Replication Controller

Replication Controller （简称 RC）是 Kubernetes 系统中的核心概念之一，简单来说，它其实是定义了一个期望的场景，即声明某种 Pod 的副本数在任意时刻都符合某个预期值，所以 RC 的定义包括如下几个部分。

- Pod 的期待的副本数（replicas）。
- 用于筛选目标 Pod 的 Label Selector。
- 当 Pod 的副本数量小于预期数量时，用于创建新 Pod 的模板（template）。

下面是一个完整的 RC 定义的例子，即确保拥有 app=nginx 标签的这个 Pod（运行 tomcat ）在整个 Kuberntes 集群中始终只有三个副本。

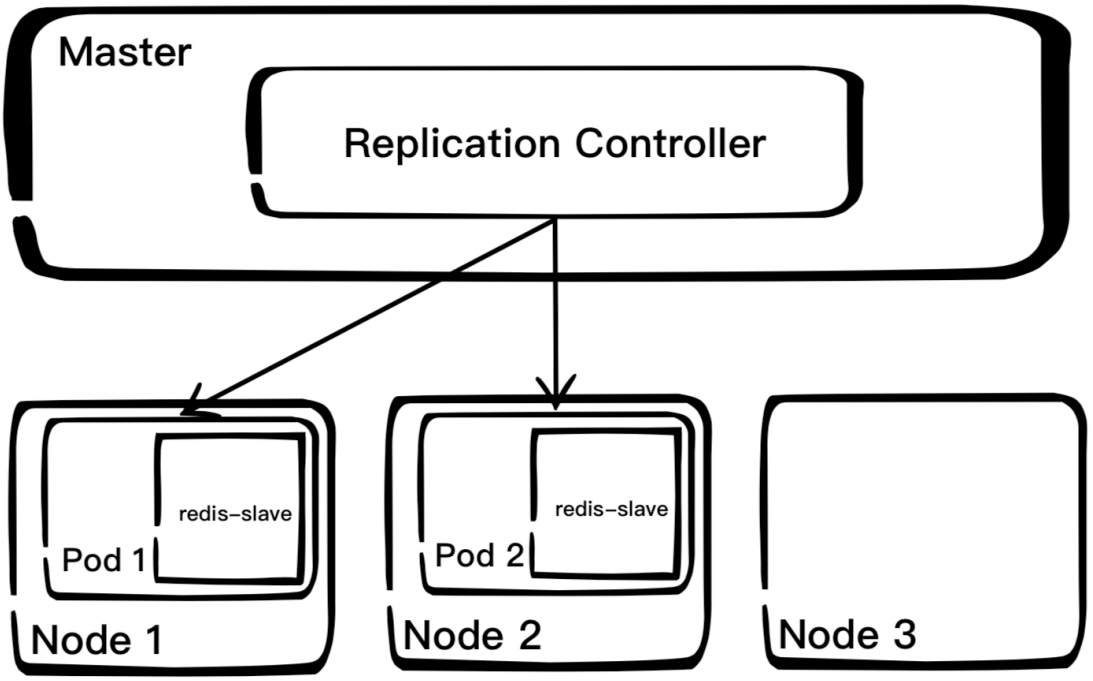
```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
    tier: frontend
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: tomcat-demo
        image: tomcat
```

```
        imagePullPolicy: IfNotPresent
        env:
        - name: GET_HOSTS_FROM
          value: dns
        # If your cluster config does not include a dns service, then to
        # instead access environment variables to find service host
        # info, comment out the 'value: dns' line above, and uncomment the
        # line below.
        # value: env
        ports:
        - containerPort: 80

[root@vlnx251101 test]# kubectl create -f frontend.yaml
replicationcontroller "frontend" created

[root@vlnx251101 test]# kubectl describe replicationcontrollers/frontend
Name:           frontend
Namespace:      default
Selector:       tier=frontend
Labels:         app=app-demo
                tier=frontend
Annotations:    <none>
Replicas:       3 current / 3 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
```

```
  Labels:  app=app-demo
           tier=frontend
  Containers:
   tomcat-demo:
    Image:       tomcat
    Port:        80/TCP
    Host Port:   0/TCP
    Environment:
      GET_HOSTS_FROM:  dns
    Mounts:            <none>
  Volumes:             <none>
Events:
  Type    Reason            Age   From                    Message
  ----    ------            ----  ----                    -------
  Normal  SuccessfulCreate  6m    replication-controller  Created pod: frontend-r8rj4
  Normal  SuccessfulCreate  6m    replication-controller  Created pod: frontend-9fqvq
  Normal  SuccessfulCreate  6m    replication-controller  Created pod: frontend-dxqcp
```

当我们定义了一个 RC 并提交到 Kubernetes 集群中以后，Master 节点上的 Controller Manager 组件就得到通知，定期巡检系统中当前存活的目标 Pod，并确保目标 Pod 示例的数量刚好等于此 RC 的期望值，如果有过多的 Pod 副本在运行，系统就会停掉一些 Pod，否则系统会再自动创建一些 Pod。可以说，通过 RC，Kubernetes 实现了用户应用集群的高可用性，并且大大减少了系统管理员在传统 IT 环境中需要完成的许多手工运维工作（如主机监控脚本、应用监控脚本。故障恢复脚本等）。

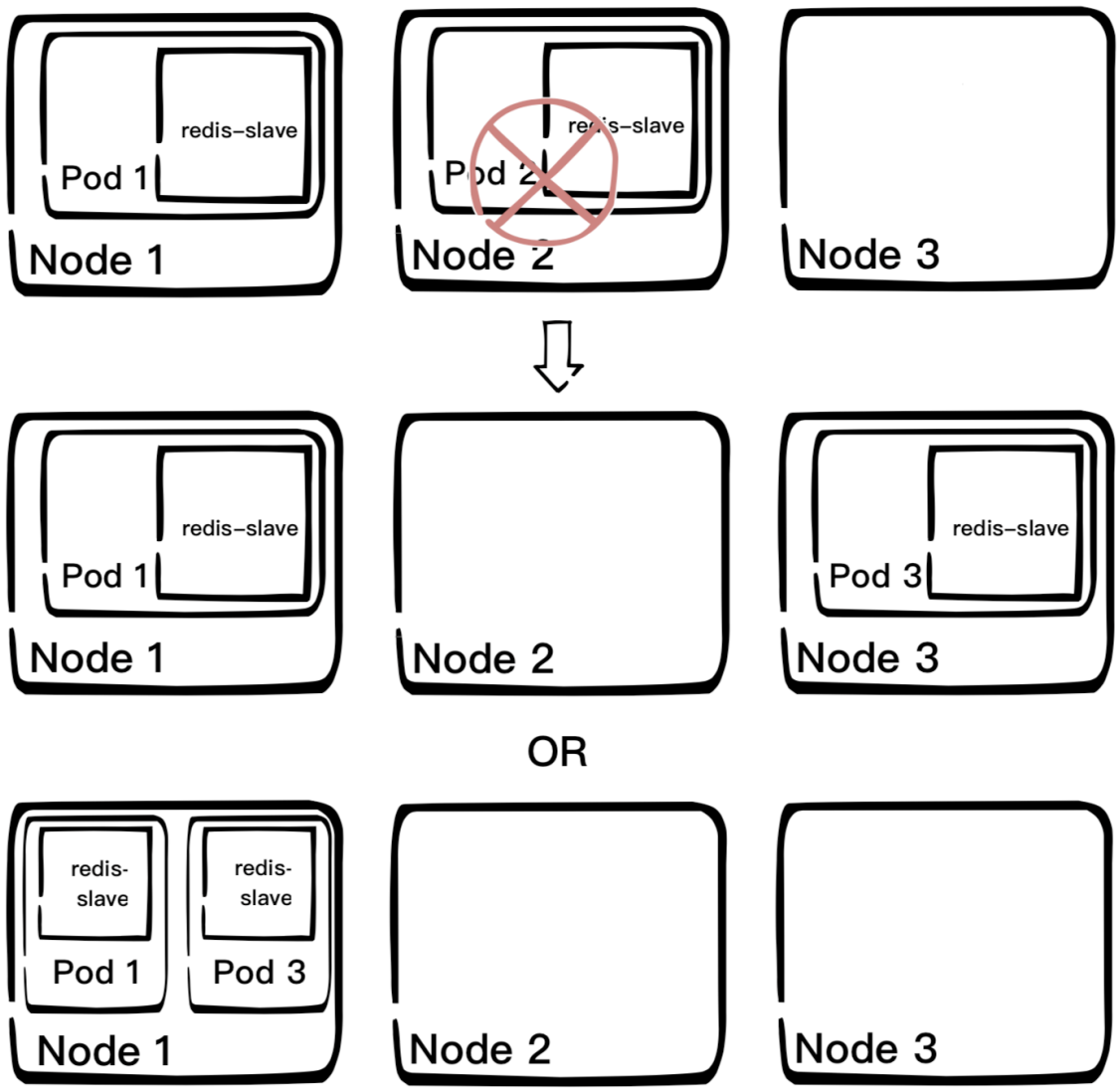
下面以 3 个 Node 节点的集群为例，说明 Kubernetes 如何通过 RC 来实现 Pod 副本数量自动控制的机制。假如 RC 里定义 redis-slave 这个 Pod 需要保持 2 个副本，系统将可能在其中的两个 Node 上创建 Pod，如下图：

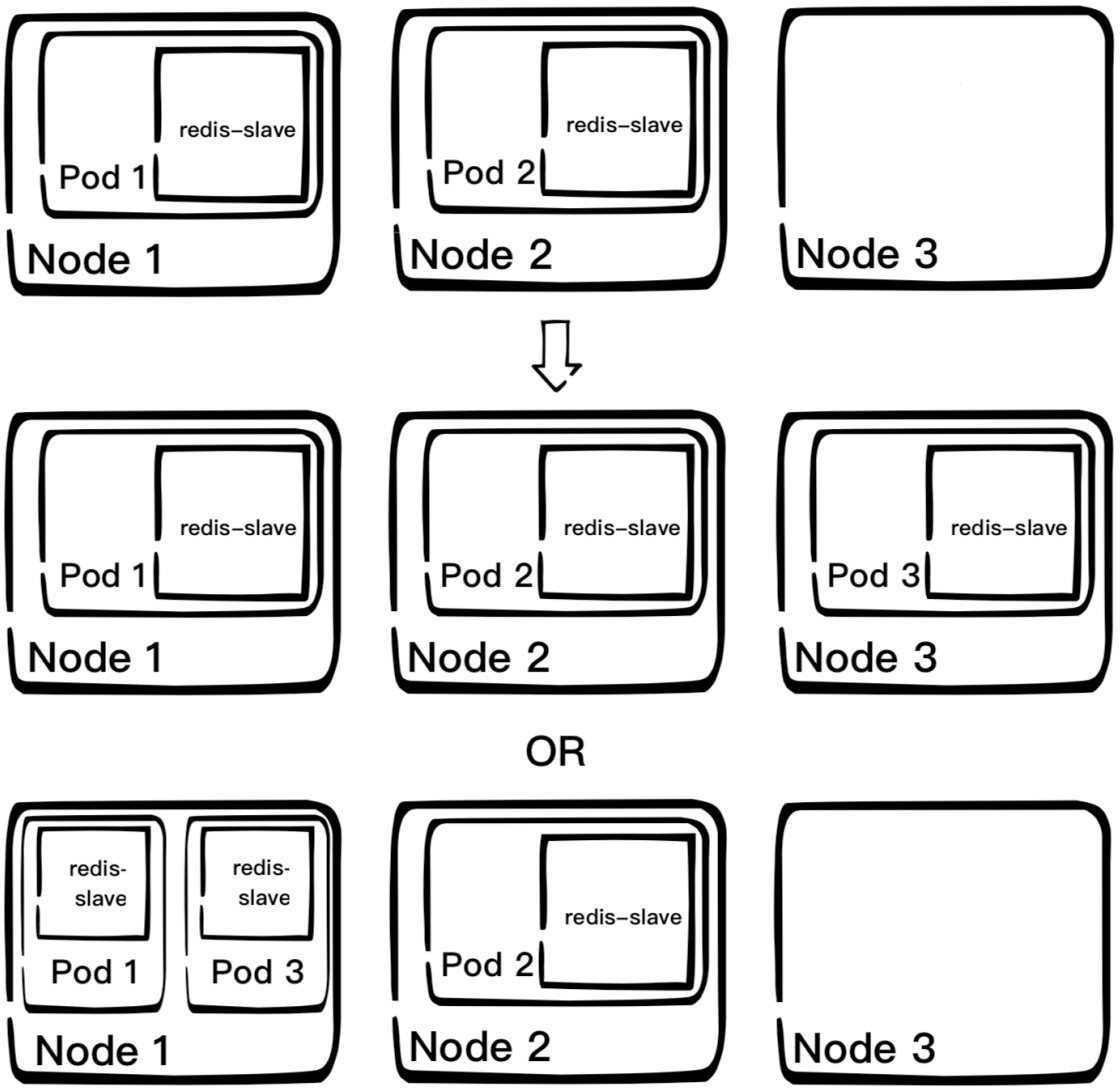


假设 Node2 上的 Pod 2 意外终止，根据 RC 定义的 replicas 数量 2，Kubernetes 将会自动创建并启动一个新的 Pod，以保证整个集群始终有两个 redis-slave Pod 运行。
如下图所示，系统可能选择 Node 3 或者 Node 1 来创建一个新的 Pod。
此外，在运行时，可以通过修改 RC 的副本数量，来实现 Pod 的动态缩放（Saling）功能，这可以通过执行 scale 命令来完成：

```
kubectl scale rc redis-slave --replicas=3
```

Scaling 的执行结果如下图：


redis-slave
Pod 1
Node 1
redis-slave
Pod 2
Node 2
Node 3
redis-slave
Pod 1
Node 1
Node 2
redis-slave
Pod 3
Node 3
OR
redis-slave
Pod 1
redis-slave
Pod 3
Node 1
Node 2
Node 3



需要注意的是，删除 RC 并不会影响该 RC 已创建好的 Pod。为了删除所有的 Pod ，可以设置 replicas 的值为 0 ，然后更新该 RC。另外，kubectl 提供了 stop 和 delete 命令来一次性删除 RC 和 RC 控制的全部 Pod。

当应用升级时，通常会通过 Build 一个新的 Docker 镜像，并用新的镜像版本来替代旧的版本的方式达到目的。在系统升级过程中，希望的是平滑的方式，比如当前系统中 10 个对应的旧版本的 Pod，最佳的方式是旧版本的 Pod 每次停止一个，同时创建一个新版本的 Pod，在整个升级过程中，此消波长，而运行中的 Pod 数量始终是 10 个，几分钟后，当所有的 Pod 都已经是新版

本时，升级过程完成。通过 RC 的机制，Kubernetes 很容易的实现了这种高级实用的特性，被称为 “滚动升级”（Rolling Update）。

Replica Set

Replica Set，官方解释为 “下一代的 RC”，它与 RC 当前存在的唯一区别是：Replica Sets 支持基于集合的 Label selector（Set-based selector），而 RC 只支持基于等式的 Label Selector（equality-based selector），这使得 Replica Set 的功能更强，下面是等价于之前 RC 例子的 Replica Set 的定义：

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
...
```

（省去了 Pod 末班部分的内容）

```
[root@vlnx251101 test]# kubectl create -f frontend.yaml
```

```
replicaset.extensions/frontend created



[root@vlnx251101 test]# kubectl describe rs/frontend
Name:         frontend
Namespace:    default
Selector:     tier=frontend,tier in (frontend)
Labels:       app=app-demo
              tier=frontend
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0
Failed
Pod Template:
  Labels:  app=app-demo
           tier=frontend
  Containers:
   tomcat-demo:
    Image:       tomcat
    Port:        80/TCP
    Host Port:   0/TCP
    Environment:
      GET_HOSTS_FROM:  dns
    Mounts:            <none>
  Volumes:             <none>
Events:
  Type    Reason            Age   From
Message
  ----    ------            ----  ----                    -
------
```

```
  Normal  SuccessfulCreate  19s   replicaset-controller
Created pod: frontend-xl4xq
  Normal  SuccessfulCreate  19s   replicaset-controller
Created pod: frontend-4wqls
  Normal  SuccessfulCreate  19s   replicaset-controller
Created pod: frontend-4bk2g
```

kubectl 命令行工具适用于 RC 的绝大部分命令都同样适用于 Replica Set。此外，当前我们很少单独使用 Replica Set，它主要被 Deployment 这个更高级的资源对象所使用，从而形成一整套 Pod 创建、删除、更新的编排机制。当使用 Deplyment 时，无需关心它是如何创建和维护 Replica Set 的，这一切都是自动发生的。

Replica Set 与 Deployment 这两个重要资源对象逐步替换了之前的 RC 的作用，是 Kubernetes v1.3 里 Pod 自动扩容（伸缩）这个告警功能实现的基础，也将继续在 Kubernetes 未来的版本中发挥重要的作用。
总结 RC（Replica Set）的一些特性：

- 在大多数情况下，通过定义一个 RC 实现 Pod 的创建过程及副本数量的自动控制。
- RC 里包括完整的 Pod 定义模板。
- RC 通过 Label Selector 机制实现对 Pod 副本的自动控制。
- 通过改变 RC 里的 Pod 副本数量，可以实现 Pod 的扩容或伸缩功能。
- 通过改变 RC 里 Pod 模板中的镜像版本，可以实现 Pod 的滚动升级功能。

Deployment

Deployment 是 Kubernetes v1.2 引入的新概念，引入的目的是为了更好的解决 Pod 的编排问题。为此，Deployment 在内部使用了 Replica Set 来实现目的，无论从 Deployment 的作用与目的、它的 YAML 定义，还是从它的具体命令操作来看，都可以把它看做 RC 的一次升级两者的相似度超过 90%。

Deployment 相对于 RC 的最大升级是可以随时知道当前 Pod “部署”的进度。实际上由于一个 Pod 的创建、调度、绑定节点以及在目标 Node 上启动对应的容器这一完整过程需要一定的时间，所以期待系统启动 N 个 Pod 副本的目标状态，实际上是一个连续变化的“部署过程”导致的最终状态。

Deployment 的典型使用场景有以下几个。

- 创建一个 Deployment 对象来生成对应的 Replica Set 并完成 Pod 副本的创建过程。
- 检查 Deployment 的状态来部署动作是否完成（Pod 副本的数量是否达到预期的值）。
- 更新 Deployment 以创建新的 Pod（比如镜像升级）。
- 如果当前 Deployment 不稳定，则回滚到一个早先的 Deployment 版本。
- 暂停 Deployment 以便于下一次性修改多个 PodTemplateSpec 的配置项，之后再回复 Deployment，进行新的发布。
- 扩展 Deployment 以应对高负载。
- 查看 Deployment 的状态，以此作为发布是否完成的指标。
- 清理不在需要的旧版本 ReplicaSets。

Deployment 的定义与 Replica Set 的定义很类似，除了 API 声明与 Kind 类型等有所区别：

```
apiVersion: extensions/v1beta1
kind: Deployment
```

```
metadata:
  name: nginx-deployment


apiVersion: v1
kind: ReplicaSet
metadata:
  name: nginx-repset
```

下面通过运行一些例子来一起直观的感受这个新概念。首先创建一个 `tomcat` 的 `Deployment` 描述文件，内容如下：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: tomcat-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
     - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: tomcat-dome
```

```
        image: tomcat
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
```

创建Deployment

```
[root@vlnx251101 ~]# kubectl apply -f tomcat-
deployment.yaml
deployment "tomcat-deployment" created
```

查看Deployment信息

```
[root@vlnx251101 ~]# kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE
AVAILABLE   AGE
tomcat-deployment   1         1         1
1           1h
```

对上述输出中涉及的数量解释如下：

- DESIRED：Pod 副本数量的期望值，即 Deployment 里定义的 Replica。
- CURRENT：当前 Replica 的值，实际上是 Deployment 所创建的 Replica Set 里的 Replica 值，这个值不断增加，直到达到 DESIRED 为止，表名整个部署过程完成。
- UP-TO-DATE：最新版本的 Pod 的副本数量，用于只是在滚动升级的过程中，有多少个 Pod 副本已经成功升级。
- AVAILABLE：当集群中可用的 Pod 副本数量，即集群中当前存活的 Pod 数量。

运行下述命令查看对应的 Replica Set，看到它的命名与 Deployment 的名字有关系：

```
[root@vlnx251101 ~]# kubectl get rs
NAME                                DESIRED   CURRENT
READY     AGE
tomcat-deployment-1473713186   1         1
1         1h
```

运行下述命令查看创建的 Pod，发现 Pod 的命名与 Deployment 对应的 Replica Set 的名字为前缀，这种命名很清晰的表明了一个 Replica Set 创建了那些 Pod，对于 Pod 滚动升级这种复杂的过程来说，很容易排查错误：

```
[root@vlnx251101 ~]# kubectl get pods
NAME                                        READY
STATUS          RESTARTS   AGE
tomcat-deployment-1473713186-6kg7m   1/1       Running
0          1h
```

运行 kubectl describe deployments ，可以清楚的看到 Deployment 控制的 Pod 的水平扩展过程。
Pod 的管理对象，除了 RC 和 Deployment，还包括ReplicaSet、Deployment、StatefulSet、、Job 等，分别用于不同的应用场景中。

Deployment 扩容

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
```

```
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

```
[root@vlnx251101 test]# kubectl create -f nginx-
deployment.yaml
deployment.apps/nginx-deployment created
```

```
[root@vlnx251101 test]# kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE
AVAILABLE   AGE
nginx-deployment    1         1         1            1
           1m
```

```
[root@vlnx251101 test]# kubectl get rs
NAME                          DESIRED   CURRENT   READY
    AGE
```

```
nginx-deployment-67594d6bf6   1         1         1
         2m
```

```
[root@vlnx251101 test]# kubectl get pods
NAME                                READY     STATUS
RESTARTS   AGE
nginx-deployment-67594d6bf6-rfgx6   1/1       Running
0          3m
```

```
[root@vlnx251101 test]# kubectl scale deployment nginx-
deployment --replicas 2
deployment.extensions/nginx-deployment scaled
```

```
[root@vlnx251101 test]# kubectl get rs
NAME                          DESIRED   CURRENT   READY
    AGE
nginx-deployment-67594d6bf6   2         2         1
         5m
```

回退Deployment

```
[root@vlnx251101 test]# kubectl set image
deployment/nginx-deployment nginx=nginx:1.9.1
deployment.extensions/nginx-deployment image updated
```

```
[root@vlnx251101 test]# kubectl rollout status deployments
nginx-deployment
Waiting for deployment "nginx-deployment" rollout to
finish: 1 old replicas are pending termination...
^C

[root@vlnx251101 test]kubectl get rs
NAME                          DESIRED   CURRENT   READY
    AGE
nginx-deployment-67594d6bf6   0         0         0
        1m
nginx-deployment-6fdbb596db   1         1         1
        36s

[root@vlnx251101 test]# kubectl get pods
NAME                                READY
STATUS               RESTARTS   AGE
nginx-deployment-67594d6bf6-xs8tq   1/1       Running
            0          33s
nginx-deployment-6fdbb596db-vmpks   0/1
ContainerCreating   0          9s

[root@vlnx251101 test]# kubectl rollout history
deployment/nginx-deployment
deployments "nginx-deployment"
REVISION  CHANGE-CAUSE
1         <none>
2         <none>

[root@vlnx251101 test]# kubectl rollout history
deployment/nginx-deployment --revision=1
```

```
deployments "nginx-deployment" with revision #1
Pod Template:
  Labels:       app=nginx
        pod-template-hash=2315082692
  Containers:
   nginx:
    Image:      nginx:1.7.9
    Port:       80/TCP
    Host Port:  0/TCP
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>
```

```
[root@vlnx251101 test]# kubectl rollout undo
deployment/nginx-deployment --to-revision=1
```

StatefulSet

在 Kubernetes 系统中，Pod 的管理对象 RC、Deployment、DaemonSet 和 Job 都是面向无状态的服务。但现实中有很多服务是有状态的，特别是一些复杂的 中间件集群，例如 MySQL 集群、MongoDB 集群、Akka 集群、Zookeeper 集群的等，这些应用集群有以下一些共同点。

- 每个节点都有固定的身份 ID，通过这个 ID，集群中的成员可以相互发现并且通信。
- 集群的规模是比较固定的，集群规模不能随意变动。
- 集群里的每个节点都是有状态的，通常会持久化数据到永久存储中。

- 如果磁盘损坏，则集群里的某个节点无法正常运行，集群功能受损。

如果用 RC/Deployment 控制 Pod 副本数的方式来实现上述有状态的集群，则会发现第一点是无法满足的，因为 Pod 的名字是随机产生的，Pod 的 IP 地址也是在运行期才确定且可能有变动的，事先无法为每个 Pod 确定唯一不变的 ID。另外，为了能够在其它节点上恢复某个失败的节点，这种集群中的 Pod 需要挂接某种共享存储，为了解决这个问题，Kubernetes 从 v1.4 版本开始引入了 PetSet 这个新的资源对象，而且在 v1.5 版本时更名为 StatefulSet，StatefulSet 从本质上来说，可以看做 Deployment/RC 的一个特殊变种，它有如下一些特性。

- StatefulSet 里的每个 Pod 都有稳定、唯一的网络标识，可以用来发现集群内的其他成员。假设 StatefulSet 的名字叫 kafka，那么第一个 Pod 叫 kafka-0，第二个叫 kafka-1，以此类推。
- StatefulSet 控制的 Pod 副本的启停顺序是受控的，操作第 n 个 Pod 时，前 n-1 个 Pod 已经是运行且准备好的状态。
- StatefulSet 里的 Pod 采用稳定的持久化存储卷，通过 PV/PVC 来实现，删除 Pod 时默认不会删除与 StatefulSet 相关的存储卷（为了保证数据的安全）。

StatefulSet 除了要与 PV 卷捆绑使用以存储 Pod 的状态数据，还要与 Headless Service 配合使用，即在每个 StatefulSet 的定义中要声明它属于哪个 Headless Service。Headless Service 与普通 Service 的关键区别在于，它没有 Custer IP，如果解析 Headless Service 的 DNS 域名，则返回的是该 Service 对应的全部 Pod 的 Endpoint 列表。StatefulSet 在 Headless Service 的基础上又为 StatefulSet 控制的每个 Pod 实例创建了一个 DNS 域名，这个域名的格式为：

```
$(podname).$(headless service name)
```

比如一个 3 节点的 Kafka 的 StatefulSet 集群，对应的 Headless Service 的名字为 kafka，StatefulSet 的名字为 kafka，则 StatefulSet 里面的 3 个 Pod 的 DNS 名称分别为 kafka-

0.kafka、kafka-1.kafka、kafka-2.kafka，这些 DNS 名称可以直接在集群的配置文件中固定下来。

```
[root@vlnx251101 test]# vim web.yaml

---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchExpressions:
    - key: app
      operator: In
      values:
```

```
      - nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
      volumes:
      - name: www
        emptyDir: {}
```

```
[root@vlnx251101 test]# kubectl create -f web.yaml
service/nginx created
statefulset.apps/web created

# 查看创建的headless service和statefulset
$ kubectl get service nginx
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
nginx     None         <none>        80/TCP     1m

$ kubectl get statefulset web
NAME      DESIRED   CURRENT   AGE
web       2         2         2m
```

```
# 根据volumeClaimTemplates自动创建PVC（在GCE中会自动创建
kubernetes.io/gce-pd类型的volume）
$ kubectl get pvc
NAME        STATUS
VOLUME                                                  CAPACITY
ACCESSMODES   AGE
www-web-0   Bound     pvc-d064a004-d8d4-11e6-b521-
42010a800002   1Gi        RWO            16s
www-web-1   Bound     pvc-d06a3946-d8d4-11e6-b521-
42010a800002   1Gi        RWO            16s

# 查看创建的Pod，他们都是有序的
$ kubectl get pods -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          5m
web-1     1/1       Running   0          4m

# 使用nslookup查看这些Pod的DNS
$ kubectl run -i --tty --image busybox dns-test --
restart=Never --rm /bin/sh
/ # nslookup web-0.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-
system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.2.10
/ # nslookup web-1.nginx
Server:    10.0.0.10
```

```
Address 1: 10.0.0.10 kube-dns.kube-
system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.3.12
/ # nslookup web-0.nginx.default.svc.cluster.local
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-
system.svc.cluster.local

Name:      web-0.nginx.default.svc.cluster.local
Address 1: 10.244.2.10
```

DaemonSet

DaemonSet 确保全部（或者一些）Node 上运行一个 Pod 的副本。当有 Node 加入集群时，也会为他们新增一个 Pod 。当有 Node 从集群移除时，这些 Pod 也会被回收。删除 DaemonSet 将会删除它创建的所有 Pod。

使用 DaemonSet 的一些典型用法：

- 运行集群存储 daemon，例如在每个 Node 上运行 `glusterd`、`ceph`。
- 在每个 Node 上运行日志收集 daemon，例如`fluentd`、`logstash`。
- 在每个 Node 上运行监控 daemon，例如 Prometheus Node Exporter、`collectd`、Datadog 代理、New Relic 代理，或 Ganglia `gmond`。

```
[root@vlnx251101 test]# vim daemonset.yaml
apiVersion: apps/v1
```

```yaml
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: k8s.gcr.io/fluentd-elasticsearch:1.20
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
```

```
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

```
[root@vlnx251101 test]# kubectl create -f daemonset.yaml
daemonset.apps/fluentd-elasticsearch created
```

```
[root@vlnx251101 test]# kubectl get daemonsets -n kube-
system
NAME                    DESIRED   CURRENT   READY     UP-
TO-DATE   AVAILABLE   NODE SELECTOR   AGE
fluentd-elasticsearch   3         3         3
3             3           <none>          4m
```

```
[root@vlnx251101 test]# kubectl describe daemonsets -n
kube-system
Name:           fluentd-elasticsearch
```

```
Selector:       name=fluentd-elasticsearch
Node-Selector:  <none>
Labels:         k8s-app=fluentd-logging
Annotations:    <none>
Desired Number of Nodes Scheduled: 3
Current Number of Nodes Scheduled: 3
Number of Nodes Scheduled with Up-to-date Pods: 3
Number of Nodes Scheduled with Available Pods: 3
Number of Nodes Misscheduled: 0
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0
Failed
Pod Template:
  Labels:  name=fluentd-elasticsearch
  Containers:
   fluentd-elasticsearch:
    Image:       k8s.gcr.io/fluentd-elasticsearch:1.20
    Port:        <none>
    Host Port:  <none>
    Limits:
      memory:  200Mi
    Requests:
      cpu:         100m
      memory:      200Mi
    Environment:  <none>
    Mounts:
      /var/lib/docker/containers from
varlibdockercontainers (ro)
      /var/log from varlog (rw)
  Volumes:
   varlog:
```

```
    Type:          HostPath (bare host directory volume)
    Path:          /var/log
    HostPathType:
  varlibdockercontainers:
    Type:          HostPath (bare host directory volume)
    Path:          /var/lib/docker/containers
    HostPathType:
Events:
  Type    Reason             Age   From                    Message
  ----    ------             ----  ----                    -------
  Normal  SuccessfulCreate   4m    daemonset-controller    Created pod: fluentd-elasticsearch-vbksg
  Normal  SuccessfulCreate   4m    daemonset-controller    Created pod: fluentd-elasticsearch-7d7rj
  Normal  SuccessfulCreate   4m    daemonset-controller    Created pod: fluentd-elasticsearch-9144x
```

```
[root@vlnx251101 test]# kubectl get pod -n kube-system
NAME                                    READY     STATUS    RESTARTS   AGE
fluentd-elasticsearch-7d7rj             1/1       Running   0          4m
fluentd-elasticsearch-9144x             1/1       Running   0          4m
fluentd-elasticsearch-vbksg             1/1       Running   0          4m
```

Horizontal Pod Autoscaler

通过手工执行 kubectl scale 命令，可以实现 Pod 扩容或缩容。如果仅仅至此为止，显然不符合 Google 对 Kubernetes 的定位目标 —— 自动化、智能化。在 Google 看来，分布式系统要能够根据当前负载的变化情况自动触发水平扩展或缩容的行为，因为这一过程可能是频繁发生的、不可预料的，所以手动控制的方式是不实现的。

因此，Kubernetes 的 v1.0 版本实现后，这帮大牛们就已经在默默研究 Pod 智能扩容的特性了，并在 Kubernetes v1.1 版本中首次发布了这一重量级新特性—— Horizontal Pod Autoscaling （Pod 横向自动扩容，简称 HPA）。随后的 v1.2 版本中 HPA 被升级为稳定版本（apiVersion: autoscaling/v1），但同时仍然保留旧版本（apiVersion: extensions/v1beta1）。从 v1.6 版本为 autoscaling/v2alpha1，仍在不断演进过程中。

HPA 与之前的RC、Deployment、一样，也属于一种 Kubernetes 资料对象。通过追踪分析 RC 控制的所有目标 Pod 的负载变化情况，来确定是否需要针对性地调整目标 Pod 的副本数，这是 HPA 的实现原理。当前，HPA 可以有以下两种方式作为 Pod 负载的调度指标。

- CPUUtilizationPercentage。
- 应用程序自定义的度量指标，比如服务在每秒内的相应的请求数（TPS 或 QPS）。

CPUUtilizationPercentage 是一个计算平均值，即目标 Pod 所有副本自身的 CPU 利用率的平均值。一个 Pod 自身的 CPU 利用率是该 Pod 当前的 CPU 的使用量除以它的 Pod Request 的值，比如定义一个 Pod 的 Pod Request 为 0.4，而当前的 Pod 的CPU 使用量为 0.2，则它的 CPU 使用率为 50%，如此一来，就可以算出来一个 RC 控制的所有 Pod 副本的 CPU 利用率的算术平均值了。如果某一时刻 CPUUtilizationPercentage 的值超过 80%，则意味着当前的 Pod 副本数很可能不足以支撑接来下更多的请求，需要进行动态扩容，而当前请求高峰时段过去后，Pod 的 CPU 利用率又会降下来，此时对应的 Pod 副本数应该自动减少到一个合理的水平。

CPUUtilizationPercentage 计算过程中使用到的 Pod 的 CPU 使用量通常是 1min 内的平均值，目前通过查询 Heapster 扩展组件来得到这个值，所以需要安装部署 Heapster，这样一来便增加了系统的复杂度和实时 HPA 特性的复杂度，因此，未来的计划是 Kubernetes 自身实现一个基础性能数据采集某块，从而更好的支持 HPA 和其他需要用到基础性能数据的功能模块。此外，如果目标 Pod 没有定义 Pod Request 的值，则无法使用 CPUUtilizationPercentage 来实现 Pod 很想自动扩容的能力。除了使用 CPUUtilizationPercentage，Kubernetes 从 v1.2 版本开始尝试支持应用程序自定义的度量指标，目前仍然为实验特性，不建议在生产环境使用。下面是 HPA 定义的一个具体例子：

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    kind: Deployment
```

```
    name: php-apache
  targetCPUUtilizationPercentage: 90
```

根据上面的定义，可以知道这个 HPA 控制的目标对象为一个名叫 php-apache 的 Deployment 里的 Pod 副本，当这些 Pod 副本的 CPUUtilizationPercentage 的值超过 90% 时会触发自动动态扩容行为，扩容或缩容时必须满足一个约束条件是 Pod 的副本数要介于 1 与 10 之间。
除了可以通过之间定义 YAML 文件并且调用 kubectl create 的命令来创建一个 HPA 资源对象的方式，还能通过下面的简单命令直接创建等价的 HPA 对象：

```
kubectl autoscale deployment php-apache --cpu-percent=90 --min-1 --max=10
```