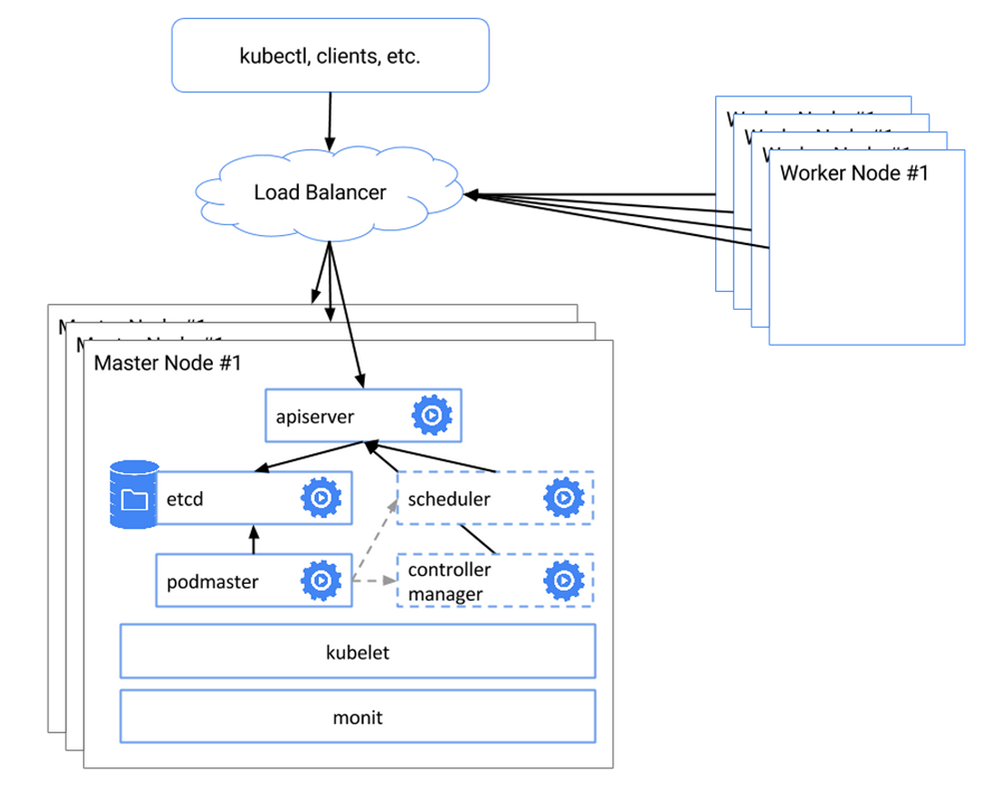
工作交接汇总

# 部署

kubernetes架构如下图所示



部署这个文档参考《kubernetes1.0实现.docx》文档。这个每个人应该都经历过一遍了。

## 文档

部署主要包括master集群的三个组件：apiserver、controller和scheduler，以及node节点上的kubelet。整体部署文档参考《kubernetes1.0实现.docx》。

## 脚本

### Master

Master的部署步骤可以参考上面的文档，涉及的脚本放在私有云打包环境上面。这个机器留着，相关脚本的目录和用途如下所示：

**/root/fangying/script/master**

etcd-initial-2.0.shetcd安装环境初始化

etcd-start-2.0.shetcd启动

master-initial-2.0.sh master集群安装初始化

master-start-2.0.sh master集群启动

nginx-initial-2.0.sh（nginx服务部署，可以选择用还是不用）

nginx-start-2.0.sh

脚本里面涉及的的安装包和文件在**/root/fangying/script/nos/v1.0.0/product/ kubernetes-1.0.0-product/**下面

apiserver-defaultkubectlkube-scheduler.manifest

controller-defaultkubeletpodmaster-auto.mainfest

docker-defaultkubelet-default schedule-policy.json

kube-apiserver.mainfestkubelet-init scheduler-default

kube-controller-manager.manifestkubernetes-default

相关文件的作用通过口述，因为大部分文件应该娄超都已经熟悉了

### Node

Node节点的初始化是通过做镜像的时候由长伟执行initial-in-building.sh脚本，然后在创建好node节点之后有nce调用nas脚本node-start-kubernetes.sh进行初始化。

这两个脚本的位置位于git仓库nas项目下

<https://git.hz.netease.com/nas/scripts-nce/tree/master/v2.0>

## 配置

**Master节点上的配置：**

Apiserver、controller和scheduler使用的配置集中在一下目录

**/var/run/kubernetes/：**

admission.conf证书生成相关配置

script证书生成相关工具

auth.csv授权配置

ca.crt根证书

server.crt服务端证书

server.key

token.csv授权配置

dockercfg镜像拉取key相关配置（弃用）

extend\_policy.csv扩展授权配置

keystone.cfgapiserver对接keystone配置

network.csv创建网络port资源zone相关配置

password.csv管理员账号相关配置

registrycfg镜像仓库相关配置

service.csv管理平台相关服务ip配置

**/etc/default/：**

kube-apiserverapiserver的启动配置

kubernetesapiserver、controller和scheduler的共有配置

kube-controller controller的启动配置

kube-scheduler scheduler的启动配置

**/etc/kubernetes/manifests/：**

kube-apiserver.mainfestapiserver的启动描述文件

kube-controller-manager.manifestcontroller的启动描述文件

kube-scheduler.manifest scheduler的启动描述文件

**/etc/kubernetes/scheduler/：**

schedule-policy.json调度器的调度策略配置

**node上的配置**：

/var/run/kubernetes:kubelet启动认证相关的文件

/etc/default/kubelet:kubelet的启动配置

## 打包

打包环境在私有云机器上：

目录：**/root/fangying/images/product/kubernetes**

kube-apiserverapiserver二进制文件

kube-scheduler scheduler二进制文件

kube-controller-managercontroller二进制文件

kube-start 镜像需要的相关文件

Dockerfile-apiserver构建apiserver镜像的dockerfile

Dockerfile-controller 构建controller镜像的dockerfile

Dockerfile-scheduler构建scheduler镜像的dockerfile

将新编译的二进制文件放到这个目录下面，分别build对应的镜像，然后push到镜像仓库就可以了。

# 功能

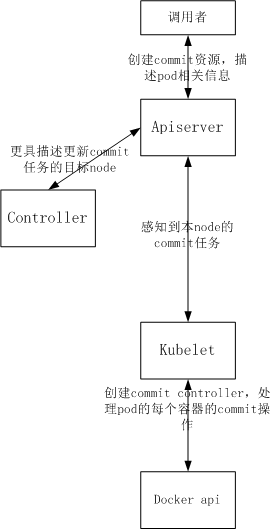
## 保存镜像

### 描述

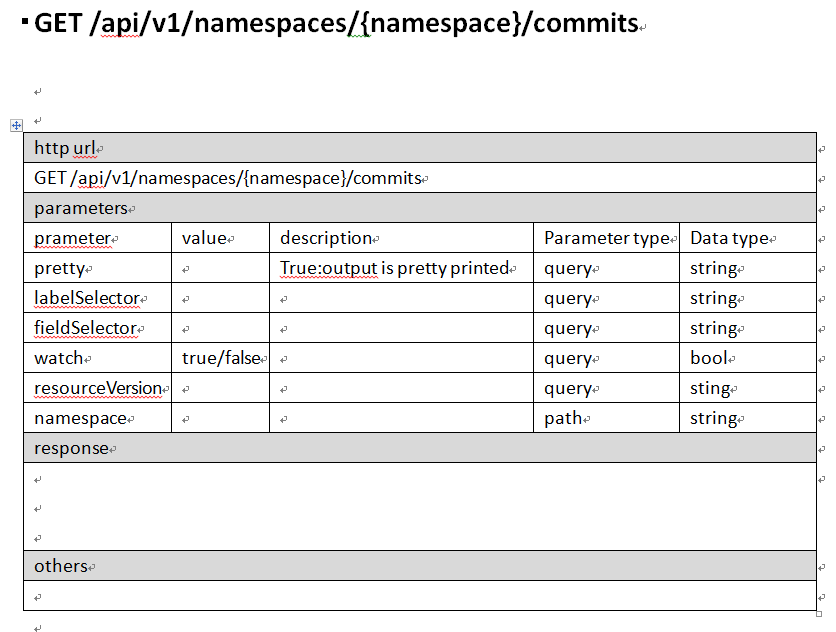
把正在运行的容器保存为镜像。

### 概要设计

K8s新增了commits资源，调用方根据说明在资源中描述需要保存镜像的容器的相关信息，包括容器id等，以及commit的策略。控制器会通过commits的描述找到对应的pod，并记录对应的node信息，并更新commits状态。Kubelet通过watch commits资源的变化获知相关的任务，每一个commit任务，kubelet创建一个控制器，根据commit提供的策略完成容器镜像的保存，如果一个pod包含了多个容器，commit 控制器会顺序保存各个容器，并单独更新commits的状态。大致流程如下图所示：



### Api接口



如图类似，详情请参考**《commit api.docx》以及《Commit镜像到镜像仓库》**

### 关键数据结构



如图所示，详情参考commit\_struct.go

### Commit 描述

{

"kind": "Commit",

"apiVersion": "v1",

"metadata": { //这个属性每个api都有

"name":"",

"selfLink":"",

"uid":"",

"resourceVersion":"",

"creationTimestamp":"",

"labels": {

"key":"value"

}

},

"spec":{

"images":[ //想要传的镜像，可以有多个，因为一个pod跑的多个container。现阶段我们的服务是一个pod对应一个container？如果是的话，只有一个

{

"containerName":"", //容器名称

"from":"", //容器镜像id

"image":"", //要保存为的容器镜像id

"imagePushPolicy":"ExistFail" // 如果镜像库中已存在，就失败

},

{

"containerName":"",

"from":"",

"image":"",

"imagePushPolicy"

},

],

"restartPolicy":"OnFaliure", //还可以选"Never"

"podName":"", //pod的名称

"nodeName":"",

"imagePullSecrets":[ //镜像库的secret

"name":""

],

},

"status": {

"containerCommitStatuses":[ // 各个commmit的状态

{

"containerName":"",

"imageID":"", //保存的镜像id

"phase":"Pending", //commit的状态

"failCount":"", //保存失败的次数

},

{

"containerName":"",

"imageID":"",

"phase":"Pending",

"failCount":"",

}

]

}

}

### Commit状态

const(

    CommitPendingCommitPhase="Pending"

    CommitRunningCommitPhase="Running"

    CommitPushingCommitPhase="Pushing"

    CommitSucceededCommitPhase="Succeeded"

    CommitFailedCommitPhase="Failed"

)

Pending状态表示commit已经创建出来，但是kubelet还没有开始处理

Running状态表示kubelet已经感知到，并撞见了commiter开始处理

Pushing状态表示kubelet正在往镜像仓库推送保存的镜像

Successed状态表示保存成功

Failed状态表示保存失败

## 认证授权扩展

### 描述

原有的认证授权机制支持从文件配置token，但是很单一不能更改。对于角色的定义也过于单调，而且不适合做分层的授权机制。所以急需完善apiserver的认证授权机制，满足一下的需求：

1. 能够动态的增删用户
2. 能够对用户权限时效性进行管理
3. 支持层级的权限管理机制

### 概要设计

由于蜂巢底层使用了iaas，为了便于和iaas整合，特别是用户管理方面进行打通，k8s使用了iaas的keystone进行授权管理。好处在于，整个蜂巢的账号体系可以独立并且统一，同时可以减少必要的开发和维护。具体的设计参考《使用keystone扩展apiserver认证授权》。

## 一对一

### 描述

一对一的特性旨在保证一个容器运行在一个node节点上，主要应用在容器上。同时使用云主机保持容器的数据，保证数据不会丢失。需求点：

1. 一个容器调度到一个node上
2. 容器挂掉之后需要保证数据不丢失
3. Node节点挂掉之后不要进行迁移

### 概要设计

主要涉及控制器和kubelet模块。使用的仍然是rc，rc的基本功能就是在挂掉之后重新创建容器或者在node离线之后删除pod，然后重新创建pod。所以对比rc controller来说，要实现一对一的特性，控制器需要做到：

1. rc创建出pod之后不能在rc没被删除的前提下删除老的pod
2. node controller在node失效的情况下超时后不能对此类rc的pod删除和重新调度

另外需要保证的就是node不会被重复使用，哪怕在pod被人为误删除的情况下，调度器的工作原理是检查一个node上是否有被调度的容器，而这个过程是通过查看pod中node name来计算的，误删除的情况下很容易出现node被误认为是空闲的。所以需要在调度之后给node上个锁，这个锁通过node的label来实现，在node上打上一个被pod占用的标签。例如{“occupy”:”podname”}。

调度完成之后kubelet会运行容器，kubelet对于pod运行的容器的基本管理方法是以镜像为基础，当容器挂掉之后，重新根据镜像创建一个新的容器，新的容器将会丢失老容器运行过程中的数据。同时kubelet面临着随时被重启或者机器重启，本地缓存会被释放，所以kubelet需要做到：

1. 识别一个pod是否为新的pod或者之前运行过容器，进行区别处理
2. 容器挂掉之后，选择重新运行老的容器而不是新建
3. 在服务重启或者node重启下能够保证恢复老的容器

K8s一对一新建流程如下：

1. apiserver创建rc，写入etcd
2. rc控制器检测pod副本数，创建所需pod
3. 调度器调度pod，检查pod类型，为一对一调度的目标node打上occupy标签
4. Kubelet根据新的pod创建容器，成功后更新container id到pod 的containerID

容器挂掉之后处理流程：

1. kubelet根据pod描述检测是否一对一类型，是否已经创建过container
2. 如果已有container则检测本地是否存在，存在则重启

一切在一堆一模式下的异常都不会触发新建流程

Node挂掉之后处理流程：

1. node controller检测到node失联超时，检测其上运行的pod类型是否为一对一类型
2. 一对一类型不删除pod，不迁移pod，非一对一删除老的pod，新pod迁移到新的节点

### 关键数据结构

**const**(

    StartNewStartPolicy="New"

    StartScheduleOldStartPolicy="ScheduleOld"

 StartUseOldStartPolicy="UseOld"

)

const(

    ScheduleShareSchedulePolicy="Share"

    ScheduleMonopolySchedulePolicy="Monopoly"

)

StartPolicy用于表明容器的启动类型，New为原生模式表示失败创建新的容器，

ScheduleOld表示使用云硬盘模式保持容器数据，UseOld表示使用云主机模式保持容器数据

SchedulePolicy表示资源调度模式，Share表示原生的共享型资源调度策略，Monopoly表示一堆独占型资源调度策略

typePodSpecstruct{

    SchedulePolicySchedulePolicy`json:"schedulePolicy,omitempty"description:"schedulepolicyofpodforscheduler;oneof'Share'or'Monopoly'"`

    //Optional:Setstartpolicy.Defaultsto"New".otherScheduleOldUseOld

StartPolicyStartPolicy`json:"startPolicy,omitempty"description:"startpolicyforpodwhenkubeletfindit;oneof'New''ScheduleOld'or'UseOld'"`

}

type Container struct{

    //Optional:ifpodwanttouseoldcontainer,thisisrequired

ContainerIDstring`json:"containerID,omitempty"`

}

ContainerId用于记录pod创建的容器的id，用于数据保持。

## 重启

### 描述

在kubernetesapi端提供操作容器重启、暂停等功能。Docker的工具和api支持直接对容器的重启等操作。Kubernetes是一个集群管理系统，不支持对单个容器进行此类操作，需要增加api来支持其管理的node上运行的容器的此类docker操作接口。

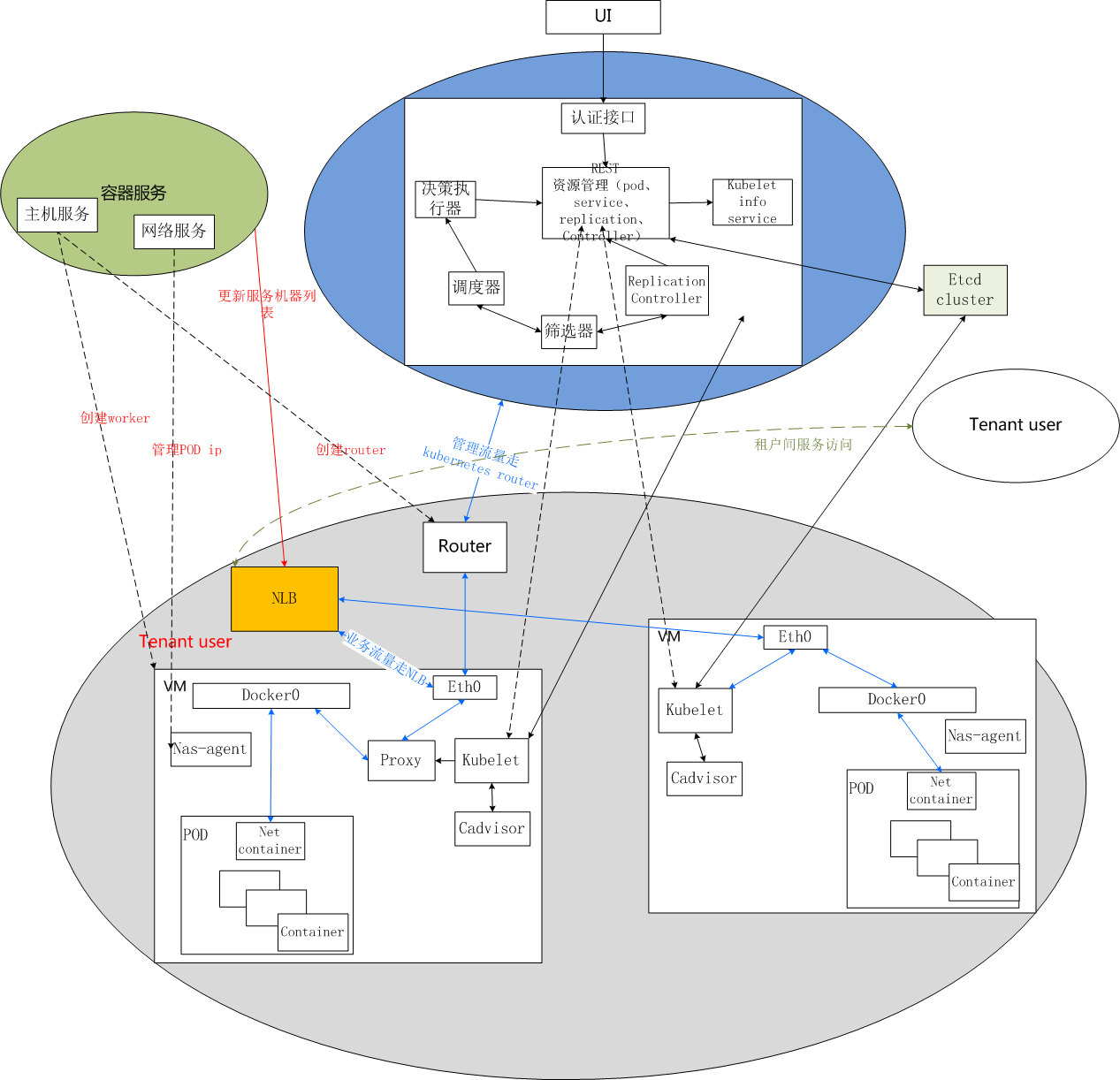
### 概要设计

通过在kubernetesapi端暴露接口，然后根据pod信息查找到container运行的node节点，再将重启等命令通过kubelet运行的代理服务发送给node上运行的dockerdamon。详细的设计和数据结构参考《容器重启等管理操作.docx》。

## Apiserver安全加固

### 描述

apiserver安全加固旨在提供一个安全的服务编排服务，保障用户数据、访问安全性。apiserver提供服务主要通过两个端口：启用https的6443端口和非安全的http端口8080.而etcd则通过非安全的客户端口进行访问。在node节点上，kubelet运行在根namespace中，用户的容器运行在其他namespace中，用户数据存储在同一个etcd。访问的结构图如图所示：



新增的磁盘和网络部分功能，需要在node节点上访问云硬盘和云网络提供的接口。

要保证集群的安全性，主要考虑一下几个问题：

1.etcd访问的安全性

2.apiserver非安全接口的防护

3.不同租户访问的安全性

4.node节点被穿透后集群的安全性

### 概要设计

针对以上问题，解决问题的方法如下：

1.etcd访问的安全性

理论上etcd应该使用https进行访问，但是作为集群内部的成员，用户无法直接访问，而且使用http效率更高。线上集群都使用nce管理网络进行访问，管理网络通常位于一个段，也会有不同的段，在etcd和apiserver上使用同一个段很容易在node节点上经过试探特定的etcd端口检测到etcd服务的正确访问地址，对于未加防护的etcd来说，无疑是一种灾难。因此apiserver和node节点使用的网络和apiserver访问etcd使用的网络需要隔离，etcd使用的服务端口也许改变，尽量不使用默认的值，etcd只监听在特定的服务ip上。

2.apiserver非安全接口访问的安全性

apiserver的非安全接口应该拒绝除了本地访问之外的其他访问来保证其安全性，所有外部访问需走安全访问端口，并启用认证授权机制。

3.不同租户访问的安全性；node节点被穿透后集群的安全性

扩展原有的老的认证授权机制，每一个租户采用自己的一套证书，这个证书在namesapce（租户）初始化的时候由apiserver签发并保存到nos上，每一个租户的node节点创建出来初始化的时候从nos下载证书，并与apiserver建立连接。生成证书的信息参考《安全加固-为每个租户生成证书》

每一个apiserver签发的证书，将会包含租户的名称，apiserver以此作为认证授权的基础。每个租户被限制访问租户自己的资源，无法访问其他租户的资源，并且可以根据不同的访问角色配置对不同资源的访问限制，这些配置通过apiserver的配置文件extend\_policy.csv来完成。配置示例如下：

{"ruleName": "kubelet", "resource": "secrets", "namespace": "self", "readonly": true}

{"ruleName": "kubelet", "resource": "services", "readonly": true}

{"ruleName": "kubelet", "resource": "endpoints", "readonly": true}

{"ruleName": "kubelet", "resource": "pods", "namespace": "self"}

{"ruleName": "kubelet", "resource": "pods", "readonly": true}

{"ruleName": "kubelet", "resource": "nodes", "namespace": "self"}

{"ruleName": "kubelet", "resource": "nodes"}

{"ruleName": "kubelet", "resource": "commits"}

{"ruleName": "kubelet", "resource": "events"}

{"ruleName": "kubelet", "resource": "persistentvolumes", "namespace": "self", "readonly": true}

{"ruleName": "kubelet", "resource": "netendpoints"}

对于卷和网络资源的操作分为两类，一是由master（apiserver、controller、scheduler）发起的请求，二是由node节点（kubelet）发起的请求。master发起的请求包括创建、删除和查询详情等操作，node节点发起的操作包括挂载、卸载和查询详情等操作。master和node节点处于不同的安全环境，master被认为是安全的，node节点认为是非安全的，用户可能通过内核bug穿透namespace从而获取系统信息。为了保证集群的访问安全性，master上发起的操作通过机器的白名单和kestone的服务角色的token进行访问，而node节点通过租户自己的token进行访问，租户自己的token被限制只能进行挂载、卸载和查询操作，而且只能针对租户自有的资源。node节点使用的token保存在租户namespace下的secret里，controller在初始化租户的时候为每个租户创建了secret资源iaastoken，里面保存了租户在node上可以使用的访问卷和网络的keystone的token。通过kubectl命令kubectl get secret iaastoken --namespace=${tenantid} -o json 查看对应租户的token。

## 镜像认证管理

### 描述

每一个租户有自有的私有仓库，访问时通过自有的密码拉取自己的镜像。用户创建pod时，必须提供必要的授权才能拉取镜像。

### 概要设计

docker在拉取用户私有仓库时，需要docker login，然后会生成一个config.json，里面包含了访问不同镜像仓库需要的密码等认证信息。其格式如下所示：

xxx

pod支持在其描述中采用secret的方式提供下载pod所需镜像而使用的docker config文件。kubelet在访问镜像仓库时需要获取该config 文件以调用docker的接口pull或者push镜像，这意味着kubelet必须知道用户的密码才能进行，而用户的密码等信息更适合保存在用户管理中心而不是k8s集群中，因此需要一种机制来帮助用户通过访问私有仓库的认证操作。在蜂巢系统中，拉取一个镜像的过程大致如下，拉取镜像的一方将密码信息通过docker接口调用registry仓库，仓库将用词密码调用管理平台进行认证，所以能否通过认证取决于管理平台。因此在不持有用户真正的密码的前提下，由管理平台为每一个用户生成一个操作镜像的特定的虚拟密码，而该密码只能进行镜像的管理操作而不能用于用户其他的操作。k8s可以通过持有该虚拟密码管理用户的镜像仓库而不会造成其他危害。具体做法是在初始化租户环境时，k8s 控制器调用管理平台的接口获取管理平台为该租户生成的虚拟密码信息，并创建secret资源registrykey以保存密码信息，pod在pull和push镜像时通过registrykey获取密码进行访问。registrykey的信息格式如下：

{

"kind": "Secret",

"apiVersion": "v1",

"metadata": {

"name": "registrykey",

"namespace": "64c3a4cd02254dee8350928d398bbda2",

"selfLink": "/api/v1/namespaces/64c3a4cd02254dee8350928d398bbda2/secrets/registrykey",

"uid": "c2cfcd65-023a-11e6-96be-246e960505ec",

"resourceVersion": "399583540",

"creationTimestamp": "2016-04-14T12:16:48Z"

},

"data": {

".dockercfg": "eyJhdXRocyI6eyJodWIuYy4xNjMuY29tIjp7ImF1dGgiOiJjWEV4TVRrMU1qQTVOVGs1UURFMk15NWpiMjA2VDFaVk5HNUhWR1ZNWkhGayIsImVtYWlsIjoiIn19fQ=="

},

"type": "kubernetes.io/dockercfg"

}

.dockercfg的内容用base64解开为

{"auths":{"hub.c.163.com":{"auth":"cXExMTk1MjA5NTk5QDE2My5jb206T1ZVNG5HVGVMZHFk","email":""}}}

这个和docker login后的本地config.json文件是一致的，保存了每一个镜像仓库的访问凭证。

## 使用云硬盘数据保持

### 描述

使用云硬盘进行数据保持分为两个部分：在k8s集群里面支持使用云硬盘；通过云硬盘进行数据保持。

支持使用云硬盘为一通用机制，而使用其进行数据保持为该功能的一个特例化用例，该功能可以为每一个容器提供采用-v方式映射的普通卷。而使用云硬盘进行数据保持为数据保持的其中一种方案，在使用本地盘进行数据保持的章节已经介绍了其接口。

### 概要设计

使用云硬盘特例化用于数据保持的方案参考《数据分离设计方案.docx》。

而使用云硬盘作为普通的卷的使用方式如下，只需在pod的描述中声明使用netease类型的卷即可。

## 外网管理

### 描述

每一个pod的容器具有两种网络：私有网和外网，私有网通过在node节点上的网桥连接，外网则每一个容器使用一个iaas层的外网port。外网的ip和mac需要保持严格的一致，否则无法通过云网络的安全限制导致网络访问失败，每一个pod的容器私有网是必须的，而外网则由用户自己选择是否使用，选择使用外网则需要在用户容器的整个生命周期保持外网不便，即便在容器发生迁移的情况下。

### 概要设计

为了方便对外网资源进行管理，新增加了一种资源NetEndpoint用于描述外网的详细信息，包括ip、mac以及路由等信息，详细的信息如下所示：

{

"kind": "NetEndpoint",

"apiVersion": "v1",

"metadata": {

"name": "three-253952-4d9a03c4-3sjpt",

"generateName": "three-253952-4d9a03c4-",

"namespace": "f2bd1478661e4e79bdc4b2d20f76fda4",

"selfLink": "/api/v1/namespaces/f2bd1478661e4e79bdc4b2d20f76fda4/netendpoints/three-253952-4d9a03c4-3sjpt",

"uid": "84eb7ebc-1a62-11e6-910b-246e960505ec",

"resourceVersion": "678926708",

"creationTimestamp": "2016-05-15T06:01:52Z",

"annotations": {

"kubernetes.io/created-by": "{\"kind\":\"SerializedReference\",\"apiVersion\":\"v1\",\"reference\":{\"kind\":\"ReplicationController\",\"namespace\":\"f2bd1478661e4e79bdc4b2d20f76fda4\",\"name\":\"three-253952-4d9a03c4\",\"uid\":\"8345b0e8-1a62-11e6-910b-246e960505ec\",\"apiVersion\":\"v1\",\"resourceVersion\":\"495582408\"}}"

}

},

"spec": {

"capacity": null,

"type": "Wan",

"neteasePort": {

"portID": "f5d783cd-e433-4da8-907b-672ce2bf77a3",

"tenantID": "f2bd1478661e4e79bdc4b2d20f76fda4",

"ip": "106.2.109.181",

"mac": "fa:16:3e:a6:c1:c7",

"route": [

"0.0.0.0/1 via 106.2.108.1 dev eth2",

"128.0.0.0/1 via 106.2.108.1 dev eth2"

],

"mask": "22"

}

},

"status": {

"podName": "three-253952-4d9a03c4-nlx1e",

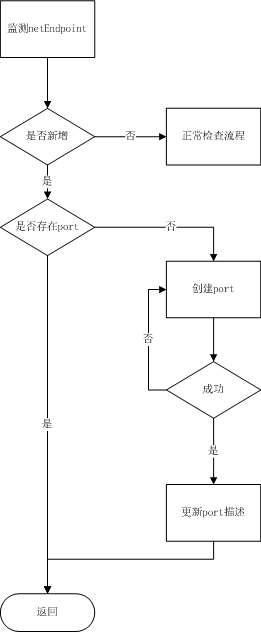
"phase": "Bound"

}

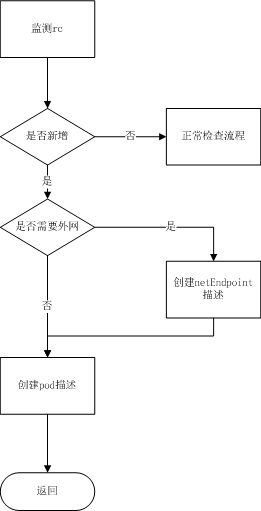
}

外网资源的控制和其他k8s资源一样，api和接口详见文档《netEndpoint api说明》

rc控制器在创建pod的时候，会根据rc中对于网络资源的需求判断当前pod是否需要使用外网，如果需要则会在创建pod之前创建外网资源，外网资源的命名和rc保持相同的前缀，只是后五位名称通过随机生成。外网（netEndpoint）资源由一个控制器进行管理，rc创建了netEndpoint资源会被控制器watch到，netEndpoint controller会根据里面的描述进行处理，例如如果发现netEndpoint为Pending状态，并且port id等为空，则会调用cinder接口创建一个外网port，根据返回的信息填充netEndpoint资源。



在创建好netEndpoint资源后，rc控制器继续创建pod资源，并将netEndpoint的名称记入pod的描述当中。pod资源创建完成之后可以通过kubectl看到pod出去Pending状态，接下来scheduler会watch到新建的pod资源进行调度，一旦调度成功，pod会处于Running状态，pod描述里面会有被调度的node的信息，此时调度器会调用cinder接口将pod里面记载的netEndpoint的port挂载到pod被调度的node上（由调度器来挂载只是一个临时方案，管理平台没有封装网络相关的接口，理论上由kubelet自己进行挂载操作）。



之后kubelet会watch到被调度给自己的pod并进行创建容器的初始化操作，每一个pod在node节点上都会具备两个以上的容器，其中一个pause容器，其余容器才是pod描述里面定义的容器。pause容器的意义在于创建一个网络namespace，初始化网络，pod里面其他的容器和pasue容器共享网络namespace。kubelet创建pod的容器之前会先创建pasue容器，创建pause容器的过程中会调用一个plugin，这个plugin实现了网络操作的相关接口，包括配置ip和路由等操作。plugin的实现详见代码。

一旦出现node宕机等原因导致pod进行迁移，controller的node controller会擦掉pod的node信息同时从node上拔出外网port，没有node信息的pod将会被重新调度，从而重复上面的流程。pod的外网进行迁移保证了用户容器在生命周期内能够保持ip恒定。

## 快照

### 描述

快照分为4部分内容：

1.对一个容器的卷做快照，并保存快照信息

2.回滚到一个老的卷或者快照

3.扩大当前卷的容量

4.从回收站回复老的容器

### 概要设计

详细设计参考《快照》和《回滚》

## 流控扩展

丰富化的流控策略利于改善失败率，区分api的优先级，提供用户体验和线上维护更加便利。

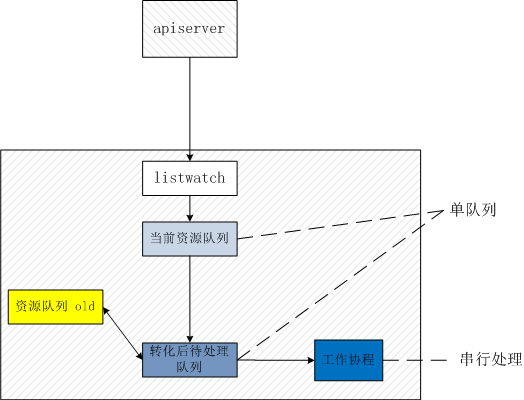
此部分由李岚清刚开发完，由他来补充文档吧。

# 性能

## Smart queue

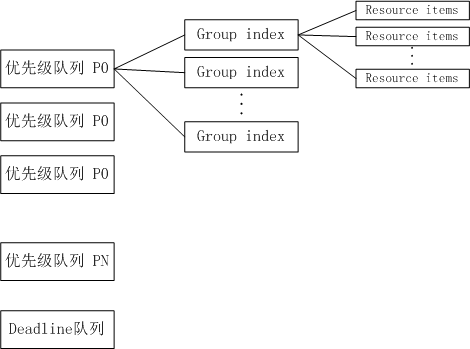
### 背景

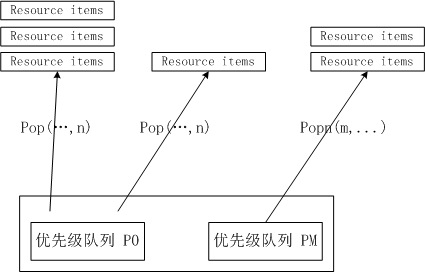
k8s里面很多地方使用到了队列，特别是在控制器和调度器当中。然而大多数的模型都是串行fifo处理，导致处理速度较慢，同时在数据量大的情况下回淹没优先级更高的创建等操作，鉴于此，实现一个更加通用的多优先级deadline队列相当必要。



### 设计

队列的设计如下图所示：

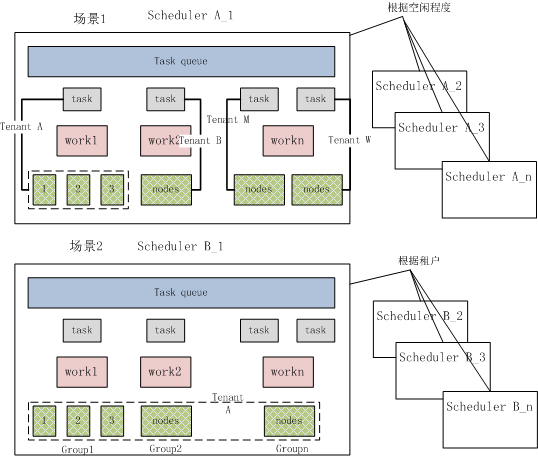




不同的优先级在不同的队列，每一个优先级队列通过一个group index来保存同一个group的resource，这里group可以特例化到namespace。除了优先级队列之外，还具备一个deadline队列。优先级队列的优先级个数由自己确定，在现有的线上环境中，优先级则分为了create、update和delete三种操作。每一个处理的worker可调用队列的接口从中弹出一个待处理的任务，队列弹出任务的顺序为按照优先级顺序弹出，在上一个优先级队列为空时才会弹出下一个优先级队列的任务，除非下一个优先级队列的任务触发了deadline。同一个优先级队列下，每次可以弹出一个group的一个或者多个任务，但是在同一个group的上一次弹出的任务没有处理完成，队列不会继续弹出同一个group的其他任务，除非worker处理完成之后调用队列的接口通知队列该group的任务已经完成，该group才会被设置为可以调度，这样是为了防止多个worker处理同一个group下任务造成的竞争。

### 用途

控制器、调度器



## Apiserver

apisever上的主要优化为使用cache、采用多etcd以及更加细致的流控策略

### Cache

在apiserver和etcd之间加入一层cache，减轻etcd的压力，目前的cache为开源的版本，可以缓存watch的请求。开源的实现还有部分不太完美的地方，目前已经在蜂巢的版本上进行了小范围优化。当然要做到缓存get等请求还需要进一步的设计和修改。

详细的内容可以参考开源或者小强的文档。

### 多etcd

鉴于所有资源存放到一个etcd上，压力太大，不便于管理。对资源进行了分类，存放于不同的etcd之上，目前资源压力最大的是pod和node，所以目前的拆分是pod和node资源单独各自一个etcd存储，其他的存储到一个etcd。后续event也需要拆出来。

在etcd 3.0性能提升以及完善cache的基础上，etcd的进一步扩展可以调低优先级。

### 流控

目前的流控策略过于简单，无法对重点的访问者和重点的操作进行区分，统一的流控总是让更多优先级低的操作挤占了优先级更高的操作，导致失败。

精细的流控策略旨在根据http header的访问者的标识和资源的分类对不同的访问者根据资源和访问类型进行流控，从而保证重点操作可以顺利完成，减小失败率。也利于升级维护的时候更好的控制集群的平稳性。

## 控制器

控制器涉及到的优化主要有rc和node controller。

rc controller使用了多优先级队列以及多并发处理，大大提高了处理速度和用户体验。详细的请参考前面的队列和代码进行理解。

node controller则是通过调整kubelet的管理方式，由kubelet汇报方式改为node controller主动查询方式，这样利于控制在kubelet重启过程中对etcd造成的冲击，便于线上维护以及水平扩展node controller。

## 调度器

使用了多优先级队列和多并发优化调度器。详细参考《xxx》

## Node

### 初始化时间优化

修改了初始化脚本，将部分操作移到打镜像的时候进行，同时对部分操作进行了并行化。这个以现在的版本为准，读脚本就可以了。

up-initial.sh为打镜像时执行的脚本。

node-start-kubernetes.sh为初始化node时调用的脚本。

### 主动注册优化

nce服务管理将初始化node的一些必要的信息比如label等传递给初始化脚本，脚本将信息写入临时文件中。kubelet在启动过程中会从临时文件中获取相关信息，同时从云主机提供的.info文件中获取租户等信息，通过主动汇报的方式向apiserver注册自己。已达到加快初始化过程的目的。