26 | 基于角色的权限控制：RBAC

26 | 基于角色的权限控制：RBAC

张磊 2018-10-22

14:02

讲述：张磊 大小：6.44M

你好，我是张磊。今天我和你分享的主题是：基于角色的权限控制之 RBAC。

在前面的文章中，我已经为你讲解了很多种 Kubernetes 内置的编排对象，以及对应的控制器模式的实现原理。此外，我还剖析了自定义 API 资源类型和控制器的编写方式。

这时候，你可能已经冒出了这样一个想法：控制器模式看起来好像也不难嘛，我能不能自己写一个编排对象呢？

答案当然是可以的。而且，这才是 Kubernetes 项目最具吸引力的地方。

毕竟，在互联网级别的大规模集群里，Kubernetes 内置的编排对象，很难做到完全满足所有需求。所以，很多实际的容器化工作，都会要求你设计一个自己的编排

对象，实现自己的控制器模式。

而在 Kubernetes 项目里，我们可以基于插件机制来完成这些工作，而完全不需要修改任何一行代码。

不过，你要通过一个外部插件，在 Kubernetes 里新增和操作 API 对象，那么就必须先了解一个非常重要的知识：RBAC。

我们知道，Kubernetes 中所有的 API 对象，都保存在 Etcd 里。可是，对这些 API 对象的操作，却一定都是通过访问 kube-apiserver 实现的。其中一个非常重要的原因，就是你需要 APIServer 来帮助你做授权工作。

而**在 Kubernetes 项目中，负责完成授权（Authorization）工作的机制，就是 RBAC**：基于角色的访问控制（Role-Based Access Control）。

如果你直接查看 Kubernetes 项目中关于 RBAC 的文档的话，可能会感觉非常复杂。但实际上，等到你用到这些 RBAC 的细节时，再去查阅也不迟。

而在这里，我只希望你能明确三个最基本的概念。

1. Role：角色，它其实是一组规则，定义了一组对 Kubernetes API 对象的操作权限。
2. Subject：被作用者，既可以是“人”，也可以是“机器”，也可以使你在 Kubernetes 里定义的“用户”。
3. RoleBinding：定义了“被作用者”和“角色”的绑定关系。

而这三个概念，其实就是整个 RBAC 体系的核心所在。

我先来讲解一下 Role。

实际上，Role 本身就是一个 Kubernetes 的 API 对象，定义如下所示：

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
namespace: mynamespace
name: example-role
rules:
- apiGroups: [""]
resources: ["pods"]
verbs: ["get", "watch", "list"]
```

□复制代码

首先，这个 Role 对象指定了它能产生作用的 Namepace 是：mynamespace。

Namespace 是 Kubernetes 项目里的一个逻辑管理单位。不同 Namespace 的 API 对象，在通过 kubectl 命令进行操作的时候，是互相隔离开的。

比如，kubectl get pods -n mynamespace。

当然，这仅限于逻辑上的“隔离”，Namespace 并不会提供任何实际的隔离或者多租户能力。而在前面文章中用到的大多数例子里，我都没有指定 Namespace，那就是使用的是默认 Namespace：default。

然后，这个 Role 对象的 rules 字段，就是它所定义的权限规则。在上面的例子里，这条规则的含义就是：允许“被作用者”，对 mynamespace 下面的 Pod 对象，进行 GET、WATCH 和 LIST 操作。

那么，这个具体的“被作用者”又是如何指定的呢？这就需要通过 RoleBinding 来实现了。

当然，RoleBinding 本身也是一个 Kubernetes 的 API 对象。它的定义如下所示：

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: example-rolebinding
namespace: mynamespace
subjects:
- kind: User
name: example-user
apiGroup: rbac.authorization.k8s.io
roleRef:
kind: Role
name: example-role
apiGroup: rbac.authorization.k8s.io
```

□复制代码

可以看到，这个 RoleBinding 对象里定义了一个 subjects 字段，即“被作用者”。它的类型是 User，即 Kubernetes 里的用户。这个用户的名字是 example-user。

可是，在 Kubernetes 中，其实并没有一个叫作“User”的 API 对象。而且，我们在前面和部署使用 Kubernetes 的流程里，既不需要 User，也没有创建过 User。

这个 User 到底是从哪里来的呢?

实际上，Kubernetes 里的“User”，也就是“用户”，只是一个授权系统里的逻辑概念。它需要通过外部认证服务，比如 Keystone，来提供。或者，你也可以直接给 APIServer 指定一个用户名、密码文件。那么 Kubernetes 的授权系统，就能够从这个文件里找到对应的“用户”了。当然，在大多数私有的使用环境中，我们只要使用 Kubernetes 提供的内置“用户”，就足够了。这部分知识，我后面马上会讲到。

接下来，我们会看到一个 roleRef 字段。正是通过这个字段，RoleBinding 对象就可以直接通过名字，来引用我们前面定义的 Role 对象（example-role），从而定义了“被作用者（Subject）”和“角色（Role）”之间的绑定关系。

需要再次提醒的是，Role 和 RoleBinding 对象都是 Namespaced 对象（Namespaced

Object），它们对权限的限制规则仅在它们自己的 Namespace 内有效，roleRef 也只能引用当前 Namespace 里的 Role 对象。

那么，**对于非 Namespaced（Non-namespaced）对象（比如：Node），或者，某一个 Role 想要作用于所有的 Namespace 的时候，我们又该如何去做授权呢?**

这时候，我们就必须要使用 ClusterRole 和 ClusterRoleBinding 这两个组合了。这两个 API 对象的用法跟 Role 和 RoleBinding 完全一样。只不过，它们的定义里，没有了 Namespace 字段，如下所示：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: example-clusterrole
rules:
- apiGroups: [""]
resources: ["pods"]
verbs: ["get", "watch", "list"]
```

复制代码

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: example-clusterrolebinding
subjects:
- kind: User
name: example-user
apiGroup: rbac.authorization.k8s.io
roleRef:
kind: ClusterRole
name: example-clusterrole
apiGroup: rbac.authorization.k8s.io
```

□复制代码

上面的例子里的 ClusterRole 和 ClusterRoleBinding 的组合，意味着名叫 example-user 的用户，拥有对所有 Namespace 里的 Pod 进行 GET、WATCH 和 LIST 操作的权限。

更进一步地，在 Role 或者 ClusterRole 里面，如果要赋予用户 example-user 所有权限，那你就可以给它指定一个 verbs 字段的全集，如下所示：

```
verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

□复制代码

这些就是当前 Kubernetes（v1.11）里能够对 API 对象进行的所有操作了。

类似的，Role 对象的 rules 字段也可以进一步细化。比如，你可以只针对某一个具体的对象进行权限设置，如下所示：

```
rules:
- apiGroups: [""]
resources: ["configmaps"]
resourceNames: ["my-config"]
verbs: ["get"]
```

□复制代码

这个例子就表示，这条规则的“被作用者”，只对名叫“my-config”的 ConfigMap 对象，有进行 GET 操作的权限。

而正如我前面介绍过的，在大多数时候，我们其实都不太使用“用户”这个功能，而是直接使用 Kubernetes 里的“内置用户”。

这个由 Kubernetes 负责管理的“内置用户”，正是我们前面曾经提到过的：ServiceAccount。

接下来，我通过一个具体的实例来为你讲解一下为 ServiceAccount 分配权限的过程。

首先，我们要定义一个 ServiceAccount。它的 API 对象非常简单，如下所示：

```
apiVersion: v1
kind: ServiceAccount
metadata:
namespace: mynamespace
name: example-sa
```

□复制代码

可以看到，一个最简单的 ServiceAccount 对象只需要 Name 和 Namespace 这两个最基本的字段。

然后，我们通过编写 RoleBinding 的 YAML 文件，来为这个 ServiceAccount 分配权限：

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: example-rolebinding
namespace: mynamespace
subjects:
- kind: ServiceAccount
name: example-sa
namespace: mynamespace
roleRef:
kind: Role
name: example-role
apiGroup: rbac.authorization.k8s.io
```

□复制代码

可以看到，在这个 RoleBinding 对象里，subjects 字段的类型（kind），不再是一个 User，而是一个名叫 example-sa 的 ServiceAccount。而 roleRef 引用的 Role 对象，依然名叫 example-role，也就是我在这篇文章一开始定义的 Role 对象。

接着，我们用 kubectl 命令创建这三个对象：

```
$ kubectl create -f svc-account.yaml
$ kubectl create -f role-binding.yaml
$ kubectl create -f role.yaml
```

□复制代码

然后，我们来查看一下这个 ServiceAccount 的详细信息：

```
$ kubectl get sa -n mynamespace -o yaml
- apiVersion: v1
kind: ServiceAccount
metadata:
creationTimestamp: 2018-09-08T12:59:17Z
name: example-sa
namespace: mynamespace
resourceVersion: "409327"
...
secrets:
- name: example-sa-token-vmfg6
```

□复制代码

可以看到，Kubernetes 会为一个 ServiceAccount 自动创建并分配一个 Secret 对象，即：上述 ServiceAcount 定义里最下面的 secrets 字段。

这个 Secret，就是这个 ServiceAccount 对应的、用来跟 APIServer 进行交互的授权文件，我们一般称它为：Token。Token 文件的内容一般是证书或者密码，它以一个 Secret 对象的方式保存在 Etcd 当中。

这时候，用户的 Pod，就可以声明使用这个 ServiceAccount 了，比如下面这个例子：

```
apiVersion: v1
kind: Pod
metadata:
```

```
namespace: mynamespace
name: sa-token-test
spec:
containers:
- name: nginx
image: nginx:1.7.9
serviceAccountName: example-sa
```

□复制代码

在这个例子里，我定义了 Pod 要使用的要使用的 ServiceAccount 的名字是：example-sa。

等这个 Pod 运行起来之后，我们就可以看到，该 ServiceAccount 的 token，也就是一个 Secret 对象，被 Kubernetes 自动挂载到了容器的 /var/run/secrets/kubernetes.io/serviceaccount 目录下，如下所示：

```
$ kubectl describe pod sa-token-test -n mynamespace
Name: sa-token-test
Namespace: mynamespace
...
Containers:
nginx:
...
Mounts:
/var/run/secrets/kubernetes.io/serviceaccount from example-sa-token-vmfg6 (ro)
```

□复制代码

这时候，我们可以通过 kubectl exec 查看到这个目录里的文件：

```
$ kubectl exec -it sa-token-test -n mynamespace -- /bin/bash
root@sa-token-test:/# ls /var/run/secrets/kubernetes.io/serviceaccount
ca.crt namespace token
```

□复制代码

如上所示，容器里的应用，就可以使用这个 ca.crt 来访问 APIServer 了。更重要的是，此时它只能够做 GET、WATCH 和 LIST 操作。因为 example-sa 这个 ServiceAccount 的权限，已经被我们绑定了 Role 做了限制。

此外，我在第 15 篇文章《深入解析 Pod 对象（二）：使用进阶》中曾经提到过，如果一个 Pod 没有声明 serviceAccountName，Kubernetes 会自动在它的 Namespace 下创建一个名叫 default 的默认 ServiceAccount，然后分配给这个 Pod。

但在这种情况下，这个默认 ServiceAccount 并没有关联任何 Role。也就是说，此时它有访问 APIServer 的绝大多数权限。当然，这个访问所需要的 Token，还是默认 ServiceAccount 对应的 Secret 对象为它提供的，如下所示。

```
$kubectl describe sa default
Name: default
Namespace: default
Labels: <none>
Annotations: <none>
Image pull secrets: <none>
Mountable secrets: default-token-s8rbq
Tokens: default-token-s8rbq
Events: <none>
$ kubectl get secret
NAME TYPE DATA AGE
kubernetes.io/service-account-token 3 82d
$ kubectl describe secret default-token-s8rbq
Name: default-token-s8rbq
Namespace: default
Labels: <none>
Annotations: kubernetes.io/service-account.name=default
kubernetes.io/service-account.uid=ffcb12b2-917f-11e8-abde-42010aa80002
Type: kubernetes.io/service-account-token
Data
====
ca.crt: 1025 bytes
namespace: 7 bytes
token: <TOKEN 数据 >
```

□复制代码

可以看到，Kubernetes 会自动为默认 ServiceAccount 创建并绑定一个特殊的 Secret：它的类型是kubernetes.io/service-account-token；它的 Annotation 字段，声明了kubernetes.io/service-account.name=default，即这个 Secret 会跟同一 Namespace 下名叫 default 的 ServiceAccount 进行绑定。

所以，在生产环境中，我强烈建议你为所有 Namespace 下的默认 ServiceAccount，绑定一个只读权限的 Role。这个具体怎么做，就当做思考题留给你了。

除了前面使用的“用户”（User），Kubernetes 还拥有“用户组”（Group）的概念，也就是一组“用户”的意思。如果你为 Kubernetes 配置了外部认证服务的话，这个“用户组”的概念就会由外部认证服务提供。

而对于 Kubernetes 的内置“用户”ServiceAccount 来说，上述“用户组”的概念也同样适用。

实际上，一个 ServiceAccount，在 Kubernetes 里对应的“用户”的名字是：

```
system:serviceaccount:<ServiceAccount 名字 >
```

复制代码

而它对应的内置“用户组”的名字，就是：

```
system:serviceaccounts:<Namespace 名字 >
```

复制代码

这两个对应关系，请你一定要牢记。

比如，现在我们可以在 RoleBinding 里定义如下的 subjects：

```
subjects:
- kind: Group
name: system:serviceaccounts:mynamespace
apiGroup: rbac.authorization.k8s.io
```

复制代码

这就意味着这个 Role 的权限规则，作用于 mynamespace 里的所有 ServiceAccount。这就用到了“用户组”的概念。

而下面这个例子：

```
subjects:
- kind: Group
```

```
    name: system:serviceaccounts
    apiGroup: rbac.authorization.k8s.io
```

□复制代码

就意味着这个 Role 的权限规则，作用于整个系统里的所有 ServiceAccount。

最后，值得一提的是，**在 Kubernetes 中已经内置了很多个为系统保留的 ClusterRole，它们的名字都以 system: 开头。**你可以通过 kubectl get clusterroles 查看到它们。

一般来说，这些系统 ClusterRole，是绑定给 Kubernetes 系统组件对应的 ServiceAccount 使用的。

比如，其中一个名叫 system:kube-scheduler 的 ClusterRole，定义的权限规则是 kube-scheduler（Kubernetes 的调度器组件）运行所需要的必要权限。你可以通过如下指令查看这些权限的列表：

```
    $ kubectl describe clusterrole system:kube-scheduler
    Name: system:kube-scheduler
    ...
    PolicyRule:
    Resources Non-Resource URLs Resource Names Verbs
    --------- ----------------- -------------- -----
    ...
    services [] [] [get list watch]
    replicasets.apps [] [] [get list watch]
    statefulsets.apps [] [] [get list watch]
    replicasets.extensions [] [] [get list watch]
    poddisruptionbudgets.policy [] [] [get list watch]
    pods/status [] [] [patch update]
```

□复制代码

这个 system:kube-scheduler 的 ClusterRole，就会被绑定给 kube-system Namesapce 下名叫 kube-scheduler 的 ServiceAccount，它正是 Kubernetes 调度器的 Pod 声明使用的 ServiceAccount。

除此之外，Kubernetes 还提供了四个预先定义好的 ClusterRole 来供用户直接使用：

1. cluster-admin；
2. admin；

3. edit；
4. view。

通过它们的名字，你应该能大致猜出它们都定义了哪些权限。比如，这个名叫 view 的 ClusterRole，就规定了被作用者只有 Kubernetes API 的只读权限。

而我还要提醒你的是，上面这个 cluster-admin 角色，对应的是整个 Kubernetes 项目中的最高权限（verbs=*），如下所示：

```
$ kubectl describe clusterrole cluster-admin -n kube-system
Name: cluster-admin
Labels: kubernetes.io/bootstrapping=rbac-defaults
Annotations: rbac.authorization.kubernetes.io/autoupdate=true
PolicyRule:
Resources Non-Resource URLs Resource Names Verbs
--------- ----------------- -------------- -----
*.* [] [] [*]
[*] [] [*]
```

复制代码

所以，请你务必要谨慎而小心地使用 cluster-admin。

总结

在今天这篇文章中，我主要为你讲解了基于角色的访问控制（RBAC）。

其实，你现在已经能够理解，所谓角色（Role），其实就是一组权限规则列表。而我们分配这些权限的方式，就是通过创建 RoleBinding 对象，将被作用者（subject）和权限列表进行绑定。

另外，与之对应的 ClusterRole 和 ClusterRoleBinding，则是 Kubernetes 集群级别的 Role 和 RoleBinding，它们的作用范围不受 Namespace 限制。

而尽管权限的被作用者可以有很多种（比如，User、Group 等），但在我们平常的使用中，最普遍的用法还是 ServiceAccount。所以，Role + RoleBinding + ServiceAccount 的权限分配方式是你要重点掌握的内容。我们在后面编写和安装各种插件的时候，会经常用到这个组合。

思考题

请问，如何为所有 Namespace 下的默认 ServiceAccount（default ServiceAccount），绑定一个只读权限的 Role 呢？请你提供 ClusterRoleBinding（或者 RoleBinding）的 YAML 文件。