# 10 | 网络通信优化之通信协议：如何优化RPC网络通信？

2019-06-11 刘超

你好，我是刘超。今天我将带你了解下服务间的网络通信优化。

上一讲中，我提到了微服务框架，其中SpringCloud和Dubbo的使用最为广泛，行业内也一直存在着对两者的比较，很多技术人会为这两个框架哪个更好而争辩。

我记得我们部门在搭建微服务框架时，也在技术选型上纠结良久，还曾一度有过激烈的讨论。当前SpringCloud炙手可热，具备完整的微服务生态，得到了很多同事的票选，但我们最终的选择却是Dubbo，这是为什么呢？

## RPC通信是大型服务框架的核心

我们经常讨论微服务，首要应该了解的就是微服务的核心到底是什么，这样我们在做技术选型时，才能更准确地把握需求。

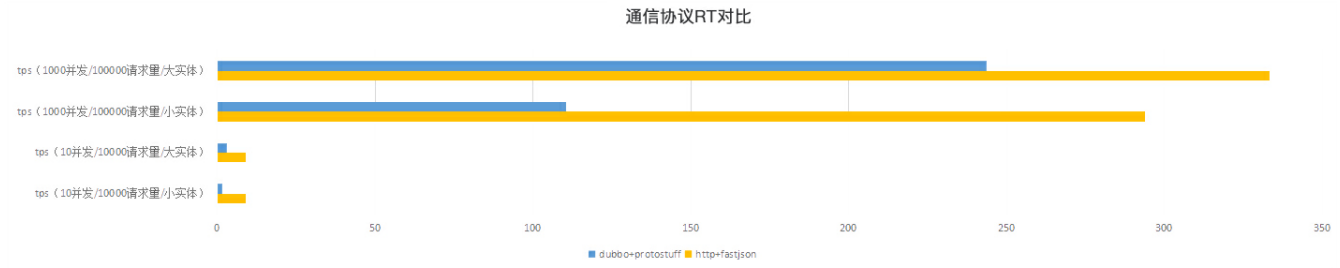
就我个人理解，我认为微服务的核心是远程通信和服务治理。远程通信提供了服务之间通信的桥梁，服务治理则提供了服务的后勤保障。所以，我们在做技术选型时，更多要考虑的是这两个核心的需求。

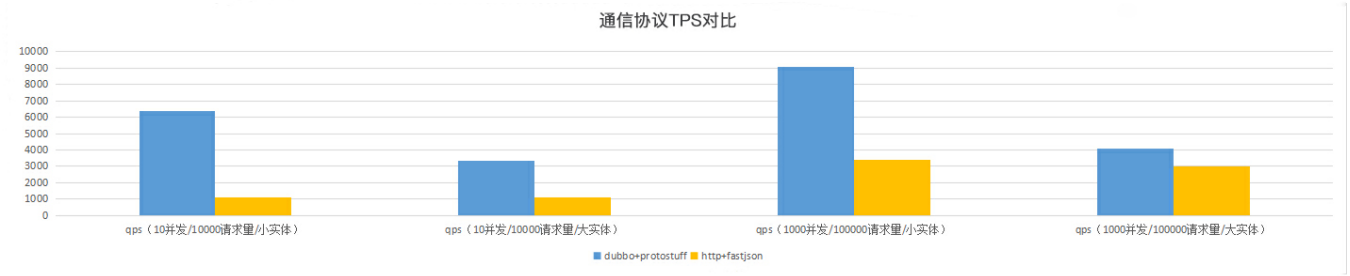
我们知道服务的拆分增加了通信的成本，特别是在一些抢购或者促销的业务场景中，如果服务之间存在方法调用，比如，抢购成功之后需要调用订单系统、支付系统、券包系统等，这种远程通信就很容易成为系统的瓶颈。所以，在满足一定的服务治理需求的前提下，对远程通信的性能需求就是技术选型的主要影响因素。

目前，很多微服务框架中的服务通信是基于RPC通信实现的，在没有进行组件扩展的前提下，SpringCloud是基于Feign组件实现的RPC通信（基于Http+Json序列化实现），Dubbo是基于SPI扩展了很多RPC通信框架，包括RMI、Dubbo、Hessian等RPC通信框架（默认是Dubbo+Hessian序列化）。不同的业务场景下，RPC通信的选择和优化标准也不同。

例如，开头我提到的我们部门在选择微服务框架时，选择了Dubbo。当时的选择标准就是RPC通信可以支持抢购类的高并发，在这个业务场景中，请求的特点是瞬时高峰、请求量大和传入、传出参数数据包较小。而Dubbo中的Dubbo协议就很好地支持了这个请求。

**以下是基于Dubbo:2.6.4版本进行的简单的性能测试。**分别测试Dubbo+Protobuf序列化以及Http+Json序列化的通信性能（这里主要模拟单一TCP长连接+Protobuf序列化和短连接的Http+Json序列化的性能对比）。为了验证在数据量不同的情况下二者的性能表现，我分别准备了小对象和大对象的性能压测，通过这样的方式我们也可以间接地了解下二者在RPC通信方面的水平。




这个测试是我之前的积累，基于测试环境比较复杂，这里我就直接给出结果了，如果你感兴趣的话，可以留言和我讨论。

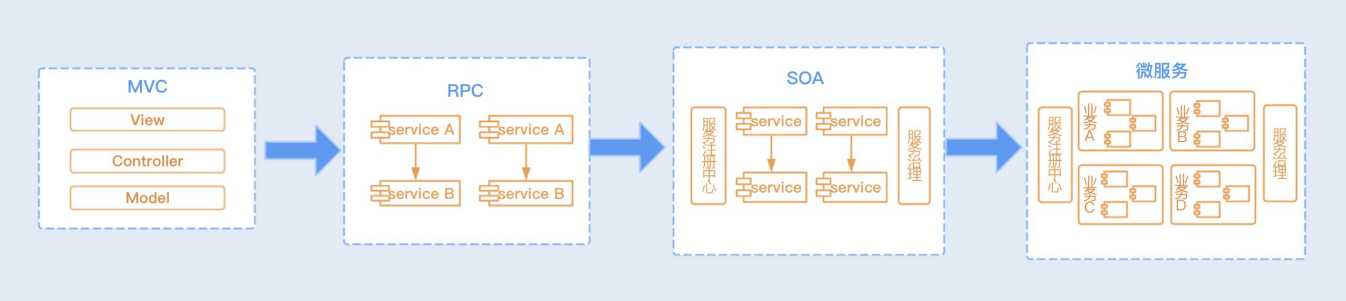
通过以上测试结果可以发现：**无论从响应时间还是吞吐量上来看，单一TCP长连接+Protobuf序列化实现的RPC通信框架都有着非常明显的优势。**

在高并发场景下，我们选择后端服务框架或者中间件部门自行设计服务框架时，RPC通信是重点优化的对象。

其实，目前成熟的RPC通信框架非常多，如果你们公司没有自己的中间件团队，也可以基于开源的RPC通信框架做扩展。在正式进行优化之前，我们不妨简单回顾下RPC。

## 什么是RPC通信

一提到RPC，你是否还想到MVC、SOA这些概念呢？如果你没有经历过这些架构的演变，这些概念就很容易混淆。**你可以通过下面这张图来了解下这些架构的演变史。**




无论是微服务、SOA、还是RPC架构，它们都是分布式服务架构，都需要实现服务之间的互相通信，我们通常把这种通信统称为RPC通信。

RPC（Remote Process Call），即远程服务调用，是通过网络请求远程计算机程序服务的通信技术。RPC框架封装好了底层网络通信、序列化等技术，我们只需要在项目中引入各个服务的接口包，就可以实现在代码中调用RPC服务同调用本地方法一样。正因为这种方便、透明的远程调用，RPC被广泛应用于当下企业级以及互联网项目中，是实现分布式系统的核心。

RMI（Remote Method Invocation）是JDK中最先实现了RPC通信的框架之一，RMI的实现对建立分布式Java应用程序至关重要，是Java体系非常重要的底层技术，很多开源的RPC通信框架也是基于RMI实现原理设计出来的，包括Dubbo框架中也接入了RMI框架。接下来我们就一起了解下RMI的实现原理，看看它存在哪些性能瓶颈有待优化。

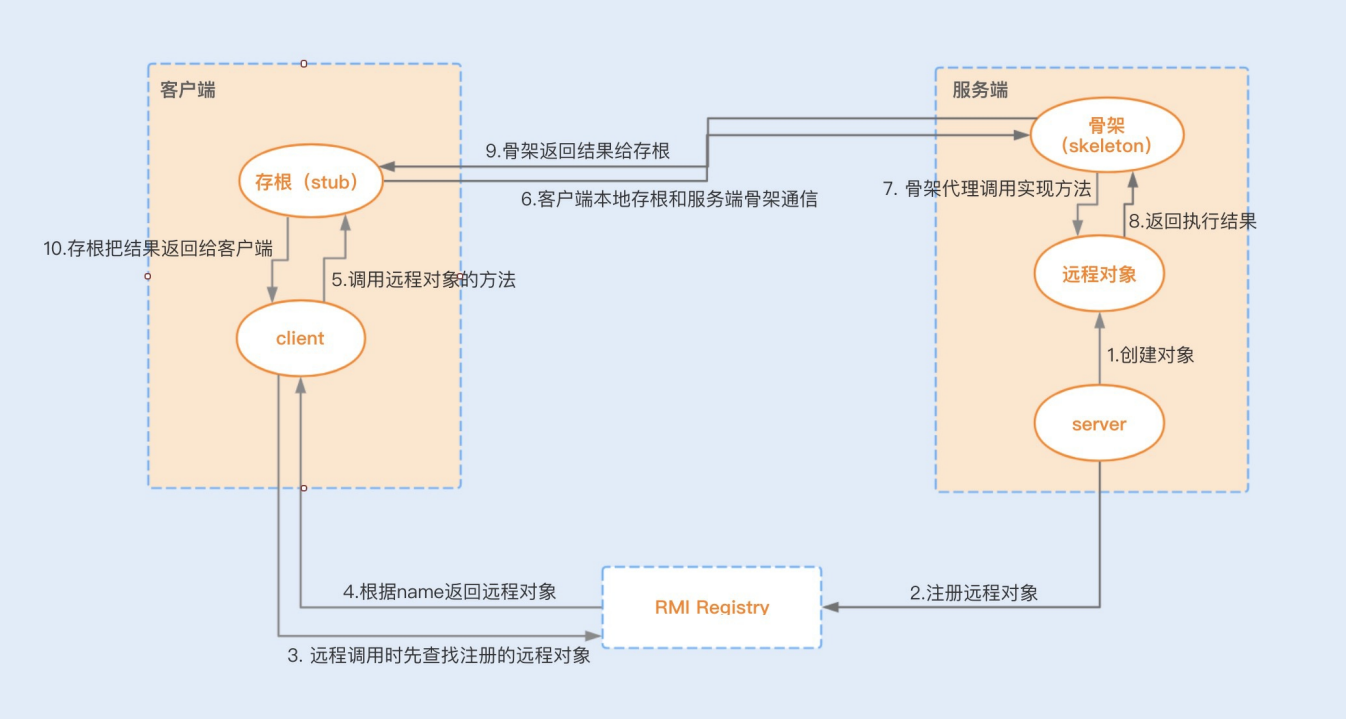
## RMI：JDK自带的RPC通信框架

目前RMI已经很成熟地应用在了EJB以及Spring框架中，是纯Java网络分布式应用系统的核心解决方案。RMI实现了一台虚拟机应用对远程方法的调用可以同对本地方法的调用一样，RMI帮我们封装好了其中关于远程通信的内容。

### RMI的实现原理

RMI远程代理对象是RMI中最核心的组件，除了对象本身所在的虚拟机，其它虚拟机也可以调用此对象的方法。而且这些虚拟机可以不在同一个主机上，通过远程代理对象，远程应用可以用网络协议与服务进行通信。

我们可以通过一张图来详细地了解下整个RMI的通信过程：




### RMI在高并发场景下的性能瓶颈

- Java默认序列化

RMI的序列化采用的是Java默认的序列化方式，我在09讲中详细地介绍过Java序列化，我们深知它的性能并不是很好，而且其它语言框架也暂时不支持Java序列化。

- TCP短连接

由于RMI是基于TCP短连接实现，在高并发情况下，大量请求会带来大量连接的创建和销毁，这对于系统来说无疑是非常消耗性能的。

- 阻塞式网络I/O

在08讲中，我提到了网络通信存在I/O瓶颈，如果在Socket编程中使用传统的I/O模型，在高并发场景下基于短连接实现的网络通信就很容易产生I/O阻塞，性能将会大打折扣。

## 一个高并发场景下的RPC通信优化路径

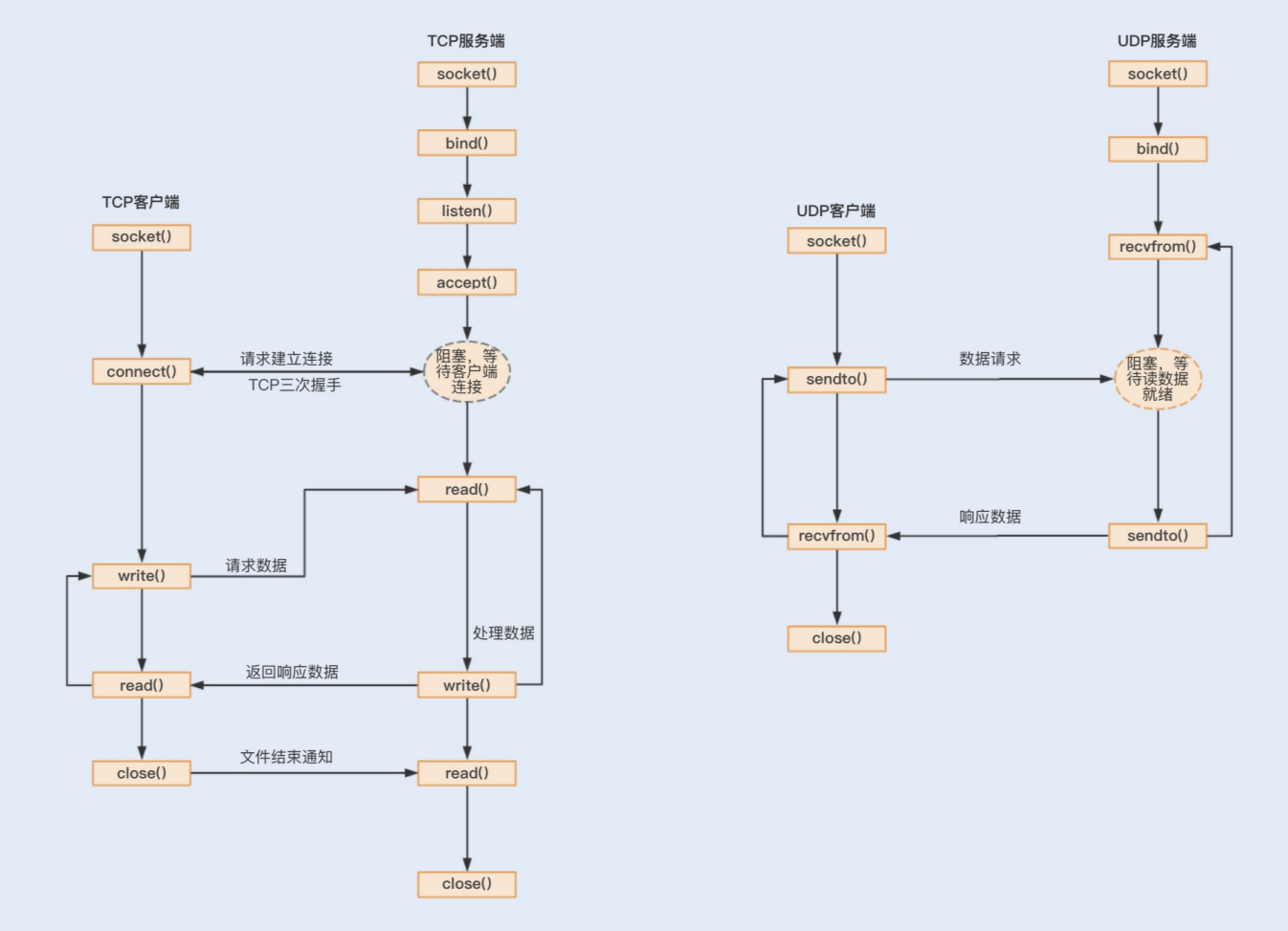
SpringCloud的RPC通信和RMI通信的性能瓶颈就非常相似。SpringCloud是基于Http通信协议（短连接）和Json序列化实现的，在高并发场景下并没有优势。 那么，在瞬时高并发的场景下，我们又该如何去优化一个RPC通信呢？

RPC通信包括了建立通信、实现报文、传输协议以及传输数据编解码等操作，接下来我们就从每一层的优化出发，逐步实现整体的性能优化。

### 1.选择合适的通信协议

要实现不同机器间的网络通信，我们先要了解计算机系统网络通信的基本原理。网络通信是两台设备之间实现数据流交换的过程，是基于网络传输协议和传输数据的编解码来实现的。其中网络

传输协议有TCP、UDP协议，这两个协议都是基于Socket编程接口之上，为某类应用场景而扩展出的传输协议。通过以下两张图，我们可以大概了解到基于TCP和UDP协议实现的Socket网络通信是怎样的一个流程。




基于TCP协议实现的Socket通信是有连接的，而传输数据是要通过三次握手来实现数据传输的可靠性，且传输数据是没有边界的，采用的是字节流模式。

基于UDP协议实现的Socket通信，客户端不需要建立连接，只需要创建一个套接字发送数据报给服务端，这样就不能保证数据报一定会达到服务端，所以在传输数据方面，基于UDP协议实现的Socket通信具有不可靠性。UDP发送的数据采用的是数据报模式，每个UDP的数据报都有一个长度，该长度将与数据一起发送到服务端。

通过对比，我们可以得出优化方法：为了保证数据传输的可靠性，通常情况下我们会采用TCP协议。如果在局域网且对数据传输的可靠性没有要求的情况下，我们也可以考虑使用UDP协议，毕竟这种协议的效率要比TCP协议高。

### 2.使用单一长连接

如果是基于TCP协议实现Socket通信，我们还能做哪些优化呢？

服务之间的通信不同于客户端与服务端之间的通信。客户端与服务端由于客户端数量多，基于短连接实现请求可以避免长时间地占用连接，导致系统资源浪费。

但服务之间的通信，连接的消费端不会像客户端那么多，但消费端向服务端请求的数量却一样多，我们基于长连接实现，就可以省去大量的TCP建立和关闭连接的操作，从而减少系统的性能消耗，节省时间。

### 3.优化Socket通信

建立两台机器的网络通信，我们一般使用Java的Socket编程实现一个TCP连接。传统的Socket通信主要存在I/O阻塞、线程模型缺陷以及内存拷贝等问题。我们可以使用比较成熟的通信框架，比如Netty。Netty4对Socket通信编程做了很多方面的优化，具体见下方。

**实现非阻塞I/O：**在08讲中，我们提到了多路复用器Selector实现了非阻塞I/O通信。

**高效的Reactor线程模型：**Netty使用了主从Reactor多线程模型，服务端接收客户端请求连接是用了一个主线程，这个主线程用于客户端的连接请求操作，一旦连接建立成功，将会监听I/O事件，监听到事件后会创建一个链路请求。

链路请求将会注册到负责I/O操作的I/O工作线程上，由I/O工作线程负责后续的I/O操作。利用这种线程模型，可以解决在高负载、高并发的情况下，由于单个NIO线程无法监听海量客户端和满足大量I/O操作造成的问题。

**串行设计：**服务端在接收消息之后，存在着编码、解码、读取和发送等链路操作。如果这些操作都是基于并行去实现，无疑会导致严重的锁竞争，进而导致系统的性能下降。为了提升性能，Netty采用了串行无锁化完成链路操作，Netty提供了Pipeline实现链路的各个操作在运行期间不进行线程切换。

**零拷贝：**在08讲中，我们提到了一个数据从内存发送到网络中，存在着两次拷贝动作，先是从用户空间拷贝到内核空间，再是从内核空间拷贝到网络I/O中。而NIO提供的ByteBuffer可以使用Direct Buffers模式，直接开辟一个非堆物理内存，不需要进行字节缓冲区的二次拷贝，可以直接将数据写入到内核空间。

**除了以上这些优化，我们还可以针对套接字编程提供的一些TCP参数配置项，提高网络吞吐量，Netty可以基于ChannelOption来设置这些参数。**

**TCP_NODELAY：**TCP_NODELAY选项是用来控制是否开启Nagle算法。Nagle算法通过缓存的方式将小的数据包组成一个大的数据包，从而避免大量的小数据包发送阻塞网络，提高网络传输的效率。我们可以关闭该算法，优化对于时延敏感的应用场景。

**SO_RCVBUF和SO_SNDBUF：**可以根据场景调整套接字发送缓冲区和接收缓冲区的大小。

**SO_BACKLOG：**backlog参数指定了客户端连接请求缓冲队列的大小。服务端处理客户端连接请求是按顺序处理的，所以同一时间只能处理一个客户端连接，当有多个客户端进来的时候，服务端就会将不能处理的客户端连接请求放在队列中等待处理。

**SO_KEEPALIVE**：当设置该选项以后，连接会检查长时间没有发送数据的客户端的连接状态，检测到客户端断开连接后，服务端将回收该连接。我们可以将该时间设置得短一些，来提高回收连接的效率。

### 4.量身定做报文格式

接下来就是实现报文，我们需要设计一套报文，用于描述具体的校验、操作、传输数据等内容。为了提高传输的效率，我们可以根据自己的业务和架构来考虑设计，尽量实现报体小、满足功能、易解析等特性。我们可以参考下面的数据格式：

| 魔数(0×12345678) | 版本号(1) | 序列化算法 | 指令 | 数据长度 | 数据 |
|---|---|---|---|---|---|
| 4字节 | 1字节 | 1字节 | 1字节 | 4字节 | N字节 |

| 字段 | 定义的作用 |
|---|---|
| 魔数 | 它的作用类似于协议内的标识，通过客户端与服务端魔数对比，我们就知道这组二进制数据是否属于当前通信协议，通常情况下魔数是一个固定数字。 |
| 版本号 | 占用 1 个字节，通常情况下是预留字段。 |
| 序列化算法 | 占用 1 个字节，表示对数据编码和解码的方式，比如，我在09讲介绍过的Java自带序列化，还有Google实现了Protobuf序列化。 |
| 指令 | 占用 1 个字节，服务端或者客户端每收到一种指令都会有相应的处理逻辑，最高支持256种指令，比如，Http中的增删改查指令。 |
| 数据长度 | 占用 4 个字节，这个字段用来截取数据。 |
| 数据 | 用于存储编码后的数据。 |

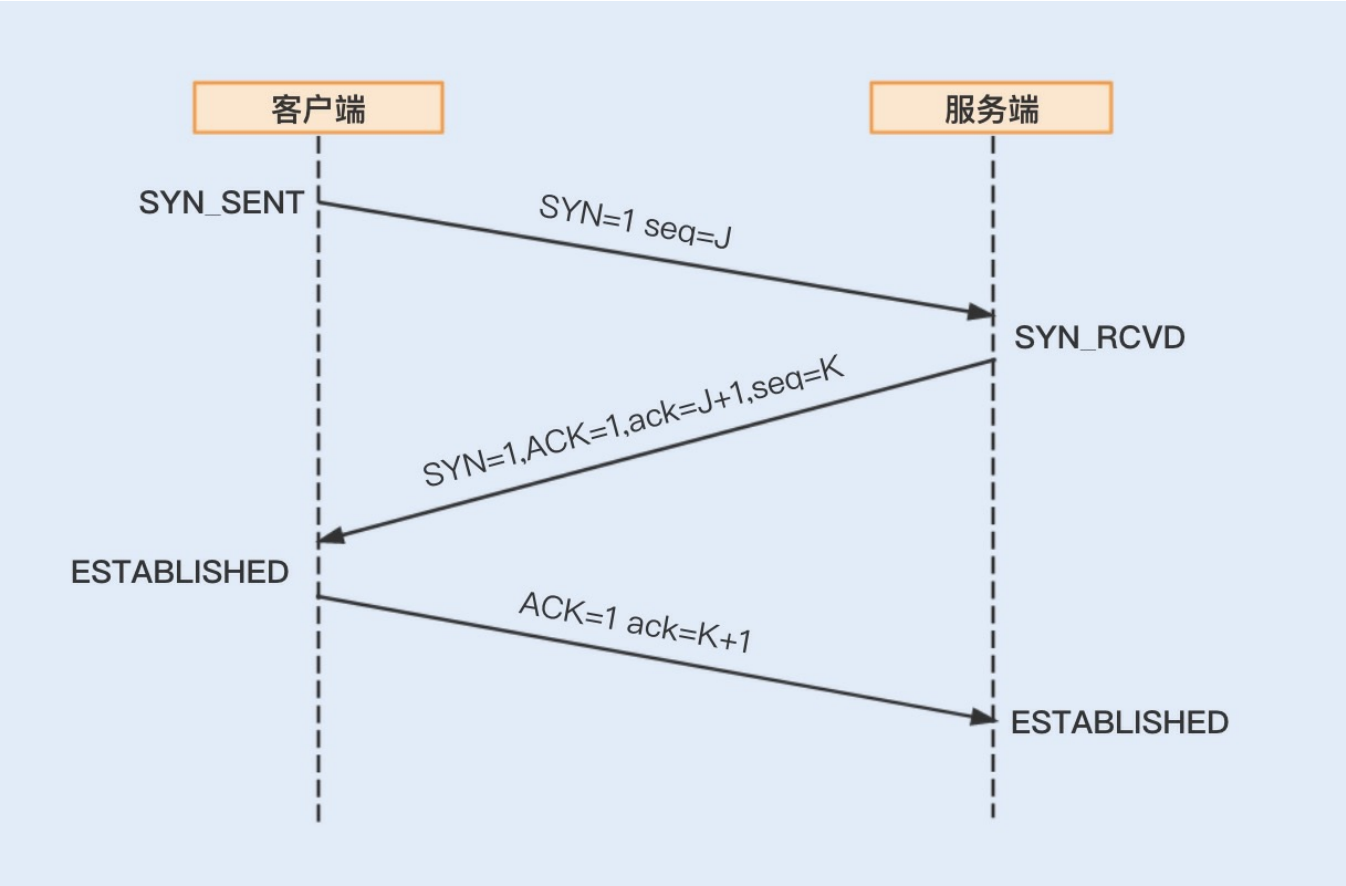
### 5.编码、解码

在09讲中，我们分析过序列化编码和解码的过程，对于实现一个好的网络通信协议来说，兼容优秀的序列化框架是非常重要的。如果只是单纯的数据对象传输，我们可以选择性能相对较好的Protobuf序列化，有利于提高网络通信的性能。

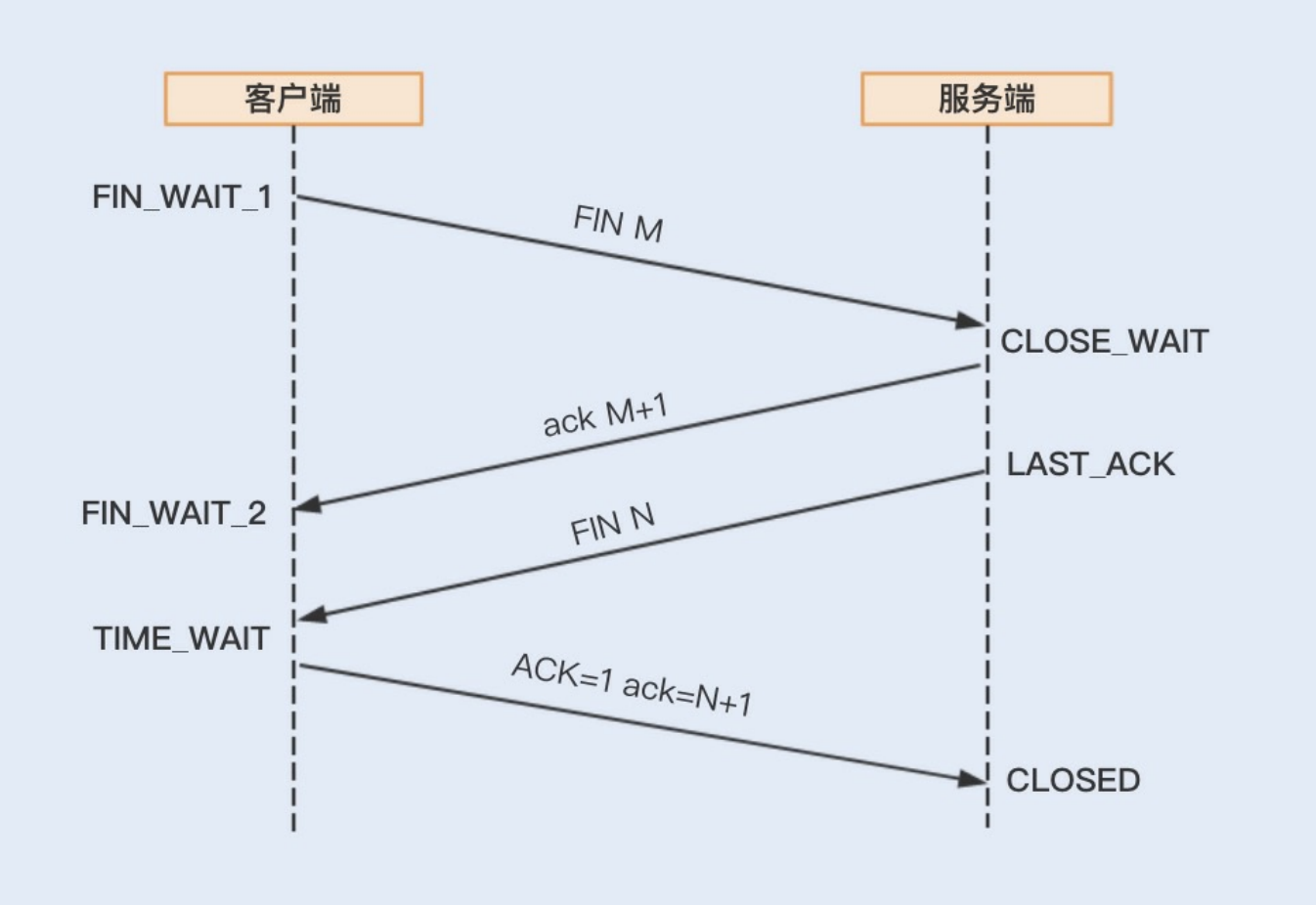
### 6.调整Linux的TCP参数设置选项

如果RPC是基于TCP短连接实现的，我们可以通过修改Linux TCP配置项来优化网络通信。开始TCP配置项的优化之前，我们先来了解下建立TCP连接的三次握手和关闭TCP连接的四次握手，这样有助后面内容的理解。

- 三次握手


客户端
服务端
SYN_SENT
SYN=1 seq=J
SYN_RCVD
SYN=1,ACK=1,ack=J+1,seq=K
ESTABLISHED
ACK=1 ack=K+1
ESTABLISHED

- 四次握手


客户端
服务端
FIN_WAIT_1
FIN M
CLOSE_WAIT
ack M+1
LAST_ACK
FIN_WAIT_2
FIN N
TIME_WAIT
ACK=1 ack=N+1
CLOSED

我们可以通过sysctl -a | grep net.xxx命令运行查看Linux系统默认的的TCP参数设置，如果需要修改某项配置，可以通过编辑 vim/etc/sysctl.conf，加入需要修改的配置项， 并通过sysctl -p命令运行生效修改后的配置项设置。通常我们会通过修改以下几个配置项来提高网络吞吐量和降低延时。

| 配置项 | 说明+优化方法 |
| --- | --- |
| file-max/ ulimit | Linux系统默认单个进程用户open files（一个socket连接相当于一个open file）数量为1024，有时应用程序会报"java.io.IOException：打开的文件过多"的错误，就说明open files数量不够。 |
| | 我们可以通过设置ulimit来增加单个用户的open files数量，file-max表示系统总的打开文件数量，如果单个进程的open files数量已经超过file-max，就也要修改file-max。 |
| net.ipv4.tcp_keepalive_time | 该配置项与Netty的SO_KEEPALIVE配置项的作用是一样的，用来检查客户端的连接状态。 |
| | 我们可以通过减少检查状态的时间，来提高连接的回收效率。 |
| net.ipv4.tcp_max_syn_backlog | 该配置项表示SYN队列的长度，默认为1024。 |
| | 加大队列长度，可以容纳更多等待连接的网络连接数。 |
| net.ipv4.tcp_syncookies | 当出现SYN等待队列溢出时，可以开启该配置项。 |
| | 启用cookies来处理，可防范少量SYN攻击。 |
| net.ipv4.ip_local_port_range | 客户端连接服务端时，需要动态分配源端口号，该配置项表示用于向外连接的端口范围。 |
| | 默认端口范围是32768到61000，我们可以扩大该范围。 |
| net.ipv4.tcp_max_tw_buckets | 当一个连接关闭时，TCP会通过四次握手来完成一次关闭连接操作。在请求量比较大的情况下，消费端会有大量TIME_WAIT状态的连接。 |
| | 由于该参数可以限制TIME_WAIT状态的连接数量，所以我们可以用减小参数的方式来应对。设置完成后，如果TIME_WAIT的数量超过参数值，TIME_WAIT将会立刻被清除并打印警告信息。 |
| net.ipv4.tcp_tw_reuse | 客户端每次连接服务端时，都会获得一个新的源端口以实现连接的唯一性。在TIME_WAIT状态的连接数量过大的情况下，会增加端口号的占用时间。 |
| | 由于处于TIME_WAIT状态的连接属于关闭连接，所以新创建的连接可以复用该端口号。 |

以上就是我们从不同层次对RPC优化的详解，除了最后的Linux系统中TCP的配置项设置调优，其它的调优更多是从代码编程优化的角度出发，最终实现了一套RPC通信框架的优化路径。

弄懂了这些，你就可以根据自己的业务场景去做技术选型了，还能很好地解决过程中出现的一些性能问题。

## 总结

在现在的分布式系统中，特别是系统走向微服务化的今天，服务间的通信就显得尤为频繁，掌握服务间的通信原理和通信协议优化，是你的一项的必备技能。

在一些并发场景比较多的系统中，我更偏向使用Dubbo实现的这一套RPC通信协议。Dubbo协议

是建立的单一长连接通信，网络I/O为NIO非阻塞读写操作，更兼容了Kryo、FST、Protobuf等性能出众的序列化框架，在高并发、小对象传输的业务场景中非常实用。

在企业级系统中，业务往往要比普通的互联网产品复杂，服务与服务之间可能不仅仅是数据传输，还有图片以及文件的传输，所以RPC的通信协议设计考虑更多是功能性需求，在性能方面不追求极致。其它通信框架在功能性、生态以及易用、易入门等方面更具有优势。

## 思考题

目前实现Java RPC通信的框架有很多，实现RPC通信的协议也有很多，除了Dubbo协议以外，你还使用过其它RPC通信协议吗？通过这讲的学习，你能对比谈谈各自的优缺点了吗？

期待在留言区看到你的见解。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起学习。



精选留言

WL

3

请教老师两个问题:

1. 在RMI的实现原理示意图中客户端的存根和服务端的骨架这两个概念是啥意思, 我感觉不太理解.

2. 在TCP的四次挥手中, 客户端最后的TIME_WAIT状态是不是就是CLOSE的状态, 如果不是那TIME_WAIT状态是在啥时候转换成CLOSE状态的.

2019-06-11

作者回复

我先回答第二个问题。

是的，TIME_WAIT状态就是主动断开方的最后状态了。主动断开连接方之所以是TIME_WAIT状态，是担心被断开方没有收到最后的ACK，这个TIME_WAIT时间内核默认设置是2MSL(报文最大生存时间)，被断开方如果超时没有收到ACK，将重新发送FIN，主动断开方收到之后又会重新发送ACK通知，重置TIME_WAIT时间。

正常情况下，当主动断开方的TIME_WAIT状态到达了定时时间后，内核就会关闭该连接。

第一个问题，这块文章中没有过多的介绍，我在这里再叙述下:

Stub是client端的远程对象的代理，负责将远程对象上的方法调用转发到实际远程对象实现所在的服务器，我们的程序要通过远程调用，底层一定是套接字的字节传输，要一个对象序列化成为字节码，传输到服务器或者客户端的对端之后，再把该对象反序列化成为对应的对象，Stub承担着底层序列化、数据组装以及协议封装等工作。

Skeleton则是server端的服务对象的代理，负责将接收解析远程调用分派到实际远程对象实现调用。Stub与Skeleton的关系以及操作是对应的关系，只有实现了java.rmi.Remote接口的类或者继承了java.rmi.Remote接口的接口，才能作为Stub与Skeleton之间通信的远程对象，Stub与Skeleton之间的通信使用Java远程消息交换协议JRMP（Java Remote Messaging Protocol）进行通信，JRMP是专为Java的远程对象制定的协议。Stub和Skeleton之间协作完成客户端与服务器之间的方法调用时的通信。

2019-06-11

Y、z

1

老师好，我想问下已经在线上跑的服务，序列化方式是hessian，如果直接换成Protobuf，那么consumer会报错吗？如果报错的话，如何避免这种情况发生呢？

2019-06-11

作者回复

服务端和消费端重启，会走protobuf序列化

2019-06-12

夏天39度

1

老师，能说一下Netty是如何实现串行无锁化完成链路操作吗，怎么做到无锁化的线程切换

2019-06-11

Stalary

1

老师，如果业务架构已经选择了SpringCloud，该如何优化远程调用呢，目前使用Feign，底层配置了HttpClient，发现qps一直上不去，暂时是对频繁的请求做了本地cache，但是需要订阅更新事件进行刷新

2019-06-11

作者回复

可以尝试扩展其他RPC框架，例如有同学提到的Google的grpc框架，也是基于Netty通信框架实现，基于protobuf实现的序列化。

2019-06-11

nightmare

0

能不能讲一下netty的串行无锁化

2019-06-13

电光火石

0

1. 老师线上有用过grpc吗，看文档说好像现在还不是特别的稳定？
2. 文中的性能测试，http是否有打开keep alive？走tcp无疑更快，我只想知道用http会慢多少，因为毕竟http更简单。有看过其他的benchmark，在打开keep alive的情况下，性能也还行，不知道老师这个测试是否打开？另外，从测试结果上看，当单次请求数据量很大的时候，http比tcp好像查不了多少是吗？

谢谢！

2019-06-11

作者回复

grpc目前很多公司在用，在Github中有源码阅读https://github.com/grpc。

对的，没有打开keep alive，如果开启效果会好一些，减少了网络连接。

2019-06-12

-W.LI-

0

老师好!文中提到了高效的 Reactor 线程模型。有适合新手的资料链接么?还有个pr啥的能一起给我个么谢谢了，文中讲的看不懂。

2019-06-11

作者回复

下一讲中会讲到Reactor、Proactor线程模型。

2019-06-12

JackJin

0

RPC要怎么学，项目中也用到了dubbo，可跟老师说的完全不一样，不是添加其他服务接口的依赖调用方法，而是发送http请求调用其他服务的接口。使我们用错了吗？

有个疑问依赖其他的服务的接口调用方法，接口都没有实现怎么调用？

2019-06-11

作者回复

请问你们用的什么协议？dubbo是消费服务类型，服务会注册接口到注册中心，消费端通过拉取注册中心的注册服务接口，与服务端通信。所以一般都是通过接口服务实现消息通信。

2019-06-12

晓杰

0

请问老师，对于大文件的传输，用哪种协议比较好

2019-06-11

作者回复

建议使用hessian协议

2019-06-12

假装自己不胖

0

对于网络编程比较迷茫，请问有没有小白一些的读物或博客推荐一下

2019-06-11

作者回复

这块知识点比较多，建议可以看一些基础书籍，例如Unix网络编程、TCP/IP网络编程，再看看netty实战，就可以进阶Java网络编程了。

2019-06-13

Liam

0

能否讲讲不同情况下，tcp各个调优参数的值应该怎么设和通常设置为多少

2019-06-11

黑崽

0

断开连接是四次挥手吧

2019-06-11

作者回复

是的，文中提到了。

2019-06-11

进阶的码农

0

我们是用spring-cloud 继承google的grpc

2019-06-11

作者回复

优秀

2019-06-11