[一个简单RPC框架是如何炼成的(VI)——引入服务注册机制](http://blog.csdn.net/crylearner/article/details/46952977)

开局篇我们说了，RPC框架的四个核心内容

1. **RPC数据的传输**。
2. **RPC消息 协议**
3. **RPC服务注册**
4. **RPC消息处理**

接下来处理RPC服务的注册机制。所谓注册机制，就是Server需要声明支持哪些rpc方法，然后当客户端发送调用某个声明的rpc方法之后，服务端能自动找到执行该请求的具体方法。以实际的例子为例，这是现在server端处理RPC请求的代码

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952977) [copy](http://blog.csdn.net/crylearner/article/details/46952977)

1. **def** procRequest(self):
2. # 循环读取并处理收到的客户端请求
3. **while** True:
4. req = self.conn.recv()
5. rsp = Response()
6. rsp.id = req.id
7. **if** req.command == 'sayHello':
8. rsp.result = self.sayHello()
9. **elif** req.command == 'whoAreYou':
10. rsp.result = self.whoAreYou()
11. **else**:
12. **raise** Exception("unknown command")
14. self.conn.send(rsp)

上面的代码有一个很不好的地方，很难稳定。Server端每次新增一个支持的rpc方法，就要修改这个procRequest方法。有什么办法可以避免吗？有，就是引入服务注册机制。在这里，实际就是将command与具体的function object绑定起来，说穿了就是生成一个dict，

{‘sayHello’ : self.sayHello,     'whoAreYou': self.whoAreYou}。

有这样的dict之后，收到req 之后，只要提取出command字段，然后从dict中找出对应的function，调用该function即可。  
  
  
  
基本想法已定，

**首先我们实现一个比较原始的服务注册机制。**

这个实现很简单，self.services就是上面的dict。通过register（）去注册服务，通过get\_service（）去获取服务名对应的function

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952977) [copy](http://blog.csdn.net/crylearner/article/details/46952977)

1. **class** ServiceRegister(object):
2. '''''
3. @服务注册  不考虑线程安全，这里简化起见，也不引入反射机制。
4. '''

7. **def** \_\_init\_\_(self):
8. '''''
9. Constructor
10. '''
11. self.services = {}
13. ## 注册具体的服务
14. #  @param servicename: 服务名
15. #  @param obj: 具体的对象
16. **def** register(self, obj, servicename):
17. **if** servicename **in** self.services:
18. **print**('warning: %s is already registered' % servicename)
19. **else**:
20. self.services[servicename] = obj
22. **def** get\_service(self, servicename):
23. **return** self.services[servicename]
25. **def** list\_service(self, servicename=None):
26. **if** servicename:
27. **return** str({servicename, self.services[servicename]})
28. **else**:
29. **return** str(self.services)

使用时，就是这个样子的

服务注册：

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952977) [copy](http://blog.csdn.net/crylearner/article/details/46952977)

1. self.services.register(self.sayHello, 'Server.sayHello', )
2. self.services.register(self.whoAreYou, 'Server.whoAreYou')
3. self.services.register(self.add, 'Server.add')

服务查找

**[html]** [view plain](http://blog.csdn.net/crylearner/article/details/46952977) [copy](http://blog.csdn.net/crylearner/article/details/46952977)

1. def proc(self, req):
2. rsp = Response()
3. rsp.id = req.id
4. rsp.result = ServiceCaller.call(self.services.get\_service(req.command), req.parameter)
5. ......

上面serviceCaller的实现，就是在RPC消息，实现带参数的RPC请求中，提到的 func(\*\*args)的技巧

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952977) [copy](http://blog.csdn.net/crylearner/article/details/46952977)

1. **class** ServiceCaller():
2. **def** \_\_init\_\_(self):
3. **pass**
5. @classmethod
6. **def** call(cls, caller, parameter):
7. **if** **not** parameter **or** len(parameter) == 0:
8. **return** caller()
9. **return** caller(\*\*parameter)

**下面我再引入一个自动注册服务的实现**

直接上代码

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952977) [copy](http://blog.csdn.net/crylearner/article/details/46952977)

1. **class** AutoServiceRegister(AbstractServiceRegister):
2. **def** register\_class(self, obj, predicate=None):
3. **if** **not** (hasattr(obj, '\_\_class\_\_') **and** inspect.isclass(obj.\_\_class\_\_)):
4. **return** False
5. servicename = obj.\_\_class\_\_.\_\_name\_\_
6. **for** (name, attr) **in** inspect.getmembers(obj, predicate):
7. # 系统方法或者私有方法，不添加
8. **if** name.startswith('\_\_') **or** name.startswith('\_' + servicename + '\_\_'): **continue**
9. #print(name)
10. **if** inspect.ismethod(attr): self.register\_method(attr)
11. **elif** inspect.isfunction(attr): self.register\_function(attr, servicename)
12. **return** True

使用

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952977) [copy](http://blog.csdn.net/crylearner/article/details/46952977)

1. **if** \_\_name\_\_ == '\_\_main\_\_':
2. **class** AServer(object):
3. **def** \_\_init\_\_(self):
4. **pass**
6. **def** sayHello(self):
7. **return** 'Hello World'
9. **def** whoAreYou(self):
10. **return** 'I am server'
12. **def** \_\_kaos(self):
13. **pass**
15. **def** \_kaos(self):
16. **pass**
18. obj = AServer()
20. service = AutoServiceRegister()
21. **print**(service.register\_class(obj))
22. **print**(service.list\_services())
23. **print**(service.get\_service('AServer.sayHello'))

执行结果如下

**[html]** [view plain](http://blog.csdn.net/crylearner/article/details/46952977) [copy](http://blog.csdn.net/crylearner/article/details/46952977)

1. True
2. {'AServer': {'sayHello': **<bound** method AServer.sayHello of **<\_\_main\_\_.AServer** object at 0x000000000294EA90**>>**, 'whoAreYou': **<bound** method AServer.whoAreYou of **<\_\_main\_\_.AServer** object at 0x000000000294EA90**>>**, '\_kaos': **<bound** method AServer.\_kaos of **<\_\_main\_\_.AServer** object at 0x000000000294EA90**>>**}}
3. **<bound** method AServer.sayHello of **<\_\_main\_\_.AServer** object at 0x000000000294EA90**>>**

详细说明 一下原理，利用了类似的反射的技术。有兴趣的同学可以先去了解一下inspect

* register\_class表示自动搜索一个类对象中的成员方法，并将其作为server端的rpc方法注册进去。  
  以上面AServer为例， 会自动将sayHello， whoAreYou 这两个方法自动注册进来。同时像\_\_init\_\_, \_\_kaos, \_kaos之类的系统固有方法，或者私有方法，会自动剔除。
* 注册时，传入的参数obj必须是class的instance，也就是类实例。虽然在python中，也支持类对象，但如果直接传递类对象，就会遇到如何初始化的难题。所以这里一致要求，必须是类的实例。

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952977) [copy](http://blog.csdn.net/crylearner/article/details/46952977)

* 1. **if** **not** (hasattr(obj, '\_\_class\_\_') **and** inspect.isclass(obj.\_\_class\_\_)):
  2. **return** False

类实例的特点就是，包含\_\_class\_\_成员，而且\_\_class\_\_成员的值就是该类的类对象。inspect.isclass就是检测是不是类对象

* inspect.getmembers（）返回的是类对象的所有成员，包括系统固有方法以及私有方法  
  所以，先要将系统方法和私有方法剔除，然后通过inspect，检查该成员是不是真的是function，就是可以被调用的。如果是，就注册进来
* register\_fucntion, register\_method与普通的服务注册基本一样。就是添加(key，value)对。

总结：

1. 引入服务注册的方式也是为了代码解耦，将req的处理与具体的req消息内容解耦。

2. 上面我们 引入了两种服务注册的方式，一种方式是普通的方式，逐个添加方法。另一种方式通过python的“反射”技术，自动查找一个类里面的方法，并自动添加。

3. 方案还是很粗糙的，实际有很多优化的地方。