンターの場合は **1**、**UNBOUNDED** カウンターの場合は **0** 、3 番目のビット：**1**（**PERSISTENT** ストレージの場合は 1、**VOLATILE** ストレージの場合）
同時実行レベル	vInt	（オプション）カウンターの同時実行レベル。カウンターが **WEAK** の場合にのみ表示されます。
下限	Long	（任意）バインドされたカウンターの下限。カウンターが **BOUNDED** の場合にのみ表示されます。
上限	Long	（任意）バインドされたカウンターの上限です。カウンターが **BOUNDED** の場合にのみ表示されます。
初期値	Long	カウンターの最初の値。

counter create operation

カウンターが存在しない場合は作成します。

表20.7 要求(0x4B)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
名前	string	カウンターの名前
カウンターの設定	variable	カウンターの設定。CounterConfiguration encode を参照してください。

表20.8 応答(0x4C)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

Response Header Status には、以下の値が使用できます。

- *0x00: 操作に成功しました。*
- *0x01: 操作に失敗しました。カウンターはすでに定義されています。*
- *エラーコードについては、「Reponse Header」を参照してください。*

カウンター get configuration 操作

カウンターの設定を返します。

表20.9 要求(0x4D)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
名前	string	カウンターの名前。

表20.10 Response (0x4E)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

フィールド名	サイズ	値
カウンターの設定	variable	（オプション）カウンターの設定。**Status==0x00** の場合にのみ表示されます。CounterConfiguration encode を参照してください。

Response Header Status には、以下の値が使用できます。

- ***0x00: 操作に成功しました。***
- ***0x02: Counter は存在しません。***
- ***エラーコードについては、「Reponse Header」を参照してください。***

カウンターは定義された操作です。

カウンターが定義されているかどうかを確認します。

表20.11 要求(0x4F)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
名前	string	カウンターの名前

表20.12 応答(0x51)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

Response Header Status には、以下の値が使用できます。

- ***0x00: Counter が定義されます。***
- ***0x01: Counter が定義されていない。***

- *エラーコードについては、「Reponse Header」を参照してください。*

カウンターの add-and-get 操作

カウンターに値を追加し、新しい値を返します。

表20.13 要求(0x52)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
名前	string	カウンターの名前
値	Long	追加する値

表20.14 応答(0x53)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
値	Long	（任意設定）カウンターの新しい値。**Status==0x00** の場合にのみ表示されます。

注記

WeakCounter は新しい値にアクセスできないため、値 はゼロになります。

Response Header Status には、以下の値が使用できます。

- *0x00: 操作に成功しました。*
- *0x02: カウンターが定義されていません。*
- *0x04: カウンターがその境界に到達しました。STRONG カウンターのみ可能です。*

- *エラーコードについては、「Reponse Header」を参照してください。*

counter reset operation

カウンターの値をリセットします。

表20.15 要求(0x54)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
名前	string	カウンターの名前

表20.16 応答(0x55)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

Response Header Status には、以下の値が使用できます。

- *0x00: 操作に成功しました。*
- *0x02: Counter が定義されていない。*
- *エラーコードについては、「Reponse Header」を参照してください。*

カウンター get 操作

カウンターの値を返します。

表20.17 要求(0x56)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
名前	string	カウンターの名前

表20.18 応答(0x57)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
値	Long	（任意設定）カウンターの値。**Status==0x00** の場合にのみ表示されます。

Response Header Status には、以下の値が使用できます。

- ***0x00: 操作に成功しました。***
- ***0x02: Counter が定義されていない。***
- ***エラーコードについては、「Reponse Header」を参照してください。***

カウンター比較およびスワップ操作

現在の値が想定される場合、比較してカウンター値のみを更新します。

表20.19 要求(0x58)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
名前	string	カウンターの名前
expect	Long	カウンターの期待値。
Update	Long	設定するカウンターの値。

表20.20 応答(0x59)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
値	Long	（任意設定）カウンターの値。**Status==0x00** の場合にのみ表示されます。

Response Header Status には、以下の値が使用できます。

- *0x00: 操作に成功しました。*
- *0x02: カウンターが定義されていません。*
- *0x04: カウンターがその境界に到達しました。STRONG カウンターのみ可能です。*
- *エラーコードについては、「Reponse Header」を参照してください。*

カウンターのリスナーの追加および削除

カウンターのリスナーを追加/削除します。

表20.21 Request ADD(0x5A)/ REMOVE(0x5C)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
名前	string	カウンターの名前
listener-id	バイト配列	リスナーの ID

表20.22 応答： ADD(0x5B)/ REMOVE(0x5D)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

Response Header Status には、以下の値が使用できます。

- *0x00: 成功し、リクエストで使用される接続を使用してイベント(add)を送信するか、または接続を削除（削除）できます。*

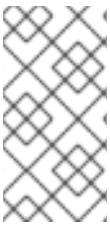
- *0x01: 成功し、現在の接続は引き続き使用中です。*
- *0x02: カウンターが定義されていません。*
- *エラーコードについては、「Reponse Header」を参照してください。*

表20.23 カウンターイベント(0x66)

フィールド名	サイズ	値
ヘッダー	variable	操作コード **0x66**のイベントヘッダー
名前	string	カウンターの名前
listener-id	バイト配列	リスナーの ID
エンコードされたカウンターの状態	byte	old および new counter の状態にエンコードされました。bit set: **------00**: Valid old state **------01**: Lower bound reached old state **------10**: Upper bound reached old state **----00--**: Valid new state **----01--**: Lower bound reached new state **----10--**: Upper bound reached new state
古い値	Long	カウンターの古い値
新しい値	Long	カウンターの新しい値

注記

CounterManager 実装内のすべてのカウンターは、同じ listener-id を使用できます。



注記

接続は単一の listener-id に専用で、異なるカウンターからイベントを受信できます。

counter remove operation

クラスターからカウンターを削除します。

注記

カウンターは、再度アクセスされている場合は再作成されます。

表20.24 要求(0x5E)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
名前	string	カウンターの名前

表20.25 応答(0x5F)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

Response Header Status には、以下の値が使用できます。

- 0x00: 操作に成功しました。
- 0x02: カウンターが定義されていません。
- エラーコードについては、「Reponse Header」を参照してください。

20.6.13. hotgitops Protocol 2.8

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 9.3 以降実装されます。

イベント

プロトコルを使用すると、クライアントは以前に Add Client Listener 操作に使用したのと同じ接続でリクエストを送信でき、プロトコルの < 2.8 はイベントをクライアントに送信するために予約されています。これには追加のリスナーの登録が含まれるため、複数のリスナーのイベントを受信します。

requests/responses/events のバイナリー形式は変更しませんが、イベントの以前の意味がない messageId を以下に設定する必要があります。

- *include-current-state イベントに対する Add Client Listener 操作の Messageid*
- *Add Client Listener 操作の完了後に送信されるイベントの場合は 0 （応答送信）。*

カウンターイベントに同じ保持： クライアントは Counter Add Listener の後にさらにリクエストを送信できます。以前のバージョンでは、カウンターイベントで messageId が常に 0 に設定されていました。

このプロトコルの変更には、クライアント側で変更は必要ありません（クライアントがこれに対応していない場合は追加の操作を送信しないため）。変更はクライアントに対して許容されますが、サーバーは接続の負荷を正しく処理する必要があります。

MediaType

この Hotgitops プロトコルバージョンには、Key and Values の MediaType の指定のサポートが追加され、異なる形式でデータの読み取り（および書き込み）が可能になります。この情報は Header の一部です。

データ形式は、以下のように表現される **MediaType** *オブジェクトを使用して記述されます。*

フィールド名	サイズ	値
type	1 バイト	0x00 = No MediaType provided 0x01 = Pre-defined MediaType provided 0x02 = Custom MediaType provided
id	vInt	（オプション）事前定義済みの MediaType(type=0x01)の場合：MediaType の ID。現在サポートされている ID は MediaTypeIdsにあります。
customString	string	（オプション）カスタム MediaType が指定されている場合 (type=0x02)、type および subtype を含むキーのカスタム MediaType。例： **text/plain**、**application/json** など

フィールド名	サイズ	値
paramSize	vInt	MediaType のパラメーターサイズ
paramKey1	string	（オプション）最初のパラメーターのキー
paramValue1	string	（オプション）最初のパラメーターの値
...	...	...
paramKeyN	string	（オプション）nth パラメーターのキー
paramValueN	string	（オプション）nth パラメーターの値

20.6.13.1. リクエストヘッダー

要求ヘッダーには以下の追加フィールドがあります。

フィールド名	タイプ	値
キーの形式	MediaType	操作中にキーに使用する MediaType。これは、送受信鍵の両方に適用されます。
値の形式	MediaType	キー形式と似ていますが、値に適用されます。

20.6.14. hotgitops Protocol 2.9

INFINISPAN のバージョン

このバージョンのプロトコルは Infinispan 9.4 以降実装されます。

互換性モードの削除

操作からの Response status フィールドの互換性モードヒントは送信されなくなりました。したがって、以下のステータスが削除されます。

- *0x06: 互換性モードのある正常なステータス。*

- *0x07: return previous value and compatibility mode で成功したステータス。*
- *0x08: return previous value and compatibility mode では実行されません。*

サーバーのストレージが何であるかを確認するには、ping 操作で設定したキーと値の MediaType と値が返されます。

Ping 応答(0x18):

フィールド名	サイズ	値
ヘッダー	variable	以前と同じ
応答のステータス	1バイト	以前と同じ
キーのタイプ	MediaType	サーバーに保存されているキーのメディアタイプ
値のタイプ	MediaType	サーバーに保存されている値のメディアタイプ

新しいクエリー形式

このバージョンは、JSON 形式のクエリー要求および応答をサポートします。操作 0x1F (Query Request)および 0x20 (Query Response)の形式は変更されません。

JSON ペイロードを送信するには、ヘッダーの「値形式」フィールドは **application/json** *である必要があります。*

クエリーリクエスト(0x1F):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
クエリーの長さ	vInt	UTF-8 でエンコードされたクエリーオブジェクトの長さ。

フィールド名	サイズ	値

クエリー	バイト配列	JSON(UTF-8)でエンコードされたクエリーオブジェクトを含むバイトアレイ。ペイロードの例：
		
		{
		"query":"From Entity where field1:'value1'",
		"offset": 12,
		"max-results": 1000,
		"query-mode": "FETCH"
		}
		
		詳細は以下のようになります。
		
		query: the Ickle query String.
		offset: the index of the first result to return.
		max_results: the maximum number of results to return.
		query_mode: the indexed query mode. Either FETCH or BROADCAST. FECTH is the default.

クエリー応答(0x20):

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
応答ペイロードの長さ	vInt	UTF-8 でエンコードされた応答オブジェクトの長さ

フィールド名	サイズ	値

応答ペイロード	バイト配列	JSON でエンコードされた応答オブジェクトを含むバイト配列。前のフィールドで長さが指定されています。ペイロードの例：
		
		{
		"total_results":801,
		"hits":[
		{
		"hit":{
		"field1":565,
		"field2":"value2"
		}
		},
		{
		"hit":{
		"field1":34,
		"field2":"value22"
		}
		}
		]
		}
		
		詳細は以下のようになります。
		
		total_results: the total number of results of the query.
		hits: an ARRAY of OBJECT representing the results.
		hit: each OBJECT above contain another OBJECT in the "hit" field, containing the result of the query, in JSON format.

また、このバージョンには Hotgitops トランザクションに新しい 3 つの操作が導入されています。

- ***Request V2 を準備します。これにより、新しいパラメーターを要求に追加します。応答は同じままになります。***
- ***Dest Transaction Request: サーバーのトランザクション情報を削除します。***
- ***In-Doubt Transactions Request の取得： インダウト状態のトランザクションの Xid をすべて取得します。***

要求 V2 の準備

要求(0x7D):

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
Xid	XID	トランザクション ID(XID)
OnePhaseCommit	byte	**1** に設定すると、サーバーは利用可能な場合は 1 フェーズコミットを使用します（XA のみ）。
recoverable	byte	このトランザクションでリカバリーを許可するには、**1** に設定します。
タイムアウト	Long	アイドルタイムアウト（ミリ秒単位）。トランザクションがリカバリー不可能な場合(**Recoverable=0)**、トランザクションがアイドル状態であった場合、サーバーはトランザクションをロールバックします。
キー数	vInt	キー数
各キーについて（キーは一意でなければなりません）		
キーの長さ	vInt	キーの長さ。vInt のサイズは最大 5 バイトで、理論では **Integer.MAX_VALUE** よりも大きな数値を生成することができる点に注意してください。ただし、Java では **Integer.MAX_VALUE** を超える単一のアレイを作成できないため、プロトコルは vInt アレイの長さを **Integer.MAX_VALUE** に制限します。
キー	バイト配列	キーを含むバイトアレイ
Control Byte	Byte	以下の意味を持つビットセット。 0x01 = **NOT_READ** 0x02 = **NON_EXISTING** 0x04 = **REMOVE_OPERATION** を同時に設定できないことに注意してください。
バージョン読み取り	Long	読み込まれたバージョン。**NOT_READ** および **NON_EXISTING** が存在しない場合のみ送信されます。
TimeUnits	Byte	ライフスパンの時間単位（最初の 4 ビット）および maxIdle（最後 4 ビット）。デフォルトのサーバーの有効期限には、特別なユニット **DEFAULT** と **INFINITE** を使用できます。使用できる値： 0x00 = **SECONDS** 0x01 = **MILLISECONDS** 0x02 = **NANOSECONDS** 0x03 = **MICROSECONDS** 0x04 = **MINUTES** 0x05 = **HOURS** 0x06 = **DAYS** 0x 07 = **DEFAULT** 0x08 = **INFINITE** のみが送信され、**REMOVE_OPERATION** が設定されていない場合のみ送信されます。

フィールド名	サイズ	値
有効期間	vLong	エントリーの有効期間。時間の単位が **DEFAULT** または **INFINITE _OPERATION** が設定されていない場合にのみ送信されます。
最大 ID	vLong	各エントリーがキャッシュからエビクトされる前にアイドル状態になる期間。時間の単位が **DEFAULT** または **INFINITE _OPERATION** が設定されていない場合にのみ送信されます。
値の長さ	vInt	値の長さ。**REMOVE_OPERATION** が設定されていない場合にのみ送信されます。
値	byte-array	保存する値。**REMOVE_OPERATION** が設定されていない場合にのみ送信されます。

応答(0x7E)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
XA 戻りコード	vInt	準備応答を表す XA コード。 **XA_OK(0**)、**XA_RDONLY(3)** またはエラーコードのいずれかを指定できます（ **XaException**を参照）。 応答の状態が **Successful** と異なる場合、このフィールドは表示されません。

トランザクションを忘れる

要求(0x79)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー
Xid	XID	トランザクション ID(XID)

応答(0x7A)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー

インダウト状態のトランザクションの取得

要求(0x7B)

フィールド名	サイズ	値
ヘッダー	variable	要求ヘッダー

応答(0x7C)

フィールド名	サイズ	値
ヘッダー	variable	応答ヘッダー
Xid の数	vInt	応答の Xid の数
各エントリーに対して以下を行います。		
Xid	XID	トランザクション ID(XID)

20.6.15. hotgitops Hash Functions

Red Hat Data Grid は一貫性のあるハッシュ機能を使用して、ハッシュホイールにノードを置き、同じ wheel にエントリーのキーを配置して、エントリーの場所を判断します。

Infinispan 4.2 以前では、ハッシュ領域は 10240 にハードコーディングされましたが 5.0 以降、ハッシュ領域は Integer.MAX_INT です。Hotgitops クライアントはデフォルトで特定のハッシュスペースを想定してはならないことに注意してください。ハッシュトポロジーの変更が検出されたときに毎回、この値は Hot336 プロトコルを介してクライアントに返信されます。

Hotgitops プロトコルを介して Red Hat Data Grid と対話する場合は、プラットフォームに依存しない動作を確保するために、キー（および値）がバイト配列であることが義務付けられます。そのため、バックエンド上のハッシュディストリビューションを認識している smart-clients は、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計算できる必要があり

ます。このため、Red Hat Data Grid で使用されるハッシュ関数はバージョン化され、文書化されるため、必要に応じて Java 以外のクライアントが再実装できます。

使用するハッシュ関数のバージョンは、ハッシュ関数のバージョンパラメーターとして HotTEMPLATES プロトコルにあります。

1. バージョン 1（単一のバイト、0x01）は、Austin Appleby の MurmurHash 2.0 アルゴリズム、高速で非暗号化のハッシュが、優れたディストリビューション、競合、および有効な動作を示す高速で非暗号化ハッシュに基づいています。使用するアルゴリズムの特定バージョンは、大規模な CPU アーキテクチャーとリトルエンディアン CPU アーキテクチャーの両方で一貫した動作を可能にする、若干遅く、エンディアンに依存しないバージョンです。また、Red Hat Data Grid のバージョンは、ハッシュを -1 としてハードコーディングします。アルゴリズムの詳細については、Austin Appleby の MurmurHash 2.0 ページ を参照してください。その他の実装は、Wikipedia に記載されています。このハッシュ関数は、Infinispan 4.2.1 まで Hotgitops サーバーによって使用されるデフォルトのものです。Infinispan 5.0 以降、サーバーはハッシュバージョン 1 を使用しません。Infinispan 9.0 以降、クライアントはハッシュバージョン 1 を無視します。

2. バージョン 2（単一バイト、0x02）、Infinispan 5.0 以降、Austin Appleby の MurmurHash 3.0 アルゴリズム を基にした新しいハッシュ関数がデフォルトで使用されます。ハッシュ関数の詳細は、この wiki を参照してください。2.0 と比較すると、パフォーマンスが向上します。Infinispan 7.0 以降、サーバーは HotRod 1.x クライアントにバージョン 2 のみを使用します。

3. バージョン 3（単一バイト、0x03）、Infinispan 7.0 以降、デフォルトで新しいハッシュ関数が使用されます。この関数は wiki に基づいていますが、サーバーの ConsistentHash で使用されるハッシュセグメントも認識します。

20.6.16. ホットエンクト管理タスク

管理操作は、よく知られたタスクのセットを持つ Exec 操作によって処理されます。管理タスクは、以下のルールに従って名前が付けられます。

@@context@name

パラメーターはすべて UTF-8 でエンコードされた文字列です。パラメーターは各タスクに固有のものですが、すべてのコマンドに共通する flags パラメーターを除きます。flags パラメーターには、コマンドの動作に影響を与える可能性があるスペースで区切られた値 0 個以上の値が含まれます。以下の表には、現在利用可能なすべてのフラグをまとめています。

管理タスクは、JSON 文字列として表示される操作の結果を返します。

表20.26 FLAGS

フラグ	説明
permanent	コマンドによる影響のリクエストは、サーバーの設定に永続化されます。サーバーがリクエストに準拠できない場合は、操作全体が失敗し、エラーが発生します。

20.6.16.1. 管理タスク

表20.27 @@cache@create

パラメーター	説明	必須
name	作成するキャッシュの名前。	はい
template	新規キャッシュに使用するキャッシュ設定テンプレートの名前。	▮
設定	使用するキャッシュ設定の XML 宣言。	▮
flags	上記のフラグ表を参照してください。	▮

表20.28 @@cache@remove

パラメーター	説明	必須
name	削除するキャッシュの名前。	はい
flags	上記のフラグ表を参照してください。	▮

@@cache@names

キャッシュ名を文字列の JSON 配列として返します（例： ["cache1", "cache2"]）。

表20.29 @@cache@reindex

パラメーター	説明	必須
name	インデックス化するキャッシュの名前。	はい

パラメーター	説明	必須
flags	上記のフラグ表を参照してください。	いいえ、すべてのフラグは無視されます。

20.7. JAVA HOTGITOPS クライアント

hotgitops はバイナリー言語に依存しないプロトコルです。この記事では、Java クライアントが Hotgitops プロトコルを介してサーバーと対話する方法を説明します。Java で書かれたプロトコルの参照実装は、すべての Red Hat Data Grid ディストリビューションにあります。この記事では、この java クライアントの機能に重点を置いています。

ヒント

その他のクライアントの検索さまざまな言語で書かれたクライアントについては、この Web サイトを参照してください。

20.7.1. 設定

Java HotTEMPLATES クライアントは、設定ファイルを介してプログラムおよび外部の両方を設定できます。

以下のコードスニペットは、利用可能な Java Fluent API を使用したクライアントインスタンスの作成を示しています。

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
    = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.tcpNoDelay(true)
  .statistics()
    .enable()
    .jmxDomain("org.infinispan")
  .addServer()
    .host("127.0.0.1")
    .port(11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

利用可能な設定オプションの完全なリファレンスは、ConfigurationBuilder 's javadoc を参照してください。

プロパティーファイルを使用して **Java Hotfaillock** クライアントを設定することもできます。以下に例を示します。

```
# Hot Rod client configuration
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.marshaller = org.infinispan.commons.marshall.jboss.GenericJBossMarshaller
infinispan.client.hotrod.async_executor_factory =
org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory
infinispan.client.hotrod.default_executor_factory.pool_size = 1
infinispan.client.hotrod.hash_function_impl.2 =
org.infinispan.client.hotrod.impl.consistenthash.ConsistentHashV2
infinispan.client.hotrod.tcp_no_delay = true
infinispan.client.hotrod.tcp_keep_alive = false
infinispan.client.hotrod.request_balancing_strategy =
org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy
infinispan.client.hotrod.key_size_estimate = 64
infinispan.client.hotrod.value_size_estimate = 512
infinispan.client.hotrod.force_return_values = false

## Connection pooling configuration
maxActive=-1
maxIdle = -1
whenExhaustedAction = 1
minEvictableIdleTimeMillis=300000
minIdle = 1
```

その後、プロパティーファイルは **RemoteCacheManager** のコンストラクターのいずれかに渡されます。プロパティー置換を使用すると、実行時に値を *Java システムプロパティー* に置き換えることができます。

```
infinispan.client.hotrod.server_list = ${server_list}
```

上記の例では、**infinispan.client.hotrod.server_list** プロパティーの値を **server_list** *Java* システムプロパティーの値に拡張します。つまり、値は **jboss.bind.address.management** というシステムプロパティーから取得し、**127.0.0.1** を使用してください。

ヒント

hotrod-client.properties をクラスパス以外の場所で使用する場合は、以下を実行します。

```
ConfigurationBuilder b = new ConfigurationBuilder();
Properties p = new Properties();
try(Reader r = new FileReader("/path/to/hotrod-client.properties")) {
  p.load(r);
  b.withProperties(p);
}
RemoteCacheManager rcm = new RemoteCacheManager(b.build());
```

プロパティーファイルに使用できる設定オプションの完全なリファレンスは、「リモートクライアント設定 Java」を参照してください。

20.7.2. 認証

サーバーが認証を設定している場合は、適切にクライアントを設定する必要があります。サーバーで有効な mechs に応じて、クライアントは必要な情報を提供する必要があります。

20.7.2.1. DIGEST-MD5

DIGEST-MD5 は単純なユーザー名/パスワード認証のシナリオで推奨される方法です。サーバーでデフォルトのレルム("ApplicationRealm")を使用している場合は、以下のように認証情報を指定する必要があります。

DIGEST-MD5 認証を使用した hotgitops クライアント設定

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         .username("myuser")
         .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

20.7.2.2. PLAIN

接続も暗号化されていない限り、PLAIN メカニズムは推奨されません。これは、クリアテキストのパスワードがネットワーク経由で送信される可能性があるためです。

DIGEST-MD5 認証を使用した hotgitops クライアント設定

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .saslMechanism("PLAIN")
         .username("myuser")
         .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

20.7.2.3. EXTERNAL

EXTERNAL メカニズムは、認証情報を明示的に提供せず、クライアント証明書をアイデンティティーとして使用する点で特殊です。これを機能させるには、TrustStore （サーバー証明書を検証するため）に加えて KeyStore を提供する必要があります（クライアント証明書を指定するため）。

EXTERNAL 認証（クライアント証明書）でのホットdb クライアント設定

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         // TrustStore is a KeyStore which contains part of the server certificate chain (e.g. the
CA Root public cert)
         .trustStoreFileName("/path/to/truststore")
         .trustStorePassword("truststorepassword".toCharArray())
         // KeyStore containing this client's own certificate
         .keyStoreFileName("/path/to/keystore")
         .keyStorePassword("keystorepassword".toCharArray())
```

```
        .authentication()
           .saslMechanism("EXTERNAL");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

詳細は、以下の「暗号化」セクションを参照してください。

20.7.2.4. GSSAPI(Kerberos)

GSSAPI/Kerberos はより複雑なセットアップを必要としますが、集中認証サーバーでは多くの企業で使用されます。Kerberos で正常に認証するには、LoginContext を作成する必要があります。これにより、サービスとの認証に使用するトークンとして使用される TGT(Ticket Granting Ticket)を取得します。

ログイン設定ファイルでログインモジュールを定義する必要があります。

gss.conf

```
GssExample {
   com.sun.security.auth.module.Krb5LoginModule required client=TRUE;
};
```

IBM JDK を使用している場合、上記は以下のようになります。

gss-ibm.conf

```
GssExample {
   com.ibm.security.auth.module.Krb5LoginModule required client=TRUE;
};
```

以下のシステムプロパティーを設定する必要もあります。

java.security.auth.login.config=gss.conf

java.security.krb5.conf=/etc/krb5.conf

krb5.conf ファイルはお使いの環境に依存しており、KDC を参照する必要があります。kinit を使用して Kerberos 経由で認証できることを確認します。

次に、以下のようにクライアントを設定します。

hotgitops クライアント GSSAPI 設定

```
LoginContext lc = new LoginContext("GssExample", new BasicCallbackHandler("krb_user",
"krb_password".toCharArray()));
lc.login();
Subject clientSubject = lc.getSubject();

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .enable()
         .serverName("infinispan-server")
         .saslMechanism("GSSAPI")
         .clientSubject(clientSubject)
         .callbackHandler(new BasicCallbackHandler());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

簡潔に、同じコールバックハンドラーを使用してクライアントサブジェクトを取得し、SASL GSSAPI mech で認証を処理するのに、異なるコールバックが実際に呼び出されます。NameCallback と PasswordCallback はクライアントサブジェクトを構築する必要がありますが、AuthorizeCallback は SASL 認証中に呼び出されます。

20.7.2.5. カスタム CallbackHandlers

上記の例で、Hot336 クライアントは、提供される認証情報を SASL メカニズムに提供するデフォルトの CallbackHandler を設定します。高度なシナリオでは、独自のカスタム CallbackHandler を提供する必要がある場合があります。

コールバックを使用した認証を使用したホット Modules クライアント設定

```
public class MyCallbackHandler implements CallbackHandler {
  final private String username;
  final private char[] password;
  final private String realm;

  public MyCallbackHandler(String username, String realm, char[] password) {
    this.username = username;
    this.password = password;
    this.realm = realm;
  }

  @Override
  public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
    for (Callback callback : callbacks) {
      if (callback instanceof NameCallback) {
        NameCallback nameCallback = (NameCallback) callback;
        nameCallback.setName(username);
      } else if (callback instanceof PasswordCallback) {
        PasswordCallback passwordCallback = (PasswordCallback) callback;
        passwordCallback.setPassword(password);
      } else if (callback instanceof AuthorizeCallback) {
        AuthorizeCallback authorizeCallback = (AuthorizeCallback) callback;
        authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationID().equals(
            authorizeCallback.getAuthorizationID()));
      } else if (callback instanceof RealmCallback) {
        RealmCallback realmCallback = (RealmCallback) callback;
        realmCallback.setText(realm);
      } else {
        throw new UnsupportedCallbackException(callback);
      }
    }
  }
}

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .enable()
```

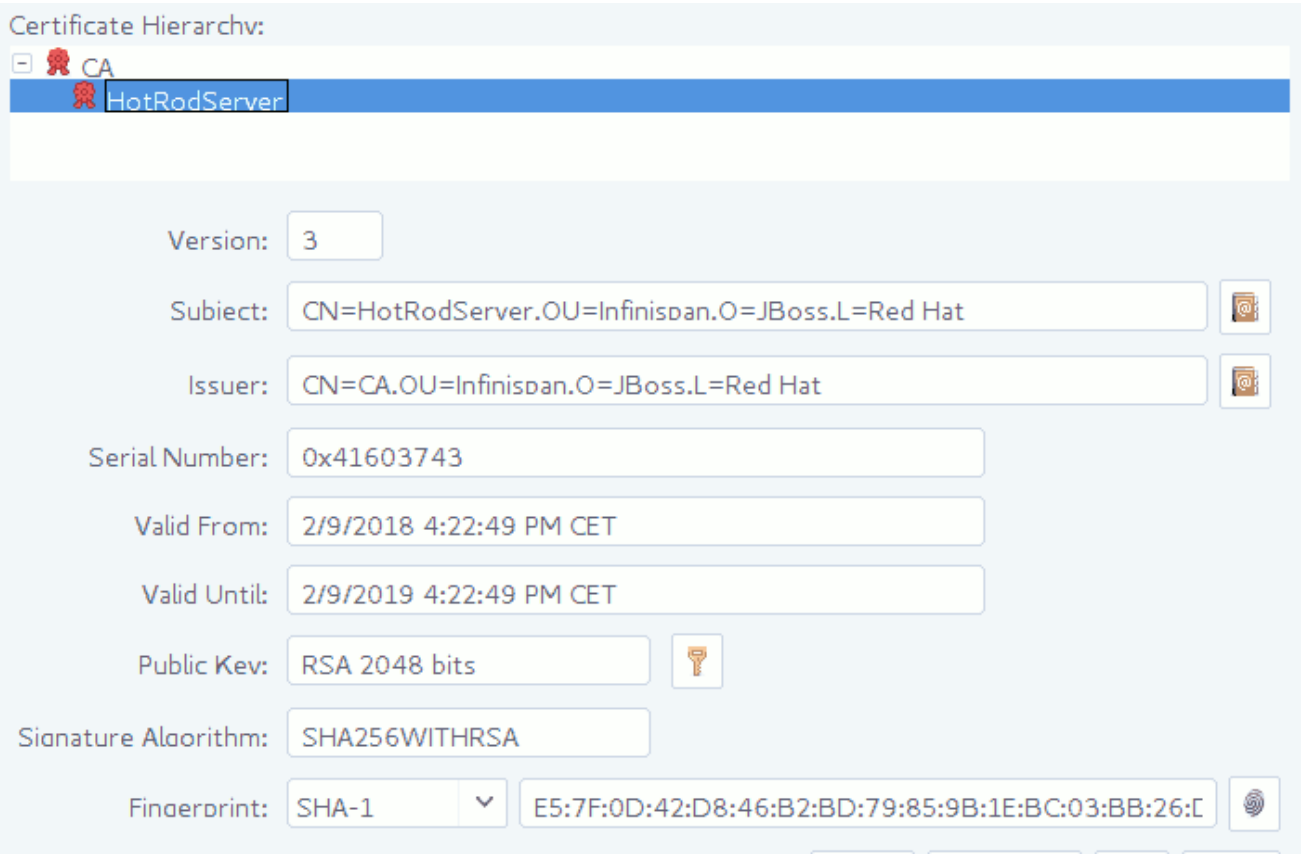
```
        .serverName("myhotrodserver")
        .saslMechanism("DIGEST-MD5")
        .callbackHandler(new MyCallbackHandler("myuser", "ApplicationRealm",
"qwer1234!".toCharArray()));
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

CallbackHandler を処理する必要がある実際のコールバックのタイプは mech 固有のため、上記の例は簡単な例です。

20.7.3. 暗号化

暗号化は TLS/SSL を使用するため、適切なサーバー証明書チェーンを設定する必要があります。通常、証明書チェーンは以下のようになります。

図20.6 証明書チェーン



上記の例では、「HotRodServer」の証明書を発行した認証局「CA」 が 1 つあります。クライアントがサーバーを信頼するには、最低でも上記のチェーンの一部が必要です（通常は、CA のパブリック証明書のみ）。この証明書はキーストアに配置する必要があります。これはクライアントの TrustStore として使用し、以下のように使用する必要があります。

TLS（サーバー証明書）を使用した hotgitops クライアント設定

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         // TrustStore is a KeyStore which contains part of the server certificate chain (e.g. the
CA Root public cert)
         .trustStoreFileName("/path/to/truststore")
         .trustStorePassword("truststorepassword".toCharArray());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

20.7.3.1. SNI

サーバーは、TLS/SNI サポート(Server Name Indication)に対応するよう設定されている可能性があります。つまり、サーバーは複数のアイデンティティーを提示していることを意味します（特に別のキャッシュコンテナーにバインドされます）。クライアントは、その名前を指定して、接続するアイデンティティーを指定できます。

SNI（サーバー証明書）を使用した hotgitops クライアント設定

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         .sniHostName("myservername")
         // TrustStore is a KeyStore which contains part of the server certificate chain (e.g. the
CA Root public cert)
         .trustStoreFileName("/path/to/truststore")
         .trustStorePassword("truststorepassword".toCharArray());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

20.7.3.2. クライアント証明書

上記の設定では、クライアントはサーバーを信頼します。セキュリティーを強化するために、サーバー管理者が相互信頼に有効な証明書を提供できるようにサーバーをセットアップしている可能性があります。このような設定には、クライアントが独自の証明書を提示する必要があります。これは、通常はサーバーと同じ認証局により発行されます。この証明書はキーストアに保存され、以下のように使用する必要があります。

TLS（サーバー証明書およびクライアント証明書）を使用した hotgitops クライアント設定

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         // TrustStore is a KeyStore which contains part of the server certificate chain (e.g. the
CA Root public cert)
         .trustStoreFileName("/path/to/truststore")
         .trustStorePassword("truststorepassword".toCharArray())
         // KeyStore containing this client's own certificate
         .keyStoreFileName("/path/to/keystore")
         .keyStorePassword("keystorepassword".toCharArray())
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

KeyStores の詳細は、KeyTool ドキュメントを参照してください。さらに、KeyStore Explorer は KeyStore を簡単に管理するための優れた GUI ツールです。

20.7.4. Basic API

以下は、Java Hotgitops クライアントを使用して、クライアント API を使用して HotTEMPLATES サーバーから情報を保存または取得するためにクライアント API を使用する方法に関するサンプルのコードスニペットです。HotTEMPLATES サーバーがデフォルトの場所(localhost:11222)にバインドされていることを前提としています(localhost:11222)。

```
//API entry point, by default it connects to localhost:11222
CacheContainer cacheContainer = new RemoteCacheManager();

//obtain a handle to the remote default cache
Cache<String, String> cache = cacheContainer.getCache();

//now add something to the cache and make sure it is there
cache.put("car", "ferrari");
```

```
assert cache.get("car").equals("ferrari");

//remove the data
cache.remove("car");
assert !cache.containsKey("car") : "Value must have been removed!";
```

クライアント API は、ローカル API をマッピングする： RemoteCacheManager は DefaultCacheManager に対応します（どちらも CacheContainer を実装します）。この一般的な API を使用すると、Hotset によるローカル呼び出しからリモート呼び出しへの簡単な移行が容易になります。すべてのニーズは DefaultCacheManager と RemoteCacheManager の間で切り替えられます。これは、両方を継承する一般的な CacheContainer インターフェースによってさらに簡素化されます。

20.7.5. RemoteCache(.keySet|.entrySet|.values)

コレクションメソッド keySet、entrySet、および values はリモートキャッシュによってサポートされます。つまり、すべてのメソッドが RemoteCache に戻されるということです。これは、さまざまなキー、エントリ、または値を遅延して取得でき、ユーザーが望まない場合にそれらすべてを一度にクライアントメモリに保存する必要がないため便利です。これらのコレクションは、addおよびaddAllはサポートされていませんが、他のすべてのメソッドはサポートされているというMap仕様に準拠しています。

注記の 1 つとして、Iterator.remove メソッドおよび Set.remove メソッドまたは Collection.remove メソッドには、操作するサーバーに 1 つ以上のラウンドトリップが必要です。RemoteCache Javadoc を確認して、これらの方法と他のメソッドの詳細を確認することができます。

イテレーターの使用

これらのコレクションの iterator メソッドは、以下で説明されている retrieveEntries を内部で使用します。retrieveEntries がバッチサイズの引数を取る場合。これをイテレータに提供する方法はありません。このようなバッチサイズは、RemoteCacheManager の設定時にシステムプロパティー infinispan.client.hotrod.batch_size を使用するか、ConfigurationBuilder を使用して設定できます。

また、返される retrieveEntries イテレーターは Closeable です。keySet、entrySet、および values からのイテレーターは AutoCloseable バリアントを返します。したがって、これらの「Iterator」の使用が終了したら、常に閉じる必要があります。

```
try (CloseableIterator<Entry<K, V>> iterator = remoteCache.entrySet().iterator) {
 ...
}
```

バッキングコレクションではなく、ディープコピーが必要な場合

以前のバージョンの RemoteCache では、keySet のディープコピーの取得が許可されました。これは新しいバッキングマップでも可能です。コンテンツを自分でコピーするだけです。また、これまでサポートしていなかったentrySetおよびvaluesを使用してこれを行うこともできます。

```
Set<K> keysCopy = remoteCache.keySet().stream().collect(Collectors.toSet());
```

鍵が多数ある場合があり、クライアントで OutOfMemoryError が発生する可能性があるため、細心に注意を払ってください。

```
Set keys = remoteCache.keySet();
```

20.7.6. リモートイテレーター

または、メモリーが問題（バッチサイズが異なる）か、サーバー側のフィルタリングまたは変換を行う場合は、リモートのイテレーター API を使用してサーバーからエントリーを取得します。このメソッドを使用すると、エントリーの全プロパティーが必要なければ、取得されたエントリーや変換された値も制限できます。

```
// Retrieve all entries in batches of 1000
int batchSize = 1000;
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(null,
batchSize)) {
   while(iterator.hasNext()) {
     // Do something
   }
}

// Filter by segment
Set<Integer> segments = ...
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(null,
segments, batchSize)) {
   while(iterator.hasNext()) {
     // Do something
   }
}

// Filter by custom filter
try (CloseableIterator<Entry<Object, Object>> iterator =
remoteCache.retrieveEntries("myFilterConverterFactory", segments, batchSize)) {
   while(iterator.hasNext()) {
     // Do something
   }
}
```

カスタムフィルターを使用するには、最初にサーバーにデプロイする必要があります。以下の手順を実施します。

- ファクトリーの名前を含む @NamedFactory アノテーションを付けて、KeyValueFilterConverterFactory を拡張するフィルターのファクトリーを作成します。以下に例を示します。

```
import java.io.Serializable;

import org.infinispan.filter.AbstractKeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverterFactory;
import org.infinispan.filter.NamedFactory;
import org.infinispan.metadata.Metadata;

@NamedFactory(name = "myFilterConverterFactory")
public class MyKeyValueFilterConverterFactory implements KeyValueFilterConverterFactory {

  @Override
  public KeyValueFilterConverter<String, SampleEntity1, SampleEntity2> getFilterConverter() {
    return new MyKeyValueFilterConverter();
  }
  // Filter implementation. Should be serializable or externalizable for DIST caches
  static class MyKeyValueFilterConverter extends AbstractKeyValueFilterConverter<String,
SampleEntity1, SampleEntity2> implements Serializable {
    @Override
    public SampleEntity2 filterAndConvert(String key, SampleEntity1 entity, Metadata
metadata) {
      // returning null will case the entry to be filtered out
      // return SampleEntity2 will convert from the cache type SampleEntity1
    }

    @Override
    public MediaType format() {
      // returns the MediaType that data should be presented to this converter.
      // When ommitted, the server will use "application/x-java-object".
      // Returning null will cause the filter/converter to be done in the storage format.
    }
  }
}
```

- META-INF/services/org.infinispan.filter.KeyValueFilterConverterFactory ファイルで jar を作成し、その中にフィルターファクトリークラス実装の完全修飾クラス名を作成します。

- オプション： フィルターがカスタムのキー/値クラスを使用する場合は、フィルターがキーや値のインスタンスを正しくアンマーシャリングできるように JAR に含める必要があります。

- JAR ファイルを Red Hat Data Grid Server にデプロイします。

20.7.7. Versioned API

RemoteCacheManager は、リモートクラスターの名前付きキャッシュまたはデフォルトキャッシュへのハンドルを表す RemoteCache インターフェースのインスタンスを提供します。API では、バージョン化された API と呼ばれる新しいメソッドも追加する Cache インターフェースを拡張します。以下の API リンクの例を確認してください。#server_hotrod_failover[motivation を理解するには、このセクションを必ず読んでください。

コードスニペット bello は、以下のバージョン管理されたメソッドの使用を示しています。

```
// To use the versioned API, remote classes are specifically needed
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
RemoteCache<String, String> cache = remoteCacheManager.getCache();

remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");

// removal only takes place only if the version has not been changed
// in between. (a new version is associated with 'car' key on each change)
assert remoteCache.remove("car", valueBinary.getVersion());
assert !cache.containsKey("car");
```

同様に、以下を置き換えます。

```
remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");
assert remoteCache.replace("car", "lamborghini", valueBinary.getVersion());
```

バージョン管理された操作の詳細は、「RemoteCache 's javadoc」を参照してください。

20.7.8. Async API

これは、Red Hat Data Grid コアの「borrowed」です。これについては、主に「」で説明しています。

20.7.9. ストリーミング API

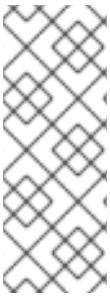
大規模なオブジェクトを送受信すると、クライアントとサーバー間でそれらをストリーミングすることが理にかなっています。Streaming API は、上記の 「Hotでベーシック API および Hot336

Versioned API と同様にメソッドを実装しますが、値をパラメーターとして取得する代わりに、InputStream および OutputStream のインスタンスを返します。以下の例は、潜在的な大きなオブジェクトを書き込む方法を示しています。

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
OutputStream os = streamingCache.put("a_large_object");
os.write(...);
os.close();
```

ストリーミングを使用したこのようなオブジェクトの読み取り：

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
for(int b = is.read(); b >= 0; b = is.read()) {
  ...
}
is.close();
```



注記

ストリーミング API はマーシャリング/アンマーシャリングを値に適用し ません。このため、この状況を検出するための独自のマーシャラーを提供しない限り、ストリーミング API と非ストリーミング API の両方を同時に使用する同じエントリーにアクセスできません。

RemoteStreamingCache.get(K key) メソッドによって返される InputStream は VersionedMetadata インターフェースを実装するため、バージョンおよび有効期限の情報を取得できます。

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
int version = ((VersionedMetadata) is).getVersion();
for(int b = is.read(); b >= 0; b = is.read()) {
  ...
}
is.close();
```

注記

条件付き書き込みメソッド(putIfAbsent,replace)は、値がサーバーへ完全に送信された（つまり OutputStream で close （）メソッドが呼び出されたときに）実際の条件チェックを実行します。

20.7.10. イベントリスナーの作成

Java Hot Rod クライアントは、リスナーを登録して cache-entry レベルのイベントを受信することができます。キャッシュエントリの作成、変更、および削除されたイベントがサポートされています。

クライアントリスナーの作成は、異なるアノテーションとイベントクラスが使用される点を除き、組み込みリスナーと非常に似ています。受信した各イベントを出力するクライアントリスナーの例を次に示します。

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

@ClientListener
public class EventPrintListener {

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

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

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

}
```

ClientCacheEntryCreatedEvent および ClientCacheEntryModifiedEvent インスタンスは、影響を受けるキーとエントリーのバージョンに関する情報を提供します。このバージョンは、replaceWithVersion、removeWithVersion などのサーバー上で条件付き操作を呼び出すために使用できます。

ClientCacheEntryRemovedEvent イベントは、削除操作が成功した場合にのみ送信されます。つまり、削除オペレーションが呼び出されても、エントリーが見つからないか、エントリを削除する必要がない場合は、イベントが生成されません。エントリーが削除されていなくても、削除されたイベントに関心のあるユーザーは、このようなイベントを生成するイベントのカスタマイズロジックを開発できます。詳細は、クライアントイベントのカスタマイズセクションを参照してください。

ClientCacheEntryCreatedEvent、ClientCacheEntryModifiedEvent、および ClientCacheEntryRemovedEvent のすべてのイベントインスタンスも、トポロジーの変更が原因でこれを引き起こした書き込みコマンドを再度実行する場合に true を返す boolean isCommandRetried() メソッドも提供します。これは、このイベントが重複したか、別のイベントが破棄されて置き換えられ

た記号になります（例: ClientCacheEntryModifiedEvent が ClientCacheEntryCreatedEvent に置き換えます）。

クライアントリスナーの実装を作成したら、サーバーに登録する必要があります。これを行うには、以下を実行します。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());
```

20.7.11. イベントリスナーの削除

クライアントイベントリスナーが必要ない場合は、以下を削除できます。

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

20.7.12. イベントのフィルタリング

クライアントにイベントが殺到するのを防ぐために、ユーザーはフィルタリング機能を提供して、特定のクライアントリスナーに対し、サーバーによって発生するイベントの数を制限できます。フィルターを有効にするには、フィルターインスタンスを生成するキャッシュイベントフィルターファクトリーを作成する必要があります。

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-filter")
class StaticCacheEventFilterFactory implements CacheEventFilterFactory {
  @Override
  public CacheEventFilterFactory<Integer, String> getFilter(Object[] params) {
    return new StaticCacheEventFilter();
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventFilter implements CacheEventFilter<Integer, String>, Serializable {
  @Override
  public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
      String newValue, Metadata newMetadata, EventType eventType) {
    if (key.equals(1)) // static key
      return true;

    return false;
  }
}
```

上記で定義されたキャッシュイベントフィルターファクトリーインスタンスは、キーが 1 であるものを除くすべてのエントリーを静的にフィルターするインスタンスを作成します。

このキャッシュイベントフィルターファクトリにリスナーを登録できるようにするには、ファクトリに一意の名前を付ける必要があり、Hot Rodサーバーに名前とキャッシュイベントフィルターファクトリインスタンスを接続する必要があります。カスタムフィルターで Red Hat Data Grid Server をプラグインするには、以下の手順を行います。

1. フィルター実装で JAR ファイルを作成します。

2. オプション： キャッシュがカスタムキー/値クラスを使用する場合は、コールバックを JAR に含める必要があります。これにより、コールバックは正しくアンマーシャリングされたキーや値インスタンス/または値インスタンスで実行できます。クライアントリスナーでuseRawDataが有効になっている場合、コールバックのキー/値インスタンスはバイナリ形式で提供されるため、これは必要ありません。

3. JAR ファイル内で META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory ファイルを作成し、フィルタークラス実装の完全修飾クラス名を作成します。

4. JAR ファイルを Red Hat Data Grid Server にデプロイします。

その上で、ファクトリーの名前を @ClientListener アノテーションに追加して、クライアントリスナーをこのキャッシュイベントフィルターファクトリーにリンクする必要があります。

```
@ClientListener(filterFactoryName = "static-filter")
public class EventPrintListener { ... }
```

そして、リスナーをサーバーに登録します。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());
```

リスナーの登録時に提供されるパラメーターに基づいてフィルターする動的フィルターインスタンスも使用できます。フィルターは、フィルターファクトリーが受信したパラメーターを使用して、このオプションを有効にします。以下に例を示します。

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventFilter;

class DynamicCacheEventFilterFactory implements CacheEventFilterFactory {
  @Override
  public CacheEventFilter<Integer, String> getFilter(Object[] params) {
    return new DynamicCacheEventFilter(params);
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class DynamicCacheEventFilter implements CacheEventFilter<Integer, String>, Serializable {
  final Object[] params;

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

  @Override
  public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
      String newValue, Metadata newMetadata, EventType eventType) {
    if (key.equals(params[0])) // dynamic key
      return true;

    return false;
  }
}
```

リスナーの登録時に、フィルタリングに必要な動的パラメーターが提供されます。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), new Object[]{1}, null);
```

警告

フィルターインスタンスは、クラスターにデプロイされるときにマーシャル可能である必要があります。これにより、リスナーの登録先の別のノードで生成された場合でも、イベントが生成される際にフィルターが適切に行われるようにします。それらをマーシャリング可能にするには、Serializable、Externalizableを拡張するか、カスタムExternalizerを提供します。

20.7.13. 通知のスキップ

サーバーからイベント通知を取得しなくても、リモート API メソッドを呼び出してオペレーション

を実行する際に、SKIP_LISTENER_NOTIFICATION フラグを含めます。たとえば、値を作成または変更する際のリスナーの通知を防ぐには、フラグを以下のように設定します。

```
remoteCache.withFlags(Flag.SKIP_LISTENER_NOTIFICATION).put(1, "one");
```

重要

SKIP_LISTENER_NOTIFICATION フラグは、Red Hat Data Grid バージョン 7.3.2 から入手できます。フラグを設定する前に、Red Hat Data Grid クライアントおよび Hotgitops クライアントの両方をこのバージョン以上にアップグレードする必要があります。

20.7.14. イベントのカスタマイズ

デフォルトで生成されるイベントには、イベントを関連させるのに十分な情報が含まれていますが、送信コストを削減するために情報を詰め込みすぎないようにしています。必要に応じて、イベントに含まれる情報は、値などの詳細情報を含むか、より少ない情報を含めるためにカスタム化できます。このカスタマイズは、CacheEventConverterFactory によって生成された CacheEventConverter インスタンスを使用して行われます。

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

@NamedFactory(name = "static-converter")
class StaticConverterFactory implements CacheEventConverterFactory {
  final CacheEventConverter<Integer, String, CustomEvent> staticConverter = new
StaticCacheEventConverter();
  public CacheEventConverter<Integer, String, CustomEvent> getConverter(final Object[]
params) {
    return staticConverter;
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
  public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata, String
newValue, Metadata newMetadata, EventType eventType) {
    return new CustomEvent(key, newValue);
  }
}

// Needs to be Serializable, Externalizable or marshallable with Infinispan Externalizers
// regardless of cluster or local caches
static class CustomEvent implements Serializable {
  final Integer key;
  final String value;
```

```
    CustomEvent(Integer key, String value) {
      this.key = key;
      this.value = value;
    }
  }
```

上記の例では、コンバーターは、イベント内の値とキーを含む新しいカスタムイベントを生成します。これにより、デフォルトのイベントと比べて大量のイベントペイロードが発生しますが、フィルターと組み合わせると、ネットワークの帯域幅コストを減らすことができます。

警告

コンバーターのターゲットタイプは、Serializable または Externalizable のいずれかである必要があります。この特定のコンバーターの場合、デフォルトの Hot Rod クライアントマーシャラーではサポートされないため、デフォルトで Externalizer を提供しません。

カスタムイベントの処理には、以前に実証されたものに対して、若干異なるクライアントリスナーの実装が必要になります。より正確な状態に戻すには、ClientCacheEntryCustomEvent インスタンスを処理する必要があります。

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

@ClientListener
public class CustomEventPrintListener {

  @ClientCacheEntryCreated
  @ClientCacheEntryModified
  @ClientCacheEntryRemoved
  public void handleCustomEvent(ClientCacheEntryCustomEvent<CustomEvent> e) {
    System.out.println(e);
  }

}
```

コールバックで受け取ったClientCacheEntryCustomEventは、getEventDataメソッドを介してカスタムイベントを公開し、getTypeメソッドは、生成されたイベントがキャッシュエントリの作成、変更、または削除の結果であったかどうかに関する情報を提供します。

フィルタリングと同様に、リスナーをこのコンバーターファクトリに登録できるようにするには、

ファクトリに一意の名前を付ける必要があり、Hot Rodサーバーに名前とキャッシュイベントコンバーターファクトリインスタンスを接続する必要があります。イベントコンバーターで Red Hat Data Grid Server をプラグインするには、以下の手順を行います。

1. コンバーターの実装を含むJARファイルを作成します。

2. オプション： キャッシュがカスタムキー/値クラスを使用する場合は、コールバックを JAR に含める必要があります。これにより、コールバックは正しくアンマーシャリングされたキーや値インスタンス/または値インスタンスで実行できます。クライアントリスナーでuseRawDataが有効になっている場合、コールバックのキー/値インスタンスはバイナリ形式で提供されるため、これは必要ありません。

3. JAR ファイル内で META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory ファイルを作成し、コンバータークラス実装の完全修飾クラス名を作成します。

4. JAR ファイルを Red Hat Data Grid Server にデプロイします。

その上で、ファクトリーの名前を @ClientListener アノテーションに追加して、クライアントリスナーをこのコンバーターファクトリーにリンクする必要があります。

```
@ClientListener(converterFactoryName = "static-converter")
public class CustomEventPrintListener { ... }
```

そして、リスナーをサーバーに登録します。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener());
```

リスナーの登録時に提供されたパラメーターを基に変換する動的コンバーターインスタンスもあります。コンバーターは、コンバーターファクトリーによって受け取ったパラメーターを使用してこのオプションを有効にします。以下に例を示します。

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

@NamedFactory(name = "dynamic-converter")
class DynamicCacheEventConverterFactory implements CacheEventConverterFactory {
   public CacheEventConverter<Integer, String, CustomEvent> getConverter(final Object[]
params) {
```

```
    return new DynamicCacheEventConverter(params);
  }
}

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

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

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

    return new CustomEvent(key, newValue);
  }
}
```

変換を行うために必要な動的パラメーターは、リスナーが登録されたときに提供されます。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), null, new Object[]{1});
```

警告

コンバーターインスタンスは、クラスターにデプロイされるときにマーシャリングする必要があります。これにより、リスナーの登録先の別のノードで生成された場合でも、イベントが生成される場所を正確に変換できます。それらをマーシャリング可能にするには、Serializable、Externalizableを拡張するか、カスタムExternalizerを提供します。

20.7.15. フィルタとカスタムイベント

イベントのフィルタリングとカスタマイズの両方を実行する場合、org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter を実装する方が簡単です。これにより、1 つのステップでフィルターとカスタマイズの両方を実行できます。便宜上、org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter を直接実装するの

ではなく、org.infinispan.notifications.cachelistener.filter.AbstractCacheEventFilterConverter を拡張することが推奨されます。以下に例を示します。

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

@NamedFactory(name = "dynamic-filter-converter")
class DynamicCacheEventFilterConverterFactory implements
CacheEventFilterConverterFactory {
  public CacheEventFilterConverter<Integer, String, CustomEvent> getFilterConverter(final
Object[] params) {
    return new DynamicCacheEventFilterConverter(params);
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed when
running in a cluster
//
class DynamicCacheEventFilterConverter extends
AbstractCacheEventFilterConverter<Integer, String, CustomEvent>, Serializable {
  final Object[] params;

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

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

    return new CustomEvent(key, newValue);
  }
}
```

フィルターとコンバーターと同様に、この組み合わせたフィルター/変換ファクトリーでリスナーを登録するには、ファクトリーに @NamedFactory アノテーション経由で一意の名前が付与される必要があります。また、Hot Rod サーバーは名前とキャッシュイベントコンバーターファクトリーインスタンスと共にプラグインする必要があります。イベントコンバーターで Red Hat Data Grid Server をプラグインするには、以下の手順を行います。

1. コンバーターの実装を含むJARファイルを作成します。

2. オプション： キャッシュがカスタムキー/値クラスを使用する場合は、コールバックを JAR に含める必要があります。これにより、コールバックは正しくアンマーシャリングされたキーや値インスタンス/または値インスタンスで実行できます。クライアントリスナーでuseRawDataが有効になっている場合、コールバックのキー/値インスタンスはバイナリ形式で提供されるため、これは必要ありません。

3. JAR ファイル内で META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverterFactory ファイルを作成し、コンバータークラス実装の完全修飾クラス名を作成します。

4. JAR ファイルを Red Hat Data Grid Server にデプロイします。

クライアントの観点からすると、組み合わせたフィルターとコンバータークラスを使用できるようにするには、クライアントリスナーは同じフィルターファクトリーとコンバーターファクトリー名を定義する必要があります。以下に例を示します。

```
@ClientListener(filterFactoryName = "dynamic-filter-converter", converterFactoryName =
"dynamic-filter-converter")
public class CustomEventPrintListener { ... }
```

上記の例で必要な動的パラメーターは、リスナーが filter パラメーターまたは converter パラメーターのいずれかに登録されている場合に提供されます。フィルターパラメーターが空でない場合は、それらが使用されます。それ以外の場合は、コンバーターパラメーターは以下のようになります。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener(), new Object[]{1}, null);
```

20.7.16. イベントマーシャリング

Hot Rod サーバーは、異なる形式でデータを保存できますが、Java Hot Rod クライアントユーザーは、タイプ化されたオブジェクトで機能する CacheEventConverter インスタンスまたは CacheEventFilter インスタンスを開発できます。デフォルトでは、フィルターとコンバーターは、データを POJO (application/x-java-object)として使用しますが、filter/converter からメソッド format() を上書きすることで、希望の形式を上書きすることができます。形式が null を返す場合、フィルター/変換は保存時にデータを受け取ります。

Marshalling Data セクションが示すように、Hot336 Java クライアントは、別の org.infinispan.commons.marshaller インスタンスを使用するよう設定でき ます。これ CacheEventConverter インスタンスや CacheEventFilter インスタンスをデプロイしたり、コンテンツをマーシャリングしたりするのではなく、Java オブジェクトでフィルター/変換できるようにするには、サーバーはオブジェクトとマーシャラーで生成されるバイナリー形式の間で変換できるようにする必要があります。

Marshaller インスタンスのサーバー側のデプロイするには、CacheEventConverterインスタンスまたはCacheEventFilterインスタンスをデプロイするのに使用されるものと同様の方法に従います。

1. コンバーターの実装を含むJARファイルを作成します。

2. JAR ファイル内で META-INF/services/org.infinispan.commons.marshall.Marshaller ファイルを作成し、フィルタークラス実装の完全修飾クラス名を作成します。

3. JAR ファイルを Red Hat Data Grid Server にデプロイします。

Marshaller は個別の jar または CacheEventConverterインスタンスおよび/またはCacheEventFilterインスタンスと同じ jar にデプロイできることに注意してください。

20.7.16.1. Protostream マーシャラーのデプロイ

キャッシュに protostream マーシャラーを使用する際に発生するように protobuf コンテンツが格納されている場合、形式はすでにサーバーによってサポートされているため、カスタムのマーシャラーをデプロイする必要はありません。protobuf 形式から JSON や POJO のような最も一般的な形式へのトランスコーダーがあります。

これらのキャッシュでフィルター/変換を使用し、バイナリー prototobuf データではなく Java オブジェクトでフィルター/変換を使用することが望ましい場合には、フィルター/変換の前にサーバーがデータをアンマーシャリングできるように、追加の protostream マーシャラーをデプロイする必要があります。これを行うには、以下の手順に従います。

1. 以下のインターフェースの実装が含まれる JAR ファイルを作成します。

```
org.infinispan.query.remote.client.ProtostreamSerializationContextInitializer
```

この実装は、Cache Manager の Serialization コンテキストに、さらにマーシャラーとオプションの Protobuf(.proto)ファイルを追加する必要があります。

2. 以下のファイルを JAR ファイル内に作成します。

```
META-INF/services/org.infinispan.query.remote.client.ProtostreamSerializationContextInitializer
```

このファイルには、ProtostreamSerializationContextInitializer 実装の完全修飾クラス名が含まれている必要があります。

3. 以下を含む JAR ファイル内に META-INF/MANIFEST.MF ファイルを作成します。

```
Dependencies: org.infinispan.protostream, org.infinispan.remote-query.client
```

4. Red Hat Data Grid Server がカスタムクラスにアクセスできるように、JBoss デプロイメント構造ファイル jboss-deployment-structure.xml を更新し、以下の内容を追加します。

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
    <deployment>
     <dependencies>
       <module name="org.infinispan.protostream" />
       <module name="org.infinispan.remote-query.client" services="import"/>
     </dependencies>
   </deployment>
</jboss-deployment-structure>
```

5. standalone/deployments フォルダーに追加して、JAR ファイルを Red Hat Data Grid Server にデプロイします。

6. 以下のように、適切な Cache Manager でデプロイメントを設定します。

```
<cache-container name="local" default-cache="default">
  <modules>
    <module name="deployment.my-file.jar"/>
  </modules>
  ...
</cache-container>
```

重要

カスタムクラスを Red Hat Data Grid Server にデプロイする JAR ファイルは起動時に利用可能である必要があります。JAR を稼働中のサーバーインスタンスにデプロイできません。

20.7.17. リスナーの状態の処理

クライアントリスナーアノテーションには、リスナーの追加時またはリスナーのフェイルオーバー時に状態がクライアントに送信されるかどうかを指定する任意の includeCurrentState 属性があります。

デフォルトでは、includeCurrentState は false ですが、true に設定され、クライアントリスナーが

データが含まれるキャッシュに追加されると、サーバーはキャッシュの内容を繰り返し処理し、各エントリーのイベント（設定されている場合はカスタムイベント）を ClientCacheEntryCreated としてクライアントに送信します。これにより、クライアントは既存のコンテンツに基づいて一部のローカルデータ構造をビルドできます。コンテンツが繰り返されると、キャッシュの更新を受け取るため、イベントは通常どおり受信されます。キャッシュがクラスター化されている場合、クラスター全体のコンテンツ全体が繰り返し処理されます。

includeCurrentState は、クライアントイベントリスナーが登録され、別のノードに移動されたノードが受信されるかどうかも制御します。次のセクションでは、このトピックを詳細に説明します。

20.7.18. リスナーの障害処理

Hot Rod クライアントがクライアントリスナーを登録すると、クラスターの単一ノードになります。そのノードに障害が発生すると、Java Hot Rod クライアントは透過的に検出し、別のノードに失敗したノードに登録されているすべてのリスナーをフェイルオーバーします。

このフェイルオーバー中に、クライアントはいくつかのイベントを見逃す可能性があります。これらのイベントが不足しないように、クライアントリスナーアノテーションには includeCurrentState という任意のパラメーターが含まれます。これは、フェイルオーバーの実行時に、キャッシュの内容が繰り返し処理され、ClientCacheEntryCreated イベント（設定されている場合はカスタムイベント）が生成されます。デフォルトでは、includeCurrentStateはfalseに設定されています。

Java Hotgitops クライアントは、処理するコールバックを追加することで、このようなフェイルオーバーイベントを認識できます。

```
@ClientCacheFailover
public void handleFailover(ClientCacheFailoverEvent e) {
  ...
}
```

これは、クライアントがいくつかのデータをキャッシュしており、フェイルオーバーの結果、いくつかのイベントが見逃される可能性を考慮して、フェイルオーバーイベントを受信したときにローカルにキャッシュされたデータを消去することを決定し、フェイルオーバーイベントの後にキャッシュ全体のコンテンツに対するイベントを受信することを知っているような場合に非常に便利です。

20.7.19. ニアキャッシング

Java HotTEMPLATES クライアントにはオプションで near キャッシュを設定できます。つまり、Hotfsprogs クライアントは最近使用されたデータを格納するローカルキャッシュを維持できます。近いキャッシングを有効にすると、データはリモートにアクセスするのではなく Hotgitops クライアント内にローカルにある可能性があるため、読み取り操作の取得と get Versioned のパフォーマンスを大幅に向上させることができます。

近いキャッシュを有効にするには、near キャッシュモードを INVALIDATED に設定する必要があります。その近いキャッシュを行うと、get または getVersioned 操作への呼び出しを介してサーバーから取得が行われます。キャッシュされたエントリーがほぼ更新または削除されると、キャッシュされたほぼキャッシュエントリーは無効になります。キーが無効になった後に要求される場合は、サーバーから再フェッチする必要があります。

警告

ほぼキャッシュでは maxIdle の有効期限を使用しないでください。near-cache の読み取りは最後のアクセスの変更をサーバーおよび他のクライアントに伝播しないためです。

ほぼキャッシュが有効な場合は、最も近いキャッシュに保持するエントリーの最大数を定義してサイズを設定する必要があります。最大に達すると、キャッシュされたエントリー付近がエビクトされます。0 または負の値を指定すると、ほぼキャッシュがバインドされていないことが想定されます。

警告

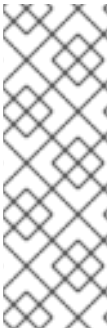
クライアント JVM の境界内に近いキャッシュのサイズを維持する責任を変化させるため、ほぼキャッシュをバインドしないようにする場合は注意してください。

Hotgitops クライアントの near キャッシュモードは、NearCacheMode 列挙を使用して設定され、呼び出します。

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
...

// Unbounded invalidated near cache
ConfigurationBuilder unbounded = new ConfigurationBuilder();
unbounded.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(-1);

// Bounded invalidated near cache
ConfigurationBuilder bounded = new ConfigurationBuilder();
bounded.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(100);
```

設定は単一の RemoteCacheManager から取得したすべてのキャッシュで共有されるため、それらすべてのキャッシュに対してほぼキャッシュを有効にしたくない場合があります。cacheNamePattern 設定属性を使用して、キャッシュの名前に一致する正規表現を定義できます。名前が正規表現に一致しないキャッシュは、ほぼキャッシュが有効ではありません。

```
// Bounded invalidated near cache with pattern matching
ConfigurationBuilder bounded = new ConfigurationBuilder();
bounded.nearCache()
  .mode(NearCacheMode.INVALIDATED)
  .maxEntries(100)
  .cacheNamePattern("near.*"); // enable near-cache only for caches whose name starts with
'near'
```



注記

ほぼキャッシュは、クラスター化されたキャッシュのようにローカルキャッシュと同様に機能しますが、クラスター化キャッシュのシナリオでは、サーバーノードが Hot336 クライアントに対してほぼキャッシュを送信するサーバーノードがダウンした場合、Hotgitops クライアントはクラスター内の別のノードに透過的に失敗し、以下の方法でほぼキャッシュをクリアします。

20.7.20. サポートされないメソッド

Cache メソッドの一部は RemoteCache ではサポートされません。これらのメソッドのいずれかを呼び出すと、UnsupportedOperationException が発生します。これらのメソッドのほとんどはリモートキャッシュ（リスナー管理操作など）では意味を持ちません。または、ローカルキャッシュでサポートされていないメソッドにも対応しません（例: containsValue）。サポートされていない別の操作のセットには、ConcurrentMap から継承される atomic 操作の一部があります。

```
boolean remove(Object key, Object value);
boolean replace(Object key, Object value);
boolean replace(Object key, Object oldValue, Object value);
```

RemoteCache は、ネットワークを介して値オブジェクト全体を送信せず、バージョン識別子を使用して値オブジェクトを送信することで、ネットワークフレンドリーなこれらのアトミック操作のための代替バージョン方法を提供します。バージョン管理された API のセクションを参照してください。

これらのサポート対象外の操作の 1 つは、RemoteCache javadoc に文書化されています。

20.7.21. 戻り値

キャッシュされたエントリーを変更し、以前の既存の値を返すメソッドのセットがあります。以下に例を示します。

```
V remove(Object key);
V put(K key, V value);
```

デフォルトでは、RemoteCache では、これらの操作は以前の値が存在する場合でも null を返します。この方法では、ネットワーク上で送信されるデータ量が減ります。ただし、これらの戻り値が必要な場合は、フラグを使用して呼び出しごとに適用できます。

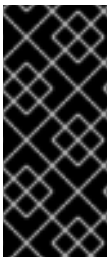
```
cache.put("aKey", "initialValue");
assert null == cache.put("aKey", "aValue");
assert "aValue".equals(cache.withFlags(Flag.FORCE_RETURN_VALUE).put("aKey",
   "newValue"));
```

このデフォルトの動作は、force-return-value=true 設定パラメーターを使用して変更できます（設定セクション bellow を参照）。

20.7.22. ホットの有効化トランザクション

JTA の トランザクション で Hot Rod クライアントを設定および使用できます。

トランザクションに参加するために、Hot Rodクライアントは、相互作用するTransactionManagerと、Synchronization インターフェイスまたは XAResourceインターフェイスを通じてトランザクションに参加するかどうかを要求します。



重要

トランザクションは、クライアントが準備フェーズでエントリーへの書き込みロックを取得するという点で最適化されます。データの不整合を回避するには、トランザクションとの競合を検出 を必ずお読みください。

20.7.22.1. サーバーの設定

クライアントが JTA トランザクションに参加するには、サーバーのキャッシュもトランザクションである必要があります。

次のサーバー構成が必要です。それ以外の場合、トランザクションはロールバックのみになります。

- 分離レベルはREPEATABLE_READである必要があります。
- ロックモードは PESSIMISTIC である必要があります。今後のリリースでは、OPTIMISTIC ロックモードがサポートされます。
- トランザクションモードは、NON_XAまたはNON_DURABLE_XAである必要があります。hotrefentry トランザクションは、パフォーマンスが低下するため、FULL_XA を使用することはできません。

Hot Rodトランザクションには、独自の復旧メカニズムがあります。

20.7.22.2. Hot Rodクライアントの設定

RemoteCacheManager の作成時に、RemoteCache が使用するデフォルトの TransactionManager および TransactionMode を設定できます。

RemoteCacheManager を使用すると、以下の例のようにトランザクションキャッシュの設定を 1 つだけ作成できます。

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
//other client configuration parameters
cb.transaction().transactionManagerLookup(GenericTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);
cb.transaction().timeout(1, TimeUnit.MINUTES)
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

上記の設定は、リモートキャッシュのすべてのインスタンスに適用されます。リモートキャッシュインスタンスに異なる設定を適用する必要がある場合は、RemoteCache 設定を上書きできます。「RemoteCacheManager 設定の上書き」を参照してください。

設定パラメーターのドキュメントについては、「ConfigurationBuilder Javadoc」を参照してください。

以下の例のように、プロパティーファイルを使用して Java Hot Rod クライアントを設定することもできます。

```
infinispan.client.hotrod.transaction.transaction_manager_lookup =
org.infinispan.client.hotrod.transaction.lookup.GenericTransactionManagerLookup
infinispan.client.hotrod.transaction.transaction_mode = NON_XA
infinispan.client.hotrod.transaction.timeout = 60000
```

20.7.22.2.1. TransactionManagerLookup インターフェース

TransactionManagerLookup は、 TransactionManager を取得するためのエントリーポイントを提供します。

TransactionManagerLookupのの利用可能な実装:

GenericTransactionManagerLookup

Java EE アプリケーションサーバーで実行しているTransactionManagerを検索するルックアップクラス。TransactionManager が見つからない場合は、デフォルトでは Remote TransactionManager に設定されます。

ヒント

ほとんどの場合、GenericTransactionManagerLookup が適しています。しかし、カスタム TransactionManager を統合する必要がある場合は、TransactionManagerLookup インターフェースを実装できます。

RemoteTransactionManagerLookup

他の実装が利用できない場合は、基本および揮発性のTransactionManagerです。この実装には、同時トランザクションと復元を処理するときに重大な制限があることに注意してください。

20.7.22.2.2. トランザクションモード

TransactionMode は、 RemoteCache が TransactionManager と対話する方法を制御します。

重要

Red Hat Data Grid サーバーとクライアントアプリケーションの両方で、トランザクションモードを設定します。クライアントが非トランザクションキャッシュでトランザクション操作を実行しようとすると、ランタイム例外が発生する可能性があります。

Red Hat Data Grid の設定とクライアント設定の両方で同じトランザクションモードが使用されま

す。クライアントで以下のモードを使用します。サーバーの Red Hat Data Grid 設定スキーマを参照してください。

NONE

RemoteCache は TransactionManager と対話しません。これはデフォルトのモードであり、非トランザクションです。

NON_XA

RemoteCache は、Syncs を使用して TransactionManager と対話 し ます。

NON_DURABLE_XA

RemoteCache は、XAResource を介して TransactionManager と対話します。復元機能は無効になっています。

FULL_XA

RemoteCache は、XAResource を介して TransactionManager と対話します。復元機能が有効になっています。XaResource.recover() メソッドを呼び出して、リカバリーするトランザクションを取得します。

20.7.22.3. キャッシュインスタンスの設定の上書き

RemoteCacheManager は各キャッシュインスタンスで異なる設定をサポートしないためです。ただし、RemoteCacheManager には、RemoteCache インスタンスを返す getCache(String) メソッドが含まれ、以下のように一部の設定パラメーターを上書きすることができます。

getCache(String cacheName, TransactionMode transactionMode)

RemoteCache を返し、設定された TransactionMode を上書きします。

getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode)

上記と同じですが、書き込み操作に対して強制的に値を返すこともできます。

getCache(String cacheName, TransactionManager transactionManager)

RemoteCache を返し、設定された TransactionManager を上書きします。

getCache(String cacheName, boolean forceReturnValue, TransactionManager transactionManager)

上記と同じですが、書き込み操作に対して強制的に値を返すこともできます。

getCache(String cacheName, TransactionMode transactionMode, TransactionManager transactionManager)

RemoteCache を返し、設定された TransactionManager および TransactionMode を上書きします。transactionManager または transactionMode が null の場合、設定された値を使用します。

getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode, TransactionManager transactionManager)

上記と同じですが、書き込み操作に対して強制的に値を返すこともできます。

注記

getCache(String) メソッドは、トランザクションかどうかに関係なく RemoteCache インスタンスを返します。RemoteCache には、キャッシュが使用する TransactionManager を返す get TransactionManager （） メソッドが含まれます。RemoteCache がトランザクションでない場合、メソッドは null を返します。

20.7.22.4. トランザクションとの競合の検出

トランザクションはキーの初期値を使用して競合を検出します。たとえば、トランザクションの開始時に「k」の値は「v」となります。準備フェーズでは、トランザクションはサーバーから "k" を取得して値を読み取ります。値が変更された場合、トランザクションは競合を回避するためにロールバックします。

注記

トランザクションはバージョンを使用して、値の等価性を確認する代わりに変更を検出します。

forceReturnValue パラメーターは RemoteCache への書き込み操作を制御し、競合を回避するのに役立ちます。次の値があります。

- *true の場合、書き込み操作を実行する前に TransactionManager はサーバーから最新の値を取得します。ただし、forceReturnValue パラメーターは、キーの初回アクセスの操作にのみ適用されます。*
- *false の場合、TransactionManager は書き込み操作を実行する前にサーバーから最新の値を取得しません。この設定があるためです。*

注記

このパラメーターは、最新の値が必要なため、replace や putIfAbsent などの 条件付き 書き込み操作には影響しません。

以下のトランザクションは、forceReturnValue パラメーターが競合する書き込み操作を防ぐ例を提供します。

トランザクション 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v1");
tm.commit();
```

トランザクション 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v2");
tm.commit();
```

この例では、TX1とTX2が並行して実行されます。「k」の初期値は「v」です。

- forceReturnValue = true の場合、cache.put() 操作はサーバーから TX1 と TX2 の両方のサーバーから「k」の値を取得します。最初に「k」のロックを取得するトランザクションはコミットします。他のトランザクションは、「k」が「v」以外の値を持っていることをトランザクションが検出できるため、コミットフェーズ中にロールバックします。
-

forceReturnValue = false の場合cache.put() オペレーションはサーバーから "k" の値を取得せず、null を返します。TX1 と TX2 の両方が正常にコミットされ、競合が生じます。これは、 "k"の初期値が変更されたことをトランザクションが検知できないために発生します。

以下のトランザクションは、cache.put()オペレーションを行う前に「K」の値を読み取るcache.get()オペレーションが含まれます。

トランザクション 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v1");
tm.commit();
```

トランザクション 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v2");
tm.commit();
```

上記の例では、TX1 および TX2 の両方が鍵を読み取るため、forceReturnValue パラメーターは有効になりません。1つのトランザクションがコミットし、もう1つのトランザクションがロールバックします。ただし、cache.get() 操作には追加のサーバー要求が必要です。サーバーリクエストが非効率な cache.put() オペレーションの戻り値が必要ない場合。

20.7.22.5. 設定済みトランザクションマネージャーおよびトランザクションモードの使用

以下の例は、RemoteCacheManager で設定した TransactionManager および TransactionMode を使用する方法を示しています。

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//The my-cache instance uses the RemoteCacheManager configuration.
RemoteCache<String, String> cache = rcm.getCache("my-cache");

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```

20.7.22.6. トランザクションマネージャーの上書き

以下の例は、getCache メソッドで TransactionManager を上書きする方法を示しています。

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//Define a custom TransactionManager.
TransactionManager myCustomTM = ...

//Override the TransactionManager for the my-cache instance. Use the default configuration if
null is returned.
RemoteCache<String, String> cache = rcm.getCache("my-cache", null, myCustomTM);

//Perform a simple transaction.
myCustomTM.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
myCustomTM.commit();
```

20.7.22.7. トランザクションモードの上書き

以下の例は、getCache メソッドで TransactionMode を上書きする方法を示しています。

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//Override the transaction mode for the my-cache instance.
RemoteCache<String, String> cache = rcm.getCache("my-cache",
TransactionMode.NON_DURABLE_XA, null);

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```

20.7.23. クライアントのインテリジェンス

クライアント情報とは、クライアントが Red Hat Data Grid サーバーにリクエストを見つけ、送信するための HotRod プロトコルのメカニズムを指します。

基本的な情報

クライアントは、Red Hat Data Grid クラスターやキーハッシュ値に関する情報を保存しません。

トポロジー対応

クライアントは Red Hat Data Grid クラスターについての情報を受信して保存します。クライアントは、サーバーがクラスターに参加したり離脱したりするたびに変更するクラスタートポロジーの内部マッピングを維持します。

クラスタートポロジーを受信するには、クライアントが少なくとも 1 つの Hot336 サーバーのアドレス(IP:HOST)が必要になります。クライアントがサーバーに接続すると、Red Hat Data Grid はトポロジーをクライアントに送信します。サーバーがクラスターに参加または退出すると、Red Hat Data Grid は更新されたトポロジーをクライアントに送信します。

distribution-aware

クライアントはトポロジーに対応し、キーの一貫したハッシュ値を保存します。

たとえば、put(k,v) 操作を実行します。クライアントは、データが存在する正確なサーバーを見つけることができるように、キーのハッシュ値を計算します。その後、クライアントは所有者に直接接続して操作をディスパッチできます。

ディストリビューション対応の利点は、Red Hat Data Grid サーバーが、サーバー側でリソースが少ないキーハッシュに基づいて値を検索する必要がないことです。もう 1 つの利点は、サーバーが追加のネットワークラウンドトリップをスキップするため、クライアントの要求により迅速に応答するという点です。

20.7.23.1. バランシングの要求

トポロジー対応の情報を使用するクライアントは、すべてのリクエストに要求の分散を使用します。デフォルトの分散ストラテジーはラウンドロビンであるため、トポロジー対応のクライアントは常にラウンドロビン順でサーバーに要求を送信します。

たとえば、s1、s2、s3 は Red Hat Data Grid クラスターのサーバーです。クライアントは以下のように要求のバランシングを実行します。

```
CacheContainer cacheContainer = new RemoteCacheManager();
Cache<String, String> cache = cacheContainer.getCache();

//client sends put request to s1
cache.put("key1", "aValue");
//client sends put request to s2
cache.put("key2", "aValue");
//client sends get request to s3
String value = cache.get("key1");
//client dispatches to s1 again
cache.remove("key2");
//and so on...
```

ディストリビューション対応情報を使用するクライアントは、失敗したリクエストに対してのみ要求の分散を使用します。要求が失敗すると、ディストリビューション対応クライアントは次に利用可能なサーバーで要求を再試行します。

カスタムバランシングポリシー

FailoverRequestBalancingStrategy を実装し、hotrod-client.properties 設定でクラスを指定できます。

20.7.24. 永続的な接続

各リクエスト（コストのかかる操作）で TCP 接続を作成しないように、クライアントは利用可能なすべてのサーバーへの永続的な接続のプールを維持し、可能な場合は常にこの接続を再利用します。接続の有効性は、プール内の接続を繰り返し処理し、HotRod ping コマンドを送信する非同期スレッドを使用してチェックされます。この接続検証プロセスを使用すると、クライアントがプロアクティブになります。プール内でアイドル状態で、アプリケーションからの実際のリクエストに無い接続が見つかる可能性が高くなっています。

サーバーごとの接続数、接続数、接続の合計数、閉じられる前に接続がアイドル状態である期間。これらすべてを設定できます。可能なすべての設定要素の一覧は、RemoteCacheManager の javadoc を参照してください。

20.7.25. マーシャリングデータ

Hotgitops クライアントを使用すると、ユーザーオブジェクトをバイトアレイに変換するためのカスタムマーシャラーをプラグインして、別の方法でプラグインできます。この変換は、Hot pressure のバイナリー性により必要となります。オブジェクトについて把握していないため、この変換が必要です。

マーシャラーは、「marshaller」設定要素でプラグインできます（設定セクションを参照）。値は、Marshaller インターフェースを実装するクラスの完全修飾名である必要があります。これは任意のパラメーターで、デフォルトで GenericJBossMarshaller に設定されます。これは、JBoss Marshalling ライブラリーに基づく非常に最適化された実装です。

バージョン 6.0 以降、移植可能なペイロードを生成する Protostream に基づいて、新しいマーシャラーが Java Hotgitops クライアントで利用できます。詳細は、 を参照してください。

警告： 独自のカスタムマーシャラーを開発する場合は、潜在的なインジェクション攻撃に注意します。

このような攻撃を回避するには、アンシャラが、予想される/許可済みのクラス名にあわせてクラス名を読み取る前にマーシャラーが読み込まれていることを確認します。

クライアント設定は、読み取りが可能なクラスの正規表現のリストで強化できます。

警告： これらのチェックはオプトインであるため、設定されていない場合はすべてのクラスを読み取ることができます。

以下の例では、Person または Employee を含む完全修飾名を持つクラスのみが許可されます。

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;

...
ConfigurationBuilder configBuilder = ...
configBuilder.addJavaSerialWhiteList(".*Person.*", ".*Employee.*");
```

20.7.26. 異なるデータフォーマットのデータの読み取り

デフォルトでは、Hotfsprogs クライアント操作はすべて、キーと値の両方にサーバーの読み取りおよび書き込み時に設定済みのマーシャラーを使用します。マーシャリングデータ を参照してください。DataFormat API を使用すると、リモートキャッシュをデコードし、すべての操作をカスタムデータ形式で発生させることができます。

20.7.26.1. キーおよび値に異なるマーシャラーの使用

キーと値のマーシャラはランタイム時に上書きできます。たとえば、Hotgitops クライアント内のすべてのシリアライズをバイパスし、サーバーに保存されるように byte[] を読み取るには、以下を実行します。

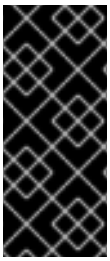
```
// Existing Remote cache instance
RemoteCache<String, Pojo> remoteCache = ...

// IdentityMarshaller is a no-op marshaller
DataFormat rawKeyAndValues =
DataFormat.builder().keyMarshaller(IdentityMarshaller.INSTANCE).valueMarshaller(IdentityMar
shaller.INSTANCE).build();

// Will create a new instance of RemoteCache with the supplied DataFormat
RemoteCache<byte[], byte[]> rawResultsCache =
remoteCache.withDataFormat(rawKeyAndValues);
```

20.7.26.2. 異なる形式のデータの読み取り

カスタムキーと値のマーシャラーの定義以外に、org.infinispan.commons.dataconversion.MediaType で指定された形式が異なる形式でデータを要求/送信することもできます。

```
// Existing remote cache using ProtostreamMarshaller
RemoteCache<String, Pojo> protobufCache = ...

// Request values returned as JSON, using the UTF8StringMarshaller that converts between
UTF-8 to String:
DataFormat jsonString =
DataFormat.builder().valueType(MediaType.APPLICATION_JSON).valueMarshaller(new
UTF8StringMarshaller().build();

RemoteCache<String, String> jsonStrCache = remoteCache.withDataFormat(jsonString);
```

```
// Alternatively, it's possible to request JSON values but marshalled/unmarshalled with a
custom value marshaller that returns `org.codehaus.jackson.JsonNode` objects:
DataFormat jsonNode =
DataFormat.builder().valueType(MediaType.APPLICATION_JSON).valueMarshaller(new
CustomJacksonMarshaller().build();

RemoteCache<String, JsonNode> jsonNodeCache = remoteCache.withDataFormat(jsonNode);
```



重要

データ変換はサーバーで行われ、ストレージ形式から要求形式への変換をサポートしない場合、エラーが返されます。サーバーデータフォーマットの設定およびサポートされる変換の詳細は、こちら を参照してください。

警告

.key Marshaller（）および .key Type （）を使用してキー に異なるマーシャラーと形式を使用すると、クライアントの情報ルーティングメカニズムに干渉し、Hoting の操作中にキーの所有者ではないサーバーと通信する可能性があります。これによりエラーが発生しませんが、クラスター内で追加のホップが発生して操作を実行できます。パフォーマンスが重要な場合は、サーバーに保存されている形式の鍵を使用するようにしてください。

20.7.27. 統計

RemoteCache .stats（）メソッドを使用して、さまざまなサーバーの使用状況の統計を取得できます。ServerStatistics オブジェクトを返します。利用可能な統計の詳細は、javadoc を参照してください。

20.7.28. 複数取得操作

Java Hotfaillock クライアントは、追加設定なしでマルチット機能を提供しませんが、クライアントは指定の API でビルドできます。

20.7.29. フェイルオーバー機能

トポロジーの変更に追いつくホットpidクライアントを使用すると、サーバーの 1 つまたは複数に障害が発生した場合に、要求のバランスをやり、より重要な操作にフェイルオーバーできます。

上記の条件操作の一部（p lfAbsent、バージョンの有無の置き換え、および条件付き 削除 ）には、厳密なメソッドの戻り値の保証や、以前の値を返す操作が強制される操作が含まれます。

失敗しても、これらのメソッドの戻り値は保証する必要があります。そのためには、障害発生時にこれらのメソッドがクラスターで部分的に適用されない必要があります。たとえば、key=k1 のサーバーの replace（） 操作は Flag.FORCE_RETURN_VALUE に、現在の値は A で、置換は B に設定します。置き換えに失敗した場合、一部のサーバーには B が含まれており、フェイルオーバー時に、B が設定されたノードにフェイルオーバーを 置き換えるか、または A を返す可能性がある場合は、元の replace( )が B を返す可能性があります。

このような状況を回避するために、Java HotTEMPLATES クライアントユーザーが条件付き操作または以前の値が必要な操作を使用する場合は、条件付き操作や戻り値を返さないようにキャッシュがトランザクションに設定されていることが重要です。

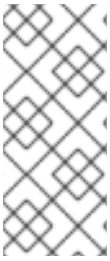
20.7.30. サイトクラスターフェイルオーバー

クラスター内フェイルオーバーの上で、Hotfsprogs クライアントは異なるクラスターにフェイルオーバーすることもでき、このクラスターは独立したサイトとして表される可能性があります。

サイトクラスターのフェイルオーバーが機能する仕組みは、すべてのメインクラスターノードが利用できない場合に、クライアントは他のクラスターが定義されており、別のクラスターにフェイルオーバーしようとするかどうかをチェックします。フェイルオーバーが成功すると、クライアントは使用できなくなるまで代替クラスターに接続されているままになります。この場合、定義された他のクラスターを試し、最終的に元のサーバー設定を試みます。

Hotgitops クライアントでクラスターを設定するには、設定した各クラスターに 1 つのホスト/ポートのペアの詳細を指定する必要があります。以下に例を示します。

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
    = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.addCluster().addClusterNode("remote-cluster-host", 11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```



注記

クラスターの定義に関係なく、初期サーバーはデフォルトのサーバーホストおよびポート情報を使用して解決されていない限り、初期サーバーの設定を提供する必要があります。

20.7.31. 手動サイトクラススイッチ

自動サイトのクラスターフェイルオーバーをサポートする他に、Java Hotgitops クライアントは、RemoteCacheManager の switchToCluster(clusterName) と switch ToDefaultCluster（）を呼び出してサイトクラスターを手動で切り替える ことができます。

switchToCluster(clusterName) を使用すると、ユーザーは、Hot336 クライアント設定で事前定義されたクラスターの 1 つに、クライアントを強制的に切り替えることができます。クライアント設定で定義された初期サーバーに切り替えるには、switchToDefaultCluster（） を呼び出します。

20.7.32. Hotgitops クライアントの監視

「管理」の章で説明されている内容と同様に、Hot336 クライアントを JMX 経由で監視および管理できます。#jmx_mgmt_tooling統計を有効にすると、MBean は RemoteCacheManager と取得した各 RemoteCache に登録されます。これらの MBean を使用して、リモートおよびほぼキャッシュヒット/中間キャッシュおよび接続プールの使用状況に関する統計を取得できます。

20.7.33. 同時更新

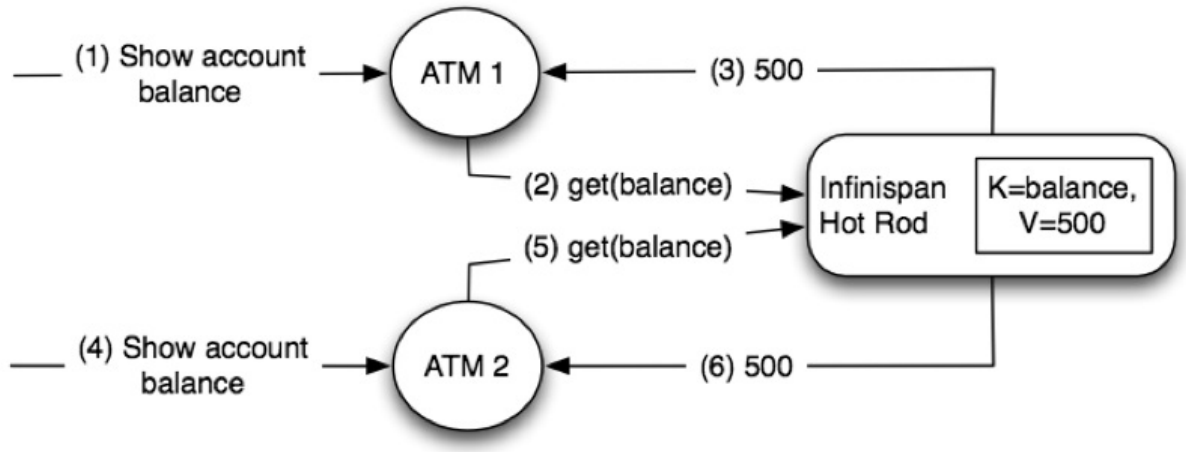
データ構造（Red Hat Data Grid Cache など）がアクセスおよび変更されると、データの正確性を保証するメカニズムがない限り、データの整合性の問題に悪影響を及ぼす可能性があります。ConcurrentMap を実装するため、Red Hat Data Grid Cache は、条件付き置換、putIfAbsent、および 条件付き削除 などの操作を提供し、データの正確性を保証します。また、クライアントは JTA トランザクション内のキャッシュインスタンスに対して操作できるため、必要なデータの整合性が保証されます。

ただし、Hot pressure プロトコル でサポートされるサーバーの場合、クライアントにはリモートトランザクションを起動する機能がありませんが、代わりにバージョン管理された操作を呼び出して、組み込み Red Hat Data Grid キャッシュインスタンス API が提供する条件メソッドを模倣できます。実際の例を見て、その仕組みを理解してみましょう。

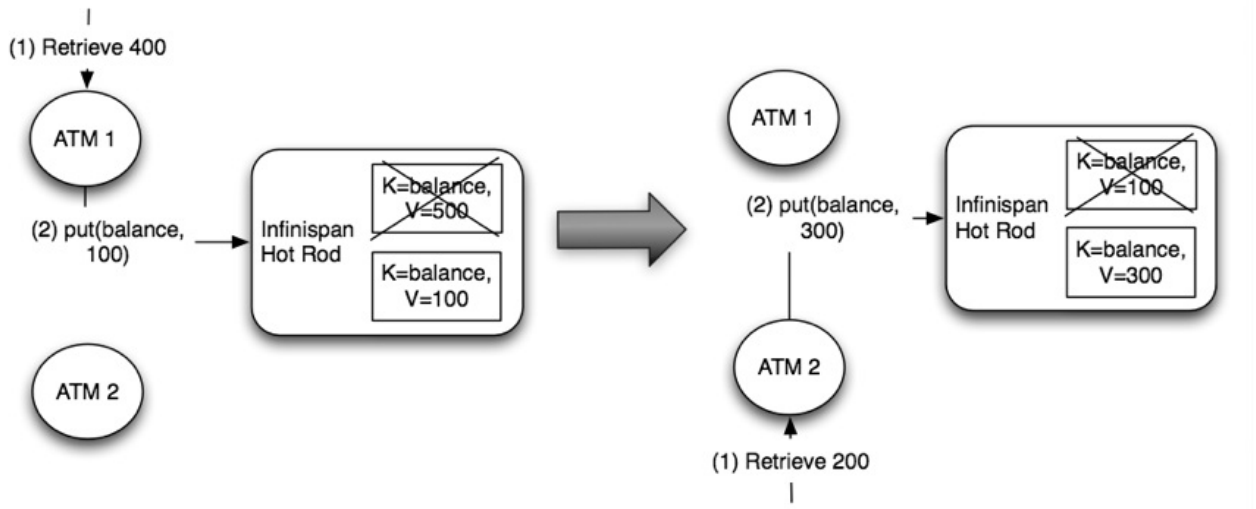
20.7.33.1. データの一貫性の問題

アカウントのバランスが格納されている銀行に Hotpid を使用して接続した 2 つの ATM があるとします。最新のバランスを取得するために 2 つの後に続く 2 つの操作は、以下のように 500 CHF(swiss francs)を返す可能性があります。

図20.7 同時リーダー



次に、顧客が最初の ATM に接続し、400 CHF を取得するよう要求します。ATM は最後の値の読み取りに基づいて、新しいバランス(100 CHF)を計算して、この新しい値で要求できます。では、別の顧客が ATM に接続するのと同時に、200 CHF を取得しても入手できることを考えてみましょう。ATM は最新のバランスを持ち、計算に基づいて新しいバランスを 300 CHF に設定すると仮定してみましょう。



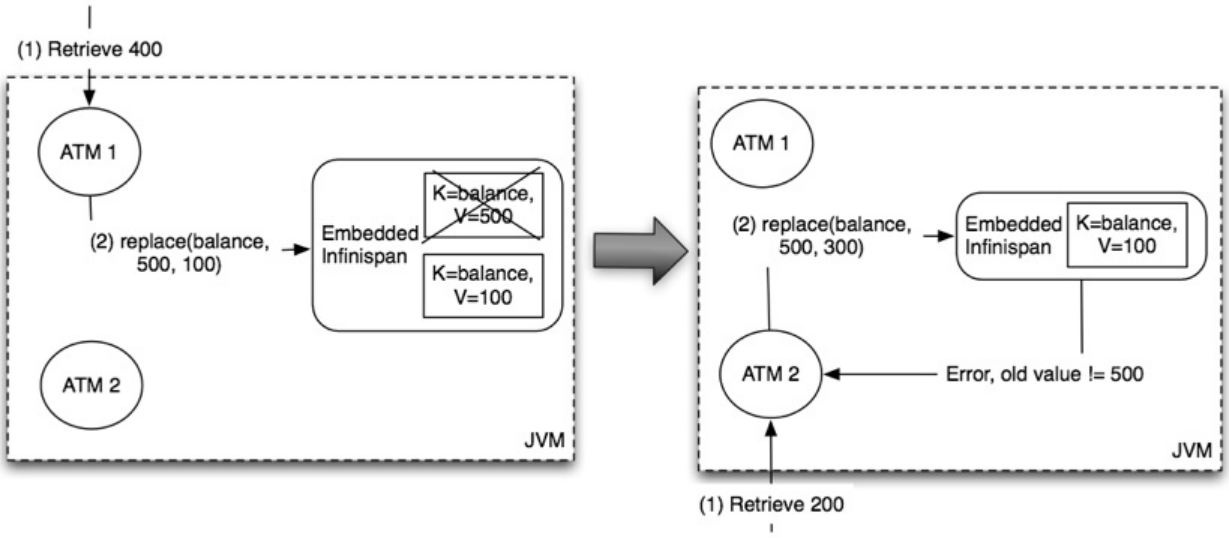
明らかに、これは間違っています。2 つの同時更新により、誤ったアカウントのバランスが生じました。2 番目の ATM が正しくないため、2 つ目の更新は許可されていません。新しいバランスを計算する前に ATM がバランスを取得している場合でも、取得される新しいバランスと更新の間で更新することができます。Hot336 のクライアントサーバーシナリオでこの問題を解決する方法を検討する前に、クライアントと Red Hat Data Grid が同じ JVM 内に配置されたピアツーピアモードで実行した場合に、この問題を解決する方法を見ていきます。

20.7.33.2. 埋め込みモードソリューション

ATM と Red Hat Data Grid インスタンスが同じ JVM に存在する銀行アカウントを保存する場合、ATM は本記事の最初に参照される 条件的な置換 API を使用できます。そのため、以前の既知の値を送

信して、最後に読み込んだ後に変更されたかどうかを確認することができます。これを実行すること で、最初の操作は、100 CHF に更新した場合のバランスが 500 CHF でもあることを再確認することが できます。2 番目の操作に達すると、現在のバランスは 500 CHF ではなく、条件付き置換呼び出しが 失敗し、データの一貫性の問題を回避できるようになりました。

図20.8 P2P ソリューション

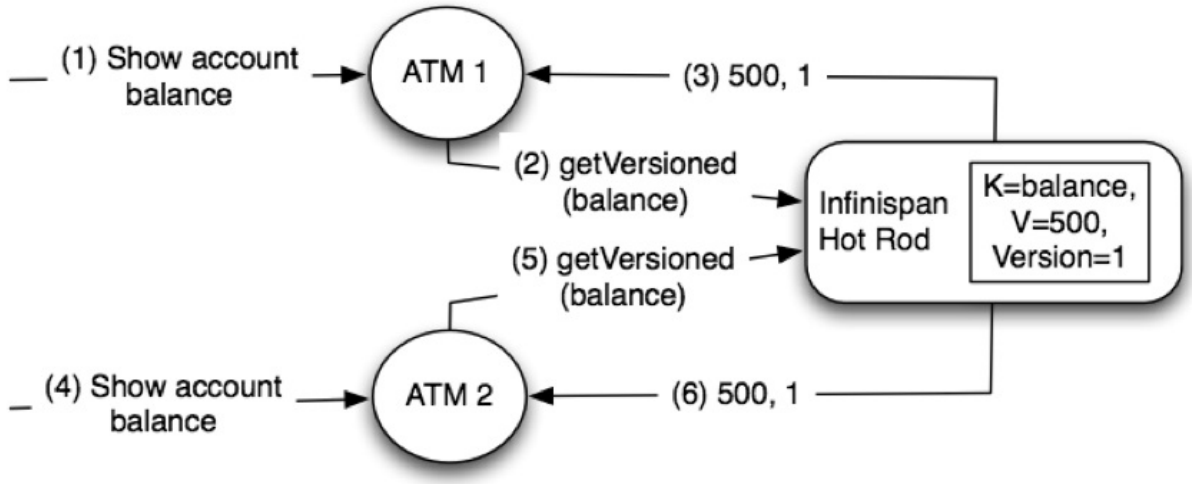


20.7.33.3. クライアントサーバーソリューション

理論では、Hotset ソリューションは同じ p2p ソリューションを使用できますが、前の値を送信す ることは実用的ではありません。この例では、以前の値は単なる整数ですが、値が大きい可能性がある ため、クライアントがサーバーに送信するように強制すると無駄になります。代わりに、Hotfsprogs は、この状況に対応するバージョン化された操作を提供します。

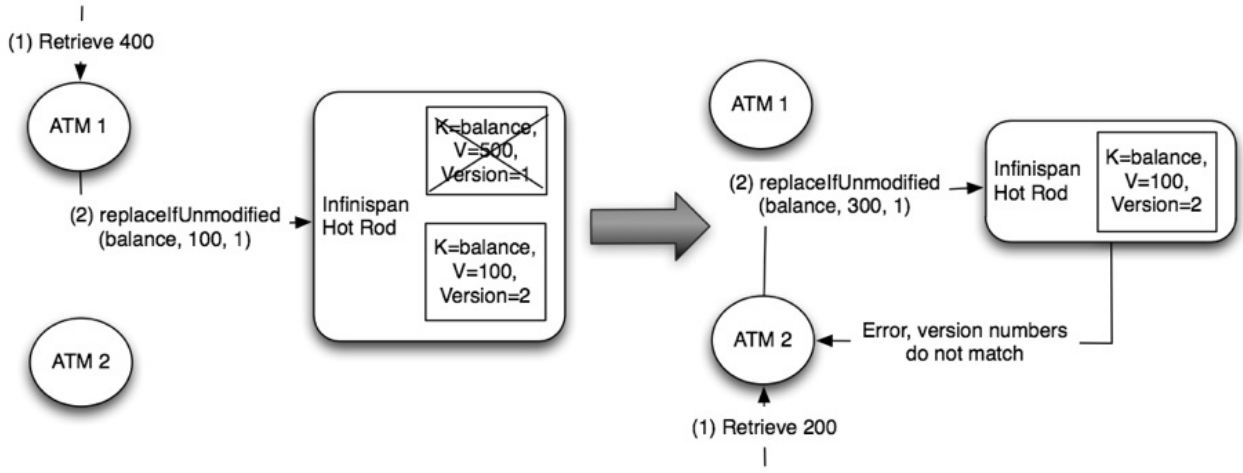
基本的に、各キー/値のペアと合わせて、Hot Lake は各変更を一意に識別するバージョン番号を保 存します。そのため、getVersioned または getWithVersion という操作を使用すると、クライアント はキーに関連付けられた値だけでなく、現行バージョンも取得できます。そのため、先ほどの例を再度 見ると、ATM は getVersioned を呼び出してバランスのバージョンを取得することができます。

図20.9 バージョン付けを取得します。



ATM がバランスを修正したい場合、単に呼び出しを行うのではなく、replaceIfUnmodified 操作を呼び出して、クライアントが認識している最新バージョン番号を渡すことができます。この操作は、渡されたバージョンがサーバーのバージョンと一致する場合にのみ成功します。そのため、クライアントが 1 をバージョンとして渡し、サーバー側バージョンは 1 でも 1 であるため、ATM による最初の変更が許可されます。一方、第 1 の ATM の変更後にバージョンを 2 にインクリメントし、サーバー側バージョン(2)は一致しないため、2 番目の ATM は変更を実行できません。

図20.10 バージョンが一致すると置き換え



20.7.34. Javadocs

以下の Java ドキュメントを読むことが強く推奨されます（これはクライアントのパブリック API の多くです）。

- *RemoteCacheManager*

- *RemoteCache*

20.8. REST サーバー

Red Hat Data Grid Server ディストリビューションには、Netty に構築された Red Hat Data Grid グリッドへの RESTful HTTP アクセスを実装するモジュールが含まれています。

20.8.1. REST サーバーの実行

REST サーバーエンドポイントは Red Hat Data Grid Server の一部で、デフォルトではポート 8080 でリッスンします。サーバーをローカルで実行するには、zip ディストリビューションをダウンロードし、展開したディレクトリーで実行します。

```
bin/standalone.sh -b 0.0.0.0
```

または、docker で以下を実行します。

```
docker run -it -p 8080:8080 -e "APP_USER=user" -e "APP_PASS=changeme" jboss/infinispan-server
```

20.8.1.1. セキュリティー

REST サーバーは認証で保護されるため、使用する前にアプリケーションログインを作成する必要があります。docker で実行する場合は、APP_USER と APP_PASS のコマンドライン引数を使用して実行できますが、ローカルで実行する場合は、以下のように実行できます。

```
bin/add-user.sh -u user -p changeme -a
```

20.8.2. サポート対象プロトコル

REST Server は HTTP/1.1 と HTTP/2 プロトコルをサポートします。HTTP/1.1 アップグレード手順を実行するか、TLS/ALPN 拡張を使用して通信プロトコルをネゴシエートして、HTTP / 2 に切り替えることができます。

注記： JDK8 を使用する TLS/ALPN には、クライアントパースペクティブからの追加手順が必要です。クライアントのドキュメンテーションを参照してくださいが、Jetty ALPN エージェントまたは OpenSSL バインディングが必要になる可能性が高くなります。

20.8.3. REST API

HTTP PUT と POST メソッドは、キャッシュ名とキーに対応する URL を使用してデータをキャッシュに配置するために使用されます。要求の本文となるデータ（データは任意のものにすることができます）。他のヘッダーを使用してキャッシュ設定と動作を制御します。

20.8.3.1. データ形式

20.8.3.1.1. 設定

REST 経由で公開される各キャッシュは、MediaType で定義される設定可能なデータ形式でデータを格納します。設定の詳細は、 を参照し て ください。

ストレージ設定の例は以下のようになります。

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

MediaType が設定されていない場合、Red Hat Data Grid は以下の例外を除き、「application/octet-stream」とみなします。

- キャッシュがインデックス化される場合、「application/x-protostream」を想定します。
- キャッシュが互換性モードで設定されている場合は、"application/x-java-object" を想定します。

20.8.3.1.2. サポート対象の形式

データはストレージ形式とは異なる形式で記述および読み取ることができます。Red Hat Data Grid は、必要に応じてこれらのフォーマット間で変換できます。

以下の「標準」形式は同じに変換できます。

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

上記の形式を変換できます。

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

最後に、以下の変換もサポートされています。

- **application/x-protostream** と **application/json** *の間*

すべての REST API コールは、書き込まれたコンテンツを記述するヘッダー、または読み取り時に必要なコンテンツの形式を提供できます。Red Hat Data Grid は、値に適用される標準の HTTP/1.1 ヘッダー「Content-Type」および「Accept」をサポートし、キーと同様の効果のある "Key-Content-Type" もサポートします。

20.8.3.1.3. Accept ヘッダー

REST サーバーは RFC-2616 Accept ヘッダーに準拠しており、サポートされる変換に基づいて正しい MediaType をネゴシエートします。たとえば、データの読み取り時に以下のヘッダーを送信します。

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

これにより、Red Hat Data Grid は最初に JSON 形式でコンテンツを返します（優先度 0.8）。ストレージの形式を JSON に変換しない場合、次の形式が text/plain （2 番目に高い優先度 0.7）にフォールバックし、最後に */* にフォールバックします。これは、キャッシュ設定に基づいて自動的に表示される形式を選択します。

20.8.3.1.4. Key-Content-Type ヘッダー

ほとんどのREST APIコールでは、URLにKeyが含まれています。Red Hat Data Grid は、これらの呼び出しを処理する際に Key が java.lang.String であることを前提としていますが、異なる形式のキーに特定のヘッダー Key-Content-Type を使用できます。

例:

- byte[] KeyをBase64文字列で指定する

API呼び出し：

```
`PUT /my-cache/AQIDBDM=`
```

ヘッダー：

Key-Content-Type: application/octet-stream

- byte[] Keyを16進数の文字列で指定する。

API呼び出し：

GET /my-cache/0x01CA03042F

ヘッダー：

```
Key-Content-Type: application/octet-stream; encoding=hex
```

- *ダブルキーの指定：*

API呼び出し：

POST /my-cache/3.141456

ヘッダー：

```
Key-Content-Type: application/x-java-object;type=java.lang.Double
```

application/x-java-object の type *パラメーターは以下に制限されます。*

- ***Primitive wrapper types***
- ***java.lang.String***
- ***bytes, making* application/x-java-object;type=Bytes *equivalent to* application/octet-stream;encoding=hex**

20.8.3.2. データの配置

20.8.3.2.1. PUT /{cacheName}/{cacheKey}

上記の URL フォームの PUT 要求により、指定のキー（名前付きキャッシュがサーバーに存在する必要がある）を持つ指定のキャッシュにペイロード（ボディー）が配置されます。たとえば、http://someserver/hr/payRoll-3 （この場合は hr はキャッシュ名で、faclroll -3 が キーになります）。既存のデータは置き換えられ、Time-To-Live および Last-Modified の値が更新されます（該当する場合）。

20.8.3.2.2. POST /{cacheName}/{cacheKey}

PUT と同じですが、キャッシュ/キーの値がすでに存在する場合のみ、Http CONFLICT ステータスが返されます（およびコンテンツは更新されません）。

Headers

- Key-Content-Type: URL に存在するキーのコンテンツタイプ。
- Content-Type: 送信される Value の MediaType を指定します。
- performAsync : OPTIONAL true/false（true の場合）は即座に返され、データを独自のクラスターに複製します。一括データの挿入/拡大クラスターに役立ちます。
- timeToLiveSeconds : OPTIONAL 番号（このエントリーが自動的に削除されるまでの秒数）。パラメーターが送信されていない場合、Red Hat Data Grid は設定のデフォルト値を想定します。負の値を渡すと、常に存在するエントリーが作成されます。
- maxIdleTimeSeconds : OPTIONAL 番号（このエントリーの最終使用後の秒数）。パラメーターが送信されていない場合、Red Hat Data Grid 設定のデフォルト値。負の値を渡すと、常に存在するエントリーが作成されます。

timeToLiveSeconds または maxIdleTimeSeconds のパラメーターとして 0 を渡す

- timeToLiveSeconds および maxIdleTimeSeconds の両方が 0 の場合、キャッシュは XML/プログラムで設定したデフォルト のライフスパン 値および maxIdle 値を使用します。
- maxIdleTimeSeconds のみ が 0 の場合、これはパラメーターとして渡される timeToLiveSeconds 値を使用します（存在しない場合は -1）、XML/プログラムで設定されるデフォルトの maxIdle を使用します。
- timeToLiveSeconds のみ が 0 の場合、これは XML/プログラムで設定されるデフォルトの ライフ スパンを使用し、maxIdle はパラメーターとして送信されるすべてのものに設定されます（存在しない場合は -1）。

JSON/Protostream 変換

キャッシュがインデックス化される場合、またはとくに application/x-protostream を保存するように設定されている場合、protostream に自動的に変換される JSON ドキュメントを送受信できま

す。変換を機能させるには、protobuf スキーマを登録する必要があります。

登録は、REST 経由で登録を行うには、getfacl protobuf_metadata *キャッシュで POST/PUT を実行します。cURL の使用例：*

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

JSON ドキュメントを作成する場合は、ドキュメントに対応する protobuf メッセージ の ID を識別するために、特別なフィールド _型 をドキュメントに存在する必要があります。

たとえば、以下のスキーマについて考えてみましょう。

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

準拠 JSON ドキュメントは以下のようになります。

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

20.8.3.3. データのバックアウト

HTTP GET および HEAD は、エントリーからデータを取得するために使用されます。

20.8.3.3.1. GET /{cacheName}/{cacheKey}

これにより、指定のキー下の指定の cacheName にあるデータを応答のボディーとして返します。Content-Type ヘッダーは、メディアタイプネゴシエーションに従って応答に表示されます。ブラウザーは(CDN)のキャッシュを直接使用できます。ETag は、指定の URL のデータの状態を示す Last-Modified ヘッダーフィールドおよび Expires ヘッダーフィールドと同様に、各エントリーに対して一意に返されます。etags により、ブラウザー（およびその他のクライアント）は変更したケース（帯域幅を節約するための）でのみデータを要求できます。これは標準 HTTP であり、Red Hat Data Grid によって初期状態です。

Headers

- Key-Content-Type: URL に存在するキーのコンテンツタイプ。省略すると、application/x-java-object; type=java.lang.String が想定されます。

- accept: コンテンツを返すために必要な形式

以下のようにクエリー文字列に "extended" パラメーターを追加して、追加情報を取得できます。

GET /cacheName/cacheKey?extended

これにより、以下のカスタムヘッダーが返されます。

- cluster-Primary-Owner: このキーのプライマリー所有者のノード名

- cluster-Node-Name: リクエストを処理したサーバーの JGroups ノード名

- cluster-Physical-Address: リクエストを処理したサーバーの物理 JGroups アドレス。

20.8.3.3.2. HEAD /{cacheName}/{cacheKey}

GET と同じですが、コンテンツのみが返されます（ヘッダーフィールドのみ）。保存した内容と同じものを受け取ります。たとえば、String を保存した場合、これは戻るものです。XML または JSON の一部を保存した場合は、それを受け取る内容になります。バイナリー（ベース 64 エンコード）の Blob を保存した場合（例： Java; オブジェクト - オブジェクト）、これを自分でデシリアライズする必要があります。

GET メソッドと同様に、HEAD メソッドはヘッダーを介した拡張情報を返すこともサポートしています。上記を参照してください。

Headers

- Key-Content-Type: URL に存在するキーのコンテンツタイプ。省略すると、application/x-java-object; type=java.lang.String が想定されます。

20.8.3.4. キーの一覧表示

20.8.3.4.1. GET /{cacheName}

これにより、指定の cacheName に存在するキーの一覧が応答のボディーとして返されます。応答の形式は、以下のように Accept ヘッダーで制御できます。

- application/xml: キーのリストが XML 形式で返されます。
- application/json - キーの一覧が JSON 形式で返されます。
- text/plain: キーのリストがプレーンテキスト形式で返されます。1 行に 1 つのキー

cacheName で識別されるキャッシュが分散されると、要求を処理するノードが所有するキーのみが返されます。すべてのキーを返すには、以下のように "global" パラメーターをクエリーに追加します。

```
GET /cacheName?global
```

20.8.3.5. データの削除

データは、キャッシュキー/要素レベルで削除することも、HTTP delete メソッドを使用してキャッシュ名全体から削除できます。

20.8.3.5.1. DELETE /{cacheName}/{cacheKey}

指定したキー名をキャッシュから削除します。

Headers

- Key-Content-Type: URL に存在するキーのコンテンツタイプ。省略すると、application/x-java-object; type=java.lang.String が想定されます。

20.8.3.5.2. DELETE /{cacheName}

指定したキャッシュ名（つまり、そのパスからすべて）のエントリーをすべて削除します。操作に成功すると、200 コードを返します。

より迅速に行うことができます。

ヘッダー performAsync を true に設定して即座に返し、バックグラウンドで削除が行われるようにします。

20.8.3.6. クエリ

REST サーバーは、JSON 形式の Ickle クエリーをサポートします。鍵と値の両方で、キャッシュを application/x-protostream で設定することが重要です。キャッシュがインデックス化される場合、設定は必要ありません。

20.8.3.6.1. GET /{cacheName}?action=search&query={ickle query}

指定したキャッシュ名で Ickle クエリーを実行します。

要求パラメーター

- クエリー文字列：クエリー 文字列は必須です。
- max_results: 返す結果の数を指定します。デフォルトは 10 です。
- offset: 返す最初の結果のインデックスを任意にします。デフォルトは 0 です。
- query_mode: サーバーによって受信されるとクエリーの 実行モード が任意になります。有効な値は FETCH および BROADCAST です。デフォルトは FETCH です。

クエリの結果

結果は、1 つ以上のヒットを含む JSON ドキュメントです。例:

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

```
      "age" : 42
    }
  }, {
    "hit" : {
      "name" : "user3",
      "age" : 12
    }
  } ]
}
```

- total_results: クエリーの結果の合計数の NUMBER
- hits: ARRAY（クエリーから一致のリスト）
- hit: OBJECT（クエリーの各結果）Select 句が使用される場合は、すべてのフィールドを含めることも、フィールドのサブセットのみを含めることもできます。

20.8.3.6.2. POST /{cacheName}?action=search

GET を使用したキュークエリーと同様ですが、クエリーパラメーターを指定する代わりに要求の本文が使用されます。

例:

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

20.8.4. CORS

REST サーバーは、リクエストの起点に基づくプリフライトとルールを含む CORS をサポートします。

例:

```
<rest-connector name="rest1" socket-binding="rest" cache-container="default">
  <cors-rules>
    <cors-rule name="restrict host1" allow-credentials="false">
      <allowed-origins>http://host1,https://host1</allowed-origins>
```

```
      <allowed-methods>GET</allowed-methods>
    </cors-rule>
    <cors-rule name="allow ALL" allow-credentials="true" max-age-seconds="2000">
      <allowed-origins>*</allowed-origins>
      <allowed-methods>GET,OPTIONS,POST,PUT,DELETE</allowed-methods>
      <allowed-headers>Key-Content-Type</allowed-headers>
    </cors-rule>
  </cors-rules>
</rest-connector>
```

ルールは、ブラウザーによって設定された "Origin" ヘッダーに基づいて順番に評価されます。オリジンが "http://host1" または "https://host1" のいずれかである場合、ルール "restrict host1" が適用されます。そうでない場合は、次のルールがテストされます。"allow ALL" ルールはすべてのオリジンを許可するため、別のオリジンから送信されるすべてのスクリプトは指定されたメソッドを実行し、指定されたヘッダーを使用できます。

<cors-rule> 要素は、以下のように設定できます。

Config	説明	必須
name	仮想マシンの名前	はい
allow-credentials	CORS リクエストがクレデンシャルを使用できるようにします。	いいえ
allowed-origins	CORS 'Access-Control-Allow-Origin' ヘッダーを設定するために使用されるコンマ区切りリストで、応答がオリジンと共有されることを示します。	はい
allowed-methods	設定されたオリジンで許可されるメソッドを指定するために、プリフライト応答に CORS の 'Access-Control-Allow-Methods' ヘッダーを設定するために使用されるカンマ区切りリスト。	はい
max-age-seconds	CORS preflight リクエストヘッダーをキャッシュできる時間	いいえ
expose-headers	設定したオリジンに公開できるヘッダーを指定するためのプリフライト応答に CORS 'Access-Control-Expose-Headers' を設定するために使用されるコンマ区切りリスト。	いいえ

20.8.5. クライアント側のコード

RESTful サービスの観点からは、クライアントライブラリー/バインディングを密接に結合する必要がないことです。必要なのは、HTTP クライアントライブラリーです。Java の場合、Apache HTTP Commons Client は問題なく機能します（統合テストで使用）、java.net API を使用できます。

20.8.5.1. Ruby の例

```
# Shows how to interact with the REST api from ruby.
# No special libraries, just standard net/http
#
# Author: Michael Neale
#
require 'net/http'

uri = URI.parse('http://localhost:8080/rest/default/MyKey')
http = Net::HTTP.new(uri.host, uri.port)

#Create new entry

post = Net::HTTP::Post.new(uri.path, {"Content-Type" => "text/plain"})
post.basic_auth('user','pass')
post.body = "DATA HERE"

resp = http.request(post)

puts "POST response code : " + resp.code

#get it back

get = Net::HTTP::Get.new(uri.path)
get.basic_auth('user','pass')
resp = http.request(get)

puts "GET response code: " + resp.code
puts "GET Body: " + resp.body

#use PUT to overwrite

put = Net::HTTP::Put.new(uri.path, {"Content-Type" => "text/plain"})
put.basic_auth('user','pass')
put.body = "ANOTHER DATA HERE"

resp = http.request(put)

puts "PUT response code : " + resp.code

#and remove...
delete = Net::HTTP::Delete.new(uri.path)
delete.basic_auth('user','pass')

resp = http.request(delete)
```

```
puts "DELETE response code : " + resp.code

#Create binary data like this... just the same...

uri = URI.parse('http://localhost:8080/rest/default/MyLogo')
put = Net::HTTP::Put.new(uri.path, {"Content-Type" => "application/octet-stream"})
put.basic_auth('user','pass')
put.body = File.read('./logo.png')

resp = http.request(put)

puts "PUT response code : " + resp.code

#and if you want to do json...
require 'rubygems'
require 'json'

#now for fun, lets do some JSON !
uri = URI.parse('http://localhost:8080/rest/jsonCache/user')
put = Net::HTTP::Put.new(uri.path, {"Content-Type" => "application/json"})
put.basic_auth('user','pass')

data = {:name => "michael", :age => 42 }
put.body = data.to_json

resp = http.request(put)

puts "PUT response code : " + resp.code

get = Net::HTTP::Get.new(uri.path)
get.basic_auth('user','pass')
resp = http.request(get)

puts "GET Body: " + resp.body
```

20.8.5.2. Python 3 の例

```
import urllib.request

# Setup basic auth
base_uri = 'http://localhost:8080/rest/default'
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(user='user', passwd='pass', realm='ApplicationRealm',
uri=base_uri)
opener = urllib.request.build_opener(auth_handler)
urllib.request.install_opener(opener)

# putting data in
data = "SOME DATA HERE \!"

req = urllib.request.Request(url=base_uri + '/Key', data=data.encode("UTF-8"), method='PUT',
                             headers={"Content-Type": "text/plain"})
with urllib.request.urlopen(req) as f:
   pass
```

```
print(f.status)
print(f.reason)

# getting data out
resp = urllib.request.urlopen(base_uri + '/Key')
print(resp.read().decode('utf-8'))
```

20.8.5.3. Java の例

```
package org.infinispan;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Base64;

/**
 * Rest example accessing a cache.
 *
 * @author Samuel Tauil (samuel@redhat.com)
 */
public class RestExample {

   /**
    * Method that puts a String value in cache.
    *
    * @param urlServerAddress URL containing the cache and the key to insert
    * @param value            Text to insert
    * @param user             Used for basic auth
    * @param password         Used for basic auth
    */
   public void putMethod(String urlServerAddress, String value, String user, String password)
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");
      addAuthorization(connection, user, password);
      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();
    }

    /**
     * Method that gets a value by a key in url as param value.
     *
     * @param urlServerAddress URL containing the cache and the key to read
     * @param user             Used for basic auth
     * @param password         Used for basic auth
     * @return String value
     */
    public String getMethod(String urlServerAddress, String user, String password) throws
IOException {
        String line;
        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");
        addAuthorization(connection, user, password);
        connection.setDoOutput(true);

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

        connection.connect();

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

    private void addAuthorization(HttpURLConnection connection, String user, String pass) {
        String credentials = user + ":" + pass;
        String basic = Base64.getEncoder().encodeToString(credentials.getBytes());
```

```
        connection.setRequestProperty("Authorization", "Basic " + basic);
    }

    /**
     * Main method example.
     */
    public static void main(String[] args) throws IOException {
        RestExample restExample = new RestExample();
        String user = "user";
        String pass = "pass";
        restExample.putMethod("http://localhost:8080/rest/default/1", "Infinispan REST Test",
user, pass);
        restExample.getMethod("http://localhost:8080/rest/default/1", user, pass);
    }
}
```

20.9. MEMCACHED サーバー

Red Hat Data Grid Server ディストリビューションには、Memcached テキストプロトコル 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 Red Hat Data Grid がサポートする 1 つまたは複数の Memcached サーバーと通信できます。これらのサーバーは、各サーバーが独立して機能し、残りの通信を行わないように、スタンドアロンで機能するか、サーバーを他の Red Hat Data Grid がサポートする Memcached サーバーに複製または配布するクラスター化が可能です。そのため、クライアントにフェイルオーバー機能を提供します。Memcached サーバーを設定し、実行する方法については、Red Hat Data Grid Server の memcached サーバーガイド を参照してください。

20.9.1. クライアントエンコーディング

memcached テキストプロトコルは、クライアントによって読み取りおよび書き込みされるデータ値が raw バイトであることを前提としています。タイプネゴシエーションのサポートは ISPN-8726 の一部として memcached バイナリープロトコル 実装と共に提供されます。

memcached クライアントがサーバーからデータを取得するためにデータ型をネゴシエートしたり、異なる形式のデータを送信することは不可能ですが、サーバーはオプションで特定の Media Type でエンコードされた値を処理するように設定できます。memcached-connector 要素に client-encoding 属性を設定すると、サーバーはこの設定された形式でコンテンツを返し、クライアントはこの形式でデータも送信します。

client-encoding は、単一キャッシュが複数のリモートエンドポイント（Rest、HotRod、Memcached、Memcached）からアクセスされ、ARM テキストクライアントへの応答/要求を調整することができる場合に便利です。エンドポイント間の相互運用性の詳細については、『エンドポイント相互運用性』を参照し てください。

20.9.2. コマンドの明確化

20.9.2.1. すべてをフラッシュします。

クラスター環境でも flush_all コマンドは、呼び出しが到達する Red Hat Data Grid Memcached サーバーを消去します。このフラッシュをクラスター内の他のノードに伝播しようとは試みません。これは、遅延ユースケースを持つ flush_all を Red Hat Data Grid Memcached サーバーで再現できるように行います。flush_all に遅延を渡す目的は、フルに異なる Memcached サーバーをフラッシュできるようにすることです。そのため、すべての Memcached サーバーが空になるため、要求でデータベースをオーバーロードしないようにします。詳細は、flush_all の Memcached テキストプロトコルのセクションを 参照してください。

20.9.3. サポートされない機能

このセクションでは、ある理由またはその他の理由で memcached テキストプロトコルのそれらの部分について説明しますが、現在 Red Hat Data Grid ベースの memcached 実装ではサポートされていません。

20.9.3.1. 個別の統計

C/C\\ ベースである元の memcached 実装と Java ベースの Red Hat Data Grid 実装の違いのため、いくつかの一般的な目的の統計がいくつかあります。このようなサポート対象外の統計では、Red Hat Data Grid memcached サーバーは常に 0 を返します。

サポートされない統計

- pid
- pointer_size
- rusage_user
- rusage_system
- bytes
- curr_connections

- *total_connections*
- *connection_structures*
- *auth_cmds*
- *auth_errors*
- *limit_maxbytes*
- *threads*
- *conn_yields*
- *回収*

20.9.3.2. 統計設定

テキストプロトコルの設定統計セクションは、自発性により実装されていません。

20.9.3.3. 引数パラメーターの設定

Memcached サーバーに送信できる引数には文書化されていないため、Red Hat Data Grid Memcached サーバーはいずれの引数を stats コマンドに渡すことはサポートしません。パラメーターが渡されると、Red Hat Data Grid Memcached サーバーは CLIENT_ERROR で応答します。

20.9.3.4. 古くなった時間パラメーターの削除

Memcached は、コマンドを削除するためにオプションの保持時間パラメーターを受け入れないため、Red Hat Data Grid ベースの memcached サーバーはそのような機能を実装しません。

20.9.3.5. 詳細コマンド

Red Hat Data Grid ロギングはログレベルのみを定義するため、詳細コマンドはサポートされていません。

20.9.4. 非 Java クライアントからの Red Hat Data Grid Memcached サーバーとの通信

本セクションでは、python スクリプトなど、java 以外のクライアントで Red Hat Data Grid memcached サーバーと通信する方法を説明します。

20.9.4.1. マルチクラスターサーバーのチュートリアル

この例では、元の memcached 実装では利用できない Red Hat Data Grid memcached severs のディストリビューション機能について取り上げます。

- 2 つのクラスター化されたノードを起動します。この設定は、GUI デモに使用されるものと同じです。

```
$ ./bin/standalone.sh -c clustered.xml -Djboss.node.name=nodeA
$ ./bin/standalone.sh -c clustered.xml -Djboss.node.name=nodeB -
Djboss.socket.binding.port-offset=100
```

または、以下を使用します。

```
$ ./bin/domain.sh
```

2 つのノードを自動的に起動します。

- ポート 11211 にバインドされている Red Hat Data Grid memcached サーバーに対して、基本的にいくつかの書き込み操作を実行する test_memcached_write.py スクリプトを実行します。スクリプトが正常に実行されると、以下のような出力が表示されます。

```
Connecting to 127.0.0.1:11211
Testing set ['Simple_Key': Simple value] ... OK
Testing set ['Expiring_Key' : 999 : 3] ... OK
Testing increment 3 times ['Incr_Key' : starting at 1 ]
Initialise at 1 ... OK
Increment by one ... OK
Increment again ... OK
Increment yet again ... OK
Testing decrement 1 time ['Decr_Key' : starting at 4 ]
Initialise at 4 ... OK
```

```
Decrement by one ... OK
Testing decrement 2 times in one call ['Multi_Decr_Key' : 3 ]
Initialise at 3 ... OK
Decrement by 2 ... OK
```

- 127.0.0.1:11311 にバインドされているサーバーに接続する test_memcached_read.py スクリプトを実行し、writer スクリプトが最初のサーバーに書き込まれたデータを読み取れることを確認します。スクリプトが正常に実行されると、以下のような出力が表示されます。

```
Connecting to 127.0.0.1:11311
Testing get ['Simple_Key'] should return Simple value ... OK
Testing get ['Expiring_Key'] should return nothing... OK
Testing get ['Incr_Key'] should return 4 ... OK
Testing get ['Decr_Key'] should return 3 ... OK
Testing get ['Multi_Decr_Key'] should return 1 ... OK
```

20.10. REMOTE GRID でのコードの実行

前述のセクションでは、グリッドでのコードの実行を説明しました。ただし、これらのメソッドは、グリッドに直接アクセスできる埋め込みのシナリオで使用するように設計されています。本セクションでは、グリッドに接続されたリモートクライアントを使用して、同様の機能を実行する方法を説明します。

20.11. スクリプト

スクリプトとは、リモートクライアントからサーバー側のスクリプトを呼び出すことができる Red Hat Data Grid Server の機能です。スクリプトスクリプトは JDK の javax.script ScriptEngines を活用するため、それを提供する JVM 言語を使用できるようにします。デフォルトでは、JDK には JavaScript を実行できる ScriptEngine である Nashorn が同梱されています。

20.11.1. スクリプトのインストール

スクリプトは、'___script_cache' という名前の特別なスクリプトキャッシュに保存されます。そのため、スクリプトをキャッシュ自体に追加するのと同じくらい簡単です。スクリプト名にファイル名の拡張子（例： +myscript.js）が含まれている場合、その拡張は実行に使用されるエンジンを決定します。または、スクリプトメタデータを使用してスクリプトエンジンを選択できます（以下を参照）。セキュリティーが有効な場合、リモートプロトコルを介してスクリプトキャッシュにアクセスするには、ユーザーが「___script_manager」ロールに属する必要があります。

20.11.2. スクリプトのメタデータ

スクリプトのメタデータは、スクリプトの実行方法に影響を与えるためにユーザーがサーバーに提供できるスクリプトに関する追加情報です。スクリプトの最初の行については、特別にフォーマットさ

れたコメントに含まれています。

プロパティーは key=value ペアとして指定され、コンマで区切ります。複数の異なるコメントスタイルを使用できます： The //, ;;, # dependsing on the scripting language you use.必要に応じて複数の行にメタデータを分割でき、一重引用符(')または二重引用符(")引用符を使用して値を区切ることができます。

以下は、有効なメタデータコメントの例です。

```
// name=test, language=javascript
// mode=local, parameters=[a,b,c]
```

20.11.2.1. メタデータのプロパティー

以下のメタデータプロパティーキーを使用できます。

- Mode: スクリプトの実行モードを定義します。以下の値のいずれかになります。
  - local: スクリプトは、要求を処理するノードによってのみ実行されます。スクリプト自体はクラスター化された操作を呼び出すことができます。
  - Distributed: 分散エグゼキューターサービスを使用してスクリプトを実行します。
- language: スクリプトの実行に使用するスクリプトエンジンを定義します（例：Javascript）。
- 拡張機能： スクリプトの実行に使用するスクリプトエンジンを指定する代替方法（js など）
- Role: スクリプトの実行に必要な特定のロール
- parameters: このスクリプトの有効なパラメーター名の配列。この一覧に含まれていないパラメーター名を指定する呼び出しにより、例外が発生します。
-

datatype: キャッシュに保存されているデータのタイプ、パラメーターおよび戻り値に関するメディアタイプ（MIME とも呼ばれる）形式の情報を提供する任意のプロパティー。現在、テキスト/警告である単一の値のみを許可します。charset=utf-8 は、データが String UTF-8 形式であることを示します。このメタデータパラメーターは、特定タイプのデータのみをサポートするリモートクライアント用に設計されており、パラメーターの取得、保存、および使用が容易になります。

実行モードはスクリプトの特徴であるため、異なるモードでスクリプトを呼び出すためにクライアントでは特別なことは必要ありません。

20.11.3. スクリプトバインディング

Red Hat Data Grid 内のスクリプトエンジンは、スクリプト実行の範囲にあるバインディングとして複数の内部オブジェクトを公開します。以下のとおりです。

- *cache: スクリプトを実行するキャッシュ*
- *マーシャラー： キャッシュへのマーシャリング/アンマーシャリングデータに使用するマーシャラー*
- *cacheManager: キャッシュの cacheManager*
- *scriptingManager: スクリプトの実行に使用されるスクリプトマネージャーのインスタンス。これは、スクリプトから他のスクリプトを実行するのに使用できます。*

20.11.4. スクリプトのパラメーター

上記の標準バインディング以外に、スクリプトを実行すると、バインディングとしても表示される名前付きパラメーターのセットを渡すことができます。パラメーターは name,value ペアとして渡されます。name は文字列で、値には、使用中のマーシャラーによって認識される値を使用できます。

以下は、乗数と乗数という 2 つのパラメーターを取る JavaScript スクリプトの例です。最後の操作は式の評価であるため、その結果は invoker に返されます。

```
// mode=local,language=javascript
multiplicand * multiplier
```

スクリプトをスクリプトキャッシュに保存するには、以下の Hotgitops コードを使用します。

```
RemoteCache<String, String> scriptCache = cacheManager.getCache("___script_cache");
scriptCache.put("multiplication.js",
  "// mode=local,language=javascript\n" +
  "multiplicand * multiplier\n");
```

20.11.5. Hotgitops Java クライアントを使用したスクリプトの実行

以下の例は、2 つの名前付きパラメーターを渡して上記のスクリプトを呼び出す方法を示しています。

```
RemoteCache<String, Integer> cache = cacheManager.getCache();
// Create the parameters for script execution
Map<String, Object> params = new HashMap<>();
params.put("multiplicand", 10);
params.put("multiplier", 20);
// Run the script on the server, passing in the parameters
Object result = cache.execute("multiplication.js", params);
```

20.11.6. 分散実行

以下は、すべてのノード上で実行されるスクリプトです。各ノードはそのアドレスを返し、すべてのノードからの結果が List で収集され、クライアントに戻ります。

```
// mode:distributed,language=javascript
cacheManager.getAddress().toString();
```

20.12. サーバータスク

サーバータスクは、Java 言語で定義されるサーバー側のスクリプトです。サーバータスクを開発するには、infinispan-tasks-api モジュールで定義される org.infinispan.tasks.ServerTask インターフェースを拡張するクラスを定義する必要があります。

一般的なサーバータスクの実装では、以下のメソッドが実装されます。

- setTaskContext は、サーバータスクが実行コンテキスト情報にアクセスできるようにします。これには、タスクパラメーター、実行されたタスクのキャッシュ参照などが含まれます。通常、実装者はこの情報をローカルに保存し、タスクが実際に実行されたときに使用します。
-

getName はタスクの一意の名前を返す必要があります。クライアントはこの名前を使用してタスクを呼び出します。

- *getExecutionMode* は、N ノードのクラスターで 1 つのノードでタスクを呼び出すか、または N ノードで呼び出すかどうかを決定するために使用されます。たとえば、ストリーム処理を呼び出すサーバータスクは、クラスター内の 1 つのノードでのみ実行する必要があります。これは、ストリーム処理自体が、処理がクラスター内のすべてのノードに分散されるためです。

- *call* は、ユーザーがサーバータスクを呼び出すときに呼び出されるメソッドです。

以下は、通知する人の名前として使用する hello greet タスクの例になります。

```
package example;

import org.infinispan.tasks.ServerTask;
import org.infinispan.tasks.TaskContext;

public class HelloTask implements ServerTask<String> {

  private TaskContext ctx;

  @Override
  public void setTaskContext(TaskContext ctx) {
    this.ctx = ctx;
  }

  @Override
  public String call() throws Exception {
    String name = (String) ctx.getParameters().get().get("name");
    return "Hello " + name;
  }

  @Override
  public String getName() {
    return "hello-task";
  }

}
```

タスクが実装されたら、jar 内でラップする必要があります。その後、jar は Red Hat Data Grid Server にデプロイされます。Red Hat Data Grid Server は サービスローダーパターン を使用してタスクをロードするため、実装はこれらの要件に従う必要があります。たとえば、サーバータスクの実装にはゼロ引数のコンストラクターが必要です。

さらに、jar には jar に含まれるサーバータスクの完全修飾名が含まれる META-

INF/services/org.infinispan.tasks.ServerTask ファイルが含まれている必要があります。以下に例を示します。

```
example.HelloTask
```

jar がパッケージ化された場合、次の手順として jar を Red Hat Data Grid Server にプッシュします。このサーバーは WildFly Application Server をサポートしているので、Maven Wildfly の Maven プラグイン をこれに使用できます。

```
<plugin>
  <groupId>org.wildfly.plugins</groupId>
  <artifactId>wildfly-maven-plugin</artifactId>
  <version>1.2.0.Final</version>
</plugin>
```

次に、コマンドラインから以下を呼び出します。

```
$ mvn package wildfly:deploy
```

Wildfly アプリケーションサーバーへのデプロイメントの他の方法も 説明 します。

タスクの実行は、以下のコードを使用して実行できます。

```
// Create a configuration for a locally-running server
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer().host("127.0.0.1").port(11222);

// Connect to the server
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());

// Obtain the remote cache
RemoteCache<String, String> cache = cacheManager.getCache();

// Create task parameters
Map<String, String> parameters = new HashMap<>();
parameters.put("name", "developer");

// Execute task
String greet = cache.execute("hello-task", parameters);
System.out.println(greet);
```

第21章 互換性モード

警告

互換性モードは非推奨となり、Red Hat Data Grid から削除されます。リモートエンドポイント間の相互運用性を実現するには、プロトコルの相互運用性機能を使用する必要があります。「プロトコル相互運用性」を 参照してください。

互換性モードは、複数の方法で Red Hat Data Grid にアクセスできるように Red Hat Data Grid キャッシュを設定します。このような互換性を実現するには、各エンドポイントの異なる形式間でコンテンツを変換できるように、Red Hat Data Grid から追加の作業が必要となります。そのため、互換性モードがデフォルトで無効になっている理由です。

21.1. 互換性モードの有効化

互換性モードが想定どおりに機能するには、すべてのエンドポイントを同じキャッシュマネージャーで設定する必要があり、同じキャッシュと通信する必要があります。ブランドの新しい Red Hat Data Grid Server ディストリビューション を使用している場合、これはすべて行われます。スタンドアロンのユニットテストで実験的な場合には、このクラスは 1 つのクラス から複数のエンドポイントを起動する方法を示しています。

そのため、Red Hat Data Grid の互換性モードの使用を開始するには、XML 経由で有効にする必要があります。

infinispan.xml

```
<local-cache>
  <compatibility/>
</local-cache>
```

または、プログラムで以下を実行します。

```
ConfigurationBuilder builder = ...
builder.compatibility().enable();
```

Red Hat Data Grid の互換性モードについて覚えておくべき重要なことは、可能な場合はデータのアンマーシャリングやデシリアライズを試みることです。最も一般的なユースケースは、Java オブジェクトを保存し、Java オブジェクトをデシリアライズされた形式で格納することです。埋め込みキャッシュから簡単に使用できます。ここでは、仮定を念頭に置いています。たとえば、Hot336 経由で何かを保存する場合は、Java で書かれた reference Hotgitops クライアントから取得され、バイナリーペイロードが非常にコンパクトにするマーシャラーを使用します。そのため、Hotset オペレーションが互換性層に到達すると、デフォルトでは Java Hotgitops クライアントによって使用されるデフォルトのマーシャラーと同じマーシャラーを使用して、アンマーシャリングを試みます。そのため、ほとんどのケースで適切な追加設定なしのサポートを提供します。

21.1.1. オプション： 互換性マーシャラの設定

クライアントは Java 以外の言語（ Ruby または Python など）用に作成された Hotで書かれたクライアントを使用している可能性があります。この場合、シリアライズされたペイロードをキャッシュに保存する Java オブジェクトに変換するか、シリアライズフォームに維持するカスタムマーシャラーを設定する必要があります。どちらのオプションも有効ですが、埋め込みキャッシュを使用している場合は、Red Hat Data Grid から取得されるオブジェクトの種類に影響します。マーシャラーは このインターフェース を実装することが予想されます。互換性マーシャラーの設定は任意で、XML で行うことができます。

infinispan.xml

```
<local-cache>
  <compatibility marshaller="com.acme.CustomMarshaller"/>
</local-cache>
```

または、プログラムで以下を実行します。

```
ConfigurationBuilder builder = ...
builder.compatibility().enable().marshaller(new com.acme.CustomMarshaller());
```

このマーシャラーロジックの具体例は、SpyMemcachedCompatibleMarshaller にあります。spy Memcached は独自のトランスコーダーを使用してオブジェクトをマーシャリングするため、作成される互換性マーシャラーは Spy Memcached クライアント経由で保存されるマーシャリング/アンマーシャリングデータに対して行われます。言い換えると、Spy Memcached 経由で保存されたデータを取得する場合は、Java Hotgitops クライアントが同じマーシャラーを使用するように設定できます。これは、Spy Memcached マーシャラーが置かれているテストを正確に降格するものです。

21.2. コードの例

アクションの互換性を示す最適なコード例は、Red Hat Data Grid Compatibility Mode テストスイート で確認できますが、近い将来的に開発される予定です。

第22章 プロトコルの相互運用性

クライアントは、REST や Hot336 などのエンドポイントを介して Red Hat Data Grid でデータを交換します。

各エンドポイントは異なるプロトコルを使用するため、クライアントは適切な形式でデータを読み書きできます。Red Hat Data Grid は同時に複数のクライアントと相互運用できるため、クライアント形式とストレージフォーマット間でデータを変換する必要があります。

Red Hat Data Grid エンドポイントの相互運用性を設定するには、キャッシュに保存されているデータの形式を設定する MediaType を定義する必要があります。

22.1. メディアタイプとエンドポイントの相互運用性に関する考慮点

特定のメディアタイプでデータを保存するように Red Hat Data Grid を設定すると、クライアントの相互運用性に影響があります。

REST クライアントは エンコードされたバイナリーデータ の送受信をサポートしますが、JSON、XML、プレーンテキストなどのテキスト形式を処理する方が適切です。

Memcached テキストクライアントは String ベースのキーおよび byte[] 値を処理できますが、サーバーでデータ型をネゴシエートすることはできません。これらのクライアントは、プロトコル定義が原因でデータ形式を処理するのに柔軟性が高くありません。

Java HotTEMPLATES クライアントは、キャッシュに存在するエンティティーを表す Java オブジェクトを処理するのに適しています。Java Hotgitops クライアントはマーシャリング操作を使用して、これらのオブジェクトをバイトアレイにシリアライズおよびデシリアライズします。

同様に、C++、C#、Javascript クライアントなどの Java Hot Warehouse クライアント以外のクライアントは、各言語でオブジェクトを処理するのに適しています。ただし、Java Hotgitops 以外のクライアントは、プラットフォームに依存しないデータ形式を使用して Java HotTEMPLATES クライアントと相互運用できます。

22.2. テキストベースのストレージによる REST、HOT336、および MEMCACHED の相互運用性

テキストベースのストレージ形式でキーと値を設定できます。

たとえば、text/plain、charset=UTF-8、またはその他の文字セットを指定して、プレーンテキストをメディアタイプとして設定します。JSON(application/json)や XML(application/xml)などのテキストベースの形式のメディアタイプを、任意の文字セットで指定することもできます。

以下の例では、text/plain: charset=UTF-8 メディアタイプでエントリーを保存するようにキャッシュを設定します。

```
<cache>
  <encoding>
    <key media-type="text/plain; charset=UTF-8"/>
    <value media-type="text/plain; charset=UTF-8"/>
  </encoding>
</cache>
```

テキストベースの形式でデータの交換を処理するには、org.infinispan.commons.marshall.StringMarshaller マーシャラーで Hotgitops クライアントを設定する必要があります。

以下のように、キャッシュから書き込みおよび読み取りを行う際に、REST クライアントは正しいヘッダーも送信する必要があります。

- *書き込み： Content-Type: text/plain; charset=UTF-8*
- *読み取り： Accept: text/plain; charset=UTF-8*

Memcached クライアントでは、テキストベースの形式を処理する設定は必要ありません。

この設定は、以下と互換性があります。	
REST クライアント	はい
Java Hot Rod クライアント	はい
Memcached クライアント	はい
Java 以外の Hot Rod クライアント	▮
クエリーおよびインデックス化	▮

この設定は、以下と互換性があります。	
カスタム Java オブジェクト	▮

22.3. カスタム JAVA オブジェクトでの REST、HOTFSPROGS、および MEMCACHED の相互運用性

エントリーをマーシャリングされたカスタム Java オブジェクトとしてキャッシュに保存する場合は、マーシャリングされたストレージの MediaType を使用してキャッシュを設定する必要があります。

Java Hotgitops クライアントは、JBoss マーシャリングストレージフォーマットをデフォルトとして使用し、エントリーをカスタム Java オブジェクトとしてキャッシュに保存します。

以下の例では、application/x-jboss-marshalling メディアタイプでエントリーを保存するようにキャッシュを設定します。

```
<distributed-cache name="my-cache">
  <encoding>
    <key media-type="application/x-jboss-marshalling"/>
    <value media-type="application/x-jboss-marshalling"/>
  </encoding>
</distributed-cache>
```

Protostream マーシャラーを使用する場合は、MediaType を application/x-protostream に設定します。UTF8Marshaller の場合は、MediaType を text/plain に設定します。

ヒント

Hotgitops クライアントのみがキャッシュと対話する場合は、MediaType を設定する必要はありません。

REST クライアントはテキスト形式の処理に最も適しているため、キーには java.lang.String などのプリミティブを使用する必要があります。それ以外の場合は、REST クライアントは、サポートされているバイナリーエンコーディング を使用してキーを bytes[] として処理する必要があります。

REST クライアントは、XML または JSON 形式でキャッシュエントリーの値を読み取ることができます。ただし、クラスがサーバーで使用できる必要があります。

Memcached クライアントからデータを読み書きするには、キーに java.lang.String を使用する必要があります。値は保存され、マーシャリングされたオブジェクトとして返されます。

一部の Java Memcached クライアントでは、オブジェクトをマーシャリングおよびアンマーシャリングするデータトランスフォーマーが許可されます。また、Memcached サーバーモジュールを設定して、レスポンスを言語に依存しない 'JSON' などの異なる形式でエンコードすることもできます。これにより、キャッシュのストレージフォーマットが Java 固有の場合でも、Java 以外のクライアントはデータと対話できます。Red Hat Data Grid Memcached サーバーモジュールの「クライアントエンコーディングの詳細」を参照してください。

注記

Java オブジェクトをキャッシュに保存するには、エンティティークラスを Data Grid にデプロイする必要があります。「エンティティークラスのデプロイ」を参照してください。

この設定は、以下と互換性があります。	
REST クライアント	はい
Java Hot Rod クライアント	はい
Memcached クライアント	はい
Java 以外の Hot Rod クライアント	▮
クエリーおよびインデックス化	▮
カスタム Java オブジェクト	はい

22.4. JAVA および非 JAVA クライアントの相互運用性と PROTOBUF の組み合わせ

キャッシュに Protobuf エンコードエントリーとしてデータを格納すると、Java および 非 Java クライアントが任意のエンドポイントからのキャッシュにアクセスしてクエリーできるようにするプラットフォームに依存しない設定が提供されます。

キャッシュに対してインデックスが設定されている場合、Red Hat Data Grid は application/x-protostream メディアタイプでキーと値を自動的に保存します。

キャッシュにインデックスが設定されていない場合は、以下のように application/x-protostream メ

ディアタイプでエントリーを保存するように設定できます。

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

Red Hat Data Grid は、application/x-protostream と application/json の間で変換を行います。これにより、REST クライアントは JSON 形式のデータを読み書きできます。ただし、REST クライアントは以下のように正しいヘッダーを送信する必要があります。

ヘッダーの読み取り

```
Read: Accept: application/json
```

ヘッダーの書き込み

```
Write: Content-Type: application/json
```

重要

application/x-protostream メディアタイプは Protobuf エンコーディングを使用します。そのため、クライアントが使用するエンティティーおよびマーシャラーを記述するプロトコルバッファースキーマ定義を登録する必要があります。「Sor ing Protobuf エンティティー」を参照してください。

この設定は、以下と互換性があります。	
REST クライアント	はい
Java Hot Rod クライアント	はい
Java 以外の Hot Rod クライアント	はい
クエリーおよびインデックス化	はい
カスタム Java オブジェクト	はい

22.5. カスタムコードの相互運用性

Red Hat Data Grid でカスタムコードをデプロイできます。たとえば、スクリプト、タスク、リス

ナー、コンバーター、およびマージポリシーをデプロイできます。カスタムコードはキャッシュで直接データにアクセスできるため、異なるエンドポイントを介してキャッシュ内のデータにアクセスするクライアントと相互運用する必要があります。

たとえば、他のクライアントはデータをバイナリー形式で保存する一方で、キャッシュに保存されているカスタムオブジェクトを処理するリモートタスクを作成できます。

カスタムコードとの相互運用性を処理するには、オンデマンドでデータを変換したり、データを POJO(Plain Old Java Objects)として保存したりできます。

22.5.1. オンデマンドデータの変換

キャッシュが application/x-protostream や application/x-jboss-marshalling などのバイナリー形式でデータを保存するように設定されている場合、Java オブジェクトをメディアタイプとして使用してキャッシュ操作を実行するように設定できます。「MediaType プログラムによるオーバーライド」を 参照してください。

このアプローチにより、リモートクライアントはキャッシュエントリーの保存にバイナリー形式を使用できますが、これが最適です。ただし、バイナリー形式と Java オブジェクト間で変換できるようにエンティティークラスをサーバーで利用できるようにする必要があります。

さらに、キャッシュが Protobuf(application/x-protostream)をバイナリー形式として使用する場合、Data Grid がカスタムコードからアンマーシャリングできるように protostream マーシャラーをデプロイする必要があります。「Deploying Protostream Marshallers 」を参照してください。

22.5.2. POJO としてのデータの保存

サーバーにアンマーシャリングされた Java オブジェクトを保存することは推奨されません。これには、リモートクライアントがキャッシュから読み取ってからリモートクライアントがキャッシュに書き込むときにデータをシリアライズするには、Red Hat Data Grid が必要です。

以下の例では、application/x-java-object のメディアタイプでエントリーを保存するようにキャッシュを設定します。

```
<distributed-cache name="my-cache">
  <encoding>
    <key media-type="application/x-java-object"/>
    <value media-type="application/x-java-object"/>
  </encoding>
</distributed-cache>
```

JBoss マーシャラーまたはデフォルトの Java シリアライゼーションメカニズムのいずれかを使用して、データをキャッシュに POJO として格納する場合は、ホットpidクライアントはサポートされるマーシャラーを使用する必要があります。また、クラスをサーバーにデプロイする必要があります。

REST クライアントは、Red Hat Data Grid が現在 JSON または XML の Java オブジェクトとの間で変換できるストレージ形式を使用する必要があります。

注記

Java オブジェクトをキャッシュに保存するには、エンティティークラスを Red Hat Data Grid にデプロイする必要があります。「エンティティークラスのデプロイ」を参照してください。

Memcached クライアントは、デフォルトで JBoss のマーシャリングされたペイロードである、シリアライズされた POJO のバージョンを送信および受信する必要があります。ただし、適切な Memcached コネクターでクライアント エンコーディング を設定する場合、Memcached クライアントが JSON などのプラットフォームに依存しない形式を使用するようにストレージ形式を変更します。

この設定は、以下と互換性があります。	
REST クライアント	はい
Java Hot Rod クライアント	はい
Java 以外の Hot Rod クライアント	▮
クエリーおよびインデックス化	必要。ただし、クエリーとインデックスは、エンティティーにアノテーションが付けられている場合にのみ POJO と連携 し ます。
カスタム Java オブジェクト	はい

22.6. エンティティークラスのデプロイ

エントリーをカスタム Java オブジェクトまたは POJO としてキャッシュに保存する予定の場合には、エンティティークラスを Red Hat Data Grid にデプロイする必要があります。クライアントは常にオブジェクトを bytes[] として交換します。エンティティークラスはこれらのカスタムオブジェクトを表すため、Red Hat Data Grid はそれらをシリアライズおよびデシリアライズすることができます。

サーバーでエンティティークラスを使用できるようにするには、以下を行います。

- エンティティーおよび依存関係が含まれる JAR ファイルを作成します。

- Red Hat Data Grid を実行している場合は停止します。

  Red Hat Data Grid は、起動時にエンティティークラスをロードします。サーバーを実行している場合は、エンティティークラスを Red Hat Data Grid で利用できるようにすることはできません。

- JAR ファイルを $RHDG_HOME/standalone/deployments/ ディレクトリーにコピーします。

- 以下の例のように、JAR ファイルをキャッシュマネージャー設定でモジュールとして指定します。

```
<cache-container name="local" default-cache="default">
  <modules>
    <module name="deployment.my-entities.jar"/>
  </modules>
  ...
</cache-container>
```

第23章 セキュリティー

Red Hat Data Grid のセキュリティーは、複数のレイヤーに実装されています。

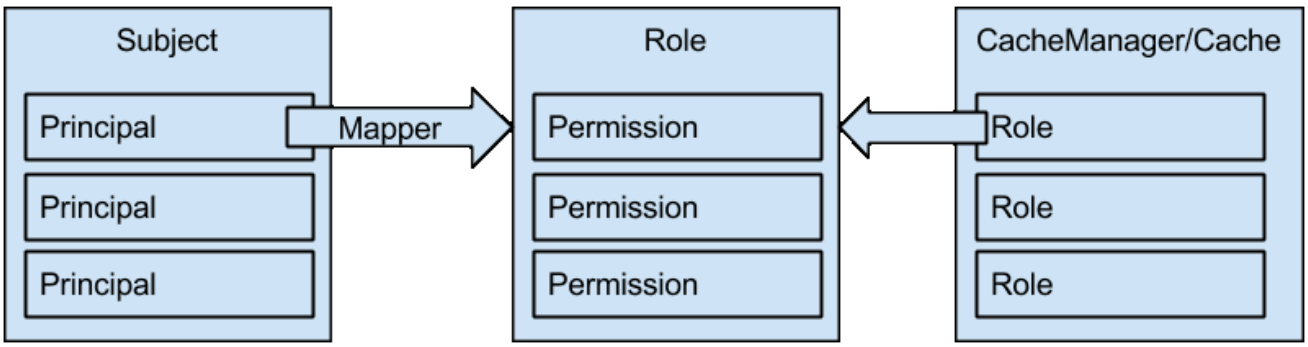
- CacheManager、Caches、およびデータに粒度の細かいアクセス制御を提供するコアライブラリー内で
- リモートプロトコルを介してリモートクライアントから認証情報を取得し、暗号化を使用してトランスポートをセキュアにする。
- 承認されたノードのみが暗号化を使用してトランスポートを保護できるようにするためのクラスター内のノード間でのノード

互換性と統合を最大化するために、Red Hat Data Grid は、X.509 証明書、SSL/TLS 暗号化、Kerberos/GSSAPI など、可能な限り幅広いセキュリティー標準を使用します。また、外部依存関係をプルし、サードパーティーライブラリーやコンテナーとの統合を容易にするために、実装では標準の Java セキュリティーライブラリー（JAAS、JSSE、JCE、SASL など）が提供する機能を利用できます。このため、Red Hat Data Grid コアライブラリーは、インターフェースと基本的な実装のセットのみを提供します。

23.1. 組み込みセキュリティー

アプリケーションは、同じ JVM 内の API を使用して Red Hat Data Grid と対話します。Red Hat Data Grid API によって公開される 2 つの主なコンポーネントは CacheManager と Caches です。アプリケーションがセキュアな CacheManager および Cache と相互作用する場合は、Red Hat Data Grid のセキュリティー層が必要なロールおよびパーミッションのセットに対して検証するアイデンティティーを提供する必要があります。ユーザーアプリケーションが提供するアイデンティティーに十分なパーミッションがある場合、アクセスは付与され、セキュリティー違反を示す例外が発生します。アイデンティティーは、複数の Principal のラッパーである javax.security.auth.Subject クラスによって表されます（例： ユーザーと所属するグループ）。Principal 名は独自のシステムに依存するため（LDAP の識別名など）、Red Hat Data Grid は Principal 名をロールにマップできる必要があります。これにより、ロールは 1 つ以上のパーミッションを表します。以下の図は、さまざまな要素間の関係を示しています。

図23.1 ロール/パーミッションのマッピング



23.1.1. 組み込みパーミッション

キャッシュマネージャーまたはキャッシュへのアクセスは、必要なパーミッションの一覧を使用して制御されます。パーミッションは、操作されるデータタイプではなく、上記のエンティティーのいずれかで実行されるアクションのタイプに関連します。これらのパーミッションの一部は、適用可能な名前付きエンティティー（名前付きキャッシュなど）に絞り込むことができます。エンティティーのタイプに応じて、利用可能なパーミッションのタイプが異なります。

23.1.1.1. Cache Manager のパーミッション

- *CONFIGURATION(defineConfiguration): 新しいキャッシュ設定を定義するかどうか。*
- *LISTEN(addListener): キャッシュマネージャーに対してリスナーを登録するかどうか。*
- *LIFECYCLE(stop): キャッシュマネージャーを停止できるかどうか*
- *ALL: 上記のすべてを含む便利なパーミッション*

23.1.1.2. キャッシュ権限

- *READ(get, contains): キャッシュからエントリーを取得できるかどうか。*
- *WRITE（計算、putIfAbsent、置換、削除、エビクト）：キャッシュからデータの書き込み/交換/削除/削除/エビクトができるかどうか。*
- *EXEC（distexec、stream）：キャッシュに対してコード実行を実行できるかどうか。*

- LISTEN(addListener): キャッシュに対してリスナーを登録できるかどうか。
- BULK_READ（keySet、value、entrySet、query）：一括取得操作を実行できるかどうか。
- BULK_WRITE(clear, putAll): 一括書き込み操作を実行できるかどうか
- LIFECYCLE（起動、停止）：キャッシュを開始/停止できるかどうか
- ADMIN(getVersion, addInterceptor*, removeInterceptor, getInterceptorChain, getEvictionManager, getComponentRegistry, getDistributionManager, getAuthorizationManager, evict, getRpcManager, getCacheConfiguration, getCacheManager, getInvocationContextContainer, setAvailability, getDataContainer, getStats, getXAResource): 基礎となるコンポーネント/内部構造にアクセスできるかどうか。
- ALL: 上記のすべてを含む便利なパーミッション
- ALL_READ: READ および BULK_READ の組み合わせ
- ALL_WRITE: WRITE と BULK_WRITE を組み合わせます。

一部のパーミッションは、より有用なものにするために他のユーザーと組み合わせる必要があります。たとえば、「supervisors」のみがストリーム操作を実行できるようにし、「標準」ユーザーは配置しか実行できず、以下のマッピングを定義します。

```
<role name="standard" permission="READ WRITE" />
<role name="supervisors" permission="READ WRITE EXEC"/>
```

23.1.2. 組み込み API

DefaultCacheManager がプログラムによる設定または宣言型の設定を使用して有効化された状態で構築されると、基礎となるキャッシュで操作を呼び出す前にセキュリティーコンテキストを確認する SecureCache を返します。SecureCache は、アプリケーションが低レベルではないオブジェクト（DataContainer など）を取得できないようにします。Java では、特定のアイデンティティーでコードを実行する場合は通常、PrivilegedAction 内で実行されるコードをラップします。

```
import org.infinispan.security.Security;

Security.doAs(subject, new PrivilegedExceptionAction<Void>() {
public Void run() throws Exception {
   cache.put("key", "value");
}
});
```

Java 8 を使用している場合は、上記の呼び出しを簡素化できます。

```
Security.doAs(mySubject, PrivilegedAction<String>() -> cache.put("key", "value"));
```

通常の Subject.doAs（）の代わりに Security.doAs（）を使用することに注意してください。Red Hat Data Grid では、Security.doAs（）を使用してアプリケーションのセキュリティーモデルに固有の理由で AccessControlContext を変更する必要がある場合を除き、どちらかを使用できます。現在のサブジェクトが必要な場合は、以下を使用します。

```
Security.getSubject();
```

Red Hat Data Grid のコンテキストまたは AccessControlContext から Subject を自動的に取得します。

Red Hat Data Grid は、完全な SecurityManager での実行もサポートします。Red Hat Data Grid ディストリビューションには、security.policy ファイルのサンプルが含まれており、このファイルを JVM に提供する前に適切なパスでカスタマイズする必要があります。

23.1.3. 埋め込み設定

設定には、global および per-cache の 2 つのレベルがあります。グローバル設定はロール/パーミッションマッピングのセットを定義し、各キャッシュは承認チェックと必要なロールを有効にするかどうかを決定できます。

プログラマティック

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
global
  .security()
    .authorization().enable()
      .principalRoleMapper(new IdentityRoleMapper())
      .role("admin")
        .permission(AuthorizationPermission.ALL)
      .role("supervisor")
```

```
            .permission(AuthorizationPermission.EXEC)
            .permission(AuthorizationPermission.READ)
            .permission(AuthorizationPermission.WRITE)
          .role("reader")
            .permission(AuthorizationPermission.READ);
ConfigurationBuilder config = new ConfigurationBuilder();
config
  .security()
    .authorization()
      .enable()
      .role("admin")
      .role("supervisor")
      .role("reader");
```

宣言的 (Declarative)

```
<infinispan>
  <cache-container default-cache="secured" name="secured">
    <security>
      <authorization>
        <identity-role-mapper />
        <role name="admin" permissions="ALL" />
        <role name="reader" permissions="READ" />
        <role name="writer" permissions="WRITE" />
        <role name="supervisor" permissions="READ WRITE EXEC"/>
      </authorization>
    </security>
    <local-cache name="secured">
      <security>
        <authorization roles="admin reader writer supervisor" />
      </security>
    </local-cache>
  </cache-container>

</infinispan>
```

23.1.3.1. ロールマッパー

Subject の Principal を、承認時に使用するロールのセットに変換するには、適切な PrincipalRoleMapper をグローバル設定に指定する必要があります。Red Hat Data Grid には 3 つのマッパーが含まれており、カスタムマッパーを指定することもできます。

-

IdentityRoleMapper(Java: org.infinispan.security.impl.IdentityRoleMapper, XML: <identity-role-mapper />): このマッパーは Principal name をロール名として使用します。

- *CommonNameRoleMapper(Java: org.infinispan.security.impl.CommonRoleMapper, XML: <common-name-role-mapper />): Principal name が識別名(DN)の場合、このマッパーは Common Name(CN)を抽出し、ロール名として使用します。たとえば、DN cn=managers,ou=people,dc=example,dc=com はロールマネージャーにマッピングされます。*
- *ClusterRoleMapper(Java: org.infinispan.security.impl.ClusterRoleMapper XML: <cluster-role-mapper />): ClusterRegistry を使用してプリンシパルをロールマッピングに保存するマッパー。これにより、CLI の GRANT および DENY コマンドを使用して、ロールをプリンシパルに追加/削除することができます。*
- *カスタムロールマッパー(XML: <custom-role-mapper class="a.b.c" />): org.infinispan.security.PrincipalRoleMapper の実装の完全修飾クラス名を指定するだけです。*

23.2. クラスターセキュリティー

JGroups は、ノードが結合/マージに参加するときに相互に認証する必要があるよう設定できます。認証は SASL を使用し、以下のように GMS プロトコルの上の JGroups XML 設定に SASL プロトコルを追加することで設定されます。

```
<SASL mech="DIGEST-MD5"
   client_name="node_user"
   client_password="node_password"

server_callback_handler_class="org.example.infinispan.security.JGroupsSaslServerCallbackHandler"

client_callback_handler_class="org.example.infinispan.security.JGroupsSaslClientCallbackHandler"
   sasl_props="com.sun.security.sasl.digest.realm=test_realm" />
```

上記の例では、SASL mech は DIGEST-MD5 になります。各ノードは、クラスターに参加する際に使用するユーザーとパスワードを宣言する必要があります。ノードの動作は、コーディネーターか他のノードかによって異なります。コーディネーターは SASL サーバーとして機能しますが、ノードが参加/以前のノードは SASL クライアントとして機能します。そのため、2 つの異なる CallbackHandler が必要です。server_callback_handler_class はコーディネーターによって使用され、client_callback_handler_class は他のノードによって使用されます。JGroups の SASL プロトコルは認証プロセスにのみ関係します。ノードの承認を実装する場合は、例外をスローしてサーバーコールバックハンドラー内で実行できます。以下の例は、これがどのように実行されるかを示しています。

```
public class AuthorizingServerCallbackHandler implements CallbackHandler {
```

```
    @Override
    public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            ...
            if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback acb = (AuthorizeCallback) callback;
                UserProfile user = UserManager.loadUser(acb.getAuthenticationID());
                if (!user.hasRole("myclusterrole")) {
                    throw new SecurityException("Unauthorized node " +user);
                }
            }
            ...
        }
    }
}
```

第24章 グリッドファイルシステム

Red Hat Data Grid の GridFileSystem は、ファイルシステムとして Red Hat Data Grid でバックアップする Data Grid を公開する実験的な API です。

警告

これは実験的な API です。自己責任でお使いください。

具体的には、API は JDK の File、InputStream クラス、および OutputStream クラス（具体的には Grid File 、 Grid InputStream 、および Grid OutputStream ）へのエクステンションとして機能します。ヘルパークラス GridFilesystem も含まれています。

基本的に、GridFilesystem は 2 つの Red Hat Data Grid キャッシュでサポートされます。1 つはメタデータ（通常は複製）、もう 1 つは実際のデータ（通常は分散）に対応します。前者は複製されるので、各ノードにメタデータ情報がローカルにあり、ファイルを一覧表示するための RPC 呼び出しは必要ありません。後者は、ストレージ領域の大部分が使用される場所であり、ここでスケーラブルなメカニズムが必要であるため、配布されます。ファイル自体はチャンクであり、各チャンクはバイトアレイとしてキャッシュエントリーとして保存されます。

以下は、使用法を示す簡単なコードスニペットです。

```
Cache<String,byte[]> data = cacheManager.getCache("distributed");
Cache<String,GridFile.Metadata> metadata = cacheManager.getCache("replicated");
GridFilesystem fs = new GridFilesystem(data, metadata);

// Create directories
File file=fs.getFile("/tmp/testfile/stuff");
fs.mkdirs(); // creates directories /tmp/testfile/stuff

// List all files and directories under "/usr/local"
file=fs.getFile("/usr/local");
File[] files=file.listFiles();

// Create a new file
file=fs.getFile("/tmp/testfile/stuff/README.txt");
file.createNewFile();
```

stuff をグリッドファイルシステムにコピーします。

```
InputStream in=new FileInputStream("/tmp/my-movies/dvd-image.iso");
OutputStream out=fs.getOutput("/grid-movies/dvd-image.iso");
byte[] buffer=new byte[20000];
int len;
while((len=in.read(buffer, 0, buffer.length)) != -1) out.write(buffer, 0, len);
in.close();
out.close();
```

グリッドから作業内容を読み込んでいる：

```
InputStream in=in.getInput("/grid-movies/dvd-image.iso");
OutputStream out=new FileOutputStream("/tmp/my-movies/dvd-image.iso");
byte[] buffer=new byte[200000];
int len;
while((len=in.read(buffer, 0, buffer.length)) != -1) out.write(buffer, 0, len);
in.close();
out.close();
```

24.1. WEBDAV デモ

Red Hat Data Grid には、グリッドファイルシステム API を使用するデモ WebDAV アプリケーションが同梱されています。このデモアプリケーションは、JBoss AS や Tomcat などのサーブレットコンテナーにデプロイできる WAR ファイルとしてパッケージされ、グリッドを WebDAV を介してファイルシステムとして公開します。これにより、オペレーティングシステムのリモートドライブとしてマウントされる可能性があります。

第25章 サイト間のレプリケーション

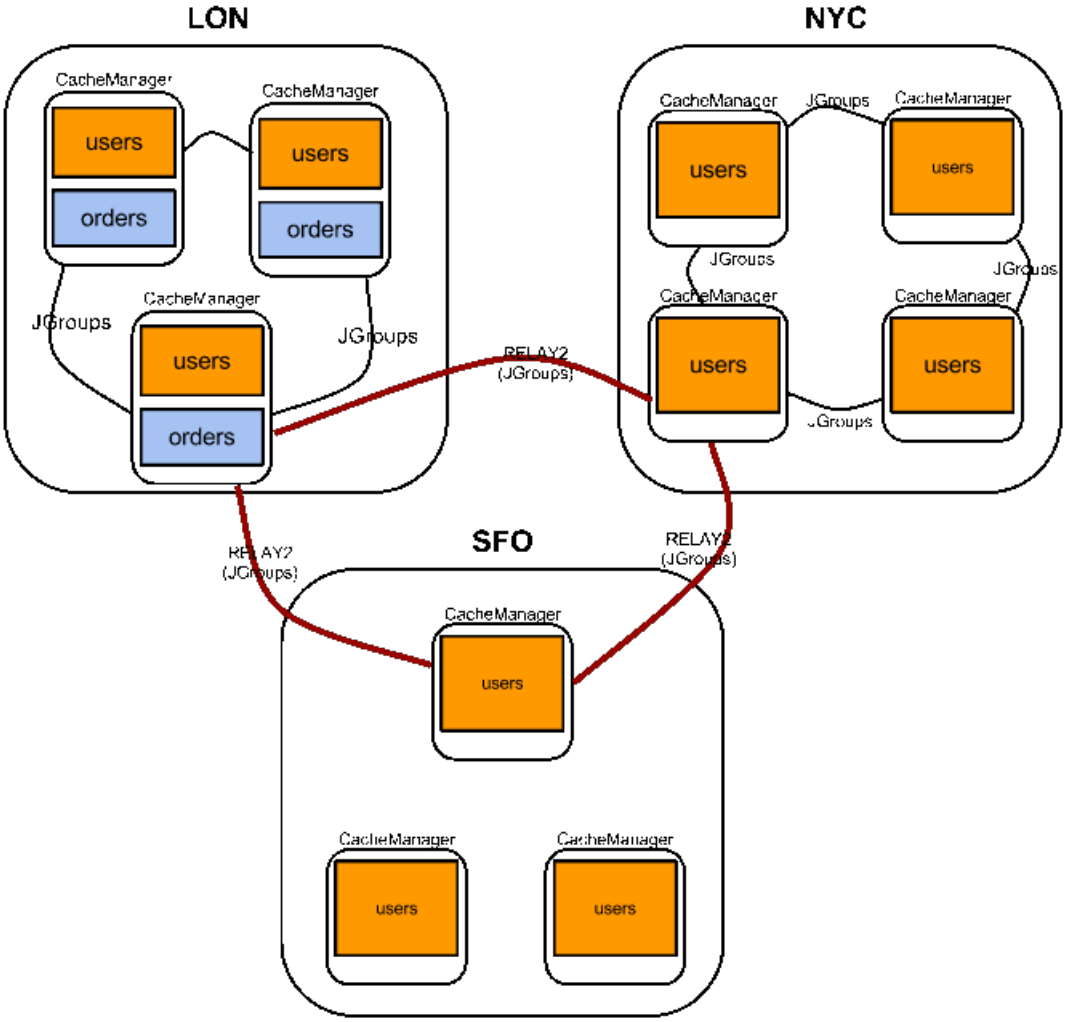
クロスサイト(x-site)レプリケーションにより、あるクラスターから他のクラスターにデータをバックアップできます。これは、地理的に異なる場所にある可能性があります。クロスサイトレプリケーションは JGroups の RELAY2 プロトコル 上に構築されます。本ガイド では、クロスサイトレプリケーションの技術的な設計について説明します。

注記

サイト間のレプリケーションには、サイトマスターノード（バックアップを受信し、これを適用するノードなど）で実行しているバックアップキャッシュが必要です。バックアップキャッシュは、最初のバックアップ要求を受け取ると自動的に起動します。

25.1. デプロイメント例

以下の図は、複製されたサイトが可能なセットアップと、デプロイメントに含まれる個々の要素の説明を示しています。その後、オプションは今後の段落で広範囲に説明されています。上記の図のコメント：



- *LON、NYC、および SFO の 3 つのサイトがあります。*
- *各サイトには、Red Hat Data Grid クラスターには（通常）異なる数の物理ノード（LON の 3 ノード、SFO の NYC ノードおよび 3 つのノード）が実行されている Red Hat Data Grid クラスターがあります。*
- *「users」キャッシュは、LON、NYC、および SFO でアクティブになっています。これらのサイトの「users」キャッシュの更新は、他のサイトにも複製されます。*
- *サイト間で異なるレプリケーションメカニズムを使用できます。例:1 つは、SFO がデータを同期的に NYC にバックアップし、非同期的に LON に設定できます。*
- *「ユーザー」キャッシュは、あるサイトから別のサイトに異なる設定を持つことができます。たとえば、LON サイトの numOwners=2 で配布され、NYC サイトの REPL、SFO サイトの numOwners=1 で配布されます。*

- JGroups はサイト間通信とサイト内通信の両方に使用されます。RELAY2 がサイト間の通信に使用されます。

- 「順序」は、LON のローカルサイトです。つまり、「orders」のデータの更新は、リモートサイトに複製されません。以下のセクションでは、サイト間のレプリケーションの特定の側面について詳細に説明しています。クロスサイトレプリケーション機能の基盤は RELAY2 であるため、クロスサイトに移動する前に JGroups の RELAY2 ドキュメントを読むことを強く推奨します。設定

クロスサイトのレプリケーション設定は、以下のファイルを分散します。

1. 各キャッシュのバックアップポリシーは、Red Hat Data Grid .xml 設定ファイル (infinispan.xml)で定義されます。

2. クラスターの JGroups xml 設定ファイル： RELAY2 プロトコルを JGroups プロトコルスタック(jgroups.xml)に追加する必要があります。

3. RELAY2 設定ファイル： RELAY2 には独自の設定ファイル(relay2.xml)があります。

4. RELAY2 によって使用される JGroups チャネルには、独自の設定ファイル(jgroups-relay2.xml)の Red Hat Data Grid XML 設定ファイルがあります。

ローカルサイトはグローバル設定セクションで定義されます。local は、この設定ファイルを使用するノードがあるサイトです（上記の例では、ローカルサイトは「LON」です）。

infinispan.xml

```
<transport site="LON" />
```

同じ設定をプログラムで実行できます。

```
GlobalConfigurationBuilder lonGc = GlobalConfigurationBuilder.defaultClusteredBuilder();
lonGc.site().localSite("LON");
```

サイトの名前（大文字）は、JGroups の RELAY2 プロトコル設定ファイル内で定義されたサイトの名前と一致する必要があります。各キャッシュはグローバル設定のほかに、「site」要素にそのバックアップポリシーを指定します。

infinispan.xml

```
<distributed-cache name="users">
  <backups>
    <backup site="NYC" failure-policy="WARN" strategy="SYNC" timeout="12000"/>
    <backup site="SFO" failure-policy="IGNORE" strategy="ASYNC"/>
    <backup site="LON" strategy="SYNC" enabled="false"/>
  </backups>
</distributed-cache>
```

「users」キャッシュは、そのデータを「NYC」および「SFO」サイトにバックアップします。「LON」がバックアップサイトとして表示されますが、「enabled」属性が false に設定されているため、無視されます。サイトのバックアップごとに、以下の設定属性を指定できます。

- strategy - データのバックアップに使用されるストラテジー（「SYNC」または「ASYNC」）。デフォルトは「ASYNC」です。
- failure-policy: バックアップ中に失敗した場合にシステムが何を行うかを決定します。以下の値が使用できます。
  - IGNORE - ローカル操作/トランザクションを成功させる
  - WARN - IGNORE と同じですが、警告メッセージをログに記録します。デフォルトです。
  - FAIL - "strategy" が "SYNC" の場合のみ有効で、ユーザーに例外をスローしてローカルクラスター操作/トランザクションが失敗します。

- CUSTOM: ユーザー提供。以下の「failurePolicyClass」を参照してください。

- failurePolicyClass: 'failure-policy' が 'CUSTOM' に設定されている場合、この属性は必須です。org.infinispan.xsite.CustomFailurePolicy を実装するクラスの完全修飾名が含まれる必要があります。

- timeout: データをリモートでバックアップする際に使用するタイムアウト（ミリ秒単位）。デフォルトは 10000（10 秒）です。

同じ設定をプログラムで実行できます。

```
ConfigurationBuilder lon = new ConfigurationBuilder();
lon.sites().addBackup()
    .site("NYC")
    .backupFailurePolicy(BackupFailurePolicy.WARN)
    .strategy(BackupConfiguration.BackupStrategy.SYNC)
    .replicationTimeout(12000)
    .sites().addInUseBackupSite("NYC")
  .sites().addBackup()
    .site("SFO")
    .backupFailurePolicy(BackupFailurePolicy.IGNORE)
    .strategy(BackupConfiguration.BackupStrategy.ASYNC)
    .sites().addInUseBackupSite("SFO")
```

上記の「ユーザー」キャッシュは、データがレプリケートされているリモートサイト上のキャッシュを認識しません。デフォルトでは、リモートサイトはバックアップデータを起点と同じ名前を持つキャッシュに書き込みます（つまり「users」）。この動作は、「backupFor」要素で上書きできます。たとえば、SFO で以下の設定により、「usersLONBackup」キャッシュが LON サイトにある「users」キャッシュのバックアップキャッシュとして機能します。

infinispan.xml

```
<infinispan>
  <cache-container default-cache="">
    <distributed-cache name="usersLONBackup">
      <backup-for remote-cache="users" remote-site="LON"/>
    </distributed-cache>
  </cache-container>
</infinispan>
```

同じ設定をプログラムで実行できます。

```
ConfigurationBuilder cb = new ConfigurationBuilder();
cb.sites().backupFor().remoteCache("users").remoteSite("LON");
```

25.1.1. ローカルクラスターの jgroups .xml 設定

これは、ローカル（オンサイト）の Red Hat Data Grid クラスターの設定ファイルです。これは、Red Hat Data Grid 設定ファイルから参照されています。以下の「configurationFile」を参照してください。

infinispan.xml

```
<infinispan>
  <jgroups>
    <stack-file name="external-file" path="jgroups.xml"/>
  </jgroups>
  <cache-container>
    <transport stack="external-file" />
  </cache-container>

  ...

</infinispan>
```

サイト間の呼び出しを許可するには、RELAY2 プロトコルを jgroups 設定で定義されたプロトコルスタックに追加する必要があります（サンプルの接続済み jgroups.xml を参照してください）。

25.1.2. RELAY2 設定ファイル

RELAY2 設定ファイルは jgroups.xml からリンクされます（割り当てられる relay2.xml を参照）。リモートサイトと通信するために、このクラスターおよび RELAY2 によって使用される JGroups 設定ファイルも定義します。

25.2. データレプリケーション

トランザクションキャッシュと非トランザクションキャッシュの両方の場合、バックアップ呼び出しは、ローカルクラスター呼び出しと並行して実行されます。たとえば、LON のノード N1 にデータを書

き込む場合、ローカルノードに N2 および N3 とリモートバックアップサイト SFO と NYC が並行してレプリケーションされます。

25.2.1. トランザクション以外のキャッシュ

トランザクション以外のキャッシュの場合は、各操作中にレプリケーションが発生します。データが並行してバックアップおよびローカルキャッシュに送信される場合、操作がローカルで成功し、リモートで失敗したり、他の方法でも行わず、不整合の原因となることがあります。

25.2.2. トランザクションキャッシュ

同期トランザクションキャッシュの場合、Red Hat Data Grid は内部的に 2 フェーズコミットプロトコルを使用します。非同期キャッシュの場合、2 つのフェーズがマージされます。「変更の適用」メッセージはデータのオーナーに非同期に送信されます。この 2PC プロトコルは、JTA トランザクションマネージャーから受け取った 2PC にマッピングします。トランザクションキャッシュの場合は、optistic および pessimistic の両方は、リモートサイトへのバックアップは準備およびコミットフェーズにのみ行われます。

25.2.2.1. 非同期バックアップを使用したローカルクラスターの同期

このシナリオでは、バックアップ呼び出しはローカルコミットフェーズ（2 番目のフェーズ）で実行されます。つまり、ローカルの準備に失敗した場合は、リモートバックアップにはリモートデータが送信されることはありません。

25.2.2.2. 同期バックアップを使用したローカルクラスターの同期

この場合、以下の 2 つのバックアップコールがあります。

- 準備中に、このトランザクション内で発生したすべての変更が含まれるメッセージが送信されます。
- リモートバックアップキャッシュがトランザクションの場合、トランザクションはリモートで起動し、これらの変更はすべてこのトランザクションのスコープ内に書き出されます。トランザクションがまだコミットされていません（以下を参照）。
- リモートバックアップキャッシュがトランザクションではない場合は、変更はリモートに適用されます。

- コミット/ロールバック時に、コミット/ロールバックメッセージが以下に送信されます。

- リモートバックアップキャッシュがトランザクションの場合、前のフェーズで開始したトランザクションはコミットまたはロールバックされます。

- リモートバックアップがトランザクションではない場合は、この呼び出しは無視されます。

ローカルの呼び出しとバックアップコール（「backupFailurePolicy」が「FAIL」）に設定すると、トランザクションの準備結果が得られます。

25.2.2.3. 非同期ローカルクラスター

非同期ローカルクラスターの場合には、コミットフェーズでバックアップデータが送信されます。バックアップ呼び出しに失敗し、「backupFailurePolicy」が「FAIL」に設定されていると、ユーザーは例外を介して通知されます。

25.2.3. サイト間での期限切れのキャッシュエントリーのレプリケーション

Red Hat Data Grid は、エントリーの有効期限が切れるように設定し、エントリーがキャッシュに残る時間を制御できます。

- ライフスパン の有効期限は、Red Hat Data Grid が他のサイトのクラスターでエントリーが期限切れになるかどうかを判断する必要のないエントリーの有効期限が切れるため、クロスサイトのレプリケーションに適しています。

- Red Hat Data Grid は、キャッシュエントリーが他のサイトのクラスターで アイドル タイムアウトに達したかどうかを判断することができないため、Maxidle の有効期限は、クロスサイトレプリケーションでは機能しません。

25.2.4. デッドロックおよび競合エントリー

アクティブ/アクティブ設定では、同じキーへの同時書き込みによりデッドロックや競合が生じます。

アクティブ/アクティブの設定は、ローカルサイトとバックアップサイトの両方がデータを相互に複

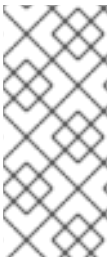
製するものです。たとえば、「SiteA」は「SiteB」までバックアップされ、「SiteB」も「SiteA」までバックアップされます。

同期モード(SYNC)では、両方のサイトが同じキーを異なる順序でロックするため、同時書き込みはデッドロックになりました。デッドロックを解決するには、クライアントアプリケーションはロックがタイムアウトするまで待つ必要があります。

非同期モード(ASYNC)では、エントリーがローカルの変更後にサイトが複製されるため、同時書き込みは競合する値になります。非同期レプリケーション操作には保証された順序がないため、以下の結果が可能になります。

- キーの値は、各サイトに異なります。たとえば、「SiteA」の場合は k=1、"SiteB" の場合は k=2 です。非同期レプリケーションは、クライアントが k への書き込み後に x=2 が「SiteA」に複製され、x= 1 が「SiteB」に複製された後に発生します。
- サイトが同時に値を複製しない場合、競合する値の 1 つが上書きされます。この場合、いずれかの値が失われ、どの値が保存されるかは保証されません。

いずれの場合も、キー値の不整合は、次の競合しない PUT 操作で値を更新すると解決されます。



注記

現在、クライアントアプリケーションが非同期モードで競合を処理するのに使用できる競合解決ポリシーはありません。ただし、競合解決技術は、今後の Red Hat Data Grid バージョンに対して予定されています。

25.3. サイトのオフラインにする

期間の特定の回数にサイトへのバックアップに失敗した場合は、そのサイトをオフラインとして自動的にマークできます。サイトがオフラインとマークされている場合、ローカルサイトはデータのバックアップを試行しなくなります。オンラインに行うには、システム管理者の介入が必要になります。

25.3.1. 設定

以下の設定では、サイトをオフラインにする例を示します。

infinispan.xml

```
<replicated-cache name="bestEffortBackup">
  ...
  <backups>
    <backup site="NYC" strategy="SYNC" failure-policy="FAIL">
       <take-offline after-failures="500" min-wait="10000"/>
    </backup>
  </backups>
   ...
</replicated-cache>
```

バックアップ 下の take-offline 要素は、サイトの使用をオフラインにします。

- after-failures: このサイトがオフラインにされるべきバックアップ操作の数。デフォルトは 0(never)です。負の値は、minTimeToWaitの後にサイトがオフラインになることを意味します。

- min-wait: 'afterFailures' の回数に達していない場合でもサイトがオフラインとマークされない期間（ミリ秒単位）。0 より小さいか等しい場合は、afterFailures のみとみなされます。

同等のプログラムによる設定は以下のとおりです。

```
lon.sites().addBackup()
    .site("NYC")
    .backupFailurePolicy(BackupFailurePolicy.FAIL)
    .strategy(BackupConfiguration.BackupStrategy.SYNC)
    .takeOffline()
      .afterFailures(500)
      .minTimeToWait(10000);
```

25.3.2. サイトのバックオンライン化

サイトをオフラインにしたら、以下の JMX MBean から bringSiteOnline(siteName) 操作を呼び出して、サイトをオンラインに戻します。

- XSiteAdmin は、リモートサイトのクラスター全体でのキャッシュのレプリケーションを有効にします。

- *GlobalXSiteAdminOperations は、リモートサイトのクラスター全体でのキャッシュコンテナーのレプリケーションを有効にします。*

bringSiteOnline(siteName) 操作はレプリケーションのみを有効にし、完全な更新は行いません。このため、サイトをオンラインに戻すと、新しいエントリーのみが含まれます。直近のデータを同期するためにオンラインに切り替えた後に、サイトへの譲渡をプッシュする必要があります。

ヒント

状態遷移をプッシュすると、サイトがオンラインに戻ります。これには、bringSiteOnline(siteName) を呼び出す代わりに実行できます。

25.4. サイトへの状態の転送のプッシュ

あるサイトから別のサイトに状態を転送すると、2 つのサイト間でデータが同期されます。

現在アクティブなサイト（最新のデータを含む）から別のサイトに状態を常に転送する必要があります。状態遷移をプッシュするサイトは、オンラインまたはオフラインにすることができます。オフラインサイトに状態遷移をプッシュすると、サイトがオンラインに戻ります。

別のサイトから状態を転送するサイト上のキャッシュは、空にする必要はありません。ただし、状態遷移は受信サイトにある既存の鍵のみを上書きし、キーを削除しません。

たとえば、キー K はサイト A およびサイト B に存在します。サイト A からサイト B に状態を移行します。今回の例では、Red Hat Data Grid はキー K を上書きします。ただし、キー Y はサイト B に存在しますが、サイト A からサイト B に状態を移動した後でもキー Y はサイト B に存在します。

注記

一度に 1 つのサイトからのみ状態遷移を受け取ることができます。同様に、サイトで状態遷移操作を呼び出すと、Red Hat Data Grid はそのサイトの後続の呼び出しを無視します。

以下の JMX MBean から pushState(String) 操作を呼び出して、サイトをオンラインに戻します。

-

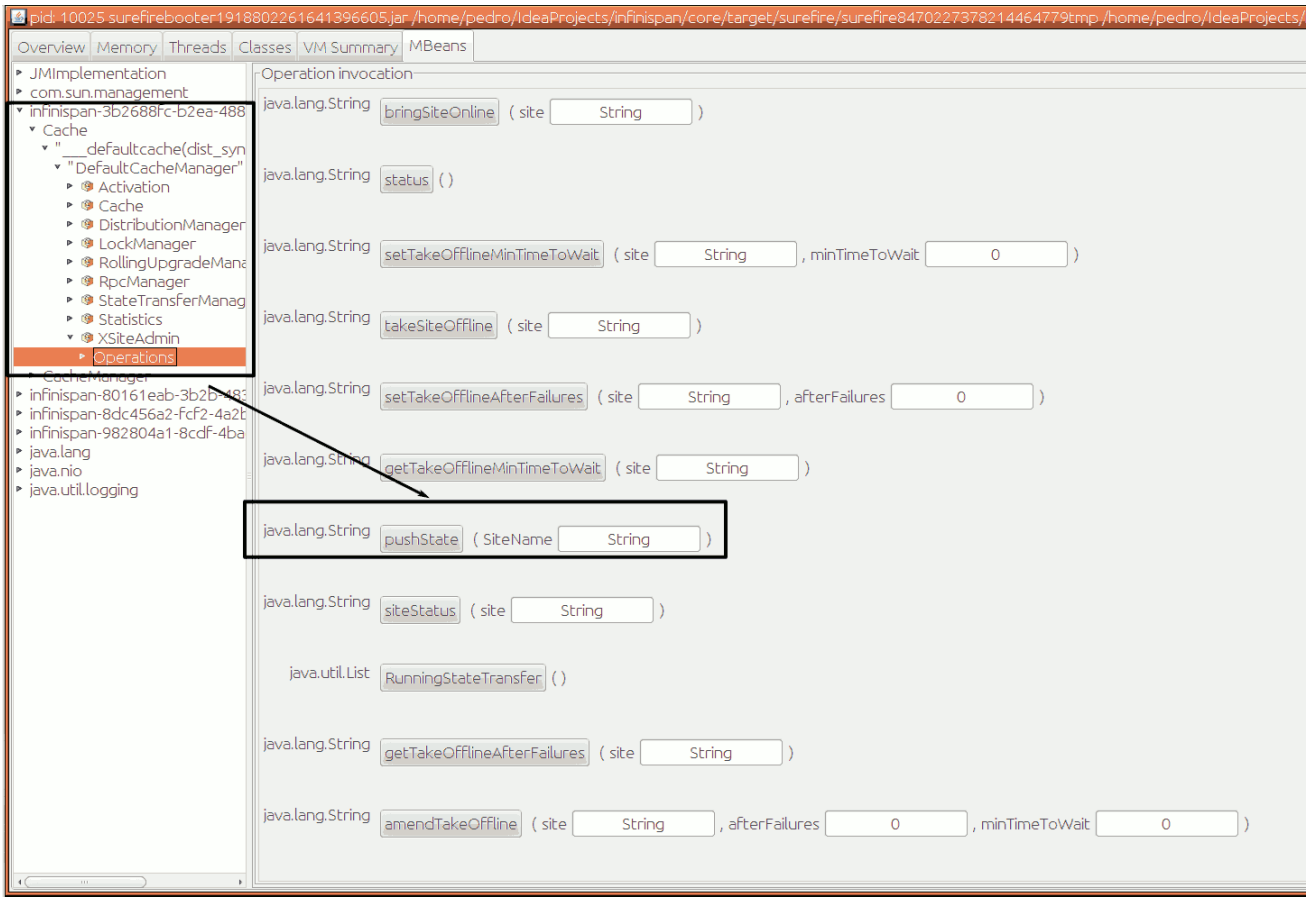
XSiteAdminOperations は、操作を呼び出すサイトとリモートサイト間でキャッシュの状態を同期します。オフラインの場合はそのサイトをオンラインにします。

- *GlobalXSiteAdminOperations は、操作を呼び出すサイトとリモートサイト間でキャッシュコンテナーの状態を同期します。オフラインの場合はそのサイトをオンラインにします。*

転送状態のサイトで pushState(String) 操作を呼び出すと、状態を受け取るサイトの名前を指定します。

以下の図は、JConsole の pushState(String)操作を示しています。

図25.1 JConsole による状態のプッシュ



25.4.1. 参加ノードの処理/ノードの処理

現在の実装は、プロデューサーまたはコンシューマーサイトのトポロジーの変更を自動的に処理します。また、クロスサイトの状態遷移は、ローカルサイトの状態遷移と並行して実行できます。

25.4.2. サイト間の破損したリンクの処理

プロデューサーとコンシューマーサイト間のリンクがサイト間の状態遷移中に破損している場合には、システム管理者のアクションが必要です（コンシューマーサイトではデータの一貫性は確保されません）。プロデューサーサイトは、停止する前にしばらく再試行します。次に、通常の状態に戻ります。ただし、コンシューマーサイトは通常の状態に戻ることができません。ここでは、システム管理者からのアクションが必要です。システム管理者は、操作 cancelReceiveState(String siteName)を使用してコンシューマーサイトを通常の状態にする必要があります。

25.4.3. システム管理者の操作

一連の操作を実行して、サイト間状態遷移を制御できます。

- pushstate(String siteName)- 指定したサイト名へのクロスサイト状態遷移を開始します。
- cancelPushState(String siteName)- 指定したサイト名へのクロスサイト状態遷移をキャンセルします。
- getRunningStateTransfer（）：このサイトが状態をプッシュするサイト名のリストを返します。
- getSendingSiteName（）- このサイトに状態をプッシュするサイト名を返します。
- cancelReceiveState(String siteName): サイトを通常の状態に復元します。状態遷移中にサイト間のリンクが壊れている場合に使用してください（上記で説明）。
- getPushStateStatus（）- 完了したクロスサイトの状態遷移のステータスを返します。
- clearPushStateStatus（）- 完了したクロスサイトの状態遷移のステータスを消去します。

技術的な情報については、「Cross Site design」を参照してください。「参考情報」を参照 してください。

25.4.4. 設定

サイト間の状態遷移は有効または無効にすることはできませんが、一部のパラメーターの調整が可能です。以下の値はデフォルト値です。

infinispan.xml

```
<replicated-cache name="xSiteStateTransfer">
  ...
  <backups>
    <backup site="NYC" strategy="SYNC" failure-policy="FAIL">
      <state-transfer chunk-size="512" timeout="1200000" max-retries="30" wait-time="2000" />
    </backup>
  </backups>
   ...
</replicated-cache>
```

同等のプログラムによる設定は以下のとおりです。

```
lon.sites().addBackup()
    .site("NYC")
    .backupFailurePolicy(BackupFailurePolicy.FAIL)
    .strategy(BackupConfiguration.BackupStrategy.SYNC)
    .stateTransfer()
      .chunkSize(512)
      .timeout(1200000)
      .maxRetries(30)
      .waitingTimeBetweenRetries(2000);
```

以下は、パラメーターの説明です。

- chunk-size - バッチ処理をコンシューマーサイトに送信する前にバッチ処理するキーの数。負の値またはゼロの値は有効な値では ありません。デフォルト値は 512 キーです。
- タイムアウト: コンシューマーサイトが、状態チャンクの受信およびアプライアンスを認識するのを待つ時間（ミリ秒単位）。負の値またはゼロの値は有効な値では ありません。デフォルト値は 20 分です。
- max-retries: push state コマンドが失敗した場合の再試行の最大数。負の値またはゼロの値は、失敗した場合にコマンドが再試行されないことを意味します。デフォルト値は 30 です。
-

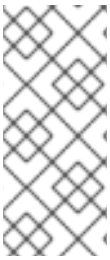
wait-time: 各再試行間の待ち時間（ミリ秒単位）。負の値またはゼロの値は有効な値ではありません。デフォルト値は 2 秒です。

25.5. 参照

本ガイド では、クロスサイトレプリケーションの技術的な設計について説明します。

第26章 ローリングアップグレードの実行

ダウンタイムやデータ損失なしで Red Hat Data Grid をアップグレードします。Remote Client/Server Mode でローリングアップグレードを実行して、より新しいバージョンの Red Hat Data Grid の使用を開始することができます。



注記

本セクションでは、Red Hat Data Grid サーバーをアップグレードする方法を説明します。アップグレード手順については、Hotset クライアントに適したドキュメントを参照してください。

ローリングアップグレードを実行するには、ハイレベルから以下を行います。

1. ターゲットクラスターを設定します。ターゲットクラスターは、データを移行する Red Hat Data Grid バージョンです。ソースクラスターは、現在使用中の Red Hat Data Grid デプロイメントです。ターゲットクラスターの実行後に、すべてのクライアントがソースクラスターの代わりにそのクライアントを参照するように設定します。

2. ソースクラスターからターゲットクラスターにデータを同期します。

26.1. ターゲットクラスターの設定

1. 一意のネットワークプロパティーまたは異なる JGroups クラスター名を使用してターゲットクラスターを起動し、ソースクラスターから分離します。

2. ソースクラスターから移行する各キャッシュに対して、ターゲットクラスターで RemoteCacheStore を設定します。

RemoteCacheStore の設定

- remote-server は、outbound-socket-binding プロパティーを使用してソースクラスターを参照する必要があります。
- remoteCacheName はソースクラスターのキャッシュ名と一致する必要があります。

- *hotrod-wrapping は true （有効）である必要があります。*
- *共有 は true （有効）である必要があります。*
- *パージ は false （無効）である必要があります。*
- *パッシベーション は false （無効）である必要があります。*
- *protocol-version は、ソースクラスターの HotTEMPLATES プロトコルバージョンと一致します。*

RemoteCacheStore の設定例

```
<distributed-cache>
   <remote-store cache="MyCache" socket-timeout="60000" tcp-no-delay="true"
protocol-version="2.5" shared="true" hotrod-wrapping="true" purge="false"
passivation="false">
      <remote-server outbound-socket-binding="remote-store-hotrod-server"/>
   </remote-store>
</distributed-cache>
...
<socket-binding-group name="standard-sockets" default-interface="public" port-
offset="${jboss.socket.binding.port-offset:0}">
  ...
  <outbound-socket-binding name="remote-store-hotrod-server">
     <remote-destination host="198.51.100.0" port="11222"/>
  </outbound-socket-binding>
  ...
</socket-binding-group>
```

3. *ソースクラスターではなくすべてのクライアント要求を処理するようにターゲットクラスターを設定します。*

   a. *すべてのクライアントがソースクラスターではなくターゲットクラスターを参照するように設定します。*

b.
各クライアントノードを再起動します。

ターゲットクラスターは、RemoteCacheStore を介してオンデマンドでソースクラスターからデータをロードします。

26.2. ソースクラスターからのデータの同期

1.
TargetMigrator インターフェースで synchronizeData（） メソッドを呼び出します。移行する各キャッシュのターゲットクラスターで、以下のいずれかを行います。

JMX

synchronizeData 操作を呼び出して、RollingUpgradeManager MBean で hotrod パラメーターを指定します。

CLI

```
$ bin/cli.sh --connect controller=127.0.0.1:9990 -c "/subsystem=datagrid-infinispan/cache-container=clustered/distributed-cache=MyCache:synchronize-data(migrator-name=hotrod)"
```

データは、ターゲットクラスターのすべてのノードに並行して移行され、各ノードはデータのサブセットを受け取ります。

以下のパラメーターを使用して操作を調整します。

- read-batch は、一度にソースクラスターから読み取るエントリーの数を設定します。デフォルト値は 10000 です。
- write-threads はデータの書き込みに使用されるスレッド数を設定します。デフォルト値は、利用可能なプロセッサーの数です。

以下に例を示します。

```
synchronize-data(migrator-name=hotrod, read-batch=100000, write-threads=3)
```

2. ターゲットクラスターの RemoteCacheStore を無効にします。次のいずれかを行います。

JMX

disconnectSource 操作を呼び出して、RollingUpgradeManager MBean で hotrod パラメーターを指定します。

CLI

```
$ bin/cli.sh --connect controller=127.0.0.1:9990 -c "/subsystem=datagrid-infinispan/cache-container=clustered/distributed-cache=MyCache:disconnect-source(migrator-name=hotrod)"
```

3. ソースクラスターの使用を停止します。== Red Hat Data Grid Red Hat Data Grid を拡張することで、エンドユーザーが Red Hat Data Grid が通常提供する設定、操作、およびコンポーネント以外の機能を追加できます。

26.3. カスタムコマンド

Red Hat Data Grid では、コマンド/アドバイザーパターンを使用し て、公開されている API で表示されるさまざまな最上位メソッドを実装します。詳細は、「仮想化 の概要」セクションを参照 してください。コアコマンド およびその対応するブリックは、Red Hat Data Grid のコアモジュールの一部としてハードコーディングされますが、新しいカスタムコマンドを作成することで、モジュール作成者は Red Hat Data Grid を拡張および強化できます。

モジュール作成者（ infinispan-query など）として、独自のコマンドを定義できます。

これは、以下の方法で行います。

1. META-INF/services/org.infinispan.commands.module.ModuleCommandExtensions ファイルを作成し、これが jar にパッケージ化されていることを確認します。

2. ModuleCommandFactory、ModuleCommandInitializer、および ModuleCommandExtensionsの実装

3. META-INF/services/org.infinispan.commands.module. ModuleCommandExtensions で ModuleCommandExtensions 実装の完全修飾クラス名の指定。

4. これらのコマンドにカスタムコマンドと方法を実装する

26.3.1. 例

以下は META-INF/services/org.infinispan.commands.module.ModuleCommandExtensions ファイルの例です。

org.infinispan.commands.module.ModuleCommandExtensions

```
org.infinispan.query.QueryModuleCommandExtensions
```

カスタムコマンドやブリックを利用するサンプルモジュールの作業例は、「Red Hat Data Grid サンプルモジュール」を参照してください。

26.3.2. 事前に割り当てられたカスタムコマンド ID 範囲

これは、Red Hat Data Grid ベースのモジュールまたはフレームワークが使用する Command 識別子の一覧です。Red Hat Data Grid ユーザーは、これらの範囲内の ID を使用しないようにする必要があります。（RANGES がまだ最終となる！） これは単一バイトであるため、範囲は大きくすることはできません。

Red Hat Data Grid Query:	100 - 119
Hibernate Search:	120 - 139
hotgitops Server:	140 - 141

26.4. 設定ビルダーおよびパーサーの拡張

カスタムモジュールの設定が必要な場合は、Red Hat Data Grid の設定ビルダーおよびパーサーを強化できます。カスタムモジュールテスト でこれを実装する方法の詳細例を確認してください。

第27章 アーキテクチャーの概要

このセクションでは、Red Hat Data Grid の内部アーキテクチャーの概要について説明します。本書は、Red Hat Data Grid の拡張または強化、または Red Hat Data Grid の内部に興味のある人を対象としています。

27.1. キャッシュ階層

Red Hat Data Grid の Cache インターフェースは、理解しやすい API を提供する JRE の ConcurrentMap インターフェースを拡張します。

```
---
public interface Cache<K, V> extends BasicCache<K, V> {
 ...
}
```

public interface BasicCache<K, V> extends ConcurrentMap<K, V> { … } ---

キャッシュは、CacheContainer インスタンス（EmbeddedCacheManager または RemoteCacheManager のいずれか）を使用して作成されます。CacheContainers は、キャッシュのファクトリーとして機能する機能に加え、キャッシュを検索するレジストリーとしても機能します。

EmbeddedCacheManagers は、同じ JVM にあるクラスター化されたキャッシュまたはスタンドアロンキャッシュのいずれかを作成します。一方、RemoteCacheManagers は、Hot Warehouse プロトコルを介してリモートキャッシュ層に接続する RemoteCaches を作成します。

27.2. コマンド

内部的には、各キャッシュ操作およびすべてのキャッシュ操作はコマンドでカプセル化されます。これらのコマンドオブジェクトは実行される操作のタイプを表し、さらに必要なパラメーターへの参照を保持します。ReplaceCommand などの特定のコマンドの実際のロジックは、コマンドの perform（）メソッドでカプセル化されます。オブジェクト指向で簡単にテストできます。

これらのコマンドはすべて VisitableCommand inte エリアを実装し、それに応じて Visitor（次のセクションで説明）を許可します。

```
---
public class PutKeyValueCommand extends VisitableCommand {
  ...
}
```

- *public class GetKeyValueCommand extends VisitableCommand { ... }*

    i. *etc ... ---*

27.3. PIDGINS

コマンドは、さまざまな Visitors で処理されます。以下に表示されているこれらのインターフェースは、システム内のさまざまなタイプのコマンドにアクセスするためのメソッドを公開します。これにより、呼び出しに動作を追加するタイプセーフメカニズムを利用できます。コマンドは 'Visitor's によって処理されます。以下に表示されているこれらのインターフェースは、システム内のさまざまなタイプのコマンドにアクセスするためのメソッドを公開します。これにより、呼び出しに動作を追加するためのタイプセーフメカニズムが提供されます。

```
---
public interface Vistor {
  Object visitPutKeyValueCommand(InvocationContext ctx, PutKeyValueCommand
command) throws Throwable;

Object visitRemoveCommand(InvocationContext ctx, RemoveCommand command) throws
Throwable;

Object visitReplaceCommand(InvocationContext ctx, ReplaceCommand command) throws
Throwable;

Object visitClearCommand(InvocationContext ctx, ClearCommand command) throws
Throwable;

Object visitPutMapCommand(InvocationContext ctx, PutMapCommand command) throws
Throwable;
```

    i. *etc ... } ---*

org.infinispan.commands パッケージの AbstractVisitor クラスは、これらの各メソッドの no-op 実装で提供されます。その後、実際の実装には、興味のあるコマンドに対してのみ上書きする必要があるため、非常に簡潔で、テスト可能な実装が可能になります。

27.4. インターセプター

インターセプターは、コマンドにアクセスすることができる Visitors の特殊なタイプですが、チェーンでも動作します。インターセプターのチェーンはすべてコマンドにアクセスします。1 つは、登録さ

れたすべてのインターセプターがコマンドにアクセスするまで行います。

注意すべきクラスは CommandInterceptor です。この抽象クラスはインターセプターパターンを実装し、Bositor も実装します。Red Hat Data Grid のインターセプターは CommandInterceptor を拡張し、ネットワーク全体での分散やディスクへの書き込みなど、特定のコマンドに特定の動作を追加します。

また、実験的な非同期インターセプターも使用できます。非同期インターセプターに使用されるインターフェースは AsyncInterceptor とベース実装で、カスタム実装に BaseCustomAsyncInterceptor が必要な場合に使用する必要があります。このクラスは Visitor インターフェースも実装することに注意してください。

27.5. すべてを 1 つにまとめる

どのようにして一緒になるのでしょうか？キャッシュを呼び出すと、キャッシュが最初に呼び出しのコンテキストを作成します。呼び出しコンテキストには、呼び出しのトランザクション特性などが含まれます。その後、キャッシュは呼び出しのコマンドを作成し、パラメーターと他のサブシステムへの参照でコマンドインスタンスを初期化するコマンドファクトリーを使用します。

その後、キャッシュは呼び出しのコンテキストおよびコマンドを InterceptorChain に渡します。最後に、コマンドの perform（）メソッドが呼び出され、戻り値がある場合は呼び出し元に伝播されます。

27.6. サブシステムマネージャー

インターセプターは単純なインターセプションポイントとして機能し、ロジック自体が多数含まれていません。ほとんどの動作ロジックは、さまざまなサブシステムのマネージャーとしてカプセル化されます。以下の小さなサブセットになります。

27.6.1. DistributionManager

クラスター全体でエントリーの分散方法を制御するマネージャー。

27.6.2. TransactionManager

Manager はトランザクションを処理するのではなく、通常はサードパーティーによって提供されます。

27.6.3. RpcManager

クラスターのノード間でコマンドを複製するマネージャーです。

27.6.4. LockManager

操作に必要なときにロック鍵を処理するマネージャー。

27.6.5. PersistenceManager

設定されたキャッシュストアへのデータの永続化を処理するマネージャー。

27.6.6. DataContainer

メモリーエントリーの実際の内容を保持するコンテナー。

27.6.7. 設定

このキャッシュのすべての設定の詳細を示すコンポーネント。

27.7. COMPONENTREGISTRY

上記のさまざまなマネージャーとコンポーネントが作成され、キャッシュで使用するために保存されるレジストリー。他のマネージャーや重要なコンポーネントはすべてレジストリーを介して実行できます。

レジストリー自体は、軽量の依存関係インジェクションフレームワークで、コンポーネントとマネージャーは相互に参照して初期化できるようにします。以下は、DataContainer と Configuration の依存関係を宣言するコンポーネントの例になります。また、DataContainerFactory はその場で DataContainers を構築する機能を宣言します。

```
---
  @Inject
  public void injectDependencies(DataContainer container, Configuration configuration) {
    this.container = container;
    this.configuration = configuration;
  }
```

```
  @DefaultFactoryFor
  public class DataContainerFactory extends AbstractNamedCacheComponentFactory {
---
```

ComponentRegistry で登録されたコンポーネントにはライフサイクルがある場合があり、@Start または @Stop アノテーションが付けられたメソッドはコンポーネントレジストリーで使用される前後に呼び出されます。

```
---
  @Start
  public void init() {
    useWriteSkewCheck = configuration.locking().writeSkewCheck();
  }
```

```
  @Stop(priority=20)
  public void stop() {
    notifier.removeListener(listener);
    executor.shutdownNow();
  }
---
```

上記の例では、@Stop へのオプションの priority パラメーターを使用して、他のコンポーネントに関連してコンポーネントが停止している順序を示すために使用されます。これは、Unix Sys-V 形式の順序付けに従います。優先順位の高い方法よりも優先度が高い方法の前に呼び出されます。デフォルトの優先度は 10 です。指定されていない場合は、デフォルトの優先度は 10 です。