资源类型 Scheme

本节主要讲解 Kubernetes 核心的资源类型 Scheme 的定义和用途。

介绍

当我们操作资源和 apiserver 进行通信的时候，需要根据资源对象类型的 Group、Version、Kind 以及规范定义、编解码等内容构成 Scheme 类型，然后 Clientset 对象就可以来访问和操作这些资源类型了，Scheme 的定义主要在 api 子项目之中，源码仓库地址: https://github.com/kubernetes/api ，被同步到 Kubernetes 源码的 staging/src/k8s.io/api 之下。

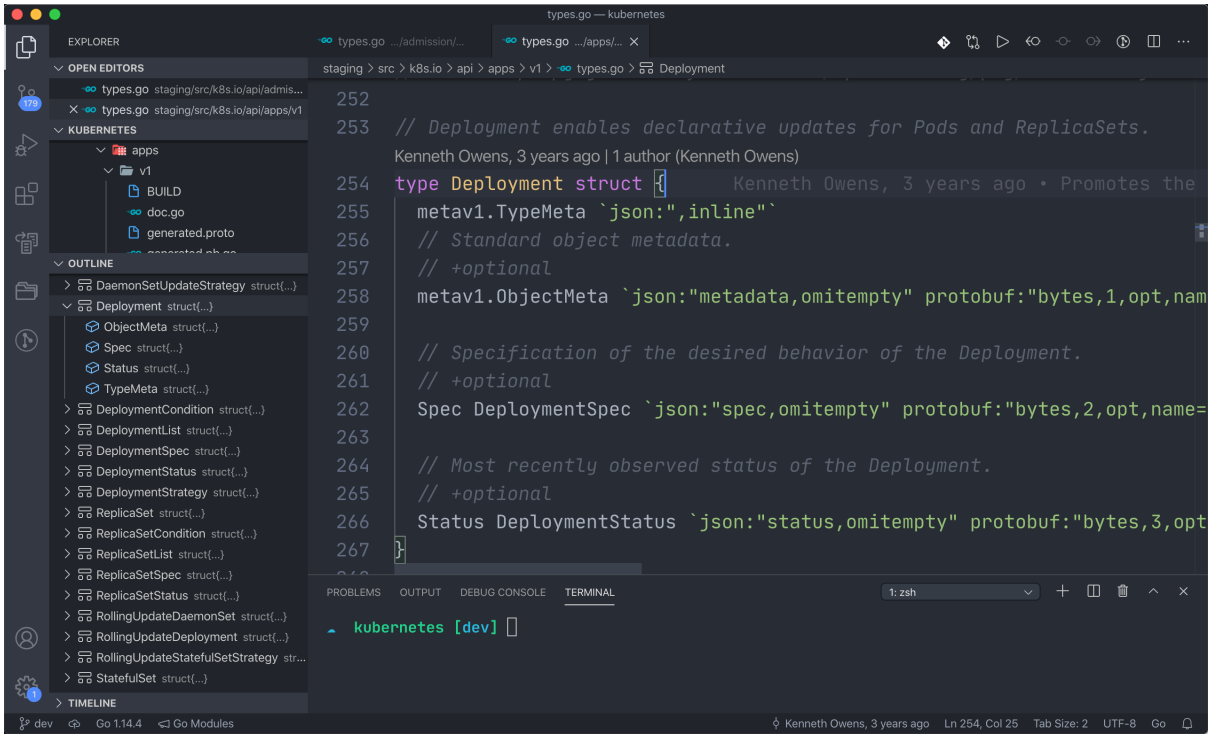
主要就是各种资源对象的原始结构体定义，比如查看 `apps/v1` 目录下面的定义：

```
$ tree staging/src/k8s.io/api/apps/v1
staging/src/k8s.io/api/apps/v1
├── BUILD
├── doc.go
├── generated.pb.go
├── generated.proto
├── register.go
├── types.go
├── types_swagger_doc_generated.go
└── zz_generated.deepcopy.go

0 directories, 8 files
```

types.go 文件

其中 `types.go` 文件里面就是 `apps/v1` 这个 `GroupVersion` 下面所有的资源对象的定义，有 Deployment、DaemonSet、StatefulSet、ReplicaSet 等几个资源对象，比如 Deployment 的结构体定义如下所示：



由 `TypeMeta`、`ObjectMeta`、`DeploymentSpec` 以及 `DeploymentStatus` 4个属性组成，和我们使用 YAML 文件定义的 Deployment 资源对象也是对应的。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  nginx-deploy
  namespace: default
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

其中 `apiVersion` 与 `kind` 就是 `TypeMeta` 属性，metadata 属性就是 `ObjectMeta`，spec 属性就是 `DeploymentSpec`，当资源部署过后也会包含一个 status 的属性，也就是 `DeploymentStatus`，这样就完整的描述了一个资源对象的模型。

zz_generated.deepcopy.go 文件

上面定义的规范在 Kubernetes 中称为`资源类型 Scheme`，此外`zz_generated.deepcopy.go`文件是由`deepcopy-gen`工具创建的定义各资源类型`DeepCopyObject()`方法的文件，所有注册到 Scheme 的资源类型都要实现`runtime.Object`接口：

```
// staging/src/k8s.io/apimachinery/pkg/runtime/interface.go
type Object interface {
  GetObjectKind() schema.ObjectKind
  DeepCopyObject() Object
}
```

而所有的资源类型都包含一个 TypeMeta 类型，而该类型实现了`GetObjectKind()`方法，所以各资源类型只需要实现`DeepCopyObject()`方法即可：

```
// staging/src/k8s.io/apimachinery/pkg/apis/meta/v1/meta.go
func (obj *TypeMeta) GetObjectKind() schema.ObjectKind { return obj }
```

各个资源类型的`DeepCopyObject()`方法也不是手动定义，而是使用`deepcopy-gen`工具命令统一自动生成的，该工具会读取`types.go`文件中的`+k8s:deepcopy-gen`注释，以 Deployment 为例：

```
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// Deployment enables declarative updates for Pods and ReplicaSets.
type Deployment struct {
......
}
```

然后将自动生成的代码保存到`zz_generated.deepcopy.go`文件中。

register.go 文件

`register.go`文件的主要作用是定义 AddToScheme 函数，将各种资源类型注册到 Clientset 使用的 Scheme 对象中去，由于每个资源自动生成了 DeepCopyObject() 方法，这样资源就实现了`runtime.Object`接口，所以可以注册到 Scheme 中去了。

```
// staging/src/k8s.io/api/apps/v1/register.go
var (
  // TODO: move SchemeBuilder with zz_generated.deepcopy.go to k8s.io/api.
  // localSchemeBuilder and AddToScheme will stay in k8s.io/kubernetes.
  SchemeBuilder      = runtime.NewSchemeBuilder(addKnownTypes)
  localSchemeBuilder = &SchemeBuilder
  // 对外暴露的 AddToScheme 方法用于注册该 Group/Verion 下的所有资源类型
  AddToScheme        = localSchemeBuilder.AddToScheme
)
```

```
// staging/src/k8s.io/client-go/kubernetes/scheme/register.go
// 新建一个 Scheme，将各类资源对象都添加到该 Scheme
var Scheme = runtime.NewScheme()
// 为 Scheme 中的所有类型创建一个编解码工厂
var Codecs = serializer.NewCodecFactory(Scheme)
// 为 Scheme 中的所有类型创建一个参数编解码工厂
var ParameterCodec = runtime.NewParameterCodec(Scheme)
// 将各 k8s.io/api/<Group>/<Version> 目录下资源类型的 AddToScheme() 方法注册到 SchemeBuilder 中
var localSchemeBuilder = runtime.SchemeBuilder{
  ......
  appsv1.AddToScheme,
  appsv1beta1.AddToScheme,
  appsv1beta2.AddToScheme,
  ......
}

var AddToScheme = localSchemeBuilder.AddToScheme

func init() {
  v1.AddToGroupVersion(Scheme, schema.GroupVersion{Version: "v1"})
  // 调用 SchemeBuilder 中各资源对象的 AddToScheme() 方法，将它们注册到到 Scheme 对象
  utilruntime.Must(AddToScheme(Scheme))
}
```

将各类资源类型注册到`全局的 Scheme 对象`中，这样 Clientset 就可以识别和使用它们了。

apimachinery 子项目主要是 Kubernetes 服务端和客户端项目都共同依赖的一些公共方法、struct、工具类的定义，主要服务于 kubernetes、client-go、apiserver 这三个项目。