```
        limits:
            cpu: 10m
            memory: 1Gi
        requests:
            cpu: 10m
            memory: 1Gi
```

（2）容器 bar 未定义资源配置而容器 foo 定义了资源配置：

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

（3）容器 foo 未定义 CPU，而容器 bar 未定义内存：

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

（4）容器 bar 未定义资源配置，而容器 foo 未定义 Limits 值：

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

4）Kubernetes QoS 的工作特点

Pod 的 CPU Requests 无法得到满足（比如节点的系统级任务占用过多的 CPU 导致无法分配给足够的 CPU 给容器使用）时，容器得到的 CPU 会被压缩限流。

内存由于是不可压缩资源，所以针对内存资源紧缺的情况，将按照以下逻辑进行处理。

（1）Best-Effort Pod 的优先级最低，这类 Pod 中运行的进程会在系统内存紧缺时被第一优先

“杀死”。当然，从另外一个角度来看，Best-Effort Pod 由于没有设置资源 Limits，所以在资源充足的时候，它们可以充分地使用所有的闲置资源。

（2）Burstable Pod 的优先级居中，这类 Pod 初始时会分配较少的可靠资源，但可以按需申请更多的资源。当然，如果整个系统内存紧缺，而又没有 Best-Effort 容器可以被“杀死”以释放资源，则这类 Pod 中的进程可能会被“杀死”。

（3）Guaranteed Pod 的优先级最高，而且一般情况下这类 Pod 只要不超过其资源 Limits 的限制就不会被“杀死”。当然，如果整个系统内存紧缺，而又没有其他更低优先级的容器可以被“杀死”以释放资源，这类 Pod 中的进程也可能会被“杀死”。

5）OOM 计分系统

OOM（Out Of Memory）计分规则包括如下内容。

- OOM 计分是一个进程消耗内存在系统中占的百分比中不含百分号的数字的值乘以 10 的结果，这个结果是进程 OOM 基础分；将进程 OOM 基础分的分值再加上这个进程的 OOM 分数调整值 OOM_SCORE_ADJ 的值作为进程 OOM 最终分值（除 root 启动的进程外）。在系统发生 OOM 时，OOM Killer 会优先杀掉 OOM 计分更高的进程。
- 进程的 OOM 计分的基本分数值范围是 0～1000，如果 A 进程的调整值 OOM_SCORE_ADJ 减去 B 进程的调整值的结果大于 1000，那么 A 进程的 OOM 计分最终值必然大于 B 进程，A 进程会比 B 进程优先被杀死。
- 不论调整值 OOM_SCORE_ADJ 为多少，任何进程的最终分值范围也是 0～1000。

在 Kubernetes，不同 QoS 的 OOM 计分调整值规则如表 5.1 所示。

表 5.1　不同 QoS 的 OOM 计分调整值

QoS 等级	oom_score_adj
Guaranteed	-998
BestEffort	1000
Burstable	min(max(2, 1000 - (1000 * memoryRequestBytes) / machineMemoryCapacityBytes), 999)

- Best-effort Pod 设置 OOM_SCORE_ADJ 调整值为 1000，因此 Best-effort Pod 中的容器里面的所有进程的 OOM 最终分肯定是 1000。
- Guaranteed Pod 设置 OOM_SCORE_ADJ 调整值为-998，因此 Guaranteed Pod 中的容器里面的所有进程的 OOM 最终分一般为 0 或者 1（因为基础分不可能为 1000）。
- Burstable Pod 规则分情况说明：如果 Burstable Pod 的内存 Requests 超过了系统可用内存的 99.8%，那么这个 Pod 的 OOM_SCORE_ADJ 调整值固定为 2；否则，设置

OOM_SCORE_ADJ 调整值为 1000 - 10（内存 Requests 占系统可用内存的百分比的无百分号的数字部分的值），而如果内存 Requests 为 0，那么 OOM_SCORE_ADJ 调整值固定为 999。这样的规则能确保 OOM_SCORE_ADJ 调整值的范围为 2～999，而 Burstable Pod 中所有进程的 OOM 最终分数范围为 2～1000。Burstable Pod 进程的 OOM 最终分数始终大于 Guaranteed Pod 的进程得分，因此它们会被优先“杀死”。如果一个 Burstable Pod 使用的内存比它的内存 Requests 少，那么可以肯定的是它的所有进程的 OOM 最终分数会小于 1000，此时能确保它的优先级高于 Best-effort Pod。如果一个 Burstable Pod 的某个容器中某个进程使用的内存比容器的 request 值高，那么这个进程的 OOM 最终分数会是 1000，否则它的 OOM 最终分会小于 1000。假设下面容器中有一个占用内存非常大的进程，那么当一个使用内存超过其 Requests 的 Burstable Pod 与另外一个使用内存少于其 Requests 的 Burstable Pod 发生内存竞争冲突时，前者的进程会被系统“杀掉”。如果一个 Burstable Pod 内部有多个进程的多个容器发生内存竞争冲突，那么此时 OOM 评分只能作为参考，不能保证完全按照资源配置的定义来执行 OOM Kill。

OOM 还有一些特殊的计分规则，如下所述。

- kubelet 进程和 Docker 进程的调整值 OOM_SCORE_ADJ 为-998。
- 如果配置进程调整值 OOM_SCORE_ADJ 为-999，那么这类进程不会被 OOM Killer“杀掉”。

6）QoS 的演进

目前 Kubernetes 基于 QoS 的超用机制日趋完善，但还有一些问题需要解决。

7）内存 Swap 的支持

当前的 QoS 策略都是假定主机不启用内存 Swap。如果主机启用了 Swap，那么上面的 QoS 策略可能会失效。举例说明：两个 Guaranteed Pod 都刚好达到了内存 Limits，那么由于内存 Swap 机制，它们还可以继续申请使用更多的内存。如果 Swap 空间不足，那么最终这两个 Pod 中的进程可能会被“杀掉”。由于 Kubernetes 和 Docker 尚不支持内存 Swap 空间的隔离机制，所以这一功能暂时还未实现。

8）更丰富的 QoS 策略

当前的 QoS 策略都是基于 Pod 的资源配置（Requests 和 Limits）来定义的，而资源配置本身又承担着对 Pod 资源管理和限制的功能。两种不同维度的功能使用同一个参数来配置，可能会导致某些复杂需求无法满足，比如当前 Kubernetes 无法支持弹性的、高优先级的 Pod。自定义 QoS 优先级能提供更大的灵活性，完美地实现各类需求，但同时会引入更高的复杂性，而且过于灵活的设置会给予用户过高的权限，对系统管理也提出了更大的挑战。

4. 资源的配额管理（Resource Quotas）

如果一个 Kubernetes 集群被多个用户或者多个团队共享使用，那么就需要考虑共享时对资源公平使用的问题，因为某个用户可能会使用超过基于公平原则分配给其的资源量。

资源配额（Resource Quotas）就是解决这个问题的工具。通过 ResourceQuota 对象，我们可以定义一项资源配额，这个资源配额可以为每一个命名空间（namespace）提供一个总体的资源使用的限制：它可以限制命名空间中某种类型的对象的总数目上限，也可以设置命名空间中 Pod 可以使用到的计算资源的总上限。

典型的资源配额（Resource Quotas）使用方式如下。

- 不同的团队工作在不同的命名空间下，目前这个是非约束性的，未来版本中可能会通过 ACLs（访问控制列表 Access Control List）的方式来实现强制性约束。
- 集群管理员为集群中的每个命名空间创建一个或者多个资源配额项。
- 当用户在命名空间中使用资源（创建 Pod 或者 Service 等）时，Kubernetes 的配额系统会统计、监控和检查资源用量，以确保使用的资源用量没有超过资源配额的配置。
- 如果创建或者更新应用时，资源使用超过了某项资源配额的限制，那么创建或者更新的请求会报错（HTTP 403 Forbidden），错误信息给出详细的出错原因说明。
- 如果命名空间中的计算资源（CPU 和内存）的资源配额启用，那么用户必须为相应的资源类型设置 Requests 或 Limits；否则配额系统可能会直接拒绝 Pod 的创建。这里可以使用 LimitRange 机制来为没有配置资源的 Pod 提供默认资源配置。

下面的例子展示了一个非常适合使用资源配额来做资源控制管理的场景。

- 集群共有 32GB 内存和 16 CPU，两个小组，A 小组使用 20GB 内存和 10 CPU，B 小组使用 10GB 内存和 2 CPU，剩下的 2GB 内存和 2 CPU 作为预留。
- 在名为 testing 的命名空间中，限制使用 1 CPU 和 1GB 内存；在名为 production 的命名空间中，资源使用不受限制。

在使用资源配额时，需要注意以下两点。

- 如果集群中总的可用资源小于各命名空间中资源配额的总和，那么可能会导致资源竞争。资源竞争时，Kubernetes 系统使用先到先得的原则。
- 不管是资源竞争还是配额的修改都不会影响到已经创建的资源使用对象。

1）在 Master 中开启资源配额选型

资源配额可以通过在 kube-apiserver 的--admission-control=参数值中添加 ResourceQuota 参数进行开启。如果某个命名空间的定义中存在 ResourceQuota，那么对于该命名空间而言，资源配

额就是开启的。一个命名空间可以有多个 ResourceQuota 配置项。

（1）计算资源配额（Compute Resource Quota）

资源配额可以限制一个命名空间中所有 Pod 的计算资源的总和。表 5.2 列出了目前 Kubernetes 资源配额支持限制的计算资源类型。

表 5.2　ResourceQuota 的计算资源类型

资 源 名 称	说　明
cpu	所有非终止状态的 Pod，CPU Requests 的总和不能超过该值
limits.cpu	所有非终止状态的 Pod，CPU Limits 的总和不能超过该值
limits.memory	所有非终止状态的 Pod，内存 Limits 的总和不能超过该值
memory	所有非终止状态的 Pod，内存 Requests 的总和不能超过该值
requests.cpu	所有非终止状态的 Pod，CPU Requests 的总和不能超过该值
requests.memory	所有非终止状态的 Pod，内存 Requests 的总和不能超过该值

（2）对象数量配额（Object Count Quota）

指定类型的对象数量可以被限制。表 5.3 列出了 Kubernetes 资源配额支持限制对象数量的对象类型。

表 5.3　ResourceQuota 的对象类型

资 源 名 称	说　明
configmaps	在该命名空间中，能存在的 ConfigMap 的总数上限
persistentvolumeclaims	在该命名空间中，能存在的持久卷的总数上限
pods	在该命名空间中，能存在的非终止状态 Pod 的总数上限。Pod 终止状态等价于 Pod 的 status.phase 状态值为 Failed 或者 Succeed is true
replicationcontrollers	在该命名空间中，能存在的 RC 的总数上限
resourcequotas	在该命名空间中，能存在的资源配额项（ResourcesQuota）的总数上限
services	在该命名空间中，能存在的 service 的总数上限
services.loadbalancers	在该命名空间中，能存在的负载均衡（LoadBalancer）的总数上限
services.nodeports	在该命名空间中，能存在的 NodePort 的总数上限
secrets	在该命名空间中，能存在的 Secret 的总数上限

例如我们可以通过资源配额来限制命名空间中能创建的 Pod 的最大数量。这种设置可以防止某些用户大量创建 Pod 而迅速耗尽整个集群的 Pod IP 和计算资源。

2）配额的作用域（Quota Scopes）

每项资源配额都可以单独配置一组作用域，配置了作用域的资源配额只会对符合其作用域的资源使用进行计量和限制，作用域范围内的且超过了资源配额的请求都会报验证错。表 5.4 列出了 ResourceQuota 的 4 种作用域。

表 5.4 ResourceQuota 的作用域

作 用 域	说 明
Terminating	匹配所有 spec.activeDeadlineSeconds >= 0 的 Pod
NotTerminating	匹配所有 spec.activeDeadlineSeconds 是 nil 的 Pod
BestEffort	匹配所有 QoS 为 Best-Effort 的 Pod
NotBestEffort	匹配所有 QoS 不是 Best-Effort 的 Pod

其中，BestEffort 作用域可以限定资源配额来追踪 pods 资源的使用，Terminating、NotTerminating 和 NotBestEffort 这三种作用域可以限定资源配额来追踪以下资源的使用。

- cpu
- limits.cpu
- limits.memory
- memory
- pods
- requests.cpu
- requests.memory

3）在资源配额（ResourceQuota）中设置 Requests 和 Limits

资源配额也可以设置 Requests 和 Limits。

如果资源配额中指定了 requests.cpu 或 requests.memory，那么它会强制要求每一个容器都必须配置自己的 CPU Requests 或 CPU Limits（可使用 LimitRange 提供的默认值）。

同理，如果资源配额中指定了 limits.cpu 或 limits.memory，那么它也会强制要求每一个容器都必须配置自己的内存 Requests 或内存 Limits（可使用 LimitRange 提供的默认值）。

4）资源配额（ResourceQuota）的定义

下面通过几个例子对资源配额进行设置和应用。

与 LimitRange 相似，ResourceQuota 也设置在 namespace 中。创建名为 myspace 的 namespace：

```
$ kubectl create namespace myspace
namespace "myspace" created
```

创建 ResourceQuota 配置文件 compute-resources.yaml，用于设置计算资源的配额：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
```

```
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
limits.memory: 2Gi
```

创建该项资源配额：

```
$ kubectl create -f compute-resources.yaml --namespace=myspace
resourcequota "compute-resources" created
```

创建另一个名为 object-counts.yaml 的文件，用于设置对象数量的配额：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"
```

创建该 ResourceQuota：

```
$ kubectl create -f object-counts.yaml --namespace=myspace
resourcequota "object-counts" created
```

查看各 ResourceQuota 的详细信息：

```
$ kubectl describe quota compute-resources --namespace=myspace
Name:                compute-resources
Namespace:           myspace
Resource             Used Hard
--------              ---- ----
limits.cpu            0    2
limits.memory         0    2Gi
pods                  0    4
requests.cpu          0    1
requests.memory       0    1Gi

$ kubectl describe quota object-counts --namespace=myspace
Name:                   object-counts
Namespace:              myspace
Resource                Used    Hard
--------                ----    ----
configmaps              0       10
```

```
persistentvolumeclaims  0      4
replicationcontrollers  0      20
secrets                 1      10
services                0      10
services.loadbalancers  0      2
```

5）资源配额与集群资源总量的关系

资源配额与集群资源总量是完全独立的。资源配额是通过绝对的单位来配置的：这也就意味着如果集群中新添加了节点，那么资源配额不会自动更新，而该资源配额所对应的命名空间下对象也不能自动地增加资源上限。

在某些情况下，我们可能希望资源配额能支持更复杂的策略，如下所述。

- 对于不同的租户，按照比例划分整个集群的资源。
- 允许每个租户都能按照需要来提高资源用量，但是有一个较宽容的限制，以防止意外的资源耗尽情况发生。
- 探测某个命名空间的需求，添加物理节点并扩大资源配额值。

这些策略可以通过将资源配额作为一个控制模块、手动编写一个控制器（controller）来监控资源使用情况，并调整命名空间上的资源配额的方式来实现。

资源配额将整个集群中的资源总量做了一个静态的划分，但它并没有对集群中的节点（Node）做任何限制：不同命名空间中的 Pod 仍然可以运行到同一个节点上。

5. ResourceQuota 和 LimitRange 实践指南

根据前面对资源管理的介绍，这里将通过一个完整的例子来说明如何通过资源配额和资源配置范围的配合来控制一个命名空间的资源使用。

集群管理员根据集群用户数量来调整集群配置，以达到如下目的：能控制特定命名空间中的资源使用量，最终实现集群的公平使用和成本的控制。

需要实现的功能如下。

- 限制运行状态的 Pod 的计算资源用量。
- 限制持久存储卷的数量以控制对存储的访问。
- 限制负载均衡器的数量以控制成本。
- 防止滥用网络端口这类稀缺资源。
- 提供默认的计算资源 Requests 以便于系统做出更优化的调度。

1）创建命名空间

创建名为 quota-example 的命名空间，namespace.yaml 文件的内容如下：

```
apiVersion: v1
kind: Namespace
metadata:
  name: quota-example

$ kubectl create -f namespace.yaml
namespace "quota-example" created
```

查看命名空间：

```
$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    2m
kube-system   Active    2m
quota-example  Active   39s
```

2）设置限定对象数目的资源配额

通过设置限定对象的数量的资源配额，可以控制以下资源的数量：

- 持久存储卷；
- 负载均衡器；
- NodePort。

创建名为 object-counts 的 ResourceQuota：

object-counts.yaml：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    persistentvolumeclaims: "2"
    services.loadbalancers: "2"
services.nodeports: "0"


$ kubectl create -f object-counts.yaml --namespace=quota-example
resourcequota "object-counts" created
```

配额系统会检测到资源项配额的创建，并且将会统计和限制该命名空间中的资源消耗。

查看该配额是否生效：

```
$ kubectl describe quota object-counts --namespace=quota-example
```

```
Name:                  object-counts
Namespace:              quota-example
Resource               Used    Hard
--------               ----    ----
persistentvolumeclaims 0       2
services.loadbalancers 0       2
services.nodeports     0       0
```

至此，配额系统会自动阻止那些使资源用量超过资源配额限定值的请求。

3）设置限定计算资源的资源配额

下面我们再来创建一项限定计算资源的资源配额，以限制该命名空间中的计算资源的使用总量。

创建名为 compute-resources 的 ResourceQuota：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
limits.memory: 2Gi

$ kubectl create -f compute-resources.yaml --namespace=quota-example
resourcequota "compute-resources" created
```

查看该配额是否生效：

```
$ kubectl describe quota compute-resources --namespace=quota-example
Name:                  compute-resources
Namespace:              quota-example
Resource               Used Hard
--------               ---- ----
limits.cpu             0    2
limits.memory          0    2Gi
pods                   0    4
requests.cpu           0    1
requests.memory        0    1Gi
```

配额系统会自动防止该命名空间下同时拥有超过 4 个非“终止态”的 Pod。此外，由于该项资源配额限制了 CPU 和内存的 Limits 和 Requests 的总量，因此会强制要求该命名空间下的所有容器都必须显示地定义 CPU 和内存的 Limits 和 Requests（可使用默认值，Requests 默认等

于 Limits）。

4）配置默认 Requests 和 Limits

在命名空间已经配置了限定计算资源的资源配额的情况下，如果尝试在该命名空间下创建一个不指定 Requests 和 Limits 的 Pod，那么 Pod 的创建可能会失败。下面是一个失败的例子。

创建一个 Nginx 的 Deployment：

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

查看创建的 Pod，会发现 Pod 没有创建成功：

```
$ kubectl get pods --namespace=quota-example
```

再查看一下 Deployment 的详细信息：

```
$ kubectl describe deployment nginx --namespace=quota-example
Name:                   nginx
Namespace:              quota-example
CreationTimestamp:      Mon, 06 Jun 2016 16:11:37 -0400
Labels:                 run=nginx
Selector:               run=nginx
Replicas:               0 updated | 1 total | 0 available | 1 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:         <none>
NewReplicaSet:          nginx-3137573019 (0/1 replicas created)
...
```

本 Deployment 尝试创建一个 Pod，但是失败了，查看其中 ReplicaSet 的详细信息：

```
$ kubectl describe rs nginx-3137573019 --namespace=quota-example
Name:                   nginx-3137573019
Namespace:              quota-example
Image(s):               nginx
Selector:               pod-template-hash=3137573019,run=nginx
Labels:                 pod-template-hash=3137573019
                        run=nginx
Replicas:               0 current / 1 desired
Pods Status:            0 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen    LastSeen    Count From                SubobjectPath Type
Reason       Message
  ---------    --------    ----- ----                ----------------------
------       -------
  4m           7s          11    {replicaset-controller }          Warning
FailedCreate  Error creating: pods "nginx-3137573019-" is forbidden: Failed quota:
```

```
compute-resources: must specify
limits.cpu,limits.memory,requests.cpu,requests.memory
```

可以看到 Pod 创建失败的原因：Master 拒绝了这个 ReplicaSet 创建 Pod，因为这个 Pod 中没有指定 CPU 和内存的 Requests 和 Limits。

为了避免这种失败，我们可以使用 LimitRange 来为这个命名空间下的所有 Pod 提供一个资源配置的默认值。下面的例子展示了如何为这个命名空间添加一个指定默认资源配置的 LimitRange。

创建一个名为 limits 的 LimitRange：

```
limits.yaml:
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
spec:
  limits:
  - default:
      cpu: 200m
      memory: 512Mi
    defaultRequest:
      cpu: 100m
      memory: 256Mi
    type: Container

$ kubectl create -f limits.yaml --namespace=quota-example
limitrange "limits" created

$ kubectl describe limits limits --namespace=quota-example
Name:          limits
Namespace:     quota-example
Type      Resource  Min  Max  Default Request   Default Limit   Max Limit/Request
Ratio
----      --------  ---  ---  ---------------   -------------
-----------------------
Container memory    -    -    256Mi             512Mi           -
Container cpu       -    -    100m              200m            -
```

LimitRange 创建成功后，用户在该命名空间下的创建未指定资源配置的 Pod 的请求时，系统会自动为该 Pod 设置默认的资源配置。

例如，每个新建的未指定资源配置的 Pod 都等价于使用下面的资源配置：

```
$ kubectl run nginx \
  --image=nginx \
  --replicas=1 \
```

```
  --requests=cpu=100m,memory=256Mi \
  --limits=cpu=200m,memory=512Mi \
  --namespace=quota-example
```

至此，我们已经为该命名空间配置好了默认的计算资源，我们的 ReplicaSet 应该能够创建 Pod 了。查看一下，创建 Pod 成功了：

```
$ kubectl get pods --namespace=quota-example
NAME                    READY     STATUS    RESTARTS   AGE
nginx-3137573019-fvrig   1/1       Running   0          6m
```

接下来，还可以随时查看资源配额的使用情况：

```
$ kubectl describe quota --namespace=quota-example
Name:          compute-resources
Namespace:     quota-example
Resource       Used    Hard
--------       ----    ----
limits.cpu     200m    2
limits.memory   512Mi   2Gi
pods           1       4
requests.cpu    100m    1
requests.memory 256Mi   1Gi


Name:               object-counts
Namespace:          quota-example
Resource            Used    Hard
--------            ----    ----
persistentvolumeclaims 0     2
services.loadbalancers 0     2
services.nodeports     0     0
```

可以看到每个 Pod 创建时都会消耗掉指定的资源量，而这些使用量都会被 Kubernetes 准确地跟踪、监控和管理。

5）指定资源配额的作用域

假设我们并不想为某个命名空间配置默认的计算资源配额，而是希望限定在命名空间内运行的 QoS 为 BestEffort 的 Pod 总数，例如将集群中的部分资源用来运行 QoS 为非 BestEffort 的服务，而将闲置的资源用来运行 QoS 为 BestEffort 的服务，即可避免集群的所有资源仅被大量的 BestEffort Pod 耗尽。这可以通过创建两个资源配额（ResourceQuota）来实现。

首先创建一个名为 quota-scopes 的命名空间：

```
$ kubectl create namespace quota-scopes
namespace "quota-scopes" created
```

创建一个名为 best-effort 的 ResourceQuota，指定 Scope 为 BestEffort：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: best-effort
spec:
  hard:
    pods: "10"
  scopes:
  - BestEffort
```

```
$ kubectl create -f best-effort.yaml --namespace=quota-scopes
resourcequota "best-effort" created
```

再创建一个名为 not-best-effort 的 ResourceQuota，指定 Scope 为 NotBestEffort：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: not-best-effort
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
  scopes:
  - NotBestEffort
```

```
$ kubectl create -f not-best-effort.yaml --namespace=quota-scopes
resourcequota "not-best-effort" created
```

查看创建成功的 ResourceQuota：

```
$ kubectl describe quota --namespace=quota-scopes
Name:       best-effort
Namespace:  quota-scopes
Scopes:     BestEffort
 * Matches all pods that have best effort quality of service.
Resource    Used  Hard
--------    ----  ----
pods        0     10

Name:            not-best-effort
Namespace:       quota-scopes
Scopes:          NotBestEffort
 * Matches all pods that do not have best effort quality of service.
Resource         Used  Hard
--------         ----  ----
```

```
limits.cpu        0     2
limits.memory     0     2Gi
pods              0     4
requests.cpu      0     1
requests.memory   0     1Gi
```

之后，对于没有配置 Requests 的 Pod 将会被名为 best-effort 的 ResourceQuota 所限制；而配置了 Requests 的 Pod 会被名为 not-best-effort 的 ResourceQuota 所限制。

创建两个 Deployment：

```
$ kubectl run best-effort-nginx --image=nginx --replicas=8
--namespace=quota-scopes
deployment "best-effort-nginx" created

$ kubectl run not-best-effort-nginx \
  --image=nginx \
  --replicas=2 \
  --requests=cpu=100m,memory=256Mi \
  --limits=cpu=200m,memory=512Mi \
  --namespace=quota-scopes
deployment "not-best-effort-nginx" created
```

名为 best-effort-nginx 的 Deployment 因为没有配置 Requests 和 Limits，所以它的 QoS 级别为 BestEffort，因此它的创建过程由 best-effort 资源配额项来限制，而 not-best-effort 资源配额项不会对它进行限制。best-effort 资源配额项没有限制 Requests 和 Limits，因此 best-effort-nginx Deployment 可以成功地创建 8 个 Pod。

名为 not-best-effort-nginx 的 Deployment 因为配置了 Requests 和 Limits，且二者不相等，所以它的 QoS 级别为 Burstable ，因此它的创建过程由 not-best-effort 资源配额项来限制，而 best-effort 资源配额项不会对它进行限制。not-best-effort 资源配额项限制了 Pod 的 Requests 和 Limits 的总上限，not-best-effort-nginx Deployment 并没有超过这个上限，所以可以成功地创建两个 Pod。

查看已经创建的 Pod：

```
$ kubectl get pods --namespace=quota-scopes
NAME                                  READY     STATUS    RESTARTS   AGE
best-effort-nginx-3488455095-2qb41    1/1       Running   0          51s
best-effort-nginx-3488455095-3go7n    1/1       Running   0          51s
best-effort-nginx-3488455095-9o2xg    1/1       Running   0          51s
best-effort-nginx-3488455095-eyg40    1/1       Running   0          51s
best-effort-nginx-3488455095-gcs3v    1/1       Running   0          51s
best-effort-nginx-3488455095-rq8p1    1/1       Running   0          51s
best-effort-nginx-3488455095-udhhd    1/1       Running   0          51s
best-effort-nginx-3488455095-zmk12    1/1       Running   0          51s
not-best-effort-nginx-2204666826-7sl61   1/1       Running   0          23s
```

```
not-best-effort-nginx-2204666826-ke746   1/1       Running   0          23s
```

可以看到 10 个 Pod 都创建成功。

再看一下两个资源配额项的使用情况：

```
$ kubectl describe quota --namespace=quota-scopes
Name:          best-effort
Namespace:      quota-scopes
Scopes:        BestEffort
 * Matches all pods that have best effort quality of service.
Resource       Used  Hard
--------       ----  ----
pods           8     10


Name:            not-best-effort
Namespace:        quota-scopes
Scopes:          NotBestEffort
 * Matches all pods that do not have best effort quality of service.
Resource         Used  Hard
--------         ----  ----
limits.cpu       400m  2
limits.memory    1Gi   2Gi
pods             2     4
requests.cpu     200m  1
requests.memory  512Mi 1Gi
```

可以看到best-effort资源配额项已经统计到了best-effort-nginx Deployment中创建的8个Pod的资源使用信息，而not-best-effort资源配额项也统计到了not-best-effort-nginx Deployment中创建的两个Pod的资源使用信息。

通过这个例子我们可以看到：资源配额的作用域（Scopes）提供了一种将资源集合分割的机制，这种机制使得集群管理员可以更加方便地监控和限制不同类型对象对于各类资源的使用，同时能为资源分配和限制提供更大的灵活度和便利性。

6. 资源管理总结

Kubernetes中的资源管理的基础是容器和Pod的资源配置（Requests和Limits）。容器的资源配置（Requests和Limits）指定了容器请求的资源和容器能使用的资源上限，而Pod的资源配置则是Pod中所有容器的资源配置总和的上限。

通过资源配额（Resource Quota）机制，我们可以对命名空间下所有Pod使用资源的总量进行限制，也可以对这个命名空间中指定类型的对象的数量进行限制。使用作用域可以让资源配额只对符合特定范围的对象加以限制，因此作用域（Scopes）机制可以使资源配额的策略更加

丰富灵活。

如果我们需要对用户的 Pod 或容器的资源配置做更多的限制，则我们可以使用资源配置范围（LimitRange）来达到这个目的。LimitRange 可以有效地限制 Pod 和容器的资源配置的最大、最小范围，也可以限制 Pod 和容器的 Limits 与 Requests 的最大比例上限，此外 LimitRange 还可以为 Pod 中的容器提供默认的资源配置。

Kubernetes 基于 Pod 的资源配置（Requests 和 Limits）实现了资源服务质量（QoS）。不同 QoS 级别的 Pod 在系统中拥有不同的优先级：高优先级的 Pod 具有更高的可靠性，可以用于运行可靠性要求较高的服务；而低优先级的 Pod 可以实现集群资源的超售，能有效地提高集群资源利用率。

上面的多种机制共同组成了当前版本 Kubernetes 的资源管理体系。这个资源管理体系已经可以满足大部分资源管理的需求了。同时，Kubernetes 资源管理体系仍然在不停地发展和进化中，对于一些目前无法满足的更复杂、更个性化的需求，我们可以继续关注 Kubernetes 未来的发展和变化。

5.1.5　Kubernetes 集群高可用部署方案

Kubernetes 作为容器应用的管理平台，通过对 Pod 的运行状况进行监控，并且根据主机或容器失效的状态将新的 Pod 调度到其他 Node 上，实现了应用层的高可用性。针对 Kubernetes 集群，高可用性还应包含以下两个层面的考虑：etcd 数据存储的高可用性和 Kubernetes Master 组件的高可用性。

1. etcd 高可用部署

etcd 在整个 Kubernetes 集群中处于中心数据库的地位，为保证 Kubernetes 集群的高可用性，首先需要保证数据库不是单故障点。一方面，etcd 需要以集群的方式进行部署，以实现 etcd 数据存储的冗余、备份与高可用性；另一方面，etcd 存储的数据本身也应考虑使用可靠的存储设备。

etcd 集群的部署可以使用静态配置，也可以通过 etcd 提供的 REST API 在运行时动态添加、修改或删除集群中的成员。本节将对 etcd 集群的静态配置进行说明。关于动态修改的操作方法请参考 etcd 官方文档的说明。

首先，规划一个至少 3 台服务器（节点）的 etcd 集群，在每台服务器上安装好 etcd。

部署一个由 3 台服务器组成的 etcd 集群，其配置如表 5.5 所示，其集群部署实例如图 5.5 所示。

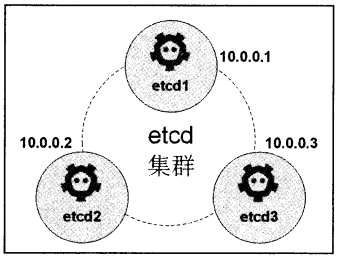

图 5.5　etcd 集群部署实例

表 5.5　etcd 集群的配置

etcd 实例名称	IP 地址
etcd1	10.0.0.1
etcd2	10.0.0.2
etcd3	10.0.0.3

然后修改每台服务器上 etcd 的配置文件/etc/etcd/etcd.conf。

以 etcd1 为创建集群的实例，需要将其 ETCD_INITIAL_CLUSTER_STATE 设置为“new”。etcd1 的完整配置如下：

```
# [member]
ETCD_NAME=etcd1                    #etcd 实例名称
ETCD_DATA_DIR="/var/lib/etcd"      #etcd 数据保存目录
ETCD_LISTEN_CLIENT_URLS="http://10.0.0.1:2379,http://127.0.0.1:2379"    #供外部客户端使用的 URL
ETCD_ADVERTISE_CLIENT_URLS="http://10.0.0.1:2379,http://127.0.0.1:2379"    #广播给外部客户端使用的 URL
#[cluster]
ETCD_LISTEN_PEER_URLS="http://10.0.0.1:2380"    #集群内部通信使用的 URL
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.0.0.1:2380"    #广播给集群内其他成员访问的 URL
ETCD_INITIAL_CLUSTER="etcd1=http://10.0.0.1:2380,etcd2=http://10.0.0.2:2380,etcd3=http://10.0.0.3:2380"      #初始集群成员列表
ETCD_INITIAL_CLUSTER_STATE="new"      #初始集群状态，new 为新建集群
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"    #集群名称
```

启动 etcd1 服务器上的 etcd 服务：

```
$ systemctl restart etcd
```

启动完成后，就创建了一个名为 etcd-cluster 的集群。

etcd2 和 etcd3 为加入 etcd-cluster 集群的实例，需要将其 ETCD_INITIAL_CLUSTER_STATE 设置为“exist”。etcd2 的完整配置如下（etcd3 的配置略）：

```
# [member]
ETCD_NAME=etcd2                    #etcd 实例名称
ETCD_DATA_DIR="/var/lib/etcd"      #etcd 数据保存目录
ETCD_LISTEN_CLIENT_URLS="http://10.0.0.2:2379,http://127.0.0.1:2379"    #供外部客户端使用的 URL
ETCD_ADVERTISE_CLIENT_URLS="http://10.0.0.2:2379,http://127.0.0.1:2379"    #广播给外部客户端使用的 URL
```

```
    #[cluster]
    ETCD_LISTEN_PEER_URLS="http://10.0.0.2:2380"    #集群内部通信使用的 URL
    ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.0.0.2:2380"    #广播给集群内其他成员
使用的 URL
    ETCD_INITIAL_CLUSTER="etcd1=http://10.0.0.1:2380,etcd2=http://10.0.0.2:2380,
etcd3=http://10.0.0.3:2380"      #初始集群成员列表
    ETCD_INITIAL_CLUSTER_STATE="new"          #初始集群状态，new 为新建集群
    ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"    #集群名称
```

启动 etcd2 和 etcd3 服务器上的 etcd 服务：

```
$ systemctl restart etcd
```

启动完成后，在任意 etcd 节点执行 etcdctl cluster-health 命令来查询集群的运行状态：

```
$ etcdctl cluster-health
cluster is healthy
member ce2a822cea30bfca is healthy
member acda82ba1cf790fc is healthy
member eba209cd0012cd2 is healthy
```

在任意 etcd 节点上执行 etcdctl member list 命令来查询集群的成员列表：

```
$ etcdctl member list
ce2a822cea30bfca: name=default peerURLs=http://10.0.0.1:2380,http://127.0.0.1:
7001 clientURLs=http://10.0.0.1:2379,http://127.0.0.1:2379
acda82ba1cf790fc: name=default peerURLs=http://10.0.0.2:2380,http://127.0.0.1:
7001 clientURLs=http://10.0.0.2:2379,http://127.0.0.1:2379
eba209cd40012cd2: name=default peerURLs=http://10.0.0.3:2380,http://127.0.0.1:
7001 clientURLs=http://10.0.0.3:2379,http://127.0.0.1:2379
```

至此，一个 etcd 集群就创建成功了。

以 kube-apiserver 为例，将访问 etcd 集群的参数设置为：

```
--etcd-servers=http://10.0.0.1:2379,http://10.0.0.2:2379,http://10.0.0.3:2379
```

在 etcd 集群成功启动之后，如果需要对集群成员进行修改，则请参考官方文档的详细说明：

```
https://github.com/coreos/etcd/blob/master/Documentation/runtime-configurati
on.md#cluster-reconfiguration-operations。
```

对于 etcd 中需要保存的数据的可靠性，可以考虑使用 RAID 磁盘阵列、高性能存储设备、共享存储文件系统，或者使用云服务商提供的存储系统等来实现。

2. Master 高可用部署

在 Kubernetes 系统中，Master 服务扮演着总控中心的角色，主要的三个服务 kube-apiserver、kube-controller-mansger 和 kube-scheduler 通过不断与工作节点上的 kubelet 和 kube-proxy 进行通信来维护整个集群的健康工作状态。如果 Master 的服务无法访问到某个 Node，则会将该 Node

标记为不可用，不再向其调度新建的 Pod。但对 Master 自身则需要进行额外的监控，使 Master 不成为集群的单故障点，所以对 Master 服务也需要进行高可用方式的部署。

以 Master 的 kube-apiserver、kube-controller-mansger 和 kube-scheduler 三个服务作为一个部署单元，类似于 etcd 集群的典型部署配置。使用至少三台服务器安装 Master 服务，并且需要保证任何时候总有一套 Master 能够正常工作。图 5.6 展示了一种典型的部署方式。

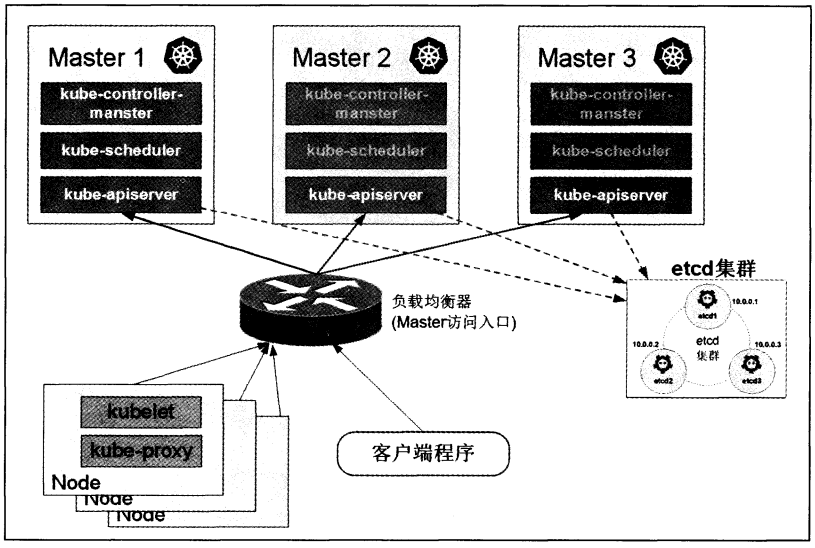

图 5.6　Kubernetes Master 高可用部署架构

Kubernetes 建议 Master 的 3 个组件都以容器的形式启动，启动它们的基础工具是 kubelet，所以它们都将以 Static Pod 的形式启动并由 kubelet 进行监控和自动重启。而 kubelet 本身的高可用则通过操作系统来完成，例如使用 Linux 的 Systemd 系统进行管理。

注意，如果之前已运行过这 3 个进程，则需要先停止它们，然后启动 kubelet 服务，这 3 个主进程将通过 kubelet 以容器的形式启动和运行。

接下来分别对 kube-apiserver 和 kube-controller-manager、kube-scheduler 的高可用部署进行说明。

1）kube-apiserver 的高可用部署

根据第 2 章的介绍，为 kube-apiserver 预先创建所有需要的 CA 证书和基本鉴权文件等内容，然后在每台服务器上创建其日志文件：

```
# touch /var/log/kube-apiserver.log
```

假设 kubelet 的启动参数指定--config=/etc/kubernetes/manifests，即 Static Pod 定义文件所在的目录，接下来就可以创建 kube-apiserver.yaml 配置文件用于启动 kube-apiserver 了。

```
    kube-apiserver.yaml
    apiVersion: v1
    kind: Pod
    metadata:
      name: kube-apiserver
    spec:
      hostNetwork: true
      containers:
      - name: kube-apiserver
        image:
gcr.io/google_containers/kube-apiserver:9680e782e08a1a1c94c656190011bd02
        command:
        - /bin/sh
        - -c
        - /usr/local/bin/kube-apiserver --etcd-servers=http://127.0.0.1:2379
    --admission-control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,Servi
ceAccount,ResourceQuota
          --service-cluster-ip-range=169.169.0.0/16 --v=2
          --allow-privileged=False 1>>/var/log/kube-apiserver.log 2>&1
        ports:
        - containerPort: 443
          hostPort: 443
          name: https
        - containerPort: 7080
          hostPort: 7080
          name: http
        - containerPort: 8080
          hostPort: 8080
          name: local
        volumeMounts:
        - mountPath: /srv/kubernetes
          name: srvkube
          readOnly: true
        - mountPath: /var/log/kube-apiserver.log
          name: logfile
        - mountPath: /etc/ssl
          name: etcssl
          readOnly: true
        - mountPath: /usr/share/ssl
          name: usrsharessl
          readOnly: true
        - mountPath: /var/ssl
          name: varssl
          readOnly: true
        - mountPath: /usr/ssl
          name: usrssl
          readOnly: true
```

```
    - mountPath: /usr/lib/ssl
      name: usrlibssl
      readOnly: true
    - mountPath: /usr/local/openssl
      name: usrlocalopenssl
      readOnly: true
    - mountPath: /etc/openssl
      name: etcopenssl
      readOnly: true
    - mountPath: /etc/pki/tls
      name: etcpkitls
      readOnly: true
  volumes:
  - hostPath:
      path: /srv/kubernetes
    name: srvkube
  - hostPath:
      path: /var/log/kube-apiserver.log
    name: logfile
  - hostPath:
      path: /etc/ssl
    name: etcssl
  - hostPath:
      path: /usr/share/ssl
    name: usrsharessl
  - hostPath:
      path: /var/ssl
    name: varssl
  - hostPath:
      path: /usr/ssl
    name: usrssl
  - hostPath:
      path: /usr/lib/ssl
    name: usrlibssl
  - hostPath:
      path: /usr/local/openssl
    name: usrlocalopenssl
  - hostPath:
      path: /etc/openssl
    name: etcopenssl
  - hostPath:
      path: /etc/pki/tls
    name: etcpkitls
```

其中，

◎ kube-apiserver 需要使用 hostNetwork 模式，即直接使用宿主机网络，以使得客户端能够

通过物理机访问其 API。

- 镜像的 tag 来源于 kubernetes 发布包中的 kube-apiserver.docker_tag 文件：kubernetes/server/kubernetes-server-linux-amd64/server/bin/kube-apiserver.docker_tag。
- --etcd-servers：指定 etcd 服务的 URL 地址。
- 再加上其他必要的启动参数，包括--admission-control、--service-cluster-ip-range、CA 证书相关配置等内容。
- 端口号的设置都配置了 hostPort，将容器内的端口号直接映射为宿主机的端口号。

将 kube-apiserver.yaml 文件复制到 kubelet 监控的/etc/kubernetes/manifests 目录下，kubelet 将会自动创建 yaml 文件中定义的 kube-apiserver 的 Pod。

接下来在另外两台服务器上重复该操作，使得每台服务器上都启动一个 kube-apiserver 的 Pod。

2）为 kube-apiserver 配置负载均衡器

至此，我们启动了三个 kube-apiserver 实例，这三个 kube-apiserver 都可以正常工作，我们需要一个统一的、可靠的、允许部分 Master 节点故障的方式来访问它们，可以通过部署一个负载均衡器来实现。

在不同的平台下，负载均衡的实现方式不同：在一些公用云比如 GCE、AWS、阿里云上都有现成的实现方案；对于本地集群，我们可以选择硬件或者软件来实现负载均衡，比如 Kubernetes 社区推荐的方案 haproxy 和 keepalived 来实现，其中 haproxy 做负载均衡，而 keepalived 负责对 haproxy 监控和进行高可用。

在完成 API Server 的负载均衡配置之后，对其访问还需要注意以下内容。

- 如果 Master 开启了安全认证机制，那么需要确保证书中包含负载均衡服务节点的 IP。
- 对于外部的访问，比如通过 kubectl 访问 API Server，那么需要配置为访问 API Server 对应的负载均衡器的 IP 地址。

3）kube-controller-manager 和 kube-scheduler 的高可用配置

不同于 API Server，Master 中另外两个核心组件 kube-controller-manager 和 kube-scheduler 会修改集群的状态信息，因此对于 kube-controller-manager 和 kube-scheduler 而言，高可用不仅意味着需要启动多个实例，还需要这多个实例能实现选举并选举出 leader，以保证同一时间只有一个实例可以对集群状态信息进行读写，避免出现同步问题和一致性问题。Kubernetes 对于这种选举机制的实现是采用租赁锁（lease-lock）来实现的，我们可以通过在 kube-controller-manager 和 kube-scheduler 的每个实例的启动参数中设置--leader-elect=true，来保证同一时间只会运行一个可修改集群信息的实例。

Scheduler 和 Controller Manager 高可用的具体实现方式如下。

首先在每个 Master 节点上创建相应的日志文件：

```
# touch /var/log/kube-scheduler.log
# touch /var/log/kube-controller-manager.log
```

然后创建 kube-controller-manager 和 kube-scheduler 的 Pod 定义文件：

kube-controller-manager.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: kube-controller-manager
spec:
  hostNetwork: true
  containers:
  - name: kube-controller-manager
    image: gcr.io/google_containers/kube-controller-manager:
fda24638d51a48baa13c35337fcd4793
    command:
    - /bin/sh
    - -c
    - /usr/local/bin/kube-controller-manager --master=127.0.0.1:8080
      --v=2 --leader-elect=true 1>>/var/log/kube-controller-manager.log 2>&1
    livenessProbe:
      httpGet:
        path: /healthz
        port: 10252
      initialDelaySeconds: 15
      timeoutSeconds: 1
    volumeMounts:
    - mountPath: /srv/kubernetes
      name: srvkube
      readOnly: true
    - mountPath: /var/log/kube-controller-manager.log
      name: logfile
    - mountPath: /etc/ssl
      name: etcssl
      readOnly: true
    - mountPath: /usr/share/ssl
      name: usrsharessl
      readOnly: true
    - mountPath: /var/ssl
      name: varssl
      readOnly: true
    - mountPath: /usr/ssl
      name: usrssl
```

```
    readOnly: true
  - mountPath: /usr/lib/ssl
    name: usrlibssl
    readOnly: true
  - mountPath: /usr/local/openssl
    name: usrlocalopenssl
    readOnly: true
  - mountPath: /etc/openssl
    name: etcopenssl
    readOnly: true
  - mountPath: /etc/pki/tls
    name: etcpkitls
    readOnly: true
volumes:
- hostPath:
    path: /srv/kubernetes
  name: srvkube
- hostPath:
    path: /var/log/kube-controller-manager.log
  name: logfile
- hostPath:
    path: /etc/ssl
  name: etcssl
- hostPath:
    path: /usr/share/ssl
  name: usrsharessl
- hostPath:
    path: /var/ssl
  name: varssl
- hostPath:
    path: /usr/ssl
  name: usrssl
- hostPath:
    path: /usr/lib/ssl
  name: usrlibssl
- hostPath:
    path: /usr/local/openssl
  name: usrlocalopenssl
- hostPath:
    path: /etc/openssl
  name: etcopenssl
- hostPath:
    path: /etc/pki/tls
  name: etcpkitls
```