```
# kubectl create configmap cm-appconf --from-file=configfiles
configmap "cm-appconf" created

# kubectl describe configmap cm-appconf
Name:         cm-appconf
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
logging.properties:    3354 bytes
server.xml:            6458 bytes
```

使用--from-literal 参数进行创建的示例如下：

```
# kubectl create configmap cm-appenv --from-literal=loglevel=info --from-literal
=appdatadir=/var/data
configmap "cm-appenv" created

# kubectl  describe configmap cm-appenv
Name:         cm-appenv
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
appdatadir:    9 bytes
loglevel:      4 bytes
```

容器应用对 ConfigMap 的使用有以下两种方法。

（1）通过环境变量获取 ConfigMap 中的内容。

（2）通过 Volume 挂载的方式将 ConfigMap 中的内容挂载为容器内部的文件或目录。

4. ConfigMap 的使用：环境变量方式

以 cm-appvars.yaml 为例：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-appvars
data:
  apploglevel: info
  appdatadir: /var/data
```

在 Pod“cm-test-pod”的定义中，将 ConfigMap“cm-appvars”中的内容以环境变量（APPLOGLEVEL 和 APPDATADIR）设置为容器内部的环境变量，容器的启动命令将显示这两个环境变量的值（"env | grep APP"）：

```
apiVersion: v1
kind: Pod
metadata:
  name: cm-test-pod
spec:
  containers:
  - name: cm-test
    image: busybox
    command: [ "/bin/sh", "-c", "env | grep APP" ]
    env:
    - name: APPLOGLEVEL          # 定义环境变量名称
      valueFrom:                 # key“apploglevel”对应的值
        configMapKeyRef:
          name: cm-appvars       # 环境变量的值取自 cm-appvars 中：
          key: apploglevel       # key 为“apploglevel”
    - name: APPDATADIR           # 定义环境变量名称
      valueFrom:                 # key“appdatadir”对应的值
        configMapKeyRef:
          name: cm-appvars       # 环境变量的值取自 cm-appvars 中：
          key: appdatadir        # key 为“appdatadir”
  restartPolicy: Never
```

使用 kubectl create -f 命令创建该 Pod，由于是测试 Pod，所以该 Pod 在执行完启动命令后将会退出，并且不会被系统自动重启（restartPolicy=Never）：

```
# kubectl create -f cm-test-pod.yaml
pod "cm-test-pod" created
```

使用 kubectl get pods --show-all 查看已经停止的 Pod：

```
# kubectl get pods --show-all
NAME             READY     STATUS       RESTARTS   AGE
cm-test-pod       0/1       Completed     0         8s
```

查看该 Pod 的日志，可以看到启动命令“env | grep APP”的执行结果如下：

```
# kubectl logs cm-test-pod
APPDATADIR=/var/data
APPLOGLEVEL=info
```

说明容器内部的环境变量使用 ConfigMap cm-appvars 中的值进行了正确的设置。

5. ConfigMap 的使用：volumeMount 模式

下面 cm-appconfigfiles.yaml 的例子中包含两个配置文件的定义：server.xml 和 logging. properties。

cm-appconfigfiles.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-serverxml
data:
  key-serverxml: |
    <?xml version='1.0' encoding='utf-8'?>
    <Server port="8005" shutdown="SHUTDOWN">
      <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
      <Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
      <Listener className="org.apache.catalina.core.
JreMemoryLeakPreventionListener" />
      <Listener className="org.apache.catalina.mbeans.
GlobalResourcesLifecycleListener" />
      <Listener className="org.apache.catalina.core.
ThreadLocalLeakPreventionListener" />
      <GlobalNamingResources>
        <Resource name="UserDatabase" auth="Container"
                type="org.apache.catalina.UserDatabase"
                description="User database that can be updated and saved"
                factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
                pathname="conf/tomcat-users.xml" />
      </GlobalNamingResources>

      <Service name="Catalina">
        <Connector port="8080" protocol="HTTP/1.1"
                 connectionTimeout="20000"
                 redirectPort="8443" />
        <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
        <Engine name="Catalina" defaultHost="localhost">
          <Realm className="org.apache.catalina.realm.LockOutRealm">
            <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                 resourceName="UserDatabase"/>
          </Realm>
          <Host name="localhost"  appBase="webapps"
               unpackWARs="true" autoDeploy="true">
            <Valve className="org.apache.catalina.valves.AccessLogValve"
directory="logs"
                 prefix="localhost_access_log" suffix=".txt"
                 pattern="%h %l %u %t &amp;quot;%r&amp;quot; %s %b" />
```

```
            </Host>
          </Engine>
        </Service>
      </Server>
    key-loggingproperties: "handlers
      = 1catalina.org.apache.juli.FileHandler,
2localhost.org.apache.juli.FileHandler,
      3manager.org.apache.juli.FileHandler,
4host-manager.org.apache.juli.FileHandler,
      java.util.logging.ConsoleHandler\r\n\r\n.handlers =
1catalina.org.apache.juli.FileHandler,

java.util.logging.ConsoleHandler\r\n\r\n1catalina.org.apache.juli.FileHandler.level
      = FINE\r\n1catalina.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n1catalina.org.apache.juli.FileHandler.prefix
      = catalina.\r\n\r\n2localhost.org.apache.juli.FileHandler.level =
FINE\r\n2localhost.org.apache.juli.FileHandler.directory
      = ${catalina.base}/logs\r\n2localhost.org.apache.juli.FileHandler.prefix =
localhost.\r\n\r\n3manager.org.apache.juli.FileHandler.level
      = FINE\r\n3manager.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n3manager.org.apache.juli.FileHandler.prefix
      = manager.\r\n\r\n4host-manager.org.apache.juli.FileHandler.level =
FINE\r\n4host-manager.org.apache.juli.FileHandler.directory
      = ${catalina.base}/logs\r\n4host-manager.org.apache.juli.FileHandler.
prefix =
      host-manager.\r\n\r\njava.util.logging.ConsoleHandler.level =
FINE\r\njava.util.logging.ConsoleHandler.formatter
      = java.util.logging.SimpleFormatter\r\n\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].level
      = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
handlers
      = 2localhost.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/manager].level
      = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/manager].handlers
      = 3manager.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/host-manager].level
      = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/host-manager].handlers
      = 4host-manager.org.apache.juli.FileHandler\r\n\r\n"
```

在 Pod“cm-test-app”的定义中，将 ConfigMap“cm-appconfigfiles”中的内容以文件的形式 mount 到容器内部的/configfiles 目录中去。Pod 配置文件 cm-test-app.yaml 的内容如下：

```
apiVersion: v1
kind: Pod
metadata:
```

```
    name: cm-test-app
  spec:
    containers:
    - name: cm-test-app
      image: kubeguide/tomcat-app:v1
      ports:
      - containerPort: 8080
      volumeMounts:
      - name: serverxml                    # 引用 volume 名
        mountPath: /configfiles            # 挂载到容器内的目录
    volumes:
    - name: serverxml                      # 定义 volume 名
      configMap:
        name: cm-appconfigfiles            # 使用 ConfigMap "cm-appconfigfiles"
        items:
        - key: key-serverxml               # key=key-serverxml
          path: server.xml                 # value 将 server.xml 文件名进行挂载
        - key: key-loggingproperties       # key=key-loggingproperties
          path: logging.properties         # value 将 logging.properties 文件名进行挂载
```

创建该 Pod：

```
# kubectl create -f cm-test-app.yaml
pod "cm-test-app" created
```

登录容器，查看到/configfiles 目录下存在 server.xml 和 logging.properties 文件，它们的内容就是 ConfigMap "cm-appconfigfiles" 中两个 key 定义的内容。

```
# kubectl exec -ti cm-test-app -- bash
root@cm-test-app:/# cat /configfiles/server.xml
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
......

root@cm-test-app:/# cat /configfiles/logging.properties
handlers = 1catalina.org.apache.juli.AsyncFileHandler,
2localhost.org.apache.juli.AsyncFileHandler,
3manager.org.apache.juli.AsyncFileHandler,
4host-manager.org.apache.juli.AsyncFileHandler, java.util.logging.ConsoleHandler
......
```

如果在引用 ConfigMap 时不指定 items，则使用 volumeMount 方式在容器内的目录中为每个 item 生成一个文件名为 key 的文件。

Pod 配置文件 cm-test-app.yaml 的内容如下：

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: cm-test-app
spec:
  containers:
  - name: cm-test-app
    image: kubeguide/tomcat-app:v1
    imagePullPolicy: Never
    ports:
    - containerPort: 8080
    volumeMounts:
    - name: serverxml                # 引用 volume 名
      mountPath: /configfiles        # 挂载到容器内的目录
  volumes:
  - name: serverxml                  # 定义 volume 名
    configMap:
      name: cm-appconfigfiles        # 使用 ConfigMap "cm-appconfigfiles"
```

创建该 Pod：

```
# kubectl create -f cm-test-app.yaml
pod "cm-test-app" created
```

登录容器，查看到/configfiles 目录下存在 key-loggingproperties 和 key-serverxml 文件，文件的名称来自 ConfigMap cm-appconfigfiles 中定义的两个 key 的名称，文件的内容则为 value 的内容：

```
# ls /configfiles
key-loggingproperties  key-serverxml
```

6. 使用 ConfigMap 的限制条件

使用 ConfigMap 的限制条件如下。

- ConfigMap 必须在 Pod 之前创建。
- ConfigMap 也可以定义为属于某个 Namespace。只有处于相同 Namespaces 中的 Pod 可以引用它。
- ConfigMap 中的配额管理还未能实现。
- kubelet 只支持可以被 API Server 管理的 Pod 使用 ConfigMap。kubelet 在本 Node 上通过 --manifest-url 或--config 自动创建的静态 Pod 将无法引用 ConfigMap。
- 在 Pod 对 ConfigMap 进行挂载（volumeMount）操作时，容器内部只能挂载为“目录”，无法挂载为“文件”。在挂载到容器内部后，目录中将包含 ConfigMap 定义的每个 item，如果该目录下原先还有其他文件，则容器内的该目录将会被挂载的 ConfigMap 进行覆盖。如果应用程序需要保留原来的其他文件，则需要进行额外的处理。可以通过将

ConfigMap 挂载到容器内部的临时目录，再通过启动脚本将配置文件复制或者链接（cp 或 link 操作）到应用所用的实际配置目录下。

2.4.6　Pod 生命周期和重启策略

Pod 在整个生命周期过程中被系统定义为各种状态，熟悉 Pod 的各种状态对于我们理解如何设置 Pod 的调度策略、重启策略是很有必要的。

Pod 的状态包括以下几种，如表 2.14 所示。

表 2.14　Pod 的状态

状 态 值	描　述
Pending	API Server 已经创建该 Pod，但 Pod 内还有一个或多个容器的镜像没有创建，包括正在下载镜像的过程
Running	Pod 内所有容器均已创建，且至少有一个容器处于运行状态、正在启动状态或正在重启状态
Succeeded	Pod 内所有容器均成功执行退出，且不会再重启
Failed	Pod 内所有容器均已退出，但至少有一个容器退出为失败状态
Unknown	由于某种原因无法获取该 Pod 的状态，可能由于网络通信不畅导致

Pod 的重启策略（**RestartPolicy**）应用于 Pod 内的所有容器，并且仅在 Pod 所处的 Node 上由 kubelet 进行判断和重启操作。当某个容器异常退出或者健康检查（详见下节）失败时，kubelet 将根据 RestartPolicy 的设置来进行相应的操作。

Pod 的重启策略包括 Always、OnFailure 和 Never，默认值为 Always。

- Always：当容器失效时，由 kubelet 自动重启该容器。
- OnFailure：当容器终止运行且退出码不为 0 时，由 kubelet 自动重启该容器。
- Never：不论容器运行状态如何，kubelet 都不会重启该容器。

kubelet 重启失效容器的时间间隔以 sync-frequency 乘以 $2n$ 来计算，例如 1、2、4、8 倍等，最长延时 5 分钟，并且在成功重启后的 10 分钟后重置该时间。

Pod 的重启策略与控制方式息息相关，当前可用于管理 Pod 的控制器包括 ReplicationController、Job、DaemonSet 及直接通过 kubelet 管理（静态 Pod）。每种控制器对 Pod 的重启策略要求如下。

- RC 和 DaemonSet：必须设置为 Always，需要保证该容器持续运行。
- Job：OnFailure 或 Never，确保容器执行完成后不再重启。
- kubelet：在 Pod 失效时自动重启它，不论 RestartPolicy 设置为什么值，并且也不会对 Pod 进行健康检查。

结合 Pod 的状态和重启策略，表 2.15 列出一些常见的状态转换场景。

表 2.15 一些常见的状态转换场景

Pod 包含的容器数	Pod 当前的状态	发生事件	Pod 的结果状态		
			RestartPolicy=Always	RestartPolicy=OnFailure	RestartPolicy=Never
包含 1 个容器	Running	容器成功退出	Running	Succeeded	Succeeded
包含 1 个容器	Running	容器失败退出	Running	Running	Failed
包含两个容器	Running	1 个容器失败退出	Running	Running	Running
包含两个容器	Running	容器被 OOM 杀掉	Running	Running	Failed

2.4.7 Pod 健康检查

对 Pod 的健康状态检查可以通过两类探针来检查：LivenessProbe 和 ReadinessProbe。

- **LivenessProbe 探针**：用于判断容器是否存活（running 状态），如果 LivenessProbe 探针探测到容器不健康，则 kubelet 将杀掉该容器，并根据容器的重启策略做相应的处理。如果一个容器不包含 LivenessProbe 探针，那么 kubelet 认为该容器的 LivenessProbe 探针返回的值永远是“Success”。
- **ReadinessProbe**：用于判断容器是否启动完成（ready 状态），可以接收请求。如果 ReadinessProbe 探针检测到失败，则 Pod 的状态将被修改。Endpoint Controller 将从 Service 的 Endpoint 中删除包含该容器所在 Pod 的 Endpoint。

kubelet 定期执行 LivenessProbe 探针来诊断容器的健康状况。LivenessProbe 有以下三种实现方式。

（1）ExecAction：在容器内部执行一个命令，如果该命令的返回码为 0，则表明容器健康。

在下面的例子中，通过执行“cat /tmp/health”命令来判断一个容器运行是否正常。而该 Pod 运行之后，在创建/tmp/health 文件的 10 秒之后将删除该文件，而 LivenessProbe 健康检查的初始探测时间（initialDelaySeconds）为 15 秒，探测结果将是 Fail，将导致 kubelet 杀掉该容器并重启它。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
```

```
    image: gcr.io/google_containers/busybox
    args:
    - /bin/sh
    - -c
    - echo ok > /tmp/health; sleep 10; rm -rf /tmp/health; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/health
      initialDelaySeconds: 15
      timeoutSeconds: 1
```

（2）TCPSocketAction：通过容器的 IP 地址和端口号执行 TCP 检查，如果能够建立 TCP 连接，则表明容器健康。

在下面的例子中，通过与容器内的 localhost:80 建立 TCP 连接进行健康检查。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-healthcheck
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    livenessProbe:
      tcpSocket:
        port: 80
      initialDelaySeconds: 30
      timeoutSeconds: 1
```

（3）HTTPGetAction：通过容器的 IP 地址、端口号及路径调用 HTTP Get 方法，如果响应的状态码大于等于 200 且小于等于 400，则认为容器状态健康。

在下面的例子中，kubelet 定时发送 HTTP 请求到 localhost:80/_status/healthz 来进行容器应用的健康检查。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-healthcheck
spec:
  containers:
  - name: nginx
    image: nginx
```

```
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
        path: /_status/healthz
        port: 80
      initialDelaySeconds: 30
      timeoutSeconds: 1
```

对于每种探测方式，都需要设置 initialDelaySeconds 和 timeoutSeconds 两个参数，它们的含义分别如下。

- **initialDelaySeconds**：启动容器后进行首次健康检查的等待时间，单位为秒。
- **timeoutSeconds**：健康检查发送请求后等待响应的超时时间，单位为秒。当超时发生时，kubelet 会认为容器已经无法提供服务，将会重启该容器。

2.4.8 玩转 Pod 调度

在 Kubernetes 系统中，Pod 在大部分场景下都只是容器的载体而已，通常需要通过 RC、Deployment、DaemonSet、Job 等对象来完成 Pod 的调度与自动控制功能。

1. RC、Deployment：全自动调度

RC 的主要功能之一就是自动部署一个容器应用的多份副本，以及持续监控副本的数量，在集群内始终维持用户指定的副本数量。

根据 frontend-controller.yaml 配置，用户需要创建 3 个 kubeguide/guestbook-php-frontend 应用的副本，在将该定义发送给 Kubernetes 之后，系统将在集群中合适的 Node 上创建 3 个 Pod，并始终维持 3 个副本的数量。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:
```

```
    name: frontend
  spec:
    containers:
    - name: frontend
      image: kubeguide/guestbook-php-frontend
      env:
      - name: GET_HOSTS_FROM
        value: env
      ports:
      - containerPort: 80
```

在调度策略上，除了使用系统内置的调度算法选择合适的 Node 进行调度，也可以在 Pod 的定义中使用 NodeSelector 或 NodeAffinity 来指定满足条件的 Node 进行调度，下面我们分别进行说明。

1）NodeSelector：定向调度

Kubernetes Master 上的 Scheduler 服务（kube-scheduler 进程）负责实现 Pod 的调度，整个调度过程通过执行一系列复杂的算法，最终为每个 Pod 计算出一个最佳的目标节点，这一过程是自动完成的，通常我们无法知道 Pod 最终会被调度到哪个节点上。在实际情况中，也可能需要将 Pod 调度到指定的一些 Node 上，可以通过 Node 的标签（Label）和 Pod 的 nodeSelector 属性相匹配，来达到上述目的。

（1）首先通过 kubectl label 命令给目标 Node 打上一些标签：

```
kubectl label nodes <node-name> <label-key>=<label-value>
```

这里，我们为 k8s-node-1 节点打上一个 zone=north 的标签，表明它是“北方”的一个节点：

```
$ kubectl label nodes k8s-node-1 zone=north
NAME          LABELS                                          STATUS
k8s-node-1    kubernetes.io/hostname=k8s-node-1,zone=north    Ready
```

上述命令行操作也可以通过修改资源定义文件的方式，并执行 kubectl replace -f xxx.yaml 命令来完成。

（2）然后，在 Pod 的定义中加上 nodeSelector 的设置，以 redis-master-controller.yaml 为例：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
  selector:
    name: redis-master
```

```
  template:
    metadata:
      labels:
        name: redis-master
    spec:
      containers:
      - name: master
        image: kubeguide/redis-master
        ports:
        - containerPort: 6379
      nodeSelector:
        zone: north
```

运行 kubectl create -f 命令创建 Pod，scheduler 就会将该 Pod 调度到拥有 zone=north 标签的 Node 上。

使用 kubectl get pods -o wide 命令可以验证 Pod 所在的 Node：

```
# kubectl get pods -o wide
NAME                READY    STATUS    RESTARTS   AGE       NODE
redis-master-f0rqj  1/1      Running   0          19s       k8s-node-1
```

如果我们给多个 Node 都定义了相同的标签（例如 zone=north），则 scheduler 将会根据调度算法从这组 Node 中挑选一个可用的 Node 进行 Pod 调度。

通过基于 Node 标签的调度方式，我们可以把集群中具有不同特点的 Node 贴上不同的标签，例如“role=frontend”“role=backend”“role=database”等标签，在部署应用时就可以根据应用的需求设置 NodeSelector 来进行指定 Node 范围的调度。

需要注意的是，如果我们指定了 Pod 的 nodeSelector 条件，且集群中不存在包含相应标签的 Node，则即使集群中还有其他可供使用的 Node，这个 Pod 也无法被成功调度。

2）NodeAffinity：亲和性调度

NodeAffinity 意为 Node 亲和性的调度策略，是将来替换 NodeSelector 的下一代调度策略。由于 NodeSelector 通过 Node 的 Label 进行精确匹配，所以 NodeAffinity 增加了 In、NotIn、Exists、DoesNotExist、Gt、Lt 等操作符来选择 Node，能够使调度策略更加灵活。同时，在 NodeAffinity 中将增加一些信息来设置亲和性调度策略。

- RequiredDuringSchedulingRequiredDuringExecution：类似于 NodeSelector，但在 Node 不满足条件时，系统将从该 Node 上移除之前调度上的 Pod。
- RequiredDuringSchedulingIgnoredDuringExecution：与第 1 个 RequiredDuringScheduling RequiredDuringExecution 的作用相似，区别是在 Node 不满足条件时，系统不一定从该 Node 上移除之前调度上的 Pod。

- PreferredDuringSchedulingIgnoredDuringExecution：指定在满足调度条件的 Node 中，哪些 Node 应更优先地进行调度。同时在 Node 不满足条件时，系统不一定从该 Node 上移除之前调度上的 Pod。

在当前的 Alpha 版本中，需要在 Pod 的 metadata.annotations 中设置 NodeAffinity 的内容：

```
apiVersion: v1
kind: Pod
metadata:
  name: with-labels
  annotations:
    scheduler.alpha.kubernetes.io/affinity: >
      {
        "nodeAffinity": {
          "requiredDuringSchedulingIgnoredDuringExecution": {
            "nodeSelectorTerms": [
              {
                "matchExpressions": [
                  {
                    "key": "kubernetes.io/e2e-az-name",
                    "operator": "In",
                    "values": ["e2e-az1", "e2e-az2"]
                  }
                ]
              }
            ]
          }
        }
      }
    another-annotation-key: another-annotation-value
spec:
  containers:
  - name: with-labels
    image: gcr.io/google_containers/pause:2.0
```

这里 NodeAffinity 的设置说明只有 Node 的 Label 中包含 key= kubernetes.io/e2e-az-name，并且其 value 为“e2e-az1”或“e2e-az2”时，才能成为该 Pod 的调度目标。其中操作符为 In，代表“或”运算，其他操作符包括 NotIn（不属于）、Exists（存在一个条件）、DoesNotExist（不存在）、Gt（大于）、Lt（小于）。

如果同时设置了 NodeSelector 和 NodeAffinity，则系统将需要同时满足两者的设置才能进行调度。

在未来的 Kubernetes 版本中，还将加入 Pod Affinity 的设置，用于控制当调度 Pod 到某个特定的 Node 上时，判断是否有其他 Pod 正在该 Node 上运行，即通过其他的相关 Pod 进行调度，

而不仅仅通过 Node 本身的标签进行调度。

2. DaemonSet：特定场景调度

DaemonSet 是 Kubernetes 1.2 版本新增的一种资源对象，用于管理在集群中每个 Node 上仅运行一份 Pod 的副本实例，如图 2.10 所示。

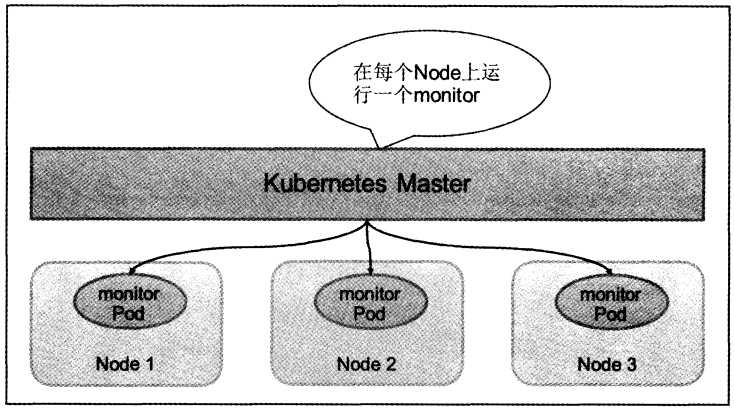

图 2.10　DaemonSet 示例

这种用法适合一些有这种需求的应用。

- 在每个 Node 上运行一个 GlusterFS 存储或者 Ceph 存储的 daemon 进程。
- 在每个 Node 上运行一个日志采集程序，例如 fluentd 或者 logstach。
- 在每个 Node 上运行一个健康程序，采集该 Node 的运行性能数据，例如 Prometheus Node Exporter、collectd、New Relic agent 或者 Ganglia gmond 等。

DaemonSet 的 Pod 调度策略与 RC 类似，除了使用系统内置的算法在每台 Node 上进行调度，也可以在 Pod 的定义中使用 NodeSelector 或 NodeAffinity 来指定满足条件的 Node 范围进行调度。

下面的例子定义为在每台 Node 上启动一个 fluentd 容器，配置文件 fluentd-ds.yaml 的内容如下，其中挂载了物理机的两个目录“/var/log”和“/var/lib/docker/containers”：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd-cloud-logging
  namespace: kube-system
  labels:
    k8s-app: fluentd-cloud-logging
spec:
  template:
```

```
  metadata:
    namespace: kube-system
    labels:
      k8s-app: fluentd-cloud-logging
  spec:
    containers:
    - name: fluentd-cloud-logging
      image: gcr.io/google_containers/fluentd-elasticsearch:1.17
      resources:
        limits:
          cpu: 100m
          memory: 200Mi
      env:
      - name: FLUENTD_ARGS
        value: -q
      volumeMounts:
      - name: varlog
        mountPath: /var/log
        readOnly: false
      - name: containers
        mountPath: /var/lib/docker/containers
        readOnly: false
    volumes:
    - name: containers
      hostPath:
        path: /var/lib/docker/containers
    - name: varlog
      hostPath:
        path: /var/log
```

使用 kubectl create 命令创建该 DaemonSet：

```
# kubectl create -f fluentd-ds.yaml
daemonset "fluentd-cloud-logging" created
```

查看创建好的 DaemonSet 和 Pod，可以看到在每个 Node 上都创建了一个 Pod：

```
# kubectl get daemonset --namespace=kube-system
NAME                    DESIRED  CURRENT   NODE-SELECTOR   AGE
fluentd-cloud-logging   2        2         <none>          3s

# kubectl get pods --namespace=kube-system
NAME                              READY     STATUS    RESTARTS   AGE
fluentd-cloud-logging-7tw9z       1/1       Running   0          1h
fluentd-cloud-logging-aqdn1       1/1       Running   0          1h
```

3. Job：批处理调度

Kubernetes 从 1.2 版本开始支持批处理类型的应用，我们可以通过 Kubernetes Job 资源对象来定义并启动一个批处理任务。批处理任务通常并行（或者串行）启动多个计算进程去处理一批工作项（work item），处理完成后，整个批处理任务结束。按照批处理任务实现方式的不同，批处理任务可以分为如图 2.11 所示的几种模式。

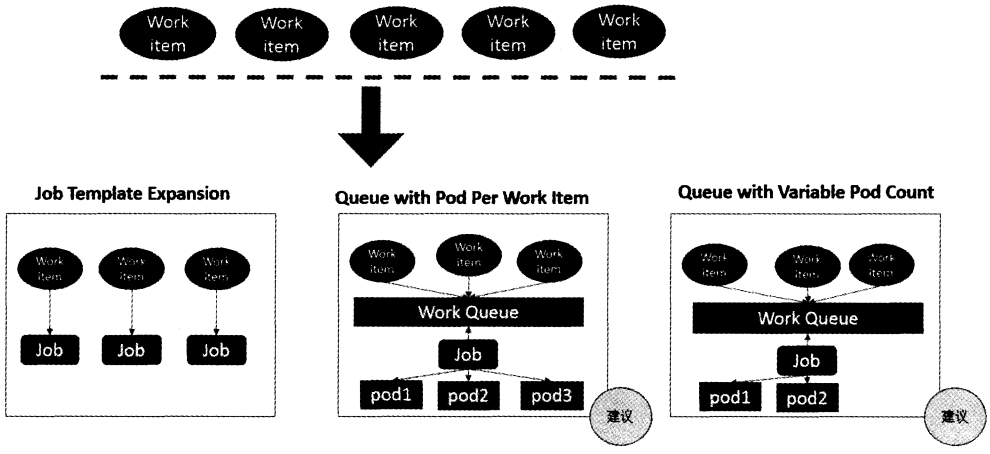

图 2.11　批处理任务的几种模式

- Job Template Expansion 模式：一个 Job 对象对应一个待处理的 Work item，有几个 Work item 就产生几个独立的 Job，通常适合 Work item 数量少、每个 Work item 要处理的数据量比较大的场景，比如有一个 100GB 的文件作为一个 Work item，总共 10 个文件需要处理。
- Queue with Pod Per Work Item 模式：采用一个任务队列存放 Work item，一个 Job 对象作为消费者去完成这些 Work item，在这种模式下，Job 会启动 *N* 个 Pod，每个 Pod 对应一个 Work item。
- Queue with Variable Pod Count 模式：也是采用一个任务队列存放 Work item，一个 Job 对象作为消费者去完成这些 Work item，但与上面的模式不同，Job 启动的 Pod 数量是可变的。

还有一种被称为 Single Job with Static Work Assignment 的模式，也是一个 Job 产生多个 Pod 的模式，但它采用程序静态方式分配任务项，而不是采用队列模式进行动态分配。

如表 2.16 所示是这几种模式的一个对比。

模 式 名 称	是否是一个 Job	Pod 的数量少于 Work item	用户程序是否要做相应的修改	Kubernetes 是否支持
Job Template Expansion	/	/	是	是
Queue with Pod Per Work Item	是	/	有时候需要	是
Queue with Variable Pod Count	是	/	/	是
Single Job with Static Work Assignment	是	/	是	/

考虑到批处理的并行问题，Kubernetes 将 Job 分以下三种类型。

1）Non-parallel Jobs

通常一个 Job 只启动一个 Pod，则除非 Pod 异常，才会重启该 Pod，一旦此 Pod 正常结束，Job 将结束。

2）Parallel Jobs with a fixed completion count

并行 Job 会启动多个 Pod，此时需要设定 Job 的.spec.completions 参数为一个正数，当正常结束的 Pod 数量达到此参数设定的值后，Job 结束。此外，Job 的.spec.parallelism 参数用来控制并行度，即同时启动几个 Job 来处理 Work Item。

3）Parallel Jobs with a work queue

任务队列方式的并行 Job 需要一个独立的 Queue，Work item 都在一个 Queue 中存放，不能设置 Job 的.spec.completions 参数，此时 Job 有以下一些特性。

- 每个 Pod 能独立判断和决定是否还有任务项需要处理。
- 如果某个 Pod 正常结束，则 Job 不会再启动新的 Pod。
- 如果一个 Pod 成功结束，则此时应该不存在其他 Pod 还在干活的情况，它们应该都处于即将结束、退出的状态。
- 如果所有 Pod 都结束了，且至少有一个 Pod 成功结束，则整个 Job 算是成功结束。

下面我们分别说说常见的三种批处理模型在 Kubernetes 中的例子。

首先是 Job Template Expansion 模式，由于这种模式下每个 Work item 对应一个 Job 实例，所以这种模式首先定义一个 Job 模板，模板里主要的参数是 Work item 的标识，因为每个 Job 处理不同的 Work item。如下所示的 Job 模板（文件名为 job.yaml.txt）中的$ITEM 可以作为任务项的标识：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: process-item-$ITEM
  labels:
    jobgroup: jobexample
```

```
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
      - name: c
        image: busybox
        command: ["sh", "-c", "echo Processing item $ITEM && sleep 5"]
      restartPolicy: Never
```

通过下面的操作，生成 3 个对应的 Job 定义文件并创建 Job：

```
# for i in apple banana cherry
> do
>   cat job.yaml.txt | sed "s/\$ITEM/$i/" > ./jobs/job-$i.yaml
> done
# ls jobs
job-apple.yaml  job-banana.yaml  job-cherry.yaml
# kubectl create -f jobs
job "process-item-apple" created
job "process-item-banana" created
job "process-item-cherry" created
```

观察 Job 的运行情况：

```
# kubectl get jobs -l jobgroup=jobexample
NAME                DESIRED   SUCCESSFUL   AGE
process-item-apple    1         1          4m
process-item-banana   1         1          4m
process-item-cherry   1         1          4m
```

其次，我们看看 Queue with Pod Per Work Item 模式，在这种模式下需要一个任务队列存放 Work item，比如 RabbitMQ，客户端程序先把要处理的任务变成 Work item 放入到任务队列，然后编写 Worker 程序并打包镜像并定义成为 Job 中的 Work Pod，Worker 程序的实现逻辑是从任务队列中拉取一个 Work item 并处理，处理完成后即结束进程，图 2.12 给出了并行度为 2 的一个 Demo 示意图。

最后，我们再看看 Queue with Variable Pod Count 模式，如图 2.13 所示，由于这种模式下，Worker 程序需要知道队列中是否还有等待处理的 Work item，如果有就取出来并处理，否则就认为所有工作完成并结束进程，所以任务队列通常要采用 Redis 或者数据库来实现。

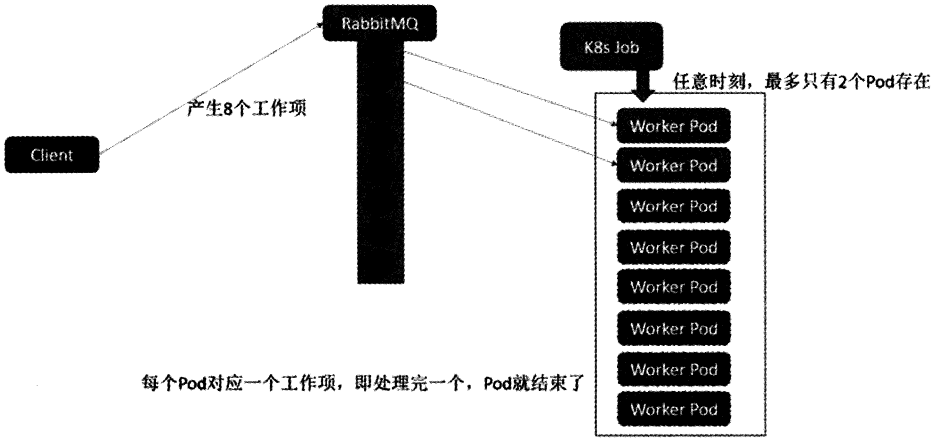

图 2.12　Queue with Pod Per Work Item 示例

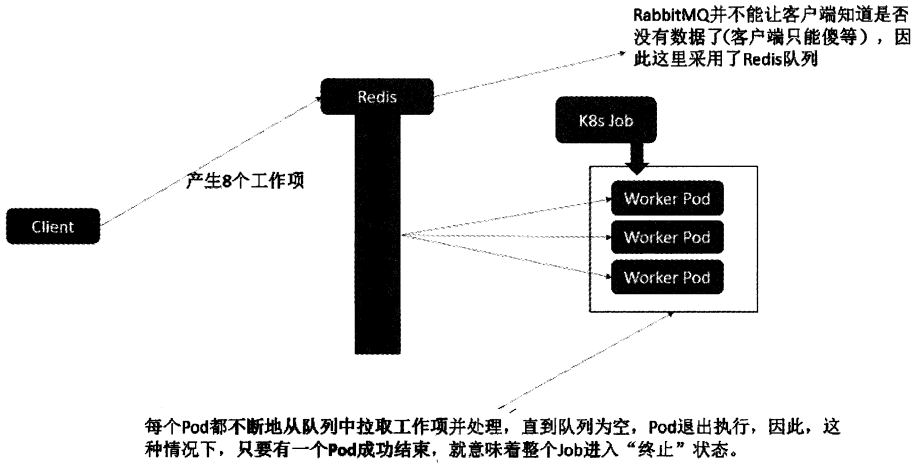

图 2.13　Queue with Variable Pod Count 示例

Kubernetes 对 Job 的支持还处于初级阶段，类似 Linux Cron 的定时任务也还没时间，计划在 Kubernetes 1.4 中实现。此外，更为复杂的流程类的批处理框架也还没有考虑，但随着 Kubernetes 生态圈的不断发展和壮大，相信 Kubernetes 在批处理方面也会有更多的规划。

2.4.9　Pod 的扩容和缩容

在实际生产系统中，我们经常会遇到某个服务需要扩容的场景，也可能会遇到由于资源紧张或者工作负载降低而需要减少服务实例数量的场景。此时我们可以利用 RC 的 Scale 机制来完

成这些工作。以 redis-slave RC 为例，已定义的最初副本数量为 2，通过 kubectl scale 命令可以将 redis-slave RC 控制的 Pod 副本数量从初始的 2 更新为 3：

```
$ kubectl scale rc redis-slave --replicas=3
replicationcontroller "redis-slave" scaled
```

执行 kubectl get pods 命令来验证 Pod 的副本数量增加到 3：

```
$ kubectl get pods
NAME                READY     STATUS    RESTARTS   AGE
redis-slave-4na2n   1/1       Running   0          1h
redis-slave-92u3k   1/1       Running   0          1h
redis-slave-palab   1/1       Running   0          2m
```

将--replicas 设置为比当前 Pod 副本数量更小的数字，系统将会“杀掉”一些运行中的 Pod，以实现应用集群缩容：

```
$ kubectl scale rc redis-slave --replicas=1
replicationcontroller "redis-slave" scaled

$ kubectl get pods
NAME                READY     STATUS    RESTARTS   AGE
redis-slave-4na2n   1/1       Running   0          1h
```

除了可以手工通过 kubectl scale 命令完成 Pod 的扩容和缩容操作，Kubernetes v1.1 版本新增了名为 Horizontal Pod Autoscaler（HPA）的控制器，用于实现基于 CPU 使用率进行自动 Pod 扩缩容的功能。HPA 控制器基于 Master 的 kube-controller-manager 服务启动参数 --horizontal-pod-autoscaler-sync-period 定义的时长（默认为 30 秒），周期性地监测目标 Pod 的 CPU 使用率，并在满足条件时对 ReplicationController 或 Deployment 中的 Pod 副本数量进行调整，以符合用户定义的平均 Pod CPU 使用率。Pod CPU 使用率来源于 heapster 组件，所以需要预先安装好 heapster。

创建 HPA 时可以使用 kubectl autoscale 命令进行快速创建或者使用 yaml 配置文件进行创建。在创建 HPA 之前，需要已经存在一个 RC 或 Deployment 对象，并且该 RC 或 Deployment 中的 Pod 必须定义 resources.requests.cpu 的资源请求值，如果不设置该值，则 heapster 将无法采集到该 Pod 的 CPU 使用情况，会导致 HPA 无法正常工作。

下面通过给一个 RC 设置 HPA，然后使用一个客户端对其进行压力测试，对 HPA 的用法进行示例。

以 php-apache 的 RC 为例，设置 cpu request 为 200m，未设置 limit 上限的值：

php-apache-rc.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
```

```
  name: php-apache
spec:
  replicas: 1
  template:
    metadata:
      name: php-apache
      labels:
        app: php-apache
    spec:
      containers:
      - name: php-apache
        image: gcr.io/google_containers/hpa-example
        resources:
          requests:
            cpu: 200m
        ports:
        - containerPort: 80

# kubectl create -f php-apache-rc.yaml
replicationcontroller "php-apache" created
```

再创建一个 php-apache 的 Service，供客户端访问：

```
php-apache-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: php-apache
spec:
  ports:
  - port: 80
  selector:
    app: php-apache

# kubectl create -f php-apache-svc.yaml
service "php-apache" created
```

接下来为 RC“php-apache”创建一个 HPA 控制器，在 1 和 10 之间调整 Pod 的副本数量，以使得平均 Pod CPU 使用率维持在 50%。

使用 kubectl autoscale 命令进行创建：

```
# kubectl autoscale rc php-apache --min=1 --max=10 --cpu-percent=50
```

或者通过 yaml 配置文件来创建 HPA，需要在 scaleTargetRef 字段指定需要管理的 RC 或 Deployment 的名字，然后设置 minReplicas、maxReplicas 和 targetCPUUtilizationPercentage 参数：

```
hpa-php-apache.yaml
apiVersion: autoscaling/v1
```

```
    kind: HorizontalPodAutoscaler
    metadata:
      name: php-apache
    spec:
      scaleTargetRef:
        apiVersion: v1
        kind: ReplicationController
        name: php-apache
      minReplicas: 1
      maxReplicas: 10
      targetCPUUtilizationPercentage: 50

    # kubectl create -f hpa-php-apache.yaml
    horizontalpodautoscaler "php-apache" created
```

查看已创建的 HPA：

```
    # kubectl get hpa
    NAME         REFERENCE                          TARGET    CURRENT   MINPODS
MAXPODS   AGE
    php-apache   ReplicationController/php-apache   50%       0%        1         10        1m
```

然后，我们创建一个 busybox Pod，用于对 php-apache 服务发起压力测试的请求：

busybox-pod.yaml

```
    apiVersion: v1
    kind: Pod
    metadata:
      name: busybox
    spec:
      containers:
      - name: busybox
        image: busybox
        command: [ "sleep", "3600" ]

    # kubectl create -f busybox-pod.yaml
    pod "busybox" created
```

登录 busybox 容器，执行一个无限循环的 wget 命令来访问 php-apache 服务：

```
    # while true; do wget -q -O- http://php-apache > /dev/null; done
```

注意这里 wget 的目的地 URL 地址是 Service 的名称“php-apache”，这要求 DNS 服务正常工作，也可以使用 Service 的虚拟 ClusterIP 地址对其进行访问，例如 http://169.169.122.145：

```
    # kubectl exec -ti busybox -- sh
    / # while true; do wget -q -O- http://php-apache > /dev/null; done
```

等待一段时间后，观察 HPA 控制器搜集到的 Pod CPU 使用率：

```
    # kubectl get hpa
```

```
NAME         REFERENCE                    TARGET    CURRENT   MINPODS   MAXPODS   AGE
php-apache   ReplicationController/php-apache   50%    3068%     1         10        3m
```

再过一会儿，查看 RC php-apache 副本数量的变化：

```
# kubectl get rc
NAME         DESIRED   CURRENT   AGE
php-apache   10        10        23m
```

可以看到 HPA 已经根据 Pod 的 CPU 使用率的提高对 RC 进行了自动扩容，Pod 副本数量变成了 10 个。这个过程如图 2.14 所示。

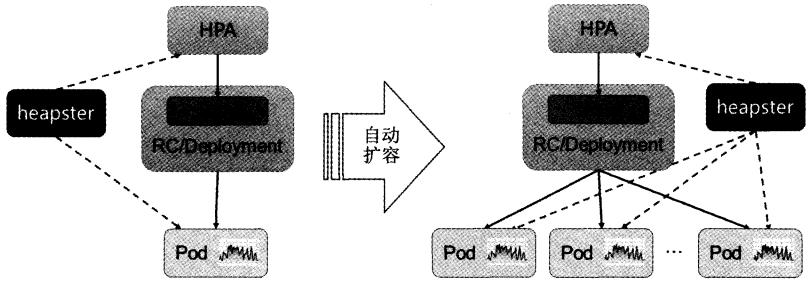

图 2.14　HPA 自动扩容

最后，我们停止压力测试，在 busybox 的控制台输入 Ctrl+C，停止无限循环操作。

等待一段时间，观察 HPA 的变化：

```
# kubectl get hpa
NAME         REFERENCE                    TARGET    CURRENT   MINPODS   MAXPODS   AGE
php-apache   ReplicationController/php-apache   50%    3%        1         10        20m
```

再次查看 RC 的副本数量：

```
NAME         DESIRED   CURRENT   AGE
php-apache   1         1         26m
```

可以看到 HPA 根据 Pod CPU 使用率的降低对副本数量进行了缩容操作，Pod 副本数量变成了 1 个。

当前 HPA 还只支持将 CPU 使用率作为 Pod 副本扩容缩容的触发条件，在将来的版本中，将会支持应用相关的指标例如 QPS（queries per second）或平均响应时间作为触发条件。

2.4.10　Pod 的滚动升级

下面我们说说 Pod 的升级问题。

当集群中的某个服务需要升级时，我们需要停止目前与该服务相关的所有 Pod，然后重新

拉取镜像并启动。如果集群规模比较大，则这个工作就变成了一个挑战，而且先全部停止然后逐步升级的方式会导致较长时间的服务不可用。Kubernetes 提供了 rolling-update（滚动升级）功能来解决上述问题。

滚动升级通过执行 kubectl rolling-update 命令一键完成，该命令创建了一个新的 RC，然后自动控制旧的 RC 中的 Pod 副本的数量逐渐减少到 0，同时新的 RC 中的 Pod 副本的数量从 0 逐步增加到目标值，最终实现了 Pod 的升级。需要注意的是，系统要求新的 RC 需要与旧的 RC 在相同的命名空间（Namespace）内，即不能把别人的资产偷偷转移到自家名下。滚动升级的过程如图 2.15 所示。

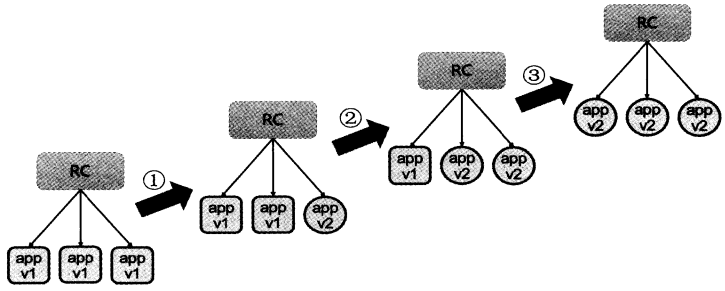

图 2.15　Pod 的滚动升级

以 redis-master 为例，假设当前运行的 redis-master Pod 是 1.0 版本，则现在需要升级到 2.0 版本。

创建 redis-master-controller-v2.yaml 的配置文件如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master-v2
  labels:
    name: redis-master
    version: v2
spec:
  replicas: 1
  selector:
    name: redis-master
    version: v2
  template:
    metadata:
      labels:
        name: redis-master
        version: v2
    spec:
      containers:
```

```
  - name: master
    image: kubeguide/redis-master:2.0
    ports:
    - containerPort: 6379
```

在配置文件中需要注意以下几点。

（1）RC 的名字（name）不能与旧的 RC 的名字相同。

（2）在 selector 中应至少有一个 Label 与旧的 RC 的 Label 不同，以标识其为新的 RC。本例中新增了一个名为 version 的 Label，以与旧的 RC 进行区分。

运行 kubectl rolling-update 命令完成 Pod 的滚动升级：

```
kubectl rolling-update redis-master -f redis-master-controller-v2.yaml
```

kubectl 的执行过程如下：

```
Creating redis-master-v2
At beginning of loop: redis-master replicas: 2, redis-master-v2 replicas: 1
Updating redis-master replicas: 2, redis-master-v2 replicas: 1
At end of loop: redis-master replicas: 2, redis-master-v2 replicas: 1
At beginning of loop: redis-master replicas: 1, redis-master-v2 replicas: 2
Updating redis-master replicas: 1, redis-master-v2 replicas: 2
At end of loop: redis-master replicas: 1, redis-master-v2 replicas: 2
At beginning of loop: redis-master replicas: 0, redis-master-v2 replicas: 3
Updating redis-master replicas: 0, redis-master-v2 replicas: 3
At end of loop: redis-master replicas: 0, redis-master-v2 replicas: 3
Update succeeded. Deleting redis-master
redis-master-v2
```

等所有新的 Pod 启动完成后，旧的 Pod 也被全部销毁，这样就完成了容器集群的更新工作。

另一种方法是不使用配置文件，直接用 kubectl rolling-update 命令，加上--image 参数指定新版镜像名称来完成 Pod 的滚动升级：

```
kubectl rolling-update redis-master --image=redis-master:2.0
```

与使用配置文件的方式不同，执行的结果是旧的 RC 被删除，新的 RC 仍将使用旧的 RC 的名字。

kubectl 的执行过程如下：

```
Creating redis-master-ea866a5d2c08588c3375b86fb253db75
At beginning of loop: redis-master replicas: 2, redis-master-ea866a5d2c08588c
3375b86fb253db75 replicas: 1
Updating redis-master replicas: 2, redis-master-ea866a5d2c08588c3375b86fb253db
75 replicas: 1
At end of loop: redis-master replicas: 2, redis-master-ea866a5d2c08588c3375b86fb
253db75 replicas: 1
At beginning of loop: redis-master replicas: 1, redis-master-ea866a5d2c08588c
```