参考：<http://blog.csdn.net/mindfloating/article/details/39473807>

近几年的项目中，服务化和[**微服务**](http://lib.csdn.net/base/microservice)化渐渐成为中大型分布式系统[**架构**](http://lib.csdn.net/base/architecture)的主流方式，而 RPC 在其中扮演着关键的作用。在平时的日常开发中我们都在隐式或显式的使用 RPC，一些刚入行的程序员会感觉 RPC 比较神秘，而一些有多年使用 RPC 经验的程序员虽然使用经验丰富，但有些对其原理也不甚了了。缺乏对原理层面的理解，往往也会造成开发中的一些误用。

本文分上下两篇《浅出篇》和《深入篇》，其目标就是想尝试深入浅出的分析下 RPC 本质，我总是这么认为理解了本质才能更好的应用。

**RPC 是什么？**

RPC 的全称是 Remote Procedure Call 是一种进程间通信方式。它允许程序调用另一个地址空间（通常是共享网络的另一台机器上）的过程或函数，而不用程序员显式编码这个远程调用的细节。即程序员无论是调用本地的还是远程的，本质上编写的调用代码基本相同。

**RPC 起源**

RPC 这个概念术语在上世纪 80 年代由 [Bruce Jay Nelson](https://en.wikipedia.org/wiki/Bruce_Jay_Nelson) 提出。这里我们追溯下当初开发 RPC 的原动机是什么？在 Nelson 的论文 ["Implementing Remote Procedure Calls"](http://birrell.org/andrew/papers/ImplementingRPC.pdf) 中他提到了几点：

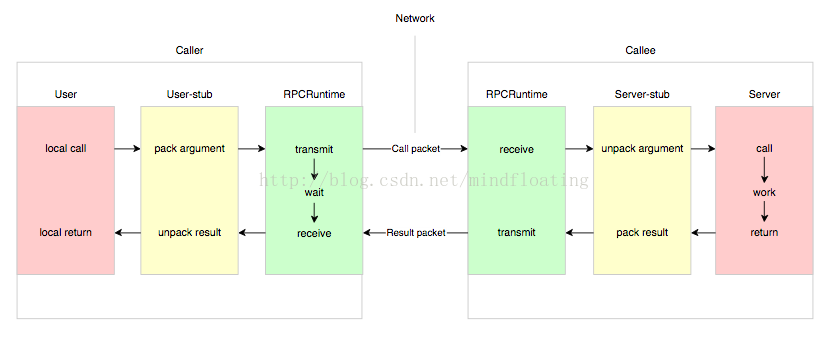
1. 简单：RPC 概念的语义十分清晰和简单，这样建立分布式计算就更容易。  
2. 高效：过程调用看起来十分简单而且高效。  
3. 通用：在单机计算中过程往往是不同算法部分间最重要的通信机制。

通俗一点说，就是一般程序员对于本地的过程调用很熟悉，那么我们把 RPC 作成和本地调用完全类似，那么就更容易被接受，使用起来毫无障碍。Nelson 的论文发表于 30 年前，其观点今天看来确实高瞻远瞩，今天我们使用的 RPC 框架基本就是按这个目标来实现的。

**RPC 结构**

Nelson 的论文中指出实现 RPC 的程序包括 5 个部分：

1. User  
2. User-stub  
3. RPCRuntime  
4. Server-stub  
5. Server

这 5 个部分的关系如下图所示  
  
这里 user 就是 client 端，当 user 想发起一个远程调用时，它实际是通过本地调用 user-stub。user-stub 负责将调用的接口、方法和参数通过约定的协议规范进行编码并通过本地的 RPCRuntime 实例传输到远端的实例。远端 RPCRuntime 实例收到请求后交给 server-stub 进行解码后发起本地端调用，调用结果再返回给 user 端。

**RPC 实现**

Nelson 论文中给出的这个实现结构也成为后来大家参考的标准范本。大约 10 年前，我最早接触分布式计算时使用的 [CORBAR](https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture) 实现结构基本与此类似。CORBAR 为了解决异构平台的 RPC，使用了 IDL（Interface Definition Language）来定义远程接口，并将其映射到特定的平台语言中。后来大部分的跨语言平台 RPC 基本都采用了此类方式，比如我们熟悉的 Web Service（SOAP），近年开源的 Thrift 等。他们大部分都通过 IDL 定义，并提供工具来映射生成不同语言平台的 user-stub 和 server-stub，并通过框架库来提供 RPCRuntime 的支持。不过貌似每个不同的 RPC 框架都定义了各自不同的 IDL 格式，导致程序员的学习成本进一步上升（苦逼啊），Web Service 尝试建立业界标准，无赖标准规范复杂而效率偏低，否则 Thrift 等更高效的 RPC 框架就没必要出现了。

IDL 是为了跨平台语言实现 RPC 不得已的选择，要解决更广泛的问题自然导致了更复杂的方案。而对于同一平台内的 RPC 而言显然没必要搞个中间语言出来，例如 [**Java**](http://lib.csdn.net/base/java) 原生的 RMI，这样对于 java 程序员而言显得更直接简单，降低使用的学习成本。目前市面上提供的 RPC 框架已经可算是五花八门，百家争鸣了。需要根据实际使用场景谨慎选型，需要考虑的选型因素我觉得至少包括下面几点：  
  
1. 性能指标  
2. 是否需要跨语言平台  
3. 内网开放还是公网开放  
4. 开源 RPC 框架本身的质量、社区活跃度

**总结**

《浅出篇》大概就到这里结束了，《深入篇》会具体深入讲解一个 RPC 框架需要实现哪里基本功能，达到什么目标，并以在 java 平台上去具体实现一个 RPC 框架为例，分析其需要考虑的实现因素。