21 | 容器化守护进程的意义：DaemonSet

21 | 容器化守护进程的意义：DaemonSet

张磊 2018-10-10

17:27

讲述：张磊 大小：8.00M

你好，我是张磊。今天我和你分享的主题是：容器化守护进程的意义之 DaemonSet。

在上一篇文章中，我和你详细分享了使用 StatefulSet 编排“有状态应用”的过程。从中不难看出，StatefulSet 其实就是对现有典型运维业务的容器化抽象。也就是说，你一定有方法在不使用 Kubernetes、甚至不使用容器的情况下，自己 DIY 一个类似的方案出来。但是，一旦涉及到升级、版本管理等更工程化的能力，Kubernetes 的好处，才会更加凸现。

比如，如何对 StatefulSet 进行“滚动更新”（rolling update）？

很简单。你只要修改 StatefulSet 的 Pod 模板，就会自动触发“滚动更新”：

```
$ kubectl patch statefulset mysql --type='json' -p='[{"op": "replace",
"path": "/spec/template/spec/containers/0/image",
"value":"mysql:5.7.23"}]'

statefulset.apps/mysql patched
```

复制代码

在这里，我使用了 kubectl patch 命令。它的意思是，以“补丁”的方式（JSON 格式的）修改一个 API 对象的指定字段，也就是我在后面指定的“spec/template/spec/containers/0/image”。

这样，StatefulSet Controller 就会按照与 Pod 编号相反的顺序，从最后一个 Pod 开始，逐一更新这个 StatefulSet 管理的每个 Pod。而如果更新发生了错误，这次“滚动更新”就会停止。此外，StatefulSet 的“滚动更新”还允许我们进行更精细的控制，比如金丝雀发布（Canary Deploy）或者灰度发布，**这意味着应用的多个实例中被指定的一部分不会被更新到最新的版本。**

这个字段，正是 StatefulSet 的 spec.updateStrategy.rollingUpdate 的 partition 字段。

比如，现在我将前面这个 StatefulSet 的 partition 字段设置为 2：

```
$ kubectl patch statefulset mysql -p '{"spec":{"updateStrategy":
{"type":"RollingUpdate","rollingUpdate":{"partition":2}}}}'

statefulset.apps/mysql patched
```

复制代码

其中，kubectl patch 命令后面的参数（JSON 格式的），就是 partition 字段在 API 对象里的路径。所以，上述操作等同于直接使用 kubectl edit 命令，打开这个对象，把 partition 字段修改为 2。

这样，我就指定了当 Pod 模板发生变化的时候，比如 MySQL 镜像更新到 5.7.23，那么只有序号大于或者等于 2 的 Pod 会被更新到这个版本。并且，如果你删除或者重启了序号小于 2 的 Pod，等它再次启动后，也会保持原先的 5.7.2 版本，绝不会被升级到 5.7.23 版本。

StatefulSet 可以说是 Kubernetes 项目中最为复杂的编排对象，希望你课后能认真消化，动手实践一下这个例子。

而在今天这篇文章中，我会为你重点讲解一个相对轻松的知识点：DaemonSet。

顾名思义，DaemonSet 的主要作用，是让你在 Kubernetes 集群里，运行一个 Daemon Pod。 所以，这个 Pod 有如下三个特征：

1. 这个 Pod 运行在 Kubernetes 集群里的每一个节点（Node）上；

2. 每个节点上只有一个这样的 Pod 实例；
3. 当有新的节点加入 Kubernetes 集群后，该 Pod 会自动地在新节点上被创建出来；而当旧节点被删除后，它上面的 Pod 也相应地会被回收掉。

这个机制听起来很简单，但 Daemon Pod 的意义确实是非常重要的。我随便给你列举几个例子：

1. 各种网络插件的 Agent 组件，都必须运行在每一个节点上，用来处理这个节点上的容器网络；
2. 各种存储插件的 Agent 组件，也必须运行在每一个节点上，用来在这个节点上挂载远程存储目录，操作容器的 Volume 目录；
3. 各种监控组件和日志组件，也必须运行在每一个节点上，负责这个节点上的监控信息和日志搜集。

更重要的是，跟其他编排对象不一样，DaemonSet 开始运行的时机，很多时候比整个 Kubernetes 集群出现的时机都要早。

这个乍一听起来可能有点儿奇怪。但其实你来想一下：如果这个 DaemonSet 正是一个网络插件的 Agent 组件呢？

这个时候，整个 Kubernetes 集群里还没有可用的容器网络，所有 Worker 节点的状态都是 NotReady（NetworkReady=false）。这种情况下，普通的 Pod 肯定不能运行在这个集群上。所以，这也就意味着 DaemonSet 的设计，必须要有某种“过人之处”才行。

为了弄清楚 DaemonSet 的工作原理，我们还是按照老规矩，先从它的 API 对象的定义说起。

```
apiVersion: apps/v1
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
```

□复制代码

这个 DaemonSet，管理的是一个 fluentd-elasticsearch 镜像的 Pod。这个镜像的功能非常实用：通过 fluentd 将 Docker 容器里的日志转发到 ElasticSearch

中。

可以看到，DaemonSet 跟 Deployment 其实非常相似，只不过是没有 replicas 字段；它也使用 selector 选择管理所有携带了 name=fluentd-elasticsearch 标签的 Pod。

而这些 Pod 的模板，也是用 template 字段定义的。在这个字段中，我们定义了一个使用 fluentd-elasticsearch:1.20 镜像的容器，而且这个容器挂载了两个 hostPath 类型的 Volume，分别对应宿主机的 /var/log 目录和 /var/lib/docker/containers 目录。

显然，fluentd 启动之后，它会从这两个目录里搜集日志信息，并转发给 ElasticSearch 保存。这样，我们通过 ElasticSearch 就可以很方便地检索这些日志了。

需要注意的是，Docker 容器里应用的日志，默认会保存在宿主机的 /var/lib/docker/containers/{{. 容器 ID}}/{{. 容器 ID}}-json.log 文件里，所以这个目录正是 fluentd 的搜集目标。

那么，**DaemonSet 又是如何保证每个 Node 上有且只有一个被管理的 Pod 呢?**

显然，这是一个典型的“控制器模型”能够处理的问题。

DaemonSet Controller，首先从 Etcd 里获取所有的 Node 列表，然后遍历所有的 Node。这时，它就可以很容易地去检查，当前这个 Node 上是不是有一个携带了 name=fluentd-elasticsearch 标签的 Pod 在运行。

而检查的结果，可能有这么三种情况：

1. 没有这种 Pod，那么就意味着要在这个 Node 上创建这样一个 Pod；
2. 有这种 Pod，但是数量大于 1，那就说明要把多余的 Pod 从这个 Node 上删除掉；
3. 正好只有一个这种 Pod，那说明这个节点是正常的。

其中，删除节点（Node）上多余的 Pod 非常简单，直接调用 Kubernetes API 就可以了。

但是，**如何在指定的 Node 上创建新 Pod 呢?**

如果你已经熟悉了 Pod API 对象的话，那一定可以立刻说出答案：用 nodeSelector，选择 Node 的名字即可。

```
nodeSelector:
```

```
    name: <Node 名字 >
```

复制代码

没错。

不过，在 Kubernetes 项目里，nodeSelector 其实已经是一个将要被废弃的字段了。因为，现在有了一个新的、功能更完善的字段可以代替它，即：nodeAffinity。我来举个例子：

```
    apiVersion: v1
    kind: Pod
    metadata:
    name: with-node-affinity
    spec:
    affinity:
    nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
    - matchExpressions:
    - key: metadata.name
    operator: In
    values:
    - node-geektime
```

复制代码

在这个 Pod 里，我声明了一个 spec.affinity 字段，然后定义了一个 nodeAffinity。其中，spec.affinity 字段，是 Pod 里跟调度相关的一个字段。关于它的完整内容，我会在讲解调度策略的时候再详细阐述。

而在这里，我定义的 nodeAffinity 的含义是：

1. requiredDuringSchedulingIgnoredDuringExecution：它的意思是说，这个 nodeAffinity 必须在每次调度的时候予以考虑。同时，这也意味着你可以设置在某些情况下不考虑这个 nodeAffinity；
2. 这个 Pod，将来只允许运行在“metadata.name”是“node-geektime”的节点上。

在这里，你应该注意到 nodeAffinity 的定义，可以支持更加丰富的语法，比如 operator: In（即：部分匹配；如果你定义 operator: Equal，就是完全匹配），这也正是 nodeAffinity 会取代 nodeSelector 的原因之一。

> *备注：其实在大多数时候，这些 Operator 语义没啥用处。所以说，在学习开源项目的时候，一定要学会抓住“主线”。不要顾此失彼。*

所以，**我们的 DaemonSet Controller 会在创建 Pod 的时候，自动在这个 Pod 的 API 对象里，加上这样一个 nodeAffinity 定义。**其中，需要绑定的节点名字，正是当前正在遍历的这个 Node。

当然，DaemonSet 并不需要修改用户提交的 YAML 文件里的 Pod 模板，而是在向 Kubernetes 发起请求之前，直接修改根据模板生成的 Pod 对象。这个思路，也正是我在前面讲解 Pod 对象时介绍过的。

此外，DaemonSet 还会给这个 Pod 自动加上另外一个与调度相关的字段，叫作 tolerations。这个字段意味着这个 Pod，会“容忍”（Toleration）某些 Node 的“污点”（Taint）。

而 DaemonSet 自动加上的 tolerations 字段，格式如下所示：

```
apiVersion: v1
kind: Pod
metadata:
name: with-toleration
spec:
tolerations:
- key: node.kubernetes.io/unschedulable
operator: Exists
effect: NoSchedule
```

复制代码

这个 Toleration 的含义是：“容忍”所有被标记为 unschedulable“污点”的 Node；“容忍”的效果是允许调度。

> *备注：关于如何给一个 Node 标记上“污点”，以及这里具体的语法定义，我会在后面介绍调度器的时候做详细介绍。这里，你可以简单地把“污点”理解为一种特殊的 Label。*

而在正常情况下，被标记了 unschedulable“污点”的 Node，是不会有任何 Pod 被调度上去的（effect: NoSchedule）。可是，DaemonSet 自动地给被管理的 Pod 加上了这个特殊的 Toleration，就使得这些 Pod 可以忽略这个限制，继而保证每个节点上都会被调度一个 Pod。当然，如果这个节点有故障的话，这个 Pod 可能会启动失败，而 DaemonSet 则会始终尝试下去，直到 Pod 启动成功。

这时，你应该可以猜到，我在前面介绍到的**DaemonSet 的“过人之处”，其实就是依靠 Toleration 实现的。**

假如当前 DaemonSet 管理的，是一个网络插件的 Agent Pod，那么你就必须在这个 DaemonSet 的 YAML 文件里，给它的 Pod 模板加上一个能够“容忍”node.kubernetes.io/network-unavailable“污点”的 Toleration。正如下面这个例子所示：

```
...
template:
metadata:
labels:
name: network-plugin-agent
spec:
tolerations:
- key: node.kubernetes.io/network-unavailable
operator: Exists
effect: NoSchedule
```

□复制代码

在 Kubernetes 项目中，当一个节点的网络插件尚未安装时，这个节点就会被自动加上名为node.kubernetes.io/network-unavailable的“污点”。

而通过这样一个 Toleration，调度器在调度这个 Pod 的时候，就会忽略当前节点上的“污点”，从而成功地将网络插件的 Agent 组件调度到这台机器上启动起来。

这种机制，正是我们在部署 Kubernetes 集群的时候，能够先部署 Kubernetes 本身、再部署网络插件的根本原因：因为当时我们所创建的 Weave 的 YAML，实际上就是一个 DaemonSet。

> *这里，你也可以再回顾一下第 11 篇文章《从 0 到 1：搭建一个完整的 Kubernetes 集群》中的相关内容。*

至此，通过上面这些内容，你应该能够明白，**DaemonSet 其实是一个非常简单的控制器。**在它的控制循环中，只需要遍历所有节点，然后根据节点上是否有被管理 Pod 的情况，来决定是否要创建或者删除一个 Pod。

只不过，在创建每个 Pod 的时候，DaemonSet 会自动给这个 Pod 加上一个 nodeAffinity，从而保证这个 Pod 只会在指定节点上启动。同时，它还会自动给这个 Pod 加上一个 Toleration，从而忽略节点的 unschedulable“污点”。

当然，**你也可以在 Pod 模板里加上更多种类的 Toleration，从而利用 DaemonSet 实现自己的目的。**比如，在这个 fluentd-elasticsearch DaemonSet 里，我就给它加上了这样的 Toleration：

```
tolerations:
- key: node-role.kubernetes.io/master
effect: NoSchedule
```

□复制代码

这是因为在默认情况下，Kubernetes 集群不允许用户在 Master 节点部署 Pod。因为，Master 节点默认携带了一个叫作node-role.kubernetes.io/master的“污点”。所以，为了能在 Master 节点上部署 DaemonSet 的 Pod，我就必须让这个 Pod“容忍”这个“污点”。

在理解了 DaemonSet 的工作原理之后，接下来我就通过一个具体的实践来帮你更深入地掌握 DaemonSet 的使用方法。

> *备注：需要注意的是，在 Kubernetes v1.11 之前，由于调度器尚不完善，DaemonSet 是由 DaemonSet Controller 自行调度的，即它会直接设置 Pod 的 spec.nodename 字段，这样就可以跳过调度器了。但是，这样的做法很快就会被废除，所以在这里我也不推荐你再花时间学习这个流程了。*

首先，创建这个 DaemonSet 对象：

```
$ kubectl create -f fluentd-elasticsearch.yaml
```

□复制代码

需要注意的是，在 DaemonSet 上，我们一般都应该加上 resources 字段，来限制它的 CPU 和内存使用，防止它占用过多的宿主机资源。

而创建成功后，你就能看到，如果有 N 个节点，就会有 N 个 fluentd-elasticsearch Pod 在运行。比如在我们的例子里，会有两个 Pod，如下所示：

```
$ kubectl get pod -n kube-system -l name=fluentd-elasticsearch
NAME READY STATUS RESTARTS AGE
fluentd-elasticsearch-dqfv9 1/1 Running 0 53m
fluentd-elasticsearch-pf9z5 1/1 Running 0 53m
```

□复制代码

而如果你此时通过 kubectl get 查看一下 Kubernetes 集群里的 DaemonSet 对象：

```
$ kubectl get ds -n kube-system fluentd-elasticsearch
NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE
fluentd-elasticsearch 2 2 2 2 2 <none> 1h
```

□复制代码

> *备注：Kubernetes 里比较长的 API 对象都有短名字，比如 DaemonSet 对应的是 ds，Deployment 对应的是 deploy。*

就会发现 DaemonSet 和 Deployment 一样，也有 DESIRED、CURRENT 等多个状态字段。这也就意味着，DaemonSet 可以像 Deployment 那样，进行版本管理。这个版本，可以使用 kubectl rollout history 看到：

```
$ kubectl rollout history daemonset fluentd-elasticsearch -n kube-system
daemonsets "fluentd-elasticsearch"
REVISION CHANGE-CAUSE
1 <none>
```

□复制代码

接下来，我们来把这个 DaemonSet 的容器镜像版本到 v2.2.0：

```
$ kubectl set image ds/fluentd-elasticsearch fluentd-elasticsearch=k8s.gcr.io/fluentd-elasticsearch:v2.2.0 --record -n=kube-system
```

□复制代码

这个 kubectl set image 命令里，第一个 fluentd-elasticsearch 是 DaemonSet 的名字，第二个 fluentd-elasticsearch 是容器的名字。

这时候，我们可以使用 kubectl rollout status 命令看到这个“滚动更新”的过程，如下所示：

```
$ kubectl rollout status ds/fluentd-elasticsearch -n kube-system
Waiting for daemon set "fluentd-elasticsearch" rollout to finish: 0 out of 2 new pods have been updated...
Waiting for daemon set "fluentd-elasticsearch" rollout to finish: 0 out of 2 new pods have been updated...
Waiting for daemon set "fluentd-elasticsearch" rollout to finish: 1 of 2 updated pods are available...
daemon set "fluentd-elasticsearch" successfully rolled out
```

□复制代码

注意，由于这一次我在升级命令后面加上了–record 参数，所以这次升级使用到的指令就会自动出现在 DaemonSet 的 rollout history 里面，如下所示：

```
$ kubectl rollout history daemonset fluentd-elasticsearch -n kube-system
daemonsets "fluentd-elasticsearch"
REVISION CHANGE-CAUSE
1 <none>
2 kubectl set image ds/fluentd-elasticsearch fluentd-elasticsearch=k8s.gcr.io/fluentd-elasticsearch:v2.2.0 --namespace=kube-system --record=true
```

□复制代码

有了版本号，你也就可以像 Deployment 一样，将 DaemonSet 回滚到某个指定的历史版本了。

而我在前面的文章中讲解 Deployment 对象的时候，曾经提到过，Deployment 管理这些版本，靠的是“一个版本对应一个 ReplicaSet 对象”。可是，DaemonSet 控制器操作的直接就是 Pod，不可能有 ReplicaSet 这样的对象参与其中。**那么，它的这些版本又是如何维护的呢？**

所谓，一切皆对象！

在 Kubernetes 项目中，任何你觉得需要记录下来的状态，都可以被用 API 对象的方式实现。当然，“版本”也不例外。

Kubernetes v1.7 之后添加了一个 API 对象，名叫**ControllerRevision**，专门用来记录某种 Controller 对象的版本。比如，你可以通过如下命令查看 fluentd-elasticsearch 对应的 ControllerRevision：

```
$ kubectl get controllerrevision -n kube-system -l name=fluentd-elasticsearch
NAME CONTROLLER REVISION AGE
fluentd-elasticsearch-64dc6799c9 daemonset.apps/fluentd-elasticsearch 2 1h
```

□复制代码

而如果你使用 kubectl describe 查看这个 ControllerRevision 对象：

```
$ kubectl describe controllerrevision fluentd-elasticsearch-64dc6799c9 -n kube-system
Name: fluentd-elasticsearch-64dc6799c9
```

```
Namespace: kube-system
Labels: controller-revision-hash=2087235575
name=fluentd-elasticsearch
Annotations: deprecated.daemonset.template.generation=2
kubernetes.io/change-cause=kubectl set image ds/fluentd-
elasticsearch fluentd-elasticsearch=k8s.gcr.io/fluentd-
elasticsearch:v2.2.0 --record=true --namespace=kube-system
API Version: apps/v1
Data:
Spec:
Template:
$ Patch: replace
Metadata:
Creation Timestamp: <nil>
Labels:
Name: fluentd-elasticsearch
Spec:
Containers:
Image: k8s.gcr.io/fluentd-elasticsearch:v2.2.0
Image Pull Policy: IfNotPresent
Name: fluentd-elasticsearch
...
Revision: 2
Events: <none>
```

□复制代码

就会看到，这个 ControllerRevision 对象，实际上是在 Data 字段保存了该版本对应的完整的 DaemonSet 的 API 对象。并且，在 Annotation 字段保存了创建这个对象所使用的 kubectl 命令。

接下来，我们可以尝试将这个 DaemonSet 回滚到 Revision=1 时的状态：

```
$ kubectl rollout undo daemonset fluentd-elasticsearch --to-revision=1
-n kube-system
daemonset.extensions/fluentd-elasticsearch rolled back
```

□复制代码

这个 kubectl rollout undo 操作，实际上相当于读取到了 Revision=1 的 ControllerRevision 对象保存的 Data 字段。而这个 Data 字段里保存的信息，就是 Revision=1 时这个 DaemonSet 的完整 API 对象。

所以，现在 DaemonSet Controller 就可以使用这个历史 API 对象，对现有的 DaemonSet 做一次 PATCH 操作（等价于执行一次 kubectl apply -f “旧的 DaemonSet 对象”），从而把这个 DaemonSet“更新”到一个旧版本。

这也是为什么，在执行完这次回滚完成后，你会发现，DaemonSet 的 Revision 并不会从 Revision=2 退回到 1，而是会增加成 Revision=3。这是因为，一个新的 ControllerRevision 被创建了出来。

总结

在今天这篇文章中，我首先简单介绍了 StatefulSet 的“滚动更新”，然后重点讲解了本专栏的第三个重要编排对象：DaemonSet。

相比于 Deployment，DaemonSet 只管理 Pod 对象，然后通过 nodeAffinity 和 Toleration 这两个调度器的小功能，保证了每个节点上有且只有一个 Pod。这个控制器的实现原理简单易懂，希望你能够快速掌握。

与此同时，DaemonSet 使用 ControllerRevision，来保存和管理自己对应的“版本”。这种“面向 API 对象”的设计思路，大大简化了控制器本身的逻辑，也正是 Kubernetes 项目“声明式 API”的优势所在。

而且，相信聪明的你此时已经想到了，StatefulSet 也是直接控制 Pod 对象的，那么它是不是也在使用 ControllerRevision 进行版本管理呢？

没错。在 Kubernetes 项目里，ControllerRevision 其实是一个通用的版本管理对象。这样，Kubernetes 项目就巧妙地避免了每种控制器都要维护一套冗余的代码和逻辑的问题。

思考题

我在文中提到，在 Kubernetes v1.11 之前，DaemonSet 所管理的 Pod 的调度过程，实际上都是由 DaemonSet Controller 自己而不是由调度器完成的。你能说出这其中有哪些原因吗？