# Red Hat Data Grid 8.4

## 数据网格性能并调整指南

计划和大小 Data Grid 部署

Last Updated: 2024-04-30

# Red Hat Data Grid 8.4 数据网格性能并调整指南

计划和大小 Data Grid 部署

## 法律通告



## 摘要

部署计划首先检查红帽数据网格使用案例并决定哪些架构适合您的需求。您应该考虑对 Data Grid 功能的各种性能考虑和权衡，如数据持久性与更高的延迟。当您开始部署并运行时，您已准备好开始基准测试数据集群并验证您的性能预期。

# 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。

# 第 1 章 DATA GRID 部署模型和使用案例

数据网格提供灵活的部署模型，支持多种用例。

- 显著提高红帽 Build of Quarkus、Red Hat JBoss EAP 和 Spring 应用程序的性能。
- 确保服务可用性和连续性。
- 降低运营成本。

## 1.1. 数据网格部署模型

数据网格有两个用于缓存、远程和嵌入的部署模型。与传统数据库系统相比，两种部署模型都允许应用程序在读取操作和更高吞吐量的情况下访问数据，同时提高写操作的速度。

**远程缓存**

数据网格服务器节点在专用的 Java 虚拟机(JVM)中运行。客户端使用 Hot Rod、二进制 TCP 协议或 REST 通过 HTTP 访问远程缓存。

**嵌入缓存**

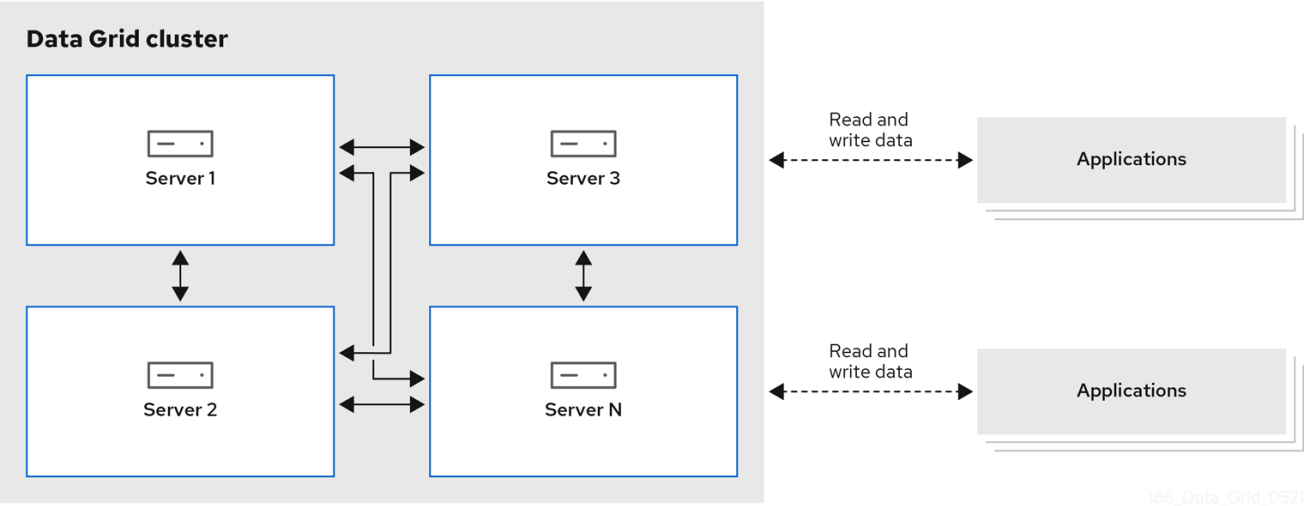
数据网格与 Java 应用程序在同一个 JVM 中运行，这意味着数据存储在执行代码的内存空间中。

红帽建议在大多数部署中使用服务器/客户端架构。使用远程缓存部署速度要快得多，因为数据层与业务逻辑分离。Data Grid Server 还提供监控和可观察性功能，以及其它内置功能以帮助您降低开发成本。

### 近乎缓存

近乎缓存功能允许远程客户端在本地存储数据，这意味着读取密集型应用程序不需要在每次调用时遍历网络。近乎缓存可显著增加读取操作速度，并获得与嵌入式缓存相同的性能。

**图 1.1. 远程缓存部署模型**




### 1.1.1. 平台和自动化工具

实现所需的服务质量意味着提供数据网格最优的 CPU 和 RAM 资源。太多资源降级了数据网格性能，同时使用过多的主机资源可以快速增加成本。

在对数据网格集群进行基准测试和调优以找到正确的 CPU 或 RAM 分配时，您应考虑哪个主机平台提供正确的自动化工具，以有效地扩展和管理资源。

### 裸机或虚拟机

结合 RHEL 或 Microsoft Windows，红帽 Ansible 用于管理数据网格配置并轮询服务，以确保可用性达到最佳资源使用。

Data Grid 的 Ansible 集合，可通过 **Automation Hub** 自动执行集群安装，并包括 Keycloak 集成和跨站点复制选项。

### OpenShift

利用 Kubernetes 编排来自动置备 pod，对资源施加限制，并自动扩展数据网格集群来满足工作负载需求。

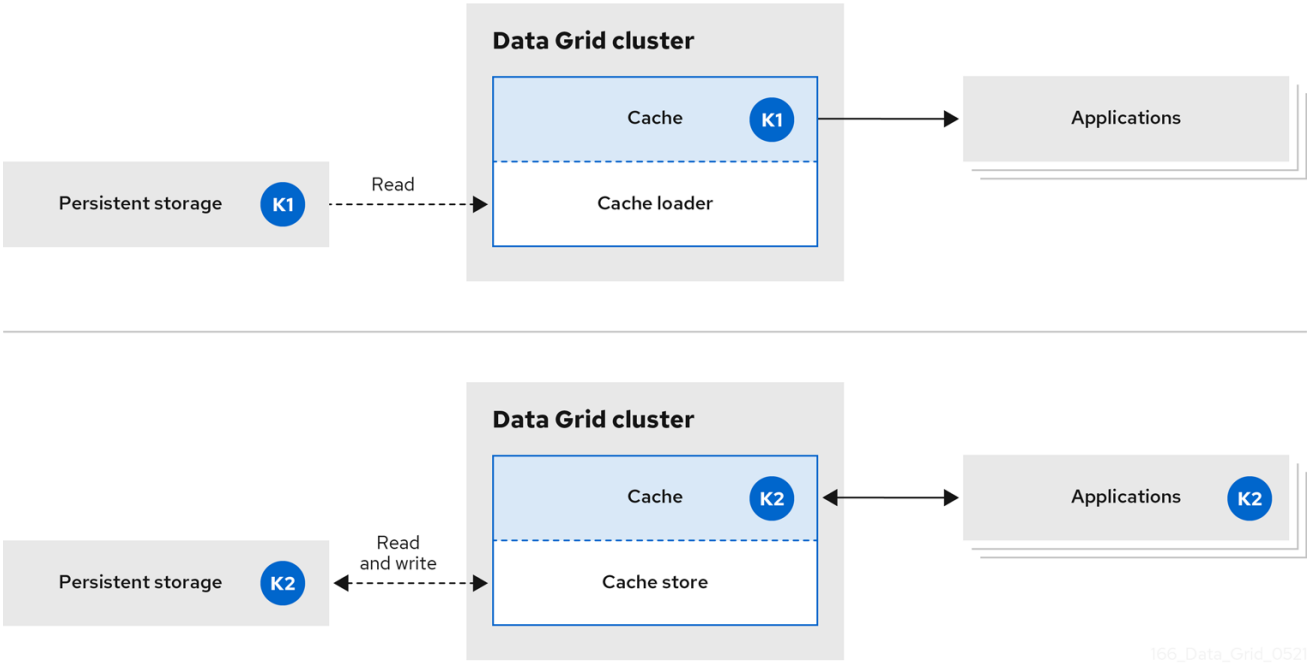
## 1.2. 在线缓存

数据网格处理位于持久存储中数据的所有应用程序请求。

通过使用在线缓存，Data Grid 使用缓存加载程序和缓存存储来操作持久性存储中的数据。

- 缓存加载程序提供对持久性存储的只读访问权限。
- 缓存存储提供对持久性存储的读写访问权限。

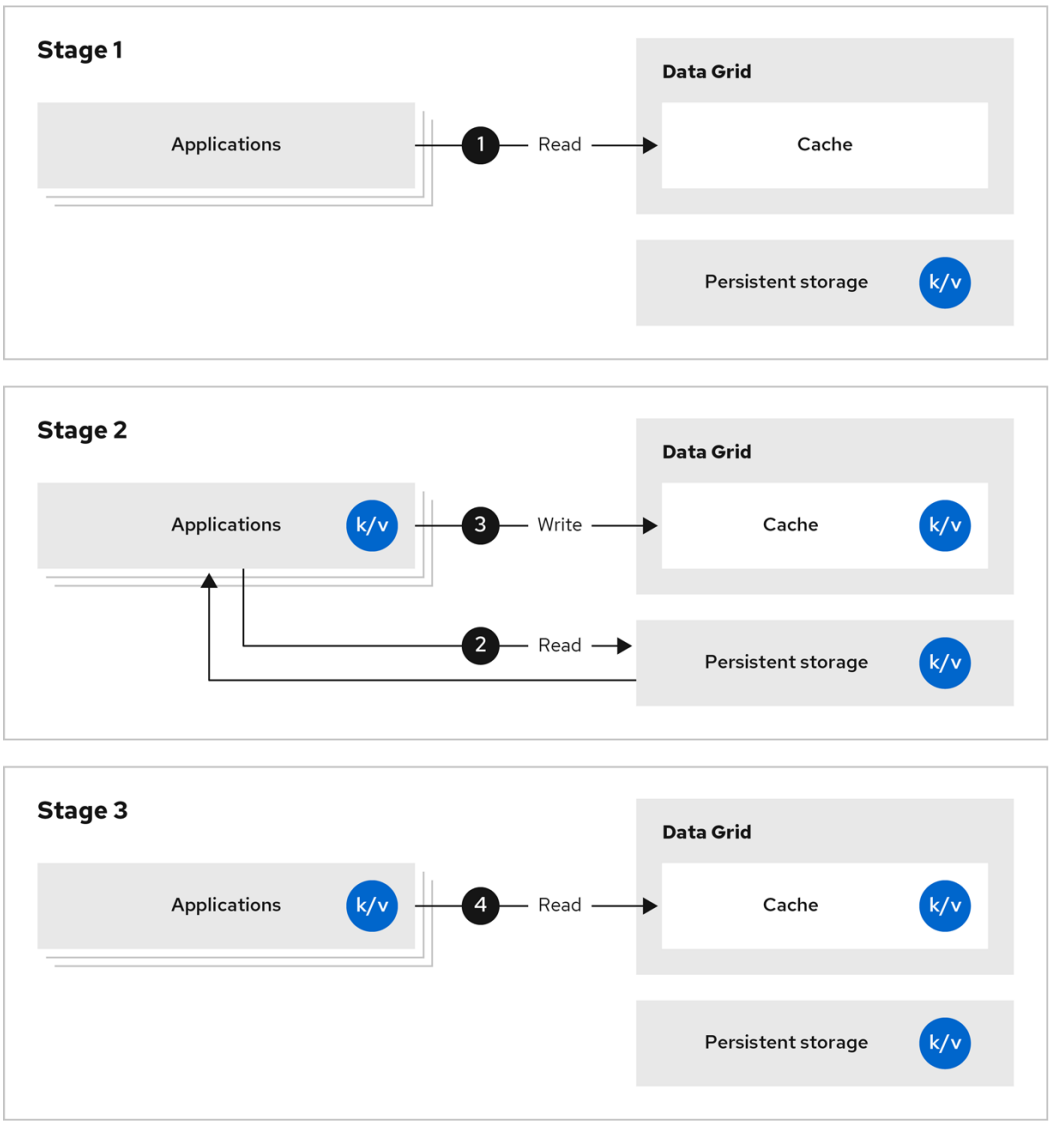
**图 1.2. 线性缓存**




## 1.3. 侧缓存

数据网格存储了应用程序从持久性存储检索的数据，从而减少了读取操作的数量到持久性存储，并延长后续读取的响应时间。

图 1.3. 侧边缓存




使用侧缓存时，应用程序控制如何将数据添加到来自持久性存储的 Data Grid 集群中。当应用程序请求条目时，会出现以下情况：

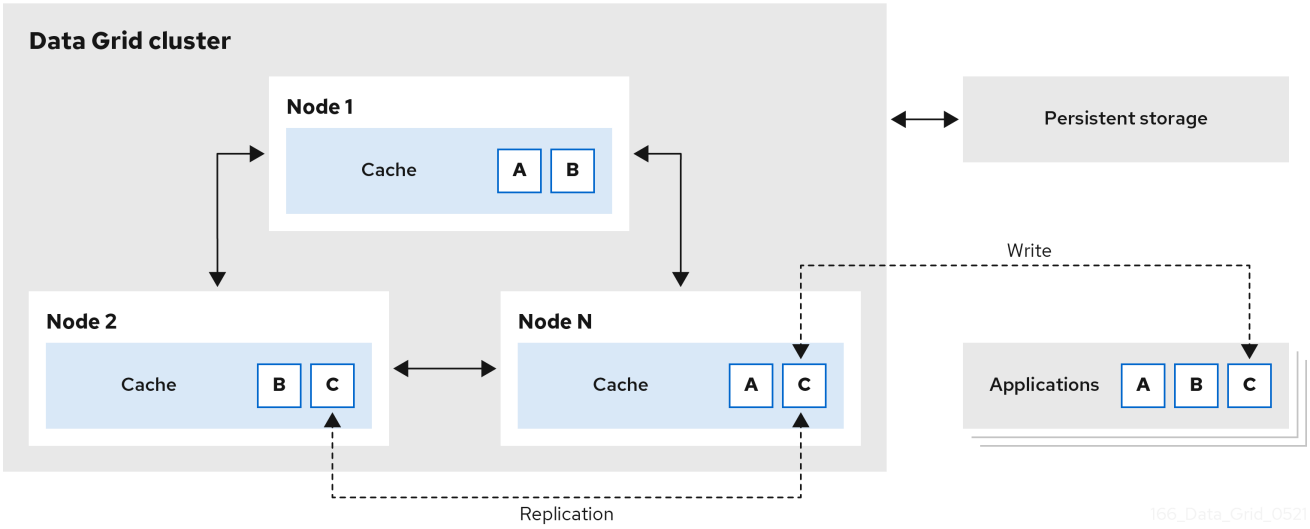
1. 读取请求进入 Data Grid。
2. 如果条目不在缓存中，应用程序会将其从持久存储中请求。
3. 应用将条目置于缓存中。
4. 该应用从 Data Grid 上检索条目，下一读信息。

## 1.4. 分布式内存

数据网格使用一致的哈希技术在集群中的缓存中存储每个条目的固定副本数。分布式缓存允许您线性扩展数据层，在节点加入时增加容量。

分布式缓存为数据网格集群添加冗余能力，以提供容错和持久保障。数据网格部署通常配置与持久性存储的集成，以保留正常关闭和从备份中恢复的群集状态。

图 1.4. 分布式缓存




## 1.5. 会话外部化

数据网格可以为在 Red Hat Build of Quarkus、Red Hat JBoss EAP、Red Hat JBoss Web Server 和 Spring 构建的应用程序提供外部缓存。这些外部缓存会独立于应用程序层存储 HTTP 会话和其他数据。

将会话外部化到数据网格将为您提供以下优势：

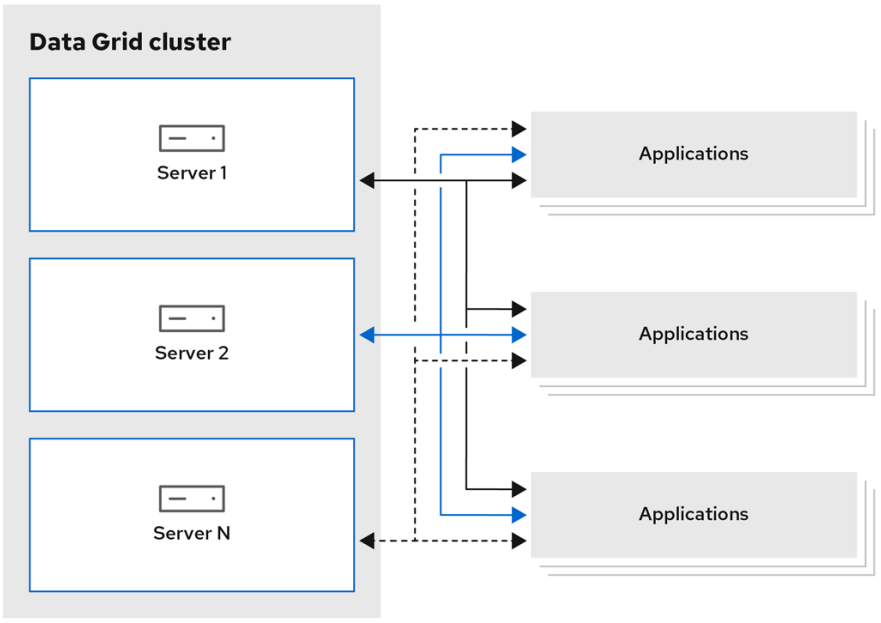
**弹性**

消除在扩展应用程序时进行重新平衡操作的需求。

**较少的内存占用量**

将会话数据存储在外部缓存中，可减少应用程序的整体内存要求。

图 1.5. 会话外部化

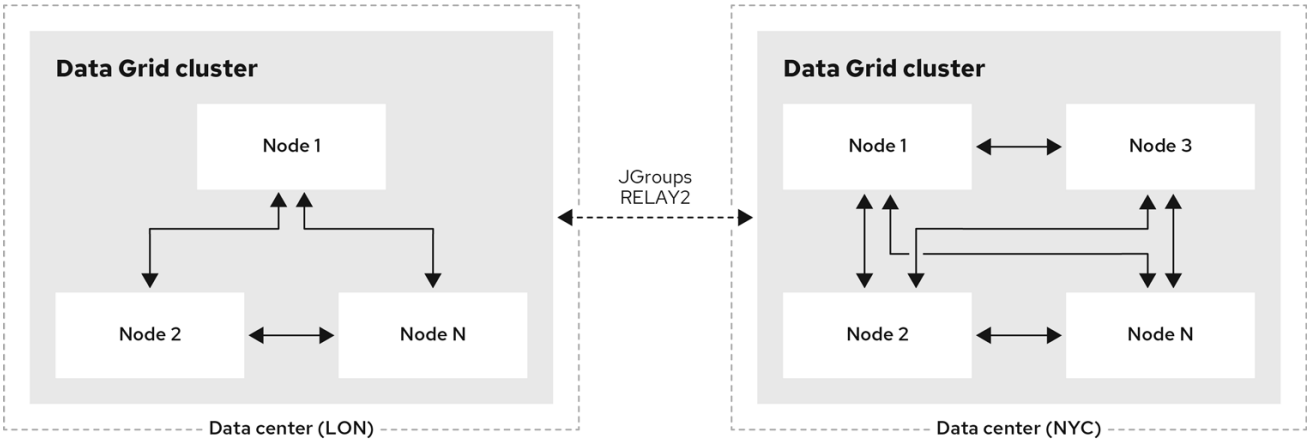



## 1.6. 跨站点复制

数据网格可在地理分散的数据中心和不同的云供应商中备份集群之间的数据。跨站点复制为数据网格提供全局集群视图和：

- 保证停机或灾难时服务连续性。
- 为客户端应用程序提供全局分布式缓存中数据的单点访问。

**图 1.6. 跨站点复制**



# 第 2 章 数据网格部署规划

要获得最好的 Data Grid 部署性能，您应该执行以下操作：

- 计算数据集合的大小。
- 确定最适合您的用例和要求的集群的缓存模式。
- 了解提供容错和一致性保障的数据网格功能的性能权衡和注意事项。

## 2.1. 性能指标注意事项

数据网格包括许多可配置的组合，决定单个公式用于涵盖所有用例的性能指标。

*数据网格性能和调整指南* 文档的目的是提供关于使用案例和架构的详细信息，可帮助您确定数据网格部署的要求。

另外，请考虑以下适用于 Data Grid 的交互因素：

- 云环境中可用的 CPU 和内存资源
- 并行使用的缓存
- get、put、query 平衡
- 峰值负载和吞吐量限制
- 使用数据集查询限制
- 每个缓存的条目数
- 缓存条目的大小

根据不同的组合和未知外部因素的数量，提供满足所有数据网格用例的性能计算是不可能的。您不能将一个性能测试与另一测试进行比较，以确定之前列出的因素是否不同。

您可以使用 Data Grid CLI 运行基本性能测试，该测试会收集有限的性能指标。您可以自定义性能测试，以便测试输出结果可能满足您的需要。测试结果可提供基线指标，可帮助您确定数据网格缓存要求的设置和资源。

测量当前设置的性能，并检查它们是否满足您的要求。如果不满足您的需求，请优化设置，然后重新保证其性能。

## 2.2. 如何计算数据集合的大小

规划数据网格部署涉及计算您的数据集的大小，然后清理正确的节点数量和 RAM 容量来保存数据集合。

您大致可以使用这个公式估算数据的总大小：

```
Data set size = Number of entries * (Average key size + Average value size + Memory overhead)
```

**注意**

使用远程缓存时，您需要在其样式中计算密钥大小和值大小。

**分布式缓存中的数据设置大小**

分布式缓存需要一些额外的计算来确定数据设置的大小。

在普通操作条件中，分布式缓存存储每个键/值条目的副本，该条目与您配置的 **所有者数** 相等。在集群重新平衡操作过程中，一些条目会有一个额外副本，因此您应该计算 **一个所有者数 + 1** 以允许这种情况。

您可以使用以下公式来调整分布式缓存的数据集合大小：

```
Distributed data set size = Data set size * (Number of owners + 1)
```

**为分布式缓存计算可用内存**

分布式缓存允许您通过添加更多节点或增加每个节点的可用内存量来增加数据设置的大小。

```
Distributed data set size <= Available memory per node * Minimum number of nodes
```

**调整节点丢失容错功能**

即使计划在集群中有固定数量的节点，也应该考虑并非所有节点始终位于集群中。分布式缓存容许丢失 **所有者的丢失 - 1** 个节点不会丢失数据，因此您可以分配很多额外的节点，除了适合您的数据集的最小节点数量外。

```
Planned nodes = Minimum number of nodes + Number of owners - 1

Distributed data set size <= Available memory per node * (Planned nodes - Number of owners + 1)
```

例如，您计划存储大小为 10KB 的 100万条目，并为每个条目配置三个所有者以供可用性。如果您计划为集群中的每个节点分配 4GB 的 RAM，您可以使用以下公式来确定数据设置所需的节点数：

```
Data set size = 1_000_000 * 10KB = 10GB
Distributed data set size = (3 + 1) * 10GB = 40GB
40GB <= 4GB * Minimum number of nodes
Minimum number of nodes >= 40GB / 4GB = 10
Planned nodes = 10 + 3 - 1 = 12
```

### 2.2.1. 内存开销

内存开销是 Data Grid 用来存储条目的额外内存。内存开销的大约估计为 JVM 堆内存中的每个条目的 200 字节，或者非堆内存中每个条目的 60 字节。但无法事先确定精确的内存开销，因为数据网格添加的开销取决于以下几个因素。例如，将数据容器与驱除绑定会导致 Data Grid 使用附加内存来跟踪条目。同样配置过期时间可为每个条目添加时间戳元数据。

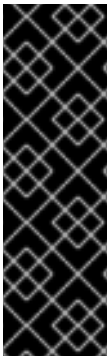
查找任何具体内存开销的唯一方法涉及 JVM 堆转储分析。当然 JVM 堆转储为您存储在非堆内存的条目提供任何信息，但内存消耗比 JVM 堆内存低得多。

**其他内存用量**

除了 Data Grid 对每个条目实施的内存开销外，重新平衡和索引等进程可能会提高总体内存用量。在节点加入并临时离开时群集的重新平衡操作需要一些额外的容量，以防止在集群成员之间复制条目时数据丢失。

### 2.2.2. JVM 堆空间分配

确定您进行数据网格部署所需的内存卷，以便有足够的数据存储容量来满足您的需求。



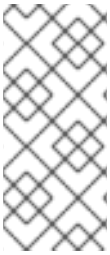
**重要**

通过以下方法分配大的内存堆大小，设置垃圾回收(GC)时间可能会影响您的数据网格部署的性能：

- 如果 JVM 只处理一个线程，GC 可能会阻止线程并降低 JVM 的性能。GC 可能会在部署前操作。这种异步行为可能会导致大 GC 暂停。
- 如果 CPU 资源较低且 GC 与部署同步运行，则 GC 可能需要更频繁地运行，从而降低部署的性能。

下表概述了为数据存储分配 JVM 堆空间的两个示例。这些示例代表部署集群的安全估算。

| | |
|---|---|
| 仅缓存操作，如读、写和删除操作。 | 为数据存储分配 50% 的 JVM 堆空间 |
| 缓存操作和数据处理，如查询和缓存事件监听程序。 | 为数据存储分配 33% 的 JVM 堆空间 |



**注意**

根据数据存储的模式变化和使用量，您可能会考虑为 JVM 堆空间设置不同百分比，而不是任何推荐的安全估算。

考虑在开始数据网格部署前设置安全估算。启动部署后，检查 JVM 的性能以及堆空间的 occupancy。当数据使用量和吞吐量显著增加时，您可能需要重新调整 JVM 堆空间。

安全估算是根据以下常见操作在 JVM 内运行的假设计算得出的。该列表并非详尽，您可能会用执行其他操作的目的来设置这些安全估算之一。

- 数据网格以序列化形式将对象转换为键值对。数据网格为缓存和持久存储添加对。
- 数据网格对缓存进行加密并解密从远程连接到客户端。
- 数据网格执行缓存的定期查询以收集数据。
- 数据网格战略性将数据划分为多个片段，以确保在群集之间高效分配数据，即使是在状态传输操作期间。
- GC 执行更频繁的垃圾回收，因为 JVM 为 GC 操作分配大量内存。
- GC 在 JVM 堆空间中动态管理和监控数据对象，以确保安全地移除未使用的对象。

为数据存储分配 JVM 堆空间以及确定数据网格部署的内存和 CPU 要求时，请考虑以下因素：

- 集群缓存模式。
- 网段数。
  - 例如，少数几个片段可能会影响服务器在节点间分发数据的方式。
- 读取或写入操作。
- 重新平衡要求。
  - 例如，在状态传输过程中可能会快速并行运行大量线程，但每个线程操作可能会使用更多内存。

- 扩展集群。
- 同步或异步复制。

需要高 CPU 资源的大多数值得注意的数据网格操作包括在 pod 重启后重新平衡节点、对数据运行索引查询以及执行 GC 操作。

### off-heap 存储

数据网格使用对象的 JVM 堆表示来处理缓存或执行其他操作的读写操作，如状态传输操作。您必须始终为 Data Grid 分配一些 JVM 堆空间，即使您将条目存储在非堆内存中。

与在 JVM 堆空间中存储数据相比，Data Grid 使用的 JVM 堆内存的卷要小得多。out-eap 存储的 JVM 堆内存要求使用并发操作数量扩展，与存储的条目数量相同。

数据网格使用拓扑缓存来为客户端提供集群视图。

如果您在 Data Grid 集群中收到任何 **OutOfMemoryError** 异常，请考虑以下选项：

- 禁用状态传输操作，如果节点加入或离开集群，这可能会导致数据丢失。
- 通过按键大小以及节点和网段数来计算 JVM 堆空间。
- 使用更多节点来更好地管理集群的内存消耗。
- 使用一个节点，因为这可能会使用较少的内存。但是，如果要将集群扩展到其原始大小，请考虑影响。

## 2.3. 集群缓存模式

您可以将集群数据网格缓存配置为复制或分布式缓存。

**分布式缓存**

通过在集群中创建每个条目的副本来最大化容量。

**复制缓存**

通过在集群中的每个节点上创建所有条目的副本来提供冗余。

### Reads:Writes

请考虑您的应用程序是否执行更多的写入操作或进行更多读取操作。通常，分布式缓存提供了在复制缓存期间写入的最佳性能，可为读取提供最佳性能。

要将 **k1** 放置在三个具有两个所有者的节点的集群上，Data Grid 写入 **k1** 两次。复制缓存中的相同操作意味着数据网格写入 **k1** 三次。每个写入复制缓存的额外网络流量等于集群中的节点数量。在 10 个节点的集群上复制的缓存会导致流量达到十倍的增加，以进行写入等。您可以通过使用带有多播进行集群传输的 UDP 堆栈来最小化流量。

要从复制的缓存中获取 **k1**，每个节点可以在本地执行读取操作。但是，要从分布式缓存中获取 **k1**，处理操作的节点可能需要从集群中的不同节点检索密钥，这会导致额外的网络跃点增加完成的操作的时间。

### 客户端智能和近乎缓存

数据网格使用一致的哈希技术使 Hot Rod 客户端拓扑感知型并避免额外的网络跃点，这意味着读取操作对分布式缓存的性能与复制缓存相同。

热环客户端也可以使用近乎缓存的功能在本地内存中保留频繁访问的条目，并避免重复读取。

**提示**

分布式缓存是大多数数据网格服务器部署的最佳选择。您可以获得读取和写入操作的最佳可能性能，以及集群扩展的弹性。

**数据保证**
因为每个节点都包含所有条目，复制的缓存会比分布式缓存提供更好的数据丢失。在三个节点的集群中，两个节点可能会崩溃，而且您不会丢失复制缓存中的数据。

在同一场景中，具有两个所有者的分布式缓存将会丢失数据。为了避免带有分布式缓存的数据丢失，您可以通过为带有 **owners** 属性声明或 **numOwners** （） 方法的每个条目配置更多所有者来增加集群中的副本数量。

**发生节点故障时重新平衡操作**

在节点失败后重新平衡操作可能会影响性能和容量。当节点离开集群时，Data Grid 在其余成员间复制缓存条目来恢复配置的所有者数量。此重新平衡操作是临时的，但增加了集群流量会对性能造成负面影响。性能下降是更多节点离开的情况。当太多节点时，集群中的节点可能没有足够的容量来保存内存中的所有数据。

**集群扩展**
数据网格集群可以随着工作负载需求而横向扩展，以便更有效地使用 CPU 和内存等计算资源。要充分利用此弹性，您应该考虑如何扩展节点数量或降低对缓存容量的影响。

对于复制的缓存，每次节点加入集群时，它都会获得数据集合的完整副本。将所有条目复制到每个节点可增加时间，使节点加入并强制实施总体容量的限制。复制的缓存永远不会超过主机可用的内存量。例如，如果您的数据集合的大小为 10 GB，则每个节点必须至少有 10 GB 的可用内存。

对于分布式缓存，添加更多节点会增加容量，因为集群的每个成员都只存储数据的一个子集。要存储 10 GB 的数据，如果所有者数量为两个，则每个节点最多可有 8 个可用内存，而不考虑内存开销。加入集群的每个附加节点会将分布式缓存的容量增加 5 GB。

分布式缓存的容量不受底层主机可用的内存量绑定。

**同步或异步复制**
当主所有者向备份节点发送复制请求时，Data Grid 可以同步或异步通信。

| 复制模式 | 对性能的影响 |
| --- | --- |
| **同步** | 同步复制有助于保持数据的一致性，但为集群流量添加延迟，以减少缓存写入吞吐量。 |
| **异步** | 异步复制会降低延迟并增加写操作的速度，但会导致数据不一致，并保证数据丢失。 |

通过同步复制，Data Grid 在复制请求在备份节点上完成时通知原始节点。如果复制请求因为集群拓扑的变化而失败，数据网格会重试操作。当复制请求因为其他错误而失败时，Data Grid 会抛出异常。

通过使用异步复制，Data Grid 并不为复制请求提供任何确认。这会对应用程序产生同样的影响，因为所有请求都成功。但是，在 Data Grid 集群中，主要所有者具有正确的条目和 Data Grid，使其在将来某个时间点备份节点。如果主所有者崩溃，备份节点可能没有条目的副本，或者它们可能没有日期副本。

集群拓扑更改也可以导致数据不一致，且使用异步复制。例如，考虑具有多个主要所有者的 Data Grid 集群。由于网络错误或某些其他问题，一个或多个主所有者会意外离开集群，因此数据网格更新哪个节点是

哪个片段的主要所有者。当发生这种情况时，某些节点可以使用旧的集群拓扑和一些节点来使用更新的拓扑。通过异步通信，这可能会导致一个短时间，因为数据网格处理来自之前拓扑的复制请求，并从写入操作应用旧的值。但是，数据网格可以检测节点崩溃并更新集群拓扑更改，因为这种情况并可能无法影响很多写操作。

使用异步复制无法保证提高了写入吞吐量，因为异步复制限制节点可在任何时间处理的备份写入数（通过 JGroups 每发送顺序）。同步复制允许节点同时处理更传入的写操作，在某些情况下，在某些配置中，单个操作需要更长的时间才能完成，从而为您提供更高的吞吐量。

当节点发送多个请求以复制条目时，JGroups一次将消息发送到集群中其他节点的其余节点，从而导致每个原始节点仅有一个复制请求。这意味着，Data Grid 节点可以和其他写入操作并行处理，一个从集群中其他节点进行写入操作。

数据网格在群集传输层使用 JGroups 流控制协议来处理对备份节点的复制请求。如果未确认的复制请求数量超过流控制阈值，使用 **max_credits** 属性（默认为 4MB）设置，在原始器节点上阻止写入操作。这适用于同步和异步复制。

**片段数量**

数据网格将数据划分为多个片段，以便在集群间平均分配数据。即使分布片段也会避免过载单个节点，并使集群重新平衡操作更高效。

默认情况下，Data Grid 为每个集群创建 256 个哈希空间片段。对于每个集群最多 20 个节点的部署，这个片段是理想的，不应更改。

对于每个集群有超过 20 个节点的部署，增加片段的数量会增加数据的粒度，使数据网格可以更有效地分发到集群中。使用以下公式来计算您应该配置多少个片段：

```
Number of segments = 20 * Number of nodes
```

例如，集群为 30 个节点，您应该配置 600 个片段。为较大的集群添加更多片段通常是一个好主意，但此公式应该为您提供了适合您部署的数量。

更改 Data Grid 创建的片段数量需要完全重启集群。如果使用持久性存储，您可能需要使用 **StoreMigrator** 程序来更改片段的数量，具体取决于缓存存储实现。

更改片段的数量也可以导致数据崩溃，因此要小心谨慎，根据您从基准测试和性能监控收集的指标。

**注意**

数据网格始终将数据存储在内存中。当您配置缓存存储时，Data Grid 并不总是将数据存储在持久性存储中。

它取决于缓存存储实施，但每当可能的情况下，您应该为缓存存储启用分段。分段缓存在迭代持久性存储中的数据时提高了数据网格性能。例如，使用基于 RocksDB 和 JDBC-string 的缓存存储，分段减少了数据网格从数据库检索的对象数量。

## 2.4. 管理陈旧数据的策略

如果 Data Grid 不是数据的主要来源，则嵌入和远程缓存的性质是陈旧的。在规划、基准测试和调整数据网格部署时，为您的应用程序选择合适的缓存过时程度。

选择一个级别，以便您可以充分利用可用的 RAM，并避免缓存未命中。如果 Data Grid 没有在内存中的条目，则在应用程序发送读写请求时，调用 转到主存储。

缓存丢失的读取和写入延迟，但在许多情况下，主要存储的调用比 Data Grid 性能损失要高得多。其中一个例子是将关系数据库管理系统(RDBMS)卸载到数据网格集群。以这种方式部署数据网格可大大降低运行传统数据库的财务成本，以便在缓存中产生更高程度的陈旧条目。

通过使用 Data Grid，您可以为条目配置最大闲置和 lifespan 值，以维护可接受的缓存过时程度。

**过期**

控制数据网格在缓存中保留条目的时长，并在集群间生效。

更高的过期值意味着，条目会保持在内存中，这会增加读操作返回过时值的可能性。较低的过期值表示缓存中存在过时的值，但缓存未命中的可能性更大。

要执行过期，Data Grid 会从现有的线程池中创建一个享受器。使用线程的主要性能考虑在过期运行之间配置正确间隔。间隔较短的间隔会执行更频繁的过期时间，但使用更多线程。

另外，您可以使用最大空闲过期来控制数据网格如何在集群中更新时间戳元数据。数据网格发送 touch 命令，以协调跨节点同步或异步的最大空闲过期时间。使用同步复制时，您可以根据首选一致性或速度选择"sync"或"async" touch 命令。

## 2.5. 使用驱除进行 JVM 内存管理

RAM 是一种昂贵的资源，通常仅限于可用性。数据网格允许您通过从内存中删除条目，来管理内存用量，为频繁使用的数据赋予优先级。

**驱除**

控制数据网格在内存中保留的数据量，并对每个节点生效。

通过以下方法驱除数据网格缓存：

- 条目总数，最多数。
- JVM 内存量，最大大小。

**重要**

数据网格按每个节点驱除条目。因为并非所有节点都驱除您应该通过持久性存储驱除的相同条目以避免数据不一致。

从驱除性能的影响源自数据网格在缓存大小达到配置的阈值时需要计算的额外处理。

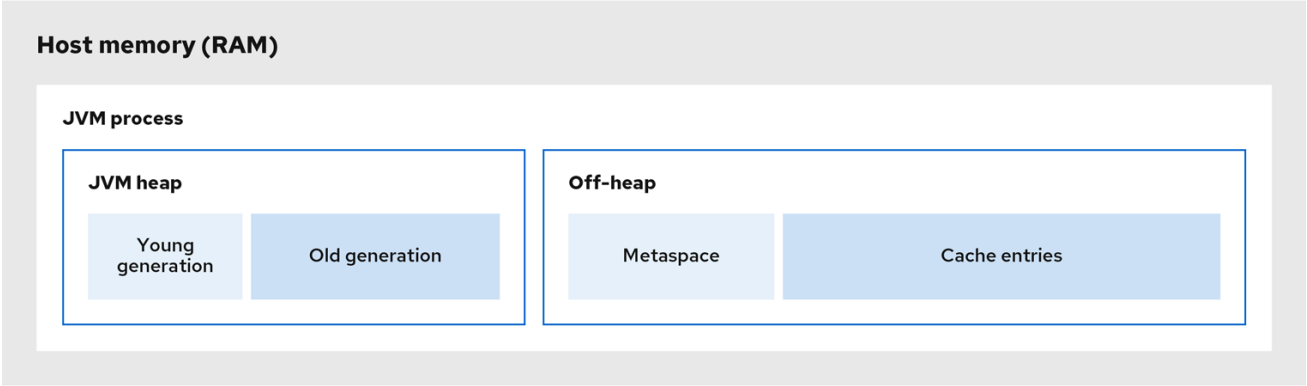
驱除也可以减慢读取操作速度。例如，如果读取操作从缓存存储检索条目，Data Grid 会将该条目引入内存中，然后驱除另一个条目。如果使用这个驱除过程，则可以使用 passivation 将新被驱除的条目添加到缓存存储中。发生这种情况时，读取操作不会返回值，直到驱除过程完成为止。

## 2.6. JVM 堆和非堆内存

默认情况下，Data Grid 将缓存条目存储在 JVM 堆内存中。您可以将 Data Grid 配置为使用非堆存储，这意味着您的数据在受管 JVM 内存空间之外发生原生内存。

下图显示了数据网格运行的 JVM 进程的内存空间简图：

图 2.1. JVM 内存空间




184_Data_Grid_0921

### JVM 堆内存

堆分成了年轻且旧的代，可帮助保持引用的 Java 对象和其他应用程序数据在内存中。GC 进程从无法访问的对象回收空间，在ng generation 内存池中运行更频繁。

当 Data Grid 将缓存条目存储在 JVM 堆内存中时，GC 运行会在开始将数据添加到缓存时完成。因为 GC 是一个密集型过程，所以较长且更频繁的运行可能会降低应用程序的性能。

### off-heap 内存

off-heap 内存是 JVM 内存管理之外的本地可用内存。*JVM 内存空间* 图显示包含类元数据的 **元空间** 内存池，以及从原生内存分配。该图还表示包含 Data Grid 缓存条目的原生内存的一部分。

非堆内存：

- 每个条目使用较少的内存。
- 通过避免 Garbage Collector (GC)运行来提高整个 JVM 性能。

然而，一个缺点是 JVM 堆转储不会显示存储在非堆内存中的条目。

## 2.6.1. 非堆数据存储

当您向非堆缓存添加条目时，Data Grid 会动态地分配原生内存到数据。

数据网格将每个密钥的序列化 **byte[]** 分为与标准 Java **HashMap** 类似的存储桶中。bucket 包括数据网格用来查找您存储在非堆内存中的条目的地址指针。

**重要**

尽管数据网格将缓存条目存储在原生内存中，但运行时操作需要这些对象的 JVM 堆表示。例如，**cache.get （）** 操作将对象读取到堆内存，然后再返回。同样，状态传输操作将对象的子集存放在堆内存中。

### 对象相等

数据网格使用每个对象的序列化字节[] 而非对象实例确定低堆存储中的 Java 对象的相等性。

### 数据一致性

数据网格使用一组锁定来保护非堆地址空间。锁定数量是两倍的内核数，然后舍入到两个最接近的幂数。这样可确保即使是 **ReadWriteLock** 实例分布，以防止写操作阻止读操作。

## 2.7. 持久性存储

将 Data Grid 配置为与持久数据源交互会严重影响性能。这种性能损失源自于以下事实：更传统的数据源本质上要慢于内存中缓存。当调用超出 JVM 时，读取和写入操作将始终更长。根据您对使用缓存存储的方式，数据网格性能的降低是性能的偏移量，导致内存数据提供了通过访问持久性存储中的数据提供了偏差。

使用持久性存储配置数据网格部署也会带来其他好处，例如，允许您为正常集群关闭保持状态。您还可以将缓存中的数据溢出到持久性存储，并在内存中仅提供容量之外获得容量。例如，您可以在总计有 1,000 万次条目，同时只保留内存中的 200万次条目。

数据网格以 write-through 模式或 write-behind 模式为缓存和持久性存储添加键/值对。由于这些写入模式对性能有不同，所以在规划数据网格部署时您必须考虑它们。

| 编写模式 | 对性能的影响 |
| --- | --- |
| **write-through** | 数据网格同时将数据写入缓存和持久性存储，这会增加一致性并避免可能导致节点故障的数据丢失。<br><br>write-through 模式的不足之处在于，同步写入添加延迟并降低吞吐量。**cache.put （）** 调用会导致应用程序线程等待到写入持久性存储完成为止。 |
| **write-behind** | 数据网格同步将数据写入缓存，但随后将修改添加到队列，以便异步写入持久性存储，从而降低一致性，同时降低写入操作的延迟。<br><br>当缓存存储无法处理写入操作数时，Data Grid 会延迟新写入操作，直到待处理的写入操作数低于配置的修改队列大小，采用类似直写的方式。如果存储通常速度快，但延迟高峰在缓存写入过程中发生，您可以提高修改队列大小来包含突发时间并缩短延迟。 |

### 激ivation

启用传递会将数据网格配置为仅在从内存中驱除条目时将条目写入持久性存储。乘客还意味着激活。对密钥执行读取或写入会使该密钥返回到内存，并将其从持久性存储中删除。在激活期间从持久性存储中删除密钥不会阻止读取或写入操作，但这会增加外部存储的负载。

传递和激活可能会导致数据网格对缓存中给定条目执行多个调用。例如，如果内存中不可用，Data Grid 会将它放回内存，即一个读取操作，以及从持久存储中删除操作。另外，如果缓存已达到大小限制，Data Grid 会执行另一个写入操作来传递新驱除条目。

### 使用数据预载入缓存

影响 Data Grid 集群性能的持久性存储的另一个方面是预加载缓存。此功能会在 Data Grid 集群启动时使用数据填充缓存，以便它们"温"，并可直接处理读取和写入。如果在持久性存储中的数据量大于可用 RAM 量时，预加载缓存可能会减慢 Data Grid 集群启动时间，导致内存不足的情况。

## 2.8. 集群安全性

保护您的数据并防止网络入侵是部署规划中最重要的方面。敏感的客户详细信息将泄漏开放互联网或数据泄漏，允许黑客公开机密信息对业务声誉的偏差。

考虑到这一点，您需要一个强大的安全策略来验证用户并加密网络通信。但您的数据网格部署性能有哪些成本？在规划过程中如何考虑这些问题？

**身份**验证

验证用户凭证的性能成本取决于机制和协议。Data Grid 通过 Hot Rod 验证凭证一次，同时可能为每个请求通过 HTTP 验证。

**表 2.1. 验证机制**

| SASL 机制 | HTTP 机制 | 性能影响 |
|---|---|---|
| **PLAIN** | **BASIC** | 尽管 **PLAIN** 和 **BASIC** 是最快的身份验证机制，但它们也是最不安全的。您应该只与 TLS/SSL 加密结合使用 **PLAIN** 或 **BASIC**。 |
| **DIGEST** 和 **SCRAM** | **摘要** | 对于 Hot Rod 和 HTTP 请求，**DIGEST** 方案使用 MD5 哈希算法来哈希凭证，因此不会以纯文本形式传输它们。如果您没有启用 TLS/SSL 加密，那么使用 **DIGEST** 的资源密集型比 **PLAIN** 或 **BASIC** 的增强性能比未安全，因为 **DIGEST** 容易受到 monkey-in-the-middle (MITM)攻击和其他入侵的情况。<br><br>对于 Hot Rod 端点，SCRAM 方案类似于 **DIGEST** 的额外级别保护，这可以提高安全性，但需要额外的处理时间才能完成。 |
| **GSSAPI** / **GS2-KRB5** | **SPNEGO** | Kerberos 服务器（密钥分发中心(KDC)）处理用户的身份验证和问题令牌。数据网格性能优势是一个独立的系统处理用户身份验证操作。但是，根据 KDC 服务本身的性能，这些机制可能会导致网络瓶颈。 |
| **OAUTHBEARER** | **BEARER_TOKEN** | 实施 OAuth 标准以向 Data Grid 用户发布临时访问令牌的联合身份提供程序。用户使用身份服务进行身份验证，而不是直接向 Data Grid 进行身份验证，并将访问令牌作为请求标头进行传递。与直接处理身份验证相比，数据网格具有较低的性能来验证用户访问令牌。与 KDC 类似，实际性能影响取决于身份提供程序本身的服务质量。 |

| SASL 机制 | HTTP 机制 | 性能影响 |
| --- | --- | --- |
| **EXTERNAL** | **CLIENT_CERT** | 您可以向 Data Grid 服务器提供信任存储，以便它通过比较客户端与信任存储提供的证书来验证入站连接。<br><br>如果信任存储仅包含签名证书（通常是证书颁发机构(CA)），则任何提供由 CA 签名的证书的客户端都可以连接到 Data Grid。这可提供较低的安全性，并容易受到 MITM 攻击的影响，但比验证每个客户端的公共证书要快。<br><br>如果信任存储除签名证书外还包含所有客户端证书，则只有信任存储中存在的签名证书的客户端才可以连接到 Data Grid。在本例中，数据网格与客户端提供的证书对比了通用名称(CN)，同时验证证书是否已签名，增加更多的开销。 |

**Encryption**
在节点间通过加密集群传输数据时保护数据，并防止您的 Data Grid 部署不受 MITM 攻击。加入集群时，节点会在加入集群时执行 TLS/SSL 握手，使用额外的往返增加延迟。但是，当每个节点建立连接后，它会始终保持连接，假设连接永远不会空闲。

对于远程缓存，Data Grid 服务器还可加密与客户端的网络通信。在客户端和远程缓存之间 TLS/SSL 连接的影响不相同。协商安全连接需要更长的时间，需要一些额外的工作，但一旦从加密建立延迟，而不是考虑数据网格性能的问题。

除了使用 TLSv1.3 外，对加密造成性能损失的唯一方式是配置数据网格运行的 JVM。对于使用 OpenSSL 库而不是标准 Java 加密的实例，可以更快地处理结果达 20%。

**授权**
基于角色的访问控制(RBAC)允许您限制对数据的操作，为部署提供额外的安全性。RBAC 是实现最小权限的策略，使用户访问在 Data Grid 集群中分发的数据具有最低权限。Data Grid 用户必须具有足够的授权级别，才能从缓存中读取、创建、修改或删除数据。

增加另一个安全层来保护数据将始终降低性能成本。授权为操作增加了一些延迟，因为数据网格在允许用户操控数据前，对每一种数据进行验证。但是，从授权性能对性能的总体影响要低于加密，因此通常可以平衡成本。

## 2.9. 客户端监听程序

当数据网格集群中添加、删除或修改数据时，客户端监听程序都会提供通知。

例如，当给定位置的温度改变时，以下实现会触发事件：

```
@ClientListener
public class TemperatureChangesListener {
  private String location;

  TemperatureChangesListener(String location) {
    this.location = location;
```

```
    }

    @ClientCacheEntryCreated
    public void created(ClientCacheEntryCreatedEvent event) {
      if(event.getKey().equals(location)) {
        cache.getAsync(location)
            .whenComplete((temperature, ex) ->
                System.out.printf(">> Location %s Temperature %s", location, temperature));
      }
    }
  }
```

在 Data Grid 集群中添加监听程序会增加部署的性能注意事项。

对于内嵌缓存，侦听程序使用与 Data Grid 相同的 CPU 内核。接收多个事件并使用大量 CPU 来处理这些事件的监听程序可减少数据网格可用的 CPU，并减慢所有其他操作的速度。

对于远程缓存，Data Grid 服务器使用内部流程来触发客户端通知。Data Grid Server 将事件从主所有者节点发送到注册侦听器的节点，然后再将其发送到客户端。数据网格服务器还包括一个后端机制，可在客户端监听器处理事件太慢时延迟写入操作以缓存缓存。

过滤监**听程序事件**

如果在每个写入操作上调用监听程序，Data Grid 会产生大量事件，在集群内部和外部客户端创建网络流量。它都取决于使用各个侦听器注册的客户端、它们触发的事件类型以及数据网格集群上的数据更改方式。

例如，如果您有 10 个客户端注册了可发出 10 个事件的监听程序，则数据网格服务器通过网络发送 100 个事件。

您可以使用自定义过滤器提供 Data Grid Server，以减少客户端的流量。过滤器允许数据网格服务器首先处理事件，并确定是否将其转发到客户端。

**持**续查询**和**监**听程序**

持续查询可让您接收匹配条目的事件，并提供部署客户端监听程序和过滤监听器事件的替代选择。当然查询需要额外的处理成本，但如果您已经索引缓存并执行查询，则可能需要使用持续查询而不是客户端监听器。

## 2.10. 索引和查询缓存

通过查询数据网格缓存，您可以分析和过滤数据以获取实时见解。例如，请考虑在线游戏，在某种程度上相互竞争。如果您要在任何一个时间使用前十个播放器实施领导板，您可以创建一个查询来找出哪个 play 是否有最多点，并将结果限制为最多十条：

```
QueryFactory queryFactory = Search.getQueryFactory(playersScores);
Query topTenQuery = queryFactory
  .create("from com.redhat.PlayerScore ORDER BY p.score DESC, p.timestamp ASC")
  .maxResults(10);
List<PlayerScore> topTen = topTenQuery.execute().list();
```

前面的示例演示了使用查询的好处，因为它可让您查找匹配有百万个缓存条目条件的十个条目。

然而，在性能影响方面，您应该考虑索引操作与查询操作相关的利弊。将 Data Grid 配置为索引缓存会导致查询更快。如果没有索引，查询必须滚动到缓存中的所有数据，根据数据的类型和数量，按量级的排序来降低结果。

在启用索引时，会有可衡量的性能损失。但是，对您要索引的内容进行仔细的规划以及对要索引的了解，您可以避免面临风险的影响。

最有效的方法是将 Data Grid 配置为仅索引所需的字段。无论您存储 Plain Old Java 对象(POJO)还是使用 Protobuf 模式，您标注的更多字段都越长，使用数据网格构建索引所需的时间。如果您有一个带有五个字段的 POJO，但您只需要查询这两个字段，请不要配置 Data Grid 来索引您不需要的三个字段。

数据网格为您提供了几个调整索引操作的选项。例如，Data Grid 存储的索引与数据不同，请将索引保存到磁盘而不是内存。每当添加、修改或删除条目时，Data Grid 使用索引写入器使索引与缓存保持同步。如果您启用索引，然后观察较慢的写操作，并认为索引会导致性能丢失，那么在写入磁盘前，将索引保留在内存中缓冲区的时间较长。这会提高索引操作，并有助于降低写入吞吐量的降级，但会消耗更多内存。对于大多数部署，默认的索引配置合适，且不会减慢写入速度。

在某些情况下，无法对缓存进行索引，例如，需要查询缓存的写重缓存，且不需要以毫秒为单位。它都取决于您要实现的目标。更快查询意味着读取速度较快，但会牺牲索引的速度较慢的写入速度。

您可以通过正确设置 **maxResults** 和 **hit-count-accuracy** 值来提高索引查询的性能。

**其他资源**

- 查询数据网格缓存

### 2.10.1. 持续查询和数据网格性能

持续查询为应用程序提供持续更新流，这样可生成大量事件。Data Grid 会临时为它生成的每个事件分配内存，这可能会导致内存压力，并可能导致 **OutOfMemoryError** 异常，特别是远程缓存。因此，您应该仔细设计持续查询，以避免产生任何性能影响。

数据网格强烈建议您将持续查询的范围限制为您所需的最小信息量。为达成此目标，您可以使用 projections 和 predicates。例如，以下语句只提供有关符合条件而非整个条目的字段子集：

```
SELECT field1, field2 FROM Entity WHERE x AND y
```

务必要确保创建的每个 **ContinuousQueryListener** 可以快速处理所有接收的事件，而不阻止线程。要达到此目的，您应该避免不必要的生成事件的缓存操作。

## 2.11. 数据一致性

驻留在分布式系统中的数据容易受到因临时网络中断、系统故障或简单的人为错误造成的错误。这些外部因素不可控制，但可能会对数据的质量造成严重后果。数据崩溃范围与客户满意度低下至昂贵的系统协调而的影响，从而会导致服务不可用。

数据网格可以执行 ACID （原子、一致、隔离、可持久的）事务，以确保缓存状态一致。

事务是指数据网格在单个操作中出出的一系列操作。事务中所有写入操作都成功完成，或者全部失败。这样，交易可以一致地修改缓存状态，提供读取和写入历史记录，或者根本不修改缓存状态。

启用事务的主要性能是，在具有更加一致的数据集和增加降低写入吞吐量的延迟之间找到平衡。

**使用事务进行写入锁定**

配置错误的锁定模式可能会对您的事务的性能造成负面影响。正确的锁定模式取决于您数据网格部署对键的高或低竞争率。

对于具有低产率的工作负载，两个或更多的交易可能无法同时写入同一密钥，而临时锁定提供了最佳性能。

数据网格在事务提交前获取对密钥的写入锁定。如果密钥有争议，则获取锁定所需的时间可能会延迟提交。另外，如果数据网格检测到冲突的写入，则它将回滚交易，应用程序必须重试，从而增加延迟。

对于具有高竞争率的工作负载，保守锁定可提供最佳性能。

当应用程序访问密钥时，数据网格获取对密钥的写入锁定，以确保其他事务无法修改密钥。事务提交在单一阶段完成，因为密钥已经被锁定。用多个关键交易锁定，会导致数据网格锁定密钥更长的时间，从而降低写吞吐量。

#### 读取隔离

除了使用 **REPEATABLE_READ** 进行优化的锁定外，隔离级别不会影响 Data Grid 性能注意事项。通过此组合，Data Grid 会检查写 skews 检测冲突，这可能会导致后续事务提交阶段。Data Grid 还使用版本元数据来检测冲突的写操作，从而可以增加每个条目的内存量，并为集群生成额外的网络流量。

#### 事务恢复和分区处理

如果网络因为分区或其他问题而变得不稳定，数据网格可将事务标记为"in-doubt"。当数据网格发生时，它会保留其获取的写入锁定，直到网络稳定，且集群会返回正常运行的运行状态。在某些情况下，系统管理员可能需要手动完成任何"对称"的事务。

## 2.12. 网络分区和降级集群

数据网格集群可能会遇到脑裂情形，因为集群中的节点子集相互隔离，并且节点之间的通信变得不连接。当发生这种情况时，在次要分区中的数据网格缓存进入 **DEGRADED** 模式，而大多数分区中的缓存仍可用。

**注意**

垃圾回收(GC)暂停是网络分区的最常见原因。当 GC 暂停导致节点变得不响应时，Data Grid 集群可以在脑裂网络中开始操作。

除了处理网络分区外，尝试通过控制 JVM 堆使用以及使用现代的低暂停 GC 实现（如 Shenandoah）来避免 GC 暂停。

#### CAP theorem 和 partition 处理策略

CAP theorem 代表分布式、键/值数据存储的限制，如 Data Grid。发生网络分区事件时，您必须在一致性或可用性之间进行选择，而数据网格修复分区并解决任何冲突的条目。

**可用性**

允许读写操作。

**一致性**

拒绝读写操作。

数据网格还可允许在将集群重新加入时进行读取。通过允许应用程序访问（潜在的过时）数据来拒绝对条目的写入和可用性，此策略具有更均衡的选择。

#### 删除分区

作为将集群重新投入到正常操作的一部分，Data Grid 会根据合并策略来解析冲突条目。

默认情况下，数据网格不会尝试解决合并冲突，这意味着集群可以更早地返回健康状态，且在正常集群重新平衡之外没有性能损失。然而，在这种情况下，缓存中的数据会更有可能不一致。

如果您配置合并策略，则数据网格需要更长的时间来修复分区。配置合并策略会导致数据网格从每个缓存中检索条目的每个版本，然后解决以下冲突：

| | |
|---|---|
| **PREFERRED_ALWAYS** | Data Grid 找到集群中大多数节点上存在的值，并应用它，这可以恢复过期的值。 |
| **PREFERRED_NON_NULL** | 数据网格应用集群中发现的第一个非null 值，可恢复日期值。 |
| **REMOVE_ALL** | 数据网格删除所有具有冲突值的条目。 |

### 2.12.1. 垃圾回收和分区处理

长垃圾回收(GC)时间可以增加 Data Grid 检测网络分区所需的时间。在某些情况下，GC 可能会导致数据网格超过检测分割的最大时间。

另外，当在分割后合并分区时，Data Grid 会尝试确认集群中存在所有节点。因为没有超时或上限应用到节点的响应时间，所以把集群视图合并的操作可能会延迟。这可能导致网络问题以及 GC 时间。

GC 在通过分区处理影响性能的另一个场景是当 GC 暂停 JVM 时，导致一个或多个节点离开集群。当发生这种情况且在 GC 完成后挂起的节点时，节点可能会过期或有冲突的集群拓扑。

如果配置了合并策略，Data Grid 会在合并节点前尝试解决冲突。但是，只有在节点有不兼容的哈希值时，才会使用合并策略。如果每个片段至少有一个通用所有者，则两个一致的哈希是兼容的，如果它们至少有一个通用所有者，则不兼容。

当节点有旧但兼容的哈希时，Data Grid 会忽略过期的集群拓扑，且不会尝试解决冲突。例如，如果一个节点因为垃圾回收而暂停，集群中的其他节点也将其从一致的哈希中移除，并将它替换为新的所有者节点。如果 **numOwners > 1**，旧的一致的哈希值以及新的一致的哈希值为每个键都有通用所有者，这使得数据网格能够兼容，并允许数据网格跳过冲突解析过程。

## 2.13. 集群备份和灾难恢复

在总体 CPU 和内存分配方面，执行跨站点复制的数据网格集群通常是"ymmetrical"。当您将跨站点复制考虑大小大小时，主要关注对群集之间的状态传输操作的影响。

例如，NYC 中的 Data Grid 集群离线，客户端在 LON 中切换到 Data Grid 集群。当 NYC 中的集群重新上线时，状态转移将从 LON 转移到 NYC。此操作可防止客户端进行过时的读取，但对集群收到状态传输的性能损失。

您可以在集群中分发该状态传输操作所需的增加。但是，状态传输操作的性能影响完全取决于数据集合的类型和大小等环境和因素。

**Active/Active 部署冲突解析**
当多个站点处理客户端请求（称为 Active/Active 站点配置）时，Data Grid 检测到并发写入操作与并发写入操作冲突。

以下示例演示了如何并发写入会导致在 **LON** 和 **NYC** 数据中心中运行的 Data Grid 集群有冲突的条目：

```
          LON       NYC

k1=(n/a)   0,0       0,0

k1=2       1,0  -->  1,0   k1=2
```

```
k1=3       1,1  <--  1,1   k1=3

k1=5       2,1       1,2   k1=8

                -->  2,1 (conflict)
(conflict)  1,2  <--
```

在 Active/Active 站点配置中，您不能使用同步备份策略，因为并发写入会导致死锁和您丢失的数据。借助异步备份策略(**strategy=async**)，数据网格为您提供处理并发写入选择跨站点合并策略。

就性能而言，数据网格用来解决冲突的合并策略需要额外的计算，但通常不会产生显著损失。例如，默认的跨站点合并策略使用字典比较，或"字符串比较"，该比较仅用几纳秒来完成。

数据网格还提供 **XSiteEntryMergePolicy** SPI 用于跨站点合并策略。如果配置数据网格来解决与自定义实施冲突，您应该始终监控性能来衡量任何负面影响。

**注意**

**XSiteEntryMergePolicy** SPI 调用非阻塞线程池中的所有合并策略。如果您实施了阻塞自定义合并策略，它可能会耗尽线程池。

您应该将复杂的或阻止策略委派给不同的线程，并且您的实施应返回完成合并策略在其他线程中执行的 **CompletionStage**。

## 2.14. 代码执行和数据处理

分布式缓存的一个优点是，您可以利用每个主机的计算资源来更有效地执行大规模数据处理。通过在 Data Grid 上直接执行您的处理逻辑，您可以将工作负载分散到多个 JVM 实例上。您的代码还在数据网格存储数据的同一内存空间中运行，这意味着您可以更快地迭代条目。

在对数据网格部署的性能影响方面，完全取决于您的代码执行。更复杂的处理操作性能更高，因此您应该谨慎规划在数据网格集群中运行任何代码。首先，测试您的代码并在较小的样本数据集中执行多次执行。收集一些指标后，您可以开始识别优化并了解您正在运行的代码的性能影响。

一个无限的考虑因素是，长时间运行的进程可能会对正常的读写操作造成负面影响。因此，您必须持续监控部署并持续评估性能。

**嵌入缓存**
通过使用嵌入式缓存，数据网格提供两个 API，可让您在与数据相同的内存空间中执行代码。

**ClusterExecutor API**

允许您使用 Cache Manager 执行任何操作，包括迭代一个或多个缓存的条目，并基于数据网格节点进行处理。

**CacheStream API**

可让您对集合执行操作，并根据数据进行处理。

如果要在单一节点、一组节点或特定地区中的所有节点上运行操作，那么您应该使用集群执行。如果要运行可保证对整个数据集的正确结果的操作，那么使用分布式流是一个更有效的选项。

**集群执行**

```
ClusterExecutor clusterExecutor = cacheManager.executor();
clusterExecutor.singleNodeSubmission().filterTargets(policy);
```

```
for (int i = 0; i < invocations; ++i) {
   clusterExecutor.submitConsumer((cacheManager) -> {
      TransportConfiguration tc =
      cacheManager.getCacheManagerConfiguration().transport();
      return tc.siteId() + tc.rackId() + tc.machineId();
   }, triConsumer).get(10, TimeUnit.SECONDS);
}
```

CacheStream

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

**其他资源**

- org.infinispan.manager.ClusterExecutor
- org.infinispan.CacheStream

远**程**缓**存**
对于远程缓存，Data Grid 提供了一个 **ServerTask** API，它可让您使用 Data Grid Server 注册自定义 Java 实施，并通过调用 Hot Rod 或使用 Data Grid Command Line Interface (CLI)来编程执行任务。**您可以在一个数据网格服务器实例或集群中所有服务器实例上执行任务。**

## 2.15. 客户端流量

**在调整远程数据网格集群大小时，您需要计算条目的数量和大小，但也需要客户端流量的数量。数据网格需要足够的 RAM 来存储您的数据和足够 CPU，以便及时处理客户端读写请求。**

**有许多影响延迟的不同因素并确定响应时间。例如，键/值对的大小会影响远程缓存的响应时间。影响远程缓存性能的其他因素包括集群接收的每秒请求数、客户端的数量以及写入操作的读操作比例。**

# 第 3 章 OPENSHIFT 的基准测试数据网格

对于在 OpenShift 上运行的 Data Grid 集群，红帽建议使用 Hyperfoil 来测量性能。Hyperfoil 是一种基准测试框架，可为分布式服务提供准确的性能结果。

## 3.1. 基准测试数据网格

设置并配置部署后，开始对数据网格集群进行基准测试，以分析和衡量性能。基准测试显示存在限制，以便您可以调整您的环境并调优您的数据网格配置以获得最佳性能，这意味着实现最低延迟和最高的吞吐量。

值得注意的是，最佳性能是连续过程，而非最终目标。当您的基准测试显示，您的数据网格部署已达到所需的性能水平时，您就无法期望这些结果已得到修复或始终有效。

## 3.2. 安装 HYPERFOIL

通过创建 operator 订阅并下载包含命令行界面(CLI)的 Hyperfoil 发行版，在 Red Hat OpenShift 上设置 Hyperfoilil。

**流程**

1. 通过 OpenShift Web 控制台中的 OperatorHub 创建 Hyperfoil Operator 订阅。

**注意**

Hyperfoil Operator 作为一个社区 Operator 提供。

红帽没有认证 Hyperfoil Operator，它不在与 Data Grid 结合使用时提供支持。安装 Hyperfoil Operator 时，系统会提示您确认有关社区版本的警告，然后再继续。

2. 从 Hyperfoil 发行页面 下载最新的 Hyperfoil 版本。

**其他资源**

- hyperfoil.io
- 在 OpenShift 上安装 Hyperfoil

### 3.3. 创建 HYPERFOIL CONTROLLER

在 Red Hat OpenShift 上实例化 Hyperfoil Controller，以便您可以使用 Hyperfoil 命令行界面(CLI)上传和运行基准测试。

**先决条件**

- 创建 Hyperfoil Operator 订阅。

**流程**

1. 定义 hyperfoil-controller.yaml。

```
$ cat > hyperfoil-controller.yaml<<EOF
apiVersion: hyperfoil.io/v1alpha2
kind: Hyperfoil
metadata:
  name: hyperfoil
spec:
  version: latest
EOF
```

2. 应用 Hyperfoil Controller。

```
$ oc apply -f hyperfoil-controller.yaml
```

3. 检索连接到 Hyperfoil CLI 的路由。

```
$ oc get routes

NAME       HOST/PORT
hyperfoil  hyperfoil-benchmark.apps.example.net
```

### 3.4. 运行 HYPERFOIL 基准

使用 Hyperfoil 运行基准测试，以收集 Data Grid 集群的性能数据。

### 先决条件

- 创建 Hyperfoil Operator 订阅。
- 在 Red Hat OpenShift 上实例化 Hyperfoil Controller。

### 流程

1. 创建基准测试。

```
$ cat > hyperfoil-benchmark.yaml<<EOF
name: hotrod-benchmark
hotrod:
 # Replace <USERNAME>:<PASSWORD> with your Data Grid credentials.
 # Replace <SERVICE_HOSTNAME>:<PORT> with the host name and port for Data
Grid.
 - uri: hotrod://<USERNAME>:<PASSWORD>@<SERVICE_HOSTNAME>:<PORT>
   caches:
    # Replace <CACHE-NAME> with the name of your Data Grid cache.
    - <CACHE-NAME>
agents:
  agent-1:
  agent-2:
  agent-3:
  agent-4:
  agent-5:
phases:
- rampupPut:
    increasingRate:
      duration: 10s
      initialUsersPerSec: 100
      targetUsersPerSec: 200
      maxSessions: 300
      scenario: &put
      - putData:
        - randomInt: cacheKey <- 1 .. 40000
        - randomUUID: cacheValue
        - hotrodRequest:
            # Replace <CACHE-NAME> with the name of your Data Grid cache.
            put: <CACHE-NAME>
            key: key-${cacheKey}
            value: value-${cacheValue}
- rampupGet:
    increasingRate:
      duration: 10s
      initialUsersPerSec: 100
```

```
    targetUsersPerSec: 200
    maxSessions: 300
    scenario: &get
    - getData:
      - randomInt: cacheKey <- 1 .. 40000
      - hotrodRequest:
          # Replace <CACHE-NAME> with the name of your Data Grid cache.
          get: <CACHE-NAME>
          key: key-${cacheKey}
- doPut:
    constantRate:
      startAfter: rampupPut
      duration: 5m
      usersPerSec: 10000
      maxSessions: 11000
      scenario: *put
- doGet:
    constantRate:
      startAfter: rampupGet
      duration: 5m
      usersPerSec: 40000
      maxSessions: 41000
      scenario: *get
EOF
```

2. 在任意浏览器中打开路由，以访问 Hyperfoil CLI。

3. 上传基准测试。

   a. 运行 上传 命令。

   ```
   [hyperfoil]$ upload
   ```

   b. 点 Select benchmark 文件，然后导航到文件系统上的基准测试并上传该文件。

4. 运行基准测试。

```
[hyperfoil]$ run hotrod-benchmark
```

5. 获取基准测试成绩.

```
[hyperfoil]$ stats
```

### 3.5. HYPERFOIL 基准结果

**Hyperfoil 使用 stats 命令以表格式输出基准测试运行的结果。**

```
[hyperfoil]$ stats
Total stats from run <run_id>
PHASE  METRIC  THROUGHPUT  REQUESTS  MEAN  p50  p90  p99  p99.9  p99.99  TIMEOUTS
ERRORS  BLOCKED
```

**表 3.1. 列描述**

| 列 | 描述 | 值 |
|---|---|---|
| **阶段** | 对于每个运行，Hyperfoil 会在两个阶段向 Data Grid 集群发出 **GET** 请求和 **PUT** 请求。 | **doGet** 或 **doPut** |
| **指标** | 在运行的两个阶段，Hyperfoil 会为每个 **GET** 和 **PUT** 请求收集指标。 | **getData** 或 **putData** |
| **吞吐量** | 捕获每秒请求总数。 | Number |
| REQUESTS | 捕获每个运行阶段的操作总数。 | Number |
| MEAN | 捕获完成 **GET** 或 **PUT** 操作的平均时间。 | 毫秒为单位的时间(**ms**) |
| p50 | 记录完成 50% 的请求所需的时间。 | 毫秒为单位的时间(**ms**) |
| p90 | 记录完成 90% 的请求所需的时间。 | 毫秒为单位的时间(**ms**) |
| p99 | 记录完成 99% 的请求所需的时间。 | 毫秒为单位的时间(**ms**) |
| p99.9 | 记录完成请求花费 99.9% 所需的时间。 | 毫秒为单位的时间(**ms**) |
| p99.99 | 记录 99.99% 的请求完成所需的时间。 | 毫秒为单位的时间(**ms**) |
| **超时** | 捕获在每次运行阶段发生的超时总数。 | Number |
| **错误** | 捕获在运行每个阶段发生的错误总数。 | Number |

| 列 | 描述 | 值 |
| --- | --- | --- |
| BLOCKED | 捕获被阻止或无法完成的操作总数。 | Number |