

RPC 服务方通过 RpcServer 去导出（export）远程接口方法，而客户方通过 RpcClient 去引入（import）远程接口方法。客户方像调用本地方法一样去调用远程接口方法，RPC 框架提供接口的代理实现，实际的调用将委托给代理RpcProxy 。代理封装调用信息并将调用转交给RpcInvoker 去实际执行。在客户端的RpcInvoker 通过连接器RpcConnector 去维持与服务端的通道RpcChannel，并使用RpcProtocol 执行协议编码（encode）并将编码后的请求消息通过通道发送给服务方。

RPC 服务端接收器 RpcAcceptor 接收客户端的调用请求，同样使用RpcProtocol 执行协议解码（decode）。解码后的调用信息传递给RpcProcessor 去控制处理调用过程，最后再委托调用给RpcInvoker 去实际执行并返回调用结果。

RPC 组件职责

上面我们进一步拆解了 RPC 实现结构的各个组件组成部分，下面我们详细说明下每个组件的职责划分。

1. RpcServer

负责导出（export）远程接口

2. RpcClient

负责导入（import）远程接口的代理实现

3. RpcProxy

远程接口的代理实现

4. RpcInvoker

客户方实现：负责编码调用信息和发送调用请求到服务方并等待调用结果返回

服务方实现：负责调用服务端接口的具体实现并返回调用结果

5. RpcProtocol

负责协议编/解码

6. RpcConnector

负责维持客户方和服务方的连接通道和发送数据到服务方

7. RpcAcceptor

负责接收客户方请求并返回请求结果

8. RpcProcessor

负责在服务方控制调用过程，包括管理调用线程池、超时时间等

9. RpcChannel

数据传输通道

协议编解码

客户端代理在发起调用前需要对调用信息进行编码，这就要考虑需要编码些什么信息并以什么格式传输到服务端才能让服务端完成调用。出于效率考虑，编码的信息越少越好（传输数据少），编码的规则越简单越好（执行效率高）。我们先看下需要编码些什么信息：

-- 调用编码 --

1. 接口方法

包括接口名、方法名

2. 方法参数

包括参数类型、参数值

3. 调用属性

包括调用属性信息，例如调用附件隐式参数、调用超时时间等

-- 返回编码 --

1. 返回结果

接口方法中定义的返回值

2. 返回码

异常返回码

3. 返回异常信息

调用异常信息

传输服务

协议编码之后，自然就是需要将编码后的 RPC 请求消息传输到服务方，服务方执行后返回结果消息或确认消息给客户方。RPC 的应用场景实质是一种可靠的请求应答消息流，和 HTTP 类似。因此选择长连接方式的 TCP 协议会更高效，与 HTTP 不同的是在协议层面我们定义了每个消息的唯一 id，因此可以更容易的复用连接。

既然使用长连接，那么第一个问题是到底 client 和 server 之间需要多少根连接？实际上单连接和多连接在使用上没有区别，对于数据传输量较小的应用类型，单连接基本足够。单连接和多连接最大的区别在于，每根连接都有自己私有的发送和接收缓冲区，因此大数据量传输时分散在不同的连接缓冲区会得到更好的吞吐效率。所以，如果你的数据传输量不足以让单连接的缓冲区一直处于饱和状态的话，那么使用多连接并不会产生任何明显的提升，反而会增加连接管理的开销。

连接是由 client 端发起建立并维持。如果 client 和 server 之间是直连的，那么连接一般不会中断（当然物理链路故障除外）。如果 client 和 server 连接经过一些负载中转设备，有可能连接一段时间不活跃时会被这些中间设备中断。为了保持连接有必要定时为每个连接发送心跳数据以维持连接不中断。心跳消息是 RPC 框架库使用的内部消息，在前文协议头结构中也有一个专门的心跳位，就是用来标记心跳消息的，它对业务应用透明。

执行调用

client stub 所做的事情仅仅是编码消息并传输给服务方，而真正调用过程发生在服务方。server stub 从前文的结构拆解中我们细分了 RpcProcessor 和 RpcInvoker 两个组件，一个负责控制调用过程，一个负责真正调用。这里我们还是以 java 中实现这两个组件为例来分析下它们到底需要做什么？

java 中实现代码的动态接口调用目前一般通过反射调用。除了原生的 jdk 自带的反射，一些第三方库也提供了性能更优的反射调用，因此 RpcInvoker 就是封装了反射调用的实现细节。

调用过程的控制需要考虑哪些因素，RpcProcessor 需要提供什么样地调用控制服务呢？下面提出几点以启发思考：

1. 效率提升

每个请求应该尽快被执行，因此我们不能每请求来再创建线程去执行，需要提供线程池服务。

2. 资源隔离

当我们导出多个远程接口时，如何避免单一接口调用占据所有线程资源，而引发其他接口执行阻塞。

3. 超时控制

当某个接口执行缓慢，而 client 端已经超时放弃等待后，server 端的线程继续执行此时显得毫无意义。

RPC 异常处理

无论 RPC 怎样努力把远程调用伪装的像本地调用，但它们依然有很大的不同点，而且有一些异常情况是在本地调用时绝对不会碰到的。在说异常处理之前，我们先比较下本地调用和 RPC 调用的一些差异：

1. 本地调用一定会执行，而远程调用则不一定，调用消息可能因为网络原因并未发送到服务方。

2. 本地调用只会抛出接口声明的异常，而远程调用还会跑出 RPC 框架运行时的其他异常。

3. 本地调用和远程调用的性能可能差距很大，这取决于 RPC 固有消耗所占的比重。

正是这些区别决定了使用 RPC 时需要更多考量。当调用远程接口抛出异常时，异常可能是一个业务异常，也可能是 RPC 框架抛出的运行时异常（如：网络中断等）。业务异常表明服务方已经执行了调用，可能因为某些原因导致未能正常执行，而 RPC 运行时异常则有可能服务方根本没有执行，对调用方而言的异常处理策略自然需要区分。

由于 RPC 固有的消耗相对本地调用高出几个数量级，本地调用的固有消耗是纳秒级，而 RPC 的固有消耗是在毫秒级。那么对于过于轻量的计算任务就并不合适导出远程接口由独立的进程提供服务，只有花在计算任务上时间远远高于 RPC 的固有消耗才值得导出为远程接口提供服务。