Red Hat Data Grid 8.4

在 Java 应用程序中嵌入数据网格

使用 Data Grid 创建嵌入式缓存

Last Updated: 2024-05-02

Red Hat Data Grid 8.4 在 Java 应用程序中嵌入数据网格

使用 Data Grid 创建嵌入式缓存

法律通告

摘要

将数据网格添加到 Java 项目并将嵌入式缓存与您的应用程序搭配使用。

目录

RED HAT DATA GRID

数据网格是高性能分布式内存数据存储。

Schemaless 数据结构

灵活性以将不同对象存储为键值对。

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数据网格的文档。

- Data Grid 8.4 文档
- Data Grid 8.4 组件详情
- Data Grid 8.4 支持的配置
- Data Grid 8 功能支持
- Data Grid 已弃用功能和功能

DATA GRID 下载

访问红帽客户门户网站中的 Data Grid 软件下载。

注意

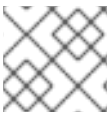
您必须有一个红帽帐户才能访问和下载 Data Grid 软件。

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。详情请查看 CTO Chris Wright 的信息。

对红帽文档提供反馈

我们非常感谢您对我们的技术内容提供反馈，并鼓励您告诉我们您的想法。如果您想添加评论，提供见解、纠正拼写错误甚至询问问题，您可以在文档中直接这样做。



注意

您必须有一个红帽帐户并登录到客户门户网站。

要从客户门户网站提交文档反馈，请执行以下操作：

1. 选择 **Multi-page HTML** 格式。
2. 点文档右上角的 **反馈** 按钮。
3. 突出显示您要提供反馈的文本部分。
4. 点高亮文本旁的**添加反馈**对话框。
5. 在页面右侧的文本框中输入您的反馈，然后单击 **Submit**。

每次提交反馈时，我们都会自动创建跟踪问题。打开在点 **Submit** 后显示的链接，并开始监视问题或添加更多注释。

感谢您的宝贵反馈。

第 1 章 在 MAVEN 存储库中添加 DATA GRID

Data Grid Java 分发包可从 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 仓库中提取 Data Grid 依赖项。

1.1. 下载 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网站。
2. 导航到 Software Downloads for Data Grid。
3. 下载 Red Hat Data Grid 8.4 Maven 存储库。
4. 将归档的 Maven 存储库提取到本地文件系统。
5. 打开 **README.md** 文件并遵循适当的安装说明。

1.2. 添加 RED HAT MAVEN 软件仓库

在 Maven 构建环境中包括 Red Hat GA 软件仓库来获取 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** 或直接在项目的 **pom.xml** 文件中添加。

```
<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 Repository

1.3. 配置项目 POM

在项目中配置项目对象模型(POM)文件，以将数据网格依赖项用于嵌入式缓存、Hot Rod 客户端和其他功能。

流程

1. 打开 **pom.xml** 项目进行编辑。
2. 使用正确的 Data Grid 版本定义 **version.infinispan** 属性。
3. 将 **infinispan-bom** 包含在 **dependencyManagement** 部分中。
   Bill Of Materials (BOM)控制依赖性版本，可以避免版本冲突，意味着您不需要为项目添加的每个数据网格构件设置版本。
4. 保存并关闭 **pom.xml**。

以下示例显示了数据网格版本和 BOM：

```
<properties>
  <version.infinispan>14.0.6.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。

第 2 章 创建嵌入缓存

数据网格提供了一个 **EmbeddedCacheManager** API，可让您以编程方式控制缓存管理器和嵌入式缓存生命周期。

2.1. 在您的项目中添加 DATA GRID

将 Data Grid 添加到项目，以便在应用程序中创建嵌入式缓存。

先决条件

- 配置项目以从 Maven 存储库获取 Data Grid 工件。

流程

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

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

2.2. 创建和使用嵌入式缓存

数据网格提供了一个 **GlobalConfigurationBuilder** API，用于控制 Cache Manager 和用于配置缓存的 **ConfigurationBuilder** API。

先决条件

- 添加 **infinispan-core** 工件作为 **pom.xml** 中的依赖项。

流程

1. 初始化 **CacheManager**。

注意

在创建缓存前，您必须始终调用 **cacheManager.start （）** 方法来初始化 **CacheManager**。默认构造器为您进行此操作，但存在不加载的构造器的超载版本。

缓存管理器也是重量级对象，而数据网格建议每个 JVM 只实例化一个实例。

2. 使用 **ConfigurationBuilder** API 定义缓存配置。
3. 使用 **getCache （）**、**createCache （）** 或 **getOrCreateCache （）** 方法获取缓存。
   数据网格建议使用 **getOrCreateCache （）** 方法，因为它可以在所有节点上创建缓存或返回现有缓存。
4. 如果需要，使用 **PERMANENT** 标志进行缓存，在重启过程中保留。

5. 通过调用 **cacheManager.stop （）** 方法来释放 JVM 资源并正常关闭任何缓存，停止 **CacheManager**。

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

getCache （） 方法

调用 **getCache (String)** 方法来获取缓存，如下所示：

```
Cache<String, String> myCache = manager.getCache("myCache");
```

前面的操作会创建一个名为 **myCache** 的缓存（如果尚不存在），并返回它。

使用 **getCache （）** 方法仅在您调用方法的节点上创建缓存。换句话说，它将执行一个本地操作，必须在集群的每个节点上调用。通常，跨越多个节点部署的应用在初始化期间获取缓存，以确保缓存是 *对称* 并存在于每个节点上。

createCache （） 方法

调用 **createCache （）** 方法，以在整个集群中动态创建缓存。

```
Cache<String, String> myCache = manager.administration().createCache("myCache",
"myTemplate");
```

前面的操作还会在之后加入集群的任何节点上自动创建缓存。

使用 **createCache （）** 方法创建的缓存默认为临时。如果整个集群关闭，则在重启时不会再次创建缓存。

PERMANENT 标志

使用 PERMANENT 标志来确保缓存可以在重新启动后保留。

```
Cache<String, String> myCache =
manager.administration().withFlags(AdminFlag.PERMANENT).createCache("myCache",
"myTemplate");
```

要使 PERMANENT 标志生效，您必须启用全局状态并设置配置存储供应商。

有关配置存储供应商的更多信息，请参阅 GlobalStateConfigurationBuilder#configurationStorage （）。

其他资源

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

2.3. 缓存 API

数据网格提供了一个 缓存 接口，用于公开添加、检索和删除条目的简单方法，包括 JDK 的 ConcurrentMap 接口公开的 atomic 机制。根据所用的缓存模式，调用这些方法将触发一些问题，甚至可能包括将条目复制到远程节点或在远程节点中查找条目，或可能缓存存储。

对于简单用法，使用 Cache API 不应与使用 JDK Map API 的不同，因此从基于 map 到数据网格缓存的简单内存缓存迁移都应该无关紧要。

Certain Map Methods 的性能调优

当与 Data Grid 搭配使用时，某些在映射中公开的方法具有某些性能后果，如 size （）、value （）、keySet （） 和 entrySet （）。有关 **keySet**、值 和 **entrySet** 的具体方法，请参见其 Javadoc 以了解更多详情。

在全球范围内尝试执行这些操作会对性能有大的影响，并成为可扩展性瓶颈。因此，这些方法应该只用于信息或调试目的。

请注意，在 withFlags （） 方法中使用某些标记可以缓解这些问题，请检查每种方法的文档以了解更多详细信息。

Mortal 和 Immortal 数据

除了仅存储条目，Data Grid 的缓存 API 可让您将分散信息附加到数据。例如，只需使用 put (key, value) 即可创建一个 *非* 仲裁条目，即：位于缓存中有的条目（位于缓存中时），直到它被删除（或被驱除内存以防止内存不足）。但是，如果使用 put (key, value, Lifespan, timeunit) 将数据放置在缓存中，这将创建一个 *mortal* 条目（例如，在那个 lifespan后具有固定寿命和过期的条目）。

除了 *lifespan* 外，数据网格还支持 *maxIdle* 作为附加指标，以确定过期时间。可以使用 lifespans 或 maxIdles 的任意组合。

putForExternalRead operation

Data Grid's Cache 类包含一个名为 putForExternalRead 的不同 'put' 操作。当将 Data Grid 用作保留在其他位置的数据临时缓存时，此操作特别有用。在大量读取方案下，缓存中的竞争不应延迟实时事务，因为缓存应只是采用优化方式，而不是采用这种方式。

要达到此目的，**putForExternalRead （）** 充当一个只能在缓存中不存在键的放置调用，并且如果另一个线程尝试同时存储同一密钥，则会失败并静默失败。在这种特定场景中，缓存数据是优化系统的一种方法，因此，缓存中的故障不会影响到持续事务中的故障，因此故障是处理失败的原因。**putForExternalRead （）** 被视为一个快速操作，因为无论是否成功完成，也不会等待任何锁定，因此无法及时返回调用者。

要了解如何使用此操作，让我们来看一下基本的示例。想象一下 Person 实例的缓存，每个由 PersonId 进行键，其数据源自于单独的数据存储。以下代码显示了在此示例中使用 putForExternalRead 的最常见模式：

```
// Id of the person to look up, provided by the application
PersonId id = ...;
```

```
// Get a reference to the cache where person instances will be stored
Cache<PersonId, Person> cache = ...;

// First, check whether the cache contains the person instance
// associated with with the given id
Person cachedPerson = cache.get(id);

if (cachedPerson == null) {
  // The person is not cached yet, so query the data store with the id
  Person person = dataStore.lookup(id);

  // Cache the person along with the id so that future requests can
  // retrieve it from memory rather than going to the data store
  cache.putForExternalRead(id, person);
} else {
  // The person was found in the cache, so return it to the application
  return cachedPerson;
}
```

请注意，putForExternalRead 不应用作使用新 Person 实例（从修改 Person 地址）更新缓存的机制。更新缓存的值时，请使用标准 放置 操作，否则可能会缓存损坏数据。

2.3.1. AdvancedCache API

除简单的缓存界面外，Data Grid 还提供了 高级缓存 接口，面向扩展作者。AdvancedCache 提供了访问某些内部组件的功能，也可应用标志来更改某些缓存方法的默认行为。以下代码片段描述了如何获取 AdvancedCache：

```
AdvancedCache advancedCache = cache.getAdvancedCache();
```

2.3.1.1. 标记

标志应用于常规缓存方法，以改变某些方法的行为。有关所有可用标志及其影响的列表，请查看 标记 枚举。使用 AdvancedCache.withFlags （） 应用标志。这个构建器方法可用于将任意数量的标志应用到缓存调用，例如：

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

2.3.2. 异步 API

除了 Cache.put （）、 Cache.remove （） 等同步 API 方法之外，数据网格还具有异步非阻塞 API，您可以实现同样的结果。

这些方法以类似的方式命名，其块程序带有 "Async" 附加。 E.g., Cache.putAsync() , Cache.removeAsync() , etc. 这些异步对应一个包含操作实际结果的 CompletableFuture 会返回一个 CompletableFuture。

例如，在 cache 参数ized as **Cache<String, String>**, **Cache.put (String value, String value) returns String** while **Cache.putAsync (String key, String value)** 返回 Complet **ableFutureString<String>**。

2.3.2.1. 为什么要使用这样的 API？

非阻塞 API 功能强大，它们能提供同步通信的所有保证 - 能够处理通信失败和异常 - 在调用完成前，无需阻止任何操作。 这可让您更好地使用系统中的并行性。 例如：

```
Set<CompletableFuture<?>> futures = new HashSet<>();
futures.add(cache.putAsync(key1, value1)); // does not block
futures.add(cache.putAsync(key2, value2)); // does not block
futures.add(cache.putAsync(key3, value3)); // does not block

// the remote calls for the 3 puts will effectively be executed
// in parallel, particularly useful if running in distributed mode
// and the 3 keys would typically be pushed to 3 different nodes
// in the cluster

// check that the puts completed successfully
for (CompletableFuture<?> f: futures) f.get();
```

2.3.2.2. 哪些进程实际发生异步发生？

数据网格有 4 个事情，可视为典型的写入操作的关键路径。即，按成本顺序排列：

- 网络调用
- marshalling
- 写入缓存存储（可选）
- 锁定

使用 async 方法将获取网络调用并汇总关键路径。 由于各种技术的原因，写入缓存存储和获取锁，但仍发生在调用者的线程中。

第 3 章 以编程方式配置用户角色和权限

在 Java 应用中使用嵌入式缓存时，以编程方式配置安全授权。

3.1. DATA GRID 用户角色和权限

数据网格包括多个角色，为用户提供访问缓存和数据网格资源的权限。

角色	权限	Description
admin	ALL	具有所有权限的超级用户，包括缓存管理器生命周期的控制。
deployer	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR, CREATE	除了 **应用程序** 权限外，还可创建和删除数据网格资源。
application	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR	除 **观察者** 权限之外，还具有对 Data Grid 资源的读写访问权限。还可以侦听事件并执行服务器任务和脚本。
observer	ALL_READ, MONITOR	除了监控权限外，还具有对数据网格 **资源** 的读取访问权限。
monitor	MONITOR	可以通过 JMX 和 **指标端点** 查看统计信息。

其他资源

- org.infinispan.security.AuthorizationPermission Enum
- Data Grid 配置模式参考

3.1.1. 权限

用户角色是具有不同访问级别的权限集合。

表 3.1. 缓存管理器权限

权限	功能	Description
配置	**defineConfiguration**	定义新的缓存配置。
LISTEN	**addListener**	针对缓存管理器注册监听程序。
生命周期	**stop**	停止缓存管理器。
创建	**createCache,removeCache**	创建和删除容器资源，如缓存、计数器、架构和脚本。

MONITOR	**getStats**	允许访问 JMX 统计数据和 **指标端点**。
ALL	-	包括所有缓存管理器权限。

表 3.2. 缓存权限

权限	功能	Description
READ	**get，包含**	从缓存检索条目。
写	**put,putIfAbsent,replace,remove,evict**	在缓存中写入、替换、删除、驱除数据。
EXEC	**distexec,流**	允许对缓存执行代码。
LISTEN	**addListener**	根据缓存注册监听程序。
BULK_READ	**keySet,值,entrySet,query**	执行批量检索操作。
BULK_WRITE	**清除，放置All**	执行批量写入操作。
生命周期	启动，**停止**	启动和停止缓存。
ADMIN	**getVersion,addInterceptor***, **removeInterceptor Chain ,getEvictionManager,getComponentRegistry,getDistributionManager,getAuthorizationManager,evict,getRpcManager**, getCacheConfiguration ,**getCacheConfiguration**, **getCacheManager,getInvocationContextContainer,setAvailability,getDataContainer,getStats,getXAResource**	允许访问底层组件和内部结构。
MONITOR	**getStats**	允许访问 JMX 统计数据和 **指标端点**。
ALL	-	包括所有缓存权限。
ALL_READ	-	组合 READ 和 BULK_READ 权限。
ALL_WRITE	-	组合 WRITE 和 BULK_WRITE 权限。

其他资源

- Data Grid Security API

3.1.2. 角色和权限映射器

数据网格用户通过 **javax.security.auth.Subject** 类实施，并代表一组类型为 **java.security.Principal** 的安全主体。

Data Grid 包含 **PrincipalRoleMapper** API，将安全主体与角色以及 **RolePermissionMapper** API 与权限集关联。数据网格还提供以下角色和权限映射程序实施：

集群角色映射器

在集群 registry 中存储到角色映射的主体。

集群权限映射程序

将角色存储到集群 registry 中的权限映射，并允许您动态修改用户角色和权限。

身份角色映射器

使用主体名称作为角色名称。主体名称的类型或格式取决于源。例如，在 LDAP 目录中，主体名称可以是可辨识的名称(DN)。

通用名称角色映射程序

使用通用名称(CN)作为角色名称。您可以将此角色 mapper 与包含可辨uished Names (DN)的 LDAP 目录一起使用；例如 **cn=managers,ou= people,dc=example,dc=com** 映射到 **managers** 角色。

其他资源

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

3.1.3. 配置角色映射器

Data Grid 默认启用集群角色映射程序和集群权限映射程序。如果要使用身份角色映射程序、通用名称(CN)角色映射程序或自定义实施，您应该配置角色映射程序。例如，如果您的部署与 LDAP 目录集成，且您想要使用可辨识的名称(DN)作为安全主体，您可以将 Data Grid 配置为使用通用名称(CN)角色映射程序。

流程

1. 打开 Data Grid 配置进行编辑。
2. 在 Cache Manager 配置中声明 role mapper 作为安全授权的一部分。
3. 保存对您的配置的更改。

通过使用嵌入式缓存，您可以使用 **principalRoleMapper** （） 和 **rolePermissionMapper** （） 方法，以编程方式配置角色和权限映射程序。

角色映射程序配置

XML

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

JSON

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

YAML

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

其他资源

- Data Grid 配置模式参考

3.2. 为嵌入式缓存启用和配置授权

在使用嵌入式缓存时，您可以使用 **GlobalSecurityConfigurationBuilder** 和 **ConfigurationBuilder** 类配置授权。

流程

1. 构建 **GlobalConfigurationBuilder**，并使用 **security （）.authorization （）.enable （）** 方法启用安全授权。
2. 使用 **principalRoleMapper （）** 方法指定 role mapper。
3. 如果需要，使用 **role （）** 和 **permission （）** 方法定义自定义角色和权限映射。

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
 global.security().authorization().enable()
```

```
        .principalRoleMapper(new ClusterRoleMapper())
        .role("myroleone").permission(AuthorizationPermission.ALL_WRITE)
        .role("myroletwo").permission(AuthorizationPermission.ALL_READ);
```

4. 在 **ConfigurationBuilder** 中为缓存启用授权。

- 从全局配置添加所有角色。

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

- 为缓存显式定义角色，以便 Data Grid 对没有角色的用户拒绝访问。

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

其他资源

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

3.3. 在运行时添加授权角色

将安全授权与 Data Grid 缓存结合使用时，将角色动态映射到权限。

先决条件

- 为内嵌缓存配置授权。
- 为 Data Grid 具有 **ADMIN** 权限。

流程

1. 获取 **RolePermissionMapper** 实例。
2. 使用 **addRole （）** 方法定义新角色。

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

其他资源

- org.infinispan.security.RolePermissionMapper

3.4. 使用安全缓存执行代码

当您为使用安全授权的嵌入式缓存构建 **DefaultCacheManager** 时，缓存管理器会返回 **SecureCache**，它会在调用任何操作前返回检查安全上下文。**SecureCache** 还确保应用无法检索较低级别的不安全对象，如 **DataContainer**。因此，您必须使用具有相应权限级别的角色的 Data Grid 用户执行代码。

先决条件

- 为内嵌缓存配置授权。

流程

1. 如有必要，从 Data Grid 上下文或 **AccessControlContext** 检索当前主题：

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

2. **PrivilegedAction** 中的嵌套方法调用，以便使用 Subject 执行它们。

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

注意

您可以使用 **Security.doAs （）** 或 **Subject.doAs （）** 方法。数据网格建议 **Security.doAs （）** 提高性能。

其他资源

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

3.5. 配置访问控制列表(ACL)缓存

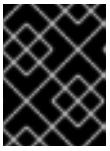
当您向用户授予或拒绝角色时，Data Grid 会存储用户可在内部访问缓存的详细信息。此 ACL 缓存避免了需要数据网格来计算安全授权的性能，如果用户具有执行每个请求的读写操作的适当权限，则此 ACL 缓存可提高性能。

注意

每当向用户授予或拒绝角色时，Data Grid 会清除 ACL 缓存以确保其正确应用用户权限。这意味着，每次授予或拒绝角色时，Data Grid 必须重新计算所有用户的缓存权限。为了获得最佳性能，您不应在生产环境中频繁或重复授予和拒绝角色。

流程

1. 打开 Data Grid 配置进行编辑。
2. 使用 **cache-size** 属性指定 ACL 缓存的最大条目数。
   ACL 缓存中的条目有卡式 **缓存：* 用户**。您应该将最大条目数设置为可保存所有缓存和用户信息的值。例如，默认大小为 **1000**，适合部署最多 100 个缓存和 10 个用户。
3. 使用 **cache-timeout** 属性以毫秒为单位设置超时值。
   如果 Data Grid 无法访问该条目的超时期间内 ACL 缓存中的条目。当用户随后尝试缓存操作时，Data Grid 会重新计算其缓存权限，并在 ACL 缓存中添加条目。



重要

为 **cache-size** 或 **cache-timeout** 属性指定值 **0** 可禁用 ACL 缓存。只有在禁用授权时才会禁用 ACL 缓存。

4. 保存对您的配置的更改。

ACL 缓存配置

XML

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

JSON

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

YAML

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

其他资源

- Data Grid 配置模式参考

第 4 章 启用和配置数据网格统计信息和 JMX 监控

数据网格可以提供缓存管理器和缓存统计信息，以及导出 JMX MBean。

4.1. 在嵌入缓存中启用统计

配置 Data Grid，以导出缓存管理器和嵌入式缓存的统计信息。

流程

1. 打开 Data Grid 配置进行编辑。
2. 添加 **statistics="true"** 属性或 **.statistics (true)** 方法。
3. 保存并关闭您的数据网格配置。

嵌入式缓存统计

XML

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

GlobalConfigurationBuilder

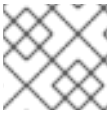
```
GlobalConfigurationBuilder global =
GlobalConfigurationBuilder.defaultClusteredBuilder().cacheContainer().statistics(true);
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());

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

4.2. 配置 DATA GRID 指标

数据网格会生成与任何监控系统兼容的指标。

- 量表提供值，如用于写操作或 JVM 运行时间的平均纳秒数。
- histograms 提供有关读取、写入和删除时间等操作执行时间的详细信息。

默认情况下，Data Grid 在启用统计数据时会生成量表，但您也可以将其配置为生成直方图。



注意

数据网格指标在 **供应商** 范围内提供。与 JVM 相关的指标在 **基础** 范围内提供。

先决条件

- 您必须将 Micrometer Core 和 Micrometer Registry Prometheus JAR 添加到 classpath 中，以便为内嵌缓存导出 Data Grid 指标。

流程

1. 打开 Data Grid 配置进行编辑。
2. 将 **metrics** 元素或对象添加到缓存容器。
3. 通过量表属性或字段启用或禁用量表。
4. **使用直 方属性或字段启用或禁用直方图。**
5. **保存并关闭您的客户端配置。**

指标配置

XML

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

JSON

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

YAML

```
infinispan:
  cacheContainer:
    statistics: "true"
    metrics:
      gauges: "true"
      histograms: "true"
```

GlobalConfigurationBuilder

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

其他资源

- Micrometer Prometheus

4.3. 注册 JMX MBEANS

数据网格可以注册 JMX MBeans，用于收集统计信息和执行管理操作。您还必须为 JMX MBeans 中的所有统计属性提供 0 值。

流程

1. 打开 Data Grid 配置进行编辑。

2. 将 jmx 元素或对象添加到缓存容器，并将 true 指定为 enabled 属性或字段的值。

3. 添加 domain 属性或字段，并根据需要指定公开 JMX MBeans 的域。

4. 保存并关闭您的客户端配置。

JMX 配置

XML

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

JSON

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

YAML

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

GlobalConfigurationBuilder

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

4.3.1. 启用 JMX 远程端口

提供唯一的远程 JMX 端口，以通过 JMXServiceURL 格式的连接公开数据网格 MBeans。

您可以使用以下方法之一启用远程 JMX 端口：

- 启用需要身份验证到其中一个数据网格服务器安全域的远程 JMX 端口。
- 使用标准的 Java 管理配置选项手动启用远程 JMX 端口。

先决条件

- 对于带有身份验证的远程 JMX，使用默认安全域定义 JMX 特定的用户角色。用户必须具有读写访问权限的 controlRole 或具有只读访问权限的 monitorRole 才能访问任何 JMX 资源。

流程

通过以下方法之一启用远程 JMX 端口启动 Data Grid Server：

- 通过端口 9999 启用远程 JMX。

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

警告

禁用了 SSL 的远程 JMX 不用于生产环境。

- 在启动时将以下系统属性传递给 Data Grid 服务器：

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

警告

在无需身份验证或 SSL 的情况下启用远程 JMX 并不安全，不建议在任何环境中使用。禁用身份验证和 SSL 可让未授权用户连接到您的服务器并访问其中托管的数据。

其他资源

- 创建安全域

4.3.2. Data Grid MBeans

数据网格公开了代表可管理资源的 JMX MBeans。

org.infinispan:type=Cache

用于缓存实例的属性和操作。

org.infinispan:type=CacheManager

用于缓存管理器的属性和操作，包括数据网格缓存和集群健康统计。

有关可用 JMX MBeans 以及描述以及可用操作和属性的完整列表，请参阅 *数据网格 JMX 组件* 文档。

其他资源

- 数据网格 JMX 组件

4.3.3. 在自定义 MBean 服务器中注册 MBeans

数据网格包含一个 MBeanServerLookup 接口，可用于在自定义 MBeanServer 实例中注册 MBeans。

先决条件

- 创建 MBeanServerLookup 的实施，使 getMBeanServer （） 方法返回自定义 MBeanServer 实例。
- 配置数据网格以注册 JMX MBeans.

流程

1. 打开 Data Grid 配置进行编辑。
2. 将 mbean-server-lookup 属性或字段添加到 Cache Manager 的 JMX 配置中。
3. 指定 MBeanServerLookup 实施的完全限定域名(FQN)。
4. 保存并关闭您的客户端配置。

JMX MBean 服务器查找配置

XML

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

JSON

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

YAML

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

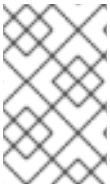
GlobalConfigurationBuilder

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

4.4. 在状态传输操作过程中导出指标

您可以为 Data Grid 在节点间重新分发的集群缓存导出时间指标。

当集群缓存拓扑更改时，会进行状态传输操作，如节点加入或离开集群。在状态传输操作期间，Data Grid 从每个缓存中导出指标，以便您可以确定缓存的状态。状态传输以属性形式公开属性，这样数据网格可以从每个缓存中导出指标。



注意

您不能以validation 模式执行状态传输操作。

数据网格会生成与 REST API 和 JMX API 兼容的时间指标。

先决条件

- 配置数据网格指标。
- 为您的缓存类型启用指标，如嵌入缓存或远程缓存。
- 通过更改集群缓存拓扑来发起状态传输操作。

流程

- 选择以下任一方法：
  - 配置 Data Grid 以使用 REST API 来收集指标。
  - 配置数据网格以使用 JMX API 来收集指标。

其他资源

- 启用和配置数据网格统计信息和 JMX 监控（数据网格缓存）
- StateTransferManager (Data Grid 14.0 API)

第 5 章 设置 DATA GRID 集群传输

数据网格需要一个传输层，以便节点可以自动加入并离开集群。传输层还使 Data Grid 节点可以在网络间复制或分发数据，并执行负载平衡和状态传输等操作。

5.1. 默认 JGROUPS 堆栈

Data Grid 在 infinispan-core-14.0.6Final-redhat-00001.jar 文件的 default-configs 目录中提供默认的 JGroups 堆栈文件 default-jgroups-*. xml。

文件名	堆栈名称	Description
default-jgroups-udp.xml	**udp**	使用 UDP 进行传输和 UDP 多播进行发现。适用于较大的集群（超过 100 个节点），或者如果您使用复制缓存或无效模式。最小化开放插槽的数量。
default-jgroups-tcp.xml	**tcp**	使用 TCP 进行传输，使用 **MPING** 协议进行发现，它使用 **UDP** 多播。*仅适用于使用分布式缓存时*（在 100 个节点下），因为 TCP 的效率比 UDP 作为点对点协议效率更高。
default-jgroups-kubernetes.xml	**kubernetes**	使用 TCP 进行传输和 **DNS_PING** 进行发现。适用于不总是可用的 UDP 多播的 Kubernetes 和 Red Hat OpenShift 节点。
default-jgroups-ec2.xml	**ec2**	使用 TCP 进行传输，并使用 **aws.S3_PING** 进行发现。适用于 UDP 多播不可用的 Amazon EC2 节点。需要额外的依赖项。
default-jgroups-google.xml	**google**	使用 TCP 进行传输和 **GOOGLE_PING2** 进行发现。适用于 UDP 多播不可用的 Google Cloud Platform 节点。需要额外的依赖项。
default-jgroups-azure.xml	**azure**	使用 TCP 进行传输和 **AZURE_PING** 进行发现。适用于不可用的 UDP 多播的 Microsoft Azure 节点。需要额外的依赖项。

其他资源

- JGroups 协议

5.2. 集群发现协议

数据网格支持不同的协议，允许节点在网络和集群中自动查找彼此。

Data Grid 可以使用两种发现机制：

- 用于处理大多数网络的通用发现协议，不依赖于外部服务。
- 依赖于外部服务的发现协议为 Data Grid 集群存储和检索拓扑信息。
例如，DNS_PING 协议通过 DNS 服务器记录执行发现。

注意

在托管平台上运行数据网格需要使用发现机制来适应各个云供应商所实施的网络限制。

其他资源

- JGroups 发现协议
- JGroup 集群传输配置 Data Grid 8.x （红帽知识库文章）

5.2.1. PING

PING 或 UDPPING 是一种通用 JGroups 发现机制，通过 UDP 协议使用动态多播。

加入后，节点会将 PING 请求发送到 IP 多播地址，以发现已位于 Data Grid 集群中的其他节点。每个节点使用包含协调器节点地址和其自身地址的数据包响应 PING 请求。c=coordinator 地址和 A=own 地址.如果没有节点响应 PING 请求，加入节点就成为新集群中的协调节点。

PING 配置示例

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

其他资源

- JGroups PING

5.2.2. TCPPING

TCPPING 是一个通用 JGroups 发现机制，为集群成员使用静态地址列表。

使用 TCPPING 时，您将把 Data Grid 集群中每个节点的 IP 地址或主机名手动指定为 JGroups 堆栈的一部分，而不是让节点能够动态发现节点。

TCPPING 配置示例

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

其他资源

- JGroups TCPPING

5.2.3. MPING

MPING 使用 IP 多播来发现数据网格集群的初始成员资格。

您可以使用 MPING 将 TCPPING 发现替换为 TCP 堆栈，并使用多caing 进行发现，而不是初始主机的静态列表。但是，您还可以将 MPING 与 UDP 堆栈搭配使用。

MPING 配置示例

```
<MPING mcast_addr="${jgroups.mcast_addr:239.6.7.8}"
```

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

其他资源

- JGroups MPING

5.2.4. TCPGOSSIP

Gossip 路由器在网络上提供一个中央位置，您的 Data Grid 集群可以检索其他节点的地址。

将 Gosssip 路由器的地址(IP:PORT)注入 Data Grid 节点，如下所示：

1. 将地址作为系统属性传递给 JVM；例如，-DGossipRouterAddress="10.10.2.4[12001]".
2. 在 JGroups 配置文件中引用该系统属性。

Gossip 路由器配置示例

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

其他资源

- JGroups Gossip Router

5.2.5. JDBC_PING

JDBC_PING 使用共享数据库存储数据网格集群的信息。此协议支持任何可以使用 JDBC 连接的数据库。

节点将其 IP 地址写入共享数据库，以便加入节点可以在网络上找到 Data Grid 集群。当节点离开 Data Grid 集群时，它们会从共享数据库中删除其 IP 地址。

JDBC_PING 配置示例

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

重要

将适当的 JDBC 驱动程序添加到类路径，以便数据网格可以使用 JDBC_PING。

其他资源

- JDBC_PING
- JDBC_PING Wiki

5.2.6. DNS_PING

JGroups DNS_PING 查询 DNS 服务器，以在 Kubernetes 环境中发现数据网格群集成员，如 OKD 和红帽 OpenShift。

DNS_PING 配置示例

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

其他资源

- JGroups DNS_PING
- Service 和 Pod 的 DNS （用于添加 DNS 条目的 Kubernetes 文档）

5.2.7. 云发现协议

数据网格包括默认的 JGroups 堆栈，它使用特定于云提供商的发现协议实施。

发现协议	默认堆栈文件	工件	版本
aws.S3_PING	default-jgroups-ec2.xml	org.jgroups.aws:jgroups-aws	2.0.1.Final
GOOGLE_PING2	default-jgroups-google.xml	org.jgroups.google:jgroups-google	1.0.0.Final
azure.AZURE_PING	default-jgroups-azure.xml	org.jgroups.azure:jgroups-azure	2.0.0.Final

为云发现协议提供依赖项

要使用 aws.S3_PING，GOOGLE_PING2 或 azure.AZURE_PING 云发现协议，您需要为数据网格提供依赖的库。

流程

- 将构件依赖项添加到项目 pom.xml 中。

然后，您可以将云发现协议配置为 JGroups 堆栈文件或系统属性的一部分。

其他资源

- JGroups aws.S3_PING

- JGroups GOOGLE_PING2
- JGroups azure.AZURE_PING

5.3. 使用默认 JGROUPS 堆栈

数据网格使用 JGroups 协议堆栈，以便节点可以在专用集群频道上互相发送其他消息。

数据网格为 UDP 和 TCP 协议提供预配置的 JGroups 堆栈。您可以使用这些默认堆栈作为构建自定义集群传输配置的起点，该配置根据您的网络要求进行了优化。

流程

之一使用默认 JGroups 堆栈之一：

- 使用 infinispan.xml 文件中的 stack 属性。

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

- 使用 addProperty （） 方法设置 JGroups 堆栈文件：

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

验证

Data Grid 记录以下信息来指示其使用的堆栈：

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

其他资源

- JGroup 集群传输配置 Data Grid 8.x （红帽知识库文章）

5.4. 自定义 JGROUPS 堆栈

调整和调优属性，以创建适用于您的网络要求的集群传输配置。

数据网格提供属性，使您能够扩展默认的 JGroups 堆栈以简化配置。您可以在组合、删除和替换其他属性的同时从默认堆栈中继承属性。

流程

1. 在 infinispan.xml 文件中创建一个新的 JGroups 堆栈声明。
2. 添加 extend 属性，并指定 JGroups 堆栈来继承属性。
3. 使用 stack.combine 属性修改继承堆栈中配置的协议属性。
4. 使用 stack.position 属性定义自定义堆栈的位置。
5. 将堆栈名称指定为 传输 配置中 stack 属性的值。

例如，您可以使用默认 TCP 堆栈使用 Gossip 路由器和对称加密进行评估，如下所示：

```
<infinispan>
  <jgroups>
    <!-- Creates a custom JGroups stack named "my-stack". -->
    <!-- Inherits properties from the default TCP stack. -->
    <stack name="my-stack" extends="tcp">
      <!-- Uses TCPGOSSIP as the discovery mechanism instead of MPING -->
      <TCPGOSSIP initial_hosts="${jgroups.tunnel.gossip_router_hosts:localhost[12001]}"
            stack.combine="REPLACE"
            stack.position="MPING" />
      <!-- Removes the FD_SOCK2 protocol from the stack. -->
```

```
    <FD_SOCK2 stack.combine="REMOVE"/>
    <!-- Modifies the timeout value for the VERIFY_SUSPECT2 protocol. -->
    <VERIFY_SUSPECT2 timeout="2000"/>
    <!-- Adds SYM_ENCRYPT to the stack after VERIFY_SUSPECT2. -->
    <SYM_ENCRYPT sym_algorithm="AES"
                 keystore_name="mykeystore.p12"
                 keystore_type="PKCS12"
                 store_password="changeit"
                 key_password="changeit"
                 alias="myKey"
                 stack.combine="INSERT_AFTER"
                 stack.position="VERIFY_SUSPECT2" />
  </stack>
  <cache-container name="default" statistics="true">
    <!-- Uses "my-stack" for cluster transport. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="my-stack"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
 </jgroups>
</infinispan>
```

6. 检查 **Data Grid** 日志，以确保其使用堆栈。

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

参考

- JGroup 集群传输配置 Data Grid 8.x （红帽知识库文章）

5.4.1. 继承属性

当您扩展 **JGroups** 堆栈时，继承属性可让您调整您要扩展的堆栈中的协议和属性。

- **stack.position** 指定要修改的协议。
- **stack.combine** 使用以下值来扩展 **JGroups** 堆栈：

值	Description
组合	覆盖协议属性。
替换	替换协议。

值	Description
INSERT_AFTER	在另一个协议后向堆栈中添加协议。不会影响您指定为插入点的协议。 JGroups 堆栈中的协议根据它们在堆栈中的位置相互影响。例如，您应该在 **SYM_ENCRYPT** 或 **ASYM_ENCRYPT** 协议后面放置等协议，以便 **NAKACK2** 已安全。
INSERT_BEFORE	在其他协议前，将协议插入到堆栈中。影响您指定为插入点的协议。
删除	从堆栈中删除协议。

5.5. 使用 JGROUPS 系统属性

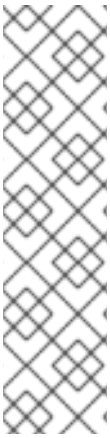
在启动时将系统属性传递给 Data Grid，以调优集群传输。

流程

- 使用 **-D<property-name>=<property-value >** 参数，根据需要设置 JGroups 系统属性。

例如，设置自定义绑定端口和 IP 地址，如下所示：

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



注意

当您在集群的红帽 JBoss EAP 应用程序中嵌入数据网格集群时，JGroups 系统属性可以清除或相互覆盖。

例如，您没有为 Data Grid 集群或红帽 JBoss EAP 应用程序设置唯一的绑定地址。在本例中，数据网格和红帽 JBoss EAP 应用都使用 JGroups 默认属性，并尝试使用相同的绑定地址来形成集群。

5.5.1. 集群传输属性

使用下列属性自定义 JGroups 集群传输：

系统属性	Description	默认值	必填/选填
jgroups.bind.address	集群传输的绑定地址。	**SITE_LOCAL**	选填
jgroups.bind.port	绑定套接字的端口。	**7800**	选填
jgroups.mcast_addr	用于多播的 IP 地址，发现和集群间通信。IP 地址必须是适合 IP 多播的有效"class D"地址。	**239.6.7.8**	选填
jgroups.mcast_port	多播套接字的端口。	**46655**	选填
jgroups.ip_ttl	IP 多播数据包的生存时间(TTL)该值定义了数据包在丢弃之前可以进行的网络跃点数。	2	选填
jgroups.thread_pool.min_threads	线程池的最小线程数量。	0	选填
jgroups.thread_pool.max_threads	线程池的最大线程数。	200	选填
jgroups.join_timeout	等待加入请求成功的最大毫秒数。	2000	选填
jgroups.thread_dumps_threshold	在记录线程转储前，线程池需要满的次数。	10000	选填
jgroups.fd.port-offset	来自 **jgroups.bind.port** 端口的偏移量 FD（失败检测协议）套接字。	**50000** （端口 **57800** ）	选填
jgroups.frag_size	消息中的最大字节数。大于碎片的消息。	60000	选填
jgroups.diag.enabled	启用 JGroups 诊断。	false	选填

其他资源

- JGroups 系统属性
- JGroups 协议列表

5.5.2. 云发现协议的系统属性

使用下列属性为托管平台配置 JGroups 发现协议。

5.5.2.1. Amazon EC2

用于配置 aws.S3_PING 的系统属性。

系统属性	Description	默认值	必填/选填
jgroups.s3.region_name	Amazon S3 区域的名称。	没有默认值。	选填
jgroups.s3.bucket_name	Amazon S3 存储桶的名称。名称必须存在，且是唯一的。	没有默认值。	选填

5.5.2.2. Google Cloud Platform

用于配置 GOOGLE_PING2 的系统属性。

系统属性	Description	默认值	必填/选填
jgroups.google.bucket_name	Google Compute Engine 存储桶的名称。名称必须存在，且是唯一的。	没有默认值。	必需

5.5.2.3. Azure

azure.AZURE_PING' 的系统属性。

系统属性	Description	默认值	必填/选填
jboss.jgroups.azure_ping.storage_account_name	Azure 存储帐户的名称。名称必须存在，且是唯一的。	没有默认值。	必需
jboss.jgroups.azure_ping.storage_access_key	Azure 存储访问密钥的名称。	没有默认值。	必需
jboss.jgroups.azure_ping.container	存储 ping 信息的容器的有效 DNS 名称。	没有默认值。	必需

5.5.2.4. OpenShift

DNS_PING 的系统属性.

系统属性	Description	默认值	必填/选填
jgroups.dns.query	设置返回群集成员的 DNS 记录。	没有默认值。	必需

5.6. 使用内联 JGROUPS 堆栈

您可以将完整的 JGroups 堆栈定义插入到 infinispan.xml 文件中。

流程

- **在您的 infinispan.xml 文件中嵌入自定义 JGroups 堆栈声明。**

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

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

5.7. 使用外部 JGROUPS 堆栈

在 infinispan.xml 文件中引用定义自定义 JGroups 堆栈的外部文件。

流程

1. 将自定义 JGroups 堆栈文件放在应用类路径上。

   另外，您还可以在声明外部堆栈文件时指定绝对路径。

2. 使用 stack-file 元素引用外部堆栈文件。

```
<infinispan>
  <jgroups>
    <!-- Creates a "prod-tcp" stack that references an external file. -->
    <stack-file name="prod-tcp" path="prod-jgroups-tcp.xml"/>
  </jgroups>
```

```
  <cache-container default-cache="replicatedCache">
    <!-- Use the "prod-tcp" stack for cluster transport. -->
    <transport stack="prod-tcp" />
    <replicated-cache name="replicatedCache"/>
  </cache-container>
  <!-- Cache configuration goes here. -->
</infinispan>
```

您还可以使用 TransportConfigurationBuilder 类中的 addProperty （） 方法来指定自定义 JGroups 堆栈文件，如下所示：

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

在本例中，my-jgroups-udp.xml 引用带有自定义属性的 UDP 堆栈，如下所示：

自定义 UDP 堆栈示例

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

其他资源

- org.infinispan.configuration.global.TransportConfigurationBuilder

5.8. 使用自定义 JCHANNELS

构建自定义 JGroups JChannel，如下例所示：

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

注意

数据网格无法使用已连接的自定义 JChannels。

其他资源

- JGroups JChannel

5.9. 加密集群传输

保护集群传输，以便节点与加密消息通信。您还可以配置 Data Grid 集群来执行证书验证，以便只有具有有效身份的节点可以加入。

5.9.1. JGroups 加密协议

为保护集群流量，您可以配置 Data Grid 节点，以使用机密密钥加密 JGroups 消息有效负载。

Data Grid 节点可以从以下任一位置获取 secret 密钥：

- 协调器节点（基本指标加密）。

- 共享密钥存储(symmetric 加密)。

从协调器节点检索 secret 密钥

您可以通过在 Data Grid 配置中将 ASYM_ENCRYPT 协议添加到 JGroups 堆栈来配置非对称加密。这允许 Data Grid 集群生成并分发 secret 密钥。

重要

在使用非对称加密时，您还应提供密钥存储，以便节点能够执行证书身份验证和安全地交换密钥。这会防止集群不受中间人(MitM)攻击。

非对称加密可以保护集群流量，如下所示：

1. Data Grid 集群中的第一个节点（协调器节点）会生成 secret 密钥。
2. 加入的节点使用协调器执行证书验证，以相互验证身份。
3. 加入的节点从协调器节点请求 secret 密钥。该请求包括加入节点的公钥。
4. 协调器节点使用公钥加密 secret 密钥，并将其返回到加入的节点。
5. 加入的节点解密并安装 secret 密钥。
6. 节点加入集群，使用 secret 密钥加密和解密信息。

从共享密钥存储检索 secret 密钥

您可以通过在 Data Grid 配置中将 SYM_ENCRYPT 协议添加到 JGroups 堆栈来配置对称加密。这允许 Data Grid 集群从您提供的密钥存储中获取 secret 密钥。

1. 节点在启动时从 Data Grid classpath 上的密钥存储安装 secret 密钥。

2. 节点加入集群，使用 secret 密钥加密和解密信息。

非对称和对称加密的比较

带有证书身份验证的 ASYM_ENCRYPT 提供了额外的加密层，与 SYM_ENCRYPT 进行比较。您提供对请求进行加密的密钥存储，以协调机密密钥的节点。Data Grid 会自动生成该 secret 密钥并处理集群流量，同时让您指定何时生成 secret 密钥。例如，您可以配置集群以在节点离开时生成新的 secret 密钥。这样可确保节点无法绕过证书身份验证，并使用旧密钥加入。

另一方面，SY M_ENCRYPT 比 ASYM_ENCRYPT 快，因为节点不需要与集群协调器交换密钥。SYM_ENCRYPT 的一个潜在的缺点是，当集群成员资格更改时，没有配置可自动生成新的 secret 密钥。用户负责生成和分发节点用于加密集群流量的 secret 密钥。

5.9.2. 使用非对称加密保护集群传输

配置 Data Grid 集群，以生成和分发加密 JGroups 消息的 secret 密钥。

流程

1. 创建具有证书链的密钥存储，使 Data Grid 能够验证节点身份。

2. 将密钥存储放在集群中的每个节点的类路径上。

   对于 Data Grid Server，您可以将密钥存储放在 $RHDG_HOME 目录中。

3. 将 SSL_KEY_EXCHANGE 和 ASYM_ENCRYPT 协议添加到数据网格配置中的 JGroups 堆栈，如下例所示：

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default
TCP stack. -->
    <stack name="encrypt-tcp" extends="tcp">
      <!-- Adds a keystore that nodes use to perform certificate authentication. -->
      <!-- Uses the stack.combine and stack.position attributes to insert
SSL_KEY_EXCHANGE into the default TCP stack after VERIFY_SUSPECT2. -->
      <SSL_KEY_EXCHANGE keystore_name="mykeystore.jks"
                        keystore_password="changeit"
                        stack.combine="INSERT_AFTER"
                        stack.position="VERIFY_SUSPECT2"/>
      <!-- Configures ASYM_ENCRYPT -->
```

```
    <!-- Uses the stack.combine and stack.position attributes to insert
ASYM_ENCRYPT into the default TCP stack before pbcast.NAKACK2. -->
    <!-- The use_external_key_exchange = "true" attribute configures nodes to use the
`SSL_KEY_EXCHANGE` protocol for certificate authentication. -->
    <ASYM_ENCRYPT asym_keylength="2048"
                  asym_algorithm="RSA"
                  change_key_on_coord_leave = "false"
                  change_key_on_leave = "false"
                  use_external_key_exchange = "true"
                  stack.combine="INSERT_BEFORE"
                  stack.position="pbcast.NAKACK2"/>
  </stack>
 </jgroups>
 <cache-container name="default" statistics="true">
  <!-- Configures the cluster to use the JGroups stack. -->
  <transport cluster="${infinispan.cluster.name}"
             stack="encrypt-tcp"
             node-name="${infinispan.node.name:}"/>
 </cache-container>
</infinispan>
```

验证

当您启动 Data Grid 集群时，以下日志消息表示集群使用 secure JGroups 堆栈：

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

只有使用 ASYM_ENCRYPT，并且可以从协调器节点获取 secret 密钥时，才可以加入集群。否则，以下消息被写入 Data Grid 日志：

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

其他资源

- JGroups 4 手册
- JGroups 4.2 Schema

5.9.3. 使用对称加密保护集群传输

配置 Data Grid 集群，以使用您提供的密钥存储的机密密钥加密 JGroups 消息。

流程

1. 创建包含机密密钥的密钥存储。

2. 将密钥存储放在集群中的每个节点的类路径上。

   对于 Data Grid Server，您可以将密钥存储放在 $RHDG_HOME 目录中。

3. 将 SYM_ENCRYPT 协议添加到数据网格配置中的 JGroups 堆栈。

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP
stack. -->
    <stack name="encrypt-tcp" extends="tcp">
      <!-- Adds a keystore from which nodes obtain secret keys. -->
      <!-- Uses the stack.combine and stack.position attributes to insert SYM_ENCRYPT into the
default TCP stack after VERIFY_SUSPECT2. -->
      <SYM_ENCRYPT keystore_name="myKeystore.p12"
                   keystore_type="PKCS12"
                   store_password="changeit"
                   key_password="changeit"
                   alias="myKey"
                   stack.combine="INSERT_AFTER"
                   stack.position="VERIFY_SUSPECT2"/>
    </stack>
  </jgroups>
  <cache-container name="default" statistics="true">
    <!-- Configures the cluster to use the JGroups stack. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="encrypt-tcp"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

验证

当您启动 Data Grid 集群时，以下日志消息表示集群使用 secure JGroups 堆栈：

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

只有使用 SYM_ENCRYPT 并且可以从共享密钥存储获取 secret 密钥时，才可以加入集群。否则，以下消息被写入 Data Grid 日志：

-

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

其他资源

- JGroups 4 手册
- JGroups 4.2 Schema

5.10. 集群流量的 TCP 和 UDP 端口

Data Grid 将以下端口用于集群传输消息：

默认端口	协议	Description
7800	TCP/UDP	JGroups 集群绑定端口
46655	UDP	JGroups 多播

跨站点复制

数据网格为 JGroups RELAY2 协议使用以下端口：

7900

对于在 OpenShift 上运行的 Data Grid 集群。

7800

如果将 UDP 用于节点和 TCP 间的流量，用于集群间的流量。

7801

如果将 TCP 用于节点和 TCP 间的流量，用于集群间的流量。

第 6 章 集群锁定

集群锁定是在数据网格集群中跨节点之间分布和共享的数据结构。集群锁定允许您运行节点间同步的代码。

6.1. 锁定 API

数据网格提供了一个 ClusteredLock API，它可让您在嵌入模式下使用数据网格时，并发执行集群中的代码。

API 由以下组成：

- ClusteredLock 会公开方法来实现集群锁定。
- ClusteredLockManager 会公开方法来定义、配置、检索和删除集群锁定。
- EmbeddedClusteredLockManagerFactory 初始化 ClusteredLockManager 实施。

所有权

数据网格支持 NODE 所有权，以便集群中的所有节点都可以使用锁定。

Reentrancy

数据网格集群锁定是非主的，因此集群中的任何节点都可以获取锁定，但只有创建锁定的节点可以释放它。

如果为同一所有者发送两个连续锁定调用，则第一个调用会获取锁定（如果锁定可用），第二个调用会被阻断。

参考

- EmbeddedClusteredLockManagerFactory

- ClusteredLockManager
- ClusteredLock

6.2. 使用集群锁定

了解如何将集群锁定与嵌入应用程序中的数据网格一起使用。

先决条件

- 将 infinispan-clustered-lock 依赖项添加到 pom.xml 中：

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

流程

1. 从缓存管理器初始化 ClusteredLockManager 接口。此接口是定义、检索和删除集群锁定的入口点。
2. 为每个集群的锁定指定唯一名称。
3. 使用 lock.tryLock (1, TimeUnit.SECONDS) 方法获取锁定。

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

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

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

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

// Define a clustered lock named 'lock'.
```

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

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

AtomicInteger counter = new AtomicInteger(0);

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

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

CompletableFuture<Boolean> call3 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r,
ex) -> {
   if (r) {
      System.out.println("lock is acquired by the call 3");
      lock.unlock().whenComplete((nil, ex2) -> {
         System.out.println("lock is released by the call 3");
         counter.incrementAndGet();
      });
   }
});

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

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

6.3. 配置内部缓存以锁定

集群锁定管理器包括存储锁定状态的内部缓存。您可以以声明性方式或以编程方式配置内部缓存。

流程

1. 定义集群中存储集群锁定状态的节点数量。默认值为 -1，它将值复制到所有节点。

2. 为缓存可靠性指定以下值之一，该可靠性控制集群锁定在集群分割为分区或多个节点时的行为方式：

- AVAILABLE ：任何分区中的节点都可以同时在锁定时操作。
- CONSISTENT ：只有属于大多数分区的节点才能锁定。这是默认值。
- 程序配置

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

GlobalConfigurationBuilder global =
GlobalConfigurationBuilder.defaultClusteredBuilder();

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

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

ClusteredLockManager clm1 =
EmbeddedClusteredLockManagerFactory.from(cm);

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

- 声明性配置

```
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:14.0
https://infinispan.org/schemas/infinispan-config-14.0.xsd"
      xmlns="urn:infinispan:config:14.0">
```

```
    <cache-container default-cache="default">
      <transport/>
      <local-cache name="default">
        <locking concurrency-level="100" acquire-timeout="1000"/>
      </local-cache>
      <clustered-locks xmlns="urn:infinispan:config:clustered-locks:14.0"
                       num-owners = "3"
                       reliability="AVAILABLE">
        <clustered-lock name="lock1" />
        <clustered-lock name="lock2" />
      </clustered-locks>
    </cache-container>
    <!-- Cache configuration goes here. -->
</infinispan>
```

参考

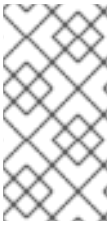
- **ClusteredLockManagerConfiguration**
- **集群锁定配置**

第 7 章 在网格中执行代码

缓存的主要优点是能够通过其键（甚至跨计算机）快速查找值。事实上，这本身是许多用户使用 Data Grid 的原因。但是，数据网格可提供并非立即明显的更多好处。由于在机器集群中通常使用 Data Grid，所以我们也有可用的功能，可帮助利用整个集群来执行用户所需的工作负载。

7.1. 集群可执行文件

既然您拥有一组计算机，因此最好利用其综合计算能力在所有这些计算机上执行代码。Cache Manager 附带一个 nice 工具，可让您在集群中执行任意代码。请注意，这个功能不需要使用缓存。通过在 EmbeddedCacheManager 上调用 executor （）来检索此集群可执行文件。https://access.redhat.com/webassets/avalon/d/red-hat-data-grid/8.4/api/org/infinispan/manager/ClusterExecutor.html这个 executor 可在集群和非集群配置中实现。



注意

ClusterExecutor 专门设计用来执行代码，这些代码不会依赖于缓存中的数据，而是用来帮助用户在集群中轻松地执行代码。

此管理器专门使用 Java 8 构建，因此在这种情况下，所有方法都采用功能接口作为参数。另外，这些参数也会发送到其他需要被序列化的节点。我们甚至使用一种不错的技巧，以确保我们都立即达到 Serializable。这种情况就是让 参数同时实现 Serializable 和 real 参数类型（例如，可运行或功能）。当确定要调用哪一种方法时，JRE 将选择最具体的类，在这种情况下，您的布局始终是序列化的。也可以使用 Externalizer 来进一步减少消息大小。

默认情况下，经理会将给定命令提交到集群中的所有节点，包括从中提交的节点。您可以使用 中的 filterTargets 方法控制任务在哪些节点上，具体如 节中所述。

7.1.1. 过滤执行节点

可以限制将要运行的命令的节点。例如，您可能希望在同一机架的机器上仅运行计算。或者，您可能要在本地站点内执行一次操作，也可以再次在不同的站点上执行操作。集群 executor 可以限制其将请求发送到同一或不同机器、机架或站点级别的节点。

SameRack.java

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

-

要使用此拓扑基本过滤，您必须通过服务器提示启用拓扑感知一致的哈希。

您还可以使用基于节点地址的 predicate 进行过滤。这也可以选择在上一个代码片段中根据拓扑过滤。

我们还允许任何方式选择目标节点，方法使用指示过滤过滤哪些节点可用于执行。请注意，这也可以与 Topology 过滤结合使用，以便更加精细地控制您在集群中执行代码的位置。

Predicate.java

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

7.1.2. Timeout（超时）

集群可执行文件允许为每个调用设置超时。默认为在传输配置中配置的分布式同步超时。此超时在集群和非集群缓存管理器中可以正常工作。执行执行器可能会在超时过期时中断执行任务的线程。但是，当超时发生任何 Consumer 或 Future 时，将完成返回 TimeoutException。这个值可通过激活 超时方法并提供 所需的持续时间来覆盖。

7.1.3. 单一节点提交

集群可执行文件也可以以单一节点提交模式运行，而不是将命令提交到所有节点，而是选择通常会收到该命令的节点之一，而是仅将其提交到一个节点。每个提交都会使用不同的节点来在其上执行任务。使用 ClusterExecutor 作为 java.util.concurrent.Executor，您可能会注意到 ClusterExecutor 实施，这非常有用。

SingleNode.java

```
EmbeddedCacheManager manager = ...;
manager.executor().singleNodeSubmission().submit(...)
```

7.1.3.1. 故障切换

当在单个节点中运行时，可能还需要允许 Cluster Executor 处理给定命令处理给定命令时出现异常情况，方法是重新重试 命令。当发生这种情况时，Cluster Executor 将再次选择一个节点，以将命令重新提交到所需的故障转移尝试数量。请注意，所选节点可以是通过拓扑或 predicate 检查的任何节点。通过调用覆盖的 singleNodeSubmission 方法来启用故障切换。指定的命令将再次重新提交到单个节点，直到命令完成前没有例外，或者提交总数等于提供的故障转移计数。

7.1.4. 示例：PI Approximation

本例演示了如何使用 ClusterExecutor 来估算 PI 的值。

通过集群可执行文件，传送传送可能会大大受益于并行分布式执行。回想一下，方括号中的区域为 Sa = 4r2，圆圈为 Ca=pi*r2。将 r2 替换为第二个 equation 到第一个 pi = 4 * Ca/Sa。现在，我们可以将大量 darts 拍摄到一个平方；如果我们占有 darts shot 的 dart 排在圆圈的比率，则我们会将大量 droximate Ca/Sa 值的比例。既然我们知道，pi = 4 * Ca/Sa，我们可以轻松获得传送价值。我们带来了更优越的应用程序。在以下示例中，我们拍摄了 1 亿多语，而是对整个数据网格集群进行并行化工作，而不是对整个数据网格群集进行并行化。请注意，这会在 1 集群中正常工作，但会较慢。

```
public class PiAppx {

  public static void main (String [] arg){
    EmbeddedCacheManager cacheManager = ..
    boolean isCluster = ..

    int numPoints = 1_000_000_000;
    int numServers = isCluster ? cacheManager.getMembers().size() : 1;
    int numberPerWorker = numPoints / numServers;

    ClusterExecutor clusterExecutor = cacheManager.executor();
    long start = System.currentTimeMillis();
    // We receive results concurrently - need to handle that
    AtomicLong countCircle = new AtomicLong();
    CompletableFuture<Void> fut = clusterExecutor.submitConsumer(m -> {
      int insideCircleCount = 0;
      for (int i = 0; i < numberPerWorker; i++) {
        double x = Math.random();
        double y = Math.random();
        if (insideCircle(x, y))
```

```
            insideCircleCount++;
        }
        return insideCircleCount;
      }, (address, count, throwable) -> {
        if (throwable != null) {
          throwable.printStackTrace();
          System.out.println("Address: " + address + " encountered an error: " + throwable);
        } else {
          countCircle.getAndAdd(count);
        }
      });
      fut.whenComplete((v, t) -> {
        // This is invoked after all nodes have responded with a value or exception
        if (t != null) {
          t.printStackTrace();
          System.out.println("Exception encountered while waiting:" + t);
        } else {
          double appxPi = 4.0 * countCircle.get() / numPoints;

          System.out.println("Distributed PI appx is " + appxPi +
              " using " + numServers + " node(s), completed in " + (System.currentTimeMillis() -
start) + " ms");
        }
      });

      // May have to sleep here to keep alive if no user threads left
    }

    private static boolean insideCircle(double x, double y) {
      return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))
          <= Math.pow(0.5, 2);
    }
}
```

第 8 章 使用 STREAMS API 进行代码执行

使用 Streams API 有效处理存储在 Data Grid 缓存中的数据。

第 9 章 流

您可能希望处理缓存中的一个子集或所有数据来生成结果。这可能使 Map Reduce 的想法。数据网格允许用户执行类似的操作，但利用标准的 JRE API 来实现此目的。Java 8 引入了 流 的概念，它允许对集合的功能式操作，无需自行迭代数据。流操作可以通过与 MapReduce 非常相似。流，就像 MapReduce 一样，您可以对整个缓存执行处理，可能是一个非常大的数据集，但效率更高。

注意

流是处理缓存中存在的数据时的首选方法，因为流会自动调整到集群拓扑更改。

另外，由于我们可以控制条目的迭代方式，如果想要同时在集群中执行所有操作，则可以在缓存中更有效地执行操作。

通过调用流或 parallelStream 方法，从 entrySet、keySet 或 值 从缓存返回的集合检索 流。

9.1. 常见流操作

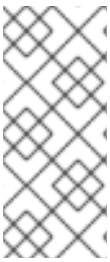
本节重点介绍了提供您正在使用的基本缓存类型的各种选项。

9.2. 密钥过滤

可以过滤流，使其仅在给定密钥的子集上运行。这可以通过调用 CacheStream 上的 filterKeys 方法来完成。这应该总是使用 Predicate 过滤器，如果 predicate 保留所有键，则更快。

如果您熟悉 AdvancedCache 接口，您可能想在这个键Filter 中使用 getAll。如果您需要像本地节点中的内存，则需要使用 getAll，则有一些小的好处（最多较小的有效负载）。但是，建议您对这些元素进行处理，因为您将同时获得分布式和线程的并行性，以便免费获得。

9.3. 基于片段的过滤



注意

这是一个高级功能，应该只用于对数据网格分段和哈希技巧的了解。如果您需要将数据分段到单独的调用中，基于这些片段的过滤很有用。这在与其他工具（如 Apache Spark ）集成时非常有用。

这个选项仅支持复制和分布式缓存。这允许用户一次处理一小部分数据，具体由 KeyPartitioner 决定。可以通过在 CacheStream 上调用 filterKeySegments 方法来过滤片段。这在键过滤器后应用，但在执行任何中间操作前。

9.4. 本地/无效

与本地或无效缓存一起使用的流只能使用常规集合上的流。如果需要，数据网格可在幕后处理所有转换，并与更为有趣的选项一起工作（例如 storeAsBinary 和 cache loader）。只有执行流操作的节点本地的数据才会被使用，例如无效仅使用本地条目。

9.5. EXAMPLE

以下代码采用缓存并返回映射，其中包含其值包含字符串"JBoss"的所有缓存条目

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

9.6. DISTRIBUTION/REPLICATION/SCATTERED

这是流逐步进入的位置。当执行流操作时，它会将各种中间和终端操作发送到每个已大数据的节点。这允许在拥有数据的节点上处理中间值，并且仅将最终结果发回到原始节点，从而提高性能。

9.6.1. Rehash Aware

数据内部被分割，每个节点仅在作为主所有者拥有的数据上执行操作。这允许均匀处理数据，假设片段足够细化，以为每个节点上提供相等的数据量。

当您使用分布式缓存时，当新节点加入或离开时，可以在节点间重新建立数据。分布式流处理自动重新处理数据，因此您不必担心在节点离开或加入集群时不必担心监控。Reshuffled 条目可能会被处理一次，我们跟踪关键级别或网段级别（取决于终端操作）的已处理条目，以限制重复处理的数量。

可能这样做，但强烈建议您禁用对流的重新哈希意识。只有在请求只能处理重新哈希时查看数据子集时才应被视为。这可通过调用 CacheStream.disableRehashAware （） 在重新哈希不完全不完全发生时，可以进行大多数操作的性能。唯一的例外是针对迭代器和每个对象，它们将使用较少的内存，因为它们不必跟踪已处理的密钥。

警告

请重新考虑禁用重新哈希意识，除非您真正了解自己所做的工作。

9.6.2. 序列化

因为操作会发送到其他节点，因此 Data Grid marshalling 必须被序列化。这允许将操作发送到其他节点。

最简单的方法是使用 CacheStream 实例，像您通常一样使用 lambda。数据网格覆盖了所有各种流中间和终端方法，以获取参数的 Serializable 版本(ie. SerializableFunction、SerializableFunction、SerializablePredicate...)，您可以在 CacheStream 找到以下方法。这依赖于 spec 来选择 此处定义 的最具体方法。

在之前的示例中，我们使用 Collector 将所有结果收集到 映射 中。遗憾的是，Collectors 类不会生成 Serializable 实例。因此，如果您需要使用这些方法，可以通过两种方式来实现：

一个选项是使用 CacheCollectors 类，它允许提供 供应商<Collector& gt;。然后，这个实例可以使用 Collectors 提供不序列化的 收集器。

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

或者，您可以避免使用 CacheCollectors，并使用采用 vendor< Collector > 的超载 收集 方法。这些超载的 收集 方法只能通过 CacheStream 接口获得。

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

但是，您无法使用 Cache 和 CacheStream 接口，则无法使用 Serializable 参数，而是必须通过将 lambdas 手动对多个接口进行序列化处理。这并不是一个非常高的，但它会让作业完成。

```
Map<Object, String> jbossValues = map.entrySet().stream()
```

```
        .filter((Serializable & Predicate<Map.Entry<Object, String>>) e ->
e.getValue().contains("Jboss"))
        .collect(CacheCollectors.serializableCollector(() ->
Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue)));
```

推荐的也是最高性能的方法是使用 高级外部化器，因为它提供了最小的有效负载。不幸的是，您不能将 lamdbas 用作高级外部化器要求在手动之前定义课程。

您可以使用高级外部化器，如下所示：

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

class ContainsFilter implements Predicate<Map.Entry<Object, String>> {
  private final String target;

  ContainsFilter(String target) {
    this.target = target;
  }

  @Override
  public boolean test(Map.Entry<Object, String> e) {
    return e.getValue().contains(target);
  }
}

class JbossFilterExternalizer implements AdvancedExternalizer<ContainsFilter> {

  @Override
  public Set<Class<? extends ContainsFilter>> getTypeClasses() {
    return Util.asSet(ContainsFilter.class);
  }

  @Override
  public Integer getId() {
    return CUSTOM_ID;
  }

  @Override
  public void writeObject(ObjectOutput output, ContainsFilter object) throws IOException {
    output.writeUTF(object.target);
  }

  @Override
  public ContainsFilter readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
    return new ContainsFilter(input.readUTF());
  }
}
```

您还可以为收集器供应商使用高级外部化器来进一步降低有效负载大小。

```
Map<Object, String> map = (Map<Object, String>) cache.entrySet().stream()
            .filter(new ContainsFilter("Jboss"))
            .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

 class ToMapCollectorSupplier<K, U> implements Supplier<Collector<Map.Entry<K, U>, ?,
Map<K, U>>> {
    static final ToMapCollectorSupplier INSTANCE = new ToMapCollectorSupplier();

    private ToMapCollectorSupplier() { }

    @Override
    public Collector<Map.Entry<K, U>, ?, Map<K, U>> get() {
      return Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue);
    }
  }

  class ToMapCollectorSupplierExternalizer implements
AdvancedExternalizer<ToMapCollectorSupplier> {

    @Override
    public Set<Class<? extends ToMapCollectorSupplier>> getTypeClasses() {
      return Util.asSet(ToMapCollectorSupplier.class);
    }

    @Override
    public Integer getId() {
      return CUSTOM_ID;
    }

    @Override
    public void writeObject(ObjectOutput output, ToMapCollectorSupplier object) throws
IOException {
    }

    @Override
    public ToMapCollectorSupplier readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      return ToMapCollectorSupplier.INSTANCE;
    }
  }
```

9.7. PARALLEL COMPUTATION

默认情况下，分布式流会尝试尽可能并行化。最终用户可以控制这一点，实际上它们必须始终控制其中一个选项。这些流的并行化方式有两种。

从最终用户从缓存集合创建流时，可以选择调用流或 parallelStream 方法时，对每个节点的 本地进程。https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#stream--根据确定的并行流

是否在本地为每个节点启用多个线程。请注意，一些操作（如重新哈希）和每个操作都会在本地使用后续流。在一些时候可以改进，允许本地并行流。

在使用本地并行性时，用户应该小心，因为它需要大量条目或操作（计算代价非常昂贵）。同样也应该注意，如果用户使用 的并行流，每个 操作不应该阻止，因为这会在通用池中执行，通常为计算操作保留。

当有多个节点时，可能需要控制远程请求一次是否同时处理远程请求。默认情况下，除了它执行并发请求外的所有终端操作。迭代器(iterator)用来减少本地节点上的总体内存压力，仅执行后续请求，实际上性能稍好。

如果用户希望更改此默认值，但是用户可以通过在 CacheStream 上调用 sequentialDistribution 或 parallelDistribution 方法来实现。

9.8. 任务超时

可以为操作请求设置超时值。此超时仅用于远程请求超时，并基于每个请求。前者意味着本地执行不会超时，后者意味着，如果您拥有故障转移方案，如后续请求上方会有一个新的超时。如果没有指定超时，它将使用复制超时作为默认超时。您可以通过执行以下操作在任务中设置超时时间：

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

有关此信息，请查看 timeout javadoc中的 java doc。

9.9. 注入

Stream 有一个名为 的终端操作，每个 操作都允许对数据运行某种副作用。在这种情况下，可能需要获得支持这个流的缓存的引用。如果您的 消费者 使用 CacheAware 接口，在来自 Consumer 接口的 accept 方法前调用 injectCache 方法。

9.10. 分布式流执行

分布式流执行的工作方式与映射减少非常相似。除非在这种情况下，我们发送零到多个中间操作（映射、过滤器等），以及单个终端操作到各个节点。操作基本上可以分为以下几项：

1. 需要的片段分组在一起，节点是给定片段的主所有者

2. 生成请求以发送到包含中间和终端操作的每个远程节点，包括它应该处理的片段

   a. 如果需要，将在本地执行终端操作

   b. 每个远程节点接收这个请求并运行操作，然后发回响应

3. 然后，本地节点将收集本地响应和远程响应，一起执行操作本身所需的任何减少。

4. 最终的减少响应将返回到用户

在大多数情况下，所有操作都完全分发，因为操作都是在每个远程节点中完全应用，而且通常仅应用最后一个操作或相关操作或相关操作，以降低多个节点的结果。个重要的一点是，中间值不一定是序列化的，它是返回所需部分的最后一个值（下将突出显示各种操作除外）。

终端操作员 分发结果 后，以下段落描述了分布式减少对各种终端操作器的工作量。其中一些是特殊的，因为中间值可能需要序列化而不是最终结果。

AllMatch noneMatch anyMatch

allMatch 操作在每个节点上运行，然后所有结果都在本地进行逻辑，以获得适当的值。noneMatch 和 anyMatch 操作使用逻辑或代替。在已知最终结果后，这些方法还会有早期终止支持，并停止远程和本地操作。

collect

收集 方法非常有趣，它执行几个额外的步骤。远程节点正常执行所有操作，但不在结果上执行最终完成程序，而是发送回完全组合的结果。https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html#finisher--然后，本地线程 会将 远程和本地结果组合成一个值，然后是最后完成的。这里要记住的一点是，最终值不一定是序列化的，而是从供应商和 组合 方法产生的值。https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html#supplier--

数量

count 方法只是将数字添加到每个节点中。

findAny findFirst

findAny 操作只返回他们找到的第一个值，无论是来自远程节点还是本地。请注意，该功能支持早期终止，当一个值被发现后就不会处理它。注意 findFirst 方法是特殊的，因为它需要排序的中间操

作，而这在 例外 部分中详述。

max min

max 和 min 方法会在每个节点上找到对应的 min 或 max 值，然后在本地执行最终缩减，以确保返回所有节点之间的 min 或 max。

reduce

各种减少方法 1、2、3 将最终对结果进行序列化，因为竞争者可以做到。如果您已提供，则会在本地将本地和远程结果集合在一起，然后再合并这些结果。请注意，这意味着来自 combiner 的值不必是 Serializable。

9.11. 基于密钥的 REHASH AWARE OPERATORS

迭代器、拆分 器和 各方面 都与其他终端操作员不同，重新哈希意识必须跟踪正在处理的每个网段的哪些键，而不是只是片段。这是为了保证，即使在集群成员资格更改下，也要保证一次(iterator 和 spliterator)或至少一次的行为（用于每个）。

当远程节点上调用时， 迭代 器和拆分 器操作器将返回条目批处理，其中下一个批处理仅在最后一次被完全使用后发回。执行此批处理来限制给定时间内存中有多少个条目。用户节点将保存它已处理哪些密钥，并且当给定片段从内存释放这些密钥时。这就是为何优先处理迭代器方法，因此仅一次性在内存中保存一个片段键子集，而不是从所有节点保存。

forEach （） 方法也会返回批处理，但它会在处理至少批量键后返回批处理密钥。这样，原始节点可以知道已处理哪些密钥已经被处理，以减少再次处理同一条目的几率。不幸的是，当节点意外停机时，至少可以有一次行为一次。在这种情况下，节点可能已被处理，尚未完成一个，并且在重新哈希失败操作时会再次运行那些在已完成的批处理中未完成的批处理中。请注意，添加节点不会造成这个问题，因为在收到所有响应前不会发生 rehash 故障转移。

这些操作批处理大小均由相同的值控制，可以通过调用 CacheStream 上的 distributedBatchSize 方法来配置。这个值默认为在状态传输中配置的 chunkSize。不幸的是，这个值会因内存使用量和至少一次性能而换取，而您的错误性可能有所不同。

将 迭代器 与复制和分布式缓存结合使用

当某个节点是分布式流中所有请求片段的主要或备份所有者时， Data Grid在本地执行 迭代器 或 分割器终端操作，这会在远程迭代更密集的情况下优化性能。

这个优化适用于复制和分布式缓存。但是，当使用 共享 并启用了 write-behind 的缓存存储时， Data

Grid 远程执行迭代。在本例中，远程执行迭代可确保一致性。

9.12. 中间操作例外

有一些具有特殊例外的中间操作，它们是 跳过 的，它们被排序 12. 和 不同的。https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#peek-java.util.function.Consumer-https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#sorted--所有这些方法都有某种程度，在流处理中指出它们以确保正确性，如下所示。请注意，这意味着这些操作可能会导致严重性能下降。

跳过

一个人会被限制在中间跳过操作中。然后，将在本地发出结果，以便它可以跳过相应的元素量。

排序

警告：此操作需要在本地节点上有内存的所有条目。一个人会被分成中间的排序操作。所有结果都会在本地排序。有可能计划具有分布式排序来返回元素的批处理，但这尚未实施。

不同的

警告：此操作需要在本地节点上具有所有或几乎所有内存条目。不同的是在每个远程节点上执行，然后一个人会返回这些不同的值。最后，所有这些结果都会对其执行不同的操作。

其余的中间操作会完全分发，就像预期一样。

9.13. 例子

字数

字数是经典的，如 map/reduce 模式。假定我们有密钥 → 句子存储在 Data Grid 节点上。Key 是字符串，每个句子都是一个字符串，我们必须计算出所有句子的可用词语。这种分布式任务的实现可以定义如下：

```
public class WordCountExample {

   /**
    * In this example replace c1 and c2 with
    * real Cache references
    *
    * @param args
    */
```

```
  public static void main(String[] args) {
    Cache<String, String> c1 = ...;
    Cache<String, String> c2 = ...;

    c1.put("1", "Hello world here I am");
    c2.put("2", "Infinispan rules the world");
    c1.put("3", "JUDCon is in Boston");
    c2.put("4", "JBoss World is in Boston as well");
    c1.put("12","JBoss Application Server");
    c2.put("15", "Hello world");
    c1.put("14", "Infinispan community");
    c2.put("15", "Hello world");

    c1.put("111", "Infinispan open source");
    c2.put("112", "Boston is close to Toronto");
    c1.put("113", "Toronto is a capital of Ontario");
    c2.put("114", "JUDCon is cool");
    c1.put("211", "JBoss World is awesome");
    c2.put("212", "JBoss rules");
    c1.put("213", "JBoss division of RedHat ");
    c2.put("214", "RedHat community");

    Map<String, Long> wordCountMap = c1.entrySet().parallelStream()
      .map(e -> e.getValue().split("\\s"))
      .flatMap(Arrays::stream)
      .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
  }
}
```

在这种情况下，执行上一示例中的字数非常简单。

但是，如果我们希望找到示例中最频繁的词语，该怎么做？如果再考虑这种情况，您需要首先计算所有词语并在本地可用。因此，我们有很多选项。

我们可以在收集器上使用 finisher，它会在收集所有结果后在用户线程上调用。已从先前示例中删除了一些冗余行。

```
public class WordCountExample {
  public static void main(String[] args) {
    // Lines removed

    String mostFrequentWord = c1.entrySet().parallelStream()
      .map(e -> e.getValue().split("\\s"))
      .flatMap(Arrays::stream)
      .collect(() -> Collectors.collectingAndThen(
        Collectors.groupingBy(Function.identity(), Collectors.counting()),
          wordCountMap -> {
            String mostFrequent = null;
            long maxCount = 0;
              for (Map.Entry<String, Long> e : wordCountMap.entrySet()) {
```

```
                int count = e.getValue().intValue();
                if (count > maxCount) {
                  maxCount = count;
                  mostFrequent = e.getKey();
                }
              }
              return mostFrequent;
            }));

  }
```

不幸的是，最后一个步骤只会在单个线程中运行，如果我们有很多词语可能会非常慢。可能还有另一种方法使用 Streams 并行化它。

在处理完本地节点之前，我们之前提到过，因此实际上可以使用有关映射结果的流。因此，我们可以对结果使用并行流。

```
public class WordFrequencyExample {
  public static void main(String[] args) {
    // Lines removed

    Map<String, Long> wordCount = c1.entrySet().parallelStream()
        .map(e -> e.getValue().split("\\s"))
        .flatMap(Arrays::stream)
        .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
    Optional<Map.Entry<String, Long>> mostFrequent =
wordCount.entrySet().parallelStream().reduce(
        (e1, e2) -> e1.getValue() > e2.getValue() ? e1 : e2);
```

这样，在计算最频繁的元素时，您仍然可以在本地使用所有内核。

删除特定条目

分布式流也可以用作修改数据的方式。例如，您可以删除包含特定词语的缓存中的所有条目。

```
public class RemoveBadWords {
  public static void main(String[] args) {
    // Lines removed
    String word = ..

    c1.entrySet().parallelStream()
      .filter(e -> e.getValue().contains(word))
      .forEach((c, e) -> c.remove(e.getKey()));
```

如果我们仔细注意的是序列化是什么，我们注意到，随着 lambda 捕获的那样，只有相关操作被序列化为其他 nods。但是，真正的保存部分是缓存操作是在主所有者上执行的，从而减少了从缓存中删除这些值所需的网络流量量。因为我们提供一个特殊的 BiConsumer 方法覆盖缓存，因此当每个节点中的调用将缓存传递给 BiConsumer 时，缓存不会被捕获。

以这种方式考虑将每个命令使用 的 一件事情是，底层流没有锁定。缓存移除操作仍然会自然而获得锁定，但该值可能会改变从流发现的结果。这意味着，在流读取但实际删除条目后，可能会更改该条目。

我们专门添加了一个新的变体，名为 LockedStream。

其他示例的方便

Streams API 是一个 JRE 工具，它使用一些示例。只需记住，您的操作需要以某种方式进行 Serializa。

第 10 章 使用 CDI 扩展

数据网格提供了一个可与 CDI (Contexts 和 Dependency Injection)编程模型集成的扩展，并允许您：

- 配置缓存并将其注入到 CDI Bean 和 Java EE 组件中。
- 配置缓存管理器。
- 接收缓存和缓存管理器级别事件。
- 使用 JCache 注释控制数据存储和检索。

10.1. CDI 依赖项

使用以下依赖项之一更新 pom.xml，以便在项目中包含 Data Grid CDI 扩展：

嵌入式(Library)模式

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

服务器模式

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

10.2. 注入嵌入式缓存

设置 CDI Bean 以注入嵌入式缓存。

流程

1. 创建缓存限定注释。

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

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

1 创建 @GreetingCache 限定符。

2. 添加用于定义缓存配置的制作者方法。

```
...
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.cdi.ConfigureCache;
import javax.enterprise.inject.Produces;

public class Config {

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

1 为要注入的缓存命名。

2

添加缓存限定符。

3. 添加制作者方法，以便在需要时创建集群缓存管理器

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

public class Config {

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

1

添加缓存限定符。

2

为应用程序创建 bean 一次。创建缓存管理器的生产者应该始终包含 @ApplicationScoped 注释，以避免创建多个缓存管理器。

3

创建一个新的 DefaultCacheManager 实例，它绑定到 @GreetingCache qualifier。

注意

缓存管理器是重量对象。在应用程序中运行多个缓存管理器可以降低性能。在注入多个缓存时，可以把每个缓存的限定符添加到 Cache Manager producer 方法中，或者不要添加任何限定符。

4. 将 @GreetingCache 限定符添加到您的缓存注入点。

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

```
public class GreetingService {

    @Inject @GreetingCache
    private Cache<String, String> cache;

    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

10.3. 注入远程缓存

设置 CDI Bean 以注入远程缓存。

流程

1. 创建缓存限定注释。

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

(1) 为要注入的缓存命名。

(2) creates a @RemoteGreetingCache qualifier.

2. 将 @RemoteGreetingCache 限定符添加到您的缓存注入点。

```
public class GreetingService {

    @Inject @RemoteGreetingCache
```

```
   private RemoteCache<String, String> cache;

   public String greet(String user) {
      String cachedValue = cache.get(user);
      if (cachedValue == null) {
         cachedValue = "Hello " + user;
         cache.put(user, cachedValue);
      }
      return cachedValue;
   }
}
```

注入远程缓存的提示

- 您可以在不使用限定符的情况下注入远程缓存。

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

- 如果您有一个多个 Data Grid 集群，可以为每个集群创建单独的远程缓存管理器制作程序。

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

public class Config {

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

(1) 为应用程序创建 bean 一次。创建缓存管理器的生产者应该始终包含 @ApplicationScoped 注释，以避免创建多个 Cache Manager，这是重量级对象。

(2) 创建一个新的 RemoteCacheManager 实例，它绑定到 @RemoteGreetingCache qualifier。

10.4. JCACHE 缓存注解

当 JCache 工件位于 classpath 中时，您可以使用带有 CDI 管理的 Bean 的以下 JCache 缓存注解：

@CacheResult

缓存方法调用的结果。

@CachePut

缓存方法参数.

@CacheRemoveEntry

从缓存中删除条目。

@CacheRemoveAll

从缓存中删除所有条目。

重要

目标类型： 您只能在方法上使用这些 JCache 缓存注解。

要使用 JCache 缓存注解，请在 application 的 beans.xml 文件中声明拦截器。

受管环境（应用服务器）

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.2" bean-discovery-mode="annotated">

 <interceptors>
  <class>org.infinispan.jcache.annotation.InjectedCacheResultInterceptor</class>
  <class>org.infinispan.jcache.annotation.InjectedCachePutInterceptor</class>
  <class>org.infinispan.jcache.annotation.InjectedCacheRemoveEntryInterceptor</class>
  <class>org.infinispan.jcache.annotation.InjectedCacheRemoveAllInterceptor</class>
 </interceptors>
</beans>
```

非管理环境（独立）

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.2" bean-discovery-mode="annotated">

 <interceptors>
  <class>org.infinispan.jcache.annotation.CacheResultInterceptor</class>
  <class>org.infinispan.jcache.annotation.CachePutInterceptor</class>
  <class>org.infinispan.jcache.annotation.CacheRemoveEntryInterceptor</class>
  <class>org.infinispan.jcache.annotation.CacheRemoveAllInterceptor</class>
 </interceptors>
</beans>
```

JCACHE 缓存注解示例

以下示例显示 @CacheResult 注释如何缓存 GreetingService.greet （） 方法的结果：

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

public class GreetingService {

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

使用 JCache 注解时，默认缓存使用带有其参数类型的注解方法的完全限定名称，例如：org.infinispan.example.GreetingService.greet (java.lang.String)

要使用默认缓存以外的缓存，请使用 cacheName 属性指定缓存名称，如下例所示：

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

10.5. 接收缓存和缓存管理器事件

您可以使用 CDI 事件接收缓存和缓存管理器级别的事件。

- 使用 @Observes 注释，如下例所示：

```
import javax.enterprise.event.Observes;
import org.infinispan.notifications.cachemanagerlistener.event.CacheStartedEvent;
import org.infinispan.notifications.cachelistener.event.*;

public class GreetingService {

   // Cache level events
   private void entryRemovedFromCache(@Observes CacheEntryCreatedEvent event) {
      ...
   }

   // Cache manager level events
   private void cacheStarted(@Observes CacheStartedEvent event) {
      ...
   }
}
```

第 11 章 使用 JCACHE API

数据网格提供了 JCache (JSR-107) API 的实施，它指定了用于在内存中缓存临时 Java 对象的标准 Java API。缓存 Java 对象可以帮助使用数据出现瓶颈问题，因为无法检索或数据难以计算。在内存中缓存这些类型的对象有助于通过直接从内存检索数据来加快应用程序性能，而不是执行昂贵的往返或重新计算。

11.1. 创建嵌入缓存

先决条件

1. 确保 cache-api 位于您的类路径上。

2. 将以下依赖项添加到 pom.xml 中：

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

流程

- 创建使用默认 JCache API 配置的内嵌缓存，如下所示：

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

// Retrieve the system wide Cache Manager
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
```

11.1.1. 配置嵌入缓存

- 将自定义数据网格配置的 URI 传递给 caching Provider.getCacheManager (URI) 调用，如下所示：

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

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

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

警告

默认情况下，JCache API 指定数据应存储为 storeByValue，因此对缓存的操作以外的对象状态变异，不会对缓存中存储的对象产生影响。到目前为止，使用 serialization/marshalling 进行了此实施，以便副本在缓存中存储，这种方式符合 spec。因此，如果将默认 JCache 配置用于 Data Grid，则存储的数据必须是一个摘要。

另外，也可以将 JCache 配置为通过参考来存储数据（就像 Data Grid 或 JDK Collections 工作一样）。要做到这一点，只需调用：

```
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>().setStoreByValue(false));
```

11.2. 存储和检索数据

虽然 JCache API 不会扩展 java.util.Map not java.util.concurrent.ConcurrentMap，它提供商一个键/值 API 来存储和检索数据：

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

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

不同于标准 java.util.Map，javax.cache.Cache 附带了两种基本的放置方法，称为 put 和 getAndPut。前者返回 void，后者会返回与该密钥关联的先前值。因此，JCache 中的 java.util.Map.put (K) 相当于 javax.cache.Cache.getAndPut (K)。

提示

虽然 JCache API 仅涵盖独立缓存，但可以使用持久性存储进行插入，但已考虑了集群或发布程序。javax.cache.Cache 提供了两种放置方法的原因是，标准的 java.util.Map 放置调用强制使用实施器计算上一个值。在使用持久性存储时，或者缓存是分布式的，返回之前的值可能是昂贵的操作，用户通常无需使用返回值即可调用标准的 java.util.Map.put (K)。因此，JCache 用户需要考虑返回值是否与它们相关，在这种情况下，需要调用 javax.cache.Cache.getAndPut (K)，否则他们可以调用 java.util.Map.put (K, V)，这样可避免返回之前值的潜在代价。

11.3. 比较 JAVA.UTIL.CONCURRENT.CONCURRENTMAP 和 JAVAX.CACHE.CACHE API

以下是 java. util.concurrent.Concurrent.ConcurrentMap 和 javax. cache.Cache API 提供的数据操作 API 的简要比较。

操作	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
存储和无返回	N/A	**void put(K key)**
存储和返回上一个值	**v 放置(K 密钥)**	**V getAndPut(K key)**
如果不存在，则存储	**v putIfAbsent (K 键， V 值)**	**布尔值 putIfAbsent (K 键， V 值)**
retrieve	**v get (Object 键)**	**v get (K 密钥)**
如果存在，请删除	**v remove (Object key)**	**boolean remove(K key)**
删除并返回前面的值	**v remove (Object key)**	**V getAndRemove(K key)**
删除条件	**布尔值 remove (Object 键、Object 值)**	**boolean remove(K key, V oldValue)**
替换 if present	**v replace (K key, V value)**	**布尔值替换(K 键， V 值)**
替换并返回前面的值	**v replace (K key, V value)**	**V getAndReplace(K key, V value)**
替换条件	**boolean replace(K key, V oldValue, V newValue)**	**boolean replace(K key, V oldValue, V newValue)**

比较两个 API 很明显，发现，在可能的情况下，JCache 可以避免返回前面的值，以避免操作昂贵的网络或 IO 操作。这是 JCache API 设计中的覆盖原则。实际上，在 java.util.concurrent.Concurrent.Concurrent.ConcurrentMap 中存在一组操作，但在

javax.cache.Cache 中不存在，因为它们可能会昂贵的分布式缓存中计算。唯一的例外是迭代缓存的内容：

操作	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
计算缓存大小	**int size （）**	N/A
返回缓存中的所有密钥	**Set<K> keySet()**	N/A
返回缓存中的所有值	**collection<V> values （）**	N/A
返回缓存中的所有条目	**Set<Map.Entry<K, V>> entrySet()**	N/A
迭代缓存	使用 **iterator （）** 方法 on keySet、value 或 entrySet	**iterator<Cache.Entry<K, V>> iterator （）**

11.4. 集群 JCACHE 实例

数据网格 JCache 实施超出了规范，从而能够使用标准 API 对集群缓存提供可能性。假定一个 Data Grid 配置文件配置为复制缓存，如下所示：

infinispan.xml

```
<infinispan>
  <cache-container default-cache="namedCache">
    <transport cluster="jcache-cluster" />
    <replicated-cache name="namedCache" />
  </cache-container>
</infinispan>
```

您可以使用这个代码创建缓存集群：

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

// For multiple Cache Managers to be constructed with the standard JCache API
// and live in the same JVM, either their names, or their classloaders, must
// be different.
```

```
// This example shows how to force their classloaders to be different.
// An alternative method would have been to duplicate the XML file and give
// it a different name, but this results in unnecessary file duplication.
ClassLoader tccl = Thread.currentThread().getContextClassLoader();
CacheManager cacheManager1 = Caching.getCachingProvider().getCacheManager(
    URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));
CacheManager cacheManager2 = Caching.getCachingProvider().getCacheManager(
    URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));

Cache<String, String> cache1 = cacheManager1.getCache("namedCache");
Cache<String, String> cache2 = cacheManager2.getCache("namedCache");

cache1.put("hello", "world");
String value = cache2.get("hello"); // Returns "world" if clustering is working

// --

public static class TestClassLoader extends ClassLoader {
  public TestClassLoader(ClassLoader parent) {
    super(parent);
  }
}
```

第 12 章 多映射缓存

MutimapCache 是数据网格缓存类型，可将键映射到值，每个键都可以包含多个值。

12.1. MULTIMAP CACHE

MutimapCache 是数据网格缓存类型，可将键映射到值，每个键都可以包含多个值。

12.1.1. 安装和配置

pom.xml

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

12.1.2. MultimapCache API

MultimapCache API 会公开几种与 Multimap 缓存交互的方法。这些方法在多数情况下是非阻塞的；如需更多信息，请参阅 限制。

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

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

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

  CompletableFuture<Void> put(K key, V value);

  CompletableFuture<Collection<V>> get(K key);

  CompletableFuture<Boolean> remove(K key);

  CompletableFuture<Boolean> remove(K key, V value);

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

  CompletableFuture<Boolean> containsKey(K key);
```

```
   CompletableFuture<Boolean> containsValue(V value);

   CompletableFuture<Boolean> containsEntry(K key, V value);

   CompletableFuture<Long> size();

   boolean supportsDuplicates();

}
```

CompletableFuture<Void> put(K key, V value)

在 multimap 缓存中放置键值对。

```
MultimapCache<String, String> multimapCache = ...;

multimapCache.put("girlNames", "marie")
        .thenCompose(r1 -> multimapCache.put("girlNames", "oihana"))
        .thenCompose(r3 -> multimapCache.get("girlNames"))
        .thenAccept(names -> {
                if(names.contains("marie"))
                   System.out.println("Marie is a girl name");

                 if(names.contains("oihana"))
                   System.out.println("Oihana is a girl name");
               });
```

这个代码的输出如下：

```
Marie is a girl name
Oihana is a girl name
```

CompletableFuture<Collection<V>> get(K key)

异步返回与这个多映射缓存中键关联的值视图集合（若有）。对检索的集合的任何更改都不会更改这个 multimap 缓存中的值。当此方法返回空集合时，这意味着没有找到密钥。

CompletableFuture<Boolean> remove(K key)

异步从 multimap 缓存中删除与密钥关联的条目（如果存在）。

CompletableFuture<Boolean> remove(K key, V value)

异步从多映射缓存中删除键值对（如果存在）。

CompletableFuture<Void> remove(Predicate<? super V> p)

异步方法.删除与给定 predicate 匹配的每个值。

CompletableFuture<Boolean> containsKey(K key)

如果这个多映射包含该密钥，则异步返回为 true。

CompletableFuture<Boolean> containsValue(V value)

如果这个多映射包含至少一个键的值，则异步返回为 true。

CompletableFuture<Boolean> containsEntry(K key, V value)

如果这个多映射至少包含一个键值对及值，则返回 true。

CompletableFuture<Long> size()

异步返回多映射缓存中的键值对数量。它并不会返回不同的密钥数。

boolean supportsDuplicates()

如果多映射缓存支持重复，则异步返回为 true。这意味着 multimap 的内容可以是 'a' → ['1', '1', '2']。现在，此方法总是返回 false，因为复制还没有支持。给定值的存在性是由 'equals' 和 'hashcode' 方法的合同决定。

12.1.3. 创建 Multimap 缓存

目前，MultimapCache 配置为常规缓存。这可以通过代码或 XML 配置完成。了解如何在配置 数据网格缓存中配置常规缓存。

12.1.3.1. 嵌入式模式

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager cm = ... ;

// create or obtain a MultimapCacheManager passing the EmbeddedCacheManager
MultimapCacheManager multimapCacheManager =
EmbeddedMultimapCacheManagerFactory.from(cm);

// define the configuration for the multimap cache
multimapCacheManager.defineConfiguration(multimapCacheName, c.build());

// get the multimap cache
multimapCache = multimapCacheManager.get(multimapCacheName);
```

12.1.4. 限制

几乎每个情况下，多映射缓存的行为是常规缓存，但当前版本中存在一些限制，如下所示：

12.1.4.1. 支持重复

副本尚不受支持。这意味着，多映射不包含任何重复的键值对。每当调用方法时，如果键-值对已存在，则不会添加此键-值对。用于检查键-值对的方法已在 multimap 中是否已存在 等于 和 hashcode。

12.1.4.2. 驱除

现在，驱除可以为每个键有效，而不是每个键值对运行。这意味着每当键被驱除时，与该密钥关联的所有值也会被驱除。

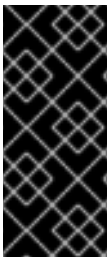
12.1.4.3. Transactions

隐式事务通过自动提交支持，所有方法都不是阻止的。在大多数情况下，显式事务可以正常发挥作用。阻塞 大小 的方法，包含Entry 和 remove (Predicate<? super V> p)

第 13 章 用于红帽 JBOSS EAP 的数据网格模块

要在部署到红帽 JBoss EAP 的应用程序中使用 Data Grid，您应该安装 Data Grid 模块：

- 让您在 WAR 或 EAR 文件中部署没有打包数据网格 JAR 文件的应用程序。
- 允许您使用独立于红帽 JBoss EAP 捆绑的数据网格版本。



重要

红帽 JBoss EAP (EAP)应用程序可以直接处理 infinispan 子系统，无需单独安装 Data Grid 模块。红帽自 EAP 7.4 版本起对这个功能提供支持。但是，您的部署需要 EAP 模块使用高级功能，如索引和查询。

13.1. 安装 DATA GRID 模块

下载并安装用于红帽 JBoss EAP 的数据网格模块。

先决条件

1. JDK 8 或更高版本。
2. 现有红帽 JBoss EAP 安装。

流程

1. 登录到红帽客户门户网站。
2. 从 Data Grid 软件下载，为模块下载 ZIP 存档。
3. 将 ZIP 归档并将 模块 内容复制到 Red Hat JBoss EAP 安装 的模块 目录中，以便您获得生成的结构：

$EAP_HOME/modules/system/add-ons/rhdg/org/infinispan/rhdg-8.4

13.2. 配置应用程序以使用数据网格模块

安装红帽 JBoss EAP 的数据网格模块后，将您的应用程序配置为使用 Data Grid 功能。

流程

1. 在项目 pom.xml 文件中，将所需的 Data Grid 依赖项标记为 *提供的*。
2. 配置工件存档器，以生成适当的 MANIFEST.MF 文件。

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-core</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-cachestore-jdbc</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-war-plugin</artifactId>
       <configuration>
         <archive>
           <manifestEntries>
             <Dependencies>org.infinispan:rhdg-8.4 services</Dependencies>
           </manifestEntries>
         </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

数据网格功能打包为单个模块 org.infinispan，您可以将其作为条目添加到应用程序清单中，如下所示：

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:rhdg-8.4 services
```

AWS 依赖项

如果您需要 AWS 依赖项，如 S3_PING，请在应用程序清单中添加以下模块：

```
Manifest-Version: 1.0
Dependencies: com.amazonaws.aws-java-sdk:rhdg-8.4 services
```