

零基础入门gRPC (一)

一、RPC基本概念

RPC(Remote Procedure Call): 远程过程调用, 它是一种通过网络从远程计算机程序上请求服务, 而不需要了解底层网络技术的思想。

RPC 是一种技术思想而非一种规范或协议, 常见 RPC 技术和框架有:

- 应用级的服务框架: 阿里的 Dubbo/Dubbox、Google gRPC、Spring Boot/Spring Cloud、Facebook 的 Thrift、Twitter 的 Finagle 等。
- 远程通信协议: RMI、Socket、SOAP(HTTP XML)、REST(HTTP JSON)。
- 通信框架: MINA 和 Netty。
- ps: Google gRPC 框架是基于 HTTP2 协议实现的, 底层使用到了 Netty 框架的支持。

1. RPC 框架

一个典型 RPC 的使用场景中, 包含了服务发现、负载、容错、网络传输、序列化等组件, 其中“RPC 协议”就指明了程序如何进行网络传输和序列化。

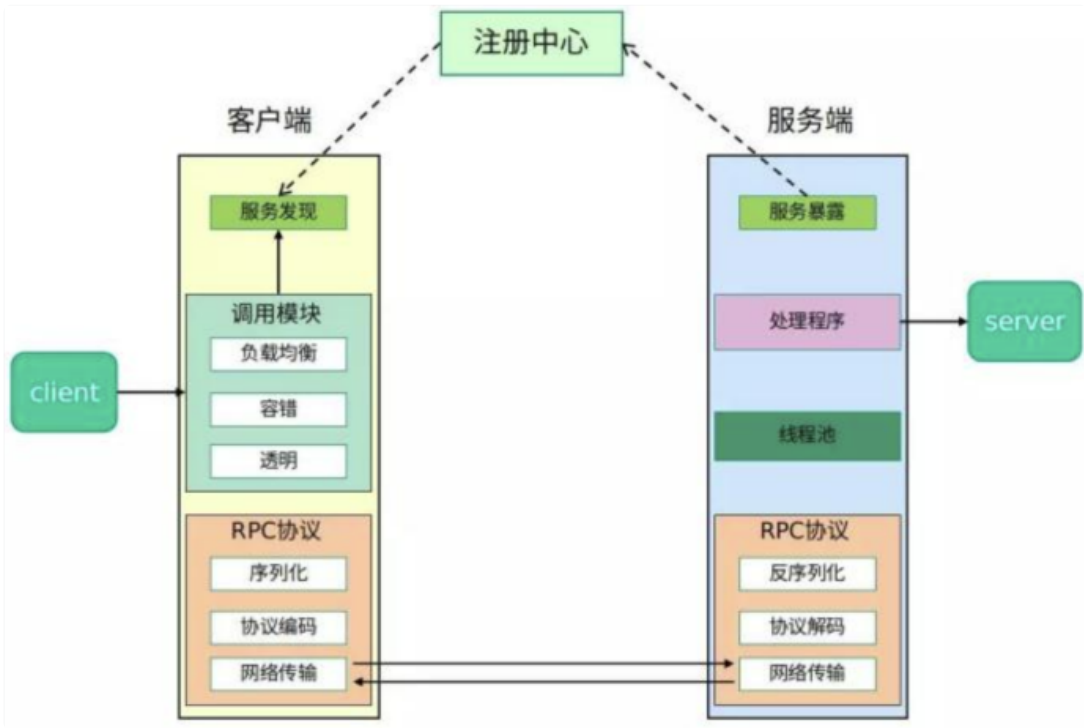


图 1: 完整 RPC 架构图

2. RPC 核心功能

RPC 的核心功能是指实现一个 RPC 最重要的功能模块，就是上图中的” RPC 协议” 部分：



一个 RPC 的核心功能主要有 5 个部分组成，分别是：客户端、客户端 Stub、网络传输模块、服务端 Stub、服务端等。

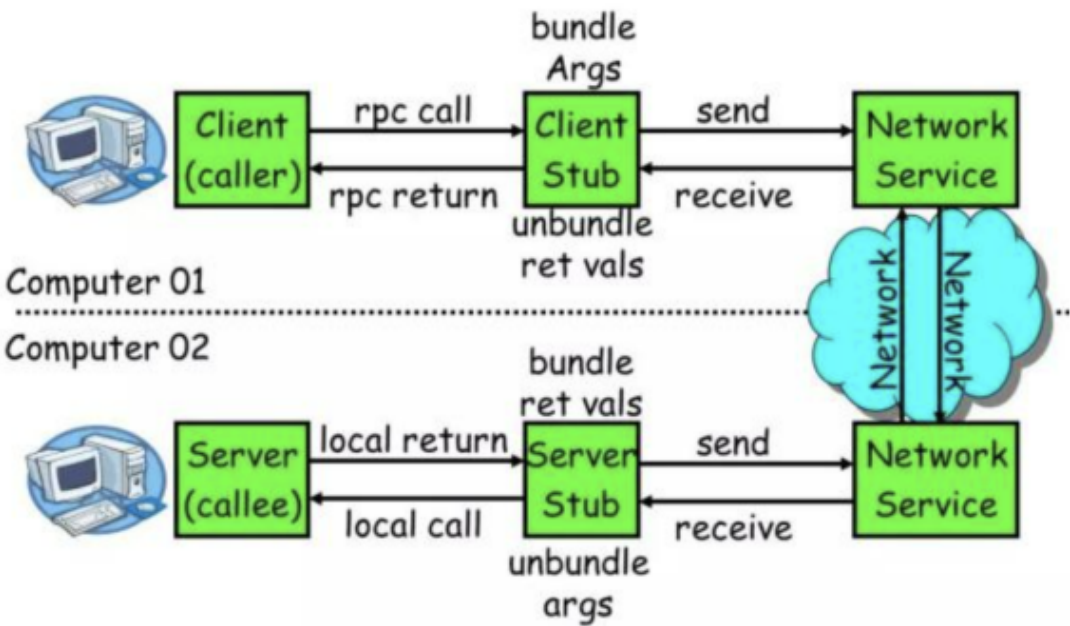


图 2：RPC 核心功能图

下面分别介绍核心 RPC 框架的重要组成：

1. 客户端(Client)：服务调用方。
2. 客户端存根(Client Stub)：存放服务端地址信息，将客户端的请求参数数据信息打包成网络消息，再通过网络传输发送给服务端。
3. 服务端存根(Server Stub)：接收客户端发送过来的请求消息并进行解包，然后再调用本地服务进行处理。
4. 服务端(Server)：服务的真正提供者。
5. Network Service：底层传输，可以是 TCP 或 HTTP。

一次 RPC 调用流程如下：

1. 服务消费者(Client 客户端)通过本地调用的方式调用服务。

2. 客户端存根(Client Stub)接收到调用请求后负责将方法、入参等信息序列化(组装)成能够进行网络传输的消息体。
3. 客户端存根(Client Stub)找到远程的服务地址, 并且将消息通过网络发送给服务端。
4. 服务端存根(Server Stub)收到消息后进行解码(反序列化操作)。
5. 服务端存根(Server Stub)根据解码结果调用本地的服务进行相关处理
6. 服务端(Server)本地服务业务处理。
7. 处理结果返回给服务端存根(Server Stub)。
8. 服务端存根(Server Stub)序列化结果。
9. 服务端存根(Server Stub)将结果通过网络发送至消费方。
10. 客户端存根(Client Stub)接收到消息, 并进行解码(反序列化)。
11. 服务消费方得到最终结果。

RPC的目标就是要2~10这些步骤都封装起来, 让用户对这些细节透明。

二、gRPC简介

官网

<https://www.grpc.io/> <<https://www.grpc.io/>>

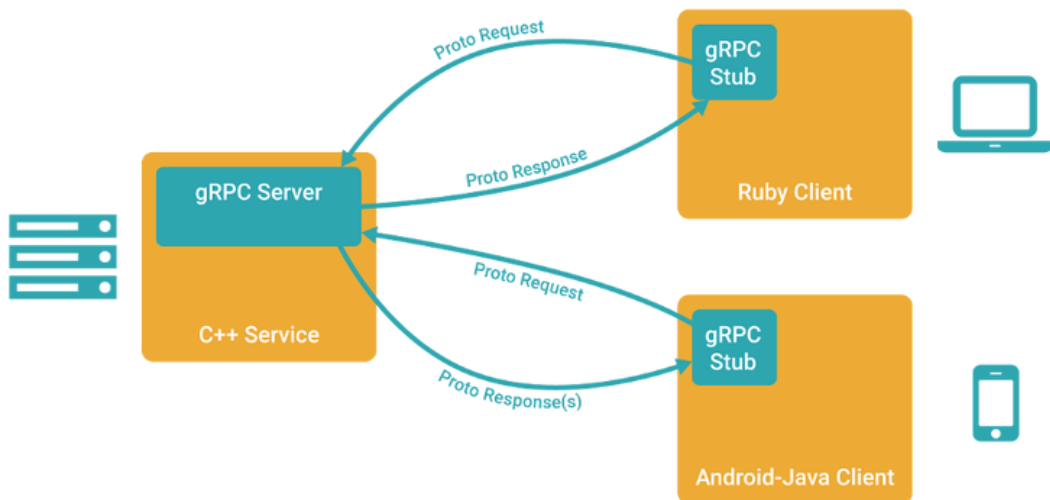
gRPC 官方文档中文版

<http://doc.oschina.net/grpc> <<http://doc.oschina.net/grpc>>

RPC 框架的目标就是让远程服务调用更加简单、透明, 其负责屏蔽底层的传输方式 (TCP/UDP)、序列化方式 (XML/Json) 和通信细节。服务调用者可以像调用本地接口一样调用远程的服务提供者, 而不需要关心底层通信细节和调用过程。

[gRPC](http://www.oschina.net/p/grpc-framework) <<http://www.oschina.net/p/grpc-framework>> 是一个高性能、开源和通用的 RPC 框架, 面向移动和 HTTP/2 设计。目前提供 C、Java 和 Go 语言版本, 分别是: grpc, grpc-java, grpc-go. 其中 C 版本支持 C, C++, Node.js, Python, Ruby, Objective-C, PHP 和 C# 支持。

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。



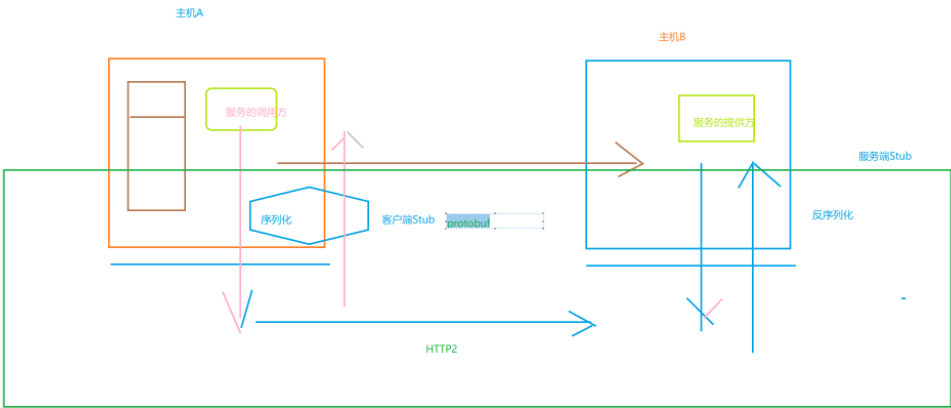
gRPC 客户端和服务端可以在多种环境中运行和交互 - 从 google 内部的服务器到你自己的笔记本，并且可以用任何 gRPC 支持的语言 <<http://doc.oschina.net/grpc?t=58008#quickstart>> 来编写。所以，你可以很容易地用 Java 创建一个 gRPC 服务端，用 Go、Python、Ruby 来创建客户端。

gRPC 的功能优点：

- 高兼容性、高性能、使用简单

gRPC 的组成部分：

- 使用 http2 作为网络传输层
- 使用 protobuf 这个高性能的数据包序列化协议
- 通过 protoc gprc 插件生成易用的 SDK



三、使用 protobuf序列化协议

什么是ProtoBuf

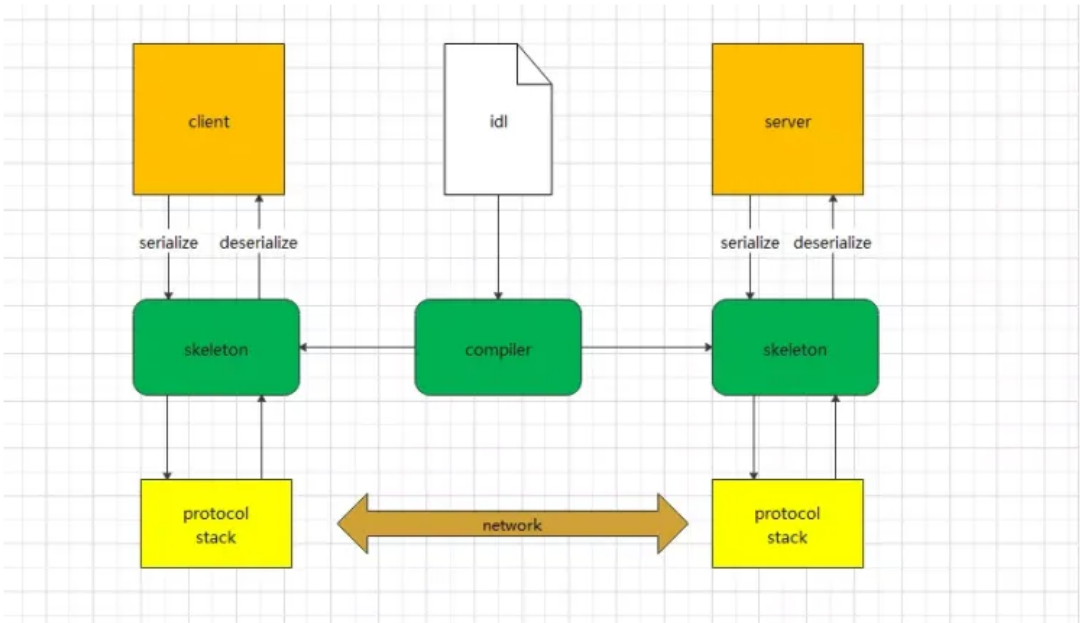
ProtoBuf(Protocol Buffers)是一种跨平台、语言无关、可扩展的序列化结构数据的方法，可用于网络数据交换及存储。

在序列化结构化数据的机制中，ProtoBuf是灵活、高效、自动化的，相对常见的XML、JSON，描述同样的信息，ProtoBuf序列化后数据量更小 (在网络中传输消耗的网络流量更少)、序列化/反序列化速度更快、更简单。

一旦定义要处理的数据的数据结构之后，就可以利用ProtoBuf的代码生成工具生成相关的代码。只需使用 Protobuf 对数据结构进行一次描述，即可利用各种不同语言(proto3支持C++, Java, Python, Go, Ruby, Objective-C, C#)或从各种不同流中对你的结构化数据轻松读写。

如何使用 ProtoBuf

ProtoBuf 协议的工作流程

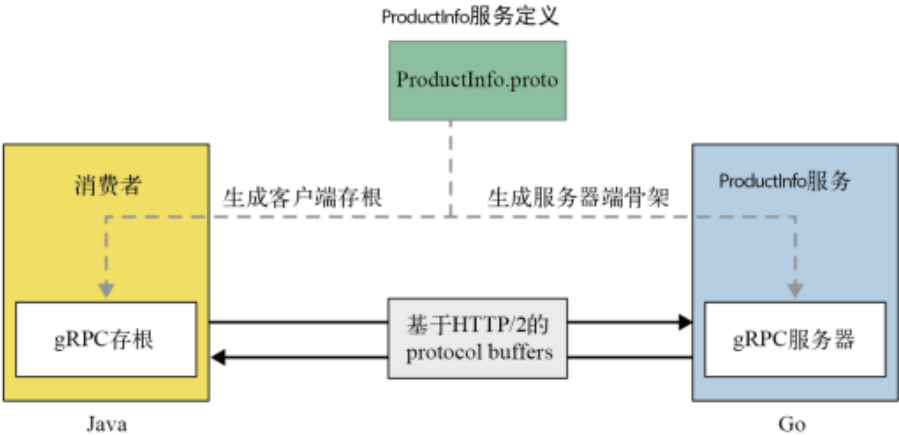


在开发 gRPC 应用程序时，先要定义服务接口，其中应包含如下信息：消费者消费服务的方式、消费者能够远程调用的方法以及调用这些方法所使用的参数和消息格式等。在服务定义中所使用的语言叫作**接口定义语言**（interface definition language, IDL）。

借助服务定义，可以生成服务器端代码，也就是服务器端骨架（这里的“骨架”和“存根”都是代理。服务器端代理叫作“骨架”（skeleton），客户端代理叫作“存根”（stub）。），它通过提供低层级的通信抽象简化了服务器端的逻辑。同时，还可以生成客户端代码，也就是客户端存根，它使用抽象简化了客户端的通信，为不同的编程语言隐藏了低层级的通信。就像调用本地函数那样，客户端能够远程调用我们在服务接口定义中所指定的方法。底层的 gRPC 框架处理所有的复杂工作，通常包括确保严格的服务契约、数据序列化、网络通信、认证、访问控制、可观察性等。

为了理解 gRPC 的基本概念，我们来看一个使用 gRPC 实现微服务的实际场景。假设我们正在构建一个在线零售应用程序，该应用程序由多个微服务组成。

如图 1-1 所示，假设我们要构建一个微服务来展现在线零售应用程序中可售商品的详情。例如，将 `ProductInfo` 服务建模为 gRPC 服务，通过网络对外暴露。



服务定义是在 `ProductInfo.proto` 文件中声明的，服务器端和客户端都会使用该文件来生成代码。这里假设 `ProductInfo` 服务使用 Go 语言来实现，消费者使用 Java 语言来实现，两者之间的通信则通过 HTTP/2 来进行。

典型的序列化和反序列化过程往往需要如下组件：

- IDL (Interface description language) 文件：参与通讯的各方需要对通讯的内容需要做相关的约定 (Specifications)。为了建立一个与语言和平台无关的约定，这个约定需要采用与具体开发语言、平台无关的语言来进行描述。这种语言被称为接口描述语言 (IDL)，采用IDL撰写的协议约定称之为IDL文件。
- IDL Compiler：IDL文件中约定的内容为了在各语言和平台可见，需要有一个编译器，将IDL文件转换成各语言对应的动态库。
- Stub/Skeleton Lib：负责序列化和反序列化的工作代码。Stub是一段部署在分布式系统客户端的代码，一方面接收应用层的参数，并对其序列化后通过底层协议栈发送到服务端，另一方面接收服务端序列化后的结果数据，反序列化后交给客户端应用层；Skeleton部署在服务端，其功能与Stub相反，从传输层接收序列化参数，反序列化后交给服务端应用层，并将应用层的执行结果序列化后最终传送给客户端Stub。
- Client/Server：指的是应用层程序代码，他们面对的是IDL所生存的特定语言的class或struct。
- 底层协议栈和互联网：序列化之后的数据通过底层的传输层、网络层、链路层以及物理层协议转换成数字信号在互联网中传递。

可以看到，对于序列化协议来说，使用方只需要关注业务对象本身，即 `idl` (Interface description language) 定义，序列化和反序列化的代码只需要通过工具生成即可。

ProtoBuf 消息定义

在Java 中，构建一个 `Person` 类的数据结构，包含成员变量 `name`、`id`、`email` 等等

Java | 复制代码

```
1 // Java类
2
3
4 public class Person
5 {
6     private String name;
7     private Int id;
8     private String email;
9     ...
10 }
```

根据上述数据结构的需求，在demo `.proto` 里 通过 `Protocol Buffer` 语法写入对应 `.proto` 对象模型的代码

proto2版本

▼ Protobuf 复制代码

```
1
2
3 package protocobuff_Demo;
4 // 关注1: 包名,防止不同 .proto 项目间命名 发生冲突
5
6 option java_package = "com.chenj.protobuf";//// 作用: 指定生成的
7 option java_outer_classname = "Demo";//作用: 生成对应.java 文件的
8 // 关注2: option选项,作用: 影响 特定环境下 的处理方式
9
10 // 关注3: 消息模型 作用: 真正用于描述 数据结构
11 // 下面详细说明
12 // 生成 Person 消息对象 (包含多个字段, 下面详细说明)
13 message Person {
14     required string name = 1;
15     required int32 id = 2;
16     optional string email = 3;
17
18     enum PhoneType {
19         MOBILE = 0;
20         HOME = 1;
21         WORK = 2;
22     }
23
24     message PhoneNumber {
25         required string number = 1;
26         optional PhoneType type = 2 [default = HOME];
27     }
28
29     repeated PhoneNumber phone = 4;
30 }
31
32 message AddressBook {
33     repeated Person person = 1;
34 }
35
```

proto3版本

Protobuf | 复制代码

```
1  syntax = "proto3"; // 协议版本(proto3中, 在第一行非空白非注释行, 必
2  package protocobuff_Demo;
3  // 关注1: 包名,防止不同 .proto 项目间命名 发生冲突
4
5  option java_package = "com.chenj.protobuf";//// 作用: 指定生成的
6  option java_outer_classname = "Demo";//作用: 生成对应.java 文件的
7  // 关注2: option选项,作用: 影响 特定环境下 的处理方式
8
9  // 关注3: 消息模型 作用: 真正用于描述 数据结构
10 // 下面详细说明
11 // 生成 Person 消息对象 (包含多个字段, 下面详细说明)
12 message Person {
13     string name = 1; //(proto3消息定义时, 移除了 "required"、 "op
14     int32 id = 2; //(proto3消息定义时, 移除了 "required"、 "optio
15     string email = 3; //(proto3消息定义时, 移除了 "required"、 "o
16
17     enum PhoneType {
18         MOBILE = 0;
19         HOME = 1;
20         WORK = 2;
21     }
22
23     message PhoneNumber {
24         string number = 1;
25         PhoneType type = 2 ; //(proto3消息定义时,移除了 default 选
26     }
27
28     repeated PhoneNumber phone = 4;
29 }
30
31 message AddressBook {
32     repeated Person person = 1;
33 }
34
```

1. 消息对象

在 `ProtocolBuffers` 中:

- 一个消息对象 (`Message`) = 一个 结构化数据
- 消息对象用 修饰符 `message` 修饰

- 消息对象 含有 字段：消息对象（ Message ）里的 字段 = 结构化数据 里的 成员变量

```
public class Person
{
    private String name;
    private Int id;
    private String email;
    ...
}
```

```
message Person {

    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
```

2. 字段

消息对象的字段 组成主要是： 字段 = 字段修饰符 + 字段类型 + 字段名 + 标识号

```
required string name = 1;
```

字段修饰符 字段类型 字段名 标识号

字段修饰符

作用： 设置该字段解析时的规则

字段修饰符	作用	备注
required	表示该字段必须赋值 (必须有且只有1个)	一个格式良好的消息对象一定要含有1个该字段（永久性） (否则消息会被认为是“未初始化的 (uninitialized)”，编译时将抛出一个 RuntimeException)
optional	表示该字段可选赋值，即 可设置 / 不设置 (最多不超过1个)	<ul style="list-style-type: none">• 如果可选字段解析时没有值，则使用默认值（自定义默认值 / 系统默认值）；• 自定义默认值在消息描述文件中指定：如 optional int32 result_per_page = 3 [default = 10];• 系统默认值规则：数字类型 = 0, 字符串类型 = 空字符串, bools值 = false, 枚举类型: 枚举类型定义中的第一个值。• 对于内嵌消息，默认值=消息的“默认实例 (default instance)” 或 “原型(prototype)”，且无字段集；• 若调用 accessor 获取未设置的 optional字段的 值，返回的是字段的默认值。
repeated	表示该字段可以被重复多次赋值 (0 / 多个)	<ul style="list-style-type: none">• 重复的值的顺序会被保留，相当于动态变化的数组；• 特别注意：由于历史原因，repeated的字段 并没有被高效编码。所以用户应该使用特殊选项：[packed=true]来保证更高效的编码。如：repeated int32 samples = 4 [packed=true]。

字段类型

字段类型主要有 三 类：

- 基本数据 类型
- 枚举 类型
- 消息对象 类型

基本数据类型

.proto 基本数据类型 对应于 各平台的基本数据类型如下

.proto类型	Java 类型	C++类型	备注
double	double	double	
bool	boolean	bool	
string	String	string	一个字符串必须是 UTF-8 编码 或者 7-bit ASCII 编码 的文本
bytes	ByteString	string	可能包含任意顺序的字节数据
int32	int	int32	使用可变长编码方式。编码负数时不够高效——如果你的字段可能含有负数，那么请使用 sint32
int64	long	int64	使用可变长编码方式。编码负数时不够高效——如果你的字段可能含有负数，那么请使用 sint64
uint32	int[1]	uint32	使用可变长编码方式。编码负数时不够高效——如果你的字段可能含有负数，那么请使用 sint64
uint64	long[1]	uint64	使用可变长编码方式。编码负数时不够高效——如果你的字段可能含有负数，那么请使用 sint64
int32	int	int32	有符号的整型值、使用可变长编码方式，编码时比通常的 int32 高效
sint64	long	sint64	有符号的整型值、使用可变长编码方式，编码时比通常的 int64 高效
fixed32	int[1]	uint32	总是4个字节 (如果数值总是比2的28次方大，这个类型会比uint32高效)
fixed64	long[1]	uint64	总是8个字节 (如果数值总是比2的56次方大，这个类型会比uint64高效)
sfixed32	int	int32	总是4个字节
sfixed64	long	int64	总是8个字节

枚举类型

作用：为字段指定一个 可能取值的字段集合，该字段只能从 该指定的字段集合里取值

下面例子，电话号码 可能是手机号、家庭电话号或工作电话号的其中一个，那么就将 PhoneType 定义为枚举类型，并将加入电话的集合（ MOBILE、HOME、WORK ）

▼ Protobuf 复制代码

```
1 // 枚举类型需要先定义才能进行使用
2
3 // 枚举类型 定义
4 enum PhoneType {
5     MOBILE = 0;
6     HOME = 1;
7     WORK = 2;
8 // 电话类型字段 只能从 这个集合里 取值
9 }
10
11 // 特别注意:
12 // 1. 枚举类型的定义可在一个消息对象的内部或外部
13 // 2. 都可以在 同一.proto文件 中的任何消息对象里使用
14 // 3. 当枚举类型是在一消息内部定义, 希望在 另一个消息中 使用时, 需要
15
16 message PhoneNumber {
17     required string number = 1;
18     optional PhoneType type = 2 [default = HOME];
19     // 使用枚举类型的字段 (设置了默认值)
20 }
21
22 // 特别注意:
23 // 1. 枚举常量必须在32位整型值的范围内
24 // 2. 不推荐在enum中使用负数: 因为enum值是使用可变编码方式的, 对负数
25
26
```

标识号

作用: 通过二进制格式唯一标识每个字段

1. 一旦开始使用就不能够再改变
2. 标识号使用范围: $[1, 2^{29} - 1]$

每个字段在进行编码时都会占用内存, 而 占用内存大小 取决于 标识号:

1. 范围 $[1, 15]$ 标识号的字段 在编码时占用1个字节;
2. 范围 $[16, 2047]$ 标识号的字段 在编码时占用2个字节

生成代码

首先安装 ProtoBuf 编译器 protoc, [这里](#)

<<https://developers.google.com/protocol-buffers/docs/downloads>> 有详细的安装教程, 安装完成后, 可以使用以下命令生成 Java 源代码:

```
▼ Protobuf | 复制代码
1
2 protoc --java_out=./src/main/java ./src/main/idl/customer.proto
```

从项目的根路径执行该命令, 并添加了两个参数: java_out, 定义./src/main/java/ 为Java代码的输出目录; 而./src/main/idl/customer.proto是.proto文件所在目录。
(编译器为每个.proto文件里的每个消息类型生成一个.java文件&一个Builder类 (Builder类用于创建消息类接口))

具体项目中使用

消息对象类介绍

通过 .proto文件 转换的 Java 源代码 = Protocol Buffer 类 + 消息对象类 (含 Builder 内部类)

消息对象类 (Message 类)

- 消息对象类 类通过 二进制数组 写 和 读 消息类型
- 使用方法包括:

Java | 复制代码

```

1  <-- 方式1: 直接序列化和反序列化 消息 -->
2  protocolBuffer.toByteArray();
3  // 序列化消息 并 返回一个包含它的原始字节的字节数组
4  protocolBuffer.parseFrom(byte[] data);
5  // 从一个字节数组 反序列化（解析） 消息
6
7
8
9
10 <-- 方式2: 通过输入/ 输出流（如网络输出流） 序列化和反序列化消息 -->
11 protocolBuffer.writeTo(OutputStream output);
12 output.toByteArray();
13 // 将消息写入 输出流 ， 然后再 序列化消息
14
15
16
17 protocolBuffer.parseFrom(InputStream input);
18 // 从一个 输入流 读取并 反序列化（解析）消息
19
20
21
22
23 // 只含包含字段的getters方法
24 // required string name = 1;
25 public boolean hasName(); // 如果字段被设置，则返回true
26 public java.lang.String getName();
27
28
29
30 // required int32 id = 2;
31 public boolean hasId();
32 public int getId();
33
34
35
36 // optional string email = 3;
37 public boolean hasEmail();
38 public String getEmail();
39
40
41 // repeated .tutorial.Person.PhoneNumber phone = 4;
42 // 重复（repeated）字段有一些额外方法
43 public List<PhoneNumber> getPhoneList();
44 public int getPhoneCount();
45 // 列表大小的速记
46 // 作用：通过索引获取和设置列表的特定元素的getters和setters

```

Builder 类

作用：创建 消息构造器 & 设置/ 获取消息对象的字段值 & 创建 消息类 实例

a. 创建 消息构造器

▼

Java | 复制代码

```
1 Demo.Person.Builder person = Person.newBuilder();
```

b. 设置/ 获取 消息对象的字段值 具体方法如下:

Java | 复制代码

```
1 // 标准的JavaBeans风格: 含getters和setters
2 // required string name = 1;
3
4 public boolean hasName(); // 如果字段被设置, 则返回true
5 public java.lang.String getName();
6 public Builder setName(String value);
7 public Builder clearName(); // 将字段设置回它的空状态
8
9
10
11 // required int32 id = 2;
12 public boolean hasId();
13 public int getId();
14 public Builder setId(int value);
15 public Builder clearId();
16
17
18
19
20 // optional string email = 3;
21 public boolean hasEmail();
22 public String getEmail();
23 public Builder setEmail(String value);
24 public Builder clearEmail();
25
26
27
28
29 // repeated .tutorial.Person.PhoneNumber phone = 4;
30 // 重复 (repeated) 字段有一些额外方法
31 public List<PhoneNumber> getPhoneList();
32 public int getPhoneCount();
33 // 列表大小的速记
34 // 作用: 通过索引获取和设置列表的特定元素的getters和setters
35
36
37 public PhoneNumber getPhone(int index);
38 public Builder setPhone(int index, PhoneNumber value);
39
40 public Builder addPhone(PhoneNumber value);
41 // 将新元素添加到列表的末尾
42
43 public Builder addAllPhone(Iterable<PhoneNumber> value);
44 // 将一个装满元素的整个容器添加到列表中
45 public Builder clearPhone();
```

具体使用

使用步骤如下：

****步骤1: ****通过 消息类的内部类Builder类 构造 消息构造器

****步骤2: ****通过 消息构造器 设置 消息字段的值

****步骤3: ****通过 消息构造器 创建 消息类 对象

****步骤4: ****序列化 / 反序列化 消息

具体使用如下：（注释非常清晰）

Java | 复制代码

```
1  package com.chenj.protobuf;
2
3
4  import java.io.ByteArrayInputStream;
5  import java.io.ByteArrayOutputStream;
6  import java.io.IOException;
7  import java.util.Arrays;
8
9  public class TestProto {
10     public static void main(String[] args) {
11         // 步骤1:通过 消息类的内部类Builder类 构造 消息类的消息构造器
12         Demo.Person.Builder personBuilder = Demo.Person.newBuilder();
13
14         // 步骤2:设置你想要设置的字段为你选择的值
15         personBuilder.setName("Lisi");// 在定义.proto文件时,该字段的名称为name
16         personBuilder.setId(123);// 在定义.proto文件时,该字段的名称为id
17         personBuilder.setEmail("lisi.ho@foxmail.com");// 在定义.proto文件时,该字段的名称为email
18
19         Demo.Person.PhoneNumber.Builder phoneNumber = Demo.Person.PhoneNumber.newBuilder();
20         phoneNumber.setType(Demo.Person.PhoneType.HOME);// 在定义.proto文件时,该字段的名称为type
21         phoneNumber.setNumber("0157-23443276");
22         // PhoneNumber消息是嵌套在Person消息里,可以理解为内部类
23         // 所以创建对象时要通过外部类来创建
24
25         // 步骤3:通过 消息构造器 创建 消息类 对象
26         Demo.Person person = personBuilder.build();
27
28         // 步骤4:序列化和反序列化消息(两种方式)
29
30         /*方式1: 直接 序列化 和 反序列化 消息 */
31         // a.序列化
32         byte[] byteArray1 = person.toByteArray();
33         // 把 person消息类对象 序列化为 byte[]字节数组
34         System.out.println(Arrays.toString(byteArray1));
35         // 查看序列化后的字节流
36
37         // b.反序列化
38         try {
39
40             Demo.Person person_Request = Demo.Person.parseFrom(byteArray1);
41             // 当接收到字节数组byte[] 反序列化为 person消息类对象
42
43             System.out.println(person_Request.getName());
44             System.out.println(person_Request.getId());
45             System.out.println(person_Request.getEmail());
```

```
46         // 输出反序列化后的消息
47     } catch (IOException e) {
48         e.printStackTrace();
49     }
50
51
52     /*方式2: 通过输入/ 输出流（如网络输出流） 序列化和反序列化
53     // a.序列化
54     ByteArrayOutputStream output = new ByteArrayOutputStream()
55     try {
56
57         person.writeTo(output);
58         // 将消息序列化 并写入 输出流(此处用 ByteArrayOutputStream
59
60     } catch (IOException e) {
61         e.printStackTrace();
62     }
63
64     byte[] byteArray = output.toByteArray();
65     // 通过 输出流 转化成二进制字节流
66
67     // b. 反序列化
68     ByteArrayInputStream input = new ByteArrayInputStream(byteArray)
69     // 通过 输入流 接收消息流(此处用 ByteArrayInputStream 代替
70
71     try {
72
73         Demo.Person person_Request = Demo.Person.parseFrom(input)
74         // 通过输入流 反序列化 消息
75
76         System.out.println(person_Request.getName());
77         System.out.println(person_Request.getId());
78         System.out.println(person_Request.getEmail());
79         // 输出消息
80     } catch (IOException e) {
81         e.printStackTrace();
82     }
83 }
84 }
85
```

序列化原理解析

请记住 Protocol Buffer 的 **三个关于数据存储** 的重要结论:

- 结论1: Protocol Buffer 将 消息里的每个字段 进行编码后, 再利用 T - L - V 存储方式 进行数据的存储, 最终得到的是一个 二进制字节流 T - L - V 的数据存储方式

即 Tag - Length - Value , 标识 - 长度 - 字段值 存储方式
以 标识 - 长度 - 字段值 表示单个数据, 最终将所有数据拼接成一个 字节流, 从而 实现 数据存储 的功能

其中 Length 可选存储, 如 储存 Varint 编码数据就不需要存储 Length

示意图



- 优点从上图可知, T - L - V 存储方式的优点是
- a. 不需要分隔符 就能 分隔开字段, 减少了 分隔符 的使用
 - b. 各字段 存储得非常紧凑, 存储空间利用率非常高
 - c. 若 字段没有被设置字段值, 那么该字段在序列化时的数据中是完全不存在的, 即不需要编码

- 结论2: Protocol Buffer 对于不同数据类型 采用不同的 序列化方式 (编码方式 & 数据存储方式) , 如下图:

Wire Type 值	编码方式	编码长度	存储方式	代表的数据类型
0	Varint (负数时以Zigzag辅助编码)	变长 (1~10个字节)	T - V	<ul style="list-style-type: none">int32, int64, uint32, uint64,bool, enumsint32, sint64(负数时使用)
1	64-bit	固定8个字节		fixed64, sfixed64, double
2	Length-delimi	变长	T - L - V	string, bytes, embedded messages, packed repeated fields
3	Start group	已弃用	已弃用	Groups (已弃用)
4	End group			
5	32-bit	固定4个字节	T - V	fixed32, sfixed32, float

从上表可以看出:

1. 对于存储 `Varint` 编码数据, 就不需要存储字节长度 `Length`, 所以实际上 `Protocol Buffer` 的存储方式是 `T - V`;
 2. 若 `Protocol Buffer` 采用其他编码方式 (如 `LENGTH_DELIMITED`) 则采用 `T - L - V`
- 结论3: 因为 `Protocol Buffer` 对于数据字段值的 **独特编码方式** & `T - L - V 数据存储方式`, 使得 `Protocol Buffer` 序列化后数据量体积如此小

总结

`Protocol Buffer`的序列化 & 反序列化简单 & 速度快的原因是:

- a. 编码 / 解码 方式简单 (只需要简单的数学运算 = 位移等等)
- b. 采用 `Protocol Buffer` 自身的框架代码 和 编译器 共同完成

`Protocol Buffer`的数据压缩效果好 (即序列化后的数据量体积小) 的原因是:

- a. 采用了独特的编码方式, 如`Varint`、`Zigzag`编码方式等等
- b. 采用 `T - L - V` 的数据存储方式: 减少了分隔符的使用 & 数据存储得紧凑

定义服务(Service)

如果想要将消息类型用在RPC(远程方法调用)系统中, 可以在`.proto`文件中定义一个RPC服务接口, `protocol buffer`编译器将会根据所选择的不同语言生成服务接口代码及存根。如, 想要定义一个RPC服务并具有一个方法, 该方法能够接收`SearchRequest`并返回一个`SearchResponse`, 此时可以在`.proto`文件中进行如下定义:

```
▼ Protobuf | 复制代码
1 service SearchService {
2   rpc Search (SearchRequest) returns (SearchResponse);
3 }
```

最直观的使用`protocol buffer`的RPC系统是[gRPC](https://github.com/grpc/grpc-experiments)

<<https://github.com/grpc/grpc-experiments>>, 一个由谷歌开发的语言和平台中的开源的RPC系统, `gRPC`在使用`protocol buffer`时非常有效, 如果使用特殊的`protocol buffer`插件可以直接为您从`.proto`文件中产生相关的RPC代码。

扩展阅读:

Protobuf2 语法指南

<https://colobu.com/2015/01/07/Protobuf-language-guide/>

<<https://colobu.com/2015/01/07/Protobuf-language-guide/>>

Protobuf3 语法指南

<https://colobu.com/2017/03/16/Protobuf3-language-guide/>

<<https://colobu.com/2017/03/16/Protobuf3-language-guide/>>

几种序列化方式的性能比较

<https://www.cjluo.top/2019/02/02/%E5%87%A0%E7%A7%8D%E5%BA%8F%E5%88%97%E5%8C%96%E5%B7%A5%E5%85%B7%E7%9A%84%E6%A8%AA%E5%90%91%E6%AF%94%E8%BE%83/>

<<https://www.cjluo.top/2019/02/02/%E5%87%A0%E7%A7%8D%E5%BA%8F%E5%88%97%E5%8C%96%E5%B7%A5%E5%85%B7%E7%9A%84%E6%A8%AA%E5%90%91%E6%AF%94%E8%BE%83/>>

四、快速使用 gRPC 框架

环境准备

安装protobuf

protobuf官网

<https://developers.google.com/protocol-buffers>

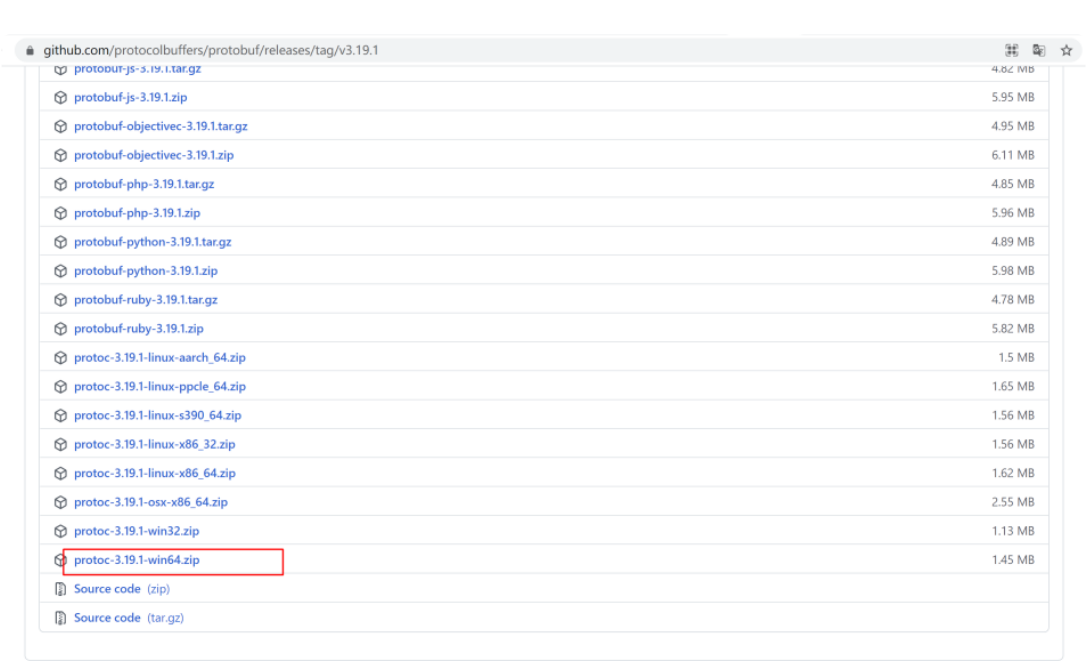
<<https://developers.google.com/protocol-buffers>>

下载地址:

<https://github.com/protocolbuffers/protobuf/releases/tag/v3.19.1>

<<https://github.com/protocolbuffers/protobuf/releases/tag/v3.19.1>>

Windows下的安装很简单，只需到github上下载Windows平台对应的压缩包然后解压即可



下载后解压，会看到该bin文件夹下有一个名为protoc.exe的应用程序，这个就是本文要使用的编译器。

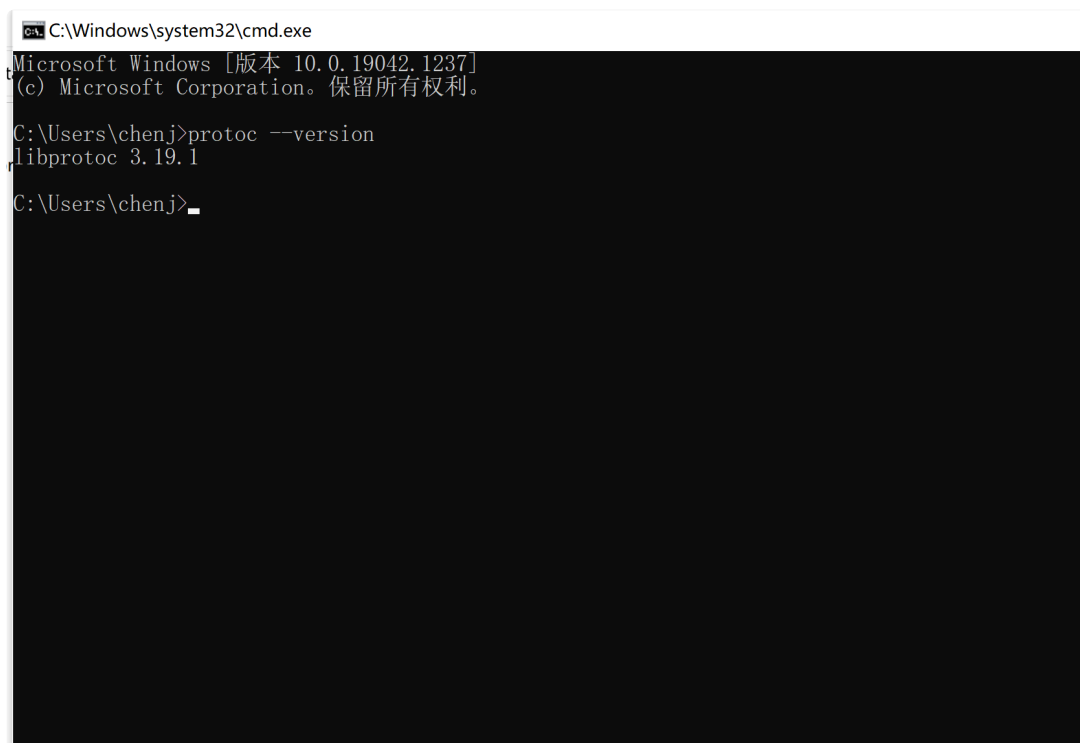
配置环境变量

- 变量名：PROTOCBUF_HOME
- 变量值：D:\protoc-3.19.1-win64

找到系统变量中的 path 变量，选中后点击 编辑，在 新建 的内容中输入：%PROTOBUF_HOME%\bin

验证是否安装成功

出现如图所示



```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.19042.1237]
(c) Microsoft Corporation。保留所有权利。

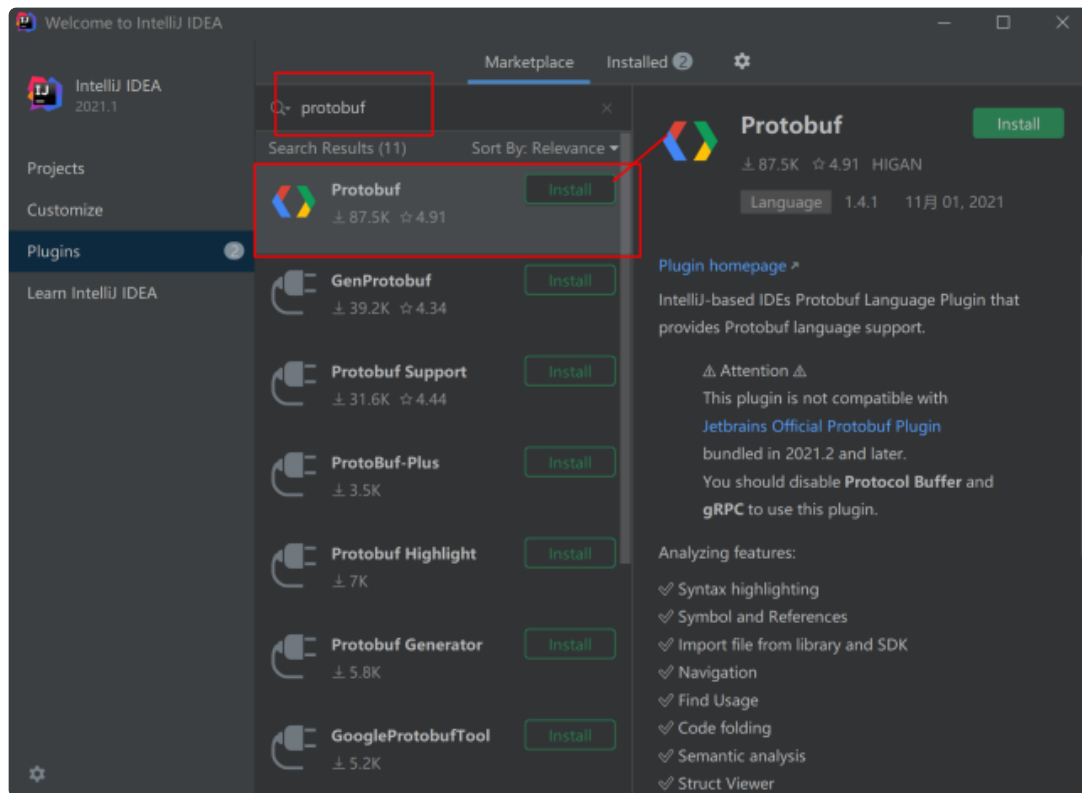
C:\Users\chenj>protoc --version
libprotoc 3.19.1

C:\Users\chenj>
```

安装成功

安装protocbuf插件

idea 建议下载一个 protobuf的插件, 可以有代码提示. 这里直接去plugging里搜就行了.

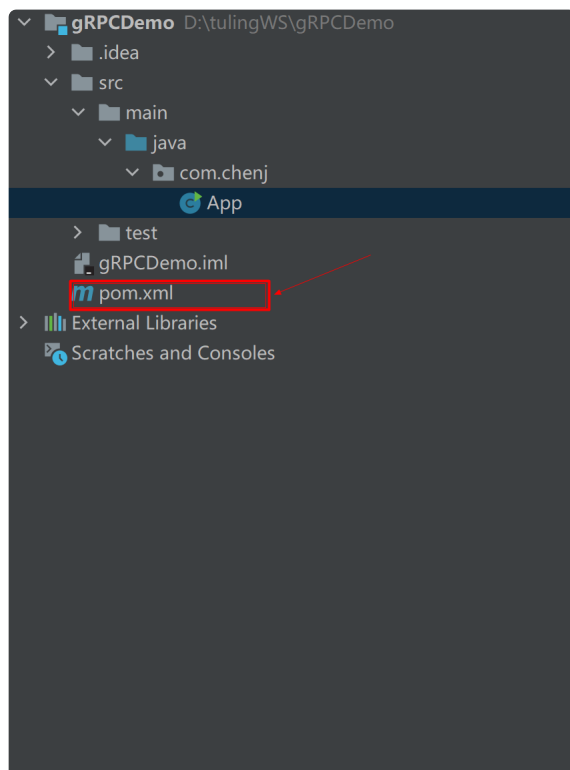


在idea的plugins中搜索 `proto` ，然后下载如下的插件就行（最多下载的那个），点击install，然后重启就可以。

gRPC项目构建

新建Maven项目并修改pom.xml

首先用IDEA新建一个maven项目



修改pom.xml，注意这个build标签和properties标签都是最顶级标签的直接子标签。

XML | 复制代码

```
1 <build>
2   <extensions>
3     <extension>
4       <groupId>kr.motd.maven</groupId>
5       <artifactId>os-maven-plugin</artifactId>
6       <version>1.6.2</version>
7     </extension>
8   </extensions>
9   <plugins>
10    <plugin>
11      <groupId>org.xolstice.maven.plugins</groupId>
12      <artifactId>protobuf-maven-plugin</artifactId>
13      <version>0.6.1</version>
14      <configuration>
15        <protocArtifact>com.google.protobuf:protoc:3.17.3:exe:$
16        <pluginId>grpc-java</pluginId>
17        <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.42.0:exe
18      </configuration>
19      <executions>
20        <execution>
21          <goals>
22            <goal>compile</goal>
23            <goal>compile-custom</goal>
24          </goals>
25        </execution>
26      </executions>
27    </plugin>
28  </plugins>
29 </build>
30
```

接着我们继续在pom.xml中添加一些，这些依赖是构造gRPC-java项目必须用到的（来自官方文档）

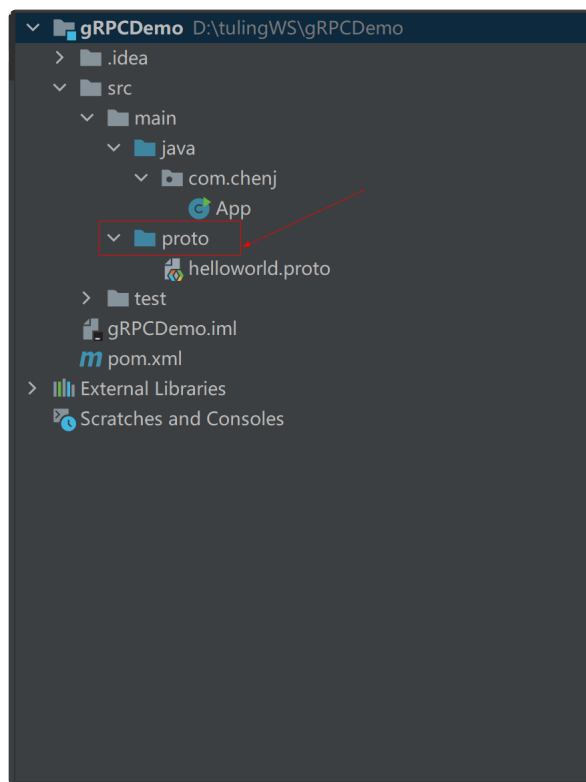
XML | 复制代码

```
1 <dependency>
2   <groupId>io.grpc</groupId>
3   <artifactId>grpc-netty-shaded</artifactId>
4   <version>1.42.0</version>
5 </dependency>
6 <dependency>
7   <groupId>io.grpc</groupId>
8   <artifactId>grpc-protobuf</artifactId>
9   <version>1.42.0</version>
10 </dependency>
11 <dependency>
12   <groupId>io.grpc</groupId>
13   <artifactId>grpc-stub</artifactId>
14   <version>1.42.0</version>
15 </dependency>
16 <dependency> <!-- necessary for Java 9+ -->
17   <groupId>org.apache.tomcat</groupId>
18   <artifactId>annotations-api</artifactId>
19   <version>6.0.53</version>
20   <scope>provided</scope>
21 </dependency>
```

添加.proto文件

proto文件用来描述rpc请求体、响应体、以及rpc提供的服务。通过插件可以根据proto文件生成Java类。

这里面有个非常重要的点要注意，就是proto文件存放的位置。一定要在和src/main/java源文件目录同级的proto源文件目录才可以。如下图所示：



我们添加一个proto文件：helloworld.proto

Protobuf | 复制代码

```
1  syntax = "proto3"; // 协议版本
2
3
4  // 选项配置
5  option java_package = "com.chenj.protobuf";
6  option java_outer_classname = "RPCDateServiceApi";
7  option java_multiple_files = true;
8
9
10
11 // 定义包名
12 package com.chenj.protobuf;
13
14
15 // 服务接口.定义请求参数和相应结果
16 service RPCDateService {
17     rpc getDate (RPCDateRequest) returns (RPCDateResponse) {
18     }
19 }
20
21 // 定义请求体
22 message RPCDateRequest {
23     string userName = 1;
24 }
25
26 // 定义响应内容
27 message RPCDateResponse {
28     string serverDate = 1;
29 }
```

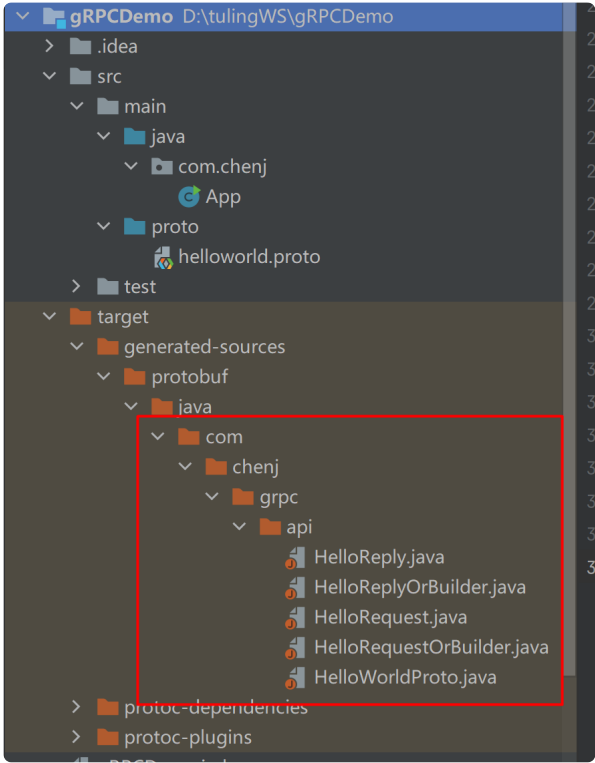
根据.proto文件生成消息体类文件和XXXGrpc类文件

使用maven命令.

在第一步修改的pom.xml的路径下, 首先执行

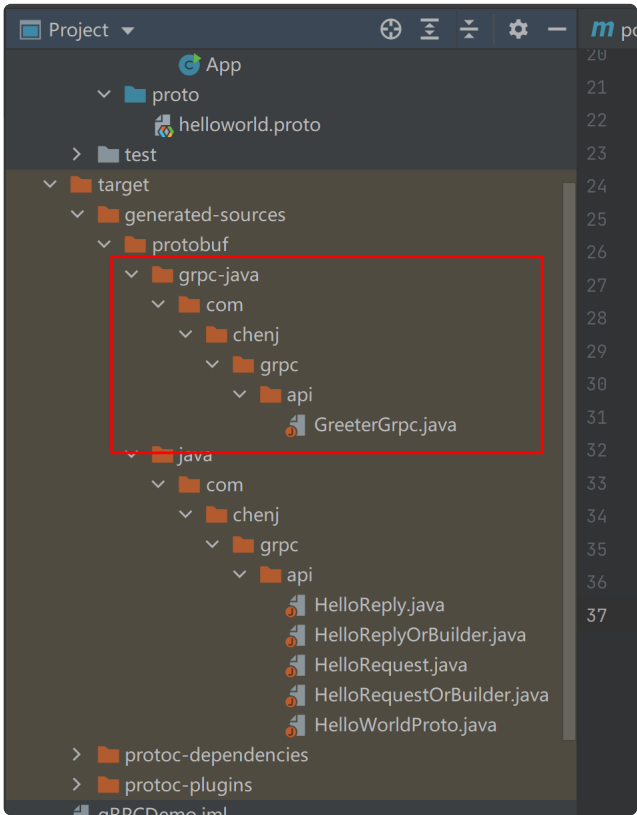
```
mvn protobuf:compile
```

 生成消息体类文件

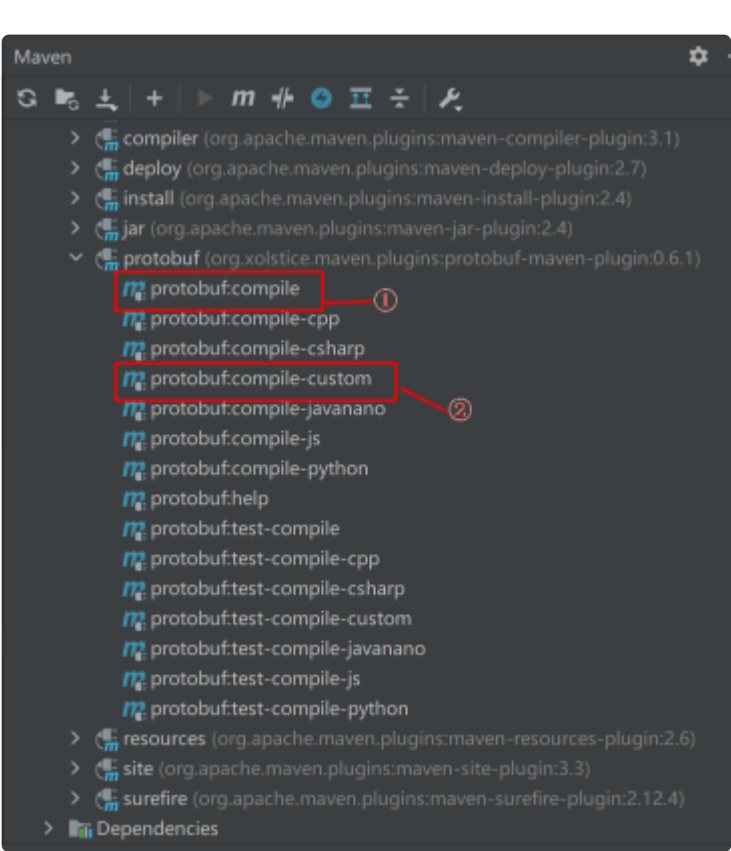


接着执行：

`mvn protobuf:compile-custom` 生成XXXGrpc类文件



使用maven插件, 编译.



第一个命令执行完. 在 target目录里找就行了. 第二个命令也是找就行了. 然后将生成的Java文件拷贝到你的目录里. 就可以了

编写接口实现类

Java | 复制代码

```
1 package com.chenj;  
2  
3 import com.chenj.grpc.api.RPCDateRequest;  
4 import com.chenj.grpc.api.RPCDateResponse;  
5 import com.chenj.grpc.api.RPCDateServiceGrpc;  
6 import io.grpc.stub.StreamObserver;  
7  
8 import java.time.LocalDate;  
9 import java.time.format.DateTimeFormatter;  
10  
11 // RPCDateServiceGrpc.RPCDateServiceImplBase 这个就是接口。  
12 // RPCDateServiceImpl 我们需要继承他的,实现方法回调  
13 public class RPCDateServiceImpl extends RPCDateServiceGrpc.RPCD  
14     @Override  
15     public void getDate(RPCDateRequest request, StreamObserver<  
16         //请求结果, 我们定义的  
17         RPCDateResponse rpcDateResponse = null;  
18         //  
19         String userName = request.getUserName();  
20         String response = String.format("你好:%s, 今天是%s.", use  
21         try {  
22             // 定义响应, 是一个builder构造器。  
23             rpcDateResponse = RPCDateResponse.newBuilder()  
24                 .setServerDate(response)  
25                 .build();  
26             //int i = 10/0;  
27         } catch (Exception e) {  
28             responseObserver.onError(e);  
29         } finally {  
30  
31             responseObserver.onNext(rpcDateResponse);  
32         }  
33  
34         responseObserver.onCompleted();  
35  
36     }  
37 }  
38
```

定义服务端

Java | 复制代码

```
1  package com.chenj;  
2  
3  
4  import io.grpc.Server;  
5  import io.grpc.ServerBuilder;  
6  
7  
8  import java.io.IOException;  
9  
10  
11  public class GRPCServer {  
12      private static final int port = 9999;  
13  
14  
15      public static void main(String[] args) throws IOException,  
16          //设置service端口  
17          Server server = ServerBuilder.forPort(port)  
18              .addService(new RPCDateServiceImpl())  
19              .build().start();  
20  
21      System.out.println(String.format("GRpc服务端启动成功，端  
22  
23          server.awaitTermination();  
  
      }  
  }
```

定义客户端

Java | 复制代码

```
1 package com.chenj;  
2  
3 import com.chenj.grpc.api.RPCDateRequest;  
4 import com.chenj.grpc.api.RPCDateResponse;  
5 import com.chenj.grpc.api.RPCDateServiceGrpc;  
6 import io.grpc.ManagedChannel;  
7 import io.grpc.ManagedChannelBuilder;  
8  
9 public class GRPCClient {  
10     private static final String host = "localhost";  
11     private static final int serverPort = 9999;  
12     public static void main(String[] args) {  
13         //1, 拿到一个通信channel  
14         ManagedChannel channel = ManagedChannelBuilder.forAddress(  
15             host, serverPort).usePlaintext().build();  
16  
17         try {  
18             //2. 拿到stub对象  
19             RPCDateServiceGrpc.RPCDateServiceBlockingStub rpcDa  
20             RPCDateRequest rpcDateRequest = RPCDateRequest.newBuilder(  
21                 .setUserName("JACK")  
22                 .build());  
23             //3, 请求  
24             RPCDateResponse rpcDateResponse = rpcDateService.ge  
25             //4, 输出结果  
26             System.out.println(rpcDateResponse.getServerDate());  
27         } finally {  
28             // 5. 关闭channel, 释放资源.  
29             channel.shutdown();  
30         }  
31     }  
32 }  
33 }  
34 }
```

然后先启动Server:

再启动Client:

可以看到执行成功。一个简单的gRPC helloworld工程就搭建好了。

7030b0d090b8.svg&title=%E9%9B%B6%E5%9F%BA%E7%A1%80%E5%85%A5%