policy本身的实现机制并不难，对我来说，难就难在python语法上，policy用到了很多高级的语法，逻辑性比较复杂，要理清其中的关系，还是要费一番功夫的。为此，还总结了另一篇blog，介绍了一下policy中用到的较为经典的语法。

1. 首先还是先来了解一下什么是policy，它是用来做什么的

在openstack的用户管理中，有三个概念：Users, Tenants, Roles。简单来说，policy就是用来控制某一个User在某个Tenant中的权限的。这个User能执行什么操作，不能执行什么操作，就是通过policy机制来实现的。直观的看，policy就是一个json文件，位于/etc/[SERVICE\_CODENAME]/policy.json中，每一个服务都有一个对应的policy.json文件，通过配置这个文件，实现了对User的权限管理。

另外就是还有一个角色(role)的概念，这个概念肯定都很熟悉了，是权限的集合，可以将role赋予某个user，使这个user拥有相应的权限，方便用户权限管理。policy.json文件可以在role的级别配置，不过默认的配置的角色只有admin，如果需要配置其他的角色，需要自己创建，然后在policy.json中进行配置。

接下来，看一下policy.json长什么样子：

查看文本打印?

1. {
2. "context\_is\_admin":  "role:admin",
3. "admin\_or\_owner":  "is\_admin:True or project\_id:%(project\_id)s",
4. "default": "rule:admin\_or\_owner",
5. "compute:create": "role:admin",
6. "compute:create:attach\_network": "",
7. "compute:create:attach\_volume": "",
8. "compute:create:forced\_host": "is\_admin:True",
9. "compute:get\_all": "",
10. ......
11. }

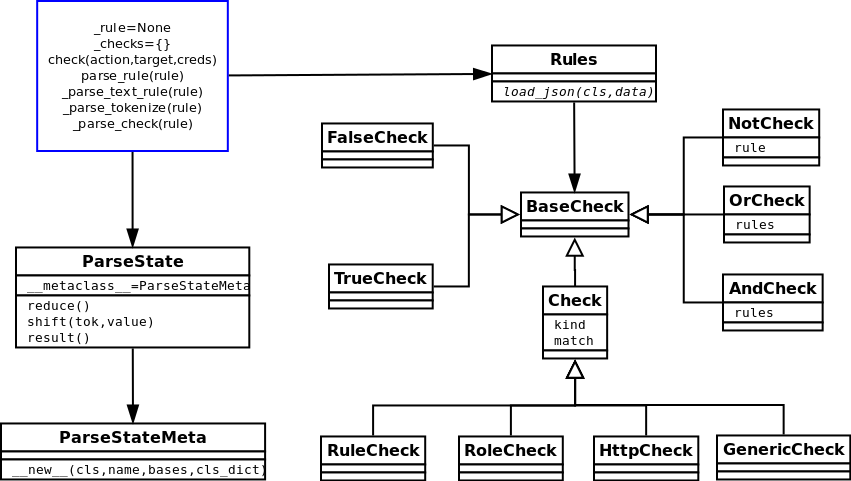
policy.json有两种写法，一种是每一行写成列表形式的，另一种就是上面的例子，是写成字符串形式的，这里只说后面一种情况。

每一行可以分为两部分：冒号前面的叫做action，即用户执行的操作，冒号后面的叫rule，即用来根据当前的上下文(context)，来判断前面的action是否能够由当前的user执行。policy做的主要工作，就是来解析这个rule的，要把这个字符串的rule，解析成相应的对象，在外部调用policy进行权限认证的时候，根据action映射到相应的rule对象，然后由这个对象判断是否能够执行这个action。

但是如上面的例子，像context\_is\_admin，admin\_or\_owner这样的action，怎么看都不像action，它们的确不是action，后面会讲到是怎么回事。

2. 如何实现

这部分功能的实现主要在nova/openstack/common/policy.py模块中实现，首先还是先来看看这个模块的整体结构图：



首先就是那个Rules类，它继承自dict，也就是说Rules是一个字典类型的，它的对象就是保存解析之后的policy.json文件得到的数据的，key保存的是action，value保存的是解析rule得到的对象。这个Rules对象被创建后，会赋值给模块中的\_rules变量。

其次是BaseCheck体系类，它们就对应的是rule字符串解析之后得到的对象，即Rules对象的value保存的就是BaseCheck对象。这个类体系可以分为三类：

1）TrueCheck和FalseCheck：解析他们是最简单的，rule如果是空字符串或者是"@"的话，那么就直接对应的是TrueCheck对象，这个对象的返回值总为True；如果rule是"!"的话，那么就对应FalseCheck对象，总是返回False，如果rule不对应任何对象，那么它就最终对应FalseCheck对象

2）RuleCheck, RoleCheck, HttpCheck, GenericCheck：rule字符串中，冒号左边是"rule"的，都会解析成RuleCheck对象，冒号左边是"role"的都会解析成RoleCheck，冒号左边是"http"的都会解析成RoleCheck对象，如果是其他的情况，那么就对应GenericCheck对象，比如rule为is\_admin:True的情况。

3）OrCheck, AndCheck, NotCheck：这些对象对应的是复合的rule，即用逻辑符号连接的几条规则，比如上面例子中的"is\_admin:True or project\_id:%(project\_id)s"，它对应的对象为OrCheck类型。上面两种情况的解析都很简单，难的就是这个复合rule的解析，费了一番功夫。上面类图中的ParseState类和ParseStateMeta类就是用来解析复合rule的。

如果不深究每条rule是怎么解析的话，看一个函数就够了，即模块中的\_parse\_check()函数，从这个函数中，就可以知道每条规则对应哪种对象：

查看文本打印?

1. # 真正的解析一个单一的rule,将它由冒号分隔开，得到kind和match，根据kind值，在\_check查找对应的Check对象，
2. # 然后以kind,match为参数，调用Check对象的\_\_call\_\_()返回最终的结果:真或假
3. **def** \_parse\_check(rule):
4. """
5. Parse a single base check rule into an appropriate Check object.
6. """
8. # Handle the special checks
9. **if** rule == '!':
10. **return** FalseCheck()
11. **elif** rule == '@':
12. **return** TrueCheck()
14. **try**:
15. kind, match = rule.split(':', 1)
16. **except** Exception:
17. LOG.exception(\_("Failed to understand rule %(rule)s") % locals())
18. # If the rule is invalid, we'll fail closed
19. **return** FalseCheck()
21. # Find what implements the check
22. **if** kind **in** \_checks:
23. **return** \_checks[kind](kind, match)#这里竟然调用的是Check的\_\_init\_\_()，把kind和match赋值给Check中的kind和match
24. **elif** None **in** \_checks:
25. **return** \_checks[None](kind, match)
26. **else**:
27. LOG.error(\_("No handler for matches of kind %s") % kind)
28. **return** FalseCheck()

如果要深究每条rule是如何解析的话，请移步这里：

最后，再来说一下转换成的这些对象有什么用，为什么要费这么大劲转换成对象，直接用字符串的rule来判断不好吗，以及怎么用这些对象去判断用户的操作是否合法？

为什么要将字符串转换成对象？这个理由太好说了，就是为了抽象，抽象是为了方便，是为了以不变应万变，这就是面向对象的好处。比如此处的Check对象，就抽象出了kind和match成员变量，分别对应rule字符串的冒号左边和右边的内容。当用这些对象来判断用户操作是否合法时，是这样来使用的：result = \_rules[action](target, creds)，因为Rules类被创建后会赋值给\_rules变量，所以这里的\_rules变量就代表Rules对象，而Rules对象又是一个字典类型的，key是action，value是BaseCheck对象，所以就相当于是直接在调用BaseCheck对象的\_\_call\_\_()方法了，参数分别是target和creds，target是action要操作的目标，creds是当前的上下文环境，再结合Check对象中的kind和match，就可以根据相应的逻辑来判断这个操作是否合法了。举个简单的例子，看RoleCheck是如何来判断的：

查看文本打印?

1. @register("role")
2. **class** RoleCheck(Check):
3. **def** \_\_call\_\_(self, target, creds):
4. """Check that there is a matching role in the cred dict."""
6. **return** self.match.lower() **in** [x.lower() **for** x **in** creds['roles']]

比如规则"role:admin"，kind为"role"，match为"admin"，所以RoleCheck就是判断一下admin这个用户是否在creds上下文中，如果在，就返回真，不在返回假。

至于OrCheck等对象判断是否合法则更简单，在他们内部都维护了一个rules列表，存放的是每条单一规则对应的Check对象，在他们的\_\_call\_\_()方法中一个for循环，来判断，OrCheck为一真即真，AndCheck为一假即假。

上面还提到有些action看起来不像action的，的确，那样的规则，一般都会转换成GenericCheck对象，而RuleCheck对象的\_\_call\_\_()在判断用户操作是否合法时，是采用递归的方法来判断的，比如下面的例子：

查看文本打印?

1. "compute\_extension:accounts": "rule:admin\_api", #-->RuleCheck
2. "admin\_api": "is\_admin:True", #-->GenericCheck

RuleCheck对象通过递归调用，最终调用了GenericCheck对象的\_\_call\_\_()方法，得出最终的结果。至于，为什么要这样做，我现在还不是很清楚。

3. 如何使用

如何使用上面已经说的差不多了，在外部只要调用nova/policy.py模块中的enforce()即可：

查看文本打印?

1. **def** enforce(context, action, target, do\_raise=True):
2. init()#读取json文件，解析，并将解析的内容封装成一个Rules对象，然后把这个对象赋值给\_rules变量
4. credentials = context.to\_dict()
6. extra = {}
7. **if** do\_raise:
8. extra.update(exc=exception.PolicyNotAuthorized, action=action)
10. **return** policy.check(action, target, credentials, \*\*extra)

有意思的是每次调用enfore()，都会去重新读取一次policy.json文件，并且重新进行一次解析，所以，对json文件的修改，是起实时作用的，不需要重启任何服务，修改之后，只要调用enfore()就会起作用，这很方便。

4. 测试

说了这么多，不能光说不练啊，这里还是举个创建实例的例子，通过调试的手段，来具体看一下效果。测试的过程如下：  
（1）修改程序  
在nova/compute/api.py中的\_check\_create\_policies()方法中添加如下几句代码：

查看文本打印?

1. **print** '\*'\*30
2. creds=context.to\_dict()
3. **print** creds['roles']
4. **print** '\*'\*30
5. **raise** Exception

即打印出当前上下文中的roles。然后抛出异常，停止。

（2）启动各项服务

（3）修改policy.json文件  
将 "compute:create" : "" 这条规则修改为："compute:create": "role:admin", 即将原来的任何角色的用户都可以创建实例的权限修改为只有admin角色的用户可以创建实例。

（4）创建实例，观看异常

查看文本打印?

1. $ source openrc demo demo
2. $ nova boot --flavor 1 --image 29936f2a-0e05-47e0-8d43-b9d579b107f9 --key-name pubkey-01 instance-01
3. ERROR: Policy doesn't allow compute:create to be performed. (HTTP 403) (Request-ID: req-bd0cda7e-2207-46af-becb-72a3a3bec598)

看，报出了异常，说policy不允许compute:create被执行，说明修改的规则生效了。  
如果使用admin角色的用户创建实例是没有问题的。也可以查看日志，看上面修改的那段代码输出的当前上下文中的角色都是什么，有没有admin角色。

======================over=======================