

# 53 | 容器化实战：怎样搭建K8s爬虫集群？

2023-02-11 郑建勋 来自北京

《Go进阶·分布式爬虫实战》

课程介绍 >



讲述：郑建勋

时长 12:08 大小 11.09M



你好，我是郑建勋。

上节课，我们介绍了 Kubernetes 架构和相关的原理。这节课让我们更进一步，将爬虫项目相关的微服务部署到 Kubernetes 中。

## 安装 Kubernetes 集群

首先，我们需要准备好 Kubernetes 的集群。部署 Kubernetes 集群的方式有很多种，典型的方式有下面几种：

领资料

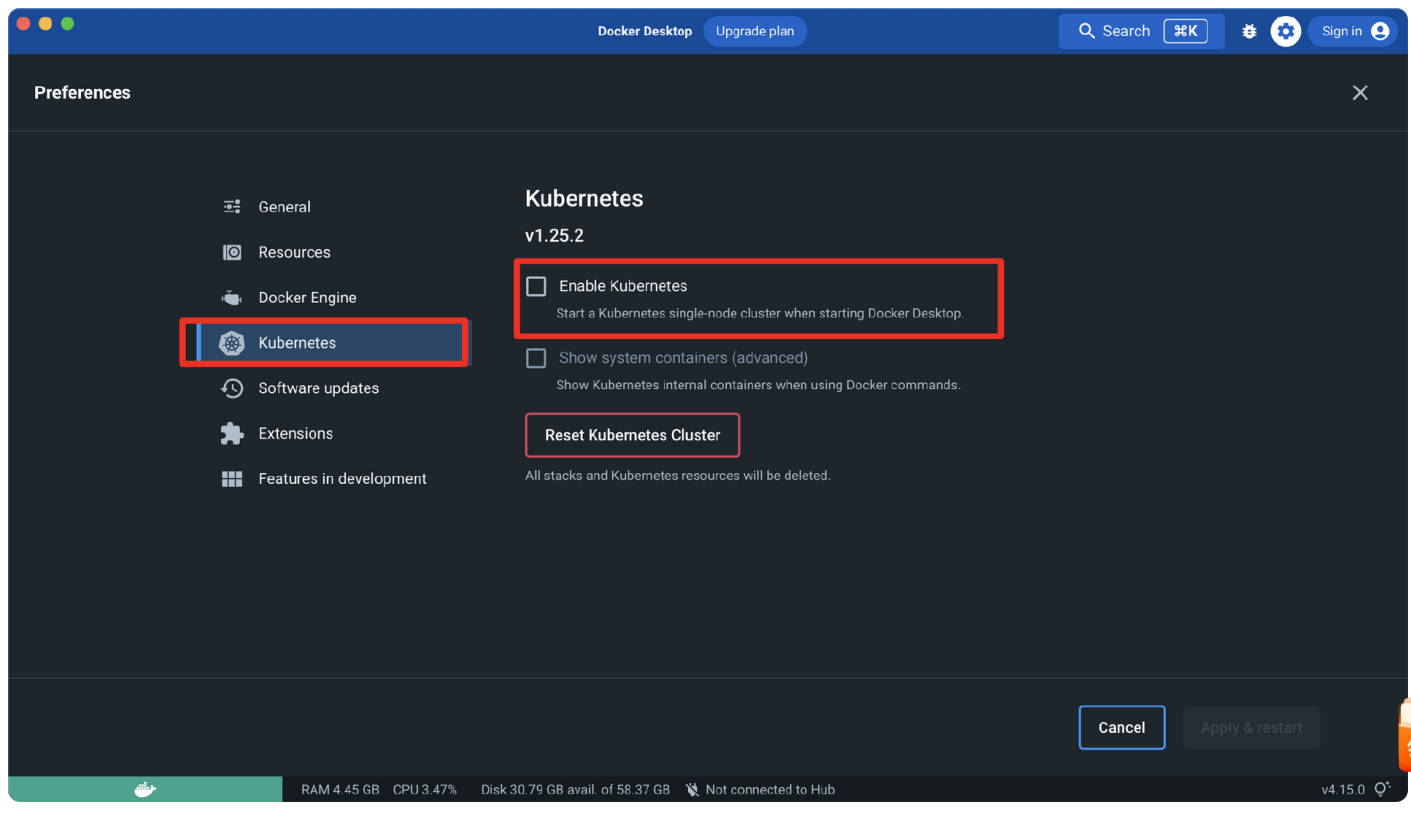


- Play with Kubernetes (PWK)
- Docker Desktop
- 云厂商的 k8s 服务，例如 Google Kubernetes Engine (GKE)

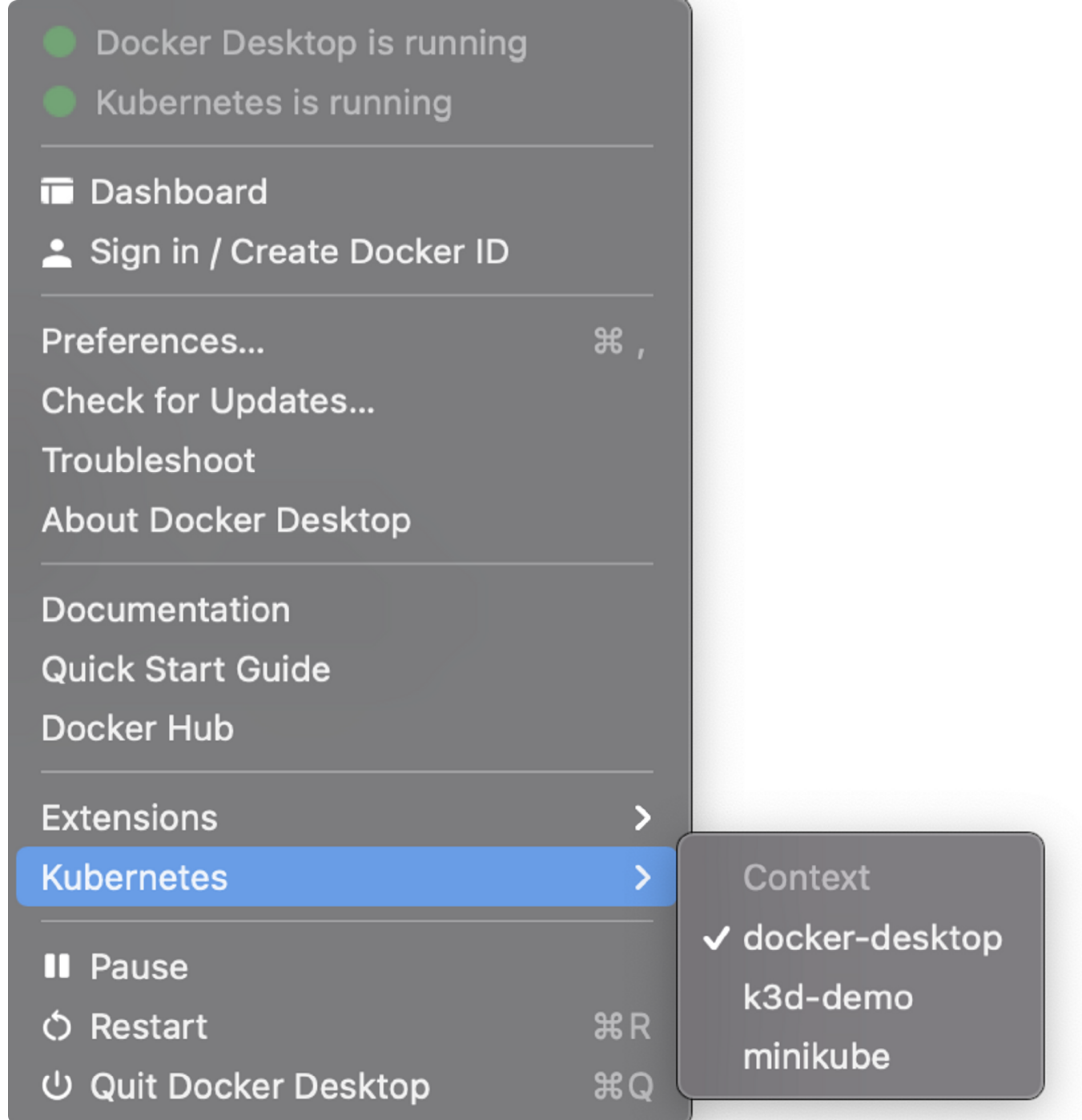
- kops
- kubeadm
- k3s
- k3d

其中，🔗 **PWK** 是试验性质的免费的 Kubernetes 集群，只要有 Docker 或者 Github 账号就可以在浏览器上一键生成 Kubernetes 集群。但是它有诸多限制，例如一次只能使用 4 个小时，并且有扩展性和性能等问题。所以 PWK 一般只用于教学或者试验。

之前，我们在 Windows 和 Mac 中用 Docker Desktop 安装包来安装了 Docker，其实利用最新的 Docker Desktop，我们还可以在本地生成 Kubernetes 集群。使用 Docker Desktop 生成 Kubernetes 集群非常简单，我们只需要点击 Docker 的鲸鱼图标，并且在 Preferences 中勾选 Enable Kubernetes，然后点击下方的 Apply & restart 就可以创建我们的 Kubernetes 集群了。



最后，Docker Desktop 还提供了切换 Kubernetes Context 的能力，我们点击 docker-desktop，这样我们通过 kubectl 发送的命令就会传到 Docker Desktop 构建的 Kubernetes 集群中。



上面的这个界面操作和下面的命令行操作在功能上是相同的。

```
1 » kubectl config use-context docker-desktop
2 Switched to context "docker-desktop".
```

复制代码

领资料

接着，我们执行 `kubectl get nodes` 可以看到当前集群为单节点的集群，版本为 `v1.25.2`。

```
1 » kubectl get nodes
2 NAME                                STATUS    ROLES    AGE    VERSION
```

复制代码

Docker Desktop 生成的 Kubernetes 集群可以用于开发测试，但是它不能模拟多节点的 Kubernetes 集群。

在一些生产环境中，我们可能需要手动部署多节点的 Kubernetes 集群。例如我们要以 Kubernetes 为基座为某企业部署一套人脸识别系统，这时我们可以使用 [🔗 kubectl](#) 工具来安装 Kubernetes 集群。

kubeadm 是 Kubernetes 1.4 版本引入的命令行工具，它致力于简化集群的安装过程。如果要更精细地调整 Kubernetes 各组件服务的参数和安全设置，还可以用 Kubernetes 二进制文件的方式进行部署。kubeadm 和二进制文件的安装方式你可以查看官方文档和《kubernetes 权威指南》。

在另一些生产环境中，例如在私有云的场景下，如果为了应对高峰期而购入机器，容易导致机器闲置，带来资源浪费，这时我们可以借助云厂商的 Kubernetes 服务（Google GKE，Microsoft AKS，Amazon EKS，腾讯云，阿里云）搭建 Kubernetes 集群。以 [🔗 GKE](#) 为例，GKE 是运行在谷歌云平台上的 Kubernetes 托管服务，它可以为我们快速部署和管理生产级的 Kubernetes 集群。要注意的是，部署在云厂商的 Kubernetes 集群一般都是需要付费的。

k3s 是轻量级的 Kubernetes 集群，它通过删除 Kubernetes 中不必要的第三方存储驱动，删除与云厂商交互的代码，将 Kubernetes 组件合并到了不到 100 MB 的二进制文件中去运行，这就减少了二进制文件与内存占用的大小。这个方案适用于物联网等对资源吃紧的环境，也可以用于 CI 及开发环境。

而 k3d 是一个社区驱动的项目，它对 k3s 进行了封装，从而可以在 Docker 中创建单节点或多节点的 Kubernetes 集群。使用 k3d，我们用一行指令就可以创建 Kubernetes 集群。这节课，我们就用 k3d 搭建多节点、轻量级的 Kubernetes 集群，实现开发与测试的目的。

## 安装 k3d

k3d 的安装方式比较简单，可以执行如下的脚本完成。



📄 复制代码

```
1 curl -s https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | bash
```

如果想安装 **k3d** 的指定版本，可以使用下面的指令。

 复制代码

```
1 curl -s https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | TAG=v5.4
```

在 **Mac** 中，还可以使用 **brew** 进行安装，如下所示。其他的安装方式你可以查看 [🔗 官方文档](#)。

 复制代码

```
1 brew install k3d
```

安装完成后，执行 **k3d version** 可以查看 **k3d** 的版本信息和 **k3d** 依赖的 **k3s** 的版本。

 复制代码

```
1 » k3d version
2 k3d version v5.4.6
3 k3s version v1.24.4-k3s1 (default)
```

**k3d** 的使用方法我推荐你阅读它的 [🔗 官方文档](#) 和 **k3d** 维护的一个交互式的 [🔗 demo 项目](#)。这个 **demo** 项目可以完成从集群的创建到销毁，从 **Pod** 的创建到销毁的完整链路。你可以下载 **k3d-demo** 项目的代码，然后 **make demo**，跟着它的提示一步步去学习。

 复制代码

```
1 git clone https://github.com/k3d-io/k3d-demo
2 make demo
```

下面让我们来看看如何借助 **k3d** 搭建 **Kubernetes** 集群。

首先执行 **k3d cluster create** 命令，创建 **Kubernetes** 集群。

 复制代码

```
1 » k3d cluster create demo --api-port 6550 --servers 1 --c 3 --port 8080:80@load
2 INFO[0000] portmapping '8080:80' targets the loadbalancer: defaulting to [serve
```

领资料



```

3 INFO[0000] Prep: Network
4 INFO[0000] Created network 'k3d-demo'
5 INFO[0000] Created image volume k3d-demo-images
6 INFO[0000] Starting new tools node...
7 INFO[0000] Starting Node 'k3d-demo-tools'
8 INFO[0007] Creating node 'k3d-demo-server-0'
9 INFO[0007] Creating node 'k3d-demo-agent-0'
10 INFO[0007] Creating node 'k3d-demo-agent-1'
11 INFO[0007] Creating node 'k3d-demo-agent-2'
12 INFO[0007] Creating LoadBalancer 'k3d-demo-serverlb'
13 INFO[0007] Using the k3d-tools node to gather environment information
14 INFO[0007] Starting new tools node...
15 INFO[0007] Starting Node 'k3d-demo-tools'
16 INFO[0009] Starting cluster 'demo'
17 INFO[0009] Starting servers...
18 INFO[0009] Starting Node 'k3d-demo-server-0'
19 INFO[0013] Starting agents...
20 INFO[0014] Starting Node 'k3d-demo-agent-2'
21 INFO[0014] Starting Node 'k3d-demo-agent-1'
22 INFO[0014] Starting Node 'k3d-demo-agent-0'
23 INFO[0019] Starting helpers...
24 INFO[0019] Starting Node 'k3d-demo-serverlb'
25 INFO[0026] Injecting records for hostAliases (incl. host.k3d.internal) and for
26 INFO[0028] Cluster 'demo' created successfully!

```

其中，运行参数 `api-port` 用于指定 Kubernetes API server 对外暴露的端口。`servers` 用于指定 master node 的数量。`agents` 用于指定 worker node 的数量。`port` 用于指定宿主机端口到 Kubernetes 集群的映射。后面我们会看到，当我们访问宿主机 8080 端口时，实际上会被转发到集群的 80 端口。`wait` 参数表示等待 k3s 集群准备就绪后指令才会返回。

创建好 Kubernetes 集群后，执行 `k3d cluster list` 可以看到当前创建好的 demo 集群信息。

```

1 » k3d cluster list
2 NAME      SERVERS  AGENTS  LOADBALANCER
3 demo      1/1      3/3     true

```

复制代码

领资料

要让 `kubectl` 客户端能够访问到我们刚创建好的 Kubernetes 集群，需要使用下面的指令，把当前集群的配置信息合并到 `kubeconfig` 文件中，然后切换 Kubernetes Context，使 `kubectl` 能够访问到新建的集群。

复制代码

```
1 k3d kubeconfig merge demo --kubeconfig-merge-default --kubeconfig-switch-context
```

接下来，输入 `kubectl get nodes`，可以看到当前集群的 Master Node 与 Worker Node。

 复制代码

```
1 » kubectl get nodes                                     jackson@localhost
2 NAME                                STATUS    ROLES    AGE    VERSION
3 k3d-demo-server-0                  Ready     control-plane,master  2m36s  v1.24.4+k3s1
4 k3d-demo-agent-1                   Ready     <none>    2m31s  v1.24.4+k3s1
5 k3d-demo-agent-0                   Ready     <none>    2m31s  v1.24.4+k3s1
6 k3d-demo-agent-2                   Ready     <none>    2m31s  v1.24.4+k3s1
```

## 部署 Worker Deployment

创建好集群之后，我们要想办法将当前的爬虫项目部署到 Kubernetes 集群中。首先让我们书写一个 `crawl-worker.yaml` 文件，用它来创建 Deployment 资源，管理我们的容器 `crawl-worker`。描述文件如下。

 复制代码

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: crawler-deployment
5   labels:
6     app: crawl-worker
7 spec:
8   replicas: 3
9   selector:
10    matchLabels:
11      app: crawl-worker
12   template:
13     metadata:
14       labels:
15         app: crawl-worker
16     spec:
17       containers:
18         - name: crawl-worker
19           image: crawler:local
20           command:
21             - sh
22             - -c
23             - './crawler worker --podip=${MY_POD_IP}'
24           ports:
25             - containerPort: 8080
26           env:
```

领资料





```
27         - name: MY_POD_IP
28           valueFrom:
29             fieldRef:
30               fieldPath: status.podIP
```

下面我们来挨个解释一下文件中的信息。

- **apiVersion**: 定义创建对象时使用的 Kubernetes API 的版本。**apiVersion** 的一般形式为 `<GROUP_NAME>/<VERSION>`。在这里，Deployment 的 API group 为 `app`，版本为 `v1`。而 Pod 对象由于位于特殊的核心组，可以省略掉 `<GROUP_NAME>`。**apiVersion** 的设计有助于 Kubernetes 对不同的资源进行单独的管理，也有助于开发者创建 Kubernetes 中实验性质的资源进行测试。
- **kind** 表示当前资源的类型，在这里我们定义的是 **Deployment** 对象。之前我们提到过，Deployment 在 Pod 之上增加了自动扩容、自动修复和滚动更新等功能。
- **metadata.name**: 定义 deployment 的名字。
- **metadata.labels**: 给 Deployment 的标签。
- **spec**: 代表对象的期望状态。
- **spec.replicas**: 代表创建和管理的 Pod 的数量。
- **spec.selector**: 定义了 Deployment Controller 要管理哪些 Pod。这里定义的通常是标签匹配的 Pod，满足该标签的 Pod 会被纳入到 Deployment Controller 中去管理。
- **spec.template.metadata**: 定义了 Pod 的属性，在上例中，定义了标签属性 `crawl-worker`。
- **spec.template.spec.containers**: 定义了 Pod 中的容器属性。
- **spec.template.spec.containers.name**: 定义了容器的名字。
- **spec.template.spec.containers.image**: 定义了容器的镜像。
- **spec.template.spec.containers.command**: 定义了容器的启动命令。注意，启动参数中的 `${MY_POD_IP}` 是从环境变量中获取的 `MY_POD_IP` 对应的值。
- **spec.template.spec.container.ports**: 描述服务暴露的端口信息，方便开发者更好地理解服务，没有实际的作用。
- **spec.template.spec.container.env**: 定义容器的环境变量。在这里，我们定义了一个环境变量 `MY_POD_IP`，并且传递了一个特殊的值，即 Pod 的 IP。并将该环境变量的值传递到了运行参数当中。





这里我将 Pod 的 IP 传入到程序中有一个妙处。我们之前在程序运行时手动传入了 Worker 的 ID，这在开发环境中是没有问题的。但是在生产环境中，我们希望 Worker 能够动态扩容，这时就不能手动地指定 ID 了。我们需要让程序在启动后自动生成一个 ID，并且这个 ID 在分布式节点中是唯一的。

一些同学可能会想到把时间当作唯一的 ID，例如使用 `time.Now().UnixNano()` 来获取 Unix 时间戳。但是，程序获取到的时间仍然可能是重复的，虽然概率很小。另一些同学可能会想到利用 MySQL 的自增主键或者借助 etcd 等分布式组件来得到分布式 ID。这当然是一种解决办法，不过却依赖了额外的外部服务。在这里，我选择了一种更为巧妙的方法，即借助 Kubernetes 中 Pod 的 IP 不重复的特性，将 Pod 的 IP 传递到程序中，借助唯一的 Pod IP 生成唯一的 ID。代码如下所示。

 复制代码

```
1 WorkerCmd.Flags().StringVar(  
2     &podIP, "podip", "", "set worker id")  
3  
4 WorkerCmd.Flags().StringVar(  
5     &workerID, "id", "", "set worker id")  
6  
7 var podIP string  
8  
9 if workerID == "" {  
10     if podIP != "" {  
11         ip := generator.GetIDbyIP(podIP)  
12         workerID = strconv.Itoa(int(ip))  
13     } else {  
14         workerID = fmt.Sprintf("%d", time.Now().UnixNano())  
15     }  
16 }  
17  
18 // generator/generator.go  
19 func GetIDbyIP(ip string) uint32 {  
20     var id uint32  
21     binary.Read(bytes.NewBuffer(net.ParseIP(ip).To4()), binary.BigEndian, &id)  
22     return id  
23 }
```

领资料

现在 workerID 的默认值为空，如果没有传递 id flag，也没有传递 podip flag，这一般是线下开发场景，我们直接使用 Unix 时间戳来生成 ID。如果没有传递 id flag，但传递了 podip flag，我们则要利用 Pod IP 的唯一性生成 ID。

准备好程序代码之后，让我们生成 Worker 的镜像，并打上镜像 tag: crawler:local。要注意的是，这里我们并没有和之前一样将镜像变为 crawler:latest。这是因为对于 crawler:latest 的镜像，Kubernetes 会在 DockerHub 中默认 [拉取最新的镜像](#)，这会导致镜像拉取失败，这里我们希望 Kubernetes 使用本地缓存的镜像。

 复制代码

```
1 docker image build -t crawler:local .
```

接着，将镜像导入到 k3d 集群中。

 复制代码

```
1 k3d image import crawler:local -c demo
```

准备好镜像之后，我们就可以创建 Kubernetes 中的 Deployment 资源，用它管理我们的 Worker 节点了。具体步骤是，执行 kubectl apply，告诉 Kubernetes 我们希望创建的 Deployment 资源的信息。

 复制代码

```
1 kubectl apply -f crawl-worker.yaml
```

接下来，输入 kubectl get po，查看 default namespace 中 Pod 的信息。可以看到，Kubernetes 已经为我们创建了 3 个 Pod。

 复制代码

```
1 » kubectl get po
2 NAME                                READY   STATUS    RESTARTS   AGE
3 crawler-deployment-6744cc644b-2mm9r 1/1     Running   0           88s
4 crawler-deployment-6744cc644b-bgjqc 1/1     Running   0           88s
5 crawler-deployment-6744cc644b-84t9g 1/1     Running   0           88s
```

领资料

使用 kubectl logs 打印出某一个 Pod 的日志，可以看到 Worker 节点已经正常地运行了。

还要注意的是，当前 Worker 节点的启动依赖于 MySQL 和 etcd 这两个组件。和之前一样，我们是先将这两个组件在宿主机中 Docker 启动的，后续这两个组件可以部署到 Kubernetes 集

群中。

复制代码

```
1 » kubectl logs -f crawler-deployment-6744cc644b-2mm9r
2 {"level":"INFO","ts":"2022-12-24T07:50:09.301Z","caller":"worker/worker.go:101"}
3 {"level":"INFO","ts":"2022-12-24T07:50:09.301Z","caller":"worker/worker.go:109"}
4 {"level":"DEBUG","ts":"2022-12-24T07:50:09.311Z","caller":"worker/worker.go:152"}
5 {"level":"DEBUG","ts":"2022-12-24T07:50:09.313Z","caller":"worker/worker.go:223"}
6 2022-12-24 07:50:09 file=worker/worker.go:196 level=info Starting [service] gc
7 2022-12-24 07:50:09 file=v4@v4.9.0/service.go:96 level=info Server [grpc] List
8 2022-12-24 07:50:09 file=grpc@v1.2.0/grpc.go:913 level=info Registry [etcd] Re
```

现在让我们来尝试从外部访问一下 Worker 节点。

由于网络存在隔离性，当前要想从外部访问 Worker 节点还没有那么容易。但是我们之前讲过，Pod 之间是可以通过 IP 相互连接的，所以我们打算通过一个 Pod 容器访问 Worker 节点。由于当前我们生成的 crawler 容器内没有内置网络请求工具，所以在这里我们用 kubectl run 启动一个带有 curl 工具的镜像 curlimages/curl，并命名为 mycurlpod。

复制代码

```
1 kubectl run mycurlpod --image=curlimages/curl -i --tty -- sh
```

如下，我们仍然使用 kubectl get pod 查看当前 Worker Pod 的 IP 地址，-o wide 可以帮助我们得到更详细的 Pod 信息。

复制代码

```
1 » kubectl get pod -o wide
2 NAME                                READY   STATUS    RESTARTS   AGE   IP
3 crawler-deployment-6744cc644b-2mm9r 1/1     Running   0           40m   10.42.
4 crawler-deployment-6744cc644b-bgjqc 1/1     Running   0           40m   10.42.
5 crawler-deployment-6744cc644b-84t9g 1/1     Running   0           40m   10.42.
6 mycurlpod                            1/1     Running   0           26m   10.42.
```

领资料

接着进入到 mycurlpod 中，利用 Worker 节点的 Pod IP 地址进行访问，成功返回预期数据。

复制代码

```
1
2 » curl -H "content-type: application/json" -d '{"name": "john"}' http://10.42.4
```

```
3 {"greeting": "Hello "}
```

## 部署 Worker Service

不过到这里还不能放松警惕。我们上节课说过，Pod 的 IP 会随时发生变化，为了有稳定的 IP 来访问我们的 Worker 与 Master，我们要创建一个文件 `crawl-worker-service.yaml`，用它来描述 Service 资源。Service 默认的类型为 ClusterIP，该类型的 Service IP 只能够在集群内部访问。

 复制代码

```
1 kind: Service
2 apiVersion: v1
3 metadata:
4   name: crawl-worker
5   labels:
6     app: crawl-worker
7 spec:
8   selector:
9     app: crawl-worker
10  ports:
11    - port: 8080
12      name: http
13    - port: 9090
14      name: grpc
```

描述 Service 资源与之前描述 Deployment 资源非常类似。

- `kind: Service`: 指的是当前的 Kubernetes 资源类型为 Service。
- `apiVersion: v1`: 代表 `apiVersion` 的版本是 `v1`，由于 Service 是核心类型，因此省略掉了 API GROUP 的命名前缀。
- `metadata.name`: 当前 Service 的名字。
- `metadata.labels`: 当前 Service 的标签。
- `spec.selector`: 选择器，表示当前 Service 管理哪些后台服务，只有标签为 `app: crawl-worker` 的 Pod 才会受到该 Service 的管理，这些 Pod 就是我们的 Worker 节点。
- `spec.ports.port`: 当前 Service 监听的端口号。例如。 `port: 8080` 指的是当前 Service 会监听 8080 端口。默认情况下，当外部访问该 Service 的 8080 端口时，会将请求转发给后端服务相同的端口。

领资料

- `spec.ports.port.name`: 描述了当前 `Service` 端口规则的名字。

接下来使用 `kubectl apply` 创建 `Service` 资源，并使用 `kubectl get service` 得到 `Service` 的 `Cluster-IP` 地址。

 复制代码

```
1 » kubectl apply -f crawl-worker-service.yaml
2 » kubectl get service
3 NAME          TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AG
4 crawl-worker   ClusterIP      10.43.245.115    <none>           8080/TCP,9090/TCP 1
```

同样，在包含 `curl` 工具的 `mycurlpod` 中，访问 `Service` 暴露的 `ClusterIP`，这时请求会负载均衡到后端任意一个 `Worker` 节点中。

 复制代码

```
1 $ curl -H "content-type: application/json" -d '{"name": "john"}' http://10.43.2
```

## 部署 Master Deployment

`Master` 节点的部署和 `Worker` 节点的部署非常类似。如下所示，创建 `crawler-master.yaml` 文件，描述 `Deployment` 资源的信息。这里和 `Worker Deployment` 不同的主要是相关的名字和程序启动的命令。

 复制代码

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: crawler-master-deployment
5   labels:
6     app: crawl-master
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: crawl-master
12   template:
13     metadata:
14       labels:
15         app: crawl-master
16     spec:
17       containers:
```

领资料



```

18     - name: crawl-master
19       image: crawler:local
20       command:
21         - sh
22         - -c
23         - "./crawler master --podip=${MY_POD_IP}"
24       ports:
25         - containerPort: 8081
26       env:
27         - name: MY_POD_IP
28           valueFrom:
29             fieldRef:
30               fieldPath: status.podIP

```

## 部署 Master Service

同样地，为了能够用固定地 IP 访问 Master，我们需要创建 Master Service。新建 `crawl-master-service.yaml` 文件，如下所示，`port` 指的是 Service 监听的端口 80，这是默认的 HTTP 端口，而 `targetPort` 指的是转发到后端服务器的端口，也就是 Master 服务的 HTTP 端口 8081。

 复制代码

```

1 kind: Service
2 apiVersion: v1
3 metadata:
4   name: crawl-master
5   labels:
6     app: crawl-master
7 spec:
8   selector:
9     app: crawl-master
10  type: NodePort
11  ports:
12    - port: 80
13      targetPort: 8081
14      name: http
15    - port: 9091
16      name: grpc

```

领资料

接下来，我们还是利用 `kubectl apply` 创建 Master Service。

 复制代码

```

1 » kubectl apply -f crawl-master-service.yaml

```

现在我们就可以和 Worker 一样，利用 Service 访问 Masrer 暴露的接口了。

## 创建 Ingress

下面让我们更进一步，尝试在宿主机中访问集群的 Master 服务。由于资源具有隔离性，之前我们一直都是在集群内从一个 Pod 中去访问另一个 Pod。现在 we 希望在集群外部使用 HTTP 访问 Master 服务。要实现这个目标，可以使用 Ingress 资源。

具体做法是，创建 ingress.yaml，在 ingress.yaml 文件中书写相关的 HTTP 路由规则，根据不同的域名和 URL 将请求路由到后端不同的 Service 中。

 复制代码

```
1 apiVersion: networking.k8s.io/v1
2 kind: Ingress
3 metadata:
4   name: crawler-ingress
5 spec:
6   rules:
7     - http:
8       paths:
9         - path: /
10           pathType: Prefix
11           backend:
12             service:
13               name: crawl-master
14               port:
15                 number: 80
```

spec.http.paths 中描述了 Ingress 的路由规则，我们对应上面这个例子解读一下。

- spec.http.paths.path 表示 URL 匹配的路径。
- spec.http.paths.pathType 表示 URL 匹配的类型为前缀匹配。
- [spec.http.paths.backend.service.name](#) 表示路由到后端的 Service 的名字。
- spec.http.paths.backend.service.port 表示路由到后端的 Service 的端口。

其实要使用 Ingress 的功能，光是定义 ingress.yaml 中的路由规则是不行的，我们还需要 Ingress Controller 控制器来实现这些路由规则的逻辑。但是和 Kubernetes 中内置的

领资料





Controller 不同，Ingress Controller 是可以灵活选择的，比较有名的 Ingress Controller 包括 Nginx、Kong 等。如果你想使用这些 Ingress Controller，需要单独安装。

好在，k3d 默认为我们内置了  traefik Ingress Controller，这样我们就不需要再额外安装了。只要创建 Ingress 资源，就可以直接访问它了。

 复制代码

```
1 » kubectl apply -f ingress.yaml
```

下一步，我们在宿主机中访问集群。在这里我们访问的是 8080 端口，因为我在创建集群时指定了端口的映射，所以当前 8080 端口的请求会转发到集群的 80 端口中。然后根据 Ingress 指定的规则，请求会被转发到后端的 Master Service 当中。

 复制代码

```
1 » curl -H "content-type: application/json" -d '{"id":"zjx","name": "douban_book
```

此外，Ingress 还可以设置规则，让不同的域名走不同的域名规则，这里就不再赘述了。

 复制代码

```
1 » curl -H "content-type: application/json" -d '{"id":"zjx","name": "douban_book
```

## 创建 ConfigMap

到这一步，我们的配置文件都是打包在镜像中的，如果想要修改程序的启动参数会非常麻烦。因此，我们可以借助 Kubernetes 中的 ConfigMap 资源，将配置挂载到容器当中，这样我们就可以更灵活地修改配置文件，而不必每一次都打包新的镜像了。

具体做法如下。我们创建一个 ConfigMap 资源，把它放到默认的 namespace 中。在 Data 下，对应地输入文件名 config.toml 和文件内容。

 复制代码

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: crawler-config
```

领资料



```

5 namespace: default
6 data:
7   config.toml: |-
8     logLevel = "debug"
9
10    [fetcher]
11    timeout = 3000
12    proxy = ["http://192.168.0.105:8888", "http://192.168.0.105:8888"]
13
14
15    [storage]
16    sqlURL = "root:123456@tcp(192.168.0.105:3326)/crawler?charset=utf8"
17
18    [GRPCServer]
19    HTTPListenAddress = ":8080"
20    GRPCListenAddress = ":9090"
21    ID = "1"
22    RegistryAddress = "192.168.0.105:2379"
23    RegisterTTL = 60
24    RegisterInterval = 15
25    ClientTimeOut    = 10
26    Name = "go.micro.server.worker"
27
28    [MasterServer]
29    RegistryAddress = "192.168.0.105:2379"
30    RegisterTTL = 60
31    RegisterInterval = 15
32    ClientTimeOut    = 10
33    Name = "go.micro.server.master"

```

然后，修改 `crawler-master.yaml` 文件，将 `ConfigMap` 挂载到容器中。

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: crawler-master-deployment
5   labels:
6     app: crawl-master
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: crawl-master
12   template:
13     metadata:
14       labels:
15         app: crawl-master
16     spec:

```

 复制代码

领资料

```

17 containers:
18   - name: crawl-master
19     image: crawler:local
20     command:
21       - sh
22       - -c
23       - "./crawler master --podip=${MY_POD_IP} --config=/app/config/config
24     ports:
25       - containerPort: 8081
26     env:
27       - name: MY_POD_IP
28         valueFrom:
29           fieldRef:
30             fieldPath: status.podIP
31     volumeMounts:
32       - name: crawler-config
33         mountPath: /app/config/
34 volumes:
35   - name: crawler-config
36     configMap:
37       name: crawler-config

```

在这个例子中，`spec.template.spec.volumes` 创建了一个存储卷 `crawler-config`，它的内容来自于名为 `crawler-config` 的 `ConfigMap`。接着，`spec.template.spec.containers.volumeMounts` 表示将该存储卷挂载到容器的 `/app/config` 目录下，这样程序就能够顺利使用 `ConfigMap` 中的配置文件了。

## 总结

这节课，我们学习了如何安装 **Kubernetes** 集群，然后使用 **k3d** 工具搭建起了轻量级的 **Kubernetes** 集群，从而在开发环境中模拟了多节点的 **Kubernetes** 集群。

我们还书写了核心的 **YAML** 描述文件，创建了 **Kubernetes** 中的资源，包括 **Deployment**、**Pod**、**Service**、**Ingress**、**ConfigMap** 等，将我们的爬虫项目部署到了 **Kubernetes** 中。

如果你是第一次接触这些 **Kubernetes** 描述文件，可能会觉得非常繁琐，但是熟悉之后就会变得简单了。创建了这些资源之后，剩下服务的扩容、维护就可以交给强大的 **Kubernetes** 来管理了。



（这节课中的 **YAML** 文件在 [🔗 最新代码分支](#) 的 **Kubernetes** 文件夹中。）


## 课后题

学完这节课，请你思考下面这个问题。

我们这节课书写的 **YAML** 文件中的资源对象，都是 **Kubernetes** 中定义好的资源类型。那么我们可以可不可以创建一个自定义的类型，自己去控制它的生命周期呢？

欢迎你在留言区留下自己思考的结果，也可以把这节课分享给对这个话题感兴趣的同事和朋友，我们下节课再见！

分享给需要的人，Ta购买本课程，你将得 **20** 元

 生成海报并分享

 赞 1     提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇    52 | 容器海洋中的舵手：Kubernetes工作机制

下一篇    特别放送 | Go泛型：用法、原理与最佳实践

## 学习推荐

全新汇总

# Go 面试必考 300+ 题

面试真题 | 进阶实战 | 专题视频 | 学习路线

限时免费 

仅限 99 名

领资料

## 精选留言 (2)

写留言



北庭

2023-03-14 来自江苏

老师，在使用kubectl get node命令后出现了这样的错误：couldn't get current server API group list: Get "https://host.docker.internal:6550/api?timeout=32s": dial tcp 10.0.0.35:6550: connect: A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.

这是为什么呢



Geek\_7873ee

2023-02-15 来自四川

master 只部署了一个节点，那代码中对master做的高可用，就不是看不出来效果了嘛



领资料

