Red Hat Data Grid 8.5

Java アプリケーションへの Data Grid の埋め込み

Data Grid を使用した組み込みキャッシュの作成

Last Updated: 2024-07-11

Red Hat Data Grid 8.5 Java アプリケーションへの Data Grid の埋め込み

Data Grid を使用した組み込みキャッシュの作成

法律上の通知

概要

Data Grid を Java プロジェクトに追加し、アプリケーションで組み込みキャッシュを使用します。

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。

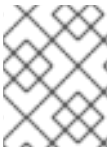
DATA GRID のドキュメント

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 8.4 ドキュメント
- Data Grid 8.4 コンポーネントの詳細
- Data Grid 7.3 でサポートされる構成
- 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、用語の置き換えは、今後の複数のリリースにわたって段階的に実施されます。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

第1章 DATA GRID の MAVEN リポジトリーへの追加

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. 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 8.1 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

- Red Hat Enterprise Maven Repository

1.3. プロジェクト POM の設定

プロジェクト内のプロジェクトオブジェクトモデル (POM) ファイルを設定して、組み込みキャッシュ、Hot Rod クライアント、およびその他の機能に Data Grid 依存関係を使用します。

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>14.0.21.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 組み込みキャッシュの作成

Data Grid は、プログラムを使用して Cache Manager と組み込みキャッシュライフサイクルの両方を制御できる **EmbeddedCacheManager** API を提供します。

2.1. DATA GRID のプロジェクトへの追加

Data Grid をプロジェクトに追加して、アプリケーションで組み込みキャッシュを作成します。

前提条件

- Maven リポジトリーから Data Grid アーティファクトを取得するようにプロジェクトを設定します。

手順

- 以下のように、**infinispan-core** アーティファクトを **pom.xml** の依存関係として追加します。

```
<dependencies>
 <dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-core</artifactId>
 </dependency>
</dependencies>
```

2.2. 組み込みキャッシュの作成と使用

Data Grid は、キャッシュマネージャーを制御する **GlobalConfigurationBuilder** API と、キャッシュを設定する **ConfigurationBuilder** API を提供します。

前提条件

- **infinispan-core** アーティファクトを **pom.xml** の依存関係として追加します。

手順

1. **CacheManager** を初期化します。

注記

キャッシュを作成する前に、必ず **cacheManager.start()** メソッドを呼び出して **CacheManager** を初期化する必要があります。デフォルトのコンストラクターはこれを行いますが、オーバーロードされたバージョンのコンストラクターはこれを行いません。

キャッシュマネージャーも重量のあるオブジェクトであり、Data Grid では、JVM ごとに 1 つのインスタンスのみをインスタンス化することを推奨します。

2. **ConfigurationBuilder** API を使用して、キャッシュ設定を定義します。
3. **getCache()**、**createCache()**、または **getOrCreateCache()** メソッドで、キャッシュを取得します。

Data Grid では、**getOrCreateCache()** メソッドを使用することを推奨します。これは、すべてのノードでキャッシュを作成するか、既存のキャッシュを返すためです。

4. 必要であれば、キャッシュが再起動しても大丈夫なように、**PERMANENT** フラグを使用します。

5. **cacheManager.stop()** メソッドを呼び出して **CacheManager** を停止し、JVM リソースを解放してキャッシュを正常にシャットダウンします。

```
// Set up a clustered Cache Manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();
// Initialize the default Cache Manager.
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());
// Create a distributed cache with synchronous replication.
ConfigurationBuilder builder = new ConfigurationBuilder();
                builder.clustering().cacheMode(CacheMode.DIST_SYNC);
// Obtain a volatile cache.
Cache<String, String> cache =
cacheManager.administration().withFlags(CacheContainerAdmin.AdminFlag.VOLATILE).getOrCreateC
ache("myCache", builder.build());
// Stop the Cache Manager.
cacheManager.stop();
```

getCache() メソッド

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

createCache() メソッドを呼び出して、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

PERMANENT フラグを使用して、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。

関連情報

- EmbeddedCacheManager
- EmbeddedCacheManager Configuration
- org.infinispan.configuration.global.GlobalConfiguration
- org.infinispan.configuration.cache.ConfigurationBuilder

2.3. CACHE API

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。

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。

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

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。

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。

2.3.1. 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();
```

2.3.1.1. Flags

フラグは通常のキャッシュメソッドに適用され、特定のメソッドの動作を変更します。利用可能なフラグの一覧と、その効果については、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");
```

2.3.2. 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。

2.3.2.1. 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();
```

2.3.2.2. 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3章 PROGRAMMATICALLY CONFIGURING USER ROLES AND PERMISSIONS

Java アプリケーションで組み込みキャッシュを使用する場合は、セキュリティー認証をプログラムで設定します。

3.1. DATA GRID のユーザーロールと権限

Data Grid には、キャッシュや Data Grid のリソースにアクセスするための権限をユーザーに提供するロールがいくつかあります。

Role	権限	説明
admin	ALL	Cache Manager ライフサイクルの制御など、すべてのパーミッションを持つスーパーユーザー。
deployer	ALL_READ、ALL_WRITE、LISTEN、EXEC、MONITOR、CREATE	**application** パーミッションに加えて、Data Grid リソースを作成および削除できます。
application	ALL_READ、ALL_WRITE、LISTEN、EXEC、MONITOR	**observer** パーミッションに加え、Data Grid リソースへの読み取りおよび書き込みアクセスがあります。また、イベントをリッスンし、サーバータスクおよびスクリプトを実行することもできます。
observer	ALL_READ、MONITOR	**monitor** パーミッションに加え、Data Grid リソースへの読み取りアクセスがあります。
monitor	MONITOR	JMX および **metrics** エンドポイント経由で統計を表示できます。

関連情報

- org.infinispan.security.AuthorizationPermission Enum
- Data Grid 設定スキーマ参照

3.1.1. 権限

ユーザーロールは、さまざまなアクセスレベルを持つパーミッションのセットです。

表3.1 Cache Manager のパーミッション

Permission	機能	説明

設定	**defineConfiguration**	新しいキャッシュ設定を定義します。
LISTEN	**addListener**	キャッシュマネージャーに対してリスナーを登録します。
ライフサイクル	**stop**	キャッシュマネージャーを停止します。
CREATE	**createCache**, **removeCache**	キャッシュ、カウンター、スキーマ、スクリプトなどのコンテナーリソースを作成および削除することができます。
MONITOR	**getStats**	JMX 統計および **metrics** エンドポイントへのアクセスを許可します。
ALL	-	すべてのキャッシュマネージャーのアクセス許可が含まれます。

表3.2 キャッシュ権限

Permission	機能	説明
READ	**get**, **contains**	キャッシュからエントリーを取得します。
WRITE	**put**, **putIfAbsent**, **replace**, **remove**, **evict**	キャッシュ内のデータの書き込み、置換、削除、エビクト。
EXEC	**distexec**, **streams**	キャッシュに対するコードの実行を許可します。
LISTEN	**addListener**	キャッシュに対してリスナーを登録します。
BULK_READ	**keySet**, **values**, **entrySet**, **query**	一括取得操作を実行します。
BULK_WRITE	**clear**, **putAll**	一括書き込み操作を実行します。
ライフサイクル	**start**, **stop**	キャッシュを開始および停止します。

ADMIN	**getVersion**, **addInterceptor***, **removeInterceptor**, **getInterceptorChain**, **getEvictionManager**, **getComponentRegistry**, **getDistributionManager**, **getAuthorizationManager**, **evict**, **getRpcManager**, **getCacheConfiguration**, **getCacheManager**, **getInvocationContextContainer**, **setAvailability**, **getDataContainer**, **getStats**, **getXAResource**	基盤となるコンポーネントと内部構造へのアクセスを許可します。
MONITOR	**getStats**	JMX 統計および **metrics** エンドポイントへのアクセスを許可します。
ALL	-	すべてのキャッシュパーミッションが含まれます。
ALL_READ	-	READ パーミッションと BULK_READ パーミッションを組み合わせます。
ALL_WRITE	-	WRITE パーミッションと BULK_WRITE パーミッションを組み合わせます。

関連情報

- Data Grid Security API

3.1.2. ロールとパーミッションマッパー

Data Grid は、ユーザーをプリンシパルのコレクションとして実装します。プリンシパルは、ユーザー名などの個々のユーザー ID、またはユーザーが属するグループのいずれかを表します。内部的には、これらは **javax.security.auth.Subject** クラスで実装されます。

承認を有効にするには、プリンシパルをロール名にマップし、その後、ロール名を一連のパーミッションに展開する必要があります。

Data Grid には、セキュリティープリンシパルをロールに関連付ける **PrincipalRoleMapper** API と、ロールを特定のパーミッションに関連付ける **RolePermissionMapper** API が含まれています。

Data Grid は以下のロールおよびパーミッションのマッパーの実装を提供します。

クラスターロールマッパー

クラスターレジストリーにロールマッピングのプリンシパルを保存します。

クラスターパーミッションマッパー

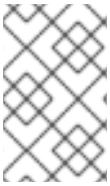
ロールとパーミッションのマッピングをクラスターレジストリーに保存します。ユーザーのロールとパーミッションを動的に変更できます。

ID ロールマッパー

ロール名としてプリンシパル名を使用します。プリンシパル名のタイプまたはフォーマットはソースに依存します。たとえば、LDAP ディレクトリーでは、プリンシパル名を識別名 (DN) にすることができます。

コモンネームロールマッパー

ロール名としてコモンネーム (CN) を使用します。このロールマッパーは、識別名 (DN) を含む LDAP ディレクトリーまたはクライアント証明書で使用できます。たとえば、**cn=managers,ou=people,dc=example,dc=com** は、**managers** ロールにマッピングされます。



注記

デフォルトでは、プリンシパルからロールへのマッピングはグループを表すプリンシパルにのみ適用されます。ユーザープリンシパルのマッピングを実行するように Data Grid を設定することもできます。

3.1.2.1. Data Grid でのロールとパーミッションへのユーザーのマッピング

LDAP サーバーから DN のコレクションとして取得された次のユーザーを考えてみましょう。

```
CN=myapplication,OU=applications,DC=mycompany
CN=dataprocessors,OU=groups,DC=mycompany
CN=finance,OU=groups,DC=mycompany
```

コモンネームロールマッパー を使用すると、ユーザーは次のロールにマッピングされます。

```
dataprocessors
finance
```

Data Grid には次のロール定義があります。

```
dataprocessors: ALL_WRITE ALL_READ
finance: LISTEN
```

ユーザーには次のパーミッションが与えられます。

```
ALL_WRITE ALL_READ LISTEN
```

関連情報

- Data Grid Security API
- org.infinispan.security.PrincipalRoleMapper
- org.infinispan.security.RolePermissionMapper
- org.infinispan.security.mappers.IdentityRoleMapper
- org.infinispan.security.mappers.CommonNameRoleMapper

3.1.3. ロールマッパーの設定

Data Grid は、デフォルトでクラスターロールマッパーとクラスターパーミッションマッパーを有効にします。ロールマッピングに別の実装を使用するには、ロールマッパーを設定する必要があります。

手順

1. Data Grid 設定を開いて編集します。
2. ロールマッパーを、キャッシュマネージャー設定のセキュリティー許可の一部として宣言します。
3. 変更を設定に保存します。

組み込みキャッシュを使用すると、**principalRoleMapper()** および **rolePermissionMapper()** メソッドを使用して、ロールおよびパーミッションマッパーをプログラムで設定できます。

ロールマッパーの設定

XML

```
<cache-container>
  <security>
    <authorization>
      <common-name-role-mapper />
    </authorization>
  </security>
</cache-container>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "security" : {
        "authorization" : {
          "common-name-role-mapper": {}
        }
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    security:
      authorization:
        commonNameRoleMapper: ~
```

関連情報

- Data Grid 設定スキーマ参照

3.2. 組み込みキャッシュの承認の有効化と設定

組み込みキャッシュを使用する場合は、**GlobalSecurityConfigurationBuilder** および **ConfigurationBuilder** クラスを使用して承認を設定できます。

手順

1. **GlobalConfigurationBuilder** を構築し、**security().authorization().enable()** メソッドでセキュリティー認証を有効にします。
2. **principalRoleMapper ()** メソッドでロールマッパーを指定します。
3. 必要に応じて、**role()** および **permission()** メソッドを使用して、カスタムのロールと権限のマッピングを定義します。

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
 global.security().authorization().enable()
        .principalRoleMapper(new ClusterRoleMapper())
        .role("myroleone").permission(AuthorizationPermission.ALL_WRITE)
        .role("myroletwo").permission(AuthorizationPermission.ALL_READ);
```

4. **ConfigurationBuilder** でキャッシュの承認を有効にします。
   - グローバル設定からすべてのロールを追加します。

```
ConfigurationBuilder config = new ConfigurationBuilder();
config.security().authorization().enable();
```

   - ロールを持たないユーザーのアクセスを Data Grid が拒否するように、キャッシュのロールを明示的に定義します。

```
ConfigurationBuilder config = new ConfigurationBuilder();
config.security().authorization().enable().role("myroleone");
```

関連情報

- org.infinispan.configuration.global.GlobalSecurityConfigurationBuilder
- org.infinispan.configuration.cache.ConfigurationBuilder

3.3. 実行時にの承認ロールの追加

Data Grid キャッシュでセキュリティー認証を使用する場合に、ロールをアクセス許可に動的にマップします。

前提条件

- 組み込みキャッシュの承認を設定しておく。
- Data Grid に **ADMIN** 権限があること。

手順

1. **RolePermissionMapper** インスタンスを取得します。
2. **addRole()** メソッドで新しいロールを定義します。

```
MutableRolePermissionMapper mapper = (MutableRolePermissionMapper)
cacheManager.getCacheManagerConfiguration().security().authorization().rolePermissionMap
per();
mapper.addRole(Role.newRole("myroleone", true, AuthorizationPermission.ALL_WRITE,
AuthorizationPermission.LISTEN));
mapper.addRole(Role.newRole("myroletwo", true, AuthorizationPermission.READ,
AuthorizationPermission.WRITE));
```

関連情報

- org.infinispan.security.RolePermissionMapper

3.4. 安全なキャッシュを使用したコードの実行

セキュリティー認証を使用する組み込みキャッシュの **DefaultCacheManager** を構築すると、Cache Manager は、操作を呼び出す前にセキュリティーコンテキストをチェックする **SecureCache** を返します。また、**SecureCache** は、アプリケーションが **DataContainer** などの低レベルの非セキュアなオブジェクトを取得できないようにします。このため、適切なレベルの権限を持つロールが割り当てられた Data Grid ユーザーでコードを実行する必要があります。

前提条件

- 組み込みキャッシュの承認を設定しておく。

手順

1. 必要に応じて、Data Grid コンテキストから現在のサブジェクトを取得します。

```
Security.getSubject();
```

2. **PrivilegedAction** でメソッド呼び出しをラップして、サブジェクトで実行します。

```
Security.doAs(mySubject, (PrivilegedAction<String>)() -> cache.put("key", "value"));
```

関連情報

- org.infinispan.security.Security
- org.infinispan.security.SecureCache

3.5. アクセス制御リスト (ACL) キャッシュの設定

ユーザーにロールを付与または拒否すると、Data Grid は、キャッシュに内部的にアクセスできるユーザーに関する詳細を保存します。この ACL キャッシュは、ユーザーがすべての要求に対して読み取りおよび書き込み操作を実行するための適切なアクセス許可を持っているかどうかを Data Grid が計算する必要をなくすことで、セキュリティー認証のパフォーマンスを向上させます。

注記

ユーザーにロールを付与または拒否するたびに、Data Grid は ACL キャッシュをフラッシュして、ユーザー権限が正しく適用されるようにします。これは、ロールを付与または拒否するたびに、Data Grid がすべてのユーザーのキャッシュ許可を再計算する必要があることを意味します。最高のパフォーマンスを得るには、本番環境でロールの付与と拒否を頻繁にまたは繰り返してはいけません。

手順

1. Data Grid 設定を開いて編集します。
2. **cache-size** 属性を使用して、ACL キャッシュのエントリーの最大数を指定します。
   ACL キャッシュ内のエントリーには、**caches * users** のカーディナリティーがあります。エントリーの最大数は、すべてのキャッシュとユーザーの情報を保持できる値に設定する必要があります。たとえば、デフォルトサイズの **1000** は、最大 100 のキャッシュとユーザーが 10 個のデプロイメントに適しています。
3. **cache-timeout** 属性を使用して、ミリ秒単位でタイムアウト値を設定します。
   Data Grid がタイムアウト期間内に ACL キャッシュ内のエントリーにアクセスしない場合、そのエントリーは削除されます。その後、ユーザーがキャッシュ操作を試みると、Data Grid はユーザーのキャッシュ許可を再計算し、ACL キャッシュにエントリーを追加します。

重要

cache-size または **cache-timeout** 属性に値 **0** を指定すると、ACL キャッシュが無効になります。認証を無効にする場合にのみ、ACL キャッシュを無効にする必要があります。

4. 変更を設定に保存します。

ACL キャッシュの設定

XML

```
<infinispan>
  <cache-container name="acl-cache-configuration">
    <security cache-size="1000"
              cache-timeout="300000">
      <authorization/>
    </security>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "name" : "acl-cache-configuration",
      "security" : {
        "cache-size" : "1000",
        "cache-timeout" : "300000",
```

```
        "authorization" : {}
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    name: "acl-cache-configuration"
    security:
      cache-size: "1000"
      cache-timeout: "300000"
      authorization: ~
```

3.5.1. ACL キャッシュのフラッシュ

JMX 経由でアクセス可能な **GlobalSecurityManager** MBean を使用して、ACL キャッシュをフラッシュできます。

関連情報

- Data Grid 設定スキーマ参照

第4章 DATA GRID 統計および JMX 監視の有効化および設定

Data Grid は、JMX MBean をエクスポートしたり、Cache Manager およびキャッシュ統計を提供できます。

4.1. 組み込みキャッシュでの統計の有効化

キャッシュマネージャーおよび組み込みキャッシュの統計をエクスポートするように Data Grid を設定します。

手順

1. Data Grid 設定を開いて編集します。
2. **statistics="true"** 属性または **.statistics(true)** メソッドを追加します。
3. Data Grid 設定を保存して閉じます。

組み込みキャッシュの統計

XML

```
<infinispan>
  <cache-container statistics="true">
    <distributed-cache statistics="true"/>
    <replicated-cache statistics="true"/>
  </cache-container>
</infinispan>
```

GlobalConfigurationBuilder

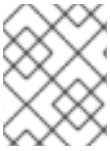
```
GlobalConfigurationBuilder global =
GlobalConfigurationBuilder.defaultClusteredBuilder().cacheContainer().statistics(true);
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());

Configuration builder = new ConfigurationBuilder();
builder.statistics().enable();
```

4.2. DATA GRID メトリックの設定

Data Grid は、あらゆる監視システムと互換性のあるメトリクスを生成します。

- ゲージは、書き込み操作または JVM アップタイムの平均数 (ナノ秒) などの値を指定します。
- ヒストグラムは、読み取り、書き込み、削除の時間などの操作実行時間の詳細を提供します。

デフォルトでは、Data Grid は統計を有効にするとゲージを生成しますが、ヒストグラムを生成するように設定することもできます。



注記

Data Grid メトリックは、**vendor** スコープで提供されます。JVM に関連するメトリックは **base** スコープで提供されます。

前提条件

- 組み込みキャッシュの Data Grid メトリクスをエクスポートするには、Micrometer Core および Micrometer Registry Prometheus JAR をクラスパスに追加する必要があります。

手順

1. Data Grid 設定を開いて編集します。
2. **metrics** 要素またはオブジェクトをキャッシュコンテナーに追加します。
3. **gauges** 属性またはフィールドを使用してゲージを有効または無効にします。
4. **histograms** 属性またはフィールドでヒストグラムを有効または無効にします。
5. クライアント設定を保存して閉じます。

メトリックの設定

XML

```
<infinispan>
  <cache-container statistics="true">
    <metrics gauges="true"
             histograms="true" />
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
      "metrics" : {
        "gauges" : "true",
        "histograms" : "true"
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    statistics: "true"
```

```
  metrics:
    gauges: "true"
    histograms: "true"
```

GlobalConfigurationBuilder

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  //Computes and collects statistics for the Cache Manager.
  .statistics().enable()
  //Exports collected statistics as gauge and histogram metrics.
  .metrics().gauges(true).histograms(true)
  .build();
```

関連情報

- Micrometer Prometheus

4.3. JMX MBEAN の登録

Data Grid は、統計の収集と管理操作の実行に使用できる JMX MBean を登録できます。統計を有効にする必要もあります。そうしないと、Data Grid は JMX MBean のすべての統計属性に **0** 値を提供します。

手順

1. Data Grid 設定を開いて編集します。
2. **jmx** 要素またはオブジェクトをキャッシュコンテナーに追加し、**enabled** 属性またはフィールドの値として **true** を指定します。
3. **domain** 属性またはフィールドを追加し、必要に応じて JMX MBean が公開されるドメインを指定します。
4. クライアント設定を保存して閉じます。

JMX の設定

XML

```
<infinispan>
  <cache-container statistics="true">
    <jmx enabled="true"
         domain="example.com"/>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
```

```
      "jmx" : {
        "enabled" : "true",
        "domain" : "example.com"
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    statistics: "true"
    jmx:
      enabled: "true"
      domain: "example.com"
```

GlobalConfigurationBuilder

```
GlobalConfiguration global = GlobalConfigurationBuilder.defaultClusteredBuilder()
   .jmx().enable()
   .domain("org.mydomain");
```

4.3.1. JMX リモートポートの有効化

一意のリモート JMX ポートを提供し、JMXServiceURL 形式の接続を介して Data Grid MBean を公開します。

次のいずれかの方法を使用して、リモート JMX ポートを有効にできます。

- Data Grid サーバーセキュリティーレルムの 1 つに対する認証を必要とするリモート JMX ポートを有効にします。
- 標準の Java 管理設定オプションを使用して、手動でリモート JMX ポートを有効にします。

前提条件

- 認証付きのリモート JMX の場合、デフォルトのセキュリティーレルムを使用して JMX 固有のユーザーロールを定義します。ユーザーが JMX リソースにアクセスするには、読み取り/書き込みアクセス権を持つ **controlRole** または読み取り専用アクセス権を持つ **monitorRole** が必要です。Data Grid は、グローバルな **ADMIN** 権限と **MONITOR** パーミッションを JMX の **controlRole** および **monitorRole** ロールに自動的にマッピングします。

手順

次のいずれかの方法を使用して、リモート JMX ポートを有効にして Data Grid サーバーを起動します。

- ポート **9999** を介してリモート JMX を有効にします。

```
bin/server.sh --jmx 9999
```

警告

SSL を無効にしてリモート JMX を使用することは、本番環境向けではありません。

- 起動時に以下のシステムプロパティーを Data Grid サーバーに渡します。

```
bin/server.sh -Dcom.sun.management.jmxremote.port=9999 -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false
```

警告

認証または SSL なしでリモート JMX を有効にすることは安全ではなく、どのような環境でも推奨されません。認証と SSL を無効にすると、権限のないユーザーがサーバーに接続し、そこでホストされているデータにアクセスできるようになります。

関連情報

- セキュリティーレルムの作成

4.3.2. Data Grid MBean

Data Grid は、管理可能なリソースを表す JMX MBean を公開します。

org.infinispan:type=Cache
: キャッシュインスタンスに使用できる属性および操作。

org.infinispan:type=CacheManager
: Data Grid キャッシュやクラスターのヘルス統計など、Cache Manager で使用できる属性および操作。

使用できる JMX MBean の詳細なリストおよび説明、ならびに使用可能な操作および属性については、**Data Grid JMX Components** のドキュメントを参照してください。

関連情報

- Data Grid JMX Components

4.3.3. カスタム MBean サーバーでの MBean の登録

Data Grid には、カスタム MBeanServer インスタンスに MBean を登録するのに使用できる **MBeanServerLookup** インターフェイスが含まれています。

前提条件

- **getMBeanServer()** メソッドがカスタム MBeanServer インスタンスを返すように **MBeanServerLookup** の実装を作成します。
- JMX MBean を登録するように Data Grid を設定します。

手順

1. Data Grid 設定を開いて編集します。
2. **mbean-server-lookup** 属性またはフィールドをキャッシュマネージャーの JMX 設定に追加します。
3. **MBeanServerLookup** 実装の完全修飾名 (FQN) を指定します。
4. クライアント設定を保存して閉じます。

JMX MBean サーバールックアップの設定

XML

```xml
<infinispan>
  <cache-container statistics="true">
    <jmx enabled="true"
         domain="example.com"
         mbean-server-lookup="com.example.MyMBeanServerLookup"/>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
      "jmx" : {
        "enabled" : "true",
        "domain" : "example.com",
        "mbean-server-lookup" : "com.example.MyMBeanServerLookup"
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    statistics: "true"
    jmx:
      enabled: "true"
      domain: "example.com"
      mbeanServerLookup: "com.example.MyMBeanServerLookup"
```

-

GlobalConfigurationBuilder

```
GlobalConfiguration global = GlobalConfigurationBuilder.defaultClusteredBuilder()
   .jmx().enable()
   .domain("org.mydomain")
   .mBeanServerLookup(new com.acme.MyMBeanServerLookup());
```

4.4. 状態遷移操作中のメトリクスのエクスポート

Data Grid がノード間で再配布するクラスター化されたキャッシュのタイムメトリクスをエクスポートできます。

ノードがクラスターに参加したりクラスターから離れたりするなど、クラスター化されたキャッシュトポロジーが変更されると、状態遷移操作が発生します。状態遷移操作中に、Data Grid は各キャッシュからメトリクスをエクスポートするため、キャッシュのステータスを判断できます。状態遷移は、Data Grid が各キャッシュからメトリクスをエクスポートできるように、属性をプロパティーとして公開します。

注記

インバリデーションモードでは状態遷移操作を実行できません。

Data Grid は、REST API および JMX API と互換性のある時間メトリクスを生成します。

前提条件

- Data Grid メトリクスを設定します。
- 埋め込みキャッシュやリモートキャッシュなど、キャッシュタイプのメトリクスを有効にします。
- クラスター化されたキャッシュトポロジーを変更して、状態遷移操作を開始します。

手順

- 以下の方法のいずれかを選択します。
  - REST API を使用してメトリクスを収集するように Data Grid を設定します。
  - JMX API を使用してメトリクスを収集するように Data Grid を設定します。

関連情報

- Data Grid 統計と JMX モニタリング (Data Grid キャッシュ) の有効化と設定
- **StateTransferManager** (Data Grid 14.0 API)

4.5. クロスサイトレプリケーションのステータスの監視

バックアップの場所のサイトステータスを監視して、サイト間の通信の中断を検出します。リモートサイトのステータスが **offline** に変わると、Data Grid はバックアップの場所へのデータのレプリケーションを停止します。データが同期しなくなるため、クラスターをオンラインに戻す前に不整合を修正する

必要があります。

問題を早期に検出するには、クロスサイトのイベントの監視が必要です。以下の監視ストラテジーのいずれかを使用します。

- REST API を使用したクロスサイトレプリケーションの監視
- Prometheus メトリクスまたはその他のモニタリングシステムを使用したクロスサイトレプリケーションの監視

REST API を使用したクロスサイトレプリケーションの監視

REST エンドポイントを使用して、すべてのキャッシュのクロスサイトレプリケーションのステータスを監視します。カスタムスクリプトを実装して REST エンドポイントをポーリングするか、以下の例を使用できます。

前提条件

- クロスサイトレプリケーションを有効にする。

手順

1. REST エンドポイントをポーリングするスクリプトを実装します。
   以下の例は、Python スクリプトを使用して、5 秒ごとにサイトのステータスをポーリングする方法を示しています。

```
#!/usr/bin/python3
import time
import requests
from requests.auth import HTTPDigestAuth


class InfinispanConnection:

    def __init__(self, server: str = 'http://localhost:11222', cache_manager: str = 'default',
                 auth: tuple = ('admin', 'change_me')) -> None:
        super().__init__()
        self.__url = f'{server}/rest/v2/container/x-site/backups/'
        self.__auth = auth
        self.__headers = {
            'accept': 'application/json'
        }

    def get_sites_status(self):
        try:
            rsp = requests.get(self.__url, headers=self.__headers, auth=HTTPDigestAuth(self.__auth[0],
self.__auth[1]))
            if rsp.status_code != 200:
                return None
            return rsp.json()
        except:
            return None


# Specify credentials for Data Grid user with permission to access the REST endpoint
USERNAME = 'admin'
PASSWORD = 'change_me'
```

```
# Set an interval between cross-site status checks
POLL_INTERVAL_SEC = 5
# Provide a list of servers
SERVERS = [
    InfinispanConnection('http://127.0.0.1:11222', auth=(USERNAME, PASSWORD)),
    InfinispanConnection('http://127.0.0.1:12222', auth=(USERNAME, PASSWORD))
]
#Specify the names of remote sites
REMOTE_SITES = [
    'nyc'
]
#Provide a list of caches to monitor
CACHES = [
    'work',
    'sessions'
]


def on_event(site: str, cache: str, old_status: str, new_status: str):
    # TODO implement your handling code here
    print(f'site={site} cache={cache} Status changed {old_status} -> {new_status}')


def __handle_mixed_state(state: dict, site: str, site_status: dict):
    if site not in state:
        state[site] = {c: 'online' if c in site_status['online'] else 'offline' for c in CACHES}
        return

    for cache in CACHES:
        __update_cache_state(state, site, cache, 'online' if cache in site_status['online'] else 'offline')


def __handle_online_or_offline_state(state: dict, site: str, new_status: str):
    if site not in state:
        state[site] = {c: new_status for c in CACHES}
        return

    for cache in CACHES:
        __update_cache_state(state, site, cache, new_status)


def __update_cache_state(state: dict, site: str, cache: str, new_status: str):
    old_status = state[site].get(cache)
    if old_status != new_status:
        on_event(site, cache, old_status, new_status)
        state[site][cache] = new_status


def update_state(state: dict):
    rsp = None
    for conn in SERVERS:
        rsp = conn.get_sites_status()
        if rsp:
            break
    if rsp is None:
        print('Unable to fetch site status from any server')
```

```
        return

    for site in REMOTE_SITES:
        site_status = rsp.get(site, {})
        new_status = site_status.get('status')
        if new_status == 'mixed':
            __handle_mixed_state(state, site, site_status)
        else:
            __handle_online_or_offline_state(state, site, new_status)


if __name__ == '__main__':
    _state = {}
    while True:
        update_state(_state)
        time.sleep(POLL_INTERVAL_SEC)
```

サイトのステータスが **online** から **offline** に変わったり、その逆の場合、関数 **on_event** が呼び出されます。

このスクリプトを使用する場合は、以下の変数を指定する必要があります。

- **USERNAME** および **PASSWORD**: REST エンドポイントにアクセスする権限を持つ Data Grid ユーザーのユーザー名およびパスワード。
- **POLL_INTERVAL_SEC**: ポーリング間の秒数。
- **SERVERS**: このサイトの Data Grid Server の一覧。このスクリプトには必要なのは、有効な応答が 1 つだけですが、フェイルオーバーを許可するためにリストが提供されています。
- **REMOTE_SITES**: これらのサーバー上で監視するリモートサイトのリスト。
- **CACHES** : 監視するキャッシュ名のリスト。

関連情報

- REST API: バックアップの場所のステータスの取得

Prometheus メトリクスを使用したクロスサイトレプリケーションの監視

Prometheus およびその他の監視システムを使用すると、サイトのステータスが **offline** に変化したことを検出するアラートを設定できます。

ヒント

クロスサイトレイテンシーメトリックを監視すると、発生する可能性のある問題を検出しやすくなります。

前提条件

- クロスサイトレプリケーションを有効にする。

手順

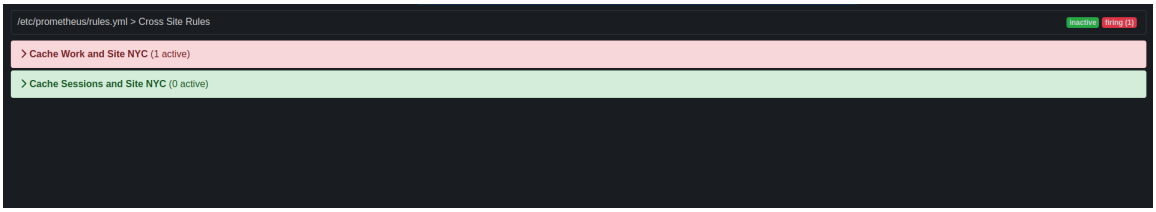
1. Data Grid メトリクスを設定します。
2. Prometheus メトリクス形式を使用してアラートルールを設定します。

- サイトの状態については、**online** の場合は **1** を、**offline** の場合は **0** を使用します。
- **expr** フィールドには、**vendor_cache_manager_default_cache_<cache name>_x_site_admin_<site name>_status** の形式を使用します。
  以下の例では、Prometheus は、**NYC** サイトが **work** または **sessions** という名前のキャッシュに対して **オフライン** になると警告を表示します。

```
groups:
- name: Cross Site Rules
  rules:
  - alert: Cache Work and Site NYC
    expr: vendor_cache_manager_default_cache_work_x_site_admin_nyc_status == 0
  - alert: Cache Sessions and Site NYC
    expr: vendor_cache_manager_default_cache_sessions_x_site_admin_nyc_status ==
0
```

次の図は、**NYC** サイトがキャッシュ **work** を行うために **offline** であるという警告を示しています。

図4.1 Prometheus 警告



関連情報

- Data Grid メトリックの設定
- Prometheus Alerting Overview
- Grafana Alerting Documentation
- Openshift Managing Alerts

第5章 DATA GRID クラストートランスポートの設定

Data Grid には、ノードがクラスターに自動的に参加および離脱できるように、トランスポート層が必要です。また、トランスポート層により、Data Grid ノードはネットワーク上でデータを複製または分散し、リバランスや状態遷移などの操作を実施することができます。

5.1. デフォルトの JGROUPS スタック

Data Grid は、**infinispan-core-14.0.21.Final-redhat-00001.jar** ファイル内の **default-configs** ディレクトリーに、デフォルトの JGroups スタックファイル **default-jgroups-*.xml** を提供します。

File name	スタック名	説明
default-jgroups-udp.xml	**udp**	トランスポートに UDP を使用し、検出に UDP マルチキャストを使用します。(100 ノードを超える) 大規模なクラスター、またはレプリケートされたキャッシュまたは無効化モードを使用している場合に適しています。オープンソケットの数を最小限に抑えます。
default-jgroups-tcp.xml	**tcp**	トランスポートには TCP を使用し、検出には **UDP** マルチキャストを使用する **MPING** プロトコルを使用します。TCP はポイントツーポイントプロトコルとして UDP よりも効率的であるため、分散キャッシュを使用している**場合にのみ**、小規模なクラスター (100 ノード未満) に適しています。
default-jgroups-kubernetes.xml	**kubernetes**	トランスポートに TCP を使用し、検出に **DNS_PING** を使用します。UDP マルチキャストが常に利用できるとは限らない Kubernetes および Red Hat OpenShift ノードに適しています。
default-jgroups-ec2.xml	**ec2**	トランスポートに TCP を使用し、検出に **aws.S3_PING** を使用します。UDP マルチキャストが利用できない Amazon EC2 ノードに適しています。追加の依存関係が必要です。
default-jgroups-google.xml	**google**	トランスポートに TCP を使用し、検出に **GOOGLE_PING2** を使用します。UDP マルチキャストが利用できない Google Cloud Platform ノードに適しています。追加の依存関係が必要です。
default-jgroups-azure.xml	**azure**	トランスポートに TCP を使用し、検出に **AZURE_PING** を使用します。UDP マルチキャストが利用できない Microsoft Azure ノードに適しています。追加の依存関係が必要です。

File name	スタック名	説明
default-jgroups-tunnel.xml	**tunnel**	トランスポートに **TUNNEL** を使用します。Data Grid がファイアウォールの背後にあり、Data Grid ノード間の直接接続できない環境に適しています。トラフィックをリダイレクトするには、外部でアクセス可能なサービス(Gossip ルーター)が必要です。Gossip ルーターおよびポートで、**jgroups.tunnel.hosts** プロパティーを **host1[port],host2[port]** の形式で設定する必要があります。

関連情報

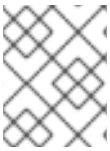
- JGroups Protocols

5.2. クラスター検出プロトコル

Data Grid は、ノードがネットワーク上でお互いを自動的に見つけてクラスターを形成できるようにするさまざまなプロトコルをサポートしています。

Data Grid が使用できる 2 種類の検出メカニズムがあります。

- ほとんどのネットワークで機能する汎用検出プロトコルで、外部サービスに依存しません。
- Data Grid クラスターのトポロジー情報を保存し、取得するために外部サービスに依存する検出プロトコル。
  たとえば、DNS_PING プロトコルは DNS サーバーレコードで検出を実行します。



注記

ホスト型プラットフォームで Data Grid を実行するには、個別のクラウドプロバイダーが課すネットワーク制約に適合する検出メカニズムを使用する必要があります。

関連情報

- JGroups Discovery Protocols
- JGroups cluster transport configuration for Data Grid 8.x (Red Hat ナレッジベースの記事)

5.2.1. PING

PING または UDPPING は、UDP プロトコルで動的なマルチキャストを使用する一般的な JGroups 検出メカニズムです。

結合時に、ノードは IP マルチキャストアドレスに PING 要求を送信し、Data Grid クラスターにある他のノードを検出します。各ノードは、コーディネーターノードのアドレスとその独自のアドレスが含まれるパケットで PING リクエストに応答します。C はコーディネーターのアドレスで、A は自分のアドレスです。ノードが PING 要求に応答すると、結合ノードは新しいクラスターのコーディネーターノードになります。

PING 設定の例

```
<PING num_discovery_runs="3"/>
```

関連情報

- JGroups PING

5.2.2. TCPPING

TCPPING は、クラスターメンバーの静的アドレスリストを使用する汎用 JGroups 検索メカニズムです。

TCPPING を使用すると、ノードが相互に動的に検出できるようにするのではなく、JGroups スタックの一部として Data Grid クラスター内の各ノードの IP アドレスまたはホスト名を手動で指定します。

TCPPING 設定の例

```
<TCP bind_port="7800" />
<TCPPING timeout="3000"
         initial_hosts="${jgroups.tcpping.initial_hosts:hostname1[port1],hostname2[port2]}"
         port_range="0"
         num_initial_members="3"/>
```

関連情報

- JGroups TCPPING

5.2.3. MPING

MPING は IP マルチキャストを使用して Data Grid クラスターの初期メンバーシップを検出します。

MPING を使用して TCPPING 検出を TCP スタックに置き換え、初期ホストの静的リストの代わりに、検出にマルチキャストを使用できます。ただし、UDP スタックで MPING を使用することもできます。

MPING 設定の例

```
<MPING mcast_addr="${jgroups.mcast_addr:239.6.7.8}"
       mcast_port="${jgroups.mcast_port:46655}"
       num_discovery_runs="3"
       ip_ttl="${jgroups.udp.ip_ttl:2}"/>
```

関連情報

- JGroups MPING

5.2.4. TCPGOSSIP

gossip ルーターは、Data Grid クラスターが他のノードのアドレスを取得できるネットワーク上の集中的な場所を提供します。

以下のように、Gosssip ルーターのアドレス (**IP:PORT**) を Data Grid ノードに挿入します。

1. このアドレスをシステムプロパティーとして JVM に渡します (例: **-DGossipRouterAddress="10.10.2.4[12001]"**)。

2. JGroups 設定ファイルのそのシステムプロパティーを参照します。

Gossip ルーター設定の例

```
<TCP bind_port="7800" />
<TCPGOSSIP timeout="3000"
        initial_hosts="${GossipRouterAddress}"
        num_initial_members="3" />
```

関連情報

- JGroups Gossip Router

5.2.5. JDBC_PING

JDBC_PING は共有データベースを使用して Data Grid クラスターに関する情報を保存します。このプロトコルは、JDBC 接続を使用できるすべてのデータベースをサポートします。

ノードは IP アドレスを共有データベースに書き込むため、ノードに結合してネットワーク上の Data Grid クラスターを検索できます。ノードが Data Grid クラスターから離脱すると、そのノードの IP アドレスが共有データベースから削除されます。

JDBC_PING 設定の例

```
<JDBC_PING connection_url="jdbc:mysql://localhost:3306/database_name"
        connection_username="user"
        connection_password="password"
        connection_driver="com.mysql.jdbc.Driver"/>
```

重要

適切な JDBC ドライバーをクラスパスに追加して、Data Grid が JDBC_PING を使用できるようにします。

5.2.5.1. JDBC_PING 検出にサーバーデータソースを使用する

管理対象データソースを Data Grid Server に追加し、これを使用してクラスタートランスポート JDBC_PING 検出プロトコルのデータベース接続を提供します。

前提条件

- Data Grid Server クラスターをインストールします。

手順

1. JDBC ドライバー JAR を Data Grid Server **server/lib** ディレクトリーにデプロイします。
2. データベースのデータソースを作成します。

```
<server xmlns="urn:infinispan:server:14.0">
 <data-sources>
   <!-- Defines a unique name for the datasource and JNDI name that you
        reference in JDBC cache store configuration.
```

```
        Enables statistics for the datasource, if required. -->
    <data-source name="ds"
                 jndi-name="jdbc/postgres"
                 statistics="true">
      <!-- Specifies the JDBC driver that creates connections. -->
      <connection-factory driver="org.postgresql.Driver"
                          url="jdbc:postgresql://localhost:5432/postgres"
                          username="postgres"
                          password="changeme">
        <!-- Sets optional JDBC driver-specific connection properties. -->
        <connection-property name="name">value</connection-property>
      </connection-factory>
      <!-- Defines connection pool tuning properties. -->
      <connection-pool initial-size="1"
                       max-size="10"
                       min-size="3"
                       background-validation="1000"
                       idle-removal="1"
                       blocking-timeout="1000"
                       leak-detection="10000"/>
    </data-source>
  </data-sources>
</server>
```

3. 検索に **JDBC_PING** プロトコルを使用する JGroups スタックを作成します。
4. **server:data-source** 属性でデータソースの名前を指定して、データソースを使用するようにクラスタートランスポートを設定します。

```
<infinispan>
   <jgroups>
      <stack name="jdbc" extends="tcp">
         <JDBC_PING stack.combine="REPLACE" stack.position="MPING" />
      </stack>
   </jgroups>
   <cache-container>
      <transport stack="jdbc" server:data-source="ds" />
   </cache-container>
</infinispan>
```

関連情報

- JDBC_PING
- JDBC_PING Wiki

5.2.6. DNS_PING

JGroups DNS_PING は DNS サーバーをクエリーし、OKD や Red Hat OpenShift などの Kubernetes 環境で Data Grid クラスターメンバーを検出します。

DNS_PING 設定の例

```
<dns.DNS_PING dns_query="myservice.myproject.svc.cluster.local" />
```

関連情報

- JGroups DNS_PING
- DNS for Services and Pods (DNS エントリーを追加するための Kubernetes ドキュメント)

5.2.7. クラウド検出プロトコル

Data Grid には、クラウドプロバイダーに固有の検出プロトコル実装を使用するデフォルトの JGroups スタックが含まれています。

検出プロトコル	デフォルトのスタックファイル	アーティファクト	バージョン
aws.S3_PING	**default-jgroups-ec2.xml**	**org.jgroups.aws:jgroups-aws**	**3.0.0.Final**
GOOGLE_PING2	**default-jgroups-google.xml**	**org.jgroups.google:jgroups-google**	**2.0.0.Final**
azure.AZURE_PING	**default-jgroups-azure.xml**	**org.jgroups.azure:jgroups-azure**	**2.0.2.Final**

クラウド検出プロトコルの依存関係の提供
aws.S3_PING、**GOOGLE_PING2**、または **azure.AZURE_PING** のクラウド検出プロトコルを使用するには、依存するライブラリーを Data Grid に提供する必要があります。

手順

- アーティファクト依存関係をプロジェクトの **pom.xml** に追加します。

続いて、JGroups スタックファイルの一部として、またはシステムプロパティーを使用して、クラウド検出プロトコルを設定できます。

関連情報

- JGroups aws.S3_PING
- JGroups GOOGLE_PING2
- JGroups azure.AZURE_PING

5.3. デフォルトの JGROUPS スタックの使用

Data Grid は JGroups プロトコルスタックを使用するため、ノードは専用のクラスターチャネルに相互に送信できるようにします。

Data Grid は、**UDP** プロトコルおよび **TCP** プロトコルに事前設定された JGroups スタックを提供します。これらのデフォルトスタックは、ネットワーク要件向けに最適化されたカスタムクラスタートランスポート設定を構築する際の開始点として使用することができます。

手順

デフォルトの JGroups スタックの 1 つを使用するには、以下のいずれかを行います。

- **infinispan.xml** ファイルの **stack** 属性を使用します。

```
<infinispan>
  <cache-container default-cache="replicatedCache">
    <!-- Use the default UDP stack for cluster transport. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="udp"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

- **addProperty()** メソッドを使用して JGroups スタックファイルを設定します。

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("qa-cluster")
        //Uses the default-jgroups-udp.xml stack for cluster transport.
        .addProperty("configurationFile", "default-jgroups-udp.xml")
        .build();
```

検証

Data Grid は、以下のメッセージをログに記録して、使用するスタックを示します。

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack udp
```

関連情報

- JGroups cluster transport configuration for Data Grid 8.x (Red Hat ナレッジベースの記事)

5.4. JGROUPS スタックのカスタマイズ

プロパティーを調整してチューニングし、ネットワーク要件に対応するクラスタートランスポート設定を作成します。

Data Grid は、設定を容易にするためにデフォルトの JGroups スタックを拡張する属性を提供します。他のプロパティーを組み合わせてデフォルトスタックからプロパティーの継承、削除、置き換えを行うことができます。

手順

1. **infinispan.xml** ファイルに新しい JGroups スタック宣言を作成します。
2. **extends** 属性を追加し、プロパティーを継承する JGroups スタックを指定します。
3. **stack.combine** 属性を使用して、継承されたスタックに設定されたプロトコルのプロパティーを変更します。
4. **stack.position** 属性を使用して、カスタムスタックの場所を定義します。
5. スタック名を **transport** 設定の **stack** 属性の値として指定します。

たとえば、以下のようにデフォルトの TCP スタックで Gosssip ルーターと対称暗号化を使用して評価できます。

```
<infinispan>
  <jgroups>
    <!-- Creates a custom JGroups stack named "my-stack". -->
    <!-- Inherits properties from the default TCP stack. -->
    <stack name="my-stack" extends="tcp">
      <!-- Uses TCPGOSSIP as the discovery mechanism instead of MPING -->
      <TCPGOSSIP initial_hosts="${jgroups.tunnel.gossip_router_hosts:localhost[12001]}"
            stack.combine="REPLACE"
            stack.position="MPING" />
      <!-- Removes the FD_SOCK2 protocol from the stack. -->
      <FD_SOCK2 stack.combine="REMOVE"/>
      <!-- Modifies the timeout value for the VERIFY_SUSPECT2 protocol. -->
      <VERIFY_SUSPECT2 timeout="2000"/>
      <!-- Adds SYM_ENCRYPT to the stack after VERIFY_SUSPECT2. -->
      <SYM_ENCRYPT sym_algorithm="AES"
                   keystore_name="mykeystore.p12"
                   keystore_type="PKCS12"
                   store_password="changeit"
                   key_password="changeit"
                   alias="myKey"
                   stack.combine="INSERT_AFTER"
                   stack.position="VERIFY_SUSPECT2" />
    </stack>
  </jgroups>
  <cache-container name="default" statistics="true">
    <!-- Uses "my-stack" for cluster transport. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="my-stack"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

6. Data Grid ログをチェックして、スタックを使用していることを確認します。

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack my-stack
```

参照

- JGroups cluster transport configuration for Data Grid 8.x (Red Hat ナレッジベースの記事)

5.4.1. 継承属性

JGroups スタックを拡張すると、継承属性により、拡張しているスタックでプロトコルやプロパティーを調整できます。

- **stack.position** は、変更するプロトコルを指定します。
- **stack.combine** は、次の値を使用して JGroups スタックを拡張します。

値	説明
COMBINE	プロトコルプロパティーをオーバーライドします。
REPLACE	プロトコルを置き換えます。
INSERT_AFTER	別のプロトコルの後にプロトコルをスタックに追加します。挿入ポイントとして指定するプロトコルには影響しません。 JGroups スタックのプロトコルは、スタック内の場所を基にして相互に影響します。**NAKACK2** がセキュリティーで保護されるように、たとえば、**SYM_ENCRYPT** プロトコルまたは **ASYM_ENCRYPT** プロトコル後に **NAKACK2** などのプロトコルを置く必要があります。
INSERT_BEFORE	別のプロトコルの前にプロトコルをスタックに挿入します。挿入ポイントとして指定するプロトコルに影響します。
REMOVE	スタックからプロトコルを削除します。

5.5. JGROUPS システムプロパティーの使用

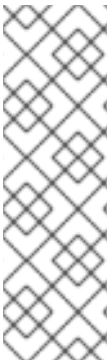
起動時にシステムプロパティーを Data Grid に渡して、クラスターのトランスポートを調整します。

手順

- **-D<property-name>=<property-value>** 引数を使用して JGroups システムプロパティーを必要に応じて設定します。

たとえば、以下のようにカスタムバインドポートと IP アドレスを設定します。

```
java -cp ... -Djgroups.bind.port=1234 -Djgroups.bind.address=192.0.2.0
```



注記

クラスター化された Red Hat JBoss EAP アプリケーションに Data Grid クラスターを組み込むと、JGroups システムプロパティーは競合したり、互いに上書きしたりする可能性があります。

たとえば、Data Grid クラスターまたは Red Hat JBoss EAP アプリケーションのいずれかに一意のバインドアドレスを設定しないでください。この場合、Data Grid と Red Hat JBoss EAP アプリケーションの両方が JGroups のデフォルトプロパティーを使用し、同じバインドアドレスを使用してクラスターを形成しようとします。

5.5.1. クラスタートランスポートプロパティー

以下のプロパティーを使用して JGroups クラスタートランスポートをカスタマイズします。

システムプロパティー	説明	デフォルト値	必須/オプション
jgroups.bind.address	クラストートランスポートのバインドアドレス。	**SITE_LOCAL**	任意
jgroups.bind.port	ソケットのバインドポート。	**7800**	任意
jgroups.mcast_addr	マルチキャストの IP アドレス (検出およびクラスター間の通信の両方)。IP アドレスは、IP マルチキャストに適した有効なクラス D アドレスである必要があります。	**239.6.7.8**	任意
jgroups.mcast_port	マルチキャストソケットのポート。	**46655**	任意
jgroups.ip_ttl	IP マルチキャストパケットの Time-to-live (TTL)この値は、パケットが破棄される前にパケットが作成できるネットワークホップの数を定義します。	2	任意
jgroups.thread_pool.min_threads	スレッドプールの最小スレッド数	0	任意
jgroups.thread_pool.max_threads	スレッドプールの最大スレッド数	200	任意
jgroups.join_timeout	結合リクエストが正常に実行されるまで待機する最大時間 (ミリ秒単位)。	2000	任意
jgroups.thread_dumps_threshold	スレッドダンプがログに記録される前にスレッドプールが満杯である必要がある回数。	10000	任意
jgroups.fd.port-offset	FD (障害検出プロトコル) ソケットの **jgroups.bind.port** ポートからのオフセット。	**50000** (port **57800**)	任意
jgroups.frag_size	メッセージの最大バイト数。それより大きいメッセージは断片化されます。	60000	任意

システムプロパティー	説明	デフォルト値	必須/オプション
jgroups.diag.enabled	JGroups 診断プローブを有効にします。	false	任意

関連情報

- JGroups system properties
- JGroups protocol list

5.5.2. クラウド検出プロトコルのシステムプロパティー

以下のプロパティーを使用して、ホストされたプラットフォームの JGroups 検出プロトコルを設定します。

5.5.2.1. Amazon EC2

aws.S3_PING を設定するためのシステムプロパティー。

システムプロパティー	説明	デフォルト値	必須/オプション
jgroups.s3.region_name	Amazon S3 リージョンの名前。	デフォルト値はありません。	任意
jgroups.s3.bucket_name	Amazon S3 バケットの名前。名前は存在し、一意でなければなりません。	デフォルト値はありません。	任意

5.5.2.2. Google Cloud Platform

GOOGLE_PING2 を設定するためのシステムプロパティー。

システムプロパティー	説明	デフォルト値	必須/オプション
jgroups.google.bucket_name	Google Compute Engine バケットの名前。名前は存在し、一意でなければなりません。	デフォルト値はありません。	必須

5.5.2.3. Azure

azure.AZURE_PING` のシステムプロパティー。

システムプロパティー	説明	デフォルト値	必須/オプション
jboss.jgroups.azure_ping.storage_account_name	Azure ストレージアカウントの名前。名前は存在し、一意でなければなりません。	デフォルト値はありません。	必須
jboss.jgroups.azure_ping.storage_access_key	Azure ストレージアクセスキーの名前。	デフォルト値はありません。	必須
jboss.jgroups.azure_ping.container	ping 情報を格納するコンテナーの有効な DNS 名。	デフォルト値はありません。	必須

5.5.2.4. OpenShift

DNS_PING のシステムプロパティー。

システムプロパティー	説明	デフォルト値	必須/オプション
jgroups.dns.query	クラスターメンバーを返す DNS レコードを設定します。	デフォルト値はありません。	必須
jgroups.dns.record	DNS レコードの種類を設定します。	A	任意

5.6. インライン JGROUPS スタックの使用

完全な JGroups スタックの定義を **infinispan.xml** ファイルに挿入することができます。

手順

- カスタム JGroups スタック宣言を **infinispan.xml** ファイルに埋め込みます。

```
<infinispan>
  <!-- Contains one or more JGroups stack definitions. -->
  <jgroups>
    <!-- Defines a custom JGroups stack named "prod". -->
    <stack name="prod">
      <TCP bind_port="7800" port_range="30" recv_buf_size="20000000"
send_buf_size="640000"/>
      <RED/>
      <MPING break_on_coord_rsp="true"
```

```
            mcast_addr="${jgroups.mping.mcast_addr:239.2.4.6}"
            mcast_port="${jgroups.mping.mcast_port:43366}"
            num_discovery_runs="3"
            ip_ttl="${jgroups.udp.ip_ttl:2}"/>
      <MERGE3 />
      <FD_SOCK2 />
      <FD_ALL3 timeout="3000" interval="1000" timeout_check_interval="1000" />
      <VERIFY_SUSPECT2 timeout="1000" />
      <pbcast.NAKACK2 use_mcast_xmit="false" xmit_interval="200"
xmit_table_num_rows="50"
                      xmit_table_msgs_per_row="1024" xmit_table_max_compaction_time="30000"
/>
      <UNICAST3 conn_close_timeout="5000" xmit_interval="200" xmit_table_num_rows="50"
              xmit_table_msgs_per_row="1024" xmit_table_max_compaction_time="30000" />
      <pbcast.STABLE desired_avg_gossip="2000" max_bytes="1M" />
      <pbcast.GMS print_local_addr="false" join_timeout="${jgroups.join_timeout:2000}" />
      <UFC max_credits="4m" min_threshold="0.40" />
      <MFC max_credits="4m" min_threshold="0.40" />
      <FRAG4 />
    </stack>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Uses "prod" for cluster transport. -->
    <transport cluster="${infinispan.cluster.name}"
          stack="prod"
          node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

5.7. 外部 JGROUPS スタックの使用

infinispan.xml ファイルでカスタム JGroups スタックを定義する外部ファイルを参照します。

手順

1. カスタム JGroups スタックファイルをアプリケーションクラスパスに配置します。
   または、外部スタックファイルを宣言する際に絶対パスを指定することもできます。
2. **stack-file** 要素を使用して、外部スタックファイルを参照します。

```
<infinispan>
  <jgroups>
    <!-- Creates a "prod-tcp" stack that references an external file. -->
    <stack-file name="prod-tcp" path="prod-jgroups-tcp.xml"/>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Use the "prod-tcp" stack for cluster transport. -->
    <transport stack="prod-tcp" />
    <replicated-cache name="replicatedCache"/>
  </cache-container>
  <!-- Cache configuration goes here. -->
</infinispan>
```

TransportConfigurationBuilder クラスで **addProperty()** メソッドを使用して、以下のようにカスタム JGroups スタックファイルを指定することもできます。

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("prod-cluster")
        //Uses a custom JGroups stack for cluster transport.
        .addProperty("configurationFile", "my-jgroups-udp.xml")
        .build();
```

この例では、**my-jgroups-udp.xml** は、以下のようなカスタムプロパティーで UDP スタックを参照します。

カスタム UDP スタックの例

```
<config xmlns="urn:org:jgroups"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:org:jgroups http://www.jgroups.org/schema/jgroups-4.2.xsd">
   <UDP bind_addr="${jgroups.bind_addr:127.0.0.1}"
        mcast_addr="${jgroups.udp.mcast_addr:239.0.2.0}"
        mcast_port="${jgroups.udp.mcast_port:46655}"
        tos="8"
        ucast_recv_buf_size="20000000"
        ucast_send_buf_size="640000"
        mcast_recv_buf_size="25000000"
        mcast_send_buf_size="640000"
        bundler.max_size="64000"
        ip_ttl="${jgroups.udp.ip_ttl:2}"
        diag.enabled="false"
        thread_naming_pattern="pl"
        thread_pool.enabled="true"
        thread_pool.min_threads="2"
        thread_pool.max_threads="30"
        thread_pool.keep_alive_time="5000" />
   <!-- Other JGroups stack configuration goes here. -->
</config>
```

関連情報

- org.infinispan.configuration.global.TransportConfigurationBuilder

5.8. カスタム JCHANNELS の使用

以下の例のように、カスタム JGroups JChannels を構築します。

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
JChannel jchannel = new JChannel();
// Configure the jchannel as needed.
JGroupsTransport transport = new JGroupsTransport(jchannel);
global.transport().transport(transport);
new DefaultCacheManager(global.build());
```

注記

Data Grid は、すでに接続されているカスタム JChannels を使用できません。

関連情報

- JGroups JChannel

5.9. クラストランスポートの暗号化

ノードが暗号化されたメッセージと通信できるように、クラストランスポートを保護します。また、有効なアイデンティティーを持つノードのみが参加できるように、証明書認証を実行するように Data Grid クラスターを設定することもできます。

5.9.1. JGroups 暗号化プロトコル

クラストラフィックのセキュリティーを保護するには、Data Grid ノードを設定し、シークレットキーで JGroups メッセージペイロードを暗号化します。

Data Grid ノードは、以下のいずれかから秘密鍵を取得できます。

- コーディネーターノード (非対称暗号化)
- 共有キーストア (対称暗号化)

コーディネーターノードからの秘密鍵の取得

非対称暗号化は、Data Grid 設定の JGroups スタックに **ASYM_ENCRYPT** プロトコルを追加して対称暗号化を設定します。これにより、Data Grid クラスターはシークレットキーを生成して配布できます。

重要

非対称暗号化を使用する場合は、ノードが証明書認証を実行し、シークレットキーを安全に交換できるようにキーストアを提供する必要もあります。これにより、中間者 (MitM) 攻撃からクラスターが保護されます。

非対称暗号化は、以下のようにクラストラフィックのセキュリティーを保護します。

1. Data Grid クラスターの最初のノードであるコーディネーターノードは、秘密鍵を生成します。
2. 参加ノードは、コーディネーターとの証明書認証を実行して、相互に ID を検証します。
3. 参加ノードは、コーディネーターノードに秘密鍵を要求します。その要求には、参加ノードの公開鍵が含まれています。
4. コーディネーターノードは、秘密鍵を公開鍵で暗号化し、参加ノードに返します。
5. 参加ノードは秘密鍵を復号してインストールします。
6. ノードはクラスターに参加し、秘密鍵でメッセージを暗号化および復号化します。

共有キーストアからの秘密鍵の取得

対称暗号化は、Data Grid 設定の JGroups スタックに **SYM_ENCRYPT** プロトコルを追加して対称暗号化を設定します。これにより、Data Grid クラスターは、指定したキーストアから秘密鍵を取得できます。

1. ノードは、起動時に Data Grid クラスパスのキーストアから秘密鍵をインストールします。

2. ノードはクラスターに参加し、秘密鍵でメッセージを暗号化および復号化します。

非対称暗号化と対称暗号化の比較

証明書認証を持つ **ASYM_ENCRYPT** は、**SYM_ENCRYPT** と比較して、暗号化の追加の層を提供します。秘密鍵のコーディネーターノードへのリクエストを暗号化するキーストアを提供します。Data Grid は、そのシークレットキーを自動的に生成し、クラスタートラフィックを処理し、秘密鍵の生成時に指定します。たとえば、ノードが離れる場合に新規のシークレットキーを生成するようにクラスターを設定できます。これにより、ノードが証明書認証を回避して古いキーで参加できなくなります。

一方、**SYM_ENCRYPT** は **ASYM_ENCRYPT** よりも高速です。ノードがクラスターコーディネーターとキーを交換する必要がないためです。**SYM_ENCRYPT** への潜在的な欠点は、クラスターのメンバーシップの変更時に新規シークレットキーを自動的に生成するための設定がないことです。ユーザーは、ノードがクラスタートラフィックを暗号化するのに使用するシークレットキーを生成して配布する必要があります。

5.9.2. 非対称暗号化を使用したクラストートランスポートのセキュア化

Data Grid クラスターを設定し、JGroups メッセージを暗号化するシークレットキーを生成して配布します。

手順

1. Data Grid がノードの ID を検証できるようにする証明書チェーンでキーストアを作成します。
2. クラスター内の各ノードのクラスパスにキーストアを配置します。
   Data Grid Server の場合は、$RHDG_HOME ディレクトリーにキーストアを配置します。
3. 以下の例のように、**SSL_KEY_EXCHANGE** プロトコルおよび **ASYM_ENCRYPT** プロトコルを Data Grid 設定の JGroups スタックに追加します。

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP
stack. -->
    <stack name="encrypt-tcp" extends="tcp">
      <!-- Adds a keystore that nodes use to perform certificate authentication. -->
      <!-- Uses the stack.combine and stack.position attributes to insert
SSL_KEY_EXCHANGE into the default TCP stack after VERIFY_SUSPECT2. -->
      <SSL_KEY_EXCHANGE keystore_name="mykeystore.jks"
                        keystore_password="changeit"
                        stack.combine="INSERT_AFTER"
                        stack.position="VERIFY_SUSPECT2"/>
      <!-- Configures ASYM_ENCRYPT -->
      <!-- Uses the stack.combine and stack.position attributes to insert ASYM_ENCRYPT into
the default TCP stack before pbcast.NAKACK2. -->
      <!-- The use_external_key_exchange = "true" attribute configures nodes to use the
`SSL_KEY_EXCHANGE` protocol for certificate authentication. -->
      <ASYM_ENCRYPT asym_keylength="2048"
                    asym_algorithm="RSA"
                    change_key_on_coord_leave = "false"
                    change_key_on_leave = "false"
                    use_external_key_exchange = "true"
                    stack.combine="INSERT_BEFORE"
                    stack.position="pbcast.NAKACK2"/>
    </stack>
```

```
  </jgroups>
  <cache-container name="default" statistics="true">
   <!-- Configures the cluster to use the JGroups stack. -->
   <transport cluster="${infinispan.cluster.name}"
              stack="encrypt-tcp"
              node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

検証

Data Grid クラスターを起動した際、以下のログメッセージは、クラスターがセキュアな JGroups スタックを使用していることを示しています。

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack
<encrypted_stack_name>
```

Data Grid ノードは **ASYM_ENCRYPT** を使用している場合のみクラスターに参加でき、コーディネーターノードからシークレットキーを取得できます。それ以外の場合は、次のメッセージが Data Grid ログに書き込まれます。

```
[org.jgroups.protocols.ASYM_ENCRYPT] <hostname>: received message without encrypt header
from <hostname>; dropping it
```

関連情報

- JGroups 4 Manual
- JGroups 4.2 Schema

5.9.3. 対称暗号化を使用したクラストランスポートのセキュア化

指定したキーストアからの秘密鍵を使用して JGroups メッセージを暗号化するように Data Grid クラスターを設定します。

手順

1. シークレットキーが含まれるキーストアを作成します。
2. クラスター内の各ノードのクラスパスにキーストアを配置します。
   Data Grid Server の場合は、$RHDG_HOME ディレクトリーにキーストアを配置します。
3. Data Grid 設定の JGroups スタックに **SYM_ENCRYPT** プロトコルを追加します。

```
<infinispan>
 <jgroups>
  <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP stack. -->
  <stack name="encrypt-tcp" extends="tcp">
   <!-- Adds a keystore from which nodes obtain secret keys. -->
   <!-- Uses the stack.combine and stack.position attributes to insert SYM_ENCRYPT into the
default TCP stack after VERIFY_SUSPECT2. -->
   <SYM_ENCRYPT keystore_name="myKeystore.p12"
                keystore_type="PKCS12"
                store_password="changeit"
```

```
                key_password="changeit"
                alias="myKey"
                stack.combine="INSERT_AFTER"
                stack.position="VERIFY_SUSPECT2"/>
    </stack>
  </jgroups>
  <cache-container name="default" statistics="true">
    <!-- Configures the cluster to use the JGroups stack. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="encrypt-tcp"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

検証

Data Grid クラスターを起動した際、以下のログメッセージは、クラスターがセキュアな JGroups スタックを使用していることを示しています。

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack
<encrypted_stack_name>
```

Data Grid ノードは、**SYM_ENCRYPT** を使用し、共有キーストアからシークレットキーを取得できる場合に限りクラスターに参加できます。それ以外の場合は、次のメッセージが Data Grid ログに書き込まれます。

```
[org.jgroups.protocols.SYM_ENCRYPT] <hostname>: received message without encrypt header from
<hostname>; dropping it
```

関連情報

- JGroups 4 Manual
- JGroups 4.2 Schema

5.10. クラスタートラフィックの TCP および UDP ポート

Data Grid は、クラスタートランスポートメッセージに以下のポートを使用します。

デフォルトのポート	プロトコル	説明
7800	TCP/UDP	JGroups クラスターバインドポート
46655	UDP	JGroups マルチキャスト

クロスサイトレプリケーション

Data Grid は、JGroups RELAY2 プロトコルに以下のポートを使用します。

7900

OpenShift で実行している Data Grid クラスターの向け。

7800

ノード間のトラフィックに UDP を使用し、クラスター間のトラフィックに TCP を使用する場合。

7801

ノード間のトラフィックに TCP を使用し、クラスター間のトラフィックに TCP を使用する場合。

第6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。

6.1. 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

- EmbeddedClusteredLockManagerFactory
- ClusteredLockManager
- ClusteredLock

6.2. 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();
});
```

6.3. 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:14.0
https://infinispan.org/schemas/infinispan-config-14.0.xsd"
      xmlns="urn:infinispan:config:14.0">

   <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:14.0"
                       num-owners = "3"
                       reliability="AVAILABLE">
         <clustered-lock name="lock1" />
         <clustered-lock name="lock2" />
      </clustered-locks>
   </cache-container>
   <!-- Cache configuration goes here. -->
</infinispan>
```

参照

- ClusteredLockManagerConfiguration
- Clustered Locks Configuration Schema

第7章 グリッドでのコード実行

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。

7.1. 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 のストリーミング API を中心に構築されているため、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。

7.1.1. 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(...)
```

7.1.2. 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。

7.1.3. 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(...)
```

7.1.3.1. 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。

7.1.4. 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8章 コード実行のための STREAMS API の使用

Streams API を使用して、Data Grid キャッシュに格納されたデータを効率的に処理します。

第9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。

9.1. 一般的なストリーム操作

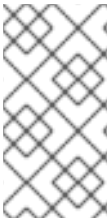
本セクションでは、使用している基礎となるキャッシュの種類に関係なく、さまざまなオプションを説明します。

9.2. 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。

9.3. 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。

9.4. 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。

9.5. 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));
```

9.6. 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。

9.6.1. 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。

9.6.2. 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終端メソッドをすべてオーバーライドして、Serializable バージョンの引数 (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;
    }
  }
```

9.7. 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。

9.8. 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。

9.9. 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。

9.10. 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、ソートされた中間操作が必要になる点で特殊です。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。

9.11. 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。

9.12. 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。

9.13. 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10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。

10.1. 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>
```

10.2. 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 jakarta.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 jakarta.transaction.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 jakarta.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;
    }
}
```

10.3. 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 jakarta.transaction.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。

10.4. 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 jakarta.transaction.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11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。

11.1. 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。

11.1.1. インストールと設定

pom.xml

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

11.1.2. 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。

11.1.3. 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。Data Grid キャッシュの設定 の通常のキャッシュの設定方法を参照してください。

11.1.3.1. 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);
```

11.1.4. 制限事項

ほとんどの場合、Multimap キャッシュは通常のキャッシュとして動作しますが、以下のように現在のバージョンにはいくつかの制限があります。

11.1.4.1. 重複のサポート

1 つのキーの重複値を格納するようにマルチマップを設定できます。重複は、値の **equals** メソッドによって決定されます。put メソッドが呼び出されるたびに、マルチマップが重複をサポートするように設定されている場合、キーと値のペアがコレクションに追加されます。マルチマップで remove を呼び出すと、存在する場合はすべての重複が削除されます。

11.1.4.2. 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。

11.1.4.3. 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。