### 声明式API在Istio中的实际使用

以Istio为例，Istio项目在每一个Pod当中安装了Envoy容器，是如何做到让那个应用程序无感的？？

使用K8s当中一个重要的功能：**Dynamic Admission Control**。

#### Dynamic Admission Control

当一个Pod或者API对象被提交到API Server运行之后，在正式创建之前需要预先进行一些工作，类似于“初始化”，这些初始化操作，基于一个**Admission**功能，它是一组Admission Controller代码，可以**选择性地被编译进API Server**,在创建完对象之后会立刻被调用到。

为了避免反复编译和重启API Server，提供了一种**“热插拔”式的Admission机制**，即Dynamic Admission Controller,也称为**Initializer**

例如，在一个Pod YAML被提交给K8s之后，Istio在其对应的API 对象里自动加入Envoy容器的配置。

**如何在用户不知情的情况下这个操作？**

Istio编写一个用来为Pod“自动注入”Envoy容器的Initializer。

逻辑如下：

for {  
// 获取当前新创建Pod的实际状态  
 pod := client.GetLatestPod()  
 //判断是否以及初始化过了  
 if !isInitialized(pod){  
 //没有，则进行初始化  
 doSomething(pod)  
 }  
}

实现流程如下：

1. 将Envoy的配置作为一个ConfigMap存储起来
2. 将ConfigMap当中的containers和volumes字段添加进Pod当中
3. Initializer使用k8s提供的TwoWayMergePatch（同时使用新旧两个Pod对象），调用k8s的Client，发起一个PATCH请求。
4. 每一个Pod都会被初始化，其Metadata携带了metadata.initializers.pending的信息，用于判断是否被初始化过。
5. 记得清除4中提到的标志

### 声明式API的独到之处

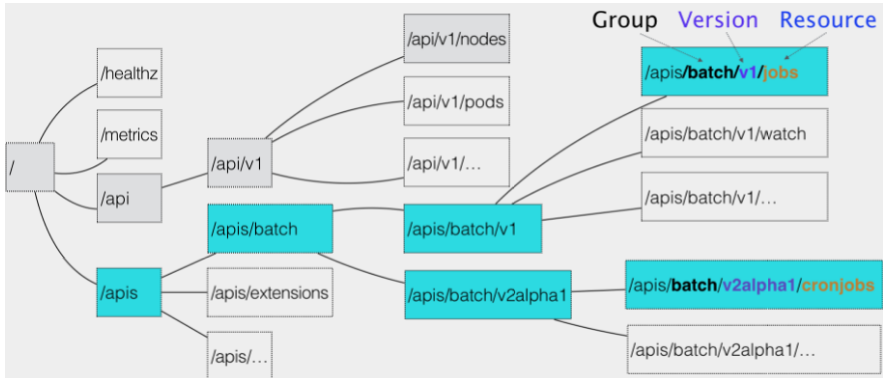
1. 只需要提交一个定义好的API对象，就可以声明期望达到的状态
2. 允许有多个API写入，以PATCH 方式对API对象进行修改，而无需关心原来的YAML文件
3. 在上两点的基础之上，k8s无需外接干预，实现对“实际状态“和期望状态的调谐过程

### 声明式API的工作原理

**提交一个YAML文件后,如何实现在k8s中创建一个API对象？？？？**

#### API的设计

k8s当中API对象的结构如下所示：



从图中可以看出：

1. API对象在etcd中的资源路径通过三部分进行表示：Group(API组)、Version（API版本）、Resource(API资源类型)
2. API对象的组织方式是层层递进的。

例如：

apiVersion: batch/v1alpha  
kind: CronJob

batch---组，v1alpha---版本，CronJob---资源类型

#### API对象建立流程

以上面的举例来说明。k8s获取到YAML文件的内容之后，需要对Resource、Vserion、Group进行解析。

1. 匹配API对象的组

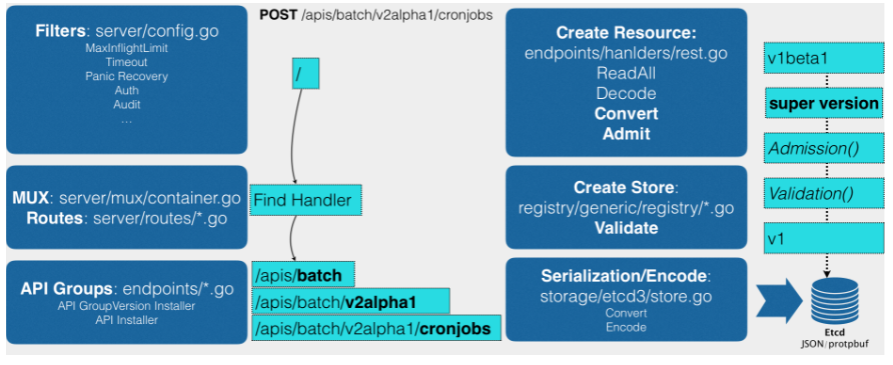
需要注意的是，对于Pod和Node这些核心的API对象，不需要Group（或者说他们的Group是“”）,**他们会直接从/api当中进行匹配**。

而对于非核心的API对象，例如CronJob等，则会在/apis下，根据Group名字进行查找。

1. 进一步匹配API的版本号
2. 匹配API对象的资源类型

完成以上三步，k8s就可以明白需要创建的是什么API对象。

下面是完整的请求处理流程：



### CRD

Custome Resource Definition，允许用户自定义API资源。

### Custom Controller——自定义控制器

在自定义API对象完成后，需要定义控制器Controller来对他进行管理。