

RBAC 权限控制

前面我们已经学习一些常用的资源对象的使用，我们知道对于资源对象的操作都是通过 APIServer 进行的，那么集群是怎样知道我们的请求就是合法的请求呢？这个就需要了解 Kubernetes 中另外一个非常重要的知识点：**RBAC**（基于角色的权限控制）。

管理员可以通过 Kubernetes API 动态配置策略来启用 **RBAC**，需要在 kube-apiserver 中添加参数 **--authorization-mode=RBAC**，如果使用的 kubeadm 安装的集群那么是默认开启了 **RBAC** 的，可以通过查看 Master 节点上 apiserver 的静态 Pod 定义文件：

```
* → cat /etc/kubernetes/manifests/kube-apiserver.yaml
...
- --authorization-mode=Node,RBAC
...
```

如果是二进制的方式搭建的集群，添加这个参数过后，记得要重启 kube-apiserver 服务。

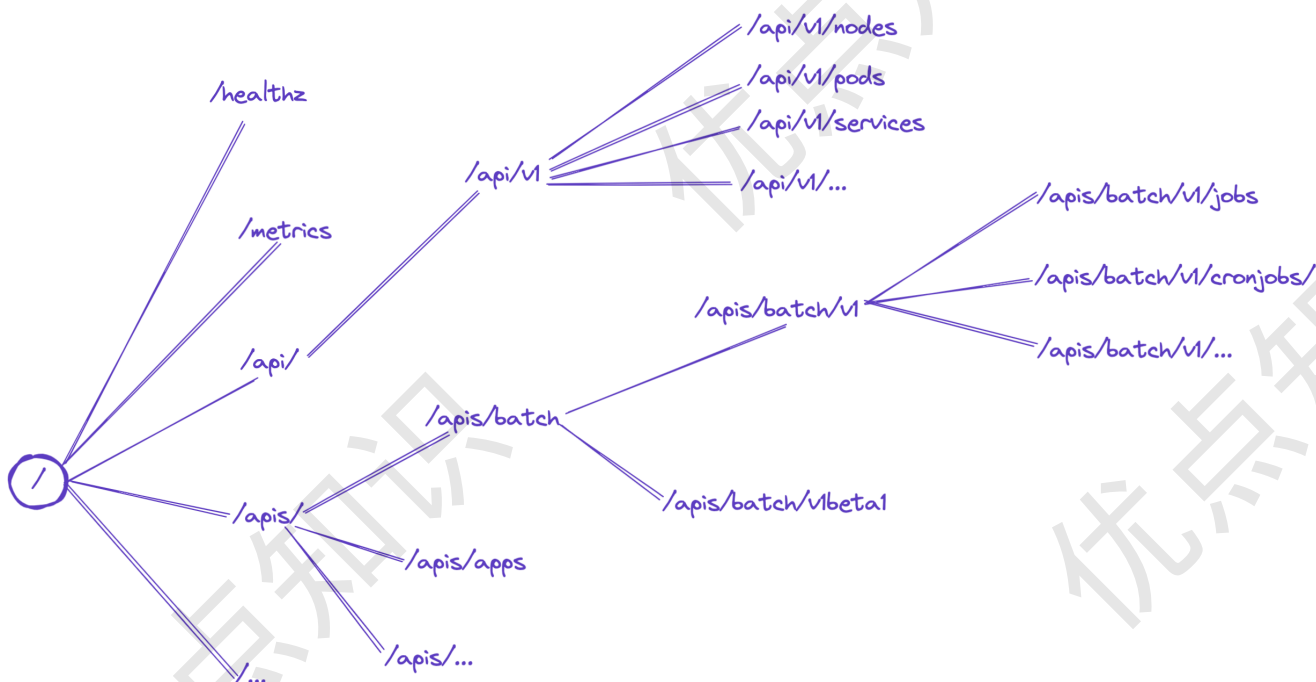
API 对象

在学习 **RBAC** 之前，我们还需要再去了解下 Kubernetes 集群中的对象，我们知道，在 Kubernetes 集群中，Kubernetes 对象是我们持久化的实体，就是最终存入 etcd 中的数据，集群中通过这些实体来表示整个集群的状态。前面我们都直接编写的 YAML 文件，通过 kubectl 来提交的资源清单文件，然后创建的对应的资源对象，那么它究竟是如何将我们的 YAML 文件转换成集群中的一个 API 对象的呢？

这个就需要去了解**声明式 API**的设计，Kubernetes API 是一个以 JSON 为主要序列化方式的 HTTP 服务，除此之外也支持 **Protocol Buffers** 序列化方式，主要用于集群内部组件间的通信。为了可扩展性，Kubernetes 在不同的 API 路径（比如 **/api/v1** 或者 **/apis/batch**）下面支持了多个 API 版本，不同的 API 版本意味着不同级别的稳定性和支持：

- Alpha 级别，例如 **v1alpha1** 默认情况下是被禁用的，可以随时删除对功能的支持，所以要慎用
- Beta 级别，例如 **v2beta1** 默认情况下是启用的，表示代码已经经过了很好的测试，但是对象的语义可能会在随后的版本中以不兼容的方式更改
- 稳定级别，比如 **v1** 表示已经是稳定版本了，也会出现在后续的很多版本中。

在 Kubernetes 集群中，一个 API 对象在 Etcd 里的完整资源路径，是由：**Group (API 组)**、**Version (API 版本)** 和 **Resource (API 资源类型)** 三个部分组成的。通过这样的结构，整个 Kubernetes 里的所有 API 对象，实际上就可以用如下的树形结构表示出来：



从上图中我们也可以看出 Kubernetes 的 API 对象的组织方式，在顶层，我们可以看到有一个核心组（由于历史原因，是 `/api/v1` 下的所有内容而不是在 `/apis/core/v1` 下面）和命名组（路径 `/apis/$NAME/$VERSION`）和系统范围内的实体，比如 `/metrics`。我们也可以用下面的命令来查看集群中的 API 组织形式：

```
* → kubectl get --raw /
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    .....
    "/version"
  ]
}
```

通常，Kubernetes API 支持通过标准 HTTP `POST`、`PUT`、`DELETE` 和 `GET` 在指定 PATH 路径上创建、更新、删除和检索操作，并使用 JSON 作为默认的数据交互格式。

GVK

我们知道要表达一个资源对象需要指定 `group/kind` 和 `version`，这个在 Kubernetes 的 API Server 中简称为 **GVK**，**GVK** 是定位一种类型的方式，例如 `daemonsets` 就是 Kubernetes 中的一种资源，当我们跟 Kubernetes 说我想要创建一个 `daemonsets` 的时候，`kubectl` 是如何知道该怎么向 API Server 发送请求呢？是所有的不同资源都发向同一个 Endpoint，还是每种资源都是不同的？

我们回顾下定义的 `daemonsets` 的资源清单文件：

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-exporter
  namespace: kube-system
spec:
# .....

```

这里声明的 `apiVersion` 是 `apps/v1`，其实就是隐含了 `Group` 是 `apps`，`Version` 是 `v1`，`Kind` 就是定义的 `DaemonSet`，而 `kubectl` 接收到这个清单之后，就可以根据这个声明去调用 API Server 对应的 URL 去获取信息，例如这里就是 `/api/apps/v1/daemonset`。所以我们说 Kubernetes 组织资源的方式是以 REST 的 URI 的形式来的，而组织的路径就是：



我们这里不是说是 `G(roup)V(ersion)K(ind)` 吗，怎么现在又变成了 `G(roup)V(ersion)R(esource)`？这就是 API Server 中的第二个概念：`GVR`。

GVR

理解了 `GVK` 之后再理解 `GVR` 就很容易了，这就是面向对象编程里面的类和对象的概念是一样的：

Kubernetes	OOP
Kind	Class
Resource	Object

`Kind` 其实就是一个类，用于描述对象的；而 `Resource` 就是具体的 `Kind`，可以理解成类已经实例化成对象了。`Resource` 只是 API 中的一个 `Kind` 的使用方式，通常情况下，`Kind` 和 `Resource` 之间有一个一对一的映射。例如，`pods` 资源对应于 `Pod` 种类，但是有时，同一类型可能由多个资源返回。例如 `Scale Kind` 是由所有 `scale` 子资源返回的，如 `deployments/scale` 或 `replicasets/scale`，这就是允许 Kubernetes `HorizontalPodAutoscaler (HPA)` 与不同资源交互的原因。

当我们要定义一个 **GVR** 的时候，那么怎么知道这个 **GVR** 是属于哪个 **GVK** 的呢？也就是前面说的，**kubectl** 是如何从 **YAML** 资源清单文件中知道该请求的是哪个 **GVR** 路径的？这就是 **REST Mapping** 的功能，**REST Mapping** 可以指定一个 **GVR**（例如 **daemonset** 的这个例子），然后它返回对应的 **GVK** 以及支持的操作等。

在代码中，其实就对应于这个接口：

```
// KindFor takes a partial resource and returns the single match.
// Returns an error if there are multiple matches.
KindFor(resource schema.GroupVersionResource) (schema.GroupVersionKind, error)

// KindsFor takes a partial resource and returns the list of potential
// kinds in priority order.
KindsFor(resource schema.GroupVersionResource) ([]schema.GroupVersionKind, error)

// ResourceFor takes a partial resource and returns the single match.
// Returns an error if there are multiple matches.
ResourceFor(input schema.GroupVersionResource) (schema.GroupVersionResource, error)

// ResourcesFor takes a partial resource and returns the list of potential
// resource in priority order.
ResourcesFor(input schema.GroupVersionResource) ([]schema.GroupVersionResource, error)

// RESTMappings returns all resource mappings for the provided group kind
// if no version search is provided. Otherwise identifies a preferred resource
// mapping for the provided version(s).
RESTMappings(gk schema.GroupKind, versions ...string) ([]*RESTMapping, error)
```

这样，就把 **GVK** 和 **GVR** 联系起来了。

现在我们知道了 **API Server** 可以通过 **URI** 的形式访问到资源，我们可以通过 **kubectl proxy** 来代理一个本地的 **API Server** 端口，这样就可以绕过认证了，直接可以以 **http** 的形式进行：

```
* → kubectl proxy
Starting to serve on 127.0.0.1:8001
```

然后我们就可以直接访问 **API Server** 了：

```
* → k8strain4 curl http://127.0.0.1:8001/apis/apps/v1/daemonsets
{
  "kind": "DaemonSetList",
  "apiVersion": "apps/v1",
  "metadata": {
    "resourceVersion": "799026"
  },
  "items": [
    {
      "metadata": {
        "name": "kindnet",
        "namespace": "kube-system",
# .....
      },
# .....
```

```
]
}
```

如果想看系统支持哪些 **GVK**，那么可以通过 `kubectl` 的命令查看：

```
* → kubectl api-resources
NAME                                SHORTNAMES  APIVERSION
NAMESPACEED  KIND
bindings                                           v1
  true      Binding
componentstatuses  cs          v1
  false     ComponentStatus
configmaps        cm          v1
  true      ConfigMap
endpoints          ep          v1
  true      Endpoints
events            ev          v1
  true      Event
limitranges       limits      v1
  true      LimitRange
namespaces        ns          v1
  false     Namespace
nodes            no          v1
  false     Node
# .....
```

这里我们可以发现一些特别的地方，对于 `Pod` 和 `Service`，它的 `API GROUP` 居然是空的，这其实就是 `Kubernetes` 核心（core）资源的含义，也就是所谓的 `Kubernetes` 中的基础资源，他们不需要 **Group**，只有 **Version** 和 **Kind**，其实主要是历史原因导致的，在 `Kubernetes` 的开始之初还不支持自定义类型的时候就没考虑过 **Group**。

RBAC

上面我们介绍了 `Kubernetes` 所有资源对象都是模型化的 `API` 对象，允许执行 **CRUD(Create、Read、Update、Delete)** 等操作（也就是我们常说的增、删、改、查操作），比如下面的这些资源：

- `Pods`
- `ConfigMaps`
- `Deployments`
- `Nodes`
- `Secrets`
- `Namespaces`
-

对于上面这些资源对象的可能存在的操作有：

- `create`
- `get`
- `delete`
- `list`
- `update`
- `edit`
- `watch`

- exec
- patch

在更上层，这些资源和 API Group 进行关联，比如 Pods 属于 Core API Group，而 Deployments 属于 apps API Group，现在我们要在 Kubernetes 中通过 RBAC 来对资源进行权限管理，除了上面的这些资源和操作以外，我们还需要了解另外几个概念：

- **Rule**：规则，规则是一组属于不同 API Group 资源上的一组操作的集合
- **Role** 和 **ClusterRole**：角色和集群角色，这两个对象都包含上面的 Rules 元素，二者的区别在于，在 Role 中，定义的规则只适用于单个命名空间，也就是和 namespace 关联的，而 ClusterRole 是集群范围内的，因此定义的规则不受命名空间的约束。另外 Role 和 ClusterRole 在 Kubernetes 中都被定义为集群内部的 API 资源，和我们前面学习过的 Pod、Deployment 这些对象类似，都是我们集群的资源对象，所以同样的可以使用 YAML 文件来描述，用 kubectl 工具来管理
- **Subject**：主题，对应集群中尝试操作的对象，集群中定义了 3 种类型的主题资源：
 - **User Account**：用户，这是有外部独立服务进行管理的，管理员进行私钥的分配，用户可以使用 KeyStone 或者 Goolge 帐号，甚至一个用户名和密码的文件列表也可以。对于用户的管理集群内部没有一个关联的资源对象，所以用户不能通过集群内部的 API 来进行管理
 - **Group**：组，这是用来关联多个账户的，集群中有一些默认创建的组，比如 **cluster-admin**
 - **Service Account**：服务帐号，通过 Kubernetes API 来管理的一些用户帐号，和 namespace 进行关联的，适用于集群内部运行的应用程序，需要通过 API 来完成权限认证，所以在集群内部进行权限操作，我们都需要使用到 ServiceAccount，这也是我们这节课的重点
- **RoleBinding** 和 **ClusterRoleBinding**：角色绑定和集群角色绑定，简单来说就是把声明的 Subject 和我们的 Role 进行绑定的过程（给某个用户绑定上操作的权限），二者的区别也是作用范围的区别：RoleBinding 只会影响到当前 namespace 下面的资源操作权限，而 ClusterRoleBinding 会影响到所有的 namespace。

接下来我们来通过几个简单的示例，来学习下在 Kubernetes 集群中如何使用 RBAC。

只能访问某个 namespace 的普通用户

我们想要创建一个 User Account，只能访问 kube-system 这个命名空间，对应的用户信息如下所示：

```
username: cnych
group: youdianzhishi
```

创建用户凭证

我们前面已经提到过，Kubernetes 没有 User Account 的 API 对象，不过要创建一个用户帐号的话也是挺简单的，利用管理员分配给你的一个私钥就可以创建了，这个我们可以参考官方文档中的方法，这里我们来使用 **OpenSSL** 证书来创建一个 User，当然我们也可以使用更简单的 **cfssl** 工具来创建：

给用户 cnych 创建一个私钥，命名成 **cnych.key**：

```
* → openssl genrsa -out cnych.key 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....
.....+++
e is 65537 (0x10001)
```

使用我们刚刚创建的私钥创建一个证书签名请求文件：**cnych.csr**，要注意需要确保在 **-subj** 参数中指定用户名和组（CN 表示用户名，O 表示组）：

```
* → openssl req -new -key cnych.key -out cnych.csr -subj "/CN=cnych/O=youdianzhishi"
```

然后找到我们的 Kubernetes 集群的 CA 证书，我们使用的是 kubeadm 安装的集群，CA 相关证书位于 `/etc/kubernetes/pki/` 目录下，如果你是二进制方式搭建的，你应该在最开始搭建集群的时候就已经指定好了 CA 的目录，我们会利用该目录下面的 `ca.crt` 和 `ca.key` 两个文件来批准上面的证书请求。生成最终的证书文件，我们这里设置证书的有效期为 500 天：

```
* → openssl x509 -req -in cnych.csr -CA /etc/kubernetes/pki/ca.crt -CAkey
/etc/kubernetes/pki/ca.key -CAcreateserial -out cnych.crt -days 500
Signature ok
subject=/CN=cnych/O=youdianzhishi
Getting CA Private Key
```

现在查看我们当前文件夹下面是否生成了一个证书文件：

```
* → ls
cnych.crt  cnych.csr  cnych.key
```

现在我们可以使用刚刚创建的证书文件和私钥文件在集群中创建新的凭证和上下文(Context)：

```
* → kubectl config set-credentials cnych --client-certificate=cnych.crt --client-
key=cnych.key
User "cnych" set.
```

我们可以看到一个用户 `cnych` 创建了，然后为这个用户设置新的 `Context`，我们这里指定特定的一个 namespace：

```
* → kubectl config get-clusters
NAME
kubernetes
kind-demo
* → kubectl config set-context cnych-context --cluster=kubernetes --namespace=kube-
system --user=cnych
Context "cnych-context" created.
```

到这里，我们的用户 `cnych` 就已经创建成功了，现在我们使用当前的这个配置文件来操作 `kubectl` 命令的时候，应该会出现错误，因为我们还没有为该用户定义任何操作的权限呢：

```
* → kubectl get pods --context=cnych-context
Error from server (Forbidden): pods is forbidden: User "cnych" cannot list resource
"pods" in API group "" in the namespace "kube-system"
```

如果使用的 `Kind` 创建的测试集群，上面的方式可能不会生效，`apiserver` 中出现类似如下所示的错误信息：

```
E0110 08:49:05.886544 1 authentication.go:63] "Unable to authenticate the request"
err="[x509: certificate signed by unknown authority, verifying certificate
SN=9538989897811029142, SKID=, AKID= failed: x509: certificate signed by unknown
authority (possibly because of \"x509: cannot verify signature: insecure algorithm SHA1-
RSA (temporarily override with GODEBUG=x509sha1=1)\") while trying to verify candidate
authority certificate \"kubernetes\")]"
```

可以用下面的方法来临时解决:

```
* → docker exec -ti <xxx>-control-plane bash
# sed -i "s@http://ports.ubuntu.com@https://mirrors.tuna.tsinghua.edu.cn@g"
/etc/apt/sources.list
* → root@xxx-control-plane:/# apt update && apt install vim -y
* → root@xxx-control-plane:/# vim /etc/kubernetes/manifests/kube-apiserver.yaml
# 添加下面的环境变量
env:
- name: GODEBUG
  value: x509sha1=1
```

创建角色

用户创建完成后, 接下来就需要给该用户添加操作权限, 我们来定义一个 YAML 文件, 创建一个允许用户操作 Deployment、Pod、ReplicaSets 的角色, 如下定义:

```
# cnych-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: cnych-role
  namespace: kube-system
rules:
- apiGroups: ["", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"] # 也可以使用 ['*']
```

其中 Pod 属于 **core** 这个 API Group, 在 YAML 中用空字符就可以, 而 Deployment 和 ReplicaSet 现在都属于 **apps** 这个 API Group (如果不知道则可以用 **kubectl explain** 命令查看), 所以 **rules** 下面的 **apiGroups** 就综合了这几个资源的 API Group: ["", "apps"], 其中 **verbs** 就是我们上面提到的可以对这些资源对象执行的操作, 我们这里需要所有的操作方法, 所以我们可以使用 **['*']** 来代替, 然后直接创建这个 Role:

```
* → kubectl create -f cnych-role.yaml
role.rbac.authorization.k8s.io/cnych-role created
```

注意这里我们没有使用上面的 **cnych-context** 这个上下文, 因为暂时还未有权限。

创建角色权限绑定

Role 创建完成了, 但是很明显现在我们这个 **Role** 和我们的用户 **cnych** 还没有任何关系, 对吧? 这里就需要创建一个 **RoleBinding** 对象, 在 kube-system 这个命名空间下面将上面的 **cnych-role** 角色和用户 **cnych** 进行绑定:

```
# cnych-rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: cnych-rolebinding
  namespace: kube-system
```



```

subjects:
  - kind: User
    name: cnych
    apiGroup: ""
roleRef:
  kind: Role
  name: cnych-role
  apiGroup: rbac.authorization.k8s.io # 留空字符串也可以, 则使用当前的apiGroup

```

上面的 YAML 文件中我们看到了 **subjects** 字段, 这里就是我们上面提到的用来尝试操作集群的对象, 这里对应上面的 **User** 帐号 **cnych**, 使用 **kubectl** 创建上面的资源对象:

```

❖ → kubectl create -f cnych-rolebinding.yaml
rolebinding.rbac.authorization.k8s.io/cnych-rolebinding created

```

测试

现在我们应该可以上面的 **cnych-context** 上下文来操作集群了:

```

❖ → kubectl get pods --context=cnych-context

```

NAME	READY	STATUS	RESTARTS
AGE			
coredns-565d847f94-9q52n	1/1	Running	13 (49s ago)
39d			
coredns-565d847f94-dh5nz	1/1	Running	13 (49s ago)
39d			
etcd-demo-control-plane	1/1	Running	1 (49s ago)
93m			
kindnet-2qdbm	1/1	Running	13 (60s ago)
39d			
kindnet-9qf4c	1/1	Running	14 (59s ago)
39d			
kindnet-zmt4n	1/1	Running	13 (49s ago)
39d			
kube-apiserver-demo-control-plane	1/1	Running	0
36s			
kube-controller-manager-demo-control-plane	1/1	Running	13 (49s ago)
39d			
.....			

```

❖ → kubectl --context=cnych-context get rs,deploy

```

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/coredns-565d847f94	2	2	2	39d
replicaset.apps/metrics-server-56c5d7b58b	1	1	1	20d

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/coredns	2/2	2	2	39d
deployment.apps/metrics-server	1/1	1	1	20d

我们可以看到我们使用 **kubectl** 的使用并没有指定 **namespace**, 这是因为我们上面创建这个 **Context** 的时候就绑定在了 **kube-system** 这个命名空间下面, 如果我们在后面加上一个 **-n default** 试看看呢?

```
* → kubectl --context=cnych-context get pods --namespace=default
Error from server (Forbidden): pods is forbidden: User "cnych" cannot list resource
"pods" in API group "" in the namespace "default"
```

如果去获取其他的资源对象呢:

```
* → kubectl --context=cnych-context get svc
Error from server (Forbidden): services is forbidden: User "cnych" cannot list resource
"services" in API group "" in the namespace "kube-system"
```

我们可以看到没有权限获取, 因为我们并没有为当前操作用户指定其他对象资源的访问权限, 是符合我们的预期的。这样我们就创建了一个只有单个命名空间访问权限的普通 User。

只能访问某个 namespace 的 ServiceAccount

首先我们需要对 **ServiceAccount** 了解下。

ServiceAccount

ServiceAccount 为 Pod 中运行的进程提供了一个身份, Pod 内的进程可以使用其关联服务账号的身份, 向集群的 API Server 进行身份认证。

当创建 Pod 的时候规范下面有一个 **spec.serviceAccount** 的属性用来指定该 Pod 使用哪个 ServiceAccount, 如果没有指定的话则默认使用 default 这个 sa。然后通过投射卷, 在 Pod 的目录 **/run/secrets/kubernetes.io/serviceaccount/** 下有一个 **token** 令牌文件。我们通过 RBAC 对该 sa 授予了什么权限, 那么容器里的应用拿着这个 token 后, 就具备了对应的权限。

但是需要注意的是不同的 K8s 版本对该 **token** 文件的使用是不一样的, 所以我们可以分几个版本来分别讨论下。

≤ 1.20 版本

使用 kind 快速创建一个小于等于 **v1.20** 版本的集群:

```
* → kind create cluster --name kind120 --image kindest/node:v1.20.15
* → kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kind120-control-plane	Ready	control-plane,master	33s	v1.20.15

我们先创建一个名为 **sa-demo** 的 ServiceAccount 对象:

```
* → kubectl create sa sa-demo
* → kubectl get sa
```

NAME	SECRETS	AGE
default	1	43s
sa-demo	1	6s

```
* → kubectl get secret
```

NAME	TYPE	DATA	AGE
default-token-dv78w	kubernetes.io/service-account-token	3	46s
sa-demo-token-4gvbw	kubernetes.io/service-account-token	3	8s

我们可以看到创建 sa 后自动生成了一个 secret，格式为 `<sa-name>-token-xxxx`，比如我们创建了一个名字为 `sa-demo` 的 sa 之后，系统自动创建了一个名字为 `sa-demo-token-4gvbw` 的 secret，这个 secret 里就包含了一个 token。

```
❖ → kubectl describe secrets sa-demo-token-4gvbw
Name:          sa-demo-token-4gvbw
Namespace:     default
Labels:        <none>
Annotations:   kubernetes.io/service-account.name: sa-demo
                kubernetes.io/service-account.uid: 1ae8eea9-acc6-4e3d-b378-07feb9146ac4

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:      1066 bytes
namespace:   7 bytes
token:
eyJhbGciOiJSUzI1NiIsImtpZCI6ImhqNmFMNjAyaDZ5OElyMmtTNGdPUWwRdHVDU1A4aGFvVkJiNHdHMkZjQ1Ui
fQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3Vud
C9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6I
nNhLWRLbW8tdG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C5uYW1lIjoic2EtZGVtb3VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZ
CI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC9zZW5yZXQubmFt
ZSI6IjFhZjZlZG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C3VudDpkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZW5yZXQubmFtZSI6IjFhZjZlZG9r
ZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb
```

可以看到自动生成的这个 `secret` 对象里面包含一个 `token`，我们也可以通过下面的命令来获取：

```
❖ → kubectl get secrets sa-demo-token-4qvbw -o jsonpath='{.data.token}' | base64 -d
```

这个 token 是 JWT 结构的，我们可以把这个 token 复制到 jwt.io 网站进行解码。

```
❖ → kubectl apply -f demo-pod.yaml
❖ → kubectl get pods
```

NAME	READY	STATUS	RESTARTS
demo	1/1	Running	0

```
❖ → kubectl get pod demo -oyaml
```

```

apiVersion: v1
kind: Pod
metadata:
  name: demo
  namespace: default
spec:
  containers:
    - image: nginx:1.7.9
      imagePullPolicy: IfNotPresent
      name: demo
# .....
  volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: sa-demo-token-4gvbw
      readOnly: true
# .....
  volumes:
    - name: sa-demo-token-4gvbw
      secret:
        defaultMode: 420
        secretName: sa-demo-token-4gvbw

```

Pod 创建后我们可以看到会自动将指定 sa 对应的 secret 挂载到容器的 `/var/run/secrets/kubernetes.io/serviceaccount` 目录下，所以现在该目录下面肯定包含对应的 token 文件，我们可以查看该值来验证下：

```

❖ → kubectl exec -it demo -- cat /run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6ImhQNmFMNjAyaDZ50ElyMmtTNGdPUWxRdHVDU1A4aGFfVkJiNHdHMkZjQlUi
fQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3Vud
C9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6I
nNhLWRLbW8tdG9rZW4tNGd2YnciLCJrdWJlcm5ldGVzLmVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3Vud
C5uYW1lIjoic2EtZGVtbyIsImt1YmVybmV0ZXMuaW8vc2VydmljZWJyY291bnQvc2VydmljZS1hY2NvdW50LnVpZ
CI6IjFhZThlZWE5LWFjYzYtNGUzCz1iMzc4LTA3ZmVi0TE0NmFjNCIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb
3VudDpkZWZhdWx0OnNhLWRLbW8ifQ.j0DQmzTeSfagKYGc2dMUuzhYqVqh2puJAoQS0EMKeiAKD6rC4bUHWXWBrC
u5Ttvpch6ZTEYwyCdRof1LDGWiLa3pJ1R1RWUNVQTCmVTZPs7tTuoGLRW0KGfEd0jyi4LU6uw4kA_6kwEsz4q2qu
WcB_fiH_Z3iKVfh1JoLYTVAWTBMWnVn6gBvIrLXV5ny2oyvcPQeVfIek8aPQqhbsct_q0xrjqpZY8mpBz0ETR_EE
LjmcZxVVPLvom0dCqEqbV-FF5KRiFxyzB3Xoh6NH3EcsxpCZNRydand-
UFHaBQC9IPwJKzxhANGmuZuWJUCqCVGGRZTo9c6eoyVz831sZ0A

```

可以看到 Pod 里通过投射卷所挂载的 token 跟 sa-demo 对应的 secret 包含的 token 是模一样的，这个 token 是永不过期的，所以即使删除了 Pod 之后重新创建，Pod 里的 token 仍然是不变的，因为 secret 对象里面的 token 数据并没有变化。

如果需要在 Pod 中去访问 K8s 集群的资源对象，现在就可以为使用的 sa 绑定上相应的权限，然后在 Pod 应用中使用该对应的 token 去和 APIServer 进行通信就可以了，这个时候的 token 就能识别到对应的权限了。

≥ 1.21 版本 && ≤ 1.23 版本

接下来我们基于 ≥ 1.21 && ≤ 1.23 版本的 K8s 集群进行测试。

这里我们使用 kind 快速创建一个 v1.22.15 版本的集群：

NAME	STATUS	ROLES	AGE	VERSION
kind122-control-plane	Ready	control-plane,master	115s	v1.22.15

同样首先创建一个名为 `sa-demo` 的 ServiceAccount 对象:

NAME	TYPE	DATA	AGE
default-token-9w9bp	kubernetes.io/service-account-token	3	116s
sa-demo-token-g7d2g	kubernetes.io/service-account-token	3	8s

同样可以看到创建 `sa` 后系统也自动创建了一个对应的 `secret` 对象，和以前版本没什么区别，我们也可以通过下面的命令来获得该 `secret` 对象里面包含的 `token` 值：

```
* → kubectl get secrets sa-demo-token-g7d2g -o jsonpath='{.data.token}' | base64 -d  
eyJhbGciOiJSUzI1NiIsImtpZCI6Im1ERkhnQ3Y3b1oxUmNHbWVhN210SDEwNXky2dVNkc0ZzdXJjTkhsY21FRVEi  
fQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWYjbnVudC9u  
C9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWYjbnVudC9zZWNyZXQubmFtZSI6I  
nNhLWRlbW8tdG9rZW4tZzdkMmciLCJrdWJlcm5ldGVzLmVlL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3Vud  
C5uYW1lIjoic2EtZGVtbysImt1YmVybmV0ZXMuaW8vc2VydmVjZWZjY291bnQvc2VydmVjZS1hY2NvdW50LnVpZ  
CI6IjIzZGI0M2FjLTdjYyItNDQ2Yi05N2Q1LWU0MGUzOWRjZTg4YyIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWYjbn  
VudDpkZWZhdWx0OnNhLWRlbW8ifQ.fnSaqrZKoLTfz2pi9t32X38Er60WSzUoRHArte6qVmQ1NTaMis4F6rESWe  
keJvGW26szTJdl16vK8KtL_IR02m6sp_fEAYfNMQMXL4CuarByXeAavDqLgMHhodf4k4Yg-  
Mj4LCQ3aH0xobjAbPT1i_h17Ewivc39fmzp-dAXbHhhWhCW2Vl_CkM-F-  
UtzLyDwThvJedkeetrFY00je7K6HpzWfqIQyMUdCJog3WnFO_4kHXacFCgYg_gNPMYyViQAsTsxB8FplGdEzRuWK  
nQ09cDE55V4l55IxME0er-dSSdG8085PzxaM_LMcTRI8YtjrJxcbxS5QkTm5R_ps0IsA
```

同样将该 token 值拷贝到 jwt.io 网站进行解码。



```
# demo-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  serviceAccount: sa-demo
  containers:
    - name: demo
      image: nginx:1.7.9
      ports:
        - containerPort: 80
```

```
❖ → kubectl apply -f demo-pod.yaml
❖ → kubectl get pods
```

NAME	READY	STATUS	RESTARTS
demo	1/1	Running	0

```
❖ → kubectl get pod demo -oyaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
  namespace: default
spec:
  containers:
  - image: nginx:1.7.9
    imagePullPolicy: IfNotPresent
    name: demo
    ports:
    - containerPort: 80
      protocol: TCP
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-6wmfb
      readOnly: true
# .....
volumes:
- name: kube-api-access-6wmfb
  projected:
    defaultMode: 420
    sources:
    - serviceAccountToken:
        expirationSeconds: 3607
        path: token
    - configMap:
        items:
        - key: ca.crt
          path: ca.crt
        name: kube-root-ca.crt
    - downwardAPI:
        items:
        - fieldRef:
            apiVersion: v1
            fieldPath: metadata.namespace
          path: namespace
```

当 Pod 创建后查看对应的资源对象，可以看到和之前的版本已经有一个很大的区别了，并不是将上面自动创建的 secret 挂载到容器的 `/var/run/secrets/kubernetes.io/serviceaccount` 目录。我们可以查看下 Pod 中的 token 值和 secret 包含的 token 值进行对比：

可以很明显看到现在 Pod 中的 token 值和自动创建 secret 的 token 值不一样了，同样在 jwt.io 解码该 token 值。



从上面查看创建后的 Pod 资源清单可以看出，现在创建 Pod 后，Kubernetes 控制平面会自动添加一个投射卷到 Pod，此卷包括了访问 Kubernetes API 的 token，该清单片段定义了由三个数据源组成的投射卷，这三个数据源是：

- **serviceAccountToken 数据源**: 包含 kubelet 从 kube-apiserver 获取的令牌, kubelet 使用 TokenRequest API 获取有时间限制的令牌。为 TokenRequest 服务的这个令牌会在 Pod 被删除或定义的生命周期 (默认为 1 小时) 结束之后过期。该令牌绑定到特定的 Pod, 并将其 audience (受众) 设置为与 kube-apiserver 的 audience 相匹配。这种机制取代了之前基于 Secret 添加卷的机制, 之前 Secret 代表了针对 Pod 的 ServiceAccount 但不会过期。
- **configMap 数据源**: ConfigMap 包含一组证书颁发机构数据, Pod 可以使用这些证书来确保自己连接到集群的 kube-apiserver (而不是连接到中间件或意外配置错误的对等点上)。
- **downwardAPI 数据源**: 用于查找包含 Pod 的名字空间的名称, 并使该名称信息可用于在 Pod 内运行的应用程序代码。

所以我们应该要指定现在版本的 K8s 集群创建的 Pod 里面包含的 token 不是使用 ServiceAccount 自动关联的 secret 对象里面的 token 了, 而是 kubelet 会向 **TokenRequest API** 发送一个请求, 申请一个新的 token 放在 Pod 的 **/run/secrets/kubernetes.io/serviceaccount/token** 里。这个 token 会在 1 个小时后由 kubelet 重新去申领一个新的 token, 所以 1 小时之后再次查看这个 token 的话会发现 token 的内容是变化的, 如果删除此 Pod 重新创建的话, 则会重新申领 token, 被删除 Pod 里的 token 会立即过期。

而且我们还可以手动使用 **kubectl create token <sa>** 命令来请求 ServiceAccount 的 token, 可以指定有效期等:

```
❖ → kubectl create token -h
Request a service account token.

Examples:
# Request a token to authenticate to the kube-apiserver as the service account "myapp"
in the current namespace
kubectl create token myapp

# Request a token for a service account in a custom namespace
kubectl create token myapp --namespace myns

# Request a token with a custom expiration
kubectl create token myapp --duration 10m

# Request a token with a custom audience
kubectl create token myapp --audience https://example.com

# Request a token bound to an instance of a Secret object
kubectl create token myapp --bound-object-kind Secret --bound-object-name mysecret

# Request a token bound to an instance of a Secret object with a specific uid
kubectl create token myapp --bound-object-kind Secret --bound-object-name mysecret --
bound-object-uid
0d4691ed-659b-4935-a832-355f77ee47cc

Options:
# .....
```

≥ 1.24 版本

现在我们再来看下 v1.24 版本以上的 K8s 集群中的 ServiceAccount token 是如何工作的。这里我们使用 kind 快速创建一个 **v1.25.3** 版本的集群:

```
* → kind create cluster --name kind125 --image kindest/node:v1.25.3
* → kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kind125-control-plane	Ready	control-plane,master	115s	v1.25.3

同样创建一个名为 `sa-demo` 的 ServiceAccount:

```
* → kubectl create sa sa-demo
* → kubectl get sa
```

NAME	SECRETS	AGE
default	0	39d
sa-demo	0	5s

```
* → kubectl get secrets
No resources found in ns1 namespace
```

我们可以看到该 ServiceAccount 创建后并没有创建对应的 Secret 对象。同样接下来创建一个如下所示的 Pod:

```
# demo-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  serviceAccount: sa-demo
  containers:
    - name: demo
      image: nginx:1.7.9
      ports:
        - containerPort: 80
```

创建上面的 Pod 后查看详情:

```
* → kubectl apply -f demo-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo
  namespace: default
spec:
  containers:
    - image: nginx:1.7.9
      imagePullPolicy: IfNotPresent
      name: demo
      ports:
        - containerPort: 80
          protocol: TCP
      volumeMounts:
        - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
          name: kube-api-access-pftqd
          readOnly: true
```

```
# .....
volumes:
- name: kube-api-access-pftqd
  projected:
    defaultMode: 420
    sources:
      - serviceAccountToken:
          expirationSeconds: 3607
          path: token
      - configMap:
          items:
            - key: ca.crt
              path: ca.crt
              name: kube-root-ca.crt
      - downwardAPI:
          items:
            - fieldRef:
                apiVersion: v1
                fieldPath: metadata.namespace
              path: namespace
```

可以看到创建 Pod 后同样会自动添加一个投射卷到 Pod，此卷包括了访问 Kubernetes API 的令牌，和 ≥ 1.21 版本 && ≤ 1.23 版本 表现是一致的。同样我们可以查看 Pod 中的 token 值来进行验证：

```
* → kubectl exec -it demo -- cat /run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6IndnazJLZENQTktiZkxVejhMnhmTHJYRTlkZ2ZnOHJGQmgwVW4td3Bwd0ki
fQ.eyJhdWQiOi0lsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjoxN
zA0ODg0MDg0LCJpYXQiOi0jE2NzMzNDgwODQsImIzcyI6Imh0dHBz0i8va3ViZXJlcy5kZWZhdWx0LnN2Yy5jb
HVzdGVyLmxvY2FsIiwia3ViZXJlcy5pbyI6eyJuYW1lczBhY2UiOiJkZWZhdWx0IiwicG9kIjp7Im5hbWUiOi
iJkZWlviIiwidWlkIjoiaMTY0ZTIwZTYtYjNjMi00ZmQ5LWI3ZTU0MDZjYUExZWY0OWM4In0sInNlcnZpY2VhY2Nvd
W50Ijp7Im5hbWUiOiJzYS1kZW1vIiwidWlkIjoiaYjJlNWZmY2NjY2VhY2NjY2VhY2NjY2VhY2NjY2VhY2NjY2Vh
n0sIndhcm5hZnRlciI6MTY3MzM1MTY5MTY3MzM0MDA4NCwic3ViIjoic3lzdGVtOnNlcnZpY2VhY2NvdW50mRlZmF1bHQ6c2EtZGVtbyJ9.LhYscyn_d9Y3GZSipSqGj4Jtsu8qsIyz34L18lv37HxjjGU_bQmUFCXY
f_CRom8DfadHppmlaskZS18KmyTV1Z09BeujJd8viUnnYCwb9K6VJB5uPBWLB0FETfgQy7Kqu8Gvk8qBKLjdCkL
8U2vr20qd2qSEdyvqhNBQXnckQRH6wyypBUc7EXSGAJf6dPVE3c6XqnbXMJ7SRZb5svE-
hv0LZKmJrouz9Ia4qxUXUtpzDlMPnH0ym2x9d1TSSZ1Lp7B0sqTnxLUQVueh9w869jAajrP1G9e5zhZwZBfzRfAR
qCVqoLid_h0QP-mo4MLfHbn61SWItlCBd75nL2WLQ
```

我们可以把上面输出的 token 值拷贝到 jwt.io 里进行解码。



为 ServiceAccount 分配权限

上面我们创建了一个只能访问某个命名空间下面的**普通用户**，我们前面也提到过 **subjects** 下面还有一种类型的主题资源：**ServiceAccount**，现在我们来创建一个集群内部的用户只能操作 kube-system 这个命名空间下面的 pods 和 deployments，首先来创建一个 **ServiceAccount** 对象：

```
* → kubectl create sa cnych-sa -n kube-system
```

当然我们也可以定义成 YAML 文件的形式来创建：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cnych-sa
  namespace: kube-system
```

然后新建一个 Role 对象：

```
# cnych-sa-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: cnych-sa-role
  namespace: kube-system
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
  - apiGroups: ["apps"]
    resources: ["deployments"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

可以看到我们这里定义的角色没有**创建、删除、更新** Pod 的权限，待会我们可以重点测试一下，创建该 Role 对象：

```
* → kubectl apply -f cnych-sa-role.yaml
```

然后创建一个 **RoleBinding** 对象，将上面的 **cnych-sa** 和角色 **cnych-sa-role** 进行绑定：

```
# cnych-sa-rolebinding.yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cnych-sa-rolebinding
  namespace: kube-system
subjects:
  - kind: ServiceAccount
    name: cnych-sa
    namespace: kube-system
roleRef:
  kind: Role
  name: cnych-sa-role
```

```
apiGroup: rbac.authorization.k8s.io
```

添加这个资源对象：

```
❖ → kubectl create -f cnych-sa-rolebinding.yaml
```

然后我们怎么去验证这个 ServiceAccount 呢？以前的版本 ServiceAccount 会自动生成一个 Secret 对象和它进行映射，这个 Secret 里面包含一个 token，可以利用这个 token 去登录 Dashboard，然后就可以在 Dashboard 中来验证我们的功能是否符合预期了，但是现在的版本默认已经不支持这种方式了。

但是我们可以手动创建一个长久有效的 ServiceAccount 令牌，要为 ServiceAccount 创建一个不过期、持久化的 API 令牌，需要创建一个类型为 `kubernetes.io/service-account-token` 的 Secret，附带引用 ServiceAccount 的注解。控制平面随后生成一个长久的令牌，并使用生成的令牌数据更新该 Secret。比如我们这里可以创建一个如下所示的 Secret 资源对象：

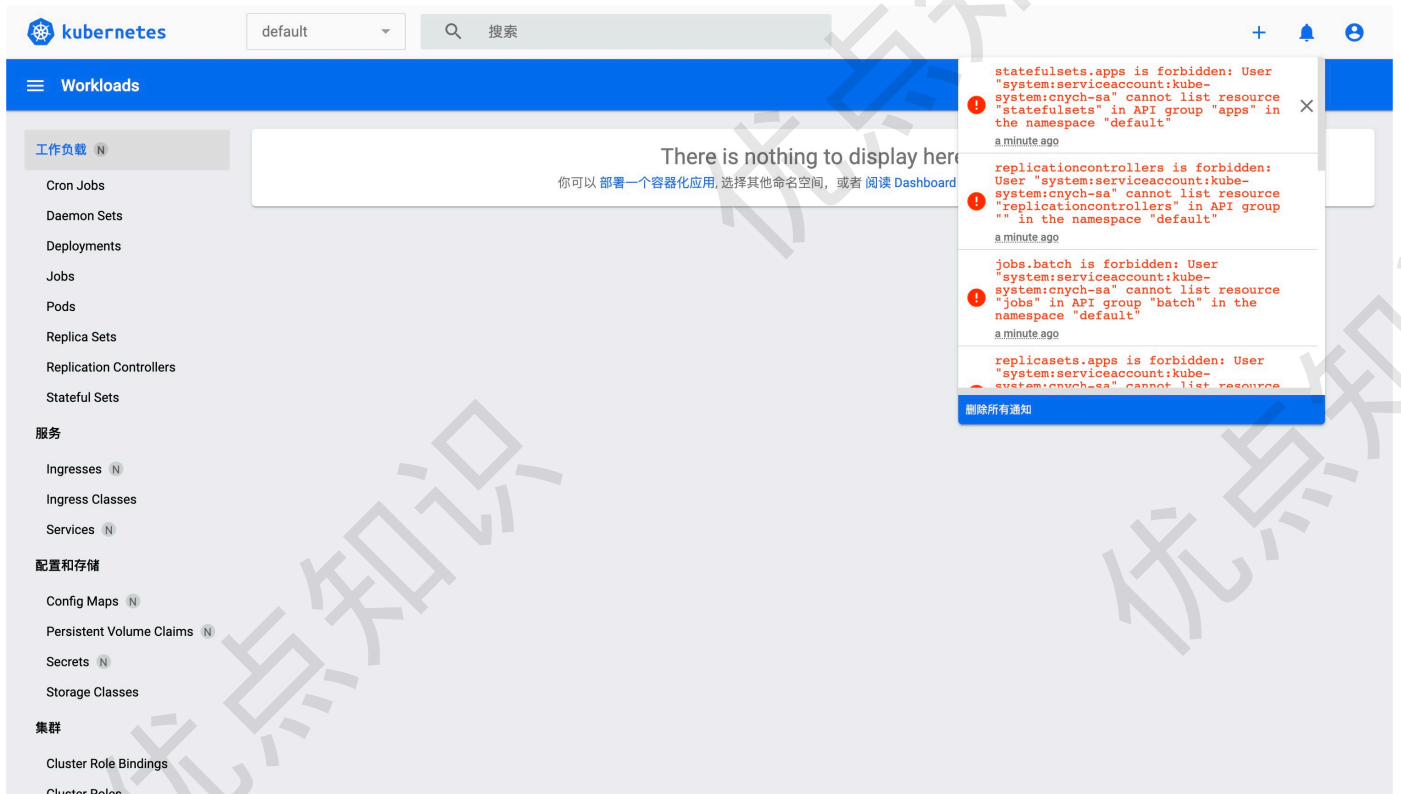
```
# cnych-token.yaml
apiVersion: v1
kind: Secret
metadata:
  name: cnych-sec
  namespace: kube-system
  annotations:
    kubernetes.io/service-account.name: cnych-sa
type: kubernetes.io/service-account-token
```

也可以直接 `kubectl create token <sa>` 命令来创建 token。

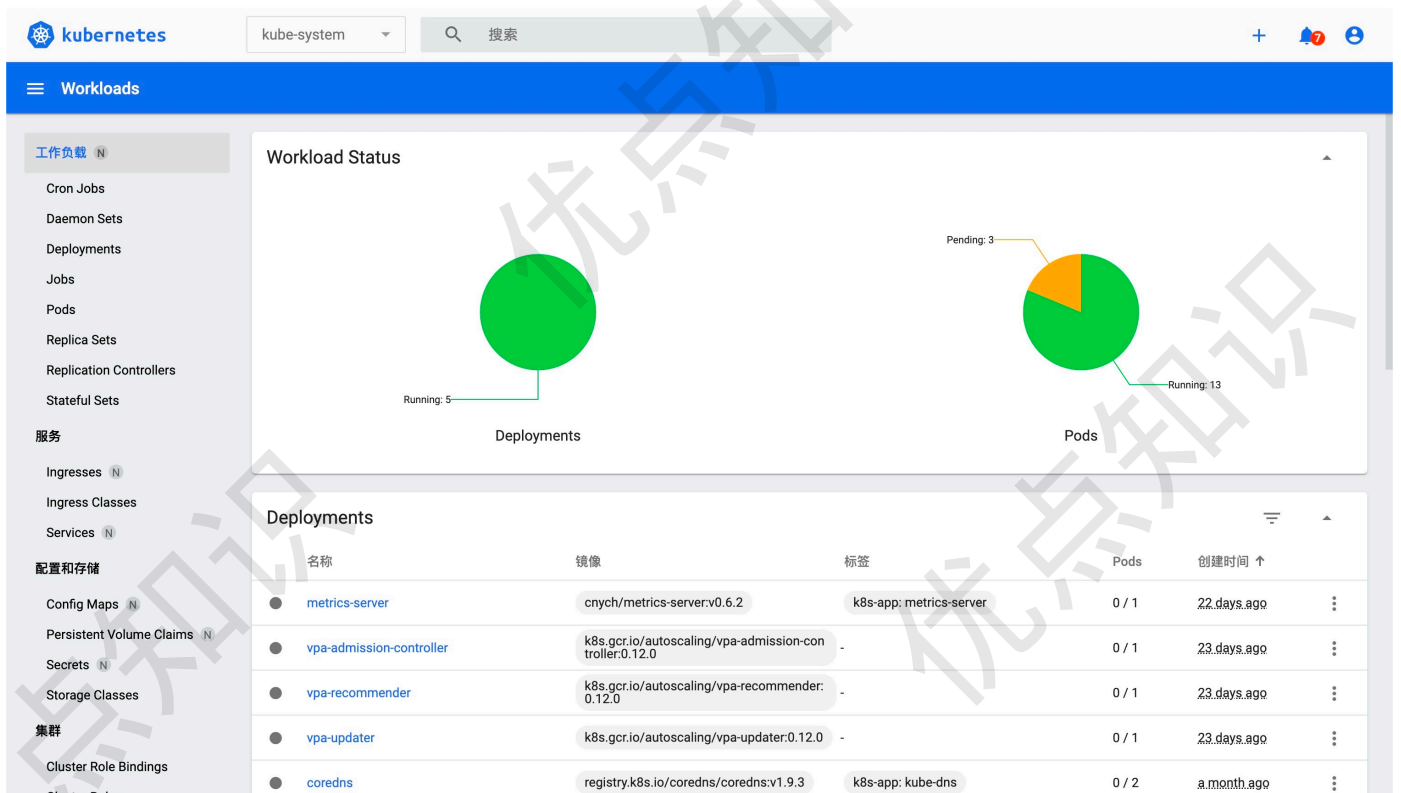
创建后我们就可以使用该 Secret 对象包含的 token 去登录我们的 Dashboard。

```
❖ → kubectl get secret cnych-sec -o jsonpath={.data.token} -n kube-system |base64 -d
eyJhbGciOiJIUzUxNiIsImtpZCI6ImdnazJLZENQTktiZkxVejhnaMnhmTHJYRTlkZ2ZnOHJGQmgwVW4td3Bwd0ki
fQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWVudC9u
YW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWJyY291bnQvc2VjcmV0Lm5hb
WUiOiJjb2lnLjaC1zZW50Lm5ldGVzLm5ldGVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC5uYW1l
IjoieY255Y2gtc2EiLCJrdWJlcm5ldGVzLm5ldGVzL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UyYWNjb3VudC51
aWY2ZlNTVkc0Z0ZjLTRmZGMtOWJmNi1mMmRhNjIyZDc2ZjMiLCJzdWIiOiJzeXN0ZW06c2VydmljZWJyY291bnQ6
a3ViZS1zeXN0ZW06Y255Y2gtc2EifQ.D-0L_KhbKYLdDnw7jdrvozb41thihaIJ7pJq0sMqM-0GNnb-
90GxmyfH0N7e0hBgJNbISmC8qCSP6oXTsJUV9uaI_aVP7bJX0ailZ7l0CjYg_aTGkdDUefDazFR1Ifzij-7-
pt0SeCqN8C-gX5qKN3mDixBJhp2m8jpTwrFSK9eN4f3gs59zjxIB7rKqk0ChnRGYsaW6ecjV5ytUqn0BWL-
3KfQh_13oYkyCqMGApRnzTZ7msoXqsUMGaqTHNeoHzmHqirRpR87SZffJaJYY6CvEbMEnSQ0ke-
nUiU4r_n4g90ASAX37G9ePMSCMNGDz40QTUdxttWD_WXKM8MbMg
```

使用这里的 token 去 Dashboard 页面进行登录：



我们可以看到上面的提示信息说我们现在使用的这个 ServiceAccount 没有权限获取当前命名空间下面的资源对象，这是因为我们登录进来后默认跳转到 default 命名空间，我们切换到 kube-system 命名空间下面就可以了（需要手动修改浏览器中的 url 地址）：



我们可以看到可以访问 pod 列表了，但是也会有一些其他额外的提示：events is forbidden: User "system:serviceaccount:kube-system:cnych-sa" cannot list events in the namespace "kube-system"，这是因为当前登录用只被授权了访问 pod 和 deployment 的权限，同样的，访问下 deployment 看看可以了吗？

同样的，你可以根据自己的需求来对访问用户的权限进行限制，可以自己通过 `Role` 定义更加细粒度的权限，也可以使用系统内置的一些权限.....

可以全局访问的 `ServiceAccount`

刚刚我们创建的 `cnych-sa` 这个 `ServiceAccount` 和一个 `Role` 角色进行绑定的，如果我们现在创建一个新的 `ServiceAccount`，需要他操作的权限作用于所有的 `namespace`，这个时候我们就需要使用到 `ClusterRole` 和 `ClusterRoleBinding` 这两种资源对象了。同样，首先新建一个 `ServiceAccount` 对象：

```
# cnych-sa2.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cnych-sa2
  namespace: kube-system
```

创建：

```
❖ → kubectl create -f cnych-sa2.yaml
```

然后创建一个 `ClusterRoleBinding` 对象

```
# cnych-clusterrolebinding.yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cnych-sa2-clusterrolebinding
subjects:
- kind: ServiceAccount
  name: cnych-sa2
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

从上面我们可以看到我们没有为这个资源对象声明 `namespace`，因为这是一个 `ClusterRoleBinding` 资源对象，是作用于整个集群的，我们也没有单独新建一个 `ClusterRole` 对象，而是使用的 `cluster-admin` 这个对象，这是 Kubernetes 集群内置的 `ClusterRole` 对象，我们可以使用 `kubectl get clusterrole` 和 `kubectl get clusterrolebinding` 查看系统内置的一些集群角色和集群角色绑定，这里我们使用的 `cluster-admin` 这个集群角色是拥有最高权限的集群角色，所以一般需要谨慎使用该集群角色。

创建上面集群角色绑定资源对象，然后我们可以使用 `kubectl create token` 命令来创建一个新的 `token` 去登录 Dashboard：

可以看到使用该 token 登录后没有出现任何权限相关错误了：

