**RPC、REST API深入理解**

**一：RPC**

RPC 即远程过程调用（Remote Procedure Call Protocol，简称RPC），**像调用本地服务(方法)一样调用服务器的服务(方法)**。通常的实现有 **XML-RPC , JSON-RPC** , 通信方式基本相同, 所不同的只是传输数据的格式.

RPC是**分布式架构的核心**，按**响应方式**分如下两种：

**同步调用：客户端调用服务方方法，等待直到服务方返回结果或者超时，再继续自己的操作**

**异步调用：客户端把消息发送给中间件，不再等待服务端返回，直接继续自己的操作。**

同步调用的实现方式有**WebService和RMI**。Web Service提供的服务是基于web容器的，底层使用http协议，因而适合不同语言异构系统间的调用。RMI实际上是Java语言的RPC实现，允许方法返回 Java 对象以及基本数据类型，适合用于JAVA语言构建的不同系统间的调用。

异步调用的JAVA实现版就是JMS(Java Message Service)，目前开源的的JMS中间件有Apache社区的ActiveMQ、Kafka消息中间件，另外有阿里的RocketMQ。

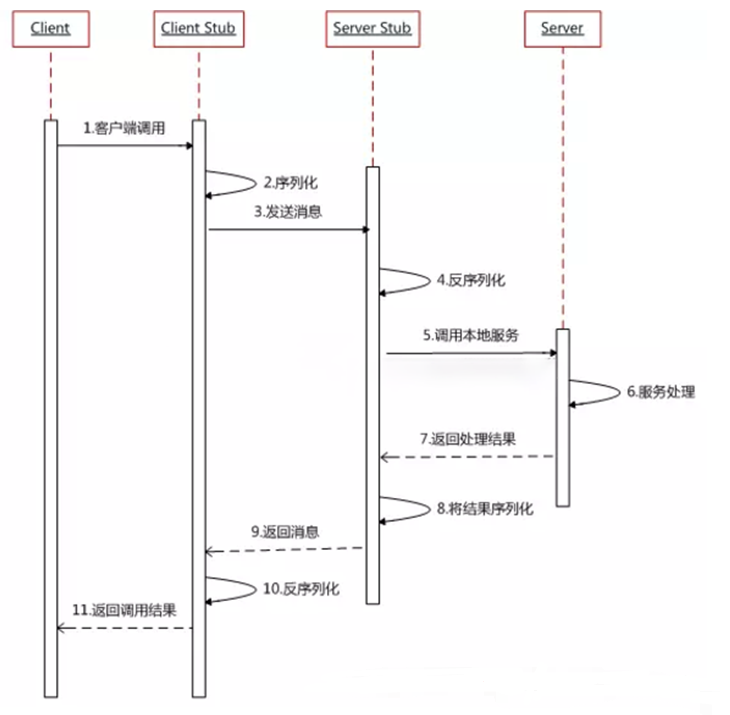
RPC架构里包含如下4个组件:

1、 **客户端(Client)：**服务调用方

2、 **客户端存根(Client Stub)：**存放服务端地址信息，将客户端的请求参数打包成网络消息，再通过网络发送给服务方

3、 **服务端存根(Server Stub)：**接受客户端发送过来的消息并解包，再调用本地服务

**4、 服务端(Server)：**真正的服务提供者。



具体实现步骤：

1、 服务调用方（client）**(客户端)以本地调用方式调用服务**；

2、 client stub接收到调用后负责**将方法、参数等组装成能够进行网络传输的消息体；**在Java里就是序列化的过程

3、 client stub**找到服务地址**，并将消息通过网络发送到服务端；

4、 server stub收到**消息后进行解码,在Java里就是反序列化的过程；**

5、 server stub根据**解码结果调用本地的服务**；

6、 本地服务**执行处理逻辑；**

7、 本地服务将结果返回给**server stub；**

8、 server stub将返回结果打包成消息，Java里的序列化；

9、 server stub将打包后的消息通过网络并发送至消费方

10、 client stub接收到消息，并进行解码, Java里的反序列化；

11、 服务调用方（client）**得到最终结果。**

RPC框架的目标就是**把2-10步封装起来**，**把调用、编码/解码的过程封装起来，让用户像调用本地服务一样的调用远程服务。**要做到对客户端（调用方）透明化服务， RPC框架需要考虑解决如下问题：   
1、通讯问题 : 主要是通过在**客户端和服务器之间建立TCP连接**，远程过程调用的**所有交换的数据都在这个连接里传输。**连接可以是按需连接，调用结束后就断掉，也可以是长连接，多个远程过程调用共享同一个连接。   
2、寻址问题 ： A服务器上的应用怎么告诉底层的RPC框架，如何连接到B服务器（如主机或IP地址）以及特定的端口，方法的名称是什么，这样才能完成调用。比如基于Web服务协议栈的RPC，就要提供一个endpoint URI，或者是从UDDI服务上查找。**如果是RMI调用的话，还需要一个RMI Registry来注册服务的地址。**3、序列化与反序列化 ： 当A服务器上的应用发起远程过程调用时，**方法的参数需要通过底层的网络协议如TCP传递到B服务器，由于网络协议是基于二进制的，内存中的参数的值要序列化成二进制的形式，也就是序列化（Serialize）或编组（marshal），通过寻址和传输将序列化的二进制发送给B服务器。   
同理，B服务器接收参数要将参数反序列化。**B服务器应用调用自己的方法处理后返回的结果也要序列化给A服务器，A服务器接收也要经过反序列化的过程。

**二：REST**

  REST即**表述性状态传递**（Representational State Transfer，简称REST），是一种软件架构风格。REST通过HTTP协议定义的通用动词方法(GET、PUT、DELETE、POST) ，以**URI对网络资源进行唯一标识**，响应端根据请求端的不同需求，**通过无状态通信**，对其请求的资源进行表述。   
  Rest架构的主要原则：

1. **网络上的所有事物都被抽象为资源**
2. **每个资源都有一个唯一的资源标识符（url代表一个实体，其他的请求操作和表现形式等可以放到请求头去）**
3. **同一个资源具有多种表现形式(xml,json等),通过http协议的context-type等去表现**
4. **对资源的各种操作不会改变资源标识符**
5. **所有的操作都是无状态的**

其中表述性状态，是指(在某个瞬间状态的)资源数据的快照，包括**资源数据的内容、表述格式(XML、JSON)等信息**。

其中无状态通信，是指**服务端(响应端)不保存任何与特定HTTP请求相关的资源，应用状态必须由请求方在请求过程中提供。**要求在网络通信过程中，任意一个Web请求必须与其他请求隔离，**当请求端提出请求时，请求本身包含了响应端为响应这一请求所需的全部信息。**

REST使用**HTTP+URI+XML /JSON** 的技术来实现其API要求的架构风格：**HTTP协议和URI用于统一接口和定位资源，文本、二进制流、XML、JSON等格式用来作为资源的表述**。

**二．Restful web service的优点：**

2.1 HTTP头中可见的统一接口和资源地址

**通过对于HTTP Head 的解析，我们便可以了解到当前所请求的资源和请求的方式。这样做**对于一些代理服务器的设置，将带来很高的处理效率。

**REST 系统中所有的动作和要访问的资源都可以从HTTP和URI中得到，**这使得代理服务器、缓存服务器和网关很好地协调工作。而RPC模型的SOAP 要访问的资源仅从 URI无法得知，要调用的方法也无法从HTTP中得知，它们都隐藏在 SOAP 消息中。

同样的，在REST系统中的代理服务器还可以通过 HTTP 的动作 (GET 、 POST)来进行控制。

2.2 返回一般的XML格式内容

一般情况下，一个RESTful Web Service将比一个传统SOAP RPC Web Service占用更少的传输带宽。

2.3 安全机制

**REST使用了简单有效的安全模型。REST中很容易隐藏某个资源，只需不发布它的URI；**而在资源上也很容易使用一些安全策略，比如可以在每个 URI 针对 4个通用接口设置权限；再者，以资源为中心的 Web服务是防火墙友好的，因为 GET的 意思就是GET， PUT 的意思就是PUT，管理员可以通过堵塞非GET请求把资源设置为只读的，而现在的基于RPC 模型的 SOAP 一律工作在 HTTP 的 POST上。而使用 SOAP RPC模型，要访问的对象名称藏在方法的参数中，因此需要创建新的安全模型。

举例：   
在Restful之前的操作： 请求的地址对应具体的业务操作   
<http://127.0.0.1/user/query/1> GET 根据用户id查询用户数据   
<http://127.0.0.1/user/save> POST 新增用户   
<http://127.0.0.1/user/update> POST 修改用户信息   
<http://127.0.0.1/user/delete> GET/POST 删除用户信息

**RESTful用法：** 请求   
<http://127.0.0.1/user/1> GET 根据用户id查询用户数据   
<http://127.0.0.1/user> POST 新增用户   
<http://127.0.0.1/user> PUT 修改用户信息   
<http://127.0.0.1/user> DELETE 删除用户信息

RESTful风格的体现，在你使用了get请求，就是查询；使用post请求,就是新增的请求；使用put请求，就是修改的请求；使用delete请求，就是删除的请求。这样做就完全没有必要对crud做具体的描述。

**满足REST约束条件和原则的架构，就被称为是RESTful架构**。就像URL都是URI(统一资源标识)的表现形式一样，RESTful是符合REST原则的表现形式。

看Url就知道要操作的资源是什么，是操作车辆还是围栏

看Http Method就知道操作动作是什么，是添加（post）还是删除（delete）

看Http Status Code就知道操作结果如何，是成功（200）还是内部错误（500）

**SpringMVC实现restful服务:**

**SpringMVC原生态的支持了REST风格的架构设计**

**所涉及到的注解:**

**--@RequestMapping**

**---@PathVariable**

**---@ResponseBody**

**[java]** [view plain](https://blog.csdn.net/chenxiaochan/article/details/73716617) [copy](https://blog.csdn.net/chenxiaochan/article/details/73716617)

1. **package** cn.itcast.mybatis.controller;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.http.HttpStatus;
5. **import** org.springframework.http.ResponseEntity;
6. **import** org.springframework.stereotype.Controller;
7. **import** org.springframework.web.bind.annotation.PathVariable;
8. **import** org.springframework.web.bind.annotation.RequestMapping;
9. **import** org.springframework.web.bind.annotation.RequestMethod;
10. **import** org.springframework.web.bind.annotation.RequestParam;
11. **import** org.springframework.web.bind.annotation.ResponseBody;
13. **import** cn.itcast.mybatis.pojo.User;
14. **import** cn.itcast.mybatis.service.NewUserService;
16. @RequestMapping("restful/user")
17. @Controller
18. **public** **class** RestUserController {
20. @Autowired
21. **private** NewUserService newUserService;
23. /\*\*
24. \* 根据用户id查询用户数据
25. \*
26. \* @param id
27. \* @return
28. \*/
29. @RequestMapping(value = "{id}", method = RequestMethod.GET)
30. @ResponseBody
31. **public** ResponseEntity<User> queryUserById(@PathVariable("id") Long id) {
32. **try** {
33. User user = **this**.newUserService.queryUserById(id);
34. **if** (**null** == user) {
35. // 资源不存在，响应404
36. **return** ResponseEntity.status(HttpStatus.NOT\_FOUND).body(**null**);
37. }
38. // 200
39. // return ResponseEntity.status(HttpStatus.OK).body(user);
40. **return** ResponseEntity.ok(user);
41. } **catch** (Exception e) {
42. e.printStackTrace();
43. }
44. // 500
45. **return** ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(**null**);
46. }
48. /\*\*
49. \* 新增用户
50. \*
51. \* @param user
52. \* @return
53. \*/
54. @RequestMapping(method = RequestMethod.POST)
55. **public** ResponseEntity<Void> saveUser(User user) {
56. **try** {
57. **this**.newUserService.saveUser(user);
58. **return** ResponseEntity.status(HttpStatus.CREATED).build();
59. } **catch** (Exception e) {
60. // TODO Auto-generated catch block
61. e.printStackTrace();
62. }
63. // 500
64. **return** ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(**null**);
65. }
67. /\*\*
68. \* 更新用户资源
69. \*
70. \* @param user
71. \* @return
72. \*/
73. @RequestMapping(method = RequestMethod.PUT)
74. **public** ResponseEntity<Void> updateUser(User user) {
75. **try** {
76. **this**.newUserService.updateUser(user);
77. **return** ResponseEntity.status(HttpStatus.NO\_CONTENT).build();
78. } **catch** (Exception e) {
79. e.printStackTrace();
80. }
81. // 500
82. **return** ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(**null**);
83. }
85. /\*\*
86. \* 删除用户资源
87. \*
88. \* @param user
89. \* @return
90. \*/
91. @RequestMapping(method = RequestMethod.DELETE)
92. **public** ResponseEntity<Void> deleteUser(@RequestParam(value = "id", defaultValue = "0") Long id) {
93. **try** {
94. **if** (id.intValue() == 0) {
95. // 请求参数有误
96. **return** ResponseEntity.status(HttpStatus.BAD\_REQUEST).build();
97. }
98. **this**.newUserService.deleteUserById(id);
99. // 204
100. **return** ResponseEntity.status(HttpStatus.NO\_CONTENT).build();
101. } **catch** (Exception e) {
102. e.printStackTrace();
103. }
104. // 500
105. **return** ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(**null**);
106. }
107. }
108. 以Apache Thrift为代表的**二进制RPC，支持多种语言（但不是所有语言）**，四层通讯协议，性能高，节省带宽。相对Restful协议，使用Thrifpt RPC，在同等硬件条件下，带宽使用率仅为前者的20%，性能却提升一个数量级。但是这种协议最大的问题在于，**无法穿透防火墙。**
109. 以Spring Cloud为代表所支持的Restful 协议，优势在于**能够穿透防火墙，**使用方便，语言无关，基本上可以使用各种开发语言实现的系统，都可以接受Restful 的请求。 但性能和带宽占用上有劣势。

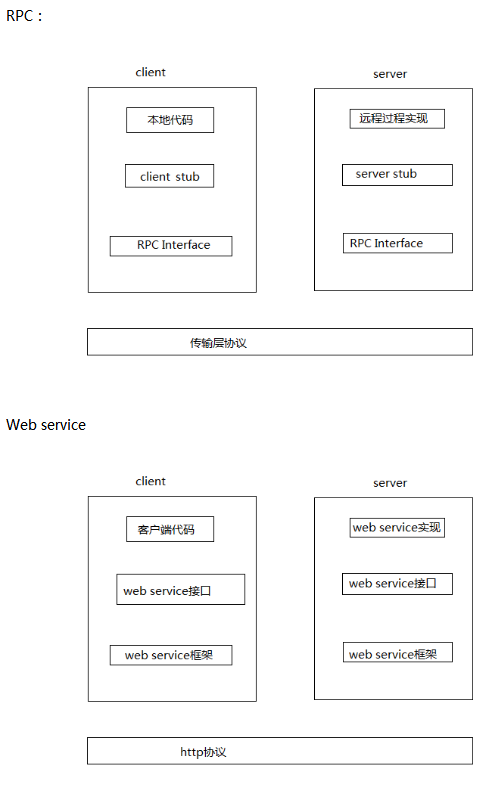
所以，业内对微服务的实现，基本是确定一个组织边界，在**该边界内，使用RPC; 边界外，使用Restful。**这个边界，可以是业务、部门，甚至是全公司。

使用RPC远程服务调用方式与传统http接口直接调用方式的差别在于：

1. 从使用方面看，**Http接口只关注服务提供方（服务端）**，对于客户端怎么调用，调用方式怎样并不关心，通常情况下，**客户端使用Http方式进行调用时，只要将内容进行传输即可，**这样**客户端**在使用时，需要**更关注网络方面的传输**，比较不适用与业务方面的开发；而RPC服务则**需要客户端接口与服务端保持一致，服务端提供一个方法，客户端通过接口直接发起调用，业务开发人员仅需要关注业务方法的调用即可，**不再关注网络传输的细节，在开发上更为高效。

2. 从性能角度看，使用Http时，Http本身提供了丰富的状态功能与扩展功能，但也正由于Http提供的功能过多，导致在网络传输时，需要携带的信息更多，从性能角度上讲，较为低效。而**RPC服务网络传输上仅传输与业务内容相关的数据**，传输数据更小，性能更高。

3. 从运维角度看，使用Http接口时，常常使用一个前端代理，来进行Http转发代理请求的操作，需要进行扩容时，则需要去修改代理服务器的配置，较为繁琐，也容易出错。而使用RPC方式的微服务，则只要增加一个服务节点即可，注册中心可自动感知到节点的变化，通知调用客户端进行负载的动态控制，更为智能，省去运维的操作。



1. 首先要解决寻址的问题，也就是说，**A服务器上的应用怎么告诉底层的RPC框架，**B服务器的IP，以及应用绑定的端口，还有方法的名称，这样才能完成调用
2. **方法的参数需要通过底层的网络协议如TCP传递到B服务器**，由于**网络协议是基于二进制的**，内存中的**参数的值要序列化**成二进制的形式
3. 在B服务器上**完成寻址后**，需要**对参数进行反序列化**，恢复为内存中的表达方式，然后找到**对应的方法进行本地调用**，然后得到**返回值，**
4. **返回值还要发送回服务器A上的应用**，也要经过序列化的方式发送，服务器A接到后，再反序列化，恢复为内存中的表达方式，交给应用

## 1、实现技术方案

     下面使用比较原始的方案实现RPC框架，采用Socket通信、动态代理与反射与Java原生的序列化。

## 2、RPC框架架构

     RPC架构分为三部分：

1. 服务提供者，运行在服务器端，提供服务接口定义与服务实现类。
2. 服务中心，运行在服务器端，负责将本地服务发布成远程服务，管理远程服务，提供给服务消费者使用。
3. 服务消费者，运行在客户端，通过远程代理对象调用远程服务。

## 3、 具体实现

1）服务提供者接口定义与实现，代码如下：

**[java]** [view plain](https://blog.csdn.net/rulon147/article/details/53814589) [copy](https://blog.csdn.net/rulon147/article/details/53814589)

1. **package** services;

4. **public** **interface** HelloService {

7. String sayHi(String name);

10. }

2）HelloServices接口实现类：

**[java]** [view plain](https://blog.csdn.net/rulon147/article/details/53814589) [copy](https://blog.csdn.net/rulon147/article/details/53814589)

1. **package** services.impl;

4. **import** services.HelloService;

7. **public** **class** HelloServiceImpl **implements** HelloService {

10. **public** String sayHi(String name) {
11. **return** "Hi, " + name;
12. }

15. }

3）服务中心代码实现，代码如下：

**[java]** [view plain](https://blog.csdn.net/rulon147/article/details/53814589) [copy](https://blog.csdn.net/rulon147/article/details/53814589)

1. **package** services;

4. **import** java.io.IOException;

7. **public** **interface** Server {
8. **public** **void** stop();

11. **public** **void** start() **throws** IOException;

14. **public** **void** register(Class serviceInterface, Class impl);

17. **public** **boolean** isRunning();

20. **public** **int** getPort();
21. }

4）服务中心实现类：

**[java]** [view plain](https://blog.csdn.net/rulon147/article/details/53814589) [copy](https://blog.csdn.net/rulon147/article/details/53814589)

1. **package** services.impl;





8. **import** java.io.IOException;
9. **import** java.io.ObjectInputStream;
10. **import** java.io.ObjectOutputStream;
11. **import** java.lang.reflect.Method;
12. **import** java.net.InetSocketAddress;
13. **import** java.net.ServerSocket;
14. **import** java.net.Socket;
15. **import** java.util.HashMap;
16. **import** java.util.concurrent.ExecutorService;
17. **import** java.util.concurrent.Executors;

20. **import** services.Server;

23. **public** **class** ServiceCenter **implements** Server {
24. **private** **static** ExecutorService              executor        = Executors.newFixedThreadPool(Runtime.getRuntime()
25. .availableProcessors());

28. **private** **static** **final** HashMap<String, Class> serviceRegistry = **new** HashMap<String, Class>();

31. **private** **static** **boolean**                      isRunning       = **false**;

34. **private** **static** **int**                          port;

37. **public** ServiceCenter(**int** port) {
38. **this**.port = port;
39. }

42. **public** **void** stop() {
43. isRunning = **false**;
44. executor.shutdown();
45. }

48. **public** **void** start() **throws** IOException {
49. ServerSocket server = **new** ServerSocket();
50. server.bind(**new** InetSocketAddress(port));
51. System.out.println("start server");
52. **try** {
53. **while** (**true**) {
54. // 1.监听客户端的TCP连接，接到TCP连接后将其封装成task，由线程池执行
55. executor.execute(**new** ServiceTask(server.accept()));
56. }
57. } **finally** {
58. server.close();
59. }
60. }

63. **public** **void** register(Class serviceInterface, Class impl) {
64. serviceRegistry.put(serviceInterface.getName(), impl);
65. }

68. **public** **boolean** isRunning() {
69. **return** isRunning;
70. }

73. **public** **int** getPort() {
74. **return** port;
75. }

78. **private** **static** **class** ServiceTask **implements** Runnable {
79. Socket clent = **null**;

82. **public** ServiceTask(Socket client) {
83. **this**.clent = client;
84. }

87. **public** **void** run() {
88. ObjectInputStream input = **null**;
89. ObjectOutputStream output = **null**;
90. **try** {
91. // 2.将客户端发送的码流反序列化成对象，反射调用服务实现者，获取执行结果
92. input = **new** ObjectInputStream(clent.getInputStream());
93. String serviceName = input.readUTF();
94. String methodName = input.readUTF();
95. Class<?>[] parameterTypes = (Class<?>[]) input.readObject();
96. Object[] arguments = (Object[]) input.readObject();
97. Class serviceClass = serviceRegistry.get(serviceName);
98. **if** (serviceClass == **null**) {
99. **throw** **new** ClassNotFoundException(serviceName + " not found");
100. }
101. Method method = serviceClass.getMethod(methodName, parameterTypes);
102. Object result = method.invoke(serviceClass.newInstance(), arguments);

105. // 3.将执行结果反序列化，通过socket发送给客户端
106. output = **new** ObjectOutputStream(clent.getOutputStream());
107. output.writeObject(result);
108. } **catch** (Exception e) {
109. e.printStackTrace();
110. } **finally** {
111. **if** (output != **null**) {
112. **try** {
113. output.close();
114. } **catch** (IOException e) {
115. e.printStackTrace();
116. }
117. }
118. **if** (input != **null**) {
119. **try** {
120. input.close();
121. } **catch** (IOException e) {
122. e.printStackTrace();
123. }
124. }
125. **if** (clent != **null**) {
126. **try** {
127. clent.close();
128. } **catch** (IOException e) {
129. e.printStackTrace();
130. }
131. }
132. }

135. }
136. }





143. }

5）客户端的远程代理对象：

**[java]** [view plain](https://blog.csdn.net/rulon147/article/details/53814589) [copy](https://blog.csdn.net/rulon147/article/details/53814589)

1. **package** client;

4. **import** java.io.ObjectInputStream;
5. **import** java.io.ObjectOutputStream;
6. **import** java.lang.reflect.InvocationHandler;
7. **import** java.lang.reflect.Proxy;
8. **import** java.net.InetSocketAddress;
9. **import** java.net.Socket;
10. **import** java.lang.reflect.Method;

13. **public** **class** RPCClient<T> {
14. @SuppressWarnings("unchecked")
15. **public** **static** <T> T getRemoteProxyObj(**final** Class<?> serviceInterface, **final** InetSocketAddress addr) {
16. // 1.将本地的接口调用转换成JDK的动态代理，在动态代理中实现接口的远程调用
17. **return** (T) Proxy.newProxyInstance(serviceInterface.getClassLoader(), **new** Class<?>[] { serviceInterface },
18. **new** InvocationHandler() {
19. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
20. Socket socket = **null**;
21. ObjectOutputStream output = **null**;
22. ObjectInputStream input = **null**;
23. **try** {
24. // 2.创建Socket客户端，根据指定地址连接远程服务提供者
25. socket = **new** Socket();
26. socket.connect(addr);

29. // 3.将远程服务调用所需的接口类、方法名、参数列表等编码后发送给服务提供者
30. output = **new** ObjectOutputStream(socket.getOutputStream());
31. output.writeUTF(serviceInterface.getName());
32. output.writeUTF(method.getName());
33. output.writeObject(method.getParameterTypes());
34. output.writeObject(args);

37. // 4.同步阻塞等待服务器返回应答，获取应答后返回
38. input = **new** ObjectInputStream(socket.getInputStream());
39. **return** input.readObject();
40. } **finally** {
41. **if** (socket != **null**)
42. socket.close();
43. **if** (output != **null**)
44. output.close();
45. **if** (input != **null**)
46. input.close();
47. }
48. }
49. });
50. }
51. }

6）最后为测试类：

**[java]** [view plain](https://blog.csdn.net/rulon147/article/details/53814589) [copy](https://blog.csdn.net/rulon147/article/details/53814589)

1. **package** client;

4. **import** java.io.IOException;
5. **import** java.net.InetSocketAddress;

8. **import** services.HelloService;
9. **import** services.Server;
10. **import** services.impl.HelloServiceImpl;
11. **import** services.impl.ServiceCenter;

14. **public** **class** RPCTest {
15. **public** **static** **void** main(String[] args) **throws** IOException {
16. **new** Thread(**new** Runnable() {
17. **public** **void** run() {
18. **try** {
19. Server serviceServer = **new** ServiceCenter(8088);
20. serviceServer.register(HelloService.**class**, HelloServiceImpl.**class**);
21. serviceServer.start();
22. } **catch** (IOException e) {
23. e.printStackTrace();
24. }
25. }
26. }).start();
27. HelloService service = RPCClient
28. .getRemoteProxyObj(HelloService.**class**, **new** InetSocketAddress("localhost", 8088));
29. System.out.println(service.sayHi("test"));
30. }

33. }

Restful里面的：（微服务里的）都要同时注册到服务的注册中心里面去。

FeignClient

 除了上面的方式，我们还可以用FeignClient。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @FeignClient(value = "users", path = "/users")  public interface UserCompositeService {  @RequestMapping(value = "/getUserDetail/{id}",          method = RequestMethod.GET,          produces = MediaType.APPLICATION\_JSON\_VALUE)  UserDTO getUserById(@PathVariable Long id);  } |

  我们只需要使用@FeignClient定义一个接口，Spring Cloud Feign会帮我们生成一个它的实现，从相应的users服务获取数据。

  其中，@FeignClient(value = “users”， path = “/users/getUserDetail”)里面的value是服务ID，path是这一组接口的path前缀。在下面的方法定义里，就好像设置Spring MVC的接口一样，对于这个方法，它对应的URL是/users/getUserDetail/{id}。然后，在使用它的时候，就像注入一个一般的服务一样注入后使用即可：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class SomeOtherServiceClass {  @Autowired  private UserCompositeService userService;  public void doSomething() {      // .....      UserDTO results = userService.getUserById(userId);      // other operation...  }  } |

从使用方面看，**Http接口只关注服务提供方（服务端）**，对于客户端怎么调用，调用方式怎样并不关心，通常情况下，**客户端使用Http方式进行调用时，只要将内容进行传输即可，**这样**客户端**在使用时，需要**更关注网络方面的传输**，比较不适用与业务方面的开发；**（restful是服务端把方法写好，客户端通过http方式调用，直接定位到方法上面去。）**

而RPC服务则**需要客户端接口与服务端保持一致，服务端提供一个方法，客户端通过接口直接发起调用，业务开发人员仅需要关注业务方法的调用即可，**不再关注网络传输的细节，在开发上更为高效。**（PRC是服务端提供好方法给客户端调用。定位到类，然后通过类去调用方法。）**



比如这种，自己要了一个计算服务，拿到计算服务类后，自己调用服务类里的加法去获得结果

如果是restful，就根据Calculate方法对应的url去传参（people），从而获得结果。

**RPC：所谓的远程过程调用 (面向方法)**

**SOA：所谓的面向服务的架构(面向消息)**

**REST：所谓的 Representational state transfer (面向资源)**

RPC 即远程过程调用, 很简单的概念, 像调用本地服务(方法)一样调用服务器的服务(方法).

通常的实现有 XML-RPC , JSON-RPC , 通信方式基本相同, 所不同的只是传输数据的格式.

REST 的三个要素是 唯一的资源标识, 简单的方法 (此处的方法是个抽象的概念), 一定的表达方式.

重要的特性：无状态