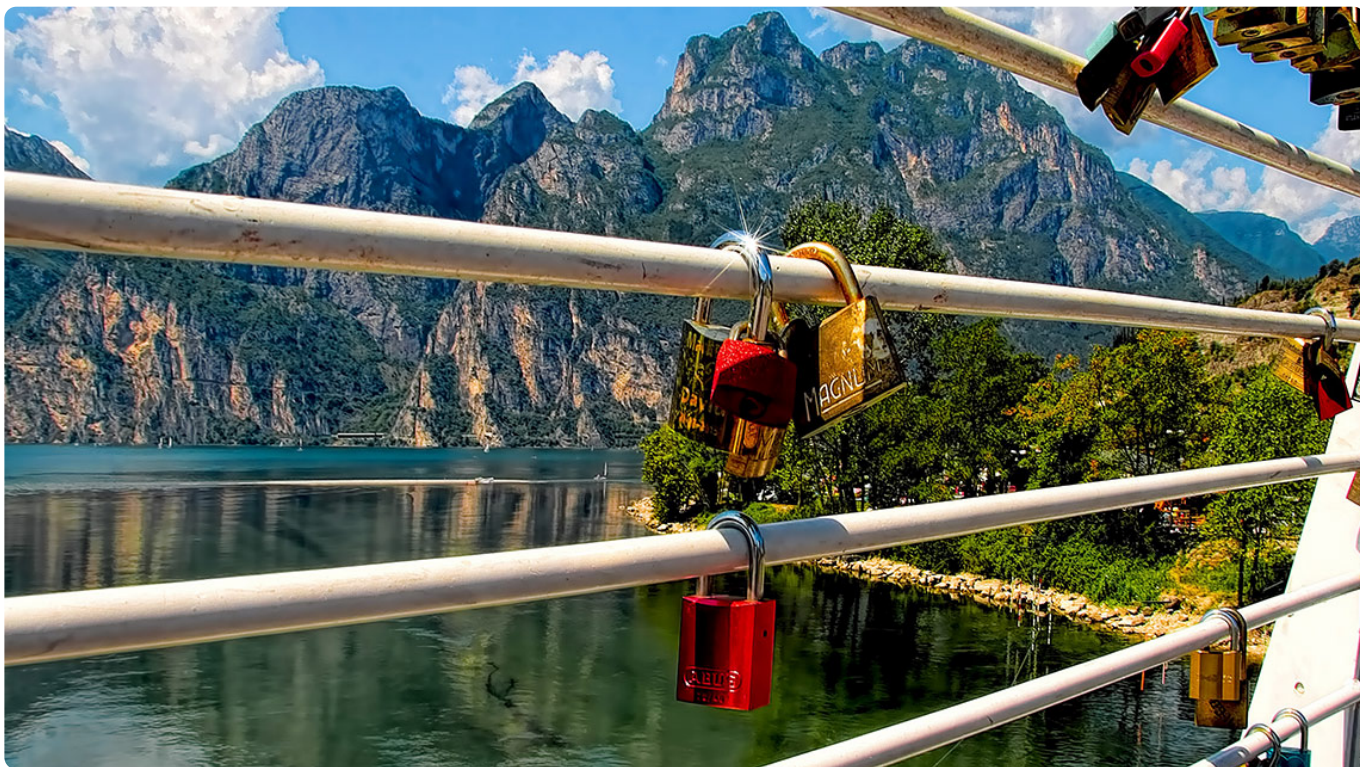


24 | 深入解析声明式API（一）：API对象的奥秘

2018-10-17 张磊

深入剖析Kubernetes

[进入课程 >](#)



讲述：张磊

时长 18:43 大小 8.58M



你好，我是张磊。今天我和你分享的主题是：深入解析声明式 API 之 API 对象的奥秘。

在上一篇文章中，我为你详细讲解了 Kubernetes 声明式 API 的设计、特点，以及使用方式。

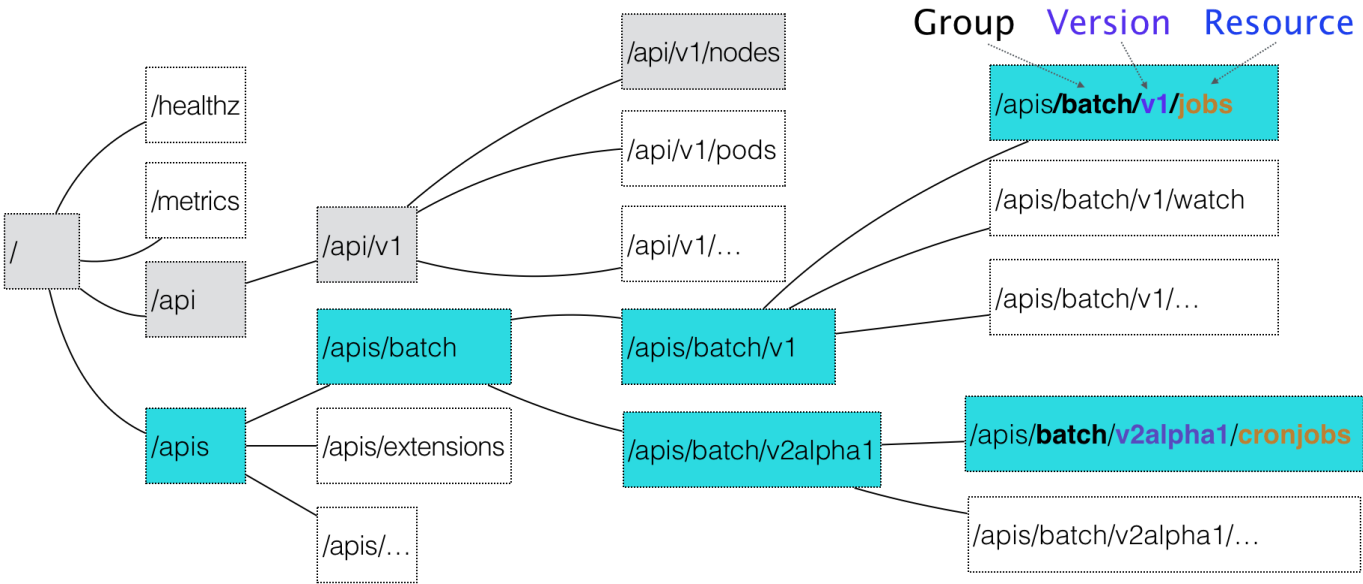
而在今天这篇文章中，我就来为你讲解一下 Kubernetes 声明式 API 的工作原理，以及如何利用这套 API 机制，在 Kubernetes 里添加自定义的 API 对象。

你可能一直就很好奇：当我把一个 YAML 文件提交给 Kubernetes 之后，它究竟是如何创建一个 API 对象的呢？

这得从声明式 API 的设计谈起了。

在 Kubernetes 项目中，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：



在这幅图中，你可以很清楚地看到 Kubernetes 里 API 对象的组织方式，其实是层层递进的。

比如，现在我要声明要创建一个 CronJob 对象，那么我的 YAML 文件的开始部分会这么写：

复制代码

```
1 apiVersion: batch/v2alpha1
2 kind: CronJob
3 ...
```

在这个 YAML 文件中，“CronJob”就是这个 API 对象的资源类型（Resource），“batch”就是它的组（Group），v2alpha1 就是它的版本（Version）。

当我们提交了这个 YAML 文件之后，Kubernetes 就会把这个 YAML 文件里描述的内容，转换成 Kubernetes 里的一个 CronJob 对象。

那么，Kubernetes 是如何对 Resource、Group 和 Version 进行解析，从而在 Kubernetes 项目里找到 CronJob 对象的定义呢？

首先，Kubernetes 会匹配 API 对象的组。

需要明确的是，对于 Kubernetes 里的核心 API 对象，比如：Pod、Node 等，是不需要 Group 的（即：它们 Group 是 ""）。所以，对于这些 API 对象来说，Kubernetes 会直接在 /api 这个层级进行下一步的匹配过程。

而对于 CronJob 等非核心 API 对象来说，Kubernetes 就必须在 /apis 这个层级里查找它对应的 Group，进而根据 "batch" 这个 Group 的名字，找到 /apis/batch。

不难发现，这些 API Group 的分类是以对象功能为依据的，比如 Job 和 CronJob 就都属于 "batch"（离线业务）这个 Group。

然后，Kubernetes 会进一步匹配到 API 对象的版本号。

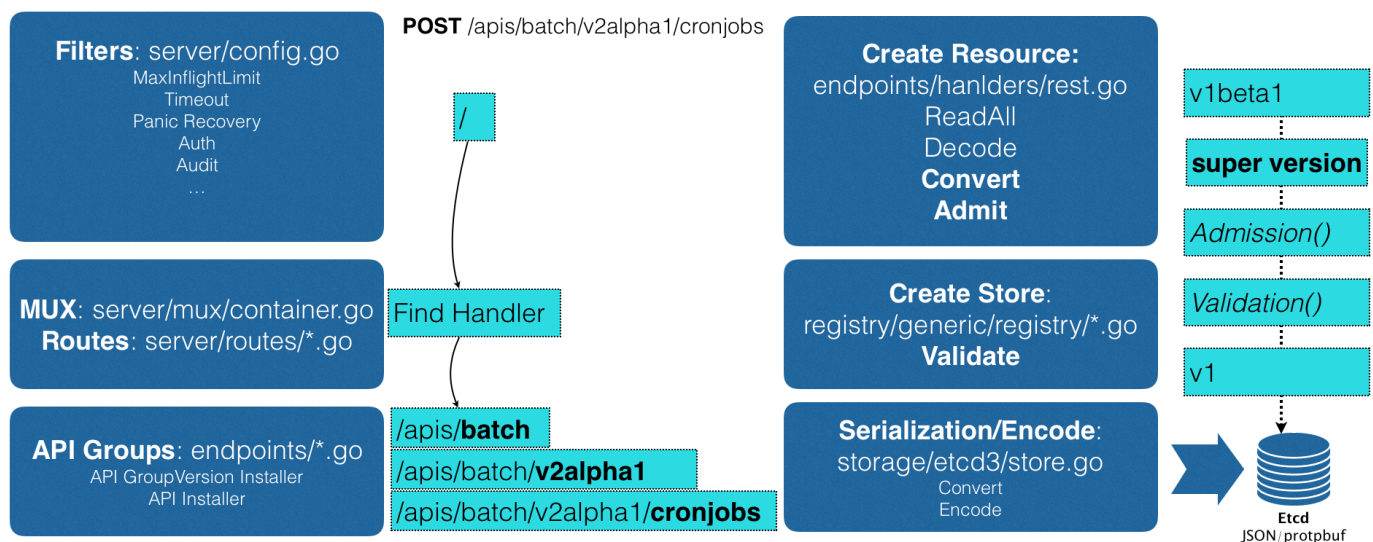
对于 CronJob 这个 API 对象来说，Kubernetes 在 batch 这个 Group 下，匹配到的版本号就是 v2alpha1。

在 Kubernetes 中，同一种 API 对象可以有多个版本，这正是 Kubernetes 进行 API 版本化管理的重要手段。这样，比如在 CronJob 的开发过程中，对于会影响到用户的变更就可以通过升级新版本来处理，从而保证了向后兼容。

最后，Kubernetes 会匹配 API 对象的资源类型。

在前面匹配到正确的版本之后，Kubernetes 就知道，我要创建的原来是一个 /apis/batch/v2alpha1 下的 CronJob 对象。

这时候，APIServer 就可以继续创建这个 CronJob 对象了。为了方便理解，我为你总结了一个如下所示流程图来阐述这个创建过程：



首先，当我们发起了创建 CronJob 的 POST 请求之后，我们编写的 YAML 的信息就被提交给了 API Server。

而 API Server 的第一个功能，就是过滤这个请求，并完成一些前置性的工作，比如授权、超时处理、审计等。

然后，请求会进入 MUX 和 Routes 流程。如果你编写过 Web Server 的话就会知道，MUX 和 Routes 是 API Server 完成 URL 和 Handler 绑定的场所。而 API Server 的 Handler 要做的事情，就是按照我刚刚介绍的匹配过程，找到对应的 CronJob 类型定义。

接着，API Server 最重要的职责就来了：根据这个 CronJob 类型定义，使用用户提交的 YAML 文件里的字段，创建一个 CronJob 对象。

而在这个过程中，API Server 会进行一个 Convert 工作，即：把用户提交的 YAML 文件，转换成一个叫作 Super Version 的对象，它正是该 API 资源类型所有版本的字段全集。这样用户提交的不同版本的 YAML 文件，就都可以用这个 Super Version 对象来进行处理了。

接下来，API Server 会先后进行 `Admission()` 和 `Validation()` 操作。比如，我在上一篇文章中提到的 Admission Controller 和 Initializer，就都属于 Admission 的内容。

而 `Validation`，则负责验证这个对象里的各个字段是否合法。这个被验证过的 API 对象，都保存在了 API Server 里一个叫作 Registry 的数据结构中。也就是说，只要一个 API 对象的定义能在 Registry 里查到，它就是一个有效的 Kubernetes API 对象。

最后，APIServer 会把验证过的 API 对象转换成用户最初提交的版本，进行序列化操作，并调用 Etcd 的 API 把它保存起来。

由此可见，声明式 API 对于 Kubernetes 来说非常重要。所以，**APIServer 这样一个在其他项目里“平淡无奇”的组件，却成了 Kubernetes 项目的重中之重**。它不仅是 Google Borg 设计思想的集中体现，也是 Kubernetes 项目里唯一一个被 Google 公司和 RedHat 公司双重控制、其他势力根本无法参与其中的组件。

此外，由于同时要兼顾性能、API 完备性、版本化、向后兼容等很多工程化指标，所以 Kubernetes 团队在 APIServer 项目里大量使用了 Go 语言的代码生成功能，来自动化诸如 Convert、DeepCopy 等与 API 资源相关的操作。这部分自动生成的代码，曾一度占到 Kubernetes 项目总代码的 20%~30%。

这也是为何，在过去很长一段时间里，在这样一个极其“复杂”的 APIServer 中，添加一个 Kubernetes 风格的 API 资源类型，是一个非常困难的工作。


不过，在 Kubernetes v1.7 之后，这个工作就变得轻松得多了。这，当然得益于一个全新的 API 插件机制：CRD。

CRD 的全称是 Custom Resource Definition。顾名思义，它指的就是，允许用户在 Kubernetes 中添加一个跟 Pod、Node 类似的、新的 API 资源类型，即：自定义 API 资源。

举个例子，**我现在要为 Kubernetes 添加一个名叫 Network 的 API 资源类型**。

它的作用是，一旦用户创建一个 Network 对象，那么 Kubernetes 就应该使用这个对象定义的网络参数，调用真实的网络插件，比如 Neutron 项目，为用户创建一个真正的“网络”。这样，将来用户创建的 Pod，就可以声明使用这个“网络”了。

这个 Network 对象的 YAML 文件，名叫 example-network.yaml，它的内容如下所示：

 复制代码

```
1 apiVersion: samplecrd.k8s.io/v1
2 kind: Network
3 metadata:
4   name: example-network
5 spec:
```

```
6   cidr: "192.168.0.0/16"
7   gateway: "192.168.0.1"
```


可以看到，我想要描述“网络”的 API 资源类型是 Network；API 组是 samplecrd.k8s.io；API 版本是 v1。

那么，Kubernetes 又该如何知道这个 API (samplecrd.k8s.io/v1/network) 的存在呢？

其实，上面的这个 YAML 文件，就是一个具体的“自定义 API 资源”实例，也叫 CR (Custom Resource)。而为了能够让 Kubernetes 认识这个 CR，你就需要让 Kubernetes 明白这个 CR 的宏观定义是什么，也就是 CRD (Custom Resource Definition)。

这就好比，你想让计算机认识各种兔子的照片，就得先让计算机明白，兔子的普遍定义是什么。比如，兔子“是哺乳动物”“有长耳朵，三瓣嘴”。

所以，接下来，我就先需编写一个 CRD 的 YAML 文件，它的名字叫作 network.yaml，内容如下所示：

 复制代码

```
1 apiVersion: apiextensions.k8s.io/v1beta1
2 kind: CustomResourceDefinition
3 metadata:
4   name: networks.samplecrd.k8s.io
5 spec:
6   group: samplecrd.k8s.io
7   version: v1
8   names:
9     kind: Network
10    plural: networks
11    scope: Namespaced
```

可以看到，在这个 CRD 中，我指定了“group: samplecrd.k8s.io”“version: v1”这样的 API 信息，也指定了这个 CR 的资源类型叫作 Network，复数 (plural) 是 networks。

然后，我还声明了它的 scope 是 Namespaced，即：我们定义的这个 Network 是一个属于 Namespace 的对象，类似于 Pod。


这就是一个 Network API 资源类型的 API 部分的宏观定义了。这就等同于告诉了计算机：“兔子是哺乳动物”。所以这时候，Kubernetes 就能够认识和处理所有声明了 API 类型是 “samplecrd.k8s.io/v1/network” 的 YAML 文件了。

接下来，我还需要让 Kubernetes “认识” 这种 YAML 文件里描述的 “网络” 部分，比如 “cidr”（网段），“gateway”（网关）这些字段的含义。这就相当于我要告诉计算机：“兔子有长耳朵和三瓣嘴”。

这时候呢，我就需要稍微做些代码工作了。

首先，我要在 GOPATH 下，创建一个结构如下的项目：


备注：在这里，我并不要求你具有完备的 Go 语言知识体系，但我会假设你已经了解了 Golang 的一些基本知识（比如，知道什么是 GOPATH）。而如果你还不了解的话，可以在涉及到相关内容时，再去查阅一些相关资料。

 复制代码

```
1 $ tree $GOPATH/src/github.com/<your-name>/k8s-controller-custom-resource
2 .
3 |— controller.go
4 |— crd
5 |   |— network.yaml
6 |— example
7 |   |— example-network.yaml
8 |— main.go
9 |— pkg
10 |   |— apis
11 |       |— samplecrd
12 |           |— register.go
13 |               |— v1
14 |                   |— doc.go
15 |                   |— register.go
16 |                   |— types.go
```


其中，pkg/apis/samplecrd 就是 API 组的名字，v1 是版本，而 v1 下面的 types.go 文件里，则定义了 Network 对象的完整描述。我已经把这个项目[上传到了 GitHub 上](#)，你可以随时参考。

然后，我在 pkg/apis/samplecrd 目录下创建了一个 register.go 文件，用来放置后面要用到的全局变量。这个文件的内容如下所示：

 复制代码

```
1 package samplecrd
2
3 const (
4     GroupName = "samplecrd.k8s.io"
5     Version   = "v1"
6 )
```

接着，我需要在 pkg/apis/samplecrd 目录下添加一个 doc.go 文件（Golang 的文档源文件）。这个文件里的内容如下所示：

 复制代码


```
1 // +k8s:deepcopy-gen=package
2
3 // +groupName=samplecrd.k8s.io
4 package v1
```

在这个文件中，你会看到 +<tag_name>[=value] 格式的注释，这就是 Kubernetes 进行代码生成要用的 Annotation 风格的注释。

其中，+k8s:deepcopy-gen=package 意思是，请为整个 v1 包里的所有类型定义自动生成 DeepCopy 方法；而+groupName=samplecrd.k8s.io，则定义了这个包对应的 API 组的名字。

可以看到，这些定义在 doc.go 文件的注释，起到的是全局的代码生成控制的作用，所以也被称为 Global Tags。

接下来，我需要添加 **types.go** 文件。顾名思义，它的作用就是定义一个 **Network** 类型到底有哪些字段（比如，**spec** 字段里的内容）。这个文件的主要内容如下所示：

 复制代码

```
1 package v1
2 ...
3 // +genclient
4 // +genclient:noStatus
5 // +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
6
7 // Network describes a Network resource
8 type Network struct {
9     // TypeMeta is the metadata for the resource, like kind and apiversion
10    metav1.TypeMeta `json:",inline"`
11    // ObjectMeta contains the metadata for the particular object, including
12    // things like...
13    // - name
14    // - namespace
15    // - self link
16    // - labels
17    // - ... etc ...
18    metav1.ObjectMeta `json:"metadata,omitempty"`
19
20    Spec networkspec `json:"spec"`
21 }
22 // networkspec is the spec for a Network resource
23 type networkspec struct {
24     Cidr      string `json:"cidr"`
25     Gateway string `json:"gateway"`
26 }
27
28 // +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
29
30 // NetworkList is a list of Network resources
31 type NetworkList struct {
32     metav1.TypeMeta `json:",inline"`
33     metav1.ListMeta `json:"metadata"`
34
35     Items []Network `json:"items"`
36 }
```

在上面这部分代码里，你可以看到 **Network** 类型定义方法跟标准的 **Kubernetes** 对象一样，都包括了 **TypeMeta**（API 元数据）和 **ObjectMeta**（对象元数据）字段。

而其中的 Spec 字段，就是需要我们自己定义的部分。所以，在 networkspec 里，我定义了 Cidr 和 Gateway 两个字段。其中，每个字段最后面的部分比如 `json:"cidr"`，指的就是这个字段被转换成 JSON 格式之后的名字，也就是 YAML 文件里的字段名字。

如果你不熟悉这个用法的话，可以查阅一下 Golang 的文档。

此外，除了定义 Network 类型，你还需要定义一个 NetworkList 类型，用来描述**一组 Network 对象**应该包括哪些字段。之所以需要这样一个类型，是因为在 Kubernetes 中，获取所有 X 对象的 List() 方法，返回值都是 List 类型，而不是 X 类型的数组。这是不一样的。

同样地，在 Network 和 NetworkList 类型上，也有代码生成注释。

其中，+genclient 的意思是：请为下面这个 API 资源类型生成对应的 Client 代码（这个 Client，我马上会讲到）。而 +genclient:noStatus 的意思是：这个 API 资源类型定义里，没有 Status 字段。否则，生成的 Client 就会自动带上 UpdateStatus 方法。

如果你的类型定义包括了 Status 字段的话，就不需要这句 +genclient:noStatus 注释了。比如下面这个例子：

 复制代码

```
1 // +genclient
2
3 // Network is a specification for a Network resource
4 type Network struct {
5     metav1.TypeMeta   `json:",inline"`
6     metav1.ObjectMeta `json:"metadata,omitempty"`
7
8     Spec   NetworkSpec   `json:"spec"`
9     Status NetworkStatus `json:"status"`
10 }
```

需要注意的是，+genclient 只需要写在 Network 类型上，而不用写在 NetworkList 上。因为 NetworkList 只是一个返回值类型，Network 才是“主类型”。

而由于我在 Global Tags 里已经定义了为所有类型生成 DeepCopy 方法，所以这里就不需要再显式地加上 `+k8s:deepcopy-gen=true` 了。当然，这也就意味着你可以用 `+k8s:deepcopy-gen=false` 来阻止为某些类型生成 DeepCopy。


你可能已经注意到，在这两个类型上面还有一句 `+k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object` 的注释。它的意思是，请在生成 DeepCopy 的时候，实现 Kubernetes 提供的 `runtime.Object` 接口。否则，在某些版本的 Kubernetes 里，你的这个类型定义会出现编译错误。这是一个固定的操作，记住即可。

不过，你或许会有这样的顾虑：这些代码生成注释这么灵活，我该怎么掌握呢？

其实，上面我所讲述的内容，已经足以应对 99% 的场景了。当然，如果你对代码生成感兴趣的话，我推荐你阅读[这篇博客](#)，它详细地介绍了 Kubernetes 的代码生成语法。

最后，我需要再编写的一个 `pkg/apis/samplecrd/v1/register.go` 文件。

在前面对 API Server 工作原理的讲解中，我已经提到，“registry”的作用就是注册一个类型（Type）给 API Server。其中，Network 资源类型在服务器端的注册的工作，API Server 会自动帮我们完成。但与之对应的，我们还需要让客户端也能“知道”Network 资源类型的定义。这就需要我们在项目里添加一个 `register.go` 文件。它最主要的功能，就是定义了如下所示的 `addKnownTypes()` 方法：

 复制代码

```
1 package v1
2 ...
3 // addKnownTypes adds our types to the API scheme by registering
4 // Network and NetworkList
5 func addKnownTypes(scheme *runtime.Scheme) error {
6     scheme.AddKnownTypes(
7         SchemeGroupVersion,
8         &Network{},
9         &NetworkList{},
10    )
11
12    // register the type in the scheme
13    metav1.AddToGroupVersion(scheme, SchemeGroupVersion)
14    return nil
15 }
```

有了这个方法，Kubernetes 就能够在后面生成客户端的时候，“知道” Network 以及 NetworkList 类型的定义了。

像上面这种register.go 文件里的内容其实是非常固定的，你以后可以直接使用我提供的这部分代码做模板，然后把其中的资源类型、GroupName 和 Version 替换成你自己的定义即可。


这样，Network 对象的定义工作就全部完成了。可以看到，它其实定义了两部分内容：

第一部分是，自定义资源类型的 API 描述，包括：组（Group）、版本（Version）、资源类型（Resource）等。这相当于告诉了计算机：兔子是哺乳动物。

第二部分是，自定义资源类型的对象描述，包括：Spec、Status 等。这相当于告诉了计算机：兔子有长耳朵和三瓣嘴。


接下来，我就要使用 Kubernetes 提供的代码生成工具，为上面定义的 Network 资源类型自动生成 clientset、informer 和 lister。其中，clientset 就是操作 Network 对象所需要的客户端，而 informer 和 lister 这两个包的主要功能，我会在下一篇文章中重点讲解。

这个代码生成工具名叫k8s.io/code-generator，使用方法如下所示：

 复制代码

```
1 # 代码生成的工作目录，也就是我们的项目路径
2 $ ROOT_PACKAGE="github.com/resouer/k8s-controller-custom-resource"
3 # API Group
4 $ CUSTOM_RESOURCE_NAME="samplecrd"
5 # API Version
6 $ CUSTOM_RESOURCE_VERSION="v1"
7
8 # 安装 k8s.io/code-generator
9 $ go get -u k8s.io/code-generator/...
10 $ cd $GOPATH/src/k8s.io/code-generator
11
12 # 执行代码自动生成，其中 pkg/client 是生成目标目录，pkg/apis 是类型定义目录
13 $ ./generate-groups.sh all "$ROOT_PACKAGE/pkg/client" "$ROOT_PACKAGE/pkg/apis" "$CUSTOM_
```

代码生成工作完成之后，我们再查看一下这个项目的目录结构：

 复制代码

```
1 $ tree
2 .
3 ├── controller.go
4 ├── crd
5 │   └── network.yaml
6 ├── example
7 │   └── example-network.yaml
8 ├── main.go
9 └── pkg
10     ├── apis
11     │   └── samplecrd
12     │       ├── constants.go
13     │       └── v1
14     │           ├── doc.go
15     │           ├── register.go
16     │           ├── types.go
17     │           └── zz_generated.deepcopy.go
18     └── client
19         ├── clientset
20         ├── informers
21         └── listers
```

其中，pkg/apis/samplecrd/v1 下面的 zz_generated.deepcopy.go 文件，就是自动生成的 DeepCopy 代码文件。

而整个 client 目录，以及下面的三个包（clientset、informers、listers），都是 Kubernetes 为 Network 类型生成的客户端库，这些库会在后面编写自定义控制器的时候用到。

可以看到，到目前为止的这些工作，其实并不要求你写多少代码，主要考验的是“复制、粘贴、替换”这样的“基本功”。

而有了这些内容，现在你就可以在 **Kubernetes 集群里创建一个 Network 类型的 API 对象** 了。我们不妨一起来实验一下。

首先，使用 network.yaml 文件，在 Kubernetes 中创建 Network 对象的 CRD（Custom Resource Definition）：

[📄 复制代码](#)

```
1 $ kubectl apply -f crd/network.yaml
2 customresourcedefinition.apiextensions.k8s.io/networks.samplecrd.k8s.io created
```

这个操作，就告诉了 Kubernetes，我现在要添加一个自定义的 API 对象。而这个对象的 API 信息，正是 network.yaml 里定义的内容。我们可以通过 kubectl get 命令，查看这个 CRD：

[📄 复制代码](#)

```
1 $ kubectl get crd
2 NAME                                CREATED AT
3 networks.samplecrd.k8s.io          2018-09-15T10:57:12Z
```

然后，我们就可以创建一个 Network 对象了，这里用到的是 example-network.yaml：

[📄 复制代码](#)

```
1 $ kubectl apply -f example/example-network.yaml
2 network.samplecrd.k8s.io/example-network created
```

通过这个操作，你就在 Kubernetes 集群里创建了一个 Network 对象。它的 API 资源路径是 samplecrd.k8s.io/v1/networks。

这时候，你可以通过 kubectl get 命令，查看到新创建的 Network 对象：

[📄 复制代码](#)

```
1 $ kubectl get network
2 NAME            AGE
3 example-network  8s
```

你还可以通过 kubectl describe 命令，看到这个 Network 对象的细节：

```
1 $ kubectl describe network example-network
2 Name:          example-network
3 Namespace:     default
4 Labels:        <none>
5 ...API Version: samplecrd.k8s.io/v1
6 Kind:          Network
7 Metadata:
8   ...
9   Generation:   1
10  Resource Version: 468239
11  ...
12 Spec:
13   Cidr:         192.168.0.0/16
14   Gateway:      192.168.0.1
```

当然，你也可以编写更多的 YAML 文件来创建更多的 Network 对象，这和创建 Pod、Deployment 的操作，没有任何区别。

总结

在今天这篇文章中，我为你详细解析了 Kubernetes 声明式 API 的工作原理，讲解了如何遵循声明式 API 的设计，为 Kubernetes 添加一个名叫 Network 的 API 资源类型。从而达到了通过标准的 `kubectl create` 和 `get` 操作，来管理自定义 API 对象的目的。

不过，创建出这样一个自定义 API 对象，我们只是完成了 Kubernetes 声明式 API 的一半工作。

接下来的另一半工作是：为这个 API 对象编写一个自定义控制器（Custom Controller）。这样，Kubernetes 才能根据 Network API 对象的“增、删、改”操作，在真实环境中做出相应的响应。比如，“创建、删除、修改”真正的 Neutron 网络。

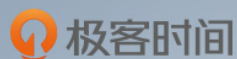
而这，正是 Network 这个 API 对象所关注的“业务逻辑”。

这个业务逻辑的实现过程，以及它所使用的 Kubernetes API 编程库的工作原理，就是我要在下一篇文章中讲解的主要内容。

思考题

在了解了 CRD 的定义方法之后，你是否已经在考虑使用 CRD（或者已经使用了 CRD）来描述现实中的某种实体了呢？能否分享一下你的思路？（举个例子：某技术团队使用 CRD 描述了“宿主机”，然后用 Kubernetes 部署了 Kubernetes）

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。



深入剖析 Kubernetes

Kubernetes 原来可以如此简单

张磊

Kubernetes 社区
资深成员与项目维护者



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | 声明式API与Kubernetes编程范式

下一篇 25 | 深入解析声明式API（二）：编写自定义控制器

精选留言 (30)

写留言



小明root

2018-11-22

12

我是运维人员，我很心碎.....我不会go语言

展开 ∨



北卡

2018-10-18

👍 10

运维人员会心碎？

我是运维人员，此刻看完我感到很兴奋。

作者回复: 没错。我觉得这部分才是运维最喜欢的。谁不喜欢写代码呢？



虎虎

2018-10-17

👍 6

一般来说，扩展api server (或者说添加自定义 resource)有两种方式：

1. 通过创建CRDs, 主API server可以处理 CRDs 的 REST 请求 (CRUD) 和持久性存储。简单，不需要其他的编程。更适用于声明式的API，和kubernetes高度集成统一。
2. API Aggregation, 一个独立的API server。主API server委托此独立的API server处理自定义resource。需要编程，但能够更加灵活的控制API的行为，更加灵活的自定义存...
展开 ∨



千寻

2018-10-17

👍 5

我从代码开始，就按着步骤走，最后创建network的CRD和example-network都成功了，但是我直接将cdr/network.yml和example/example-network.yml文件单独拿出来，并没有执行代码生成那些步骤，发现也创建成功了，搞得有点懵。
老师可以说一下这大概是怎么回事吗？

展开 ∨

作者回复: 因为生成的代码是给下一篇写控制器代码用的，哈哈



Joe

2019-01-10

👍 3

Generating deepcopy funcs

F0110 15:05:38.168635 11946 deepcopy.go:866] Hit an unsupported type invalid type.

按文档走了一半，报错~ 这个是什么问题呀？

展开 ∨



yuanlinio...

2018-12-28

👍 3

./generate-groups.sh all "

ROOT_PACKAGE/pkg/client "" *ROOT_PACKAGE/pkg/apis* "

CUSTOM_RESOURCE_NAME :CUSTOM_RESOURCE_VERSION"

Generating deepcopy funcs

F1228 01:11:20.543446 2908 deepcopy.go:866] Hit an unsupported type invalid type.

展开 ▾



小金刚

2018-10-28

👍 3

可以用 kubebuild 自动生成项目框架，添加自己的 CRD 并实现 controller 即可。



小宇宙

2018-10-17

👍 3

用CRD来开发定义自主化的operator，将有状态的应用自动化



Leon

2019-01-26

👍 2

我是开发人员，会go,也会C++，也自己设计过业务协议，看到这篇异常兴奋



圣诞使者

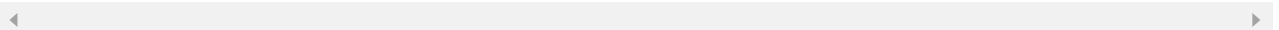
2018-10-22

👍 2

老师，我照着你的代码敲了一遍，这个pkg/signals目录是自己创建的吗？我这个生成完代码也没有这个目录。

展开 ▾

作者回复: 这是个自己的帮助库



Vincen

👍 2

2018-10-17

从kubernetes用户到kubernetes玩家成长中...

展开 ▾



蜗牛

2018-10-23

👍 1

有一个问题张老师，一直不是特别清楚... 通过 crd 创建的自定义资源我还并没有定义他的结构，为什么就可以通过 `kubectl get` 拿到这个资源的详情呢？

展开 ▾

作者回复: 你提交了一个CR yaml了不是？反正是存在etcd里，给你返回即可。不过kubernetes 并不懂字段的意思。

◀ ▶



小伟

2018-10-18

👍 1

老师好，作为一个运维，觉得你讲的内容好深，学得好吃力，不知能否分享一些实际生产环境的troubleshooting 的案例和思路？

展开 ▾

作者回复: 建议还是多实践这里讲解的内容

◀ ▶



mazhen

2018-10-18

👍 1

register.go会将自定义Type注册到APIServer，那register.go本身是怎么交给APIServer，然后被APIServer调用注册过程的？

`kubectlapply -f crd/network.yaml` `kubectl apply -f example/example-network.yaml`

执行完这两步，自定义的Newwork对象被创建出来，怎么感觉register.go并没有被用到

展开 ▾

作者回复: 因为你还没有调用生成的client, register.go是客户端用来进行初始化的代码的一部分。你这两步做完, 服务端已经知道这个crd了, 客户端也得有办法知道啊。



kyleqian

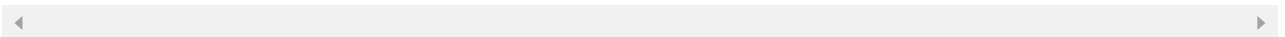
2018-10-17

👍 1

这章会让运维人员心碎的...幸好俺会Go语言!

请教下这个生成的代码, 还得合并到apiserver的代码树中吧? 还是得重新编译apiserver吧? 毕竟Go目前还没有动态加载机制。

作者回复: 当然不需要。你就按文章里的步骤来就全妥了。



iuwai

2019-03-06

👍

看到能对k8s进行二次开发, 很兴奋! 不过, 自定义APIserver在具体的实践中有哪些实际的用途呢?



Charles

2019-02-14

👍

老师, 你讲课中使用到的容器的镜像有没有统一的地址可以给一下, 找对应的镜像真心不容易啊



Charles

2019-02-14

👍

正好学过go, 可以派上用场了

展开 ▾



疯狂的小企...

2019-02-09

👍

提示 Hit an unsupported type invalid type的同学, 可以先安装下k8s.io/apimachinery包。应该是deepcopy找不到metav1.ObjectMeta和metav1.TypeMeta

go get -u k8s.io/apimachinery



骨汤鸡蛋面

2019-01-05



自定义resource的Controller 单独运行，只是通过client-go 与api 交互？是否可以认为，k8s内建的Resource 对应的Controller，由Controller-manager 统一管理呢？

作者回复: 对的，内置和外挂的区别

