11 | 从0到1：搭建一个完整的Kubernetes集群

11 | 从0到1：搭建一个完整的Kubernetes集群

张磊 2018-09-17

17:20

讲述：张磊 大小：7.95M

你好，我是张磊。今天我和你分享的主题是：从 0 到 1 搭建一个完整的 Kubernetes 集群。

在上一篇文章中，我介绍了 kubeadm 这个 Kubernetes 半官方管理工具的工作原理。既然 kubeadm 的初衷是让 Kubernetes 集群的部署不再让人头疼，那么这篇文章，我们就来使用它部署一个完整的 Kubernetes 集群吧。

> *备注：这里所说的“完整”，指的是这个集群具备 Kubernetes 项目在 GitHub 上已经发布的所有功能，并能够模拟生产环境的所有使用需求。但并不代表这个集群是生产级别可用的：类似于高可用、授权、多租户、灾难备份等生产级别集群的功能暂时不在本篇文章的讨论范围。*
> *目前，kubeadm 的高可用部署已经有了第一个发布。但是，这个特性还没有 GA（生产可用），所以包括了大量的手动工作，跟我们所预期的一键部署还有一定距离。GA 的日期预计是 2018 年底到 2019 年初。届时，如果有机会我会再和你分享这部分内容。*

这次部署，我不会依赖于任何公有云或私有云的能力，而是完全在 Bare-metal 环境中完成。这样的部署经验会更有普适性。而在后续的讲解中，如非特殊强调，我也都会以本次搭建的这个集群为基础。

准备工作

首先，准备机器。最直接的办法，自然是到公有云上申请几个虚拟机。当然，如果条件允许的话，拿几台本地的物理服务器来组集群是最好不过了。这些机器只要满足如下几个条件即可：

1. 满足安装 Docker 项目所需的要求，比如 64 位的 Linux 操作系统、3.10 及以上的内核版本；
2. x86 或者 ARM 架构均可；
3. 机器之间网络互通，这是将来容器之间网络互通的前提；
4. 有外网访问权限，因为需要拉取镜像；
5. 能够访问到gcr.io、quay.io这两个 docker registry，因为有小部分镜像需要在这里拉取；
6. 单机可用资源建议 2 核 CPU、8 GB 内存或以上，再小的话问题也不大，但是能调度的 Pod 数量就比较有限了；
7. 30 GB 或以上的可用磁盘空间，这主要是留给 Docker 镜像和日志文件用的。

在本次部署中，我准备的机器配置如下：

1. 2 核 CPU、 7.5 GB 内存；
2. 30 GB 磁盘；
3. Ubuntu 16.04；
4. 内网互通；
5. 外网访问权限不受限制。

> *备注：在开始部署前，我推荐你先花几分钟时间，回忆一下 Kubernetes 的架构。*

然后，我再和你介绍一下今天实践的目标：

1. 在所有节点上安装 Docker 和 kubeadm；
2. 部署 Kubernetes Master；
3. 部署容器网络插件；
4. 部署 Kubernetes Worker；
5. 部署 Dashboard 可视化插件；
6. 部署容器存储插件。

好了，现在，就来开始这次集群部署之旅吧！

安装 kubeadm 和 Docker

我在上一篇文章《 Kubernetes 一键部署利器：kubeadm》中，已经介绍过 kubeadm 的基础用法，它的一键安装非常方便，我们只需要添加 kubeadm 的源，然后直接使用 apt-get 安装即可，具体流程如下所示：

> *备注：为了方便讲解，我后续都直接会在 root 用户下进行操作*

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
$ cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
$ apt-get update
$ apt-get install -y docker.io kubeadm
```

复制代码

在上述安装 kubeadm 的过程中，kubeadm 和 kubelet、kubectl、kubernetes-cni 这几个二进制文件都会被自动安装好。

另外，这里我直接使用 Ubuntu 的 docker.io 的安装源，原因是 Docker 公司每次发布的最新的 Docker CE（社区版）产品往往还没有经过 Kubernetes 项目的验证，可能会有兼容性方面的问题。

部署 Kubernetes 的 Master 节点

在上一篇文章中，我已经介绍过 kubeadm 可以一键部署 Master 节点。不过，在本篇文章中既然要部署一个“完整”的 Kubernetes 集群，那我们不妨稍微提高一下难度：通过配置文件来开启一些实验性功能。

所以，这里我编写了一个给 kubeadm 用的 YAML 文件（名叫：kubeadm.yaml）：

```
apiVersion: kubeadm.k8s.io/v1alpha1
kind: MasterConfiguration
controllerManagerExtraArgs:
horizontal-pod-autoscaler-use-rest-clients: "true"
horizontal-pod-autoscaler-sync-period: "10s"
node-monitor-grace-period: "10s"
apiServerExtraArgs:
runtime-config: "api/all=true"
kubernetesVersion: "stable-1.11"
```

复制代码

这个配置中，我给 kube-controller-manager 设置了：

```
horizontal-pod-autoscaler-use-rest-clients: "true"
```

复制代码

这意味着，将来部署的 kube-controller-manager 能够使用自定义资源（Custom Metrics）进行自动水平扩展。这是我后面文章中会重点介绍的一个内容。

其中，“stable-1.11”就是 kubeadm 帮我们部署的 Kubernetes 版本号，即：Kubernetes release 1.11 最新的稳定版，在我的环境下，它是 v1.11.1。你也可以直接指定这个版本，比如：kubernetesVersion: “v1.11.1”。

然后，我们只需要执行一句指令：

```
$ kubeadm init --config kubeadm.yaml
```

复制代码

就可以完成 Kubernetes Master 的部署了，这个过程只需要几分钟。部署完成后，kubeadm 会生成一行指令：

```
kubeadm join 10.168.0.2:6443 --token 00bwbx.uvnaa2ewjflwu1ry --discovery-token-ca-cert-hash sha256:00eb62a2a6020f94132e3fe1ab721349bbcd3e9b94da9654cfe15f2985ebd711
```

复制代码

这个 kubeadm join 命令，就是用来给这个 Master 节点添加更多工作节点（Worker）的命令。我们在后面部署 Worker 节点的时候马上会用到它，所以找一个地方把这条命令记录下来。

此外，kubeadm 还会提示我们第一次使用 Kubernetes 集群所需要的配置命令：

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

复制代码

而需要这些配置命令的原因是：Kubernetes 集群默认需要加密方式访问。所以，这几条命令，就是将刚刚部署生成的 Kubernetes 集群的安全配置文件，保存到当前用户的.kube 目录下，kubectl 默认会使用这个目录下的授权信息访问 Kubernetes 集群。

如果不这么做的话，我们每次都需要通过 export KUBECONFIG 环境变量告诉 kubectl 这个安全配置文件的位置。

现在，我们就可以使用 kubectl get 命令来查看当前唯一一个节点的状态了：

```
$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
master NotReady master 1d v1.11.1
```

复制代码

可以看到，这个 get 指令输出的结果里，Master 节点的状态是 NotReady，这是为什么呢?

在调试 Kubernetes 集群时，最重要的手段就是用 kubectl describe 来查看这个节点（Node）对象的详细信息、状态和事件（Event），我们来试一下：

```
$ kubectl describe node master
...
Conditions:
...
Ready False ... KubeletNotReady runtime network not ready:
NetworkReady=false reason:NetworkPluginNotReady message:docker:
network plugin is not ready: cni config uninitialized
```

复制代码

通过 kubectl describe 指令的输出，我们可以看到 NodeNotReady 的原因在于，我们尚未部署任何网络插件。

另外，我们还可以通过 kubectl 检查这个节点上各个系统 Pod 的状态，其中，kube-system 是 Kubernetes 项目预留的系统 Pod 的工作空间（Namepsace，注意它并不是 Linux Namespace，它只是 Kubernetes 划分不同工作空间的单位）：

```
$ kubectl get pods -n kube-system
NAME READY STATUS RESTARTS AGE
coredns-78fcdf6894-j9s52 0/1 Pending 0 1h
```

```
coredns-78fcdf6894-jm4wf 0/1 Pending 0 1h
etcd-master 1/1 Running 0 2s
kube-apiserver-master 1/1 Running 0 1s
kube-controller-manager-master 0/1 Pending 0 1s
kube-proxy-xbd47 1/1 NodeLost 0 1h
kube-scheduler-master 1/1 Running 0 1s
```

复制代码

可以看到，CoreDNS、kube-controller-manager 等依赖于网络的 Pod 都处于 Pending 状态，即调度失败。这当然是符合预期的：因为这个 Master 节点的网络尚未就绪。

部署网络插件

在 Kubernetes 项目“一切皆容器”的设计理念指导下，部署网络插件非常简单，只需要执行一句 kubectl apply 指令，以 Weave 为例：

```
$ kubectl apply -f https://git.io/weave-kube-1.6
```

复制代码

部署完成后，我们可以通过 kubectl get 重新检查 Pod 的状态：

```
$ kubectl get pods -n kube-system
NAME READY STATUS RESTARTS AGE
coredns-78fcdf6894-j9s52 1/1 Running 0 1d
coredns-78fcdf6894-jm4wf 1/1 Running 0 1d
etcd-master 1/1 Running 0 9s
kube-apiserver-master 1/1 Running 0 9s
kube-controller-manager-master 1/1 Running 0 9s
kube-proxy-xbd47 1/1 Running 0 1d
kube-scheduler-master 1/1 Running 0 9s
weave-net-cmk27 2/2 Running 0 19s
```

复制代码

可以看到，所有的系统 Pod 都成功启动了，而刚刚部署的 Weave 网络插件则在 kube-system 下面新建了一个名叫 weave-net-cmk27 的 Pod，一般来说，这些 Pod 就是容器网络插件在每个节点上的控制组件。

Kubernetes 支持容器网络插件，使用的是一个名叫 CNI 的通用接口，它也是当前容器网络的事实标准，市面上的所有容器网络开源项目都可以通过 CNI 接入 Kubernetes，比如 Flannel、Calico、Canal、Romana 等等，它们的部署方式也都是类似的“一键部署”。关于这些开源项目的实现细节和差异，我会在后续的网络部分详细介绍。

至此，Kubernetes 的 Master 节点就部署完成了。如果你只需要一个单节点的 Kubernetes，现在你就可以使用了。不过，在默认情况下，Kubernetes 的 Master 节点是不能运行用户 Pod 的，所以还需要额外做一个小操作。在本篇的最后部分，我会介绍到它。

部署 Kubernetes 的 Worker 节点

Kubernetes 的 Worker 节点跟 Master 节点几乎是相同的，它们运行着的都是一个 kubelet 组件。唯一的区别在于，在 kubeadm init 的过程中，kubelet 启动后，Master 节点上还会自动运行 kube-apiserver、kube-scheduler、kube-controller-manger 这三个系统 Pod。

所以，相比之下，部署 Worker 节点反而是最简单的，只需要两步即可完成。

第一步，在所有 Worker 节点上执行“安装 kubeadm 和 Docker”一节的所有步骤。

第二步，执行部署 Master 节点时生成的 kubeadm join 指令：

```
$ kubeadm join 10.168.0.2:6443 --token 00bwbx.uvnaa2ewjflwu1ry --discovery-token-ca-cert-hash sha256:00eb62a2a6020f94132e3fe1ab721349bbcd3e9b94da9654cfe15f2985ebd711
```

复制代码

通过 Taint/Toleration 调整 Master 执行 Pod 的策略

我在前面提到过，默认情况下 Master 节点是不允许运行用户 Pod 的。而 Kubernetes 做到这一点，依靠的是 Kubernetes 的 Taint/Toleration 机制。

它的原理非常简单：一旦某个节点被加上了一个 Taint，即被“打上了污点”，那么所有 Pod 就都不能在这个节点上运行，因为 Kubernetes 的 Pod 都有“洁癖”。

除非，有个别的 Pod 声明自己能“容忍”这个“污点”，即声明了 Toleration，它才可以在这个节点上运行。

其中，为节点打上“污点”（Taint）的命令是：

```
$ kubectl taint nodes node1 foo=bar:NoSchedule
```

复制代码
这时，该 node1 节点上就会增加一个键值对格式的 Taint，即：foo=bar:NoSchedule。其中值里面的 NoSchedule，意味着这个 Taint 只会在调度新 Pod 时产生作用，而不会影响已经在 node1 上运行的 Pod，哪怕它们没有 Toleration。

那么 Pod 又如何声明 Toleration 呢?

我们只要在 Pod 的.yaml 文件中的 spec 部分，加入 tolerations 字段即可：

```
apiVersion: v1
kind: Pod
...
spec:
tolerations:
```

```
- key: "foo"
operator: "Equal"
value: "bar"
effect: "NoSchedule"
```

复制代码

这个 Toleration 的含义是，这个 Pod 能“容忍”所有键值对为 foo=bar 的 Taint（ operator: “Equal”， “等于”操作）。

现在回到我们已经搭建的集群上来。这时，如果你通过 kubectl describe 检查一下 Master 节点的 Taint 字段，就会有所发现了：

```
$ kubectl describe node master
Name: master
Roles: master
Taints: node-role.kubernetes.io/master:NoSchedule
```

复制代码

可以看到，Master 节点默认被加上了node-role.kubernetes.io/master:NoSchedule 这样一个“污点”，其中“键”是node-role.kubernetes.io/master，而没有提供“值”。

此时，你就需要像下面这样用“Exists”操作符（operator: “Exists”， “存在”即可）来说明，该 Pod 能够容忍所有以 foo 为键的 Taint，才能让这个 Pod 运行在该 Master 节点上：

```
apiVersion: v1
kind: Pod
...
spec:
tolerations:
- key: "foo"
operator: "Exists"
effect: "NoSchedule"
```

复制代码

当然，如果你就是想要一个单节点的 Kubernetes，删除这个 Taint 才是正确的选择：

```
$ kubectl taint nodes --all node-role.kubernetes.io/master-
```

复制代码

如上所示，我们在“node-role.kubernetes.io/master”这个键后面加上了一个短横线“-”，这个格式就意味着移除所有以“node-role.kubernetes.io/master”为键的 Taint。

到了这一步，一个基本完整的 Kubernetes 集群就部署完毕了。是不是很简单呢?

有了 kubeadm 这样的原生管理工具，Kubernetes 的部署已经被大大简化。更重要的是，像证书、授权、各个组件的配置等部署中最麻烦的操作，kubeadm 都已经帮你完成了。

接下来，我们再在这个 Kubernetes 集群上安装一些其他的辅助插件，比如 Dashboard 和存储插件。

部署 Dashboard 可视化插件

在 Kubernetes 社区中，有一个很受欢迎的 Dashboard 项目，它可以给用户提供一个可视化的 Web 界面来查看当前集群的各种信息。毫不意外，它的部署也相当简单：

```
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/ku
dashboard.yaml
```

复制代码

部署完成之后，我们就可以查看 Dashboard 对应的 Pod 的状态了：

```
$ kubectl get pods -n kube-system

kubernetes-dashboard-6948bdb78-f67xk 1/1 Running 0 1m
```

复制代码

需要注意的是，由于 Dashboard 是一个 Web Server，很多人经常会在自己的公有云上无意地暴露 Dashboard 的端口，从而造成安全隐患。所以，1.7 版本之后的 Dashboard 项目部署完成后，默认只能通过 Proxy 的方式在本地访问。具体的操作，你可以查看 Dashboard 项目的官方文档。

而如果你想从集群外访问这个 Dashboard 的话，就需要用到 Ingress，我会在后面的文章中专门介绍这部分内容。

部署容器存储插件

接下来，让我们完成这个 Kubernetes 集群的最后一块拼图：容器持久化存储。

我在前面介绍容器原理时已经提到过，很多时候我们需要用数据卷（Volume）把外面宿主机上的目录或者文件挂载进容器的 Mount Namespace 中，从而达到容器和宿主机共享这些目录或者文件的目的。容器里的应用，也就可以在这些数据卷中新建和写入文件。

可是，如果你在某一台机器上启动的一个容器，显然无法看到其他机器上的容器在它们的数据卷里写入的文件。**这是容器最典型的特征之一：无状态。**

而容器的持久化存储，就是用来保存容器存储状态的重要手段：存储插件会在容器里挂载一个基于网络或者其他机制的远程数据卷，使得在容器里创建的文件，实际上是保存在远程存储服务器上，或者以分布式的方式保存在多个节点上，而与当前宿主机没有任何绑定关系。这样，无论你在其他哪个宿主机上启动新的容器，都可以请求挂载指定的持久化存储卷，从而访问到数据卷里保存的内容。**这就是“持久化”的含义。**

由于 Kubernetes 本身的松耦合设计，绝大多数存储项目，比如 Ceph、GlusterFS、NFS 等，都可以为 Kubernetes 提供持久化存储能力。在这次的部署实战中，我会选择部署一个很重要的 Kubernetes 存储插件项目：Rook。

Rook 项目是一个基于 Ceph 的 Kubernetes 存储插件（它后期也在加入对更多存储实现的支持）。不过，不同于对 Ceph 的简单封装，Rook 在自己的实现中加入

了水平扩展、迁移、灾难备份、监控等大量的企业级功能，使得这个项目变成了一个完整的、生产级别可用的容器存储插件。

得益于容器化技术，用两条指令，Rook 就可以把复杂的 Ceph 存储后端部署起来：

```
$ kubectl apply -f https://raw.githubusercontent.com/rook/rook/master/cluster/examples/kubernetes/ceph/opera
$ kubectl apply -f https://raw.githubusercontent.com/rook/rook/master/cluster/examples/kubernetes/ceph/cluste
```

复制代码

在部署完成后，你就可以看到 Rook 项目会将自己的 Pod 放置在由它自己管理的两个 Namespace 当中：

```
$ kubectl get pods -n rook-ceph-system
NAME READY STATUS RESTARTS AGE
rook-ceph-agent-7cv62 1/1 Running 0 15s
rook-ceph-operator-78d498c68c-7fj72 1/1 Running 0 44s
rook-discover-2ctcv 1/1 Running 0 15s
$ kubectl get pods -n rook-ceph
NAME READY STATUS RESTARTS AGE
rook-ceph-mon0-kxnzh 1/1 Running 0 13s
rook-ceph-mon1-7dn2t 1/1 Running 0 2s
```

复制代码

这样，一个基于 Rook 持久化存储集群就以容器的方式运行起来了，而接下来在 Kubernetes 项目上创建的所有 Pod 就能够通过 Persistent Volume（PV）和 Persistent Volume Claim（PVC）的方式，在容器里挂载由 Ceph 提供的数据卷了。

而 Rook 项目，则会负责这些数据卷的生命周期管理、灾难备份等运维工作。关于这些容器持久化存储的知识，我会在后续章节中专门讲解。

这时候，你可能会有个疑问：为什么我要选择 Rook 项目呢？

其实，是因为这个项目很有前途。

如果你去研究一下 Rook 项目的实现，就会发现它巧妙地依赖了 Kubernetes 提供的编排能力，合理的使用了很多诸如 Operator、CRD 等重要的扩展特性（这些特性我都会在后面的文章中逐一讲解到）。这使得 Rook 项目，成为了目前社区中基于 Kubernetes API 构建的最完善也最成熟的容器存储插件。我相信，这样的发展路线，很快就会得到整个社区的推崇。

> *备注：其实，在很多时候，大家说的所谓“云原生”，就是“Kubernetes 原生”的意思。而像 Rook、Istio 这样的项目，正是贯彻这个思路的典范。在我们后面讲解了声明式 API 之后，相信你对这些项目的设计思想会有更深刻的体会。*

总结

在本篇文章中，我们完全从 0 开始，在 Bare-metal 环境下使用 kubeadm 工具部署了一个完整的 Kubernetes 集群：这个集群有一个 Master 节点和多个 Worker 节点；使用 Weave 作为容器网络插件；使用 Rook 作为容器持久化存储插件；使用 Dashboard 插件提供了可视化的 Web 界面。

这个集群，也将会是我进行后续讲解所依赖的集群环境，并且在后面的讲解中，我还会给它安装更多的插件，添加更多的新能力。

另外，这个集群的部署过程并不像传说中那么繁琐，这主要得益于：

1. kubeadm 项目大大简化了部署 Kubernetes 的准备工作，尤其是配置文件、证书、二进制文件的准备和制作，以及集群版本管理等操作，都被 kubeadm 接管了。
2. Kubernetes 本身“一切皆容器”的设计思想，加上良好的可扩展机制，使得插件的部署非常简便。

上述思想，也是开发和使用 Kubernetes 的重要指导思想，即：基于 Kubernetes 开展工作时，你一定要优先考虑这两个问题：

1. 我的工作是不是可以容器化？
2. 我的工作是不是可以借助 Kubernetes API 和可扩展机制来完成？

而一旦这项工作能够基于 Kubernetes 实现容器化，就很有可能像上面的部署过程一样，大幅简化原本复杂的运维工作。对于时间宝贵的技术人员来说，这个变化的重要性是不言而喻的。

思考题

1. 你是否使用其他工具部署过 Kubernetes 项目？经历如何？
2. 你是否知道 Kubernetes 项目当前（v1.11）能够有效管理的集群规模是多少个节点？你在生产环境中希望部署或者正在部署的集群规模又是多少个节点呢？