# 云原生

## Tips

ownerReferences 字段表示了该资源所属的资源的，表示由谁控制。

Controlled By 表示正在被哪个资源控制。

Finalizers 字段属于 Kubernetes GC 垃圾收集器，是一种删除拦截机制，能够让控制器实现异步的删除前（Pre-delete）回调。其存在于任何一个资源对象的 Meta[1] 中，在 k8s 源码中声明为 []string，该 Slice 的内容为需要执行的拦截器名称。

Endpoints(即Pod-IP)。



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

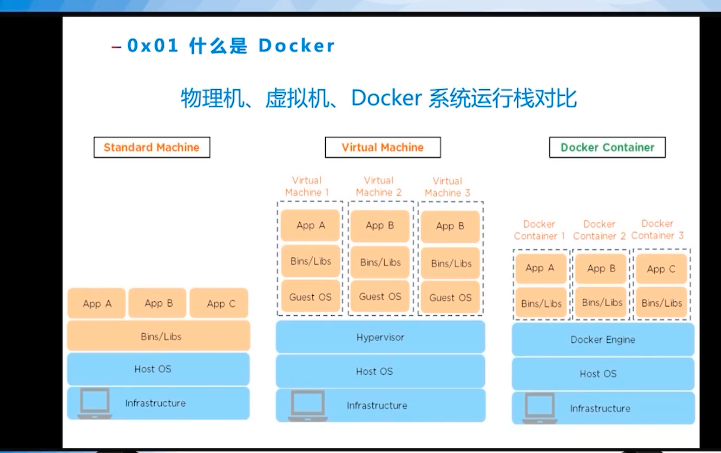
如果不指定namespace 就会创建到默认的namespace,可以给namespace指定quato

每个resource的meta里面的name和namespace是必须的.

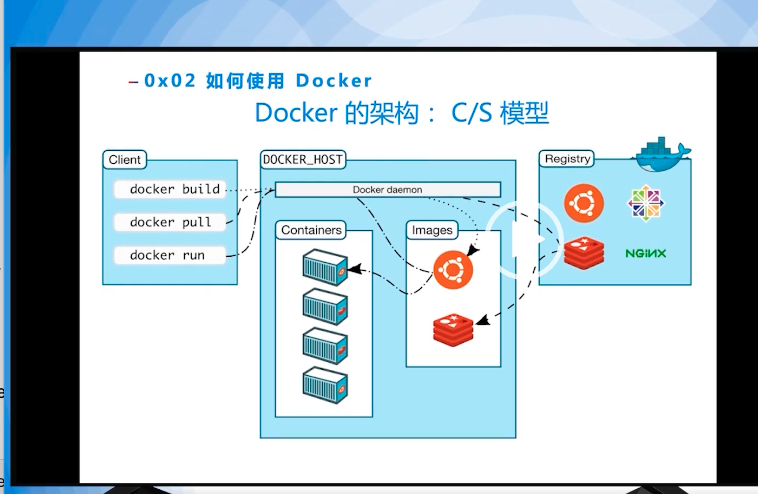
包括nfs pvc之类的

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。

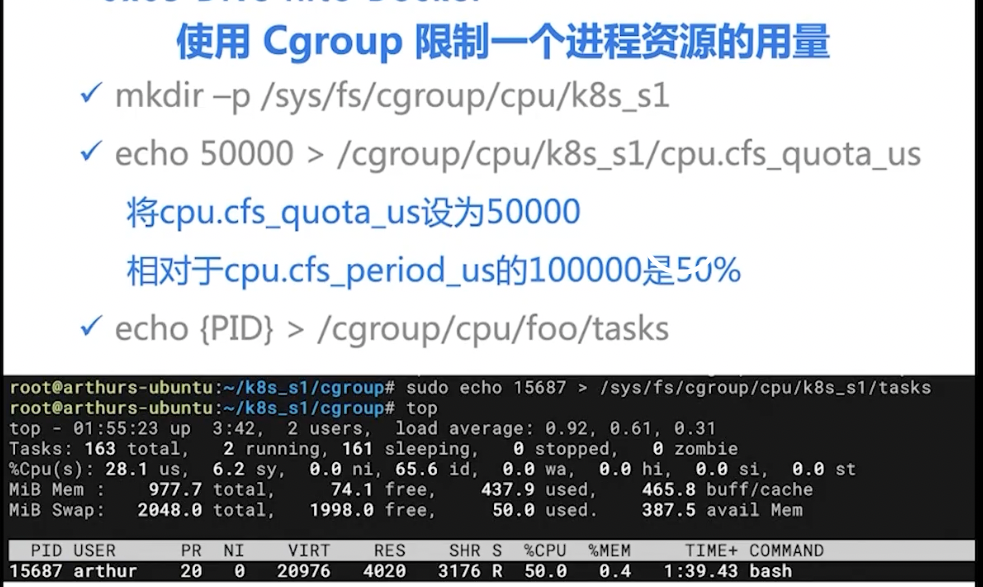
如果你使用containerd作为K8S容器运行时的话，由于containerd内置了CRI插件，kubelet可以直接调用containerd。

## 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

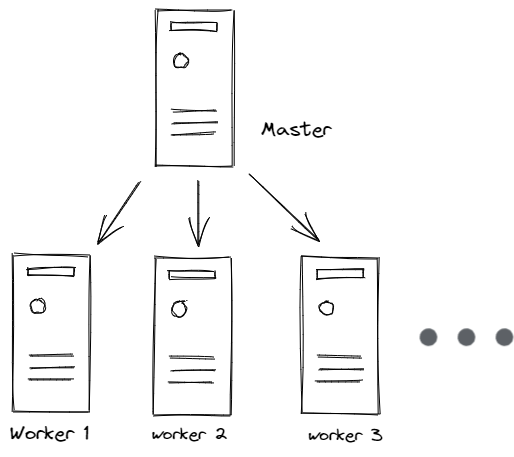
docker-compose：单机器多镜像编排容器

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

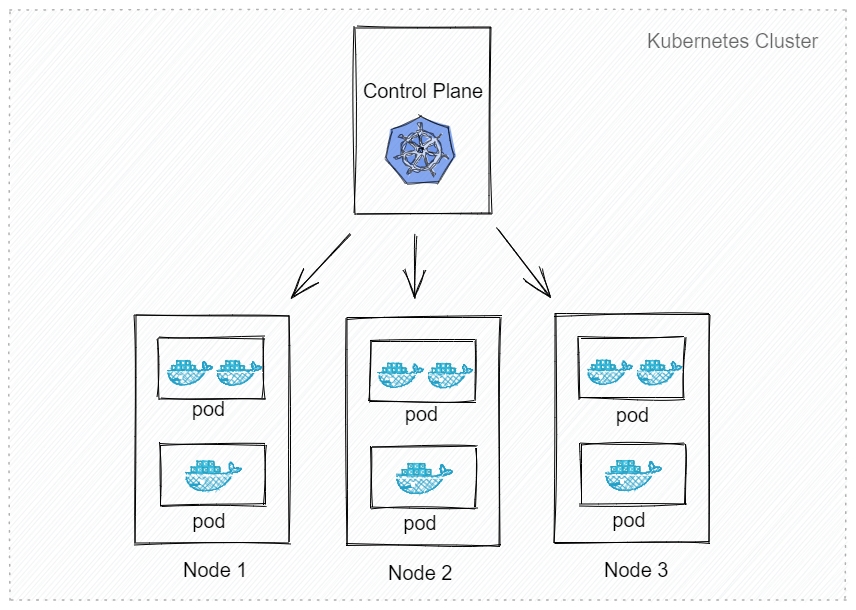


master

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

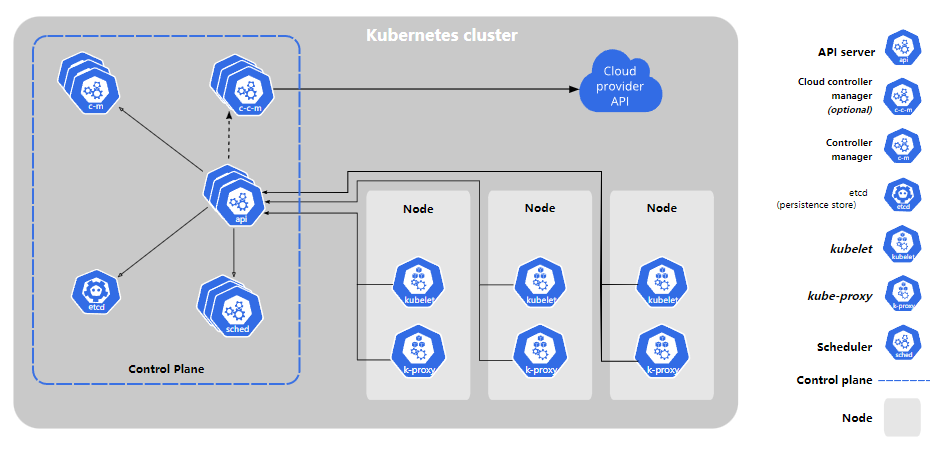
### 控制面组件

kube-apiserver API 服务器，公开了 Kubernetes API, 模块间的通信枢纽, 只有apiserver可以有权限操作etcd.

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



### Node组件

## Quta

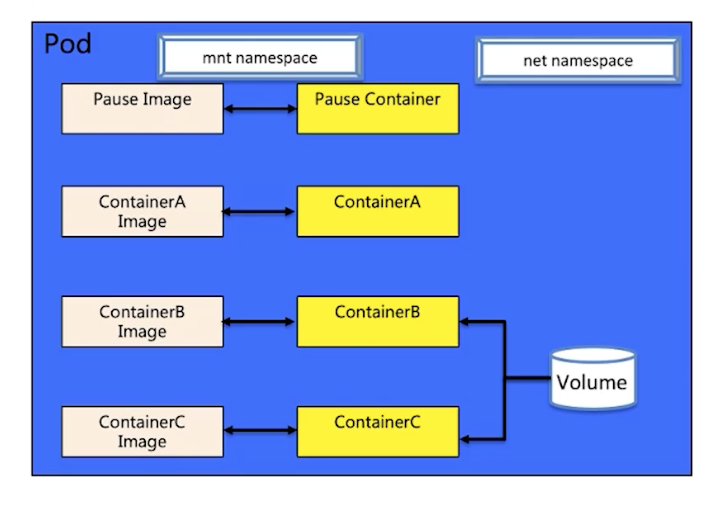
CPU的计量单位叫毫核(m)。一个节点的CPU核心数量乘以1000，得到的就是节点总的CPU总数量。如，一个节点有两个核，那么该节点的CPU总量为2000m.

## 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

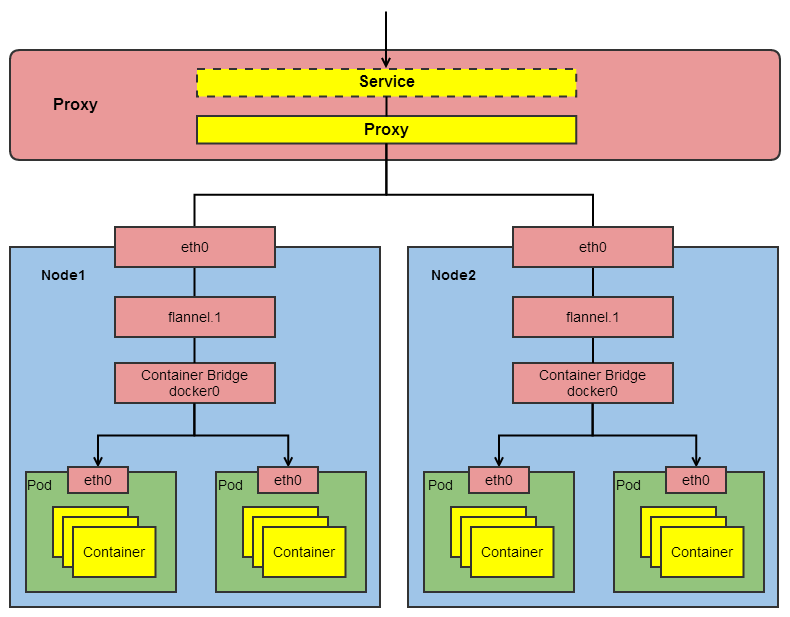
## K8S网络

1, 每个Pod都拥有一个独立的IP地址，而且假定所有Pod都在一个可以直接连通的、扁平的网络空间中，不管是否运行在同一Node上都可以通过Pod的IP来访问.

2,k8s中Pod的IP是最小粒度IP。同一个Pod内所有的容器共享一个网络堆栈，该模型称为IP-per-Pod模型。

3, IP-per-Pod模型从端口分配、域名解析、服务发现、负载均衡、应用配置等角度看，Pod可以看作是一台独立的VM或物理机。

4, Pod由docker0实际分配的IP，Pod内部看到的IP地址和端口与外部保持一致。同一个Pod内的不同容器共享网络，可以通过localhost来访问对方的端口，类似同一个VM内的不同进程。



可以看到, 在所有的node外面还有一个proxy层.

|  |  |
| --- | --- |
| Proxy-IP | 代理层公网地址IP，外部访问应用的网关服务器。[实际需要关注的IP] |
| Service-IP | Service的固定虚拟IP，Service-IP是内部，外部无法寻址到。 |
| Node-IP | 容器宿主机的主机IP。 |
| Container-Bridge-IP | 容器网桥（docker0）IP，容器的网络都需要通过容器网桥转发。 |
| Pod-IP | Pod的IP，等效于Pod中网络容器的Container-IP。 |
| Container-IP | 容器的IP，容器的网络是个隔离的网络空间。 |

Underlay网络是由各类物理设备构成，通过使用路由协议保证其设备之间的IP连通性的承载网络。  
Overlay网络是通过网络[虚拟化技术](https://so.csdn.net/so/search?q=%E8%99%9A%E6%8B%9F%E5%8C%96%E6%8A%80%E6%9C%AF&spm=1001.2101.3001.7020" \t "/Users/yangwenshuo/Documents\\x/_blank)，在同一张Underlay网络上构建出的一张或者多张虚拟的逻辑网络。

相互连接的Overlay设备之间建立隧道，数据包准备传输出去时，设备为数据包添加新的IP头部和隧道头部，并且被屏蔽掉内层的IP头部，数据包根据新的IP头部进行转发。当数据包传递到另一个设备后，外部的IP报头和隧道头将被丢弃，得到原始的数据包，在这个过程中Overlay网络并不感知Underlay网络。

k8s的网络就是overlay网络, 进来后ip的报头就全丢了, 全靠隧道的ip了.

## Flannel

使集群中的不同Node主机创建的Docker容器都具有全集群唯一的虚拟IP地址。

创建一个新的虚拟网卡flannel.1

接收docker网桥的数据，通过维护路由表，对接收到的数据进行封包和转发.

etcd保证了所有node上flanned所看到的配置是一致的。同时每个node上的flanned监听etcd上的数据变化，实时感知集群中node的变化。

Flanneld：flannel在每个主机中运行flanneld作为agent，它会为所在主机从集群的网络地址空间中，获取一个小的网段subnet，本主机内所有容器的IP地址都将从中分配。同时Flanneld监听K8s集群数据库，为flannel.1设备提供封装数据时必要的mac，ip等网络数据信息。

就是每个node上都有的flanneld的模块, 这个模块连着etcd, 实时同步着每个pod的ip和其宿主机的mac地址,

比如poda想发消息到podb, 那么poda的flanneld模块一查就知道podb的ip和他的node的mac地址, 那么flanneld就直接把这个包的源mac地址设置好, 通过宿主node的eth0发出去, 就OK了, podb的宿主node的eth0收到后再到flanneld,到docker0, 到podb了就.

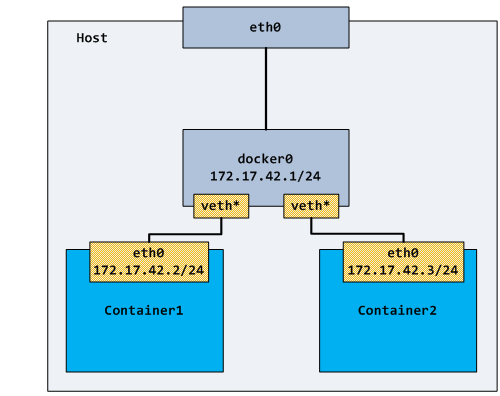
核心就是etcd实时同步所有pod的node的mac地址.

## Docker网络

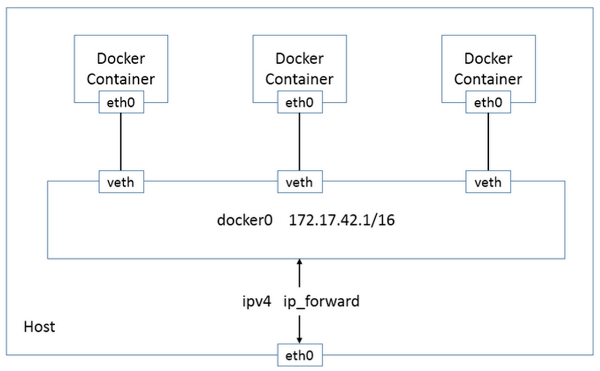
Docker使用Linux桥接，在宿主机虚拟一个Docker容器网桥(docker0)，Docker启动一个容器时会根据Docker网桥的网段分配给容器一个IP地址，称为Container-IP，同时Docker网桥是每个容器的默认网关。因为在同一宿主机内的容器都接入同一个网桥，这样容器之间就能够通过容器的Container-IP直接通信。

这个对应的就是Containerport

Docker网桥是宿主机虚拟出来的，并不是真实存在的网络设备，外部网络是无法寻址到的，这也意味着外部网络无法通过直接Container-IP访问到容器。如果容器希望外部访问能够访问到，可以通过映射容器端口到宿主主机（端口映射），即docker run创建容器时候通过 -p 或 -P 参数来启用，访问容器的时候就通过[宿主机IP]:[容器端口]访问容器。



可以看到node上都会有eth0做为物理网卡, 还有有一个docker0做为容器专用网卡.



可以登录到容器里面看到, 都是有个eth0, 这个容器的eth0, 通过虚拟以太设备对（veth pair）来通信.

外部访问容器的流程如下：

外界网络通过宿主机的IP和映射的端口port\_1访问。

当宿主机收到此类请求，会通过DNAT将请求的目标IP即宿主机IP和目标端口即映射端口port\_1替换成容器的IP和容器的端口port\_0。

由于宿主机上可以识别容器IP，所以宿主机将请求发给veth pair。

veth pair将请求发送给容器内部的eth0，由容器内部的服务进行处理。

## CNI容器网络

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

## CRD

<https://blog.csdn.net/boling_cavalry/article/details/88924194> 实战

 CRD 本身是 Kubernetes 的一种资源，允许用户自定义新的资源类型

除了 CRD 还需要用户提供一个 Controller 以实现自己的逻辑

CRD 允许用户基于已有的 Kubernetes 资源，例如 Deployment、Configmap 等，拓展集群能力

CRD 本身是一种 Kubernetes 内置的资源类型，即自定义资源的定义，用于描述用户定义的资源是什么样子。CRD 的相关概念：① 从 Kubernetes 的用户角度来看，所有东西都叫资源 Resource，就是 Yaml 里的字段 Kind 的内容，例如 Service、Deployment 等。② 除了常见内置资源之外，Kubernetes 允许用户自定义资源 Custom Resource，而 CRD 表示自定义资源的定义。

创建一个crd resource

apiVersion: apiextensions.k8s.io/v1beta1

kind: CustomResourceDefinition

metadata:

# 名称必须与下面的spec字段匹配，格式为: <plural>.<group>

name: crontabs.stable.example.com

spec:

# 用于REST API的组名称: /apis/<group>/<version>

group: stable.example.com

# 此CustomResourceDefinition支持的版本列表

versions:

- name: v1

# 每个版本都可以通过服务标志启用/禁用。

served: true

# 必须将一个且只有一个版本标记为存储版本。

storage: true

# 指定crd资源作用范围在命名空间或集群

scope: Namespaced

names:

# URL中使用的复数名称: /apis/<group>/<version>/<plural>

plural: crontabs

# 在CLI(shell界面输入的参数)上用作别名并用于显示的单数名称

singular: crontab

# kind字段使用驼峰命名规则. 资源清单使用如此

kind: CronTab

# 短名称允许短字符串匹配CLI上的资源，意识就是能通过kubectl 在查看资源的时候使用该资源的简名称来获取。

shortNames:

- ct

创建这个resource的对象

apiVersion: "stable.example.com/v1"

kind: CronTab

metadata:

name: my-new-cron-object

spec:

cronSpec: "\* \* \* \* \*/5"

image: my-awesome-cron-image

Student crd

apiVersion: apiextensions.k8s.io/v1beta1

kind: CustomResourceDefinition

metadata:

# metadata.name的内容是由"复数名.分组名"构成，如下，students是复数名，bolingcavalry.k8s.io是分组名

name: students.bolingcavalry.k8s.io

spec:

# 分组名，在REST API中也会用到的，格式是: /apis/分组名/CRD版本

group: bolingcavalry.k8s.io

# list of versions supported by this CustomResourceDefinition

versions:

- name: v1

# 是否有效的开关.

served: true

# 只有一个版本能被标注为storage

storage: true

# 范围是属于namespace的

scope: Namespaced

names:

# 复数名

plural: students

# 单数名

singular: student

# 类型名

kind: Student

# 简称，就像service的简称是svc

shortNames:

- stu

apiVersion: bolingcavalry.k8s.io/v1

kind: Student

metadata:

name: object-student

spec:

name: "张三"

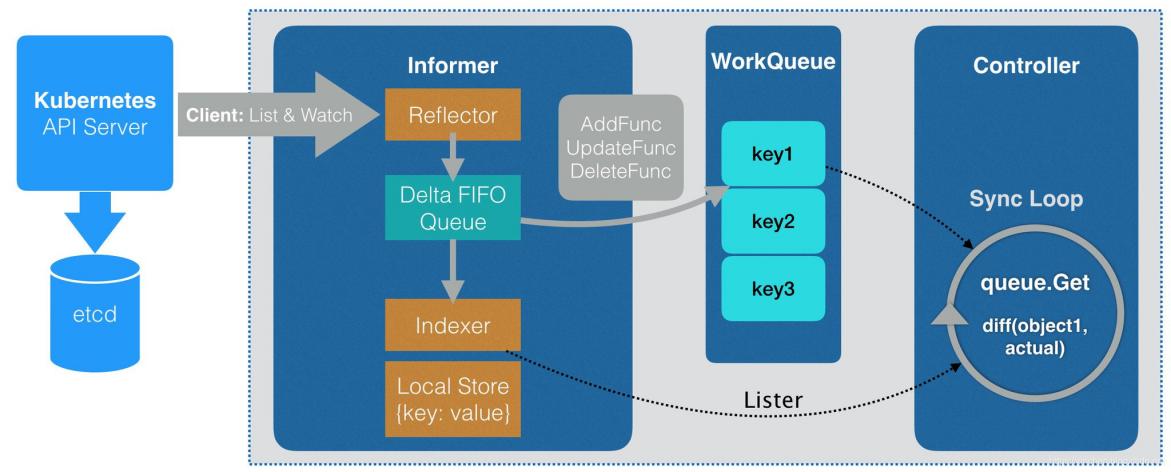
school: "深圳中学"

在k8s环境注册了resource Student，此时如果创建Student对象就会在etcd保存该对象信息

但如果仅仅是在etcd保存Student对象是没有什么意义的，试想通过deployment创建pod时，如果只在etcd创建pod对象，而不去node节点创建容器，那这个pod对象只是一条数据而已，没有什么实质性作用，其他对象如service、pv也是如此.

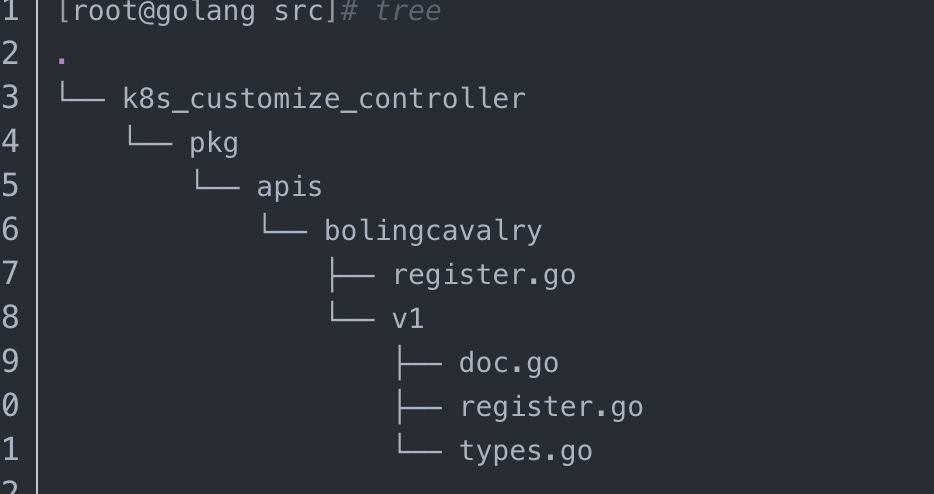
controller的作用就是监听指定对象的新增、删除、修改等变化，针对这些变化做出相应的响应（例如新增pod的响应为创建docker容器）.

API对象的变化会通过Informer存入队列（WorkQueue），在Controller中消费队列的数据做出响应，响应相关的具体代码就是我们要做的真正业务逻辑.



从上图可以发现整个逻辑还是比较复杂的，为了简化我们的自定义controller开发，k8s的大师们利用自动代码生成工具将controller之外的事情都做好了，我们只要专注于controller的开发就好.

就是你自己写一段这个, 定义文件



执行个自动生成脚本会生成这个

[root@master k8s\_customize\_controller]# tree

.

└── pkg

├── apis

│ └── bolingcavalry

│ ├── register.go

│ └── v1

│ ├── doc.go

│ ├── register.go

│ ├── types.go

│ └── zz\_generated.deepcopy.go

└── client

├── clientset

│ └── versioned

│ ├── clientset.go

│ ├── doc.go

│ ├── fake

│ │ ├── clientset\_generated.go

│ │ ├── doc.go

│ │ └── register.go

│ ├── scheme

│ │ ├── doc.go

│ │ └── register.go

│ └── typed

│ └── bolingcavalry

│ └── v1

│ ├── bolingcavalry\_client.go

│ ├── doc.go

│ ├── fake

│ │ ├── doc.go

│ │ ├── fake\_bolingcavalry\_client.go

│ │ └── fake\_student.go

│ ├── generated\_expansion.go

│ └── student.go

├── informers

│ └── externalversions

│ ├── bolingcavalry

│ │ ├── interface.go

│ │ └── v1

│ │ ├── interface.go

│ │ └── student.go

│ ├── factory.go

│ ├── generic.go

│ └── internalinterfaces

│ └── factory\_interfaces.go

└── listers

└── bolingcavalry

└── v1

├── expansion\_generated.go

└── student.go

21 directories, 27 files

然后新增一个controller.go文件

类似这样写

package main

import (

"fmt"

"time"

"github.com/golang/glog"

corev1 "k8s.io/api/core/v1"

"k8s.io/apimachinery/pkg/api/errors"

"k8s.io/apimachinery/pkg/util/runtime"

utilruntime "k8s.io/apimachinery/pkg/util/runtime"

"k8s.io/apimachinery/pkg/util/wait"

"k8s.io/client-go/kubernetes"

"k8s.io/client-go/kubernetes/scheme"

typedcorev1 "k8s.io/client-go/kubernetes/typed/core/v1"

"k8s.io/client-go/tools/cache"

"k8s.io/client-go/tools/record"

"k8s.io/client-go/util/workqueue"

bolingcavalryv1 "github.com/zq2599/k8s-controller-custom-resource/pkg/apis/bolingcavalry/v1"

clientset "github.com/zq2599/k8s-controller-custom-resource/pkg/client/clientset/versioned"

studentscheme "github.com/zq2599/k8s-controller-custom-resource/pkg/client/clientset/versioned/scheme"

informers "github.com/zq2599/k8s-controller-custom-resource/pkg/client/informers/externalversions/bolingcavalry/v1"

listers "github.com/zq2599/k8s-controller-custom-resource/pkg/client/listers/bolingcavalry/v1"

)

const controllerAgentName = "student-controller"

const (

SuccessSynced = "Synced"

MessageResourceSynced = "Student synced successfully"

)

// Controller is the controller implementation for Student resources

type Controller struct {

// kubeclientset is a standard kubernetes clientset

kubeclientset kubernetes.Interface

// studentclientset is a clientset for our own API group

studentclientset clientset.Interface

studentsLister listers.StudentLister

studentsSynced cache.InformerSynced

workqueue workqueue.RateLimitingInterface

recorder record.EventRecorder

}

// NewController returns a new student controller

func NewController(

kubeclientset kubernetes.Interface,

studentclientset clientset.Interface,

studentInformer informers.StudentInformer) \*Controller {

utilruntime.Must(studentscheme.AddToScheme(scheme.Scheme))

glog.V(4).Info("Creating event broadcaster")

eventBroadcaster := record.NewBroadcaster()

eventBroadcaster.StartLogging(glog.Infof)

eventBroadcaster.StartRecordingToSink(&typedcorev1.EventSinkImpl{Interface: kubeclientset.CoreV1().Events("")})

recorder := eventBroadcaster.NewRecorder(scheme.Scheme, corev1.EventSource{Component: controllerAgentName})

controller := &Controller{

kubeclientset: kubeclientset,

studentclientset: studentclientset,

studentsLister: studentInformer.Lister(),

studentsSynced: studentInformer.Informer().HasSynced,

workqueue: workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLimiter(), "Students"),

recorder: recorder,

}

glog.Info("Setting up event handlers")

// Set up an event handler for when Student resources change

studentInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{

AddFunc: controller.enqueueStudent,

UpdateFunc: func(old, new interface{}) {

oldStudent := old.(\*bolingcavalryv1.Student)

newStudent := new.(\*bolingcavalryv1.Student)

if oldStudent.ResourceVersion == newStudent.ResourceVersion {

//版本一致，就表示没有实际更新的操作，立即返回

return

}

controller.enqueueStudent(new)

},

DeleteFunc: controller.enqueueStudentForDelete,

})

return controller

}

//在此处开始controller的业务

func (c \*Controller) Run(threadiness int, stopCh <-chan struct{}) error {

defer runtime.HandleCrash()

defer c.workqueue.ShutDown()

glog.Info("开始controller业务，开始一次缓存数据同步")

if ok := cache.WaitForCacheSync(stopCh, c.studentsSynced); !ok {

return fmt.Errorf("failed to wait for caches to sync")

}

glog.Info("worker启动")

for i := 0; i < threadiness; i++ {

go wait.Until(c.runWorker, time.Second, stopCh)

}

glog.Info("worker已经启动")

<-stopCh

glog.Info("worker已经结束")

return nil

}

func (c \*Controller) runWorker() {

for c.processNextWorkItem() {

}

}

// 取数据处理

func (c \*Controller) processNextWorkItem() bool {

obj, shutdown := c.workqueue.Get()

if shutdown {

return false

}

// We wrap this block in a func so we can defer c.workqueue.Done.

err := func(obj interface{}) error {

defer c.workqueue.Done(obj)

var key string

var ok bool

if key, ok = obj.(string); !ok {

c.workqueue.Forget(obj)

runtime.HandleError(fmt.Errorf("expected string in workqueue but got %#v", obj))

return nil

}

// 在syncHandler中处理业务

if err := c.syncHandler(key); err != nil {

return fmt.Errorf("error syncing '%s': %s", key, err.Error())

}

c.workqueue.Forget(obj)

glog.Infof("Successfully synced '%s'", key)

return nil

}(obj)

if err != nil {

runtime.HandleError(err)

return true

}

return true

}

// 处理

func (c \*Controller) syncHandler(key string) error {

// Convert the namespace/name string into a distinct namespace and name

namespace, name, err := cache.SplitMetaNamespaceKey(key)

if err != nil {

runtime.HandleError(fmt.Errorf("invalid resource key: %s", key))

return nil

}

// 从缓存中取对象

student, err := c.studentsLister.Students(namespace).Get(name)

if err != nil {

// 如果Student对象被删除了，就会走到这里，所以应该在这里加入执行

if errors.IsNotFound(err) {

glog.Infof("Student对象被删除，请在这里执行实际的删除业务: %s/%s ...", namespace, name)

return nil

}

runtime.HandleError(fmt.Errorf("failed to list student by: %s/%s", namespace, name))

return err

}

glog.Infof("这里是student对象的期望状态: %#v ...", student)

glog.Infof("实际状态是从业务层面得到的，此处应该去的实际状态，与期望状态做对比，并根据差异做出响应(新增或者删除)")

c.recorder.Event(student, corev1.EventTypeNormal, SuccessSynced, MessageResourceSynced)

return nil

}

// 数据先放入缓存，再入队列

func (c \*Controller) enqueueStudent(obj interface{}) {

var key string

var err error

// 将对象放入缓存

if key, err = cache.MetaNamespaceKeyFunc(obj); err != nil {

runtime.HandleError(err)

return

}

// 将key放入队列

c.workqueue.AddRateLimited(key)

}

// 删除操作

func (c \*Controller) enqueueStudentForDelete(obj interface{}) {

var key string

var err error

// 从缓存中删除指定对象

key, err = cache.DeletionHandlingMetaNamespaceKeyFunc(obj)

if err != nil {

runtime.HandleError(err)

return

}

//再将key放入队列

c.workqueue.AddRateLimited(key)

}

然后编译, 然后

./k8s\_customize\_controller -kubeconfig=$HOME/.kube/config -alsologtostderr=true

这样你就把当前的进程注册到k8s里面成为一个controller了, 这个类型的crd有操作你都可以知道了.

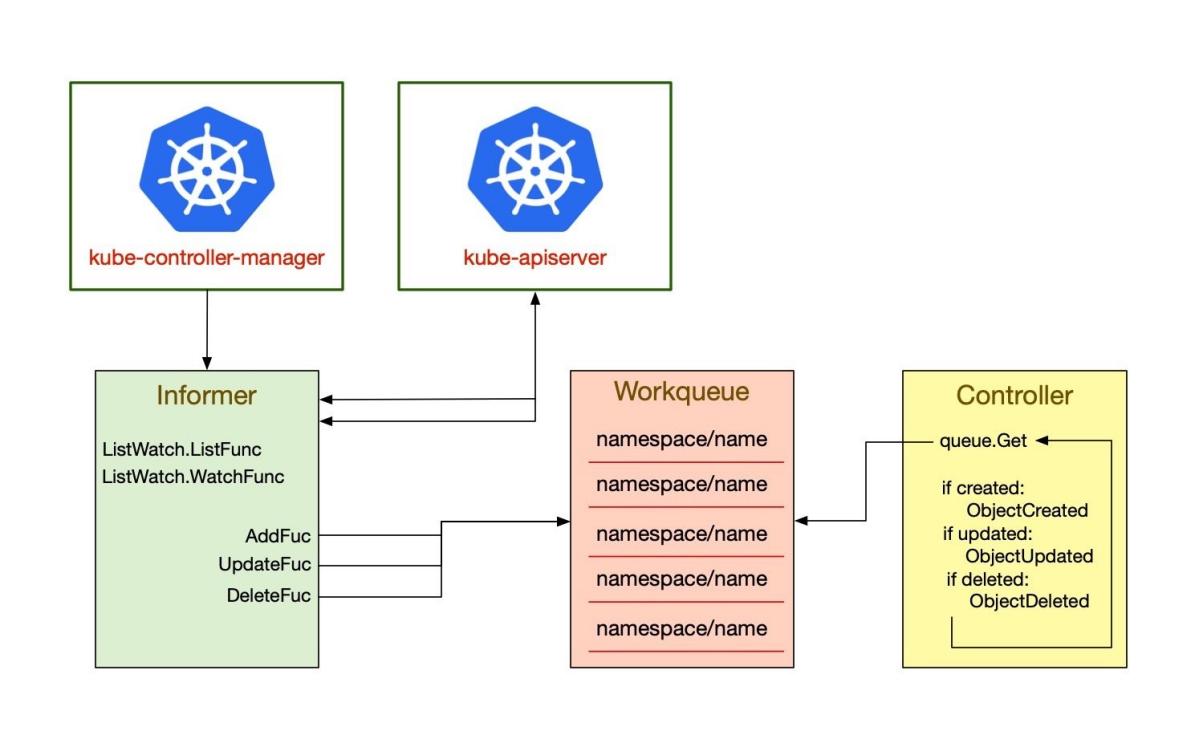


## Controller与Manager

Kubernetes 中运行了一系列控制器来确保集群的当前状态与期望状态保持一致，它们就是 Kubernetes 的大脑。例如，ReplicaSet 控制器负责维护集群中运行的 Pod 数量；Node 控制器负责监控节点的状态，并在节点出现故障时及时做出响应。总而言之， 在 [Kubernetes](https://so.csdn.net/so/search?q=Kubernetes&spm=1001.2101.3001.7020" \t "/Users/yangwenshuo/Documents\\x/_blank) 中，控制器通过监控集群的公共状态，并致力于将当前状态转变为期望的状态。

在 Kubernetes 中，每个控制器只负责某种类型的特定资源。对于集群管理员来说，了解每个控制器的角色分工至关重要.

每个控制器内部都有两个核心组件：Informer/SharedInformer 和 Workqueue。其中 Informer/SharedInformer 负责 watch Kubernetes 资源对象的状态变化，然后将相关事件（evenets）发送到 Workqueue 中，最后再由控制器的 worker 从 Workqueue 中取出事件交给控制器处理程序进行处理



[kube-controller-manager](https://kubernetes.io/zh-cn/docs/reference/command-line-tools-reference/kube-controller-manager/) 是[控制平面](https://kubernetes.io/zh-cn/docs/reference/glossary/?all=true" \l "term-control-plane" \o "" \t "/Users/yangwenshuo/Documents\\x/_blank)的组件， 负责运行[控制器](https://kubernetes.io/zh-cn/docs/concepts/architecture/controller/" \o "" \t "/Users/yangwenshuo/Documents\\x/_blank)进程。

从逻辑上讲， 每个[控制器](https://kubernetes.io/zh-cn/docs/concepts/architecture/controller/" \o "" \t "/Users/yangwenshuo/Documents\\x/_blank)都是一个单独的进程， 但是为了降低复杂性，它们都被编译到同一个可执行文件，并在同一个进程中运行。

这些控制器包括：

节点控制器（Node Controller）：负责在节点出现故障时进行通知和响应

任务控制器（Job Controller）：监测代表一次性任务的 Job 对象，然后创建 Pods 来运行这些任务直至完成

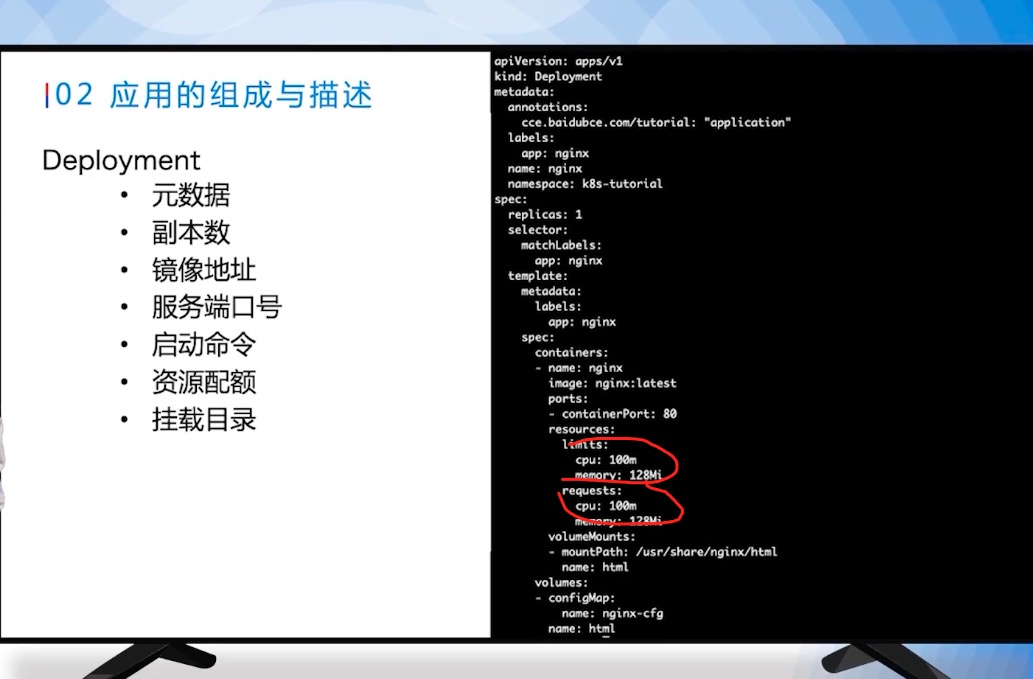
端点控制器（Endpoints Controller）：填充端点（Endpoints）对象（即加入 Service 与 Pod）

服务帐户和令牌控制器（Service Account & Token Controllers）：为新的命名空间创建默认帐户和 API 访问令牌

## 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

### 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。

### 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

## PORT

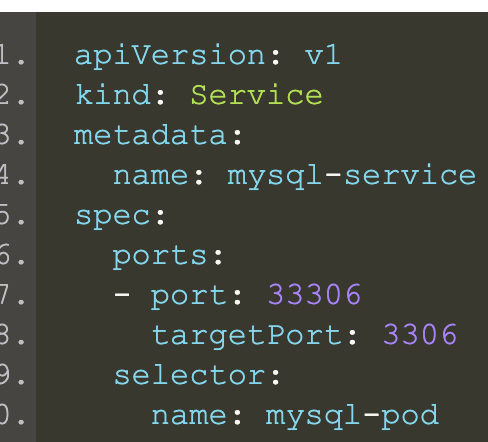
nodePort

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。

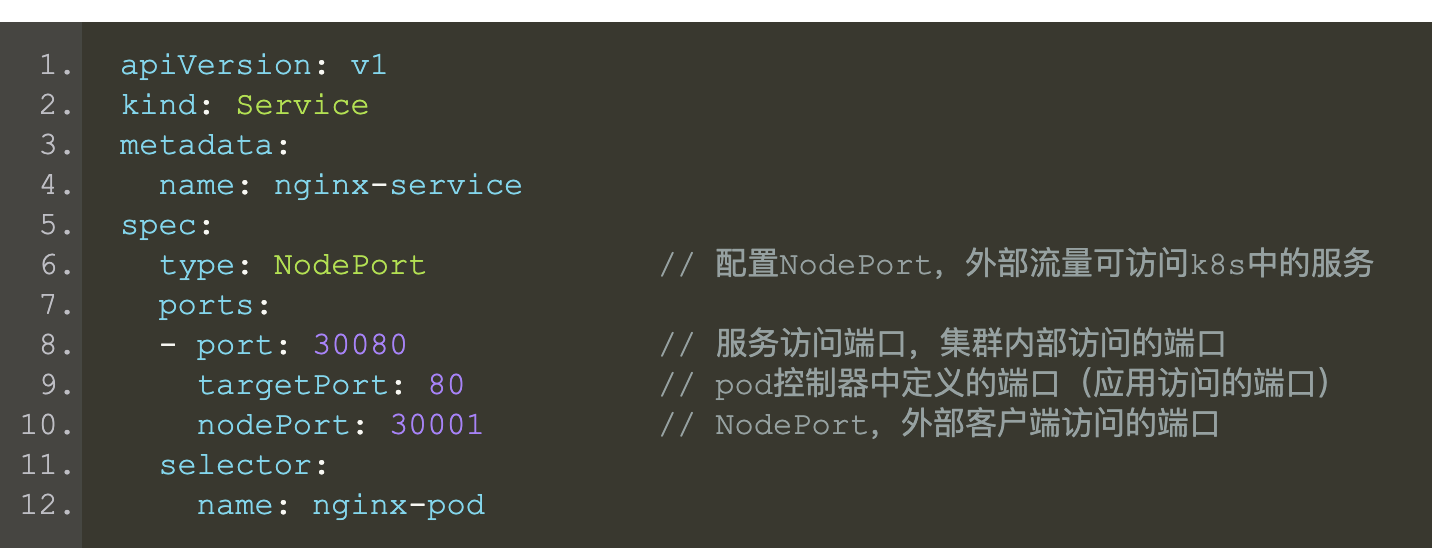
mysql容器暴露了3306端口（参考[DockerFile](https://github.com/docker-library/mysql/)），集群内其他容器通过33306端口访问mysql服务，但是外部流量不能访问mysql服务，因为mysql服务没有配置NodePort。对应的service.yaml如下：



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

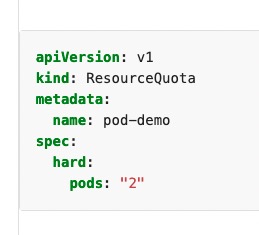
metadata - 帮助唯一性标识对象的一些数据，包括一个 name 字符串、UID 和可选的 namespace

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

## GVK GVR

GVK 就是 group、verison、kind

GVR 就是 group、version、resource

GVK是k8s的概念, 这三个玩意精准描述一个资源.

GVR是apiserver的http path的概念, 可以和GVK一一对应, 内部维护了一个mapper

## 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

wathc可以执行版本监听获取历史事件 而且这玩意是kv的 本质上只有新增 删除更新 启动的时候获取最新的就可以了

Refector1 连接是ApiServer1, Reflector 2连接的是ApiServer2, 当ApiServer1 收到一个update Pod 的请求时， Refelctor1 可以收到，但是这个请求没有发生在ApiServer2 上，如何让Reflector 2 也能感知到Pod 的event 呢？

这里的关键就是ETCD集群也可以有watch 机制，如果ApiServer1，写入ETCD，ApiServer2能够watch ETCD 的event的话，那就可以实现在ApiServer集群内部的Event 同步了。K8S 就是利用这个etcd机制来实现ApiServer 的同步，这对集群功能来说是至关重要的

## 监控

Client-go只能获取cpu和内存.

可以在集群里面装一个k8s-metrics的组件就可以监控了.

## 发布

连续执行以下命令，就可以实现先部署一个ｐｏｄ

kubectl set image deployment mynginxdeployment nginx=nginx:1.16 --record

kubectl rollout pause deployment mynginxdeployment

蓝绿发布

所谓蓝绿部署，是指同时运行两个版本的应用，

如上图所示，蓝绿部署的时候，并不停止掉老版本，而是直接部署一套新版本，等新版本运行起来后，再将流量切换到新版本上。但是蓝绿部署要求在升级过程中，同时运行两套程序，对硬件的要求就是日常所需的二倍，比如日常运行时，需要10台服务器支撑业务，那么使用蓝绿部署，你就需要购置二十台服务器。

滚动发布

所谓滚动升级，就是在升级过程中，并不一下子启动所有新版本，是先启动一台新版本，再停止一台老版本，然后再启动一台新版本，再停止一台老版本，直到升级完成，这样的话，如果日常需要10台服务器，那么升级过程中也就只需要11台就行了。

但是滚动升级有一个问题，在开始滚动升级后，流量会直接流向已经启动起来的新版本，但是这个时候，新版本是不一定可用的，比如需要进一步的测试才能确认。那么在滚动升级期间，整个系统就处于非常不稳定的状态，如果发现了问题，也比较难以确定是新版本还是老版本造成的问题

金丝雀(灰度)发布

灰度发布也叫金丝雀发布，起源是，矿井工人发现，金丝雀对瓦斯气体很敏感，矿工会在下井之前，先放一只金丝雀到井中，如果金丝雀不叫了，就代表瓦斯浓度高

在灰度发布开始后，先启动一个新版本应用，但是并不直接将流量切过来，而是测试人员对新版本进行线上测试，启动的这个新版本应用，就是我们的金丝雀。如果没有问题，那么可以将少量的用户流量导入到新版本上，然后再对新版本做运行状态观察，收集各种运行时数据，如果此时对新旧版本做各种数据对比，就是所谓的A/B测试

## 声明式API

K8s就是声明式api, 你告诉他最终的状态, 中间的各种流程由系统去处理.

命令式api就是中间各种流程你自己挨个去调用.