Red Hat Data Grid 8.3

Hot Rod Java クライアントガイド

Hot Rod Java クライアントの設定および使用

Last Updated: 2023-12-06

Red Hat Data Grid 8.3 Hot Rod Java クライアントガイド

Hot Rod Java クライアントの設定および使用

法律上の通知

概要

Hot Rod Java クライアントは、Data Grid クラスターへの高性能リモートアクセスを提供します。

目次

RED HAT DATA GRID

Data Grid は、高性能の分散型インメモリーデータストアです。

スキーマレスデータ構造

さまざまなオブジェクトをキーと値のペアとして格納する柔軟性があります。

グリッドベースのデータストレージ

クラスター間でデータを分散および複製するように設計されています。

エラスティックスケーリング

サービスを中断することなく、ノードの数を動的に調整して要件を満たします。

データの相互運用性

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

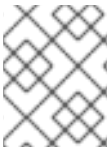
DATA GRID のドキュメント

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

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

DATA GRID のダウンロード

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



注記

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

多様性を受け入れるオープンソースの強化

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

第1章 HOT ROD JAVA クライアント

Hot Rod Java クライアント API を介してリモートで Data Grid にアクセスします。

1.1. HOT ROD プロトコル

Hot Rod は、Data Grid が次の機能を備えた高性能のクライアントサーバー相互作用を提供するバイナリー TCP プロトコルです。

- 負荷分散Hot Rod クライアントは、さまざまな戦略を使用して、Data Grid クラスター間でリクエストを送信できます。
- フェイルオーバーHot Rod クライアントは、Data Grid クラストポロジーの変更を監視し、使用可能なノードに自動的に切り替えることができます。
- 効率的なデータの場所。Hot Rod クライアントは、主要な所有者を見つけてそれらのノードに直接要求を行うことができるため、待ち時間が短縮されます。

1.2. クライアントのインテリジェンス

Hot Rod クライアントは、インテリジェンスメカニズムを使用して、Data Grid Server クラスターにリクエストを効率的に送信します。デフォルトでは、Hot Rod プロトコルでは **HASH_DISTRIBUTION_AWARE** インテリジェンスメカニズムが有効になっています。

BASIC インテリジェンス

クライアントは、ノードの参加や離脱など、Data Grid クラスターのトポロジー変更イベントを受け取らず、クライアント設定に追加した Data Grid サーバーネットワークの場所のリストのみを使用します。

注記

Data Grid サーバーが内部および非表示のクラストポロジーを Hot Rod クライアントに送信しない場合に、**BASIC** インテリジェンスが Hot Rod クライアント設定を使用できるようにします。

TOPOLOGY_AWARE インテリジェンス

クライアントは、Data Grid クラスターのトポロジー変更イベントを受け取って保存し、ネットワーク上の Data Grid サーバーを動的に追跡します。

クラストポロジーを受け取るために、クライアントは起動時に少なくとも 1 つの Hot Rod サーバーのネットワークロケーション (IP アドレスまたはホスト名) を必要とします。クライアントが接続した後、Data Grid Server はトポロジーをクライアントに送信します。Data Grid Server ノードがクラスターに参加またはクラスターから離脱すると、Data Grid は更新されたトポロジーをクライアントに送信します。

HASH_DISTRIBUTION_AWARE インテリジェンス

クライアントは、特定のキーを格納しているノードをクライアントが識別できるようにするハッシュ情報に加えて、Data Grid クラスターのトポロジー変更イベントを受け取って格納します。

たとえば、**put(k,v)** オペレーションについて考えてみましょう。クライアントはキーのハッシュ値を計算して、データが存在する正確な Data Grid Server ノードを見つけられるようにします。その後、クライアントはそのノードに直接接続して、読み取りおよび書き込み操作を実行できます。

HASH_DISTRIBUTION_AWARE インテリジェンスの利点は、Data Grid Server がキーハッシュに基づいて値を検索する必要がないことです。これにより、サーバー側のリソースの使用量が少なくなります。もう 1 つの利点は、クライアントが追加のネットワークラウンドトリップを行う必要がないため、Data Grid Server がクライアントの要求により迅速に応答することです。

設定

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clientIntelligence(ClientIntelligence.BASIC);
```

hotrod-client.properties

```
infinispan.client.hotrod.client_intelligence=BASIC
```

関連情報

- org.infinispan.client.hotrod.configuration.ClientIntelligence

1.3. バランシングの要求

Hot Rod Java クライアントは、Data Grid サーバークラスターへの要求のバランスを取り、読み取りおよび書き込み操作がノード全体に分散されるようにします。

BASIC または **TOPOLOGY_AWARE** インテリジェンスを使用するクライアントは、すべての要求に対して要求バランシングを使用します。**HASH_DISTRIBUTION_AWARE** インテリジェンスを使用するクライアントは、目的のキーを格納するノードに直接要求を送信します。ノードが応答しない場合、クライアントはフォールバックしてバランシングを要求します。

デフォルトのバランシング戦略はラウンドロビンであるため、Hot Rod クライアントは次の例のようにリクエストバランシングを実行します。ここで、 **s1**、**s2**、**s3** は Data Grid クラスター内のノードです。

```
// Connect to the Data Grid cluster
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());
// Obtain the remote cache
RemoteCache<String, String> cache = cacheManager.getCache("test");

//Hot Rod client sends a request to the "s1" node
cache.put("key1", "aValue");
//Hot Rod client sends a request to the "s2" node
cache.put("key2", "aValue");
//Hot Rod client sends a request to the "s3" node
String value = cache.get("key1");
//Hot Rod client sends the next request to the "s1" node again
cache.remove("key2");
```

カスタムバランシングポリシー

Hot Rod クライアント設定にクラスを追加する場合は、カスタムの **FailoverRequestBalancingStrategy** 実装を使用できます。

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer()
        .host("127.0.0.1")
        .port(11222)
        .balancingStrategy(new MyCustomBalancingStrategy());
```

hotrod-client.properties

```
infinispan.client.hotrod.request_balancing_strategy=my.package.MyCustomBalancingStrategy
```

関連情報

- org.infinispan.client.hotrod.FailoverRequestBalancingStrategy

1.4. クライアントフェイルオーバー

Hot Rod クライアントは、Data Grid クラスタートポロジーが変更されたときに自動的にフェイルオーバーできます。たとえば、トポロジー対応の Hot Rod クライアントは、1 つ以上の Data Grid サーバーに障害が発生したことを検出できます。

クラスター化された Data Grid サーバー間のフェイルオーバーに加えて、HotRod クライアントは Data Grid クラスター間でフェイルオーバーをすることができます。

たとえば、ニューヨーク (**NYC**) で実行している Data Grid クラスターと、ロンドン (**LON**) で実行している別のクラスターがあるとします。**NYC**に要求を送信するクライアントは、使用可能なノードがないことを検出したため、**LON**のクラスターに切り替えます。その後、クライアントは、手動でクラスターを切り替えるか、フェイルオーバーが再度発生するまで、**LON**への接続を維持します。

フェイルオーバーを伴うトランザクションキャッシュ

putIfAbsent()、**replace()**、**remove()** などの条件付きオペレーションには、厳密なメソッドの戻り保証があります。同様に、一部のオペレーションでは、以前の値を返さないといけない場合があります。

Hot Rod クライアントはフェイルオーバーが可能ですが、トランザクションキャッシュを使用して、操作が部分的に完了せず、異なるノードに競合するエントリーが残らないようにする必要があります。

1.5. HOT ROD クライアントの DATA GRID SERVER との互換性

Data Grid Server を使用すると、異なるバージョンの Hot Rod クライアントを接続することができます。たとえば、Data Grid クラスターへの移行またはアップグレードの際に、Hot Rod クライアントのバージョンが Data Grid Server よりも低い Data Grid バージョンになることがあります。

ヒント

Data Grid は、最新の機能およびセキュリティー機能強化の恩恵を受けるために、最新の Hot Rod クライアントバージョンを使用することを推奨しています。

Data Grid 8 以降

Hot Rod プロトコルバージョン 3.x は、Data Grid Server のクライアントに対して、可能な限り高いバージョンを自動的にネゴシエートします。

Data Grid 7.3 以前

Data Grid Server バージョンよりも高い Hot Rod プロトコルバージョンを使用するクライアントは、**infinispan.client.hotrod.protocol_version** プロパティーを設定する必要があります。

関連情報

- Hot Rod protocol reference
- Connecting Hot Rod clients to servers with different versions (Red Hat ナレッジベース)

第2章 DATA GRID MAVEN リポジトリーの設定

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

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

2.1. DATA GRID MAVEN リポジトリーのダウンロード

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

手順

1. Red Hat カスタマーポータルにログインします。
2. Software Downloads for Data Grid に移動します。
3. Red Hat Data Grid 8.3 Maven リポジトリーをダウンロードします。
4. アーカイブされた Maven リポジトリーをローカルファイルシステムに展開します。
5. **README.md** ファイルを開き、適切なインストール手順に従います。

2.2. RED HAT MAVEN リポジトリーの追加

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

手順

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

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

参照資料

- Red Hat Enterprise Maven Repository

2.3. DATA GRID POM の設定

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

手順

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

以下の例は、Data Grid のバージョンと BOM を示しています。

```
<properties>
  <version.infinispan>13.0.10.Final-redhat-00001</version.infinispan>
</properties>

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

次のステップ

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

第3章 HOT ROD JAVA クライアント設定

Data Grid は、設定プロパティーを公開する Hot Rod Java クライアント設定 API を提供します。

3.1. HOT ROD JAVA クライアントの依存関係の追加

Hot Rod Java クライアントの依存関係を追加して、プロジェクトに含めます。

前提条件

- Java 8 または Java 11

手順

- 次のように、**pom.xml** の依存関係として **infinispan-client-hotrod** アーティファクトを追加します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-client-hotrod</artifactId>
</dependency>
```

参照資料

Data Grid Server の要件

3.2. HOT ROD クライアント接続の設定

Data Grid Server への Hot Rod Java クライアント接続を設定します。

手順

- **RemoteCacheManager** に渡すか、アプリケーションのクラスパスの **hotrod-client.properties** ファイルを使用できることを不変の設定オブジェクトを生成する **ConfigurationBuilder** をクラス使用します。

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer()
        .host("127.0.0.1")
        .port(ConfigurationProperties.DEFAULT_HOTROD_PORT)
       .addServer()
        .host("192.0.2.0")
        .port(ConfigurationProperties.DEFAULT_HOTROD_PORT)
       .security().authentication()
        .username("username")
        .password("changeme")
        .realm("default")
        .saslMechanism("SCRAM-SHA-512");
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());
```

hotrod-client.properties

```
infinispan.client.hotrod.server_list = 127.0.0.1:11222,192.0.2.0:11222
infinispan.client.hotrod.auth_username = username
infinispan.client.hotrod.auth_password = changeme
infinispan.client.hotrod.auth_realm = default
infinispan.client.hotrod.sasl_mechanism = SCRAM-SHA-512
```

Hot Rod URI の設定

次のように、URI を使用して Hot Rod クライアント接続を設定することもできます。

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.uri("hotrod://username:changeme@127.0.0.1:11222,192.0.2.0:11222?
auth_realm=default&sasl_mechanism=SCRAM-SHA-512");
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());
```

hotrod-client.properties

```
infinispan.client.hotrod.uri = hotrod://username:changeme@127.0.0.1:11222,192.0.2.0:11222?
auth_realm=default&sasl_mechanism=SCRAM-SHA-512
```

クラスパスの外部へのプロパティーの追加

hotrod-client.properties ファイルがアプリケーションのクラスパスにない場合は、次の例のように場所を指定する必要があります。

```
ConfigurationBuilder builder = new ConfigurationBuilder();
Properties p = new Properties();
try(Reader r = new FileReader("/path/to/hotrod-client.properties")) {
   p.load(r);
   builder.withProperties(p);
}
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());
```

関連情報

- Hot Rod Client Configuration
- org.infinispan.client.hotrod.configuration.ConfigurationBuilder
- org.infinispan.client.hotrod.RemoteCacheManager

3.2.1. クライアント設定での Data Grid クラスターの定義

Hot Rod クライアント設定で Data Grid クラスターの場所を指定します。

手順

- **ClusterConfigurationBuilder** クラスを使用して、少なくとも 1 つのノードのホスト名とポートとともに少なくとも 1 つの Data Grid クラスター名を指定します。

クラスターをデフォルトとして定義し、クライアントが常に最初にクラスターに接続を試みるようにする場合は、**addServers("<host_name>:<port>; <host_name>:<port>")** メソッドを使用してサーバーリストを定義します。

複数のクラスター接続

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addCluster("siteA")
               .addClusterNode("hostA1", 11222)
               .addClusterNode("hostA2", 11222)
             .addCluster("siteB")
               .addClusterNodes("hostB1:11222; hostB2:11222");
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

フェールオーバークラスターを含むデフォルトのサーバーリスト

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServers("hostA1:11222; hostA2:11222")
             .addCluster("siteB")
               .addClusterNodes("hostB1:11222; hostB2:11223");
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

3.2.2. Data Grid クラスターの手動切り替え

Data Grid クラスター間で Hot Rod Java クライアント接続を手動で切り替えます。

手順

- **RemoteCacheManager** クラスで次のいずれかのメソッドを呼び出します。
  switchToCluster(clusterName) は、クライアント設定で定義された特定のクラスターに切り替えます。

  switchToDefaultCluster() は、Data Grid サーバーのリストとして定義されているクライアント設定のデフォルトクラスターに切り替えます。

関連情報

- RemoteCacheManager

3.2.3. 接続プールの設定

Hot Rod Java クライアントは、Data Grid サーバーへの永続的な接続のプールを保持して、要求ごとに TCP 接続を作成するのではなく、TCP 接続を再利用します。

手順

- 次の例のように、Hot Rod クライアント接続プール設定を設定します。

ConfigurationBuilder

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
```

```
            .host("127.0.0.1")
            .port(11222)
          .connectionPool()
            .maxActive(10)
            exhaustedAction(ExhaustedAction.valueOf("WAIT"))
            .maxWait(1)
            .minIdle(20)
            .minEvictableIdleTime(300000)
            .maxPendingRequests(20);
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

hotrod-client.properties

```
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.connection_pool.max_active = 10
infinispan.client.hotrod.connection_pool.exhausted_action = WAIT
infinispan.client.hotrod.connection_pool.max_wait = 1
infinispan.client.hotrod.connection_pool.min_idle = 20
infinispan.client.hotrod.connection_pool.min_evictable_idle_time = 300000
infinispan.client.hotrod.connection_pool.max_pending_requests = 20
```

3.3. HOT ROD クライアントの認証メカニズムの設定

Data Grid Server は、さまざまなメカニズムを使用して Hot Rod クライアント接続を認証します。

手順

- **AuthenticationConfigurationBuilder** クラスの **saslMechanism()** メソッド、または **infinispan.client.hotrod.sasl_mechanism** プロパティーを使用して認証メカニズムを指定します。

SCRAM

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
            .host("127.0.0.1")
            .port(11222)
          .security()
            .authentication()
              .saslMechanism("SCRAM-SHA-512")
              .username("myuser")
              .password("qwer1234!");
```

DIGEST

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
            .host("127.0.0.1")
            .port(11222)
          .security()
            .authentication()
```

```
            .saslMechanism("DIGEST-MD5")
            .username("myuser")
            .password("qwer1234!");
```

PLAIN

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
            .host("127.0.0.1")
            .port(11222)
          .security()
            .authentication()
              .saslMechanism("PLAIN")
              .username("myuser")
              .password("qwer1234!");
```

OAUTHBEARER

```
String token = "..."; // Obtain the token from your OAuth2 provider
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
            .host("127.0.0.1")
            .port(11222)
          .security()
            .authentication()
              .saslMechanism("OAUTHBEARER")
              .token(token);
```

EXTERNAL

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
  .addServer()
    .host("127.0.0.1")
    .port(11222)
  .security()
    .ssl()
      // TrustStore stores trusted CA certificates for the server.
      .trustStoreFileName("/path/to/truststore")
      .trustStorePassword("truststorepassword".toCharArray())
      .trustStoreType("PCKS12")
      // KeyStore stores valid client certificates.
      .keyStoreFileName("/path/to/keystore")
      .keyStorePassword("keystorepassword".toCharArray())
      .keyStoreType("PCKS12")
    .authentication()
      .saslMechanism("EXTERNAL");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

GSSAPI

```
LoginContext lc = new LoginContext("GssExample", new BasicCallbackHandler("krb_user",
"krb_password".toCharArray()));
lc.login();
Subject clientSubject = lc.getSubject();

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
             .host("127.0.0.1")
             .port(11222)
            .security()
             .authentication()
               .saslMechanism("GSSAPI")
               .clientSubject(clientSubject)
               .callbackHandler(new BasicCallbackHandler());
```

基本的なコールバックハンドラー
BasicCallbackHandler は、GSSAPI の例に示されているように、次のコールバックを呼び出します。

- **NameCallback** および **PasswordCallback** は、クライアントサブジェクトを構築します。
- **AuthorizeCallback** は、SASL 認証中に呼び出されます。

トークンコールバックハンドラーを備えた OAUTHBEARER
次の例のように、**TokenCallbackHandler** を使用して、OAuth2 トークンが期限切れになる前に更新します。

```
String token = "..."; // Obtain the token from your OAuth2 provider
TokenCallbackHandler tokenHandler = new TokenCallbackHandler(token);
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
             .host("127.0.0.1")
             .port(11222)
            .security()
             .authentication()
             .saslMechanism("OAUTHBEARER")
             .callbackHandler(tokenHandler);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
// Refresh the token
tokenHandler.setToken("newToken");
```

カスタムの CallbackHandler
Hot Rod クライアントは、SASL メカニズムに認証情報を渡すためにデフォルトの **CallbackHandler** を設定します。次の例のように、カスタムの **CallbackHandler** を提供しないといけない場合があります。

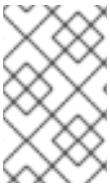
```
public class MyCallbackHandler implements CallbackHandler {
  final private String username;
  final private char[] password;
  final private String realm;

  public MyCallbackHandler(String username, String realm, char[] password) {
    this.username = username;
    this.password = password;
    this.realm = realm;
```

```
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
      for (Callback callback : callbacks) {
        if (callback instanceof NameCallback) {
          NameCallback nameCallback = (NameCallback) callback;
          nameCallback.setName(username);
        } else if (callback instanceof PasswordCallback) {
          PasswordCallback passwordCallback = (PasswordCallback) callback;
          passwordCallback.setPassword(password);
        } else if (callback instanceof AuthorizeCallback) {
          AuthorizeCallback authorizeCallback = (AuthorizeCallback) callback;
          authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationID().equals(
              authorizeCallback.getAuthorizationID()));
        } else if (callback instanceof RealmCallback) {
          RealmCallback realmCallback = (RealmCallback) callback;
          realmCallback.setText(realm);
        } else {
          throw new UnsupportedCallbackException(callback);
        }
      }
    }
}

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
            .host("127.0.0.1")
            .port(11222)
           .security().authentication()
            .serverName("myhotrodserver")
            .saslMechanism("DIGEST-MD5")
            .callbackHandler(new MyCallbackHandler("myuser","default","qwer1234!".toCharArray()));
```



注記

カスタムの **CallbackHandler** は、使用する認証メカニズムに固有のコールバックを処理する必要があります。ただし、考えられる各コールバックタイプの例を提供することは、このドキュメントの範囲を超えています。

3.3.1. GSSAPI ログインコンテキストの作成

GSSAPI メカニズムを使用するには、**LoginContext**を作成して、Hot Rod クライアント Ticket Granting Ticket (TGT) を取得できるようにする必要があります。

手順

1. ログイン設定ファイルでログインモジュールを定義します。

gss.conf

```
GssExample {
    com.sun.security.auth.module.Krb5LoginModule required client=TRUE;
};
```

IBM JDK の場合:

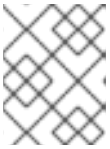
gss-ibm.conf

```
GssExample {
   com.ibm.security.auth.module.Krb5LoginModule required client=TRUE;
};
```

2. 次のシステムプロパティーを設定します。

```
java.security.auth.login.config=gss.conf

java.security.krb5.conf=/etc/krb5.conf
```



注記

krb5.conf は、KDC の場所を提供します。**kinit**コマンドを使用して、Kerberos で認証し、**krb5.conf** を確認します。

3.3.2. SASL 認証メカニズム

Data Grid Server は、Hot Rod エンドポイントで以下の SASL 認証メカニズムをサポートします。

認証メカニズム	説明	セキュリティーレルムタイプ	関連する詳細
PLAIN	プレーンテキスト形式の認証情報を使用します。**PLAIN** 認証は、暗号化された接続でのみ使用する必要があります。	プロパティーレルムおよび LDAP レルム	**BASIC** HTTP メカニズムと同様です。
DIGEST-*	ハッシュアルゴリズムとナンス値を使用します。ホットロッドコネクターは、強度の順に、**DIGEST-MD5**、**DIGEST-SHA**、**DIGEST-SHA-256**、**DIGEST-SHA-384**、および **DIGEST-SHA-512** ハッシュアルゴリズムをサポートします。	プロパティーレルムおよび LDAP レルム	**Digest** HTTP メカニズムに似ています。

認証メカニズム	説明	セキュリティーレルムタイプ	関連する詳細
SCRAM-*	ハッシュアルゴリズムとナンス値に加えて**ソルト**値を使用します。ホットロッドコネクターは、**SCRAM-SHA**、**SCRAM-SHA-256**、**SCRAM-SHA-384**、および **SCRAM-SHA-512** ハッシュアルゴリズムを強度順にサポートします。	プロパティーレルムおよび LDAP レルム	**Digest** HTTP メカニズムに似ています。
GSSAPI	Kerberos チケットを使用し、Kerberos ドメインコントローラーが必要です。対応する **Kerberos** サーバー ID をレルム設定に追加する必要があります。ほとんどの場合、ユーザーメンバーシップ情報を提供するために **ldap-realm** も指定します。	Kerberos レルム	**SPNEGO** HTTP メカニズムに似ています。
GS2-KRB5	Kerberos チケットを使用し、Kerberos ドメインコントローラーが必要です。対応する **Kerberos** サーバー ID をレルム設定に追加する必要があります。ほとんどの場合、ユーザーメンバーシップ情報を提供するために **ldap-realm** も指定します。	Kerberos レルム	**SPNEGO** HTTP メカニズムに似ています。
EXTERNAL	クライアント証明書を使用します。	トラストストアレルム	**CLIENT_CERTHTTP** メカニズムに似ています。
OAUTHBEARER	OAuth トークンを使用し、**token-realm** 設定が必要です。	トークンレルム	**BEARER_TOKEN** HTTP メカニズムに似ています。

3.4. HOT ROD クライアントの暗号化の設定

Data Grid Server は、SSL/TLS 暗号化を実施し、Hot Rod クライアントに証明書を提示して、信頼を確立し、安全な接続をネゴシエートできます。

Data Grid Server に発行された証明書を検証するには、Hot Rod クライアントでは、完全な証明書チェーンまたは Root CA で始まる部分的なチェーンのいずれかが必要です。サーバー証明書をトラストストアとして Hot Rod クライアントに提供します。

ヒント

トラストストアを提供する代わりに、共有システム証明書を使用できます。

前提条件

- Hot Rod クライアントが Data Grid Server アイデンティティーを検証するのに使用できるトラストストアを作成します。
- Data Grid Server を設定してクライアント証明書を検証または認証する場合は、必要に応じてキーストアを作成します。

手順

1. **trustStoreFileName()** メソッドおよび **trustStorePassword()** メソッドまたは対応するプロパティーを使用して、トラストストアをクライアント設定に追加します。
2. クライアント証明書認証を設定する場合は、次のようにします。
   a. **keyStoreFileName()** メソッドおよび **keyStorePassword()** メソッドまたは対応するプロパティーを使用して、クライアント設定にキーストアを追加します。
   b. **EXTERNAL** 認証メカニズムを使用するようにクライアントを設定します。

ConfigurationBuilder

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
  .addServer()
    .host("127.0.0.1")
    .port(11222)
  .security()
    .ssl()
      // Server SNI hostname.
      .sniHostName("myservername")
      // Keystore that contains the public keys for Data Grid Server.
      // Clients use the trust store to verify Data Grid Server identities.
      .trustStoreFileName("/path/to/server/truststore")
      .trustStorePassword("truststorepassword".toCharArray())
      .trustStoreType("PCKS12")
      // Keystore that contains client certificates.
      // Clients present these certificates to Data Grid Server.
      .keyStoreFileName("/path/to/client/keystore")
      .keyStorePassword("keystorepassword".toCharArray())
      .keyStoreType("PCKS12")
    .authentication()
      // Clients must use the EXTERNAL mechanism for certificate authentication.
      .saslMechanism("EXTERNAL");
```

hotrod-client.properties

```
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.use_ssl = true
infinispan.client.hotrod.sni_host_name = myservername
# Keystore that contains the public keys for Data Grid Server.
# Clients use the trust store to verify Data Grid Server identities.
infinispan.client.hotrod.trust_store_file_name = server_truststore.pkcs12
infinispan.client.hotrod.trust_store_password = changeme
infinispan.client.hotrod.trust_store_type = PCKS12
# Keystore that contains client certificates.
# Clients present these certificates to Data Grid Server.
infinispan.client.hotrod.key_store_file_name = client_keystore.pkcs12
infinispan.client.hotrod.key_store_password = changeme
infinispan.client.hotrod.key_store_type = PCKS12
# Clients must use the EXTERNAL mechanism for certificate authentication.
infinispan.client.hotrod.sasl_mechanism = EXTERNAL
```

次のステップ

クライアントトラストストアを **$RHDG_HOME/server/conf** ディレクトリーに追加し、必要に応じてそれを使用するように Data Grid Server を設定します。

関連情報

- Data Grid Server 接続の暗号化
- SslConfigurationBuilder
- Hot Rod クライアント設定プロパティー
- Using Shared System Certificates (Red Hat Enterprise Linux 7 Security Guide)

3.5. HOT ROD クライアント統計の有効化

Hot Rod Java クライアントは、リモートキャッシュヒット、ニアキャッシュヒット、ミス、および接続プールの使用状況などの統計を提供できます。

手順

1. Hot Rod Java クライアント設定を開いて編集します。
2. **true** を **statistics** プロパティーの値として指定するか、**statistics().enable()** メソッドを呼び出します。
3. **jmx** および **jmx_domain** プロパティーを使用して Hot Rod クライアントの JMX MBean をエクスポートするか、**jmxEnable()** メソッドおよび **jmxDomain()** メソッドを呼び出します。
4. クライアント設定を保存して閉じます。

Hot Rod Java クライアントの統計

ConfigurationBuilder

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

```
        .jmxDomain("my.domain.org")
      .addServer()
        .host("127.0.0.1")
        .port(11222);
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(builder.build());
```

hotrod-client.properties

```
infinispan.client.hotrod.statistics = true
infinispan.client.hotrod.jmx = true
infinispan.client.hotrod.jmx_domain = my.domain.org
```

3.6. ニアキャッシュ

ニアキャッシュは Hot Rod クライアントに対してローカルであり、最近使用されたデータを格納し、すべての読み取り操作でネットワークを通過する必要がないため、パフォーマンスが大幅に向上します。

ニアキャッシュ:

- 読み取り操作、**get()** または **getVersioned()** メソッドの呼び出しが取り込まれます。
  次の例では、**put()** 呼び出しはニアキャッシュにデータを入力せず、エントリーがすでに存在する場合にエントリーを無効にする効果しかありません。

  ```
  cache.put("k1", "v1");
  cache.get("k1");
  ```

- クライアントリスナーを登録して、Data Grid Server のリモートキャッシュでエントリーが更新または削除されたときにエントリーを無効にします。
  エントリーが無効化された後に要求された場合、クライアントはそれらをリモートキャッシュから再度取得する必要があります。

- クライアントが別のサーバーにフェイルオーバーするとクリアされます。

バインドされているニアキャッシュ

キャッシュに含めることができるエントリーの最大数を指定して、常にバウンドニアキャッシュを使用する必要があります。ニアキャッシュがエントリーの最大数に達すると、古いエントリーを削除するため自動的に削除が行われます。そのため、クライアント JVM の境界内にキャッシュサイズを手動で維持する必要はありません。

重要

ニアキャッシュ読み取りはエントリーの最終アクセス時間を伝播しないため、ニアキャッシュで最大アイドル有効期限を使用しないでください。

Bloom フィルター

Bloom フィルターは、無効化メッセージの総数を減らすことにより、書き込み操作のパフォーマンスを最適化します。

Bloom フィルター:

- Data Grid Server に存在し、クライアントが要求したエントリーを追跡します。

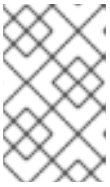
- サーバーごとに最大1つのアクティブな接続があり、 **WAIT** 枯渇アクションを使用する接続プール設定が必要です。
- 無制限のニアキャッシュでは使用できません。

3.6.1. ニアキャッシュの設定

ニアキャッシュで Hot Rod Java クライアントを設定し、最近使用されたデータをクライアント JVM に保存します。

手順

1. Hot Rod Java クライアント設定を開きます。
2. **nearCacheMode(NearCacheMode.INVALIDATED)** メソッドでニアキャッシュを実行するように各キャッシュを設定します。



注記

Data Grid は、グローバルニアキャッシュ設定プロパティーを提供します。ただし、これらのプロパティーは非推奨となり、使用すべきではありませんが、代わりにキャッシュごとにニアキャッシュを設定する必要があります。

3. **nearCacheMaxEntries()** メソッドでエビクションが発生する前に、ニアキャッシュが保持できるエントリーの最大数を指定します。
4. **nearCacheUseBloomFilter()** メソッドでニアキャッシュの bloom フィルターを有効にします。

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
import org.infinispan.client.hotrod.configuration.ExhaustedAction;

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer()
    .host("127.0.0.1")
    .port(ConfigurationProperties.DEFAULT_HOTROD_PORT)
  .security().authentication()
    .username("username")
    .password("password")
    .realm("default")
    .saslMechanism("SCRAM-SHA-512")
  // Configure the connection pool for bloom filters.
  .connectionPool()
    .maxActive(1)
    .exhaustedAction(ExhaustedAction.WAIT);
// Configure near caching for specific caches
builder.remoteCache("books")
    .nearCacheMode(NearCacheMode.INVALIDATED)
    .nearCacheMaxEntries(100)
    .nearCacheUseBloomFilter(false);
builder.remoteCache("authors")
    .nearCacheMode(NearCacheMode.INVALIDATED)
    .nearCacheMaxEntries(200)
    .nearCacheUseBloomFilter(true);
```

–

関連情報

- org.infinispan.client.hotrod.configuration.NearCacheConfiguration
- org.infinispan.client.hotrod.configuration.ExhaustedAction

3.7. 戻り値の強制

不必要にデータを送信しないようにするために、リモートキャッシュでの書き込み操作は、以前の値ではなく **null** を返します。

たとえば、以下のメソッド呼び出しでは、キーに対する以前の値は返されません。

```
V remove(Object key);
V put(K key, V value);
```

ただし、デフォルトの動作を変更して、呼び出しがキーの以前の値を返すようにすることができます。

手順

- 次のいずれかの方法で、メソッド呼び出しがキーの以前の値を返すように Hot Rod クライアントを設定します。

FORCE_RETURN_VALUE フラグ

```
cache.withFlags(Flag.FORCE_RETURN_VALUE).put("aKey", "newValue")
```

キャッシュごと

```
ConfigurationBuilder builder = new ConfigurationBuilder();
// Return previous values for keys for invocations for a specific cache.
builder.remoteCache("mycache")
      .forceReturnValues(true);
```

hotrod-client.properties

```
# Use the "*" wildcard in the cache name to return previous values
# for all caches that start with the "somecaches" string.

infinispan.client.hotrod.cache.somecaches*.force_return_values = true
```

関連情報

- org.infinispan.client.hotrod.Flag

3.8. HOT ROD クライアントからのリモートキャッシュの作成

Data Grid Hot Rod API を使用して、Java、C++、.NET/C#、JS クライアントなどから Data Grid Server にリモートキャッシュを作成します。

この手順では、最初のアクセスでリモートキャッシュを作成する Hot Rod Java クライアントを使用する方法を示します。他の Hot Rod クライアントのコード例は、Data Grid Tutorials を参照してください。

前提条件

- **admin** パーミッションを持つ Data Grid ユーザーを作成します。
- 1 つ以上の Data Grid Server インスタンスを起動します。
- Data Grid キャッシュ設定があります。

手順

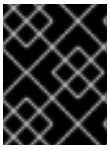
- **ConfigurationBuilder** の一部として **remoteCache()** メソッドを呼び出します。
- クラスパスの **hotrod-client.properties** ファイルで **configuration** または **configuration_uri** プロパティーを設定します。

ConfigurationBuilder

```
File file = new File("path/to/infinispan.xml")
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.remoteCache("another-cache")
      .configuration("<distributed-cache name=\"another-cache\"/>");
builder.remoteCache("my.other.cache")
      .configurationURI(file.toURI());
```

hotrod-client.properties

```
infinispan.client.hotrod.cache.another-cache.configuration=<distributed-cache name=\"another-cache\"/>
infinispan.client.hotrod.cache.[my.other.cache].configuration_uri=file:///path/to/infinispan.xml
```



重要

リモートキャッシュの名前に **.** が含まれる場合は、**hotrod-client.properties** ファイルを使用する場合は角括弧で囲む必要があります。

関連情報

- Hot Rod Client Configuration
- org.infinispan.client.hotrod.configuration.RemoteCacheConfigurationBuilder

第4章 HOT ROD クライアント API

Data Grid Hot Rod クライアント API は、リモートでのキャッシュ作成、データ操作、クラスター化されたキャッシュのトポロジー監視を行うためのインターフェイスを提供します。

4.1. REMOTECACHE API

コレクションメソッド **keySet**、**entrySet**、および **values** はリモートキャッシュによってサポートされます。つまり、すべてのメソッドが **RemoteCache** に戻されるということです。これは、さまざまなキー、エントリー、または値を遅延して取得でき、ユーザーが望まない場合にそれらすべてを一度にクライアントメモリーに保存する必要がないため便利です。

これらのコレクションは、**add** および **addAll** はサポートされていませんが、他のすべてのメソッドはサポートされているという **Map** 仕様に準拠しています。

注記の 1 つとして、**Iterator.remove** メソッドおよび **Set.remove** メソッドまたは **Collection.remove** メソッドには、操作するサーバーに 1 つ以上のラウンドトリップが必要です。RemoteCache Javadoc を参照して、これらの詳細と他のメソッドの詳細を確認できます。

イテレーターの使用

これらのコレクションの iterator メソッドは、以下で説明されている **retrieveEntries** を内部で使用します。**retrieveEntries** がバッチサイズの引数を取る場合。これをイテレータに提供する方法はありません。そのため、バッチサイズは、**RemoteCacheManager** の設定時に、システムプロパティー **infinispan.client.hotrod.batch_size** で設定できます。または、ConfigurationBuilder で設定できます。

また、返される **retrieveEntries** イテレーターは **Closeable** です。**keySet**、**entrySet**、および **values** からのイテレーターは **AutoCloseable** バリアントを返します。したがって、これらの Iterator の使用が終了したら、常に閉じる必要があります。

```
try (CloseableIterator<Map.Entry<K, V>> iterator = remoteCache.entrySet().iterator()) {

   }
```

バッキングコレクションではなく、ディープコピーが必要な場合

以前のバージョンの **RemoteCache** では、**keySet** のディープコピーの取得が許可されました。これは新しいバッキングマップでも可能です。コンテンツを自分でコピーするだけです。また、これまでサポートしていなかった **entrySet** および **values** を使用してこれを行うこともできます。

```
Set<K> keysCopy = remoteCache.keySet().stream().collect(Collectors.toSet());
```

4.1.1. サポートされていないメソッド

Data Grid の **RemoteCache** API は、**Cache** API で利用可能なすべてのメソッドをサポートしておらず、サポート対象外のメソッドが呼び出されると **UnsupportedOperationException** を出力します。

これらのメソッドのほとんどはリモートキャッシュ (リスナー管理操作など) では意味を持ちません。または、ローカルキャッシュでサポートされていないメソッドにも対応しません (例: containsValue)。

ConcurrentMap から継承された特定のアトミック操作も、**RemoteCache** API ではサポートされていません。以下に例を示します。

```
boolean remove(Object key, Object value);
boolean replace(Object key, Object value);
boolean replace(Object key, Object oldValue, Object value);
```

ただし、**RemoteCache** は、値オブジェクト全体ではなく、ネットワーク経由でバージョン識別子を送信する、これらのアトミック操作の代替バージョンメソッドを提供します。

参照資料

- Cache
- RemoteCache
- UnsupportedOperationException
- ConcurrentMap

4.2. リモート ITERATOR API

Data Grid は、メモリーリソースが制限されているエントリーを取得するため、またはサーバー側のフィルタリングや変換を行う予定がある場合に、リモートイテレータ API を提供します。

```
// Retrieve all entries in batches of 1000
int batchSize = 1000;
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(null,
batchSize)) {
    while(iterator.hasNext()) {
      // Do something
    }
}

// Filter by segment
Set<Integer> segments = ...
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(null, segments,
batchSize)) {
    while(iterator.hasNext()) {
      // Do something
    }
}

// Filter by custom filter
try (CloseableIterator<Entry<Object, Object>> iterator =
remoteCache.retrieveEntries("myFilterConverterFactory", segments, batchSize)) {
    while(iterator.hasNext()) {
      // Do something
    }
}
```

4.2.1. Data Grid サーバーへのカスタムフィルターのデプロイ

カスタムフィルターを Data Grid サーバーインスタンスにデプロイします。

手順

1. **KeyValueFilterConverterFactory** を拡張するファクトリーを作成します。

```
import java.io.Serializable;

import org.infinispan.filter.AbstractKeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverterFactory;
import org.infinispan.filter.NamedFactory;
import org.infinispan.metadata.Metadata;

//@NamedFactory annotation defines the factory name
@NamedFactory(name = "myFilterConverterFactory")
public class MyKeyValueFilterConverterFactory implements KeyValueFilterConverterFactory
{

  @Override
  public KeyValueFilterConverter<String, SampleEntity1, SampleEntity2>
getFilterConverter() {
    return new MyKeyValueFilterConverter();
  }
  // Filter implementation. Should be serializable or externalizable for DIST caches
  static class MyKeyValueFilterConverter extends AbstractKeyValueFilterConverter<String,
SampleEntity1, SampleEntity2> implements Serializable {
    @Override
    public SampleEntity2 filterAndConvert(String key, SampleEntity1 entity, Metadata
metadata) {
      // returning null will case the entry to be filtered out
      // return SampleEntity2 will convert from the cache type SampleEntity1
    }

    @Override
    public MediaType format() {
      // returns the MediaType that data should be presented to this converter.
      // When omitted, the server will use "application/x-java-object".
      // Returning null will cause the filter/converter to be done in the storage format.
    }
  }
}
```

2. **META-INF/services/org.infinispan.filter.KeyValueFilterConverterFactory** ファイルが含まれる JAR を作成します。このファイルには、フィルターファクトリークラス実装の完全修飾クラス名を含める必要があります。
フィルターがカスタムのキー/値クラスを使用する場合は、フィルターがキーや値のインスタンスを正しくアンマーシャリングできるように、それらを JAR ファイルに含める必要があります。

3. JAR ファイルを Data Grid サーバーのインストールディレクトリーの **server/lib** ディレクトリーに追加します。

参照資料

- KeyValueFilterConverterFactory

4.3. METADATAVALUE API

バージョン管理された操作には、**MetadataValue** インターフェイスを使用します。

次の例は、エントリーの値のバージョンが変更されていない場合にのみ発生する削除操作を示しています。

```
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
    RemoteCache<String, String> remoteCache = remoteCacheManager.getCache();

    remoteCache.put("car", "ferrari");
    VersionedValue valueBinary = remoteCache.getWithMetadata("car");

    assert remoteCache.remove("car", valueBinary.getVersion());
    assert !remoteCache.containsKey("car");
```

参照資料

- org.infinispan.client.hotrod.MetadataValue

4.4. ストリーミング API

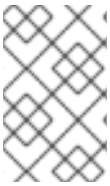
Data Grid は、**InputStream** および **OutputStream** のインスタンスを返すメソッドを実装する Streaming API を提供するため、Hot Rod クライアントと Data Grid サーバー間で大きなオブジェクトをストリーミングできます。

大きなオブジェクトの次の例を考えてみましょう。

```
StreamingRemoteCache<String> streamingCache = remoteCache.streaming();
OutputStream os = streamingCache.put("a_large_object");
os.write(...);
os.close();
```

次のように、ストリーミングを通じてオブジェクトを読み取ることができます。

```
StreamingRemoteCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
for(int b = is.read(); b >= 0; b = is.read()) {
  // iterate
}
is.close();
```



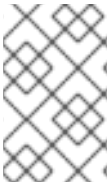
注記

Streaming API は値をマーシャリング **しません**。つまり、Streaming API と非 Streaming API の両方を同時に使用することはできません。ただし、このケースを処理するためにカスタムマーシャラーを実装することもできます。

RemoteStreamingCache.get(K key) メソッドによって返される **InputStream** は **VersionedMetadata** インターフェイスを実装しているため、以下のようにバージョンと有効期限の情報を取得できます。

```
StreamingRemoteCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
long version = ((VersionedMetadata) is).getVersion();
for(int b = is.read(); b >= 0; b = is.read()) {
```

```
   // iterate
}
is.close();
```



注記

条件付き書き込みメソッド (**putIfAbsent()**、**replace()**) は、値がサーバーに完全に送信されてから、実際の条件チェックを実行します。つまり、**close()** メソッドが **OutputStream** で呼び出される場合です。

参照資料

- org.iinfinispan.client.hotrod.StreamingRemoteCache

4.5. カウンター API

CounterManager インターフェイスは、カウンターを定義、取得、および削除するエントリーポイントです。

Hot Rod クライアントは、以下の例のように **CounterManager** インターフェイスを取得できます。

```
// create or obtain your RemoteCacheManager
RemoteCacheManager manager = ...;

// retrieve the CounterManager
CounterManager counterManager = RemoteCounterManagerFactory.asCounterManager(manager);
```

参照資料

- クラスター化カウンター

4.6. イベントリスナーの作成

Java Hot Rod クライアントは、リスナーを登録して cache-entry レベルのイベントを受信することができます。キャッシュエントリーの作成、変更、および削除されたイベントがサポートされています。

クライアントリスナーの作成は、異なるアノテーションとイベントクラスが使用される点を除き、組み込みリスナーと非常に似ています。受信した各イベントを出力するクライアントリスナーの例を次に示します。

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener(converterFactoryName = "static-converter")
public class EventPrintListener {

  @ClientCacheEntryCreated
  public void handleCreatedEvent(ClientCacheEntryCreatedEvent e) {
    System.out.println(e);
  }

  @ClientCacheEntryModified
  public void handleModifiedEvent(ClientCacheEntryModifiedEvent e) {
```

```
    System.out.println(e);
  }

  @ClientCacheEntryRemoved
  public void handleRemovedEvent(ClientCacheEntryRemovedEvent e) {
    System.out.println(e);
  }

}
```

ClientCacheEntryCreatedEvent および **ClientCacheEntryModifiedEvent** インスタンスは、影響を受けるキーとエントリーのバージョンに関する情報を提供します。このバージョンは、**replaceWithVersion**、**removeWithVersion** などのサーバー上で条件付き操作を呼び出すために使用できます。

ClientCacheEntryRemovedEvent イベントは、削除操作が成功した場合にのみ送信されます。つまり、削除オペレーションが呼び出されても、エントリーが見つからないか、エントリーを削除する必要がない場合は、イベントが生成されません。エントリーが削除されていなくても、削除されたイベントに関心のあるユーザーは、このようなイベントを生成するイベントのカスタマイズロジックを開発できます。詳細は、クライアントイベントのカスタマイズ セクションを参照してください。

ClientCacheEntryCreatedEvent、**ClientCacheEntryModifiedEvent**、および **ClientCacheEntryRemovedEvent** のすべてのイベントインスタンスも、トポロジーの変更が原因でこれを引き起こした書き込みコマンドを再度実行する場合に true を返す **boolean isCommandRetried()** メソッドも提供します。これは、このイベントが重複したか、別のイベントが破棄されて置き換えられた記号になります (例: ClientCacheEntryModifiedEvent が ClientCacheEntryCreatedEvent に置き換えます)。

クライアントリスナーの実装を作成したら、サーバーに登録する必要があります。これを行うには、以下を実行します。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());
```

4.6.1. イベントリスナーの削除

クライアントイベントリスナーが必要ない場合は、以下を削除できます。

```
EventPrintListener listener = ...
cache.removeClientListener(listener);
```

4.6.2. イベントのフィルタリング

クライアントにイベントが殺到するのを防ぐために、ユーザーはフィルタリング機能を提供して、特定のクライアントリスナーに対し、サーバーによって発生するイベントの数を制限できます。フィルターを有効にするには、フィルターインスタンスを生成するキャッシュイベントフィルターファクトリーを作成する必要があります。

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-filter")
public static class StaticCacheEventFilterFactory implements CacheEventFilterFactory {
```

```
  @Override
  public StaticCacheEventFilter getFilter(Object[] params) {
    return new StaticCacheEventFilter();
  }
}


// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventFilter implements CacheEventFilter<Integer, String>, Serializable {
  @Override
  public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
      String newValue, Metadata newMetadata, EventType eventType) {
    if (key.equals(1)) // static key
      return true;

    return false;
  }
}
```

上記で定義されたキャッシュイベントフィルターファクトリーインスタンスは、キーが **1** であるものを除くすべてのエントリーを静的にフィルターするインスタンスを作成します。

このキャッシュイベントフィルターファクトリーにリスナーを登録できるようにするには、ファクトリーに一意の名前を付ける必要があり、Hot Rod サーバーに名前とキャッシュイベントフィルターファクトリーインスタンスを接続する必要があります。

1. フィルターの実装を含む JAR ファイルを作成します。
   キャッシュがカスタムのキー/値クラスを使用する場合は、コールバックを正しくマーシャリングされていないキー/値インスタンスで実行できるように、これらを JAR に含める必要があります。クライアントリスナーで **useRawData** が有効になっている場合、コールバックのキー/値インスタンスはバイナリー形式で提供されるため、これは必要ありません。

2. JAR ファイル内で **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory** ファイルを作成し、フィルタークラス実装の完全修飾クラス名を作成します。

3. JAR ファイルを Data Grid サーバーのインストールディレクトリーの **server/lib** ディレクトリーに追加します。

4. ファクトリー名を **@ClientListener** アノテーションに追加して、クライアントリスナーをこのキャッシュイベントフィルターファクトリーとリンクします。

   ```
   @ClientListener(filterFactoryName = "static-filter")
   public class EventPrintListener { ... }
   ```

5. リスナーをサーバーに登録します。

   ```
   RemoteCache<?, ?> cache = ...
   cache.addClientListener(new EventPrintListener());
   ```

リスナーの登録時に提供されたパラメーターに基づいてフィルタリングする動的フィルターインスタンスを登録することもできます。フィルターは、フィルターファクトリーが受け取ったパラメーターを使用してこのオプションを有効にします。以下に例を示します。

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventFilter;

class DynamicCacheEventFilterFactory implements CacheEventFilterFactory {
  @Override
  public CacheEventFilter<Integer, String> getFilter(Object[] params) {
    return new DynamicCacheEventFilter(params);
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class DynamicCacheEventFilter implements CacheEventFilter<Integer, String>, Serializable {
  final Object[] params;

  DynamicCacheEventFilter(Object[] params) {
    this.params = params;
  }

  @Override
  public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
      String newValue, Metadata newMetadata, EventType eventType) {
    if (key.equals(params[0])) // dynamic key
      return true;

    return false;
  }
}
```

リスナーの登録時に、フィルタリングに必要な動的パラメーターが提供されます。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), new Object[]{1}, null);
```

警告

フィルターインスタンスは、クラスターにデプロイされるときにマーシャル可能である必要があります。これにより、リスナーの登録先の別のノードで生成された場合でも、イベントが生成される際にフィルターが適切に行われるようにします。それらをマーシャリング可能にするには、**Serializable**、**Externalizable** を拡張するか、カスタム **Externalizer** を提供します。

4.6.3. 通知のスキップ

サーバーからイベント通知を取得しなくても、リモート API メソッドを呼び出してオペレーションを実行する際に、**SKIP_LISTENER_NOTIFICATION** フラグを含めます。たとえば、値を作成または変更する際のリスナーの通知を防ぐには、フラグを以下のように設定します。

```
remoteCache.withFlags(Flag.SKIP_LISTENER_NOTIFICATION).put(1, "one");
```

4.6.4. イベントのカスタマイズ

デフォルトで生成されるイベントには、イベントを関連させるのに十分な情報が含まれていますが、送信コストを削減するために情報を詰め込みすぎないようにしています。必要に応じて、イベントに含まれる情報は、値などの詳細情報を含むか、より少ない情報を含めるためにカスタム化できます。このカスタマイズは、**CacheEventConverterFactory** によって生成された **CacheEventConverter** インスタンスを使用して行われます。

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-converter")
class StaticConverterFactory implements CacheEventConverterFactory {
  final CacheEventConverter<Integer, String, CustomEvent> staticConverter = new
StaticCacheEventConverter();
  public CacheEventConverter<Integer, String, CustomEvent> getConverter(final Object[] params) {
    return staticConverter;
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventConverter implements CacheEventConverter<Integer, String, CustomEvent>,
Serializable {
  public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata, String newValue,
Metadata newMetadata, EventType eventType) {
    return new CustomEvent(key, newValue);
  }
}

// Needs to be Serializable, Externalizable or marshallable with Infinispan Externalizers
// regardless of cluster or local caches
static class CustomEvent implements Serializable {
  final Integer key;
  final String value;
  CustomEvent(Integer key, String value) {
    this.key = key;
    this.value = value;
  }
}
```

上記の例では、コンバーターは、イベント内の値とキーを含む新しいカスタムイベントを生成します。これにより、デフォルトのイベントと比べて大量のイベントペイロードが発生しますが、フィルターと組み合わせると、ネットワークの帯域幅コストを減らすことができます。

警告

コンバーターのターゲットタイプは、**Serializable** または **Externalizable** のいずれかである必要があります。この特定のコンバーターの場合、デフォルトの Hot Rod クライアントマーシャラーではサポートされないため、デフォルトで Externalizer を提供しません。

カスタムイベントの処理には、以前に実証されたものに対して、若干異なるクライアントリスナーの実装が必要になります。より正確な状態に戻すには、**ClientCacheEntryCustomEvent** インスタンスを処理する必要があります。

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class CustomEventPrintListener {

  @ClientCacheEntryCreated
  @ClientCacheEntryModified
  @ClientCacheEntryRemoved
  public void handleCustomEvent(ClientCacheEntryCustomEvent<CustomEvent> e) {
    System.out.println(e);
  }

}
```

コールバックで受け取った **ClientCacheEntryCustomEvent** は、**getEventData** メソッドを介してカスタムイベントを公開し、**getType** メソッドは、生成されたイベントがキャッシュエントリーの作成、変更、または削除の結果であったかどうかに関する情報を提供します。

フィルタリングと同様に、リスナーをこのコンバーターファクトリーに登録できるようにするには、ファクトリーに一意の名前を付ける必要があり、Hot Rod サーバーに名前とキャッシュイベントコンバーターファクトリーインスタンスを接続する必要があります。

1. コンバーターの実装を含む JAR ファイルを作成します。
   キャッシュがカスタムのキー/値クラスを使用する場合は、コールバックを正しくマーシャリングされていないキー/値インスタンスで実行できるように、これらを JAR に含める必要があります。クライアントリスナーで **useRawData** が有効になっている場合、コールバックのキー/値インスタンスはバイナリー形式で提供されるため、これは必要ありません。

2. JAR ファイル内で **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory** ファイルを作成し、コンバータークラス実装の完全修飾クラス名を作成します。

3. JAR ファイルを Data Grid サーバーのインストールディレクトリーの **server/lib** ディレクトリーに追加します。

4. ファクトリー名を **@ClientListener** アノテーションに追加して、クライアントリスナーをこのコンバーターファクトリーとリンクします。

```
@ClientListener(converterFactoryName = "static-converter")
public class CustomEventPrintListener { ... }
```

5. リスナーをサーバーに登録します。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener());
```

リスナーの登録時に提供されたパラメーターを基に変換する動的コンバーターインスタンスもあります。コンバーターは、コンバーターファクトリーによって受け取ったパラメーターを使用してこのオプションを有効にします。以下に例を示します。

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-converter")
class DynamicCacheEventConverterFactory implements CacheEventConverterFactory {
  public CacheEventConverter<Integer, String, CustomEvent> getConverter(final Object[] params) {
    return new DynamicCacheEventConverter(params);
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed when running in a
cluster
class DynamicCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
  final Object[] params;

  DynamicCacheEventConverter(Object[] params) {
    this.params = params;
  }

  public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata,
      String newValue, Metadata newMetadata, EventType eventType) {
    // If the key matches a key given via parameter, only send the key information
    if (params[0].equals(key))
      return new CustomEvent(key, null);

    return new CustomEvent(key, newValue);
  }
}
```

変換を行うために必要な動的パラメーターは、リスナーが登録されたときに提供されます。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), null, new Object[]{1});
```

警告

コンバーターインスタンスは、クラスターにデプロイされるときにマーシャル可能である必要があります。これにより、リスナーの登録先の別のノードで生成された場合でも、イベントが生成される際に変換が適切に行われるようにします。それらをマーシャリング可能にするには、**Serializable**、**Externalizable** を拡張するか、カスタム **Externalizer** を提供します。

4.6.5. フィルターとカスタムイベント

イベントのフィルタリングとカスタマイズの両方を実行する場合、**org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter** を実装する方が簡単です。これにより、1つのステップでフィルターとカスタマイズの両方を実行できます。便宜上、**org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter** を直接実装するのではなく、**org.infinispan.notifications.cachelistener.filter.AbstractCacheEventFilterConverter** を拡張することが推奨されます。以下に例を示します。

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-filter-converter")
class DynamicCacheEventFilterConverterFactory implements CacheEventFilterConverterFactory {
  public CacheEventFilterConverter<Integer, String, CustomEvent> getFilterConverter(final Object[]
params) {
    return new DynamicCacheEventFilterConverter(params);
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed when running in a
cluster
//
class DynamicCacheEventFilterConverter extends AbstractCacheEventFilterConverter<Integer,
String, CustomEvent>, Serializable {
  final Object[] params;

  DynamicCacheEventFilterConverter(Object[] params) {
    this.params = params;
  }

  public CustomEvent filterAndConvert(Integer key, String oldValue, Metadata oldMetadata,
      String newValue, Metadata newMetadata, EventType eventType) {
    // If the key matches a key given via parameter, only send the key information
    if (params[0].equals(key))
      return new CustomEvent(key, null);

    return new CustomEvent(key, newValue);
  }
}
```

フィルターとコンバーターと同様に、この組み合わせたフィルター/変換ファクトリーでリスナーを登録するには、ファクトリーに **@NamedFactory** アノテーション経由で一意の名前が付与される必要が

あります。また、Hot Rod サーバーは名前とキャッシュイベントコンバーターファクトリーインスタンスと共にプラグインする必要があります。

1. コンバーターの実装を含む JAR ファイルを作成します。
   キャッシュがカスタムのキー/値クラスを使用する場合は、コールバックを正しくマーシャリングされていないキー/値インスタンスで実行できるように、これらを JAR に含める必要があります。クライアントリスナーで **useRawData** が有効になっている場合、コールバックのキー/値インスタンスはバイナリー形式で提供されるため、これは必要ありません。

2. JAR ファイル内で **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverterFactory** ファイルを作成し、コンバータークラス実装の完全修飾クラス名を作成します。

3. JAR ファイルを Data Grid サーバーのインストールディレクトリーの **server/lib** ディレクトリーに追加します。

クライアントの観点からすると、組み合わせたフィルターとコンバータークラスを使用できるようにするには、クライアントリスナーは同じフィルターファクトリーとコンバーターファクトリー名を定義する必要があります。以下に例を示します。

```
@ClientListener(filterFactoryName = "dynamic-filter-converter", converterFactoryName = "dynamic-filter-converter")
public class CustomEventPrintListener { ... }
```

上記の例で必要な動的パラメーターは、リスナーが filter パラメーターまたは converter パラメーターのいずれかに登録されている場合に提供されます。フィルターパラメーターが空でない場合は、それらが使用されます。それ以外の場合は、コンバーターパラメーターは以下のようになります。

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener(), new Object[]{1}, null);
```

4.6.6. イベントマーシャリング

Hot Rod サーバーは、異なる形式でデータを保存できますが、Java Hot Rod クライアントユーザーは、タイプ化されたオブジェクトで機能する **CacheEventConverter** インスタンスまたは **CacheEventFilter** インスタンスを開発できます。デフォルトでは、フィルターとコンバーターは、データを POJO (application/x-java-object) として使用しますが、filter/converter からメソッド **format()** を上書きすることで、希望の形式を上書きすることができます。形式が **null** を返す場合、フィルター/変換は保存時にデータを受け取ります。

Hot Rod Java クライアントは、異なる **org.infinispan.commons.marshall.Marshaller** インスタンスを使用するように設定できます。これ **CacheEventConverter** インスタンスや **CacheEventFilter** インスタンスをデプロイしたり、コンテンツをマーシャリングしたりするのではなく、Java オブジェクトでフィルター/変換できるようにするには、サーバーはオブジェクトとマーシャラーで生成されるバイナリー形式の間で変換できるようにする必要があります。

Marshaller インスタンスのサーバー側のデプロイするには、**CacheEventConverter** インスタンスまたは **CacheEventFilter** インスタンスをデプロイするのに使用されるものと同様の方法に従います。

1. コンバーターの実装を含む JAR ファイルを作成します。

2. JAR ファイル内で **META-INF/services/org.infinispan.commons.marshall.Marshaller** ファイルを作成し、フィルタークラス実装の完全修飾クラス名を作成します。

3. JAR ファイルを Data Grid サーバーのインストールディレクトリーの **server/lib** ディレクトリーに追加します。

Marshaller は個別の jar または **CacheEventConverter** インスタンスおよび/または **CacheEventFilter** インスタンスと同じ jar にデプロイできることに注意してください。

4.6.6.1. Protostream マーシャラーのデプロイ

キャッシュが Protobuf コンテンツを保存する場合は、Hot Rod クライアントで ProtoStream マーシャラーを使用する際に発生する場合は、サーバーで形式がすでにサポートされているため、カスタムマーシャラーをデプロイする必要はありません。Protobuf 形式から JSON や POJO などの最も一般的な形式に変換するトランスコーダーがあります。

これらのキャッシュでフィルター/変換を使用し、バイナリー Protobuf データではなく Java オブジェクトでフィルター/変換を使用すると、追加の ProtoStream マーシャラーを設定し、フィルタリング/変換の前にデータをアンマーシャリングできるようにする必要があります。これには、Data Grid サーバー設定の一部として、必要な **SerializationContextInitializer(s)** を設定する必要があります。

詳細は、Cache Encoding and Marshalling を参照してください。

4.6.7. リスナーの状態の処理

クライアントリスナーアノテーションには、リスナーの追加時またはリスナーのフェイルオーバー時に状態がクライアントに送信されるかどうかを指定する任意の **includeCurrentState** 属性があります。

デフォルトでは、**includeCurrentState** は false ですが、true に設定され、クライアントリスナーがデータが含まれるキャッシュに追加されると、サーバーはキャッシュの内容を繰り返し処理し、各エントリーのイベント (設定されている場合はカスタムイベント) を **ClientCacheEntryCreated** としてクライアントに送信します。これにより、クライアントは既存のコンテンツに基づいて一部のローカルデータ構造をビルドできます。コンテンツが繰り返されると、キャッシュの更新を受け取るため、イベントは通常どおり受信されます。キャッシュがクラスター化されている場合、クラスター全体のコンテンツ全体が繰り返し処理されます。

4.6.8. リスナーの障害処理

Hot Rod クライアントがクライアントリスナーを登録すると、クラスターの単一ノードになります。そのノードに障害が発生すると、Java Hot Rod クライアントは透過的に検出し、別のノードに失敗したノードに登録されているすべてのリスナーをフェイルオーバーします。

このフェイルオーバー中に、クライアントはいくつかのイベントを見逃す可能性があります。これらのイベントが不足しないように、クライアントリスナーアノテーションには **includeCurrentState** という任意のパラメーターが含まれます。これは、フェイルオーバーの実行時に、キャッシュの内容が繰り返し処理され、**ClientCacheEntryCreated** イベント (設定されている場合はカスタムイベント) が生成されます。デフォルトでは、**includeCurrentState** は false に設定されています。

コールバックを使用してフェイルオーバーイベントを処理します。

```
@ClientCacheFailover
public void handleFailover(ClientCacheFailoverEvent e) {
  ...
}
```

これは、クライアントがいくつかのデータをキャッシュしており、フェイルオーバーの結果、いくつかのイベントが見逃される可能性を考慮して、フェイルオーバーイベントを受信したときにローカルにキャッシュされたデータを消去することを決定し、フェイルオーバーイベントの後にキャッシュ全体の

コンテンツに対するイベントを受信することを知っているような場合に非常に便利です。

4.7. HOT ROD JAVA クライアントトランザクション

JTA の トランザクション で Hot Rod クライアントを設定および使用できます。

トランザクションに参加するために、Hot Rod クライアントは、相互作用する TransactionManager と、Synchronization インターフェイスまたは XAResource インターフェイスを通じてトランザクションに参加するかどうかを要求します。

重要

トランザクションは、クライアントが準備フェーズでエントリーへの書き込みロックを取得するという点で最適化されます。データの不整合を回避するには、トランザクションとの競合を検出 を必ずお読みください。

4.7.1. サーバーの設定

クライアントが JTA トランザクション に参加するには、サーバーのキャッシュもトランザクションである必要があります。

次のサーバー設定が必要です。それ以外の場合、トランザクションはロールバックのみになります。

- 分離レベルは **REPEATABLE_READ** である必要があります。
- **PESSIMISTIC** ロックモードが推奨されますが、**OPTIMISTIC** を使用できます。
- トランザクションモードは、**NON_XA** または **NON_DURABLE_XA** である必要があります。Hot Rod トランザクションはパフォーマンスが低下するため、**FULL_XA** を使用しないでください。

以下に例を示します。

```
<replicated-cache name="hotrodReplTx">
  <locking isolation="REPEATABLE_READ"/>
  <transaction mode="NON_XA" locking="PESSIMISTIC"/>
</replicated-cache>
```

Hot Rod トランザクションには、独自の復旧メカニズムがあります。

4.7.2. Hot Rod クライアントの設定

トランザクションの RemoteCache は、キャッシュごとに設定されます。例外は、単一のトランザクションが複数の RemoteCache と対話できるため、グローバルなトランザクションの **timeout** です。

以下の例は、キャッシュ **my-cache** にトランザクション RemoteCache を設定する方法を示しています。

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
//other client configuration parameters
cb.transactionTimeout(1, TimeUnit.MINUTES);
```

```
cb.remoteCache("my-cache")
  .transactionManagerLookup(GenericTransactionManagerLookup.getInstance())
  .transactionMode(TransactionMode.NON_XA);
```

設定パラメーターに関するドキュメントは、ConfigurationBuilder および RemoteCacheConfigurationBuilder Javadoc を参照してください。

以下の例のように、プロパティーファイルを使用して Java Hot Rod クライアントを設定することもできます。

```
infinispan.client.hotrod.cache.my-cache.transaction.transaction_manager_lookup =
org.infinispan.client.hotrod.transaction.lookup.GenericTransactionManagerLookup
infinispan.client.hotrod.cache.my-cache.transaction.transaction_mode = NON_XA
infinispan.client.hotrod.transaction.timeout = 60000
```

4.7.2.1. TransactionManagerLookup インターフェイス

TransactionManagerLookup は、TransactionManager を取得するためのエントリーポイントを提供します。

TransactionManagerLookup の の利用可能な実装:

GenericTransactionManagerLookup

Java EE アプリケーションサーバーで実行している TransactionManager を検索するルックアップクラス。TransactionManager が見つからなかった場合、デフォルトでは RemoteTransactionManager に設定されます。これは、Hot Rod Java クライアントのデフォルトです。

ヒント

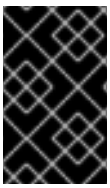
ほとんどの場合は、GenericTransactionManagerLookup が適しています。しかし、カスタム TransactionManager を統合する必要がある場合は、**TransactionManagerLookup** インターフェイスを実装できます。

RemoteTransactionManagerLookup

他の実装が利用できない場合は、基本および揮発性の TransactionManager です。この実装には、同時トランザクションと復元を処理するときに重大な制限があることに注意してください。

4.7.3. トランザクションモード

TransactionMode は RemoteCache が TransactionManager と相互作用する方法を制御します。



重要

Data Grid サーバーとクライアントアプリケーションの両方でトランザクションモードを設定します。クライアントが非トランザクションキャッシュでトランザクション操作を実行しようとすると、ランタイム例外が発生する可能性があります。

トランザクションモードは、Data Grid 設定とクライアント設定の両方で同じです。クライアントで以下のモードを使用します。サーバーの Data Grid 設定スキーマを参照してください。

NONE

RemoteCache は TransactionManager と対話しません。これはデフォルトのモードであり、非トランザクションです。

NON_XA

RemoteCache は、Synchronization を介して TransactionManager と対話します。

NON_DURABLE_XA

RemoteCache は、XAResource を介して TransactionManager と対話します。復元機能は無効になっています。

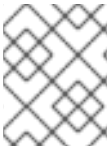
FULL_XA

RemoteCache は、XAResource を介して TransactionManager と対話します。復元機能が有効になっています。**XaResource.recover()** メソッドを呼び出して、リカバリーするトランザクションを取得します。

4.7.4. トランザクションとの競合の検出

トランザクションはキーの初期値を使用して競合を検出します。

たとえば、トランザクションの開始時に k の値は v となります。準備フェーズでは、トランザクションはサーバーから "k" を取得して値を読み取ります。値が変更された場合、トランザクションは競合を回避するためにロールバックします。



注記

トランザクションはバージョンを使用して、値の等価性を確認する代わりに変更を検出します。

forceReturnValue パラメーターは、RemoteCache への書き込み操作を制御し、競合を回避するのに役立ちます。次の値があります。

- **true** の場合、書き込み操作を実行する前に TransactionManager はサーバーから最新の値を取得します。ただし、**forceReturnValue** パラメーターは、キーの初回アクセスの操作にのみ適用されます。
- **false** の場合、TransactionManager は書き込み操作を実行する前にサーバーから最新の値を取得しません。

注記

このパラメーターは、最新の値が必要なため、**replace** や **putIfAbsent** などの **条件付き** 書き込み操作には影響しません。

以下のトランザクションは、**forceReturnValue** パラメーターが競合する書き込み操作を防ぐ例を提供します。

トランザクション 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v1");
tm.commit();
```

トランザクション 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v2");
tm.commit();
```

この例では、TX1 と TX2 が並行して実行されます。k の初期値は v です。

- **forceReturnValue = true** の場合、**cache.put()** 操作はサーバーから TX1 と TX2 の両方のサーバーから k の値を取得します。最初に k のロックを取得するトランザクションはコミットします。他のトランザクションは、k が v 以外の値を持っていることをトランザクションが検出できるため、コミットフェーズ中にロールバックします。

- **forceReturnValue = false** の場合 **cache.put()** オペレーションはサーバーから "k" の値を取得せず、null を返します。TX1 と TX2 の両方が正常にコミットされ、競合が生じます。これは、"k"の初期値が変更されたことをトランザクションが検知できないために発生します。

以下のトランザクションは、**cache.put()** オペレーションを行う前に K の値を読み取る **cache.get()** オペレーションが含まれます。

トランザクション 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v1");
tm.commit();
```

トランザクション 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v2");
tm.commit();
```

上記の例では、TX1 および TX2 の両方が鍵を読み取るため、**forceReturnValue** パラメーターは有効になりません。1 つのトランザクションがコミットし、もう 1 つのトランザクションがロールバックします。ただし、**cache.get()** 操作には追加のサーバー要求が必要です。サーバーリクエストが非効率な **cache.put()** オペレーションの戻り値が必要ない場合。

4.7.5. 設定済みトランザクションマネージャーおよびトランザクションモードの使用

以下の例は、**RemoteCacheManager** で設定した **TransactionManager** および **TransactionMode** を使用する方法を示しています。

```
//Configure the transaction manager and transaction mode.
```

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.remoteCache("my-cache")
    .transactionManagerLookup(RemoteTransactionManagerLookup.getInstance())
    .transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//The my-cache instance uses the RemoteCacheManager configuration.
RemoteCache<String, String> cache = rcm.getCache("my-cache");

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```