Red Hat Data Grid 8.1

热 Rod Java 客户端指南

配置和使用 Hot Rod Java 客户端

Last Updated: 2023-12-15

Red Hat Data Grid 8.1 热 Rod Java 客户端指南

配置和使用 Hot Rod Java 客户端

法律通告

摘要

热 Rod Java 客户端可让您对 Data Grid 集群进行高性能远程访问。

目录

RED HAT DATA GRID

Data Grid 是一个高性能分布式内存数据存储。

无架构数据结构

将不同对象存储为键值对的灵活性。

基于网格的数据存储

旨在在集群中分发和复制数据。

弹性扩展

动态调整节点数量，以便在不中断服务的情况下满足需求。

数据互操作性

从不同端点在网格中存储、检索和查询数据。

DATA GRID 文档

红帽客户门户网站中提供了 Data Grid 的文档。

- Data Grid 8.1 文档
- Data Grid 8.1 组件详情
- Data Grid 8.1 支持的配置
- Data Grid 8 功能支持
- 数据中心已弃用的功能和功能

DATA GRID 下载

访问红帽客户门户上的 Data Grid 软件下载。

注意

您必须有一个红帽帐户才能访问和下载数据中心软件。

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。有关更多详情，请参阅我们的首席技术官 Chris Wright 提供的消息。

第 1 章 热 ROD JAVA 客户端

通过 Hot Rod Java 客户端 API 远程访问数据网格。

1.1. 热 ROD 协议

hot Rod 是一个二进制 TCP 协议，Data Grid 提供高性能客户端-服务器与以下功能的交互：

- 负载平衡。热 Rod 客户端可以使用不同的策略在 Data Grid 集群中发送请求。
- 故障切换.热 Rod 客户端可以监控数据网格集群拓扑更改，并自动切换到可用节点。
- 有效的数据位置.热 Rod 客户端可以找到密钥所有者，并直接向这些节点发出请求，从而缩短延迟。

1.2. 配置 DATA GRID MAVEN 存储库

Data Grid Java 发行版可从 Maven 获取。

您可以从客户门户网站下载 Data Grid Maven 存储库，或者从公共 Red Hat Enterprise Maven 存储库拉取 Data Grid 依赖项。

1.2.1. 下载 Data Grid Maven 存储库

如果您不想使用公共 Red Hat Enterprise Maven 存储库，将 Data Grid Maven 存储库下载并安装到本地文件系统、Apache HTTP 服务器或 Maven 存储库管理器。

流程

1. 登录到红帽客户门户。
2. 导航到 Data Grid 的软件下载。
3. 下载 Red Hat Data Grid 8.1 Maven 存储库。
4. 将存档的 Maven 存储库提取到本地文件系统。
5. 打开 **README.md** 文件，并按照适当的安装说明进行操作。

1.2.2. 添加 Red Hat Maven 存储库

在您的 Maven 构建环境中包括红帽 GA 存储库，以获取 Data Grid 工件和依赖项。

流程

- 将 Red Hat GA 存储库添加到 Maven 设置文件中，通常为 **~/.m2/settings.xml**，或者直接在项目的 **pom.xml** 文件中。

```
<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>
```

参考

- Red Hat Enterprise Maven 存储库

1.2.3. 配置数据网格 POM

Maven 使用名为 Project Object Model (POM)文件的配置文件来定义项目并管理构建。POM 文件采用 XML 格式，描述生成的项目打包和输出的模块和组件依赖项、构建顺序和目标。

流程

1. 打开您的项目 **pom.xml** 进行编辑。
2. 使用正确的 Data Grid 版本定义 **version.infinispan** 属性。
3. 在 **dependencyManagement** 部分中包含 **infinispan-bom**。
   Bill Of Materials (BOM)控制依赖项版本，从而避免了版本冲突，这意味着您不需要为添加到项目的每个 Data Grid 工件设置版本。
4. 保存并关闭 **pom.xml**。

以下示例显示了 Data Grid 版本和 BOM：

```
<properties>
  <version.infinispan>11.0.9.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>
```

后续步骤

根据需要，将 Data Grid 工件作为依赖项添加到 **pom.xml** 中。

1.3. 获取 HOT ROD JAVA 客户端

将 Hot Rod Java 客户端添加到您的项目。

先决条件

热 Rod Java 客户端可以使用 Java 8 或 Java 11。

流程

- 将 **infinispan-client-hotrod** 工件作为依赖项添加到 **pom.xml** 中，如下所示：

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

参考

数据网格服务器要求

第 2 章 配置 HOT ROD JAVA 客户端

2.1. 以编程方式配置 HOT ROD JAVA 客户端

使用 **ConfigurationBuilder** 类来生成不可变配置对象，您可以传递给 **RemoteCacheManager**。

例如，使用 Java fluent API 创建客户端实例，如下所示：

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
    = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.marshaller(new org.infinispan.commons.marshall.ProtoStreamMarshaller())
  .statistics()
    .enable()
    .jmxDomain("org.example")
  .addServer()
    .host("127.0.0.1")
    .port(11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

参考

org.infinispan.client.hotrod.configuration.ConfigurationBuilder

2.2. 配置 HOT ROD JAVA 客户端属性文件

将 **hotrod-client.properties** 添加到 classpath 中，以便客户端将配置传递给 **RemoteCacheManager**。

hotrod-client.properties 示例

```
# Hot Rod client configuration
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.marshaller = org.infinispan.commons.marshall.ProtoStreamMarshaller
infinispan.client.hotrod.async_executor_factory =
org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory
infinispan.client.hotrod.default_executor_factory.pool_size = 1
infinispan.client.hotrod.hash_function_impl.2 =
org.infinispan.client.hotrod.impl.consistenthash.ConsistentHashV2
infinispan.client.hotrod.tcp_no_delay = true
infinispan.client.hotrod.tcp_keep_alive = false
infinispan.client.hotrod.request_balancing_strategy =
org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy
infinispan.client.hotrod.key_size_estimate = 64
infinispan.client.hotrod.value_size_estimate = 512
infinispan.client.hotrod.force_return_values = false

## Connection pooling configuration
maxActive = -1
maxIdle = -1
whenExhaustedAction = 1
minEvictableIdleTimeMillis=300000
minIdle = 1
```

要使用 classpath 以外的 **hotrod-client.properties**，请执行以下操作：

-

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

参考

- 热 Rod 客户端配置
- org.infinispan.client.hotrod.RemoteCacheManager
- Java 系统属性

2.3. 客户端 INTELLIGENCE

热 Rod 客户端智能是指查找用于高效路由请求的数据网格服务器的机制。

基本智能

客户端不存储任何有关 Data Grid 集群或密钥哈希值的信息。

topology-aware

客户端接收和存储有关 Data Grid 集群的信息。客户端维护集群拓扑的内部映射，该映射会在服务器加入或离开集群时进行更改。

要接收集群拓扑，客户端需要启动时至少一个 Hot Rod 服务器的地址(**IP:HOST**)。客户端连接到服务器后，Data Grid 将拓扑传送到客户端。当服务器加入或离开集群时，Data Grid 将更新的拓扑传输到客户端。

分发感知

客户端是拓扑感知型，并存储键的一致性哈希值。

例如，使用 **put (k,v)** 操作。客户端计算键的哈希值，以便它可以找到数据所在的确切服务器。然后，客户端可以直接连接到所有者来分配操作。

分发情报的好处在于，Data Grid 服务器不需要根据键哈希查找值，这在服务器端使用较少的资源。另一个好处是，服务器可以更快地响应客户端请求，因为它跳过了额外的网络往返。

2.3.1. 请求负载平衡

使用拓扑感知智能的客户端对所有请求使用请求平衡。默认平衡策略是 round-robin，因此拓扑感知客户端始终以轮循顺序向服务器发送请求。

例如，**s1**、 **s2**、 **s3** 是 Data Grid 集群中的服务器。客户端执行请求平衡，如下所示：

```
CacheContainer cacheContainer = new RemoteCacheManager();
Cache<String, String> cache = cacheContainer.getCache();

//client sends put request to s1
cache.put("key1", "aValue");
//client sends put request to s2
```

```
cache.put("key2", "aValue");
//client sends get request to s3
String value = cache.get("key1");
//client dispatches to s1 again
cache.remove("key2");
//and so on...
```

使用分布感知智能的客户端仅对失败的请求使用请求平衡。当请求失败时，分发感知客户端会在下一个可用服务器上重试请求。

自定义平衡策略

您可以实现 FailoverRequestBalancingStrategy，并使用以下属性在 **hotrod-client.properties** 配置中指定您的类：

infinispan.client.hotrod.request_balancing_strategy

2.3.2. 客户端故障切换

当 Data Grid 集群拓扑更改时，热 Rod 客户端可以自动进行故障转移。例如，具有拓扑感知的 Hot Rod 客户端可以检测一个或多个数据网格服务器何时失败。

除了集群数据网格服务器之间故障转移外，Hot Rod 客户端还可在 Data Grid 集群之间进行故障转移。

例如，您有一个 Data Grid 集群在 New York (**NYC**)中运行，另一个集群在伦敦(**LON**)中运行。向 **NYC** 发送请求的客户端检测到没有可用的节点，以便它们在 **LON** 中切换到集群。然后，客户端会维护到 **LON** 的连接，直到您手动切换集群或故障转移再次发生。

带有故障切换的事务缓存

条件操作（如 **putIfAbsent** （），**replace** （），**remove** （） ）具有严格的方法返回保证。同样，一些操作可能需要返回前面的值。

虽然 Hot Rod 客户端可以故障转移，但您应该使用事务缓存来确保操作不会部分完成，并在不同的节点上保留冲突的条目。

2.4. 为 HOT ROD 客户端配置身份验证机制

数据网格服务器使用不同的机制来验证 Hot Rod 客户端连接。

流程

- 指定 Data Grid 服务器与 **SecurityConfigurationBuilder** 类中的 **saslMechanism** （） 方法使用的身份验证机制。

SCRAM

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

```
        .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

摘要

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
  .addServer()
    .host("127.0.0.1")
    .port(11222)
  .security()
    .authentication()
      .saslMechanism("DIGEST-MD5")
      .username("myuser")
      .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

PLAIN

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
  .addServer()
    .host("127.0.0.1")
    .port(11222)
  .security()
    .authentication()
      .saslMechanism("PLAIN")
      .username("myuser")
      .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

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);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

OAUTHBEARER 身份验证使用 TokenCallbackHandler

您可以使用 **TokenCallbackHandler** 配置客户端，以便在 OAuth2 令牌过期前刷新 OAuth2 令牌，如下例所示：

```
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");
```

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())
      // KeyStore stores valid client certificates.
      .keyStoreFileName("/path/to/keystore")
      .keyStorePassword("keystorepassword".toCharArray())
    .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()
      .enable()
      .saslMechanism("GSSAPI")
      .clientSubject(clientSubject)
```

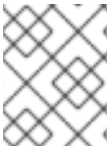
```
        .callbackHandler(new BasicCallbackHandler());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

以上配置使用 **BasicCallbackHandler** 检索客户端主题和处理身份验证。但是，这实际上调用不同的回调：

- **NameCallback** 和 **PasswordCallback** 构造客户端主题。
- 在 SASL 身份验证过程中调用 **AuthorizeCallback**。

自定义 CallbackHandler

热 Rod 客户端设置默认 **CallbackHandler**，将凭证传递给 SASL 机制。在某些情况下，您可能需要提供自定义 **CallbackHandler**。



注意

您的 **CallbackHandler** 需要处理特定于您使用的身份验证机制的回调。但是，超出了本文档的范围，为每个可能的回调类型提供示例。

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

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

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

```
   ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
  .addServer()
    .host("127.0.0.1")
    .port(11222)
  .security()
    .authentication()
      .enable()
      .serverName("myhotrodserver")
      .saslMechanism("DIGEST-MD5")
      .callbackHandler(new MyCallbackHandler("myuser","default","qwer1234!".toCharArray()));
remoteCacheManager=new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache=remoteCacheManager.getCache("secured");
```

2.4.1. 热 Rod 端点身份验证机制

Data Grid 支持带有 Hot Rod 连接器的以下 SASL 验证机制：

身份验证机制	描述	相关详情
PLAIN	以纯文本格式使用凭据。您应该只在加密连接中使用 **PLAIN** 身份验证。	与 **基本** HTTP 机制类似。
DIGESTJPEG	使用哈希算法和非ce 值。热 Rod 连接器支持 **DIGEST-MD** 5、**DIGEST-SHA** - 256、**DIGEST-SHA-256**、**DIGEST-SHA-384** 和 **DIGEST-SHA-512** 哈希算法，以强度顺序。	与 **Digest** HTTP 机制类似。
SCRAM-*	除了哈希算法和非ce 值外，还使用 *salt* 值。热 Rod 连接器支持 **SCRAM-SHA**、**SCRAM-SHA-256**、**SCRAM-SHA-384** 和 **SCRAM-SHA-512** 哈希算法（按强度排序）。	与 **Digest** HTTP 机制类似。
GSSAPI	使用 Kerberos 票据并需要一个 Kerberos 域控制器。您必须在 realm 配置中添加对应的 **kerberos** 服务器身份。在大多数情况下，您还指定一个 **ldap-realm** 来提供用户成员资格信息。	与 **SPNEGO** HTTP 机制类似。
GS2-KRB5	使用 Kerberos 票据并需要一个 Kerberos 域控制器。您必须在 realm 配置中添加对应的 **kerberos** 服务器身份。在大多数情况下，您还指定一个 **ldap-realm** 来提供用户成员资格信息。	与 **SPNEGO** HTTP 机制类似。

身份验证机制	描述	相关详情
EXTERNAL	使用客户端证书。	与 **CLIENT_CERT** HTTP 机制类似。
OAUTHBEARER	使用 OAuth 令牌并需要一个 **token-realm** 配置。	与EARER **_TOKEN** HTTP 机制类似。

2.4.2. 创建 GSSAPI 登录上下文

要使用 GSSAPI 机制，您必须创建一个 *LoginContext*，以便您的 Hot Rod 客户端可以获取 Ticket Granting Ticket Granting Ticket Granting Ticket Granting Ticket Granting Ticket (TGT)。

流程

1. 在登录配置文件中定义登录模块。

   gss.conf

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

   对于 IBM JDK：

   gss-ibm.conf

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

2. 设置以下系统属性：

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

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

注意

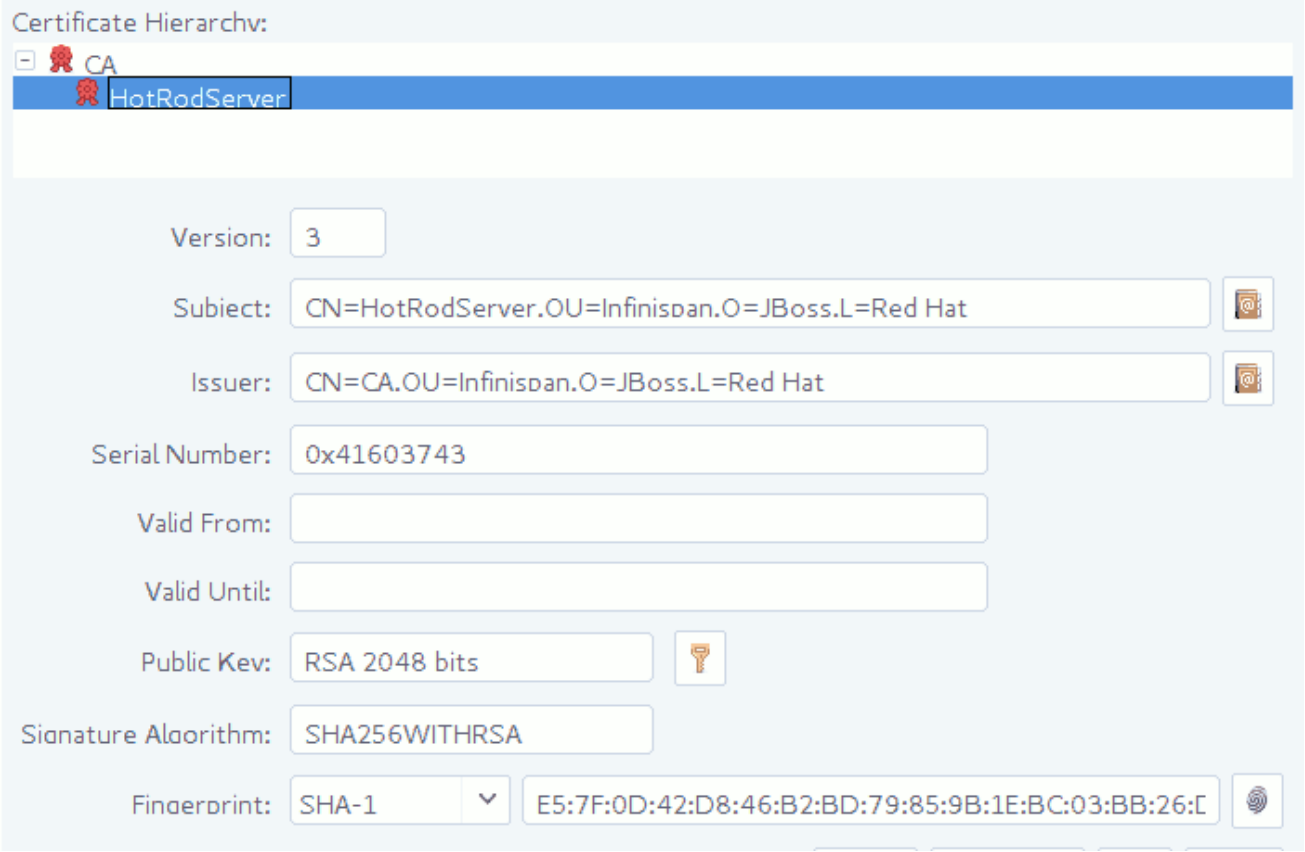
krb5.conf 提供 KDC 的位置。使用 *kinit* 命令与 Kerberos 进行身份验证并验证 **krb5.conf**。

2.5. 配置 HOT ROD 客户端加密

使用 SSL/TLS 加密的数据网格服务器向 Hot Rod 客户端提供证书，以便他们可以建立信任并协商安全连接。

要验证服务器发布的证书，Hot Rod 客户端需要 TLS 证书链的一部分。例如，下图显示名为"CA"的证书颁发机构(CA)，该服务器为名为"HotRodServer"的服务器发布证书：

图 2.1. 证书链



流程

1. 使用服务器证书链的一部分创建 Java 密钥存储。在大多数情况下，您应该为 CA 使用公共证书。
2. 在客户端配置中使用 **SslConfigurationBuilder** 类，将密钥存储指定为 *TrustStore*。

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
  .addServer()
    .host("127.0.0.1")
    .port(11222)
  .security()
    .ssl()
      // Server SNI hostname.
      .sniHostName("myservername")
      // Server certificate keystore.
      .trustStoreFileName("/path/to/truststore")
      .trustStorePassword("truststorepassword".toCharArray())
      // Client certificate keystore.
      .keyStoreFileName("/path/to/client/keystore")
      .keyStorePassword("keystorepassword".toCharArray());
RemoteCache<String, String> cache=remoteCacheManager.getCache("secured");
```

提示

指定包含 PEM 格式和 Hot Rod 客户端自动生成信任存储的路径。

使用 **.trustStorePath ("/path/to/certificate")**。

2.6. 监控 HOT ROD 客户端统计信息

启用 Hot Rod 客户端统计信息，包括 remote 和 near-cache hits 和 misses，以及连接池使用情况。

流程

- 使用 **StatisticsConfigurationBuilder** 类来启用和配置 Hot Rod 客户端统计信息。

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

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
  .statistics()
   //Enable client statistics.
   .enable()
  //Register JMX MBeans for RemoteCacheManager and each RemoteCache.
  .jmxEnable()
   //Set JMX domain name to which MBeans are exposed.
   .jmxDomain("org.example")
  .addServer()
   .host("127.0.0.1")
   .port(11222);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

2.7. 在客户端配置中定义 DATA GRID 集群

在 Hot Rod 客户端配置中提供 Data Grid 集群的位置。

流程

- 至少提供一个 Data Grid 集群名称、主机名和端口，并带有 **ClusterConfigurationBuilder** 类。

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

默认集群

将集群添加到 Hot Rod 客户端配置时，您可以以 **hostname1:port; hostname2:port** 的格式定义数据网格服务器列表。然后，Data Grid 使用服务器列表作为默认集群配置。

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

2.7.1. 手动切换 Data Grid 集群

在 Data Grid 集群间手动切换 Hot Rod Java 客户端连接。

流程

- 在 **RemoteCacheManager** 类中调用以下方法之一：
  switchToCluster (clusterName) 切换到客户端配置中定义的特定集群。

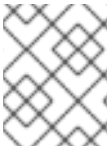
  switchToDefaultCluster （） 切换到客户端配置中的默认集群，该集群被定义为 Data Grid 服务器列表。

参考

RemoteCacheManager

2.8. 使用 HOT ROD 客户端创建缓存

通过 **RemoteCacheManager** API 在 Data Grid Server 上以编程方式创建缓存。



注意

以下流程演示了使用 Hot Rod Java 客户端进行编程缓存创建。但是，Hot Rod 客户端以 Javascript 或 C++ 等不同语言提供。

先决条件

- 创建用户，并至少启动一个 Data Grid 服务器实例。
- 获取 Hot Rod Java 客户端。

流程

1. 使用 **ConfigurationBuilder** 类配置您的客户端。

```
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.DefaultTemplate;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.commons.configuration.XMLStringConfiguration;
...

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer()
        .host("127.0.0.1")
        .port(11222)
      .security().authentication()
        .enable()
        .username("username")
        .password("password")
        .realm("default")
        .saslMechanism("DIGEST-MD5");

manager = new RemoteCacheManager(builder.build());
```

2. 使用 **XMLStringConfiguration** 类以 XML 格式添加缓存定义。

3. 调用 **getOrCreateCache （）** 方法，以添加缓存（如果已存在）或创建缓存（如果不存在）。

```
private void createCacheWithXMLConfiguration() {
   String cacheName = "CacheWithXMLConfiguration";
   String xml = String.format("<infinispan>" +
                              "<cache-container>" +
                              "<distributed-cache name=\"%s\" mode=\"SYNC\"
                              statistics=\"true\">" +
                                "<locking isolation=\"READ_COMMITTED\"/>" +
                                "<transaction mode=\"NON_XA\"/>" +
                                "<expiration lifespan=\"60000\" interval=\"20000\"/>" +
                              "</distributed-cache>" +
                              "</cache-container>" +
                             "</infinispan>"
                              , cacheName);
   manager.administration().getOrCreateCache(cacheName, new
XMLStringConfiguration(xml));
   System.out.println("Cache created or already exists.");
}
```

4. 使用 **org.infinispan** 模板创建缓存，如下例所示，使用 **createCache （）** 调用：

```
private void createCacheWithTemplate() {
   manager.administration().createCache("myCache", "org.infinispan.DIST_SYNC");
   System.out.println("Cache created.");
}
```

后续步骤

尝试一些工作代码示例，其中演示了如何使用 Hot Rod Java 客户端创建远程缓存。访问 Data Grid Tutorials。

参考

- **RemoteCacheManager** Javadoc
- 获取 Hot Rod Java 客户端

2.9. 在首次访问时创建缓存

当 Hot Rod Java 客户端试图访问不存在的缓存时，它们会返回 **null** for **getCache ("$cacheName")** 调用。

您可以更改此默认行为，以便客户端使用默认配置模板或 Data Grid 缓存定义在首次访问时自动创建缓存。

Programmatic 过程

- 使用 **remoteCache （）** 方法在 Hot Rod **ConfigurationBuilder** 类中创建每个缓存配置，如下所示：

```
import org.infinispan.client.hotrod.DefaultTemplate;
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
```

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

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.remoteCache("my-cache") 1
   .templateName(DefaultTemplate.DIST_SYNC)
builder.remoteCache("another-cache") 2
   .configuration("<infinispan><cache-container><distributed-cache name=\"another-cache\"/>
</cache-container></infinispan>");
builder.remoteCache("my-other-cache") 3
   .configurationURI(URI.create("file:/path/to/configuration.xml"));
```

1. 从 **org.infinispan.DIST_SYNC** 模板创建一个名为"my-cache"的缓存。
2. 从 XML 定义创建一个名为"another-cache"的缓存。
3. 从 XML 文件创建一个名为"my-other-cache"的缓存。

热 Rod 客户端属性

- 在 **hotrod-client.properties 文件中添加 infinispan.client.hotrod.cache.<cache-name>** 属性来创建每个缓存配置，如下所示：

```
infinispan.client.hotrod.cache.my-cache.template_name=org.infinispan.DIST_SYNC 1
infinispan.client.hotrod.cache.another-cache.configuration=<infinispan><cache-container>
<distributed-cache name=\"another-cache\"/></cache-container></infinispan> 2
infinispan.client.hotrod.cache.my-other-cache.configuration_uri=file:/path/to/configuration.xml 3
```

1. 从 **org.infinispan.DIST_SYNC** 模板创建一个名为"my-cache"的缓存。
2. 从 XML 定义创建一个名为"another-cache"的缓存。
3. 从 XML 文件创建一个名为"my-other-cache"的缓存。

参考

- 热 Rod 客户端配置
- org.infinispan.client.hotrod.configuration.RemoteCacheConfigurationBuilder
- org.infinispan.client.hotrod.DefaultTemplate

2.10. 创建永久缓存配置

除了在首次访问时创建缓存外，您还可以远程配置单个缓存的某些方面，例如：

- 强制返回值
- 接近缓存
- 事务模式

流程

- 为名为 **a-cache** 的缓存启用 *强制返回值*，如下所示：

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

ConfigurationBuilder builder = new ConfigurationBuilder();
builder
  .remoteCache("a-cache")
    .forceReturnValues(true);
```

- 在远程缓存名称中使用通配符 globbing，为以字符串 **somecaches** 开头的所有缓存启用强制返回值：

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

ConfigurationBuilder builder = new ConfigurationBuilder();
builder
  .remoteCache("somecaches*")
    .forceReturnValues(true);
```

注意

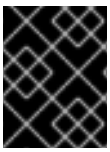
当使用声明性配置和缓存名称包含 **.** 字符时，您必须将缓存名称放在方括号中，如 **infinispan.client.hotrod.cache.[example.MyCache].template=…**

2.11. 配置 NEAR 缓存

热 Rod Java 客户端可以保留存储最近使用的数据的本地缓存，这显著提高 **get （）** 和 **getVersioned （）** 操作的性能，因为数据对客户端是本地的。

当您使用 Hot Rod Java 客户端启用接近缓存时，对 **get （）** 或 **getVersioned （）** 调用的调用会在从服务器检索条目时填充最接近的缓存。当在服务器端更新或删除条目时，接近缓存中的条目将无效。如果在密钥无效后请求密钥，客户端必须再次从服务器获取密钥。

您还可以配置接近缓存可能包含的条目数。当达到最大值时，near-cached 条目将被驱除。



接近缓存注意事项

不要将最大闲置过期时间与接近缓存一起使用，因为 near-cache 读取不会传播条目的最后一次访问时间。

- 当使用集群缓存模式时，客户端故障转移到不同的服务器时会清除接近缓存。
- 您应该始终配置可以驻留在最接近的缓存中的最大条目数。unbounded 接近缓存要求您在客户端 JVM 的界限内保持接近缓存的大小。
- 接近缓存无效消息可能会降低写入操作的性能

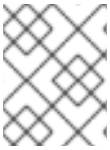
流程

1. 在客户端配置中，将您想要的缓存模式设置为 **INVALIDATED**

2. 通过指定条目的最大数量来定义最接近的缓存的大小。

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

// Configure different near cache settings for specific caches
ConfigurationBuilder builder = new ConfigurationBuilder();
builder
  .remoteCache("bounded")
    .nearCacheMode(NearCacheMode.INVALIDATED)
    .nearCacheMaxEntries(100);
  .remoteCache("unbounded").nearCache()
    .nearCacheMode(NearCacheMode.INVALIDATED)
    .nearCacheMaxEntries(-1);
```



注意

您应该始终基于每个缓存配置接近缓存。虽然 Data Grid 提供全局近的缓存配置属性，但不应使用它们。

2.12. 强制返回值

为了避免不必要地发送数据，对远程缓存写入操作会返回 **null**，而不是之前的值。

例如，以下方法调用不会返回键以前的值：

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

您可以使用 **FORCE_RETURN_VALUE** 标志更改此默认行为，以便您的调用返回前面的值。

流程

- 使用 **FORCE_RETURN_VALUE** 标志获取以前的值而不是 **null**，如下例所示：

```
cache.put("aKey", "initialValue");
assert null == cache.put("aKey", "aValue");
assert "aValue".equals(cache.withFlags(Flag.FORCE_RETURN_VALUE).put("aKey",
  "newValue"));
```

参考

org.infinispan.client.hotrod.Flag

2.13. 配置连接池

热 Rod Java 客户端保持与数据网格服务器的持久连接池，以重复利用 TCP 连接，而不是在每个请求上创建它们。

客户端使用异步线程，通过迭代连接池并将 ping 发送到 Data Grid 服务器来检查连接的有效性。这通过在池中闲置时发现有问题的连接来提高性能，而不是在应用程序请求中。

流程

流程

- 使用 **ConnectionPoolConfigurationBuilder** 类配置 Hot Rod 客户端连接池设置。

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
  .addServer()
   .host("127.0.0.1")
   .port(11222)
   //Configure client connection pools.
   .connectionPool()
   //Set the maximum number of active connections per server.
   .maxActive(10)
   //Set the minimum number of idle connections
   //that must be available per server.
   .minIdle(20);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

2.14. HOT ROD JAVA CLIENT MARSHALLING

热 Rod 是一种二进制 TCP 协议，要求您将 Java 对象转换为二进制格式，以便可以通过线路或存储到磁盘传输它们。

默认情况下，Data Grid 使用 ProtoStream API 将 Java 对象编码并解码为协议缓冲器(Protobuf)；语言中立、向后兼容的格式。但是，您也可以实施并使用自定义 marshallers。

参考

- Marshalling Java 对象
- 使用 ProtoStream Marshaller

2.14.1. 配置 SerializationContextInitializer 实现

您可以将 ProtoStream **SerializationContextInitializer** 接口的实现添加到 Hot Rod 客户端配置，以便 Data Grid marshalls 自定义 Java 对象。

流程

- 将 **SerializationContextInitializer** 实现添加到 Hot Rod 客户端配置中，如下所示：

hotrod-client.properties

```
infinispan.client.hotrod.context-
initializers=org.infinispan.example.LibraryInitializerImpl,org.infinispan.example.AnotherExampleSciImpl
```

编程配置

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder
    .addServer()
    .host("127.0.0.1")
```

```
    .port(11222)
    .addContextInitializers(new LibraryInitializerImpl(), new AnotherExampleSciImpl());
RemoteCacheManager rcm = new RemoteCacheManager(builder.build());
```

- 使用 ProtoStream Marshaller
- ProtoStream Serialization 上下文

2.14.2. 配置自定义 Marshallers

将 Hot Rod 客户端配置为使用自定义 marshallers。

流程

1. 实施 **org.infinispan.commons.marshall.Marshaller** 接口。
2. 在 Hot Rod 客户端配置中指定您的类的完全限定名称。
3. 将您的 Java 类添加到 Data Grid deserialization whitelist 中。
   在以下示例中，只允许带有 **Person** 或 **Employee** 的完全限定名称的类：

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.marshaller("org.infinispan.example.marshall.CustomMarshaller")
  .addJavaSerialWhiteList(".*Person.*", ".*Employee.*");
  ...
```

参考

- org.infinispan.commons.marshall.Marshaller
- 使用自定义 Marshallers
- 将 Java 类添加到 Deserialization White 列表中

2.15. 配置 HOT ROD 客户端数据格式

默认情况下，Hot Rod 客户端操作在读取和写入 Data Grid 服务器时使用配置的 marshaller。

但是，**DataFormat** API 允许您分离远程缓存，以便所有操作都可以使用自定义数据格式发生。

将不同的 marshallers 用于键和值

在运行时可以覆盖键和值的 Marshallers。例如，要绕过 Hot Rod 客户端中的所有序列化，并读取 **byte[]**，因为它们存储在服务器中：

```
// Existing RemoteCache instance
RemoteCache<String, Pojo> remoteCache = ...

// IdentityMarshaller is a no-op marshaller
DataFormat rawKeyAndValues =
DataFormat.builder()
        .keyMarshaller(IdentityMarshaller.INSTANCE)
        .valueMarshaller(IdentityMarshaller.INSTANCE)
        .build();
```

```
// Creates a new instance of RemoteCache with the supplied DataFormat
RemoteCache<byte[], byte[]> rawResultsCache =
remoteCache.withDataFormat(rawKeyAndValues);
```

重要

对键使用不同的 marshallers 和格式，**keyMarshaller （）** 和 **keyType （）** 方法可能会影响客户端智能路由机制，并导致 Data Grid 集群中的额外跃点执行该操作。如果性能至关重要，您应该使用服务器存储的格式的密钥。

返回 XML 值

```
Object xmlValue = remoteCache
    .withDataFormat(DataFormat.builder()
    .valueType(APPLICATION_XML)
    .valueMarshaller(new UTF8StringMarshaller())
    .build())
    .get(key);
```

前面的代码示例返回 XML 值，如下所示：

```
<?xml version="1.0" ?><string>Hello!</string>
```

以不同格式读取数据

请求并发送由 **org.infinispan.commons.dataconversion.MediaType** 指定的不同格式的数据，如下所示：

```
// Existing remote cache using ProtostreamMarshaller
RemoteCache<String, Pojo> protobufCache = ...

// Request values returned as JSON
// Use the UTF8StringMarshaller to convert UTF-8 to String
DataFormat jsonString =
DataFormat.builder()
        .valueType(MediaType.APPLICATION_JSON)
        .valueMarshaller(new UTF8StringMarshaller())
        .build();
RemoteCache<byte[], byte[]> rawResultsCache =
protobufCache.withDataFormat(jsonString);
```

```
// Alternatively, use a custom value marshaller
// that returns `org.codehaus.jackson.JsonNode` objects
DataFormat jsonNode =
DataFormat.builder()
        .valueType(MediaType.APPLICATION_JSON)
        .valueMarshaller(new CustomJacksonMarshaller()
        .build();

RemoteCache<String, JsonNode> jsonNodeCache =
remoteCache.withDataFormat(jsonNode);
```

在上例中，数据转换发生在 Data Grid 服务器中。如果数据源不支持从存储格式进行转换，则数据网格会抛出异常。

参考

org.infinispan.client.hotrod.DataFormat

第 3 章 HOT ROD CLIENT API

Data Grid Hot Rod 客户端 API 提供了用于远程创建缓存、处理数据、监控集群缓存拓扑等的接口。

3.1. 基本 API

以下是客户端 API 如何使用 Java Hot Rod 客户端存储或检索信息的示例代码片段。它假定 Data Grid 服务器在 **localhost:11222** 上运行。

```
//API entry point, by default it connects to localhost:11222
CacheContainer cacheContainer = new RemoteCacheManager();

//obtain a handle to the remote default cache
Cache<String, String> cache = cacheContainer.getCache();

//now add something to the cache and make sure it is there
cache.put("car", "ferrari");
assert cache.get("car").equals("ferrari");

//remove the data
cache.remove("car");
assert !cache.containsKey("car") : "Value must have been removed!";
```

客户端 API 映射本地 API: RemoteCacheManager 对应于 DefaultCacheManager （实施 CacheContainer ）。这种通用 API 有助于通过 Hot Rod 从本地调用进行简单迁移：所有需要做的是在 DefaultCacheManager 和 RemoteCacheManager 之间切换 - 这进一步简化了这两者继承的通用 CacheContainer 接口。

3.2. REMOTECACHE API

集合方法 **keySet**、**entrySet** 和 值 由远程缓存支持。也就是说，每种方法都调用回 **RemoteCache**。这很有用，因为它允许准确检索各种键、条目或值，如果用户不希望，则不必将它们全部存储在客户端内存中。

这些集合遵循正在添加的 **Map** 规格，而不支持 **add All**，但所有其他方法都被支持。

要注意的一件事是 **Iterator.remove** 和 **Set.remove** 或 **Collection.remove** 方法需要超过 1 个往返操作。您可以检查 RemoteCache Javadoc，以查看这些和其他方法的更多详情。

迭代器使用情况

这些集合的迭代方法在内部使用 **retrieveEntries**，如下所述。如果您注意到 **retrieveEntries** 使用批处理大小的参数。无法向迭代器提供此操作。因此，批处理大小可以通过系统属性 **infinispan.client.hotrod.batch_size** 配置，或者在配置 **RemoteCacheManager** 时通过 ConfigurationBuilder 配置。

另外，返回的 **retrieveEntries** iterator **可以被关闭**，如来自 **keySet**,**entrySet** 和 值 返回 **AutoCloseable** 变体。因此，在使用完这些"Iterator"时，您应该始终关闭这些"Iterator"。

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

    }
```

如果我想要深度副本而不是后备集合，该怎么办？

之前版本的 **RemoteCache** 允许检索 **keySet** 的深度副本。您仍然可以通过新的后备映射进行这一操作，只需自行复制内容。另外，您可以使用 **entrySet** 和 值 （在之前不支持）进行此操作。

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

3.2.1. 不支持的方法

Data Grid **RemoteCache** API 不支持 **Cache** API 中的所有方法，并在调用不支持的方法时抛出 **UnsupportedOperationException**。

大多数方法不适用于远程缓存（如侦听器管理操作），或者对应于本地缓存不支持的方法（例如，containsValue）。

RemoteCache API 不支持从 **ConcurrentMap** 继承的某些原子操作，例如：

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

但是，**remoteCache** 为这些原子操作提供替代的方法，它们通过网络发送版本标识符，而不是整个值对象。

参考

- Cache
- RemoteCache
- UnsupportedOperationException
- ConcurrentMap

3.3. 远程 ITERATOR API

Data Grid 提供了一个远程迭代器 API，用于检索内存资源受限制或者计划进行服务器端过滤或转换的条目。

```
// 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
    }
}
```

3.3.1. 将自定义过滤器部署到 Data Grid 服务器

将自定义过滤器部署到 Data Grid 服务器实例。

流程

1. 创建扩展 **KeyValueFilterConverterFactory** 的工厂。

```
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 ommitted, the server will use "application/x-java-object".
      // Returning null will cause the filter/converter to be done in the storage format.
    }
  }
}
```

2. 创建一个 JAR 其中包含 **META**

2. 创建一个 JAR，其中包含 **META-INF/services/org.infinispan.filter.KeyValueFilterConverterFactory** 文件。此文件应包含过滤器工厂类实施的完全限定类名称。
   如果过滤器使用自定义键/值类，您必须将它们包含在 JAR 文件中，以便过滤器可以正确地 unmarshall 键和/或值实例。

3. 将 JAR 文件添加到 Data Grid 服务器安装目录的 **server/lib** 目录中。

参考

- KeyValueFilterConverterFactory

3.4. METADATAVALUE API

将 **MetadataValue** 接口用于版本控制操作。

以下示例显示了只有在条目值版本没有改变时才发生的删除操作：

```
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");
```

参考

- org.infinispan.client.hotrod.MetadataValue

3.5. STREAMING API

Data Grid 提供了一个流 API，它实现了返回 **InputStream** 和 **OutputStream** 实例的方法，以便您可以在 Hot Rod 客户端和 Data Grid 服务器之间流传输大型对象。

考虑以下大型对象示例：

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

您可以通过流读取对象，如下所示：

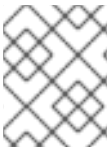
```
StreamingRemoteCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
for(int b = is.read(); b >= 0; b = is.read()) {
  // iterate
}
is.close();
```

注意

Streaming API **不会** marshall 值，这意味着您无法同时使用 Streaming 和 Non-Streaming API 访问相同的条目。但是，您可以实施一个自定义 marshaller 来处理这个问题单。

RemoteStreamingCache.get (K key) 方法返回的 **InputStream** 实现了 **VersionedMetadata** 接口，以便您可以检索版本和过期信息，如下所示：

```
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();
```



注意

条件写入方法(**putIfAbsent （）**）在值完全发送到服务器后执行实际条件检查。换句话说，当在 **OutputStream** 上调用 **close （）** 方法时。

参考

- org.iinfinispan.client.hotrod.StreamingRemoteCache

3.6. 计数器 API

CounterManager 接口是定义、检索和删除计数器的入口点。

热 Rod 客户端可以检索 **CounterManager** 接口，如下例所示：

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

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

参考

- 集群的计数

3.7. 创建事件 LISTENERS

Java Hot Rod 客户端可以注册监听程序来接收 cache-entry 级别事件。支持创建、修改和删除的事件的缓存条目。

创建客户端监听程序与嵌入式监听程序非常相似，但使用不同的注解和事件类。以下是打印收到的每个事件的客户端监听程序示例：

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

```
@ClientListener
public class EventPrintListener {

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

  @ClientCacheEntryModified
  public void handleModifiedEvent(ClientCacheEntryModifiedEvent e) {
    System.out.println(e);
  }

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

}
```

ClientCacheEntryCreatedEvent 和 **ClientCacheEntryModifiedEvent** 实例提供有关受影响密钥的信息，以及条目的版本。此版本可用于在服务器上调用条件操作，如 **replaceWithVersion** 或 **removeWithVersion**。

只有 remove 操作成功时才会发送 **ClientCacheEntryRemovedEvent** 事件。换句话说，如果调用删除操作但没有找到条目，或者不应删除任何条目，则不会生成事件。有兴趣删除的事件的用户（即使没有删除条目）可以开发事件自定义逻辑来生成此类事件。如需更多信息，请参阅自定义 客户端事件部分。

所有 **ClientCacheEntryCreatedEvent**,**ClientCacheEntryModifiedEvent** 和 **ClientCacheEntryRemovedEvent** 事件实例也会提供一个 **布尔值 isCommandRetried （）** 方法，如果因为拓扑更改而需要再次重试的写命令返回 true。这可能是此事件已被重复或另一个事件已被丢弃并替换（例如： ClientCacheEntryModifiedEvent 替换 ClientCacheEntryCreatedEvent）的符号。

创建了客户端侦听器实施后，需要向服务器注册。要做到这一点，请执行：

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

3.7.1. 删除事件 Listener

当不需要客户端事件监听程序时，可以删除它：

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

3.7.2. 过滤事件

为了避免用事件取消客户端，用户可以提供过滤功能来限制服务器为特定客户端侦听器触发的事件数量。要启用过滤，需要创建一个缓存事件过滤器工厂来生成过滤器实例：

```
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;
  }
}
```

上面定义的缓存事件过滤器工厂实例会创建过滤器实例，它们静态过滤掉除其键为 **1** 的所有条目。

为了能够使用此缓存事件过滤器工厂注册侦听器，必须赋予一个唯一的名称，并且 Hot Rod 服务器需要插入名称和缓存事件过滤器工厂实例。

1. 创建包含过滤器实现的 JAR 文件。
   如果缓存使用自定义键/值类，则必须将它们包含在 JAR 中，以便可以使用正确的 unmarshalled 键和/或值实例来执行回调。如果客户端侦听器启用了 **useRawData**，则不需要此功能，因为回调键/值实例将以二进制格式提供。

2. 在 JAR 文件中创建一个 **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory** 文件，并在其中编写过滤器类实施的完全限定类名称。

3. 将 JAR 文件添加到 Data Grid 服务器安装目录的 **server/lib** 目录中。

4. 通过将工厂名称添加到 **@ClientListener** 注释，将客户端监听程序链接到此缓存事件过滤器工厂：

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

5. 使用服务器注册监听程序：

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

您还可以根据在监听器注册时提供的参数注册动态过滤器实例。过滤器使用过滤器工厂接收的参数启用这个选项，例如：

```
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;
   }
}
```

在注册监听器时，提供了执行过滤所需的动态参数：

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

警告

当它们部署到集群中时，过滤实例必须可以被处理，以便过滤可以在生成事件的位置发生，即使即使即使在被注册了监听器的不同节点中也是如此。为了使其可以被编译，可以使它们扩展 **Serializable**、**Externalizable** 或为它们提供自定义外部工具。

3.7.3. 跳过通知

在调用远程 API 方法来执行操作时，包括 SKIP_LISTENER_NOTIFICATION 标志，而无需从服务器获取事件通知。例如，要在创建或修改值时防止监听程序通知，请设置标志，如下所示：

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

3.7.4. 自定义事件

默认情况下生成的事件仅包含足够的信息，以便使事件相关，但可以避免产生太多的信息，以降低发送它们的成本。（可选）事件中提供的信息可以自定义，使其包含更多信息，如值，或者包含较少的信息。此自定义通过 CacheEventConverter Factory 生成的 CacheEventConverterFactory 实例进行：

```
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;
  }
}
```

在上例中，转换器生成新的自定义事件，该事件包括值以及事件中的键。与默认事件相比，这会导致更大的事件有效负载，但如果与过滤结合使用，则可能会降低其网络带宽成本。

警告

转换程序的目标类型必须是 Serializable 或 Externalizable。在这个特殊情况下，提供外部程序的转换器默认不起作用，因为默认的 Hot Rod 客户端 marshaller 不支持它们。

处理自定义事件需要略有不同的客户端监听程序实现。要更精确地处理 ClientCacheEntryCustomEvent 实例：

```
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);
  }

}
```

回调中收到的 ClientCacheEntryCustomEvent 通过 getEventData 方法公开自定义事件，getType 方法提供了有关生成的事件的信息，这是缓存条目创建、修改或删除的结果。

与过滤类似，若要使用此转换器工厂注册监听程序，必须授予唯一的名称，并且 Hot Rod 服务器需要插入名称和缓存事件转换器工厂实例。

1. 创建一个 JAR 文件，其中带有转换器实现。

   如果缓存使用自定义键/值类，则必须将它们包含在 JAR 中，以便可以使用正确的 unmarshalled 键和/或值实例来执行回调。如果客户端侦听器启用了 useRawData，则不需要此功能，因为回调键/值实例将以二进制格式提供。

2. 在 JAR 文件中创建一个 META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory

文件，并编写转换器类实施的完全限定类名称。

3. 将 JAR 文件添加到 Data Grid 服务器安装目录的 server/lib 目录中。

4. 通过将工厂名称添加到 @ClientListener 注释，将客户端监听程序与这个转换器工厂连接：

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

5. 使用服务器注册监听程序：

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

也可以根据在侦听器注册时提供的参数转换的动态转换器实例。转换器使用转换器接收的参数启用此选项。例如：

```
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);
  }
}
```

在注册监听器时，提供了进行转换所需的动态参数：

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

警告

当集群部署到集群中时，转换器实例必须可以被处理，因此即使事件被注册了监听程序的不同节点中也会发生转换。为了使其可以被编译，可以使它们扩展 Serializable、Externalizable 或为它们提供自定义外部工具。

3.7.5. 过滤和自定义事件

如果要同时进行事件过滤和自定义，可以更轻松地实施 org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter，它允许过滤和自定义在一个步骤中进行。为方便起见，建议直接扩展 org.infinispan.notifications.cachelistener.filter.AbstractCacheEventFilterConverter，而不是直接实施 org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter。例如：

```
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);
  }
}
```

与过滤器和转换器类似，要能够使用组合的 filter/converter 工厂注册监听程序，工厂必须通过 @NamedFactory 注解指定唯一名称，并且 Hot Rod 服务器需要与名称和缓存事件转换器工厂实例插入。

1. 创建一个 JAR 文件，其中带有转换器实现。

   如果缓存使用自定义键/值类，则必须将它们包含在 JAR 中，以便可以使用正确的 unmarshalled 键和/或值实例来执行回调。如果客户端侦听器启用了 useRawData，则不需要此功能，因为回调键/值实例将以二进制格式提供。

2. 在 JAR 文件中创建一个 META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverterFactory 文件，并编写转换器类实施的完全限定类名称。

3. 将 JAR 文件添加到 Data Grid 服务器安装目录的 server/lib 目录中。

从客户端的角度来看，要使用组合过滤器和转换器类，客户端监听程序必须定义相同的过滤器工厂和转换器工厂名称，例如：

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

当监听器通过 filter 或 converter 参数注册时，会提供上例中的动态参数。如果过滤器参数是非空的，则会使用这些参数，否则会使用转换器参数：

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

3.7.6. event Marshalling

热 Rod 服务器可以以不同的格式存储数据，但是尽管如此，Java Hot Rod 客户端用户仍然可以开发在键入的对象上运行的 CacheEventConverter 或 CacheEventFilter 实例。默认情况下，过滤器和转换器将数据用作 POJO (application/x-java-object)，但可以通过覆盖 filter/converter 中的方法 格式（）来覆盖所需的格式。如果格式返回 null，则过滤器/转换器将接收存储的数据。

热 Rod Java 客户端可以配置为使用不同的 org.infinispan.commons.marshall.Marshaller 实例。如果这样做和部署 CacheEventConverter 或 CacheEventFilter 实例，则可以使用 Java 对象而不是 marshaller 显示过滤器/转换，服务器需要能够在对象和 marshaller 生成的二进制格式之间进行转换。

要部署 Marshaller 实例服务器端，请按照类似的方法部署 CacheEventConverter 或 CacheEventFilter 实例：

1. 创建一个 JAR 文件，其中带有转换器实现。
2. 在 JAR 文件中创建 META-INF/services/org.infinispan.commons.marshall.Marshaller 文件，写入 marshaller 类实施的完全限定类名称。
3. 将 JAR 文件添加到 Data Grid 服务器安装目录的 server/lib 目录中。

请注意，Marshaller 可以部署到单独的 jar 中，或者在与 CacheEventConverter 和/或 CacheEventFilter 实例相同的 jar 中进行部署。

3.7.6.1. 部署 Protostream Marshallers

如果缓存存储 Protobuf 内容，就像在 Hot Rod 客户端中使用 ProtoStream marshaller 时发生，则不需要部署自定义 marshaller，因为服务器已经支持格式：有 Protobuf 格式到最常见的格式，如 JSON 和 POJO。

在将过滤器/转换器与这些缓存搭配使用时，需要使用带有 Java 对象的 filter/converters 而不是二进制 Protobuf 数据时，需要配置额外的 ProtoStream marshallers，以便服务器可以在过滤/转换前处理数据。要做到这一点，您必须将所需的 SerializationContextInitializer (s) 配置为 Data Grid 服务器配置的一部分。

如需更多信息，请参阅 ProtoStream。

3.7.7. 侦听器状态处理

客户端侦听器注解具有可选的 includeCurrentState 属性，用于指定在添加监听程序时是否将状态发送到客户端，或者是监听器故障转移时。

默认情况下，includeCurrentState 为 false，但如果设置为 true，并且客户端监听程序添加到已包含数据的缓存中，服务器会迭代缓存内容，并将每个条目的事件作为 ClientCacheEntryCreated （如果配置）发送一个 ClientCacheEntryCreated （如果配置了自定义事件）。这允许客户端基于现有内容构建一些本地数据结构。迭代内容后，事件会正常接收，因为接收缓存更新。如果缓存被集群，则整个集群范围的内容都会迭代。

3.7.8. 侦听器故障处理

当 Hot Rod 客户端注册客户端监听程序时，它会在集群的单个节点中执行此操作。如果该节点失败，Java Hot Rod 客户端会检测到透明且在节点中注册的所有监听器失败。

在这种故障切换过程中，客户端可能会错过一些事件。为了避免缺少这些事件，client 侦听器注解包含一个名为 includeCurrentState 的可选参数，如果设为 true，则缓存内容可以迭代，并且生成 ClientCacheEntryCreated 事件（如果配置了自定义事件）。默认情况下，includeCurrentState 设置为 false。

使用回调来处理故障转移事件：

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

当客户端缓存了一些数据的用例中，这非常有用，因此，考虑到一些事件可能会丢失，它决定在收到事件失败时清除任何本地缓存的数据，了解事件故障转移后，它将收到整个缓存的内容的事件。

3.8. 热 ROD JAVA 客户端事务

您可以在 JTA {tx}s 中配置和使用 Hot Rod 客户端。

要参与事务，Hot Rod 客户端需要与之交互的 {tm}，以及它是否通过 {sync} 或 {xa} 接口参与事务。

重要

事务在准备阶段获取条目的写锁是最佳的。为了避免数据不一致，请务必阅读有关 使用 Transactions 的冲突。

3.8.1. 配置服务器

服务器中的缓存还必须是事务处理，客户端才能参与 JTA {tx}s。

需要以下服务器配置，否则只进行事务回滚：

- 隔离级别必须是 REPEATABLE_READ。
- 锁定模式必须是 PESSIMISTIC。在以后的发行版本中，支持 OPTIMISTIC 锁定模式。
- 事务模式应该是 NON_XA 或 NON_DURABLE_XA。热 Rod 事务不应使用 FULL_XA，因为它会降低性能。

例如：

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

热 Rod 事务有自己的恢复机制。

3.8.2. 配置 Hot Rod 客户端

在创建 {rcm} 时，您可以设置 {rc} 使用的默认 {tm} 和 {tx-mode}。

{rcm} 可让您只为事务缓存创建一个配置，如下例所示：

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
```

```
//other client configuration parameters
cb.transaction().transactionManagerLookup(GenericTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);
cb.transaction().timeout(1, TimeUnit.MINUTES)
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

前面的配置应用到远程缓存的所有实例。如果需要将不同的配置应用到远程缓存实例，您可以覆盖 {rc} 配置。请参阅 覆盖 RemoteCacheManager 配置。

有关配置参数的文档，请参阅 {cb} Javadoc。

您还可以使用属性文件配置 Java Hot Rod 客户端，如下例所示：

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

3.8.2.1. TransactionManagerLookup Interface

TransactionManagerLookup 提供了一个入口点，用于获取 {tm}。

TransactionManagerLookup 的可用实现：

{gtml}

查找类，用于查找在 Java EE 应用服务器中运行的 {tm}。如果找不到 {tm}，则默认为 {rtm}。这是 Hot Rod Java 客户端的默认设置。

提示

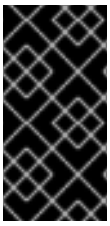
在大多数情况下，{gtml} 适合。但是，如果您需要集成自定义 {tm}，您可以实现 TransactionManagerLookup 接口。

{rtml}

如果没有其他实现，则基本和易失性 {tm}。请注意，这个实现在处理并发事务和恢复时有很大的限制。

3.8.3. 事务模式

{tx-mode} 控制 {rc} 与 {tm} 交互的方式。



重要

在 Data Grid 服务器和您的客户端应用程序上配置事务模式。如果客户端试图对非事务缓存执行事务操作，则可能会出现运行时异常。

在 Data Grid 配置和客户端设置中，事务模式都相同。将以下模式与客户端搭配使用，请参阅服务器的 Data Grid 配置模式：

NONE

{rc} 不与 {tm} 交互。这是默认模式，不是事务处理模式。

NON_XA

{rc} 通过 {sync} 与 {tm} 交互。

NON_DURABLE_XA

{rc} 通过 {xa} 与 {tm} 交互。恢复功能被禁用。

FULL_XA

{rc} 通过 {xa} 与 {tm} 交互。启用恢复功能。调用 XaResource.recover （） 方法，以检索要恢复的事务。

3.8.4. 覆盖缓存实例的配置

因为 {rcm} 不支持每个缓存实例的不同配置。但是，{rcm} 包含返回 {rc} 实例的 getCache (String) 方法，可让您覆盖一些配置参数，如下所示：

getCache (String cacheName, TransactionMode transactionMode)

返回 {rc} 并覆盖配置的 {tx-mode}。

getCache (String cacheName, boolean forceReturnValue, TransactionMode transactionMode)

与之前的相同，但也可为写操作强制执行返回值。

getCache (String cacheName, TransactionManager transactionManager)

返回 {rc} 并覆盖配置的 {tm}。

getCache (String cacheName, boolean forceReturnValue, TransactionManager transactionManager)

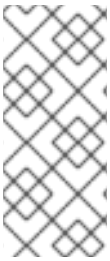
与之前的相同，但也可为写操作强制执行返回值。

getCache (String cacheName, TransactionMode transactionMode, TransactionManager transactionManager)

返回 {rc} 并覆盖配置的 {tm} 和 {tx-mode}。如果 transactionManager 或 transactionMode 为 null，则使用配置的值。

getCache (String cacheName, boolean forceReturnValue, TransactionMode transactionMode, TransactionManager transactionManager)

与之前的相同，但也可为写操作强制执行返回值。



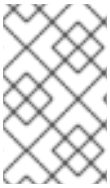
注意

getCache (String) 方法返回 {rc} 实例，无论它们是事务。{rc} 包含 getTransactionManager （） 方法，它返回了缓存使用的 {tm} 方法。如果 {rc} 不是事务处理，则方法会返回 null。

3.8.5. 使用事务检测冲突

事务使用键的初始值来检测冲突。

例如，当事务开始时，"k" 的值为 "v"。在准备阶段，事务从服务器获取"k"以读取值。如果值已更改，事务会回滚以避免冲突。



注意

事务使用版本来检测更改，而不是检查值相等。

forceReturnValue 参数控制对 {rc} 的写入操作，并帮助避免冲突。它有以下值：

- 如果为 true，则 {tm} 在执行写入操作前从服务器获取最新的值。但是，forceReturnValue

参数仅适用于首次访问密钥的操作。

- 如果为 false，则 {tm} 在执行写入操作前不会从服务器获取最新的值。

注意

此参数不会影响 替换 或放置 if Absent 等 *条件* 写入操作，因为它们需要最新的值。

以下事务提供了一个示例，其中 forceReturnValue 参数可以防止出现冲突的写入操作：

事务 1 (TX1)

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

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

事务 2 (TX2)

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

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

在这个示例中，TX1 和 TX2 并行执行。"k"的初始值为 "v"。

- 如果 forceReturnValue = true，则 cache.put （） 操作从 TX1 和 TX2 的服务器获取"k"的

值。首先获取"k"锁定的事务，然后提交。其他事务会在提交阶段回滚，因为事务可以检测到 "k" 的值为 "v"。

- 如果 forceReturnValue = false，则 cache.put （） 操作不会从服务器获取"k"的值并返回 null。TX1 和 TX2 都可以成功提交，这会导致冲突。这是因为事务都无法检测到初始值为 "k" 已更改。

以下事务包括 cache.get （） 操作，以便在执行 cache.put （） 操作前读取 "k" 的值：

事务 1 (TX1)

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

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

事务 2 (TX2)

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

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

在前面的示例中，TX1 和 TX2 都是读取密钥，因此 forceReturnValue 参数不会生效。一个事务提交，另一个回滚。但是 cache.get （） 操作需要额外的服务器请求。如果您不要求服务器请求的 cache.put （） 操作返回值效率低下。

3.8.6. 使用 Configured Transaction Manager 和 Transaction Mode

以下示例演示了如何使用在 RemoteCacheManager 中配置的 TransactionManager 和 TransactionMode ：

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

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

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

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

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

3.8.7. 覆盖 Transaction Manager

以下示例演示了如何使用 getCache 方法覆盖 TransactionManager ：

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

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

//Define a custom TransactionManager.
TransactionManager myCustomTM = ...

//Override the TransactionManager for the my-cache instance. Use the default configuration if
null is returned.
RemoteCache<String, String> cache = rcm.getCache("my-cache", null, myCustomTM);

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

3.8.8. 覆盖事务模式

以下示例演示了如何使用 getCache 方法覆盖 TransactionMode ：

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

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

//Override the transaction mode for the my-cache instance.
RemoteCache<String, String> cache = rcm.getCache("my-cache",
TransactionMode.NON_DURABLE_XA, null);

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

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