代码生成器

本文主要介绍如何使用代码生成器为 CRD 资源自动生成代码

介绍

上节课我们介绍了 CRD 的使用，了解到 CRD 仅仅是一种资源的定义而已，需要一个对应的控制器去监听 CRD 的各种事件来添加自己的业务逻辑才有实际意义，接下来我们就来介绍如何为 CRD 创建一个自定义控制器。

代码生成器

要实现自己的控制器原理比较简单，前面我们也介绍过如何编写控制器，最重要的就是要去实现 ListAndWatch 操作、获取资源的 Informer 和 Indexer、以及通过一个 workqueue 去接收事件来进行处理，所以我们就要想办法来编写我们自定义的 CRD 资源对应的 Informer、ClientSet 这些工具，前面我们已经了解了对于内置的 Kubernetes 资源对象这些都是已经内置到源码中了，对于我们自己的 CRD 资源肯定不会内置到源码中的，所以就需要我们自己去实现，比如要为 CronTab 这个资源对象实现一个 `DeepCopyObject` 函数，这样才会将我们自定义的对象转换成 `runtime.Object`，系统才能够识别，但是客户端相关的操作实现又非常多，而且实现方式基本上都是一致的，所以 Kubernetes 就为我们提供了代码生成器这样的工具，我们可以来自动生成客户端访问的一些代码，比如 Informer、ClientSet 等等。

code-generator

code-generator 就是 Kubernetes 提供的一个用于代码生成的项目，它提供了以下工具为 Kubernetes 中的资源生成代码：

- **deepcopy-gen:** 生成深度拷贝方法，为每个 T 类型生成 `func (t* T) DeepCopy() *T` 方法，API 类型都需要实现深拷贝
- **client-gen:** 为资源生成标准的 clientset
- **informer-gen:** 生成 informer，提供事件机制来响应资源的事件
- **lister-gen:** 生成 Lister，为 get 和 list 请求提供只读缓存层（通过 indexer 获取）

Informer 和 Lister 是构建控制器的基础，使用这4个代码生成器可以创建全功能的、和 Kubernetes 上游控制器工作机制相同的 production-ready 的控制器。

code-generator 还包含一些其它的代码生成器，例如 Conversion-gen 负责产生内外部类型的转换函数、Defaulter-gen 负责处理字段默认值。大部分的生成器支持 `--input-dirs` 参数来读取一系列输入包，处理其中的每个类型，然后生成代码：

1. 部分代码生成到输入包所在目录，例如 `deepcopy-gen` 生成器，也可以使用参数 `--output-file-base "zz_generated.deepcopy"` 来定义输出文件名
2. 其它代码生成到 `--output-package` 指定的目录，例如 `client-gen`、`informer-gen`、`lister-gen` 等生成器

在开发 CRD 的控制器的时候，我们可以编写一个脚本来统一调用生成器生成代码，我们可以直接使用 sample-controller 仓库中提供的 hack/update-codegen.sh 脚本。

```
#!/usr/bin/env bash

set -o errexit
set -o nounset
set -o pipefail

SCRIPT_ROOT=$(dirname "${BASH_SOURCE[0]}")/..
# 代码生成器包的位置
CODEGEN_PKG=${CODEGEN_PKG:-$(cd "${SCRIPT_ROOT}"; ls -d -1 ./vendor/k8s.io/code-generator 2>/dev/null || echo ../code-generator)}
```

```
# generate-groups.sh <generators> <output-package> <apis-package> <groups-versions>
#                     使用哪些生成器，可选值 deepcopy,defaulter,client,lister,informer，逗号分隔，all表示全部使用
#                     输出包的导入路径
#                     CR 定义所在路径
#                     API 组和版本
bash "${CODEGEN_PKG}"/generate-groups.sh "deepcopy,client,informer,lister" \
  k8s.io/sample-controller/pkg/generated k8s.io/sample-controller/pkg/apis \
  samplecontroller:v1alpha1 \
  --output-base "$(dirname "${BASH_SOURCE[0]}")/../../.." \
  --go-header-file "${SCRIPT_ROOT}"/hack/boilerplate.go.txt

# 自动生成的源码头部附加的内容：
#   --go-header-file "${SCRIPT_ROOT}"/hack/custom-boilerplate.go.txt
```

执行上面的脚本后，所有 API 代码会生成在 `pkg/apis` 目录下，clientsets、informers、listers 则生成在 `pkg/generated` 目录下。不过从脚本可以看出需要将 code-generator 的包放置到 vendor 目录下面，现在我们都是使用 `go modules` 来管理依赖保，我们可以通过执行 `go mod vendor` 命令将依赖包放置到 vendor 目录下面来。

我们还可以进一步提供 `hack/verify-codegen.sh` 脚本，用于判断生成的代码是否 up-to-date：

```
#!/usr/bin/env bash

set -o errexit
set -o nounset
set -o pipefail

# 先调用 update-codegen.sh 生成一份新代码
# 然后对比新老代码是否一样

SCRIPT_ROOT=$(dirname "${BASH_SOURCE[0]}")/..

DIFFROOT="${SCRIPT_ROOT}/pkg"
TMP_DIFFROOT="${SCRIPT_ROOT}/_tmp/pkg"
_tmp="${SCRIPT_ROOT}/_tmp"

cleanup() {
  rm -rf "${_tmp}"
}
trap "cleanup" EXIT SIGINT

cleanup

mkdir -p "${TMP_DIFFROOT}"
cp -a "${DIFFROOT}"/* "${TMP_DIFFROOT}"

"${SCRIPT_ROOT}/hack/update-codegen.sh"
echo "diffing ${DIFFROOT} against freshly generated codegen"
ret=0
diff -Naupr "${DIFFROOT}" "${TMP_DIFFROOT}" || ret=$?
cp -a "${TMP_DIFFROOT}"/* "${DIFFROOT}"
if [[ $ret -eq 0 ]]
then
  echo "${DIFFROOT} up to date."
else
  echo "${DIFFROOT} is out of date. Please run hack/update-codegen.sh"
  exit 1
fi
```

代码生成 tag

除了通过命令行标记控制代码生成器之外，我们一般是在源码中使用 `tag` 来标记一些供生成器使用的属性，这些 `tag` 主要分为两类：

1. 在 `doc.go` 的 package 语句之上提供的全局 tag
2. 在需要被处理的类型上提供的局部 tag

tag 的使用方法如下所示：

```
// +tag-name
// 或者
// +tag-name=value
```

我们可以看到 tag 是通过注释的形式存在的，另外需要注意的是 tag 的位置非常重要，很多 tag 必须直接位于 type 或 package 语句的上一行，另外一些则必须和 go 语句隔开至少一行空白。

全局 tag

必须在目标包的 `doc.go` 文件中声明，一般路径为 `pkg/apis/<apigroup>/<version>/doc.go` 如下所示：

```
// 为包中任何类型生成深拷贝方法，可以在局部 tag 覆盖此默认行为
// +k8s:deepcopy-gen=package

// groupName 指定 API 组的全限定名
// 此 API 组的 v1 版本，放在同一个包中
// +groupName=example.com
package v1
```

> 注意: 空行不能省

局部 tag

要么直接声明在类型之前，要么位于类型之前的第二个注释块中。下面的 `types.go` 中声明了 CR 对应的类型：

```
// 为当前类型生成客户端，如果不加此注解则无法生成 lister·informer 等包
// +genclient

// 提示此类型不基于 /status 子资源来实现 spec-status 分离，产生的客户端不具有 UpdateStatus 方法
// 否则，只要类型具有 Status 字段，就会生成 UpdateStatus 方法
// +genclient:noStatus

// 为每个顶级 API 类型添加，自动生成 DeepCopy 相关代码
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// K8S 资源，数据库
type Database struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec DatabaseSpec `json:"spec"`
}

// 不为此类型生成深拷贝方法
// +k8s:deepcopy-gen=false

// 数据库的规范
type DatabaseSpec struct {
    User     string `json:"user"`
    Password string `json:"password"`
    Encoding string `json:"encoding,omitempty"`
}

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

// 数据库列表，因为 list 获取的是列表，所以需要定义该结构
type DatabaseList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata"`

    Items []Database `json:"items"`
}
```

在上面 CR 的定义上面就通过 tag 来添加了自动生成相关代码的一些注释。此外对于集群级别的资源，我们还需要提供如下所示的注释：

```
// +genclient:nonNamespaced

// 下面的 Tag 不能少
// +genclient
```

另外我们还可以控制客户端提供哪些 HTTP 方法：

```
// +genclient:noVerbs
// +genclient:onlyVerbs=create,delete
```

```
// +genclient:skipVerbs=get,list,create,update,patch,delete,deleteCollection,watch
// 仅仅返回 Status 而非整个资源
// +genclient:method=Create,verb=create,result=k8s.io/apimachinery/pkg/apis/meta/v1.Status

// 下面的 Tag 不能少
// +genclient
```

使用 tag 定义完需要生成的代码规则后，执行上面提供的代码生成脚本即可自动生成对应的代码了。