RPC 的概念模型与实现解析

mindwind 2016-05-22 阅读 7 分钟

今天分布式应用、云计算、微服务大行其道，作为其技术基石之一的 RPC 你了解多少？一篇 RPC 的技术总结文章，数了下 5k+ 字，略长，可能也不适合休闲的碎片化时间阅读，可以先收藏抽空再细读:)

全文目录如下：

两年前写过两篇关于 RPC 的文章，如今回顾发现结构和逻辑略显凌乱，特作整理重新整合成一篇，想了解 RPC 原理的同学可以看看。

近几年的项目中，服务化和微服务化渐渐成为中大型分布式系统架构的主流方式，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。

定义

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函数，本质上编写的调用代码基本相同。

起源

RPC 这个概念术语在上世纪 80 年代由 Bruce Jay Nelson（参考[1]）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_（参考[2]） 中他提到了几点：

> 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。

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 做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。

目标

RPC 的主要目标是让构建分布式计算（应用）更容易，在提供强大的远程调用能力时不损失本地调用的语义简洁性。 为实现该目标，RPC 框架需提供一种透明调用机制让使用者不必显式的区分本地调用和远程调用。

分类

RPC 调用分以下两种：

1. __同步调用__：客户端等待调用执行完成并获取到执行结果。
2. __异步调用__：客户端调用后不用等待执行结果返回，但依然可以通过回调通知等方式获取返回结果。若客户端不关心调用返回结果，则变成单向异步调用，单向调用不用返回结果。

异步和同步的区分在于是否等待服务端执行完成并返回结果。

结构

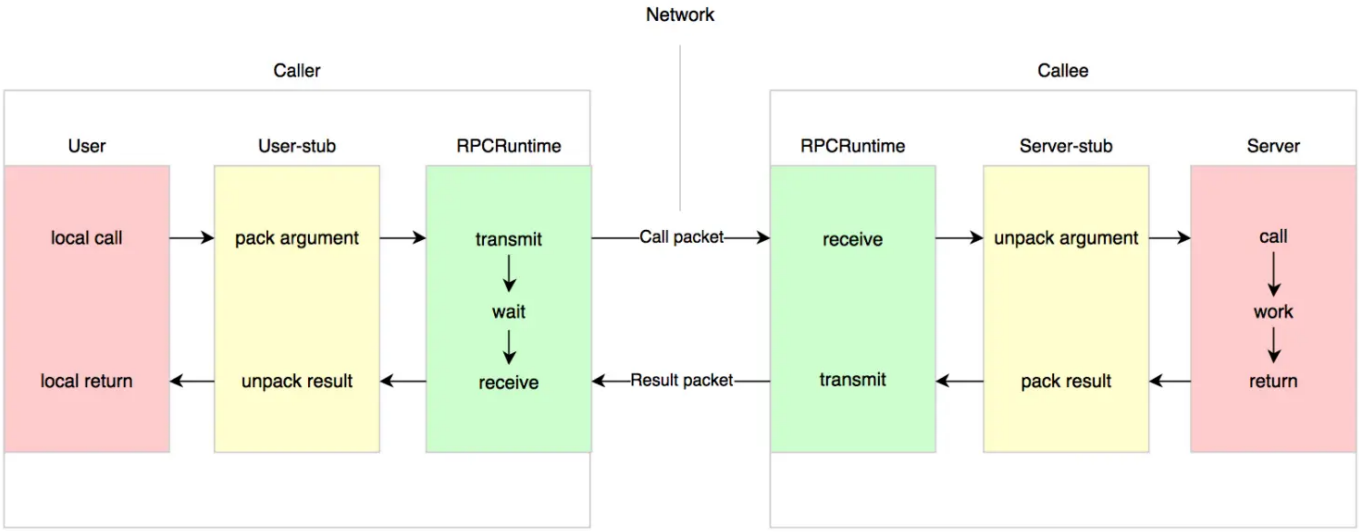
下面我们对 RPC 的结构从理论模型到真实组件一步步抽丝剥茧。

模型

最早在 Nelson 的论文中指出实现 RPC 的程序包括 5 个理论模型部分：

> User
> User-stub
> RPCRuntime
> Server-stub
> Server

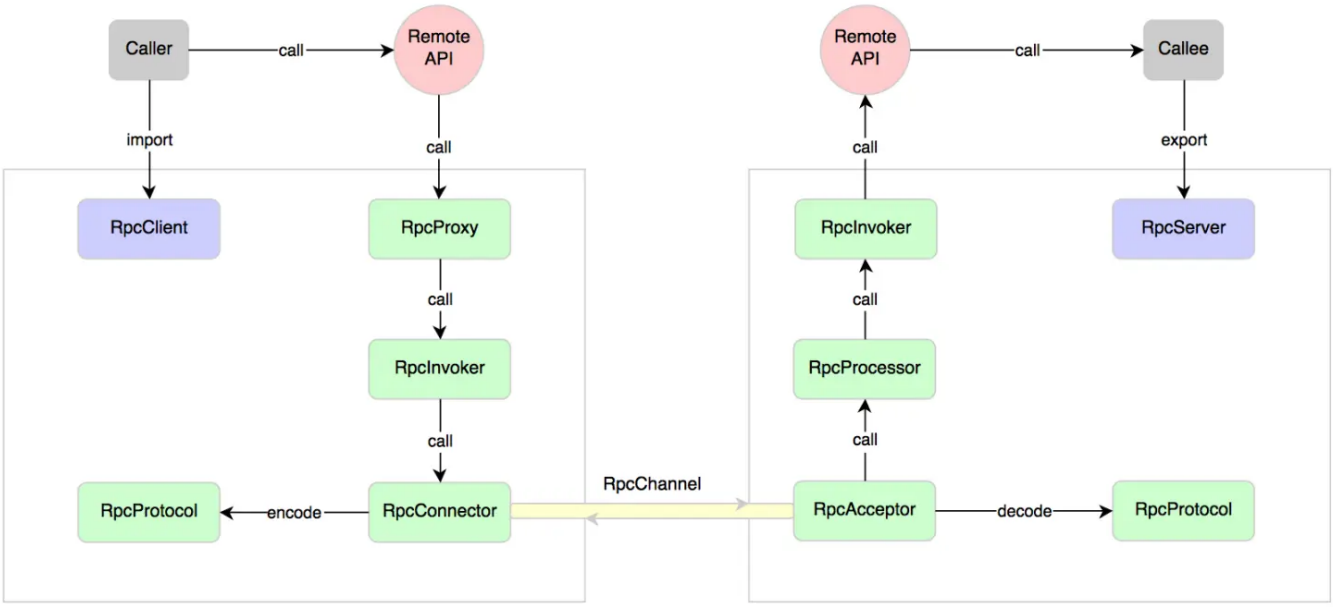
这 5 个部分的关系如下图所示：



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 进行解码后发起向本地端 Server 的调用，调用结果再返回给 User 端。

拆解

上面给出了一个比较粗粒度的 RPC 实现理论模型概念结构，这里我们进一步细化它应该由哪些组件构成，如下图所示。



RPC 服务端通过 `RpcServer` 去导出（export）远程接口方法，而客户端通过 `RpcClient` 去导入（import）远程接口方法。客户端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

`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端。

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。

组件

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**
   客户端：负责编码调用信息和发送调用请求到服务端并等待调用结果返回
   服务端：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**
   负责维持客户端和服务端的连接通道和发送数据到服务端
7. **RpcAcceptor**
   负责接收客户端请求并返回请求结果
8. **RpcProcessor**
   负责在服务端控制调用过程，包括管理调用线程池、超时时间等
9. **RpcChannel**
   数据传输通道

实现

Nelson 论文中给出的这个概念模型也成为后来大家参考的标准范本。十多年前，我最早接触分布式计算时使用的 CORBAR（参考[3]）实现结构基本与此基本类似。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

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。而 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 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。

在上文进一步拆解了组件并划分了职责之后，下面就以在 Java 平台实现该 RPC 框架概念模型为例，详细分析下实现中需要考虑的因素。

导出

导出是指暴露远程接口的意思，只有导出的接口可以供远程调用，而未导出的接口则不能。 在 Java 中导出接口的代码片段可能如下：

```
DemoService demo   = new ...;
RpcServer   server = new ...;
server.export(DemoService.class, demo, options);
```

我们可以导出整个接口，也可以更细粒度一点只导出接口中的某些方法，如下：

```
// 只导出 DemoService 中签名为 hi(String s) 的方法
server.export(DemoService.class, demo, "hi", new Class<?>[] { String.class }, options);
```

Java 中还有一种比较特殊的调用就是多态，也就是一个接口可能有多个实现，那么远程调用时到底调用哪个？这个本地调用的语义是通过 JVM 提供的引用多态性隐式实现的，那么对于 RPC 来说跨进程的调用就没法隐式实现了。如果前面 DemoService 接口有 2 个实现，那么在导出接口时就需要特殊标记不同的实现，如下：

```
DemoService demo   = new ...;
DemoService demo2  = new ...;
RpcServer   server = new ...;
server.export(DemoService.class, demo, options);
server.export("demo2", DemoService.class, demo2, options);
```

上面 demo2 是另一个实现，我们标记为 demo2 来导出，
那么远程调用时也需要传递该标记才能调用到正确的实现类，这样就解决了多态调用的语义。

导入

导入相对于导出而言，客户端代码为了能够发起调用必须要获得远程接口的方法或过程定义。目前，大部分跨语言平台 RPC 框架采用根据 IDL 定义通过 code generator 去生成 User-stub 代码，这种方式下实际导入的过程就是通过代码生成器在编译期完成的。我所使用过的一些跨语言平台 RPC 框架如 CORBAR、WebService、ICE、Thrift 均是此类方式。

代码生成的方式对跨语言平台 RPC 框架而言是必然的选择，而对于同一语言平台的 RPC 则可以通过共享接口定义来实现。
在 Java 中导入接口的代码片段可能如下：

```
RpcClient client = new ...;
DemoService demo = client.refer(DemoService.class);
demo.hi("how are you?");
```

在 Java 中 `import` 是关键字，所以代码片段中我们用 refer 来表达导入接口的意思。 这里的导入方式本质也是一种代码生成技术，只不过是在运行时生成，比静态编译期的代码生成看起来更简洁些。Java 里至少提供了两种技术来提供动态代码生成，一种是 JDK 动态代理，另外一种是字节码生成。 动态代理相比字节码生成使用起来更方便，但动态代理方式在性能上是要逊色于直接的字节码生成的，而字节码生成在代码可读性上要差很多。两者权衡起来，作为一种底层通用框架，个人更倾向于选择性能优先。

协议

协议指 RPC 调用在网络传输中约定的数据封装方式，包括三个部分：**编解码**、**消息头** 和 **消息体**。

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

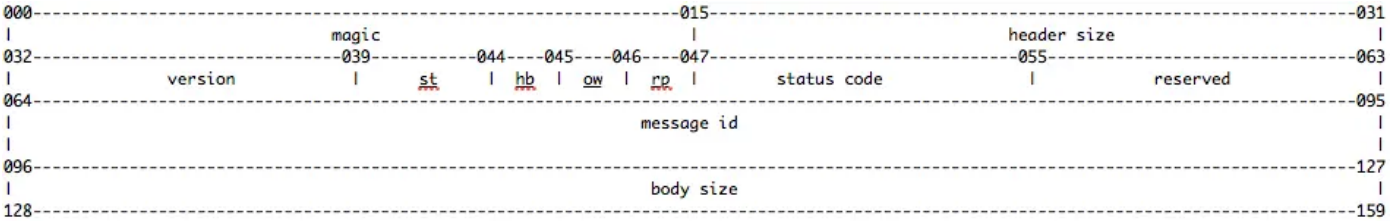
2. **方法参数**
   包括参数类型、参数值
3. **调用属性**
   包括调用属性信息，例如调用附加的隐式参数、调用超时时间等

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

消息头

除了以上这些必须的调用信息，我们可能还需要一些元信息以方便程序编解码以及未来可能的扩展。这样我们的编码消息里面就分成了两部分，一部分是元信息、另一部分是调用的必要信息。如果设计一种 RPC 协议消息的话，元信息我们把它放在协议消息头中，而必要信息放在协议消息体中。下面给出一种概念上的 RPC 协议消息头设计格式：



- `magic`
  协议魔数，为解码设计
- `header size`
  协议头长度，为扩展设计
- `version`
  协议版本，为兼容设计
- `st`
  消息体序列化类型
- `hb`
  心跳消息标记，为长连接传输层心跳设计
- `ow`
  单向消息标记，
- `rp`
  响应消息标记，不置位默认是请求消息

- `status code`
  响应消息状态码
- `reserved`
  为字节对齐保留
- `message id`
  消息 id
- `body size`
  消息体长度

消息体

消息体常采用序列化编码，常见有以下序列化方式：

- `xml`
  如 webservie SOAP
- `json`
  如 JSON-RPC
- `binary`
  如 thrift; hession; kryo 等

格式确定后编解码就简单了，由于头长度一定所以我们比较关心的就是消息体的序列化方式。 序列化我们关心三个方面：

1. __效率__：序列化和反序列化的效率，越快越好。
2. __长度__：序列化后的字节长度，越小越好。
3. __兼容__：序列化和反序列化的兼容性，接口参数对象若增加了字段，是否兼容。

上面这三点有时是鱼与熊掌不可兼得，这里面涉及到具体的序列化库实现细节，就不在本文进一步展开分析了。

传输

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端，服务方执行后返回结果消息或确认消息给客户端。RPC 的应用场景实质是一种可靠的请求应答消息流，这点和 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客户端和服务端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

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。

连接是由客户端发起建立并维持的，如果客户端和服务端之间是直连的，那么连接一般不会中断（当然物理链路故障除外）。如果客户端和服务端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

客户端 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端。服务端 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 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，而客户端已经超时放弃等待后，服务端的线程继续执行此时显得毫无意义。

异常

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适合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的时间远远高于 RPC 的固有消耗才值得导出为远程接口提供服务。

总结

至此我们提出了一个 RPC 实现的概念框架，并详细分析了需要考虑的一些实现细节。无论 RPC 的概念是如何优雅，但是“草丛中依然有几条蛇隐藏着”，只有深刻理解了 RPC 的本质，才能更好地应用。

看到这里的同学也许会想按这个概念模型和实现解析真得能开发实现一个 RPC 框架库么？这个问题我能肯定的回答，真得可以。因为我就按这个模型开发实现了一个最小化的 RPC 框架库来学习验证，相关的代码放在 Github 上，感兴趣的同学可以自己去阅读。这是我自己的一个实验性质的学习验证用开源项目，地址是 https://github.com/mindwind/craft-atom，其中的 `craft-atom-rpc` 即是按这个模型实现的微型 RPC 框架库，代码量相对工业级使用的 RPC 框架库少的多，方便阅读学习。

最后，读到这里的肯定都是好学不倦的同学，谢谢大家的时间，让我写作的意义更多了一点:)。

参考

1] Bruce Jay Nelson. [Bruce Jay Nelson
2] BIRRELL, NELSON. [Implementing Remote Procedure Calls. 1983
3] CORBAR. [CORBAR
4] DUBBO. [DUBBO