

讲堂 > 深入剖析Kubernetes > 文章详情

26 | 基于角色的权限控制：RBAC

2018-10-22 张磊



26 | 基于角色的权限控制：RBAC

朗读人：张磊 14'02" | 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：

```
1 kind: Role
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   namespace: mynamespace
5   name: example-role
6 rules:
7 - apiGroups: [""]
8   resources: ["pods"]
9   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 Namespace 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：

```
1 kind: RoleBinding
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: example-rolebinding
5   namespace: mynamespace
6 subjects:
7 - kind: User
8   name: example-user
9   apiGroup: rbac.authorization.k8s.io
10 roleRef:
11   kind: Role
12   name: example-role
13   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：

```
1 kind: ClusterRole
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: example-clusterrole
5 rules:
6 - apiGroups: [""]
7   resources: ["pods"]
8   verbs: ["get", "watch", "list"]
```

[复制代码](#)

```
1 kind: ClusterRoleBinding
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: example-clusterrolebinding
5 subjects:
6 - kind: User
7   name: example-user
8   apiGroup: rbac.authorization.k8s.io
9 roleRef:
10  kind: ClusterRole
11  name: example-clusterrole
12  apiGroup: rbac.authorization.k8s.io
```

[复制代码](#)

上面的例子中的 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，那你就给它指定一个 verbs 字段的全集，如下所示：

```
1 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：

```
1 rules:
2   - apiGroups: [""]
3     resources: ["configmaps"]
4     resourceNames: ["my-config"]
5     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：

```
1 apiVersion: v1
2 kind: ServiceAccount
3 metadata:
4   namespace: mynamespace
5   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：

```
1 kind: RoleBinding
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: example-rolebinding
5   namespace: mynamespace
6 subjects:
7   - kind: ServiceAccount
8     name: example-sa
9     namespace: mynamespace
10 roleRef:
11   kind: Role
```

[复制代码](#)


```
12   name: example-role
13   apiGroup: rbac.authorization.k8s.io
```

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：

```
1 $ kubectl create -f svc-account.yaml
2 $ kubectl create -f role-binding.yaml
3 $ 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：

```
1 $ kubectl get sa -n mynamespace -o yaml
2 - apiVersion: v1
3   kind: ServiceAccount
4   metadata:
5     creationTimestamp: 2018-09-08T12:59:17Z
6     name: example-sa
7     namespace: mynamespace
8     resourceVersion: "409327"
9     ...
10  secrets:
11    - 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 ServiceAccount 定义里最下面的 secrets 字段。

这个 Secret，就是这个 ServiceAccount 对应的、用来跟 API Server 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：

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   namespace: mynamespace
5   name: sa-token-test
6 spec:
7   containers:
8     - name: nginx
```

[复制代码](#)

```
9     image: nginx:1.7.9
10    serviceAccountName: example-sa
```

在这个例子里，我定义了 Pod 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：

```
1 $ kubectl describe pod sa-token-test -n mynamespace
2 Name:          sa-token-test
3 Namespace:     mynamespace
4 ...
5 Containers:
6   nginx:
7     ...
8   Mounts:
9     /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：

```
1 $ kubectl exec -it sa-token-test -n mynamespace -- /bin/bash
2 root@sa-token-test:/# ls /var/run/secrets/kubernetes.io/serviceaccount
3 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 API Server 了。更重要的是，此时它只能做 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 API Server 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。

```
1 $ kubectl describe sa default
2 Name:          default
3 Namespace:     default
4 Labels:        <none>
5 Annotations:   <none>
6 Image pull secrets: <none>
```

[复制代码](#)

```

7 Mountable secrets:  default-token-s8rbq
8 Tokens:             default-token-s8rbq
9 Events:             <none>
10
11 $ kubectl get secret
12 NAME                                TYPE                                DATA    AGE
13 kubernetes.io/service-account-token  3                                82d
14
15 $ kubectl describe secret default-token-s8rbq
16 Name:                             default-token-s8rbq
17 Namespace:                         default
18 Labels:                           <none>
19 Annotations: kubernetes.io/service-account.name=default
20                  kubernetes.io/service-account.uid=ffcb12b2-917f-11e8-abde-42010aa80002
21
22 Type: kubernetes.io/service-account-token
23
24 Data
25 ====
26 ca.crt:      1025 bytes
27 namespace:   7 bytes
28 token:       <TOKEN 数据 >

```

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：

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

[复制代码](#)

而它对应的内置“用户组”的名字，就是：

[复制代码](#)


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

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：

```
1 subjects:
2 - kind: Group
3   name: system:serviceaccounts:mynamespace
4   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：

```
1 subjects:
2 - kind: Group
3   name: system:serviceaccounts
4   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：

```
1 $ kubectl describe clusterrole system:kube-scheduler
2 Name:          system:kube-scheduler
3 ...
4 PolicyRule:
5   Resources          Non-Resource URLs  Resource Names      Verbs
6   -----
7   ...
8   services           []                 []                  [get list watch]
9   replicaset.apps    []                 []                  [get list watch]
10  statefulsets.apps   []                 []                  [get list watch]
11  replicaset.extensions []                 []                  [get list watch]
```

[复制代码](#)

12	poddisruptionbudgets.policy	[]	[]	[get list watch]
13	pods/status	[]	[]	[patch update]

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，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：

```

1 $ kubectl describe clusterrole cluster-admin -n kube-system
2 Name:          cluster-admin
3 Labels:        kubernetes.io/bootstrapping=rbac-defaults
4 Annotations:   rbac.authorization.kubernetes.io/autoupdate=true
5 PolicyRule:
6   Resources      Non-Resource URLs  Resource Names  Verbs
7   -----      -
8   *.*           []                  []              [*]
9               [*]                  []              [*]
```

[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。

感谢你的收听，欢迎你给我留言，也欢迎分享给更多的朋友一起阅读。



版权归极客邦科技所有，未经许可不得转载

写留言

精选留言



kyleqian

0

好像可以通过RoleBinding去绑定ClusterRole，这样能拥有后者的所有权限，但是只局限在所属namespace中。好像是这样。

2018-10-22



Geek_zz

0

rbac 和token是怎么联系的呢

2018-10-22



Ethan

赞一个

2018-10-22



huan

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: readonly-all-default

subjects:

- kind: User

name: system.serviceaccount.default

roleRef:

kind: ClusterRole

name: view

apiGroup: rbac.authorization.k8s.io

2018-10-22

