Red Hat Data Grid 8.0

Data Grid 開発者ガイド

Data Grid のドキュメント

Last Updated: 2023-06-04

Red Hat Data Grid 8.0 Data Grid 開発者ガイド

Data Grid のドキュメント

法律上の通知

Copyright © 2023 Red Hat, Inc.

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

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

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, 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

Data Grid API について説明し、Data Grid と対話するコードを作成する方法を説明します。

目次

第1章 RED HAT DATA GRID **5**
1.1. DATA GRID のドキュメント 5
1.2. DATA GRID のダウンロード 5

第2章 DATA GRID MAVEN リポジトリーの設定 **6**
2.1. DATA GRID MAVEN リポジトリーのダウンロード 6
2.2. RED HAT GA MAVEN リポジトリーの追加 6
2.3. DATA GRID POM の設定 7

第3章 キャッシュマネージャー **8**
3.1. キャッシュの取得 8
3.2. クラスターリング情報 9
3.3. メンバー情報 9

第4章 DATA GRID キャッシュインターフェイス **10**
4.1. キャッシュ API 10
4.2. ADVANCEDCACHE API 11
4.3. リスナーおよび通知 11
4.4. ASYNCHRONOUS API 15

第5章 データエンコーディングおよび MEDIATYPES **17**
5.1. 概要 17
5.2. デフォルトのエンコーダー 17
5.3. プログラムでの上書き 18
5.4. カスタムエンコーダーの定義 18
5.5. MEDIATYPE 20

第6章 プロトコルの相互運用性 **24**
6.1. メディアタイプおよびエンドポイント相互運用性に関する考慮事項 24
6.2. REST、HOT ROD、および MEMCACHED のテキストベースのストレージとの相互運用性 24
6.3. REST、HOT ROD、および MEMCACHED とカスタム JAVA オブジェクトとの相互運用性 25
6.4. PROTOBUF を使用した JAVA および非 JAVA クライアントの相互運用性 26
6.5. カスタムコードの相互運用性 27
6.6. エンティティークラスのデプロイ 29

第7章 MARSHALLING JAVA OBJECTS **30**
7.1. USING THE PROTOSTREAM MARSHALLER 30
7.2. JBOSS MARSHALLING の使用 31
7.3. JAVA シリアライゼーションの使用 31
7.4. KRYO MARSHALLER の使用 32
7.5. PROTOSTUFF MARSHALLER の使用 33
7.6. カスタムマーシャラーの使用 34
7.7. ADDING JAVA CLASSES TO DESERIALIZATION WHITE LISTS 34
7.8. DATA GRID サーバーでシリアル化されたオブジェクトの保存 35
7.9. バイナリー形式でのデータの保存 35

第8章 MARSHALLING CUSTOM JAVA OBJECTS WITH PROTOSTREAM **37**
8.1. PROTOBUF スキーマ 37
8.2. PROTOSTREAM シリアライゼーションコンテキスト 37
8.3. PROTOSTREAM タイプ 38
8.4. シリアル化コンテキストイニシャライザーの生成 38
8.5. シリアライゼーションコンテキストイニシャライザーの手動実装 42

第9章 クラスター化されたロック **45**
9.1. インストール 45
9.2. CLUSTEREDLOCK 設定 45
9.3. CLUSTEREDLOCKMANAGER インターフェイス 45
9.4. CLUSTEREDLOCK インターフェイス 47

第10章 クラスター化カウンター **49**
10.1. インストールおよび設定 49
10.2. COUNTERMANAGER インターフェイス 52
10.3. カウンター 52
10.4. 通知およびイベント 57

第11章 ロックおよび同時実行 **59**
11.1. 実装の詳細のロック 59
11.2. データのバージョン管理 61

第12章 DATA GRID CDI エクステンションの使用 **62**
12.1. CDI 依存関係 62
12.2. 組み込みキャッシュのインジェクト 62
12.3. リモートキャッシュの注入 64
12.4. JCACHE キャッシングアノテーション 65
12.5. キャッシュおよびキャッシュマネージャーイベントの受信 67

第13章 DATA GRID トランザクション **68**
13.1. トランザクションの設定 68
13.2. 分離レベル 70
13.3. トランザクションのロック 71
13.4. スキューの書き込み 72
13.5. 例外への対処 73
13.6. 同期の登録 73
13.7. バッチ処理 74
13.8. トランザクションリカバリー 75
13.9. TOTAL ORDER ベースのコミットプロトコル 78

第14章 インデックスとクエリー **84**
14.1. 概要 84
14.2. 埋め込みクエリー 84
14.3. リモートクエリー 114
14.4. 統計 118
14.5. パフォーマンスチューニング 118

第15章 GRID でのコードの実行 **120**
15.1. クラスターエグゼキューター 120

第16章 ストリーム **124**
16.1. 一般的なストリーム操作 124
16.2. キーのフィルターリング 124
16.3. セグメントベースのフィルターリング 124
16.4. ローカル/無効化 125
16.5. 例 125
16.6. 配布/複製/散在 125
16.7. 並列計算 128
16.8. タスクのタイムアウト 128
16.9. 注入 129
16.10. 分散ストリームの実行 129

16.11. キーベースの再ハッシュ対応 OPERATOR 130
16.12. 中間オペレーションの例外 131
16.13. 例 131

第17章 JCACHE (JSR-107) API 134
17.1. 組み込みキャッシュの作成 134
17.2. リモートキャッシュの作成 135
17.3. データの保管および取得 136
17.4. JAVA.UTIL.CONCURRENT.CONCURRENTMAP と JAVAX.CACHE.CACHE APIS の比較 136
17.5. JCACHE インスタンスのクラスターリング 138

第18章 マルチマップキャッシュ 139
18.1. インストールと設定 139
18.2. MULTIMAPCACHE API 139
18.3. マルチマップキャッシュの作成 141
18.4. 制限事項 141

第19章 カスタムインターセプター 142
19.1. カスタムインターセプターの宣言的追加 142
19.2. プログラムによるカスタムインターセプターの追加 142
19.3. カスタムインターセプターの設計 142

第1章 RED HAT DATA GRID

Data Grid は、高性能の分散型インメモリーデータストアです。

スキーマレスデータ構造

さまざまなオブジェクトをキーと値のペアとして格納する柔軟性があります。

グリッドベースのデータストレージ

クラスター間でデータを分散および複製するように設計されています。

エラスティックスケーリング

サービスを中断することなく、ノードの数を動的に調整して要件を満たします。

データの相互運用性

さまざまなエンドポイントからグリッド内のデータを保存、取得、およびクエリーします。

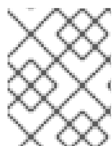
1.1. DATA GRID のドキュメント

Data Grid のドキュメントは、Red Hat カスタマーポータルで入手できます。

- Data Grid 8.0 ドキュメント
- Data Grid 8.0 コンポーネントの詳細
- Data Grid 8.0 でサポートされる設定

1.2. DATA GRID のダウンロード

Red Hat カスタマーポータルで Data Grid Software Downloads にアクセスします。



注記

Data Grid ソフトウェアにアクセスしてダウンロードするには、Red Hat アカウントが必要です。

第2章 DATA GRID MAVEN リポジトリーの設定

Data Grid Java ディストリビューションは Maven から入手できます。

顧客ポータルから Data Grid Maven リポジトリーをダウンロードするか、パブリック Red Hat Enterprise Maven リポジトリーから Data Grid 依存関係をプルできます。

2.1. DATA GRID MAVEN リポジトリーのダウンロード

パブリック Red Hat Enterprise Maven リポジトリーを使用しない場合は、ローカルファイルシステム、Apache HTTP サーバー、または Maven リポジトリーマネージャーに Data Grid Maven リポジトリーをダウンロードし、インストールします。

手順

1. Red Hat カスタマーポータルにログインします。
2. Software Downloads for Data Grid に移動します。
3. Red Hat Data Grid 8.0 Maven リポジトリーをダウンロードします。
4. アーカイブされた Maven リポジトリーをローカルファイルシステムに展開します。
5. **README.md** ファイルを開き、適切なインストール手順に従います。

2.2. RED HAT GA MAVEN リポジトリーの追加

Maven 設定ファイル (通常は **~/.m2/settings.xml**) を設定して、Red Hat GA リポジトリーを追加します。または、リポジトリーをプロジェクトの **pom.xml** ファイルに直接追加します。

以下の設定では、パブリックの Red Hat Enterprise Maven リポジトリーを使用します。Red Hat カスタマーポータルからダウンロードした Data Grid Maven リポジトリーを使用するには、**url** 要素の値を正しい場所に変更します。

```
<repositories>
  <repository>
    <id>redhat-ga</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>redhat-ga</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </pluginRepository>
</pluginRepositories>
```

参照資料

- Red Hat Enterprise Maven Repository

2.3. DATA GRID POM の設定

Maven は、プロジェクトオブジェクトモデル (POM) ファイルと呼ばれる設定ファイルを使用して、プロジェクトを定義し、ビルドを管理します。POM ファイルは XML 形式であり、モジュールとコンポーネントの依存関係、ビルドの順序、および結果となるプロジェクトのパッケージ化と出力のターゲットを記述します。

手順

1. プロジェクト **pom.xml** を開いて編集します。
2. 正しい Data Grid バージョンで **version.infinispan** プロパティーを定義します。
3. **dependencyManagement** セクションに **infinispan-bom** を含めます。
   BOM(Bill of Materials) は、依存関係バージョンを制御します。これにより、バージョンの競合が回避され、プロジェクトに依存関係として追加する Data Grid アーティファクトごとにバージョンを設定する必要がなくなります。
4. **pom.xml** を保存して閉じます。

以下の例は、Data Grid のバージョンと BOM を示しています。

```
<properties>
  <version.infinispan>10.1.8.Final-redhat-00001</version.infinispan>
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-bom</artifactId>
      <version>${version.infinispan}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

次のステップ

必要に応じて、Data Grid アーティファクトを依存関係として **pom.xml** に追加します。

第3章 キャッシュマネージャー

CacheManager インターフェイスは Data Grid の主なエントリーポイントであり、以下を可能にします。

- キャッシュを設定および取得します。
- ノードの管理およびモニター
- クラスター全体でコードを実行する
- 詳細情報

Data Grid をアプリケーションに埋め込むか、リモートサーバーとして実行するかに応じて、**EmbeddedCacheManager** または **RemoteCacheManager** のいずれかを使用します。それらは一部のメソッドとプロパティーを共有しますが、それらにはセマンティックの違いがあることに注意してください。以下の章では、**組み込み** 実装の多くに重点を置いています。

CacheManager はヘビーウェイト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複数の設定が必要でない限り、いずれの方法でも最小および有限のインスタンス数になります）。

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();
```

キャッシュなしで最も基本的なローカルモード、クラスター化されていないキャッシュマネージャーを開始します。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も Lifecycle.start() を呼び出します。コンストラクターの過負荷バージョンが利用できます。CacheManager は起動しませんが、CacheManager を使用して Cache インスタンスを作成する前に、CacheManager を起動する必要があることに注意してください。

構築後は、JNDI、ServletContext、または IoC コンテナーなどの他のメカニズムを使用して対話する必要のあるコンポーネントで CacheManagers を利用できるようにする必要があります。

CacheManager で終了したら、**manager.stop();** のリソースを解放できるように停止する必要があります。

これにより、スコープ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、クラスターも適切に残ります。

3.1. 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。

以下のように、キャッシュを取得するために **getCache(String)** メソッドを呼び出します。

```
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。クラスター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。

3.2. クラスターリング情報

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。

3.3. 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、ローカルノードがコーディネーターであるかどうかを確認することもできます。

第4章 DATA GRID キャッシュインターフェイス

Data Grid は、JDK の ConcurrentMap インターフェイスによって公開されるアトミックメカニズムを含む、エントリーを追加、取得、および削除するための簡単なメソッドを公開する Cache インターフェイスを提供します。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ーを複製したり、リモートノードからエントリーを検索することやキャッシュストアからエントリーを検索することなど、数多くのことが発生します。

4.1. キャッシュ API

単純な使用の場合、Cache API の使用は JDK Map API の使用と違いがないはずです。したがって、マップに基づく単純なインメモリーキャッシュから Data Grid のキャッシュへの移行は簡単になります。

4.1.1. 特定のマップメソッドのパフォーマンスに関する懸念

size()、values()、keySet()、および entrySet() など、マップで公開される特定のメソッドは、Data Grid と使用すると特定のパフォーマンスに影響します。**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、これらの問題の一部を軽減できます。詳細は、各メソッドのドキュメントを参照してください

4.1.2. Mortal および Immortal データ

単にエントリーを格納するだけでなく、Data Grid のキャッシュ API を使用すると、期限付き情報をデータに添付できます。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、put(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。これは固定のライフスパンのあるエントリーで、そのライフスパンの後に期限切れになります。

Data Grid は、**lifespan**の他に、有効期限を決定する追加のメトリクスとして**maxIdle**もサポートします。lifespans または maxIdles の任意の組み合わせを使用できます。

4.1.3. putForExternalRead 操作

Data Grid の Cache クラスには、putForExternalRead と呼ばれる異なる 'put' 操作が含まれます。この操作は、他の場所で保持されるデータの一時キャッシュとして Data Grid が使用され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。

これを実現するため、キーがキャッシュ内に存在しない場合にのみ動作する put 呼び出しとして**putForExternalRead()**が動作し、別のスレッドが同じキーを同時に格納しようとすると、通知なしに即座に失敗します。このシナリオでは、データのキャッシュはシステムを最適化する方法で、キャッシングの失敗が実行中のトランザクションに影響するのは望ましくないため、失敗の処理方法が異なります。成功したかどうかに関わらず、ロックを待たず、読み出し元に即座に返されるため、**putForExternalRead()**は高速な操作とみなされます。

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、アプリケーションの実行元 (Person のアドレスを変更するトランザクションからなど) となる新しい 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。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。

4.2. ADVANCEDCACHE API

簡単なキャッシュインターフェイスの他に、Data Grid はエクステンション作成者向けに AdvancedCache インターフェイスを提供します。AdvancedCache は、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();
```

4.2.1. フラグ

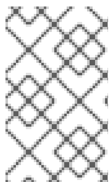
フラグは通常のキャッシュメソッドに適用され、特定のメソッドの動作を変更します。利用可能なフラグの一覧と、その効果については、Flag 列挙を参照してください。フラグは、AdvancedCache.withFlags() を使用して適用されます。このビルダーメソッドを使用して、キャッシュ呼び出しに任意の数のフラグを適用できます。次に例を示します。

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

4.3. リスナーおよび通知

Data Grid はリスナー API を提供し、クライアントはイベントが発生したときに登録して通知を受け取ることができます。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、Listenable インターフェイスで定義されたメソッドを使用して登録された単純な POJO です。



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、次のクラスは、新しいエントリーがキャッシュに追加されるたびに、ブロックしない方法で、一部の情報を出力するようにリスナーを定義します。

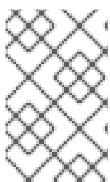
```
@Listener
public class PrintWhenAdded {
  Queue<CacheEntryCreatedEvent> events = new ConcurrentLinkedQueue<>();

  @CacheEntryCreated
  public CompletionStage<Void> print(CacheEntryCreatedEvent event) {
    events.add(event);
    return null;
  }

}
```

より包括的な例は Javadocs for @Listener を参照してください。

4.3.1. 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、エントリーの追加、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とそれぞれのメソッドレベルのアノテーションの包括的なリストについては、org.infinispan.notifications.cachelistener.annotation パッケージの Javadocs を参照してください。



注記

Data Grid で使用可能なキャッシュレベルの通知のリストについては org.infinispan.notifications.cachelistener.annotation パッケージの Javadocs を参照してください。

4.3.1.1. 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。

4.3.1.2. イベントのフィルターリングおよび変換

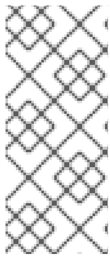
リスナーがインストールされているノードで適用可能なすべてのイベントがリスナーに発生します。KeyFilter(キーのフィルターリングのみを許可) または CacheEventFilter(キー、古い値、古いメタデータ、新しい値、新しいメタデータ、コマンドの再実行の有無、イベントがイベント (isPre など) の前であるか、およびコマンドタイプのフィルターに使用) を使用して、どのイベントが発生したかを動的にフィルターできます。

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;
   }

   public 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、イベントが発生する前に値を別の値に変換できるようにする CacheEventConverter もあります。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。



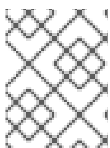
注記

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ー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。

4.3.1.3. 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。

4.3.1.4. 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。

Data Grid は、指定のキーの新規プライマリー所有者に put 操作を自動的に送信することで、put 操作を内部で修正しますが、最初に書き込みが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た場合、Data Grid は再試行コマンドによって生成されたイベントをマークし、変更の表示に注意を払いなくても、このイベントが発生したタイミング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、EventTypeには、再試行によりイベントが生成されたかどうかを確認するために、メソッド **isRetry** が含まれます。

4.3.2. 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。

キャッシュマネージャーレベルのすべての通知とそれぞれのメソッドレベルのアノテーションの包括的なリストは、org.infinispan.notifications.cachemanagerlistener.annotation package を参照してください。

4.3.3. イベントの同期

デフォルトでは、すべての非同期通知は通知スレッドプールにディスパッチされます。同期通知は、リスナーメソッドが完了するか (スレッドがブロックする原因となる)、または CompletionStage が完了するまで、操作の続行を遅らせます。または、リスナーに**非同期**としてアノテーションを付けることもできます。この場合、操作は即座に継続され、通知は通知スレッドプールで非同期に完了します。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。

非同期リスナー

```
@Listener (sync = false)
public class MyAsyncListener {
  @CacheEntryCreated
  void listen(CacheEntryCreatedEvent event) { }
}
```

同期リスナーのブロック

```
@Listener
public class MySyncListener {
  @CacheEntryCreated
  void listen(CacheEntryCreatedEvent event) { }
}
```

ノンブロッキングリスナー

```
@Listener
public class MyNonBlockingListener {
  @CacheEntryCreated
  CompletionStage<Void> listen(CacheEntryCreatedEvent event) { }
}
```

4.3.3.1. 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。

4.4. ASYNCHRONOUS API

Cache.put()、Cache.remove() などの同期 API メソッドの他に、Data Grid には非同期のノンブロッキング API も含まれ、同じ結果をノンブロッキング方式で達成できます。

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。E.g., Cache.putAsync() , Cache.removeAsync() , etc. これらの非同期のメソッドは、操作の実際の結果が含まれる CompletableFuture を返します。

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

4.4.1. 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<CompletableFuture<?>> futures = new HashSet<>();
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 (CompletableFuture<?> f: futures) f.get();
```

4.4.2. 実際に非同期で発生するプロセス

Data Grid には、通常の書き込み操作の重要なパスにあると見なされる 4 つの項目があります。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

非同期メソッドを使用すると、ネットワーク呼び出しとマーシャリングがクリティカルパスから除外されます。 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。

第5章 データエンコーディングおよび MEDIATYPES

エンコーディングは、データを保存する前、およびストレージから読み戻すときに、Data Grid キャッシュによって実行されるデータ変換操作です。

5.1. 概要

エンコーディングを使用すると、API 呼び出し中に特定のデータ形式 (マップ、リスナー、ストリームなど) を処理できますが、効果的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();
}
```

5.2. デフォルトのエンコーダー

Data Grid は、キャッシュ設定に応じてエンコーダー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

Mode	設定	エンコーダー	説明
組み込み	StorageType.OFF_HEAP	GlobalMarshallerEncoder	Data Grid の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。
組み込み	StorageType.BINARY	BinaryEncoder	Data Grid の内部マーシャラーを使用して、プリミティブおよび String を除いて byte[] に変換します。
Server	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

5.3. 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().withEncoding(IdentityEncoder.class);
byte[] marshalled = (byte[]) 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。

5.4. 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。

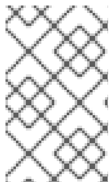
```
GlobalComponentRegistry registry = cacheManager.getGlobalComponentRegistry();
EncoderRegistry encoderRegistry = registry.getComponent(EncoderRegistry.class);
encoderRegistry.registerEncoder(new GzipEncoder());
```

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 stringValue = 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
```

5.5. MEDIATYPE

任意で、キャッシュはキーと値の **org.infinispan.commons.dataconversion.MediaType** で設定できます。キャッシュのデータフォーマットを記述することで、Data Grid はキャッシュ操作中にデータを即座に変換できます。



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は、MediaType が設定され、REST または Hot Rod などのクライアントが異なる形式で読み取りおよび書き込みを行うのが一般的です。

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();
}
```

5.5.1. 設定

宣言的 (Declarative)

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

プログラマティック

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

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

5.5.2. プログラムで MediaType の上書き

異なる MediaType でキャッシュを切り離すことができるため、キャッシュ操作を異なるデータ形式を送受信できます。

以下に例を示します。

```
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 Transcoders は、サーバーモードの使用時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します。

5.5.3. 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。

```
class Scrambler implements Encoder {

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

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

  @Override
  public boolean isStorageFormatFilterable() {

  }
```

```
    public MediaType getStorageFormat() {
    return new MediaType("application", "scrambled");
    }

    @Override
    public short id() {
    //return id
    }
}
```

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、Data Grid は最初にエンコーダー **fromStorage** 操作を適用して、application/x-jboss-marshalling 形式のエントリーを取得し、次に適切なトランスコーダーを使用して application/xml に順次変換を適用します。

第6章 プロトコルの相互運用性

クライアントは、REST や Hot Rod などのエンドポイントを介して Data Grid と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、クライアントが適切な形式でデータの読み取りと書き込みを行うことができます。Data Grid は複数のクライアントと同時に相互運用できるため、クライアント形式とストレージ形式間でデータを変換する必要があります。

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。

6.1. メディアタイプおよびエンドポイント相互運用性に関する考慮事項

特定のメディアタイプでデータを保存するように Data Grid を設定すると、クライアントの相互運用性に影響します。

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 ベースのキーと byte[] の値を処理できますが、サーバーと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際に柔軟性が高くありません。

Java Hot Rod 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 Hot Rod 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 Rod クライアントは、それぞれの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 以外の Hot Rod 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 Hot Rod クライアントと相互運用できます。

6.2. REST、HOT ROD、および MEMCACHED のテキストベースのストレージとの相互運用性

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。オプションの文字セットを使用して、JSON (**application/json**)や XML (**application/xml**)などの他のテキストベースの形式のメディアタイプを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 Hot Rod クライアントを設定する必要があります。

以下のように、キャッシュへの書き込みおよび読み取り時に、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ための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	いいえ
クエリーおよびインデックス	いいえ
カスタム Java オブジェクト	いいえ

6.3. REST、HOT ROD、および MEMCACHED とカスタム JAVA オブジェクトとの相互運用性

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保存する場合は、マーシャリングされたストレージの MediaType でキャッシュを設定する必要があります。

Java Hot Rod クライアントは、JBoss マーシャリングストレージ形式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** として設定します。UTF8Marshaller の場合は、MediaType を **text/plain** として設定します。

ヒント

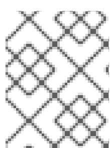
Hot Rod クライアントのみがキャッシュと対話する場合は、MediaType を設定する必要はありません。

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、REST クライアントは、サポートされ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、クラスは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。値はマーシャリングされたオブジェクトとして保存され、返されます。

一部の Java Memcached クライアントは、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を許可します。また、言語に依存しない JSON などの異なる形式で応答をエンコードするように Memcached サーバーモジュールを設定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の形式が 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。



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	いいえ
クエリーおよびインデックス	いいえ
カスタム Java オブジェクト	はい

6.4. PROTOBUF を使用した JAVA および非 JAVA クライアントの相互運用性

Protobuf でエンコードされたエントリーとしてデータをキャッシュに保存すると、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ると、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。

キャッシュに対して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>
```

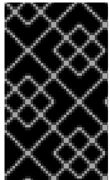
Data Grid は、**application/x-protostream** と **application/json** の間で変換します。これにより、REST クライアントは JSON 形式のデータの読み取りと書き込みが可能になります。ただし、REST クライアントは以下のように正しいヘッダーを送信する必要があります。

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。これには、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 Protocol Buffers スキーマ定義を登録する必要があります。

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	はい
クエリーおよびインデックス	はい
カスタム Java オブジェクト	はい

6.5. カスタムコードの相互運用性

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内のデータに直接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、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し、他のクライアントはデータをバイナリー形式で保存します。

カスタムコードとの相互運用性を処理するには、データをオンデマンドで変換するか、Plain Old Java Objects (POJO)としてデータを保存することができます。

6.5.1. Converting Data On Demand

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

するようにキャッシュが設定されている場合は、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デプロイされたコードを設定できます。MediaType プログラムの上書き 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バイナリー形式を使用してキャッシュエントリーを保存することができます。これは最適です。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、キャッシュがバイナリー形式として Protobuf (**application/x-protostream**)を使用する場合は、Data Grid がカスタムコードからのデータを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。

6.5.2. データを POJO として保存

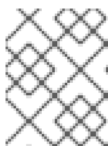
アンマーシャリングされた Java オブジェクトをサーバーに保存することは推奨されません。これを実行するには、リモートクライアントがキャッシュから読み取られたときに Data Grid がデータをシリアライズし、リモートクライアントがキャッシュに書き込むときにデータをデシリアライズする必要があります。

以下の例では、**application/x-java-object** 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>
```

Hot Rod クライアントは、データが 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である POJO として保存されている場合は、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、Data Grid が Java オブジェクト（現在 JSON または XML）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 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 マーシャリングされたペイロードである、保存された POJO のシリアライズされたバージョンを送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	いいえ

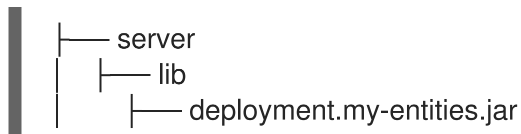
この設定は、以下と互換性があります。	
クエリーおよびインデックス	有効。ただし、クエリーとインデックスは、エンティティーにアノテーションが付けられている場合にのみ POJO で機能します。
カスタム Java オブジェクト	はい

6.6. 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。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し、Data Grid がそれらをシリアライズおよびデシリアライズできるようにします。

エンティティークラスをサーバーで利用可能にするには、以下の手順を実施します。

1. 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。
2. Data Grid が実行している場合は停止します。Data Grid は、起動時にエンティティークラスのみを読み込みます。
3. JAR を Data Grid サーバーのインストールの **server/lib** ディレクトリーにコピーします。

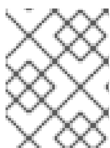


第7章 MARSHALLING JAVA OBJECTS

マーシャリングは、Java オブジェクトをバイナリー形式に変換して、ネットワーク経由で転送したり、ディスクに保存したりできるようにします。逆プロセスのアンマーシャリングは、データをバイナリー形式から Java オブジェクトに変換します。

Data Grid は、マーシャリングとアンマーシャリングを実行し、以下を行います。

- クラスターの他の Data Grid ノードにデータを送信します。
- データを永続キャッシュストアに保存します。
- データをバイナリー形式で格納し、遅延デシリアライズ機能を提供します。



注記

Data Grid は、すべての内部タイプのマーシャリングを処理します。格納する Java オブジェクトに対してのみマーシャリングを設定する必要があります。

Data Grid は、Java オブジェクトをバイナリー形式にマーシャリングするデフォルトとして ProtoStream を使用します。Data Grid は、使用可能な他の Marshaller 実装も提供します。

7.1. USING THE PROTOSTREAM MARSHALLER

Data Grid は ProtoStream API と統合し、言語に依存しない下位互換性のある形式である Protocol Buffers (Protobuf)に Java オブジェクトをエンコードおよびデコードします。

手順

1. Data Grid が Java オブジェクトをマーシャリングできるように、ProtoStream **SerializationContextInitializer** インターフェイスの実装を作成します。
2. 実装を使用するように Data Grid を設定します。
   - プログラムで行う:

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
       .addContextInitializers(new LibraryInitializerImpl(), new SCIImpl());
```

   - 宣言的に

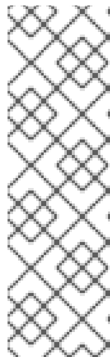
```
<serialization>
   <context-initializer class="org.infinispan.example.LibraryInitializerImpl"/>
   <context-initializer class="org.infinispan.example.another.SCIImpl"/>
</serialization>
```

参照資料

- ProtoStream マーシャリング用のシリアル化コンテキストの作成
- プロトコルバッファー

7.2. JBOSS MARSHALLING の使用

JBoss Marshalling はシリアル化ベースのマーシャリングライブラリーであり、以前の Data Grid バージョンではデフォルトのマーシャラーでした。



注記

- Data Grid では、シリアル化ベースのマーシャリングを使用しないでください。代わりに、後方互換性を保証する高性能のバイナリーワイヤー形式である Protostream を使用する必要があります。
- JBoss Marshalling および **AdvancedExternalizer** インターフェイスは非推奨となり、今後のリリースで削除される予定です。しかし、Data Grid は、JBoss Marshalling を使用せずにデータを永続化したときに **AdvancedExternalizer** 実装を無視します。

手順

1. **infinispan-jboss-marshalling** 依存関係をクラスパスに追加します。
2. **JBossUserMarshaller** を使用するように Data Grid を設定します。
   - プログラムで行う:

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization().marshaller(new JBossUserMarshaller());
```

   - 宣言的に行う:

```
<serialization marshaller="org.infinispan.jboss.marshalling.core.JBossUserMarshaller"/>
```

参照資料

- Adding Java Classes to Deserialization White Lists
- AdvancedExternalizer

7.3. JAVA シリアライゼーションの使用

Data Grid で Java シリアライゼーションを使用すると、Java オブジェクトが Java **Serializable** インターフェイスを実装する場合にのみ、オブジェクトをマーシャリングできます。

手順

1. **JavaSerializationMarshaller** をマーシャラーとして使用するように Data Grid を設定します。
2. Java クラスをデシリアライズホワイトリストに追加します。
   - プログラムで行う:

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
       .marshaller(new JavaSerializationMarshaller())
```

```
    .whiteList()
    .addRegexps("org.infinispan.example.", "org.infinispan.concrete.SomeClass");
```

- 宣言的に行う:

```
<serialization
marshaller="org.infinispan.commons.marshall.JavaSerializationMarshaller">
  <white-list>
    <class>org.infinispan.concrete.SomeClass</class>
    <regex>org.infinispan.example.*</regex>
  </white-list>
</serialization>
```

参照資料

- Adding Java Classes to Deserialization White Lists
- Serializable
- org.infinispan.commons.marshall.JavaSerializationMarshaller

7.4. KRYO MARSHALLER の使用

Data Grid は、Kryo ライブラリーを使用するマーシャリング実装を提供します。

Data Grid サーバーの前提条件

Data Grid サーバーで Kryo マーシャリングを使用するには、以下のように Kryo マーシャリング実装のランタイムクラスファイルが含まれる JAR を追加します。

1. Data Grid Maven リポジトリーから **infinispan-marshaller-kryo-bundle.jar** をコピーします。
2. JAR ファイルを Data Grid サーバーのインストールディレクトリーにある **server/lib** ディレクトリーに追加します。

Data Grid ライブラリーモードの前提条件

Data Grid で Kryo マーシャリングをアプリケーションの埋め込みライブラリーとして使用するには、以下の手順を実行します。

1. **infinispan-marshaller-kryo** 依存関係を **pom.xml** に追加します。

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

2. **org.infinispan.marshaller.kryo.KryoMarshaller** クラスをマーシャラーとして指定します。

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
      .marshaller(new org.infinispan.marshaller.kryo.KryoMarshaller());
```

手順

1. **SerializerRegistryService.java** インターフェイスのサービスプロバイダーを実装します。
2. **register(Kryo)** メソッドにすべてのシリアライザー登録を配置します。シリアライザーは、Kryo API を使用して提供された **Kryo** オブジェクトに登録されます。以下に例を示します。

```
kryo.register(ExampleObject.class, new ExampleObjectSerializer())
```

3. 以下の範囲内で、デプロイメント JAR ファイルにクラスを実装するためのフルパスを指定します。

```
META-INF/services/org/infinispan/marshaller/kryo/SerializerRegistryService
```

参照資料

- Kryo on GitHub

7.5. PROTOSTUFF MARSHALLER の使用

Data Grid は、Protostuff ライブラリーを使用するマーシャリング実装を提供します。

Data Grid サーバーの前提条件

Data Grid サーバーで Protostuff マーシャリングを使用するには、以下のように Protostuff マーシャリング実装のランタイムクラスファイルが含まれる JAR を追加します。

1. Data Grid Maven リポジトリーから **infinispan-marshaller-protostuff-bundle.jar** をコピーします。
2. JAR ファイルを Data Grid サーバーのインストールディレクトリーにある **server/lib** ディレクトリーに追加します。

Data Grid ライブラリーモードの前提条件

Data Grid で Protostuff マーシャリングをアプリケーションの埋め込みライブラリーとして使用するには、以下の手順を実行します。

1. **infinispan-marshaller-protostuff** 依存関係を **pom.xml** に追加します。

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

2. **org.infinispan.marshaller.protostuff.ProtostuffMarshaller** クラスをマーシャラーとして指定します。

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
       .marshaller(new org.infinispan.marshaller.protostuff.ProtostuffMarshaller());
```

手順

オブジェクトマーシャリングのカスタム Protostuff スキーマを登録するには、以下のいずれかを行います。

- **register()** メソッドを呼び出します。

```
RuntimeSchema.register(ExampleObject.class, new ExampleObjectSchema());
```

- **register()** メソッドにすべてのスキーマ登録を配置する **SerializerRegistryService.java** インターフェイスのサービスプロバイダーを実装します。
その後、以下の範囲内で、デプロイメント JAR ファイルにクラスを実装するためのフルパスを指定します。

```
META-INF/services/org/infinispan/marshaller/protostuff/SchemaRegistryService
```

参照資料

- Protostuff on GitHub

7.6. カスタムマーシャラーの使用

Data Grid は、カスタムマーシャラーの **Marshaller** インターフェイスを提供します。

プログラムによる手順

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
      .marshaller(new org.infinispan.example.marshall.CustomMarshaller())
      .whiteList().addRegexp("org.infinispan.example.*");
```

宣言型手順

```
<serialization marshaller="org.infinispan.example.marshall.CustomMarshaller">
   <white-list>
      <class>org.infinispan.concrete.SomeClass</class>
      <regex>org.infinispan.example.*</regex>
   </white-list>
</serialization>
```

ヒント

カスタムマーシャラーの実装は、起動時に呼び出される initialize() メソッドを使用して設定済みの許可リストにアクセスできます。

参照資料

- org.infinispan.commons.marshall.Marshaller

7.7. ADDING JAVA CLASSES TO DESERIALIZATION WHITE LISTS

Data Grid では、セキュリティー上の理由から、任意の Java クラスのデシリアルは許可されません。これは、JSON、XML、およびマーシャリングされた **byte[]** コンテンツが該当します。

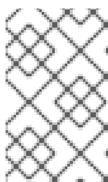
システムプロパティーを使用するか、または Data Grid 設定で Java クラスを指定して、デシリアライズホワイトリストに Java クラスを追加する必要があります。

システムプロパティー

```
// Specify a comma-separated list of fully qualified class names
-Dinfinispan.deserialization.whitelist.classes=java.time.Instant,com.myclass.Entity

// Specify a regular expression to match classes
-Dinfinispan.deserialization.whitelist.regexps=.*
```

宣言的 (Declarative)

```
<cache-container>
  <serialization version="1.0" marshaller="org.infinispan.marshall.TestObjectStreamMarshaller">
    <white-list>
      <class>org.infinispan.test.data.Person</class>
      <regex>org.infinispan.test.data.*</regex>
    </white-list>
  </serialization>
</cache-container>
```



注記

デシリアライズホワイトリストに追加する Java クラスは、Data Grid **CacheContainer** に適用され、**CacheContainer** が制御するすべてのキャッシュによってデシリアライズできます。

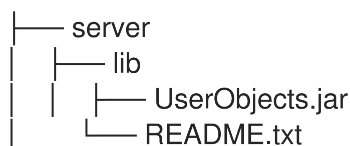
7.8. DATA GRID サーバーでシリアル化されたオブジェクトの保存

application/x-java-object MediaType をデータの形式として使用するように Data Grid を設定できます。つまり、Data Grid はデータをバイナリーコンテンツではなく Plain Old Java Objects (POJO)として保存します。

POJO を保存する場合は、すべてのカスタムオブジェクトのクラスファイルを Data Grid サーバークラスパスに配置する必要があります。

手順

- **server/lib** ディレクトリーにマーシャラー実装用のカスタムクラスやサービスプロバイダーを含む JAR ファイルを追加します。



```
├── server
│   ├── lib
│   │   ├── UserObjects.jar
│       └── README.txt
```

7.9. バイナリー形式でのデータの保存

Data Grid は、データをバイナリー形式でシリアライズされた形式で保存し、必要に応じて Java オブジェクトをシリアライズまたはデシリアライズできます。この動作は Lazy deserialization とも呼ばれます。

プログラムによる手順

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

宣言型手順

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

等価性に関する考慮事項

データをバイナリー形式で保存する場合、Data Grid はキーと値に **WrappedBytes** インターフェイスを使用します。このラッパークラスは、オンデマンドでシリアライズおよびデシリアライズを透過的に処理し、内部的には、ラップされているオブジェクト自体への参照、または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。これは等価性の動作に影響します。これは、キーに **equals()** メソッドを実装する場合に注意することが重要です。

ラッパークラスの **equals()** メソッドは、比較対象の両方のインスタンスが比較時にシリアライズされた形式かデシリアライズされた形式かによって、バイナリー表現 (バイト配列) を比較するか、ラップされたオブジェクトインスタンスの **equals()** メソッドに委任します。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。

参照資料

- org.infinispan.commons.marshall.WrappedBytes.

第8章 MARSHALLING CUSTOM JAVA OBJECTS WITH PROTOSTREAM

Data Grid は ProtoStream API を使用して、言語に依存しない下位互換性のある形式で、Java オブジェクトをプロトコルバッファー (Protobuf) にエンコードおよびデコードします。

8.1. PROTOBUF スキーマ

プロトコルバッファー、Protobuf は Java オブジェクトの構造化表現を提供します。

Protobuf メッセージタイプの **.proto** スキーマファイルを以下の例のように定義します。

```
package book_sample;

message Book {
    optional string title = 1;
    optional string description = 2;
    optional int32 publicationYear = 3; // no native Date type available in Protobuf

    repeated Author authors = 4;
}

message Author {
    optional string name = 1;
    optional string surname = 2;
}
```

前述の **.library.proto** ファイルは、**book_sample** パッケージに含まれる **Book** という名前のエンティティー (Protobuf メッセージタイプ) を定義します。**Book** は、プリミティブ型のいくつかのフィールドと、**Author** メッセージタイプである **authors** という名前のアレイ (Protobuf 反復可能フィールド) を宣言します。

Protobuf メッセージ

- メッセージをネストできますが、結果的に構造は厳密にツリーであり、グラフではありません。
- 型の継承はできません。
- コレクションはサポート対象外ですが、フィールドを繰り返してアレイをエミュレートできます。

参照資料

- Protocol Buffers Developer Guide

8.2. PROTOSTREAM シリアライゼーションコンテキスト

ProtoStream **SerializationContext** には、**.proto** スキーマファイルからロードされたカスタム Java オブジェクトの Protobuf タイプ定義と、オブジェクトに付随するマーシャラーが含まれています。

SerializationContextInitializer インターフェイスは Java オブジェクトとマーシャラーを登録し、ProtoStream ライブラリーがカスタムオブジェクトを Protobuf 形式にエンコードできるようにします。これにより、Data Grid はデータを送信および保存できます。

8.3. PROTOSTREAM タイプ

ProtoStream は、追加の設定なしで、以下のタイプやプリミティブタイプの場合に同等のタイプを処理できます。

- **String**
- **整数**
- **Long**
- **double**
- **Float**
- **ブール値**
- **byte[]**
- **Byte**
- **Short**
- **Character**
- **java.util.Date**
- **java.time.Instant**

他の Java オブジェクトをマーシャリングするには、**SerializationContext** で **.proto** スキーマとマーシャラーを登録する **SerializationContextInitializer** 実装を生成または手動で作成する必要があります。

8.4. シリアル化コンテキストイニシャライザーの生成

Data Grid は、アノテーション付きの Java クラスから **.proto** スキーマと **SerializationContextInitializer** 実装を生成できる **protostream-processor** アーティファクトを提供します。

手順

1. **protostream-processor** 依存関係を **pom.xml** に追加します。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-bom</artifactId>
      <version>${version.infinispan}</version>
      <type>pom</type>
    </dependency>
    <dependency>
```

```
      <groupId>org.infinispan.protostream</groupId>
      <artifactId>protostream-processor</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

2. マーシャリングする Java オブジェクトに **@ProtoField** と**@ProtoFactory** アノテーションを付けます。

Book.java

```
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;
...

public class Book {
   @ProtoField(number = 1)
   final String title;

   @ProtoField(number = 2)
   final String description;

   @ProtoField(number = 3, defaultValue = "0")
   final int publicationYear;

   @ProtoField(number = 4, collectionImplementation = ArrayList.class)
   final List<Author> authors;

   @ProtoFactory
   Book(String title, String description, int publicationYear, List<Author> authors) {
      this.title = title;
      this.description = description;
      this.publicationYear = publicationYear;
      this.authors = authors;
   }
   // public Getter methods omitted for brevity
}
```

Author.java

```
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

public class Author {
   @ProtoField(number = 1)
   final String name;

   @ProtoField(number = 2)
   final String surname;

   @ProtoFactory
   Author(String name, String surname) {
      this.name = name;
      this.surname = surname;
```

```
    }
    // public Getter methods omitted for brevity
}
```

3. **SerializationContextInitializer** を拡張し、 **@AutoProtoSchemaBuilder** アノテーションが付けられたインターフェイスを定義します。

```
@AutoProtoSchemaBuilder(
      includeClasses = {
            Book.class,
            Author.class,
      },
      schemaFileName = "library.proto", (1)
      schemaFilePath = "proto/", (2)
      schemaPackageName = "book_sample")
interface LibraryInitializer extends SerializationContextInitializer {
}
```

(1) 生成された **.proto** スキーマファイルに名前を付けます。

(2) スキーマファイルが生成される **target/classes** の下にパスを設定します。

コンパイル時に、**protostream-processor** は、ProtoStream **SerializationContext** の初期化に使用できるインターフェイスの具体的な実装を生成します。デフォルトでは、実装名は Impl 接尾辞が付いたアノテーション付きクラス名です。

例

以下は、生成されたスキーマファイルと実装の例です。

target/classes/proto/library.proto

```
// File name: library.proto
// Generated from : org.infinispan.commons.marshall.LibraryInitializer

syntax = "proto2";

package book_sample;



message Book {

   optional string title = 1;

   optional string description = 2;

   optional int32 publicationYear = 3 [default = 0];

   repeated Author authors = 4;
}


message Author {
```

```
    optional string name = 1;

    optional string surname = 2;
}
```

LibraryInitializerImpl.java

```
/*
 Generated by
org.infinispan.protostream.annotations.impl.processor.AutoProtoSchemaBuilderAnnotationProcessor
 for class org.infinispan.commons.marshall.LibraryInitializer
 annotated with @org.infinispan.protostream.annotations.AutoProtoSchemaBuilder(dependsOn=,
service=false, autoImportClasses=false, excludeClasses=,
includeClasses=org.infinispan.commons.marshall.Book,org.infinispan.commons.marshall.Author,
basePackages={}, value={}, schemaPackageName="book_sample", schemaFilePath="proto/",
schemaFileName="library.proto", className="")
 */

package org.infinispan.commons.marshall;

/**
 * WARNING: Generated code!
 */
@javax.annotation.Generated(value =
"org.infinispan.protostream.annotations.impl.processor.AutoProtoSchemaBuilderAnnotationProcessor"
,
    comments = "Please do not edit this file!")
@org.infinispan.protostream.annotations.impl.OriginatingClasses({
    "org.infinispan.commons.marshall.Author",
    "org.infinispan.commons.marshall.Book"
})
/*@org.infinispan.protostream.annotations.AutoProtoSchemaBuilder(
   className = "LibraryInitializerImpl",
   schemaFileName = "library.proto",
   schemaFilePath = "proto/",
   schemaPackageName = "book_sample",
   service = false,
   autoImportClasses = false,
   classes = {
      org.infinispan.commons.marshall.Author.class,
      org.infinispan.commons.marshall.Book.class
   }
)*/
public class LibraryInitializerImpl implements org.infinispan.commons.marshall.LibraryInitializer {

   @Override
   public String getProtoFileName() { return "library.proto"; }

   @Override
   public String getProtoFile() { return
org.infinispan.protostream.FileDescriptorSource.getResourceAsString(getClass(),
"/proto/library.proto"); }

   @Override
   public void registerSchema(org.infinispan.protostream.SerializationContext serCtx) {
```

```
serCtx.registerProtoFiles(org.infinispan.protostream.FileDescriptorSource.fromString(getProtoFileName
(), getProtoFile()));
  }

  @Override
  public void registerMarshallers(org.infinispan.protostream.SerializationContext serCtx) {
    serCtx.registerMarshaller(new
org.infinispan.commons.marshall.Book$___Marshaller_cdc76a682a43643e6e1d7e43ba6d1ef6f794949
a45e1a8bc961046cda44c9a85());
    serCtx.registerMarshaller(new
org.infinispan.commons.marshall.Author$___Marshaller_9b67e1c1ecea213b4207541b411fb9af2ae6f65
8610d2a4ca9126484d57786d1());
  }
}
```

8.5. シリアライゼーションコンテキストイニシャライザーの手動実装

場合によっては、**.proto** スキーマファイルを手動で定義し、ProtoStream マーシャラーを実装する必要がある場合があります。たとえば、Java オブジェクトクラスを変更してアノテーションを追加できない場合などです。

手順

1. Protobuf メッセージで **.proto** スキーマを作成します。

```
package book_sample;

message Book {
   optional string title = 1;
   optional string description = 2;
   optional int32 publicationYear = 3; // no native Date type available in Protobuf

   repeated Author authors = 4;
}

message Author {
   optional string name = 1;
   optional string surname = 2;
}
```

2. **org.infinispan.protostream.MessageMarshaller** インターフェイスを使用して、クラスのマーシャラーを実装します。

BookMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class BookMarshaller implements MessageMarshaller<Book> {

  @Override
  public String getTypeName() {
    return "book_sample.Book";
  }
```

```
  @Override
  public Class<? extends Book> getJavaClass() {
    return Book.class;
  }

  @Override
  public void writeTo(MessageMarshaller.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(MessageMarshaller.ProtoStreamReader reader) throws IOException
{
    String title = reader.readString("title");
    String description = reader.readString("description");
    int publicationYear = reader.readInt("publicationYear");
    List<Author> authors = reader.readCollection("authors", new ArrayList<>(), Author.class);
    return new Book(title, description, publicationYear, authors);
  }
}
```

AuthorMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class AuthorMarshaller implements MessageMarshaller<Author> {

  @Override
  public String getTypeName() {
    return "book_sample.Author";
  }

  @Override
  public Class<? extends Author> getJavaClass() {
    return Author.class;
  }

  @Override
  public void writeTo(MessageMarshaller.ProtoStreamWriter writer, Author author) throws
IOException {
    writer.writeString("name", author.getName());
    writer.writeString("surname", author.getSurname());
  }

  @Override
  public Author readFrom(MessageMarshaller.ProtoStreamReader reader) throws
IOException {
    String name = reader.readString("name");
    String surname = reader.readString("surname");
    return new Author(name, surname);
  }
}
```

3. **.proto** スキーマと ProtoStream マーシャラー実装を **SerializationContext** に登録する**SerializationContextInitializer** 実装を作成します。

ManualSerializationContextInitializer.java

```
import org.infinispan.protostream.FileDescriptorSource;
import org.infinispan.protostream.SerializationContext;
import org.infinispan.protostream.SerializationContextInitializer;
...

public class ManualSerializationContextInitializer implements SerializationContextInitializer {
  @Override
  public String getProtoFileName() {
    return "library.proto";
  }

  @Override
  public String getProtoFile() throws UncheckedIOException {
    // Assumes that the file is located in a Jar's resources, we must provide the path to the
library.proto file
    return FileDescriptorSource.getResourceAsString(getClass(), "/" + getProtoFileName());
  }

  @Override
  public void registerSchema(SerializationContext serCtx) {
    serCtx.registerProtoFiles(FileDescriptorSource.fromString(getProtoFileName(),
getProtoFile()));
  }

  @Override
  public void registerMarshallers(SerializationContext serCtx) {
    serCtx.registerMarshaller(new AuthorMarshaller());
    serCtx.registerMarshaller(new BookMarshaller());
  }
}
```

第9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、Data Grid クラスターのすべてのノードに分散され、共有されるロックで、現在、特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。

9.1. インストール

クラスター化されたロックの使用を開始するには、Maven **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>
</dependency>
```

9.2. CLUSTEREDLOCK 設定

現在、**ClusteredLock** の 1 つのタイプがサポートされています。reentrant、NODE 所有権ロックです。

9.2.1. 所有権

- **NODE** は、**ClusteredLock** が定義されると、Data Grid クラスターのすべてのノードでこのロックを使用できます。所有権が NODE タイプである場合、ロックの所有者は、特定の時点でロックを取得する Data Grid ノードになります。つまり、**ClusteredCacheManager** を使用して **ClusteredLock** インスタンスを取得するたびに、このインスタンスは各 Data Grid ノードの同じインスタンスになります。このロックは、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、ノード内のすべてのスレッドが特定のタイミングでロックを解放できることです。

- **INSTANCE** - まだサポートされていません

ClusteredLock が定義されると、Data Grid クラスターのすべてのノードでこのロックを使用できます。所有権が INSTANCE タイプである場合、ロックの所有者は、**ClusteredLockManager.get ("lockName")** が呼び出される際に取得した実際のインスタンスになります。

これは、**ClusteredCacheManager** を使用して **ClusteredLock** インスタンスを取得するたびに、このインスタンスは新しいインスタンスになります。このロックを使用して、Data Grid ノード間でコードと各 Data Grid ノード間でコードを同期します。このロックの利点は、ロックを呼び出したインスタンスのみがロックを解放できることです。

9.2.2. 再入可能性

ClusteredLock がリエントラントに構成されている場合、ロックの所有者は、ロックを保持している間、何度でも連続してロックを再取得することができます。

現在、非リエントラントロックのみサポートされています。つまり、2 つの連続した **lock** 呼び出しが同じ所有者に対して送信されると、最初の呼び出しは利用可能な場合はロックを取得し、2 番目の呼び出しはブロックされます。

9.3. CLUSTEREDLOCKMANAGER インターフェイス

ClusteredLockManager インターフェースは、**実験的とマークされて** おり、ロックを定義、取得、削除するためのエントリポイントです。**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。ロック状態を保存するために必要な内部キャッシュを起動します。

ClusteredLockManager の取得は、以下の例のように **EmbeddedClusteredLockManagerFactory.from(EmbeddedCacheManager)** を呼び出すのと同じくらい簡単です。

```
// 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** : 指定された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** の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)** : 名前を指定して **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、所有権 レベルセクションを参照してください。

現在、サポートされる唯一の所有権レベルは NODE のみです。

- **ClusteredLockConfiguration getConfiguration(String name)** :

ClusteredLock の設定を返します（存在する場合）。

- **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)** : そのような場合は **ClusteredLock** を削除します。
- **CompletableFuture<Boolean> forceRelease(String name)** : Releases - または unlocks - 存在

する場合は、特定の時間に保持するユーザーに関係なく、**ClusteredLock**。このメソッドを呼び出すと、並行処理の問題が発生する可能性があるため、**例外的な状況** で使用する必要があります。

9.4. CLUSTEREDLOCK インターフェイス

実験的としてマーク されている **ClusteredLock** インターフェイスは、クラスター化されたロックを実装するインターフェイスです。

```
@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、ロックが取得されるまでブロックされます。現在、**ロックリクエストが失敗する最大時間が指定されていないため**、これによりスレッドが枯渇する可能性があります。
- **tryLock** は、呼び出し時に解放されている場合にのみロックを取得し、その場合は **true** を返します。このメソッドは、ロックの取得をブロック（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、このメソッドは **true** で即座に返します。ロックが利用できない場合、呼び出しは以下になるまで待機します。
  - ロックが取得される
  - 指定された待機時間の経過時間

時間がゼロ以下の場合、メソッドはまったく待機しません。

- **unlock**

ロックを解放します。ロックの所有者のみがロックを解放できます。

- **isLocked** がロックされている場合は **true**、ロックがリリースされると **false** を返します。
- **isLockedByMe** ロックが呼び出し元によって所有されている場合は **true** を、ロックが他の誰かによって所有されているか、解放されている場合は **false** を返します。

9.4.1. 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
     }
  });
}
```

9.4.2. ClusteredLockManager の設定

次の属性を使用して、宣言的またはプログラム的に、ロックに異なるストラテジーを使用するように **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格納する各クラスターのノードの総数を定義します。デフォルト値は **-1** で、値をすべてのノードに複製します。

reliability

クラスターがパーティションに分割したり、複数のノードがクラスターから離れる場合にクラスター化されたロックがどのように動作するかを制御します。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** の宣言型設定の例です。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns="urn:infinispan:config:10.1">
   ...
   <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:10.1"
                  num-owners = "3"
                  reliability="AVAILABLE">
         <clustered-lock name="lock1" />
         <clustered-lock name="lock2" />
      </clustered-locks>
   </cache-container>
   ...
</infinispan>
```

第10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、Data Grid クラスターのすべてのノードで分散され、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。

10.1. 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>
</dependency>
```

このカウンターは、本書で後述する **CounterManager** インターフェイスを介して、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、起動時に Data Grid 設定ファイルに設定されたカウンターが作成します。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

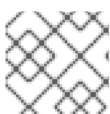
```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan>
    <cache-container ...>
        <!-- if needed to persist counter, global state needs to be configured -->
        <global-state>
            ...
        </global-state>
        <!-- your caches configuration goes here -->
         <counters xmlns="urn:infinispan:config:counters:10.1" 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.VOLATILE);
builder.addStrongCounter().name("c3").initialValue(3).upperBound(5).storage(Storage.PERSISTENT)
;
builder.addStrongCounter().name("c4").initialValue(4).lowerBound(0).upperBound(10).storage(Storag
e.VOLATILE);
builder.addWeakCounter().name("c5").initialValue(5).concurrencyLevel(1).storage(Storage.PERSIST
ENT);
```

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(Storage.
PERSISTENT).build());
manager.defineCounter("c2",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(2).lowerBound(0).stora
ge(Storage.VOLATILE).build());
manager.defineCounter("c3",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(3).upperBound(5).stor
age(Storage.PERSISTENT).build());
manager.defineCounter("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(4).lowerBound(0).uppe
rBound(10).storage(Storage.VOLATILE).build());
manager.defineCounter("c2",
CounterConfiguration.builder(CounterType.WEAK).initialValue(5).concurrencyLevel(1).storage(Storag
e.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", ...);
}
```

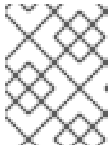
クラスターごとの属性:

- **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:

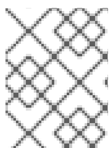
- **initial-value** [common]: カウンターの初期値を設定します。デフォルトは **0** (ゼロ) です。
- **storage** [common]: クラスターのシャットダウンおよび再起動時のカウンターの動作を設定します。デフォルト値は **VOLATILE** で、有効な値は次のとおりです。
    - **VOLATILE**: カウンターの値はメモリーでのみ利用できます。この値は、クラスターがシャットダウンすると失われます。
    - **PERSISTENT**: カウンターの値はプライベートおよびローカル永続ストアに保存されます。この値は、クラスタ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。

10.1.1. 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。

10.2. COUNTERMANAGER インターフェイス

CounterManager インターフェイスは、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);
```

Hot Rod クライアントの場合、**CounterManager** は RemoteCacheManager に登録されており、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);
```

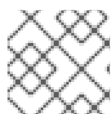
10.2.1. CounterManager を介したカウンターの削除

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。

10.3. 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

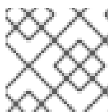
ます。各インターフェイスには特定のインターフェイスがありますが、ロジック (つまり各操作により **CompletableFuture** が返される) を共有しているため、更新イベントが返され、初期値にリセットできます。

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イ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。

10.3.1. StrongCounter インターフェイス: 一貫性または境界が明確になります。

strong カウンターは、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、必要な一貫性を提供します。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");
```

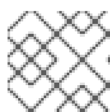
警告

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 インターフェイスでは、以下のメソッドを追加します。

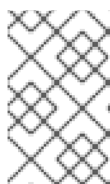
```
default CompletableFuture<Long> incrementAndGet() {
   return addAndGet(1L);
}

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。

10.3.1.1. 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** を出力します。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();
```

10.3.1.2. 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)

10.3.1.3. 使用例

```
StrongCounter counter = counterManager.getStrongCounter("unbounded_counter");

// 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);
```

10.3.2. WeakCounter インターフェイス: 速度が必要な場合

WeakCounter は、カウンターの値を 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);
```

10.3.2.1. weak カウンターインターフェイス

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。

10.3.2.2. 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。

10.3.2.3. 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());

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());
```

10.4. 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、これを以下の方法で登録できます。

```
<T extends CounterListener> Handle<T> addListener(T listener);
```

–

CounterListener には以下のインターフェイスがあります。

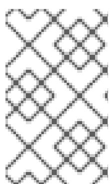
```
public interface CounterListener {
   void onUpdate(CounterEvent entry);
}
```

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イスがあります。

```
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イスがあります。

```
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11章 ロックおよび同時実行

Data Grid は、マルチバージョン同時実行制御 (MVCC) を利用します (MVCC)。これは、リレーショナルデータベースや他のデータストアでよく使用され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

11.1. 実装の詳細のロック

Data Grid の MVCC 実装では、ロックと同期が最小限に抑えられており、可能な限り compare-and-swap などのロックフリー技術やロックフリーのデータ構造などに重点を置いています。これにより、マルチ CPU 環境とマルチコア環境の最適化に役立ちます。

特に、Data Grid の MVCC 実装はリーダーに対して高度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。

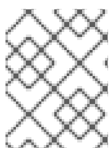
同時読み取りを可能にするため、ライターはエントリーを **MVCCEntry** でラップして、変更する予定のエントリーのコピーを作成します。このコピーは、同時リーダーが部分的に変更された状態を認識できないようにします。書き込みが完了したら、**MVCCEntry.commit()** はデータコンテナーへの変更をフラッシュし、後続のリーダーに変更内容が反映されます。

11.1.1. 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。

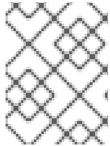
11.1.1.1. 非トランザクションキャッシュ

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、プライマリー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

操作がプライマリー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。

11.1.2. 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、トランザクションのロックを参照してください。

11.1.3. 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、分離レベルを参照してください。

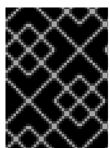
11.1.4. LockManager

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。

11.1.5. 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ー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で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、以下を実行します。

```
new ConfigurationBuilder().locking().useLockStriping(false|true);
```

11.1.6. 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。このパラメーターは、Data Grid でもまったく同じ方法で使用されているため、同時実行レベルの詳細については、JDK **ConcurrentHashMap** Javadocs を参照してください。

設定例:

```
<locking concurrency-level="32"/>
```

または、以下を実行します。

```
new ConfigurationBuilder().locking().concurrencyLevel(32);
```

11.1.7. 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、以下を実行します。

```
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10000);
//alternatively
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10, TimeUnit.SECONDS);
```

11.1.8. 一貫性

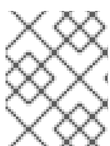
(すべての所有者がロックされているのとは対照的に) 単一の所有者がロックされるという事実により、次の一貫性の保証が失われることはありません。キー **K** がノード **{A, B}** に対してハッシュ化され、トランザクション **TX1** が、たとえば、**A** 上の **K** のロックを取得したとします。別のトランザクション **TX2** が **B** (またはその他のノード) 上で開始され、**TX2** が **K** のロックを試みる場合、ロックがすでに **TX1** によって保持されているため、タイムアウトでロックに失敗します。理由は、キー **K** のロックがト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。

11.2. データのバージョン管理

Data Grid は、simple と external の 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。

external バージョン管理は、Data Grid を Hibernate で使用する場合など、Data Grid 内のデータバージョン管理の外部ソースをカプセル化するために使用され、そのデータバージョン情報をデータベースから直接取得します。

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12章 DATA GRID CDI エクステンションの使用

Data Grid は、CDI (Contexts and Dependency Injection) プログラミングモデルと統合し、以下を可能にするエクステンションを提供します。

- CDI Bean および Java EE コンポーネントにキャッシュを設定し、インジェクトします。
- キャッシュマネージャーを設定します。
- キャッシュおよびキャッシュマネージャーレベルのイベントを受信します。
- JCache アノテーションを使用してデータストレージおよび取得を制御します。

12.1. CDI 依存関係

以下の依存関係のいずれかで **pom.xml** を更新し、プロジェクトに Data Grid CDI エクステンションを追加します。

埋め込み (Library) モード

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-embedded</artifactId>
</dependency>
```

サーバーモード

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-remote</artifactId>
</dependency>
```

12.2. 組み込みキャッシュのインジェクト

組み込みキャッシュをインジェクトするために CDI Bean を設定します。

手順

1. キャッシュ修飾子アノテーションを作成します。

```
...
import javax.inject.Qualifier;

@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GreetingCache { (1)
}
```

(1) **@GreetingCache** 修飾子を作成します。

2. キャッシュ設定を定義するプロデューサーメソッドを追加します。

```
...
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("mygreetingcache") (1)
   @GreetingCache (2)
   @Produces
   public Configuration greetingCacheConfiguration() {
      return new ConfigurationBuilder()
               .memory()
                  .size(1000)
               .build();
   }
}
```

(1) インジェクトするキャッシュに名前を付けます。

(2) キャッシュ修飾子を追加します。

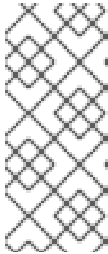
3. 必要に応じて、クラスター化されたキャッシュマネージャーを作成するプロデューサーメソッドを追加します。

```
...
package org.infinispan.configuration.global.GlobalConfigurationBuilder;

public class Config {

   @GreetingCache (1)
   @Produces
   @ApplicationScoped (2)
   public EmbeddedCacheManager defaultClusteredCacheManager() { (3)
     return new DefaultCacheManager(
       new GlobalConfigurationBuilder().transport().defaultTransport().build();
   }
}
```

(1) キャッシュ修飾子を追加します。

(2) アプリケーションに対して Bean を 1 度作成します。キャッシュマネージャーを作成するプロデューサーには、複数のキャッシュマネージャーを作成しないように、常に **@ApplicationScoped** アノテーションを含める必要があります。

(3) **@GreetingCache** 修飾子にバインドされた新規の **DefaultCacheManager** インスタンスを作成します。



注記

キャッシュマネージャーは、ヘビーウェイトオブジェクトです。アプリケーションで複数のキャッシュマネージャーを実行すると、パフォーマンスが低下する可能性があります。複数のキャッシュを挿入する場合は、各キャッシュの修飾子をキャッシュマネージャープロデューサーメソッドに追加するか、修飾子を追加しないでください。

4. **@GreetingCache** 修飾子をキャッシュインジェクションポイントに追加します。

```
...
import javax.inject.Inject;

public class GreetingService {

    @Inject @GreetingCache
    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;
    }
}
```

12.3. リモートキャッシュの注入

リモートキャッシュを注入するために CDI Bean を設定します。

手順

1. キャッシュ修飾子アノテーションを作成します。

```
@Remote("mygreetingcache") (1)
@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RemoteGreetingCache { (2)
}
```

(1) インジェクトするキャッシュに名前を付けます。

(2) **@RemoteGreetingCache** 修飾子を作成します。

2. キャッシュインジェクションポイントに **@RemoteGreetingCache** 修飾子を追加します。

```
public class GreetingService {

    @Inject @RemoteGreetingCache
```

```
    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;
    }
}
```

リモートキャッシュをインジェクトするためのヒント

- 修飾子を使用せずにリモートキャッシュをインジェクトできます。

```
...
@Inject
@Remote("greetingCache")
private RemoteCache<String, String> cache;
```

- 複数の Data Grid クラスターがある場合は、クラスターごとに個別のリモートキャッシュマネージャープロデューサーを作成できます。

```
...
import javax.enterprise.context.ApplicationScoped;

public class Config {

    @RemoteGreetingCache
    @Produces
    @ApplicationScoped (1)
    public ConfigurationBuilder builder = new ConfigurationBuilder(); (2)
        builder.addServer().host("localhost").port(11222);
        return new RemoteCacheManager(builder.build());
    }
}
```

1. アプリケーションに対して Bean を 1 度作成します。キャッシュマネージャーを作成するプロデューサーには、ヘビーウェイトオブジェクトである複数のキャッシュマネージャーが作成されないように、常に **@ApplicationScoped** アノテーションが含まれる必要があります。

2. **@RemoteGreetingCache** 修飾子にバインドされる新しい **RemoteCacheManager** インスタンスを作成します。

12.4. JCACHE キャッシングアノテーション

JCache アーティファクトがクラスパスにある場合、以下の JCache キャッシングアノテーションを CDI 管理 Bean で使用できます。

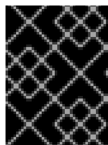
@CacheResult

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
キャッシュからすべてのエントリーを削除します。



重要

Target type: これらの JCache キャッシングアノテーションはメソッドでのみ使用できます。

JCache キャッシュアノテーションを使用するには、アプリケーションの **beans.xml** ファイルでインターセプターを宣言します。

管理対象環境 (アプリケーションサーバー)

```
<?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>
```

管理対象外環境 (スタンドアロン)

```
<?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>
```

JCache キャッシングアノテーションの例

以下の例は、**@CacheResult** アノテーションが **GreetingService.greet()** メソッドの結果をキャッシュする方法を示しています。

```
import javax.cache.interceptor.CacheResult;

public class GreetingService {

    @CacheResult
    public String greet(String user) {
        return "Hello" + user;
    }
}
```

JCache アノテーションを使用すると、デフォルトのキャッシュは、アノテーションが付けられたメソッドの完全修飾名をパラメータータイプで使用します (例: **org.infinispan.example.GreetingService.greet(java.lang.String)**)。

デフォルト以外のキャッシュを使用するには、以下の例のように、**cacheName** 属性を使用してキャッシュ名を指定します。

```
@CacheResult(cacheName = "greeting-cache")
```

12.5. キャッシュおよびキャッシュマネージャーイベントの受信

CDI イベントを使用して、キャッシュおよびキャッシュマネージャーレベルのイベントを受信します。

- 以下の例のように **@Observes** アノテーションを使用します。

```
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) {
        ...
    }
}
```

第13章 DATA GRID トランザクション

Data Grid は、JTA 準拠のトランザクションを使用し、参加するように設定できます。

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)。

すべてのキャッシュ操作で 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 インターフェイスの実装のクラス名を使用してキャッシュを設定することで行います。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。

Data Grid には複数のトランザクションマネージャールックアップクラスが同梱されます。

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、埋め込みモードのみ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、または JBoss AS 7 以前、および WildFly 8、9、10 で Data Grid を実行している場合、トランザクションマネージャーのデフォルトとしてこれを選択します。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 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としてこれを選択します。
- 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();
```

13.1. 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"/>
<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"/>
```

プログラムを使用する場合

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.locking()
   .isolationLevel(IsolationLevel.READ_COMMITTED);
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__");
```

- **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。
- **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** - プロトコルが使用するように設定します。デフォルトは **DEFAULT**です。使用できる値は次のとおりです。
  - **DEFAULT** - 従来の 2 フェーズコミットプロトコルを使用します。以下で説明します。
  - **TOTAL_ORDER**: **トランスポート** によって保証された合計順序を使用してトランザクションをコミットします。詳細は Total Order based commit protocol のセクションを参照してください。
- **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。

2 フェーズコミット (2PC) が Data Grid に実装される方法、およびロックが取得される方法についての詳細は、以下のセクションを参照してください。設定の詳細については、設定リファレンス を参照してください。

13.2. 分離レベル

Data Grid は、READ_COMMITTED および 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、Data Grid のコンテキストの **READ_COMMITTED** および **REPEATABLE_READ** の違いを理解する上で役立つ詳細な例です。**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()
```

13.3. トランザクションのロック

13.3.1. 悲観的なトランザクションキャッシュ

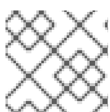
ロック取得の観点では、悲観的トランザクションはキーの書き込み時にキーのロックを取得します。

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。

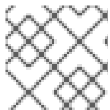
13.3.2. 楽観的トランザクションキャッシュ

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。

13.3.3. 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。

13.4. スキューの書き込み

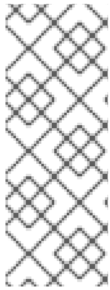
書き込みスキューは、2 つのトランザクションが独立して同時に同じキーの読み取りと書き込みを行うときに発生します。書き込みスキューの結果、両方のトランザクションは同じキーに対して更新を正常にコミットしますが、値は異なります。

Data Grid は、書き込みスキューチェックを自動的に実行し、楽観的トランザクションで **REPEATABLE_READ** 分離レベルのデータの一貫性を確保します。これにより、Data Grid はトランザクションの 1 つを検出し、ロールバックできます。

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。

13.4.1. 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

悲観的トランザクションでの書き込みスキューを回避するには、**Flag.FORCE_WRITE_LOCK** で読み取り時に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
```

13.5. 例外への対処

CacheException (またはそのサブクラス) が 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出力される場合、トランザクションは自動的にロールバックに対してマークされます。

13.6. 同期の登録

デフォルトでは、Data Grid は XAResource を介して、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。トランザクションの参加者として Data Grid が必要ではなく、ライフサイクル (準備、完了) によってのみ通知される状況があります (例: Data Grid が Hibernate で 2 次レベルキャッシュとして使用される場合など)。

Data Grid は、同期 を介したトランザクションのエンリストを許可します。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)。つまり、Hibernate 2 次キャッシュ: Data Grid がコミット時よりも **XAResource** として **TransactionManager** に登録する場合、**TransactionManager** は 2 つの **XAResource** (キャッ

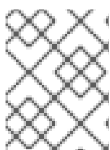
シュとデータベース) を認識し、この最適化を行いません。2 つのリソース間で調整する必要があるため、tx ログをディスクに書き込む必要があります。一方、Data Grid を **Synchronization** として登録すると、**TransactionManager** はディスクへのログの書き込みを省略します (パフォーマンスが向上)。

13.7. 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 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。両方の口座が Data Grid 内に保存されている場合は、バッチ処理を使用できます。ある口座がデータベースにあり、もう 1 つの口座が 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**。

13.7.1. 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
```

13.7.2. 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。

13.8. 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。

13.8.1. 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たアカウントから Data Grid に保管された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 (第 2) フェーズでは、データベースは、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け取る前に、Data Grid の変更を問題なく適用します。この時点では、システムが一貫性のない状態です。お金は外部データベースのアロ座から取得されますが、まだ Data Grid には表示されません (ロックは 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リリースされます)。リカバリーはこの状況に対応することで、データベースと Data Grid の両方のデータが一貫した状態で終了します。

13.8.2. 仕組み

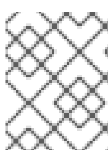
リカバリーはトランザクションマネージャーによって調整されます。トランザクションマネージャーは Data Grid と連携して、手動による介入が必要な未確定のトランザクションのリストを決定し、システム管理者に (電子メール、ログアラートなどを介して) 通知します。このプロセスはトランザクションマネージャーに固有のものですが、通常トランザクションマネージャーで設定が必要になります。

未確定のトランザクション ID を把握すると、システム管理者は 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できるようになりました。Data Grid は、この JMX ツールを提供します。これは、トランザクションのリカバリーおよび調整セクション で広範囲に説明されています。

13.8.3. リカバリーの設定

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 Data Grid と動作しないため、インダウト状態のトランザクションを決定できません。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。

13.8.3.1. 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。

13.8.4. リカバリーキャッシュ

未確定のトランザクションを追跡し、それらに応答できるようにするために、Data Grid は将来の使用の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設定できます。このキャッシュは、**recovery-cache** 設定属性を介して指定できます。指定のない場合は、Data Grid がローカルキャッシュを設定します。

リカバリーが有効になっているすべての 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は可能です (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、キャッシュ自体が使用す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。

13.8.5. 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。Data Grid **XAResource** の以下の API への参照を取得するには、以下を行います。

```
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。

13.8.6. 調整

トランザクションマネージャーは、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。

- **ステップ 1**: システム管理者は、JMX を介して Data Grid サーバーに接続し、未確定のトランザクションを一覧表示します。以下のイメージは、未確定のトランザクションを持つ Data Grid ノードに接続する JConsole を示しています。

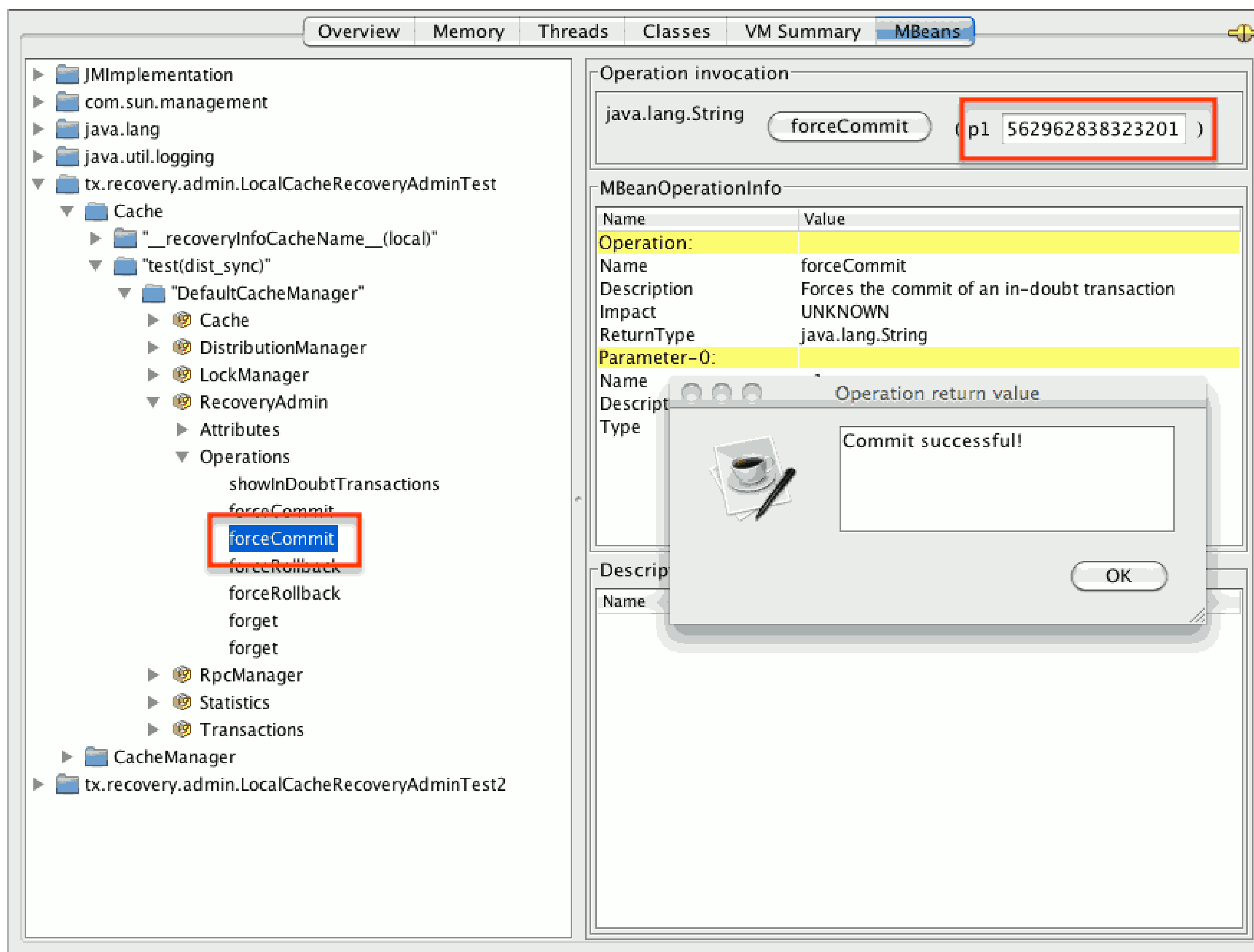
図13.1 未確定のトランザクションの表示

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。

- **ステップ 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 を数字で表した Data Grid 内部 ID に視覚的にマッピングします。XID(バイトアレイ) は、JMX ツール (JConsole など) に渡して Data Grid 側で再アセンブルされるため、このステップが必要です。

- **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13.2 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。

13.8.6.1. 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。

13.8.7. 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。

13.9. TOTAL ORDER ベースのコミットプロトコル

Total Order ベースのプロトコルは、マルチマスタースキーム (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 Data Grid に実装された (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) です。このコミットプロトコルは、メッセージの完全な順序付けされた配信の概念に依存します。これは、メッセージの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、それらを受信するノードによって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. 分散のデッドロックを軽減します。

このアプローチの弱点は、その実装は、トランザクションとその変更を提供するノードごとに単一のスレッドに依存し、**Transport** でメッセージを完全に順序付けする若干のコストです。

したがって、このプロトコルは **競合が高い** シナリオで最高のパフォーマンスを提供します。これは、シングルフェーズコミットを活用でき、配信スレッドはボトルネックではありません。

現在、Total Order ベースのプロトコルは、**replicated** および **distributed** 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。

13.9.1. 概要

Total Order ベースのコミットプロトコルは、トランザクションが Data Grid によってコミットされる方法と、分離レベルと書き込みスキューが動作に影響する方法にのみ影響します。

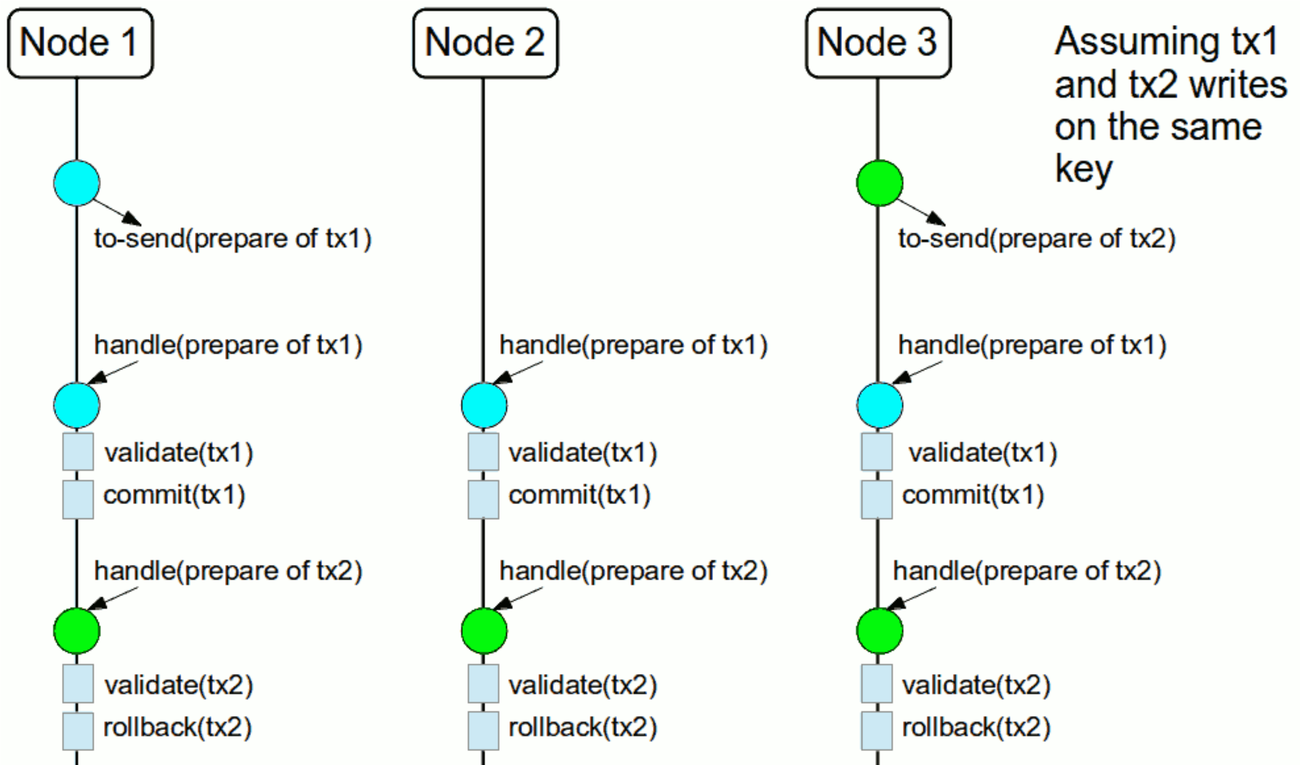
書き込みスキューが無効になっていると、トランザクションは単一のフェーズでコミット/ロールバックできます。データの整合性は **Transport** によって保証され、キーのすべての所有者が同じ順序で設定された同じトランザクションを提供します。

一方、書き込みスキューを有効にすると、プロトコルは適合し、安全であれば 1 フェーズコミットを使用します。**XaResource** enlistment では、**TransactionManager** が 1 つのフェーズでコミットを要求し (最後のリソースコミットの最適化)、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に 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チェック検証を実行するため、分散モードではこの最適化は安全ではありません。**Synchronization** enlistment では、Data Grid が唯一のリソースエンリストされたリソース (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。

13.9.1.1. 1 つのフェーズでのコミット

トランザクションが終了すると、Data Grid はトランザクション (およびその変更) を合計順に送信します。これにより、関連するすべての Data Grid ノードですべてのトランザクションが同じ順序で配信されます。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) で決定的な書き込みスキューチェックが実行され、同じ結果 (トランザクションのコミットまたはロールバック) が発生します。

図13.3 1 フェーズコミット

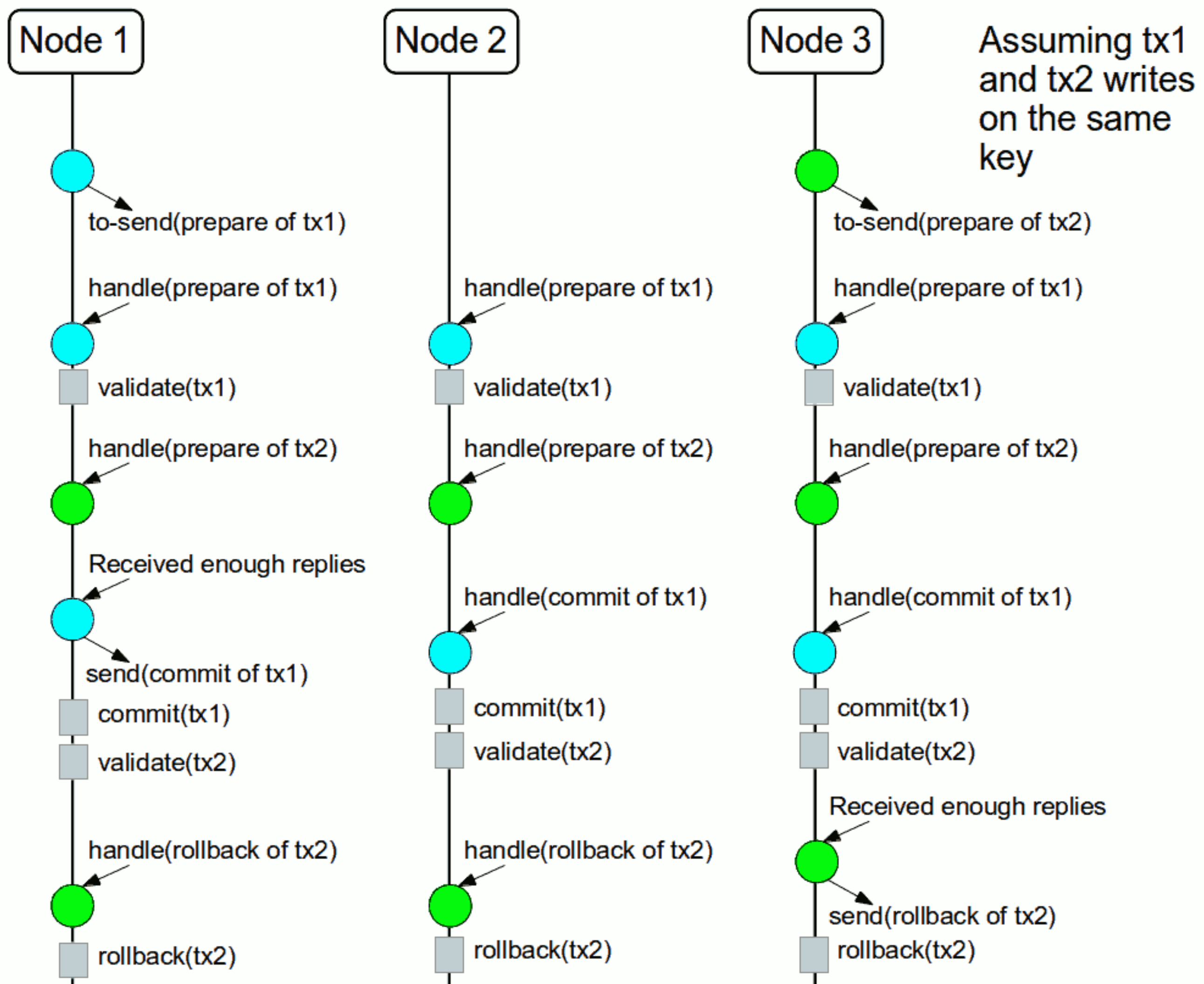


上の図は、3 つのノードがある高レベルの例を示しています。**Node1** と **Node3** はそれぞれ 1 つのトランザクションを実行しており、両方のトランザクションが同じキーに書き込むことを想定できます。より興味深いものにするために、図の最初の色付きの円で表される、両方のノードが同時にコミットしようとすると仮定します。**青い** 円はトランザクション **tx1**、**緑** はトランザクション **tx2** を表しています。どちらのノードも、トランザクションの変更とともに、合計順序でリモート呼び出しを実行します(**to-send**)。この時点で、すべてのノードが同じ配信順序で同意します (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** についても同じステップが実行されますが、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。

13.9.1.2. 2 つのフェーズでのコミット

最初のフェーズでは、合計順に変更を送信し、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は、発信者に返されます。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** に正の応答が付与されます。一方、負の応答を受信すると、**TransactionManager** に負の応答が返されます。最後に、トランザクションは **TransactionManager** リクエストに応じて 2 番目のフェーズでコミットまたは中止されます。

図13.4 2 フェーズコミット



上の図は、最初の図で説明した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します。tx1 が配信されると検証が実行され、**TransactionManager** に応答します。次に、**TransactionManager** が tx1 の 2 番目のフェーズをリクエストする前に、 tx2 が配信されると仮定します。この場合、tx2 はキューに入れられ、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）を要求し、すべてのノードが tx2 の検証を自由に実行できます。

13.9.1.3. トランザクションリカバリー

現在、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。

13.9.1.4. 状態遷移

簡略化のため、Total Order ベース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ています。主な相違点は以下のとおりです。

1. 状態遷移中にトランザクションを配信します。

2. 状態遷移制御メッセージ(**CacheTopologyControlCommand**)は、合計順序で送信されます。

これにより、状態遷移と、すべてのノードであるトランザクション配信間の同期が可能になります。ただし、新しい参加者に関連する新しい合計順序を見つけるには、トランザクションは状態遷移の開始前に送信され（つまり、状態遷移の開始後に送信される）再送信する必要があります。

図13.5 トランザクション中のノード参加

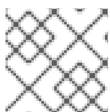
上の図は、ノードの参加 を示しています。シナリオでは、**tx2** は **topologyId=1** で送信されますが、受信時に **topologyId=2 にあります**。そのため、トランザクションは新規ノードに関連する再送信されます。

13.9.2. 設定

キャッシュで total order を使用するには、**jgroups.xml** 設定ファイルに **TOA** プロトコルを追加する必要があります。

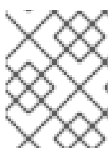
jgroups.xml

```
<tom.TOA />
```



注記

詳細は JGroups Manual を参照してください。



注記

JGroups が total order を保証する方法の詳細は、link::http://jgroups.org/manual/index.html#TOA[TOA manual] を確認してください。

また、トランザクション設定 に示すように、**<transaction>** 要素に **protocol=TOTAL_ORDER**を設定する必要があります。

13.9.3. いつ使用するか?

Total order は、書き込み集約型およびコンテンツ化されたワークロードで使用される利点を示します。ロックキーの競合を回避します。

第14章 インデックスとクエリー

14.1. 概要

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ている Protocol Buffers を介してエンコードされた Java Pojo またはオブジェクトのインデックス作成と検索をサポートします。

クエリーはライブラリーモードとクライアント/サーバーモード(Java、C#、Node.js、その他のクライアントの場合) の両方で可能であり、DataGrid は ApacheLucene を使用してデータをインデックス化でき、広い範囲のデータ取得ユースケースをカバーするために効率的なフルテキスト対応の検索エンジンを提供します。

インデックス設定はスキーマ定義に依存し、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 クラスを使用でき、他の言語で書かれたリモートクライアントには protobuf スキーマを使用できます。protobuf で標準化することで、Data Grid は Java クライアントと 非 Java クライアントとの間の完全な相互運用性を可能にします。

インデックス付きクエリーとは別に、Data Grid はインデックス化されていないデータまたは部分的にインデックス化されたデータでクエリーを実行できます。

Search API の観点では、Data Grid には、文字列ベースである **Ickle** と呼ばれる独自のクエリー言語があり、完全なテキストクエリーのサポートが追加されました。Query DSL は、フルテキストが必要ない場合に埋め込みおよびリモート java クライアントの両方に使用できます。Java 埋め込みクライアント Data Grid は、Faceted や Spatial 検索などの高度な検索機能とは別に、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 を提供します。

最後に、Data Grid は 継続的なクエリー をサポートします。これは、他の API とは逆に動作します。クエリーの作成、クエリーの実行、結果の取得の代わりに、クライアントは、クラスター内のデータの変更に応じて継続的に評価されるクエリーを登録でき、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します。

14.2. 埋め込みクエリー

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使用できます。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 を使用できます。また、柔軟なインデックス設定により、レイテンシーを最小限に抑えることができます。

14.2.1. 簡単な例

books と呼ばれる Data Grid キャッシュに **Book** インスタンスを保存します。**Book** インスタンスはインデックス化されるため、キャッシュのインデックスを有効にし、Data Grid が インデックスを自動的に設定 できるようにします。

Data Grid の設定:

infinispan.xml

```
<infinispan>
   <cache-container>
      <transport cluster="infinispan-cluster"/>
      <distributed-cache name="books">
         <indexing index="PRIMARY_OWNER" 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 **Book** は、次の例のように定義されます。インデックスを作成するプロパティーを選択する必要があります。プロパティーごとに、Hibernate Search プロジェクトで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任意で選択できます。

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
}
```

Data Grid **Cache** に複数の **Book** インスタンスを保存したとすると、次の例のように、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 クエリーは、"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 によって提供され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() とは別に、結果をストリーミングするか、ページネーションを使用できます。

Lucene またはフルテキスト機能を必要としない検索で、ほとんどの場合集計と完全一致に関するものである場合は、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して、1 つ以上の述語で Lucene 構文を可能にします。

```
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();
```

14.2.2. インデックス化

Data Grid 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。インデックスを有効にすることは必須ではありませんが、インデックスを使用したクエリーのパフォーマンスは大幅に高くなります。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、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、キャッシュタイプとユースケースに応じてこの影響を最小限に抑えるようにしてください。

14.2.2.1. 設定

14.2.2.1.1. 一般的な形式

XML によるインデックス作成を有効にするには、**<indexing>** 要素と **index** (インデックスモード) をキャッシュ設定に追加し、オプションで追加のプロパティーを渡す必要があります。

```
<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>
```

プログラマティック

```
import org.infinispan.configuration.cache.*;

ConfigurationBuilder cacheCfg = ...
cacheCfg.indexing().index(Index.ALL)
     .addProperty("property name", "propery value")
```

14.2.2.1.2. Index names

index 要素内の各プロパティーの前には、**org.infinispan.sample.Car** というインデックス名が付けられます。**directory_provider** 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")
```

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、アノテーションの **name** 引数で上書きされない限り、インデックス名は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。**Airplane** エンティティーも **local-heap** に保存されます。他のエンティティーのインデックスは、**default** 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>
```

14.2.2.1.3. インデックス化されたエンティティーの指定

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し、管理できます。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宣言することが推奨されます（完全修飾クラス名で一覧表示します）。これは、xml を介して実行できます。

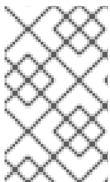
```
<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="default">
      <indexing index="ALL">
        <indexed-entities>
          <indexed-entity>com.acme.query.test.Car</indexed-entity>
          <indexed-entity>com.acme.query.test.Truck</indexed-entity>
```

```
            </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 要素に一覧表示されているクラス名は、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設定される拡張クラス名を使用し、これら 3 つのコンポーネントは ':' 文字 (例: "com.acme.my-module-with-entity-classes:my-slot:com.acme.query.test.Car") を使用する必要があります。エンティティークラスは参照モジュールに配置する必要があります。このモジュールは、サーバーの modules フォルダーにデプロイされた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たプレーン jar のいずれかです。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

サーバーの場合のみ、extended クラス名の使用要件に従い、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(Data Grid の内部クラスパスが使用されます)。

14.2.2.2. インデックスモード

Data Grid ノードは通常、ローカルとリモートの 2 つのソースからデータを受け取ります。ローカルは、同じ JVM のマップ API を使用してデータを操作するクライアントに変換されます。リモートデータは、レプリケーションまたはリバランス時に他の 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、クラスターがインデックス化されるビューのノードから定義します。

値:

- all: すべてのデータがインデックス化され、ローカル、およびリモートになります。
- 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作成元に関係なく、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: データがインデックス化されません。インデックスを設定しないのと同等です。

14.2.2.3. 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、Data Grid クエリーの central コンポーネントで、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、配布、および内部ライフサイクルを行います。各インデックスマネージャーは **Directory Provider** に関連付けられており、インデックスの物理ストレージを定義します。

インデックスの分散に関して、Data Grid は共有インデックスまたは非共有インデックスで設定できます。

14.2.2.4. 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から表示され、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 して結果を集約する必要がないことです。欠点は、Lucene がインデックスに同時に書き込むことができないため、ロック取得の連携を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が行う必要があることです。いずれの場合も、クラスター単位で 1 つの書き込みロックがあると、書き込みが非常に多い場合に一定レベルの競合が発生する可能性があります。

Data Grid は、Data Grid Directory Provider を利用する共有インデックスをサポートします。これは、インデックスを InfinispanIndexManager と呼ばれる個別のキャッシュセットに保存します。

14.2.2.4.1. インデックスモードの影響

共有インデックスは冗長なインデックス作成につながるため、**ALL** インデックスモードは使用しないでください。クラスタ全体で1つのインデックスが存在するため、Cache API を介して挿入されたときにインデックスが作成され、データグリッドが別のノードに複製したときに別のインデックスが作成されます。**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、非共有インデックス に関連付けます。

14.2.2.4.2. InfinispanIndexManager

このインデックスマネージャーは 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="PRIMARY_OWNER">
      <property
name="default.indexmanager">org.infinispan.query.indexmanager.InfinispanIndexManager</property
>
      <!-- Optional: tailor each cache used internally by the InfinispanIndexManager -->
      <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">
   <indexing index="NONE" />
   <!-- extra configuration -->
</distributed-cache>
```

インデックスは、デフォルトで **LuceneIndexesData**、**LuceneIndexesMetadata**、および **LuceneIndexesLocking** と呼ば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 ロックを保存するために使用されます。これは非常に小さなキャッシュであり、エンティティー (インデックス) ごとに 1 つのエントリーが含まれます。

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に関する情報を格納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配置されます。これは他の 2 つのインデックスよりもはるかに大きくなりますが、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なければなりません。

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。Data Grid は適切なデフォルトを選択します。それらを再定義する理由は、特定のシナリオにおけるパフォーマンスチューニングや、キャッシュストアを設定して永続化させる理由です。

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たときにインデックスが破損するのを回避するために、**InfinispanIndexManager** はクラスター内のマスター (JGroups コーディネーター) を内部で選択し、すべてのインデックスがこのマスターに機能します。

14.2.2.5. 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、各ノードにすべてのクラスターデータがある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、すべてのインデックスを保持することができ、クエリー時のネットワークレイテンシーがゼロで最適なクエリーパフォーマンスが得られます。もう 1 つの利点は、インデックスは各ノードにローカルであり、書き込み中に競合が発生するため、各ノードはクラスター全体ではなく、独自のインデックスロックの対象であることから、競合が少なくなります。

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、正しい検索結果を取得するために、リンク#query_clustered_query_api[broadcast] クエリーが必要になる場合があります。これにより、レイテンシーを追加できます。ただし、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し、ローカルインデックスを利用する最適な速度でクエリーが実行されます。

Data Grid には、**directory-based** と **near-real-time** の 2 つのインデックスマネージャーが非共有インデックスに適しています。ストレージでは、共有されていないインデックスは ram、filesystem、または Data Grid のローカルキャッシュに配置できます。

14.2.2.5.1. インデックスモードの影響

directory-based および **near-real-time** の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。これにより、インデックスディストリビューションが異なります。

REPL キャッシュは **ALL** インデックスモードと組み合わせて、各ノードのクラスター全体のインデックスの完全なコピーになります。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を効果的にローカルにすることができます。これは、REPL キャッシュをインデックス化するのに推奨されるモードであり、REPL キャッシュが検出されると auto-config によって選択されます。**ALL** モードは、DIST キャッシュでは使用できません。

REPL または DIST キャッシュと **LOCAL** インデックスモードを組み合わせると、各ノードが同じ JVM から挿入されたデータのみをインデックス化するため、インデックスの分散が不均一になります。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 または DIST キャッシュを **PRIMARY_OWNER** と組み合わせるには、ブロードキャストクエリーも必要です。**LOCAL** モードとは異なり、各ノードのインデックスには、キーが一貫したハッシュに従ってノードが所有するインデックス化されたエントリーが含まれ、ノード間でより均等に配分されたインデックスが作成されます。

14.2.2.5.2. 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。**directory-based** 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**、**filesystem**、およびクラスター化されていない **infinispan** ストレージをサポートします。

ファイルシステムストレージ

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る場合に使用されます。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。ローカルインデックスに推奨されるストレージです。インデックスはディスク上で永続化されますが、Lucene によってマップされたメモリーを取得するため、適切なクエリーパフォーマンスが提供されます。

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>
```

Data Grid は、キャッシュに存在するエンティティー（インデックス）ごとに **default.indexBase** の下に異なるフォルダーを作成します。

RAM ストレージ

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同時の状況では推奨されません。Ram に保存されてい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>
```

Data Grid ストレージ

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 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、たとえば、キャッシュストアを追加して、メモリー以外の別の場所でインデックスを永続化するなどして、他の Data Grid キャッシュと同様に設定できます。共有されていないインデックスで 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>
```

14.2.2.5.3. Near-real-time インデックスマネージャー

directory-based 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が少ないため、**directory-based** 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 ストレージと共に使用できます。異なるストレージタイプの設定は、directory-based のインデックスマネージャーと同じです。

ram を使用する例：

```
<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>
```

filesystem を使用した例:

```
<replicated-cache name="default">
   <indexing index="ALL">
      <property name="default.indexmanager">near-real-time</property>
   </indexing>
</replicated-cache>
```

14.2.2.6. 外部インデックス

Data Grid 自体によって管理されている共有インデックスと非共有インデックスを持つこと以外に、サードパーティーの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、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ます。

14.2.2.6.1. 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、一部の機能は利用できません。たとえば、**@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:

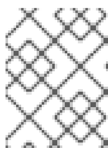
```
<indexing index="PRIMARY_OWNER">
   <property name="default.indexmanager">elasticsearch</property>
   <property name="default.elasticsearch.host">link:http://elasticHost:9200</property>
   <!-- other elasticsearch configurations -->
</indexing>
```

Data Grid は Elasticsearch を単一の共有インデックスとみなすため、インデックスモードは **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など、Elasticsearch 統合の詳細は、Hibernate Search マニュアル を参照してください。

14.2.2.7. 自動設定

属性 auto-config 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、他のプロセスと共有されず、ディスク上で永続化されるように設定されます。また、オブジェクトがインデックス化されてから、検索に使用可能な時点 (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="PRIMARY_OWNER" auto-config="true"/>
</local-cache>
```



注記

auto-config を介して追加されたプロパティーを再定義したり、新しいプロパティーを追加したりできるため、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、および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	排他モードでのインデックス化操作が可能で、Hibernate Search による書き込みの最適化が可能になります。
default.indexmanager	Near Real Time	Lucene の near real ti me機能を利用します。つまり、インデックス化されたオブジェクトは検索にすぐに利用できます。

プロパティー名	value	description
default.reader.strategy	shared	複数のクエリーでインデックスリーダーを再利用するため、再度開くのを防ぎます。

分散キャッシュの場合、auto-config は Data Grid 自体のインデックスを設定し、インデックス化操作がインデックスに書き込む単一のノードに送信されるマスター/スレーブメカニズムとして内部的に処理されます。

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	Data Grid に保存されているインデックス。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	排他モードでのインデックス化操作が可能で、Hibernate Search による書き込みの最適化が可能になります。
default.indexmanager	org.infinispan.query.indexmanager.InfinispanIndexManager	Data Grid クラスターの単一ノードへのインデックス書き込みを委譲します。
default.reader.strategy	shared	複数のクエリーでインデックスリーダーを再利用し、再度開くのを防ぎます。

14.2.2.8. 再インデックス化

場合によっては、Cache に保存されているデータから Lucene インデックスを再構築する必要がある場合があります。Analyzers はインデックスの記述方法に影響を与えるため、タイプでインデックスの定義を変更する場合や、一部の **Analyzer** パラメーターを変更する場合は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エラーによって破棄された場合は、インデックスを再構築する必要がある場合があります。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するには、グリッド内のすべてのデータを再処理する必要があるため、時間がかかる場合があります。

```
// Blocking execution
SearchManager searchManager = Search.getSearchManager(cache);
searchManager.getMassIndexer().start();

// Non blocking execution
CompletableFuture<Void> future = searchManager.getMassIndexer().startAsync();
```

ヒント

これは、**org.infinispan:type=Query,manager="{name-of-cache-manager}",cache="{name-of-cache}",component=MassIndexer** 配下に登録されている MassIndexer MBean で **start** JMX 操作としても利用できます。

14.2.2.9. マッピングエンティティー

Data Grid は、エンティティーレベルでインデックス作成の詳細な設定を定義するため Hibernate Search の 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。詳細なドキュメントは the Hibernate Search manual を参照してください。

14.2.2.9.1. @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付けすると、Data Grid は値を保存するために使用されるキーになります。すべての **@Indexed** オブジェクトの識別子は、値を保存するために使用されるキーになります。**@Transformable**、カスタム型、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。

14.2.2.9.2. @Transformable keys

各値のキーはインデックス化する必要があり、キーインスタンスを **String** で変換する必要があります。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。

アノテーションを使用したキートランスフォーマーの登録

キークラスに **org.infinispan.query.Transformable** のアノテーションを付け、カスタムトランスフォーマー実装が自動的に選択されます。

```
@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 ...
   }
}
```

キャッシュインデックス設定を介したキートランスフォーマーの登録

埋め込みおよびサーバー設定の両方で、**key-transformers** xml 要素を使用できます。

```
<replicated-cache name="test">
    <indexing index="ALL" auto-config="true">
        <key-transformers>
            <key-transformer key="com.mycompany.CustomKey"
transformer="com.mycompany.CustomTransformer"/>
        </key-transformers>
    </indexing>
</replicated-cache>
```

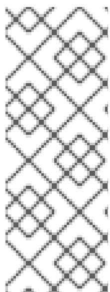
または、Java 設定 API (組み込みモード) を使用して同じ効果を得ることができます。

```
ConfigurationBuilder builder = ...
builder.indexing().autoConfig(true)
    .addKeyTransformer(CustomKey.class, CustomTransformer.class);
```

実行時のプログラムによるトランスフォーマーの登録

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も、キャッシュインデックス設定にトランスフォーマーを追加する必要もなく、代わりに **org.infinispan.query.spi.SearchManagerImplementor.registerKeyTransformer(Class<?>, Class<? extends Transformer>)** を呼び出すことで、実行時に **SearchManagerImplementor** に追加することができます。

```
org.infinispan.query.spi.SearchManagerImplementor manager =
Search.getSearchManager(cache).unwrap(SearchManagerImplementor.class);
manager.registerKeyTransformer(keyClass, keyTransformerClass);
```



注記

10.0 以降、このアプローチは非推奨になりました。これは、新しく起動したノードが初期状態遷移でキャッシュエントリーを受信し、必要なキートランスフォーマーがまだ登録されていないため (キャッシュが完全に起動した後にのみ登録可能)、それらをインデックス化できない可能性があるためです。他の利用可能なアプローチ (設定およびアノテーション) を使用してキートランスフォーマーを登録すると、望ましくない状況を回避できます。

14.2.2.9.3. 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。

次の例では、グリッドに格納され、クラスにアノテーションを付けなくても 2 つのプロパティーで検索可能にするオブジェクト **Author** をマップします。

```
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.NONE)
       .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;
```

14.2.3. API のクエリー

以下を使用して Data Grid にクエリーを実行できます。

- Lucene または Hibernate Search クエリー。Data Grid は、Lucene クエリーを生成する Hibernate Search DSL を公開します。Lucene クエリーを単一ノードで実行するか、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できます。
- フルテキスト拡張を含むカスタム文字列ベースのクエリー言語である Ickle クエリー。

14.2.3.1. Hibernate Search

インデックスを設定するために 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。

14.2.3.1.1. Lucene クエリーの実行

Lucene クエリーを直接実行するには、**CacheQuery** でこれを作成してラップするだけです。

```
import org.apache.lucene.search.Query;
import org.infinispan.query.CacheQuery;
```

```
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;


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);
```

14.2.3.1.2. 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 のマニュアル の関連のセクションを参照してください。

14.2.3.1.3. ファセット検索

Data Grid は、Hibernate Search **FacetManager** を使用して、ファセット検索 をサポートしています。

```
// 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");
```

上記のファセット検索は、年ごとにリリースされた 'bitcoin' に一致する数字ブックを返します。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}
```

ファセット検索の詳細は、Hibernate Search Faceting を参照してください。

14.2.3.1.4. 空間クエリー

Data Grid は Spatial Queries もサポートしているため、完全テキストを距離、地理、地理コーディネートに基づく制限と組み合わせることができます。

この例では、**@Latitude** および **@Longitude** とともに検索されるエンティティーで **@Spatial** アノテーションを使用することから開始します。

```
@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();
```

詳細については、Hibernate Search のマニュアル を参照してください。

14.2.3.1.5. 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、結果を取得してから呼び出し元に戻すことができます。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 と併用する場合に適しています。

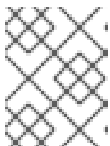
IndexedQueryMode.FETCH は呼び出し元でクエリーを実行します。クラスター全体のデータのすべてのインデックスがローカルで利用可能な場合、パフォーマンスが最適になります。それ以外の場合は、このクエリーモードにはリモートノードからインデックスデータの取得が含まれる場合があります。

IndexedQueryMode は、Ickle クエリーおよび Lucene クエリーでサポートされます(Query DSL ではサポートされません)。

以下に例を示します。

```
CacheQuery<Person> broadcastQuery = Search.getSearchManager(cache).getQuery(new
MatchAllDocsQuery(), IndexedQueryMode.BROADCAST);

List<Person> result = broadcastQuery.list();
```

14.2.3.2. Data Grid Query DSL



注記

Query DSL (QueryBuilder および関連インターフェイス)は非推奨となり、次のメジャーバージョンで削除されます。代わりに Ickle クエリーを使用してください。

Data Grid は、Lucene および Hibernate Search に依存しない独自のクエリー 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メカニズムからクエリー API を切り離すと、Lucene だけでなく、同じ統一されたクエリー API を使用できるように、今後も新しい代替エンジンを導入できます。インデックス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、この章で説明するすべてのインデックス関連側面は引き続き適用されます。

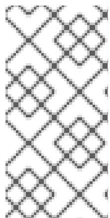
新しい API は、Lucene クエリーオブジェクトを構築する低レベルの詳細にユーザーを公開せず、リモートの Hot Rod クライアントで使用できるという利点があります。ただし、詳細を分解する前に、最初に前の例から **Book** エンティティーのクエリーを書き込む簡単な例を見てみましょう。

Data Grid のクエリー DSL を使用したクエリーの例

```
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** から取得した **QueryFactory** インスタンスを使用して作成されます。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、それ以外の場合は、複数のクエリーを並行して作成するために使用できるステートレスおよびスレッドセーフオブジェクトになります。

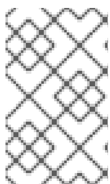
クエリーの作成は **from(Class entityType)** メソッドの呼び出しで始まります。このメソッドは、指定のキャッシュから特定されたエンティティークラスにターゲットとするクエリーを作成する **QueryBuilder** オブジェクトを返します。



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ーを実行したり、複数のエンティティータイプ (結合) を対象とするクエリー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によって指定された検索条件や設定を蓄積し、最終的に **QueryBuilder.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し、構築を完了させます。ステートフルオブジェクトであるため、(ネスト されたクエリーを除く) 複数のクエリーを同時に作成することはできませんが、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、同様の目的があるため、同じ名前になります。あいまいさを防ぐために、まもなく名前を変更することを検討しています。

クエリーの実行と結果のフェッチは、**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これを再実行する必要がある場合は、**QueryBuilder.build()** を呼び出して新しいインスタンスを取得する必要があります。

14.2.3.2.1. Operator のフィルタリング

クエリーの構築は、複数の条件を設定する階層的なプロセスで、この階層に従って説明するのが最適です。

クエリー条件の最も単純な形式は、ゼロ以上の引数を受け入れるフィルター Operator に応じて、エン

ティティー属性の値を制限することです。エンティティー属性は、利用可能なすべての Operator 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** で定義された各メソッドは、2 つの引数を持つ **between** と引数を持たない **isNull** を除いて、引数を受け取る演算子です。引数はクエリーの作成時に静的に評価されるため、SQL の相関サブクエリーと同様の機能を探している場合は、現在利用できません。

```
// a single query criterion
QueryBuilder qb = ...
qb.having("title").eq("Hibernate Search in Action");
```

表14.1 FilterConditionEndContext は、以下のフィルターリング Operator を公開します。

フィルター	引数	説明
in	コレクション値	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	オブジェクト値	左側のオペランドが、値として指定された値のリスト (固定) のいずれかと同じであ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として想定される) に指定の要素が含まれていることを確認します。
containsAll	コレクション値	左側の引数 (配列またはコレクションとして想定される) に、指定されたコレクションのすべての要素を任意の順序で含まれていることを確認します。
containsAll	オブジェクト値	左側の引数 (配列またはコレクションとして想定される) に、指定したすべての要素を任意の順序で含まれていることを確認します。
containsAny	コレクション値	左側の引数 (配列またはコレクションとして想定される) に、指定された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	オブジェクト値	左側の引数 (配列またはコレクションとして想定される) に指定の要素が含まれていることを確認します。

フィルター	引数	説明
isNull		左側の引数が null であることを確認します。
like	文字列のパターン	(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	左側の引数が指定の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	Object from、Object to	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。

クエリ構築には、適切な順序で行わなければならない、正確に **1 回** で完了しなければならない、2 回行うとエラーになる、といったメソッド呼び出しの多段階の連鎖が必要であることに注意することが重要です。以下の例は無効であり、各ケースによって基準が無視される (あいまいなケース) または例外が出力されます (より深刻なもの)。

```
// 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();
```

14.2.3.2.2. 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** 属性を参照するためのドット区切りの属性パスも受け付けますので、以下は有効なクエリーとなります。

```
// 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の既存のインデックス付き属性を参照する必要があります。複数のレベルの埋め込みが可能です。

14.2.3.2.3. 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。ブール値演算子のよく知られている Operator の優先順位ルールはここで適用されるため、構築中に 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、論理演算子の中で最も優先され、次の単純な属性条件にのみ適用される **not** 演算子で実現されます。

```
// 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();
```

14.2.3.2.4. ネストされた条件

論理演算子の優先順位の変更は、ネストされたフィルター条件で行います。論理演算子を使用すると、以前示される 2 つの単純な属性条件を接続できますが、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を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();
```

14.2.3.2.5. プロジェクション

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ーにエンティティー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。クエリー言語を使用すると、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。展開が使用される場合、**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 **List** (プロジェクト化された属性に対応する配列) 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();
```

14.2.3.2.6. ソート

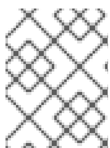
1 つ以上の属性または属性パスに基づいて結果の順序は、属性パスとソート方向を受け入れる **QueryBuilder.orderBy( )** メソッドで行われます。複数の並べ替え基準を指定すると、**orderBy** メソッドの呼び出し順序が優先順位を決定します。ただし、複数の並べ替え基準は、各属性の個別のソート操作のシーケンスではなく、指定された属性のタプルで動作します。

```
// 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();
```

14.2.3.2.7. ページネーション

QueryBuilder の **maxResults** プロパティーを設定することにより、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

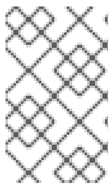
フェッチされる結果が **maxResults** に制限されている場合でも、**Query.getResultSize()** を呼び出すことで、一致する結果の合計数を見つけることができます。

14.2.3.2.8. グループ化およびアグリゲーション

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結果をグループ化し、各グループに分類される値のセットに集計関数を適用することにより、各グループからの結果の集計を構築する機能があります。グループ化および集計は、プロジェクションクエリーにのみ適用できます。サポートされる集約は 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、複数回呼び出すことができます。グループ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。グループ化フィールドのみを選択し、集計フィールドは選択しないクエリーは有効です。

例: ブックマークは作成者別にグループ化し、それらをカウントします。

```
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ーが許可されます。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。

14.2.3.2.9. 集約

avg、sum、count、max、min の集約関数をフィールドに適用できます。

- avg() - 一連の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** で表されます。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() - 見つかった最も大きな値を返します。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() - 数字のセット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14.2 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

フィールドタイプ	戻り値のタイプ
BigDecimal	BigDecimal

14.2.3.2.10. 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。フィルターリングは、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。グループ化操作の実行前に **groupBy()** メソッドを起動する前に定義されたフィルター条件はすべて、キャッシュエントリーに直接 (最終的な展開ではなく) キャッシュエントリーに適用されます。これらのフィルター条件は、照会されたエンティティータイプの任意のフィールドを参照し、グループ化ステージの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** メソッドの呼び出し後に定義されたフィルター条件はすべて、展開およびグループ化操作の結果が展開されます。このフィルター条件は、**groupBy** フィールドまたは集約された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ーリングすると、プロパティーに基づいてグループの数が減ります。通常のクエリーと同様にソートを指定することもできます。順序付け操作は、グループ化操作後に実行され、**groupBy** フィールドまたは集約されたフィールドのいずれかを参照できます。

14.2.3.2.11. 名前付きクエリーパラメーターの使用

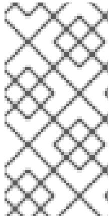
実行ごとに新しい Query オブジェクトを作成する代わりに、実行前に実際の値に置き換えることができる名前付きパラメーターをクエリーに含めることができます。これにより、クエリーを 1 度定義し、複数回効率的に実行できます。パラメーターは、Operator の右側でのみ使用でき、通常の定数値ではなく、**org.infinispan.query.dsl.Expression.param(String paramName)** メソッドによって生成されたオブジェクトを Operator に提供することで、クエリーの作成時に定義されます。パラメーターが定義されたら、以下の例に示すように **Query.setParameter(parameterName, value)** または **Query.setParameters(parameterMap)** のいずれか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プを指定して、複数のパラメーターを一度に設定することもできます。

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。

14.2.3.2.12. その他のクエリー DSL サンプル

Query DSL API の使用を調べる最善の方法は、テストスイートを確認することです。QueryDslConditionsTest は簡単な例です。

14.2.3.3. 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 は文字列ベースであり、以下の特徴があります。

- Java クラスをクエリーし、Protocol Buffers をサポートします。
- クエリーは単一のエンティティータイプをターゲットにすることができます。
- クエリーは、コレクションを含む埋め込みオブジェクトのプロパティーをフィルタリングできます。
- プロジェクション、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)** などの式の計算をサポートしません。
- 結合をサポートしません。
- サブクエリーをサポートしません。
- さまざまな Data Grid API でサポートされています。Query が受け入れるたびに、継続的なクエリーやリスナーのイベントフィルターに QueryBuilder が生成されます。

API を使用するには、まず QueryFactory をキャッシュに取得してから、.create() メソッドを呼び出し、クエリーで使用する文字列を渡します。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");
```

Ickle を使用する場合、フルテキスト演算子で使用されるすべてのフィールドは、**Indexed** および **Analysed** の両方である必要があります。

14.2.3.3.1. 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。

14.2.3.3.2. 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");
```

14.2.3.3.3. 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]");
```

14.2.3.3.4. フレーズクエリー

次の例に示すように、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'");
```

14.2.3.3.5. 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ーを実行するには、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 ");
```

14.2.3.3.6. ワイルドカードクエリー

単一文字およびマルチ文字のワイルドカード検索の両方を実行できます。

- 単一文字のワイルドカード検索は ? 文字で使用できます。
- マルチ文字のワイルドカード検索は * 文字で使用できます。

テキストを検索するか、テストするには、以下の単一文字のワイルドカード検索を使用します。

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where description : 'te?t'");
```

test、tests、または tester を検索するには、以下のマルチ文字のワイルドカード検索を使用します。

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where description :
'test*'");
```

14.2.3.3.7. 正規表現クエリー

正規表現クエリーは、/ 間のパターンを指定することで実行できます。Ickle は Lucene の正規表現構文を使用しているため、単語 moat または boat を検索するには、以下を使用できます。

```
Query regExpQuery = qf.create("from sample_library.Book  where title : /[mb]oat/");
```

14.2.3.3.8. クエリーのブースト

用語は、指定のクエリーにおける耐障害性を高めるために ^ を追加し、条件を強化できます。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");
```

14.2.3.4. 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。

- 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 18～25 歳の人を返す。

- $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)。

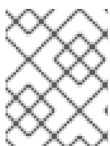
14.2.3.4.1. 連続クエリー実行

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** イベントによって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が更新され、**Update** event によって表されるクエリーの一致が継続されます。
- エントリーは、**Leave** イベントで表されるクエリーの一致を停止します。

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、すぐにクエリーに一致する結果の受信を開始します (上記のように **Join** イベントとして受信された)。さらに、通常は作成、変更、削除、または有効期限イベントを生成するキャッシュ操作の結果として、他のエントリーが **Join** イベントとしてクエリーの一致を開始したとき、または **Leave** イベントとしてクエリーの一致を停止したときに、後続の通知を受信します。更新されたキャッシュエントリーは、操作の前後でエントリーがクエリーフィルターに一致する場合、**Update** イベントを生成します。要約すると、リスナーが **Join**、**Update**、または **Leave** イベントを受信するかどうかを決定するために使用されるロジックは次のとおりです。

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. 古い値に対するクエリーが false と評価され、新しい値に対するクエリーが true と評価された場合、**Join** イベントが送信されます。
3. 古い値と新しい値の両方のクエリーが true と評価されると、**Update** イベントが送信されます。
4. 古い値に対するクエリーが true と評価され、新しい値に対するクエリーが false と評価された場合、**Leave** イベントが送信されます。
5. 古い値に対するクエリーが true と評価され、エントリーが削除または期限切れになると、**Leave** 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き、Query DSL の完全な機能を使用できます。

14.2.3.4.2. 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。これは、クエリー 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。キャッシュに関連付けられた ContinuousQuery オブジェクトは、リモートモードで実行されている場合は静的メソッド **org.infinispan.client.hotrod.Search.getContinuousQuery(RemoteCache<K, V> cache)** を、組み込みモードで実行されている場合は **org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache)** を呼び出して取得することができます。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 歳未満の Person インスタンスがキャッシュに追加されると、リスナーによって受信され、 **matches** マップに配置されます。これらのエントリーがキャッシュから削除されるか、年齢が 21 歳以上に変更されると、それらは **matches** から削除されます。

14.2.3.4.3. 継続的なクエリーの削除

クエリー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);
```

14.2.3.4.4. 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。この動作によりメモリーが不足し、クエリーが適切に設計されていない場合、**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ために、一致するエントリーの数と各一致のサイズの両方の観点から、各クエリーが必要な最小限の情報をキャプチャーし (プロジェクションを使用して興味深いプロパティーをキャプチャできます)、各 **ContinuousQueryListener** が受信したすべてのイベントをブロックせずにすばやく処理し、リッスンするキャッシュから一致する新しいイベントの生成につながるアクションの実行を回避するように設計されていることが強く推奨されます。

14.3. リモートクエリー

Java エンティティーのインデックス作成および埋め込みクライアントへの検索をサポートする以外に、Data Grid はリモート、言語に依存しないクエリーのサポートを導入しました。

この場合、2 つの主要な変更が必要でした。

- JVM 以外のクライアントは Apache Lucene の Java API を直接使用することはできないため、Data Grid は、現在 Hot Rod クライアントを実装するすべての言語に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、クライアントによるキャッシュに配置さ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。これらの構造はサーバーとクライアントの両方に認識されるため、構造化データをエンコードする一般的な方法を使用する必要があります。さらに、マルチ言語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、堅牢性、優れた複数言語のサポート、スキーマの進化のサポートにより、有線およびストレージの両方のエンコーディング形式として選択されました。

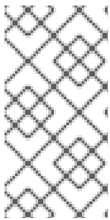
14.3.1. Protobuf でエンコードされたエンティティーの保存

保存されたエンティティーにインデックスを付けてクエリーできるリモートクライアントは、ProtoStream マーシャラーを使用して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。ただし、Protobuf エンティティーを保存して、プラットフォームの独立性の利点を得ることや、不要な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。

14.3.2. Protobuf でエンコードされたエントリーのインデックス化

前のセクションで説明したよう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。インデックスとさまざまなインデックス固有の設定は組み込みモードと同じで、Data Grid のクエリー で説明されています。

ただし、追加の設定手順が必要になります。埋め込みモードでは、インデックス作成メタデータは、エントリーのクラス上のさまざまな Hibernate Search アノテーションの存在を分析することによって Java リフレクションを介して取得されますが、エントリーが protobuf エンコードされている場合、これは明らかに不可能です。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で専用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。したがって、新しいスキーマの登録は、スキーマ名をキーとして、スキーマファイル自体を値として使用して、このキャッシュに対して **put** 操作を実行するのと同じくらい簡単です。または、CLI (cache-container=*:register-proto-schemas() オペレーション経由)、管理コンソール、または JMX 経由の **ProtobufMetadataManager** MBean を使用することもできます。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

キャッシュのインデックス作成が有効になっている場合でも、インデックスを作成する必要があるフィールドを指定するために **@Indexed** および **@Field** の protobuf スキーマドキュメント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フィールドにはインデックスが作成されません。

14.3.3. リモートクエリーの例

この例では、LibraryInitializerImpl の例を利用するようにクライアントを設定し、いくつかのデータをキャッシュに配置し、検索する方法を示します。以下の例では、必要な .proto ファイルを **___protobuf_metadata** キャッシュに登録することで、インデックスが有効になっている ことを前提としています。

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
   .host("10.1.2.3").port(11234)
   .addContextInitializers(new LibraryInitializerImpl());

RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());

Book book1 = new Book();
book1.setTitle("Infinispan 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!
```

クエリ作成で重要なのは、**org.infinispan.client.hotrod.Search.getQueryFactory()** メソッドを使用してリモートキャッシュ用の **QueryFactory** を取得することです。これ以降、クエリーの作成は、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。

14.3.4. 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。

14.3.4.1. デフォルトのアナライザー

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。

14.3.4.2. 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それらを参照します。

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。

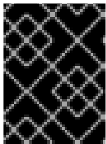
以下は、参照されたアナライザー定義の例になります。

```
/* @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;
}
```

14.3.4.3. 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** インターフェイ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 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

起動中に Data Grid サーバーでデプロイメントが利用可能である必要があります。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** インターフェイスの実装例です。

```
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. 以下のように、キャッシュコンテナー設定で **JAR** を指定します。

```
<cache-container name="mycache" default-cache="default">
  <modules>
    <module name="deployment.analyzers.jar"/>
  </modules>
...
```

14.4. 統計

以下のコードスニペットが示すように、クエリー Statistics 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 Data Grid によって登録されますが、統計はキャッシュレベルで統計収集が有効になっている場合にのみ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。Data Grid Query でそれらを使用することは、重複した MBean や予測不能な結果のみが発生するため、禁止されています。

14.5. パフォーマンスチューニング

14.5.1. SYNC モードでのバッチ書き込み

デフォルトでは、Index Managers は sync モードで動作します。つまり、データが 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。

したがって、インデックスを有効にして 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処理を活用してみてください。

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");
```

14.5.2. 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="PRIMARY_OWNER">
        <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>
```

14.5.3. 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。Data Grid でデフォルトで使用されるインデックスマネージャーは、すべてのクエリーの前にインデックスリーダーが更新されているかをチェックし、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。

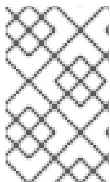
```
<distributed-cache name="default">
    <indexing index="PRIMARY_OWNER">
        <property
name="default.indexmanager">org.infinispan.query.affinity.InfinispanIndexManager</property>
        <property name="default.reader.strategy">async</property>
        <!-- refresh reader every 1s, default is 5s -->
        <property name="default.reader.async_refresh_period_ms">1000</property>
    </indexing>
</distributed-cache>
```

14.5.4. 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15章 GRID 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、この理由だけで、おそらく多くのユーザーが Data Grid を使用しています。ただし、Data Grid には、すぐには明らかにならない多くの利点があります。通常、Data Grid はマシンのクラスターで使用されるため、ユーザーのニーズのワークロードを実行するためにクラスター全体を利用するのに役立つ機能もあります。

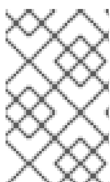


注記

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についてのみ説明します。リモートキャッシュを使用している場合は、リモートグリッドでのコードの実行に関する詳細を確認する必要があります。

15.1. 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 クラスターエグゼキューター 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、クラスター設定と非クラスター設定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。

15.1.1. 実行ノードのフィルターリング

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フィルターリングを使用するには、サーバーヒントを介してトポロジー対応のコンシステント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ー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ー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 フィルター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(...)
```

15.1.2. Timeout

クラスターエグゼキューターを使用すると、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、timeout メソッドを呼び出して、希望の期間を指定することでオーバーライドすることができます。

15.1.3. 単一ノードの提出

クラスターエグゼキューターは、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(...)
```

15.1.3.1. 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クラスターエグゼキューター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、クラスターエグゼキューター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、オーバーライドされた singleNodeSubmission メソッドを呼び出すことで有効になります。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。

15.1.4. 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、クラスターエグゼキューター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。

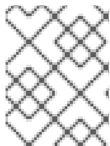
正方形の面積は Sa = 4r2 であり、円の面積は Ca=pi*r2 であることを思い出してください。2 つ目の式からの r2 を置き換えると、pi = 4 * Ca/S になります。ここで、正方形に非常に多くのダーツを射ることができると仮定して、射ったダーツの総数に対して円の中に入ったダーツの割合を取ると、Ca/Sa の値が近似します。pi = 4 * Ca/Sa であるため、pi の近似値を簡単に導き出すことができます。ダーツを多く撃つほど、より良い近似が得られます。以下の例では、10 億本のダーツを撃ちますが、それらを連続して撃つのではなく、Data Grid クラスター全体でダーツ射撃の作業を並列化します。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);
    }
}
```

第16章 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。Data Grid を使用すると、ユーザーは非常によく似た操作を実行できますが、標準の JRE API を使用して実行できます。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、エントリー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 メソッドまたは 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。

16.1. 一般的なストリーム操作

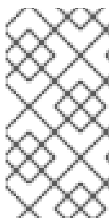
本セクションでは、使用している基礎となるキャッシュの種類に関係なく、さまざまなオプションを説明します。

16.2. キーのフィルターリング

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。これは、**CacheStream** で filterKeys メソッドを呼び出して実行できます。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 インターフェイスに慣れている人なら、なぜこの keyFilter ではなく getAll を使うのか不思議に思うかもしれません。エントリー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ーに必要とする場合、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。

16.3. セグメントベースのフィルターリング



注記

これは高度な機能で、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ーリングは、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タイミングで、データのサブセットで操作することができます。このセグメントは、**CacheStream** で filterKeySegments メソッドを呼び出してフィルターリングできます。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。

16.4. 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。Data Grid は、必要に応じてすべての変換をバックグラウンドで処理し、より興味深いすべてのオプション (つまり storeAsBinary およびキャッシュローダー) で機能します。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。

16.5. 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));
```

16.6. 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。

16.6.1. 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ーを追跡します。

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、CacheStream.disableRehashAware() を呼び出すことで実行できます。再ハッシュが発生しない場合、ほとんどの操作のパフォーマンスの向上は、完全に無視できます。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、使用するメモリーが少ない 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。

16.6.2. シリアル化

操作は他のノード全体に送信されるため、Data Grid マーシャリングでシリアライズできる必要があります。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。Data Grid は、さまざまな Stream 中間メソッドおよび端末メソッドをすべて上書きして、引数の Serializable バージョン (SerializableFunction、SerializablePredicate など) を取ります。これらのメソッドは CacheStream にあります。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、**Supplier<Collector>** の指定を可能にする 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。

```
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、代わりに **Supplier<Collector>** を取得するオーバーロードされた **collect** 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** インターフェイ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** インターフェイ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イ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)));
```

推奨され最も高性能な方法は、最小限のペイロードを提供するために、**AdvancedExternalizer** を使用することです。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> map = (Map<Object, String>) cache.entrySet().stream()
            .filter(new ContainsFilter("Jboss"))
            .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

 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;
      }
    }
```

16.7. 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は、**CacheStream** で sequentialDistribution または parallelDistribution メソッドを呼び出して実行できます。

16.8. 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<Map.Entry<Object, String>> stream = cache.entrySet().stream();
stream.timeout(1, TimeUnit.MINUTES);
```

詳細は、java ドキュメントの timeout を確認してください。

16.9. 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。**Consumer** が CacheAware インターフェイスを実装する場合は、**Consumer** インターフェイスからの accept メソッドの前に **injectCache** メソッドが呼び出されます。

16.10. 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。

16.11. キーベースの再ハッシュ対応 OPERATOR

iterator、spliterator、および forEach は、再ハッシュ認識が、セグメントだけでなくセグメントごとに処理されたキーを追跡する必要がある点で、他のターミナル operator 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ーから解放します。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ーは、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。

これらのオペレーションのバッチサイズは両方とも、**CacheStream** で distributedBatchSize メソッドを呼び出して設定できる値と同じ値で制御されます。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、メモリー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に要求されたすべての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、Data Grid は **iterator** または **spliterator** の端末操作をローカルで実行します。これにより、リモートの反復がリソース集約型であるためにパフォーマンスが最適化されます。

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、Data Grid は、**shared** および **write-behind** の両方が有効なキャッシュストアを使用する場合に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。

16.12. 中間オペレーションの例外

特別な例外を持つ中間オペレーションがあります。これらは、skip、peek、ソートされた 1 2. & distinct です。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。

16.13. 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。Data Grid ノードに key → sentence が保存されていると仮定します。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ー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ーを読み取った後にエントリー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。

第17章 JCACHE (JSR-107) API

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ー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ー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、メモリー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 Data Grid 実装で JCache を使用する方法と、API の主要な側面が説明されています。

17.1. 組み込みキャッシュの作成

前提条件

1. **cache-api** がクラスパスにあることを確認します。
2. 以下の依存関係を **pom.xml** に追加します。

```xml
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-jcache</artifactId>
</dependency>
```

手順

- 以下のように、デフォルトの JCache API 設定を使用する組み込みキャッシュを作成します。

```java
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>());
```

17.1.1. 組み込みキャッシュの設定

- 以下のように、カスタム Data Grid 設定の URI を **CachingProvider.getCacheManager(URI)** 呼び出しに渡します。

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

// Load configuration from an absolute filesystem path
URI uri = URI.create("file:///path/to/infinispan.xml");
// Load configuration from a classpath resource
// URI uri = this.getClass().getClassLoader().getResource("infinispan.xml").toURI();

// Create a cache manager using the above configuration
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager(uri,
this.getClass().getClassLoader(), null);
```

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。Data Grid はこれまで、シリアル化/マーシャリングを使用してこれを実装し、コピーを作成してキャッシュに保存しており、その方法は仕様に準拠しています。したがって、Data Grid でデフォルトの JCache 設定を使用する場合、保存されるデータはマーシャリング可能である必要があります。

または、(Data Grid または JDK Collections が機能するのと同じように) 参照によってデータを格納するように JCache を設定することもできます。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));
```

17.2. リモートキャッシュの作成

前提条件

1. **cache-api** がクラスパスにあることを確認します。
2. 以下の依存関係を **pom.xml** に追加します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-jcache-remote</artifactId>
</dependency>
```

手順

- リモート Data Grid サーバーでキャッシュを作成し、以下のようにデフォルトの JCache API 設定を使用します。

```
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").getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("remoteNamedCache",
    new MutableConfiguration<String, String>());
```

17.2.1. リモートキャッシュの設定

Hot Rod 設定ファイルには、リモートキャッシュのカスタマイズに使用できる **infinispan.client.hotrod.cache.*** プロパティーが含まれます。

- 以下のように、**hotrod-client.properties** ファイルの URI を **CachingProvider.getCacheManager(URI)** 呼び出しに渡します。

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

// Load configuration from an absolute filesystem path
URI uri = URI.create("file:///path/to/hotrod-client.properties");
// Load configuration from a classpath resource
// URI uri = this.getClass().getClassLoader().getResource("hotrod-client.properties").toURI();

// Retrieve the system wide cache manager via org.infinispan.jcache.remote.JCachingProvider
CacheManager cacheManager =
Caching.getCachingProvider("org.infinispan.jcache.remote.JCachingProvider")
    .getCacheManager(uri, this.getClass().getClassLoader(), null);
```

17.3. 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.getCachingProvider().getCacheManager();
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
cache.put("hello", "world"); // Notice that javax.cache.Cache.put(K) returns void!
String value = cache.get("hello"); // Returns "world"
```

標準の java.util.Map とは異なり、javax.cache.Cache には put 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、クラスター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。

17.4. 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)**
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

操作	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
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()**

17.5. JCACHE インスタンスのクラスターリング

Data Grid JCache 実装は仕様を越え、標準 API を使用してクラスターキャッシュを使用できるようになります。次のようにキャッシュを複製するように設定された 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18章 マルチマップキャッシュ

MutimapCache は、各キーに複数の値を含めることができる値にキーをマップする Data Grid キャッシュの一種です。

18.1. インストールと設定

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-multimap</artifactId>
</dependency>
```

18.2. 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、ほとんどの場合、ノンブロッキングです。詳細については、制限 を参照してください。

```
public interface MultimapCache<K, V> {

   CompletableFuture<Optional<CacheEntry<K, Collection<V>>>> getEntry(K key);

   CompletableFuture<Void> remove(SerializablePredicate<? super V> p);

   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();

}
```

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
```

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。

CompletableFuture<Boolean> remove(K key)

キーに関連付けられたエントリー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。

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。

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。

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。

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。

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。

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。

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。

18.3. 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設定する方法を参照してください。

18.3.1. 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);
```

18.4. 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動作しますが、以下のように現在のバージョンにはいくつかの制限があります。

18.4.1. 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。

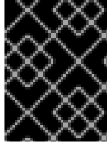
18.4.2. エビクション

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。

18.4.3. 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19章 カスタムインターセプター



重要

カスタムインターセプターは Data Grid で非推奨となり、今後のバージョンで削除されます。

カスタムインターセプターは、キャッシュへの変更に影響を与えたり、それに応答したりできるようにすることで、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。

19.1. 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>
```

19.2. プログラムによるカスタムインターセプターの追加

そのためには、**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。

19.3. 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、構築を有効にするために、パブリックの空のコンストラクターを宣言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すべてのプロパティーに対するセッターがあります。