

Kubernetes 工作负载

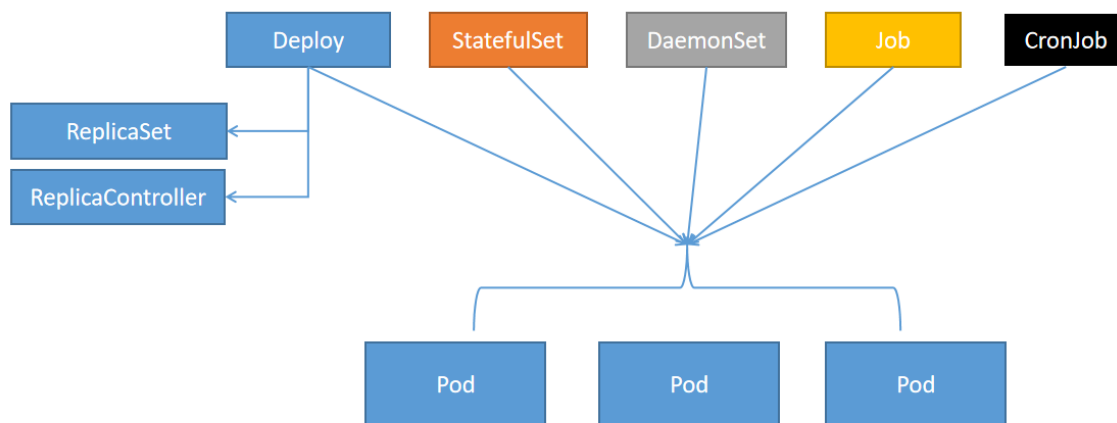
总：Workloads

```
1 #获取控制台访问令牌
2 kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-dashboard
  get secret | grep admin-user | awk '{print $1}')
```

什么是工作负载 (Workloads)

- 工作负载是运行在 Kubernetes 上的一个应用程序。
- 一个应用很复杂，可能由单个组件或者多个组件共同完成。无论如何我们可以用一组Pod来表示一个应用，也就是一个工作负载
- Pod又是一组容器 (Containers)
- 所以关系又像是这样
 - 工作负载 (Workloads) 控制一组Pod
 - Pod控制一组容器 (Containers)
 - 比如Deploy (工作负载) 3个副本的nginx (3个Pod)，每个nginx里面是真正的nginx容器 (container)

工作负载



工作负载能让Pod能拥有自恢复能力。

会写Pod。研究不同的工作负载怎么控制Pod的行为

一、Pod

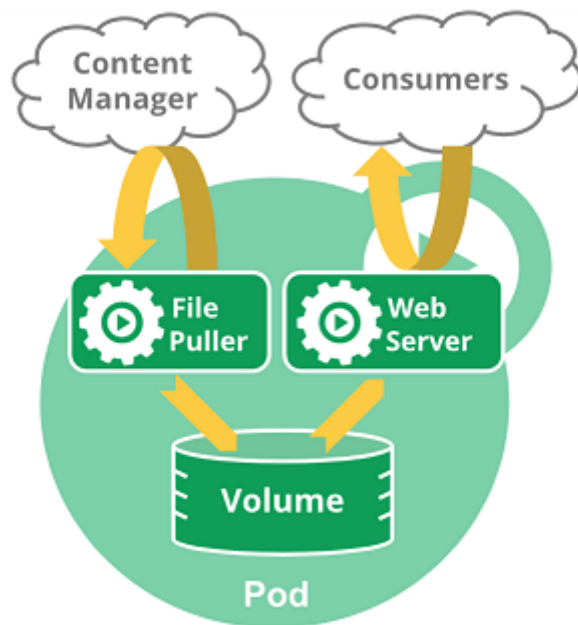
1、什么是Pod

- *Pod* 是一组（一个或多个） **容器（docker容器）** 的集合（就像在豌豆荚中）；这些容器共享存储、网络、以及怎样运行这些容器的声明。

-



- 我们一般不直接创建Pod，而是创建一些工作负载由他们来创建Pod
- Pod的形式
 - Pod对容器有自恢复能力（Pod自动重启失败的容器）
 - Pod自己不能恢复自己，Pod被删除就真的没了（100, MySQL、Redis、Order）还是希望k8s集群能自己在其他地方再启动这个Pod
 - 单容器Pod
 - 多容器协同Pod。我们可以把另外的容器称为 **SideCar（为应用赋能）**
 - Pod天生地为其成员容器提供了两种共享资源：**网络**和**存储**。
- 一个Pod由一个**Pause容器**设置好整个Pod里面所有容器的网络、名称空间等信息
- systemctl status可以观测到。Pod和容器进程关系
 - kubelet启动一个Pod，准备两个容器，一个是Pod声明的应用容器（nginx），另外一个Pause。Pause给当前应用容器设置好网络空间各种的。
 -



编写yaml测试：多容器协同

2、Pod使用

- 可以编写deploy等各种工作负载的yaml文件，最终创建出pod，也可以直接创建
- Pod的模板如下

```
1      # 这里是 Pod 模版
2      apiVersion: v1
3      kind: Pod
4      metadata:
5        name: my-pod
6      spec:
7        containers:
8          - name: hello
9            image: busybox
10           command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
11           restartPolicy: OnFailure
12     # 以上为 Pod 模版
```

3、Pod生命周期



- Pod启动，会先**依次**执行所有初始化容器，有一个失败，则Pod不能启动
- 接下来**启动所有的应用容器**（每一个应用容器都必须能一直运行起来），Pod开始正式工作，一个启动失败就会**尝试重启Pod内的这个容器**，Pod只要是NotReady，Pod就不对外提供服务了

编写yaml测试生命周期

- 应用容器生命周期钩子
- 初始化容器（也可以有钩子）

my-nginx666	1/1	Running	0	2d
pod-life-02	0/2	Init:0/1	0	5s

临时容器：线上排错。

有些容器基础镜像。线上没法排错。使用临时容器进入这个Pod。临时容器共享了Pod的所有。临时容器有Debug的一些命令，拍错完成以后，只要exit退出容器，临时容器自动删除

Java: dump, jre 50mb。jdk 150mb

jre 50mb。: jdk作为临时容器

临时容器需要开启特性门控 `--feature-gates="EphemeralContainers=true"`

在所有组件，api-server、kubelet、scheduler、controller-manager都得配置

1.21.0: 生产环境 .5

使用临时容器的步骤：

1、声明一个临时容器。准备好json文件

```

1  {
2      "apiVersion": "v1",
3      "kind": "EphemeralContainers",
4      "metadata": {
5          "name": "my-nginx666" //指定Pod的名字
6      },
7      "ephemeralContainers": [{
8          "command": [

```

```

9         "sh"
10     ],
11     "image": "busybox",    //jre的需要jdk来调试
12     "imagePullPolicy": "IfNotPresent",
13     "name": "debugger",
14     "stdin": true,
15     "tty": true,
16     "terminationMessagePolicy": "File"
17 }
18 }

```

2、使用临时容器，应用一下即可

```

1 kubectl replace --raw /api/v1/namespaces/default/pods/my-nginx666【pod
   名】/ephemeralcontainers -f ec.json

```

4、静态Pod

在 `/etc/kubernetes/manifests` 位置放的所有Pod.yaml文件，机器启动kubelet自己就把他启动起来。

静态Pod一直守护在他的这个机器上

5、Probe 探针机制（健康检查机制）

- 每个容器三种探针（Probe）
 - **启动探针****（后来才加的）** **一次性成功探针**。只要启动成功了
 - kubelet 使用启动探针，来检测应用是否已经启动。如果启动就可以进行后续的探测检查。慢容器一定指定启动探针。一直在等待启动
 - **启动探针 成功以后就不用了，剩下存活探针和就绪探针持续运行**
 - 存活探针
 - kubelet 使用存活探针，来检测容器是否正常存活。（有些容器可能产生死锁【应用程序在运行，但是无法继续执行后面的步骤】），**如果检测失败就会**重新启动这个容器****
 - `initialDelaySeconds`: 3600（长了导致可能应用一段时间不可用） 5（短了陷入无限启动循环）
 - 就绪探针
 - kubelet 使用就绪探针，来检测容器是否准备**好了可以接收流量**。当一个 Pod 内的所有容器都准备好了，才能把这个 Pod 看作就绪了。用途就是：Service后端负载均衡多个 Pod，如果某个Pod还没就绪，就会从service负载均衡里面剔除
 - 谁利用这些探针探测
 - kubelet会主动按照配置给Pod里面的所有容器发送响应的探测请求

- **Probe** 配置项

- `initialDelaySeconds`：容器启动后要等待多少秒后存活和就绪探测器才被初始化，默认是 0 秒，最小值是 0。这是针对以前没有
- `periodSeconds`：执行探测的时间间隔（单位是秒）。默认是 10 秒。**最小值是 1。**
- `successThreshold`：探测器在失败后，被视为成功的最小连续成功数。**默认值是 1。**
 - 存活和启动探针的这个值必须是 1。最小值是 1。
- `failureThreshold`：当探测失败时，Kubernetes 的重试次数。存活探测情况下的放弃就意味着重新启动容器。就绪探测情况下的放弃 Pod 会被打上未就绪的标签。**默认值是 3。**最小值是 1。
- `timeoutSeconds`：探测的超时后等待多少秒。**默认值是 1 秒。**最小值是 1。

<https://kubernetes.io/zh/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#configure-probes>

```

1      exec、httpGet、tcpSocket 【那种方式探测】
2
3
4
5
6      failureThreshold
7
8
9
10     initialDelaySeconds
11
12     periodSeconds
13
14     successThreshold
15
16
17
18     terminationGracePeriodSeconds
19
20     timeoutSeconds    <integer>
21

```

编写yaml测试探针机制

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: "nginx-start-probe02"
5    namespace: default
6    labels:
7      app: "nginx-start-probe02"
8  spec:
9    volumes:
10     - name: nginx-vol
11       hostPath:
12         path: /app
13     - name: nginx-html
14       hostPath:
15         path: /html
16   containers:

```

```

17   - name: nginx
18     image: "nginx"
19     ports:
20     - containerPort: 80
21     startupProbe:
22       exec:
23         command: ["/bin/sh", "-c", "cat /app/abc"] ## 返回不是0，那就是探测失败
24       # initialDelaySeconds: 20 ## 指定的这个秒以后才执行探测
25       periodSeconds: 5 ## 每隔几秒来运行这个
26       timeoutSeconds: 5 ##探测超时，到了超时时间探测还没返回结果说明失败
27       successThreshold: 1 ## 成功阈值，连续几次才算成功
28       failureThreshold: 3 ## 失败阈值，连续几次失败才算真失败
29     volumeMounts:
30     - name: nginx-vol
31       mountPath: /app
32     - name: nginx-html
33       mountPath: /usr/share/nginx/html
34     livenessProbe: ## nginx容器有没有 /abc.html，就绪探针
35       # httpGet:
36       #   host: 127.0.0.1
37       #   path: /abc.html
38       #   port: 80
39       #   scheme: HTTP
40       # periodSeconds: 5 ## 每隔几秒来运行这个
41       # successThreshold: 1 ## 成功阈值，连续几次才算成功
42       # failureThreshold: 5 ## 失败阈值，连续几次失败才算真失败
43       exec:
44         command: ["/bin/sh", "-c", "cat /usr/share/nginx/html/abc.html"] ## 返回
不是0，那就是探测失败
45       # initialDelaySeconds: 20 ## 指定的这个秒以后才执行探测
46       periodSeconds: 5 ## 每隔几秒来运行这个
47       timeoutSeconds: 5 ##探测超时，到了超时时间探测还没返回结果说明失败
48       successThreshold: 1 ## 成功阈值，连续几次才算成功
49       failureThreshold: 3 ## 失败阈值，连续几次失败才算真失败
50     readinessProbe: ##就绪检测，都是http
51       httpGet:
52         # host: 127.0.0.1 ###不行
53         path: /abc.html ## 给容器发请求
54         port: 80
55         scheme: HTTP ## 返回不是0，那就是探测失败
56       initialDelaySeconds: 2 ## 指定的这个秒以后才执行探测
57       periodSeconds: 5 ## 每隔几秒来运行这个
58       timeoutSeconds: 5 ##探测超时，到了超时时间探测还没返回结果说明失败
59       successThreshold: 3 ## 成功阈值，连续几次才算成功
60       failureThreshold: 5 ## 失败阈值，连续几次失败才算真失败
61
62     # livenessProbe:
63     #   exec: ["/bin/sh", "-c", "sleep 30;abc "] ## 返回不是0，那就是探测失败
64     #   initialDelaySeconds: 20 ## 指定的这个秒以后才执行探测
65     #   periodSeconds: 5 ## 每隔几秒来运行这个
66     #   timeoutSeconds: 5 ##探测超时，到了超时时间探测还没返回结果说明失败
67     #   successThreshold: 5 ## 成功阈值，连续几次才算成功
68     #   failureThreshold: 5 ## 失败阈值，连续几次失败才算真失败

```

微服务。 /health

K8S检查当前应用的状态； connection refuse；

SpringBoot 优雅停机： gracefulShutdown: true

pod.spec.terminationGracePeriodSeconds = 30s 优雅停机； 给一个缓冲时间

健康检查+优雅停机 = 0宕机

start完成以后，liveness和readiness并存。 liveness失败导致重启。 readiness失败导致不给Service负载均衡网络中加，不接受流量。 kubectl exec -it 就进不去。 Kubectl describe 看看咋了。

二、Deployment

1、什么是Deployment

- 一个 *Deployment* 为 **Pods** 和 **ReplicaSets** 提供声明式的更新能力。
- 你负责描述 Deployment 中的 *目标状态*，而 Deployment **控制器（Controller）** 以受控速率更改**实际状态**，使其变为**期望状态**
- 不要管理 Deployment 所拥有的 ReplicaSet
- 我们部署一个应用一般不直接写Pod，而是部署一个Deployment
- Deploy编写规约 <https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#writing-a-deployment-spec>

2、Deployment创建

- 基本格式
 - **.metadata.name** 指定deploy名字
 - **replicas** 指定副本数量
 - **selector** 指定匹配的Pod模板。
 - **template** 声明一个Pod模板

编写一个Deployment的yaml

- 在检查集群中的 Deployment 时，所显示的字段有：
 - **NAME** 列出了集群中 Deployment 的名称。
 - **READY** 显示应用程序的可用的 *副本* 数。显示的模式是“就绪个数/期望个数”。
 - **UP-TO-DATE** 显示为了达到期望状态已经更新的副本数。

- `AVAILABLE` 显示应用可供用户使用的副本数。
- `AGE` 显示应用程序运行的时间。
- ReplicaSet 输出中包含以下字段：
 - `NAME` 列出名字空间中 ReplicaSet 的名称；
 - `DESIRED` 显示应用的期望副本个数，即在创建 Deployment 时所定义的值。此为期望状态；
 - `CURRENT` 显示当前运行状态中的副本个数；
 - `READY` 显示应用中有多少副本可以为用户提供服务；
 - `AGE` 显示应用已经运行的时间长度。
 - 注意：ReplicaSet 的名称始终被格式化为 `[Deployment名称]-[随机字符串]`。其中的随机字符串是使用 pod-template-hash 作为种子随机生成的。

3、Deployment 更新机制

- 仅当 Deployment Pod 模板（即 `.spec.template`）发生改变时，例如模板的标签或容器镜像被更新，才会触发 Deployment 上线。其他更新（如对 Deployment 执行扩缩容的操作）不会触发上线动作。
- 上线动作原理：创建新的rs，准备就绪后，替换旧的rs（此时不会删除，因为 `revisionHistoryLimit` 指定了保留几个版本）
- 常用的kubectl 命令

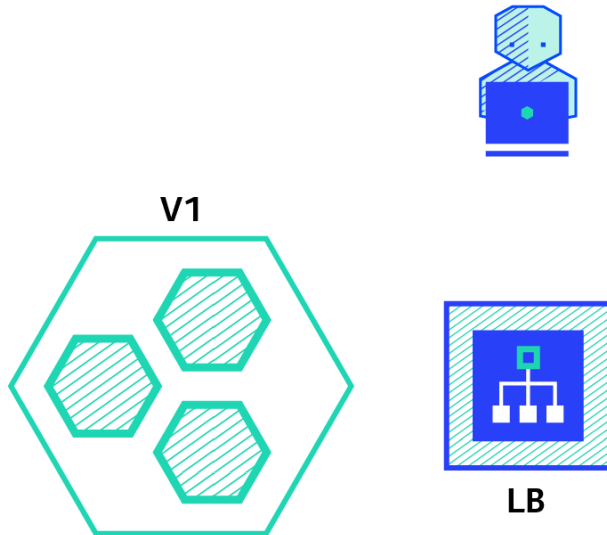
```

1  #####更新#####
2  #kubectl set image deployment资源名 容器名=镜像名
3  kubectl set image deployment.apps/nginx-deployment php-redis=tomcat:8 --record
4  #或者直接修改定义也行
5  kubectl edit deployment.v1.apps/nginx-deployment
6  #查看状态
7  kubectl rollout status deployment.v1.apps/nginx-deployment
8
9  #####查看历史并回滚#####
10 #查看更新历史-看看我们设置的历史总记录数是否生效了
11 kubectl rollout history deployment.v1.apps/nginx-deployment
12 #回滚
13 kubectl rollout undo deployment.v1.apps/nginx-deployment --to-revision=2
14
15 #####累计更新#####
16 #暂停记录版本
17 kubectl rollout pause deployment.v1.apps/nginx-deployment
18 #多次更新操作。
19 ##比如更新了资源限制
20 kubectl set resources deployment.v1.apps/nginx-deployment -c=nginx --
    limits=cpu=200m,memory=512Mi
21 ##比如更新了镜像版本
22 kubectl set image deployment.apps/nginx-deployment php-redis=tomcat:8
23 ##在继续操作多次
24 ##看看历史版本有没有记录变化
25 kubectl rollout history deployment.v1.apps/nginx-deployment
26 #让多次累计生效
27 kubectl rollout resume deployment.v1.apps/nginx-deployment
  
```

1、比例缩放 (Proportional Scaling)

maxSurge (最大增量)：除当前数量外还要添加多少个实例。

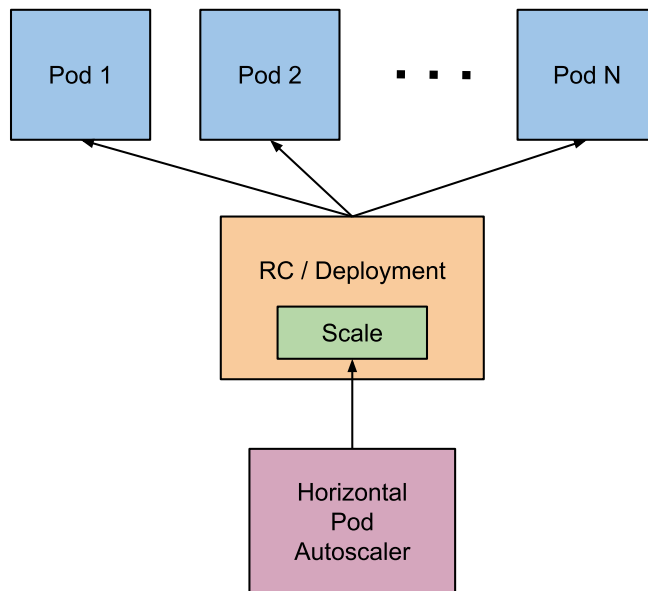
maxUnavailable (最大不可用量)：滚动更新过程中的不可用实例数。



2、HPA (动态扩缩容)

概念：<https://kubernetes.io/zh/docs/tasks/run-application/horizontal-pod-autoscale/#scaling-policies>

实战：<https://kubernetes.io/zh/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/>



- 需要先安装metrics-server

<https://github.com/kubernetes-sigs/metrics-server>

- 配置hpa测试

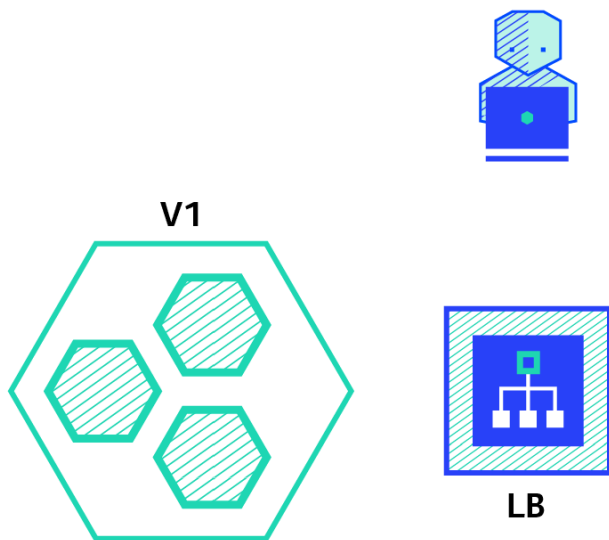
```
1  ##应用的yaml已经做好
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: php-apache
6  spec:
7    ports:
8      - port: 80
9      protocol: TCP
10     targetPort: 80
11    selector:
12      run: php-apache
13  ---
14  apiVersion: apps/v1
15  kind: Deployment
16  metadata:
17    labels:
18      run: php-apache
19    name: php-apache
20  spec:
21    replicas: 1
22    selector:
23      matchLabels:
24        run: php-apache
25    template:
```

```
26     metadata:
27       creationTimestamp: null
28     labels:
29       run: php-apache
30     spec:
31       containers:
32       - image: registry.cn-hangzhou.aliyuncs.com/lfy_k8s_images/php-hpa:latest
33         name: php-apache
34         ports:
35         - containerPort: 80
36       resources:
37         requests:
38           cpu: 200m
39
40     ##hpa配置 hpa.yaml
41     apiVersion: autoscaling/v1
42     kind: HorizontalPodAutoscaler
43     metadata:
44       name: php-apache
45     spec:
46       maxReplicas: 10
47       minReplicas: 1
48       scaleTargetRef:
49         apiVersion: apps/v1
50         kind: Deployment
51         name: php-apache
52       targetCPUUtilizationPercentage: 50
53
54     #3、进行压力测试
55     kubectl run -i --tty load-generator --image=busybox /bin/sh
56
57     #回车然后敲下面的命令
58     while true; do wget -q -O- http://php-apache; done
```

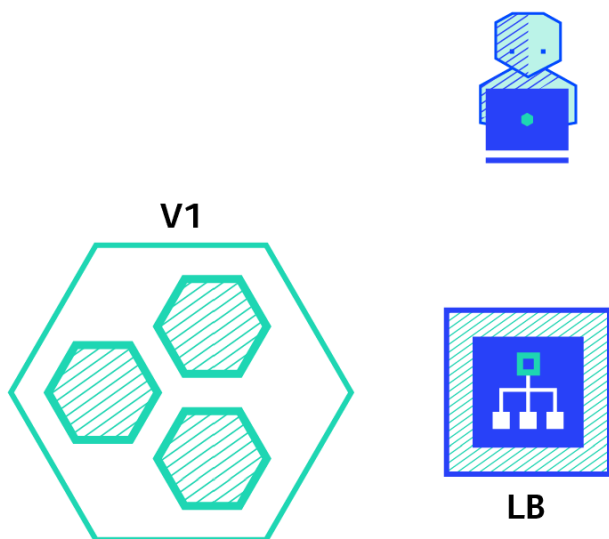
3、Canary (金丝雀部署)

1、蓝绿部署VS金丝雀部署

蓝绿部署



金丝雀部署



2、金丝雀的简单测试

```
1 ##### 使用这个镜像测试registry.cn-hangzhou.aliyuncs.com/lfy_k8s_images/nginx-test
2 ##### 这个镜像docker run 的时候 -e msg=aaaa, 访问这个nginx页面就是看到aaaa
```

步骤原理

- 准备一个Service，负载均衡Pod
- 准备版本v1的deploy，准备版本v2的deploy

4、Deployment状态与排错

<https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#deployment-status>

三、RC、RS

四、DaemonSet

DemonSet 控制器确保所有（或一部分）的节点都运行了一个指定的 Pod 副本。

- 每当向集群中添加一个节点时，指定的 Pod 副本也将添加到该节点上
- 当节点从集群中移除时，Pod 也就被垃圾回收了
- 删除一个 DaemonSet 可以清理所有由其创建的 Pod

DemonSet 的典型使用场景有：

- 在每个节点上运行集群的存储守护进程，例如 glusterd、ceph
- 在每个节点上运行日志收集守护进程，例如 fluentd、logstash
- 在每个节点上运行监控守护进程，例如 **Prometheus Node Exporter**、**Sysdig Agent**、collectd、**Dynatrace OneAgent**、**APCDynamics Agent**、**Datadog agent**、**New Relic agent**、Ganglia gmond、**Instana Agent** 等

```
1   apiVersion: apps/v1
2   kind: DaemonSet
3   metadata:
4     name: logging
5     labels:
6       app: logging
7   spec:
8     selector:
9       matchLabels:
```

```

10     name: logging
11   template:
12     metadata:
13       labels:
14         name: logging
15     spec:
16       containers:
17       - name: logging
18         image: nginx
19         resources:
20           limits:
21             memory: 200Mi
22           requests:
23             cpu: 100m
24             memory: 200Mi
25         tolerations: #设置容忍master的污点
26         - key: node-role.kubernetes.io/master
27           effect: NoSchedule
28 #查看效果
29 kubectl get pod -l name=logging -o wide

```

五、StatefulSet

有状态副本集；Deployment等属于无状态的应用部署（stateless）

- **StatefulSet 使用场景**；对于有如下要求的应用程序，StatefulSet 非常适用：
 - **稳定、唯一的网络标识（dnsname）**
 - StatefulSet通过与其相关的无头服务为每个pod提供DNS解析条目。假如无头服务的DNS条目为：
 "\$(service name).\$(namespace).svc.cluster.local",
 那么pod的解析条目就是 "\$(pod name).\$(service name).\$(namespace).svc.cluster.local",
 每个pod name也是唯一的。
 - **稳定的、持久的存储；【每个Pod始终对应各自的存储路径（PersistentVolumeClaimTemplate）】**
 - **有序的、优雅的部署和缩放。【按顺序地增加副本、减少副本，并在减少副本时执行清理】**
 - **有序的、自动的滚动更新。【按顺序自动地执行滚动更新】**
- **限制**
 - 给定 Pod 的存储必须由 **PersistentVolume 驱动** 基于所请求的 **storage class** 来提供，或者由管理员预先提供。
 - 删除或者收缩 StatefulSet 并 不会 删除它关联的存储卷。这样做是为了保证数据安全，它通常比自动清除 StatefulSet 所有相关的资源更有价值。
 - StatefulSet 当前需要 **无头服务** 来负责 Pod 的网络标识。你需要负责创建此服务。
 - 当删除 StatefulSets 时，StatefulSet 不提供任何终止 Pod 的保证。为了实现 StatefulSet 中的 Pod 可以有序地且体面地终止，可以在删除之前将 StatefulSet 缩放为 0。
 - 在默认 **Pod 管理策略（OrderedReady）** 时使用 **滚动更新**，可能进入需要 **人工干预** 才能修复的损坏状态。

如果一个应用程序不需要稳定的网络标识，或者不需要按顺序部署、删除、增加副本，**就应该考虑使用 Deployment 这类无状态（stateless）的控制器**

```
1  apiVersion: v1
2  kind: Service    #定义一个负载均衡网络
3  metadata:
4    name: stateful-tomcat
5    labels:
6      app: stateful-tomcat
7  spec:
8    ports:
9      - port: 8123
10     name: web
11     targetPort: 8080
12     clusterIP: None    #NodePort: 任意机器+NodePort都能访问，ClusterIP: 集群内能用这个ip、
                           #service域名能访问，clusterIP: None; 不要分配集群ip。headless; 无头服务。稳定的域名
13     selector:
14       app: stateful-tomcat
15 ---
16 apiVersion: apps/v1
17 kind: StatefulSet  #控制器。
18 metadata:
19   name: stateful-tomcat
20 spec:
21   selector:
22     matchLabels:
23       app: stateful-tomcat # has to match .spec.template.metadata.labels
24   serviceName: "stateful-tomcat" #这里一定注意，必须提前有个service名字叫这个的
25   replicas: 3 # by default is 1
26   template:
27     metadata:
28       labels:
29         app: stateful-tomcat # has to match .spec.selector.matchLabels
30     spec:
31       terminationGracePeriodSeconds: 10
32       containers:
33         - name: tomcat
34           image: tomcat:7
35           ports:
36             - containerPort: 8080
37               name: web
38
39   #观察效果。
40   删除一个，重启后名字，ip等都是是一样的。保证了状态
41
42
43   #细节
44   kubectl explain StatefulSet.spec
45   podManagementPolicy:
46     OrderedReady（按序）、Parallel（并发）
47
48   serviceName -required-
49     设置服务名，就可以用域名访问pod了。
50     pod-specific-string.serviceName.default.svc.cluster.local
51
52
53   #测试
54   kubectl run -i --tty --image busybox dns-test --restart=Never --rm /bin/sh
```



```

55 ping stateful-tomcat-0.stateful-tomcat
56
57 #我们在这里没有加存储卷。如果有的话 kubectl get pvc -l app=stateful-tomcat 我们就能看到
    即使Pod删了再拉起，卷还是同样的。

```

六、Job、CronJob

1、Job

Kubernetes中的 Job 对象将创建一个或多个 Pod，并确保指定数量的 Pod 可以成功执行到进程正常结束：

- 当 Job 创建的 Pod 执行成功并正常结束时，Job 将记录成功结束的 Pod 数量
- 当成功结束的 Pod 达到指定的数量时，Job 将完成执行
- 删除 Job 对象时，将清理掉由 Job 创建的 Pod

pi-mtdmq	0/1	Completed	0	84s
imagepullbackoff	0/1	ImagePullBackoff	0	47h

```

1  apiVersion: batch/v1
2  kind: Job
3  metadata:
4    name: pi
5  spec:
6    template:
7      spec:
8        containers:
9          - name: pi
10            image: perl
11            command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
12            restartPolicy: Never #Job情况下，不支持Always
13            backoffLimit: 4 #任务4次都没成，认为失败
14            activeDeadlineSeconds: 10
15
16
17
18 #默认这个任务需要成功执行一次。
19
20 #查看job情况
21 kubectl get job
22
23 #修改下面参数设置再试试
24 #千万不要用阻塞容器。nginx。job由于Pod一直running状态。下一个永远得不到执行，而且超时了，当前
    running的Pod还会删掉
25
26 kubectl api-resources

```

```

1  #参数说明
2  kubectl explain job.spec
3    activeDeadlineSeconds: 10 总共维持10s
4    #该字段限定了 Job 对象在集群中的存活时长，一旦达到 .spec.activeDeadlineSeconds
    指定的时长，该 Job 创建的所有的 Pod 都将被终止。但是Job不会删除，Job需要手动删除，或者使用ttl
    进行清理

```

```

5     backoffLimit:
6         #设定 Job 最大的重试次数。该字段的默认值为 6；一旦重试次数达到了 backoffLimit 中的
        值, Job 将被标记为失败, 且尤其创建的所有 Pod 将被终止;
7     completions: #Job结束需要成功运行的Pods。默认为1
8     manualSelector:
9     parallelism: #并行运行的Pod个数, 默认为1
10    ttlSecondsAfterFinished:
11        ttlSecondsAfterFinished: 0 #在job执行完时马上删除
12        ttlSecondsAfterFinished: 100 #在job执行完后, 等待100s再删除
13    #除了 CronJob 之外, TTL 机制是另外一种自动清理已结束Job (Completed 或 Finished)
    的方式:
14        #TTL 机制由 TTL 控制器 提供, ttlSecondsAfterFinished 字段可激活该特性
15        #当 TTL 控制器清理 Job 时, TTL 控制器将删除 Job 对象, 以及由该 Job 创建的所有 Pod
    对象。
16
17    # job超时以后 已经完成的不删, 正在运行的Pod就删除
18    #单个Pod时, Pod成功运行, Job就结束了
19    #如果Job中定义了多个容器, 则Job的状态将根据所有容器的执行状态来变化。
20    #Job任务不建议去运行nginx, tomcat, mysql等阻塞式的, 否则这些任务永远完不了。
21    ##如果Job定义的容器中存在http server、mysql等长期的容器和一些批处理容器, 则Job状态不会发生变
    化(因为长期运行的容器不会主动结束)。此时可以通过Pod的.status.containerStatuses获取指定容
    器的运行状态。

```

- manualSelector:
 - job同样可以指定selector来关联pod。需要注意的是job目前可以使用两个API组来操作, batch/v1和extensions/v1beta1。当用户需要自定义selector时, 使用两种API组时定义参数有所差异。
 - 使用batch/v1时, 用户需要将job.spec.manualSelector设置为true, 才可以定制selector。默认为false。
 - 使用extensions/v1beta1时, 用户不需要额外的操作。因为extensions/v1beta1的spec.autoSelector默认为false, 该项与batch/v1的spec.manualSelector含义正好相反。换句话说, 使用extensions/v1beta1时, 用户不想定制selector时, 需要手动将spec.autoSelector设置为true。

2、CronJob

CronJob 按照预定的时间计划 (schedule) 创建 Job (注意: 启动的是Job不是Deploy, rs)。一个 CronJob 对象类似于 crontab (cron table) 文件中的一行记录。该对象根据 **Cron** 格式定义的时间计划, 周期性地创建 Job 对象。

Schedule

所有 CronJob 的 **schedule** 中所定义的时间, 都是基于 master 所在时区来进行计算的。

一个 CronJob 在时间计划中的每次执行时刻, 都创建 **大约** 一个 Job 对象。这里用到了 **大约**, 是因为在少数情况下会创建两个 Job 对象, 或者不创建 Job 对象。尽管 K8S 尽最大的可能性避免这种情况的出现, 但是并不能完全杜绝此现象的发生。因此, Job 程序必须是 **幂等的**。

当以下两个条件都满足时, Job 将至少运行一次:

- **startingDeadlineSeconds** 被设置为一个较大的值, 或者不设置该值 (默认值将被采纳)
- **concurrencyPolicy** 被设置为 **Allow**

```

1  # kubectl explain cronjob.spec
2
3  concurrencyPolicy: 并发策略
4  "Allow" (允许, default):
5  "Forbid" (禁止): forbids: 前个任务没执行完, 要并发下一个的话, 下一个会被跳过
6  "Replace" (替换): 新任务, 替换当前运行的任务
7
8  failedJobsHistoryLimit: 记录失败数的上限, Defaults to 1.
9  successfulJobsHistoryLimit: 记录成功任务的上限。 Defaults to 3.
10 #指定了 CronJob 应该保留多少个 completed 和 failed 的 Job 记录。将其设置为 0, 则
    CronJob 不会保留已经结束的 Job 的记录。
11
12 jobTemplate: job怎么定义 (与前面我们说的job一样定义法)
13
14 schedule: cron 表达式:
15
16 startingDeadlineSeconds: 表示如果Job因为某种原因无法按调度准时启动, 在
    spec.startingDeadlineSeconds时间段之内, CronJob仍然试图重新启动Job, 如果
    在.spec.startingDeadlineSeconds时间之内没有启动成功, 则不再试图重新启动。如果
    spec.startingDeadlineSeconds的值没有设置, 则没有按时启动的任务不会被尝试重新启动。
17
18
19
20 suspend 暂停定时任务, 对已经执行了的任务, 不会生效; Defaults to false.

```

```

1  apiVersion: batch/v1beta1
2  kind: CronJob
3  metadata:
4    name: hello
5  spec:
6    schedule: "*/1 * * * *" #分、时、日、月、周
7    jobTemplate:
8      spec:
9        template:
10         spec:
11           containers:
12             - name: hello
13               image: busybox
14               args:
15                 - /bin/sh
16                 - -c
17                 - date; echo Hello from the Kubernetes cluster
18           restartPolicy: OnFailure

```

七、GC

<https://kubernetes.io/zh/docs/concepts/workloads/controllers/ttlafterfinished/>

这是alpha版本

这个特性现在在v1.12版本是alpha阶段, 而且默认关闭的, 需要手动开启。

- 需要修改的组件包括apiserver、controller还要scheduler。
- apiserver、controller还要scheduler都是以pod的形式运行的，所以直接修改/etc/kubernetes/manifests下面对应的三个.yaml静态文件，加入 `--feature-gates=TTLAfterFinished=true` 命令，然后重启对应的pod即可。

例如修改后的kube-scheduler.yaml的spec部分如下， kube-apiserver.yaml和kube-controller-manager.yaml也在spec部分加入 `--feature-gates=TTLAfterFinished=true` 即可。

什么是垃圾回收

Kubernetes garbage collector（垃圾回收器）的作用是删除那些曾经有 owner，后来又不再有 owner 的对象。描述

垃圾收集器如何删除从属对象

当删除某个对象时，可以指定该对象的从属对象是否同时被自动删除，这种操作叫做级联删除（cascading deletion）。级联删除有两种模式：后台（background）和前台（foreground）

如果删除对象时不删除自动删除其从属对象，此时，从属对象被认为是孤儿（或孤立的 orphaned）

通过参数 `--cascade`， kubectl delete 命令也可以选择不同的级联删除策略：

- `--cascade=true` 级联删除
- `--cascade=false` 不级联删除 orphan

```
1  #删除rs，但不删除级联Pod
2  kubectl delete replicaset my-repset --cascade=false
```