Red Hat Data Grid 8.3

Data Grid 開発者ガイド

Data Grid API を使用して、デプロイメントをカスタマイズ、設定、および拡張します

Last Updated: 2023-12-06

Red Hat Data Grid 8.3 Data Grid 開発者ガイド

Data Grid API を使用して、デプロイメントをカスタマイズ、設定、および拡張します

法律上の通知

概要

Data GridAPI を使用して、デプロイメントをカスタマイズ、設定、および拡張します。

目次

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。

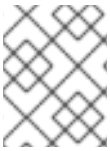
DATA GRID のドキュメント

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 8.3 ドキュメント
- Data Grid 8.3 コンポーネントの詳細
- Data Grid 8.3 でサポートされる設定
- Data Grid 8 機能のサポート
- Data Grid で非推奨の機能

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。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージ を参照してください。

第1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。

1.1. 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 8.3 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デプロイメントします。
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。

1.2. RED HAT MAVEN リポジトリーの追加

Red Hat GA リポジトリーを Maven ビルド環境に組み込み、Data Grid アーティファクトおよび依存関係を取得します。

手順

- Red Hat GA リポジトリーを Maven 設定ファイル (通常は **~/.m2/settings.xml**) に追加するか、プロジェクトの **pom.xml** ファイルに直接追加します。

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

1.3. 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>13.0.10.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2章 キャッシュマネージャー

Data Grid への主なエントリーポイントは **CacheManager** インターフェイスであり、以下を行うことができます。

- キャッシュを設定および取得します。
- クラスター化された Data Grid ノードを管理および監視します。
- クラスター全体でコードを実行します。

Data Grid をアプリケーションに埋め込む場合は、**EmbeddedCacheManager** を使用します。Data Grid をリモートサーバーとして実行する場合は、**RemoteCacheManager** を使用します。

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あるため、ほとんどの場合、JVM ごとに 1 つの **CacheManager** インスタンスのみをインスタンス化する必要があります。

```
EmbeddedCacheManager manager = new DefaultCacheManager(); 1
```

1 キャッシュのないローカルのクラスター化されていないキャッシュマネージャーを起動します。

キャッシュマネージャーにはライフサイクルがあり、デフォルトのコンストラクターは **start()** メソッドも呼び出します。コンストラクターのオーバーロードされたバージョンが利用可能ですが、それらは **CacheManager** を開始しません。ただし、キャッシュを作成する前に必ず **CacheManager** を起動する必要があります。

同様に、実行中の **CacheManager** が不要になった際に **stop()** を呼び出して、リソースを解放する必要があります。これにより、キャッシュマネージャーは制御するキャッシュを安全に停止することもできます。

2.1. 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。

2.2. クラスタリング情報

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

2.3. 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3章 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。

3.1. 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。

3.1.1. 特定のマップメソッドのパフォーマンスに関する懸念

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

3.1.2. Mortal および Immortal データ

単にエントリーを格納するだけでなく、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ックとして**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。

3.1.3. 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。

3.2. 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();
```

3.2.1. フラグ

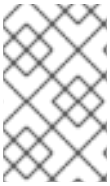
フラグは通常のキャッシュメソッドに適用され、特定のメソッドの動作を変更します。利用可能なフラグの一覧と、その効果については、Flag 列挙を参照してください。フラグは、AdvancedCache.withFlags() を使用して適用されます。このビルダーメソッドを使用して、キャッシュ呼び出しに任意の数のフラグを適用できます。次に例を示します。

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

3.3. 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。

3.3.1. キャッシュレベルの通知

キャッシュレベルのイベントはキャッシュごとに発生し、デフォルトでは、イベントが発生したノードでのみ発生します。分散キャッシュでは、これらのイベントは影響を受けるデータの所有者に対してのみ発生することに注意してください。キャッシュレベルのイベントの例としては、エントリーの追加、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。

3.3.1.1. クラスターリスナー

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

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

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

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

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

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

3.3.1.2. イベントのフィルタリングおよび変換

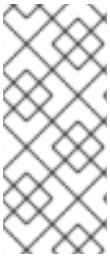
リスナーがインストールされているノードで適用可能なすべてのイベントがリスナーに発生します。KeyFilter(キーのフィルタリングのみを許可) または CacheEventFilter(キー、古い値、古いメタデータ、新しい値、新しいメタデータ、コマンドの再実行の有無、イベントがイベント (isPre など) の前であるか、およびコマンドタイプのフィルターに使用) を使用して、どのイベントが発生したかを動的にフィルターできます。

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。



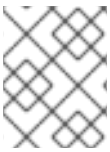
注記

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リングと変換が行われるためです。これにより、クラスター全体でイベントを複製する必要がない (フィルター)、またはペイロードを減らす (コンバーター) という利点があります。

3.3.1.3. 初期状態のイベント

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

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



注記

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

3.3.1.4. 重複イベント

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

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。

3.3.2. キャッシュマネージャーレベルの通知

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

クラスターに参加または退出するノード、または開始または停止するキャッシュがあります。

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。

3.3.3. 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) { }
}
```

3.3.3.1. 非同期スレッドプール

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

3.4. ASYNCHRONOUS API

Cache.put()、Cache.remove() などの同期 API メソッドの他に、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。 例: Cache.putAsync()、Cache.removeAsync() など。 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。

3.4.1. このような API を使用する理由

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

```
Set<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();
```

3.4.2. 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4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 クラスターのノード間で分散され、共有されるデータ構造です。クラスター化されたロックにより、ノード間で同期されるコードを実行できます。

4.1. ロック API

Data Grid は、埋め込みモードで Data Grid を使用するときに、クラスター上でコードを同時に実行できる **ClusteredLock** API を提供します。

API は以下で設定されます。

- **ClusteredLock** は、クラスター化されたロックを実装するメソッドを公開します。
- **ClusteredLockManager** は、クラスター化されたロックの定義、設定、取得、および削除を行うメソッドを公開します。
- **EmbeddedClusteredLockManagerFactory** は、**ClusteredLockManager** の実装を初期化します。

所有権

Data Grid は、クラスター内のすべてのノードがロックを使用できるように、**NODE** 所有権をサポートします。

再入可能性

Data Grid のクラスター化ロックは、再入可能ではないため、クラスター内のすべてのノードがロックを取得できますが、ロックを作成したノードのみがロックを解放することができます。

同じ所有者に対して 2 つの連続したロック呼び出しが送信された場合、最初の呼び出しが使用可能であればロックを取得し、2 番目の呼び出しはブロックされます。

参照資料

- EmbeddedClusteredLockManagerFactory
- ClusteredLockManager
- ClusteredLock

4.2. クラスター化されたロックの使用

アプリケーションに埋め込まれた Data Grid でクラスター化されたロックを使用する方法について説明します。

前提条件

- **infinispan-clustered-lock** 依存関係を **pom.xml** に追加します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-lock</artifactId>
</dependency>
```

手順

1. キャッシュマネージャーから **ClusteredLockManager** インターフェイスを初期化します。このインターフェイスは、クラスター化されたロックの定義、取得、および削除を行うエントリーポイントです。
2. クラスター化されたロックごとに一意の名前を指定します。
3. **lock.tryLock(1, TimeUnit.SECONDS)** メソッドでロックを取得します。

```
// Set up a clustered Cache Manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();

// Configure the cache mode, in this case it is distributed and synchronous.
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC);

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

// Initialize a Clustered Lock Manager.
ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);

// Define a clustered lock named 'lock'.
clm1.defineLock("lock");

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

AtomicInteger counter = new AtomicInteger(0);

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

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

CompletableFuture<Boolean> call3 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
```

```
    if (r) {
        System.out.println("lock is acquired by the call 3");
        lock.unlock().whenComplete((nil, ex2) -> {
            System.out.println("lock is released by the call 3");
            counter.incrementAndGet();
        });
    }
});

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

    // Stop the Cache Manager.
    cm.stop();
});
```

4.3. ロックの内部キャッシュの設定

クラスター化されたロックマネージャーには、ロック状態を格納する内部キャッシュが含まれます。内部キャッシュは、宣言的またはプログラムのいずれかに設定できます。

手順

1. クラスター化されたロックの状態を保存するクラスター内のノード数を定義します。デフォルト値は **-1** で、値をすべてのノードに複製します。

2. キャッシュの信頼性に以下のいずれかの値を指定します。これは、クラスターがパーティションに分割するか、複数のノードが残った場合にクラスター化ロックがどのように動作するかを制御します。

   - **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。

   - プログラムによる設定

     ```
     import org.infinispan.lock.configuration.ClusteredLockManagerConfiguration;
     import org.infinispan.lock.configuration.ClusteredLockManagerConfigurationBuilder;
     import org.infinispan.lock.configuration.Reliability;
     ...

     GlobalConfigurationBuilder global =
     GlobalConfigurationBuilder.defaultClusteredBuilder();

     final ClusteredLockManagerConfiguration config =
     global.addModule(ClusteredLockManagerConfigurationBuilder.class).numOwner(2).relia
     bility(Reliability.AVAILABLE).create();

     DefaultCacheManager cm = new DefaultCacheManager(global.build());
     ```

```
ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);

clm1.defineLock("lock");
```

- 宣言型設定

```
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:13.0
https://infinispan.org/schemas/infinispan-config-13.0.xsd"
      xmlns="urn:infinispan:config:13.0">

   <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:13.0"
                       num-owners = "3"
                       reliability="AVAILABLE">
         <clustered-lock name="lock1" />
         <clustered-lock name="lock2" />
      </clustered-locks>
   </cache-container>
   <!-- Cache configuration goes here. -->
</infinispan>
```

参照資料

- ClusteredLockManagerConfiguration
- Clustered Locks Configuration Schema

第5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。

5.1. 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

```
<infinispan>
   <cache-container ...>
      <!-- To persist counters, you need to configure the global state. -->
      <global-state>
      <!-- Global state configuration goes here. -->
      </global-state>
      <!-- Cache configuration goes here. -->
       <counters xmlns="urn:infinispan:config:counters:13.0" 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="0"/>
          <strong-counter name="c3" initial-value="3" storage="PERSISTENT" upper-bound="5"/>
          <strong-counter name="c4" initial-value="4" storage="VOLATILE" lower-bound="0" upper-
bound="10"/>
          <strong-counter name="c5" initial-value="0" upper-bound="100" lifespan="60000"/>
          <weak-counter name="c6" initial-value="5" storage="PERSISTENT" concurrency-level="1"/>
      </counters>
   </cache-container>
</infinispan>
```

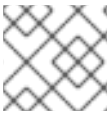
または、プログラムを使用して **GlobalConfigurationBuilder** で以下を行います。

```
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.addStrongCounter().name("c5").initialValue(0).upperBound(100).lifespan(60000);
builder.addWeakCounter().name("c6").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("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(0).upperBound(100).lif
espan(60000).build());
manager.defineCounter("c6",
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", ...);
}
```

関連情報

- Data Grid 設定スキーマ参照

5.1.1. カウンター名のリスト表示

定義されたすべてのカウンターをリスト表示するには、**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。

5.2. 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。

埋め込みデプロイメント

CounterManager は **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);
```

サーバーデプロイメント

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);
```

5.2.1. CounterManager を介したカウンターの削除

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。

5.3. 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。

5.3.1. 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。

5.3.1.1. 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();
```

5.3.1.2. 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)

5.3.1.3. 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());
```

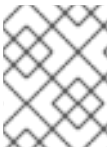
compare-and-swap では、呼び出しカウンターの呼び出し (**counter.getValue()**) が 1 つ保存されます。

```
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);
```

stong カウンターをレートリミッターとして使用するには、以下のように **upper-bound** パラメーターおよび **lifespan** パラメーターを設定します。

```
// 5 request per minute
CounterConfiguration configuration =
CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
      .upperBound(5)
      .lifespan(60000)
      .build();
counterManager.defineCounter("rate_limiter", configuration);
StrongCounter counter = counterManager.getStrongCounter("rate_limiter");

// on each operation, invoke
try {
   counter.incrementAndGet().get();
   // continue with operation
} catch (InterruptedException e) {
   Thread.currentThread().interrupt();
} catch (ExecutionException e) {
```

```
    if (e.getCause() instanceof CounterOutOfBoundsException) {
      // maximum rate. discard operation
      return;
    } else {
      // unexpected error, handling property
    }
  }
```



注記

lifespan パラメーターは実験的な機能で、今後のバージョンで削除される可能性があります。

5.3.2. 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);
```

5.3.2.1. 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。

5.3.2.2. 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。

5.3.2.3. 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());
```

5.4. 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);
}
```

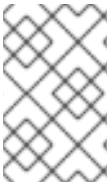
返される **Handle** オブジェクトには、**CounterListener** が必要なくなったときに削除するという主な目的があります。また、処理している **CounterListener** インスタンスにアクセスできます。これには、以下のインターフェイスがあります。

```
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6章 CDI 拡張機能の使用

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。

6.1. 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>
```

6.2. 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;
    }
}
```

6.3. 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。

6.4. 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

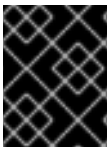
メソッドパラメーターをキャッシュします。

@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。

マネージド環境 (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>
```

マネージド外環境 (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。例を以下に示します。
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")
```

6.5. 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7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

7.1. 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。

7.1.1. クラスター化されたキャッシュおよびロック

Data Grid クラスターでは、プライマリー所有者ノードがキーをロックします。

非トランザクションキャッシュの場合、Data Grid は書き込み操作をキーのプライマリー所有者に転送して、ロックを試行できるようにします。次に、Data Grid は、他の所有者に書き込み操作を転送するか、キーをロックできない場合は例外を出力します。

注記

操作が条件付きで、プライマリー所有者で失敗した場合、Data Grid はこれを他の所有者には転送しません。

トランザクションキャッシュの場合、プライマリーの所有者は楽観的および悲観的ロックモードでキーをロックできます。Data Grid は、トランザクション間の同時読み取りを制御する異なる分離レベルもサポートします。

7.1.2. LockManager

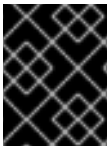
LockManager は、書き込み用にエントリーをロックするコンポーネントです。**LockManager** は、**LockContainer** を使用して、ロックを検索、保持、作成します。**LockContainers** には、ロックス

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。

7.1.3. 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);
```

7.1.4. 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);
```

7.1.5. 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);
```

7.1.6. 一貫性

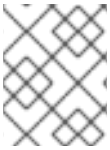
(すべての所有者がロックされているのとは対照的に) 単一の所有者がロックされるという事実により、次の一貫性の保証が失われることはありません。キー **K** がノード **{A, B}** に対してハッシュ化され、トランザクション **TX1** が、たとえば、**A** 上の **K** のロックを取得したとします。別のトランザクション **TX2** が **B** (またはその他のノード) 上で開始され、**TX2** が **K** のロックを試みる場合、ロックがすでに **TX1** によって保持されているため、タイムアウトでロックに失敗します。理由は、キー **K** のロックがト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

7.2. 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8章 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();
```

8.1. トランザクションの設定

トランザクションはキャッシュレベルで設定されます。以下はトランザクションの動作に影響する設定と、各設定属性の簡単な説明になります。

```
<locking
  isolation="READ_COMMITTED"/>
<transaction
  locking="OPTIMISTIC"
  auto-commit="true"
  complete-timeout="60000"
  mode="NONE"
  notifications="true"
  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)
   .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。
- **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。

8.2. 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()
```

8.3. トランザクションのロック

8.3.1. 悲観的なトランザクションキャッシュ

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

1. ロック要求がプライマリー所有者に送信されます (明示的なロック要求または操作のいずれか)。
2. プライマリーの所有者はロックの取得を試みます。
   a. 成功した場合は、正の応答が返されます。
   b. そうでない場合は、負の応答が送信され、トランザクションはロールバックされます。

たとえば、以下のようになります。

```
transactionManager.begin();
cache.put(k1,v1); //k1 is locked.
cache.remove(k2); //k2 is locked when this returns
transactionManager.commit();
```

cache.put(k1,v1) が返されると、**k1** はロックされ、クラスター内のどこかで実行中の他のトランザクションは、これに書き込むことができません。**k1** の読み取りは引き続き可能です。トランザクションの完了時に **k1** のロックが解放されます (コミットまたはロールバック)。

注記

条件付き操作の場合、検証はオリジネーターで実行されます。

8.3.2. 楽観的トランザクションキャッシュ

楽観的トランザクションロックはトランザクションの準備時に取得され、トランザクションのコミット (またはロールバック) まで保持されます。これは、書き込みでローカルロックを取得し、準備中にクラスターのロックが取得される 5.0 デフォルトロックモデルとは異なります。

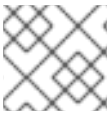
1. 準備はすべての所有者に送信されます。
2. プライマリーの所有者は、必要なロックの取得を試みます。
   a. ロックに成功すると、書き込みのスキューチェックが実行されます。

b. 書き込みスキューチェックが成功した場合 (または無効化された場合) は、正の応答を送信します。

c. それ以外の場合は、負の応答が送信され、トランザクションはロールバックされます。

たとえば、以下のようになります。

```
transactionManager.begin();
cache.put(k1,v1);
cache.remove(k2);
transactionManager.commit(); //at prepare time, K1 and K2 is locked until committed/rolled back.
```



注記

条件付きコマンドの場合、検証は引き続きオリジネーターで実行されます。

8.3.3. 悲観的または楽観的トランザクションのどちらが必要か

ユースケースの観点からは、同時に実行されている複数のトランザクション間で多くの競合が **ない** 場合は、楽観的トランザクションを使用する必要があります。これは、読み取り時と、コミット時 (書き込みスキューチェックが有効) の間でデータが変更された場合に、楽観的トランザクションがロールバックするためです。

一方、キーでの競合が多く、トランザクションのロールバックがあまり望ましくない場合は、悲観的トランザクションの方が適している可能性があります。悲観的トランザクションは、その性質上、よりコストがかかります。各書き込み操作ではロックの取得に RPC が関係する可能性があります。

8.4. スキューの書き込み

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

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。

8.4.1. 悲観的トランザクションでのキーへの書き込みロックの強制

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
```

8.5. 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。

8.6. 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)。

8.7. バッチ処理

バッチ処理は、トランザクションの原子性といくつかの特性を許可しますが、本格的な JTA または XA 機能は許可しません。多くの場合、バッチ処理は本格的なトランザクションよりもはるかに軽量で安価です。

ヒント

一般的には、トランザクションの参加者のみが 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**。

8.7.1. API

バッチ処理を使用するようにキャッシュを設定したら、**Cache** で **startBatch()** と **endBatch()** を呼び出して使用します。例:

```
Cache cache = cacheManager.getCache();
// not using a batch
cache.put("key", "value"); // will replicate immediately

// using a batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k2", "value");
cache.endBatch(true); // This will now replicate the modifications since the batch was started.

// a new batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false); // This will "discard" changes made in the batch
```

8.7.2. バッチ処理と JTA

裏では、バッチ機能が JTA トランザクションを開始し、そのスコープ内のすべての呼び出しがそれに関連付けられます。これには、内部 **TransactionManager** 実装が非常に簡単な (例: リカバリーなし) を使用します。バッチ処理では、以下を取得します。

1. 呼び出し中に取得したロックはバッチが完了するまで保持されます。
2. 変更はすべて、バッチ完了プロセスの一部として、クラスター内でバッチ内に複製されます。バッチの各更新のレプリケーションチャット数を減らします。
3. 同期のレプリケーションまたは無効化が使用された場合は、レプリケーション/無効化の失敗により、バッチがロールバックされます。
4. すべてのトランザクション関連の設定は、バッチ処理にも適用されます。

8.8. トランザクションリカバリー

リカバリーは XA トランザクションの機能であり、リソースの不測の事態、場合によってはトランザクションマネージャーの障害を対処し、それに応じてそのような状況から回復します。

8.8.1. リカバリーを使用するタイミング

外部データベースに保存されたアカウントから 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口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。

8.8.2. 仕組み

リカバリーはトランザクションマネージャーによって調整されます。トランザクションマネージャーは 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。

8.8.3. 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。

8.8.3.1. JMX サポートの有効化

リカバリー JMX サポートの管理に JMX を使用できるようにするには、明示的に有効にする必要があります。

8.8.4. 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。

8.8.5. トランザクションマネージャーとの統合

これはトランザクションマネージャーに固有のものですが、通常トランザクションマネージャーは **XAResource.recover()** を呼び出すために **XAResource** 実装への参照が必要になります。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。

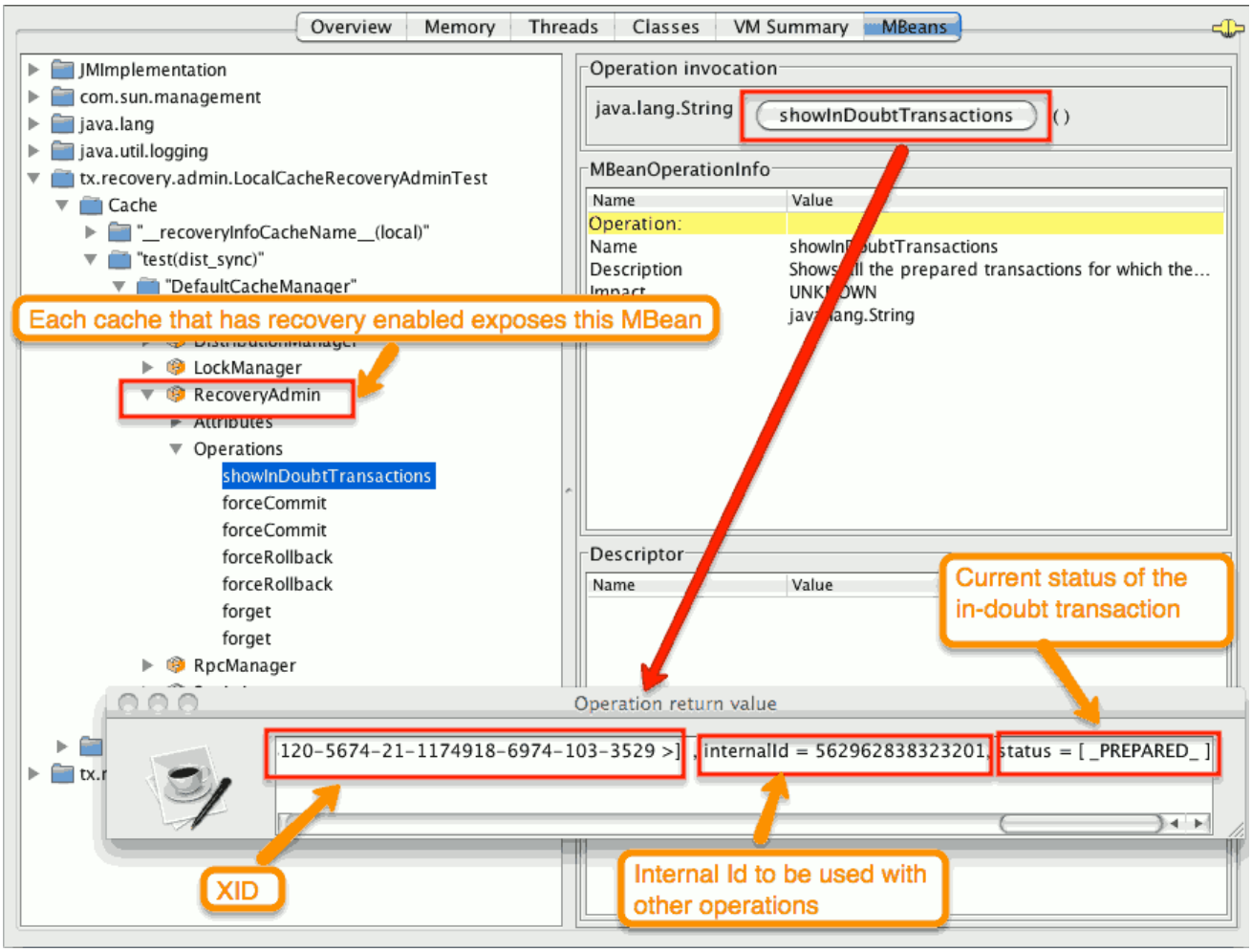
8.8.6. 調整

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リスト表示します。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8.1 未確定のトランザクションの表示

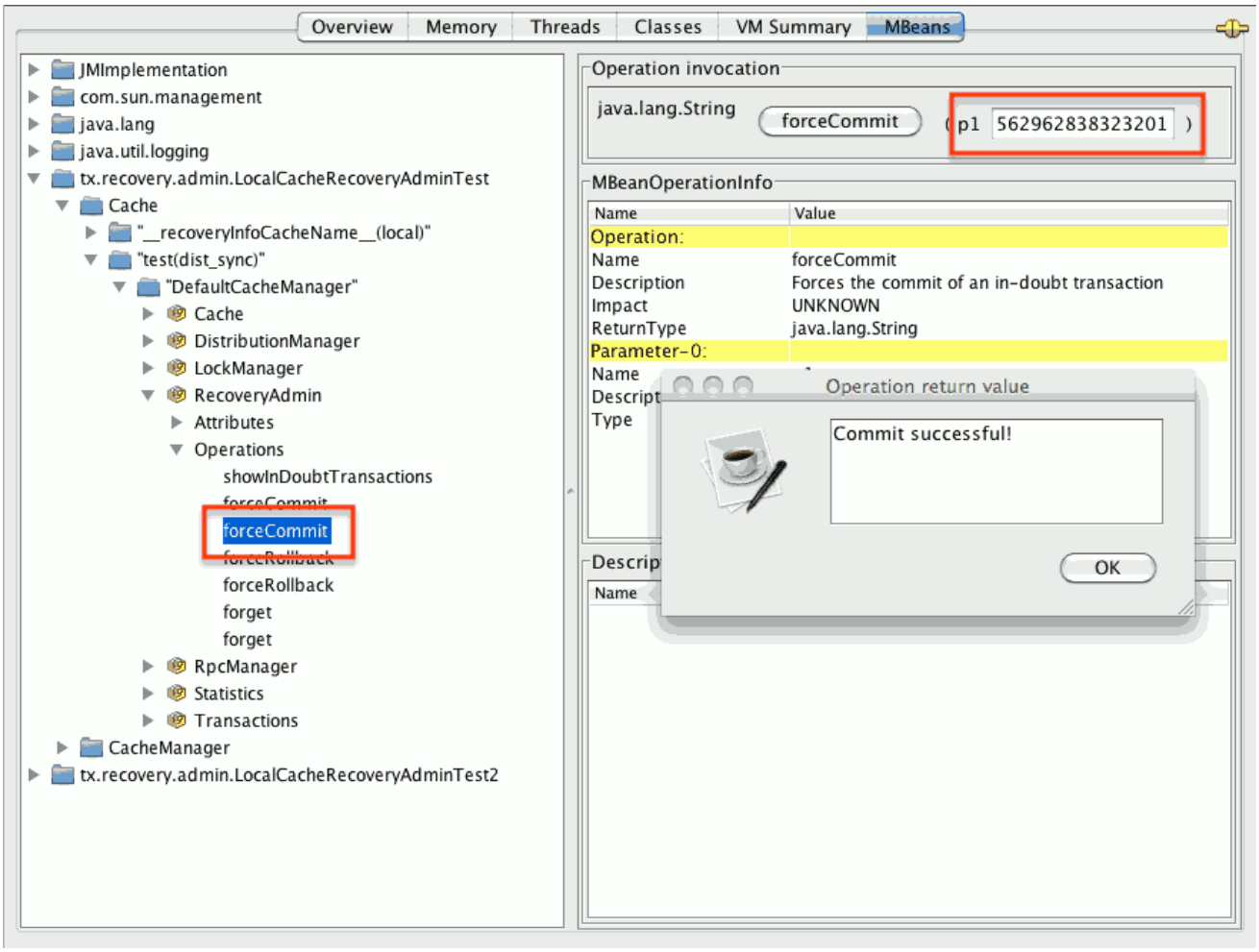


未確定の各トランザクションのステータスが表示されます (この例では **PREPARED** です)。status フィールドに複数の要素が存在する可能性があります。たとえば、トランザクションが特定ノードでコミットされていても、それらのノードでコミットされない場合は PREPARED および COMMITTED です。

- **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8.2 コミットの強制



ヒント

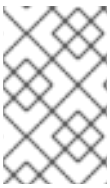
上記のすべての JMX 操作は、トランザクションの発信場所に関係なく、任意のノードで実行できます。

8.8.6.1. XID に基づくコミット/ロールバックの強制

未確定のトランザクションのコミット/ロールバックの強制を行う XID ベースの JMX 操作も使用できます。これらのメソッドはトランザクションに関連する番号ではなく、XID を記述する byte[] アレイを受け取ります (前述のステップ 2 で説明)。これらは、たとえば、特定の未確定トランザクションの自動完了ジョブを設定する場合に役立ちます。このプロセスはトランザクションマネージャーのリカバリーにプラグインされ、トランザクションマネージャーの XID オブジェクトにアクセスできます。

第9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。

9.1. 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。

9.1.1. 実行ノードのフィルタリング

コマンドを実行するノードを制限できます。たとえば、同じラック内のマシンでのみ計算を実行したい場合があります。または、ローカルサイトで 1 回、別のサイトで操作を再実行することもできます。クラスターエグゼキューターは、同じマシン、ラック、またはサイトレベルのスコープで要求を送信するノードを制限できます。

SameRack.java

```
EmbeddedCacheManager manager = ...;
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_RACK).submit(...)
```

このトポロジーベースフィルタ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リングすることもできます。これは任意で、以前のコードスニペットでトポロジーベースのフィルタリングと組み合わせることもできます。

また、実行対象と見なすことができるノードを除外する **Predicate** を使用して、任意の方法でターゲットノードを選択することもできます。これは同時に Topology フィルタリングと組み合わせて、クラスター内でコードを実行する場所をより詳細に制御できるようにすることもできます。

Predicate.java

```
EmbeddedCacheManager manager = ...;
// Just filter
manager.executor().filterTargets(a -> a.equals(..)).submit(...)
// Filter only those in the desired topology
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_SITE, a -> a.equals(..)).submit(...)
```

9.1.2. 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。

9.1.3. 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(...)
```

9.1.3.1. 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。

9.1.4. 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。

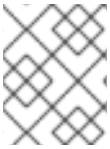
正方形の面積は Sa = 4r2 であり、円の面積は Ca=pi*r2 であることを思い出してください。2 つ目の式からの r2 を置き換えると、pi = 4 * Ca/S になります。ここで、正方形に非常に多くのダーツを射ることができると仮定して、射ったダーツの総数に対して円の中に入ったダーツの割合を取ると、Ca/Sa の値が近似します。pi = 4 * Ca/Sa であるため、pi の近似値を簡単に導き出すことができます。ダーツを多く撃つほど、より良い近似が得られます。以下の例では、10 億本のダーツを撃ちますが、それらを連続して撃つのではなく、Data Grid クラスター全体でダーツ射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

10.1. 一般的なストリーム操作

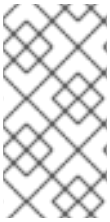
本セクションでは、使用している基礎となるキャッシュの種類に関係なく、さまざまなオプションを説明します。

10.2. キーのフィルタリング

特定のキーのサブセットでのみ動作するようにストリームをフィルターできます。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、ストリームを推奨します。

10.3. セグメントベースのフィルタリング



注記

これは高度な機能で、Data Grid セグメントおよびハッシュ技術の深い知識でのみ使用する必要があります。これらのセグメントベースのフィルタリングは、データを個別の呼び出しに分割する必要がある場合に便利です。これは、Apache Spark などの他のツールと統合する際に便利です。

このオプションは、レプリケートされたキャッシュと分散されたキャッシュでのみサポートされます。これにより、ユーザーは KeyPartitioner によって決定されるタイミングで、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。

10.4. 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。

10.5. 例

以下のコードはキャッシュを取得し、値に "JBoss" の文字列が含まれるすべてのキャッシュエントリーを持つマップを返します。

```
Map<Object, String> jbossValues =
cache.entrySet().stream()
    .filter(e -> e.getValue().contains("JBoss"))
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

10.6. 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。

10.6.1. 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。

10.6.2. 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;
      }
    }
```

10.7. 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。

10.8. 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。

10.9. 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。

10.10. 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。

10.11. 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。

10.12. 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 12 および 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。

10.13. 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11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。

11.1. 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。

```
<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。

```
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>());
```

11.1.1. 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。

```
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));
```

11.2. 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>());
```

11.2.1. 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);
```

11.3. 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。

11.4. 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()**

11.5. JCACHE インスタンスのクラスタリング

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12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。

12.1. インストールと設定

pom.xml

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

12.2. 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。

12.3. 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。

12.3.1. 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);
```

12.4. 制限

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。

12.4.1. 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。

12.4.2. 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。

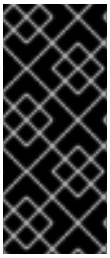
12.4.3. トランザクション

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13章 RED HAT JBOSS EAP のデータグリッドモジュール

Red Hat JBoss EAP にデプロイされたアプリケーション内で Data Grid を使用するには、以下を実行する Data Grid モジュールをインストールする必要があります。

- WAR または EAR ファイルに Data Grid JAR ファイルをパッケージ化せずにアプリケーションをデプロイできます。
- Red Hat JBoss EAP にバンドルされているバージョンとは独立した Data Grid を使用できるようにします。



重要

Red Hat JBoss EAP (EAP) アプリケーションは、Data Grid モジュールを別途インストールすることなく、**infinispan** サブシステムを直接処理することができます。Red Hat は、EAP バージョン 7.4 以降、この機能をサポートしています。ただし、デプロイでは、EAP モジュールがインデックス作成やクエリーなどの高度な機能を使用する必要があります。

13.1. DATA GRID モジュールのインストール

Red Hat JBoss EAP の Data Grid モジュールをダウンロードしてインストールします。

前提条件

1. JDK 8 以降。
2. 既存の Red Hat JBoss EAP インストール

手順

1. Red Hat カスタマーポータルにログインします。
2. Data Grid ソフトウェアダウンロード からモジュールの ZIP アーカイブをダウンロードします。
3. ZIP アーカイブを抽出し、**modules** の内容を Red Hat JBoss EAP インストールの **modules** ディレクトリーにコピーして、結果の構造を取得します。
   $EAP_HOME/modules/system/add-ons/rhdg/org/infinispan/rhdg-8.4

13.2. DATA GRID モジュールを使用するためのアプリケーションの設定

Red Hat JBoss EAP 用の DataGrid モジュールをインストールした後、Data Grid 機能を使用するようにアプリケーションを設定します。

手順

1. プロジェクトの **pom.xml** ファイルで、必要な Data Grid の依存関係を **提供されている**ものとしてマークします。
2. 適切な **MANIFEST.MF** ファイルを生成するようにアーティファクトアーカイバを設定します。

pom.xml

```
<dependencies>
 <dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-core</artifactId>
  <scope>provided</scope>
 </dependency>
 <dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cachestore-jdbc</artifactId>
  <scope>provided</scope>
 </dependency>
</dependencies>
<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-war-plugin</artifactId>
   <configuration>
    <archive>
     <manifestEntries>
      <Dependencies>org.infinispan:rhdg-8.3 services</Dependencies>
     </manifestEntries>
    </archive>
   </configuration>
  </plugin>
 </plugins>
</build>
```

Data Grid 機能は、以下のようにアプリケーションのマニフェストにエントリーとして追加できる単一のモジュール **org.infinispan** としてパッケージ化されます。

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:rhdg-8.3 services
```

AWS の依存関係

S3_PING などの AWS 依存関係が必要な場合は、アプリケーションのマニフェストに以下のモジュールを追加します。

```
Manifest-Version: 1.0
Dependencies: com.amazonaws.aws-java-sdk:rhdg-8.3 services
```

第14章 RED HAT JBOSS WEB SERVER から RED HAT DATA GRID への HTTP セッションの外部化

org.apache.catalina.Manager インターフェイスを使用して、HTTP セッションデータを JBoss Web Server デプロイメントから Data Grid Server クラスターに外部化して、高可用性を実現します。

14.1. TOMCAT セッションクライアントのインストール

Tomcat セッションクライアントをインストールし、Red Hat JBoss Web Server アプリケーションから Red Hat Data Grid に HTTP セッションを外部化します。

手順

1. Data Grid Software Downloads から **redhat-datagrid-8.1.1-tomcat<$version>-session-client.zip** アーカイブをダウンロードします。
2. アーカイブをローカルのファイルシステムにデプロイメントします。
3. デプロイメントしたアーカイブから **$CATALINA_HOME/lib** に **lib/** ディレクトリーの内容をコピーします。

14.2. セッションマネジャーの設定

セッションマネージャーの **HotRodManager** クラスを設定し、Tomcat セッションクライアントが Red Hat Data Grid Server に接続し、データをリモートキャッシュに保存する方法を定義します。

前提条件

- Tomcat セッションクライアントをインストールします。
- 1 つ以上の Data Grid Server インスタンスをインストールします。
- HTTP セッションデータを保存するテンプレートとして使用する Data Grid Server にキャッシュを作成します。

手順

1. **$CATALINA_HOME/conf/context.xml** または **/WEB-INF/context.xml** を開いて編集します。
2. **org.wildfly.clustering.tomcat.hotrod.HotRodManager** を **className** プロパティーの値として指定します。
3. **configurationName** プロパティーで、テンプレートとして使用するキャッシュの名前を指定します。
4. **HotRodManager** クラスの他の設定プロパティーを必要に応じて定義します。
5. **infinispan.client.hotrod.** 接頭辞なしで Hot Rod クライアント設定プロパティーを設定します。
    a. **server_list** プロパティーで Data Grid Server ノードのリストを指定します。
    b. **auth_username** および **auth_password** プロパティーで Data Grid のクレデンシャルを指定します。

6. 必要に応じて、Tomcat セッションマネージャーの一般的な属性を指定します。
7. **context.xml** を保存して閉じます。

設定例

```
<Manager className="org.wildfly.clustering.tomcat.hotrod.HotRodManager"
        configurationName="mycache"
        persistenceStrategy="FINE"
        maxActiveSessions="100"
        server_list="192.0.2.0:11222;192.0.2.0:11223;192.0.2.0:11224"
        protocol_version="2.9"
        auth_username="admin"
        auth_password="changeme"
        auth_realm="default"
        sasl_mechanism="DIGEST-MD5"
        auth_server_name="infinispan"/>
```

検証

Tomcat セッションクライアントがリモートキャッシュにデータを保存することを確認するには、以下を行います。

1. 任意のブラウザーで Data Grid コンソールを開きます。
   デフォルトでは、コンソールは **http://127.0.0.1:11222/console/** で利用できます。
2. Tomcat セッションクライアントが、デプロイされたアプリケーションごとにキャッシュを作成することを確認します。

14.2.1. Hot Rod マネージャー設定プロパティー

以下の表は、**HotRodManager** クラスの設定プロパティーをリストおよび説明しています。

プロパティー	説明
className	**org.wildfly.clustering.tomcat.hotrod.HotRodManager** をセッションマネージャーとして指定します。
configurationName	HTTP セッションデータを保存するテンプレートとして使用する Data Grid Server のリモートキャッシュを指定します。
persistenceStrategy	セッションをキャッシュのエントリーにマップする方法を定義します。 **COARSE** は、単一のキャッシュエントリー内にすべてのセッション属性を保存します。これはデフォルトになります。 **FINE** は、セッション属性を個別のキャッシュエントリーに保存します。

プロパティー	説明
maxActiveSessions	キャッシュに保存するセッションの最大数を定義します。デフォルトは最大値なし (制限なし) です。

関連情報

- Data Grid のドキュメント
- Tomcat 7.0 Manager の共通属性
- Tomcat 8.0 Manager の共通属性
- Tomcat 8.5 Manager の共通属性
- Tomcat 9.0 Manager の共通属性

第15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。

15.1. 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. -->
  <!-- Custom interceptors should extend
       org.infinispan.interceptors.BaseCustomAsyncInterceptor -->
  <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>
```

15.2. 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。

15.3. 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。