Red Hat build of Keycloak 26.4

高可用性指南

Last Updated: 2025-12-13

Legal Notice

Abstract

本指南由管理员在高可用性部署/架构中配置和使用红帽构建的 Keycloak 26.4 的建议实践。

Table of Contents

第 1 章 高可用性概述

探索红帽构建 Keycloak 高可用性架构的不同。

红帽构建的 Keycloak 可以部署到多个高可用性架构中，允许系统管理员选择最适合其需要的部署类型。在为您的部署确定正确的架构时，易于部署、成本和容错保证是重要的考虑因素。

任何红帽构建的 Keycloak 部署都必须安装到正确配置的 OpenShift 集群上，该集群根据 OpenShift 安装说明 进行了配置。对于跨越多个站点（例如，跨多个可用区的单一集群或多集群）的 OpenShift 集群上的红帽 Keycloak 安装，OpenShift 集群必须遵循 Red Hat OpenShift Container Platform 集群的指南 。

除了运行的软件之外，高可用性架构还涉及许多组件，并且对服务的可用性负责。本指南具有在这样的高可用性架构中使用红帽构建的 Keycloak 的建议实践。

1.1. 构架

本文档描述了部署红帽构建的 Keycloak 的两个高可用性架构： 单集群部署和多集群部署。

1.1.1. 单集群部署

使用单 集群部署 在单个集群中（可选）在带有所需网络延迟和数据库配置的多个可用区或数据中心中部署红帽构建的 Keycloak。

优点

- 没有外部依赖项
- 在单个 OpenShift 集群中部署
- 如果部署到多个可用区或数据中心，则容许可用区或数据中心故障

缺点

- OpenShift 集群是一个单点故障：
  - control-plane 失败可能会影响所有红帽构建的 Keycloak pod

1.1.2. 多集群部署

连接两个红帽构建的 Keycloak 集群，例如，在两个可用区或具有所需网络延迟和数据库配置中的不同 OpenShift 集群中使用 Multi-cluster 部署进行数据库配置。

优点

- 容许可用区失败
- 容许 OpenShift 集群失败
- 桥接不提供透明网络的两个网络
- 需要不同部署时的法规合规性

缺点

- 复杂性：

    - 需要外部负载均衡器
    - 每个站点所需的独立 Data Grid 集群
- Cost:
    - 需要额外的负载均衡器
    - 外部 Data Grid 集群需要额外的计算
    - 必须置备两个 OpenShift control-planes
- 不支持三个或更多可用区

1.1.3. 后续步骤

要了解有关不同高可用性架构和支持的配置的更多信息，请参阅各个章节。

第 2 章 单集群部署

在多个可用区间部署单个 Keycloak 集群（可选）。

2.1. 何时使用单集群设置

Red Hat build of Keycloak 单集群设置适用于以下用例：

- 部署到具有透明网络的基础架构，如单个 OpenShift 集群。
- 需要所有健康的红帽 Keycloak 实例构建来处理用户请求。
- 限制到单个区域（如单个 AWS 区域）
- 允许计划停机进行维护。
- 适合定义的用户和请求数。
- 可以接受定期中断的影响。
- 在具有所需网络延迟和数据库配置的数据中心中部署

2.2. 测试的配置

我们定期使用以下配置测试红帽构建的 Keycloak：

- 在同一区域中的三个 AWS 可用区部署的 OpenShift 集群。
  - 使用 ROSA HCP 在 AWS (ROSA)上置备 Red Hat OpenShift Service 。
  - 每个可用区至少有一个 worker 节点
  - OpenShift 版本 4.17。
- Amazon Aurora PostgreSQL 数据库
  - 在一个可用区中带有主 DB 实例的高可用性，以及同步复制器到其他可用区
  - 版本 17.5
- 在这些配置中支持红帽构建的 Keycloak 可能需要在此测试集中复制问题。

2.3. 配置

- 部署在 OpenShift 集群版本 4.17 或更高版本上的 Red Hat build of Keycloak
  - 对于云设置，如果 OpenShift 支持跨越该环境中的多个可用区，且满足 Keycloak 的延迟要求，则可以将 Pod 调度到同一区域内三个可用区。
  - 对于内部设置，如果 OpenShift 支持跨越该环境中的多个数据中心，则 Pod 可以调度到三个数据中心，并满足红帽构建的 Keycloak 延迟要求。
- 部署需要在红帽构建的 Keycloak 实例之间有 10 毫秒的往返延迟。
- 数据库

- 有关支持的数据库列表，请参阅配置数据库。
- 跨越多个可用区的部署必须使用可以容忍区域故障的数据库，并在副本之间同步复制数据。

上述配置中的任何偏差都未测试，红帽构建的 Keycloak 中的任何问题都可能需要复制到经过测试的环境中，以便获得支持。

了解有关 构建块单集群部署 一章中的每个项目的更多信息。

2.4. 最大负载

我们定期使用以下负载测试红帽构建的 Keycloak：

- 100,000 个用户
- 每秒 300 个请求

如需更多信息，请参阅有关调整 CPU 和内存资源 大小的概念 章节。

2.5. 限制：

即使具有三个可用区的额外冗余性，在以下情况下仍可发生停机：

- 同时发生节点故障
- 推出红帽构建的 Keycloak 升级
- 基础架构失败，如 OpenShift 集群

有关限制的详情，请参阅 单集群部署的概念 章节。

2.6. 后续步骤

不同的章节介绍了必要的概念和构建块。对于每个构建块，蓝图演示了如何部署全功能示例。在准备生产环境设置时，仍然建议使用额外的性能调整和安全强化。

2.7. 单集群部署的概念

通过同步复制了解单集群部署。

本主题描述了单集群设置和预期的行为。它概述了高可用性架构的要求，并描述了优点和权衡。

2.7.1. 何时使用此设置

使用此设置将红帽构建的 Keycloak 部署到 OpenShift 集群。

2.7.2. 单个或多个可用区

红帽构建的 Keycloak 部署的行为和高可用性性能最终由 OpenShift 集群的配置决定。通常，OpenShift 集群部署在单个可用区中，但为了提高容错能力，可以在 多个可用区间部署集群。

Red Hat build of Keycloak Operator 默认定义了以下拓扑分布约束，首选红帽构建的 Keycloak pod 部署到不同的节点上，并尽可能使用不同的可用区：

```
topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: "topology.kubernetes.io/zone"
    whenUnsatisfiable: "ScheduleAnyway"
    labelSelector:
      matchLabels:
        app: "keycloak"
        app.kubernetes.io/managed-by: "keycloak-operator"
        app.kubernetes.io/instance: "keycloak"
        app.kubernetes.io/component: "server"
  - maxSkew: 1
    topologyKey: "kubernetes.io/hostname"
    whenUnsatisfiable: "ScheduleAnyway"
    labelSelector:
      matchLabels:
        app: "keycloak"
        app.kubernetes.io/managed-by: "keycloak-operator"
        app.kubernetes.io/instance: "keycloak"
        app.kubernetes.io/component: "server"
```

重要

为了通过多个可用区配置高可用性，数据库也能够处理区域故障，因为红帽构建的 Keycloak 依赖于底层数据库来保持可用。

2.7.3. 此设置可以存活的故障

在单一区中的单个集群中部署红帽构建的 Keycloak，或跨多个可用区，或带有所需网络延迟和数据库配置的数据中心，并显著改变高可用性特性，因此我们会独立考虑这些架构。

2.7.3.1. 单区

在测试高可用性 单集群 部署过程中，我们观察到以下恢复时间，以了解上述事件：

失败	恢复	RPO[1]	RT[2]
红帽构建的 Keycloak Pod	在集群中运行多个 Keycloak Pod 的红帽构建。如果一个实例失败，一些传入的请求可能会收到错误消息，或者延迟几秒钟。	没有数据丢失	少于 30 秒
OpenShift 节点	在集群中运行多个 Keycloak Pod 的红帽构建。如果主机节点停止，则该节点上的所有 pod 都将失败，一些传入的请求可能会收到错误消息，或者延迟一段时间（以秒为单位）。	没有数据丢失	少于 30 秒

失败	恢复	RPO[1]	RT[2]
红帽构建的 Keycloak 集群连接	如果 OpenShift 节点之间的连接丢失，则数据不能在这些节点上托管的 Keycloak pod 之间发送。传入的请求可能会收到错误消息，或者延迟（以秒为单位）。红帽构建的 Keycloak 最终会从其本地视图中删除无法访问的 pod，并停止向它们发送数据。	没有数据丢失	秒到分钟

表注：

[1] 测试的恢复点目标，假设设置的所有部分都健康。
[2] 最大恢复时间观察到。

2.7.3.2. 多个区

在测试高可用性多集群部署过程中，我们观察到以下恢复时间，以了解上述事件： #multi-cluster-introduction#multi-cluster-supported-configuration

失败	恢复	RPO[1]	RT[2]
数据库节点[3]	如果写入器实例失败，则数据库可以提升相同或其他区域中的读者实例，使其成为新的写入器。	没有数据丢失	分钟的秒数（取决于数据库）
Red Hat build of Keycloak pod	在集群中运行多个 Keycloak 实例的红帽构建。如果一个实例失败，一些传入的请求可能会收到错误消息，或者延迟几秒钟。	没有数据丢失	少于 30 秒
OpenShift 节点	在集群中运行多个 Keycloak pod 的红帽构建。如果主机节点停止，则该节点上的所有 pod 都将失败，一些传入的请求可能会收到错误消息，或者延迟一段时间（以秒为单位）。	没有数据丢失	少于 30 秒

失败	恢复	RPO[1]	RT[2]
可用区失败	如果一个可用区失败，则该区中托管的 Keycloak pod 的所有红帽构建也会失败。至少部署与可用区相同的红帽构建的 Keycloak 副本数量，应该确保不会丢失数据，停机时间最小，因为存在其他 pod 可用于服务请求。	没有数据丢失	秒
连接数据库	如果可用区间的连接丢失，同步复制将失败。有些请求可能会收到错误消息，或延迟几秒钟。根据数据库，可能需要手动操作。	没有数据丢失[3]	分钟的秒数（取决于数据库）
红帽构建的 Keycloak 集群连接	如果 OpenShift 节点之间的连接丢失，则数据不能在这些节点上托管的 Keycloak pod 之间发送。传入的请求可能会收到错误消息，或者延迟（以秒为单位）。红帽构建的 Keycloak 最终会从其本地视图中删除无法访问的 pod，并停止向它们发送数据。	没有数据丢失	秒到分钟

表注：

[1] 测试的恢复点目标，假设设置的所有部分都健康。
[2] 最大恢复时间观察到。
[3] 假设数据库也在多个可用区间复制

2.7.4. 已知限制

1. 红帽构建的 Keycloak 升级过程中的停机时间
   如果 可能出现滚动更新，可以通过启用检查补丁版本来解决此问题。

2. 如果节点故障数大于或等于缓存配置的 **num_owners**，则多次节点故障可能会导致 **authenticationSessions**、**loginFailures** 和 **actionTokens** 缓存丢失条目，默认为 2。

3. 使用带有 **whenUnsatisfiable: ScheduleAnyway** s 的默认 **topologySpreadConstraints** 的部署，如果多个 pod 调度到失败的节点/区，则可能会在节点/区上出现数据。
   用户可以通过使用 **whenUnsatisfiable: DoNotSchedule** 定义 **topologySpreadConstraints** 来缓解这种情况，以确保 pod 始终在区和节点间平均调度。但是，如果无法满足限制，这可能会导致一些红帽构建的 Keycloak 实例不会被部署。

因为 Infinispan 在分发缓存条目时不知道网络拓扑，因此如果缓存数据的所有 **num_owner** 副本都存储在失败的节点/区域中，则数据术语仍可发生在 node/availability-zone 失败时发生。您可以通过为节点和区定义 **requiredDuringSchedulingIgnoredDuringExecution** 将红帽构建的 Keycloak 实例总数限制为可用的节点或可用区的数量。但是，这代价是可扩展性，作为红帽构建的 Keycloak 实例的数量，这些实例可以被置备到 OpenShift 集群中的节点/可用区数量。

有关如何配置自定义反关联性 **拓扑SpreadConstraints** 策略，请参阅 Operator 高级配置详情。

4. Operator 不会在 Pod 中配置站点的名称（请参阅 配置分布式缓存），因为它的值无法通过 Downward API 提供。machine name 选项使用调度 Pod 的节点的 **spec.nodeName** 进行配置。

2.7.5. 后续步骤

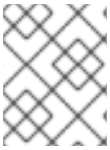
继续阅读 构建块单集群部署 章节中的蓝图，以查找不同构建块的蓝图。

2.8. 构建会阻止单集群部署

了解构建块以及为单集群部署推荐的设置。

设置单集群部署需要以下构建块。

构建块链接到带有示例配置的蓝图。它们按照需要安装的顺序列出。



注意

我们提供这些蓝图来显示最小功能完整示例，为常规安装提供良好的基准性能。您仍然需要根据您的环境以及您的组织的标准和安全性最佳实践进行调整。

2.8.1. 先决条件

- 了解 单集群部署概念 一章中的概念。

2.8.2. 带有低延迟连接的多个可用区

红帽构建的 Keycloak 需要低延迟网络连接，用于由数据库和红帽构建的 Keycloak 集群同步数据。

建议少于 5 ms 的往返延迟，且需要低于 10 ms，以及区间的可靠的网络，以避免出现延迟、吞吐量或连接的问题。

网络延迟和延迟延迟在服务的响应时间中扩展，并可能导致排队的请求、超时和失败请求。网络问题可能会导致停机，直到失败检测隔离有问题的节点。

建议设置： 由同一 AWS 区域内的两个或多个 AWS 可用区组成的 OpenShift 集群。

不考虑： OpenShift 集群分布在相同或不同处的多个区域，因为它会增加延迟和网络故障的可能性。在 AWS 上，同步将数据库复制为带有 Aurora 区域部署的服务。

2.8.3. 数据库

在所有可用区间有一个同步复制的数据库。

蓝图： 在多个可用区中部署 AWS Aurora。

2.8.4. 红帽构建的 Keycloak

红帽构建的 Keycloak 集群部署，带有在可用区间分布的 pod。

蓝图： 使用 Operator 在多个可用区部署红帽构建的 Keycloak。

2.9. 数据库连接池的概念

了解避免资源耗尽和拥塞的概念。

本节适用于对 Red Hat build of Keycloak 配置数据库连接池的注意事项和最佳实践。对于应用此配置，请访问 使用 Operator 在多个可用区间部署红帽 Keycloak 构建。

2.9.1. 概念

创建新数据库连接所需的时间非常昂贵。当请求到达时，创建它们会延迟响应，因此在请求到达前，最好创建它们。它还有助于对在短时间内创建大量连接而造成问题的影响，因为它会减慢系统和块线程的速度。https://en.wikipedia.org/wiki/Cache_stampede关闭连接也会使连接的所有服务器端语句缓存无效。

为了获得最佳性能，初始、最小和最大数据库连接池大小的值都应相等。这可避免在新请求出现成本高时创建新的数据库连接。

只要可能，保持数据库连接打开，从而允许绑定至连接的服务器端语句缓存。如果是 PostgreSQL，若要使用服务器端准备好的声明，需要至少执行（默认）查询（默认为 ）。

如需更多信息，请参阅有关准备的语句的 PostgreSQL 文档。

2.10. 配置线程池的概念

了解避免资源耗尽和拥塞的概念。

本节旨在了解如何为红帽构建的 Keycloak 配置线程池连接池的注意事项和最佳实践。对于应用此配置，请访问 使用 Operator 在多个可用区间部署红帽 Keycloak 构建。

2.10.1. 概念

2.10.1.1. JGroups 通信

JGroups 通信（在单集群设置中用于红帽构建的 Keycloak 节点间的通信）得益于使用虚拟线程，在 OpenJDK 21 中使用虚拟线程（如果至少有两个内核可用于 Keycloak）。这可减少内存用量，并不再需要配置线程池大小。因此，建议使用 OpenJDK 21。

2.10.1.2. Quarkus executor 池

红帽构建的 Keycloak 请求以及阻塞探测由 executor 池处理。根据可用的 CPU 内核，它的默认最大值为 50 个或更多线程。线程根据需要创建，并在不再需要时结束，因此系统将自动扩展和缩减。红帽构建的 Keycloak 允许通过 **http-pool-max-threads** 配置选项配置最大线程池大小。

2.10.1.3. Load Shedding

默认情况下，红帽构建的 Keycloak 将无限地将所有传入的请求排队，即使请求处理停止也是如此。这将在 Pod 中使用额外的内存，可能会耗尽负载均衡器中的资源，请求最终会在客户端一侧超时，而无需了解请求是否已被处理。要限制红帽构建的 Keycloak 中排队请求数，请设置额外的 Quarkus 配置选项。

配置 **http-max-queued-requests**，以指定在超过此队列大小后有效的负载均衡的最大队列长度。假设红帽构建的 Keycloak Pod 进程每秒约 200 个请求，则队列为 1000 会导致最大等待时间为 5 秒。

当此设置处于活跃状态时，超过排队请求数的请求将返回 HTTP 503 错误。Red Hat build of Keycloak 会在日志中记录错误消息。

2.10.1.4. probes

Red Hat build of Keycloak 的存活度探测是非阻止的，以避免在高负载下重启 Pod。

在一些情况下，整个健康探测和就绪度探测可能会检查与数据库的连接，因此它们可能会在高负载下失败。因此，Pod 可能会在高负载下变为未就绪。

2.10.1.5. OS 资源

为了使 Java 创建线程，在 Linux 上运行时，它需要有文件句柄。因此，打开的文件数量（如 **ulimit -n** on Linux）需要为红帽构建的 Keycloak 提供头空间，以增加所需的线程数量。每个线程也会消耗内存，容器内存限值需要设置为允许此的值，或 Pod 将由 OpenShift 终止。

2.11. 调整 CPU 和内存资源大小的概念

了解避免资源耗尽和拥塞的概念。

使用此选项作为调整产品环境的起点。根据您的负载测试，根据需要调整您的环境值。

2.11.1. 性能建议

警告

- 当扩展到更多 Pod （因为额外的开销）并使用多集群设置（因为额外的流量和操作）时，性能会降低。
- 当红帽构建用于长时间的 Keycloak 实例时，增加的缓存大小可以提高性能。这将减少响应时间并减少数据库的 IOPS。仍然需要在实例重启时填充这些缓存，因此不要根据缓存填写后测量的稳定状态设置资源。
- 使用这些值作为起点，并在进入生产环境前执行您自己的负载测试。

Summary：

- 已使用的 CPU 根据以下测试的限制来线性扩展。

建议：

- Pod 的基本内存用量，包括 Realm 数据和 10,000 个缓存会话的缓存 1250 MB RAM。
- 在容器中，Keycloak 为堆内存分配 70% 的内存限值。它还将使用大约 300 MB 的非堆内存。要计算请求的内存，请使用上面的计算。作为内存限制，从上面的值中减去非堆内存，并将结果除以 0.7 划分。
- 对于每秒 15 个基于密码的用户登录，请将 1 个 vCPU 分配给集群（每秒测试最多 300 个）。红帽构建的 Keycloak 使用大多数 CPU 时间哈希为用户提供的密码，并与哈希迭代数量成比例。

- 对于每个 120 个客户端凭证，每秒授予 1 个 vCPU 到集群（每秒测试最多 2000 个）。* 大多数 CPU 时间都会创建新的 TLS 连接，因为每个客户端仅运行一个请求。
- 对于每个 120 每秒刷新令牌请求，1 个 vCPU 到集群（每秒测试了 435 刷新令牌请求）。*
- 保留 150% 额外头条以便 CPU 使用量处理负载高峰。这样可确保节点快速启动，并有足够的容量来处理故障转移任务。当其 Pod 在我们的测试中节流时，红帽构建的 Keycloak 的性能会显著下降。
- 当同时执行具有 2500 多个不同客户端的请求时，并非所有客户端信息都适合红帽构建的 Keycloak 的缓存，分别使用 10000 个条目的标准缓存。因此，数据库可能会成为瓶颈，因为客户端数据经常从数据库中重新加载。要减少数据库使用量，请将 **用户** 缓存大小增加到同时使用的客户端的两次，**域** 缓存大小由并发使用的客户端数的四倍增加。

Red Hat build of Keycloak （默认情况下，在数据库中存储用户会话）需要以下资源在 Aurora PostgreSQL 多AZ 数据库中获得最佳性能：

每个 100 个登录/logout/refresh 请求每秒：

- 1400 写 IOPS 预算.
- 在 0.35 和 0.7 vCPU 之间分配。

vCPU 要求作为范围提供，因为数据库主机上的 CPU 饱和度增加，每个请求的 CPU 使用量会降低，响应时间会增加。数据库上的 CPU 配额较低，可能会导致高峰负载期间的响应时间较慢。如果在峰值负载期间快速响应时间至关重要，请选择较大的 CPU 配额。请参见以下示例。

2.11.1.1. 测量运行红帽构建的 Keycloak 实例的活动

红帽构建的 Keycloak 实例的大小取决于基于密码的用户登录、刷新令牌请求和客户端凭证的实际和预测的数量，如上一节中所述。

要检索这三个密钥输入的运行红帽构建的 Keycloak 实例的实际数量，请使用 Red Hat build of Keycloak 提供的指标：

- 事件类型登录的用户事件指标 **keycloak_user_events_total** 包括基于密码的登录和基于 Cookie 的登录，仍可作为此大小指南的第一个大约输入。
- **要查找红帽构建的 Keycloak 执行的密码验证数量，请使用 metric keycloak_credentials_password_hashing_validations_total。指标还包含一些有关使用的哈希算法和验证结果的标签。以下是可用标签的列表： realm, algorithm, hash_strength,结果.**
- **将用户事件指标 keycloak_user_events_total 用于事件类型 refresh_token 和 client_login 分别用于刷新令牌请求和客户端凭证授权。**

如需更多信息，请参阅使用事件指标和 HTTP 指标 章节监控用户活动。

这些指标对于跟踪用户活动负载中的每日和每周波动至关重要，找出可能表示需要调整系统大小并验证大小计算的新兴趋势。通过系统测量和评估这些用户事件指标，您可以确保您的系统保持正确扩展，并响应用户行为和需求的变化。

2.11.1.2. 计算示例（单集群）

目标大小：

- 45 个登录，每秒退出登录
- 360 客户端凭证每秒授予每秒*
- 360 刷新令牌请求每秒(1:8 ratio for login)*
- 3 个 pod

计算的限制：

- 每个 Pod 请求的 CPU: 3 个 vCPU

  （每秒45 个登录 = 3 个 vCPU，360 客户端凭据授予每秒 = 3 个 vCPU，360 刷新令牌 = 3 个 vCPU。总和最多 9 个 vCPU。当集群中运行的 3 个 Pod 时，每个 Pod 都会请求 3 个 vCPU。
- 每个 Pod 的 CPU 限制： 7.5 vCPU

  （允许额外的 150% CPU 处理峰值、启动和故障转移任务）
- 每个 Pod 请求的内存： 1250 MB

  (1250 MB 基本内存)

- 每个 Pod 的内存限值： 1360 MB

  (1250 MB 预期内存使用减 300 非堆使用，以 0.7 划分)

- Aurora Database 实例： db.t4g.large 或 db.t4g.xlarge，具体取决于高峰负载期间所需的响应时间。

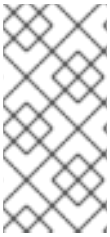
  （每秒45 个登录，每秒 5 个退出一次，360 每秒刷新令牌。这个总和每秒 410 请求。这个预期 DB 使用为 1.4 到 2.8 vCPU，DB 闲置负载为 0.3 vCPU。这表示 2 个 vCPU db.t4g.large 实例或 4 个 vCPU db.t4g.xlarge 实例。如果在高峰使用期间允许响应时间更高，则 2 个 vCPU db.t4g.large 将更具成本效益。在我们的测试中，当 CPU 饱和达到了这个场景的 2 个 vCPU db.t4g.large 实例时，登录的介质和令牌刷新增加了 120 毫秒。为了在高峰使用期间更快的响应时间，请针对这种情况考虑 4 个 vCPU db.t4g.xlarge 实例。）

2.12. 在多个可用区部署 AWS AURORA

在单集群部署中部署一个 AWS Aurora 作为数据库构建块。

本节论述了如何在多个可用区间部署 PostgreSQL 实例的 Aurora 区域部署，以便在给定 AWS 区域中容忍一个或多个可用区失败。

此部署旨在与 单集群部署 一章中介绍的设置一起使用。将此部署与构建块 单集群部署一章中介绍的其他构建块 一起使用。



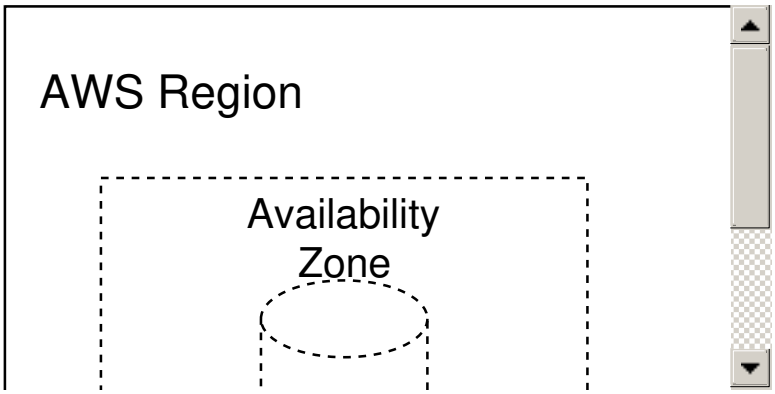
注意

我们提供这些蓝图来显示最小功能完整示例，为常规安装提供良好的基准性能。您仍然需要根据您的环境以及您的组织的标准和安全性最佳实践进行调整。

2.12.1. 架构

Aurora 数据库集群由多个 Aurora 数据库实例组成，一个实例指定为主写入器，所有其他实例都指定为备份读取器。为确保在可用区失败时高可用性，Aurora 允许在单个 AWS 区域中的多个区域部署数据库实例。如果在托管 Primary 数据库实例的可用区失败时，Aurora 会自动修复自身，并将读取器实例从非失败的可用区提升为新的写入器实例。

图 2.1. Aurora 多可用区部署



有关 Aurora 数据库提供的语义的详情，请参阅 AWS Aurora 文档。

本文档遵循 AWS 最佳实践并创建不向互联网公开的私有 Aurora 数据库。要从 ROSA 集群访问数据库，请在 数据库和 ROSA 集群之间建立对等连接。

2.12.2. 流程

以下流程包含两个部分：

- 在 eu-west-1 中使用名称 "keycloak-aurora" 创建 Aurora Multi-AZ 数据库集群。
- 在 ROSA 集群和 Aurora VPC 之间创建对等连接，以允许 ROSA 集群上部署的应用程序与数据库建立连接。

2.12.2.1. 创建 Aurora 数据库集群

1. 为 Aurora 集群创建一个 VPC

命令：

```
aws ec2 create-vpc \
  --cidr-block 192.168.0.0/16 \
  --tag-specifications "ResourceType=vpc, Tags=[{Key=AuroraCluster,Value=keycloak-aurora}]" \ 1
  --region eu-west-1
```

1

我们使用 Aurora 集群的名称添加可选标签，以便我们可以轻松地检索 VPC。

输出：

```
{
    "Vpc": {
        "CidrBlock": "192.168.0.0/16",
        "DhcpOptionsId": "dopt-0bae7798158bc344f",
        "State": "pending",
        "VpcId": "vpc-0b40bd7c59dbe4277",
        "OwnerId": "606671647913",
        "InstanceTenancy": "default",
        "Ipv6CidrBlockAssociationSet": [],
        "CidrBlockAssociationSet": [
            {
                "AssociationId": "vpc-cidr-assoc-09a02a83059ba5ab6",
                "CidrBlock": "192.168.0.0/16",
                "CidrBlockState": {
                    "State": "associated"
                }
            }
        ],
        "IsDefault": false
    }
}
```

2. 使用新创建的 VPC 的 VpcId 为 Aurora 部署到的每个可用区创建一个子网。

注意

为每个可用区指定的 cidr-block 范围不得互相重叠。

a. 区域 A

命令：

```
aws ec2 create-subnet \
  --availability-zone "eu-west-1a" \
  --vpc-id vpc-0b40bd7c59dbe4277 \
  --cidr-block 192.168.0.0/19 \
  --region eu-west-1
```

输出：

```
{
    "Subnet": {
        "AvailabilityZone": "eu-west-1a",
        "AvailabilityZoneId": "euw1-az3",
        "AvailableIpAddressCount": 8187,
        "CidrBlock": "192.168.0.0/19",
        "DefaultForAz": false,
        "MapPublicIpOnLaunch": false,
        "State": "available",
        "SubnetId": "subnet-0d491a1a798aa878d",
        "VpcId": "vpc-0b40bd7c59dbe4277",
        "OwnerId": "606671647913",
        "AssignIpv6AddressOnCreation": false,
        "Ipv6CidrBlockAssociationSet": [],
        "SubnetArn": "arn:aws:ec2:eu-west-1:606671647913:subnet/subnet-
0d491a1a798aa878d",
        "EnableDns64": false,
        "Ipv6Native": false,
        "PrivateDnsNameOptionsOnLaunch": {
            "HostnameType": "ip-name",
            "EnableResourceNameDnsARecord": false,
            "EnableResourceNameDnsAAAARecord": false
        }
    }
}
```

b.

zone B

命令：

```
aws ec2 create-subnet \
  --availability-zone "eu-west-1b" \
  --vpc-id vpc-0b40bd7c59dbe4277 \
  --cidr-block 192.168.32.0/19 \
  --region eu-west-1
```

输出：

```
{
    "Subnet": {
        "AvailabilityZone": "eu-west-1b",
        "AvailabilityZoneId": "euw1-az1",
        "AvailableIpAddressCount": 8187,
        "CidrBlock": "192.168.32.0/19",
        "DefaultForAz": false,
        "MapPublicIpOnLaunch": false,
        "State": "available",
        "SubnetId": "subnet-057181b1e3728530e",
        "VpcId": "vpc-0b40bd7c59dbe4277",
        "OwnerId": "606671647913",
        "AssignIpv6AddressOnCreation": false,
        "Ipv6CidrBlockAssociationSet": [],
        "SubnetArn": "arn:aws:ec2:eu-west-1:606671647913:subnet/subnet-
057181b1e3728530e",
        "EnableDns64": false,
        "Ipv6Native": false,
        "PrivateDnsNameOptionsOnLaunch": {
            "HostnameType": "ip-name",
            "EnableResourceNameDnsARecord": false,
            "EnableResourceNameDnsAAAARecord": false
        }
    }
}
```

3. 获取 Aurora VPC 路由表 ID

命令：

```
aws ec2 describe-route-tables \
  --filters Name=vpc-id,Values=vpc-0b40bd7c59dbe4277 \
  --region eu-west-1
```

输出：


```
{
    "RouteTables": [
        {
            "Associations": [
                {
                    "Main": true,
                    "RouteTableAssociationId": "rtbassoc-02dfa06f4c7b4f99a",
                    "RouteTableId": "rtb-04a644ad3cd7de351",
                    "AssociationState": {
                        "State": "associated"
                    }
                }
            ],
            "PropagatingVgws": [],
            "RouteTableId": "rtb-04a644ad3cd7de351",
            "Routes": [
                {
                    "DestinationCidrBlock": "192.168.0.0/16",
                    "GatewayId": "local",
                    "Origin": "CreateRouteTable",
                    "State": "active"
                }
            ],
            "Tags": [],
            "VpcId": "vpc-0b40bd7c59dbe4277",
            "OwnerId": "606671647913"
        }
    ]
}
```


4. 关联 Aurora VPC 路由表每个可用区的子网

a.

区域 A

命令：

```
aws ec2 associate-route-table \
  --route-table-id rtb-04a644ad3cd7de351 \
  --subnet-id subnet-0d491a1a798aa878d \
  --region eu-west-1
```

b.

zone B

命令：

```
aws ec2 associate-route-table \
  --route-table-id rtb-04a644ad3cd7de351 \
  --subnet-id subnet-057181b1e3728530e \
  --region eu-west-1
```

5.

创建 **Aurora** 子网组

命令：

```
aws rds create-db-subnet-group \
  --db-subnet-group-name keycloak-aurora-subnet-group \
  --db-subnet-group-description "Aurora DB Subnet Group" \
  --subnet-ids subnet-0d491a1a798aa878d subnet-057181b1e3728530e \
  --region eu-west-1
```

6. 创建 Aurora 安全组

命令：

```
aws ec2 create-security-group \
  --group-name keycloak-aurora-security-group \
  --description "Aurora DB Security Group" \
  --vpc-id vpc-0b40bd7c59dbe4277 \
  --region eu-west-1
```

输出：

```
{
    "GroupId": "sg-0d746cc8ad8d2e63b"
}
```

7. 创建 Aurora DB 集群

命令：

```
aws rds create-db-cluster \
    --db-cluster-identifier keycloak-aurora \
    --database-name keycloak \
    --engine aurora-postgresql \
    --engine-version ${properties["aurora-postgresql.version"]} \
    --master-username keycloak \
    --master-user-password secret99 \
    --vpc-security-group-ids sg-0d746cc8ad8d2e63b \
    --db-subnet-group-name keycloak-aurora-subnet-group \
    --region eu-west-1
```

注意

您应该替换 **--master-username** 和- **-master-user-password** 值。在配置红帽构建 **Keycloak** 数据库凭证时，必须使用此处指定的值。

输出：


```
{
   "DBCluster": {
      "AllocatedStorage": 1,
      "AvailabilityZones": [
         "eu-west-1b",
         "eu-west-1c",
         "eu-west-1a"
      ],
      "BackupRetentionPeriod": 1,
      "DatabaseName": "keycloak",
      "DBClusterIdentifier": "keycloak-aurora",
      "DBClusterParameterGroup": "default.aurora-postgresql15",
      "DBSubnetGroup": "keycloak-aurora-subnet-group",
      "Status": "creating",
      "Endpoint": "keycloak-aurora.cluster-clhthfqe0h8p.eu-west-
1.rds.amazonaws.com",
      "ReaderEndpoint": "keycloak-aurora.cluster-ro-clhthfqe0h8p.eu-west-
1.rds.amazonaws.com",
      "MultiAZ": false,
      "Engine": "aurora-postgresql",
      "EngineVersion": "15.5",
      "Port": 5432,
      "MasterUsername": "keycloak",
      "PreferredBackupWindow": "02:21-02:51",
      "PreferredMaintenanceWindow": "fri:03:34-fri:04:04",
      "ReadReplicaIdentifiers": [],
      "DBClusterMembers": [],
      "VpcSecurityGroups": [
         {
            "VpcSecurityGroupId": "sg-0d746cc8ad8d2e63b",
            "Status": "active"
         }
      ],
      "HostedZoneId": "Z29XKXDKYMONMX",
      "StorageEncrypted": false,
      "DbClusterResourceId": "cluster-IBWXUWQYM3MS5BH557ZJ6ZQU4I",
      "DBClusterArn": "arn:aws:rds:eu-west-1:606671647913:cluster:keycloak-aurora",
      "AssociatedRoles": [],
      "IAMDatabaseAuthenticationEnabled": false,
```

```
        "ClusterCreateTime": "2023-11-01T10:40:45.964000+00:00",
        "EngineMode": "provisioned",
        "DeletionProtection": false,
        "HttpEndpointEnabled": false,
        "CopyTagsToSnapshot": false,
        "CrossAccountClone": false,
        "DomainMemberships": [],
        "TagList": [],
        "AutoMinorVersionUpgrade": true,
        "NetworkType": "IPV4"
    }
}
```

8. **创建 Aurora DB 实例**

a. **Create Zone A Writer 实例**

命令：

```
aws rds create-db-instance \
  --no-auto-minor-version-upgrade \
  --db-cluster-identifier keycloak-aurora \
  --db-instance-identifier "keycloak-aurora-instance-1" \
  --db-instance-class db.t4g.large \
  --engine aurora-postgresql \
  --region eu-west-1
```

b. **Create Zone B Reader 实例**

命令：

```
aws rds create-db-instance \
  --no-auto-minor-version-upgrade \
  --db-cluster-identifier keycloak-aurora \
  --db-instance-identifier "keycloak-aurora-instance-2" \
```

```
    --db-instance-class db.t4g.large \
    --engine aurora-postgresql \
    --region eu-west-1
```

9. 等待所有 Writer 和 Reader 实例就绪

命令：

```
aws rds wait db-instance-available --db-instance-identifier keycloak-aurora-instance-1 --region eu-west-1
aws rds wait db-instance-available --db-instance-identifier keycloak-aurora-instance-2 --region eu-west-1
```

10. 获取用于 Keycloak 的 Writer 端点 URL

命令：

```
aws rds describe-db-clusters \
  --db-cluster-identifier keycloak-aurora \
  --query 'DBClusters[*].Endpoint' \
  --region eu-west-1 \
  --output text
```

输出：

```
[
    "keycloak-aurora.cluster-clhthfqe0h8p.eu-west-1.rds.amazonaws.com"
]
```

2.12.2.2. 使用 ROSA 集群建立对等连接

1. 检索 Aurora VPC

命令：

```
aws ec2 describe-vpcs \
  --filters "Name=tag:AuroraCluster,Values=keycloak-aurora" \
  --query 'Vpcs[*].VpcId' \
  --region eu-west-1 \
  --output text
```

输出：

```
vpc-0b40bd7c59dbe4277
```

2. 检索 ROSA 集群 VPC

   a. 使用 oc登录到 ROSA 集群

   b. 检索 ROSA VPC

命令：

```
NODE=$(oc get nodes --selector=node-role.kubernetes.io/worker -o
```

```
jsonpath='{.items[0].metadata.name}')
aws ec2 describe-instances \
  --filters "Name=private-dns-name,Values=${NODE}" \
  --query 'Reservations[0].Instances[0].VpcId' \
  --region eu-west-1 \
  --output text
```

输出：

```
vpc-0b721449398429559
```

3. **创建对等连接**

命令：

```
aws ec2 create-vpc-peering-connection \
  --vpc-id vpc-0b721449398429559 \ 1
  --peer-vpc-id vpc-0b40bd7c59dbe4277 \ 2
  --peer-region eu-west-1 \
  --region eu-west-1
```

1. **ROSA 集群 VPC**

2. **Aurora VPC**

输出：

```
{
    "VpcPeeringConnection": {
        "AccepterVpcInfo": {
            "OwnerId": "606671647913",
            "VpcId": "vpc-0b40bd7c59dbe4277",
            "Region": "eu-west-1"
        },
        "ExpirationTime": "2023-11-08T13:26:30+00:00",
        "RequesterVpcInfo": {
            "CidrBlock": "10.0.17.0/24",
            "CidrBlockSet": [
                {
                    "CidrBlock": "10.0.17.0/24"
                }
            ],
            "OwnerId": "606671647913",
            "PeeringOptions": {
                "AllowDnsResolutionFromRemoteVpc": false,
                "AllowEgressFromLocalClassicLinkToRemoteVpc": false,
                "AllowEgressFromLocalVpcToRemoteClassicLink": false
            },
            "VpcId": "vpc-0b721449398429559",
            "Region": "eu-west-1"
        },
        "Status": {
            "Code": "initiating-request",
            "Message": "Initiating Request to 606671647913"
        },
        "Tags": [],
        "VpcPeeringConnectionId": "pcx-0cb23d66dea3dca9f"
    }
}
```


4. 等待 Peering 连接存在

命令：

```
aws ec2 wait vpc-peering-connection-exists --vpc-peering-connection-ids pcx-0cb23d66dea3dca9f
```

5. 接受对等连接

命令：

```
aws ec2 accept-vpc-peering-connection \
  --vpc-peering-connection-id pcx-0cb23d66dea3dca9f \
  --region eu-west-1
```

输出：


```
{
    "VpcPeeringConnection": {
        "AccepterVpcInfo": {
            "CidrBlock": "192.168.0.0/16",
            "CidrBlockSet": [
                {
                    "CidrBlock": "192.168.0.0/16"
                }
            ],
            "OwnerId": "606671647913",
            "PeeringOptions": {
                "AllowDnsResolutionFromRemoteVpc": false,
                "AllowEgressFromLocalClassicLinkToRemoteVpc": false,
                "AllowEgressFromLocalVpcToRemoteClassicLink": false
            },
            "VpcId": "vpc-0b40bd7c59dbe4277",
            "Region": "eu-west-1"
        },
        "RequesterVpcInfo": {
            "CidrBlock": "10.0.17.0/24",
            "CidrBlockSet": [
                {
                    "CidrBlock": "10.0.17.0/24"
                }
            ],
            "OwnerId": "606671647913",
            "PeeringOptions": {
                "AllowDnsResolutionFromRemoteVpc": false,
                "AllowEgressFromLocalClassicLinkToRemoteVpc": false,
                "AllowEgressFromLocalVpcToRemoteClassicLink": false
            },
            "VpcId": "vpc-0b721449398429559",
            "Region": "eu-west-1"
```

```
        },
        "Status": {
            "Code": "provisioning",
            "Message": "Provisioning"
        },
        "Tags": [],
        "VpcPeeringConnectionId": "pcx-0cb23d66dea3dca9f"
    }
}
```

6. 更新 ROSA 集群 VPC 路由表

命令：

```
ROSA_PUBLIC_ROUTE_TABLE_ID=$(aws ec2 describe-route-tables \
  --filters "Name=vpc-id,Values=vpc-0b721449398429559"
"Name=association.main,Values=true" \ 1
  --query "RouteTables[*].RouteTableId" \
  --output text \
  --region eu-west-1
)
aws ec2 create-route \
  --route-table-id ${ROSA_PUBLIC_ROUTE_TABLE_ID} \
  --destination-cidr-block 192.168.0.0/16 \ 2
  --vpc-peering-connection-id pcx-0cb23d66dea3dca9f \
  --region eu-west-1
```

1. ROSA 集群 VPC

2. 这必须与创建 Aurora VPC 时使用的 cidr-block 相同

7. 更新 Aurora 安全组

命令：

```
AURORA_SECURITY_GROUP_ID=$(aws ec2 describe-security-groups \
  --filters "Name=group-name,Values=keycloak-aurora-security-group" \
  --query "SecurityGroups[*].GroupId" \
  --region eu-west-1 \
  --output text
)
aws ec2 authorize-security-group-ingress \
  --group-id ${AURORA_SECURITY_GROUP_ID} \
  --protocol tcp \
  --port 5432 \
  --cidr 10.0.17.0/24 \ 1
  --region eu-west-1
```

1 ROSA 集群的 "machine_cidr"

输出：

```
{
    "Return": true,
    "SecurityGroupRules": [
        {
            "SecurityGroupRuleId": "sgr-0785d2f04b9cec3f5",
            "GroupId": "sg-0d746cc8ad8d2e63b",
            "GroupOwnerId": "606671647913",
            "IsEgress": false,
            "IpProtocol": "tcp",
            "FromPort": 5432,
            "ToPort": 5432,
            "CidrIpv4": "10.0.17.0/24"
        }
    ]
}
```

2.12.3. 验证连接

验证 ROSA 集群和 Aurora DB 集群之间是否可以连接的最简单方法是在 OpenShift 集群上部署 psql，并尝试连接到写器端点。

以下命令在 default 命名空间中创建 pod，并在可能的情况下建立与 Aurora 集群的 psql 连接。退出 pod shell 后，会删除 pod。

```
USER=keycloak 1
PASSWORD=secret99 2
DATABASE=keycloak 3
HOST=$(aws rds describe-db-clusters \
  --db-cluster-identifier keycloak-aurora \ 4
  --query 'DBClusters[*].Endpoint' \
  --region eu-west-1 \
  --output text
)
oc run -i --tty --rm debug --image=postgres:15 --restart=Never -- psql
postgresql://${USER}:${PASSWORD}@${HOST}/${DATABASE}
```

1 Aurora DB 用户，这可以与创建 DB 时使用的 -master-username 相同。

2 Aurora DB 用户密码，与创建 DB 时使用的 master-user-password 相同。

3 Aurora DB 的名称，如 --database-name。

4 Aurora DB 集群的名称。

2.12.4. 使用红帽构建的 Keycloak 连接 Aurora 数据库

现在，Aurora 数据库已经建立并链接到所有 ROSA 集群，以下是相关的红帽构建的 Keycloak CR 选项，用于将 Aurora 数据库与红帽构建的 Keycloak 连接。使用 Operator 在多个可用区部署红帽构建的 Keycloak 中，需要这些更改。JDBC url 被配置为使用 Aurora 数据库写入器端点。

1. 将 spec.db.url 更新为 jdbc:aws-wrapper:postgresql://$HOST:5432/keycloak，其中

$HOST 是 Aurora writer 端点 URL。

2. 确保 spec.db.usernameSecret 和 spec.db.passwordSecret 引用的 Secret 包含创建 Aurora 时定义的用户名和密码。

2.12.5. 后续步骤

在成功部署 Aurora 数据库后，使用 Operator 在多个可用区上部署红帽构建的 Keycloak

2.13. 使用 OPERATOR 在多个可用区部署红帽构建的 KEYCLOAK

使用红帽构建的 Keycloak Operator 作为构建块，部署红帽构建的 Keycloak 以实现高可用性。

本章论述了 OpenShift 的 Keycloak 配置的高级红帽构建，这些配置已测试，并将恢复可用性区域失败。

这些说明 适用于单集群部署 一章中介绍的设置。将其与构建块 单集群部署一章中介绍的其他构建块 一起使用。

2.13.1. 先决条件

- 在多个可用区部署的 OpenShift 集群，为每个配置了一个 worker-pool。
- 了解使用 红帽构建的 Keycloak Operator 进行红帽构建的 Keycloak 部署的基本红帽构建。
- 使用在 多个可用区中的 Deploying AWS Aurora 部署的 AWS Aurora 数据库。

2.13.2. 流程

1. 确定使用 概念调整 CPU 和内存资源章节的部署大小。
2. 按照 Red Hat build of Keycloak Operator 安装中的内容 安装 Red Hat build of Keycloak

Operator。

3. 请注意，以下配置文件包含与 多个可用区部署 AWS Aurora 中连接到 Aurora数据库相关的选项

4. 构建自定义红帽构建的 Keycloak 镜像，准备与 Amazon Aurora PostgreSQL 数据库一起使用。

5. 使用在第一步中计算的资源请求和限值，使用以下值部署红帽 Keycloak CR：

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  labels:
    app: keycloak
  name: keycloak
  namespace: keycloak
spec:
  hostname:
    hostname: <KEYCLOAK_URL_HERE>
  resources:
    requests:
      cpu: "2"
      memory: "1250M"
    limits:
      cpu: "6"
      memory: "2250M"
  db:
    vendor: postgres
    url: jdbc:aws-wrapper:postgresql://<AWS_AURORA_URL_HERE>:5432/keycloak
    poolMinSize: 30 (1)
    poolInitialSize: 30
    poolMaxSize: 30
    usernameSecret:
      name: keycloak-db-secret
      key: username
    passwordSecret:
      name: keycloak-db-secret
      key: password
  image: <KEYCLOAK_IMAGE_HERE> (2)
  startOptimized: false (3)
  additionalOptions:
    - name: log-console-output
      value: json
    - name: metrics-enabled (4)
      value: 'true'
    - name: event-metrics-user-enabled
      value: 'true'
```

```
  - name: db-driver
    value: software.amazon.jdbc.Driver
http:
  tlsSecret: keycloak-tls-secret
instances: 3
```

1

数据库连接池 initial、max 和 min 大小应当相同，以允许数据库的声明缓存。调整这个数字以满足您的系统需求。由于大多数请求不会因为红帽构建的 Keycloak 嵌入式缓存而影响数据库，因此这个更改可以每秒负责几百个请求。详情请参阅数据库连接池的概念 章节。

2 3

指定自定义红帽构建的 Keycloak 镜像的 URL。如果您的镜像经过优化，请将 startOptimized 标志设置为 true。

4

为了能够在负载下分析系统，启用指标端点。

2.13.3. 验证部署

确认红帽构建的 Keycloak 部署已就绪。

```
oc wait --for=condition=Ready keycloaks.k8s.keycloak.org/keycloak
oc wait --for=condition=RollingUpdate=False keycloaks.k8s.keycloak.org/keycloak
```

2.13.4. 可选： Load shedding

要启用负载均衡，请限制排队的请求数。

使用最大排队的 http 请求加载她

```
spec:
  additionalOptions:
    - name: http-max-queued-requests
      value: "1000"
```

所有超过的请求都使用 HTTP 503 提供。

您可以考虑进一步限制 http-pool-max-threads 的值，因为在达到请求的 CPU 限制后，OpenShift 会导致 OpenShift 节流。

有关详细信息，请参阅有关配置线程池 的概念 一章。

2.13.5. 可选：禁用粘性会话

当在 OpenShift 上运行以及由 Red Hat build of Keycloak Operator 提供的默认 passthrough Ingress 设置时，HAProxy 所做的负载均衡会根据源的 IP 地址使用粘性会话来实现。在运行负载测试时，或者在 HAProxy 前面运行反向代理时，您可能需要禁用此设置以避免在单个红帽构建的 Keycloak Pod 上接收所有请求。

在红帽构建的 Keycloak 自定义资源的 spec 下添加以下补充配置，以禁用粘性会话。

```
spec:
  ingress:
    enabled: true
    annotations:
      # When running load tests, disable sticky sessions on the OpenShift HAProxy router
      # to avoid receiving all requests on a single Red Hat build of Keycloak Pod.
      haproxy.router.openshift.io/balance: roundrobin
      haproxy.router.openshift.io/disable_cookies: 'true'
```

第 3 章 多集群部署

在独立的 OpenShift 集群中连接多个 Keycloak 部署的红帽构建。

红帽构建的 Keycloak 支持部署由多个红帽构建的 Keycloak 实例组成，这些实例使用嵌入式 Infinispan 缓存互相连接。负载均衡器可以在这些实例之间平均分配负载。这些设置适用于透明网络，请参阅 单集群部署 以了解更多详细信息。

多集群设置添加了额外的组件，允许桥接非转换网络，以便提供额外的高可用性。

3.1. 何时使用多集群设置

红帽构建的 Keycloak 的多集群部署功能针对以下用例：

- 受单个 AWS 区域的限制。
- 允许计划停机进行维护。
- 适合定义的用户和请求数。
- 可以接受定期中断的影响。
- 在具有所需网络延迟和数据库配置的数据中心中部署

3.2. 测试的配置

我们定期使用以下配置测试红帽构建的 Keycloak：

- 同一 AWS 区域中的两个 OpenShift 单AZ 集群

- 使用 ROSA HCP 在 AWS (ROSA)上置备 Red Hat OpenShift Service。
- 所有 worker 节点都位于单个可用区中。
- OpenShift 版本 4.17。

- Amazon Aurora PostgreSQL 数据库
  - 在一个可用区中带有主 DB 实例的高可用性，以及第二个可用区中的同步复制器
  - 版本 17.5
- AWS Global Accelerator，将流量发送到两个 ROSA 集群
- ROSA Prometheus 和 Alert Manager 触发的 AWS Lambda 来自动故障切换

3.3. 支持的配置

支持以下配置：

- 同一 AWS 区域中的两个 OpenShift 单AZ 集群
  - 使用 Red Hat OpenShift Service on AWS (ROSA)置备，可以是 ROSA HCP 或 ROSA 经典。
  - 每个 OpenShift 集群都在单个可用区中都有其所有 worker。
  - OpenShift 版本 4.17 （或更新版本）。

- Amazon Aurora PostgreSQL 数据库
  - 在一个可用区中带有主 DB 实例的高可用性，以及第二个可用区中的同步复制器
  - 版本 17.5
- AWS Global Accelerator，将流量发送到两个 ROSA 集群
- AWS Lambda 用于自动故障切换

上述配置中的任何偏差都未测试，红帽构建的 Keycloak 中的任何问题都可能需要复制到经过测试的环境中，以便获得支持。

了解有关 构建块多集群部署 一章中的每个项目的更多信息。

3.4. 最大负载

我们定期使用以下负载测试红帽构建的 Keycloak：

- 100,000 个用户
- 每秒 300 个请求

如需更多信息，请参阅有关调整 CPU 和内存资源 大小的概念 章节。

3.5. 限制：

- 在升级期间，Red Hat build of Keycloak 或 Data Grid 的升级过程中都需要离线。

- 在某些故障情况下，可能最多需要 5 分钟。
- 在某些故障情境后，可能需要手动干预来恢复冗余，方法是使失败的站点恢复在线。
- 在某些切换场景中，可能会有最多 5 分钟的停机时间。

有关限制的详情，请参阅多集群部署的概念 章节。

3.6. 后续步骤

不同的章节介绍了必要的概念和构建块。对于每个构建块，蓝图演示了如何设置全功能示例。在准备生产环境设置时，仍然建议使用额外的性能调整和安全强化。

3.7. 多集群部署的概念

通过同步复制了解多集群部署。

本主题描述了高度可用的多集群设置，以及预期的行为。它概述了高可用性架构的要求，并描述了优点和权衡。

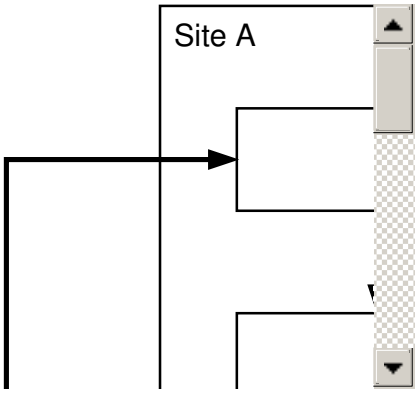
3.7.1. 何时使用此设置

使用此设置来提供红帽构建的 Keycloak 部署，以容许 OpenShift 集群故障，从而减少停机时间的可能性。

3.7.2. 部署、数据存储和缓存

在不同站点中运行的 Keycloak 部署的两个独立的红帽构建与低延迟网络连接连接。用户、域、客户端、会话和其他实体存储在同步在两个站点之间复制的数据库中。数据也缓存在红帽构建的 Keycloak Infinispan 缓存中，作为本地缓存。当数据在一个红帽构建的 Keycloak 实例中更改时，该数据会在数据库中更新，并使用 工作 缓存将无效消息发送到其他站点。

在以下段落和图表中，部署数据网格的参考适用于外部 Data Grid。



3.7.3. 数据丢失的原因

虽然此设置旨在实现高可用性，但以下情况仍然可以导致服务或数据丢失：

- 红帽构建的 Keycloak 站点失败可能会导致请求在故障和 loadbalancer 检测它之间失败，因为请求可能仍然路由到失败的站点。
- 当站点之间的通信出现问题后，需要手动步骤来重新同步降级设置。
- 如果其他组件失败，降级设置可能会导致服务或数据丢失。监控需要检测降级设置。

3.7.4. 此设置可以存活的故障

失败	恢复	RPO[1]	RT[2]
数据库节点	如果写入器实例失败，数据库可以提升相同或其他站点中的读取器实例，使其成为新的写入器。	没有数据丢失	分钟的秒数（取决于数据库）
Red Hat build of Keycloak 节点	多个红帽构建的 Keycloak 实例在每个站点上运行。如果一个实例失败，一些传入的请求可能会收到错误消息，或者延迟几秒钟。	没有数据丢失	少于 30 秒

失败	恢复	RPO[1]	RT[2]
Data Grid 节点	在每个站点中运行多个 Data Grid 实例。如果一个实例失败，则其他节点需要几秒钟才能注意到更改。实体至少存储在两个 Data Grid 节点上，因此单个节点故障不会造成数据丢失。	没有数据丢失	少于 30 秒
Data Grid 集群故障	如果 Data Grid 集群在其中一个站点中失败，红帽构建的 Keycloak 将无法与该站点的外部 Data Grid 通信，并且红帽构建的 Keycloak 服务将不可用。loadbalancer 将检测到情况，因为 **/lb-check** 返回错误，并将所有流量定向到其他站点。 设置会降级，直到 Data Grid 集群恢复且数据被重新同步。	没有数据丢失[3]	秒数（取决于负载均衡器设置）
连接数据网格	如果两个站点之间的连接丢失，则无法将数据发送到其他站点。传入的请求可能会收到错误消息，或者延迟（以秒为单位）。Data Grid 将标记其他站点离线，并将停止发送数据。其中一个站点需要在负载均衡器中离线，直到恢复连接并在两个站点间重新同步数据。在蓝图中，我们展示了如何进行自动化。	没有数据丢失[3]	秒数（取决于负载均衡器设置）
连接数据库	如果两个站点之间的连接丢失，同步复制将失败。有些请求可能会收到错误消息，或延迟几秒钟。根据数据库，可能需要手动操作。	没有数据丢失[3]	分钟的秒数（取决于数据库）

失败	恢复	RPO[1]	RT[2]
站点失败	如果没有红帽构建的 Keycloak 节点可用，则 loadbalancer 将检测到中断，并将流量重定向到其他站点。有些请求可能会收到错误消息，直到 loadbalancer 检测到失败。	没有数据丢失[3]	少于两分钟

表注：

[1] 测试的恢复点目标，假设设置的所有部分都健康。

[2] Maximum Recovery Time observed.

[3] 3 Manual operations to restore the degraded setup。

语句 "No data loss" 依赖于之前失败中的设置，包括完成任何待处理的手动操作来重新同步站点之间的状态。

3.7.5. 已知限制

站点故障

成功故障转移需要设置不会从以前的故障中降级。所有手动操作（如在上一个故障后重新同步）都必须完成，以防止数据丢失。使用监控确保及时检测和处理降级。

不同步站点

当同步 Data Grid 请求失败时，站点可能会不同步。这种情况目前很难监控，可能需要完全重新同步 Data Grid 才能恢复。监控站点中的缓存条目数量，以及 Red Hat build of Keycloak 日志文件时，可以在需要重新同步时显示。

手动操作

在站点间重新同步 Data Grid 状态的手动操作将发布完整状态转移，这将给系统带来压力。

两个站点限制

此设置只适用于两个站点。每个额外站点都会增加整体延迟，因为数据需要同步写入每个站点。此外，网络故障的可能性也会增加。因此，我们不支持超过两个站点，因为我们认为部署具有低端稳定性和性能。

3.7.6. 问题和答案

为什么要进行同步数据库复制？

同步复制的数据库可确保在站点失败后，在一个站点写入的数据始终会在其他站点中可用，且没有数据丢失。它还确保下一个请求不会返回过时的数据，与其提供服务的站点无关。

为什么要进行同步的 Data Grid 复制？

同步复制的 Data Grid 可确保在站点出现故障时，缓存的数据始终会在其他站点上可用，且不会丢失数据。它还确保下一个请求不会返回过时的数据，与其提供服务的站点无关。

为什么在站点间需要低延迟网络？

同步复制会延迟对调用者的响应，直到数据在其他站点收到为止。对于同步数据库复制和同步数据网格复制，因此当数据更新时，每个请求在站点间可能有多个交互，这会放大延迟。

同步集群是否低于异步集群？

异步设置会安全地处理站点之间的网络故障，而同步设置会延迟请求，并将抛出错误到调用者，异步设置会延迟到 Data Grid 或另一站点上的数据库。但是，因为两个站点永远不会完全更新，所以这个设置可能会导致失败过程中数据丢失。这包括：

- 丢失了导致用户无法使用旧密码登录的更改，因为在使用异步数据库时数据库更改不会复制到其他站点。
- 无效的缓存导致用户使用旧密码登录，因为在使用异步 Data Grid 复制时，无效的缓存不会传播到其他站点的故障点。

因此，高可用性和一致性之间存在权衡。本主题的重点是，与红帽构建的 Keycloak 相比，确定一致性优先级。

3.7.7. 后续步骤

继续阅读 构建块多集群部署 一章，以查找不同构建块的蓝图。

3.8. 构建会阻止多集群部署

了解构建块以及为多集群部署推荐的设置。

要使用同步复制设置多集群部署，需要以下构建块。

构建块链接到带有示例配置的蓝图。它们按照需要安装的顺序列出。

注意

我们提供这些蓝图来显示最小功能完整示例，为常规安装提供良好的基准性能。您仍然需要根据您的环境以及您的组织的标准和安全性最佳实践进行调整。

3.8.1. 先决条件

- 了解 多集群部署的概念 一章中的概念。

3.8.2. 两个具有低延迟连接的站点

红帽构建的 Keycloak 需要一个低延迟网络连接，用于由数据库和外部 Data Grid 同步数据。

建议少于 5 ms 的往返延迟，且需要低于 10 ms，以及区间的可靠的网络，以避免出现延迟、吞吐量或连接的问题。

网络延迟和延迟延迟在服务的响应时间中扩展，并可能导致排队的请求、超时和失败请求。网络问题可能会导致停机，直到失败检测隔离有问题的节点。

建议设置： 同一 AWS 区域中的两个 AWS 可用区。

不考虑： 位于相同或不同连续的两个区域，因为它会增加延迟和网络故障的可能性。在 AWS 上，同步将数据库复制为带有 Aurora 区域部署的服务。

3.8.3. 用于红帽构建的 Keycloak 和 Data Grid 的环境

确保实例已部署，并根据需要重启。

建议设置： 在每个可用区部署的 Red Hat OpenShift Service on AWS (ROSA)。

不考虑： 一个跨越多个可用区的 ROSA 集群，因为如果配置错误，这可能是单点故障。

3.8.4. 数据库

两个站点间同步复制的数据库。

蓝图： 在多个可用区中部署 AWS Aurora。

3.8.5. Data Grid

用于利用 Data Grid 的跨DC功能的 Data Grid 部署。

蓝图： 使用 Data Grid Operator 部署用于 HA 的 Data Grid，并使用 Data Grid 的 Gossip Router 连接两个站点。

未考虑： 在网络层上的 OpenShift 集群之间直接干预。以后可能会考虑它。

3.8.6. 红帽构建的 Keycloak

每个站点中红帽构建的 Keycloak 的集群部署，连接到外部 Data Grid。

蓝图： 使用 Operator 部署红帽构建的 Keycloak for HA，其中包括连接到 Aurora 数据库和 Data Grid 服务器。

3.8.7. 负载均衡器

一个负载均衡器，用于检查每个站点中红帽构建的 Keycloak 部署的 /lb-check URL，以及用于检测两

个站点之间的 Data Grid 连接问题的自动化。

蓝图： 将 AWS 全局加速器负载均衡器 与部署 AWS Lambda 一起部署，以禁用非响应站点。

3.9. 数据库连接池的概念

了解避免资源耗尽和拥塞的概念。

本节适用于对 Red Hat build of Keycloak 配置数据库连接池的注意事项和最佳实践。对于应用此功能的配置，请访问 使用 Operator 为 HA 部署红帽构建的 Keycloak。

3.9.1. 概念

创建新数据库连接所需的时间非常昂贵。当请求到达时，创建它们会延迟响应，因此在请求到达前，最好创建它们。它还有助于对在短时间内创建大量连接而造成问题的影响，因为它会减慢系统和块线程的速度。https://en.wikipedia.org/wiki/Cache_stampede关闭连接也会使连接的所有服务器端语句缓存无效。

为了获得最佳性能，初始、最小和最大数据库连接池大小的值都应相等。这可避免在新请求出现成本高时创建新的数据库连接。

只要可能，保持数据库连接打开，从而允许绑定至连接的服务器端语句缓存。如果是 PostgreSQL，若要使用服务器端准备好的声明，需要至少执行（默认）查询（默认为 ）。

如需更多信息，请参阅有关准备的语句的 PostgreSQL 文档。

3.10. 配置线程池的概念

了解避免资源耗尽和拥塞的概念。

本节旨在了解如何为红帽构建的 Keycloak 配置线程池连接池的注意事项和最佳实践。对于应用此功能的配置，请访问 使用 Operator 为 HA 部署红帽构建的 Keycloak。

3.10.1. 概念

3.10.1.1. Quarkus executor 池

红帽构建的 Keycloak 请求以及阻塞探测由 executor 池处理。根据可用的 CPU 内核，它的默认最大值为 50 个或更多线程。线程根据需要创建，并在不再需要时结束，因此系统将自动扩展和缩减。红帽构建的 Keycloak 允许通过 http-pool-max-threads 配置选项配置最大线程池大小。

3.10.1.2. Load Shedding

默认情况下，红帽构建的 Keycloak 将无限地将所有传入的请求排队，即使请求处理停止也是如此。这将在 Pod 中使用额外的内存，可能会耗尽负载均衡器中的资源，请求最终会在客户端一侧超时，而无需了解请求是否已被处理。要限制红帽构建的 Keycloak 中排队请求数，请设置额外的 Quarkus 配置选项。

配置 http-max-queued-requests，以指定在超过此队列大小后有效的负载均衡的最大队列长度。假设红帽构建的 Keycloak Pod 进程每秒约 200 个请求，则队列为 1000 会导致最大等待时间为 5 秒。

当此设置处于活跃状态时，超过排队请求数的请求将返回 HTTP 503 错误。Red Hat build of Keycloak 会在日志中记录错误消息。

3.10.1.3. probes

Red Hat build of Keycloak 的存活度探测是非阻止的，以避免在高负载下重启 Pod。

在一些情况下，整个健康探测和就绪度探测可能会检查与数据库的连接，因此它们可能会在高负载下失败。因此，Pod 可能会在高负载下变为未就绪。

3.10.1.4. OS 资源

为了使 Java 创建线程，在 Linux 上运行时，它需要有文件句柄。因此，打开的文件数量（如 ulimit -n on Linux）需要为红帽构建的 Keycloak 提供头空间，以增加所需的线程数量。每个线程也会消耗内存，容器内存限值需要设置为允许此的值，或 Pod 将由 OpenShift 终止。

3.11. 调整 CPU 和内存资源大小的概念

了解避免资源耗尽和拥塞的概念。

使用此选项作为调整产品环境的起点。根据您的负载测试，根据需要调整您的环境值。

3.11.1. 性能建议

警告

- 当扩展到更多 Pod （因为额外的开销）并使用多集群设置（因为额外的流量和操作）时，性能会降低。
- 当红帽构建用于长时间的 Keycloak 实例时，增加的缓存大小可以提高性能。这将减少响应时间并减少数据库的 IOPS。仍然需要在实例重启时填充这些缓存，因此不要根据缓存填写后测量的稳定状态设置资源。
- 使用这些值作为起点，并在进入生产环境前执行您自己的负载测试。

Summary：

- 已使用的 CPU 根据以下测试的限制来线性扩展。

建议：

- Pod 的基本内存用量，包括 Realm 数据和 10,000 个缓存会话的缓存 1250 MB RAM。
- 在容器中，Keycloak 为堆内存分配 70% 的内存限值。它还将使用大约 300 MB 的非堆内存。要计算请求的内存，请使用上面的计算。作为内存限制，从上面的值中减去非堆内存，并将结果除以 0.7 划分。

- 对于每秒 15 个基于密码的用户登录，请将 1 个 vCPU 分配给集群（每秒测试最多 300 个）。

  红帽构建的 Keycloak 使用大多数 CPU 时间哈希为用户提供的密码，并与哈希迭代数量成比例。

- 对于每个 120 个客户端凭证，每秒授予 1 个 vCPU 到集群（每秒测试最多 2000 个）。*

  大多数 CPU 时间都会创建新的 TLS 连接，因为每个客户端仅运行一个请求。

- 对于每个 120 每秒刷新令牌请求，1 个 vCPU 到集群（每秒测试了 435 刷新令牌请求）。*

- 保留 150% 额外头条以便 CPU 使用量处理负载高峰。这样可确保节点快速启动，并有足够的容量来处理故障转移任务。当其 Pod 在我们的测试中节流时，红帽构建的 Keycloak 的性能会显著下降。

- 当同时执行具有 2500 多个不同客户端的请求时，并非所有客户端信息都适合红帽构建的 Keycloak 的缓存，分别使用 10000 个条目的标准缓存。因此，数据库可能会成为瓶颈，因为客户端数据经常从数据库中重新加载。要减少数据库使用量，请将 用户 缓存大小增加到同时使用的客户端的两次，域 缓存大小由并发使用的客户端数的四倍增加。

Red Hat build of Keycloak （默认情况下，在数据库中存储用户会话）需要以下资源在 Aurora PostgreSQL 多AZ 数据库中获得最佳性能：

每个 100 个登录/logout/refresh 请求每秒：

- 1400 写 IOPS 预算.

- 在 0.35 和 0.7 vCPU 之间分配。

vCPU 要求作为范围提供，因为数据库主机上的 CPU 饱和度增加，每个请求的 CPU 使用量会降低，响应时间会增加。数据库上的 CPU 配额较低，可能会导致高峰负载期间的响应时间较慢。如果在峰值负

载期间快速响应时间至关重要，请选择较大的 CPU 配额。请参见以下示例。

3.11.1.1. 测量运行红帽构建的 Keycloak 实例的活动

红帽构建的 Keycloak 实例的大小取决于基于密码的用户登录、刷新令牌请求和客户端凭证的实际和预测的数量，如上一节中所述。

要检索这三个密钥输入的运行红帽构建的 Keycloak 实例的实际数量，请使用 Red Hat build of Keycloak 提供的指标：

- 事件类型登录的用户事件指标 keycloak_user_events_total 包括基于密码的登录和基于 Cookie 的登录，仍可作为此大小指南的第一个大约输入。
- 要查找红帽构建的 Keycloak 执行的密码验证数量，请使用 metric keycloak_credentials_password_hashing_validations_total。指标还包含一些有关使用的哈希算法和验证结果的标签。以下是可用标签的列表： realm, algorithm, hash_strength,结果.
- 将用户事件指标 keycloak_user_events_total 用于事件类型 refresh_token 和 client_login 分别用于刷新令牌请求和客户端凭证授权。

如需更多信息，请参阅使用事件指标和 HTTP 指标 章节监控用户活动。

这些指标对于跟踪用户活动负载中的每日和每周波动至关重要，找出可能表示需要调整系统大小并验证大小计算的新兴趋势。通过系统测量和评估这些用户事件指标，您可以确保您的系统保持正确扩展，并响应用户行为和需求的变化。

3.11.1.2. 计算示例（单集群）

目标大小：

- 45 个登录，每秒退出登录

- 360 客户端凭证每秒授予每秒*

- 360 刷新令牌请求每秒(1:8 ratio for login)*

- 3 个 pod

计算的限制：

- 每个 Pod 请求的 CPU: 3 个 vCPU

  （每秒45 个登录 = 3 个 vCPU，360 客户端凭据授予每秒 = 3 个 vCPU，360 刷新令牌 = 3 个 vCPU。总和最多 9 个 vCPU。当集群中运行的 3 个 Pod 时，每个 Pod 都会请求 3 个 vCPU。

- 每个 Pod 的 CPU 限制： 7.5 vCPU

  （允许额外的 150% CPU 处理峰值、启动和故障转移任务）

- 每个 Pod 请求的内存： 1250 MB

  (1250 MB 基本内存)

- 每个 Pod 的内存限值： 1360 MB

  (1250 MB 预期内存使用减 300 非堆使用，以 0.7 划分)

- Aurora Database 实例： db.t4g.large 或 db.t4g.xlarge，具体取决于高峰负载期间所需的响应时间。

  （每秒45 个登录，每秒 5 个退出一次，360 每秒刷新令牌。这个总和每秒 410 请求。这个预期 DB 使用为 1.4 到 2.8 vCPU，DB 闲置负载为 0.3 vCPU。这表示 2 个 vCPU db.t4g.large

实例或 4 个 vCPU db.t4g.xlarge 实例。如果在高峰使用期间允许响应时间更高，则 2 个 vCPU db.t4g.large 将更具成本效益。在我们的测试中，当 CPU 饱和达到了这个场景的 2 个 vCPU db.t4g.large 实例时，登录的介质和令牌刷新增加了 120 毫秒。为了在高峰使用期间更快的响应时间，请针对这种情况考虑 4 个 vCPU db.t4g.xlarge 实例。）

3.11.1.3. 调整多集群设置的大小

要创建在 AWS 区域中使用两个 AZ 的主动-主动 Keycloak 设置大小，请按照以下步骤操作：

- 创建与第二个站点上相同的内存大小相同的 Pod 数。
- 数据库大小保持不变。两个站点都将连接到同一数据库写入器实例。

根据 CPU 请求和限值的大小，根据预期的故障转移行为，有不同的方法：

快速故障转移和成本更高

为第二个站点保留 CPU 请求和限值。这样，任何剩余的站点都可以立即接管主站点的流量，而无需扩展。

故障转移速度较慢，更具成本效益

为第二个站点减少 CPU 请求和限值，超过 50%。当其中一个站点失败时，将剩余的站点从 3 个 Pod 扩展到 6 个 Pod，也可以手动、自动化或使用 Horizontal Pod Autoscaler。这需要在集群或集群自动扩展功能上有足够的备用容量。

某些环境的备用设置

为第二个站点将 CPU 请求减少 50%，但会保留以上 CPU 限值。这样，其余站点就可以获取流量，但只有在节点遇到 CPU 压力时，才会遇到 CPU 压力，因此在高峰流量期间的响应时间较慢。此设置的好处是，在故障切换过程中不需要扩展 Pod 的数量，这更易于设置。

3.11.2. 参考架构

以下设置用于检索以上设置，在不同场景中运行大约 10 分钟的设置：

- OpenShift 4.17.x 通过 ROSA 部署在 AWS 上。

- 具有 c7g.2xlarge 实例的机器池。*
- 在具有主动/主动模式下有两个站点的高可用性设置中使用 Operator 和 3 个 pod 部署的红帽 Keycloak 构建。
- OpenShift 的反向代理在 passthrough 模式下运行，其中客户端的 TLS 连接在 Pod 上终止。
- 在多AZ 设置中的数据库 Amazon Aurora PostgreSQL。
- 带有 Argon2 和 5 哈希迭代的默认用户密码散列，以及 OWASP （默认） 推荐的最小内存大小 7 MiB。
- 客户端凭证授予不使用刷新令牌（这是默认设置）。
- 数据库看到有 20,000 个用户和 20,000 个客户端。
- Infinispan local 缓存默认是 10,000 个条目，因此并非所有客户端和服务器都适合缓存，一些请求需要从数据库中获取数据。
- 默认情况下，分布式缓存中的所有身份验证会话都会有两个所有者，每个条目有两个所有者，允许一个失败的 Pod 而不丢失数据。
- 所有用户和客户端会话都存储在数据库中，不会缓存在内存中，因为这是在多集群设置中测试的。预期将缓存单站点设置的性能，因为固定用户和客户端会话将被缓存。
- OpenJDK 21

* 对于 AWS 上的非 ARM CPU 架构（c7i/c7a.c7g）发现，客户端凭证授予和刷新令牌工作负载能够为每个 CPU 内核提供两倍操作，而密码散列则为每个 CPU 内核提供持续数量的操作。根据您的工作负载

和云定价，请运行您自己的测试，并为混合工作负载进行自己的计算结果，以了解哪个架构为您提供了更好的定价。

3.12. 自动执行 DATA GRID CLI 命令的概念

通过创建一个 'Batch' CR 实例，可以自动执行 Data Grid CLI 命令。

在 OpenShift 中与外部 Data Grid 交互时，Batch CR 允许您使用标准 oc 命令自动执行它。

3.12.1. 何时使用它

在自动化 OpenShift 上的交互时，请使用此选项。这可避免提供用户名和密码，并检查 shell 脚本的输出及其状态。

对于人类交互，CLI shell 可能仍然更加适合。

3.12.2. Example

以下 批处理 CR 将站点离线，如操作流程 离线 所述。

```
apiVersion: infinispan.org/v2alpha1
kind: Batch
metadata:
  name: take-offline
  namespace: keycloak (1)
spec:
  cluster: infinispan (2)
  config: | (3)
    site take-offline --all-caches --site=site-a
    site status --all-caches --site=site-a
```

(1) Batch CR 必须与 Data Grid 部署在同一命名空间中创建。

(2) Infinispan CR 的名称。

3

包含一个或多个 Data Grid CLI 命令的多行字符串。

创建 CR 后，等待状态显示完成。

```
oc -n keycloak wait --for=jsonpath='{.status.phase}'=Succeeded Batch/take-offline
```

注意

修改 Batch CR 实例无效。批处理操作是修改 Infinispan 资源的"一次性"事件。要更新 CR 的 .spec 字段，或者在批处理操作失败时，您必须创建 Batch CR 的新实例。

3.12.3. 进一步阅读

如需更多信息，请参阅 Data Grid Operator Batch CR 文档。

3.13. 在多个可用区部署 AWS AURORA

在多集群部署中部署一个 AWS Aurora 作为数据库构建块。

本节论述了如何在多个可用区间部署 PostgreSQL 实例的 Aurora 区域部署，以便在给定 AWS 区域中容忍一个或多个可用区失败。

此部署旨在与 多集群部署 一章中介绍的设置一起使用。将此部署与构建块 多集群部署一章中介绍的其他构建块 一起使用。

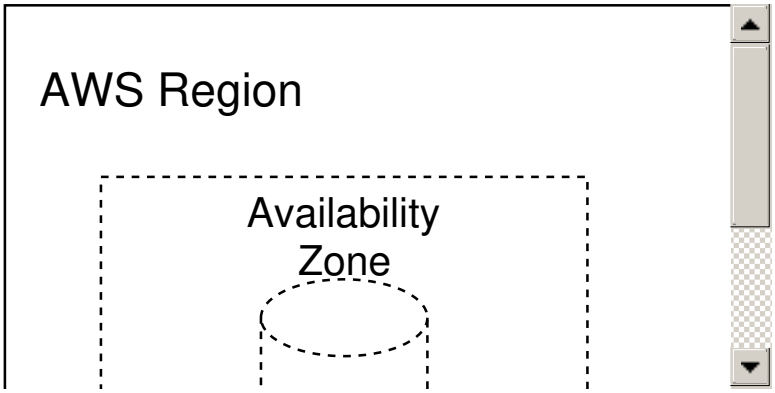
注意

我们提供这些蓝图来显示最小功能完整示例，为常规安装提供良好的基准性能。您仍然需要根据您的环境以及您的组织的标准和安全性最佳实践进行调整。

3.13.1. 架构

Aurora 数据库集群由多个 Aurora 数据库实例组成，一个实例指定为主写入器，所有其他实例都指定为备份读取器。为确保在可用区失败时高可用性，Aurora 允许在单个 AWS 区域中的多个区域部署数据库实例。如果在托管 Primary 数据库实例的可用区失败时，Aurora 会自动修复自身，并将读取器实例从非失败的可用区提升为新的写入器实例。

图 3.1. Aurora 多可用区部署



有关 Aurora 数据库提供的语义的详情，请参阅 AWS Aurora 文档。

本文档遵循 AWS 最佳实践并创建不向互联网公开的私有 Aurora 数据库。要从 ROSA 集群访问数据库，请在 数据库和 ROSA 集群之间建立对等连接。

3.13.2. 流程

以下流程包含两个部分：

- 在 eu-west-1 中使用名称 "keycloak-aurora" 创建 Aurora Multi-AZ 数据库集群。
- 在 ROSA 集群和 Aurora VPC 之间创建对等连接，以允许 ROSA 集群上部署的应用程序与数据库建立连接。

3.13.2.1. 创建 Aurora 数据库集群

1. 为 Aurora 集群创建一个 VPC

命令：

```
aws ec2 create-vpc \
  --cidr-block 192.168.0.0/16 \
  --tag-specifications "ResourceType=vpc, Tags=[{Key=AuroraCluster,Value=keycloak-aurora}]" \ 1
  --region eu-west-1
```

1

我们使用 Aurora 集群的名称添加可选标签，以便我们可以轻松地检索 VPC。

输出：

```
{
    "Vpc": {
        "CidrBlock": "192.168.0.0/16",
        "DhcpOptionsId": "dopt-0bae7798158bc344f",
        "State": "pending",
        "VpcId": "vpc-0b40bd7c59dbe4277",
        "OwnerId": "606671647913",
        "InstanceTenancy": "default",
        "Ipv6CidrBlockAssociationSet": [],
        "CidrBlockAssociationSet": [
            {
                "AssociationId": "vpc-cidr-assoc-09a02a83059ba5ab6",
                "CidrBlock": "192.168.0.0/16",
                "CidrBlockState": {
                    "State": "associated"
                }
            }
        ],
        "IsDefault": false
    }
}
```

2.

使用新创建的 VPC 的 VpcId 为 Aurora 部署到的每个可用区创建一个子网。

注意

为每个可用区指定的 cidr-block 范围不得互相重叠。

a. **区域 A**

命令：

```
aws ec2 create-subnet \
  --availability-zone "eu-west-1a" \
  --vpc-id vpc-0b40bd7c59dbe4277 \
  --cidr-block 192.168.0.0/19 \
  --region eu-west-1
```

输出：

```
{
    "Subnet": {
        "AvailabilityZone": "eu-west-1a",
        "AvailabilityZoneId": "euw1-az3",
        "AvailableIpAddressCount": 8187,
        "CidrBlock": "192.168.0.0/19",
        "DefaultForAz": false,
        "MapPublicIpOnLaunch": false,
        "State": "available",
        "SubnetId": "subnet-0d491a1a798aa878d",
        "VpcId": "vpc-0b40bd7c59dbe4277",
        "OwnerId": "606671647913",
        "AssignIpv6AddressOnCreation": false,
        "Ipv6CidrBlockAssociationSet": [],
        "SubnetArn": "arn:aws:ec2:eu-west-1:606671647913:subnet/subnet-
0d491a1a798aa878d",
        "EnableDns64": false,
        "Ipv6Native": false,
        "PrivateDnsNameOptionsOnLaunch": {
            "HostnameType": "ip-name",
            "EnableResourceNameDnsARecord": false,
            "EnableResourceNameDnsAAAARecord": false
        }
    }
}
```

-

b.
zone B

命令：

```
aws ec2 create-subnet \
  --availability-zone "eu-west-1b" \
  --vpc-id vpc-0b40bd7c59dbe4277 \
  --cidr-block 192.168.32.0/19 \
  --region eu-west-1
```

输出：

```
{
    "Subnet": {
        "AvailabilityZone": "eu-west-1b",
        "AvailabilityZoneId": "euw1-az1",
        "AvailableIpAddressCount": 8187,
        "CidrBlock": "192.168.32.0/19",
        "DefaultForAz": false,
        "MapPublicIpOnLaunch": false,
        "State": "available",
        "SubnetId": "subnet-057181b1e3728530e",
        "VpcId": "vpc-0b40bd7c59dbe4277",
        "OwnerId": "606671647913",
        "AssignIpv6AddressOnCreation": false,
        "Ipv6CidrBlockAssociationSet": [],
        "SubnetArn": "arn:aws:ec2:eu-west-1:606671647913:subnet/subnet-
057181b1e3728530e",
        "EnableDns64": false,
        "Ipv6Native": false,
        "PrivateDnsNameOptionsOnLaunch": {
            "HostnameType": "ip-name",
            "EnableResourceNameDnsARecord": false,
            "EnableResourceNameDnsAAAARecord": false
        }
    }
}
```

3. 获取 Aurora VPC 路由表 ID

命令：

```
aws ec2 describe-route-tables \
  --filters Name=vpc-id,Values=vpc-0b40bd7c59dbe4277 \
  --region eu-west-1
```

输出：


```
{
    "RouteTables": [
        {
            "Associations": [
                {
                    "Main": true,
                    "RouteTableAssociationId": "rtbassoc-02dfa06f4c7b4f99a",
                    "RouteTableId": "rtb-04a644ad3cd7de351",
                    "AssociationState": {
                        "State": "associated"
                    }
                }
            ],
            "PropagatingVgws": [],
            "RouteTableId": "rtb-04a644ad3cd7de351",
            "Routes": [
                {
                    "DestinationCidrBlock": "192.168.0.0/16",
                    "GatewayId": "local",
                    "Origin": "CreateRouteTable",
                    "State": "active"
                }
            ],
            "Tags": [],
            "VpcId": "vpc-0b40bd7c59dbe4277",
            "OwnerId": "606671647913"
        }
    ]
}
```

4.

关联 Aurora VPC 路由表每个可用区的子网

a.

区域 A

命令：

```
aws ec2 associate-route-table \
  --route-table-id rtb-04a644ad3cd7de351 \
  --subnet-id subnet-0d491a1a798aa878d \
  --region eu-west-1
```

b.

zone B

命令：

```
aws ec2 associate-route-table \
  --route-table-id rtb-04a644ad3cd7de351 \
  --subnet-id subnet-057181b1e3728530e \
  --region eu-west-1
```

5.

创建 Aurora 子网组

命令：

```
aws rds create-db-subnet-group \
```

```
  --db-subnet-group-name keycloak-aurora-subnet-group \
  --db-subnet-group-description "Aurora DB Subnet Group" \
  --subnet-ids subnet-0d491a1a798aa878d subnet-057181b1e3728530e \
  --region eu-west-1
```

6. 创建 Aurora 安全组

命令：

```
aws ec2 create-security-group \
  --group-name keycloak-aurora-security-group \
  --description "Aurora DB Security Group" \
  --vpc-id vpc-0b40bd7c59dbe4277 \
  --region eu-west-1
```

输出：

```
{
    "GroupId": "sg-0d746cc8ad8d2e63b"
}
```

7. 创建 Aurora DB 集群

命令：

```
aws rds create-db-cluster \
    --db-cluster-identifier keycloak-aurora \
    --database-name keycloak \
    --engine aurora-postgresql \
    --engine-version ${properties["aurora-postgresql.version"]} \
```

```
--master-username keycloak \
--master-user-password secret99 \
--vpc-security-group-ids sg-0d746cc8ad8d2e63b \
--db-subnet-group-name keycloak-aurora-subnet-group \
--region eu-west-1
```

注意

您应该替换 **--master-username** 和- **-master-user-password** 值。在配置红帽构建 **Keycloak** 数据库凭证时，必须使用此处指定的值。

输出：


```
{
  "DBCluster": {
    "AllocatedStorage": 1,
    "AvailabilityZones": [
      "eu-west-1b",
      "eu-west-1c",
      "eu-west-1a"
    ],
    "BackupRetentionPeriod": 1,
    "DatabaseName": "keycloak",
    "DBClusterIdentifier": "keycloak-aurora",
    "DBClusterParameterGroup": "default.aurora-postgresql15",
    "DBSubnetGroup": "keycloak-aurora-subnet-group",
    "Status": "creating",
    "Endpoint": "keycloak-aurora.cluster-clhthfqe0h8p.eu-west-
1.rds.amazonaws.com",
    "ReaderEndpoint": "keycloak-aurora.cluster-ro-clhthfqe0h8p.eu-west-
1.rds.amazonaws.com",
    "MultiAZ": false,
    "Engine": "aurora-postgresql",
    "EngineVersion": "15.5",
    "Port": 5432,
    "MasterUsername": "keycloak",
    "PreferredBackupWindow": "02:21-02:51",
    "PreferredMaintenanceWindow": "fri:03:34-fri:04:04",
    "ReadReplicaIdentifiers": [],
    "DBClusterMembers": [],
    "VpcSecurityGroups": [
      {
        "VpcSecurityGroupId": "sg-0d746cc8ad8d2e63b",
        "Status": "active"
      }
    ],
```

```
        "HostedZoneId": "Z29XKXDKYMONMX",
        "StorageEncrypted": false,
        "DbClusterResourceId": "cluster-IBWXUWQYM3MS5BH557ZJ6ZQU4I",
        "DBClusterArn": "arn:aws:rds:eu-west-1:606671647913:cluster:keycloak-aurora",
        "AssociatedRoles": [],
        "IAMDatabaseAuthenticationEnabled": false,
        "ClusterCreateTime": "2023-11-01T10:40:45.964000+00:00",
        "EngineMode": "provisioned",
        "DeletionProtection": false,
        "HttpEndpointEnabled": false,
        "CopyTagsToSnapshot": false,
        "CrossAccountClone": false,
        "DomainMemberships": [],
        "TagList": [],
        "AutoMinorVersionUpgrade": true,
        "NetworkType": "IPV4"
    }
}
```


8. 创建 Aurora DB 实例

   a. Create Zone A Writer 实例

   命令：

```
aws rds create-db-instance \
  --no-auto-minor-version-upgrade \
  --db-cluster-identifier keycloak-aurora \
  --db-instance-identifier "keycloak-aurora-instance-1" \
  --db-instance-class db.t4g.large \
  --engine aurora-postgresql \
  --region eu-west-1
```

   b. Create Zone B Reader 实例

   命令：

```
aws rds create-db-instance \
  --no-auto-minor-version-upgrade \
  --db-cluster-identifier keycloak-aurora \
  --db-instance-identifier "keycloak-aurora-instance-2" \
  --db-instance-class db.t4g.large \
  --engine aurora-postgresql \
  --region eu-west-1
```

9. 等待所有 Writer 和 Reader 实例就绪

命令：

```
aws rds wait db-instance-available --db-instance-identifier keycloak-aurora-instance-1 --region eu-west-1
aws rds wait db-instance-available --db-instance-identifier keycloak-aurora-instance-2 --region eu-west-1
```

10. Obtain the Writer 端点 URL 用于 Keycloak

命令：

```
aws rds describe-db-clusters \
  --db-cluster-identifier keycloak-aurora \
  --query 'DBClusters[*].Endpoint' \
  --region eu-west-1 \
  --output text
```

输出：

```
[
  "keycloak-aurora.cluster-clhthfqe0h8p.eu-west-1.rds.amazonaws.com"
]
```

3.13.2.2. 使用 ROSA 集群建立对等连接

为每个包含红帽构建的 Keycloak 部署 ROSA 集群执行这些步骤。

1. 检索 Aurora VPC

命令：

```
aws ec2 describe-vpcs \
  --filters "Name=tag:AuroraCluster,Values=keycloak-aurora" \
  --query 'Vpcs[*].VpcId' \
  --region eu-west-1 \
  --output text
```

输出：

```
vpc-0b40bd7c59dbe4277
```

2. 检索 ROSA 集群 VPC

a. 使用 oc登录到 ROSA 集群

b. 检索 ROSA VPC

命令：

```
NODE=$(oc get nodes --selector=node-role.kubernetes.io/worker -o
jsonpath='{.items[0].metadata.name}')
aws ec2 describe-instances \
  --filters "Name=private-dns-name,Values=${NODE}" \
  --query 'Reservations[0].Instances[0].VpcId' \
  --region eu-west-1 \
  --output text
```

输出：

```
vpc-0b721449398429559
```

3. 创建对等连接

命令：

```
aws ec2 create-vpc-peering-connection \
  --vpc-id vpc-0b721449398429559 \ 1
  --peer-vpc-id vpc-0b40bd7c59dbe4277 \ 2
  --peer-region eu-west-1 \
  --region eu-west-1
```

1

ROSA 集群 VPC

2

Aurora VPC

输出：


```
{
    "VpcPeeringConnection": {
        "AccepterVpcInfo": {
            "OwnerId": "606671647913",
            "VpcId": "vpc-0b40bd7c59dbe4277",
            "Region": "eu-west-1"
        },
        "ExpirationTime": "2023-11-08T13:26:30+00:00",
        "RequesterVpcInfo": {
            "CidrBlock": "10.0.17.0/24",
            "CidrBlockSet": [
                {
                    "CidrBlock": "10.0.17.0/24"
                }
            ],
            "OwnerId": "606671647913",
            "PeeringOptions": {
                "AllowDnsResolutionFromRemoteVpc": false,
                "AllowEgressFromLocalClassicLinkToRemoteVpc": false,
                "AllowEgressFromLocalVpcToRemoteClassicLink": false
            },
            "VpcId": "vpc-0b721449398429559",
            "Region": "eu-west-1"
        },
        "Status": {
            "Code": "initiating-request",
            "Message": "Initiating Request to 606671647913"
        },
        "Tags": [],
        "VpcPeeringConnectionId": "pcx-0cb23d66dea3dca9f"
    }
}
```

4. **等待 Peering 连接存在**

命令：

```
aws ec2 wait vpc-peering-connection-exists --vpc-peering-connection-ids pcx-0cb23d66dea3dca9f
```

5. **接受对等连接**

命令：

```
aws ec2 accept-vpc-peering-connection \
  --vpc-peering-connection-id pcx-0cb23d66dea3dca9f \
  --region eu-west-1
```

输出：

```
{
    "VpcPeeringConnection": {
        "AccepterVpcInfo": {
            "CidrBlock": "192.168.0.0/16",
            "CidrBlockSet": [
                {
                    "CidrBlock": "192.168.0.0/16"
                }
            ],
            "OwnerId": "606671647913",
            "PeeringOptions": {
                "AllowDnsResolutionFromRemoteVpc": false,
                "AllowEgressFromLocalClassicLinkToRemoteVpc": false,
                "AllowEgressFromLocalVpcToRemoteClassicLink": false
            },
            "VpcId": "vpc-0b40bd7c59dbe4277",
            "Region": "eu-west-1"
```

```
        },
        "RequesterVpcInfo": {
            "CidrBlock": "10.0.17.0/24",
            "CidrBlockSet": [
                {
                    "CidrBlock": "10.0.17.0/24"
                }
            ],
            "OwnerId": "606671647913",
            "PeeringOptions": {
                "AllowDnsResolutionFromRemoteVpc": false,
                "AllowEgressFromLocalClassicLinkToRemoteVpc": false,
                "AllowEgressFromLocalVpcToRemoteClassicLink": false
            },
            "VpcId": "vpc-0b721449398429559",
            "Region": "eu-west-1"
        },
        "Status": {
            "Code": "provisioning",
            "Message": "Provisioning"
        },
        "Tags": [],
        "VpcPeeringConnectionId": "pcx-0cb23d66dea3dca9f"
    }
}
```

6. 更新 ROSA 集群 VPC 路由表

命令：

```
ROSA_PUBLIC_ROUTE_TABLE_ID=$(aws ec2 describe-route-tables \
  --filters "Name=vpc-id,Values=vpc-0b721449398429559"
"Name=association.main,Values=true" \ 1
  --query "RouteTables[*].RouteTableId" \
  --output text \
  --region eu-west-1
)
aws ec2 create-route \
  --route-table-id ${ROSA_PUBLIC_ROUTE_TABLE_ID} \
  --destination-cidr-block 192.168.0.0/16 \ 2
  --vpc-peering-connection-id pcx-0cb23d66dea3dca9f \
  --region eu-west-1
```

1

ROSA 集群 VPC

2

这必须与创建 Aurora VPC 时使用的 cidr-block 相同

7. 更新 Aurora 安全组

命令：

```
AURORA_SECURITY_GROUP_ID=$(aws ec2 describe-security-groups \
  --filters "Name=group-name,Values=keycloak-aurora-security-group" \
  --query "SecurityGroups[*].GroupId" \
  --region eu-west-1 \
  --output text
)
aws ec2 authorize-security-group-ingress \
  --group-id ${AURORA_SECURITY_GROUP_ID} \
  --protocol tcp \
  --port 5432 \
  --cidr 10.0.17.0/24 \ 1
  --region eu-west-1
```

1

ROSA 集群的 "machine_cidr"

输出：


```
{
    "Return": true,
    "SecurityGroupRules": [
        {
            "SecurityGroupRuleId": "sgr-0785d2f04b9cec3f5",
            "GroupId": "sg-0d746cc8ad8d2e63b",
            "GroupOwnerId": "606671647913",
```

```
            "IsEgress": false,
            "IpProtocol": "tcp",
            "FromPort": 5432,
            "ToPort": 5432,
            "CidrIpv4": "10.0.17.0/24"
        }
    ]
}
```

3.13.3. 验证连接

验证 ROSA 集群和 Aurora DB 集群之间是否可以连接的最简单方法是在 OpenShift 集群上部署 psql，并尝试连接到写器端点。

以下命令在 default 命名空间中创建 pod，并在可能的情况下建立与 Aurora 集群的 psql 连接。退出 pod shell 后，会删除 pod。

```
USER=keycloak 1
PASSWORD=secret99 2
DATABASE=keycloak 3
HOST=$(aws rds describe-db-clusters \
  --db-cluster-identifier keycloak-aurora \ 4
  --query 'DBClusters[*].Endpoint' \
  --region eu-west-1 \
  --output text
)
oc run -i --tty --rm debug --image=postgres:15 --restart=Never -- psql
postgresql://${USER}:${PASSWORD}@${HOST}/${DATABASE}
```

1

Aurora DB 用户，这可以与创建 DB 时使用的 -master-username 相同。

2

Aurora DB 用户密码，与创建 DB 时使用的 master-user-password 相同。

3

Aurora DB 的名称，如 --database-name。

4

Aurora DB 集群的名称。

3.13.4. 使用红帽构建的 Keycloak 连接 Aurora 数据库

现在，Aurora 数据库已经建立并链接到所有 ROSA 集群，以下是相关的红帽构建的 Keycloak CR 选项，用于将 Aurora 数据库与红帽构建的 Keycloak 连接。在使用 Operator 部署 Red Hat build of HA 中， 需要这些更改。JDBC url 被配置为使用 Aurora 数据库写入器端点。

1. 将 spec.db.url 更新为 jdbc:aws-wrapper:postgresql://$HOST:5432/keycloak，其中 $HOST 是 Aurora writer 端点 URL。

2. 确保 spec.db.usernameSecret 和 spec.db.passwordSecret 引用的 Secret 包含创建 Aurora 时定义的用户名和密码。

3.13.5. 后续步骤

在成功部署 Aurora 数据库后，使用 Data Grid Operator 为 HA 部署 Data Grid

3.14. 使用 DATA GRID OPERATOR 部署用于 HA 的 DATA GRID

在 OpenShift 上的多可用区部署 Data Grid 以实现高可用性。

本章论述了在多集群环境（跨站点）中部署 Data Grid 所需的步骤。为了简单起见，本主题使用最小的配置，允许红帽构建 Keycloak 与外部 Data Grid 一起使用。

本章假定两个 OpenShift 集群名为 Site-A 和 Site-B。

这是一个构建块，它遵循 多集群部署 一章中介绍的概念。如需了解概述，请参阅 多集群部署 章节。

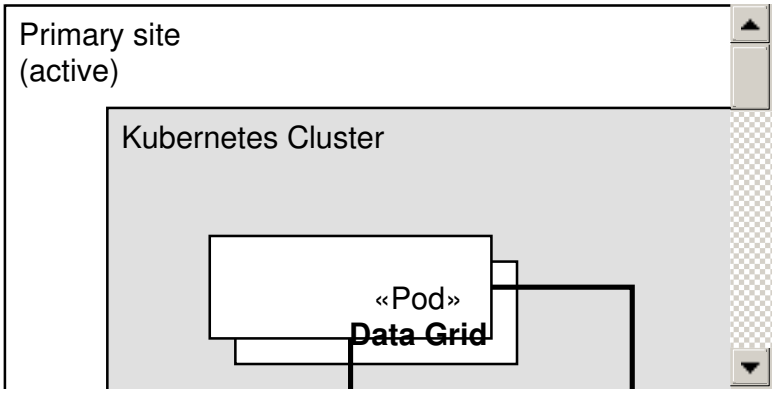
重要

对于外部 Data Grid 部署，只支持 Data Grid 版本 8.5.3 或最新的补丁版本。

3.14.1. 架构

此设置会在具有低延迟网络连接的两个站点中部署两个同步复制 Data Grid 集群。例如，在这种情况下，可能是一个 AWS 区域的两个可用区。

为了简单起见，红帽构建的 Keycloak, loadbalancer 和 database 已从下图中删除。



3.14.2. 先决条件

- 正在运行的 OpenShift 集群
- 了解 Data Grid Operator

3.14.3. 流程

1. 安装 Data Grid Operator
2. 配置用于访问 Data Grid 集群的凭证。

   红帽构建的 Keycloak 需要此凭证才能与 Data Grid 集群进行身份验证。以下 identity.yaml 文件使用 admin 权限设置用户名和密码

```
credentials:
 - username: developer
   password: strong-password
   roles:
     - admin
```

identity.yaml 可以在 secret 中设置为以下之一：

- 作为 OpenShift 资源：

凭证 Secret

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: connect-secret
  namespace: keycloak
data:
  identities.yaml:
Y3JlZGVudGlhbHM6CiAgLSB1c2VybmFtZTogZGV2ZWxvcGVyCiAgICBwYXNzd29y
ZDogc3Ryb25nLXBhc3N3b3JkCiAgICByb2xlczoKICAgICAgLSBhZG1pbgo= (1)
```

(1) 前面示例 base64 编码中的 identity .yaml。

- 使用 CLI

```
oc create secret generic connect-secret --from-file=identities.yaml
```

如需了解更多详细信息，请参阅配置身份验证文档。https://docs.redhat.com/en/documentation/red_hat_data_grid/8.5/html-single/data_grid_operator_guide/index#configuring-authentication

这两个 OpenShift 集群上都必须执行这些命令。

3. 创建一个服务帐户。

在集群间建立连接需要服务帐户。Data Grid Operator 用来检查远程站点的网络配置，并相应地配置本地 Data Grid 集群。

如需了解更多详细信息，请参阅管理跨站点连接 文档。

a. 按照如下所示，创建一个 service-account-token 机密类型：同一 YAML 文件可用于两个 OpenShift 集群。

xsite-sa-secret-token.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: ispn-xsite-sa-token 1
  annotations:
    kubernetes.io/service-account.name: "xsite-sa" 2
type: kubernetes.io/service-account-token
```

1 机密名称。

2 服务帐户名称。

b. 创建服务帐户，并在两个 OpenShift 集群中生成访问令牌。

在 Site-A中创建服务帐户

```
oc create sa -n keycloak xsite-sa
oc policy add-role-to-user view -n keycloak -z xsite-sa
oc create -f xsite-sa-secret-token.yaml
oc get secrets ispn-xsite-sa-token -o jsonpath="{.data.token}" | base64 -d > Site-A-token.txt
```

在 Site-B中创建服务帐户

```
oc create sa -n keycloak xsite-sa
oc policy add-role-to-user view -n keycloak -z xsite-sa
oc create -f xsite-sa-secret-token.yaml
oc get secrets ispn-xsite-sa-token -o jsonpath="{.data.token}" | base64 -d > Site-B-token.txt
```

c. 下一步是将来自 Site-A 的令牌部署到 Site-B，反向部署：

将 Site-B 令牌部署到 Site-A

```
oc create secret generic -n keycloak xsite-token-secret \
  --from-literal=token="$(cat Site-B-token.txt)"
```

将 Site-A 令牌部署到 Site-B

```
oc create secret generic -n keycloak xsite-token-secret \
  --from-literal=token="$(cat Site-A-token.txt)"
```

4.

创建 TLS secret

在本章中，Data Grid 使用 OpenShift 路由进行跨站点通信。它使用 TLS 的 SNI 扩展将流量定向到正确的 Pod。为达到此目的，JGroups 使用 TLS 套接字，这需要一个含有正确证书的 Keystore 和 Truststore。

如需更多信息，请参阅安全 跨站点连接 文档或本 红帽开发人员指南。

上传 OpenShift Secret 中的 Keystore 和 Truststore。secret 包含文件内容、访问它的密码以及存储的类型。创建证书和存储的说明超出了本章的讨论范围。

要将 Keystore 作为 Secret 上传，请使用以下命令：

部署密钥存储

```
oc -n keycloak create secret generic xsite-keystore-secret \
  --from-file=keystore.p12="./certs/keystore.p12" \ 1
  --from-literal=password=secret \ 2
  --from-literal=type=pkcs12 3
```

1

文件名和到密钥存储的路径。

2

用于访问密钥存储的密码。

3

Keystore 类型。

要将 Truststore 上传为 Secret，请使用以下命令：

部署 Truststore

```
oc -n keycloak create secret generic xsite-truststore-secret \
    --from-file=truststore.p12="./certs/truststore.p12" \ 1
    --from-literal=password=caSecret \ 2
    --from-literal=type=pkcs12 3
```

1

文件名和到 Truststore 的路径。

2

访问 Truststore 的密码。

3

Truststore 类型。

注意

keystore 和 Truststore 必须在两个 OpenShift 集群中上传。

5. 为启用了跨站点的 Data Grid 创建一个集群

设置跨站点 文档提供了关于如何在启用了跨站点的情况下创建和配置 Data Grid 集群的所有信息，包括前面的步骤。

本章提供了一个基本示例，它使用前面步骤中命令创建的凭据、令牌和 TLS Keystore/Truststore。

Site-A的 Infinispan CR

```
apiVersion: infinispan.org/v1
kind: Infinispan
metadata:
  name: infinispan (1)
  namespace: keycloak
  annotations:
    infinispan.org/monitoring: 'true' (2)
spec:
  replicas: 3
  jmx:
    enabled: true
  security:
    endpointSecretName: connect-secret (3)
  service:
    type: DataGrid
    sites:
      local:
        name: site-a (4)
        expose:
          type: Route (5)
        maxRelayNodes: 128
        encryption:
          transportKeyStore:
            secretName: xsite-keystore-secret (6)
            alias: xsite (7)
            filename: keystore.p12 (8)
          routerKeyStore:
            secretName: xsite-keystore-secret (9)
            alias: xsite (10)
            filename: keystore.p12 (11)
          trustStore:
            secretName: xsite-truststore-secret (12)
            filename: truststore.p12 (13)
      locations:
        - name: site-b (14)
          clusterName: infinispan
          namespace: keycloak (15)
          url: openshift://api.site-b (16)
          secretName: xsite-token-secret (17)
```

1

集群名称

2

允许 Prometheus 监控集群。

3

如果使用自定义凭证，请在此处配置 secret 名称。

4

本地站点的名称，本例中为 Site-A。

5

使用 OpenShift Route 公开跨站点连接。

6 9

上一步中定义的 Keystore 所在的 secret 名称。

7 10

Keystore 中证书的别名。

8 11

上一步中定义的密钥存储的机密密钥(filename)。

12

Truststore 存在于上一步中定义的 secret 名称。

13

上一步中定义的密钥存储的 Truststore 密钥(filename)。

14

远程站点的名称，本例中为 Site-B。您可以在红帽构建的 Keycloak 选项 cache-remote-backup-sites 中使用这个值来创建自动缓存。

15

来自远程站点的 Data Grid 集群的命名空间。

16

远程站点的 OpenShift API URL。

17

带有访问令牌的 secret，以向远程站点进行身份验证。

对于 Site-B，Infinispan CR 类似于上述内容。请注意点 4、11 和 13 的不同。

Site-B的 Infinispan CR

```
apiVersion: infinispan.org/v1
kind: Infinispan
metadata:
  name: infinispan 1
  namespace: keycloak
  annotations:
    infinispan.org/monitoring: 'true' 2
spec:
  replicas: 3
  jmx:
    enabled: true
  security:
    endpointSecretName: connect-secret 3
  service:
    type: DataGrid
    sites:
      local:
        name: site-b 4
        expose:
          type: Route 5
        maxRelayNodes: 128
        encryption:
          transportKeyStore:
            secretName: xsite-keystore-secret 6
            alias: xsite 7
            filename: keystore.p12 8
          routerKeyStore:
```

```
      secretName: xsite-keystore-secret 9
      alias: xsite 10
      filename: keystore.p12 11
    trustStore:
      secretName: xsite-truststore-secret 12
      filename: truststore.p12 13
  locations:
  - name: site-a 14
    clusterName: infinispan
    namespace: keycloak 15
    url: openshift://api.site-a 16
    secretName: xsite-token-secret 17
```

6. 为红帽构建的 Keycloak 创建缓存。

如果不存在，Red Hat build of Keycloak 会自动在第一次启动时创建所需的缓存。

重要

它要求红帽构建的 Keycloak 部署到两个集群中，因为红帽构建的 Keycloak 不会启动，直到两个集群中都存在所有缓存。

要 缓存 CR 是在 OpenShift 中继续进行的建议方法。为了生效，必须在任何红帽构建的 Keycloak Pod 启动前部署 Cache CR。

以下示例显示了 Site-A 的 Cache CR。

1. 在 Site-A 中，为上述每个缓存创建一个缓存 CR，其内容如下：

cache actionTokens

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
```

```
  name: actiontokens
  namespace: keycloak
spec:
  clusterName: infinispan
  name: actionTokens
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
        media-type: "application/x-protostream"
      locking:
        acquireTimeout: "4000"
      transaction:
        mode: "NON_DURABLE_XA" 1
        locking: "PESSIMISTIC" 2
      stateTransfer:
        chunkSize: "16"
      backups:
        site-b: 3
          backup:
            strategy: "SYNC" 4
            timeout: "4500" 5
            failurePolicy: "FAIL" 6
            stateTransfer:
              chunkSize: "16"
```

Cache authenticationSessions

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
  name: authenticationsessions
  namespace: keycloak
spec:
  clusterName: infinispan
  name: authenticationSessions
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
        media-type: "application/x-protostream"
      locking:
```

```
      acquireTimeout: "4000"
    transaction:
      mode: "NON_DURABLE_XA" (1)
      locking: "PESSIMISTIC" (2)
    stateTransfer:
      chunkSize: "16"
    indexing:
      enabled: true
      indexed-entities:
      - keycloak.RootAuthenticationSessionEntity
    backups:
      site-b: (3)
        backup:
          strategy: "SYNC" (4)
          timeout: "4500" (5)
          failurePolicy: "FAIL" (6)
          stateTransfer:
            chunkSize: "16"
```

缓存 loginFailures

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
  name: loginfailures
  namespace: keycloak
spec:
  clusterName: infinispan
  name: loginFailures
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
        media-type: "application/x-protostream"
      locking:
        acquireTimeout: "4000"
      transaction:
        mode: "NON_DURABLE_XA" (1)
        locking: "PESSIMISTIC" (2)
      stateTransfer:
        chunkSize: "16"
      indexing:
        enabled: true
        indexed-entities:
        - keycloak.LoginFailureEntity
```

```
    backups:
      site-b: 3
        backup:
          strategy: "SYNC" 4
          timeout: "4500" 5
          failurePolicy: "FAIL" 6
          stateTransfer:
            chunkSize: "16"
```

缓存 工作

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
  name: work
  namespace: keycloak
spec:
  clusterName: infinispan
  name: work
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
        media-type: "application/x-protostream"
      locking:
        acquireTimeout: "4000"
      transaction:
        mode: "NON_DURABLE_XA" 1
        locking: "PESSIMISTIC" 2
      stateTransfer:
        chunkSize: "16"
      backups:
        site-b: 3
          backup:
            strategy: "SYNC" 4
            timeout: "4500" 5
            failurePolicy: "FAIL" 6
            stateTransfer:
              chunkSize: "16"
```

1 1 1 1 1

事务模式。

2 2 2 2 2

事务使用的锁定模式。

3 3 3 3 3

远程站点名称。

4 4 4 4 4

跨站点通信策略，本例中为 SYNC。

5 5 5 5 5

跨站点复制超时。

6 9 6 6 6 6

跨站点复制失败策略。

上面的例子是建议配置来实现最佳数据一致性。

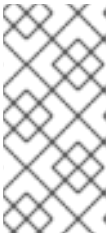
背景信息

当两个站点同时修改条目时，主动设置中可能会出现死锁。

transaction.mode: NON_DURABLE_XA 确保在发生这种情况时事务被回滚一致。本例中需要设置 backup.failurePolicy: FAIL。它将抛出一个错误，允许安全回滚事务。当发生这种情况时，红帽构建的 Keycloak 将尝试重试。

transaction.locking: PESSIMISTIC 是唯一支持的锁定模式；由于网络成本，不建议使用 OPTIMISTIC。相同的设置也会阻止在一个站点更新，而其他站点无法访问。

backup.strategy: SYNC 确保数据在完成红帽构建的 Keycloak 请求时可见并存储在其他站点。



注意

在死锁场景中，可以将 locking.acquireTimeout 减少为快速失败。backup.timeout 必须始终高于 locking.acquireTimeout。

对于 Site-B，缓存 CR 非常相似，但 backup.& lt;name> 在上图的第 3 点概述。

Site-B中的 actionTokens 缓存示例

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
  name: actiontokens
  namespace: keycloak
spec:
  clusterName: infinispan
  name: actionTokens
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
        media-type: "application/x-protostream"
      locking:
        acquireTimeout: "4000"
      transaction:
        mode: "NON_DURABLE_XA" 1
        locking: "PESSIMISTIC" 2
      stateTransfer:
        chunkSize: "16"
      backups:
        site-a: 3
          backup:
            strategy: "SYNC" 4
            timeout: "4500" 5
            failurePolicy: "FAIL" 6
            stateTransfer:
              chunkSize: "16"
```

== 验证部署

确认 Data Grid 集群已形成，且跨站点连接已在 OpenShift 集群之间建立。

等待 Data Grid 集群被形成

```
oc wait --for condition=WellFormed --timeout=300s infinispans.infinispan.org -n keycloak infinispan
```

等待 Data Grid 跨站点连接建立

```
oc wait --for condition=CrossSiteViewFormed --timeout=300s infinispans.infinispan.org -n keycloak infinispan
```

3.14.4. 使用红帽构建的 Keycloak 连接 Data Grid

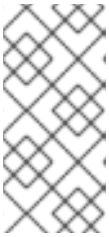
现在，Data Grid 服务器正在运行，以下是需要与红帽构建的 Keycloak CR 更改相关的红帽构建，以将其连接到红帽构建的 Keycloak。在使用 Operator 部署 Red Hat build of HA 中， 需要这些更改。

1. 使用用户名和密码创建一个 Secret，以连接到外部 Data Grid 部署：

```
apiVersion: v1
kind: Secret
metadata:
  name: remote-store-secret
  namespace: keycloak
type: Opaque
```

```
data:
  username: ZGV2ZWxvcGVy # base64 encoding for 'developer'
  password: c2VjdXJlX3Bhc3N3b3Jk # base64 encoding for 'secure_password'
```

2. 使用 additionalOptions 扩展 Red Hat build of Keycloak 自定义资源，如下所示。



注意

所有内存、资源和数据库配置都会从 CR 跳过，因为 使用 Operator 部署红帽构建的 Keycloak for HA 所述。管理员应使这些配置保持不变。

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  labels:
    app: keycloak
  name: keycloak
  namespace: keycloak
spec:
  additionalOptions:
    - name: cache-remote-host (1)
      value: "infinispan.keycloak.svc"
    - name: cache-remote-port (2)
      value: "11222"
    - name: cache-remote-username (3)
      secret:
        name: remote-store-secret
        key: username
    - name: cache-remote-password (4)
      secret:
        name: remote-store-secret
        key: password
    - name: cache-remote-backup-sites
      value: "site-2" (5)
```

(1) (1)

远程 Data Grid 集群的主机名。

(2) (2)

远程 Data Grid 集群的端口。这是可选的，默认为 11222。

(3) (3)

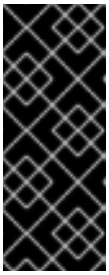
带有 Data Grid 用户名凭证的 Secret 名称和 密钥。

4 4

使用 Data Grid 密码凭证的机密名称和密钥。

5 5

（可选）远程站点的名称。只有在缓存不存在时，才会创建缓存。



重要

使用 cache-remote-backup-sites 选项仅在本地站点中创建缓存。您还必须在其他集群中部署 KeycloakCR，才能创建缓存，否则红帽构建的 Keycloak 无法启动，直到它们存在为止。

3.14.4.1. 架构

这使用 TLS 1.3 保护的 TCP 连接将 Keycloak 的红帽构建连接到 Data Grid。它使用红帽构建的 Keycloak 的信任存储来验证 Data Grid 的服务器证书。因为红帽构建的 Keycloak 是使用 OpenShift 在下面列出的先决条件中的 Operator 部署，Operator 已将 service-ca.crt 添加到用于为 Data Grid 的服务器证书签名的信任存储中。在其他环境中，将所需的证书添加到红帽构建的 Keycloak 的信任存储中。

3.14.5. 后续步骤

在部署并运行 AWS Aurora 数据库和 Data Grid 后，请使用 使用 Operator 部署红帽构建的 Keycloak for HA 中的步骤来部署红帽构建的 Keycloak，并将其连接到之前创建的构建块。

3.14.6. 相关选项

	value
cache-remote-backup-sites 配置备份站点名称列表到外部 Infinispan 集群备份 Keycloak 数据的位置。 CLI: **--cache-remote-backup-sites** Env: **KC_CACHE_REMOTE_BACKUP_SITES** 仅在设置远程主机时可用	

	value
cache-remote-host 外部 Infinispan 集群的主机名。 仅在设置功能 **多站点** 或无 **集群** 时才可用。 CLI: **--cache-remote-host** Env: **KC_CACHE_REMOTE_HOST**	
cache-remote-password 对外部 Infinispan 集群进行身份验证的密码。 如果连接到不安全的外部 Infinispan 集群，则它是可选的。如果指定了这个选项，则需要 **cache-remote-username**。 CLI: **--cache-remote-password** Env: **KC_CACHE_REMOTE_PASSWORD** 仅在设置远程主机时可用	
cache-remote-port 外部 Infinispan 集群的端口。 CLI: **--cache-remote-port** Env: **KC_CACHE_REMOTE_PORT** 仅在设置远程主机时可用	**11222** （默认）
cache-remote-tls-enabled 启用 TLS 支持与安全远程 Infinispan 服务器通信。 建议在生产环境中启用。 CLI: **--cache-remote-tls-enabled** Env: **KC_CACHE_REMOTE_TLS_ENABLED** 仅在设置远程主机时可用	**true** （默认），**false**
cache-remote-username 外部 Infinispan 集群身份验证的用户名。 如果连接到不安全的外部 Infinispan 集群，则它是可选的。如果指定了 选项，则需要 **cache-remote-password**。 CLI: **--cache-remote-username** Env: **KC_CACHE_REMOTE_USERNAME** 仅在设置远程主机时可用	

3.15. 使用 OPERATOR 部署红帽构建的 KEYCLOAK FOR HA

使用红帽构建的 Keycloak Operator 作为构建块，部署红帽构建的 Keycloak 以实现高可用性。

本章论述了 OpenShift 的 Keycloak 配置的高级红帽构建，这些配置已测试，并将从单个 Pod 失败中恢复。

这些说明 适用于多集群部署 一章中介绍的设置。将其与构建块 多集群部署一章中介绍的其他构建块 一起使用。

3.15.1. 先决条件

- 正在运行的 OpenShift 集群.
- 了解使用 红帽构建的 Keycloak Operator 进行红帽构建的 Keycloak 部署的基本红帽构建。
- 使用在 多个可用区中的 Deploying AWS Aurora 部署的 AWS Aurora 数据库。
- Data Grid 服务器通过 Data Grid Operator 使用 Deploying Data Grid for HA 部署。

3.15.2. 流程

1. 确定使用 概念调整 CPU 和内存资源章节的部署大小。
2. 按照 Red Hat build of Keycloak Operator 安装中的内容 安装 Red Hat build of Keycloak Operator。
3. 请注意，以下配置文件包含与 多个可用区部署 AWS Aurora 中连接到 Aurora数据库相关的选项
4. 请注意以下与 Data Grid Operator 部署 Data Grid 相关的选项。

5. 构建自定义红帽构建的 Keycloak 镜像，准备与 Amazon Aurora PostgreSQL 数据库一起使用。

6. 使用在第一步中计算的资源请求和限值，使用以下值部署红帽 Keycloak CR：

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  labels:
    app: keycloak
  name: keycloak
  namespace: keycloak
spec:
  hostname:
    hostname: <KEYCLOAK_URL_HERE>
  resources:
    requests:
      cpu: "2"
      memory: "1250M"
    limits:
      cpu: "6"
      memory: "2250M"
  db:
    vendor: postgres
    url: jdbc:aws-wrapper:postgresql://<AWS_AURORA_URL_HERE>:5432/keycloak
    poolMinSize: 30 (1)
    poolInitialSize: 30
    poolMaxSize: 30
    usernameSecret:
      name: keycloak-db-secret
      key: username
    passwordSecret:
      name: keycloak-db-secret
      key: password
  image: <KEYCLOAK_IMAGE_HERE> (2)
  startOptimized: false (3)
  features:
    enabled:
      - multi-site (4)
  additionalOptions:
    - name: log-console-output
      value: json
    - name: metrics-enabled (5)
      value: 'true'
    - name: event-metrics-user-enabled
      value: 'true'
    - name: cache-remote-host
      value: "infinispan.keycloak.svc"
    - name: cache-remote-port
      value: "11222"
    - name: cache-remote-username
```

```
    secret:
      name: remote-store-secret
      key: username
  - name: cache-remote-password
    secret:
      name: remote-store-secret
      key: password
  - name: db-driver
    value: software.amazon.jdbc.Driver
http:
  tlsSecret: keycloak-tls-secret
instances: 3
```

1

数据库连接池 initial、max 和 min 大小应当相同，以允许数据库的声明缓存。调整这个数字以满足您的系统需求。因为大多数请求不会因为红帽构建的 Keycloak 嵌入式缓存而影响数据库，所以这个更改每秒可以提供几百个请求。详情请参阅数据库连接池的概念 章节。

2 3

指定自定义红帽构建的 Keycloak 镜像的 URL。如果您的镜像经过优化，请将 startOptimized 标志设置为 true。

4

为多集群支持启用额外的功能，如 loadbalancer probe /lb-check。

5

为了能够在负载下分析系统，启用指标端点。

3.15.3. 验证部署

确认红帽构建的 Keycloak 部署已就绪。

```
oc wait --for=condition=Ready keycloaks.k8s.keycloak.org/keycloak
oc wait --for=condition=RollingUpdate=False keycloaks.k8s.keycloak.org/keycloak
```

3.15.4. 可选： Load shedding

要启用负载均衡，请限制排队的请求数。

使用最大排队的 http 请求加载她

```
spec:
  additionalOptions:
    - name: http-max-queued-requests
      value: "1000"
```

所有超过的请求都使用 HTTP 503 提供。

您可以考虑进一步限制 http-pool-max-threads 的值，因为在达到请求的 CPU 限制后，OpenShift 会导致 OpenShift 节流。

有关详细信息，请参阅有关配置线程池 的概念 一章。

3.15.5. 可选：禁用粘性会话

当在 OpenShift 上运行以及由 Red Hat build of Keycloak Operator 提供的默认 passthrough Ingress 设置时，HAProxy 所做的负载均衡会根据源的 IP 地址使用粘性会话来实现。在运行负载测试时，或者在 HAProxy 前面运行反向代理时，您可能需要禁用此设置以避免在单个红帽构建的 Keycloak Pod 上接收所有请求。

在红帽构建的 Keycloak 自定义资源的 spec 下添加以下补充配置，以禁用粘性会话。

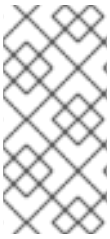
```
spec:
  ingress:
    enabled: true
    annotations:
      # When running load tests, disable sticky sessions on the OpenShift HAProxy router
      # to avoid receiving all requests on a single Red Hat build of Keycloak Pod.
      haproxy.router.openshift.io/balance: roundrobin
      haproxy.router.openshift.io/disable_cookies: 'true'
```

3.16. 部署 AWS 全局加速器负载均衡器

在多集群部署中部署一个 AWS 全局加速器作为负载平衡器构建块。

本节论述了部署 AWS 全局加速器所需的流程，以便在多集群红帽构建 Keycloak 部署之间路由流量。

此部署旨在与 多集群部署 一章中介绍的设置一起使用。将此部署与构建块 多集群部署一章中介绍的其他构建块 一起使用。



注意

我们提供这些蓝图来显示最小功能完整示例，为常规安装提供良好的基准性能。您仍然需要根据您的环境以及您的组织的标准和安全性最佳实践进行调整。

3.16.1. 受众

本章论述了如何部署 AWS 全局加速器实例，以处理红帽构建的 Keycloak 客户端连接故障切换。

3.16.2. 架构

为确保用户请求路由到每个红帽构建的 Keycloak 站点，我们需要使用负载均衡器。为了防止客户端上 DNS 缓存出现问题，实施应使用在将客户端路由到两个可用区时保持不变的静态 IP 地址。

在本章中，我们描述了如何通过 AWS Global Accelerator 负载均衡器路由所有红帽构建的 Keycloak 客户端请求。如果红帽构建的 Keycloak 站点构建失败，则加速器确保所有客户端请求都路由到剩余的健康站点。如果两个站点都标记为不健康，则加速器将"fail-open"并将请求转发到随机选择的站点。

图 3.2. AWS Global Accelerator Failover

在两个 ROSA 集群上都创建一个 AWS Network Load Balancer (NLB)，以便将 Keycloak pod 作为 Endpoints 提供给 AWS 全局加速器实例。每个集群端点都会被分配一个 128 （最大权重为 255）的权重），以确保在两个集群都健康时加速器流量平均路由到两个可用区。

3.16.3. 先决条件

- 基于 ROSA 的 Multi-AZ 红帽构建的 Keycloak 部署

3.16.4. 流程

1. 创建网络负载平衡器

   在每个红帽构建的 Keycloak 集群中执行以下操作：

   a. 登录到 ROSA 集群

   b. 创建 OpenShift 负载均衡器服务

   命令：

```
cat <<EOF | oc apply -n $NAMESPACE -f - 1
  apiVersion: v1
  kind: Service
  metadata:
    name: accelerator-loadbalancer
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags:
accelerator=${ACCELERATOR_NAME},site=${CLUSTER_NAME},namespace=${NA
MESPACE} 2
      service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-path: "/lb-check"
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-protocol: "https"
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval: "10" 3
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-healthy-threshold:
"3" 4
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-unhealthy-
threshold: "3" 5
  spec:
    ports:
    - name: https
```

```
    port: 443
    protocol: TCP
    targetPort: 8443
  selector:
    app: keycloak
    app.kubernetes.io/instance: keycloak
    app.kubernetes.io/managed-by: keycloak-operator
  sessionAffinity: None
  type: LoadBalancer
EOF
```

1 $NAMESPACE 应该替换为红帽构建的 Keycloak 部署的命名空间

2 在 AWS 创建的资源中添加附加标签，以便稍后可以检索它们。ACCELERATOR_NAME 应当是后续步骤中创建的全局加速器的名称，CLUSTER_NAME 应该是当前站点的名称。

3 健康检查探测的执行频率（以秒为单位）

4 NLB 必须通过多少健康检查才被认为是健康的

5 NLB 被视为不健康的健康检查必须失败

c. 记录 DNS 主机名，因为稍后需要：

命令：

```
oc -n $NAMESPACE get svc accelerator-loadbalancer --template="{{range
.status.loadBalancer.ingress}}{{.hostname}}{{end}}"
```

输出：

```
abab80a363ce8479ea9c4349d116bce2-6b65e8b4272fa4b5.elb.eu-west-
1.amazonaws.com
```

2. 创建全局加速器实例

命令：

```
aws globalaccelerator create-accelerator \
  --name example-accelerator \ 1
  --ip-address-type DUAL_STACK \ 2
  --region us-west-2 3
```

1 要创建的加速器的名称，根据需要更新

2 可以是 'DUAL_STACK' 或 'IPV4'

3 所有 globalaccelerator 命令必须使用区域 'us-west-2'

输出：

```
{
   "Accelerator": {
      "AcceleratorArn":
"arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-391f-4e3e-9a3d-
d5ad22a78c71", (1)
      "Name": "example-accelerator",
      "IpAddressType": "DUAL_STACK",
      "Enabled": true,
      "IpSets": [
         {
            "IpFamily": "IPv4",
            "IpAddresses": [
               "75.2.42.125",
               "99.83.132.135"
            ],
            "IpAddressFamily": "IPv4"
         },
         {
            "IpFamily": "IPv6",
            "IpAddresses": [
               "2600:9000:a400:4092:88f3:82e2:e5b2:e686",
               "2600:9000:a516:b4ef:157e:4cbd:7b48:20f1"
            ],
            "IpAddressFamily": "IPv6"
         }
      ],
      "DnsName": "a099f799900e5b10d.awsglobalaccelerator.com", (2)
      "Status": "IN_PROGRESS",
      "CreatedTime": "2023-11-13T15:46:40+00:00",
      "LastModifiedTime": "2023-11-13T15:46:42+00:00",
      "DualStackDnsName": "ac86191ca5121e885.dualstack.awsglobalaccelerator.com"
(3)
   }
}
```


(1) 与创建的加速器实例关联的 ARN，这将在后续命令中使用

(2) 哪个 IPv4 红帽构建的 Keycloak 客户端应该连接到的 DNS 名称

(3) 用于 Red Hat build of Keycloak 客户端的 IPv6 红帽构建的 DNS 名称

3. 为加速器创建一个 Listener

命令：

```
aws globalaccelerator create-listener \
  --accelerator-arn 'arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-391f-4e3e-9a3d-d5ad22a78c71' \
  --port-ranges '[{"FromPort":443,"ToPort":443}]' \
  --protocol TCP \
  --region us-west-2
```

输出：

```
{
    "Listener": {
        "ListenerArn": "arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-391f-4e3e-9a3d-d5ad22a78c71/listener/1f396d40",
        "PortRanges": [
            {
                "FromPort": 443,
                "ToPort": 443
            }
        ],
        "Protocol": "TCP",
        "ClientAffinity": "NONE"
    }
}
```

4. 为 Listener 创建端点组

命令：

```
CLUSTER_1_ENDPOINT_ARN=$(aws elbv2 describe-load-balancers \
    --query "LoadBalancers[?DNSName=='abab80a363ce8479ea9c4349d116bce2-
```

```
6b65e8b4272fa4b5.elb.eu-west-1.amazonaws.com'].LoadBalancerArn" \ 1
  --region eu-west-1 \ 2
  --output text
)
CLUSTER_2_ENDPOINT_ARN=$(aws elbv2 describe-load-balancers \
  --query "LoadBalancers[?DNSName=='a1c76566e3c334e4ab7b762d9f8dcbcf-
985941f9c8d108d4.elb.eu-west-1.amazonaws.com'].LoadBalancerArn" \ 3
  --region eu-west-1 \ 4
  --output text
)
ENDPOINTS='[
 {
  "EndpointId": "${CLUSTER_1_ENDPOINT_ARN}",
  "Weight": 128,
  "ClientIPPreservationEnabled": false
 },
 {
  "EndpointId": "${CLUSTER_2_ENDPOINT_ARN}",
  "Weight": 128,
  "ClientIPPreservationEnabled": false
 }
]'
aws globalaccelerator create-endpoint-group \
 --listener-arn 'arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-391f-
4e3e-9a3d-d5ad22a78c71/listener/1f396d40' \ 5
 --traffic-dial-percentage 100 \
 --endpoint-configurations ${ENDPOINTS} \
 --endpoint-group-region eu-west-1 \ 6
 --region us-west-2
```

1 3

集群的 NLB 的 DNS 主机名

2 4 5

上一步中创建的 Listener 的 ARN

6

这应该是托管集群的 AWS 区域

输出：

-

```
{
   "EndpointGroup": {
      "EndpointGroupArn":
"arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-391f-4e3e-9a3d-
d5ad22a78c71/listener/1f396d40/endpoint-group/2581af0dc700",
      "EndpointGroupRegion": "eu-west-1",
      "EndpointDescriptions": [
         {
            "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/abab80a363ce8479ea9c4349d116bce2/6b65e8b4272f
a4b5",
            "Weight": 128,
            "HealthState": "HEALTHY",
            "ClientIPPreservationEnabled": false
         },
         {
            "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a1c76566e3c334e4ab7b762d9f8dcbcf/985941f9c8d10
8d4",
            "Weight": 128,
            "HealthState": "HEALTHY",
            "ClientIPPreservationEnabled": false
         }
      ],
      "TrafficDialPercentage": 100.0,
      "HealthCheckPort": 443,
      "HealthCheckProtocol": "TCP",
      "HealthCheckPath": "undefined",
      "HealthCheckIntervalSeconds": 30,
      "ThresholdCount": 3
   }
}
```


5. 可选：配置自定义域

   如果您使用自定义域，请通过在自定义域中配置 Alias 或 CNAME 将自定义域指向 AWS Global Load Balancer。

6. 创建或更新红帽构建的 Keycloak 部署

   在每个红帽构建的 Keycloak 集群中执行以下操作：

   a. 登录到 ROSA 集群

b.

确保 Keycloak CR 具有以下配置

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: keycloak
spec:
  hostname:
    hostname: $HOSTNAME (1)
  ingress:
    enabled: false (2)
```

(1)

用于连接到 Keycloak 的主机名客户端

(2)

禁用默认入口，因为所有红帽构建的 Keycloak 访问都应通过置备的 NLB

为确保请求转发按预期工作，Keycloak CR 需要指定要通过其访问红帽构建的 Keycloak 实例的主机名。这可以是与全局加速器关联的 DualStack DnsName 或 DnsName 主机名。如果您使用自定义域，请将自定义域指向 AWS 全局加速器，并在这里使用您的自定义域。

3.16.5. 验证

要验证 Global Accelerator 是否已正确配置为连接到集群，请导航到上述主机名，您应该会看到红帽构建的 Keycloak 管理控制台。

3.16.6. 进一步阅读

- 在线提供站点
- 使站点离线

3.17. 部署 AWS LAMBDA 以禁用非响应站点

在多集群部署中部署一个 AWS Lambda 作为负载均衡器构建块的一部分。

本章介绍了如何在多集群部署的两个站点间解决脑裂场景。它还禁用复制（如果一个站点失败），因此其他站点可以继续为请求提供服务。

此部署旨在与 多集群部署 一章中介绍的设置一起使用。将此部署与构建块 多集群部署一章中介绍的其他构建块 一起使用。

注意

我们提供这些蓝图来显示最小功能完整示例，为常规安装提供良好的基准性能。您仍然需要根据您的环境以及您的组织的标准和安全性最佳实践进行调整。

3.17.1. 架构

如果多集群部署中站点间的网络通信失败，则两个站点无法再继续在它们之间复制数据。Data Grid 配置有 FAIL 失败策略，该策略可确保可用性上的一致性。因此，所有用户请求都会显示错误消息，直到失败被解析（通过恢复网络连接或禁用跨站点复制）。

在这种情况下，仲裁通常用于确定哪些站点被标记为在线或离线。但是，因为多集群部署只能由两个站点组成，因此无法实现。相反，我们利用"隔离"来确保当某个站点无法连接到其他站点时，只有一个站点保留在负载均衡器配置中，因此只有此站点能够为后续用户请求提供服务。

除了负载均衡器配置外，隔离程序会禁用两个 Data Grid 集群之间的复制，以允许从负载均衡器配置中保留的站点提供用户请求。因此，在禁用复制后，站点将不同步。

要从不同步状态中恢复，需要手动重新同步，如 Synchroniz ing site 所述。这就是为什么在解决网络通信失败时，不会自动重新添加通过隔离删除的站点。只有在使用概述的步骤创建站点 在线时，才应重新添加删除站点。

在本章中，我们介绍了如何使用 Prometheus Alerts 和 AWS Lambda 功能的组合实现隔离。当 Data Grid 服务器指标检测到脑裂时，会触发 Prometheus Alert，这会导致 Prometheus AlertManager 调用基于 AWS Lambda 的 Webhook。触发的 Lambda 功能检查当前的全局加速器配置，并删除报告离线的站点。

在真正脑裂的情况中，两个站点仍然在线，但网络通信都可能同时触发 Webhook。我们通过确保给定时间只能执行单个 Lambda 实例来保护这一点。AWS Lambda 中的逻辑可确保始终在负载均衡器配置中保留一个站点条目。

3.17.2. 先决条件

- 基于 ROSA HCP 的多集群 Keycloak 部署
- 已安装 AWS CLI
- AWS Global Accelerator 负载均衡器
- 已安装 jq 工具

3.17.3. 流程

1. 启用 OpenShift 用户警报路由

命令：

```
oc apply -f - << EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: user-workload-monitoring-config
  namespace: openshift-user-workload-monitoring
data:
  config.yaml: |
    alertmanager:
      enabled: true
      enableAlertmanagerConfig: true
EOF
oc -n openshift-user-workload-monitoring rollout status --watch
statefulset.apps/alertmanager-user-workload
```

2. Decide 根据用户名/密码组合，用于验证 Lambda Webhook 并创建存储密码的 AWS Secret

命令：

```
aws secretsmanager create-secret \
  --name webhook-password \ 1
  --secret-string changeme \ 2
  --region eu-west-1 3
```

1 secret 的名称

2 用于身份验证的密码

3 托管 secret 的 AWS 区域

3. 创建用于执行 Lambda 的 Role。

命令：

```
FUNCTION_NAME= 1
ROLE_ARN=$(aws iam create-role \
  --role-name ${FUNCTION_NAME} \
  --assume-role-policy-document \
  '{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
```

```
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}' \
--query 'Role.Arn' \
--region eu-west-1 \ 2
--output text
)
```

1 与 Lambda 和相关资源关联的选择名称

2 托管 OpenShift 集群的 AWS 区域

4. 创建并附加 'LambdaSecretManager' 策略，以便 Lambda 可以访问 AWS Secret

命令：

```
POLICY_ARN=$(aws iam create-policy \
--policy-name LambdaSecretManager \
--policy-document \
'{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "*"
        }
    ]
}' \
--query 'Policy.Arn' \
--output text
)
```

```
aws iam attach-role-policy \
  --role-name ${FUNCTION_NAME} \
  --policy-arn ${POLICY_ARN}
```

5. 附加 ElasticLoadBalancingReadOnly 策略，以便 Lambda 可以查询置备的 Network Load Balancers

命令：

```
aws iam attach-role-policy \
  --role-name ${FUNCTION_NAME} \
  --policy-arn arn:aws:iam::aws:policy/ElasticLoadBalancingReadOnly
```

6. 附加 GlobalAcceleratorFullAccess 策略，以便 Lambda 可以更新全局加速器 EndpointGroup

命令：

```
aws iam attach-role-policy \
  --role-name ${FUNCTION_NAME} \
  --policy-arn arn:aws:iam::aws:policy/GlobalAcceleratorFullAccess
```

7. 创建包含所需的隔离逻辑的 Lambda ZIP 文件

命令：

```
LAMBDA_ZIP=/tmp/lambda.zip
```

```
cat << EOF > /tmp/lambda.py

from urllib.error import HTTPError

import boto3
import jmespath
import json
import os
import urllib3

from base64 import b64decode
from urllib.parse import unquote

# Prevent unverified HTTPS connection warning
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)


class MissingEnvironmentVariable(Exception):
    pass


class MissingSiteUrl(Exception):
    pass


def env(name):
    if name in os.environ:
        return os.environ[name]
    raise MissingEnvironmentVariable(f"Environment Variable '{name}' must be set")


def handle_site_offline(labels):
    a_client = boto3.client('globalaccelerator', region_name='us-west-2')

    acceleratorDNS = labels['accelerator']
    accelerator = jmespath.search(f"Accelerators[?(DnsName=='{acceleratorDNS}'||
DualStackDnsName=='{acceleratorDNS}')]", a_client.list_accelerators())
    if not accelerator:
        print(f"Ignoring SiteOffline alert as accelerator with DnsName '{acceleratorDNS}'
not found")
        return

    accelerator_arn = accelerator[0]['AcceleratorArn']
    listener_arn = a_client.list_listeners(AcceleratorArn=accelerator_arn)['Listeners'][0]
['ListenerArn']

    endpoint_group = a_client.list_endpoint_groups(ListenerArn=listener_arn)
['EndpointGroups'][0]
    endpoints = endpoint_group['EndpointDescriptions']

    # Only update accelerator endpoints if two entries exist
    if len(endpoints) > 1:
        # If the reporter endpoint is not healthy then do nothing for now
        # A Lambda will eventually be triggered by the other offline site for this reporter
        reporter = labels['reporter']
        reporter_endpoint = [e for e in endpoints if endpoint_belongs_to_site(e, reporter)]
```

```
[0]
        if reporter_endpoint['HealthState'] == 'UNHEALTHY':
            print(f"Ignoring SiteOffline alert as reporter '{reporter}' endpoint is marked
UNHEALTHY")
            return

        offline_site = labels['site']
        endpoints = [e for e in endpoints if not endpoint_belongs_to_site(e, offline_site)]
        del reporter_endpoint['HealthState']
        a_client.update_endpoint_group(
            EndpointGroupArn=endpoint_group['EndpointGroupArn'],
            EndpointConfigurations=endpoints
        )
        print(f"Removed site={offline_site} from Accelerator EndpointGroup")

        take_infinispan_site_offline(reporter, offline_site)
        print(f"Backup site={offline_site} caches taken offline")
    else:
        print("Ignoring SiteOffline alert only one Endpoint defined in the EndpointGroup")


def endpoint_belongs_to_site(endpoint, site):
    lb_arn = endpoint['EndpointId']
    region = lb_arn.split(':')[3]
    client = boto3.client('elbv2', region_name=region)
    tags = client.describe_tags(ResourceArns=[lb_arn])['TagDescriptions'][0]['Tags']
    for tag in tags:
        if tag['Key'] == 'site':
            return tag['Value'] == site
    return false


def take_infinispan_site_offline(reporter, offlinesite):
    endpoints = json.loads(INFINISPAN_SITE_ENDPOINTS)
    if reporter not in endpoints:
        raise MissingSiteUrl(f"Missing URL for site '{reporter}' in
'INFINISPAN_SITE_ENDPOINTS' json")

    endpoint = endpoints[reporter]
    password = get_secret(INFINISPAN_USER_SECRET)
    url = f"https://{endpoint}/rest/v2/container/x-site/backups/{offlinesite}?action=take-
offline"
    http = urllib3.PoolManager(cert_reqs='CERT_NONE')
    headers = urllib3.make_headers(basic_auth=f"{INFINISPAN_USER}:{password}")
    try:
        rsp = http.request("POST", url, headers=headers)
        if rsp.status >= 400:
            raise HTTPError(f"Unexpected response status '%d' when taking site offline",
rsp.status)
        rsp.release_conn()
    except HTTPError as e:
        print(f"HTTP error encountered: {e}")


def get_secret(secret_name):
    session = boto3.session.Session()
```

```
    client = session.client(
        service_name='secretsmanager',
        region_name=SECRETS_REGION
    )
    return client.get_secret_value(SecretId=secret_name)['SecretString']


def decode_basic_auth_header(encoded_str):
    split = encoded_str.strip().split(' ')
    if len(split) == 2:
        if split[0].strip().lower() == 'basic':
            try:
                username, password = b64decode(split[1]).decode().split(':', 1)
            except:
                raise DecodeError
        else:
            raise DecodeError
    else:
        raise DecodeError

    return unquote(username), unquote(password)


def handler(event, context):
    print(json.dumps(event))

    authorization = event['headers'].get('authorization')
    if authorization is None:
        print("'Authorization' header missing from request")
        return {
            "statusCode": 401
        }

    expectedPass = get_secret(WEBHOOK_USER_SECRET)
    username, password = decode_basic_auth_header(authorization)
    if username != WEBHOOK_USER and password != expectedPass:
        print('Invalid username/password combination')
        return {
            "statusCode": 403
        }

    body = event.get('body')
    if body is None:
        raise Exception('Empty request body')

    body = json.loads(body)
    print(json.dumps(body))

    if body['status'] != 'firing':
        print("Ignoring alert as status is not 'firing', status was: '%s'" % body['status'])
        return {
            "statusCode": 204
        }

    for alert in body['alerts']:
        labels = alert['labels']
```

```
        if labels['alertname'] == 'SiteOffline':
            handle_site_offline(labels)

    return {
        "statusCode": 204
    }


INFINISPAN_USER = env('INFINISPAN_USER')
INFINISPAN_USER_SECRET = env('INFINISPAN_USER_SECRET')
INFINISPAN_SITE_ENDPOINTS = env('INFINISPAN_SITE_ENDPOINTS')
SECRETS_REGION = env('SECRETS_REGION')
WEBHOOK_USER = env('WEBHOOK_USER')
WEBHOOK_USER_SECRET = env('WEBHOOK_USER_SECRET')

EOF
zip -FS --junk-paths ${LAMBDA_ZIP} /tmp/lambda.py
```

8. 创建 Lambda 功能。

命令：

```
aws lambda create-function \
  --function-name ${FUNCTION_NAME} \
  --zip-file fileb://${LAMBDA_ZIP} \
  --handler lambda.handler \
  --runtime python3.12 \
  --role ${ROLE_ARN} \
  --region eu-west-1 (1)
```

(1) 托管 OpenShift 集群的 AWS 区域

9. 公开功能 URL，以便 Lambda 可以作为 Webhook 触发

命令：

```
aws lambda create-function-url-config \
  --function-name ${FUNCTION_NAME} \
  --auth-type NONE \
  --region eu-west-1 1
```

1

托管 OpenShift 集群的 AWS 区域

10. 允许对功能 URL 进行公共调用

命令：

```
aws lambda add-permission \
  --action "lambda:InvokeFunctionUrl" \
  --function-name ${FUNCTION_NAME} \
  --principal "*" \
  --statement-id FunctionURLAllowPublicAccess \
  --function-url-auth-type NONE \
  --region eu-west-1 1
```

1

托管 OpenShift 集群的 AWS 区域

11. 配置 Lambda 的环境变量：

a. 在每个 OpenShift 集群中，检索公开的 Data Grid URL 端点：

```
oc -n ${NAMESPACE} get route infinispan-external -o
jsonpath='{.status.ingress[].host}' 1
```

1

将 ${NAMESPACE} 替换为包含 Data Grid 服务器的命名空间

b.

上传所需的环境变量

```
ACCELERATOR_NAME= 1
LAMBDA_REGION= 2
CLUSTER_1_NAME= 3
CLUSTER_1_ISPN_ENDPOINT= 4
CLUSTER_2_NAME= 5
CLUSTER_2_ISPN_ENDPOINT= 6
INFINISPAN_USER= 7
INFINISPAN_USER_SECRET= 8
WEBHOOK_USER= 9
WEBHOOK_USER_SECRET= 10

INFINISPAN_SITE_ENDPOINTS=$(echo "
{\"${CLUSTER_NAME_1}\":\"${CLUSTER_1_ISPN_ENDPOINT}\",\"${CLUSTER_2_N
AME}\":\"${CLUSTER_2_ISPN_ENDPOINT\"}" | jq tostring)
aws lambda update-function-configuration \
  --function-name ${ACCELERATOR_NAME} \
  --region ${LAMBDA_REGION} \
  --environment "{
   \"Variables\": {
     \"INFINISPAN_USER\" : \"${INFINISPAN_USER}\",
     \"INFINISPAN_USER_SECRET\" : \"${INFINISPAN_USER_SECRET}\",
     \"INFINISPAN_SITE_ENDPOINTS\" : ${INFINISPAN_SITE_ENDPOINTS},
     \"WEBHOOK_USER\" : \"${WEBHOOK_USER}\",
     \"WEBHOOK_USER_SECRET\" : \"${WEBHOOK_USER_SECERT}\",
     \"SECRETS_REGION\" : \"eu-central-1\"
   }
  }"
```

1

您的部署使用的 AWS Global Accelerator 的名称

2

托管 OpenShift 集群的 AWS 区域和 Lambda 功能

3

使用 Data Grid Operator 部署 HA 中定义的 Data Grid站点的名称

4

与 CLUSER_1_NAME 站点关联的 Data Grid 端点 URL

5

第二个 Data Grid 站点的名称

6

与 CLUSER_2_NAME 站点关联的 Data Grid 端点 URL

7

Data Grid 用户的用户名，该用户有足够的特权在服务器上执行 REST 请求

8

包含与 Data Grid 用户关联的密码的 AWS secret 名称

9

用于验证对 Lambda Function 的请求的用户名

10

包含用于验证向 Lambda 功能请求的密码的 AWS secret 名称

12. 检索 Lambda Function URL

命令：

```
aws lambda get-function-url-config \
  --function-name ${FUNCTION_NAME} \
  --query "FunctionUrl" \
  --region eu-west-1 \ 1
  --output text
```

1 创建 Lambda 的 AWS 区域

输出：

```
https://tjqr2vgc664b6noj6vugprakoq0oausj.lambda-url.eu-west-1.on.aws
```

13. 在每个 OpenShift 集群中，配置 Prometheus Alert 路由，以便在脑裂时触发 Lambda

命令：

```
NAMESPACE= # The namespace containing your deployments
oc apply -n ${NAMESPACE} -f - << EOF
apiVersion: v1
kind: Secret
type: kubernetes.io/basic-auth
metadata:
  name: webhook-credentials
stringData:
  username: 'keycloak' 1
  password: 'changme' 2
---
apiVersion: monitoring.coreos.com/v1beta1
kind: AlertmanagerConfig
metadata:
  name: example-routing
spec:
  route:
    receiver: default
    groupBy:
      - accelerator
    groupInterval: 90s
    groupWait: 60s
    matchers:
      - matchType: =
        name: alertname
```

```
      value: SiteOffline
  receivers:
    - name: default
      webhookConfigs:
        - url: 'https://tjqr2vgc664b6noj6vugprakoq0oausj.lambda-url.eu-west-1.on.aws/'
3
          httpConfig:
            basicAuth:
              username:
                key: username
                name: webhook-credentials
              password:
                key: password
                name: webhook-credentials
            tlsConfig:
              insecureSkipVerify: true
---
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: xsite-status
spec:
  groups:
    - name: xsite-status
      rules:
        - alert: SiteOffline
          expr: 'min by (namespace, site)
(vendor_jgroups_site_view_status{namespace="default",site="site-b"}) == 0' 4
          labels:
            severity: critical
            reporter: site-a 5
            accelerator: a3da6a6cbd4e27b02.awsglobalaccelerator.com 6
```

1

验证 Lambda 请求所需的用户名

2

验证 Lambda 请求所需的密码

3

Lambda Function URL

4

5

在 Infinispan CR 中由 spec.service.sites.local.name 定义的本地站点的名称

6

全球加速器的 DNS

3.17.4. 验证

要测试 Prometheus 警报是否会如预期触发 Webhook，请执行以下步骤来模拟脑裂：

1. 在每个集群中执行以下操作：

命令：

```
oc -n openshift-operators scale --replicas=0 deployment/infinispan-operator-controller-manager 1
oc -n openshift-operators rollout status -w deployment/infinispan-operator-controller-manager
oc -n ${NAMESPACE} scale --replicas=0 deployment/infinispan-router 2
oc -n ${NAMESPACE} rollout status -w deployment/infinispan-router
```

1

缩减 Data Grid Operator，以便下一步不会导致操作器重新创建部署

2

使用包含 Data Grid 服务器的命名空间缩减 Gossip Router deployment.Replace ${NAMESPACE}

2. 通过检查 OpenShift 控制台中的 Observe → Alerting 菜单来验证集群中是否已触发 SiteOffline 事件

3. 检查 AWS 控制台中的 Global Accelerator EndpointGroup，且只有一个端点

4. 扩展 Data Grid Operator 和 Gossip Router，以在站点间重新建立连接：

命令：

```
oc -n openshift-operators scale --replicas=1 deployment/infinispan-operator-controller-manager
oc -n openshift-operators rollout status -w deployment/infinispan-operator-controller-manager
oc -n ${NAMESPACE} scale --replicas=1 deployment/infinispan-router (1)
oc -n ${NAMESPACE} rollout status -w deployment/infinispan-router
```

(1) 将 ${NAMESPACE} 替换为包含 Data Grid 服务器的命名空间

5. 检查每个站点中的 vendor_jgroups_site_view_status 指标。值 1 表示站点可以访问。

6. 更新加速器 EndpointGroup，使其包含两个端点。详情请参阅 在线品牌品牌 章节。

3.17.5. 进一步阅读

- 在线提供站点
- 使站点离线

3.18. 使站点离线

使站点离线，使其不再处理客户端请求。

3.18.1. 何时使用此流程

在部署生命周期中，可能需要其中一个站点暂时离线进行维护，或允许软件升级。为确保没有用户请求路由到需要维护的站点，需要从负载均衡器配置中删除站点。

3.18.2. 流程

按照以下步骤从负载均衡器中删除站点，以便无法路由到它。

3.18.2.1. 全局加速器

1. 确定与要在线站点关联的 Network Load Balancer (NLB)的 ARN

命令：

```
NAMESPACE= 1
REGION= 2
HOSTNAME=$(oc -n $NAMESPACE get svc accelerator-loadbalancer --template='
{{range .status.loadBalancer.ingress}}{{.hostname}}{{end}}')
aws elbv2 describe-load-balancers \
  --query "LoadBalancers[?DNSName=='${HOSTNAME}'].LoadBalancerArn" \
  --region ${REGION} \
  --output text
```

1 包含 Keycloak 部署的 OpenShift 命名空间

2 托管 OpenShift 集群的 AWS 区域

输出：

```
arn:aws:elasticloadbalancing:eu-west-1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80ed4eba3d
```

2. 更新加速器 EndpointGroup，使其只包含单个集群

   a. 列出 Global Accelerator 的 EndpointGroup 中的当前端点

   命令：

```
ACCELERATOR_NAME= (1)
ACCELERATOR_ARN=$(aws globalaccelerator list-accelerators \
  --query "Accelerators[?Name=='${ACCELERATOR_NAME}'].AcceleratorArn" \
  --region us-west-2 \ (2)
  --output text
)
LISTENER_ARN=$(aws globalaccelerator list-listeners \
  --accelerator-arn ${ACCELERATOR_ARN} \
  --query "Listeners[*].ListenerArn" \
  --region us-west-2 \
  --output text
)
aws globalaccelerator list-endpoint-groups \
  --listener-arn ${LISTENER_ARN} \
  --region us-west-2
```

(1) 要更新的加速器的名称

(2) 查询 AWS 全局加速器时，区域必须始终设置为 us-west-2

输出：


```
{
   "EndpointGroups": [
      {
         "EndpointGroupArn":
"arn:aws:globalaccelerator::606671647913:accelerator/d280fc09-3057-4ab6-9330-
6cbf1f450748/listener/8769072f/endpoint-group/a30b64ec1700",
         "EndpointGroupRegion": "eu-west-1",
         "EndpointDescriptions": [
            {
               "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80e
d4eba3d",
               "Weight": 128,
               "HealthState": "HEALTHY",
               "ClientIPPreservationEnabled": false
            },
            {
               "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a3c75f239541c4a6e9c48cf8d48d602f/5ba333e870
19ccf0",
               "Weight": 128,
               "HealthState": "HEALTHY",
               "ClientIPPreservationEnabled": false
            }
         ],
         "TrafficDialPercentage": 100.0,
         "HealthCheckPort": 443,
         "HealthCheckProtocol": "TCP",
         "HealthCheckIntervalSeconds": 30,
         "ThresholdCount": 3
      }
   ]
}
```


b. 更新 EndpointGroup，使其只包含在第 1 步中获取的 NLB。

命令：

```
aws globalaccelerator update-endpoint-group \
  --endpoint-group-arn
arn:aws:globalaccelerator::606671647913:accelerator/d280fc09-3057-4ab6-9330-
```

```
6cbf1f450748/listener/8769072f/endpoint-group/a30b64ec1700 \
 --region us-west-2 \
 --endpoint-configurations '
 [
  {
     "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80e
d4eba3d",
     "Weight": 128,
     "ClientIPPreservationEnabled": false
  }
 ]
'
```

3.19. 在线提供站点

使站点上线，以便它可以处理客户端请求。

3.19.1. 何时使用此流程

此流程描述了如何在之前离线后将 Keycloak 站点重新添加到全局加速器，以便它可以在服务客户端请求后再次添加。

3.19.2. 流程

按照以下步骤将 Keycloak 站点重新添加到 AWS 全局加速器，以便它可以处理客户端请求。

3.19.2.1. 全局加速器

1. 确定与要上线的站点关联的 Network Load Balancer (NLB)的 ARN

命令：

```
NAMESPACE= 1
REGION= 2
HOSTNAME=$(oc -n $NAMESPACE get svc accelerator-loadbalancer --template="
{{range .status.loadBalancer.ingress}}{{.hostname}}{{end}}")
```

```
aws elbv2 describe-load-balancers \
  --query "LoadBalancers[?DNSName=='${HOSTNAME}'].LoadBalancerArn" \
  --region ${REGION} \
  --output text
```

1. 包含 Keycloak 部署的 OpenShift 命名空间

2. 托管 OpenShift 集群的 AWS 区域

输出：

```
arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80ed4e
ba3d
```

2. 更新加速器 EndpointGroup，使其包含两个站点

   a. 列出 Global Accelerator 的 EndpointGroup 中的当前端点

   命令：

```
ACCELERATOR_NAME= 1
ACCELERATOR_ARN=$(aws globalaccelerator list-accelerators \
  --query "Accelerators[?Name=='${ACCELERATOR_NAME}'].AcceleratorArn" \
  --region us-west-2 \ 2
  --output text
)
LISTENER_ARN=$(aws globalaccelerator list-listeners \
  --accelerator-arn ${ACCELERATOR_ARN} \
  --query "Listeners[*].ListenerArn" \
```

```
  --region us-west-2 \
  --output text
)
aws globalaccelerator list-endpoint-groups \
  --listener-arn ${LISTENER_ARN} \
  --region us-west-2
```

1 要更新的加速器的名称

2 查询 AWS 全局加速器时，区域必须始终设置为 us-west-2

输出：

```
{
   "EndpointGroups": [
      {
         "EndpointGroupArn":
"arn:aws:globalaccelerator::606671647913:accelerator/d280fc09-3057-4ab6-9330-
6cbf1f450748/listener/8769072f/endpoint-group/a30b64ec1700",
         "EndpointGroupRegion": "eu-west-1",
         "EndpointDescriptions": [
            {
               "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a3c75f239541c4a6e9c48cf8d48d602f/5ba333e870
19ccf0",
               "Weight": 128,
               "HealthState": "HEALTHY",
               "ClientIPPreservationEnabled": false
            }
         ],
         "TrafficDialPercentage": 100.0,
         "HealthCheckPort": 443,
         "HealthCheckProtocol": "TCP",
         "HealthCheckIntervalSeconds": 30,
         "ThresholdCount": 3
      }
   ]
}
```

b.

更新 EndpointGroup，使其包含现有的 Endpoint 和第 1 步中获取的 NLB。

命令：

```
aws globalaccelerator update-endpoint-group \
  --endpoint-group-arn
arn:aws:globalaccelerator::606671647913:accelerator/d280fc09-3057-4ab6-9330-
6cbf1f450748/listener/8769072f/endpoint-group/a30b64ec1700 \
  --region us-west-2 \
  --endpoint-configurations '
  [
    {
      "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a3c75f239541c4a6e9c48cf8d48d602f/5ba333e870
19ccf0",
      "Weight": 128,
      "ClientIPPreservationEnabled": false
    },
    {
      "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80e
d4eba3d",
      "Weight": 128,
      "ClientIPPreservationEnabled": false
    }
  ]
'
```

3.20. 同步站点

将离线站点与在线站点同步。

3.20.1. 何时使用此流程

当两个站点的 Data Grid 集群的状态断开连接且缓存的内容不同步时，请使用此选项。例如，在脑裂或一个站点离线进行维护后执行此操作。

在流程结束时，次要站点中的数据已被丢弃，并替换为活动站点的数据。离线站点中的所有缓存都会被清除，以防止无效的缓存内容。

3.20.2. 流程

3.20.2.1. Data Grid 集群

对于本章的上下文中，site-a 是当前活跃的站点，site-b 是不是 AWS 全局加速器 EndpointGroup 的一部分的离线站点，因此不接收用户请求。

警告

通过增加响应时间和/或资源使用量，传输状态可能会影响 Data Grid 集群性能。

第一步是从离线站点中删除过时的数据。

1. 登录到离线站点。

2. 关闭红帽构建的 Keycloak。这将清除所有红帽构建的 Keycloak 缓存，并防止红帽构建的 Keycloak 状态与 Data Grid 不兼容。

   当使用红帽构建的 Keycloak Operator 部署红帽构建的 Keycloak 时，请将红帽构建的 Keycloak 实例中的红帽构建的 Keycloak 实例数量改为 0。

3. 使用 Data Grid CLI 工具连接到 Data Grid 集群：

   命令：

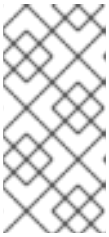
```
oc -n keycloak exec -it pods/infinispan-0 -- ./bin/cli.sh --trustall --connect
https://127.0.0.1:11222
```

-

它要求输入 Data Grid 集群的用户名和密码。这些凭据是在配置凭证部分中的 使用 Data Grid Operator 的 Deploying Data Grid for HA 一章中的设置。

输出：

```
Username: developer
Password:
[infinispan-0-29897@ISPN//containers/default]>
```



注意

pod 名称取决于 Data Grid CR 中定义的集群名称。连接可通过 Data Grid 集群中的任何 pod 完成。

4. 运行以下命令，将复制从离线站点禁用到活动站点。它可防止访问活动站点的清除请求并删除所有正确的缓存数据。

命令：

```
site take-offline --all-caches --site=site-a
```

输出：

```
{
  "authenticationSessions" : "ok",
```

```
  "work" : "ok",
  "loginFailures" : "ok",
  "actionTokens" : "ok"
}
```

5. 检查**复制状**态是否为 。

命令：

```
site status --all-caches --site=site-a
```

输出：

```
{
  "status" : "offline"
}
```

如果状态不是 离线，请重复上一步。

警告

确保复制 离线，否则清除数据将清除两个站点。

6. 使用以下命令清除离线站点中的所有缓存数据：

命令：

```
clearcache actionTokens
clearcache authenticationSessions
clearcache loginFailures
clearcache work
```

这些命令不会打印任何输出。

7. 重新启用从离线站点到活动站点的跨站点复制。

命令：

```
site bring-online --all-caches --site=site-a
```

输出：

```
{
  "authenticationSessions" : "ok",
  "work" : "ok",
  "loginFailures" : "ok",
  "actionTokens" : "ok"
}
```

8. 检查复制状态为 在线。

命令：

```
site status --all-caches --site=site-a
```

输出：

```
{
  "status" : "online"
}
```

现在，我们已准备好将状态从活动站点转移到离线站点。

1. 登录到您的活跃站点

2. 使用 Data Grid CLI 工具连接到 Data Grid 集群：

命令：

```
oc -n keycloak exec -it pods/infinispan-0 -- ./bin/cli.sh --trustall --connect
https://127.0.0.1:11222
```

它要求输入 Data Grid 集群的用户名和密码。这些凭据是在配置凭证部分中的 使用 Data

Grid Operator 的 Deploying Data Grid for HA 一章中的设置。

输出：

```
Username: developer
Password:
[infinispan-0-29897@ISPN//containers/default]>
```

注意

pod 名称取决于 Data Grid CR 中定义的集群名称。连接可通过 Data Grid 集群中的任何 pod 完成。

3. 触发从活动站点到离线站点的状态转移。

命令：

```
site push-site-state --all-caches --site=site-b
```

输出：

```
{
  "authenticationSessions" : "ok",
  "work" : "ok",
  "loginFailures" : "ok",
  "actionTokens" : "ok"
}
```

4. 检查所有缓存的复制状态是 在线的。

命令：

```
site status --all-caches --site=site-b
```

输出：

```
{
  "status" : "online"
}
```

5. 通过检查所有缓存的 push-site-status 命令的输出，等待状态传输完成。

命令：

```
site push-site-status --cache=actionTokens
site push-site-status --cache=authenticationSessions
site push-site-status --cache=loginFailures
site push-site-status --cache=work
```

输出：

```
{
  "site-b" : "OK"
}
```

```
{
  "site-b" : "OK"
}
{
  "site-b" : "OK"
}
{
  "site-b" : "OK"
}
```

检查本节中的表，了解 Cross-Site Documentation 中的可能状态值。

如果报告错误，请对该特定缓存重复状态传输。

命令：

```
site push-site-state --cache=<cache-name> --site=site-b
```

6. 使用以下命令清除/重置状态

命令：

```
site clear-push-site-status --cache=actionTokens
site clear-push-site-status --cache=authenticationSessions
site clear-push-site-status --cache=loginFailures
site clear-push-site-status --cache=work
```

输出：

```
"ok"
"ok"
"ok"
"ok"
```

现在，这个状态包括在离线站点中，可以再次启动 Red Hat build of Keycloak:

1. 登录到您的次要站点。

2. 启动红帽构建的 Keycloak。

当使用红帽构建的 Keycloak Operator 部署红帽构建的 Keycloak 时，将红帽构建的 Keycloak 实例中的红帽构建的 Keycloak 实例数量改为原始值。

3.20.2.2. AWS Aurora 数据库

不需要任何操作。

3.20.2.3. AWS Global Accelerator

同步两个站点后，可以安全地按照 Bring a site online 章节中的步骤将之前离线站点重新添加到 Global Accelerator EndpointGroup 中。

3.20.3. 进一步阅读

请参阅 概念来自动执行 Data Grid CLI 命令。

3.21. 多集群部署的健康检查

验证多集群部署的健康状态。

在 OpenShift 环境中运行多集群部署时，您应该自动检查所有内容是否已启动并按预期运行。#multi-

cluster-introduction

本页提供了可用于验证红帽构建的 Keycloak 的多集群设置的 URL、OpenShift 资源和 Healthcheck 端点的概述。

3.21.1. 概述

主动监控策略旨在检测和警告问题，然后再对用户造成影响。这个策略是红帽构建的 Keycloak 应用程序的关键。

跨各种架构组件的健康检查（如应用程序健康、负载均衡、缓存和整体系统状态）非常重要：

确保高可用性

验证所有站点和负载平衡器是否正常运行是确保系统可以处理请求的关键，即使一个站点停机也是如此。

维护性能

检查 Data Grid 缓存的健康状态和发布，确保红帽构建的 Keycloak 可以通过高效地处理会话和其他临时数据来保持最佳性能。

操作弹性

通过持续监控红帽构建的 Keycloak 及其依赖项在 OpenShift 环境中的健康状况，系统可以快速识别并可能自动修复的问题，从而减少停机时间。

3.21.2. 先决条件

1. 已安装并配置了 kubectl CLI。
2. 如果还没有在您的操作系统中安装 jq。

3.21.3. 特定的健康检查

3.21.3.1. 红帽构建的 Keycloak 负载均衡器和站点

通过负载均衡器和主和备份站点验证红帽构建的 Keycloak 应用程序的健康状况。这样可确保红帽构建的 Keycloak 可访问，且负载平衡机制在不同地理位置或网络位置正常工作。

此命令返回红帽构建的 Keycloak 应用程序连接到其配置数据库的健康状态，从而确认数据库连接的可靠性。此命令仅适用于管理端口，而不可用于外部 URL。在 OpenShift 设置中，会定期检查子状态 健康/就绪状态 以使 Pod 就绪。

```
curl -s https://keycloak:managementport/health
```

此命令验证负载均衡器的 lb-check 端点，并确保红帽构建的 Keycloak 应用程序集群已启动并在运行。

```
curl -s https://keycloak-load-balancer-url/lb-check
```

这些命令将在多集群设置中返回红帽构建的 Keycloak 的 Site A 和 Site B 的运行状态。

```
curl -s https://keycloak_site_a_url/lb-check
curl -s https://keycloak_site_b_url/lb-check
```

3.21.3.2. Data Grid Cache 健康状况

检查外部 Data Grid 集群中默认缓存管理器和单个缓存的健康状态。此检查对于红帽构建的 Keycloak 性能和可靠性至关重要，因为 Data Grid 通常用于红帽构建的 Keycloak 部署中的分布式缓存和会话集群。

此命令返回 Data Grid 缓存管理器的整体健康状况，这很有用，因为 Admin 用户不需要提供用户凭证来获取健康状态。

```
curl -s https://infinispan_rest_url/rest/v2/cache-managers/default/health/status
```

与前面的健康检查不同，以下健康检查需要 Admin 用户提供 Data Grid 用户凭据，作为请求的一部分，以便了解外部 Data Grid 集群缓存的整体健康状况。

```
curl -u <infinispan_user>:<infinispan_pwd> -s https://infinispan_rest_url/rest/v2/cache-managers/default/health \
 | jq 'if .cluster_health.health_status == "HEALTHY" and (all(.cache_health[].status; . == "HEALTHY")) then "HEALTHY" else "UNHEALTHY" end'
```

jq 过滤器是根据单个缓存健康状况计算整体健康状况的一个方便。您还可以选择在没有 jq 过滤器的情况下运行上述命令来查看完整详情。

3.21.3.3. Data Grid 集群分发

评估 Data Grid 集群的分布健康状况，确保集群节点正确分布数据。此步骤对于缓存层的可扩展性和容错至关重要。

您可以修改 expectedCount 3 参数，以匹配集群中的节点总数，并验证它们是否正常运行。

```
curl <infinispan_user>:<infinispan_pwd> -s https://infinispan_rest_url/rest/v2/cluster\?
action\=distribution \
 | jq --argjson expectedCount 3 'if map(select(.node_addresses | length > 0)) | length ==
$expectedCount then "HEALTHY" else "UNHEALTHY" end'
```

3.21.3.4. 总体，Data Grid 系统健康状况

使用 oc CLI 工具查询 Data Grid 集群的健康状况，以及指定命名空间中的红帽构建的 Keycloak 服务。此综合检查可确保红帽构建的 Keycloak 部署的所有组件都正常运行，并在 OpenShift 环境中正确配置。

```
oc get infinispan -n <NAMESPACE> -o json  \
| jq '.items[].status.conditions' \
| jq 'map({(.type): .status})' \
| jq 'reduce .[] as $item ([]; . + [keys[] | select($item[.] != "True")]) | if length == 0 then
"HEALTHY" else "UNHEALTHY: " + (join(", ")) end'
```

3.21.3.5. Red Hat build of Keycloak 在 OpenShift 中就绪

具体来说，检查 OpenShift 中红帽构建的 Keycloak 部署的就绪度和滚动更新条件，确保红帽构建的 Keycloak 实例完全运行，且不会执行可能会影响可用性的更新。

```
oc wait --for=condition=Ready --timeout=10s keycloaks.k8s.keycloak.org/keycloak -n
<NAMESPACE>
oc wait --for=condition=RollingUpdate=False --timeout=10s
keycloaks.k8s.keycloak.org/keycloak -n <NAMESPACE>
```