

讲堂 > 深入剖析Kubernetes > 文章详情

## 17 | 经典PaaS的记忆：作业副本与水平扩展

2018-10-01 张磊



### 17 | 经典PaaS的记忆：作业副本与水平扩展

朗读人：张磊 17'53" | 8.20M

你好，我是张磊。今天我和你分享的主题是：经典 PaaS 的记忆之作业副本与水平扩展。

在上一篇文章中，我为你详细介绍了 Kubernetes 项目中第一个重要的设计思想：控制器模式。


而在今天这篇文章中，我就来为你详细讲解一下，Kubernetes 里第一个控制器模式的完整实现：Deployment。

Deployment 看似简单，但实际上，它实现了 Kubernetes 项目中一个非常重要的功能：**Pod 的“水平扩展 / 收缩”（horizontal scaling out/in）**。这个功能，是从 PaaS 时代开始，一个平台级项目就必须具备的编排能力。

举个例子，如果你更新了 Deployment 的 Pod 模板（比如，修改了容器的镜像），那么 Deployment 就需要遵循一种叫作“滚动更新”（rolling update）的方式，来升级现有的容器。

而这个能力的实现，依赖的是 Kubernetes 项目中的一个非常重要的概念（API 对象）：ReplicaSet。

ReplicaSet 的结构非常简单，我们可以通过这个 YAML 文件查看一下：

 复制代码

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-set
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
```

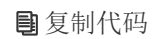
从这个 YAML 文件中，我们可以看到，一个 ReplicaSet 对象，其实就是由副本数目的定义和一个 Pod 模板组成的。不难发现，它的定义其实是 Deployment 的一个子集。

更重要的是，Deployment 控制器实际操纵的，正是这样的 ReplicaSet 对象，而不是 Pod 对象。

还记不记得我在上一篇文章[《编排其实很简单：谈谈“控制器”模型》](#)中曾经提出过这样一个问题：对于一个 Deployment 所管理的 Pod，它的 ownerReference 是谁？

所以，这个问题的答案就是：ReplicaSet。

明白了这个原理，我再来和你一起分析一个如下所示的 Deployment：

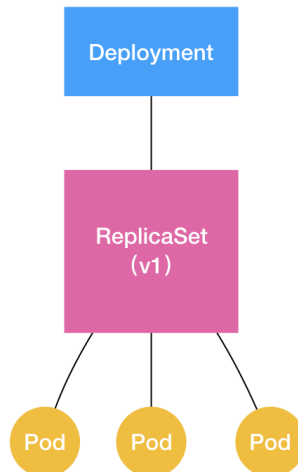


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

可以看到，这就是一个我们常用的 nginx-deployment，它定义的 Pod 副本个数是 3 ( spec.replicas=3 )。

那么，在具体的实现上，这个 Deployment，与 ReplicaSet，以及 Pod 的关系是怎样的呢？

我们可以用一张图把它描述出来：



通过这张图，我们就很清楚的看到，一个定义了 replicas=3 的 Deployment，与它的 ReplicaSet，以及 Pod 的关系，实际上是一种“层层控制”的关系。

其中，ReplicaSet 负责通过“控制器模式”，保证系统中 Pod 的个数永远等于指定的个数（比如，3 个）。这也正是 Deployment 只允许容器的 restartPolicy=Always 的主要原因：只有在容器能保证自己始终是 Running 状态的前提下，ReplicaSet 调整 Pod 的个数才有意义。

而在此基础上，Deployment 同样通过“控制器模式”，来操作 ReplicaSet 的个数和属性，进而实现“水平扩展 / 收缩”和“滚动更新”这两个编排动作。

其中，“水平扩展 / 收缩”非常容易实现，Deployment Controller 只需要修改它所控制的 ReplicaSet 的 Pod 副本个数就可以了。

比如，把这个值从 3 改成 4，那么 Deployment 所对应的 ReplicaSet，就会根据修改后的值自动创建一个新的 Pod。这就是“水平扩展”了；“水平收缩”则反之。

而用户想要执行这个操作的指令也非常简单，就是 `kubectl scale`，比如：

```
$ kubectl scale deployment nginx-deployment --replicas=4
deployment.apps/nginx-deployment scaled
```

[📄 复制代码](#)

那么，“滚动更新”又是什么意思，是如何实现的呢？

接下来，我还以这个 Deployment 为例，来为你讲解“滚动更新”的过程。

首先，我们来创建这个 nginx-deployment：

```
$ kubectl create -f nginx-deployment.yaml --record
```

[复制代码](#)

注意，在这里，我额外加了一个--record 参数。它的作用，是记录下你每次操作所执行的命令，以方便后面查看。

然后，我们来检查一下 nginx-deployment 创建后的状态信息：

```
$ kubectl get deployments
```

[复制代码](#)

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3	0	0	0	1s

在返回结果中，我们可以看到四个状态字段，它们的含义如下所示。

1. DESIRED：用户期望的 Pod 副本个数（spec.replicas 的值）；
2. CURRENT：当前处于 Running 状态的 Pod 的个数；
3. UP-TO-DATE：当前处于最新版本的 Pod 的个数，所谓最新版本指的是 Pod 的 Spec 部分与 Deployment 里 Pod 模板里定义的完全一致；
4. AVAILABLE：当前已经可用的 Pod 的个数，即：既是 Running 状态，又是最新版本，并且已经处于 Ready（健康检查正确）状态的 Pod 的个数。

可以看到，只有这个 AVAILABLE 字段，描述的才是用户所期望的最终状态。

而 Kubernetes 项目还为我们提供了一条指令，让我们可以实时查看 Deployment 对象的状态变化。这个指令就是 kubectl rollout status：


```
$ kubectl rollout status deployment/nginx-deployment
```

[复制代码](#)

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...  
deployment.apps/nginx-deployment successfully rolled out
```


在这个返回结果中，“2 out of 3 new replicas have been updated”意味着已经有 2 个 Pod 进入了 UP-TO-DATE 状态。

继续等待一会儿，我们就能看到这个 Deployment 的 3 个 Pod，就进入到了 AVAILABLE 状态：

 复制代码

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3	3	3	3	20s

此时，你可以尝试查看一下这个 Deployment 所控制的 ReplicaSet：

 复制代码

```
$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-3167673210	3	3	3	20s

如上所示，在用户提交了一个 Deployment 对象后，Deployment Controller 就会立即创建一个 Pod 副本个数为 3 的 ReplicaSet。这个 ReplicaSet 的名字，则是由 Deployment 的名字和一个随机字符串共同组成。


这个随机字符串叫作 pod-template-hash，在我们这个例子里就是：3167673210。

ReplicaSet 会把这个随机字符串加在它所控制的所有 Pod 的标签里，从而保证这些 Pod 不会与集群里的其他 Pod 混淆。

而 ReplicaSet 的 DESIRED、CURRENT 和 READY 字段的含义，和 Deployment 中是一致的。所以，相比之下，Deployment 只是在 ReplicaSet 的基础上，添加了 UP-TO-DATE 这个跟版本有关的状态字段。

这个时候，如果我们修改了 Deployment 的 Pod 模板，“滚动更新”就会被自动触发。

修改 Deployment 有很多方法。比如，我可以直接使用 kubectl edit 指令编辑 Etcd 里的 API 对象。

 复制代码

```
$ kubectl edit deployment/nginx-deployment
```

```
...
```

```
spec:
```

```
  containers:
```

```
  - name: nginx
```

```
    image: nginx:1.9.1 # 1.7.9 -> 1.9.1
```

```
    ports:
```

```
    - containerPort: 80
```

```
...
```

```
deployment.extensions/nginx-deployment edited
```

这个 `kubectl edit` 指令，会帮你直接打开 `nginx-deployment` 的 API 对象。然后，你就可以修改这里的 Pod 模板部分了。比如，在这里，我将 `nginx` 镜像的版本升级到了 1.9.1。

备注：`kubectl edit` 并不神秘，它不过是把 API 对象的内容下载到了本地文件，让你修改完成后再提交上去。

`kubectl edit` 指令编辑完成后，保存退出，Kubernetes 就会立刻触发“滚动更新”的过程。你还可以通过 `kubectl rollout status` 指令查看 `nginx-deployment` 的状态变化：

```
$ kubectl rollout status deployment/nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment.extensions/nginx-deployment successfully rolled out
```

[复制代码](#)

这时，你可以通过查看 Deployment 的 Events，看到这个“滚动更新”的流程：

```
$ kubectl describe deployment nginx-deployment
...
Events:
  Type     Reason             Age   From                    Message
  ----     -
  ...
  Normal   ScalingReplicaSet   24s   deployment-controller   Scaled up replica set nginx-deplo
  Normal   ScalingReplicaSet   22s   deployment-controller   Scaled down replica set nginx-deplo
  Normal   ScalingReplicaSet   22s   deployment-controller   Scaled up replica set nginx-deplo
  Normal   ScalingReplicaSet   19s   deployment-controller   Scaled down replica set nginx-deplo
  Normal   ScalingReplicaSet   19s   deployment-controller   Scaled up replica set nginx-deplo
  Normal   ScalingReplicaSet   14s   deployment-controller   Scaled down replica set nginx-deplo
```

[复制代码](#)

可以看到，首先，当你修改了 Deployment 里的 Pod 定义之后，Deployment Controller 会使用这个修改后的 Pod 模板，创建一个新的 ReplicaSet ( hash=1764197365 )，这个新的 ReplicaSet 的初始 Pod 副本数是：0。

然后，在 Age=24 s 的位置，Deployment Controller 开始将这个新的 ReplicaSet 所控制的 Pod 副本数从 0 个变成 1 个，即：“水平扩展”出一个副本。

紧接着，在 Age=22 s 的位置，Deployment Controller 又将旧的 ReplicaSet ( hash=3167673210 ) 所控制的旧 Pod 副本数减少一个，即：“水平收缩”成两

个副本。

如此交替进行，新 ReplicaSet 管理的 Pod 副本数，从 0 个变成 1 个，再变成 2 个，最后变成 3 个。而旧的 ReplicaSet 管理的 Pod 副本数则从 3 个变成 2 个，再变成 1 个，最后变成 0 个。这样，就完成了这一组 Pod 的版本升级过程。

像这样，将一个集群中正在运行的多个 Pod 版本，交替地逐一升级的过程，就是“滚动更新”。

在这个“滚动更新”过程完成之后，你可以查看一下新、旧两个 ReplicaSet 的最终状态：

```
$ kubectl get rs
```

[复制代码](#)

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1764197365	3	3	3	6s
nginx-deployment-3167673210	0	0	0	30s

其中，旧 ReplicaSet ( hash=3167673210 ) 已经被“水平收缩”成了 0 个副本。

这种“滚动更新”的好处是显而易见的。

比如，在升级刚开始的时候，集群里只有 1 个新版本的 Pod。如果这时，新版本 Pod 有问题启动不起来，那么“滚动更新”就会停止，从而允许开发和运维人员介入。而在这个过程中，由于应用本身还有两个旧版本的 Pod 在线，所以服务并不会受到太大的影响。

当然，这也就要求你一定要使用 Pod 的 Health Check 机制检查应用的运行状态，而不是简单地依赖于容器的 Running 状态。要不然的话，虽然容器已经变成 Running 了，但服务很有可能尚未启动，“滚动更新”的效果也就达不到了。

而为了进一步保证服务的连续性，Deployment Controller 还会确保，在任何时间窗口内，只有指定比例的 Pod 处于离线状态。同时，它也会确保，在任何时间窗口内，只有指定比例的新 Pod 被创建出来。这两个比例的值都是可以配置的，默认都是 DESIRED 值的 25%。

所以，在上面这个 Deployment 的例子中，它有 3 个 Pod 副本，那么控制器在“滚动更新”的过程中永远都会确保至少有 2 个 Pod 处于可用状态，至多只有 4 个 Pod 同时存在于集群中。这个策略，是 Deployment 对象的一个字段，名叫 RollingUpdateStrategy，如下所示：

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

[复制代码](#)



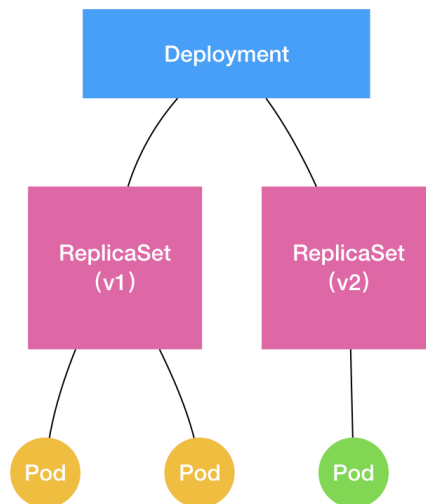
```
name: nginx-deployment
labels:
  app: nginx
spec:
...
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 1
```

在上面这个 RollingUpdateStrategy 的配置中，maxSurge 指定的是除了 DESIRED 数量之外，在一次“滚动”中，Deployment 控制器还可以创建多少个新 Pod；而 maxUnavailable 指的是，在一次“滚动”中，Deployment 控制器可以删除多少个旧 Pod。

同时，这两个配置还可以用前面我们介绍的百分比形式来表示，比如：

maxUnavailable=50%，指的是我们最多可以一次删除“50%\*DESIRED 数量”个 Pod。

结合以上讲述，现在我们可以扩展一下 Deployment、ReplicaSet 和 Pod 的关系图了。



如上所示，Deployment 的控制器，实际上控制的是 ReplicaSet 的数目，以及每个 ReplicaSet 的属性。

而一个应用的版本，对应的正是一个 ReplicaSet；这个版本应用的 Pod 数量，则由 ReplicaSet 通过它自己的控制器（ReplicaSet Controller）来保证。

通过这样的多个 ReplicaSet 对象，Kubernetes 项目就实现了对多个“应用版本”的描述。

而明白了“应用版本和 ReplicaSet 一一对应”的设计思想之后，我就可以为你讲解一下 **Deployment 对应用进行版本控制的具体原理**了。

这一次，我会使用一个叫 `kubectl set image` 的指令，直接修改 `nginx-deployment` 所使用的镜像。这个命令的好处就是，你可以不用像 `kubectl edit` 那样需要打开编辑器。

不过这一次，我把这个镜像名字修改成为了一个错误的名字，比如：`nginx:1.91`。这样，这个 Deployment 就会出现一个升级失败的版本。

我们一起来实践一下：

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.91
deployment.extensions/nginx-deployment image updated
```

[复制代码](#)

由于这个 `nginx:1.91` 镜像在 Docker Hub 中并不存在，所以这个 Deployment 的“滚动更新”被触发后，会立刻报错并停止。

这时，我们来检查一下 ReplicaSet 的状态，如下所示：

```
$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1764197365	2	2	2	24s
nginx-deployment-3167673210	0	0	0	35s
nginx-deployment-2156724341	2	2	0	7s

[复制代码](#)


通过这个返回结果，我们可以看到，新版本的 ReplicaSet（`hash=2156724341`）的“水平扩展”已经停止。而且此时，它已经创建了两个 Pod，但是它们都没有进入 READY 状态。这当然是因为这两个 Pod 都拉取不到有效的镜像。

与此同时，旧版本的 ReplicaSet（`hash=1764197365`）的“水平收缩”，也自动停止了。此时，已经有一个旧 Pod 被删除，还剩下两个旧 Pod。

那么问题来了，我们如何让这个 Deployment 的 3 个 Pod，都回滚到以前的旧版本呢？

我们只需要执行一条 `kubectl rollout undo` 命令，就能把整个 Deployment 回滚到上一个版本：

```
$ kubectl rollout undo deployment/nginx-deployment
deployment.extensions/nginx-deployment
```


 复制代码

很容易想到，在具体操作上，Deployment 的控制器，其实就是让这个旧 ReplicaSet ( hash=1764197365 ) 再次“扩展”成 3 个 Pod，而让新的 ReplicaSet ( hash=2156724341 ) 重新“收缩”到 0 个 Pod。

更进一步地，如果我想回滚到更早之前的版本，要怎么办呢？

首先，我需要使用 `kubectl rollout history` 命令，查看每次 Deployment 变更对应的版本。而由于我们在创建这个 Deployment 的时候，指定了 `--record` 参数，所以我们创建这些版本时执行的 `kubectl` 命令，都会被记录下来。这个操作的输出如下所示：


```
$ kubectl rollout history deployment/nginx-deployment
deployments "nginx-deployment"
REVISION    CHANGE-CAUSE
1           kubectl create -f nginx-deployment.yaml --record
2           kubectl edit deployment/nginx-deployment
3           kubectl set image deployment/nginx-deployment nginx=nginx:1.91
```

 复制代码

可以看到，我们前面执行的创建和更新操作，分别对应了版本 1 和版本 2，而那次失败的更新操作，则对应的是版本 3。


当然，你还可以通过这个 `kubectl rollout history` 指令，看到每个版本对应的 Deployment 的 API 对象的细节，具体命令如下所示：

```
$ kubectl rollout history deployment/nginx-deployment --revision=2
```

 复制代码

然后，我们就可以在 `kubectl rollout undo` 命令行最后，加上要回滚到的指定版本的版本号，就可以回滚到指定版本了。这个指令的用法如下：

```
$ kubectl rollout undo deployment/nginx-deployment --to-revision=2
deployment.extensions/nginx-deployment
```

 复制代码

这样，Deployment Controller 还会按照“滚动更新”的方式，完成对 Deployment 的降级操作。

不过，你可能已经想到了一个问题：我们对 Deployment 进行的每一次更新操作，都会生成一个新的 ReplicaSet 对象，是不是有些多余，甚至浪费资源呢？

没错。

所以，Kubernetes 项目还提供了—个指令，使得我们对 Deployment 的多次更新操作，最后只生成一个 ReplicaSet。

具体的做法是，在更新 Deployment 前，你要先执行—条 `kubectl rollout pause` 指令。它的用法如下所示：

```
$ kubectl rollout pause deployment/nginx-deployment
deployment.extensions/nginx-deployment paused
```

[复制代码](#)

这个 `kubectl rollout pause` 的作用，是让这个 Deployment 进入了一个“暂停”状态。

所以接下来，你就可以随意使用 `kubectl edit` 或者 `kubectl set image` 指令，修改这个 Deployment 的内容了。

由于此时 Deployment 正处于“暂停”状态，所以我们对 Deployment 的所有修改，都不会触发新的“滚动更新”，也不会创建新的 ReplicaSet。

而等到我们对 Deployment 修改操作都完成之后，只需要再执行—条 `kubectl rollout resume` 指令，就可以把这个 Deployment “恢复”回来，如下所示：

```
$ kubectl rollout resume deploy/nginx-deployment
deployment.extensions/nginx-deployment resumed
```

[复制代码](#)

而在这个 `kubectl rollout resume` 指令执行之前，在 `kubectl rollout pause` 指令之后的这段时间里，我们对 Deployment 进行的所有修改，最后只会触发—次“滚动更新”。

当然，我们可以通过检查 ReplicaSet 状态的变化，来验证—下 `kubectl rollout pause` 和 `kubectl rollout resume` 指令的执行效果，如下所示：

```
$ kubectl get rs
NAME                                DESIRED    CURRENT    READY    AGE
```

[复制代码](#)

nginx-1764197365	0	0	0	2m
nginx-3196763511	3	3	3	28s

通过返回结果，我们可以看到，只有一个 hash=3196763511 的 ReplicaSet 被创建了出来。

不过，即使你像上面这样小心翼翼地控制了 ReplicaSet 的生成数量，随着应用版本的不断增加，Kubernetes 中还是会为同一个 Deployment 保存很多很多不同的 ReplicaSet。

那么，我们又该如何控制这些“历史” ReplicaSet 的数量呢？

很简单，Deployment 对象有一个字段，叫作 spec.revisionHistoryLimit，就是 Kubernetes 为 Deployment 保留的“历史版本”个数。所以，如果把它设置为 0，你就再也不能做回滚操作了。

## 总结

在今天这篇文章中，我为你详细讲解了 Deployment 这个 Kubernetes 项目中最基本的编排控制器的实现原理和使用方法。

通过这些讲解，你应该了解到：Deployment 实际上是一个两层控制器。首先，它通过 ReplicaSet 的个数来描述应用的版本；然后，它再通过 ReplicaSet 的属性（比如 replicas 的值），来保证 Pod 的副本数量。

备注：Deployment 控制 ReplicaSet（版本），ReplicaSet 控制 Pod（副本数）。这个两层控制关系一定要牢记。

不过，相信你也能够感受到，Kubernetes 项目对 Deployment 的设计，实际上是代替我们完成了对“应用”的抽象，使得我们可以使用这个 Deployment 对象来描述应用，使用 kubectl rollout 命令控制应用的版本。

可是，在实际使用场景中，应用发布的流程往往千差万别，也可能有很多的定制化需求。比如，我的应用可能有会话黏连（session sticky），这就意味着“滚动更新”的时候，哪个 Pod 能下线，是不能随便选择的。

这种场景，光靠 Deployment 自己就很难应对了。对于这种需求，我在专栏后续文章中重点介绍的“自定义控制器”，就可以帮我们实现一个功能更加强大的 Deployment Controller。

当然，Kubernetes 项目本身，也提供了另外一种抽象方式，帮我们应对其他一些用 Deployment 无法处理的应用编排场景。这个设计，就是对有状态应用的管理，也是我在下一篇文章中要重点讲解的内容。

## 思考题

你听说过金丝雀发布 ( Canary Deployment ) 和蓝绿发布 ( Blue-Green Deployment ) 吗？你能说出它们是什么意思吗？

实际上，有了 Deployment 的能力之后，你可以非常轻松地用它来实现金丝雀发布、蓝绿发布，以及 A/B 测试等很多应用发布模式。这些问题的答案都在[这个 GitHub 库](#)，建议你在课后实践一下。

感谢你的收听，欢迎你给我留言。



版权归极客邦科技所有，未经许可不得转载

写留言

#### 精选留言



Tigerfive

👍 2

半夜从火车上醒来，就来看看有没有更新，果然没有让我失望！

2018-10-01

作者回复

国庆可以充电啦

2018-10-01