

第35讲 | 二进制类RPC协议：还是叫NBA吧，总说全称多费劲

2018-08-06 刘超



朗读人：刘超

时长12:46 大小5.86M



前面我们讲了两个常用文本类的 RPC 协议，对于陌生人之间的沟通，用 NBA、CBA 这样的缩略语，会使得协议约定非常不方便。

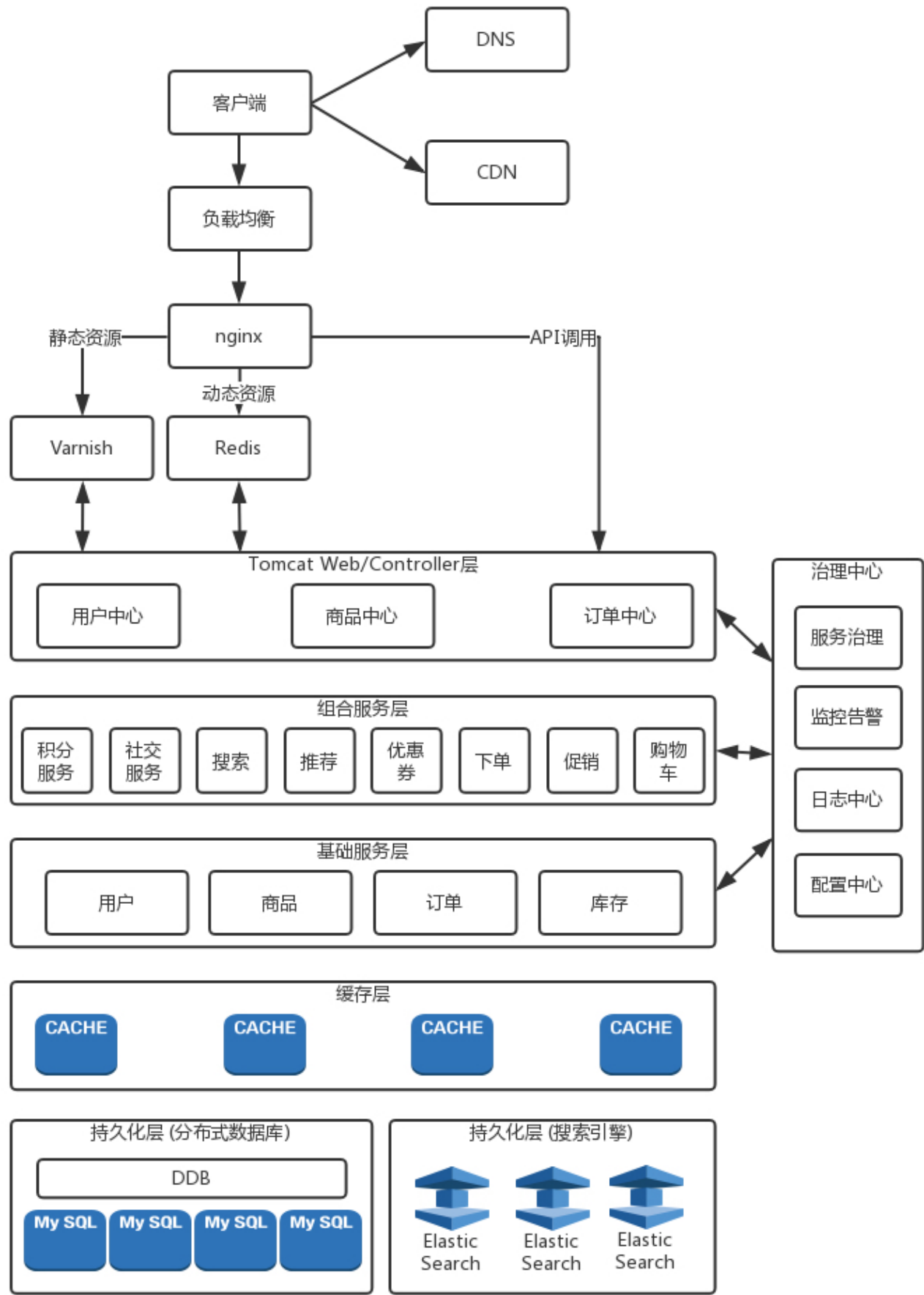
在讲 CDN 和 DNS 的时候，我们讲过接入层的设计，对于静态资源或者动态资源静态化的部分都可以做缓存。但是对于下单、支付等交易场景，还是需要调用 API。

对于微服务的架构，API 需要一个 API 网关统一的管理。API 网关有多种实现方式，用 Nginx 或者 OpenResty 结合 Lua 脚本是常用的方式。在上一节讲过的 Spring Cloud 体系中，有个组件 Zuul 也是干这个的。

数据中心内部是如何相互调用的？

API 网关用来管理 API，但是 API 的实现一般在一个叫作 **Controller 层** 的地方。这一层对外提供 API。由于是让陌生人访问的，我们能看到目前业界主流的，基本都是 RESTful 的

API，是面向大规模互联网应用的。



在 Controller 之内，就是咱们互联网应用的业务逻辑实现。上节讲 RESTful 的时候，说过业务逻辑的实现最好是无状态的，从而可以横向扩展，但是资源的状态还需要服务端去维护。资源的状态不应该维护在业务逻辑层，而是在最底层的持久化层，一般会使用分布式数据库和 ElasticSearch。

这些服务端的状态，例如订单、库存、商品等，都是重中之重，都需要持久化到硬盘上，数据不能丢，但是由于硬盘读写性能差，因而持久化层往往吞吐量不能达到互联网应用要求的吞吐量，因而前面要有一层缓存层，使用 Redis 或者 memcached 将请求拦截一道，不能让所有的请求都进入数据库“中军大营”。

缓存和持久化层之上一般是**基础服务层**，这里面提供一些原子化的接口。例如，对于用户、商品、订单、库存的增删查改，将缓存和数据库对再上层的业务逻辑屏蔽一道。有了这一层，上层业务逻辑看到的都是接口，而不会调用数据库和缓存。因而对于缓存层的扩容，数据库的分库分表，所有的改变，都截止到这一层，这样有利于将来对于缓存和数据库的运维。

再往上就是**组合层**。因为基础服务层只是提供简单的接口，实现简单的业务逻辑，而复杂的业务逻辑，比如下单，要扣优惠券，扣减库存等，就要在组合服务层实现。

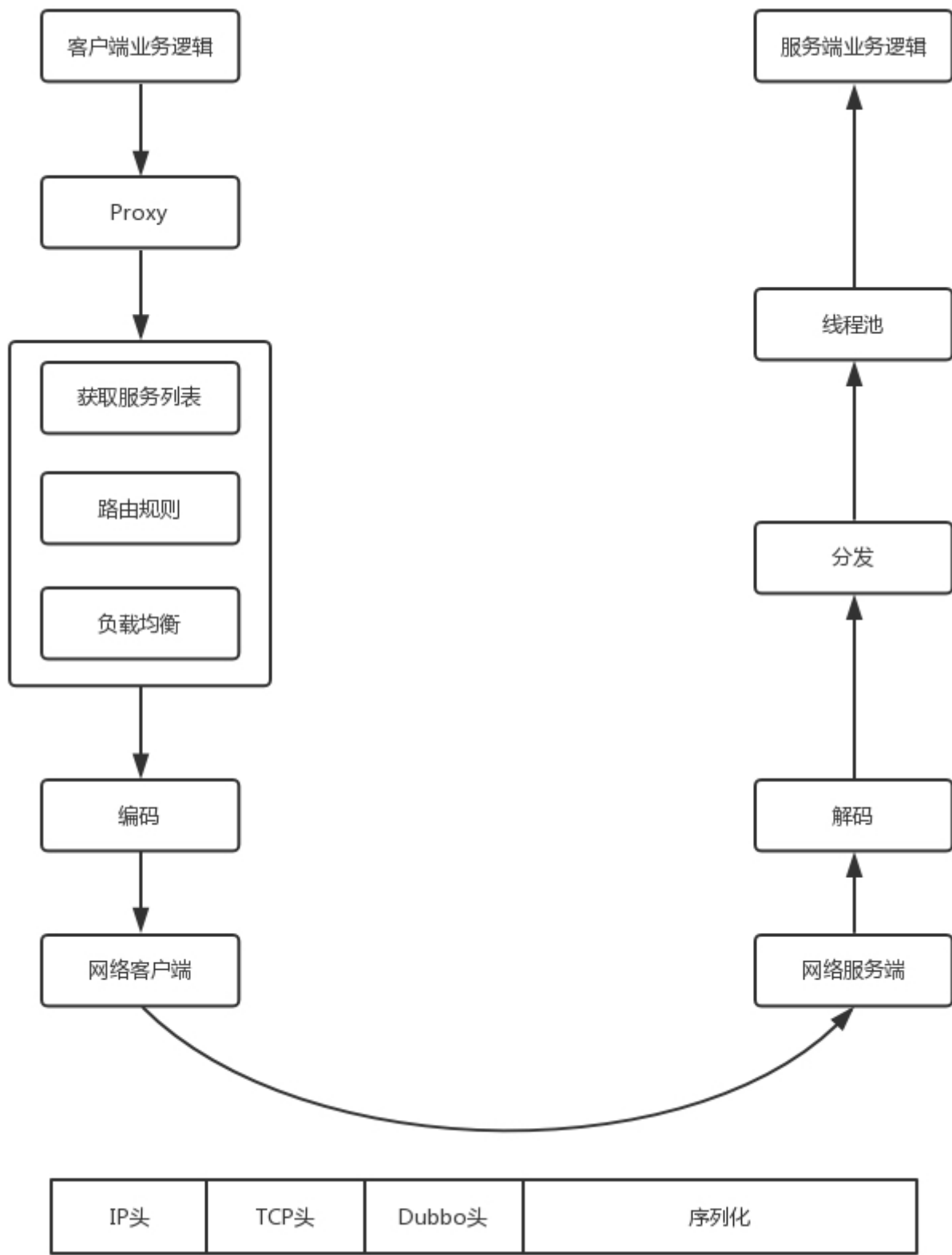
这样，Controller 层、组合服务层、基础服务层就会相互调用，这个调用是在数据中心内部的，量也会比较大，还是使用 RPC 的机制实现的。

由于服务比较多，需要一个单独的注册中心来做服务发现。服务提供方会将自己提供哪些服务注册到注册中心中去，同时服务消费方订阅这个服务，从而可以对这个服务进行调用。

调用的时候有一个问题，这里的 RPC 调用，应该用二进制还是文本类？其实文本的最大问题是，占用字节数目比较多。比如数字 123，其实本来二进制 8 位就够了，但是如果变成文本，就成了字符串 123。如果是 UTF-8 编码的话，就是三个字节；如果是 UTF-16，就是六个字节。同样的信息，要多费好多的空间，传输起来也更加占带宽，时延也高。

因而对于数据中心内部的相互调用，很多公司选型的时候，还是希望采用更加省空间和带宽的二进制的方案。

这里一个著名的例子就是 Dubbo 服务化框架二进制的 RPC 方式。



Dubbo 会在客户端的本地启动一个 Proxy，其实就是客户端的 Stub，对于远程的调用都通过这个 Stub 进行封装。

接下来，Dubbo 会从注册中心获取服务端的列表，根据路由规则和负载均衡规则，在多个服务端中选择一个最合适的服务端进行调用。

调用服务端的时候，首先要进行编码和序列化，形成 Dubbo 头和序列化的方法和参数。将编码好的数据，交给网络客户端进行发送，网络服务端收到消息后，进行解码。然后将任务分发给某个线程进行处理，在线程中会调用服务端的代码逻辑，然后返回结果。

这个过程和经典的 RPC 模式何其相似啊！

如何解决协议约定问题？

接下来我们还是来看 RPC 的三大问题，其中注册发现问题已经通过注册中心解决了。我们下面就来看协议约定问题。

Dubbo 中默认的 RPC 协议是 Hessian2。为了保证传输的效率，Hessian2 将远程调用序列化为二进制进行传输，并且可以进行一定的压缩。这个时候你可能会疑惑，同为二进制的序列化协议，Hessian2 和前面的二进制的 RPC 有什么区别呢？这不绕了一圈又回来了吗？

Hessian2 是解决了一些问题的。例如，原来要定义一个协议文件，然后通过这个文件生成客户端和服务端的 Stub，才能进行相互调用，这样使得修改就会不方便。Hessian2 不需要定义这个协议文件，而是自描述的。什么是自描述呢？

所谓自描述就是，关于调用哪个函数，参数是什么，另一方不需要拿到某个协议文件、拿到二进制，靠它本身根据 Hessian2 的规则，就能解析出来。

原来有协议文件的场景，有点儿像两个人事先约定好，0 表示方法 add，然后后面会传两个数。服务端把两个数加起来，这样一方发送 012，另一方知道是将 1 和 2 加起来，但是不知道协议文件的，当它收到 012 的时候，完全不知道代表什么意思。

而自描述的场景，就像两个人说的每句话都带前因后果。例如，传递的是“函数：add，第一个参数 1，第二个参数 2”。这样无论谁拿到这个表述，都知道是什么意思。但是只不过都是以二进制的形式编码的。这其实相当于综合了 XML 和二进制共同优势的一个协议。


Hessian2 是如何做到这一点的呢？这就需要去看 Hessian2 的序列化的[语法描述文件](#)。

top	# starting production ::= value	long	# 64-bit signed long integer ::= 'L' b7 b6 b5 b4 b3 b2 b1 b0 ::= [xd8-xef] # -x08 to x0f ::= [xf0-fff] b0 # -x800 to x7ff ::= [x38-x3f] b1 b0 # -x40000 to x3ffff ::= x59 b3 b2 b1 b0 # 32-bit integer cast to long
binary	# 8-bit binary data split into 64k chunks ::= x41 b1 b0 <binary-data> binary # non-final chunk ::= 'B' b1 b0 <binary-data> # final chunk ::= [x20-x2f] <binary-data> # binary data of # length 0-15 ::= [x34-x37] <binary-data> # binary data of # length 0-1023	map	# map/object ::= 'M' type (value value)* 'Z' # key, value map pairs ::= 'H' (value value)* 'Z' # untyped key, value
boolean	# boolean true/false ::= 'T' ::= 'F'	null	# null value ::= 'N'
class-def	# definition for an object (compact map) ::= 'C' string int string*	object	# Object instance ::= 'O' int value* ::= [x60-x6f] value*
date	# time in UTC encoded as 64-bit long milliseconds since # epoch ::= x4a b7 b6 b5 b4 b3 b2 b1 b0 ::= x4b b3 b2 b1 b0 # minutes since epoch	ref	# value reference (e.g. circular trees and graphs) ::= x51 int # reference to nth map/list/object
double	# 64-bit IEEE double ::= 'D' b7 b6 b5 b4 b3 b2 b1 b0 ::= x5b # 0.0 ::= x5c # 1.0 ::= x5d b0 # byte cast to double # (-128.0 to 127.0) ::= x5e b1 b0 # short cast to double ::= x5f b3 b2 b1 b0 # 32-bit float cast to double	string	# UTF-8 encoded character string split into 64k chunks ::= x52 b1 b0 <utf8-data> string # non-final chunk ::= 'S' b1 b0 <utf8-data> # string of length # 0-65535 ::= [x00-x1f] <utf8-data> # string of length # 0-31 ::= [x30-x34] <utf8-data> # string of length # 0-1023
int	# 32-bit signed integer ::= 'I' b3 b2 b1 b0 ::= [x80-xbf] # -x10 to x3f ::= [xc0-xcf] b0 # -x800 to x7ff ::= [xd0-xd7] b1 b0 # -x40000 to x3ffff	type	# map/list types for OO languages ::= string # type name ::= int # type reference
list	# list/vector ::= x55 type value* 'Z' # variable-length list ::= 'V' type int value* # fixed-length list ::= x57 value* 'Z' # variable-length untyped list ::= x58 int value* # fixed-length untyped list ::= [x70-77] type value* # fixed-length typed list ::= [x78-7f] value* # fixed-length untyped list	value	# main production ::= null ::= binary ::= boolean ::= class-def value ::= date ::= double ::= int ::= list ::= long ::= map ::= object ::= ref ::= string

看起来很复杂，编译原理里面是有这样的语法规则的。

我们从 Top 看起，下一层是 value，直到形成一棵树。这里面的有个思想，为了防止歧义，每一个类型的起始数字都设置成为独一无二的。这样，解析的时候，看到这个数字，就知道后面跟的是什么了。

这里还是以加法为例子，“add(2,3)”被序列化之后是什么样的呢？

 复制代码

```

1 H x02 x00      # Hessian 2.0
2 C              # RPC call
3  x03 add       # method "add"
4  x92           # two arguments
5  x92           # 2 - argument 1
6  x93           # 3 - argument 2

```

H 开头，表示使用的协议是 Hessian，H 的二进制是 0x48。

C 开头，表示这是一个 RPC 调用。


0x03，表示方法名是三个字符。

0x92，表示有两个参数。其实这里存的应该是 2，之所以加上 0x90，就是为了防止歧义，表示这里一定是一个 int。

第一个参数是 2，编码为 0x92，第二个参数是 3，编码为 0x93。

这个就叫作**自描述**。

另外，Hessian2 是面向对象的，可以传输一个对象。

 复制代码

```
1 class Car {
2   String color;
3   String model;
4 }
5 out.writeObject(new Car("red", "corvette"));
6 out.writeObject(new Car("green", "civic"));
7 ---
8 C          # object definition (#0)
9 0xb example.Car  # type is example.Car
10 0x92        # two fields
11 0x05 color   # color field name
12 0x05 model   # model field name
13
14 0          # object def (long form)
15 0x90        # object definition #0
16 0x03 red    # color field value
17 0x08 corvette  # model field value
18
19 0x60        # object def #0 (short form)
20 0x05 green  # color field value
21 0x05 civic  # model field value
```

首先，定义这个类。对于类型的定义也传过去，因而也是自描述的。类名为 example.Car，字符长 11 位，因而前面长度为 0x0b。有两个成员变量，一个是 color，一个是 model，字符长 5 位，因而前面长度 0x05。

然后，传输的对象引用这个类。由于类定义在位置 0，因而对象会指向这个位置 0，编码为 0x90。后面 red 和 corvette 是两个成员变量的值，字符长分别为 3 和 8。

接着又传输一个属于相同类的对象。这时候就不保存对于类的引用了，只保存一个 0x60，表示同上就可以了。

可以看出，Hessian2 真的是能压缩尽量压缩，多一个 Byte 都不传。

如何解决 RPC 传输问题？

接下来，我们再来看 Dubbo 的 RPC 传输问题。前面我们也说了，基于 Socket 实现一个高性能的服务端，是很复杂的一件事情，在 Dubbo 里面，使用了 Netty 的网络传输框架。

Netty 是一个非阻塞的基于事件的网络传输框架，在服务端启动的时候，会监听一个端口，并注册以下的事件。

连接事件：当收到客户端的连接事件时，会调用 `void connected(Channel channel)` 方法。

当**可写事件**触发时，会调用 `void sent(Channel channel, Object message)`，服务端向客户端返回响应数据。

当**可读事件**触发时，会调用 `void received(Channel channel, Object message)`，服务端在收到客户端的请求数据。

当**发生异常**时，会调用 `void caught(Channel channel, Throwable exception)`。

当事件触发之后，服务端在这些函数中的逻辑，可以选择直接在这个函数里面进行操作，还是将请求分发到线程池去处理。一般异步的数据读写都需要另外的线程池参与，在线程池中会调用真正的服务端业务代码逻辑，返回结果。

Hessian2 是 Dubbo 默认的 RPC 序列化方式，当然还有其他选择。例如，Dubbox 从 Spark 那里借鉴 Kryo，实现高性能的序列化。

到这里，我们说了数据中心里面的相互调用。为了高性能，大家都愿意用二进制，但是为什么后期 Spring Cloud 又兴起了呢？这是因为，并发量越来越大，已经到了微服务的阶段。同原来的 SOA 不同，微服务粒度更细，模块之间的关系更加复杂。

在上面的架构中，如果使用二进制的方式进行序列化，虽然不用协议文件来生成 Stub，但是对于接口的定义，以及传的对象 DTO，还是需要共享 JAR。因为只有客户端和服务端都有这个 JAR，才能成功地序列化和反序列化。

但当关系复杂的时候，JAR 的依赖也变得异常复杂，难以维护，而且如果在 DTO 里加一个字段，双方的 JAR 没有匹配好，也会导致序列化不成功，而且还有可能循环依赖。这个时候，一般有两种选择。

第一种，建立严格的项目管理流程。

不允许循环调用，不允许跨层调用，只准上层调用下层，不允许下层调用上层。

接口要保持兼容性，不兼容的接口新添加而非改原来的，当接口通过监控，发现不用的时候，再下掉。

升级的时候，先升级服务提供端，再升级服务消费端。

第二种，改用 RESTful 的方式。

使用 Spring Cloud，消费端和提供端不用共享 JAR，各声明各的，只要能变成 JSON 就行，而且 JSON 也是比较灵活的。

使用 RESTful 的方式，性能会降低，所以需要通过横向扩展来抵消单机的性能损耗。

这个时候，就看架构师的选择喽！

小结

好了，这节就到这里了，我们来总结一下。

RESTful API 对于接入层和 Controller 层之外的调用，已基本形成事实标准，但是随着内部服务之间的调用越来越多，性能也越来越重要，于是 Dubbo 的 RPC 框架有了用武之地。

Dubbo 通过注册中心解决服务发现问题，通过 Hessian2 序列化解决协议约定的问题，通过 Netty 解决网络传输的问题。

在更加复杂的微服务场景下，Spring Cloud 的 RESTful 方式在内部调用也会被考虑，主要是 JAR 包的依赖和管理问题。

最后，给你留两个思考题。

1. 对于微服务模式下的 RPC 框架的选择, Dubbo 和 SpringCloud 各有优缺点, 你能做个详细的对比吗?
2. 到目前为止, 我们讲过的 RPC, 还没有跨语言调用的场景, 你知道如果跨语言应该怎么办吗?

我们的专栏更新到第 35 讲, 不知你掌握得如何? 每节课后我留的思考题, 你都有没有认真思考, 并在留言区写下答案呢? 我会从**已发布的文章中选出一批认真留言的同学**, 赠送**学习奖励礼券**和我整理的**独家网络协议知识图谱**。

欢迎你留言和我讨论。趣谈网络协议, 我们下期见!

 极客时间

趣谈网络协议

像小说一样的网络协议入门课

刘超 网易研究院
云计算技术部首席架构师



新版升级: 点击「 请朋友读」, 10位好友免费读, 邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有, 未经许可不得转载

上一篇 第34讲 | 基于JSON的RESTful接口协议: 我不关心过程, 请给我结果

下一篇 第36讲 | 跨语言类RPC协议: 交流之前, 双方先来个专业术语表

精选留言

 写留言



问题究竟系边度 置顶



2018-08-10

1

dubbo 是这个rpc框架包括服务发现，服务均衡负载，接口层面监控。对于rpc中的扩展点比较多。后面会用servicemesh ,传输协议较多选择

...

展开 ▾

**blackpiglet**

2018-08-10

3

第二题，可以使用 thrift 和 protobuf

**忆水寒**

2018-08-07

1

跨语言调用的场景，可以使用序列化工具，比如Thrift、protobuf等序列化框架。

**阿痕**

2018-08-06

1

请教下，文中说的dubbo的jar包，具体是指啥？我们公司正在用dubbo，不需要在应用离单独部署jar包啊

展开 ▾

**及子龙**

2018-08-06

1

我们用的是gRpc，对多语言支持的比较好。

**andy**

2018-08-06

1

spring cloud的restful方式虽然基于json，但是服务端在发送数据之前会将DTO对象转换为JSON，客户端收到JSON之后还会转换为DTO。这时会在客户端和服务端分别创建各自的DTO对象，会出现代码的重复，如果共享jar，又出现jar管理的问题。

作者回复: 是的，我们是各自定义



**Jay**

2018-08-06

1

题目2:

可以使用Thrift和Protocol Buffers。

Thrift是Facebook提供的跨语言轻量级RPC消息和数据交换框架； ...

展开 ▾

**Jay**

2018-08-06

1

题目1:

1.Dubbo只实现了服务治理，而Spring Cloud子项目分别覆盖了微服务架构下的众多部件。 ...

展开 ▾

**stany**

2018-08-06

0

深入浅出，条理很清晰了。

**_CountingStars**

2018-08-06

0

2.跨语言如果使用 restful 基本可以直接用 如果用二进制rpc需要分别实现相应的客户端 sdk