## 一．概念普及：

**protocol buffers（protobuf）：**结构数据序列化机制

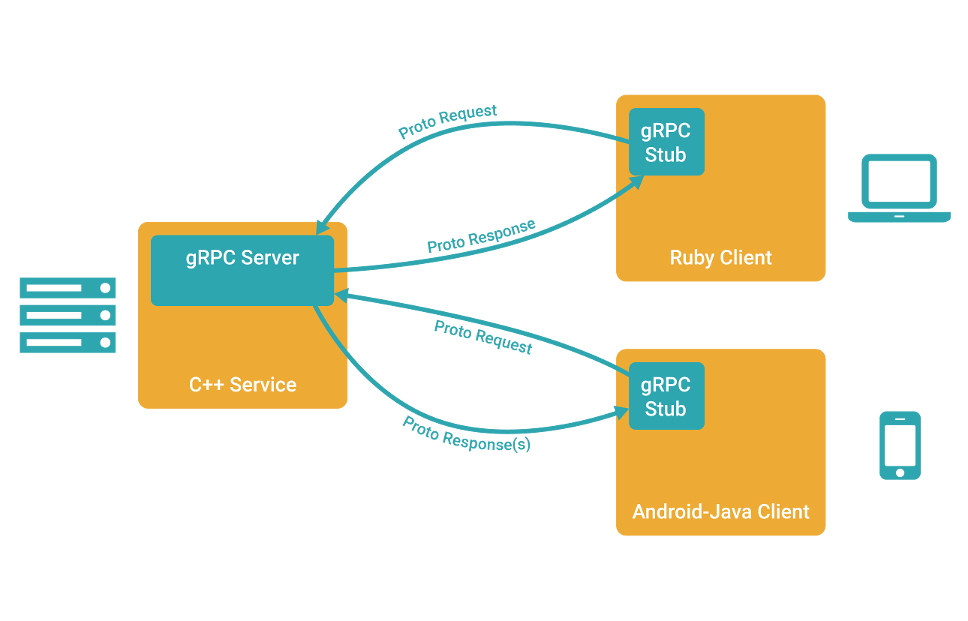
主要思想是:定义一个服务, 指定其可以被远程调用的方法及其参数和返回类型。用来实现不同进程间的通信。

gRPC 默认使用 protocol buffers，用 proto files 创建 gRPC 服务，用 protocol buffers 消息类型来定义方法参数和返回类型。

**GRPC :**

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

基于以下理念：定义一个服务，指定其能够被远程调用的方法（包含参数和返回类型）。在服务端实现这个接口，并运行一个 GRPC 服务器来处理客户端调用。在客户端拥有一个存根能够像服务端一样的方法。



**Stub存根：**

为屏蔽客户调用远程主机上的对象，必须提供某种方式来模拟本地对象,这种本地对象称为存根(stub),存根负责接收本地方法调用,并将它们委派给各自的具体实现对象。

## 二．GRPC JAVA demo实现

针对你选择的语言构建和安装 gRPC 插件和相关工具。 Java gRPC 除了 JDK 外不需要其他工具。

1. 在一个.proto文件内定义服务
2. 用protocol buffer编译器生成指定编程语言的服务器和客户端代码（maven compile + deploy）
3. 使用grpc的对应的语言（grpc-java）的API，实现服务端和客户端的进程间的通信。

**1.protobuf使用包括以下几个步骤**：

1. 定义数据结构。
2. 编译成你喜欢的开发语言的相关工具类。
3. 在程序中调用工具类来处理数据。

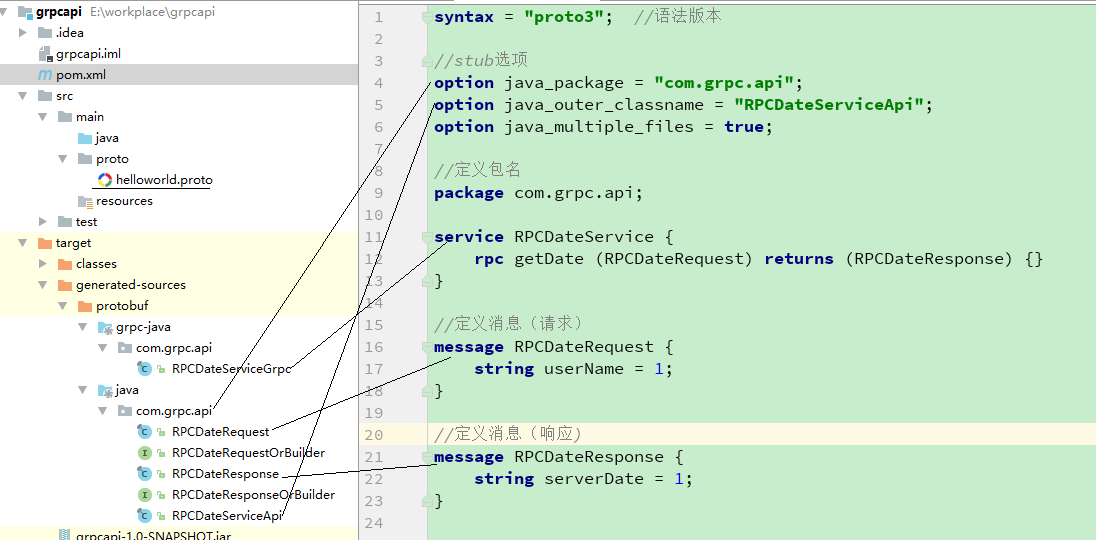
一旦定义好服务，可以使用 protocol buffer 编译器 protoc 来生成创建应用所需的特定客户端和服务端的代码，生成的代码同时包括客户端的存根和服务端要实现的抽象接口。

需要支持：Idea 插件 protobuf support

通过：Maven compile+deploy

生成java版本的数据结构类：RPCDateRequest、RPCResponse的java类，我们在服务端、客户端直接使用即可。

RPCDateRequest.java， RPCResponse.java和其他文件包含所有 protocol buffer 用来填充、序列化和提取 RPCDateRequest和 RPCResponse消息类型的代码。



RPCDateServiceGrpc.java：包含 (还有其他有用的代码)调用RPCDateService服务，服务端需要实现的抽象类

**public static abstract class** RPCDateServiceImplBase **implements** io.grpc.BindableService {  
**public void** getDate(com.grpc.api.RPCDateRequest request,  
 io.grpc.stub.StreamObserver<com.grpc.api.RPCDateResponse> responseObserver) {  
 *asyncUnimplementedUnaryCall*(***METHOD\_GET\_DATE***, responseObserver);  
 }

客户端用来与 RPCDateService服务端进行对话的存根类。

**public static final class** RPCDateServiceBlockingStub **extends** io.grpc.stub.AbstractStub<RPCDateServiceBlockingStub> {  
 **public** com.grpc.api.RPCDateResponse getDate(com.grpc.api.RPCDateRequest request) {  
 **return** *blockingUnaryCall*(  
 getChannel(), ***METHOD\_GET\_DATE***, getCallOptions(), request);  
 }  
}

**服务端的主要工作**：

1. 实现定义的服务接口函数
2. 开启服务端，监听来自客户端的请求并响应客户端的请求。

服务实现

[RPCDateServiceImpl.java](http://doc.oschina.net/https%EF%BC%9A/github.com/grpc/grpc-java/blob/master/examples/src/main/java/io/grpc/examples/helloworld/HelloWorldServer.java#L51) 继承自RPCDateServiceGrpc.RPCDateServiceImplBase，实现其定义的getDate方法。。

**public void** getDate(RPCDateRequest request, StreamObserver<RPCDateResponse> responseObserver) {  
 RPCDateResponse rpcDateResponse = **null**;  
 Date now = **new** Date();  
 SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat(**"今天是"**+**"yyyy年MM月dd日 E kk点mm分"**);  
 String nowTime = simpleDateFormat.format(now);  
 **try** {  
 rpcDateResponse = RPCDateResponse  
 .*newBuilder*()  
 .setServerDate(**"hello "** + request.getUserName() + **", "** + nowTime)  
 .build();  
 } **catch** (Exception e) {  
 responseObserver.onError(e);  
 } **finally** {  
 responseObserver.onNext(rpcDateResponse);  
 }  
 responseObserver.onCompleted();  
}

getDate 有两个参数：

* + RPCDateRequest：请求。
  + StreamObserver<RPCDateResponse>： 应答观察者，一个特殊的接口，服务器用应答来调用它。

为了返回给客户端应答并且完成调用：

1.进行消息构建并填充一个在我们接口定义的 RPCDateResponse 应答对象。

2.将 RPCDateResponse 返回给客户端，然后表明我们已经完成了对 RPC 的处理。

**public class** GRPCServer {  
 **private static final int *port*** = 9999;  
 **public static void** main(String[] args) **throws** Exception {  
 Server server = ServerBuilder  
 .*forPort*(***port***) *//设置服务端口* .addService(**new** RPCDateServiceImpl()) *//添加服务* .build().start();  
 System.***out***.print(**"grpc服务端启动成功，端口="** + ***port***);  
 server.awaitTermination();  
 }  
}

**客户端的主要工作：**

1. 创建Client实例并连接server
2. 调用服务端方法（请求）并获取服务端回应。

连接服务

需要创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口。然后我们用这个频道创建存根实例

调用RPC

1. 我们创建并填充一个 RPCDateRequest 发送给服务。
2. 我们用请求调用存根的 getDate()，如果 RPC 成功，会得到一个填充的 RPCDateResponse ，从其中我们可以获得 响应数据。

**public class** GRPCClient {  
 **private static final** String ***host*** = **"localhost"**;  
 **private static final int *serverPort*** = 9999;  
  
 **public static void** main(String[] args) **throws** Exception {  
 *//根据端口和IP连接服务端，创建grpc channel* ManagedChannel managedChannel = ManagedChannelBuilder.*forAddress*(***host***, ***serverPort***).usePlaintext().build();  
 **try** {  
 *//创建存根，存根是根据.proto文件中生成的类RPCDateServiceGrpc的代理。  
 //blockingStub为阻塞式，需要阻塞等待服务端的回应。 另有asyncStub为非阻塞，可以异步执行。* RPCDateServiceGrpc.RPCDateServiceBlockingStub rpcDateService = RPCDateServiceGrpc.*newBlockingStub*(managedChannel);  
 *//构造请求对象* RPCDateRequest rpcDateRequest = RPCDateRequest  
 .*newBuilder*()  
 .setUserName(**"ChaiShuai"**)  
 .build();  
 RPCDateResponse rpcDateResponse = rpcDateService.getDate(rpcDateRequest);  
 System.***out***.println(rpcDateResponse.getServerDate());  
 } **finally** {  
 managedChannel.shutdown();  
 }  
 }  
}

## 三．FAQ

在建立.proto文件时，通过new>file，命名为.proto格式创建。添加了protobuf support插件后，idea自动识别，前面的显示图标也变了。

通过maven compile后，相应的Java类会生成到target>generated-sources>java和grpc-java+ java\_package（自主命名目录）下。

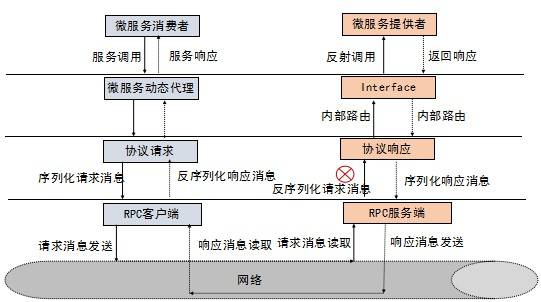
在服务端和客户端需要使用通过.proto定义的服务，要添加相应的maven依赖，这时由.proto生成的java类要进行发布（maven deploy）.

## 四．GRPC原理学习

**RPC是什么**

<https://blog.csdn.net/mindfloating/article/details/39474123#comments>

 通用RPC调用流程



**GRPC**

gRPC 是一个高性能、开源和通用的 RPC 框架，面向服务端和移动端，基于 HTTP/2 设计

语言特点：

语言中立，支持多种语言。

基于 IDL 文件定义服务，通过 proto3 工具生成指定语言的数据结构、服务端接口以及客户端 Stub。

通信协议基于标准的 HTTP/2 设计，支持双向流、消息头压缩、单 TCP 的多路复用、服务端推送等特性，这些特性使得 gRPC 在移动端设备上更加省电和节省网络流量。

序列化支持 PB（Protocol buffer）和 JSON，PB 是一种语言无关的高性能序列化框架，基于 HTTP/2 + PB, 保障了 RPC 调用的高性能。

**gRPC 服务端创建和调用原理解析：**

<https://www.sohu.com/a/159931611_355140>

**一．gRPC 服务端创建关键流程分析**：

a.NettyServer 实例创建：gRPC 服务端创建，首先需要初始化 NettyServer，它是 gRPC 基于 Netty4.1 HTTP/2 协议栈之上封装的 HTTP/2 服务端。NettyServer 实例由 NettyServerBuilder 的 buildTransportServer 方法构建。NettyServer 构建完成之后，监听指定的 Socket 地址，即可实现基于 HTTP/2 协议的请求消息接入。

b.绑定 IDL 定义的服务接口实现类：gRPC 与其它一些 RPC 框架的差异点是服务接口实现类的调用并不是通过动态代理和反射机制，而是通过 proto 工具生成代码。在服务端启动时，将服务接口实现类实例注册到 gRPC 内部的服务注册中心上。请求消息接入之后，可以根据服务名和方法名，直接调用启动时注册的服务实例，而不需要通过反射的方式进行调用，性能更优。

c.gRPC 服务实例（ServerImpl）构建：ServerImpl 负责整个 gRPC 服务端消息的调度和处理，创建 ServerImpl 实例过程中，会对服务端依赖的对象进行初始化。例如 Netty 的线程池资源、gRPC 的线程池、内部的服务注册类（InternalHandlerRegistry）等。ServerImpl 初始化完成之后，就可以调用 NettyServer 的 start 方法启动 HTTP/2 服务端，接收 gRPC 客户端的服务调用请求。

**二.服务端 service 调用流程**

gRPC 的客户端请求消息由 Netty Http2ConnectionHandler 接入，由 gRPC 负责将 PB 消息（或者 JSON）反序列化为 POJO 对象，然后通过服务定义查询到该消息对应的接口实例，发起本地 Java 接口调用，调用完成之后，将响应消息反序列化为 PB(或者 JSON)，通过 HTTP2 Frame 发送给客户端。

流程并不复杂，但是细节却比较多，整个 service 调用可以划分为如下四个过程：

gRPC 请求消息接入

gRPC 消息头和消息体处理

内部的服务路由和调用

响应消息发送

GRPC Server端，还有一个最终要的方法：addService。

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）。

**三．Protobuf 上的 GRPC**

用 protobuf 定义的服务接口可以通过 protoc 的代码生成扩展简单地映射成 GRPC ，以下定义了所用的映射：

路径 → / 服务名 / {方法名}

服务名 → ?( {proto 包名} "." ) {服务名}

消息类型 → {全路径 proto 消息名}

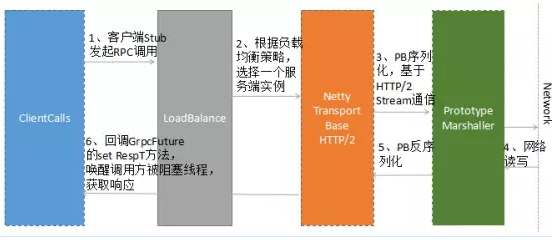
内容类型 → "application/grpc+proto"

**gRPC客户端创建和调用原理解析**

<https://blog.csdn.net/omnispace/article/details/80167076> （内容好多）

gRPC的客户端调用主要包括基于Netty的HTTP/2客户端创建、客户端负载均衡、请求消息的发送和响应接收处理四个流程。

gRPC的客户端调用总体流程如下图所示



参考文章：<https://mp.weixin.qq.com/s/w6hHkme-JwuDgEv95XLptA>

<http://shift-alt-ctrl.iteye.com/blog/2292862>

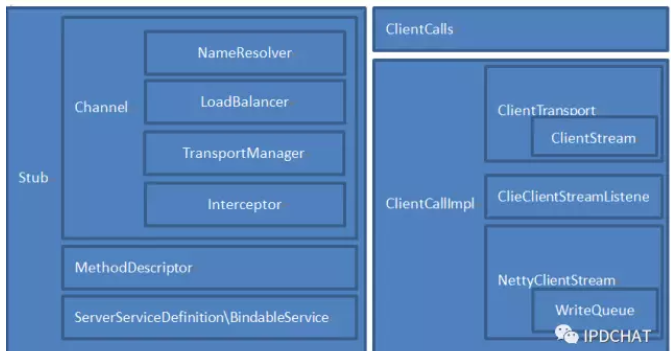
ManagedChannelBuilder来创建客户端channel，ManagedChannelBuilder使用了provider机制，具体是创建了哪种channel有provider决定，可以参看META-INF下同类名的文件中的注册信息。当前Channel有2种：NettyChannelBuilder与OkHttpChannelBuilder。当前版本中为NettyChannelBuilder；可以直接使用NettyChannelBuilder来构建channel。

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()）

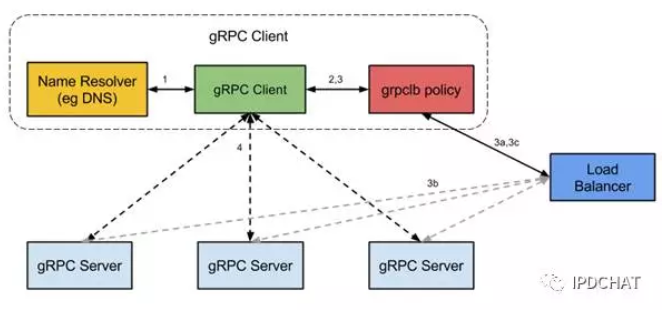
通常情况下，我们不需要在RPC调用结束后就关闭Channel，Channel可以被一直重用，直到Client不再需要请求为止或者Channel无法真的异常中断而无法继续使用。

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。

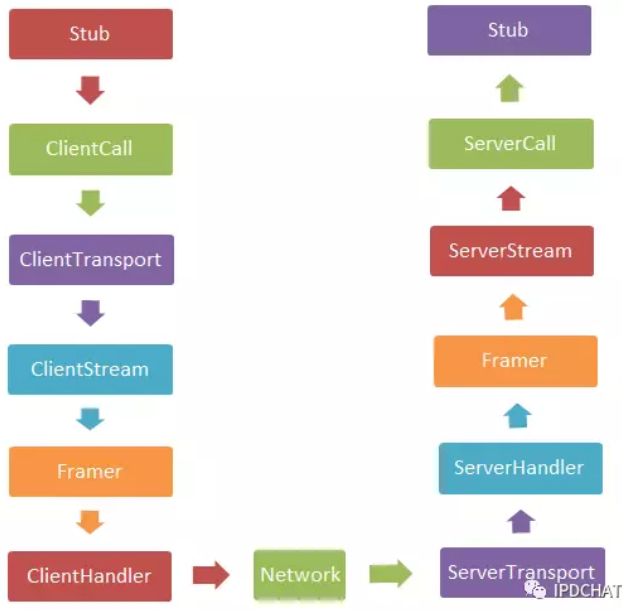
Client端关键组件：



关于Client负载均衡



Grpc分层设计



参考文章：

RPC框架实践之：Google gRPC：<https://www.imooc.com/article/30184>

Protobuf3语言指南：<https://blog.csdn.net/u011518120/article/details/54604615>

gRPC 官方文档中文版：<http://doc.oschina.net/grpc?t=58011>

GRPC:<https://blog.csdn.net/xuduorui/article/details/78278808>

gRPC 服务端创建和调用原理解析:<https://www.sohu.com/a/159931611_355140>

gRPC客户端创建调用原理解析:<https://juejin.im/entry/59bb30f76fb9a00a616f1b73>

GRPC原理解析：<http://shift-alt-ctrl.iteye.com/blog/2292862>