[一个简单RPC框架是如何炼成的(V)——引入传输层](http://blog.csdn.net/crylearner/article/details/46952765)

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**

接下来处理数据传输。实际应用场景一般都是基于socket。socket代码比较多，使用起来也比较麻烦。而且具体的传输通道使用socket或者其他的方式，如更上层的http，或者android里的binder，都是可替换的，只是具体的一种实现而已。所以，这里我就偷个懒，只是引入一个很简单的Connection类，用来描述一下如何将数据传输 这一层给独立出来。

首先简单列出Connection类的实现，很简单，就是两个list，一个管发送，一个管接收。（实现没有考虑多线程安全，实际是必须考虑的）。

需要说明的是，这里的recv的实现约定是阻塞式的，也就是如果没有收到任何数据，recv调用会一直阻塞。

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952765) [copy](http://blog.csdn.net/crylearner/article/details/46952765)

1. **class** Connection(object):
2. '''''
3. @RPC 连接。一般说来，都是socket连接，这里简化起见，直接本地变量实现。
4. '''

7. **def** \_\_init\_\_(self, sending\_msg\_list, recving\_msg\_list):
8. '''''
9. Constructor
10. '''
11. self.sending\_msg\_list = sending\_msg\_list
12. self.recving\_msg\_list = recving\_msg\_list
14. **def** send(self, message):
15. self.sending\_msg\_list.append(message)
17. **def** recv(self):
18. **while** len(self.recving\_msg\_list) == 0: time.sleep(0.01)
19. **return** self.recving\_msg\_list.pop(0)
21. **def** isClosed(self):
22. **return** False

有了这个connection，剩下的就只要将rpc消息统统通过这个connection去发送，通过这个Connection去接收。

接着修改客户端的request请求，不再直接调用server端的procRequest方法，而是将请求交给connection，去发送。  然后等待connection收到server端的回复，将回复消息从connection中取出来。

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952765) [copy](http://blog.csdn.net/crylearner/article/details/46952765)

1. **def** request(self, req):
2. # 全部简化处理，不考虑线程安全问题，不考虑异步
3. # 先是将RPC消息发送到服务端，然后服务端就会处理，并将结果发回到客户端，客户端这边接收处理结果。
4. # self.remote.procRequest(req) // 删除
5. self.conn.send(req)
6. rsp = self.conn.recv()
7. **return** rsp.result

同样的，修改服务端收到request请求后的处理。首先反复调用connection.recv()方法读取客户端发过来的请求。当请求处理完成后，不再直接以函数返回的方式return，而是将rsp交给connection，由connection负责传输给client

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952765) [copy](http://blog.csdn.net/crylearner/article/details/46952765)

1. # def procRequest(self, req): 调整参数列表，不再需要req
2. **def** procRequest(self):
3. # 循环读取并处理收到的客户端请求
4. **while** True:
5. req = self.conn.recv()
6. rsp = Response()
7. rsp.id = req.id
8. **if** req.command == 'sayHello':
9. rsp.result = self.sayHello()
10. **elif** req.command == 'whoAreYou':
11. rsp.result = self.whoAreYou()
12. **else**:
13. **raise** Exception("unknown command")
15. # return rsp  # rsp也是通过connection最终传给client，而不是直接函数返回
16. self.conn.send(rsp)

最后，列一下connection的初始化

**[python]** [view plain](http://blog.csdn.net/crylearner/article/details/46952765) [copy](http://blog.csdn.net/crylearner/article/details/46952765)

1. slist = []
2. rlist = []
3. client = Client(Connection(slist, rlist))
4. server = Server(Connection(rlist, slist))
5. server.start()

总结，引入传输层的意义在于

1. 实现client与server端的解耦，client端不再需要持有server端的对象了。 这也是实现“远程调用 ”所必须的。

2. 传输层的实现有很大的自由度，一般说来，他无需关心具体的RPC消息的格式，只需要完成数据的可靠传输就可以了。

3. 传输层具体基于socket，binder， 是采用http，udp，tcp这些都是自由的，根据需要选择就可以了。也就是相当于一个可以自由拼接的组件。

4. 上面的模型实在过于简单，没有考虑多线程保护，没有考虑异常。实际比较理想的情况，应该起码有个类，Connector，以及Channel。其中channel只负责数据的传输，Connector负责管理channel。

后续如果有时间，会再进行完善