第 5 章
Kubernetes 运维指南

为了让容器应用在 Kubernetes 集群中运行得更加有效，对 Kubernetes 集群本身也需要进行相应的配置和管理。本章将从 Kubernetes 集群管理、高级案例及 Trouble Shooting 等方面对 Kubernetes 集群的运维和查错进行详细说明，最后对 Kubernetes 1.3 版本开发中的新功能进行介绍。

5.1 Kubernetes 集群管理指南

本节将从 Node 的管理、Label 的管理、Namespace 资源共享、资源配额管理、集群 Master 高可用及集群监控等方面，对 Kubernetes 集群本身的运维管理进行详细说明。

5.1.1 Node 的管理

1. Node 的隔离与恢复

在硬件升级、硬件维护等情况下，我们需要将某些 Node 进行隔离，脱离 Kubernetes 集群的调度范围。Kubernetes 提供了一种机制，既可以将 Node 纳入调度范围，也可以将 Node 脱离调度范围。

创建配置文件 unschedule_node.yaml，在 spec 部分指定 unschedulable 为 true：

```
apiVersion: v1
kind: Node
metadata:
```

```
  name: k8s-node-1
  labels:
    kubernetes.io/hostname: k8s-node-1
spec:
  unschedulable: true
```

然后，通过 kubectl replace 命令完成对 Node 状态的修改：

```
$ kubectl replace -f unschedule_node.yaml
node "k8s-node-1" replaced
```

查看 Node 的状态，可以观察到在 Node 的状态中增加了一项 SchedulingDisabled：

```
# kubectl get nodes
NAME         STATUS                     AGE
k8s-node-1   Ready,SchedulingDisabled   1h
```

对于后续创建的 Pod，系统将不会再向该 Node 进行调度。

也可以不使用配置文件，直接使用 kubectl patch 命令完成：

```
$ kubectl patch node k8s-node-1 -p '{"spec":{"unschedulable":true}}'
```

需要注意的是，将某个 Node 脱离调度范围时，在其上运行的 Pod 并不会自动停止，管理员需要手动停止在该 Node 上运行的 Pod。

同样，如果需要将某个 Node 重新纳入集群调度范围，则将 unschedulable 设置为 false，再次执行 kubectl replace 或 kubectl patch 命令就能恢复系统对该 Node 的调度。

在 Kubernetes 当前的版本中，kubectl 的子命令 cordon 和 uncordon 也用于实现将 Node 进行隔离和恢复调度的操作。

例如，使用 kubectl cordon <node_name>对某个 Node 进行隔离调度操作：

```
# kubectl cordon k8s-node-1
node "k8s-node-1" cordoned

# kubectl get nodes
NAME         STATUS                     AGE
k8s-node-1   Ready,SchedulingDisabled   1h
```

使用 kubectl uncordon <node_name>对某个 Node 进行恢复调度操作：

```
# kubectl uncordon k8s-node-1
node "k8s-node-1" uncordoned

# kubectl get nodes
NAME         STATUS    AGE
k8s-node-1   Ready     1h
```

2. Node 的扩容

在实际生产系统中会经常遇到服务器容量不足的情况，这时就需要购买新的服务器，然后将应用系统进行水平扩展来完成对系统的扩容。

在 Kubernetes 集群中，一个新 Node 的加入是非常简单的。在新的 Node 节点上安装 Docker、kubelet 和 kube-proxy 服务，然后配置 kubelet 和 kube-proxy 的启动参数，将 Master URL 指定为当前 Kubernetes 集群 Master 的地址，最后启动这些服务。通过 kubelet 默认的自动注册机制，新的 Node 将会自动加入现有的 Kubernetes 集群中，如图 5.1 所示。

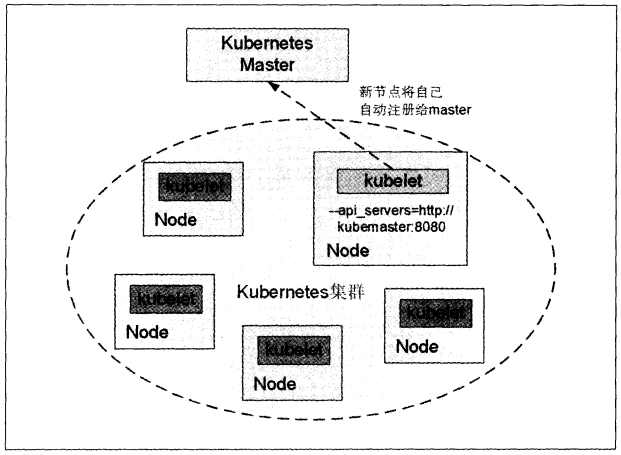

图 5.1 新节点自动注册完成扩容

Kubernetes Master 在接受了新 Node 的注册之后，会自动将其纳入当前集群的调度范围内，在之后创建容器时，就可以向新的 Node 进行调度了。

通过这种机制，Kubernetes 实现了集群中 Node 的扩容。

5.1.2 更新资源对象的 Label

Label（标签）作为用户可灵活定义的对象属性，在正在运行的资源对象上，仍然可以随时通过 kubectl label 命令对其进行增加、修改、删除等操作。

例如，我们要给已创建的 Pod “redis-master-bobr0” 添加一个标签 role=backend：

```
$ kubectl label pod redis-master-bobr0 role=backend
pod "redis-master-bobr0" labeled
```

查看该 Pod 的 Label：

```
$ kubectl get pods -Lrole
```

```
NAME                READY     STATUS    RESTARTS   AGE       ROLE
redis-master-bobr0   1/1       Running   0          3m        backend
```

删除一个 Label 时，只需在命令行最后指定 Label 的 key 名并与一个减号相连即可：

```
$ kubectl label pod redis-master-bobr0 role-
pod "redis-master-bobr0" labeled
```

修改一个 Label 的值时，需要加上--overwrite 参数：

```
$ kubectl label pod redis-master-bobr0 role=master --overwrite
pod "redis-master-bobr0" labeled
```

5.1.3　Namespace：集群环境共享与隔离

在一个组织内部，不同的工作组可以在同一个 Kubernetes 集群中工作，Kubernetes 通过命名空间和 Context 的设置来对不同的工作组进行区分，使得它们既可以共享同一个 Kubernetes 集群的服务，也能够互不干扰，如图 5.2 所示。

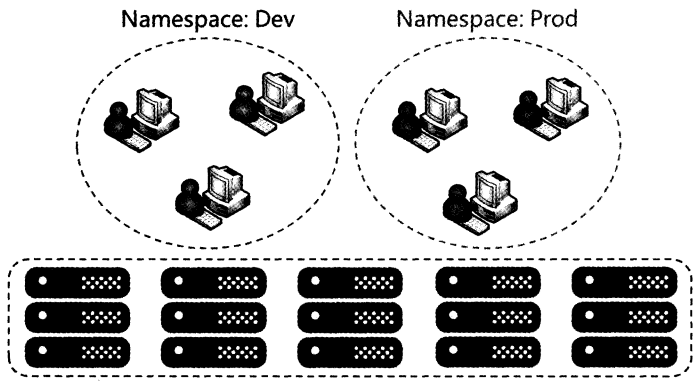

图 5.2　集群环境共享和隔离

假设在我们的组织中有两个工作组：开发组和生产运维组。开发组在 Kubernetes 集群中需要不断创建、修改、删除各种 Pod、RC、Service 等资源对象，以便实现敏捷开发的过程。而生产运维组则需要使用严格的权限设置来确保生产系统中的 Pod、RC、Service 处于正常运行状态且不会被误操作。

1. 创建 namespace

为了在 Kubernetes 集群中实现这两个分组，首先需要创建两个命名空间。

namespace-development.yaml：

```
apiVersion: v1
```

```
kind: Namespace
metadata:
  name: development
```

namespace-production.yaml：

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
```

使用 kubectl create 命令完成命名空间的创建：

```
$ kubectl create -f namespace-development.yaml
namespaces/development

$ kubectl create -f namespace-production.yaml
namespaces/production
```

查看系统中的命名空间：

```
$ kubectl get namespaces
NAME         LABELS           STATUS
default      <none>           Active
development   name=development   Active
production    name=production    Active
```

2. 定义 Context（运行环境）

接下来，需要为这两个工作组分别定义一个 Context，即运行环境。这个运行环境将属于某个特定的命名空间。

通过 kubectl config set-context 命令定义 Context，并将 Context 置于之前创建的命名空间中：

```
$ kubectl config set-cluster kubernetes-cluster --server=https://192.168.1.
128:8080
$ kubectl config set-context ctx-dev --namespace=development --cluster=kubernetes-
cluster --user=dev
$ kubectl config set-context ctx-prod --namespace=production --cluster=kubernetes-
cluster --user=prod
```

使用 kubectl config view 命令查看已定义的 Context：

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    server: http://192.168.1.128:8080
  name: kubernetes-cluster
contexts:
- context:
```

```
    cluster: kubernetes-cluster
    namespace: development
  name: ctx-dev
- context:
    cluster: kubernetes-cluster
    namespace: production
  name: ctx-prod
current-context: ctx-dev
kind: Config
preferences: {}
users: []
```

注意，通过 kubectl config 命令在${HOME}/.kube 目录下生成了一个名为 config 的文件，文件内容即以 kubectl config view 命令查看到的内容。所以，也可以通过手工编辑该文件的方式来设置 Context。

3. 设置工作组在特定 Context 环境中工作

使用 kubectl config use-context <context_name>命令来设置当前的运行环境。

下面的命令将把当前运行环境设置为“ctx-dev”：

```
$ kubectl config use-context ctx-dev
```

通过这个命令，当前的运行环境即被设置为开发组所需的环境。之后的所有操作都将在名为“development”的命名空间中完成。

现在，以 redis-slave RC 为例创建两个 Pod：

redis-slave-controller.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  replicas: 2
  selector:
    name: redis-slave
  template:
    metadata:
      labels:
        name: redis-slave
    spec:
      containers:
      - name: slave
```

```
        image: kubeguide/guestbook-redis-slave
        ports:
        - containerPort: 6379
```

```
$ kubectl create -f  redis-slave-controller.yaml
replicationcontrollers/redis-slave
```

查看创建好的 Pod：

```
$ kubectl get pods
NAME                READY     STATUS    RESTARTS   AGE
redis-slave-0feq9   1/1       Running   0          6m
redis-slave-6i0g4   1/1       Running   0          6m
```

可以看到容器被正确创建并运行起来了。而且，由于当前的运行环境是 ctx-dev，所以不会影响到生产运维组的工作。

让我们切换到生产运维组的运行环境：

```
$ kubectl config use-context ctx-prod
```

查看 RC 和 Pod：

```
$ kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR   REPLICAS

$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
```

结果为空，说明看不到开发组创建的 RC 和 Pod。

现在我们为生产运维组也创建两个 redis-slave 的 Pod：

```
$ kubectl create -f  redis-slave-controller.yaml
replicationcontrollers/redis-slave
```

查看创建好的 Pod：

```
$ kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
redis-slave-a4m7s   1/1       Running   0          12s
redis-slave-xyrkk   1/1       Running   0          12s
```

可以看到容器被正确创建并运行起来了，并且当前的运行环境是 ctx-prod，也不会影响开发组的工作。

至此，我们为两个工作组分别设置了两个运行环境，在设置好当前的运行环境时，各工作组之间的工作将不会相互干扰，并且它们都能够在同一个 Kubernetes 集群中同时工作。

5.1.4　Kubernetes 资源管理

本章从计算资源管理（Compute Resources）、资源配置范围管理（LimitRange）、服务质量管理（QoS）及资源配额管理（ResourceQuota）等方面，对 Kubernetes 集群内的资源管理进行详细说明，结合实践操作、常见问题分析和一个完整的示例，对 Kubernetes 集群资源管理相关的运维工作提供指导。

1. 计算资源管理（Compute Resources）

在配置 Pod 的时候，我们可以为其中的每个容器指定需要使用的计算资源（CPU 和内存）。

计算资源的配置项分为两种：一种是资源请求（Resource Requests，简称 Requests），表示容器希望被分配到的、可完全保证的资源量，Requests 的值会提供给 Kubernetes 调度器（Kubernetes Scheduler）以便于优化基于资源请求的容器调度；另外一种是资源限制（Resource Limits，简称 Limits），Limits 是容器最多能使用到的资源量的上限，这个上限值会影响节点上发生资源竞争时的解决策略。

当前版本的 Kubernetes 中，计算资源的资源类型分为两种：CPU 和内存（Memory）。这两种资源类型都有一个基本单位：对于 CPU 而言，基本单位是核心数（Cores）；而内存的基本单位是字节数（Bytes）。CPU 和内存一起构成了目前 Kubernetes 中的计算资源（也可简称为资源）。

计算资源是可计量的，能被申请、分配和使用的基础资源，这使之区别于 API 资源（API Resources，例如 Pod 和 services 等）。

1）Pod 和容器的 Requests 和 Limits

Pod 中的每个容器都可以配置以下 4 个参数。

- spec.container[].resources.requests.cpu。
- spec.container[].resources.limits.cpu。
- spec.container[].resources.requests.memory。
- spec.container[].resources.limits.memory。

这四个参数分别对应容器的 CPU 和内存的 Requests 和 Limits，它们具有以下特点。

- Requests 和 Limits 都是可选的。在某些集群中如果在 Pod 创建或者更新的时候，没设置资源限制或者资源请求值，那么可能会使用系统提供一个默认值，这个默认值取决于集群的配置。
- 如果 Request 没有配置，那么默认会被设置为等于 Limits。

◎ 而任何情况下 Limits 都应该设置为大于或者等于 Requests。

以 CPU 为例，图 5.3 显示了未设置 CPU Limits 和设置 CPU Limits 的 CPU 使用率的区别。

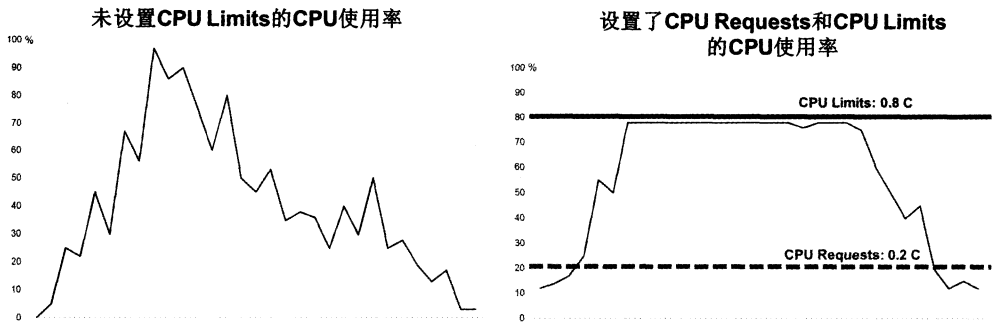

图 5.3 未设置和设置了 CPU Limits 的 CPU 使用率样例

尽管 Requests 和 Limits 只能设置到容器上，但是设置 Pod 级别的 Requests 和 Limits 能极大程度上提高我们对 Pod 管理的便利性和灵活性，因此 Kubernetes 中提供对 Pod 级别的 Requests 和 Limits 配置。对于 CPU 和内存而言，Pod 的 Requests 或 Limits 是指该 Pod 中所有容器的 Requests 或 Limits 的总和（Pod 中没设置 Request 或 Limits 的容器，该项的值被当作 0 或者按照集群配置的默认值来计算）。下面对 CPU 和内存这两种计算资源各自的特点进行说明。

（1）CPU

CPU 的 Requests 和 Limits 是通过 CPU 数（cpus）来度量的。CPU 资源值支持最多三位小数：如果一个容器的 spec.container[].resources.requests.cpu 设置为 0.5，那么它会获得半个 CPU；同理如果设置为 1，就会获得 1 个 CPU。0.1CPU 等价于 100m CPU（100 millicpu），而在 Kubernetes API 中自动将这种小数 0.1 转化为 100m，因此 CPU 的小数最多支持三位数字，而 Kubernetes 官方也更推荐直接使用形如 100m 的 millicpu 作为计量单位。

CPU 资源值是绝对值，而不是相对值：比如 0.1CPU 不管是在单核或者多核机器上都是一样的，都严格等于 0.1 CPU core。

（2）内存（Memory）

内存的 Requests 和 Limits 计量单位是字节数（Bytes）。内存值用使用整数或者定点整数加上国际单位制（International System of Units）来表示。国际单位制包括十进制的 E、P、T、G、M、K、m，或二进制的 Ei、Pi、Ti、Gi、Mi、Ki。比如：KiB 与 MiB 是二进制表示的字节单位，而常见的 KB 与 MB 则是十进制表示的字节单位。两种方式的区别举例说明如下：

1 KB（kilobyte）= 1000 bytes = 8000 bits

1 KiB（kibibyte）= 2^10 bytes = 1024 bytes = 8192 bits

因此，下面几种内存配置的意思是一样的：128974848、129e6、129M、123Mi

Kubernetes 的计算资源单位是大小写敏感的，因为 m 可以表示千分之一单位（milli unit），而 M 可以表示十进制的 1000，两者的含义不同；同理可知，小写的 k 不是一个合法的资源单位。

以一个 Pod 中的资源配置为例：

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```


该 Pod 包含两个容器，每个容器配置的 Requests 都是 0.25 CPU 和 64MiB（2^{26} bytes）内存，而配置的 Limits 都是 0.5 CPU 和 128MiB（2^{27} bytes）内存。

这个 Pod 的 Requests 和 Limits 等于 Pod 中所有容器对应配置的总和，所以 Pod 的 Requests 是 0.5 CPU 和 128MiB（2^{27} bytes）内存，Limits 是 1 CPU 和 256MiB（2^{28} bytes）内存。

2）基于 Requests 和 Limits 的 Pod 调度机制

当一个 Pod 创建成功时，Kubernetes 调度器（Scheduler）为该 Pod 选择一个节点（Node）来执行。对于每种计算资源（CPU 和内存）而言，每个节点都有一个能用于运行 Pod 的最大容量值。调度器在调度时，首先要确保调度后该节点上所有 Pod 的 CPU 和内存的 Requests 总和不能超过该节点能提供给 Pod 使用的 CPU 和内存的最大容量值。

例如某个节点上 CPU 资源充足，而内存为 4GB，其中 3GB 可以运行 Pod，某 Pod 的内存

Requests 为 1GB、Limits 为 2GB，那么这个节点上最多可运行 3 个这种 Pod。

这里需要注意的是：可能某些节点上的实际资源使用量非常低，但是如果该节点上已运行 Pod 配置的 Requests 值的总和已经非常高，再加上需要调度的 Pod 的 Requests 值会直接超过该节点提供给 Pod 的资源容量上限，Kubernetes 仍然不会将 Pod 调度到这个节点上。这是因为如果 Kubernetes 将 Pod 调度到该节点上，那么如果后面该节点上运行的 Pod 面临服务峰值等情况，可能会导致 Pod 资源短缺的情况发生。

接着上面的例子，假设该节点已经启动 3 个 Pod 实例，而这 3 个 Pod 的实际内存使用都不足 500MB，那么理论上该节点的可用内存应该大于 1.5GB，但是由于该节点的 Pod Requests 总和已经等于节点的可用内存上限，因此 Kubernetes 不会再将任何 Pod 实例调度到该节点上执行。

3）Requests 和 Limits 资源配置机制

当 kubelet 启动 Pod 的一个容器时，它会将容器的 Requests 和 Limits 值转化为相应的容器启动参数传递给容器执行器（Docker 或者是 rkt）。

如果容器的执行环境是 Docker，那么容器的 4 个参数是这样传递给 Docker 的。

（1）spec.container[].resources.requests.cpu

这个参数会转化为 core 数（比如配置的 100m 会转化为 0.1），然后乘以 1024，再将这个结果作为--cpu-shares 参数的值传递给 docker run 命令。在 docker run 命令中，--cpu-share 参数是一个相对权重值（Relative Weight），这个相对权重值会决定 Docker 在资源竞争时分配给容器的资源比例。举例说明--cpu-shares 参数在 Docker 中的含义：比如两个容器的 CPU Requests 分别设置为 1 和 2，那么容器在 docker run 启动时对应的--cpu-shares 参数值分别为 1024 和 2048，在主机 CPU 资源产生竞争时，Docker 会尝试按照 1∶2 的配比将 CPU 资源分配给这两个容器使用。

这里需要区分清楚的是：这个参数对于 Kubernetes 而言是绝对值，主要用于 Kubernetes 调度和管理的依据（参见下文 QoS 章节）；同时这个参数值会设置为--cpu-shares 参数传递给 Docker，--cpu-shares 参数对于 Docker 而言又是相对值，主要用于资源分配比例。这两种用途的作用范围不同，所以并不会发生冲突。

（2）spec.container[].resources.limits.cpu

这个参数会转化为 millicore 数（比如配置的 1 会转化为 1000，而配置的 100m 转化为 100），将此值乘以 100000，再除以 1000，然后将结果值作为--cpu-quota 参数的值传递给 docker run 命令。docker run 命令中另外一个参数--cpu-period 默认设置为 100000，表示 Docker 重新计量和分配 CPU 的使用时间间隔为 100000 微秒（100 毫秒）。

Docker 的--cpu-quota 参数和--cpu-period 参数一起配合完成对容器 CPU 的使用限制：比如 Kubernetes 中配置容器的 CPU Limits 为 0.1，那么计算后--cpu-quota 为 10000，而--cpu-period

为 100000，这意味着 Docker 在 100 毫秒内最多给该容器分配 10 毫秒*core 的计算资源用量，10/100=0.1 core 的结果与 Kubernetes 配置的意义是一致的。

注意：如果 kubelet 启动参数--cpu-cfs-quota 设置为 true，那么 kubelet 会强制要求所有 Pod 都必须配置 CPU Limits（如果 Pod 没配置，而集群提供了默认配置也可以）。而从 Kubernetes 1.2 版本开始，这个--cpu-cfs-quota 启动参数的默认值就是 true。

（3）spec.container[].resources.requests.memory

这个参数值只提供给 Kubernetes 调度器（Kubernetes Scheduler）作为调度和管理的依据，不会作为任何参数传递给 Docker。

（4）spec.container[].resources.limits.memory

这个参数值会转化为单位为 bytes 的整数，数值会作为--memory 参数传递给 docker run 命令。

如果一个容器在运行过程中使用了超出了其内存 Limits 配置的内存限制值，那么它可能会被“杀掉”，如果这个容器是一个可重启的容器，那么之后它会被 kubelet 重新启动起来。因此容器的 Limits 配置需要进行准确的测试和评估。

与内存 Limits 不同的是 CPU 在容器技术中属于可压缩资源，因此对于 CPU 的 Limits 配置一般不会引发因偶然超标使用而导致容器被系统“杀掉”的情况。

4）计算资源使用情况监控

Pod 的资源用量会作为 Pod 的状态信息一同上报给 Master。如果集群中配置了 Heapster 来监控集群的性能数据，那么还可以从 Heapster 中查看 Pod 的资源用量信息。

5）计算资源相关常见问题分析

（1）Pod 状态为 pending，错误信息为 FailedScheduling。

如果 Kubernetes 调度器（Kubernetes Scheduler）在集群中找不到合适的节点来运行 Pod，那么这个 Pod 会一直处于未调度状态，直到调度器找到合适的节点为止。每次调度器尝试调度失败，Kubernetes 都会产生一个事件（event），我们可以通过下面这种方式来查看事件的信息：

```
$ kubectl describe pod frontend | grep -A 3 Events
Events:
  FirstSeen    LastSeen    Count From      Subobject   PathReason      Message
  36s      5s      6     {scheduler }    FailedScheduling Failed for reason
PodExceedsFreeCPU and possibly others
```

在上面这个例子中，名为 frontend 的 Pod 由于节点的 CPU 资源不足而调度失败（PodExceedsFreeCPU），同样，如果内存不足也可能导致调度失败（PodExceedsFreeMemory）。

如果一个或者多个 Pod 调度失败且有这类错误，那么我们可以尝试以下几种解决方法。

- 添加更多的节点到集群中。
- 停止一些不必要的运行中的 Pod，释放资源。
- 检查 Pod 的配置，错误的配置可能导致该 Pod 永远都无法被调度执行。比如如果整个集群中所有节点都只有 1 CPU，而 Pod 配置的 CPU Requests 为 2，那么该 Pod 就不会被调度执行。

我们可以使用 kubectl describe nodes 命令来查看集群中节点的计算资源容量和已使用量：

```
$ kubectl describe nodes k8s-node-1
Name:         k8s-node-1
...
Capacity:
 cpu:     1
 memory:  464Mi
 pods:    40
Allocated resources (total requests):
 cpu:     910m
 memory:  2370Mi
 pods:    4
...
Pods:             (4 in total)
  Namespace           Name                                CPU(milliCPU)
Memory(bytes)
  frontend            webserver-ffj8j                          500 (50% of total)
   2097152000 (50% of total)
  kube-system         fluentd-cloud-logging-k8s-node-1         100 (10% of total)
   209715200 (5% of total)
  kube-system         kube-dns-v8-qopgw                    310 (31% of total)
178257920 (4% of total)
TotalResourceLimits:
  CPU(milliCPU):      910 (91% of total)
  Memory(bytes):      2485125120 (59% of total)
...
```

超过可用资源容量上限（Capacity）和已分配资源量（Allocated resources）差额的 Pod 无法运行在该 Node 上。这个例子中，如果一个 Pod 的 Requests 超过 90 millicpus 或者超过 1341MiB 内存，那么就无法运行在这个节点上。

在后面的资源配额（Resource Quota）章节中，我们还可以配置针对一组 Pod 的 Requests 和 Limits 总量的限制，这种限制可以作用于命名空间，通过这种方式我们可以防止一个命名空间下的用户将所有资源全部据为己有。

（2）容器被强行终止（Terminated）

如果容器使用的资源超过了它配置的 Limits，那么该容器可能会被强制终止。我们可以通

过 kubectl describe pod 命令来确认容器是否是因为这个原因被终止的：

```
    $ kubectl describe pod simmemleak-hra99
    Name:                       simmemleak-hra99
    Namespace:                   default
    Image(s):                    saadali/simmemleak
    Node:                       192.168.18.3
    Labels:                      name=simmemleak
    Status:                      Running
    Reason:
    Message:
    IP:                         172.17.1.3
    Replication Controllers:       simmemleak (1/1 replicas created)
    Containers:
      simmemleak:
        Image:  saadali/simmemleak
        Limits:
          cpu:                  100m
          memory:               50Mi
        State:                  Running
          Started:              Tue, 07 Jul 2015 12:54:41 -0700
        Last Termination State:    Terminated
          Exit Code:            1
          Started:              Fri, 07 Jul 2015 12:54:30 -0700
          Finished:             Fri, 07 Jul 2015 12:54:33 -0700
        Ready:                  False
        Restart Count:           5
    Conditions:
      Type      Status
      Ready     False
    Events:
      FirstSeen                     LastSeen                      Count  From
SubobjectPath                   Reason      Message
      Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1
{scheduler }                                                    scheduled
Successfully assigned simmemleak-hra99 to kubernetes-node-tf0f
      Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1     {kubelet
kubernetes-node-tf0f}    implicitly required container POD   pulled     Pod
container image "gcr.io/google_containers/pause:0.8.0" already present on machine
      Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1     {kubelet
kubernetes-node-tf0f}    implicitly required container POD   created    Created
with docker id 6a41280f516d
      Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1     {kubelet
kubernetes-node-tf0f}    implicitly required container POD   started    Started
with docker id 6a41280f516d
      Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1     {kubelet
kubernetes-node-tf0f}    spec.containers{simmemleak}        created    Created
```

```
with docker id 87348f12526a
```

Restart Count: 5 说明这个名为 simmemleak 的容器被强制终止并重启了 5 次。

我们可以在使用 kubectl get pod 命令时添加-o go-template=...格式参数来读取已终止容器之前的状态信息：

```
    $ kubectl get pod -o
go-template='{{range.status.containerStatuses}}{{"Container Name:
"}}{{.name}}{{"\r\nLastState: "}}{{.lastState}}{{end}}'  simmemleak-60xbc
    Container Name: simmemleak
    LastState: map[terminated:map[exitCode:137 reason:OOM Killed
startedAt:2015-07-07T20:58:43Z finishedAt:2015-07-07T20:58:43Z
containerID:docker://0e4095bba1feccdfe7ef9fb6ebffe972b4b14285d5acdec6f0d3ae8a22f
ad8b2]]
```

这里我们可以看到这个容器因为 reason:OOM Killed 而被强制终止，说明这个容器的内存超过了限制（Out of Memory）。

6）计算资源管理的演进

当前版本的 Kubernetes 中的 Requests 和 Limits 都是作用于容器级别的，未来 Kubernetes 计划增加对直接作用于 Pod 级别的资源配置的支持，这种资源配置是能被 Pod 内的所有容器共享的，包括 emptyDir 这种 Pod 级别的 Volume。

从资源的种类来看，目前 Kubernetes 只能支持 CPU 和内存两种计算资源类型，在后续的版本中，Kubernetes 计划支持更多的资源类型，包括节点磁盘空间资源，还将支持自定义的资源类型。

2. 资源的配置范围管理（LimitRange）

默认情况下，Kubernetes 的 Pod 会以无限制的 CPU 和内存运行。这也就意味着 Kubernetes 系统中任何的 Pod 都可以使用其所在节点上的所有可用的 CPU 和内存。通过配置 Pod 的计算资源 Requests 和 Limits，我们可以限制 Pod 的资源使用，但对于 Kubernetes 集群管理员而言，配置每一个 Pod 的 Requests 和 Limits 是烦琐且限制性过强的。更多的时候，我们需要的是对集群内 Request 和 Limits 的配置做一个全局的统一的限制。常见的配置场景如下。

- 集群中的每个节点有 2GB 内存，集群管理员不希望任何 Pod 申请超过 2GB 的内存：因为整个集群中没有任何节点能满足超过 2GB 内存的请求。如果某个 Pod 的内存配置超过 2GB，那么该 Pod 将永远都无法被调度到任何节点上执行。为了防止这种情况的发生，集群管理员希望能在系统管理功能中设置禁止 Pod 申请超过 2GB 内存。
- 集群由同一个组织中的两个团队共享，各自分别用来运行生产环境和开发环境。生产环境最多可以使用 8GB 内存，而开发环境最多可以使用 512MB 内存。集群管理员希望通过为这两个环境创建不同的命名空间（namespace）并为每个命名空间设置不同的

限制来满足这个需求。

◎ 用户创建 Pod 时使用的资源可能会刚好比整个机器资源的上限稍小一点，而恰好剩下的资源大小非常尴尬：不足以运行其他任务但整个集群加起来又非常浪费。因此，集群管理员希望设置每个 Pod 必须至少使用集群平均资源值（CPU 和内存）的 20%，这样集群能够提供更好的资源一致性的调度，从而减少了资源浪费。

针对这些需求，Kubernetes 提供了 LimitRange 机制对 Pod 和容器的 Requests 和 Limits 配置进一步做出限制。在下面的示例中，将说明如何将 LimitsRange 应用到一个 Kubernetes 的命名空间（namespace）中，然后说明 LimitRange 的几种限制方式，比如最大及最小范围、Requests 和 Limits 的默认值、Limits 与 Requests 最大比例上限等。

下面通过 LimitRange 的设置和应用对其进行说明。

1）创建一个 namespace

创建一个名为 limit-example 的 namespace：

```
$ kubectl create namespace limit-example
namespace "limit-example" created
```

2）为 namespace 设置 LimitRange

为 namespace “limit-example” 创建一个简单的 LimitRange。创建 limits.yaml 配置文件，内容如下：

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mylimits
spec:
  limits:
  - max:
      cpu: "4"
      memory: 2Gi
    min:
      cpu: 200m
      memory: 6Mi
    maxLimitRequestRatio:
      cpu: 3
      memory: 2
    type: Pod
  - default:
      cpu: 300m
      memory: 200Mi
    defaultRequest:
      cpu: 200m
```

```
      memory: 100Mi
    max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 100m
      memory: 3Mi
    maxLimitRequestRatio:
      cpu: 5
      memory: 4
    type: Container
```

创建该 LimitRange：

```
$ kubectl create -f limits.yaml --namespace=limit-example
limitrange "mylimits" created
```

查看 namespace limit-example 中的 LimitRange：

```
$ kubectl describe limits mylimits --namespace=limit-example
Name:   mylimits
Namespace:  limit-example
Type        Resource     Min      Max   Default Request      Default Limit
Max Limit/Request Ratio
----        --------     ---      ---     ---------------      -------------
-----------------------
Pod         cpu          200m     4       -            -           3
Pod         memory       6Mi      2Gi     -            -           2
Container   cpu          100m     2       200m         300m        5
Container   memory       3Mi      1Gi      100Mi        200Mi      4
```

下面解释一下 LimitRange 中各项配置的意义和特点。

（1）不论是 CPU 还是内存，在 LimitRange 中，Pod 和 Container 都可以设置 Min、Max 和 Max Limit/Requests Ratio 这三种参数。Container 还可以设置 Default Request 和 Default Limit 这两种参数，而 Pod 不能设置 Default Request 和 Default Limit。

（2）对 Pod 和 Container 的五种参数的解释如下。

- Container 的 Min（上面的 100m 和 3Mi）是 Pod 中所有容器的 Requests 值的下限；Container 的 Max（上面的 2 和 1Gi）是 Pod 中所有容器的 Limits 值的上限；Container 的 Default Request（上面的 200m 和 100Mi）是 Pod 中所有未指定 Requests 值的容器的默认 Requests 值；Container 的 Default Limit（上面的 300m 和 200Mi）是 Pod 中所有未指定 Limits 值的容器的默认 Limits 值。对于同一资源类型，这 4 个参数必须满足以下关系：Min ≤ Default Request ≤ Default Limit ≤ Max。
- Pod 的 Min（上面的 200m 和 6Mi）是 Pod 中所有容器的 Requests 值的总和的下限；Pod

的 Max（上面的 4 和 2Gi）是 Pod 中所有容器的 Limits 值的总和的上限。当容器未指定 Requests 值或者 Limits 值时，将使用 Container 的 Default Request 值或者 Default Limit 值。

◎ Container 的 Max Limit/Requests Ratio（上面的 5 和 4）限制了 Pod 中所有容器的 Limits 值与 Requests 值的比例上限；而 Pod 的 Max Limit/Requests Ratio（上面的 3 和 2）限制了 Pod 中所有容器的 Limits 值总和与 Requests 值总和的比例上限。

（3）如果设置了 Container 的 Max，那么对于该类资源而言，整个集群中的所有容器都必须设置 Limits，否则将无法成功创建。Pod 内的容器未配置 Limits 时，将使用 Default Limit 的值（本例中的 300m CPU 和 200Mi 内存），而如果 Default 也未配置则无法成功创建。

（4）如果设置了 Container 的 Min，那么对于该类资源而言，整个进群中的所有容器都必须设置 Requests。如果创建 Pod 的容器时未配置该类资源的 Requests，那么创建过程会报验证错误。Pod 里容器的 Requests 在未配置时，可以使用默认值 defaultRequest（本例中的 200m CPU 和 100Mi 内存）；如果未配置而又没有 defaultRequest，那么会默认等于该容器的 Limits；如果此时 Limits 也未定义，那么就会报错。

（5）对于任意一个 Pod 而言，该 Pod 中所有容器的 Requests 总和必须大于或等于 6Mi，而且所有容器的 Limits 总和必须小于或等于 1Gi；同样，所有容器的 CPU Requests 总和必须大于或等于 200m，而且所有容器的 CPU Limits 总和必须小于或等于 2。

（6）Pod 里任何容器的 Limits 与 Requests 的比例不能超过 Container 的 Max Limit/Requests Ratio；Pod 里所有容器的 Limits 总和与 Requests 的总和的比例不能超过 Pod 的 Max Limit/Requests Ratio。

3）创建 Pod 时触发 LimitRange 限制

最后，让我们看看 LimitRange 生效时对容器的资源限制效果。

命名空间中的限制（LimitRange）只会在 Pod 创建或者更新的时候执行检查。如果手动修改限制（LimitRange）为一个新的值，那么这个新的值不会去检查或限制之前已经在该命名空间中创建好的 Pod。

如果用户创建 Pod 时，配置的资源值（CPU 或者内存）超过了 LimitRange 的限制，那么该创建过程会报错，在错误信息中会说明详细的错误原因。

下面通过创建一个单容器 Pod 来展示默认限制是如何配置到 Pod 上的：

```
$ kubectl run nginx --image=nginx --replicas=1 --namespace=limit-example
deployment "nginx" created
```

查看已创建的 Pod：

```
$ kubectl get pods --namespace=limit-example
NAME                     READY     STATUS    RESTARTS   AGE
```

```
nginx-2040093540-s8vzu   1/1       Running   0          11s
```

查看该 Pod 的 resources 相关信息：

```
$ kubectl get pods nginx-2040093540-s8vzu --namespace=limit-example -o yaml |
grep resources -C 8
    resourceVersion: "57"
    selfLink: /api/v1/namespaces/limit-example/pods/nginx-2040093540-ivimu
    uid: 67b20741-f53b-11e5-b066-64510658e388
  spec:
    containers:
    - image: nginx
      imagePullPolicy: Always
      name: nginx
      resources:
        limits:
          cpu: 300m
          memory: 200Mi
        requests:
          cpu: 200m
          memory: 100Mi
      terminationMessagePath: /dev/termination-log
      volumeMounts:
```

由于该 Pod 未配置资源 Requests 和 Limits，所以使用了 namespace limit-example 中的默认 CPU 和内存定义的 Requests 和 Limits 值。

下面创建一个超出资源限制的 Pod（使用 3 CPU）：

invalid-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: invalid-pod
spec:
  containers:
  - name: kubernetes-serve-hostname
    image: gcr.io/google_containers/serve_hostname
    resources:
      limits:
        cpu: "3"
        memory: 100Mi
```

创建该 Pod，可以看到系统报错了，并且提供了错误原因为超过了限制。

```
$ kubectl create -f invalid-pod.yaml --namespace=limit-example
Error from server: error when creating "invalid-pod.yaml": Pod "invalid-pod" is
forbidden: [Maximum cpu usage per Pod is 2, but limit is 3., Maximum cpu usage per
Container is 2, but limit is 3.]
```

接下来的例子展示了 LimitRange 对 maxLimitRequestRatio 的限制：

limit-test-nginx.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: limit-test-nginx
  labels:
    name: limit-test-nginx
spec:
  containers:
  - name: limit-test-nginx
    image: nginx
    resources:
      limits:
        cpu: "1"
        memory: 512Mi
      requests:
        cpu: "0.8"
        memory: 250Mi
```

由于 limit-test-nginx 这个 Pod 的全部内存 Limits 总和与 Requests 总和的比例为 512∶250，大于 LimitRange 中定义的 Pod 的内存 maxLimitRequestRatio 值 2，因此创建会失败：

```
$ kubectl create -f limit-test-nginx.yaml --namespace=limit-example
Error from server: error when creating "limit-test-nginx.yaml": pods
"limit-test-nginx" is forbidden: [memory max limit to request ratio per Pod is 2,
but provided ratio is 2.048000.]
```

下面的例子为满足 LimitRange 限制的 Pod：

valid-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: valid-pod
  labels:
    name: valid-pod
spec:
  containers:
  - name: kubernetes-serve-hostname
    image: gcr.io/google_containers/serve_hostname
    resources:
      limits:
        cpu: "1"
        memory: 512Mi
```

创建 Pod 将会成功：

```
$ kubectl create -f valid-pod.yaml --namespace=limit-example
```

```
pod "valid-pod" created
```

查看该 Pod 的资源信息：

```
$ kubectl get pods valid-pod --namespace=limit-example -o yaml | grep -C 6
resources
  uid: 3b1bfd7a-f53c-11e5-b066-64510658e388
spec:
  containers:
  - image: gcr.io/google_containers/serve_hostname
    imagePullPolicy: Always
    name: kubernetes-serve-hostname
    resources:
      limits:
        cpu: "1"
        memory: 512Mi
      requests:
        cpu: "1"
        memory: 512Mi
```

可以看到该 Pod 配置了明确的 Limits 和 Requests，因此该 Pod 不会使用 namespace limit-example 中定义的 default 和 defaultRequest。

需要注意的是，CPU Limits 强制配置这个选项在 Kubernetes 集群中默认是开启的；除非集群管理员在部署 kubelet 时，通过设置参数--cpu-cfs-quota=false 来关闭该限制：

```
$ kubelet --help
Usage of kubelet
....
--cpu-cfs-quota[=true]: Enable CPU CFS quota enforcement for containers that
specify CPU limits
$ kubelet --cpu-cfs-quota=false ...
```

如果集群管理员希望对整个集群中容器或者 Pod 配置的 Requests 和 Limits 做限制，那么可以通过配置 Kubernetes 的命名空间（namespace）上的 LimitRange（资源限制区间）来达到该目的。在 Kubernetes 集群中，如果 Pod 没有显式定义 Limits 和 Requests，那么 Kubernetes 系统会将该 Pod 所在的命名空间中定义的 LimitRange 的 default 和 defaultRequests 配置到该 Pod 上。

3. 资源的服务质量管理（Resource QoS）

本节对 Kubernetes 如何根据 Pod 的 Requests 和 Limits 配置来实现针对 Pod 的不同级别的资源服务质量控制（QoS）进行说明。

在 Kubernetes 的资源 QoS 体系中，需要保证高可靠性的 Pod 可以申请可靠资源，而一些不需要高可靠性的 Pod 可以申请可靠性较低或者不可靠的资源。在计算资源一节中，我们讲到了容器的资源配置分为 Requests 和 Limits，其中 Requests 是 Kubernetes 调度时能为容器提供的完

全可保障的资源量（最低保障），而 Limits 是系统允许容器运行时可能使用到的资源量的上限（最高上限）。Pod 级别的资源配置是通过计算 Pod 内所有容器的资源配置的总和得出来的。

Kubernetes 中 Pod 的 Requests 和 Limits 资源配置有如下特点：如果 Pod 配置的 Requests 值等于 Limits 值，那么该 Pod 可以获得的资源是完全可靠的；而如果 Pod 的 Requests 值小于 Limits 值，那么该 Pod 获得的资源可分成两部分：一部分是完全可靠的资源，资源量大小等于 Requests 值；另外一部分是不可靠的资源，这部分资源最大等于 Limits 与 Requests 的差额值，这份不可靠的资源能够申请到多少，则取决于当时主机上容器可用资源的余量。

通过这种机制，Kubernetes 可以实现节点资源的超售（Over Subscription），比如在 CPU 完全充足的情况下，某机器共有 32GiB 内存可提供给容器使用，容器配置为 Requests 值 1GiB，Limits 值为 2GiB，那么该机器上最多可以同时运行 32 个容器，每个容器最多可使用 2GiB 内存，如果这些容器的内存使用峰值错开，那么所有容器也可以一直正常运行。

超售机制能有效地提高资源的利用率，同时不会影响容器申请的完全可靠资源的可靠性。

1）Requests 和 Limits 对不同计算资源类型的限制机制

根据计算资源章节的内容我们知道，容器的资源配置满足以下两个条件。

- Requests<=节点可用资源。
- Requests<=Limits。

Kubernetes 根据 Pod 配置的 Requests 值来调度 Pod，Pod 在成功调度之后会得到 Requests 值定义的资源来运行；而如果 Pod 所在机器上的资源有空余，则 Pod 可以申请更多的资源，最多不能超过 Limits 的值。我们下面看一下 Requests 和 Limits 针对不同计算资源类型的限制机制的差异。这种差异主要取决于计算资源类型是可压缩资源还是不可压缩资源。

（1）可压缩资源

- Kubernetes 目前支持的可压缩资源是 CPU。
- Pod 可以得到 Pod 的 Requests 配置的 CPU 使用量，而是否能使用超过 Requests 值的部分取决于系统的负载和调度。不过由于目前 Kubernetes 和 Docker 的 CPU 隔离机制都是在容器级别隔离的，所以 Pod 级别的资源配置并不能完全得到保障；Pod 级别的 cgroups 正在紧锣密鼓地开发中，如果将来引入，就可以确保 Pod 级别的资源配置准确运行。
- 空闲 CPU 资源按照容器 Requests 值的比例分配。举例说明：容器 A 的 CPU 配置为 Requests 1 Limits 10，容器 B 的 CPU 配置为 request 2 Limits 8，A 和 B 同时运行在一个节点上，初始状态下容器的可用 CPU 为 3cores，那么 A 和 B 恰好得到它们的 Requests 中定义的 CPU 用量，即 1CPU 和 2CPU。如果 A 和 B 都需要更多的 CPU 资源，而恰

好此时系统的其他任务释放出 1.5CPU，那么这 1.5CPU 将按照 A 和 B 的 Requests 值的比例 1∶2 分配给 A 和 B，即最终 A 可使用 1.5CPU，B 可使用 3CPU。

- 如果 Pod 使用了超过 Limits 10 中配置的 CPU 用量，那么 cgroups 会对 Pod 中的容器的 CPU 使用进行限流（throttled）；如果 Pod 没有配置 Limits 10，那么 Pod 会尝试抢占所有空闲的 CPU 资源（Kubernetes 从 1.2 版本开始默认开启--cpu-cfs-quota，因此默认情况下必须配置 Limits）。

（2）不可压缩资源

- Kubernetes 目前支持的可压缩资源是内存。
- Pod 可以得到 Requests 中配置的内存。如果 Pod 使用的内存量小于它的 Requests 的配置，那么这个 Pod 可以正常运行（除非出现操作系统级别的内存不足等严重问题）；如果 Pod 使用的内存量超过了它的 Requests 的配置，那么这个 Pod 有可能被 Kubernetes “杀掉”：比如 Pod A 使用了超过 Requests 而不到 Limits 的内存量，此时同一机器上另外一个 Pod B 之前只使用了远少于自己的 Requests 值的内存，而此时程序压力增大，Pod B 向系统申请的总量不超过自己的 Requests 值的内存，那么 Kubernetes 可能会直接杀掉 Pod A；另外一种情况是 Pod A 使用了超过 Requests 而不到 Limits 的内存量，此时 Kubernetes 将一个新的 Pod 调度到这台机器上，新的 Pod 需要使用内存，而只有 Pod A 使用了超过了自己的 Requests 值的内存，那么 Kubernetes 也可能会杀掉 Pod A 来释放内存资源。
- 如果 Pod 使用的内存量超过了它的 Limits 设置，那么操作系统内核会杀掉 Pod 所有容器的所有进程中使用内存最多的一个，直到内存不超过 Limits 为止。

2）对调度策略的影响

- Kubernetes 的 kubelet 通过计算 Pod 中所有容器的 Requests 的总和来决定对 Pod 的调度。
- 不管是 CPU 还是内存，Kubernetes 调度器和 kubelet 都会确保节点上所有 Pod 的 Requests 的总和不会超过该节点上可分配给容器使用的资源容量上限。

3）服务质量等级（QoS Classes）

在一个超用（Over Committed，即容器 Limits 总和大于系统容量上限）系统中，由于容器负载的波动可能导致操作系统的资源不足，最终可能会导致部分容器被“杀掉”。在这种情况下，我们当然会希望优先“杀掉”那些不太重要的容器，那么如何衡量重要程度呢？Kubernetes 将容器划分成 3 个 QoS 等级：Guaranteed（完全可靠的）、Burstable（弹性波动、较可靠的）和 Best-Effort（尽力而为、不太可靠的），这三种优先级依次递减，如图 5.4 所示。

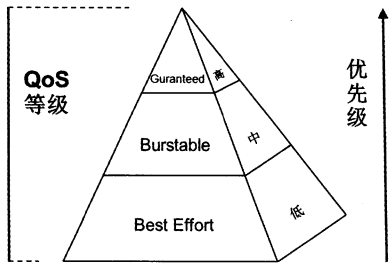

图 5.4　QoS 等级和优先级的关系

从理论上来说，QoS 级别应该作为一个单独的参数来提供 API，并由用户对 Pod 进行配置，这种配置应该与 Requests 和 Limits 无关。但在当前版本的 Kubernetes 的设计中，为了简化模式及避免引入太多的复杂性，QoS 级别直接由 Requests 和 Limits 来定义。在 Kubernetes 中容器的 QoS 级别等于容器所在 Pod 的 QoS 级别，而 Kubernetes 的资源配置定义了 Pod 的三种 QoS 级别，如下所述。

1）Guaranteed（完全可靠的）

如果 Pod 中的所有容器对所有资源类型都定义了 Limits 和 Requests，并且所有容器的 Limits 值都和 Requests 值全部相等（且都不为 0），那么该 Pod 的 QoS 级别就是 Guaranteed。注意：在这种情况下，容器可以不定义 Requests，因为 Requests 值在未定义的时候默认等于 Limits。

下面这两个例子中定义的 Pod QoS 级别就是 Guaranteed:

```
containers:
   name: foo
      resources:
         limits:
            cpu: 10m
            memory: 1Gi
   name: bar
      resources:
         limits:
            cpu: 100m
            memory: 100Mi
```

在上面的例子中未定义 Requests 值，所以其默认等于 Limits 值。而下面这个例子中定义的 Requests 和 Limits 的值完全相同：

```
containers:
   name: foo
      resources:
         limits:
            cpu: 10m
            memory: 1Gi
```

```
            requests:
               cpu: 10m
               memory: 1Gi
      name: bar
         resources:
            limits:
               cpu: 100m
               memory: 100Mi
            requests:
               cpu: 10m
               memory: 1Gi
```

2）Best-Effort（尽力而为、不太可靠的）

如果 Pod 中所有容器都未定义资源配置（Requests 和 Limits 都未定义），那么该 Pod 的 QoS 级别就是 Best-Effort。

例如下面这个 Pod 定义：

```
containers:
   name: foo
      resources:
   name: bar
      resources:
```

3）Burstable（弹性波动、较可靠的）

当一个 Pod 既不是 Guaranteed 级别的，也不是 Best-Effort 级别的时，该 Pod 的 QoS 级别就是 Burstable。Burstable 级别的 Pod 包括两种情况。第 1 种情况是：Pod 中的一部分容器在一种或多种资源类型的资源配置中，定义了 Requests 值和 Limits 值（都不为 0），且 Requests 值小于 Limits 值；第 2 种情况是：Pod 中的一部分容器未定义资源配置（Requests 和 Limits 都未定义）。注意：容器未定义 Limits 时，Limits 值默认等于节点资源容量上限。

下面几个例子中的 Pod 的 QoS 等级都是 Burstable。

（1）容器 foo 的 CPU Requests 不等于 Limits：

```
containers:
   name: foo
      resources:
         limits:
            cpu: 10m
            memory: 1Gi
         requests:
            cpu: 5m
            memory: 1Gi
   name: bar
      resources:
```