Cinder-API 自定义接口扩展

Openstack中的xxx-api都是WSGI Service，负责接收、分析、处理HTTP request，并返回response。其中最重要的两个Python package是delopy和routes。

Python delopy负责查找、配置、启动WSGI app。

Python Routes是用 Python 重新实现的 Rails routes system，它被用来做将 URL 映射为 App 的 action，以及为 App的action 产生 URL（即，把RESTful的HTTP requests分发到相应的处理方法上）。

**启动Cinder-API**

Python paste delopy通过名为XXX.ini的配置文件，初始化WSGI中对应的服务。

配置文件有以下几个元素构成：

|  |  |
| --- | --- |
| 构成元素 | 说明 |
| composite | 将传入的request根据其请求的URL地址分发到不同的APP。WSGI应用从一个root composite开始，可能包含多级相互串联的composite。 |
| filter | request请求从composite到app的过程中会经过多级filter，filter会对request中请求的报文进行鉴权、修改等操作。多个级联的filter相当于整个WSGI服务的中间件。 |
| app | app：request最终开始响应的逻辑代码，在app的代码中会检查body中的内容进行相应操作。 |
| pipeline | 若干filter和一个app组成的完整过程链。 |

在Cinder中该配置文件为/etc/cinder/api-paste.ini，配置文件的格式如下：

|  |
| --- |
| [composite:XXXXXX]  # composite调用use方法将解析request，并且传递到下一个模块  # composite匹配request中url的path，将其传递到下一个composite，或者直接传递给pipeline  use = call: class\_moudule\_path: method  path1:other\_composite\_1  path2: other\_composite\_2 path3: pipeline\_name  [composite: other\_composite\_1]  # composite中可以配置工作链，工作链由若干filter和一个app组成  use = call: class\_moudule\_path: method  woker\_line = filter\_name1 filer\_name2 …... filer\_nameN app\_name1 [filter: XXXXXX] paste.filter\_factory = class\_moudule\_path:Class\_name.factory  [app: XXXXXX] paste.app\_factory =class\_moudule\_path:Class\_name.factory [pipeline:XXXXXX] pipeline = filter\_name1 filer\_name2 …... filer\_nameN app\_name1 |

在Cinder中c-api的每个进程对应的实现是cinder.service.WSGIService，其他模块进程对应的cinder.service.Service。启动c-api的过程中，启动程序cinder.cmd.api会根据配置文件/etc/cinder/api-paste.ini创建一个名为“osapi\_volume”的WSGIService，其构造函数如下：

def \_\_init\_\_(self, name, loader=None):  
 # 对应的服务名”osapi\_volume”  
 self.name = name  
 # 服务对应的管理类，如VolumeManager、SchedulerManager，但是对于Cinder-API而言没有管理类。  
 self.manager = self.\_get\_manager()  
 # 读取配置文件CONF.api\_paste\_config,加载WSGI对应的Middleware，APP  
 self.loader = loader or wsgi.Loader(CONF)  
 self.app = self.loader.load\_app(name)  
 # 读取其他配置项

……  
 # 通过oslo\_Service创建wsgi服务  
 self.server = wsgi.Server(CONF, name, self.app,  
 host=self.host,  
 port=self.port,  
 use\_ssl=self.use\_ssl)

在osapi\_volume 的 loading过程中，各个 filter middleware 以及 WSGI Application 都被加载/注册，并被初始化。WSGI Server 在被启动后，开始等待 HTTP request，然后进入HTTP request 处理过程。

HTTP request的处理过程：

1. HTTP request从进入osapi\_volume开始，根据请求的API版本，将请求转向相应版本的composite。
2. HTTP request顺序经过若干过滤器。
3. HTTP request到达app接口类APIRouter，APIRouter通过Python Routes将其定位到某个Controller

配置文件api-paste.ini中定义的app接口类APIRouter，利用了Python Routes的方式将URL中的action，最终映射到某一个具体的Controller，每种Controller在OpenStack中称为一种资源。如VolumeController对应卷操作的API接口，SnapshotController对应快照操作的的API接口。

app接口类APIRouter实现了\_setup\_routes方法，该方法加载不同类型的Controller，并且为这些Controller定义路由规则。

cinder.api.v2.router. APIRouter

|  |
| --- |
| class APIRouter(cinder.api.openstack.APIRouter):  """Routes requests on the API to the appropriate controller and method."""  ExtensionManager = extensions.ExtensionManager  def \_setup\_routes(self, mapper, ext\_mgr):  …….  # 将VolumeController对应的资源注册到Python Routes中，  # 这样 POST /v2/<tenant\_id> /volumes/<volume\_id>/action，  # 以及GET /v2/<tenant\_id> /volumes/detail  # 格式的URL请求就被定向到VolumeController类中执行  self.resources['volumes'] = volumes.create\_resource(ext\_mgr)  mapper.resource("volume", "volumes",  controller=self.resources['volumes'],  collection={'detail': 'GET'},  member={'action': 'POST'})  # 其他核心资源  ……. |

**OpenStack中的资源管理**

OpenStack 定义了两种类型的资源，及核心资源和扩展资源。

**Core resource核心资源：**

核心资源的定义文件在 /cinder/api/v2/ 目录下面。Cinder 的核心资源包括：volumes，types，snapshots，limits等。APIRouter类在初始化时加载的就是核心资源

**Extension resource扩展资源：**

扩展资源定义在 ./cinder/api/contrib/ 目录下，这些文件定义的都是扩展资源，例如 quotas.py 等。扩展资源又分为两种情况：一种扩展资源本身也是一种资源，只是没那么核心，比如 os-quota-sets。对扩展资源的访问方法同核心资源，比如 PUT /v2/< tenant\_id >/os-quota-sets/<quota\_id>；另一种扩展资源是对核心资源的扩展(Resource extension)，包括对 action 的扩展和基本操作（如GET）的扩展，现在的 Cinder 中只有对 Resource 基本操作的扩展，例如 SchedulerHints 是对 volumes 提供了扩展方法。一些扩展资源同时具备这两种功能。

APIRouter 是 Cinder 中的核心类之一，它负责分发 HTTP Request 到其管理的某个 Resource。核心资源、扩展资源、以及核心资源扩展都是在APIRouter的\_\_init\_\_方法中被加载的。

它的几个重要属性：

**resources：**

保存所有的 Resource 的 Controller 类的instance；每个Resource 拥有一个该数组的数组项，比如 self.resources['versions'] = versions.create\_resource() 会为 versions 核心资源创建一个 Resource 并保存到 resources 中。

**mapper:**

用来保存保存所有routes信息给Middleware 使用。它其实是一张路由表。 每个表项表示一个 URL 和 controller 以及 action 的映射关系。

**routes：**

routes.middleware.RoutesMiddleware 的实例，它其实是一个 WSGI app，它使用 mapper 和 \_dispatch\_进行初始化。功能是根据 URL 得到 controller 和它的 action。

|  |
| --- |
| def \_\_init\_\_(self, ext\_mgr=None):  #初始化 ext\_mgr，用于管理所有的 extensions  # ExtensionManager在初始化时根据配置项CONF. osapi\_volume\_extension  #加载cinder/api/contrib目录下的所有py文件，每个python文件就是一个扩展资源。  if ext\_mgr is None:  if self.ExtensionManager:  ext\_mgr = self.ExtensionManager()  else:  raise Exception(\_("Must specify an ExtensionManager class"))  #生成 mapper  mapper = ProjectMapper()  #初始化 resources 数组  self.resources = {}  #\_setup\_routes会将Controller资源加到self.resource中，并且在mapper中添加路由信息  self.\_setup\_routes(mapper, ext\_mgr)  #\_setup\_ext\_routes会将self.ExtensionManager中的所有资源加到self.resource中，  # 并且在mapper中添加路由信息  self.\_setup\_ext\_routes(mapper, ext\_mgr)  #\_setup\_extensions加载所有资源扩展  self.\_setup\_extensions(ext\_mgr)  super(APIRouter, self).\_\_init\_\_(mapper) |