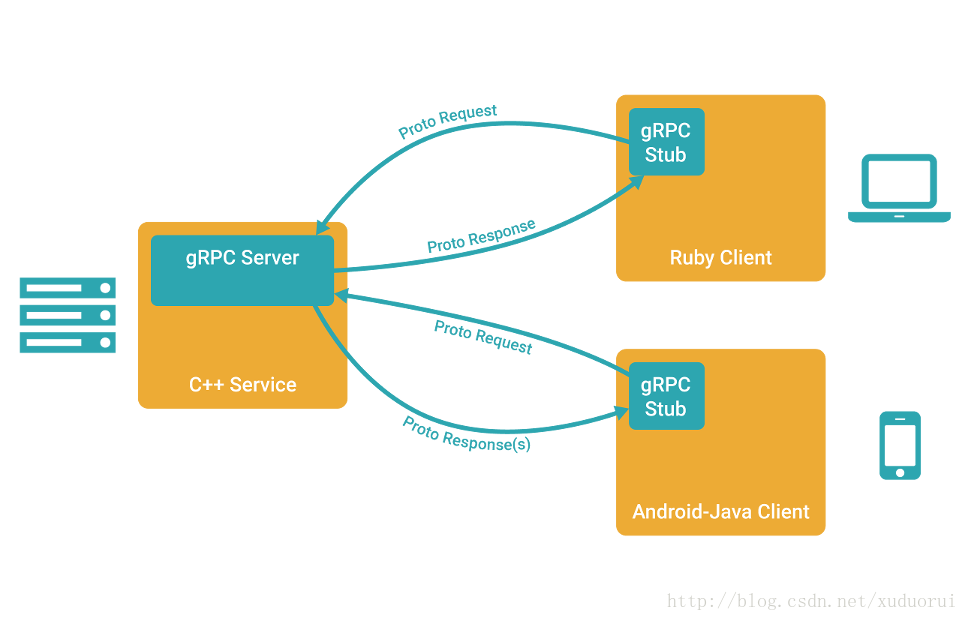
# Springboot整合gRPC

## 概述：

gRPC 一开始由 google 开发，是一款语言中立、平台中立、开源的远程过程调用(RPC)系统。

在 gRPC 里客户端应用可以像调用本地对象一样直接调用另一台不同的机器上服务端应用的方法，使得您能够更容易地创建分布式应用和服务。与许多 RPC 系统类似，gRPC 也是基于以下理念：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。



## 特性

* **基于HTTP/2** HTTP/2 提供了连接多路复用、双向流、服务器推送、请求优先级、首部压缩等机制。可以节省带宽、降低TCP链接次数、节省CPU，帮助移动设备延长电池寿命等。gRPC 的协议设计上使用了HTTP2 现有的语义，请求和响应的数据使用HTTP Body 发送，其他的控制信息则用Header 表示。
* **IDL使用ProtoBuf** gRPC使用ProtoBuf来定义服务，ProtoBuf是由Google开发的一种数据序列化协议（类似于XML、JSON、hessian）。ProtoBuf能够将数据进行序列化，并广泛应用在数据存储、通信协议等方面。压缩和传输效率高，语法简单，表达力强。
* **多语言支持**（C, C++, Python, PHP, Nodejs, C#, Objective-C、Golang、Java） gRPC支持多种语言，并能够基于语言自动生成客户端和服务端功能库。目前已提供了C版本grpc、Java版本grpc-java 和 Go版本grpc-go，其它语言的版本正在积极开发中，其中，grpc支持C、C++、Node.js、Python、Ruby、Objective-C、PHP和C#等语言，grpc-java已经支持Android开发。

## gRPC已经应用在Google的云服务和对外提供的API中，其主要应用场景如下

* 低延迟、高扩展性、分布式的系统
* 同云服务器进行通信的移动应用客户端
* 设计语言独立、高效、精确的新协议
* 便于各方面扩展的分层设计，如认证、负载均衡、日志记录、监控等

## gRPC优缺点

### 优点

protobuf二进制消息，性能好/效率高（空间和时间效率都很不错） proto文件生成目标代码，简单易用 序列化反序列化直接对应程序中的数据类，不需要解析后在进行映射(XML,JSON都是这种方式) 支持向前兼容（新加字段采用默认值）和向后兼容（忽略新加字段），简化升级 支持多种语言（可以把proto文件看做IDL文件） Netty等一些框架集成

### 缺点

1）GRPC尚未提供连接池，需要自行实现 2）尚未提供“服务发现”、“负载均衡”机制 3）因为基于HTTP2，绝大部多数HTTP Server、Nginx都尚不支持，即Nginx不能将GRPC请求作为HTTP请求来负载均衡，而是作为普通的TCP请求。（nginx1.9版本已支持） 4） Protobuf二进制可读性差（貌似提供了Text\_Fromat功能） 默认不具备动态特性（可以通过动态定义生成消息类型或者动态编译支持）

## 使用协议缓冲区

默认情况下，gRPC使用Protocol Buffs，这是Google成熟的开源机制，用于序列化结构化数据（尽管它可以与其他数据格式（如JSON）一起使用）。 这是一个如何工作的快速介绍。 如果您已经熟悉协议缓冲区，请随时跳到下一部分。

使用协议缓冲区的第一步是定义要在*proto文件中*序列化的数据的结构：这是一个扩展名为.proto的普通文本文件。 协议缓冲区数据被构造为*消息* ，其中每条消息是包含一系列称为*字段*的名称 - 值对的信息的小型逻辑记录。这是一个简单的例子：

message Person {   
 string name = 1;   
 int32 id = 2;   
 bool has\_ponycopter = 3;   
}

然后，一旦指定了数据结构，就可以使用协议缓冲区编译器protoc从原型定义生成首选语言的数据访问类。 这些为每个字段提供了简单的访问器（如name()和set\_name() ），以及将整个结构序列化/解析为原始字节的方法 - 例如，如果您选择的语言是C ++，则运行编译器上面的例子将生成一个名为Person的类。 然后，您可以在应用程序中使用此类来填充，序列化和检索Person协议缓冲区消息。

正如您将在我们的示例中更详细地看到的那样，您可以在普通的proto文件中定义gRPC服务，并将RPC方法参数和返回类型指定为协议缓冲区消息：

// The greeter service definition.   
service Greeter {   
//Sends a greeting   
rpc SayHello (HelloRequest) returns (HelloReply) {}   
}  
   
 // The request message containing the user's name.   
message HelloRequest {   
 string name = 1;   
}   
   
// The response message containing the greetings   
message HelloReply {   
 string message = 1;   
}

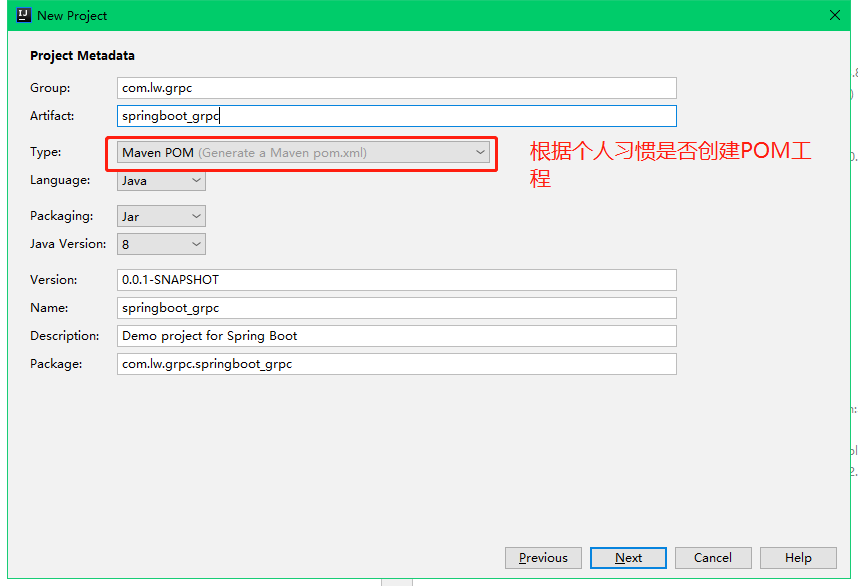
gRPC还使用带有特殊gRPC插件的protoc来生成proto文件中的代码。 但是，使用gRPC插件，您可以生成gRPC客户端和服务器代码，以及用于填充，序列化和检索消息类型的常规协议缓冲区代码。 我们将在下面更详细地看一下这个例子。

## 协议缓冲版本

虽然协议缓冲区已经有一段时间可用于开源用户，但我们的示例使用了一种新的协议缓冲区，称为proto3，它具有略微简化的语法，一些有用的新功能，并支持更多语言。 目前提供Java，C ++，Python，Objective-C，C＃，（Android Java），Ruby和JavaScript，

通常，虽然您可以使用proto2（当前默认协议缓冲版本），但我们建议您将proto3与gRPC一起使用，因为它允许您使用全系列gRPC支持的语言，并避免与proto2客户端通信时的兼容性问题proto3服务器，反之亦然。

## 创建springboot父工程 springboot\_grpc

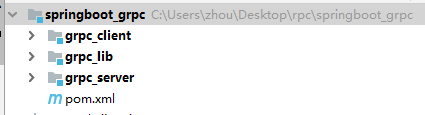


## 创建子工程

创建服务端grpc\_server

创建客户端grpc\_cilent

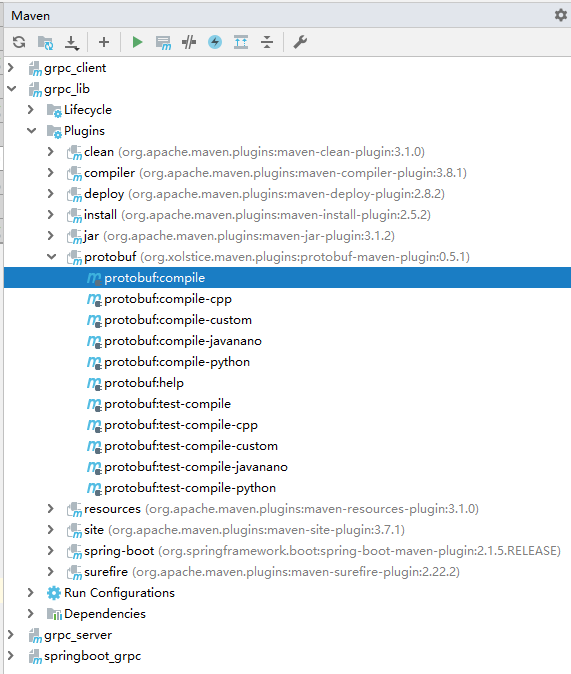
创建存放文件的工程grpc\_lib



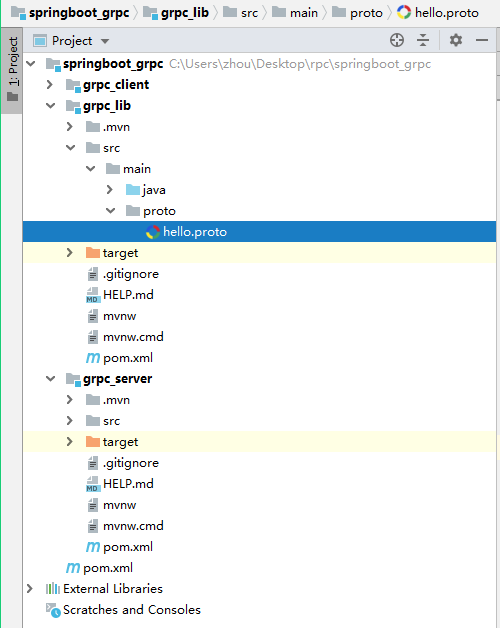
## 修改父工程pom文件

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.1.5.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
 <groupId>com.lw.grpc</groupId>  
 <artifactId>springboot\_grpc</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <name>springboot\_grpc</name>  
 <description>Demo project for Spring Boot</description>  
  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>io.grpc</groupId>  
 <artifactId>grpc-netty-shaded</artifactId>  
 <version>1.14.0</version>  
 </dependency>  
 <dependency>  
 <groupId>io.grpc</groupId>  
 <artifactId>grpc-protobuf</artifactId>  
 <version>1.14.0</version>  
 </dependency>  
 <dependency>  
 <groupId>io.grpc</groupId>  
 <artifactId>grpc-stub</artifactId>  
 <version>1.14.0</version>  
 </dependency>  
 <dependency>  
 <groupId>com.rpc.proto</groupId>  
 <artifactId>grpc\_lib</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>com.rpc.proto</groupId>  
 <artifactId>grpc\_lib</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <scope>compile</scope>  
 </dependency>  
 </dependencies>  
  
 <build>  
  
 <extensions>  
 <extension>  
 <groupId>kr.motd.maven</groupId>  
 <artifactId>os-maven-plugin</artifactId>  
 <version>1.5.0.Final</version>  
 </extension>  
 </extensions>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 <plugin>  
 <groupId>org.xolstice.maven.plugins</groupId>  
 <artifactId>protobuf-maven-plugin</artifactId>  
 <version>0.5.1</version>  
 <configuration>  
 <protocArtifact>com.google.protobuf:protoc:3.5.1-1:exe:${os.detected.classifier}</protocArtifact>  
 <pluginId>grpc-java</pluginId>  
 <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.14.0:exe:${os.detected.classifier}</pluginArtifact>  
 </configuration>  
 <executions>  
 <execution>  
 <goals>  
 <goal>compile</goal>  
 <goal>compile-custom</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <configuration>  
 <source>6</source>  
 <target>6</target>  
 </configuration>  
 </plugin>  
 </plugins>  
 </build>  
</project>

#### 导入plugin后再maven plugin中可以找到protobuf插件



## 在grpc\_lib下创建.proto文件生成java配置文件

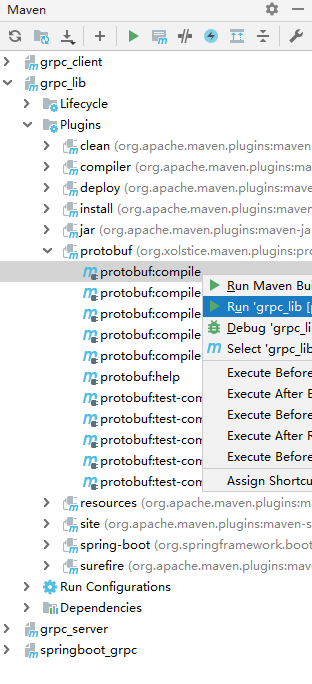


#### 官方给出的proto文件内容

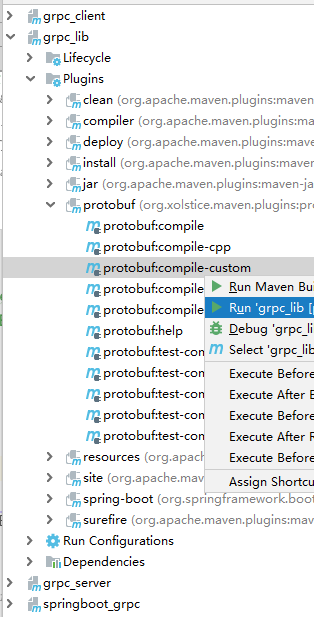
// Copyright 2015 The gRPC Authors  
//  
// Licensed under the Apache License, Version 2.0 (the "License");  
// you may not use this file except in compliance with the License.  
// You may obtain a copy of the License at  
//  
// http://www.apache.org/licenses/LICENSE-2.0  
//  
// Unless required by applicable law or agreed to in writing, software  
// distributed under the License is distributed on an "AS IS" BASIS,  
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
// See the License for the specific language governing permissions and  
// limitations under the License.  
syntax = "proto3";  
  
option java\_multiple\_files = true;  
option java\_package = "io.grpc.examples.helloworld";  
option java\_outer\_classname = "HelloWorldProto";  
option objc\_class\_prefix = "HLW";  
  
package helloworld;  
  
// The greeting service definition.  
service Greeter {  
 // Sends a greeting  
 rpc SayHello (HelloRequest) returns (HelloReply) {}  
}  
  
// The request message containing the user's name.  
message HelloRequest {  
 string name = 1;  
}  
  
// The response message containing the greetings  
message HelloReply {  
 string message = 1;  
}

## 使用插件生成java代码

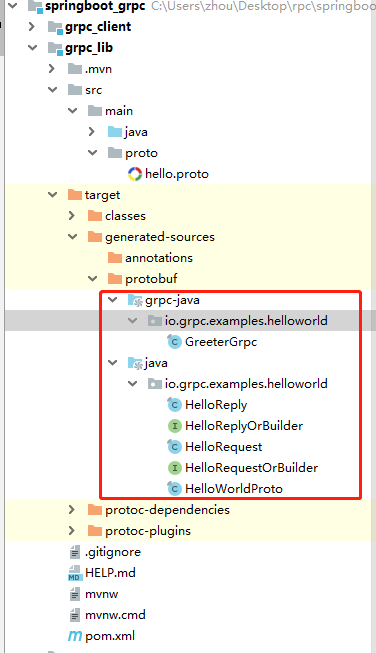
##### 右键Run protobuf:compile插件



##### 右键使用protobuf:compile-custom生成service所需文件



##### 生成文件如下



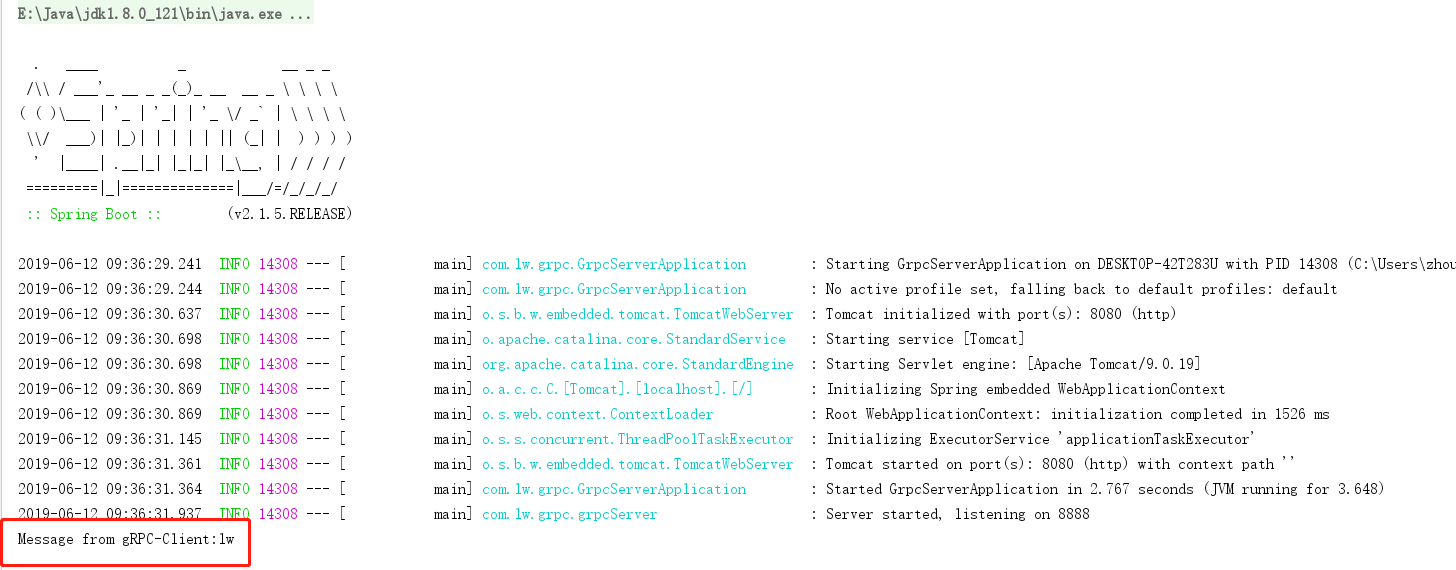
## 配置server端代码

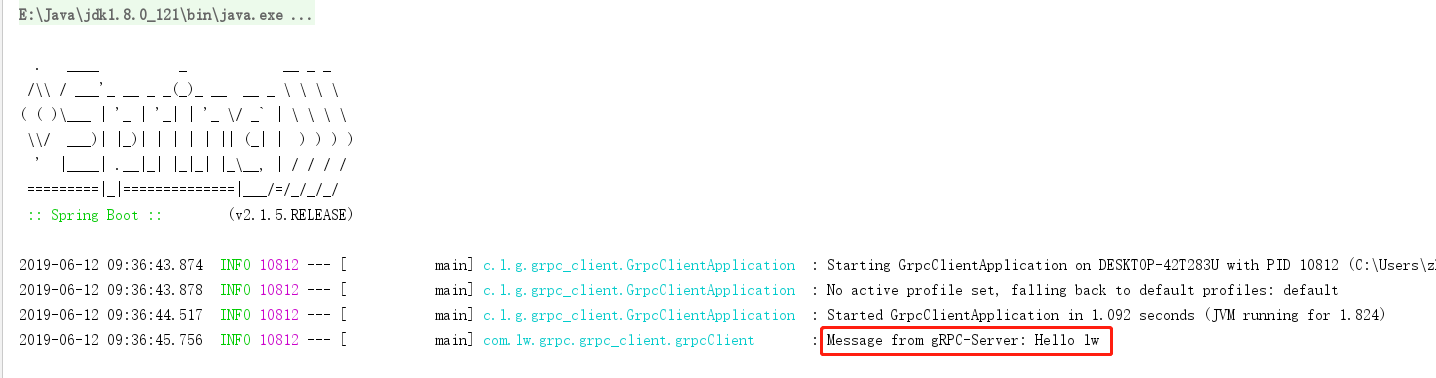
package com.lw.grpc;  
  
import io.grpc.Server;  
import io.grpc.ServerBuilder;  
import io.grpc.examples.helloworld.GreeterGrpc;  
import io.grpc.examples.helloworld.HelloReply;  
import io.grpc.examples.helloworld.HelloRequest;  
import io.grpc.stub.StreamObserver;  
import org.springframework.beans.factory.annotation.Value;  
  
import java.io.IOException;  
import java.util.logging.Logger;  
  
/\*\*  
 \* 类的描述  
 \*  
 \* @author Wangjinghao  
 \* @version v1.0.0  
 \* @date 2019/6/11  
 \*/  
public class grpcServer {  
 private static final Logger logger = Logger.getLogger(grpcServer.class.getName());  
  
  
// @Value("${grpc.server.port}")  
// Integer port;  
  
 private Integer port=8888;  
 private Server server;  
  
 protected void start() throws IOException {  
 server = ServerBuilder.forPort(port)  
 .addService(new GreeterImpl())  
 .build()  
 .start();  
 logger.info("Server started, listening on "+ port);  
  
 Runtime.getRuntime().addShutdownHook(new Thread(){  
  
 @Override  
 public void run(){  
  
 System.err.println("\*\*\* shutting down gRPC server since JVM is shutting down");  
 grpcServer.this.stop();  
 System.err.println("\*\*\* server shut down");  
 }  
 });  
 }  
  
 private void stop(){  
 if (server != null){  
 server.shutdown();  
 }  
 }  
  
 // block 一直到退出程序  
 protected void blockUntilShutdown() throws InterruptedException {  
 if (server != null){  
 server.awaitTermination();  
 }  
 }  
  
 // 实现 定义一个实现服务接口的类  
 private class GreeterImpl extends GreeterGrpc.GreeterImplBase {  
  
 @Override  
 public void sayHello(HelloRequest req, StreamObserver<HelloReply> responseObserver){  
 HelloReply reply = HelloReply.newBuilder().setMessage(("Hello "+req.getName())).build();  
 responseObserver.onNext(reply);  
 responseObserver.onCompleted();  
 System.out.println("Message from gRPC-Client:" + req.getName());  
 }  
 }  
}

## 配置Client端代码

package com.lw.grpc.grpc\_client;  
  
import io.grpc.ManagedChannel;  
import io.grpc.ManagedChannelBuilder;  
import io.grpc.StatusRuntimeException;  
import io.grpc.examples.helloworld.GreeterGrpc;  
import io.grpc.examples.helloworld.HelloReply;  
import io.grpc.examples.helloworld.HelloRequest;  
  
import java.util.concurrent.TimeUnit;  
import java.util.logging.Level;  
import java.util.logging.Logger;  
  
/\*\*  
 \* 类的描述  
 \*  
 \* @author Wangjinghao  
 \* @version v1.0.0  
 \* @date 2019/6/11  
 \*/  
public class grpcClient {  
 private final ManagedChannel channel;  
 private final GreeterGrpc.GreeterBlockingStub blockingStub;  
 private static final Logger logger = Logger.getLogger(grpcClient.class.getName());  
  
 public grpcClient(String host,int port){  
 channel = ManagedChannelBuilder.forAddress(host,port)  
 .usePlaintext(true)  
 .build();  
  
 blockingStub = GreeterGrpc.newBlockingStub(channel);  
 }  
  
  
 public void shutdown() throws InterruptedException {  
 channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);  
 }  
  
 public void greet(String name){  
 HelloRequest request = HelloRequest.newBuilder().setName(name).build();  
 HelloReply response;  
 try{  
 response = blockingStub.sayHello(request);  
 } catch (StatusRuntimeException e)  
 {  
 logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());  
 return;  
 }  
 logger.info("Message from gRPC-Server: "+response.getMessage());  
 }  
  
}

## 运行图如下





声明：我在这里并没有做太多的扩展，springboot和grpc使用同一个端口，若有扩展需求根据自己需求进行扩展