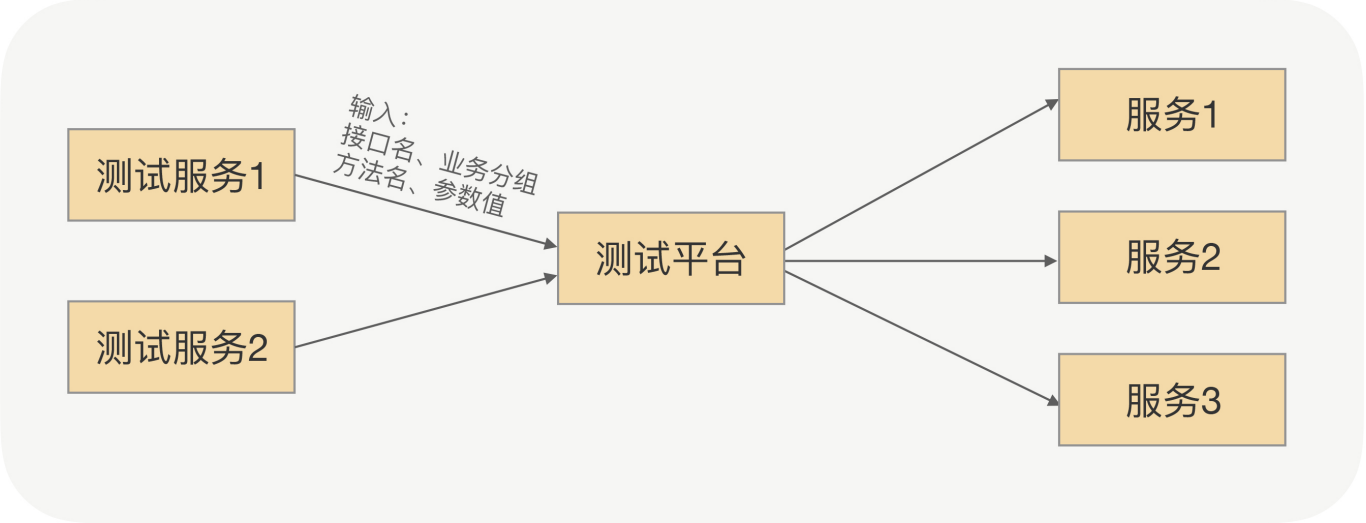
23 | 如何在没有接口的情况下进行RPC调用？

你好，我是何小锋。上一讲我们学习了 RPC 如何通过动态分组来实现秒级扩缩容，其关键点就是“动态”与“隔离”。今天我们来聊聊如何在没有接口的情况下进行 RPC 调用。

应用场景有哪些？

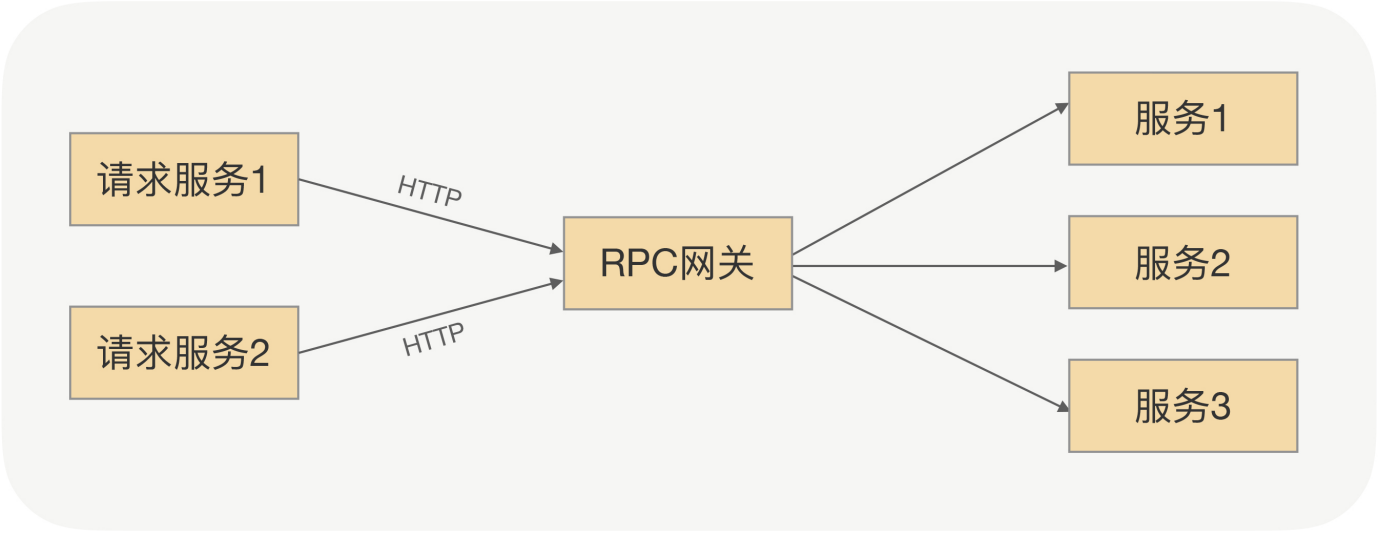
在 RPC 运营的过程中，让调用端在没有接口 API 的情况下发起 RPC 调用的需求，不只是一个业务方和我提过，这里我列举两个非常典型的场景例子。

场景一：我们要搭建一个统一的测试平台，可以让各个业务方在测试平台中通过输入接口、分组名、方法名以及参数值，在线测试自己发布的 RPC 服务。这时我们就有一个问题要解决，我们搭建统一的测试平台实际上是作为各个 RPC 服务的调用端，而在 RPC 框架的使用中，调用端是需要依赖服务提供方提供的接口 API 的，而统一测试平台不可能依赖所有服务提供方的接口 API。我们不能因为每有一个新的服务发布，就去修改平台的代码以及重新上线。这时我们就需要让调用端在没有服务提供方提供接口的情况下，仍然可以正常地发起 RPC 调用。



示意图

场景二：我们要搭建一个轻量级的服务网关，可以让各个业务方用 HTTP 的方式，通过服务网关调用其它服务。这时就有与场景一相同的问题，服务网关要作为所有 RPC 服务的调用端，是不能依赖所有服务提供方的接口 API 的，也需要调用端在没有服务提供方提供接口的情况下，仍然可以正常地发起 RPC 调用。



示意图

这两个场景都是我们经常会碰到的，而让调用端在没有服务提供方提供接口 API 的情况下仍然可以发起 RPC 调用的功能，在 RPC 框架中也是非常有价值的。

怎么做？

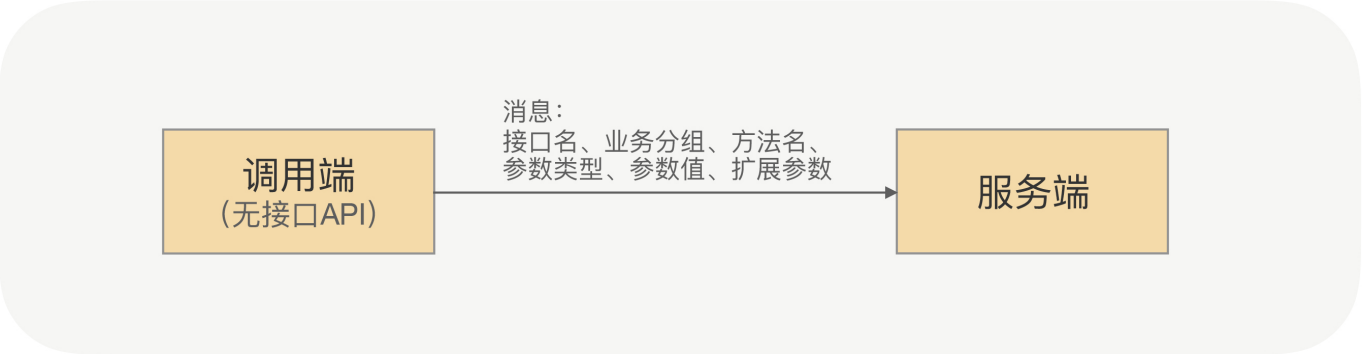
RPC 框架要实现这个功能，我们可以使用泛化调用。那什么是泛化调用呢？我们带着这个问题，先学习下如何在没有接口的情况下进行 RPC 调用。

我们先回想下我在基础篇讲过的内容，通过前面的学习我们了解到，在 RPC 调用的过程中，调用端向服务端发起请求，首先要通过动态代理，正如[第 05 讲] 中我说过的，动态代理可以帮助我们屏蔽 RPC 处理流程，真正地让我们发起远程调用就像调用本地一样。

那么在 RPC 调用的过程中，既然调用端是通过动态代理向服务端发起远程调用的，那么在调用端的程序中就一定要依赖服务提供方提供的接口 API，因为调用端是通过这个接口 API 自动生成动态代理的。那如果没有接口 API 呢？我们该如何让调用端仍然能够发起 RPC 调用呢？

所谓的 RPC 调用，本质上就是调用端向服务端发送一条请求消息，服务端接收并处理，之后向调用端发送一条响应消息，调用端处理完响应消息之后，一次 RPC 调用就完成了。那是不是说我们只要能够让调用端在没有服务提供方提供接口的情况下，仍然能够向服务端发送正确的请求消息，就能够解决这个问题了呢？

没错，只要调用端将服务端需要知道的信息，如接口名、业务分组名、方法名以及参数信息等封装成请求消息发送给服务端，服务端就能够解析并处理这条请求消息，这样问题就解决了。过程如下图所示：



示意图

现在我们已经清楚了解决问题的关键，但 RPC 的调用端向服务端发送消息是需要以动态代理作为入口的，我们现在得继续想办法让调用端发送我刚才讲过的那条请求消息。

我们可以定义一个统一的接口（GenericService），调用端在创建 GenericService 代理时指定真正需要调用的接口的接口名以及分组名，而 GenericService 接口的 $invoke 方法的入参就是方法名以及参数信息。

这样我们传递给服务端所需要的所有信息，包括接口名、业务分组名、方法名以及参数信息等都可以通过调用 GenericService 代理的 $invoke 方法来传递。具体的接口定义如下：

复制代码

```
class GenericService {

 Object $invoke(String methodName, String[] paramTypes, Object[] params);

}
```

这个通过统一的 GenericService 接口类生成的动态代理，来实现在没有接口的情况下进行 RPC 调用的功能，我们就称之为泛化调用。

通过泛化调用功能，我们可以解决在没有服务提供方提供接口 API 的情况下进行 RPC 调用，那么这个功能是否就完美了呢？

回顾下[第 17 讲] 我过的内容，RPC 框架可以通过异步的方式提升吞吐量，还有如何实现全异步的 RPC 框架，其关键点就是 RPC 框架对 CompletableFuture 的支持，那么我们的泛化调用是否也可以支持异步呢？

当然可以。我们可以给 GenericService 接口再添加一个异步方法 $asyncInvoke，方法的返回值就是 CompletableFuture，GenericService 接口的具体定义如下：

复制代码

```
class GenericService {

 Object $invoke(String methodName, String[] paramTypes, Object[] params);

  CompletableFuture<Object> $asyncInvoke(String methodName, String[] paramTypes, Object[] params);

}
```

学到这里相信你已经对泛化调用的功能有一定的了解了，那你有没有想过这样一个问题？在没有服务提供方提供接口 API 的情况下，我们可以用泛化调用的方式实现 RPC 调用，但是如果没有服务提供方提供接口 API，我们就没法得到入参以及返回值的 Class 类，也就不能对入参对象进行正常的序列化。这时我们会面临两个问题：

问题 1：调用端不能对入参对象进行正常的序列化，那调用端、服务端在接收到请求消息后，入参对象又该如何序列化与反序列化呢？

回想下[第 07 讲]，在这一讲中我讲解了如何设计可扩展的 RPC 框架，我们通过插件体系来提高 RPC 框架的可扩展性，在 RPC 框架的整体架构中就包括了序列化插件，我们可以为泛化调用提供专属的序列化插件，通过这个插件，解决泛化调用中的序列化与反序列化问题。

问题 2：调用端的入参对象（params）与返回值应该是什么类型呢？

在服务提供方提供的接口 API 中，被调用的方法的入参类型是一个对象，那么使用泛化调用功能的调用端，可以使用 Map 类型的对象，之后通过泛化调用专属的序列化方式对这个 Map 对象进行序列化，服务端收到消息后，再通过泛化调用专属的序列化方式将其反序列成对象。

总结

今天我们主要讲解了如何在没有接口的情况下进行 RPC 调用，泛化调用的功能可以实现这一目的。

这个功能的实现原理，就是 RPC 框架提供统一的泛化调用接口（GenericService），调用端在创建 GenericService 代理时指定真正需要调用的接口的接口名以及分组名，通过调用 GenericService 代理的 $invoke 方法将服务端所需要的所有信息，包括接口名、业务分组名、方法名以及参数信息等封装成请求消息，发送给服务端，实现在没有接口的情况下进行 RPC 调用的功能。

而通过泛化调用的方式发起调用，由于调用端没有服务端提供方提供的接口 API，不能正常地进行序列化与反序列化，我们可以为泛化调用提供专属的序列化插件，来解决实际问题。

课后思考

在讲解泛化调用时，我讲到服务端在收到调用端通过泛化调用的方式发送过来的请求时，会使用泛化调用专属的序列化插件实现对其进行反序列化，那么服务端是如何判定这个请求消息是通过泛化调用的方式发送过来的消息呢？

欢迎留言和我分享你的答案，也欢迎你把文章分享给你的朋友，邀请他加入学习。我们下节课再见！

上一篇 22 | 动态分组：超高效实现秒级扩缩容

24 | 如何在线上环境里兼容多种RPC协议？

你好，我是何小锋。上一讲我们学习了如何在没有接口的情况下完成 RPC 调用，其关键在于你要理解接口定义在 RPC 里面的作用。除了我们前面说的，动态代理生成的过程中需要用到接口定义，剩余的其它过程中接口的定义只是被当作元数据来使用，而动态代理在 RPC 中并不是一个必须的环节，所以在没有接口定义的情况下我们同样也是可以完成 RPC 调用的。

回顾完上一讲的重点，咱们就言归正传，切入今天的主题，一起看看如何在线上环境里兼容多种 RPC 协议。

看到这个问题后，可能你的第一反应就是，在真实环境中为什么会存在多个协议呢？我们说过，RPC 是能够帮助我们屏蔽网络编程细节，实现调用远程方法就跟调用本地一样的体验。大白话说就是，RPC 是能够帮助我们在开发过程中完成应用之间的通信，而又不需要我们关心具体通信细节的工具。

为什么要支持多协议？

既然应用之间的通信都是通过 RPC 来完成的，而能够完成 RPC 通信的工具有很多，比如像 Web Service、Hessian、gRPC 等都可以用来充当 RPC 使用。这些不同的 RPC 框架都是随着互联网技术的发展而慢慢涌现出来的，而这些 RPC 框架可能在不同时期会被我们引入到不同的项目中解决当时应用之间的通信问题，这样就导致我们线上的生成环境中存在各种各样的 RPC 框架。

很显然，这种混乱使用 RPC 框架的方式肯定不利于公司技术栈的管理，最明显的一个特点就是我们维护 RPC 框架的成本越来越高，因为每种 RPC 框架都需要有专人去负责升级维护。

为了解决早期遗留的一些技术负债，我们通常会去选择更高级的、更好用的工具来解决，治理 RPC 框架混乱的问题也是一样。为了解决同时维护多个 RPC 框架的困难，我们肯定希望能够用统一用一种 RPC 框架来替代线上所有的 RPC 框架，这样不仅能降低我们的维护成本，而且还可以让我们在一种 RPC 上面去精进。

既然目标明确后，我们该如何实施呢？

可能你会说这很简单啊，我们只要把所有的应用都改造成新 RPC 的使用方式，然后同时上线所有改造后的应用就可以了。如果在团队比较小的情况下，这种断崖式的更新可能确实是最快的方法，但如果是在团队比较大的情况下，要想做到同时上线所有改造后的应用，暂且不讨论这种方式是否存在风险，光从多个团队同一时间上线所有应用来看，这也几乎是一件不可能做到的事儿。

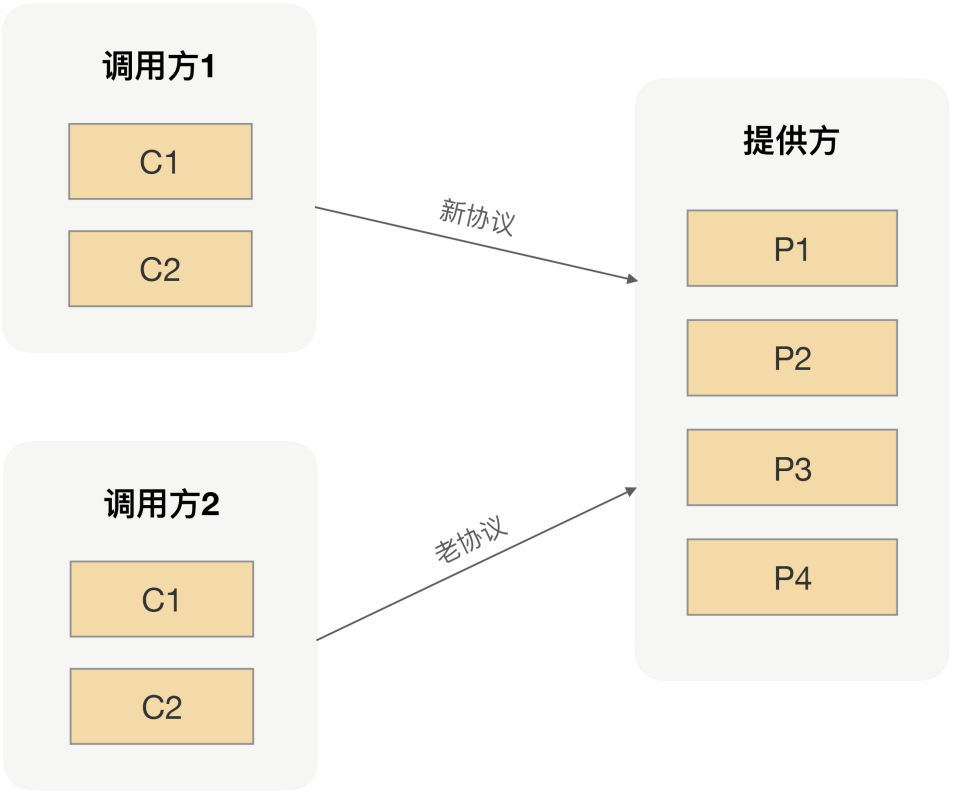
那对于多人团队来说，有什么办法可以让其把多个 RPC 框架统一到一个工具上呢？我们先看下多人团队在升级过程中所要面临的困难，人数多就意味着要维护的应用会比较多，应用多了之后线上应用之间的调用关系就会相对比较复杂。那这时候如果单纯地把任意一个应用目前使用的 RPC 框架换成新的 RPC 框架的话，就需要让所有调用这个应用的调用方去改成新的调用方式。

通过这种自下而上的滚动升级方式，最终是可以让所有的应用都切换到统一的 RPC 框架上，但是这种升级方式存在一定的局限性，首先要求我们能够清楚地梳理出各个应用之间的调用关系，只有这样，我们才能按部就班把所有应用都升级到新的 RPC 框架上；其次要求应用之间的关系不能存在互相调用的情况，最好的情况就是应用之间的调用关系像一颗树，有一定的层次关系。但实际上我们应用的调用关系可能已经变成了网状结构，这时候想再按照这种方式去推进升级的话，就可能寸步难行了。

为了解决上面升级过程中遇到的问题，你可能还会想到另外一个方案，那就是在应用升级的过程中，先不移除原有的 RPC 框架，但同时接入新的 RPC 框架，让两种 RPC 同时提供服务，然后等所有的应用都接入完新的 RPC 以后，再让所有的应用逐步接入到新的 RPC 上。这样既解决了上面存在的问题，同时也可以让所有的应用都能无序地升级到统一的 RPC 框架上。

在保持原有 RPC 使用方式不变的情况下，同时引入新的 RPC 框架的思路，是可以让所有的应用最终都能升级到我们想要升级的 RPC 上，但对于开发人员来说，这样切换成本还是有点儿高，整个过程最少需要两次上线才能彻底地把应用里面的旧 RPC 都切换成新 RPC。

那有没有更好的方式可以让应用上线一次就可以完成新老 RPC 的切换呢？关键就在于要让新的 RPC 能同时支持多种 RPC 调用，当一个调用方切换到新的 RPC 之后，调用方和服务提供方之间就可以用新的协议完成调用；当调用方还是用老的 RPC 进行调用的话，调用方和服务提供方之间就继续沿用老的协议完成调用。对于服务提供方来说，所要处理的请求关系如下图所示：



调用关系

怎么优雅处理多协议？

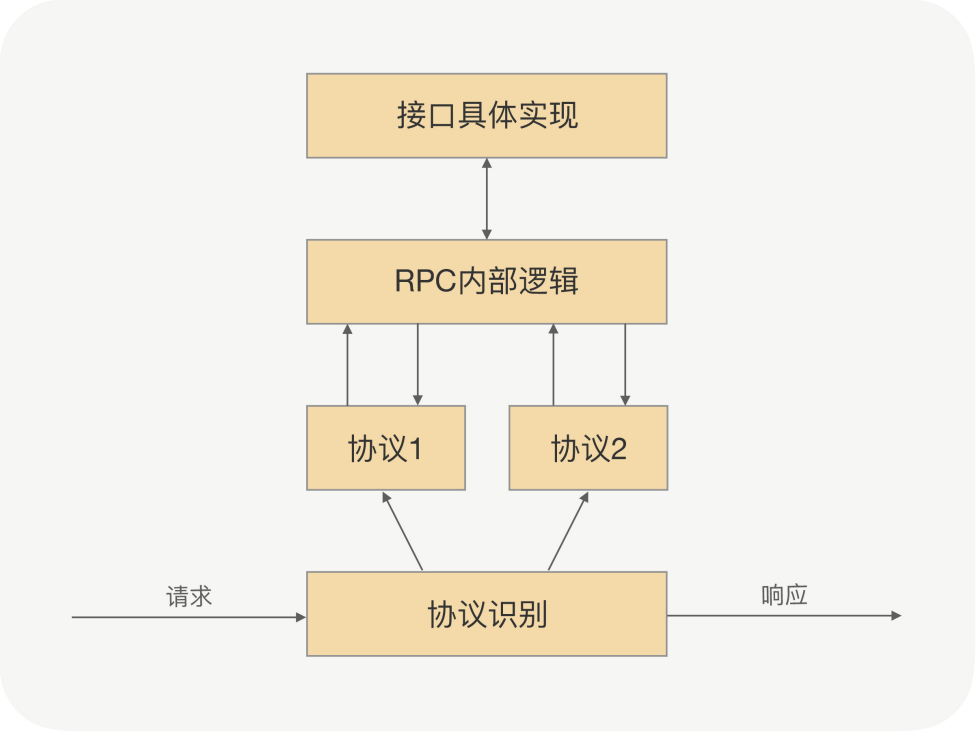
要让新的 RPC 同时支持多种 RPC 调用，关键就在于要让新的 RPC 能够原地支持多种协议的请求。怎么才能做到？在[第 02 讲] 我们说过，协议的作用就是用于分割二进制数据流。每种协议约定的数据包格式是不一样的，而且每种协议开头都有一个协议编码，我们一般叫做 magic number。

当 RPC 收到了数据包后，我们可以先解析出 magic number 来。获取到 magic number 后，我们就很容易地找到对应协议的数据格式，然后用对应协议的数据格式去解析收到的二进制数据包。

协议解析过程就是把一连串的二进制数据变成一个 RPC 内部对象，但这个对象一般是跟协议相关的，所以为了能让 RPC 内部处理起来更加方便，我们一般都会把这个协议相关的对象转成一个跟协议无关的 RPC 对象。这是因为在 RPC 流程中，当服务提供方收到反序列化后的请求的时候，我们需要根据当前请求的参数找到对应接口的实现类去完成真正的方法调用。如果这个请求参数是跟协议相关的话，那后续 RPC 的整个处理逻辑就会变得很复杂。

当完成了真正的方法调用以后，RPC 返回的也是一个跟协议无关的通用对象，所以在真正往调用方写回数据的时候，我们同样需要完成一个对象转换的逻辑，只不过这时候是把通用对象转成协议相关的对象。

在收发数据包的时候，我们通过两次转换实现 RPC 内部的处理逻辑跟协议无关，同时保证调用方收到的数据格式跟调用请求过来的数据格式是一样的。整个流程如下图所示：



多协议处理流程

总结

在我们日常开发的过程中，最难的环节不是从 0 到 1 完成一个新应用的开发，而是把一个老应用通过架构升级完成从 70 分到 80 分的跳跃。因为在老应用升级的过程中，我们不仅需要考虑既有的功能逻辑，也需要考虑切换到新架构上的成本，这就要求我们在设计新架构的时候要考虑如何让老应用能够平滑地升级，就像在 RPC 里面支持多协议一样。

在 RPC 里面支持多协议，不仅能让我们更从容地推进应用 RPC 的升级，还能为未来在 RPC 里面扩展新协议奠定一个良好的基础。所以我们平时在设计应用架构的时候，不仅要考虑应用自身功能的完整性，还需要考虑应用的可运维性，以及是否能平滑升级等一些软性能力。

课后思考

在 RPC 里面支持多协议的时候，有一个关键点就是能够识别出不同的协议，并且根据不同的 magic number 找到不同协议的解析逻辑。如果线上协议存在很多种的话，就需要我们事先在 RPC 里面内置各种协议，但通过枚举的方式可能会遗漏，不知道针对这种问题你有什么好的办法吗？

欢迎留言和我分享你的答案，也欢迎你把文章分享给你的朋友，邀请他加入学习。我们下节课再见！

上一篇 23 | 如何在没有接口的情况下进行RPC调用？

下一篇 特别放送 | 谈谈我所经历过的RPC

特别放送 | 谈谈我所经历过的RPC

你好，我是何小锋。上一讲我们学习了如何在线上环境里兼容多种 RPC 协议，目的就是为了能够平滑地升级线上环境中已经存在的 RPC 框架，同时我们也可以利用多协议的特点来支持不同的使用场景。

以上就是我们整个专栏关于技术内容的最后一讲了，很幸运能够和你一起携手并肩走过这些日子，这段时间我们跨了新年，也经历了让人猝不及防的新冠肺炎。现在我才发现原来能够自由地呼吸新鲜空气也是一种幸福，祝你平平安安度过这段艰难的日子，期待我们能早日摘下口罩。为了感谢你这些日子的陪伴，今天我们来换换脑子，聊一些轻松的话题。我就和你分享分享我所经历过的 RPC 框架吧。

与 RPC 结缘

我 1998 从大学毕业的时候，互联网并不是像今天这样如火如荼地进行着。那时候大部分 IT 公司都在耕耘数字化办公相关领域，所以大学毕业的时候我也就随了大流，进入了一个从事办公软件开发的公司，也就是我们今天经常说的“传统软件行业”。

后来随着互联网的快速普及，各个行业都想借着这个机会跟互联网发生点关系，因为大家都知道互联网代表着未来，在咱国内借助互联网技术发展最快的行业代表就是电商和游戏。

得益于互联网的普及，各个互联网公司有关技术的文章也是扑面而来，比如 A 公司因为流量激增而导致整站长时间瘫痪，B 公司因为订单持续上涨而导致数据库压力太大等等。每当我看到类似技术文章的时候，我都在想自己什么时候也可以遇到能用技术优化的方式来解决的问题呢。后来发现这些“设想”在我当时所在的行业里很难体会到，所以 2011 年我毅然选择了加入京东。

那时候我们公司刚好处于一个业务高速发展期，但系统稳定性相比今天来讲的话，还处于一个比较稚嫩的阶段。在日常工作中，我们的研发人员不仅要面对来自业务方需求进度的压力，还要面对因为线上业务增长而导致的系统压力。这样的双重压力对我来说很新鲜，这种体验是我之前从未有过的，这让我感到很兴奋。

之前我自己也做了很长一段时间的业务系统开发，虽然积累了不少实用的编程技巧，也认真阅读过国内外很多优秀的开源代码，包括像 Spring、Netty 等等这样优秀的框架，但因为从来没有经历过这么多系统之间复杂调用的场景，所以在入职后的一段时间里，我的内心一直处于忐忑状态。

得幸于同事的热情帮助，我很快地就融入了团队氛围。因为我们大团队是负责整个公司基础架构的，可以简单理解成我们就是负责公司 PASS 平台的建设，既然是 PASS 平台那肯定少不了中间件，而中间件里面最核心的应该就是 RPC 了，因为 RPC 相对其它中间件来说，使用频率是最高的，也是我们构建分布式系统的基石。

从此我就踏上了 RPC 这条“不归路”了，当然并不是因为这是条绝路，而是这条路一直充满着挑战，并没有让我觉得有停下来的想法。

使用过的 RPC

因为我工作年限比较长，所以编程经历相对也比较丰富，在转 Java 之前我做过一段时间的.Net。早期因为主要做数字化办公软件，所以那时候用.Net 编程的机会比较多，主要是因为很多应用采用.Net 相对使用 Java 来

说，开发效率会高很多。但后面应用复杂之后，.Net 在性能、可维护性上都不如 Java 有优势，而且.Net 在很长一段时间内并不支持在 Linux 环境下部署，所以我们就把应用都逐渐改成 Java 了。

ICE

这就存在一个问题，我们是怎么把.Net 应用平滑地切换到 Java 上呢？原本是.Net 应用之间的互相调用，需要改成.Net 跟 Java 之间互相通信。为了解决这个问题，我们当时选择了一个比较古老的 RPC 框架 ICE（https://zeroc.com），可能现在很多人都没有听说过了。

Hessian

后面使用 Java 开发应用的机会越来越多，而且随着 Spring 开发方式的大流行，在 Java 应用里面使用 ICE 来完成应用之间的 RPC 调用就变得比较鸡肋了。所以当时我们用了一种新的 RPC 框架 Hessian（http://hessian.caucho.com/），这时我们就把 Java 应用之间的 RPC 调用方式改成了 Hessian 方式。

用 Hessian 的原因就是因为它可以很好地跟 Spring 进行集成，对 Spring 项目的开发人员来说，开发一个 RPC 接口就变得很容易了，我们可以直接把 Java 类对外进行暴露，用作 RPC 接口的定义，而不需要像 ICE 一样先定义 Stub。

单纯的从 RPC 框架角度出发，Hessian 是一款很优秀的产品，即使放到今天，它的性能和鲁棒性都有着很强的参考意义。但当业务发展壮大到一定程度后，应用之间的调用就不仅仅需要考虑用什么 RPC 框架了，更多的是需要考虑怎么去完成服务治理。

另外一个原因就是因为 Hessian 是没有服务发现功能的，我们只能通过 VIP 暴露的方式完成调用，我们需要给每个应用分配一个 VIP，把同一接口的所有服务提供方实例挂载到同一个 VIP 上。这种集中式流量转发架构就会使得提供 VIP 服务的 LVS 存在很大的压力，而且集中式流量的转发会让调用方响应时间相对变长。

Dubbo

为了解决类似 Hessian 这种集中式问题，实现大规模应用服务化的落地，国内的 RPC 框架 Dubbo 的做法就显得比较先进了。随着业务越来越复杂，应用之间的调用关系也就变得更加错综复杂，所以后面我们也是选择基于 Dubbo 进行扩展，以完成 RPC 通信，而服务发现则通过接入 Zookeeper 集群来完成。通过这种现有框架的搭配，我们完成了应用服务化的快速落地，也同时完成了统一公司内部所有应用 RPC 框架的目标。

但随着微服务理念越来越流行，很多应用的接口也是越拆越细，导致我们 Zookeeper 集群需要接入的接口数量越来越多；还有就是因为我们每年的业务量是成倍增长，为了让应用能够抗足够的调用量应用，我们也需要经常扩容，从而导致 Zookeeper 集群接入的 IP 实例数也是呈数量级增长的，这使得我们的 Zookeeper 集群负荷特别重。

再有就是 Dubbo 相对有点复杂，而且性能还有提高空间，这使得我们不得不考虑新的方案。

自研 RPC

在这种背景下，我们决定自行研发一套适合自己业务场景的微服务解决方案，包括 RPC 框架、服务治理以及多语言解决方案。

至此我们自研的 RPC 就一直平稳地支持着公司内的各种业务。

这几年，在以 K8S 为代表的基础设施演进过程中，一个重要的关键词就是应用基础设施能力的下沉。在过去我们给应用提供 RPC 能力的时候，都是需要应用引入 Jar 包方式来解决的，在 RPC 里面，我们要把服务发现、路由等一整套 RPC 解决方案都融入到这个 Jar 里面去。

未来

目前，K8S 已成为基础设施的事实标准。而原先通过 Jar 包的方式封装的各种基础设施能力，现在全都被 K8S 项目从应用层拽到了基础设施中。那对于我们 RPC 来说也是一样，我们需要把非业务功能从传统的 RPC 框架中剥离出来，下沉到基础设施并且融入基础设施，然后通过 Mesh 去连接应用和基础设施。

这也是 RPC 发展的下一个阶段，完成所有应用的 Mesh 化。所以说，RPC 这条路没有尽头，只有不断的挑战和乐趣。希望你也能爱上它！

那最后我还想和你谈谈我对 RPC 的看法。

可能大家在谈论 RPC 时候，都想着 RPC 只是解决应用之间调用的工具。从本质上来讲，这没有什么问题，但在现实中，我们需要 RPC 解决更多的实际问题，比如服务治理，这些东西都是在使用 RPC 的过程中需要考虑的问题，所以我个人认为 RPC 应该是一个比较泛的概念。

当然，可能我们中大多数人现在是没有机会去完整实现一个新的 RPC 的，这不仅是精力的问题，更多是实际需求的问题，那为什么我们还需要学好 RPC 呢？我的想法很简单也非常实在，就是因为 RPC 是我们构建分布式系统的基石，就好比我们每次都是从“Hello World”开始学习一门新的编程语言。期待你能打牢这个基础，总有一天你会体验到它的能量！

今天的特别放送就到这里，也非常期待听到你和 RPC 的故事。

上一篇 24 | 如何在线上环境里兼容多种RPC协议？

下一篇 结束语 | 学会从优秀项目的源代码中挖掘知识

结束语 | 学会从优秀项目的源代码中挖掘知识

你好，我是何小锋。

今天是咱们专栏的最后一课。首先，我的读者，非常感谢你对这个专栏的支持。当你看到这儿，恭喜你“毕业”了，我也“毕业”了！从专栏筹备到结课，转眼已经过去了半年的时间。这段经历令我很难忘，可以说是痛苦与快乐并存吧，今天结课我还真想和你说道说道。

我其实不太善言辞，挺典型的一类技术人，做这件事的出发点还是对技术的热爱，我希望能把自己这些年的经验去沉淀一下，分享出去供更多的人参考，与更多的人探讨和交流，很开心我做到了，有一种成就感和满足感在。但写作的过程确实还挺难的，有点出乎我的意料，讲课和写技术文档的差别不小，我觉得这其中最大的区别就是讲课我需要去设计内容，怎么让内容“既浅又深”，让不同阶段的人都能“看得懂、学得会、用得着”，就是我的标准。

我们应该学会分享与交流，这是我在写作过程中最大的一个感触。

在写的过程中，我会不停地调动自己积累的知识和经验，每一节课，每一个问题，都需要我在脑子里一遍遍梳理好才能成文，对我自己的知识体系是一次很好的加固和升级的机会。当然，我也会碰到新的疑惑和问题，甚至是自我否定，但这不重要，重要的是我解决了它，我的知识面得到了拓展。还有就是，我会感觉到一种快乐，别人会因为我的分享而有收获，这是一种肯定。

当然，分享和兴趣虽是初衷，但我打的也是“有准备之仗”。在写这个专栏之前，我曾对我所运营的 RPC 框架进行了重定义，整体架构完全重新设计，代码也完全重写，这里面的主要原因还是在于旧版本的 RPC 框架由于一次次的代码迭代，已经有了太多的补丁，代码维护起来已经十分困难了，它很难再满足业务新增的需求。

在设计与编写 RPC 框架的过程中，我从业界的 RPC 框架中吸取经验，我先对自己的认知做了一次全面的升级。正如我在 [开篇词] 里说的，RPC 是解决分布式系统通信问题的一大利器，在我所接触的分布式系统中，基本都离不开 RPC。这一点令我自己都是兴奋的。

决定了重定义 RPC 框架，我就坚持对每一行代码，甚至是每一行注释进行 review，对代码的扩展性、可阅读性以及性能都尽量做到完美。但是在写专栏的时候，我并没有写到太多的代码，因为在我看来，在设计整体框架、实现每个功能、解决某一个难题时，我们首先要有一个或多个完整的并且可行的思路和想法，而编码的过程不过是将解决问题的思路和想法以代码的形式翻译出来。当然编码也需要很扎实的基本功，这一点我们不能否认。

那讲到这儿，我还是想再强调一下咱们专栏的重点，以免你迷路。我把重点放在了 RPC 框架的一些基础功能与高级功能的实现思路上，以及某一类问题的解决办法上。

从内容结构上来说，为了能让你更好地学习 RPC 框架，整个专栏的内容设计还是比较系统的，我们从 RPC 框架的基础功能，讲到集群、治理等相关的高级功能，再到 RPC 框架的性能优化以及问题排查等等，整个学习过程由浅入深。

希望看到最后一课的你，已经完全理解了我的用意和用心！

那最后，我还想给你一个小建议，那就是**你一定要学会从优秀项目的源代码中挖掘知识，结合自己的见解与经验，去解决一个又一个的难题，形成自己的知识体系**，而我前面所说的分享就是检验成果的一个好办法。

最后的最后，还是要感谢你，我的读者，多谢你的支持，你的鼓励和批评是我前进的最大动力。如果你有什么意见或建议，欢迎你通过结课问卷告知我，我会正视大家的声音。

我是何小锋，我们后会有期！

上一篇 特别放送 | 谈谈我所经历过的RPC

答疑课堂 | 基础篇与进阶篇思考题答案合集

你好，我是何小锋。到今天为止，基础篇和进阶篇我们就都学习完了，在这个过程中我一直在看大家的留言，知道你可能还有很多地方存在着疑问，今天这一讲我整理了一些关注度比较高的课后思考题答案，希望能给你带来帮助。

第二讲

思考题：在 RPC 里面，我们是怎么实现请求跟响应关联的？

首先我们要弄清楚为什么要把请求与响应关联。这是因为在 RPC 调用过程中，调用端会向服务端发送请求消息，之后它还会收到服务端发送回来的响应消息，但这两个操作并不是同步进行的。在高并发的情况下，调用端可能会在某一时刻向服务端连续发送很多条消息之后，才会陆续收到服务端发送回来的各个响应消息，这时调用端需要一种手段来区分这些响应消息分别对应的是之前的哪条请求消息，所以我们说 RPC 在发送消息时要请求跟响应关联。

解决这个问题不难，只要调用端在收到响应消息之后，从响应消息中读取到一个标识，告诉调用端，这是哪条请求消息的响应消息就可以了。在这一讲中，你会发现我们设计的私有协议都会有消息 ID，这个消息 ID 的作用就是起到请求跟响应关联的作用。调用端为每一个消息生成一个唯一的消息 ID，它收到服务端发送回来的响应消息如果是同一消息 ID，那么调用端就可以认为，这条响应消息是之前那条请求消息的响应消息。

第五讲

思考题：如果没有动态代理帮我们完成方法调用拦截，用户该怎么完成 RPC 调用？

这个问题我们可以参考下 gRPC 框架。gRPC 框架中就没有使用动态代理，它是通过代码生成的方式生成 Service 存根，当然这个 Service 存根起到的作用和 RPC 框架中的动态代理是一样的。

gRPC 框架用代码生成的 Service 存根来代替动态代理主要是为了实现多语言的客户端，因为有些语言是不支持动态代理的，比如 C++、go 等，但缺点也是显而易见的。如果你使用过 gRPC，你会发现这种代码生成 Service 存根的方式与动态代理相比还是很麻烦的，并不如动态代理的方式使用起来方便、透明。

第六讲

思考题：在 gRPC 调用的时候，我们有一个关键步骤就是把对象转成可传输的二进制，但是在 gRPC 里面，我们并没有直接转成二进制数组，而是返回一个 InputStream，你知道这样做的好处是什么吗？

RPC 调用在底层传输过程中也是需要使用 Stream 的，直接返回一个 InputStream 而不是二进制数组，可以避免数据的拷贝。

第八讲

思考题：目前服务提供者上线后会自动注册到注册中心，服务调用方会自动感知到新增的实例，并且流量会很快打到该新增的实例。如果我想把某些服务提供者实例的流量切走，除了下线实例，你有没有想到其它更便捷的办法呢？

解决这个问题的方法还是有很多的，比如留言中提到的改变服务提供者实例的权重，将权重调整为 0，或者通过路由的方式也可以。

但解决这个问题最便捷的方式还是使用动态分组，在[第 16 讲] 中我讲解了业务分组的概念，通过业务分组来实现流量隔离。如果业务分组是动态的，我们就可以在管理平台动态地自由调整，那是不是就可以实现动态地流量切换了呢？这个问题我们还会在高级篇中详解，期待一下。

第十二讲

思考题：在整个 RPC 调用的流程中，异常重试发生在哪个环节？

在回答这个问题之前，我们先回想下这一讲中讲过的内容。我在讲 RPC 为什么需要异常重试时我说过，如果在发出请求时恰好网络出现问题了，导致我们的请求失败，我们可能需要进行异常重试。从这一点我们可以看出，异常重试的操作是要在调用端进行的。因为如果在调用端发出请求时恰好网络出现问题导致请求失败，那么这个请求很可能还没到达服务端，服务端当然就没办法去处理重试了。

另外，我还讲过，我们需要在所有发起重试、负载均衡选择节点的时候，去掉重试之前出现过问题的那个节点，以保证重试的成功率。由此可见异常重试的操作应该发生在负载均衡之前，在发起重试的时候，会调用负载均衡插件来选择一个服务节点，在调用负载均衡插件时我们要告诉负载均衡需要刨除哪些有问题的服务节点。

在整个 RPC 调用的过程中，从动态代理到负载均衡之间还有一系列的操作，如果你研究过开源的 RPC 框架，你会发现在调用端发送请求消息之前还会经过过滤链，对请求消息进行层层的过滤处理，之后才会通过负载均衡选择服务节点，发送请求消息，而异常重试操作就发生在过滤链处理之后，调用负载均衡选择服务节点之前，这样的重试是可以减少很多重复操作的。

第十四讲

思考题：在启动预热那部分，我们特意提到过一个问题，就是“当大批量重启服务提供方的时候，会导致请求大概率发到没有重启的机器上，这时服务提供方有可能扛不住”，不知道你是怎么看待这个问题的，是否有好的解决方案呢？

我们可以考虑在非流量高峰的时候重启服务，将影响降到最低；也可以考虑分批次重启，控制好每批重启的服务节点的数量，当一批服务节点的权重与访问量都到正常水平时，再去重启下一批服务节点。

第十五讲

思考题：在使用 RPC 的过程中业务要实现自我保护，针对这个问题你是否还有其他的解决方案？

通过这一讲我们知道，在 RPC 调用中无论服务端还是调用端都需要自我保护，服务端自我保护的最简单有效的方式是“限流”，调用端则可以通过“熔断”机制来进行自我保护。

除了“熔断”和“限流”外，相信你一定听过“降级”这个词。简单来说就是当一个服务处理大量的请求达到一定压力的时候，我们可以让这个服务在处理请求时减少些非必要的功能，从而降低这个服务的压力。

还有就是我们可以通过服务治理，降低一个服务节点的权重来减轻某一方服务节点的请求压力，达到保护这个服务节点的目的。

第十六讲

思考题：在我们的实际工作中，测试人员和开发人员的工作一般都是并行的，这就导致一个问题经常出现：开发人员在开发过程中可能需要启动自身的应用，而测试人员为了能验证功能，会在测试环境中部署同样的应用。如果开发人员和测试人员用的接口分组名刚好一样，在这种情况下，就可能会干扰其它正在联调的调用方进行功能验证，进而影响整体的工作效率。不知道面对这种情况，你有什么好办法吗？

我们可以考虑配置不同的注册中心，开发人员将自己的服务注册到注册中心 A 上，而测试人员可以将自己的服务注册到测试专属的注册中心 B 上，这样测试人员在验证功能的时候，调用端会从注册中心 B 上拉取服务节点，开发人员重启自己的服务是影响不到测试人员的。

如果你使用过或者了解 k8s 的话，你一定知道“命名空间”的概念，RPC 框架如果支持“命名空间”，也是可以解决这一问题的。

今天的答疑就到这里，如果你有更多问题，欢迎继续在留言区中告知，我们共同讨论。下节课再见！

上一篇 16 | 业务分组：如何隔离流量？

下一篇 17 | 异步RPC：压榨单机吞吐量