深度解析gRPC以及京东分布式服务框架跨语言实战

2017-02-22 闫家宝 开涛的博客

本文转载自IPD-Chat，IPD-Chat为京东商城基础平台部门官方公众号，扫一扫二维码进行关注。

gRPC是什么

gRPC是Google 开发的基于HTTP/2和Protocol Buffer 3的RPC 框架。

gRPC是开源的，有 C、Java、Go等多种语言的实现，可以轻松实现跨语言调用。

声称是"一个高性能,开源,将移动和HTTP/2放在首位的通用的RPC框架"

当前版本1.1.1，主要技术栈：Netty-4.1.8，Protobuff-3.1.0，Guava-20.0
Multi-language, multi-platform framework

- Native implementations in C, Java, and Go
- C stack wrapped by C++, C#, Node, ObjC, Python, Ruby, PHP
- Platforms supported: Linux, Android, iOS, MacOS, Windows

gRPC设计动机和原则

最初由Louis Ryan在谷歌其他同事帮助下写成，如下文：

设计动机

十多年来，谷歌一直使用一个叫做Stubby的通用RPC基础框架，用它来连接在其数据中心内和跨数据中心运行的大量微服务。其内部系统早就接受了如今越来越流行的微服务架构。拥有一个统一的、跨平台的RPC的基础框架，使得服务的首次发行在效率、安全性、可靠性和行为分析上得到全面提升，这是支撑这一时期谷歌快速增长的关键。

Stubby有许多非常棒的特性，然而，它没有基于任何标准，而且与其内部的基础框架耦合得太紧密以至于被认为不适合公开发布。随着SPDY、HTTP/2和QUIC的到来，许多类似特性在公共标准中出现，并提供了Stubby不支持的其它功能。很明显，是时候利用这些标准来重写Stubby，并将其适用性扩展到移动、物联网和云场景。

设计原则

- 服务非对象、消息非引用 —— 促进微服务的系统间粗粒度消息交互设计理念，同时避免分布式对象的陷阱和分布式计算的谬误。
- 普遍并且简单 —— 该基础框架应该在任何流行的开发平台上适用，并且易于被个人在自己的平台上构建。它在CPU和内存有限的设备上也应该切实可行。
- 免费并且开源 —— 所有人可免费使用基本特性。以友好的许可协议开源方式发布所有交付件。
- 互通性 —— 该报文协议(Wire Protocol)必须遵循普通互联网基础框架。
- 通用并且高性能 —— 该框架应该适用于绝大多数用例场景，相比针对特定用例的框架，该框架只会牺牲一点性能。
- 分层的 —— 该框架的关键是必须能够独立演进。对报文格式(Wire Format)的修改不应该影响应用层。
- 负载无关的 —— 不同的服务需要使用不同的消息类型和编码，例如protocol buffers、JSON、XML和Thrift，协议上和实现上必须满足这样的诉求。类似地，对负载压缩的诉求也因应用场景和负载类型不同而不同，协议上应该支持可插拔的压缩机制。
- 流 —— 存储系统依赖于流和流控来传递大数据集。像语音转文本或股票代码等其它服务，依靠流表达时间相关的消息序列。

● 阻塞式和非阻塞式 —— 支持异步和同步处理在客户端和服务端间交互的消息序列。这是在某些平台上缩放和处理流的关键。
● 取消和超时 —— 有的操作可能会用时很长，客户端运行正常时，可以通过取消操作让服务端回收资源。当任务因果链被追踪时，取消可以级联。客户端可能会被告知调用超时，此时服务就可以根据客户端的需求来调整自己的行为。
● Lameducking —— 服务端必须支持优雅关闭，优雅关闭时拒绝新请求，但继续处理正在运行中的请求。
● 流控 —— 在客户端和服务端之间，计算能力和网络容量往往是不平衡的。流控可以更好的缓冲管理，以及保护系统免受来自异常活跃对端的拒绝服务(DOS)攻击。
● 可插拔的 —— 数据传输协议(Wire Protocol)只是功能完备API基础框架的一部分。大型分布式系统需要安全、健康检查、负载均衡和故障恢复、监控、跟踪、日志等。实　现上应该提供扩展点，以允许插入这些特性和默认实现。
● API扩展 —— 可能的话，在服务间协作的扩展应该最好使用接口扩展，而不是协议扩展。这种类型的扩展可以包括健康检查、服务内省、负载监测和负载均衡分配。
● 元数据交换 —— 常见的横切关注点，如认证或跟踪，依赖数据交换，但这不是服务公共接口中的一部分。部署依赖于他们将这些特性以不同速度演进到服务暴露的个别API的能力。
● 标准化状态码 —— 客户端通常以有限的方式响应API调用返回的错误。应该限制状态代码名字空间，使得这些错误处理决定更清晰。如果需要更丰富的特定域的状态，可以使用元数据交换机制来提供。

关于HTTP/2和Protocol Buffer 3简介

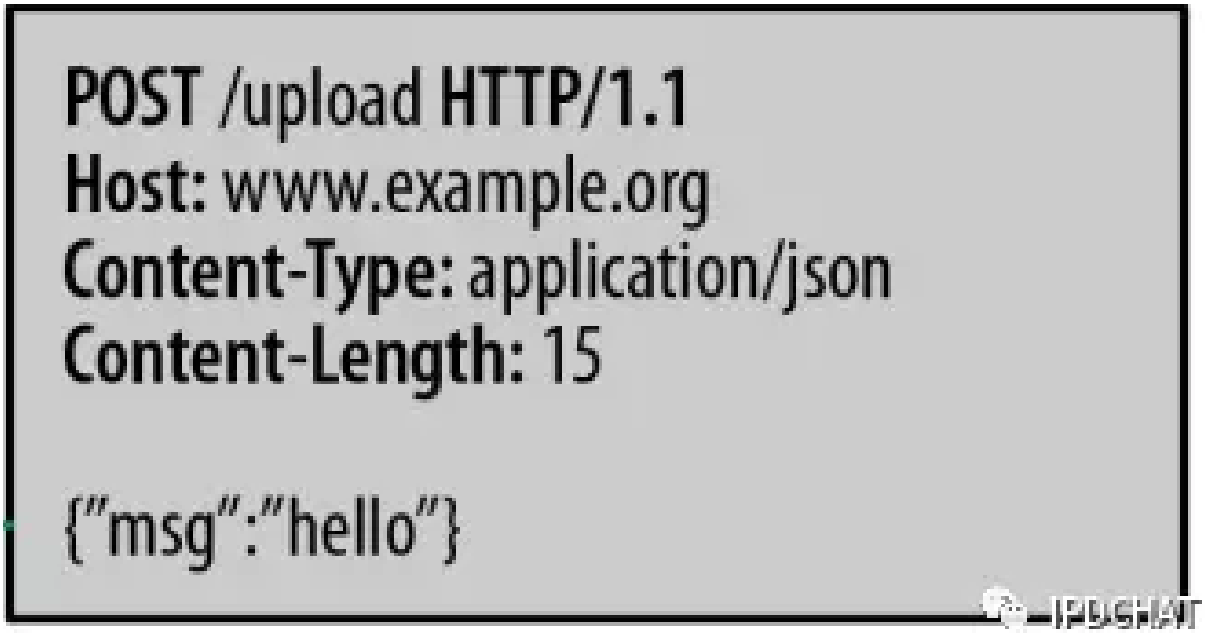
HTTP/2是什么

HTTP/2是下一代的HTTP协议。
起源于 GOOGLE 带头开发的 SPDY 协议，由 IETF 的 HTTPbis 工作组修改发布。

由两个RFC组成：
● RFC 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2)
● RFC 7541 - HPACK: Header Compression for HTTP/2
这两个 RFC 目前的状态是 PROPOSED STANDARD

HTTP/1的主要问题

Head-of-line blocking，新请求的发起必须等待服务器对前一个请求的回应，无法同时发起多个请求，导致很难充分利用TCP连接。

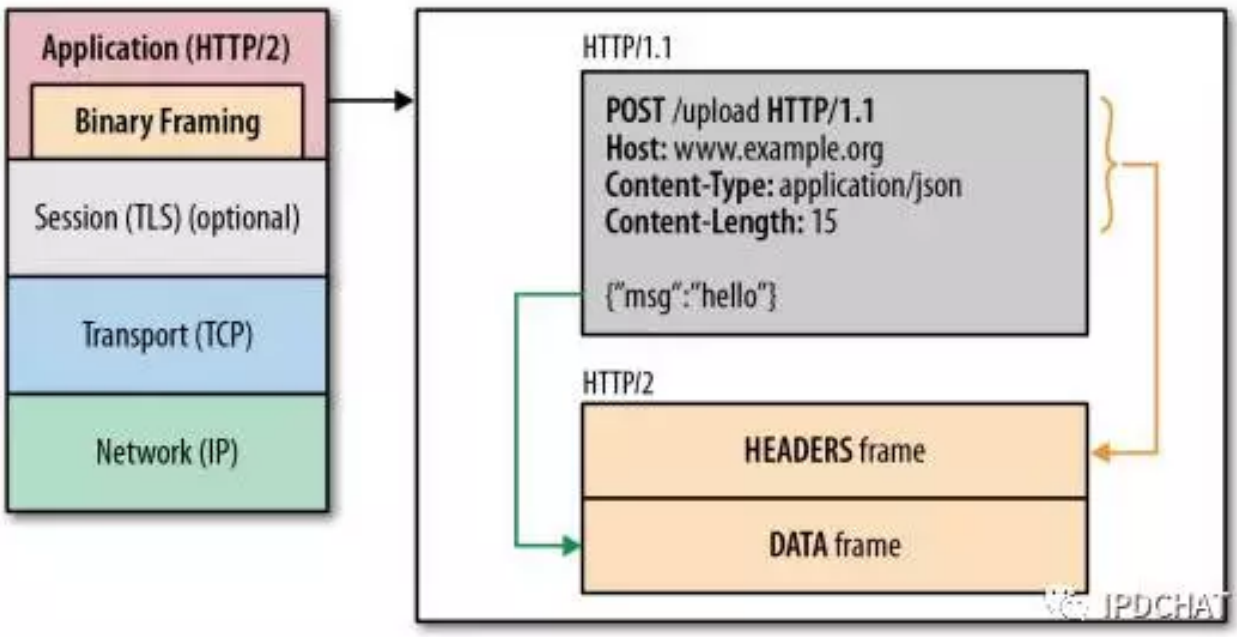


- 头部冗余

HTTP头部包含大量重复数据，比如cookies，多个请求的cookie可能完全一样

HTTP/2改进

- 二进制协议、分帧（Frame）
- 双向流，多路复用
- 头部压缩
- 服务器推送（Server Push）
- 优先级
- 流量控制
- 流重置



HTTP/2-帧

●HTTP/2抛弃HTTP/1的文本协议改为二进制协议。HTTP/2的基本传输单元为帧。每个帧都从属于某个流。

- Length： Payload 长度
- Type： 帧类型
- Stream identifier：流ID
- Frame Payload： 依帧类型而不同

HTTP/2-帧的类型

- HEADERS 对应HTTP/1的 Headers
- DATA 对应HTTP/1的 Body
- CONTINUATION 头部太大，分多个帧传输（一个HEADERS+若干CONTINUATION）
- SETTINGS 连接设置
- WINDOW_UPDATE 流量控制
- PUSH_PROMISE 服务端推送
- PRIORITY 流优先级更改
- PING 心跳或计算RTT
- RST_STREAM 马上中止一个流
- GOAWAY 关闭连接并且发送错误信息

HTTP/2-流

HTTP/2连接上传输的每个帧都关联到一个流，一个连接上可以同时有多个流。同一个流的帧按序传输，不同流的帧交错混合传输。客户端、服务端双方都可以建立流，流也可以被任意一方关闭。客户端发起的流使用奇数流ID，服务端发起的使用偶数。

Protocol Buffers是什么

一个语言无关，平台无关，可扩展的结构化数据序列化方案，用于协议通讯，数据存储和其他更多用途。

一个灵活，高效，自动化的结构化数据序列化机制(想象xml)，但是更小，更快并且更简单，一旦定义好数据如何构造， 就可以使用特殊的生成的源代码来轻易的读写你的结构化数据到和从不同的数据流，用不同的语言。你甚至可以更新你的数据结构而不打破已部署的使用"旧有"格式编译的程序。

为什么使用HTTP协议

将移动和HTTP/2放在首位的通用的RPC框架，

- 网络基础设施设计良好的支持HTTP，比如防火墙, 负载, 加密, 认证, 压缩, ...

gRPC原理-从一个HelloWorld开始

第1步. 定义 hello-dto.proto 文件

```
syntax = "proto3";
  option java_package = "com.jd.jsf.grpc.dto";
```

```
option java_multiple_files = true;
option java_outer_classname = "HelloServiceDTO";
package grpc;
// The request message containing the user's name.
message HelloRequest {
      string name = 1;
}
// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

第2步. 定义hello-service.proto文件（可以和第一步合并）

```
syntax = "proto3";
  import "hello-dto.proto";
  option java_package = "com.jd.jsf.grpc.service";
  option java_multiple_files = true;
  option java_outer_classname = "IHelloService";
  package grpc;
  // The greeting service definition.
  service HelloService {
    // Sends a greeting
    rpc SayHello (grpc.HelloRequest) returns (grpc.HelloReply) {}
  }
```

第3步. 生成 源代码 文件

```
#! /bin/bash
PROTOC3="/grpc/protoc-3.1.0"
PROJECT_HOME="./"
echo "gen dto"
${PROTOC3}/bin/protoc \
  -I=${PROJECT_HOME}/src/main/proto/ \
  --java_out=${PROJECT_HOME}/src/main/java \
  ${PROJECT_HOME}/src/main/proto/hello-dto.proto
echo "gen service"
${PROTOC3}/bin/protoc \
  -I=${PROJECT_HOME}/src/main/proto/ \
  --java_out=${PROJECT_HOME}/src/main/java \
  ${PROJECT_HOME}/src/main/proto/hello-service.proto
echo "gen grpc service"
${PROTOC3}/bin/protoc \
```

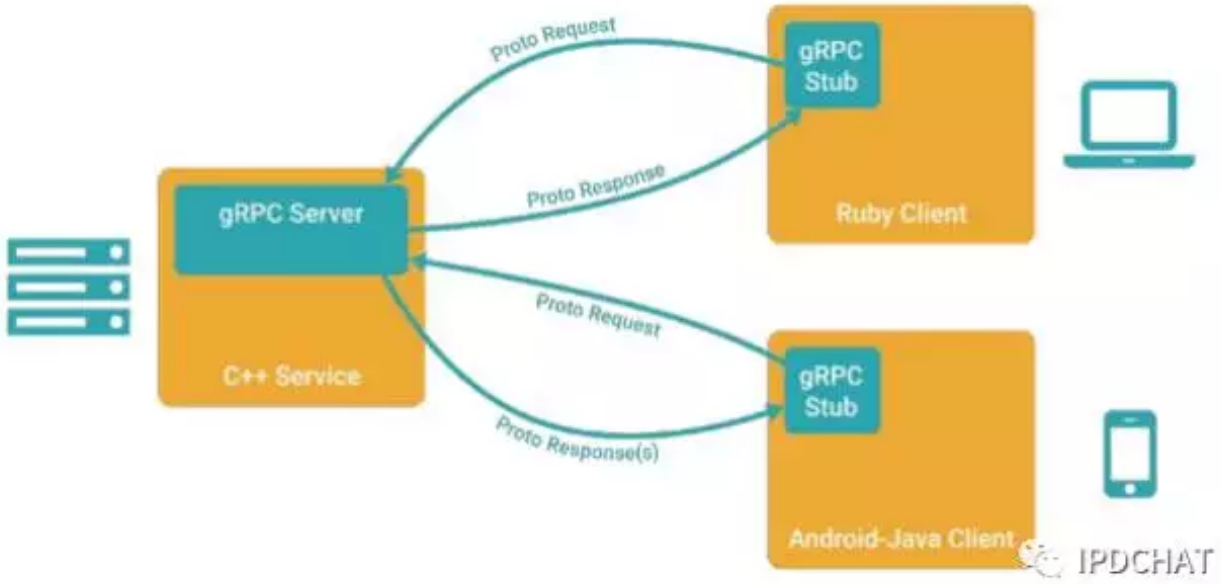
```
    --plugin=protoc-gen-grpc-java=${PROTOC3}/bin/protoc-gen-grpc-java-1.1.1-linux-
x86_64.exe \
    --grpc-java_out=${PROJECT_HOME}/src/main/java \
    -I=${PROJECT_HOME}/src/main/proto/ \
    ${PROJECT_HOME}/src/main/proto/hello-service.proto
echo "over!"
```

第4步. 编写Server端

```
 int port = 50051;
 server = ServerBuilder.forPort(port).addService(new GreeterImpl()).build() .start();
 server.awaitTermination();
 class GreeterImpl extends HelloServiceGrpc.HelloServiceImplBase {
  @Override
  public void sayHello(HelloRequest req, StreamObserver<HelloReply> responseObserver) {
      HelloReply reply = HelloReply.newBuilder().setMessage("Hello " +
                                          req.getName()).build();
      responseObserver.onNext(reply);
      responseObserver.onCompleted();
  }
 }
```

第5步. 编写Client端

```
 ManagedChannel
channel = ManagedChannelBuilder.forAddress(host, port).usePlaintext(true);
HelloServiceGrpc.HelloServiceBlockingStub
            blockingStub = HelloServiceGrpc.newBlockingStub(channel);
HelloServiceGrpc.HelloServiceFutureStub
            futureStubStub = HelloServiceGrpc.newFutureStub(channel);
HelloRequest request = HelloRequest.newBuilder().setName(name).build();
HelloReply response = blockingStub.sayHello(request);
ListenableFuture<HelloReply> future = futureStubStub.sayHello(request);
HelloReply response = future.get();
```



gRPC原理- 概念

从HelloWorld中，映射出gRPC的基本概念

1. Channels

在创建客户端存根时，一个gRPC通道提供一个特定主机和端口服务端的连接。客户端可以通过指定通道参数来修改gRPC的默认行为，比如打开关闭消息压缩。一个通道具有状态，包含已连接和空闲 。

2. Stub

Proxy, Channel, Marshaller, MethodDescriptor

利用代码生成器生成client和server端stub代码，为了跨语言只能这么玩，这也体现了静态语言和动态语言的区别。Stub代码包含了客户端和服务端静态代理类，分别处理消息的加工和发送。还包括序列化方法，服务定义相关的方法描述。

3. Service Def

gRPC 基于如下思想：定义一个服务， 指定其可以被远程调用的方法及其参数和返回类型。gRPC 默认使用 protocol buffers 3 作为接口定义语言.

```
service HelloService {
        rpc SayHello (HelloRequest) returns (HelloResponse);
    }
    message HelloRequest {
        required string greeting = 1;
    }
    message HelloResponse {
        required string reply = 1;
    }
```

gRPC 允许你定义四类服务方法：

1). 单项 RPC，即客户端发送一个请求给服务端，从服务端获取一个应答，就像一次普通的函数调用

```
rpc SayHello(HelloRequest) returns (HelloResponse){}
```

2). 服务端流式 RPC，即客户端发送一个请求给服务端，可获取一个数据流用来读取一系列消息。客户端从返回的数据流里一直读取直到没有更多消息为止。

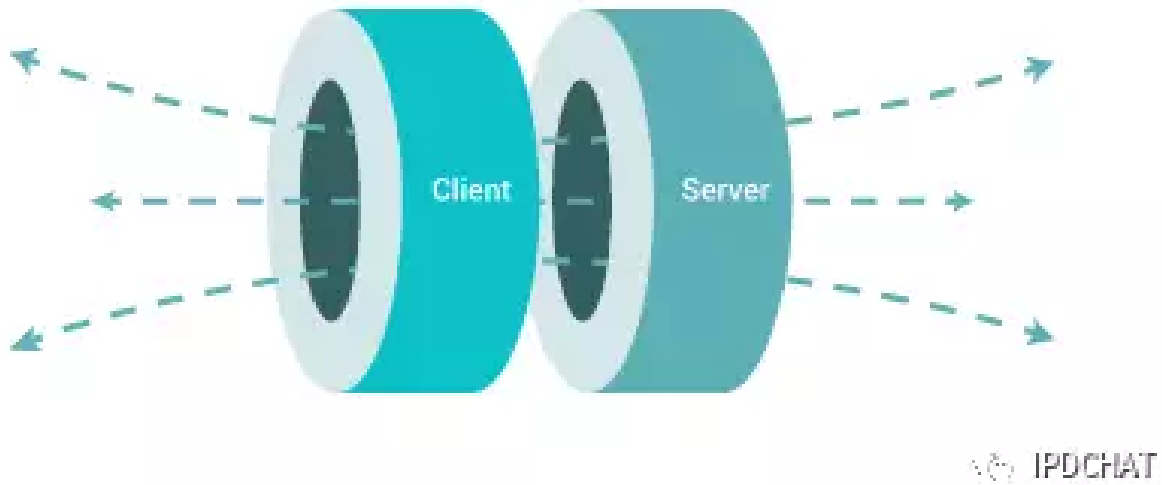
```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse){}
```

3). 客户端流式 RPC，即客户端用提供的一个数据流写入并发送一系列消息给服务端。一旦客户端完成消息写入，就等待服务端读取这些消息并返回应答。

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse) {}
```

4). 双向流式 RPC，即两边都可以分别通过一个读写数据流来发送一系列消息。这两个数据流操作是相互独立的，所以客户端和服务端能按其希望的任意顺序读写，例如：服务端可以在写应答前等待所有的客户端消息，或者它可以先读一个消息再写一个消息，或者是读写相结合的其他方式。每个数据流里消息的顺序会被保持。

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse){}
```



4. DEADLINE

gRPC 允许客户端在调用一个远程方法前指定一个最后期限值。这个值指定了在客户端可以等待服务端多长时间来应答，超过这个时间值 RPC 将结束并返回DEADLINE_EXCEEDED错误。在服务端可以查询这个期限值来看是否一个特定的方法已经过期，或者还剩多长时间来完成这个方法。

5. Metadata

Headers

RPC原理- 协议与实现

gRPC中，把HTTP2的Steam Identifier当作调用标识，每一次请求都发起一个新的流。

每个请求的调用哪个服务和方法、回应的调用结果状态码都在HEADER Frame中指定。

请求内容和回应内容由Protocol Buffer序列化后使用DATA Frame传输

以下是 gRPC 请求和应答消息流中一般的消息顺序：
请求 → 请求报头 * 有定界符的消息 EOS
应答 → 应答报头 * 有定界符的消息 EOS
应答 → (应答报头 * 有定界符的消息 跟踪信息) / 仅仅跟踪时
界定的消息的重复序列通过数据帧来进行传输。
界定的消息 → 压缩标志 消息长度 消息
压缩标志 → 0 / 1 # 编码为 1 byte 的无符号整数
消息长度 → {消息长度} # 编码为 4 byte 的无符号整数
消息 → *{二进制字节}

1. 请求
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = http
:path = /google.pubsub.v2.PublisherService/CreateTopic
:authority = pubsub.googleapis.com
grpc-timeout = 1S
content-type = application/grpc+proto
grpc-encoding = gzip
authorization = Bearer y235.wef315yfh138vh31hv93hv8h3v
DATA (flags = END_STREAM)
<Delimited Message>

2. 应答
HEADERS (flags = END_HEADERS)
:status = 200
grpc-encoding = gzip
DATA
<Delimited Message>
HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0 # OK
trace-proto-bin = jher831yy13JHy3hc

gRPC原理- Server端

gRPC而言,只是对Netty Server的简单封装,底层使用了PlaintextHandler、Http2ConnectionHandler的相关封装等。具体Framer、Stream方式请参考Http2相关文档。

followControlWindow：
流量控制的窗口大小，单位：字节，默认值为1M，HTTP2中的“Flow Control”特性；连接上，已经发送尚未ACK的数据帧大小，比如window大小为100K，且winow已满，每次向Client发送消息时，如果客户端反馈ACK（携带此次ACK数据的大小），window将会减掉此大小；每次向window中添加亟待发送的数据时，window增加；如果window中的数据已达到限定值，它将不能继续添加数据，只能等待Client端ACK。

maxConcurrentCallPerConnection：
每个connection允许的最大并发请求数，默认值为Integer.MAX_VALUE;如果此连接上已经接受但尚未响应的streams个数达到此值，新的请求将会被拒绝。为了避免TCP通道的过度拥堵，我们可以适度调整此值，以便Server端平稳处理，毕竟buffer太多的streams会对server的内存造成巨大压力。

maxMessageSize：每次调用允许发送的最大数据量，默认为100M。

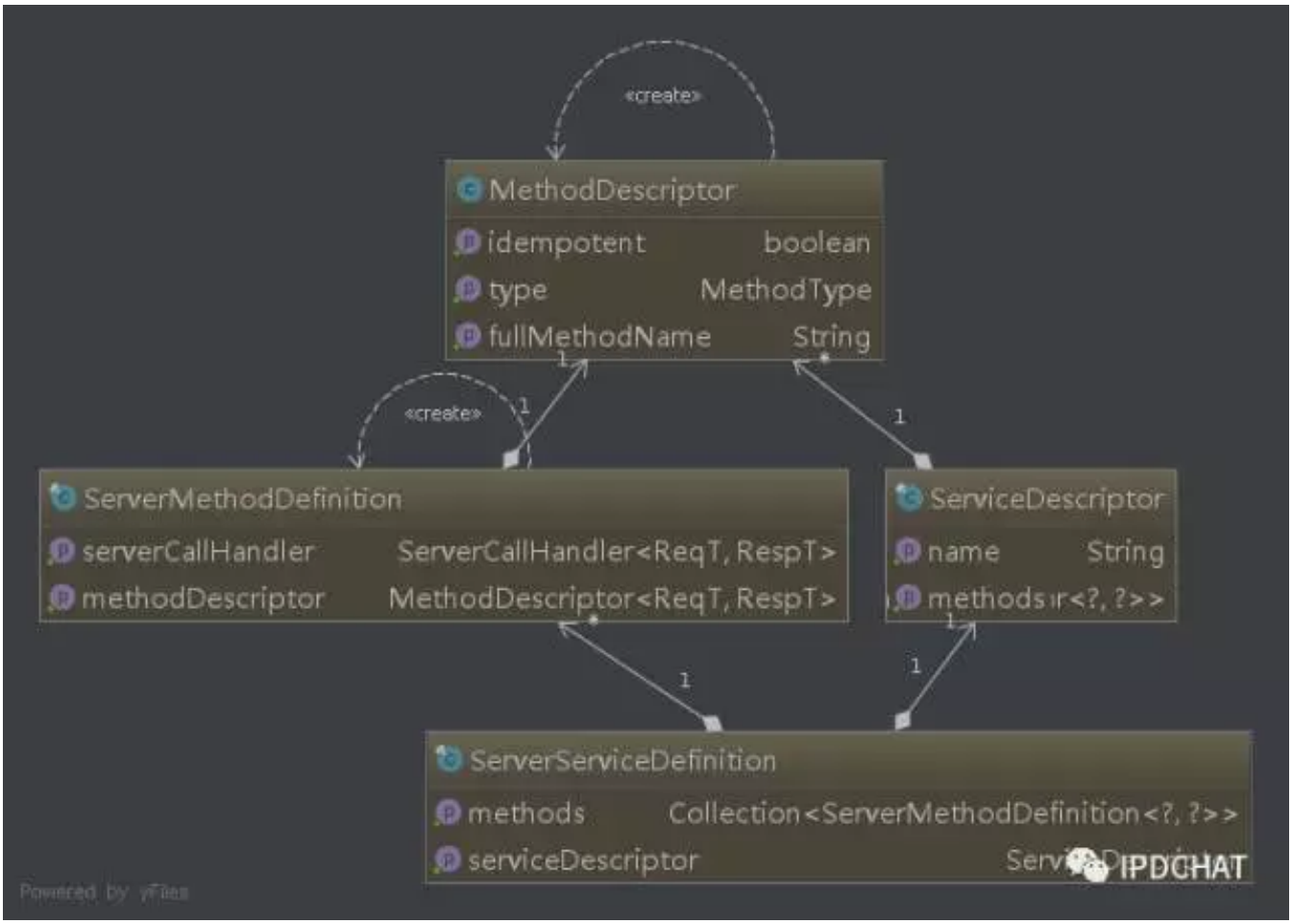
maxHeaderListSize：每次调用允许发送的header的最大条数，gRPC中默认为8192。

gRPC Server端，有个重要的方法：addService。【如下文service代理模式】

在此之前，我们需要介绍一下bindService方法，每个gRPC生成的service代码中都有此方法，它以硬编码的方式遍历此service的方法列表，将每个方法的调用过程都与“被代理实例”绑定，这个模式有点类似于静态代理，比如调用sayHello方法时，其实内部直接调用“被代理实例”的sayHello方法（参见MethodHandler.invoke方法，每个方法都有一个唯一的index，通过硬编码方式执行）；bindService方法的最终目的是创建一个ServerServiceDefinition对象，这个对象内部位置一个map，key为此Service的方法的全名（fullname，{package}.{service}.{method}）,value就是此方法的gRPC封装类（ServerMethodDefinition）

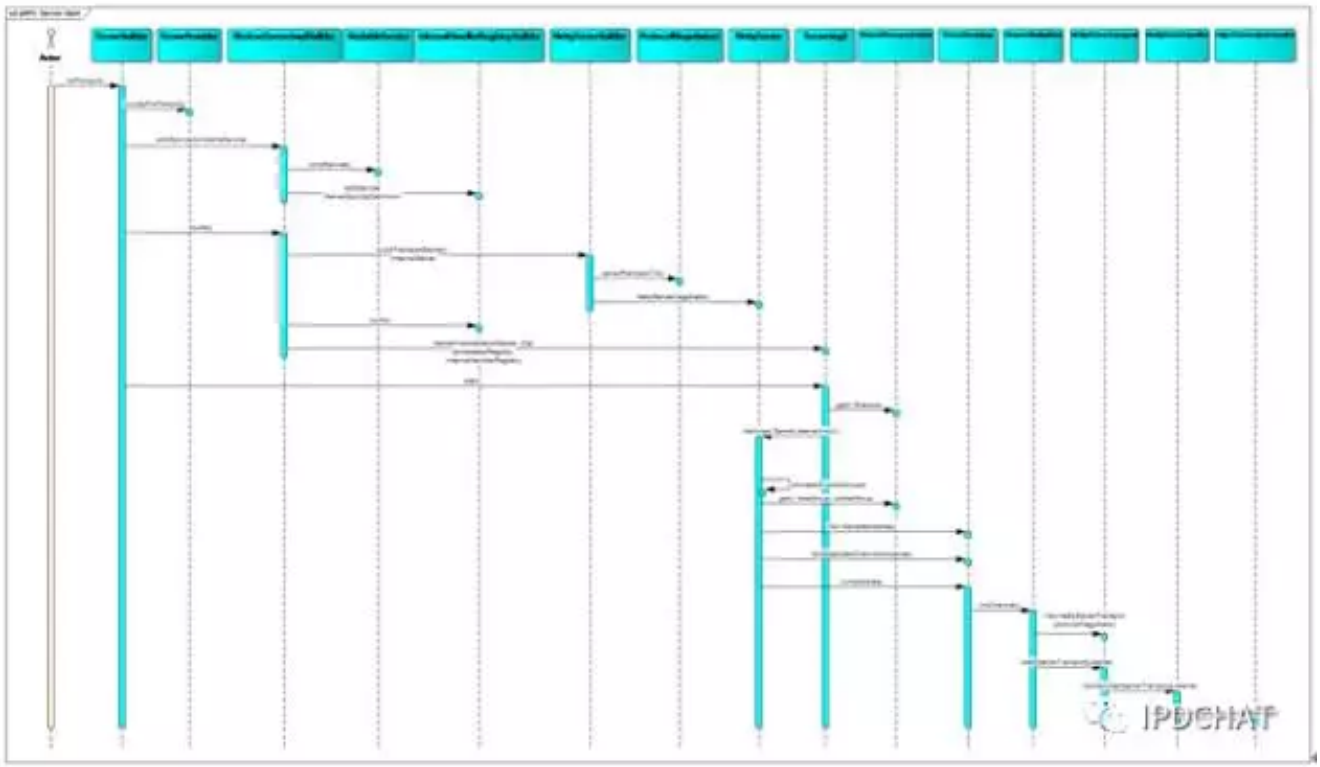
addService方法可以添加多个Service，即一个Netty Server可以为多个service服务，这并不违背设计模式和架构模式。addService方法将会把service保存在内部的一个map中，key为serviceName（即{package}.{service}）,value就是上述bindService生成的对象。

如下是服务定义的类结构：



那么究竟Server端是如何解析RPC过程的？Client在调用时会将调用的service名称 + method信息保存在一个GRPC“保留”的header中，那么Server端即可通过获取这个特定的header信息，就可以得知此stream需要请求的service、以及其method，那么接下来只需要从上述提到的map中找到service，然后找到此method，直接代理调用即可。执行结果在Encoder之后发送给Client。

如下是Server端启动过程：



gRPC原理- Client端

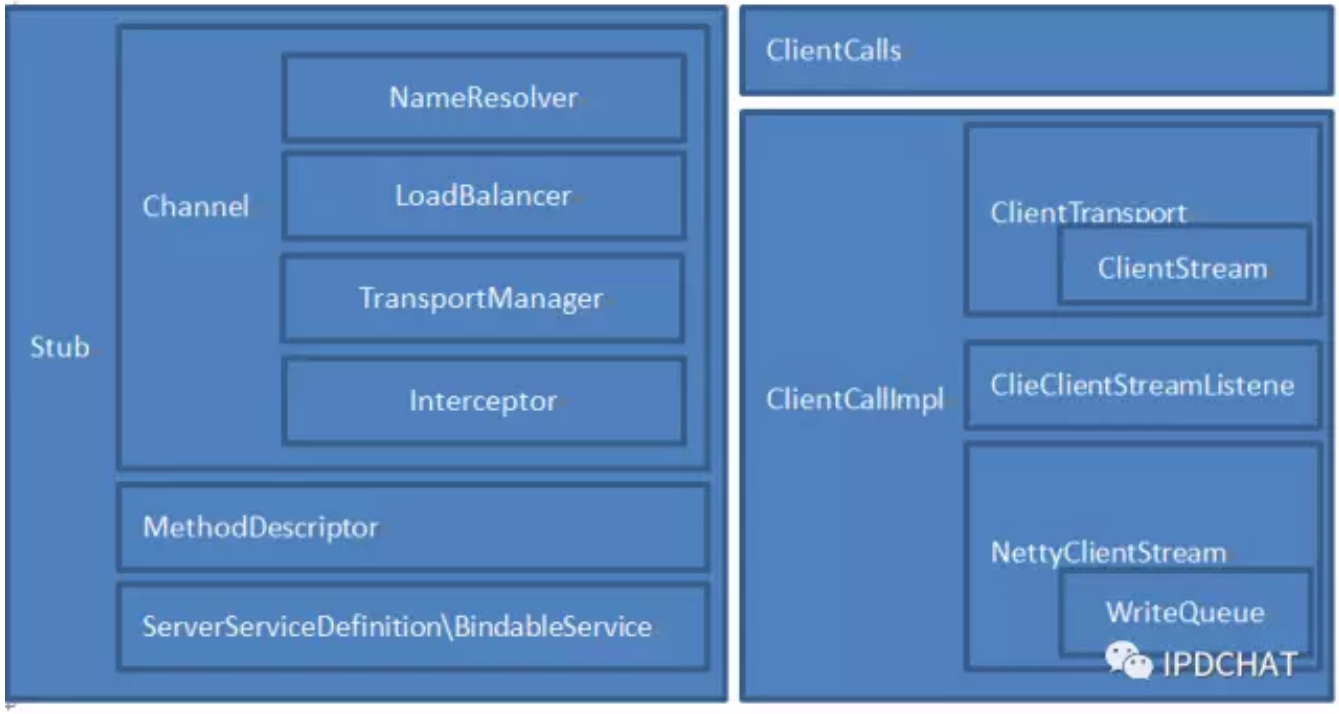
ManagedChannelBuilder来创建客户端channel，ManagedChannelBuilder使用了provider机制，具体是创建了哪种channel有provider决定，可以参看META-INF下同类名的文件中的注册信息。当前Channel有2种：NettyChannelBuilder与OkHttpChannelBuilder。当前版本中为NettyChannelBuilder；可以直接使用NettyChannelBuilder来构建channel。

ManagedChannel是客户端最核心的类，它表示逻辑上的一个channel；底层持有一个物理的transport（TCP通道，参见NettyClientTransport），并负责维护此transport的活性；即在RPC调用的任何时机，如果检测到底层transport处于关闭状态（terminated），将会尝试重建transport。（参见TransportSet.obtainActiveTransport()）

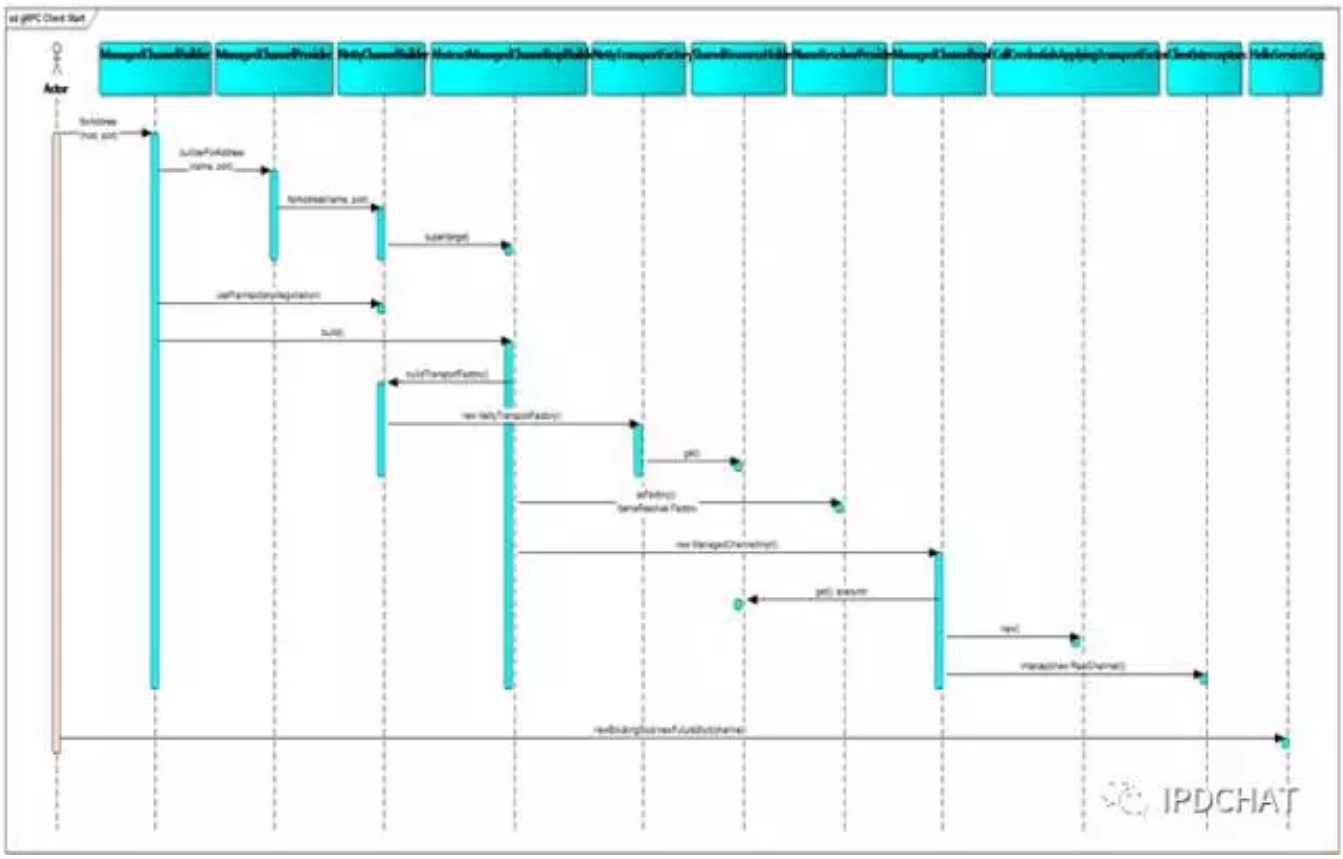
通常情况下，我们不需要在RPC调用结束后就关闭Channel，Channel可以被一直重用，直到Client不再需要请求为止或者Channel无法真的异常中断而无法继续使用。

每个Service客户端，都生成了2种stub：BlockingStub和FutureStub；这两个Stub内部调用过程几乎一样，唯一不同的是BlockingStub的方法直接返回Response, 而FutureStub返回一个Future对象。BlockingStub内部也是基于Future机制，只是封装了阻塞等待的过程。

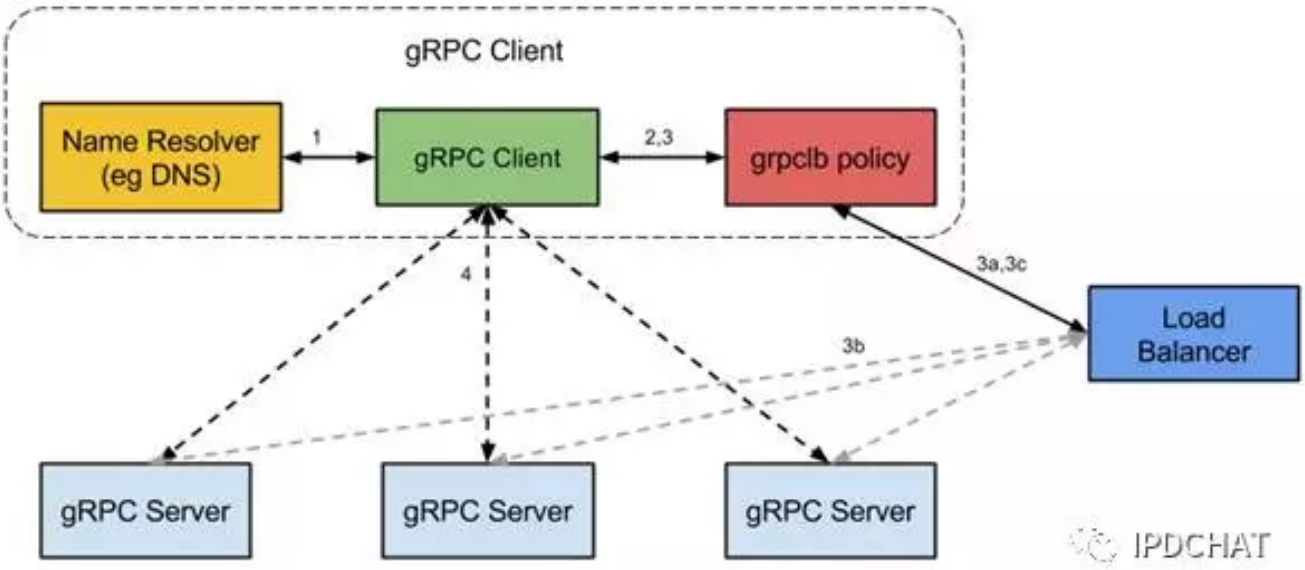
如下是Client端关键组件：



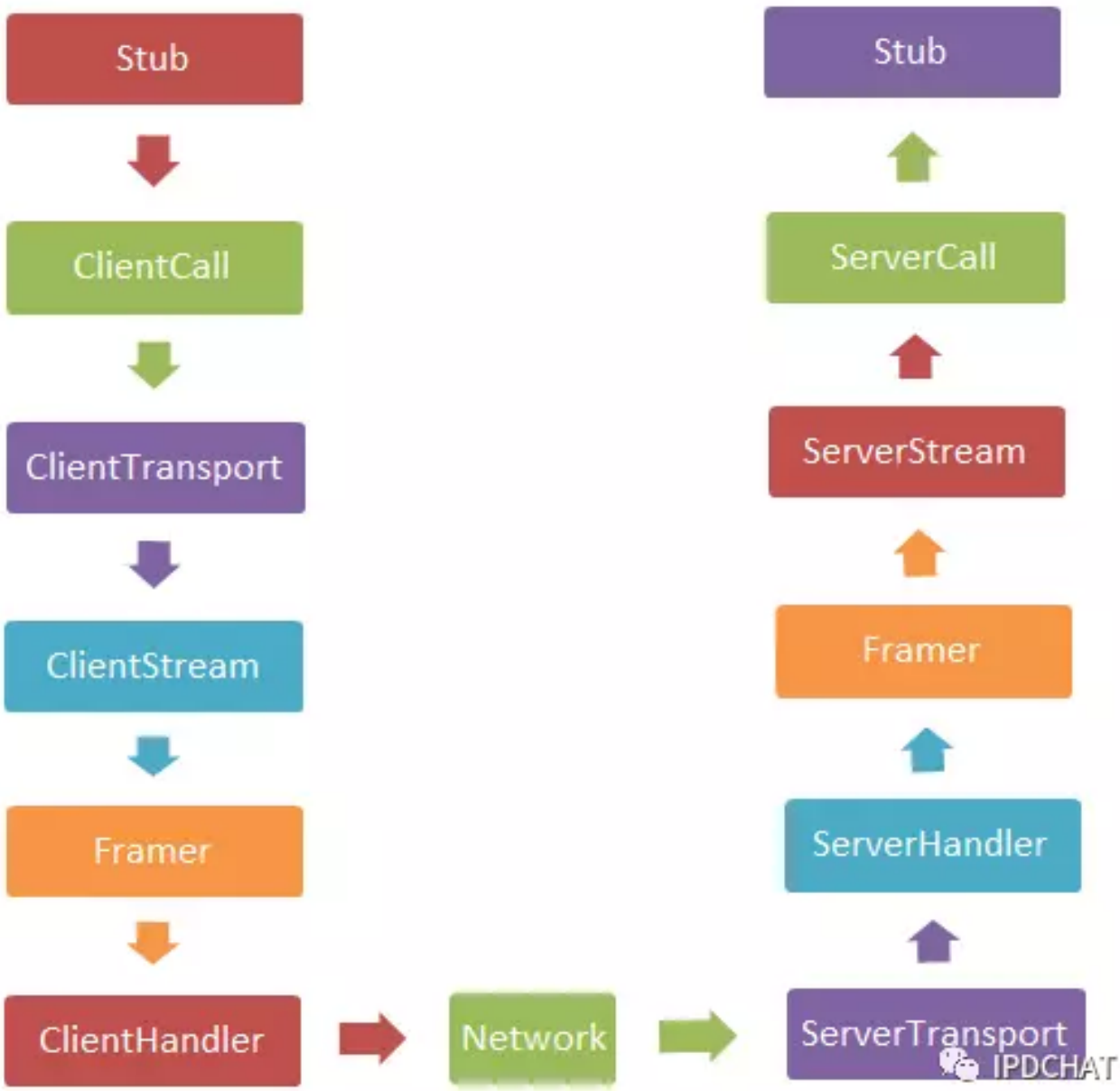
如下是Client端的启动流程:



关于Client负载均衡

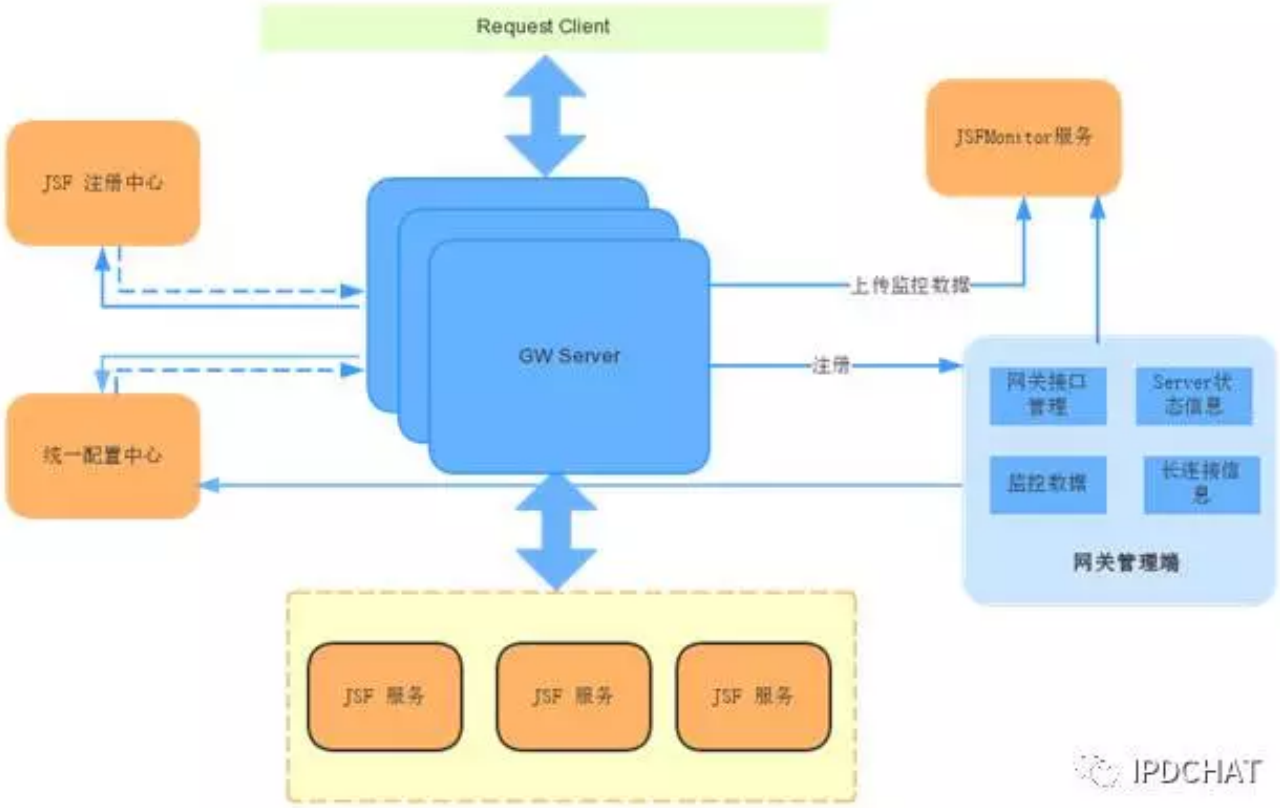

gRPC Client
Name Resolver
(eg DNS)
1
gRPC Client
2,3
grpclb policy
4
3a,3c
Load
Balancer
3b
gRPC Server
gRPC Server
gRPC Server
IPDCHAT

gRPC分层设计


Stub
ClientCall
ClientTransport
ClientStream
Framer
ClientHandler
Network
Stub
ServerCall
ServerStream
Framer
ServerHandler
ServerTransport
IPDCHAT

JSF兼容gRPC

目前JSF支持Java和C++两种客户端，其他小众语言无法支持，为了解决跨语言问题，JSF系统增加了基于HTTP/1的网关服务。这可能是目前业内RPC框架解决跨语言问题的普遍解决方案。



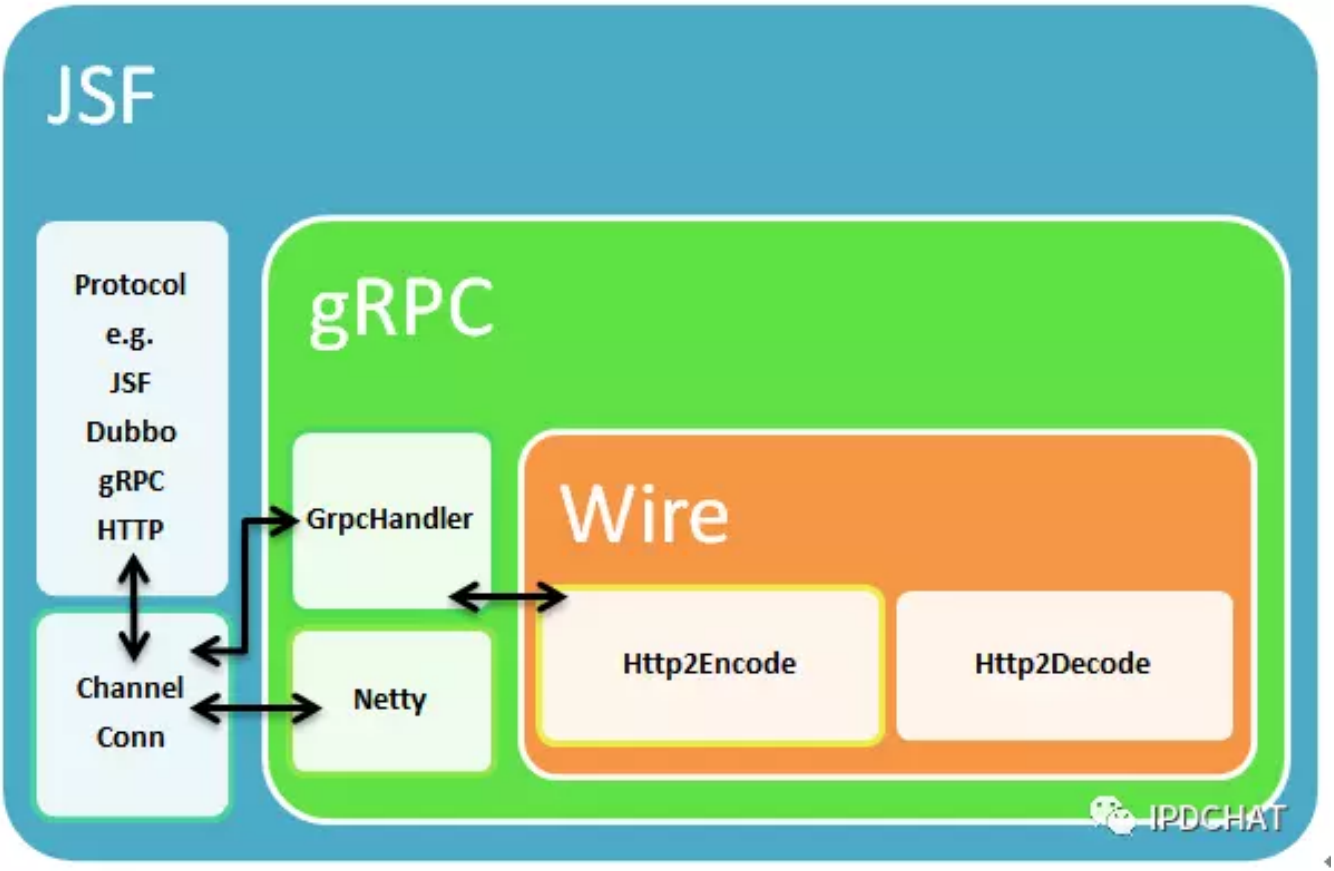
针对gRPC的技术预言，就是为了解决JSF跨语言问题，如何解决？目前JSF框架发布的JSF协议服务，天然支持JSF、HTTP、Dubbo、Telnet协议。这都得益于Netty的伟大。就Netty而言，客户端与服务端建立TCP连接后，初始化Channel时，可以根据报文头的特征码进行协议匹配，进而针对当前连接设置相应协议的解码器。就gRPC而言，其报文头就是HTTP/2的报文头-棱镜。

```
▶ Secure Sockets Layer
▼ HyperText Transfer Protocol 2
  ▼ Stream: Magic
      Magic: PRI * HTTP/2.0\r\n\r\nS
```

针对JSF服务提供端，解析gRPC协议报文，获取接口、方法、参数，然后进行方法调用，最后模拟gRPC协议返回给客户端。

针对JSF服务调用端，模拟gRPC协议，发送gRPC协议报文。

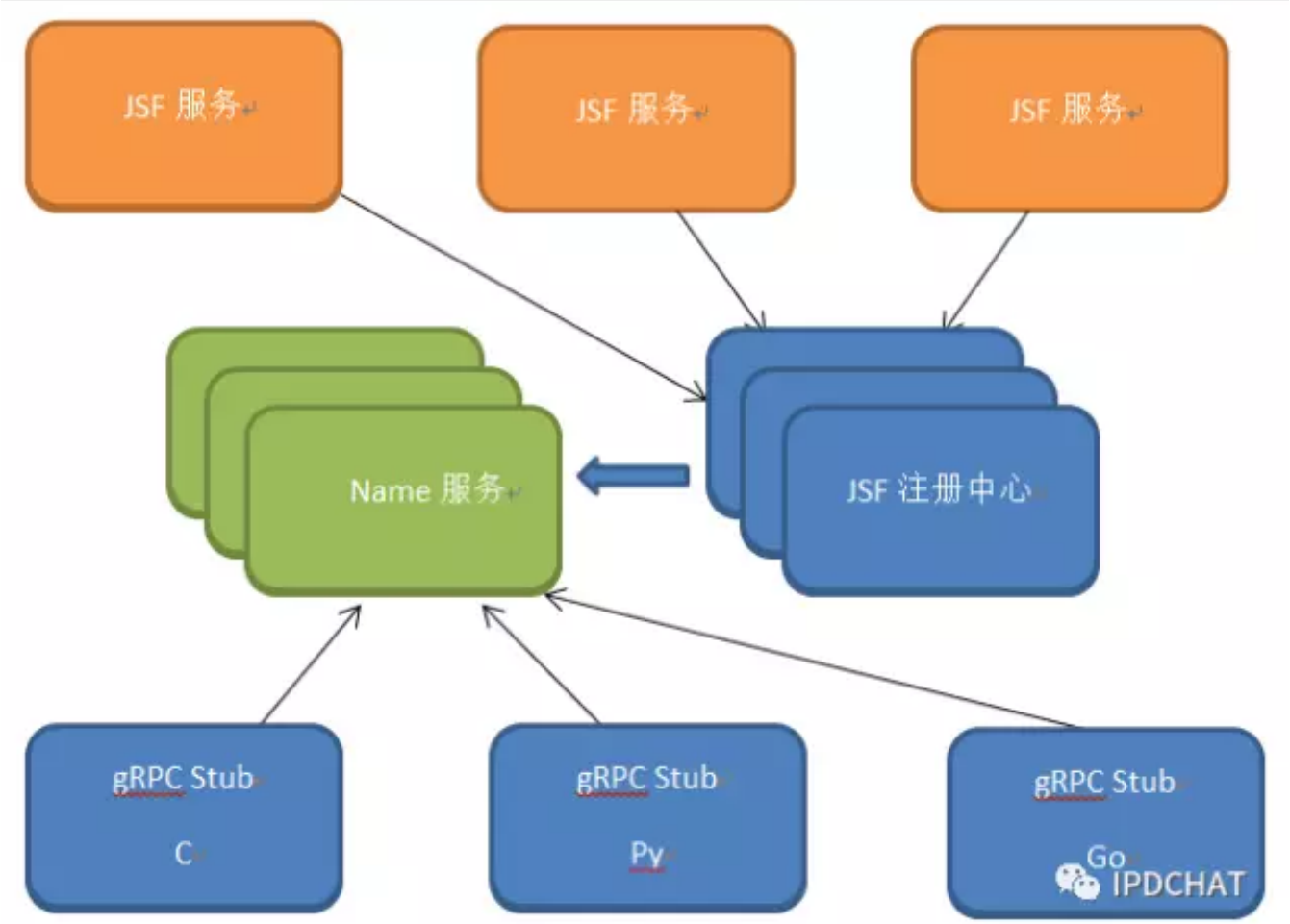
JSF兼容gRPC如下图：



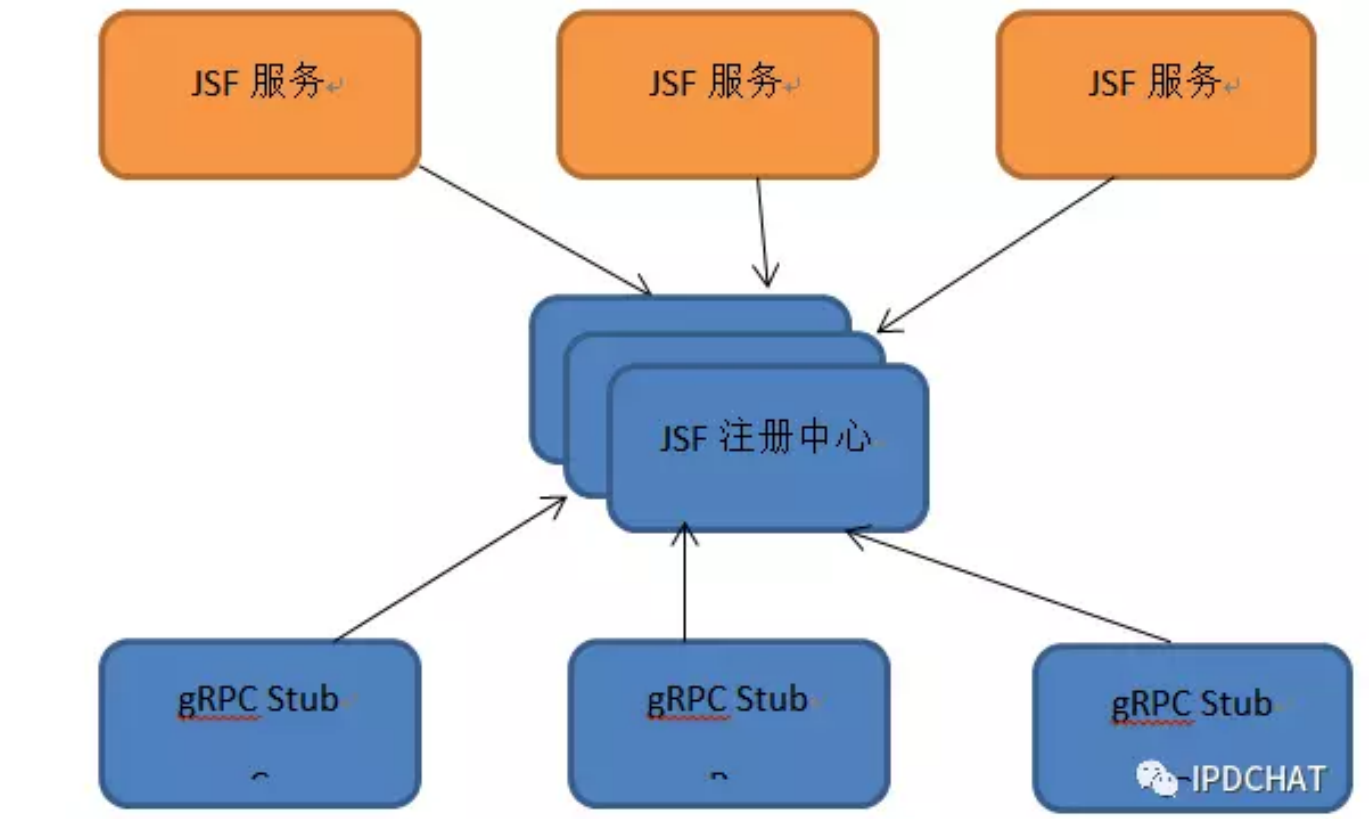
至此，JSF跨语言问题解决了。NO！目前gRPC各种语言客户端可以访问Java版的JSF服务，Java版的JSF客户端也可以访问gRPC各种语言服务端。我们要解决的问题是Java版的JSF服务，可以让其他gRPC各种语言客户端访问，目前仅解决了一小步。gRPC各种语言客户端不具备JSF的服务订阅功能，只能借道gRPC自身的负载策略DNS。

默认gRPC通过本地的域名解析，拉取服务列表，进而负载均衡。为了支持这种策略，Java版的JSF服务注册服务时，需要将信息同步注册到DNS服务，其他gRPC各种语言客户端访问DNS服务实现服务发现。这里需要按照服务申请域名，这是个弊端。这也违背了gRPC移动端为主、跨数据中心访问的初衷。

采用DNS服务发现的设计如下图：



为了解决DNS服务发现带来的系统复杂度，正能对gRPC进行动刀，由于gRPC的扩展性良好，而且只需要将C、Go语言的客户端进行扩展即可。gRPC服务发现的机制是通过NameResolver来解决的，而且是基于Plugin方式，故NameResolver的实现目标指向JSF系统现有的注册中心即可，同时为了更彻底的改变gRPC服务注册、订阅，又将C、Go语言的客户端增加了服务注册、订阅功能。至此，Java版的JSF服务与gRPC版的服务之间相互调用打通了。



gRPC总结

跨语言，针对移动端：省电、省流量、高性能、双向流、支持DNS负载。关于性能，肯定比HTTP/1好，比TCP差，网上好多性能对比，都是和TCP相关的RPC对比，没有可比性。

本文转载自IPD-Chat，IPD-Chat为京东商城基础平台部门官方公众号，扫一扫二维码进行关注。

==========友情推荐==========