性能调优通常是个持续和权衡的过程，某些调优需要根据自身硬件条件和实际需求，选择合适的配置。本章节从两个层面进行说明。

1. 主机层面优化

（1） 操作系统

默认在 OpenShift 安装完成之后，已经启用了 Tuned 服务进行 OpenShft 节点操作系统优化。Tuned 是 Red Hat Enterprise Linux 和其他红帽产品中默认启用的调优配置文件交付机制。Tuned 可以自定义一些 Linux 配置，例如 sysctls、电源管理和内核命令行选项，以针对不同的工作负载和可伸缩性要求优化操作系统，关于这个服务的更多细节可以查阅网络资料学习。

OpenShift 默认提供了 openshift-control-plane 和 openshift-node 两个 Tuned 配置文件，分别用于控制节点和计算节点。

在节点上执行以下命令查看当前使用的 Tuned 配置文件，以 Master 节点为例。

tuned-adm active

Current active profile: openshift-control-plane

所有调优的配置文件存放在每个节点的/etc/tuned 目录下，包含的内容如下：

cd /etc/tuned/

ls

active_profile openshift openshift-node recommend.conf tuned-main.conf

bootcmdline openshift-control-plane profile_mode recommend.d

可以看到 OpenShift 激活的调优配置文件保存在 openshift-node 和 openshift-control-plane 目录下。每个目录下有一个 tuned.conf 文件，该配置文件安全地增加了一些内核中常见的一些调优参数，以 openshift-control-plane 为例，内容如下：

```
[main]
summary=Optimize systems running OpenShift control plane
include=openshift


[sysctl]
```

```
   # ktune sysctl settings, maximizing i/o throughput
   #
   # Minimal preemption granularity for CPU-bound tasks:
   # (default: 1 msec#  (1 + ilog(ncpus)), units: nanoseconds)
   kernel.sched_min_granularity_ns=10000000


   # The total time the scheduler will consider a migrated process
   # "cache hot" and thus less likely to be re-migrated
   # (system default is 500000, i.e. 0.5 ms)
   kernel.sched_migration_cost_ns=5000000


   # SCHED_OTHER wake-up granularity.
   # Preemption granularity when tasks wake up.  Lower the value to
improve
   # wake-up latency and throughput for latency critical tasks.
   kernel.sched_wakeup_granularity_ns = 4000000
```

可以看到，在 main 中包含了名为 openshift 的 Tuned 配置文件，同时在 sysctl 中增加了三个额外的内核参数。openshift 的 Tuned 配置文件/etc/tuned/openshift/tuned.conf 的内容就不再介绍了。

除了自动配置的 Tuned 调优之外，如果想要进一步优化，通常还可以调节以下内容：

- ulimit 中的最大打开文件数、最大进程数等。
- sysctl 调优参数：

net.ipv4.tcp_max_tw_buckets = 60000

net.ipv4.tcp_timestamps = 0

net.ipv4.ip_local_port_range = 1024 65000

net.ipv4.tcp_tw_recycle = 1

net.ipv4.tcp_tw_reuse = 1

net.ipv4.tcp_syncookies = 1

net.core.netdev_max_backlog = 262144

net.ipv4.tcp_max_orphans = 262144

net.ipv4.tcp_max_syn_backlog = 262144

net.ipv4.tcp_synack_retries = 1

net.ipv4.tcp_syn_retries = 1

net.ipv4.tcp_fin_timeout = 1

net.ipv4.tcp_keepalive_time = 30

net.netfilter.nf_conntrack_max = 1048576（tuned 已经包含）

net.netfilter.nf_conntrack_tcp_timeout_time_wait = 30

vm.overcommit_memory=1

vm.panic_on_oom=0

由于篇幅有限，列出的内核调优参数的具体含义读者可以查阅网络资料获取。鉴于内核版本及操作系统环境的差异性，列出的调优参数仅供参考，如果要应用到生产环境中，必须在开发测试环境中自行验证。

（2） OpenShift 节点

1）Master 节点

在 OpenShift 集群中，除了 Pod 间的网络通信外，最大的开销就是 Master 节点和 ETCD 间的通信了，Master 节点通过与 ETCD 交互更新获取节点状态、网络配置、Secrets 和卷挂载等等信息。

Master 节点主要优化点包括：

- Master 和 ETCD 尽量部署在一起，修改 Master 配置文件中关于 ETCD 集群节点的列表，使得每个 Master 节点的第一个 ETCD 节点是本机节点，这样默认所有 Master 节点都连接本机的 ETCD，减少网络上消耗和延迟。这是由于 Master 默认会选择连接 ETCD 列表中第一个节点的机制决定的，除非，Master 配置文件中的 ETCD 节点列表的第一个实例不可用，才会连接后续节点。
- 在高可用集群中，尽量将 Master 节点部署在低延迟的网络里。
- Master 节点反序列化缓存：Master 使用缓存反序列化的资源以降低 CPU 负载。但是，在小于 1000 个 Pod 的小规模集群中，降低的 CPU 负载可以忽略不计，这样导致这部分缓存浪费了大量内存。默认缓存大小为 50,000 个条目，根据资源的大

小可以增加占用 1 到 2 GB 的内存。建议根据集群 Pod 的规模增加或减少此缓存的大小，通过修改 Master 配置文件中的参数实现。配置参数如下：

```
kubernetesMasterConfig:
  apiServerArguments:
    deserialization-cache-size:
    - "1000"
```

- 资源配额同步时间：在设置资源配额后，创建和删除资源都需要同步信息，可以设置同步的时间，默认 10s，修改 Master 配置文件实现，配置参数如下：

```
kubernetesMasterConfig:
  controllerArguments:
    resource-quota-sync-period:
      - "10s"
```

- 清理无用的历史对象：通过命令行清理无用的资源对象，支持 groups、builds、deployments、和 images，通过 oc adm prune 实现。
- 增加 API QPS 限制：对于大规模集群，Master API 可能会超过默认的 QPS 限制。如果 Master 节点有足够的 CPU 和内存资源, 则可以将默认的 QPS 增大或翻倍，修改 Master 配置文件实现，配置参数如下：

```
masterClients:
  externalKubernetesClientConnectionOverrides:
    burst: 1600
    qps: 800
  openshiftLoopbackClientConnectionOverrides:
    burst: 2400
    qps: 1200
```

- Pod 迁移参数：节点在出现故障之后，Pod 会在一定时间内（默认 5 分钟）迁移，可以通过参数配置迁移的时间。修改 Master 配置文件实现，配置参数如下：

```
controllerArgments:
  node-monitor-grace-period:
  - “10s”
 pod-eviction-timeout:
```

- “10s”

Pod 迁移的时间是这两个参数时间的总和。node-monitor-grace-period 必须配置为 5 的倍数，极限情况下，pod-eviction-timeout 可配置为 0s。

2）Etcd 节点

Etcd 节点配置默认不需要修改，Etcd 节点优化点在于资源和节点数，主要包括：

- CPU：Etcd 通常不需要大量的 CPU，通常 2 到 4 个核心就能顺利运行。对于负载超高的 Etcd，如每秒有数千个客户端或数万个请求，这时候往往会受 CPU 限制，此时才会考虑将 CPU 扩容 8 到 16 个 CPU 核心。
- 内存：Etcd 对内存占用量也相对较小，但其性能仍然取决于是否有足够的内存。Etcd 会尽可能的缓存键值数据，这时内存主要的消耗，通常 8GB 就足够了。对于有数千个客户端和数百万个键的大规模集群，需要相应地分配 16GB 到 64GB 内存。
- 网络：Etcd 集群节点内部通信依赖于快速可靠的网络。Etcd 选举和保持数据一致性都需要低延迟可靠的网络，低延迟确保 Etcd 成员可以快速通信，高带宽可以减少恢复失败的 Etcd 成员的时间。通常 1GbE 带宽是足够的。对于大规模 Etcd 集群，10GbE 网络将缩短平均恢复时间。关于延迟，默认集群节点间的心跳为 100ms，最好将集群部署在单个数据中心中，如果多数据中心部署，选择距离近延迟的数据中心，而且要调整 Etcd 一些关于时间的参数。
- 磁盘：磁盘性能是 Etcd 集群性能和稳定性最关键的因素。低速磁盘会增加 Etcd 请求延迟并可能损害群集稳定性。由于 Etcd 的一致性协议依赖于持久化存储保存日志，如果这些写入花费的时间太长，心跳可能会超时并触发选举，从而破坏群集的稳定性。通常，为了判断磁盘是否满足 Etcd，可以使用诸如 fio、diskbench 之类的基准测试工具。

  官方给出一组参考数据，关于读写延迟，通常需要顺序读写达到 50 IOPS 的磁盘，如 7200 RPM 磁盘。对于负载较高的集群，建议使用顺序读写达到 50 IOPS 的磁盘，如典型的本地 SSD 或高性能虚拟化块设备；关于读写带宽，Etcd 运行中只需要适量的磁盘带宽，但当故障成员恢复时，更大的磁盘带宽可以更快地完成恢复。通常，10MB/s 将在 15 秒内恢复 100MB 数据。对于大规模集群，建议在 15 秒内恢复 1GB 数据，需要 100MB/s 或更高的磁盘带宽。

在可能的情况下，使用 SSD 作为 Etcd 的存储。SSD 通常提供较低的写入延迟并且具有比旋转磁盘更小的变化，从而提高了 Etcd 的稳定性和可靠性。如果使用旋转磁盘，可以使用最快的磁盘（15,000 RPM）。对于旋转磁盘和 SSD，使用 RAID 0 也是提高磁盘速度的有效方法，对于 Etcd 集群通过一致性复制保证数据的高可用，没必要在底层 RAID 做冗余。

- 节点数：Etcd 需要时奇数节点，通常为 3、5、7，最大不要超过 7 个节点，节点越多，数据完成写入的时间越长。

根据上述的资源需求，整理四种 OpenShift 规模的集群下的资源需求如下表：

表 3-2 不同规模集群下的 Etcd 资源需求

OpenShif 节点数	**vCPUs**	**内存（G）**	**最大并发 IOPS**	**磁盘带宽（MB/s）**
50	2	8	3600	56
250	4	16	6000	94
1000	8	32	8000	125
3000	16	64	16000	250

需要强调的是，建议在投入生产之前，进行模拟的工作负载测试以评估是否满足集群需求，官方提供了工具 cluster-loader 用于模拟负载，Github 链接 https://github.com/openshift/svt/tree/master/openshift_scalability。

如果是优化 Etcd 性能，通常优先考虑扩展集群节点资源，除非是资源已经不是瓶颈的时候，才会考虑添加集群节点。

3）计算节点

Node 节点主要优化点包括：

- iptables 同步周期：在 OpenShift 中，很多通信需要依赖 iptables，可以通过参数调整 iptables 刷新的最大时间间隔以提高响应，默认 30s。修改 node 配置文件实现，配置参数如下：

  iptablesSyncPeriod: 30s

- 网络 MTU 值：包含节点网络 MTU 和 SDN MTU。关于节点网络 MTU 必须小于或等于网卡支持的最大 MTU，如果想要优化吞吐，则增大 MTU，如果想要优化延迟，则减小 MTU；关于 SDN MTU，必须至少小于节点网络 MTU 50 个字节，因为 SDN 的 overlay 包头为 50 个字节，在通常的以太网中，默认为 1450，在巨型帧以太网中，默认为 8950。修改 node 配置文件实现，配置参数如下：

networkConfig:

mtu: 1450

- 限制每个节点允许运行的最大 Pod 数：一旦节点运行 Pod 数过多，会导致调度变慢、容器进程 OOM、资源抢占导致应用性能下降，修改 Node 配置文件实现，配置参数如下：

kubeletArguments:

pods-per-core:

- "10"

max-pods:

- "100"

- 合理保留系统资源：配置节点为 kubelet 和操作系统操作保留一定的资源，避免 Pod 对资源的过度消耗，导致系统服务缺乏资源，修改 Node 配置文件实现，配置参数如下：

kubeletArguments:

kube-reserved:

- "cpu=200m,memory=512Mi"

system-reserved:

- "cpu=200m,memory=512Mi"

- 并行拉取镜像：如果使用 docker 1.9+，建议开启并行拉取镜像，提升效率，修改 Node 配置文件实现，配置参数如下：

kubeletArguments:

serialize-image-pulls:

- "false"

- 配置垃圾清理阈值：默认集群有 GC 自动清理无用的垃圾数据，如 image。修改 Node 配置文件实现，配置参数如下：

kubeletArguments:

image-gc-high-threshold:

- "90"

image-gc-low-threshold:

- "80"

- 容器清理：可以通过 Kubelet 自动清理退出的容器，避免在节点上保留多余无用的退出容器。修改 Node 配置文件实现，配置参数如下：

```
kubeletArguments:
  minimum-container-ttl-duration:
    - "10s"
  maximum-dead-containers-per-container:
    - "2"
  maximum-dead-containers:
    - "10"
```

（3） Docker 进程

Docker 进程优化点包括：

- Docker 使用的裸磁盘做存储卷，尽量使用 overlay2 存储驱动。绝不要在生产环境使用 Device Mapper 存储驱动的 loop-lvm 模式。
- 尽量周期性清理无用的垃圾镜像和退出容器。

（4） 网络优化

这里的网络优化主要指 Pod 与 Pod 的网络性能优化。在第二章中，我们介绍 OpenShift 的网络模型中提到 Pod 与 Pod 的网络通信主要依靠于 CNI 插件。所以性能上也主要取决于选择的 CNI 插件，不同的 CNI 插件调优的方法也不尽相同。如使用 Calico 网络时，建议设置 CALICO_IPV4POOL_IPIP=off 关闭同网段 IPIP 隧道。

这部分建议对选型的 CNI 进行网络性能测试，然后再基于选型的插件进行性能优化，这里就不一一列举了。

2. 容器层面优化

对容器层面的优化比较多，可以从 Pod 资源层面、应用配置层面等等进行优化，下面我们仅说明几个比较关键的容器优化。

（1） Router 优化

对于 Router，我们通常最关心每秒可以处理的请求数，但是通常与很多因素有关，如后端页面大小、HTTP keep-alive/close 模式、Route 类型、Router 中注册应用的个数等等。为了便于我们评估 Router 性能，官方给出 Router 的一组性能基线数据，测试场景为单个 Router 运行在公有云 4 vCPU/16GB 内存的节点上，后端应用为 1kB 的静态页面，Router 中注册 100 个应用，分别对不同的 HTTP 模式测试结果如下表 3-3、3-4 所示：

表 3-3 HTTP keep-alive 模式下测试数据

Route 加密类型	**未设置 Router 线程数**	**设置 Router 线程数为 4**
none	23681	24327
edge	14981	22768
passthrough	34358	34331
re-encrypt	13288	24605

表 3-4 HTTP close 模式下测试数据

Route 加密类型	**未设置 Router 线程数**	**设置 Router 线程数为 4**
none	3245	4527
edge	1910	3043
passthrough	3408	3922
re-encrypt	1333	2239

由于 Router 性能相关的因素较多，上表中的基准仅为我们测试提供对比参考。我们需要在实际的环境中测试并对比，得到单个 Router 的基准值，然后再进行下面的参数调优，确认是否对性能有所改善。

- 调整最大连接数：设定 Router 允许的每秒允许的最大并发连接数。通过 Router 环境变量 ROUTER_MAX_CONNECTIONS 设置，默认为 20000。
- 增加 Router 线程数：再 OpenShift 3.11 版本中，Router 支持多线程运行，通过 Router 环境变量 ROUTER_THREADS 设置，默认为 1。
- 优化超时：对于长连接，结合长的 client/server 超时时间和短的 reload 周期可以提升性能，通过 Router 环境变量 ROUTER_DEFAULT_TUNNEL_TIMEOUT、ROUTER_DEFAULT_CLIENT_TIMEOUT、

ROUTER_DEFAULT_SERVER_TIMEOUT、RELOAD_INTERVAL 来设置。

上述列出了一部分的 Router 调优手段，Router 本质上就是容器化的 Haproxy，可以通过部署自定义 Router 可以实现任何 Haproxy 可以使用的调优参数和方法。

需要注意的一点，Router 作为所有业务的入口，对性能和可用性要求是很高的。受 Haproxy 本身性能限制，单纯的依靠 Router 调优很难大幅提升性能，对于大量请求场景，利用 Router 可无限水平扩展的特性，建议使用如下方案或多种方案的结合：

- 扩展多个 Router，并在 Router 前使用硬件负载均衡（如 F5），直到满足性能要求。
- 使用 OpenShift 提供的 Router Sharding 技术将流量分隔为多个组，每个组有多个 Router。
- 使用 Nginx Ingress Controller 替换默认 OpenShift 提供的 Router。这种方案会导致某些 Router 相关的功能失效，如基于 Route 的灰度发布。但是在实际客户测试中发现，同等配置情况下，Nginx Ingress Controller 性能确实比 Router 好，TPS 大约在 2 到 3 倍左右。

上述三种方案在选择时，需要综合考虑企业现状和实际性能需求等，我们这里就不展开阐述了。

（2） Jenkins 优化

默认参数启动的 Jenkins 在大量 Pipeline 使用的情况下，会出现经常重启，一般是发生了内存溢出，建议在 Jenkins DeploymentConfig 调整如下环境变量：

OPENSHIFT_JENKINS_JVM_ARCH=x86_64

JAVA_GC_OPTS=-XX:+UseParallelGC -XX:MinHeapFreeRatio=20 -XX:MaxHeapFreeRatio=40 -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight=90 -XX:MaxMetaspaceSize=1024m

并把 Jenkins Pod 的内存资源调整为 8G 或 16G。

（3） 容器资源优化

OCP 应用资源的管理使用 Request 和 Limit 控制，分别表示下限和上限。在资源超量

使用时，需要注意不同资源配置的 Pod 有不同的 QoS 级别：

- Guaranteed：表示 Request=Limit，这种类型的 Pod 优先级最高。
- Burstable：表示 Request<Limit。
- BestEffort：未设置任何资源限制，这种类型的 Pod 优先级最低。

当集群资源严重不足（尤其是内存资源）时，对不同优先级的 Pod 处理的策略是不一样的，会优先强制停止低优先级的 Pod 释放资源。

在实际使用过程中，建议结合应用特性混合使用三种 QoS 级别的资源配置，而不是仅使用一种。因为集群的调度和资源 Quota 的统计是以 Request 为准，如果只使用第一种或第二种，导致集群可运行的 Pod 减少，但是资源却很空闲。例如，8 核 CPU 的计算节点，如果每个 Pod 设置 CPU Request 为 2 核，那么节点上可运行的 Pod 数小于 4 个。在运行 4 个 Pod 之后，只要新创建的 Pod 配置了 CPU 资源的 Request，就无法调度到该节点。但是 4 个 Pod 通常无法充分利用 8 核 CPU，导致节点资源空闲造成浪费。

优化是一个持续改进的过程，需要根据实际情况在多个层面进行优化。