# 云原生

## Tips



Kubernetes 在计算 availableReplicas 数值时不考虑终止过程中的 Pod， availableReplicas 的值一定介于 replicas - maxUnavailable 和 replicas + maxSurge 之间。 因此，你可能在上线期间看到 Pod 个数比预期的多，Deployment 所消耗的总的资源也大于 replicas + maxSurge 个 Pod 所用的资源，直到被终止的 Pod 所设置的 terminationGracePeriodSeconds 到期为止。意思就是给pod发了杀死进程的命令后, 留给你自己退出的时间,到这个时间不管你有没有优雅退出都会把pod干掉

kubectl get pd yws-test-2-6b94c5f8bf-dpdfk -o yaml

kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1

kubectl rollout history deployment yws-test-2 -n test -v=9

kubectl get pod -n test -l app=yws-test-2

-l是根据label的意思



注意：Dockerfile 的指令每执行一次都会在 docker 上新建一层。所以过多无意义的层，会造成镜像膨胀过大。例如：

FROM centos

RUN yum -y install wget

RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz"

RUN tar -xvf redis.tar.gz

以上执行会创建 3 层镜像。可简化为以下格式：

FROM centos

RUN yum -y install wget \

&& wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz" \

&& tar -xvf redis.tar.gz

如上，以 && 符号连接命令，这样执行后，只会创建 1 层镜像

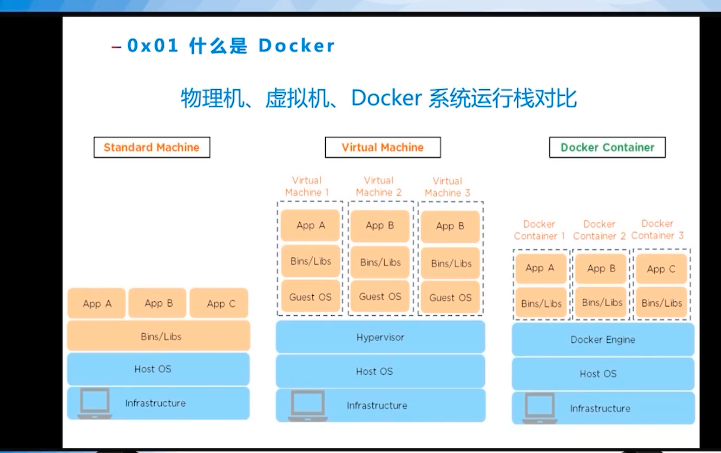
如果不指定namespace 就会创建到默认的namespace,可以给namespace指定quato

每个resource的meta里面的name和namespace是必须的.

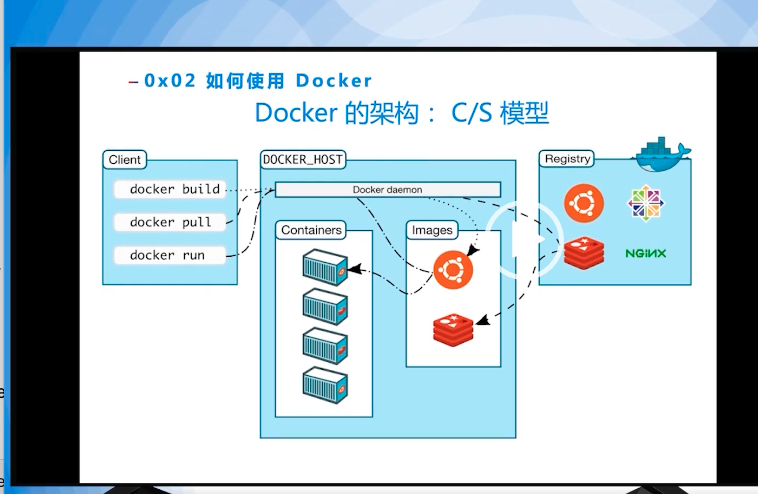
包括nfs pvc之类的

pod漂移指的这个pod部署在node1, 下次可能就部署在node2

同一个pod多个容器,只是保证这两个容器部署到了同一个node上,这两个容器通信比较方便 ,但是还是两个container 两个进程的



虚拟机多了个hypervisor 这个是个把os弄成多个硬件接口的技术,还多个了guestos.



<https://hub.docker.com/>

Dockerfile 构建docker镜像的语法. 后面docker build的时候会用到.

## Contaird

containerd是从Docker中分离出来的一个项目，可以作为一个底层容器运行时，现在它成了Kubernete容器运行时更好的选择。

不仅仅是Docker，还有很多云平台也支持containerd作为底层容器运行时，具体参考下图。

K8S发布CRI（Container Runtime Interface），统一了容器运行时接口，凡是支持CRI的容器运行时，皆可作为K8S的底层容器运行时。

K8S为什么要放弃使用Docker作为容器运行时，而使用containerd呢？

如果你使用Docker作为K8S容器运行时的话，kubelet需要先要通过dockershim去调用Docker，再通过Docker去调用containerd。

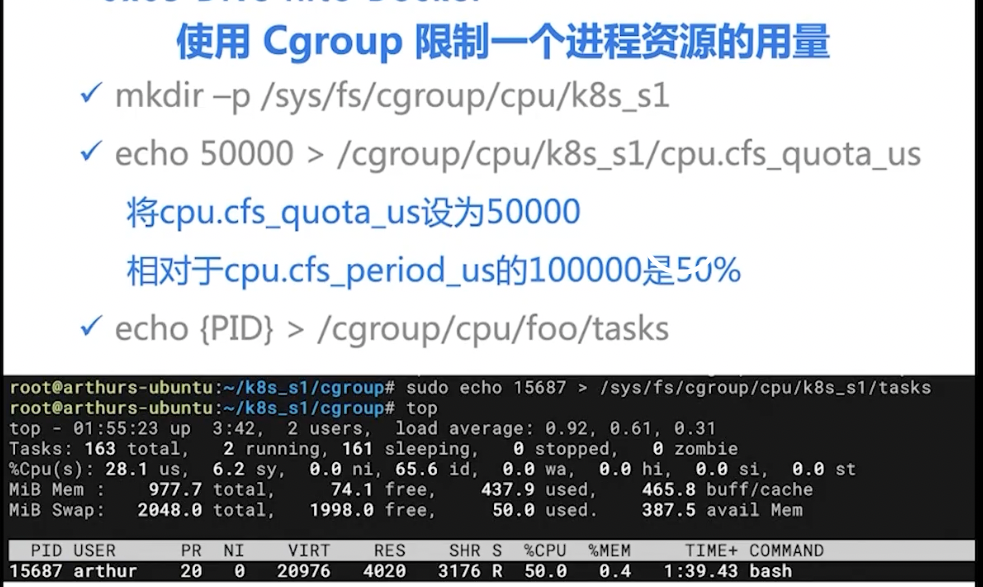
如果你使用containerd作为K8S容器运行时的话，由于containerd内置了CRI插件，kubelet可以直接调用containerd。

## Cgroup

Linux control group 限制一个进程组的资源,包括cpu 内存 网络io等

How much you can use

一组docker进程放到一个控制组里, 通过给这个控制组分配指定的可用资源,达到控制这一组进程可用资源的目的.



这就是个案例,一个死循环的程序占cpu100%, 然后你执行上面三个命令,把这个程序的pid写到cgroup的配置里面,这个程序的cpu就会降到50%

## Namespace

What you can see

主要做访问隔离, 即同一个命名空间的多个资源(cpu mem network)可以互相按到,但是之外的看不到.

就是给你一个单独的隔离空间, 你在这个新空间里创建的进程的pid都是从1开始的,是和其他的namespace的pid不冲突的, 和root namespace的pid都不冲突的.

## AUFS

高性能分层文件系统, 镜像可以通过分层实现继承.大大降低docker image的体积.

## DockerCompose

dockerfile: 构建单个服务镜像，以脚本形式

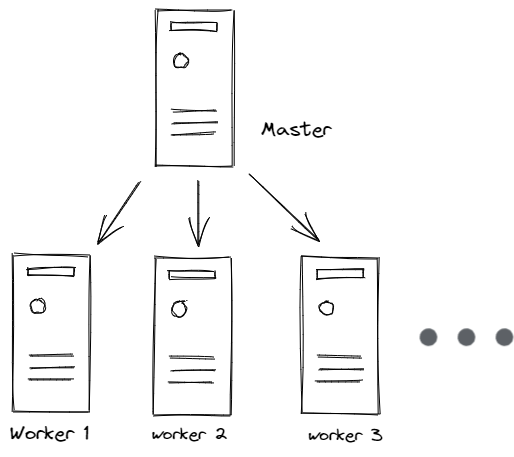
docker-compose：单机器多镜像编排容器

k8s：跨服务编排

可以通过yml文件描述 一个工程下的多个service, 每个service用那个镜像或者dockerfile创建容器, 容器有多少个.

还可以更新容器配置等.

## 集群架构

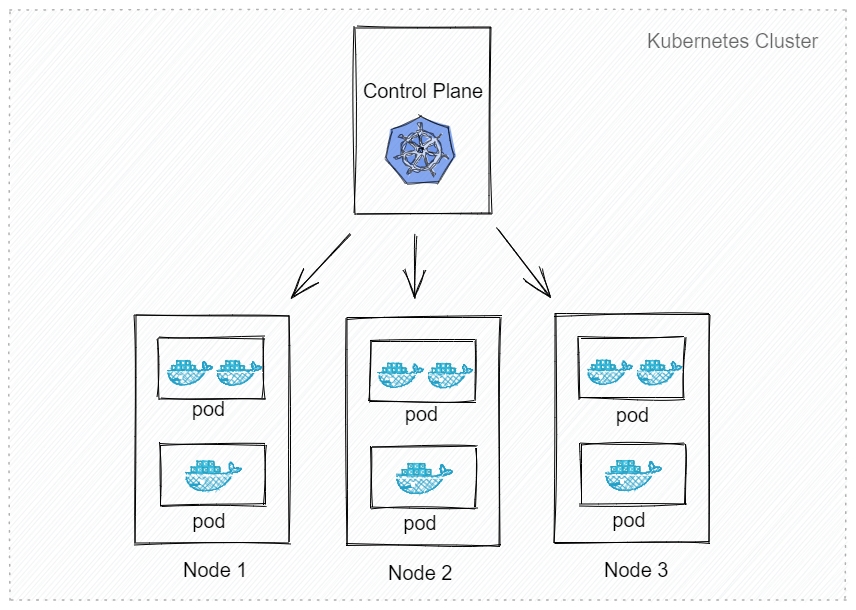


master

主节点，控制平台，不需要很高性能，不跑任务，通常一个就行了，也可以开多个主节点来提高集群可用度。

worker

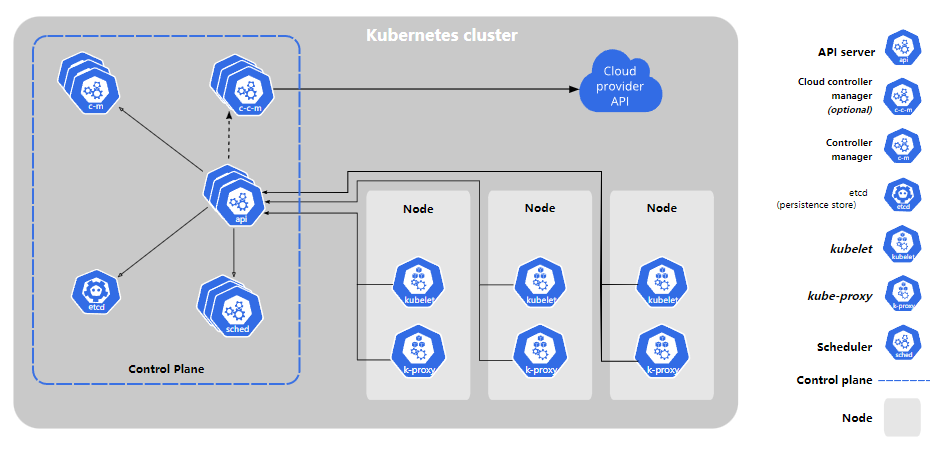
工作节点，可以是虚拟机或物理计算机，任务都在这里跑，机器性能需要好点；通常都有很多个，可以不断加机器扩大集群；每个工作节点由主节点管理.



## 组件

kube-apiserver API 服务器，公开了 Kubernetes API  
etcd 键值数据库，可以作为保存 Kubernetes 所有集群数据的后台数据库  
kube-scheduler 调度 Pod 到哪个节点运行

scheduler 分配那个pod到哪个node,根据打分 看那个node分最高 这个pod就给谁  
kube-controller 集群控制器  
cloud-controller 与云服务商交互



## Quta

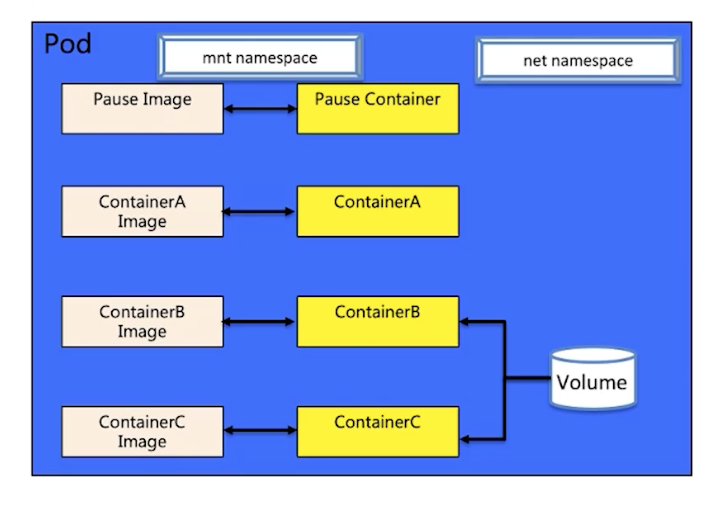
CPU的计量单位叫毫核(m)。一个节点的CPU核心数量乘以1000，得到的就是节点总的CPU总数量。如，一个节点有两个核，那么该节点的CPU总量为2000m.

## POD

Pod 在其生命周期中只会被调度一次。 一旦 Pod 被调度（分派）到某个节点，Pod 会一直在该节点运行，直到 Pod 停止或者 被终止。

一组容器 共享namespace 原子的可部署单元

豆荚，K8S 调度、管理的最小单位，一个 Pod 可以包含一个或多个容器，每个 Pod 有自己的虚拟IP。一个工作节点可以有多个 pod，主节点会考量负载自动调度 pod 到哪个节点运行



pause是随着pod一起诞生的第一个container.

InitContainer 举一个最简单的例子，假设我们有一个 Web 服务，该服务又依赖于另外一个数据库服务。

但是在在启动这个 Web 服务的时候，我们并不能保证依赖的这个数据库服务就已经启动起来了，所以可能会出现一段时间内 Web 服务连接数据库异常。

要解决这个问题的话我们就可以在 Web 服务的 Pod 中使用一个InitContainer，在这个初始化容器中去检查数据库是否已经准备好了，准备好了过后初始化容器就结束退出，然后我们的主容器 Web 服务被启动起来，这个时候去连接数据库就不会有问题了。

但其实不用 initContainer 的话，这部分依赖的检查实际也可以移入应用容器的程序中，确实是如此。

## CRI container runtime interface

每个node都需要安装容器运行时 CRI 这东西是docker或者container

## CNI容器网络

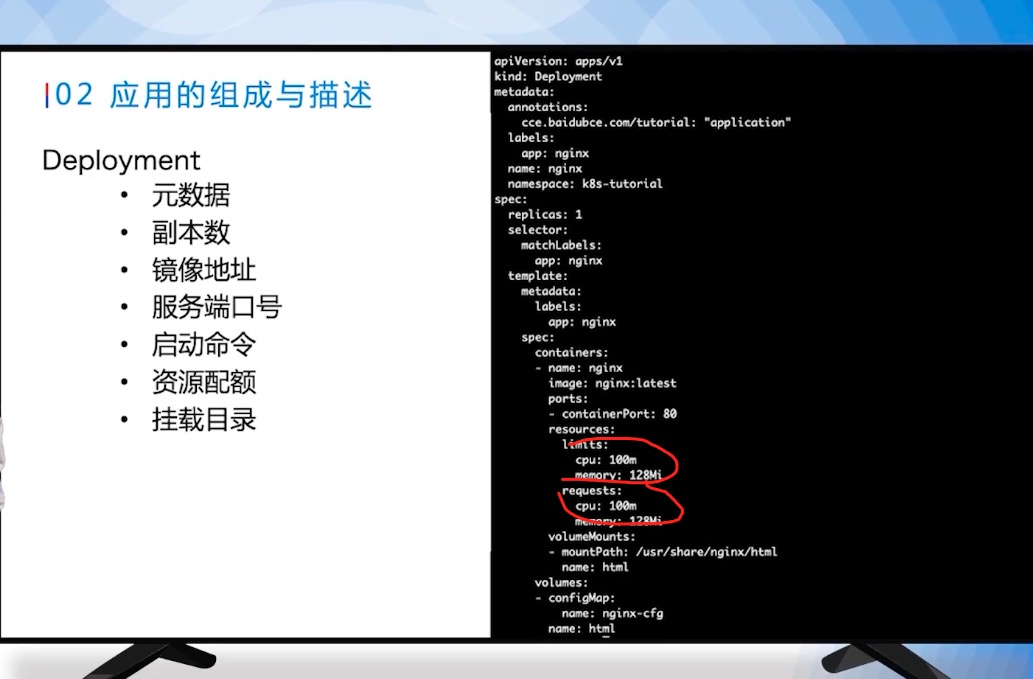
<https://www.sohu.com/a/510047266_121124376>

## CRD

## Deployment和ReplicaSet

实例数量,镜像,资源配置都是挂在deployment上的,所以他是部署的英文

新的 ReplicaSet 会被创建，Deployment 以受控速率将 Pod 从旧 ReplicaSet 迁移到新 ReplicaSet。 每个新的 ReplicaSet 都会更新 Deployment 的修订版本。



使用 Deployment 而不直接创建 ReplicaSet 是因为 Deployment 对象拥有许多 ReplicaSet 没有的特性，例如滚动升级和回滚。

Deployment 控制器是建立在 rs 之上的一个控制器，可以管理多个 rs，每次更新镜像版本，都会生 成一个新的 rs，把旧的 rs 替换掉，多个 rs 同时存在，但是只有一个 rs 运行

## Service和StatefulSet

Service 通过 label 关联对应的 Pod

Servcie 生命周期不跟 Pod 绑定，不会因为 Pod 重创改变 IP

提供了负载均衡功能，自动转发流量到不同 Pod

可对集群外部提供访问端口

集群内部可通过服务名字访问.

是用来控制有状态服务，StatefulSet 中的每个 Pod 的名字都是事先确定的，不能更改。 StatefulSet 中的 Pod，每个 Pod 挂载自己独立的存储，如果一个 Pod 出现故障，从其他节点启动一个同样名字的 Pod，要挂载上原来 Pod 的存储继续以它的状态提供服务。

## DaemonSet

确保全部（或者某些）node上运行一个 Pod 的副本。 当有node加入集群时， 也会为他们新增一个 Pod 。 当有节点从集群移除时，这些 Pod 也会被回收。删除 DaemonSet 将会删除它创建的所有 Pod。

DaemonSet 的一些典型用法：

在每个node上运行集群守护进程

在每个node上运行日志收集守护进程

在每个node上运行监控守护进程

一种简单的用法是为每种类型的守护进程在所有的node上都启动一个 DaemonSet。 一个稍微复杂的用法是为同一种守护进程部署多个 DaemonSet；每个具有不同的标志， 并且对不同硬件类型具有不同的内存、CPU 要求。

## 存储

临时存储

   resources:

      limits:

        cpu: "1"

        eks.baidu-int.com/cpu: "15"

        ephemeral-storage: 50Gi

        memory: 2Gi

写在resource里面就行了 一般无状态服务就是这样

半永久存储

hostpath类型则是映射node文件系统中的文件或者目录到pod里。因为在k8s中pod会漂移，当pod漂移到其他node节点的时候，pod不会跨节点的去读取目录。所以说是一种半持久化的存储方式

  mountpath指的是pod的路径

spec:

containers:

- image: nginx

name: mynginx

volumeMounts:

- mountPath: /usr/share/nginx/html

name: html

volumes:

- name: html # 名称

hostPath: # 存储类型

path: /data # 物理节点上的真实路径

type: Directory # 如果该路径不存在讲如何处理，Directory是要求目录必须存在

持久化存储

mysql会这样

谓容器挂载卷就是将宿主机的目录挂载到容器中的某个目录。而持久化则意味着这个目录里面的内容不会因为容器被删除而清除，也不会和当前宿主机有什么直接关系，而是一个外部的。这样当POD重建以后或者在其他主机节点上启动后依然可以访问这些内容。

Pv 就是定义了一块磁盘, 如下面, 磁盘是由nfs提供, 这个磁盘有自己唯一的名字

nfs-pv001

apiVersion: v1

kind: PersistentVolume

metadata: # PV建立不要加名称空间，因为PV属于集群级别的

name: nfs-pv001 # PV名称

labels: # 这些labels可以不定义

name: nfs-pv001

storetype: nfs

spec: # 这里的spec和volumes里面的一样

storageClassName: normal

accessModes: # 设置访问模型

- ReadWriteMany

- ReadWriteOnce

- ReadOnlyMany

capacity: # 设置存储空间大小

storage: 500Mi

persistentVolumeReclaimPolicy: Retain # 回收策略

nfs:

path: /work/volumes/v1

server: stroagesrv01.contoso.com

Pvc 这个pvc通过label和storage和storageClassName关联到了上面的pv

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: nfs-pvc001

namespace: default

labels: # 这些labels可以不定义

name: nfs-pvc001

storetype: nfs

capacity: 500Mi

spec:

storageClassName: normal

accessModes: # PVC也需要定义访问模式，不过它的模式一定是和现有PV相同或者是它的子集，否则匹配不到PV

- ReadWriteMany

resources: # 定义资源要求PV满足这个PVC的要求才会被匹配到

requests:

storage: 500Mi # 定义要求有多大空间

pod中这样使用pvc

spec:

containers:

- name: myapp

image: tomcat:8.5.38-jre8

ports:

- name: http

containerPort: 8080

protocol: TCP

volumeMounts:

- name: tomcatedata

mountPath : "/data"

volumes:

- name: tomcatedata

persistentVolumeClaim:

claimName: nfs-pvc001

半永久存储这里是hostpath

所以说pv是op创建的一块磁盘, pvc是研发对这个磁盘的使用,然后挂载到pod的一个目录.

StorageClass

PV是运维人员来创建的，开发操作PVC，可是大规模集群中可能会有很多PV，如果这些PV都需要运维手动来处理这也是一件很繁琐的事情，所以就有了动态供给概念，也就是Dynamic Provisioning。而我们上面的创建的PV都是静态供给方式，也就是Static Provisioning。而动态供给的关键就是StorageClass，它的作用就是创建PV模板。

## PORT

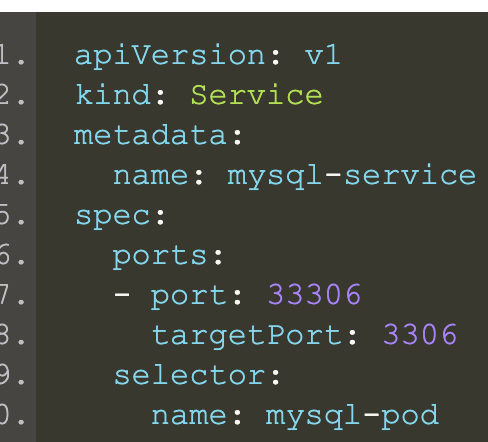
nodePort

提供了集群外部客户端访问 Service 的一种方式，nodePort 提供了集群外部客户端访问 Service 的端口，通过 nodeIP:nodePort 提供了外部流量访问k8s集群中service的入口。比如外部用户要访问k8s集群中的一个Web应用，那么我们可以配置对应service的type=NodePort，nodePort=30001。其他用户就可以通过浏览器http://node:30001访问到该web服务。

port

port是暴露在cluster ip上的端口，:port提供了集群内部客户端访问service的入口，即clusterIP:port。

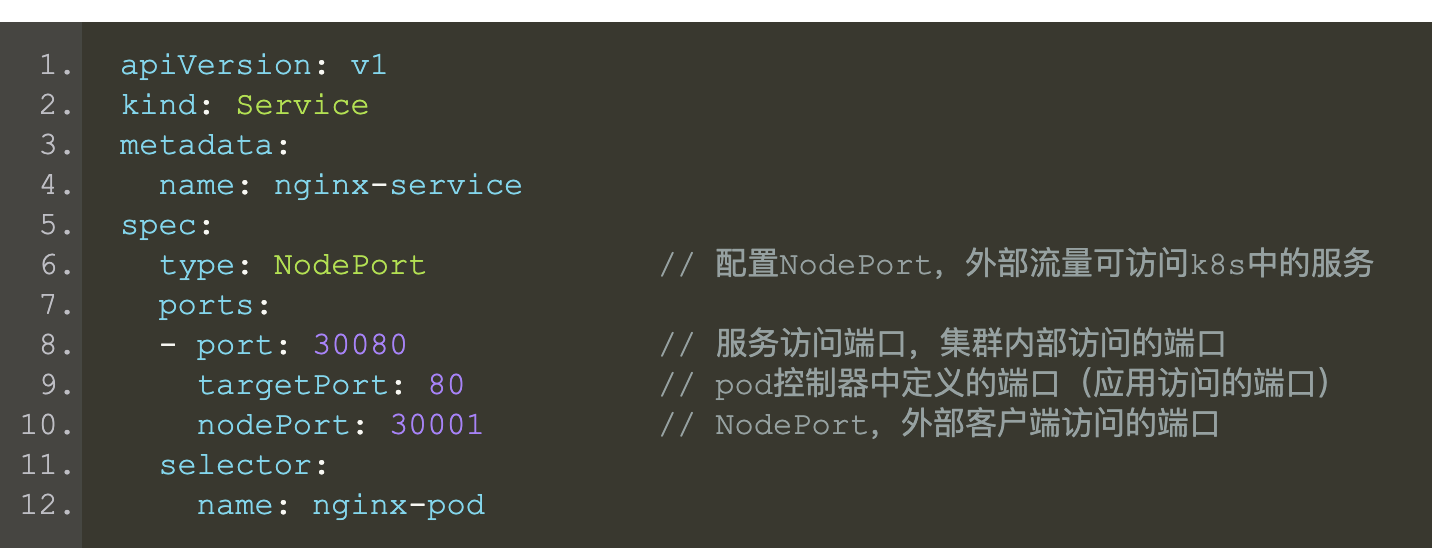
mysql容器暴露了3306端口（参考[DockerFile](https://github.com/docker-library/mysql/)），集群内其他容器通过33306端口访问mysql服务，但是外部流量不能访问mysql服务，因为mysql服务没有配置NodePort。对应的service.yaml如下：



Targetport

targetPort是pod上的端口，从port/nodePort上来的数据，经过kube-proxy流入到后端pod的targetPort上，最后进入容器。

所以这玩意是定义在service上, 和port或者nodeport一起用的



containerPort

这个是定义在deployment里面的container里面的.

如果对应的service有targetPort的话 这两个要一样



## 就绪探针存活探针

## Ingress

Ingress 和 ingress controller是两个东西

一般来说，ingress-controller的形式都是一个pod，里面跑着daemon程序和反向代理程序（典型的有nginx负载均衡器）。daemon负责不断监控集群的变化，根据ingress对象生成配置并应用新配置到反向代理，比如nginx-ingress就是动态生成nginx配置，动态更新upstream，并在需要的时候reload程序应用新配置。为了方便，后面的例子一般都以k8s官方维护的nginx-ingress为例。

ingress和ingress-controller的关系：类似于路由器与路由表的关系

与其他k8s对象一样，ingress配置也包含了apiVersion、kind、metadata、spec等关键字段。

有几个关注的在spec字段中，tls用于定义https秘钥、证书；rule用于指定请求路由规则；

这里值得关注的还有metadata.annotations字段，在ingress配置中，annotations很重要,ingress-controller有很多不同的实现，而不同的ingress-controller就可以根据“kubernetes.io/ingress.class:”来判断要使用哪些ingress配置，

同时，不同的ingress-controller也有对应的annotations配置，用于自定义一些参数，例如上面配置的‘nginx.ingress.kubernetes.io/use-regex:"true"’,最终是在生成nginx配置中，会采用location~来表示正则匹配

ingress-controller并不是k8s自带的组件，实际上ingress-controller只是一个统称，用户可以选择不同的ingress-controller实现，目前，由k8s维护的ingress-controller只有google云的GCE与ingress-nginx两个，其他还有很多第三方维护的ingress-controller，具体可以参考官方文档。但是不管哪一种ingress-controller，实现的机制都大同小异，只是在具体配置上有差异：

一般来说，ingress-controller的形式都是一个pod，里面跑着daemon程序和反向代理程序（典型的有nginx负载均衡器）。daemon负责不断监控集群的变化，根据ingress对象生成配置并应用新配置到反向代理，比如nginx-ingress就是动态生成nginx配置，动态更新upstream，并在需要的时候reload程序应用新配置。为了方便，后面的例子一般都以k8s官方维护的nginx-ingress为例

简单来说，ingress-controller才是负责具体转发的组件，通过各种方式将它暴露在集群入口，外部对集群的请求流量会先到ingress-controller，而ingress对象是用来告诉ingress-controller改如何转发请求，比如哪些域名哪些path要转发到哪些服务等等

ingress 所以代理的是service 具体pod的变化啥的都是交给service的机制去保证的

ingress这个玩意，简单的理解就是你原来要改nginx配置，然后配置各种域名对应哪个service，现在把这个动作抽象出来，变成一个ingress对象，你可以用yaml创建，每次不要去改nginx了，直接改yaml然后创建/更新就行；那么问题来了：“nginx咋整？”

ingress controller这东西就是解决“nginx咋整”的;ingress controller通过与k8s api交互，动态的去感知集群中ingress规则变化，然后读取它，按照他自己模板生成一段nginx配置，再写到nginx pod里，最后reload一下，工作流程如下：

当然咱实际应用中，最新版本k8s已经将nginx与ingress controller合并为一个组件，所以ngxin无需单独部署，只需要部署ingress controller即可。

ingress controller是一个pod服务，封装了一个web前端负载均衡器，同时在其基础上实现了动态感知ingress并根据ingress的定义生成前端web负载均衡器的配置文件，ingress-nginx-controller本质上就是一个nginx，只不过它能根据ingress资源定义的动态生成nginx的配置文件，然后动态reload。个人觉得ingress controller的重大作用是将前端负载均衡器和k8s完美地结合起来，一方面在云、容器平台下方便配置管理，另一方面实现了集群统一的流量入口，而不是像nodeport那样给集群打多个孔

总的来说要使用ingress，得先部署ingress controller实体（相当于前端nginx），然后再创建ingress（相当于nginx配置的k8s资源体现），ingress controller部署后之后会动态检测到ingress的创建清楚并生成相应的配置。

## 对象

每一个创建的pod, rs, ss, deploy, service都是一个对象

集群中的每一个对象都有一个[名称]（#names）来标识在同类资源中的唯一性。

每个 Kubernetes 对象也有一个 [UID]（#uids）来标识在整个集群中的唯一性。

apiVersion - 创建该对象所使用的 Kubernetes API 的版本

kind - 想要创建的对象的类别

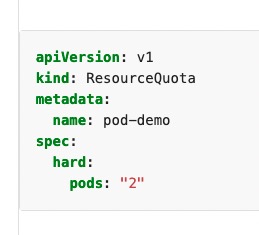
metadata - 帮助唯一性标识对象的一些数据，包括一个 name 字符串、UID 和可选的 namespace

spec - 你所期望的该对象的状态

api对象

api/v1

前面是api后面是版本





namespace的创建的apiversion也是v1

而pod, deploy,service的创建都是apps/v1

## 自动扩缩容

HorizontalPodAutoscaler 自动扩缩容 k8s的一种kind

## Watch

k8s也是watch机制监听pod变化啥的

etcd也是watch机制的