<https://blog.csdn.net/u014590757/article/details/80233901>

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

**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。）

## 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。

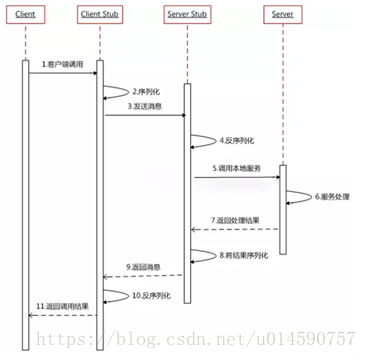
RPC架构里包含如下4个组件:

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

3. 同一个资源具有多种表现形式(xml,json等)

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。

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。

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

1.           以ApacheThrift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。**

2.           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、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。

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。

@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：