**加我微信可进群学习交流：**

**微信号：**

luckylucky421302

也可通过扫描下面二维码添加



**课程更新的知识点会通过微信公众号免费分享给大家，可以关注我的公众号**

****

k8s对我们整个系统的认证，授权，访问控制做了精密的设置；对于k8s集群来说，apiserver是整个就集群访问控制的唯一入口，我们在k8s集群之上部署应用程序的时候，也可以通过宿主机的NodePort暴露的端口访问里面的程序，用户访问kubernetes集群需要经历如下认证过程：认证->授权->准入控制（adminationcontroller）

1.认证(Authenticating)是对客户端的认证，通俗点就是用户名密码验证

2.授权(Authorization)是对资源的授权，k8s中的资源无非是容器，最终其实就是容器的计算，网络，存储资源，当一个请求经过认证后，需要访问某一个资源（比如创建一个pod），授权检查都会通过访问策略比较该请求上下文的属性，（比如用户，资源和Namespace），根据授权规则判定该资源（比如某namespace下的pod）是否是该客户可访问的。

3.准入(Admission Control)机制是一种在改变资源的持久化之前（比如某些资源的创建或删除，修改等之前）的机制。

k8s的整体架构也是一个微服务的架构，所有的请求都是通过一个GateWay，也就是kube-apiserver这个组件（对外提供REST服务），k8s中客户端有两类，一种是普通用户，一种是集群内的Pod，这两种客户端的认证机制略有不同，但无论是哪一种，都需要依次经过认证，授权，准入这三个机制。

**一、认证**

**认证支持多种插件**

（1）令牌（token）认证：

双方有一个共享密钥，服务器上先创建一个密码，存下来，客户端登陆的时候拿这个密码登陆即可，这个就是对称密钥认证方式；k8s提供了一个restful风格的接口，它的所有服务都是通过http协议提供的，因此认证信息只能经由http协议的认证首部进行传递，这种认证首部进行传递通常叫做令牌；

（2）ssl认证：

对于k8s访问来讲，ssl认证能让客户端确认服务器的认证身份，我们在跟服务器通信的时候，需要服务器发过来一个证书，我们需要确认这个证书是不是ca签署的，如果是我们认可的ca签署的，里面的subj信息与我们访问的目标主机信息保持一致，没有问题，那么我们就认为服务器的身份得到认证了，k8s中最重要的是服务器还需要认证客户端的信息，kubectl也应该有一个证书，这个证书也是server所认可的ca签署的证书，双方需要互相认证，实现加密通信，这就是ssl认证

**二、授权**

如果用户通过认证，那么什么权限都没有，需要一些后续的授权操作，如对资源的增删该查等，kubernetes1.6之后开始有RBAC（基于角色的访问控制机制）授权检查机制。

**三、准入控制**

一般而言，准入控制只是用来定义我们授权检查完成之后的后续的其他安全检查操作的，进一步补充了授权机制，由多个插件组合实行，一般而言在创建，删除，修改或者做代理时做补充；

**第一部分、认证**

**kubernetes上的账号**

客户端对apiserver发起请求，apiserver要识别这个用户是否有请求的权限，要识别用户本身能否通过apiserver执行相应的操作，那么需要哪些信息才能识别用户信息来完成对用户的相关的访问控制呢？

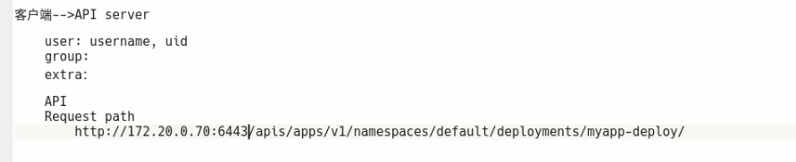
kubectl explain pods.spec可以看到有一个字段serviceAccountName（服务账号名称），这个就是我们pod连接apiserver时使用的账号，因此整个kubernetes集群中的账号有两类，ServiceAccount（服务账号），User account（用户账号）

User account：实实在在现实中的人，人可以登陆的账号，客户端想要对apiserver发起请求，apiserver要识别这个客户端是否有请求的权限，那么不同的用户就会有不同的权限，靠用户账号表示，叫做username

ServiceAccount：方便Pod里面的进程调用Kubernetes API或其他外部服务而设计的，是kubernetes中的一种资源

**sa账号：**登陆dashboard使用的账号

**user account：**这个是登陆k8s物理机器的用户



kubectl客户端拿着token进行认证，然后把请求发送给https协议

**1.ServiceAccount**

Service account是为了方便Pod里面的进程调用Kubernetes API或其他外部服务而设计的。它与User account不同，User account是为人设计的，而service account则是为Pod中的进程调用Kubernetes API而设计；User account是跨namespace的，而service account则是仅局限它所在的namespace；每个namespace都会自动创建一个default service account；

开启ServiceAccount Admission Controller后

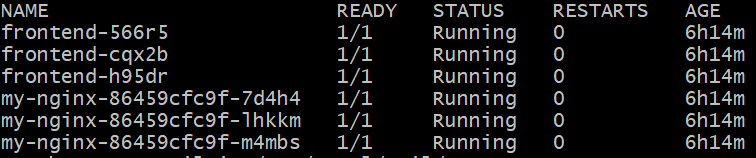
1）每个Pod在创建后都会自动设置spec.serviceAccount为default（除非指定了其他ServiceAccout）

2）验证Pod引用的service account已经存在，否则拒绝创建；

3）如果Pod没有指定ImagePullSecrets，则把service account的ImagePullSecrets加到Pod中

当创建 pod 的时候，如果没有指定一个 serviceaccount，系统会自动在与该pod 相同的 namespace 下为其指派一个default service account。而pod和apiserver之间进行通信的账号，称为serviceAccountName。如下：

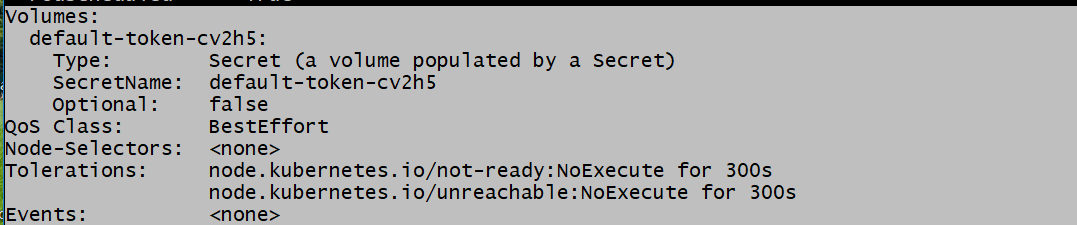
**kubectl get pods**



**kubectl get pods frontend-4tbz4 -o yaml | grep "serviceAccountName"**



**kubectl describe pods frontend-4tbz4**

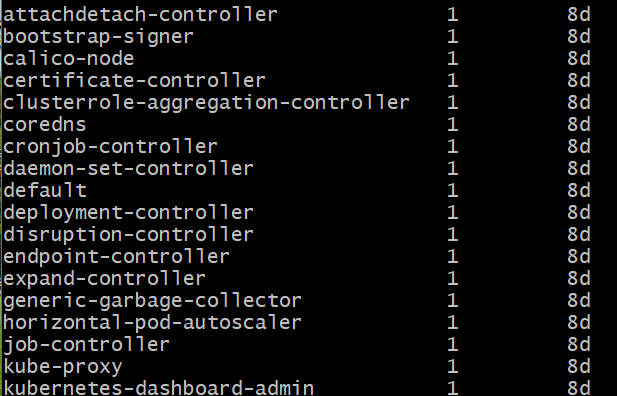


从上面可以看到每个Pod无论定义与否都会有个存储卷，这个存储卷为default-token-\*\*\*的 token令牌，这就是pod和serviceaccount认证信息。通过secret进行定义，由于认证信息属于敏感信息，所以需要保存在secret资源当中，并以存储卷的方式挂载到Pod当中。从而让Pod内运行的应用通过对应的secret中的信息来连接apiserver，并完成认证。每个 namespace 中都有一个默认的叫做 default 的 serviceaccount 资源。进行查看名称空间内的secret，也可以看到对应的default-token。让当前名称空间中所有的pod在连接apiserver时可以使用的预制认证信息，从而保证pod之间的通信。

**kubectl get sa**



**kubectl get sa -n kube-system**



**kubectl get secret**



**kubectl get secret -n kube-system**

默认的service account 仅仅只能获取当前Pod自身的相关属性，无法观察到其他名称空间Pod的相关属性信息。如果想要扩展Pod，假设有一个Pod需要用于管理其他Pod或者是其他资源对象，是无法通过自身的名称空间的serviceaccount进行获取其他Pod的相关属性信息的，此时就需要进行手动创建一个serviceaccount，并在创建Pod时进行定义。那么serviceaccount该如何进行定义呢？实际上，service accout也属于一个k8s资源，serviceAccount也属于标准的k8s资源，可以创建一个serviceAccount，创建之后由我们创建的pod使用serviceAccountName去加载自己定义的serviceAccount就可以了，如下查看service account的定义方式：

**（1）创建一个serviceaccount：**

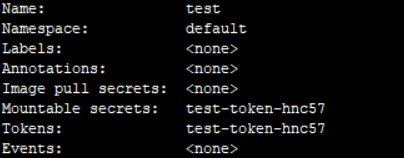
kubectl create serviceaccount test

kubectl get sa 显示如下：



可以看到已经创建了test的serviceacount

kubectl describe sa test 查看test这个账号的详细信息

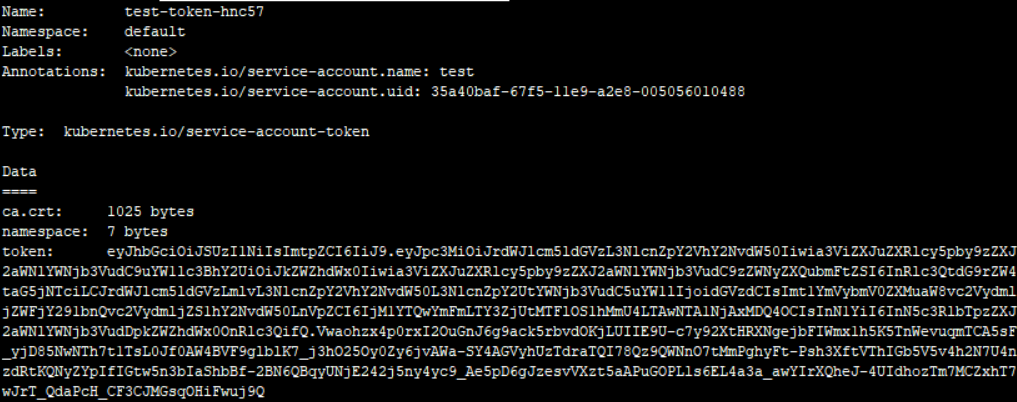


上面可以看到生成了一个test-token-hnc57的secret和test-token-hnc57的token

kubectl get secret 显示如下：



kubectl describe secret test-token-hnc57 显示如下：



上面可以看到生成了test-token-hnc57的token详细信息，这个账号就是sa连接apiserver的账号，这个token是登陆k8s的token，这些是一个认证信息，能够登陆k8s，能认证到k8s，但是不能做别的事情，不代表权限，想要做其他事情，需要授权

**（2）kubeconfig文件**

在K8S集群当中，每一个用户对资源的访问都是需要通过apiserver进行通信认证才能进行访问的，那么在此机制当中，对资源的访问可以是token，也可以是通过配置文件的方式进行保存和使用认证信息，可以通过kubectl config进行查看配置，如下：

**kubectl config view**

apiVersion: v1

clusters:

- cluster:

certificate-authority-data: DATA+OMITTED

server: https://192.168.80.199:6443 #apiserver的地址

name: kubernetes #集群的名字

contexts:

- context:

cluster: kubernetes

user: kubernetes-admin

name: kubernetes-admin@kubernetes #上下文的名字

current-context: kubernetes-admin@kubernetes #当前上下文的名字

kind: Config

preferences: {}

users:

- name: kubernetes-admin

user:

client-certificate-data: REDACTED

client-key-data: REDACTED

在上面的配置文件当中，定义了集群、上下文以及用户。其中Config也是K8S的标准资源之一，在该配置文件当中定义了一个集群列表，指定的集群可以有多个；用户列表也可以有多个，指明集群中的用户；而在上下文列表当中，是进行定义可以使用哪个用户对哪个集群进行访问，以及当前使用的上下文是什么。

**第二部分、授权**

 Kubernetes的授权是基于插件形式的，其常用的授权插件有以下几种：

1）Node（节点认证）

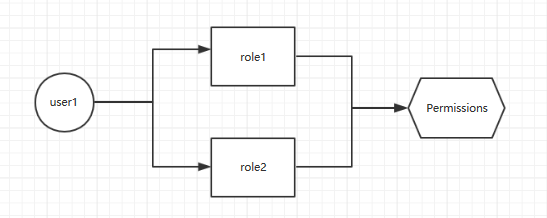
2）ABAC(基于属性的访问控制)

**3）RBAC（基于角色的访问控制）\*\*\***

4）Webhook（基于http回调机制的访问控制）

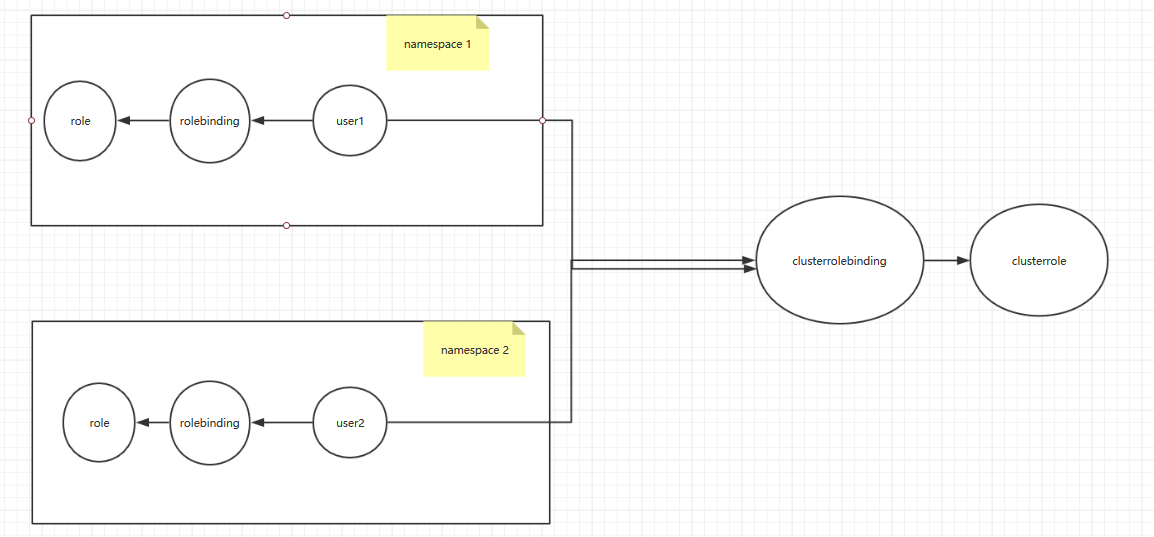
什么是**RBAC（基于角色的访问控制）？**

 让一个用户（Users）扮演一个角色（Role），角色拥有权限，从而让用户拥有这样的权限，随后在授权机制当中，只需要将权限授予某个角色，此时用户将获取对应角色的权限，从而实现角色的访问控制。如图：



在k8s的授权机制当中，采用RBAC的方式进行授权，其工作逻辑是，把对对象的操作权限定义到一个角色当中，再将用户绑定到该角色，从而使用户得到对应角色的权限。如果通过rolebinding绑定role，只能对rolebingding所在的名称空间的资源有权限，上图user1这个用户绑定到role1上，只对role1这个名称空间的资源有权限，对其他名称空间资源没有权限，属于名称空间级别的；

另外，k8s为此还有一种集群级别的授权机制，就是定义一个集群角色（ClusterRole），对集群内的所有资源都有可操作的权限，从而将User2通过ClusterRoleBinding到ClusterRole，从而使User2拥有集群的操作权限。Role、RoleBinding、ClusterRole和ClusterRoleBinding的关系如下图：



通过上图可以看到，可以通过rolebinding绑定role，**rolebinding绑定clusterrole**，clusterrolebinding绑定clusterrole。

上面我们说了两个角色绑定：

（1）用户通过rolebinding绑定role

（2）用户通过clusterrolebinding绑定clusterrole

还有一种：rolebinding绑定clusterrole

假如有6个名称空间，每个名称空间的用户都需要对自己的名称空间有管理员权限，那么需要定义6个role和rolebinding，然后依次绑定，如果名称空间更多，我们需要定义更多的role，这个是很麻烦的，所以我们引入clusterrole，定义一个clusterrole，对clusterrole授予所有权限，然后用户通过rolebinding绑定到clusterrole，就会拥有自己名称空间的管理员权限了

**注：**RoleBinding仅仅对当前名称空间有对应的权限。

**补充知识点，感兴趣可自行查看**

**kubernetes 中的认证机制**

kubernetes 支持多种认证机制，可以配置成多个认证体制共存，这样，只要有一个认证通过，这个request就认证通过了。下面介绍官网列举的几种常见认证机制：

**1.X509 Client Certs**

也叫作双向数字证书认证，HTTPS证书认证，是基于CA根证书签名的双向数字证书认证方式，是所有认证方式中最严格的认证。默认在kubeadm创建的集群中是enabled的，可以在master node上查看kube-apiserver的pod配置文件：

# cat /etc/kubernetes/manifests/kube-apiserver.json  
.................  
containers": [  
 {  
 "name": "kube-apiserver",  
 "image": "gcr.io/google\_containers/kube-apiserver-amd64:v1.5.2",  
 "command": [  
 "kube-apiserver",  
 "--insecure-bind-address=127.0.0.1",  
 "--admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,PersistentVolumeLabel,DefaultStorageClass,ResourceQuota",  
 "--service-cluster-ip-range=10.96.0.0/12",  
 "--service-account-key-file=/etc/kubernetes/pki/apiserver-key.pem",  
 "--client-ca-file=/etc/kubernetes/pki/ca.pem",  
 "--tls-cert-file=/etc/kubernetes/pki/apiserver.pem",  
 "--tls-private-key-file=/etc/kubernetes/pki/apiserver-key.pem",  
 "--token-auth-file=/etc/kubernetes/pki/tokens.csv",  
 "--secure-port=6443",  
 "--allow-privileged",  
 "--advertise-address=192.168.61.100",  
 "--kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname",  
 "--anonymous-auth=false",  
 "--etcd-servers=http://127.0.0.1:2379"  
 ]

**相关的三个启动参数：**

1）client-ca-file: 指定CA根证书文件为/etc/kubernetes/pki/ca.pem，内置CA公钥用于验证某证书是否是CA签发的证书

2）tls-private-key-file: 指定ApiServer私钥文件为/etc/kubernetes/pki/apiserver-key.pem

3）tls-cert-file：指定ApiServer证书文件为/etc/kubernetes/pki/apiserver.pem

只要有这三个启动参数，就说明开启了https的认证方式，这时，如果在集群外访问 [https://masterIP:6443/api](https://masterip:6443/api) 会提示Unauthorized，只有在客户端配置相关认证才可以访问,客户端的认证证书生成与操作可以参考[Creating Certificates](https://kubernetes.io/docs/admin/authentication/" \l "x509-client-certs" \t "_blank)。证书的生成是kubeadm使用openssl自动生成的，如果是手动配置双向认证，相对比较麻烦，主要配置流程如下：

1）生成根证书、API Server服务端证书、服务端私钥、各个组件所用的客户端证书和客户端私钥。

2）修改 Kubernetes 各个服务进程的启动参数，启用双向认证模式.

详细配置可以参考[Kubernetes集群安全配置案例](http://www.cnblogs.com/breg/p/5923604.html" \t "_blank)

注意，在启动参数中还有一个参数：–insecure-bind-address=127.0.0.1，这个参数主要用与master node上的其他核心组件，比如kube-scheduler，kube-controller-manager通过masterIP:8080与APIserver直接通信，而不用通过双向认证。这一点可以从他们的启动参数–master=127.0.0.1:8080看出。

**2.Static Token File**

静态token文件认证，同样，在kubeadm创建的集群中也是默认enabled的，比如，上面的apiserver启动参数中，我们可以看到有参数 ：

–token-auth-file=/etc/kubernetes/pki/tokens.csv ，这个静态token文件的格式为 token,user,uid,”group1,group2,group3”，如下示例：

# cat /etc/kubernetes/pki/tokens.csv  
7db2f1c02d721320,kubeadm-node-csr,0615e0ac-7d70-11e7-ad94-fa163eb9dfdd,system:kubelet-bootstrap

客户端请求的时候需要在http header中加入：”Authorization: Bearer THETOKEN”，如下实例：

curl -k --header "Authorization: Bearer 7db2f1c02d721320" https://192.168.21.34:6443/api

或者使用brctl:

|  |  |
| --- | --- |
| 1 2 3 4 | kubectl --server=https://192.168.21.34:6443 \ --token=7db2f1c02d721320 \ --insecure-skip-tls-verify=true \ cluster-info |

注意，如果该静态token文件更改的话，需要重启apiserver。

**3. Bootstrap Tokens**

bootstrap token认证目前处于alpha阶段，目前主要是kubeadm创建k8s集群时使用。使用这种认证方式，k8s会动态的管理一种type为bootstrap token的token，这些token作为secret放在kube-system namespace中。controller-manager中的tokencleaner controller会在bootstrap token 过期时进行删除。  
使用这种认证方式，apiserver的启动参数中需要有–experimental-bootstrap-token-auth，Controller Manager的启动参数中有–controllers=\*,tokencleaner 类似参数。

**4. Static Password File**

比较简单，kubeadm默认没有开启，生产环境也不建议使用。  
apiserver启动参数指定–basic\_auth\_file=/etc/kubernetes/basic\_auth。然后在指定的文件中加入用户名密码等就可以了，文件格式为password,user,uid,”group1,group2,group3”。

**5. Service Account Tokens**

Service Account Token 是一种比较特殊的认证机制，适用于上文中提到的pod内部服务需要访问apiserver的认证情况，默认enabled。  
还是看上文中apiserver 的启动配置参数有–service-account-key-file=/etc/kubernetes/pki/apiserver-key.pem，如果没有指明文件，默认使用–tls-private-key-file的值，即API Server的私钥。  
service accout本身是作为一种资源在k8s集群中，我们可以通过命令行获取：

[root@k8s-master pki]# kubectl get serviceaccount --all-namespaces  
NAMESPACE NAME SECRETS AGE  
default default 1 7d  
for-test default 1 3d  
kube-system default 1 7d  
kube-system weave-net 1 7d  
sock-shop default 1 7d

可以看到k8s集群为所有的namespace创建了一个默认的service account，利用命令describe会发现service account只是关联了一个secret作为token，也就是service-account-token。

[root@k8s-master pki]# kubectl describe serviceaccount/default -n kube-system  
Name: default  
Namespace: kube-system  
Labels: <none>  
  
Image pull secrets: <none>  
  
Mountable secrets: default-token-nbldr  
  
Tokens: default-token-nbldr  
  
[root@k8s-master pki]# kubectl get secret default-token-nbldr -o yaml -n kube-system  
apiVersion: v1  
data:  
 ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN5RENDQWJDZ0F3SUJBZ0lCQURBTkJna3Foa2lHOXcwQkFRc0ZBREFWTVJNd0VRWURWUVFERXdwcmRXSmwKY201bGRHVnpNQjRYRFRFM0  
 ........................略....................  
 namespace: a3ViZS1zeXN0ZW0=  
 token: ZXlKaGJHY2lPaUpTVXpJMU5pSXNJblI1Y0..................................  
kind: Secret  
metadata:  
 annotations:  
 kubernetes.io/service-account.name: default  
 kubernetes.io/service-account.uid: 67aae699-7d70-11e7-a8a9-fa163eb9dfdd  
 creationTimestamp: 2017-08-10T02:05:38Z  
 name: default-token-nbldr  
 namespace: kube-system  
 resourceVersion: "88"  
 selfLink: /api/v1/namespaces/kube-system/secrets/default-token-nbldr  
 uid: 67b20b73-7d70-11e7-a8a9-fa163eb9dfdd  
type: kubernetes.io/service-account-token

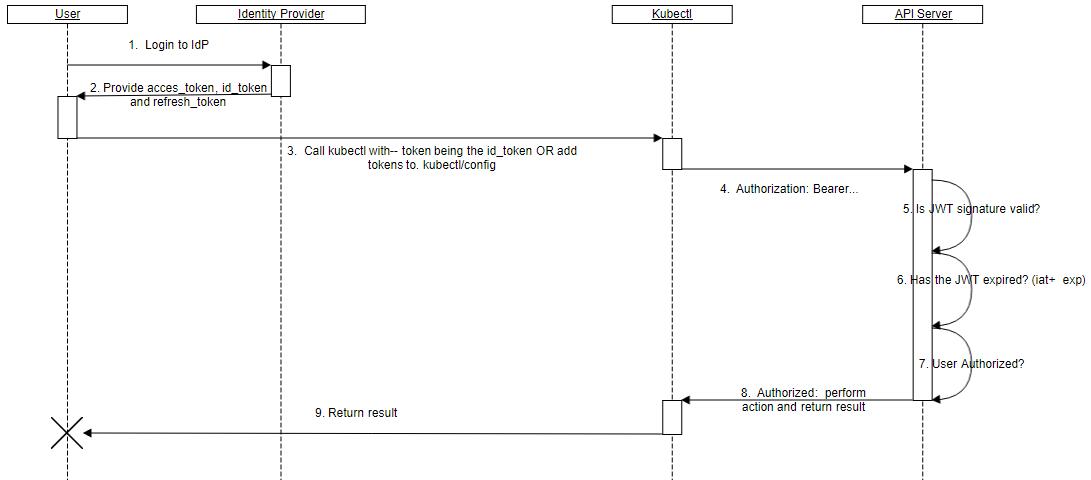
可以看到service-account-token的secret资源包含的数据有三部分：

* ca.crt，这是API Server的CA公钥证书，用于Pod中的Process对API Server的服务端数字证书进行校验时使用的；
* namespace，这是Secret所在namespace的值的base64编码：# echo -n “kube-system”|base64 => “a3ViZS1zeXN0ZW0=”
* token：该token就是由service-account-key-file的值签署(sign)生成。

这种认证方式主要由k8s集群自己管理，用户用到的情况比较少。我们创建一个pod时，默认就会将该namespace对应的默认service account token mount到Pod中，所以无需我们操作便可以直接与apiserver通信，相关示例参考[在Kubernetes Pod中使用Service Account访问API Server](http://tonybai.com/2017/03/03/access-api-server-from-a-pod-through-serviceaccount/" \t "_blank)，当然也可以指定多个service account token,参考[Configure Service Accounts for Pods](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/)。

**6.OpenID Connect Tokens**

类似 OAuth2的认证方式，大致认证过程如下：



除了以上几种认证方式外，还有几种比如Webhook Token Authentication，Keystone Password等，详情见[官网](https://kubernetes.io/docs/admin/authentication/#x509-client-certs)。

**Kubernetes中的授权机制**

k8s中的授权策略也支持开启多个授权插件，只要一个验证通过即可。k8s授权处理主要是根据以下请求属性：

* user, group, extra
* API、请求方法（如get、post、update、patch和delete）和请求路径（如/api）
* 请求资源和子资源
* Namespace
* API Group

目前k8s支持的授权模式主要有以下几种：

* Node Authorization
* ABAC Authorization
* RBAC Authorization
* Webhook Authorization

### Node Authorization

1.7+版本才release的一种授权机制，通过配合NodeRestriction control准入控制插件来限制kubelet访问node，endpoint、pod、service以及secret、configmap、PV和PVC等相关的资源。配置方式为：  
–authorization-mode=Node,RBAC –admission-control=…,NodeRestriction,…

### ABAC Authorization

ABAC(Attribute-based access control),使用这种模式需要配置参数：  
–authorization-mode=ABAC –authorization-policy-file=SOME\_FILENAME。  
这种模式的实现相对比较生硬，就是在master node保存一份policy文件，指定不用用户（或用户组）对不同资源的访问权限,当修改该文件后，需要重启apiserver,跟openstack 的ABAC类似。policy文件的格式如下：

# Alice can do anything to all resources:  
{  
 "apiVersion": "abac.authorization.kubernetes.io/v1beta1",  
 "kind": "Policy",  
 "spec": {  
 "user": "alice",  
 "namespace": "\*",  
 "resource": "\*",  
 "apiGroup": "\*"  
 }  
}  
# Kubelet can read any pods:  
{  
 "apiVersion": "abac.authorization.kubernetes.io/v1beta1",  
 "kind": "Policy",  
 "spec": {  
 "user": "kubelet",  
 "namespace": "\*",  
 "resource": "pods",  
 "readonly": true  
 }  
}  
  
# Kubelet can read and write events:  
{  
 "apiVersion": "abac.authorization.kubernetes.io/v1beta1",  
 "kind": "Policy",  
 "spec": {  
 "user": "kubelet",  
 "namespace": "\*",  
 "resource": "events"  
 }  
}

使用这种模式需要配置参数：  
–authorization-mode=ABAC –authorization-policy-file=SOME\_FILENAME

### RBAC Authorization

RBAC（Role-Based Access Control）依然处于Beta阶段，通过启动参数–authorization-mode=RBAC，使用kubeadm安装k8s默认会enabled。  
RBAC API定义了四个资源对象用于描述RBAC中用户和资源之间的连接权限：

* Role
* ClusterRole
* RoleBinding
* ClusterRoleBinding

Role是定义在某个Namespace下的资源，在这个具体的Namespace下使用。 ClusterRole与Role相似，只是ClusterRole是整个集群范围内使用的。  
RoleBinding把Role绑定到账户主体Subject，让Subject继承Role所在namespace下的权限。 ClusterRoleBinding把ClusterRole绑定到Subject，让Subject集成ClusterRole在整个集群中的权限。

我们可以通过kubectl命令获取对应的Role相关资源进行增删改查：

kubectl get roles --all-namespaces  
  
kubectl get ClusterRoles  
  
kubectl get rolebinding --all-namespaces  
  
kubectl get clusterrolebinding

API Server已经创建一系列ClusterRole和ClusterRoleBinding。这些资源对象中名称以system:开头的，表示这个资源对象属于Kubernetes系统基础设施。 也就说RBAC默认的集群角色已经完成足够的覆盖，让集群可以完全在 RBAC的管理下运行。 修改这些资源对象可能会引起未知的后果，例如对于system:node这个ClusterRole定义了kubelet进程的权限，如果这个角色被修改，可能导致kubelet无法工作。

### Webhook Authorization

用户在外部提供 HTTPS 授权服务，然后配置 apiserver 调用该服务去进行授权。apiserver配置参数：  
–authorization-webhook-config-file=SOME\_FILENAME  
配置文件的格式跟kubeconfig的格式类似，具体参考[官方文档](https://kubernetes.io/docs/admin/authorization/webhook/" \t "_blank)

## kubernetes 中的准入机制

Kubernetes的Admission Control实际上是一个准入控制器(Admission Controller)插件列表，发送到APIServer的请求都需要经过这个列表中的每个准入控制器插件的检查，如果某一个控制器插件准入失败，就准入失败。  
控制器插件如下：

* AlwaysAdmit：允许所有请求通过
* AlwaysPullImages：在启动容器之前总是去下载镜像，相当于每当容器启动前做一次用于是否有权使用该容器镜像的检查
* AlwaysDeny：禁止所有请求通过，用于测试
* DenyEscalatingExec：拒绝exec和attach命令到有升级特权的Pod的终端用户访问。如果集中包含升级特权的容器，而要限制终端用户在这些容器中执行命令的能力，推荐使用此插件
* ImagePolicyWebhook
* ServiceAccount：这个插件实现了serviceAccounts等等自动化，如果使用ServiceAccount对象，强烈推荐使用这个插件
* SecurityContextDeny：将Pod定义中定义了的SecurityContext选项全部失效。SecurityContext包含在容器中定义了操作系统级别的安全选型如fsGroup，selinux等选项
* ResourceQuota：用于namespace上的配额管理，它会观察进入的请求，确保在namespace上的配额不超标。推荐将这个插件放到准入控制器列表的最后一个。ResourceQuota准入控制器既可以限制某个namespace中创建资源的数量，又可以限制某个namespace中被Pod请求的资源总量。ResourceQuota准入控制器和ResourceQuota资源对象一起可以实现资源配额管理。
* LimitRanger：用于Pod和容器上的配额管理，它会观察进入的请求，确保Pod和容器上的配额不会超标。准入控制器LimitRanger和资源对象LimitRange一起实现资源限制管理
* NamespaceLifecycle：当一个请求是在一个不存在的namespace下创建资源对象时，该请求会被拒绝。当删除一个namespace时，将会删除该namespace下的所有资源对象
* DefaultStorageClass
* DefaultTolerationSeconds
* PodSecurityPolicy

当Kubernetes版本>=1.6.0，官方建议使用这些插件：  
–admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,PersistentVolumeLabel,DefaultStorageClass,ResourceQuota,DefaultTolerationSeconds  
当Kubernetes版本>=1.4.0，官方建议使用这些插件：  
–admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,ResourceQuota  
以上是标准的准入插件，如果是自己定制的话，k8s1.7版 出了两个alpha features, Initializers 和 External Admission Webhooks，详情可以参考[Dynamic Admission Control](https://kubernetes.io/docs/admin/extensible-admission-controllers/).

参考<https://zhangchenchen.github.io/2017/08/17/kubernetes-authentication-authorization-admission-control/>