# ProtoBuf学习和使用指南 - 董德钰

# 友好开放的ProtoBuf

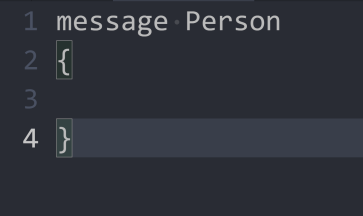
ProtoBuf是Google公司开发的一款与语言无关、与平台无关的通信协议，是一种可扩展的用于序列化和结构化数据的协议，常用于高负载网络通讯环境。

故在偏向快速序列化和反序列化数据，快速、大量传输二进制数据的网络通讯场景中，ProtoBuf举着“拼爹”大旗，一直凭实力占据着一席之地。

## ProtoBuf的语法

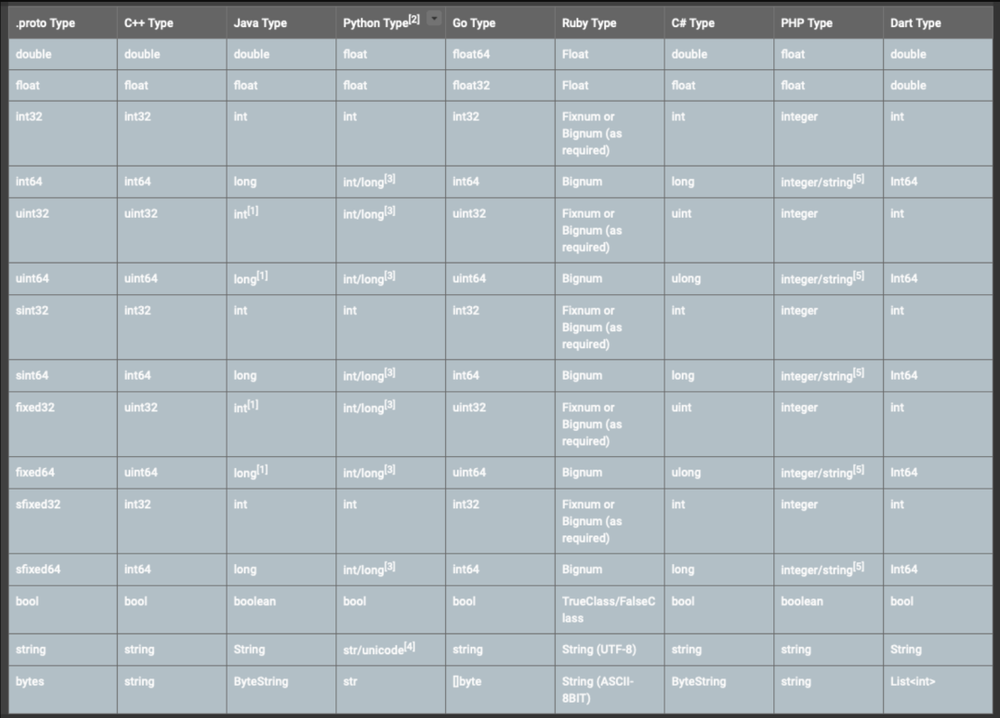
ProtoBuf为了成为一款与语言无关、与平台无光的通讯协议，自身定义了一种“表示语言”，用于表示传输的数据结构，这样便脱离了终端平台和平台语言的控制。我们使用这一中间语言，便可以约定通信双端收发过程中序列化和反序列化的规则。

对于一个在网络传输上的数据包，我们首先要定义一个Message，这个Message便是我们在网络上传输的被序列化的数据包。当然这个数据包要有一个自己的名字，像类名一样：



之后我们要在数据包中定义各种不同数据类型的数据，这些数据中，有些在数据包中一定存在（required）,有些在数据包中至多存在一次（optional），有些在数据包中存在次数未知（repeated），所以数据包中一种数据的声明格式为：

**字段规则 字段类型 字段名 = 字段编号;**



PS：

* 1. 字符串默认值为空字符串。

* 1. 字节类型默认值是空字节。

* 1. 布尔类型默认值为 false。

* 1. 数值类型默认值为 0。

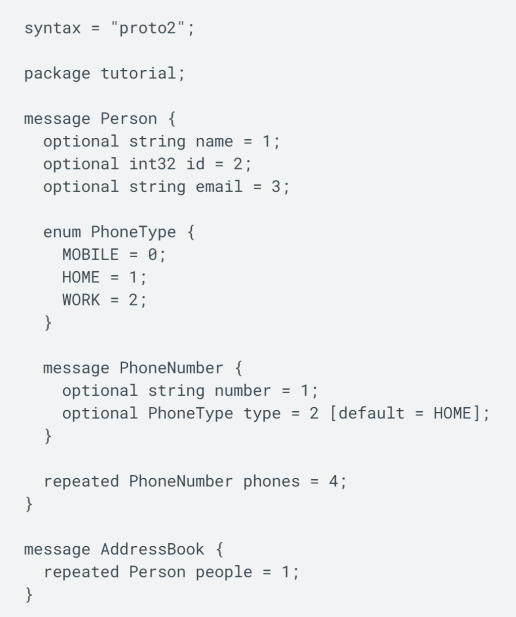
* 1. 枚举类型默认值是其定义中的第一个值，它必须为 0。

* 1. 消息类型的默认值没有设置。它的具体值与使用的编程语言有关。

* 1. repeated字段的默认值为空（通常是相应编程语言的空列表）。



当然 数据类型 可以是编程语言中的基本数据类型，也可以是一个之前定义过的message结构，其中字段编号和字段名在一个message中都应该保证是唯一的：



上述.Proto文件中，package关键字意思是定义当前文件的包名，proto文件的包名可以看作C++ .h/.cpp文件的namespace，主要用于防止文件中的类型名和生成的接口名与include的文件中的名称重复。Syntax关键字用于声明文件使用的proto版本，默认使用Proto3。

PS：

* 1. proto中的枚举值定义结尾用;而不是用,

* 1. 枚举类型中必须包含值为0的元素，这样我们才可以使用0作为数值型字段的默认值。这个为 0 的元素必须是第一个元素，是为了兼容 proto2 语法（proto2 中枚举类型的第一个元素总是默认值）。

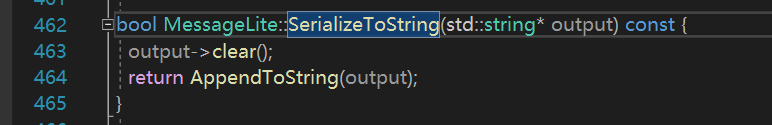
* 1. enum值是使用Varints编码方式的，对负数不够高效，因此不推荐在enum中使用负数。

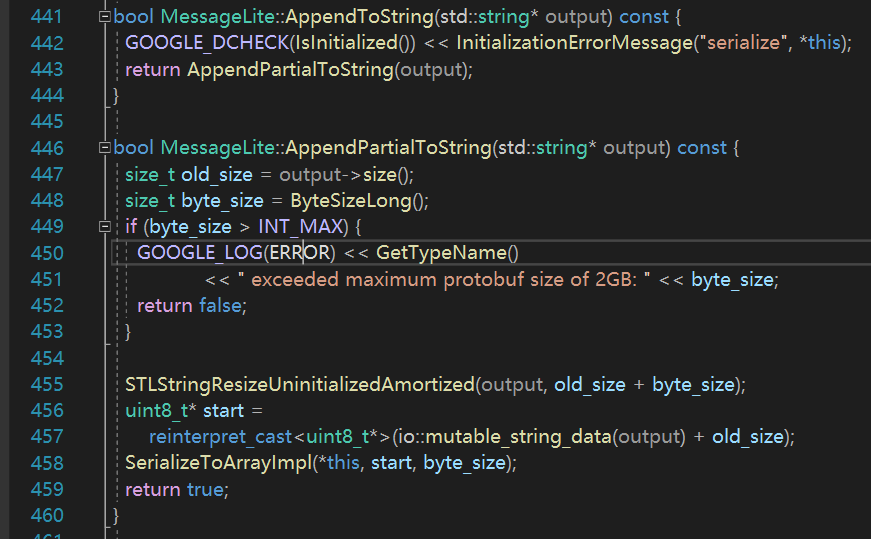
 在使用Protoc.exe生成协议接口的时候，会根据字段编号由小到大的顺序序列化数据，接口名也会是以（set\_、clear\_、has\_、mutable\_）开头＋字段名，或者直接使用字段名：

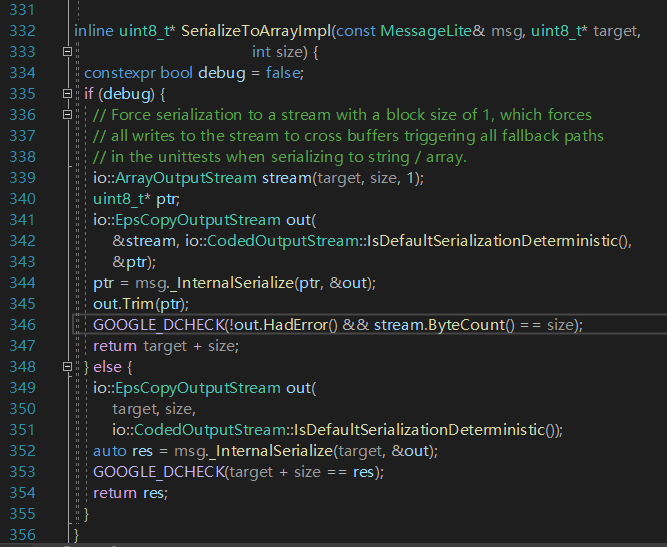
在填充完message对象后，可以使用继承的下列接口对其本身进行操作：

**bool ParseFromString(const string& data)**：对传入的序列化字符串解析，并将结果缓存到自身数据结构。

**bool SerializeToString(string\* output) const**：序列化自身存储的数据，将生成的二进制字符串拷贝到传入的字符串指针指向的字符串里。







**string DebugString()**：以key-value格式化message对象内部数据结构，生成一个字符串并返回，仅用于调试。

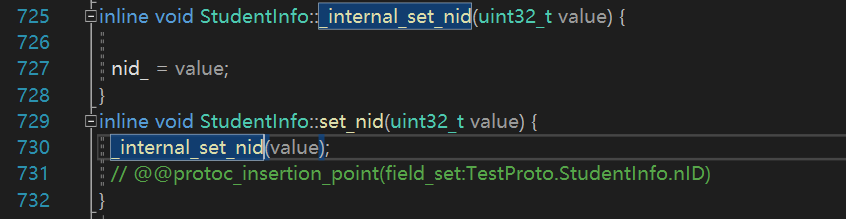
还可以使用生成的字段API接口对message的派生对象中包含的字段进行“增删改查”操作：

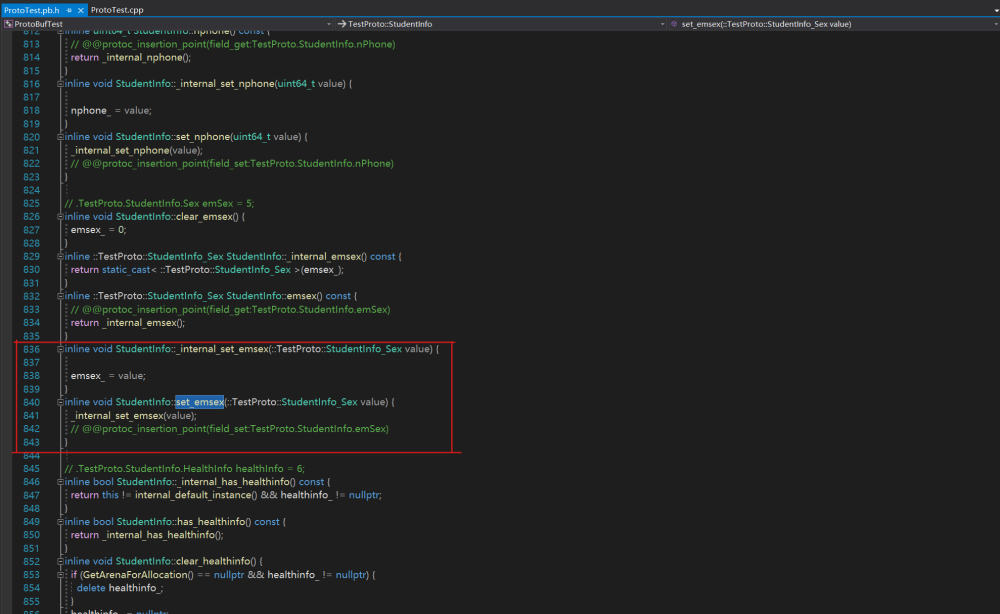
### 单一成员字段操作API

**bool has\_字段名() const：**如果这个字段被赋值，则返回true。在Proto3中被废弃。

**[const ]字段类型[&] 字段名() const：**返回这个字段的当前值，如果没有被赋值则返回默认值。

**void set\_字段名([const ]字段类型[&] value)：**设置字段值为传入的常量值。调用这个函数之后，**has\_字段名();** 将会返回true，且**字段名();** 将会返回这个value值的拷贝。

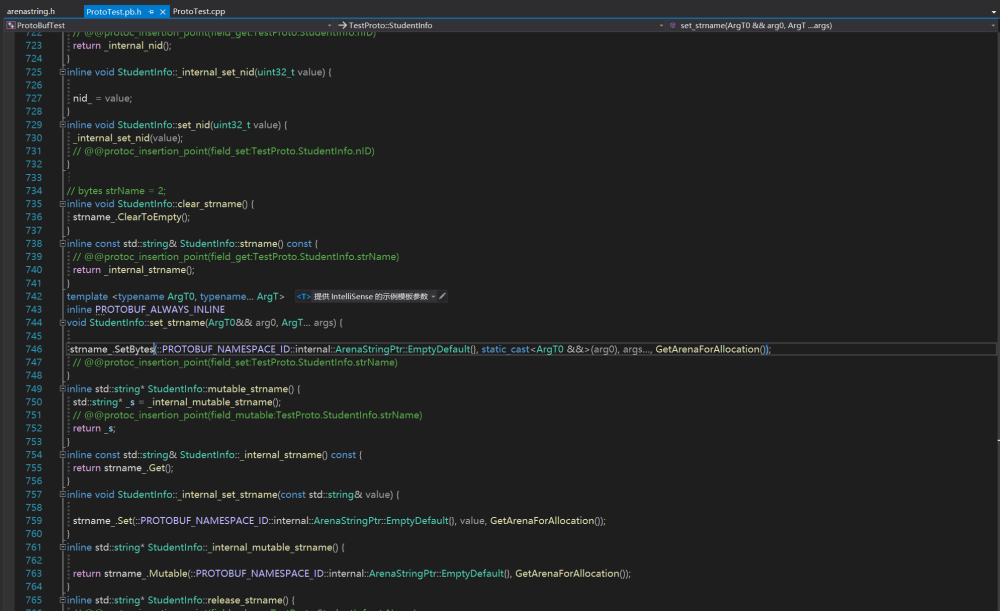


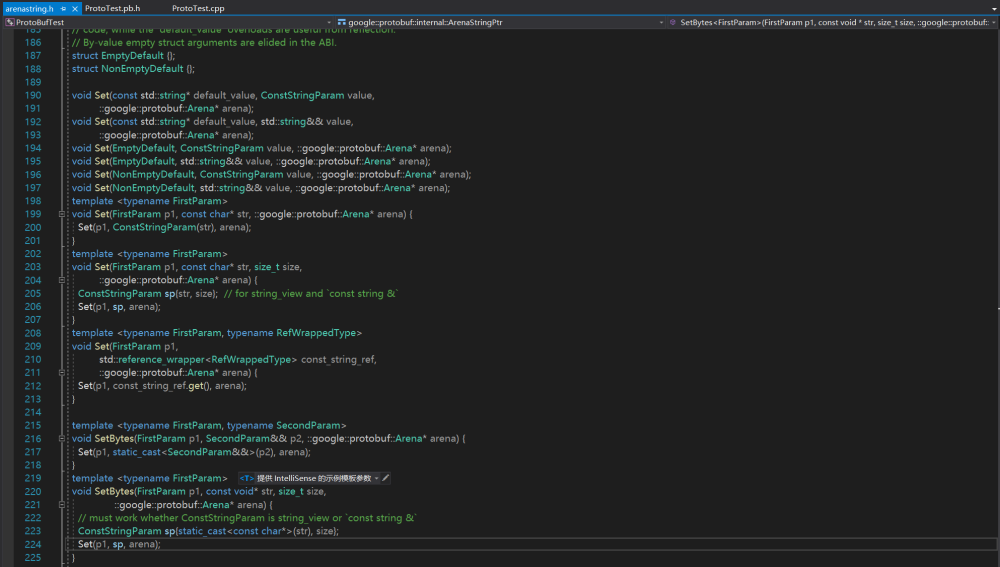




**void set\_字符串字段名(string&& value)：**C++11及以上版本支持的接口。设置字段值为传入的右值。调用这个函数之后，**has\_字段名();** 将会返回true，且**字段名();** 将会返回这个value值的拷贝。

**void set\_字符串字段名(const char\* value)：**设置字段值为传入的C语言风格以空字符（\0）结尾的字符串。调用这个函数后，**has\_字段名();** 将会返回true，且**字段名();** 将会返回这个value值的拷贝。



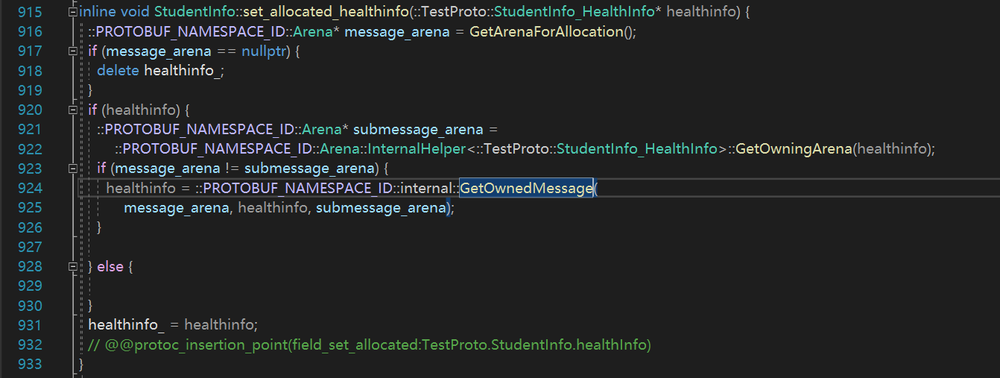


**void set\_字符串字段名(const char\* value, int size)**：同上，只是传入的字符串的长度被显式限定，不需要再去查找结尾的空字符（\0）。

**string\* mutable\_字符串字段名()**：返回一个可变字符串的指针，如果这个字段没有被赋值此函数就被调用了，则返回空（不返回默认值）。调用此函数之后，**has\_字段名();** 将会返回true，且**字段名();** 将会返回该字段被写入的值。

**void clear\_字段名()**：清空这个字段值（释放指向对象的内存，指针置空）。调用此函数后，**has\_字段名();** 将会返回false，且**字段名();** 将会返回该字段数据类型的默认值。

**void set\_allocated\_字符串字段名(string\* value)**：设置一个字符串对象的值为字段值，并释放之前的字段值的内存（如果它存在）。如果传入的字符串指针不是空，则message就接受它所指向的字符串对象的值，且**has\_字段名();** 将会返回true。message将会释放删除之前存在的字符串对象的内存，所以索引之前缓存的字符串对象可能会失败。但是，如果value为空，则message会直接调用该字段的**clear\_字段名();** 。



**string\* release\_字符串字段名()**：放弃字段值的所有权（指针置空）并返回这个字符串对象的指针。调用这个函数之后，调用处会获得这个已分配字符串对象的所有权，且**has\_字段名();** 将会返回false，**字段名();** 将会返回该字段数据类型的默认值。

### 重复成员字段操作API

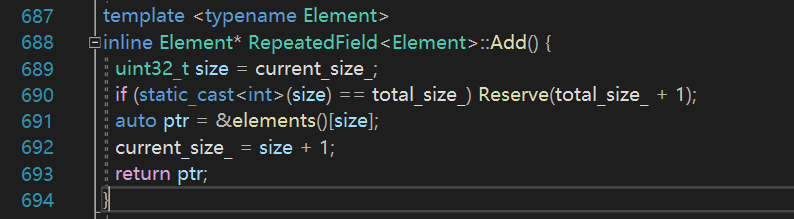
**int 字段名\_size() const**：返回该重复字段当前包含的元素总个数。

**[const ]字段类型[&] 字段名(int index) const**：返回字段中第index个元素（从0开始数）。传入的index值在[0,字段名\_size())范围之外时，会导致未定义错误。

**void set\_字段名(int index, [const ]字段类型[&] value)**：设置字段中第index个元素（从0开始数）的值为value。

**void add\_字段名([const ]字段类型[&] value)**：添加一个新的元素到字段尾部，值为value。





**void clear\_字段名()**：移除该字段所有元素。调用这个接口后，字段名\_size(); 将会返回0。

**const RepeatedField<字段类型>& 字段名() const**：返回一个存储该字段所有元素的底层容器。这个容器提供类STL的迭代器和其他方法。

**RepeatedField<字段类型>\* mutable\_字段名()**：返回一个指向存储该字段所有元素的底层容器的指针。这个容器提供类STL的迭代器和其他方法。

**void set\_字符串字段名(int index, const char\* value)**：设置字段中第index个元素（从0开始数）为value字符串（一个C风格的以空字符（\0）结尾的字符串）。

**void set\_字符串字段名(int index, const char\* value, int size)**：同上，只是替换的value字符串被显式限定了长度，不再必须以空字符（\0）结尾。

**void add\_字符串字段名(const char\* value, int size)**：添加一个新的元素到字段尾部，值为value，长度限定为size。

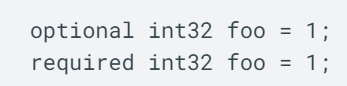
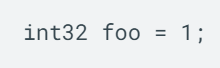
**字段类型\* add\_字段名()**：添加一个新元素到字段尾部，并返回一个指向这个新添加元素的指针。

## Proto2到Proto3的变化

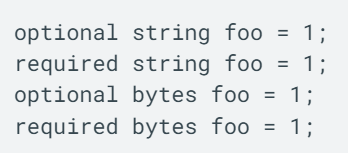
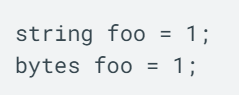
### 字段规则的变化

Proto2的字段规则有三种：required、optional、repeated，Proto3废弃了之前的required和optional，取而代之的使用singular作为默认规则，这个默认规则可以隐藏，即在声明字段时不写字段规则，就默认为至多出现一次（出现0次或1次）的singular。

Proto2： Proto3:

Proto2： Proto3：

### 生成接口的变化

Proto2到Proto3常用接口基本没有什么太大的变化，但是Proto3删除了Proto2中的返回成员是否被设置的接口：**bool has\_字段名() const**

### 字段类型的变化

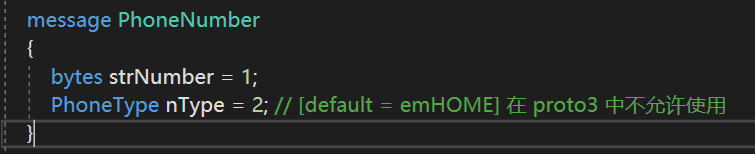
Proto2中的extensions在Proto3中将会被google.protobuf.Any替代，用于存储一段任意字节的二进制序列化数据。

PS：Any 类型的运行时库还在开发中。使用Any类型要

import google/protobuf/any.proto。

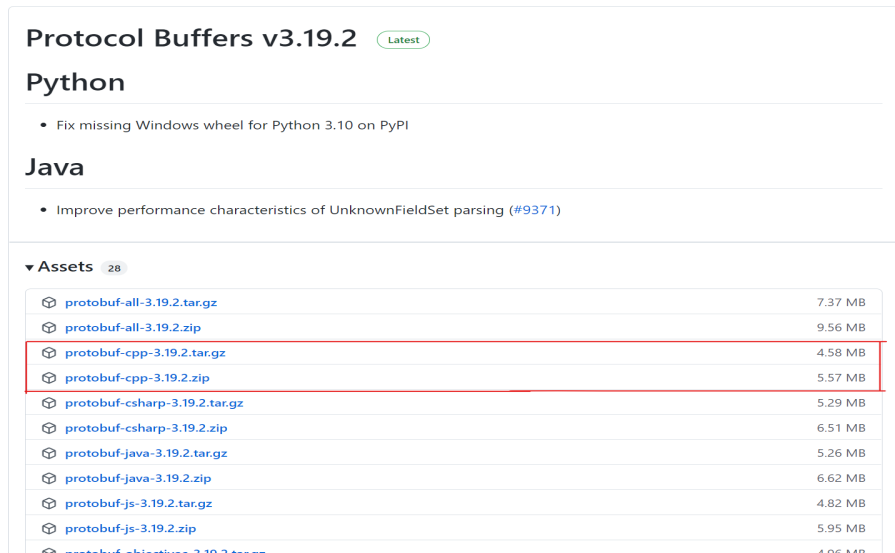
### 字段选项的变化

Proto2中的字段选项default，用于声明字段默认值，在Proto3中部分被废弃了：



## ProtoBuf的安装与使用

首先到GitHub上下载Protobuf的cpp源码包，有两种压缩格式，选一种下载就行：



如果只需要protoc.exe这个编译.proto文件生成编程语言接口的编译工具，可以直接下载对应平台的protoc压缩包（应该都是release版本，不知道编译出来的接口文件能不能用于debug）：



安装CMake（<https://cmake.org/download/>），使用CMake生成VS2019可加载的X64平台运行的项目。

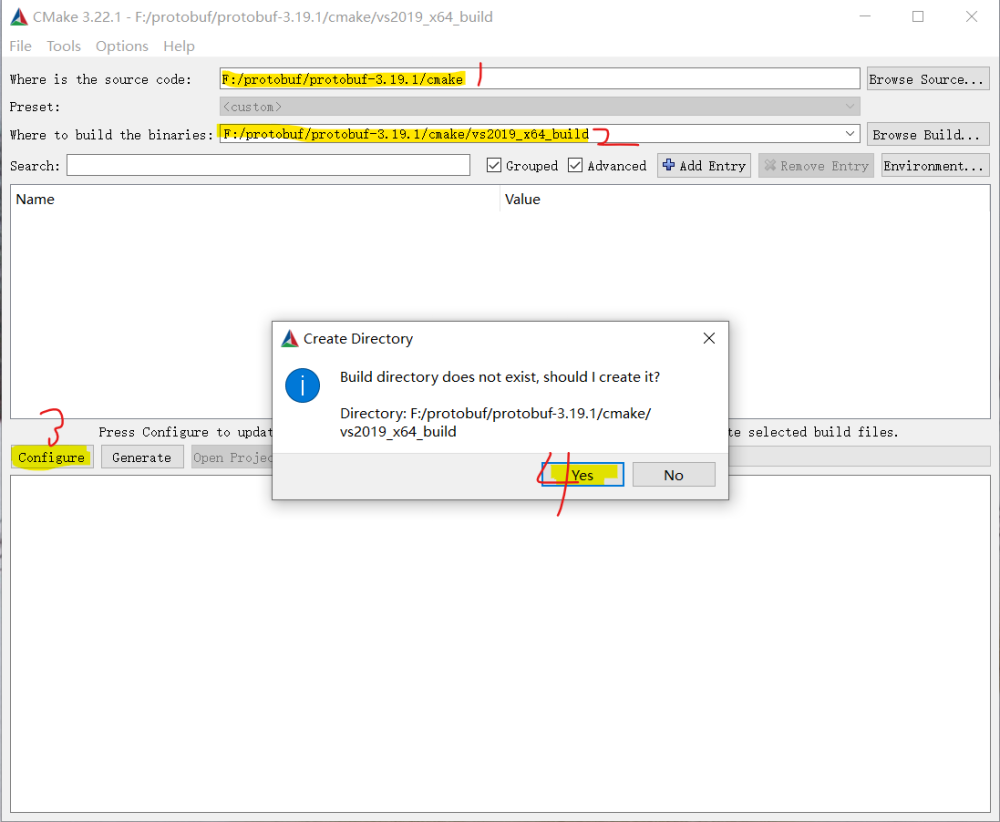
PS：

* 1. 项目文件中的CMake配置目录

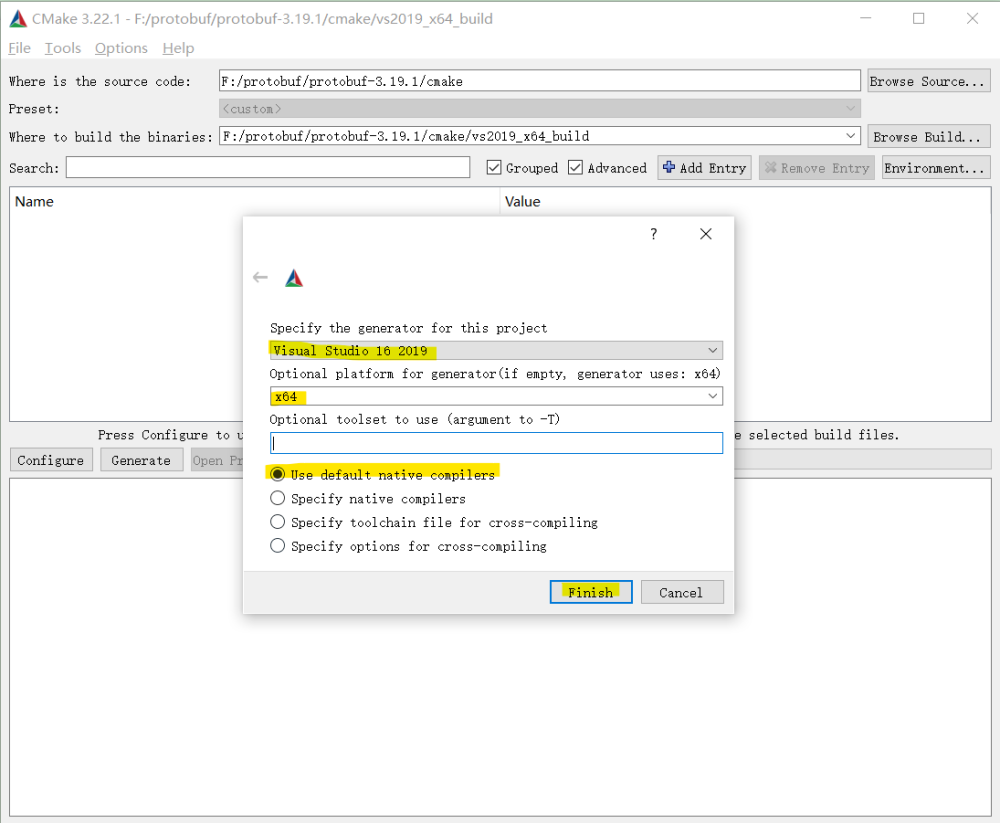
* 1. 构建生成的项目文件存储的位置（目录名可自定义）

* 1. 先解析CMake配置文件，明确需要生成的目录和生成顺序

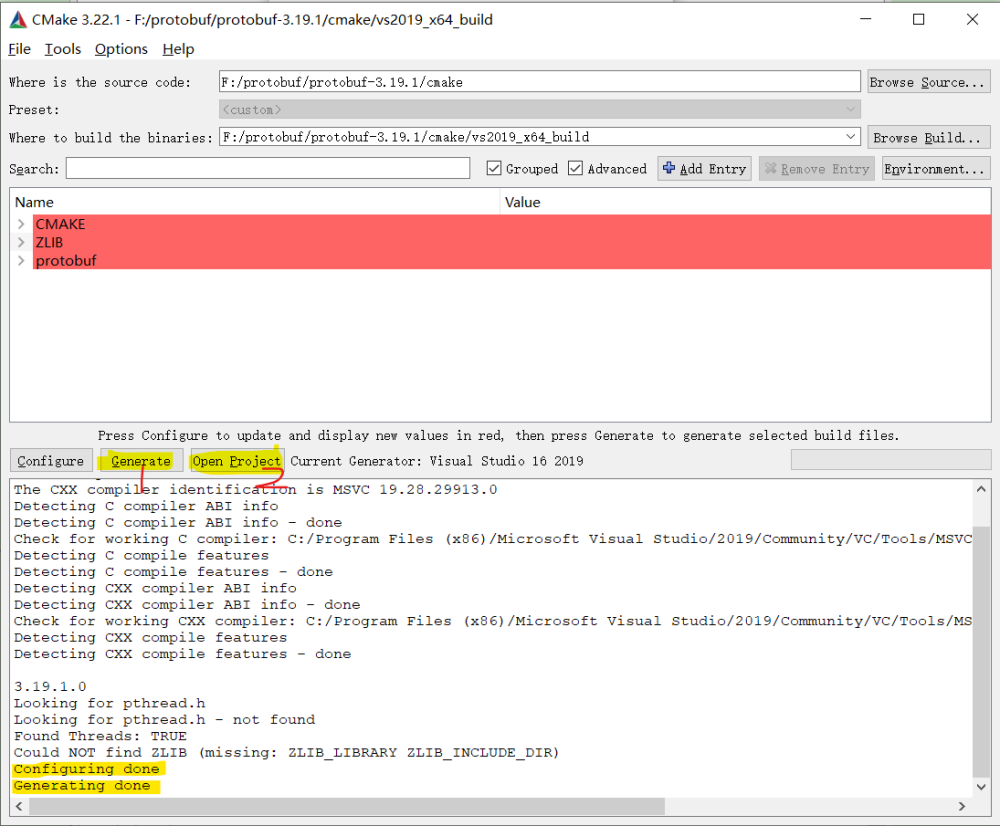
* 1. （如果第2点输入的目录不存在，则需要确认是否自动创建的输入目录）



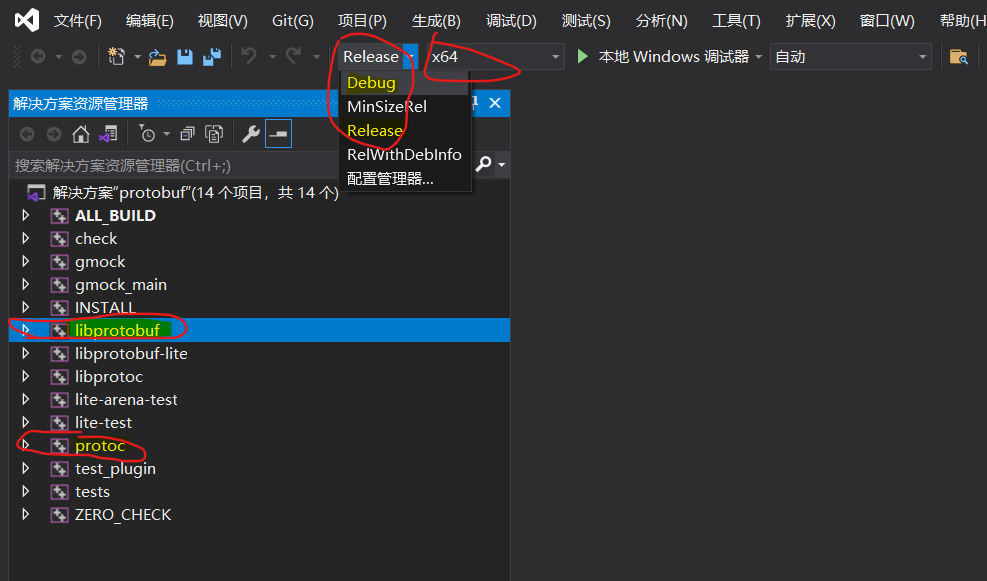
选择对应的VS版本（2019），以及平台类型（X64），使用本地默认编译器构建项目，点击Finish：

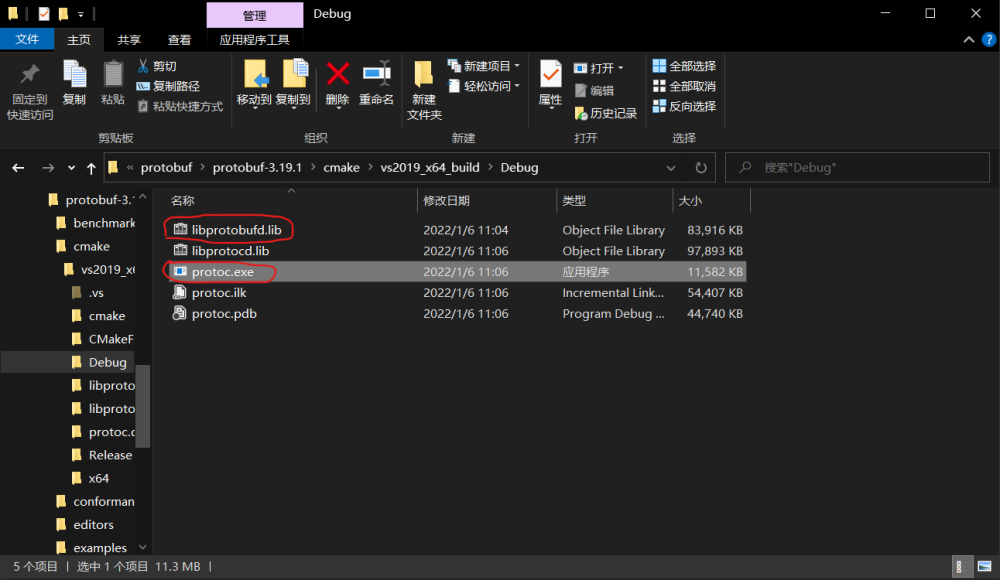


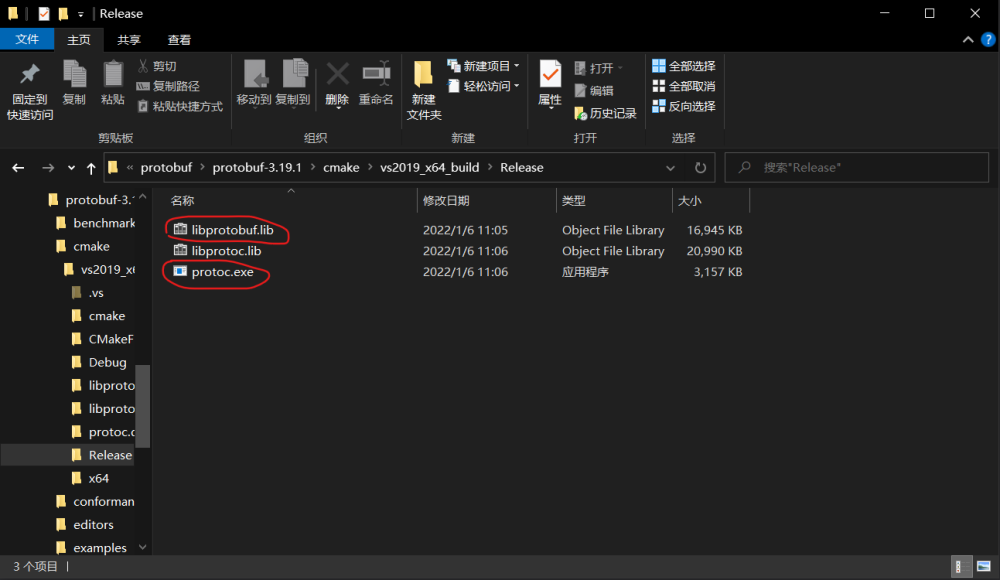
解析CMake配置完成后，点击Generate开始构建VS项目。完成项目构建后，可以点击Open Project使用之前设置的VS平台打开构建好的项目：



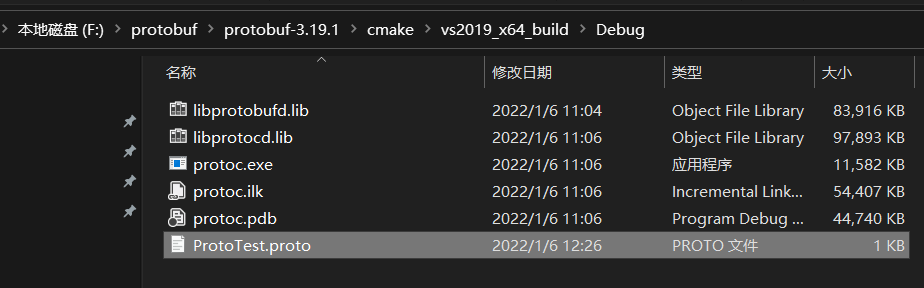
在打开的项目中，分别编译生成Debug版和Release版的x64平台的libprotobuf和protoc项目（protoc项目编译生成的protoc.exe可以在GitHub上直接下载）:







在protoc.exe所在目录创建一个.proto文件，以proto3语法定义一个包含两个message的包：



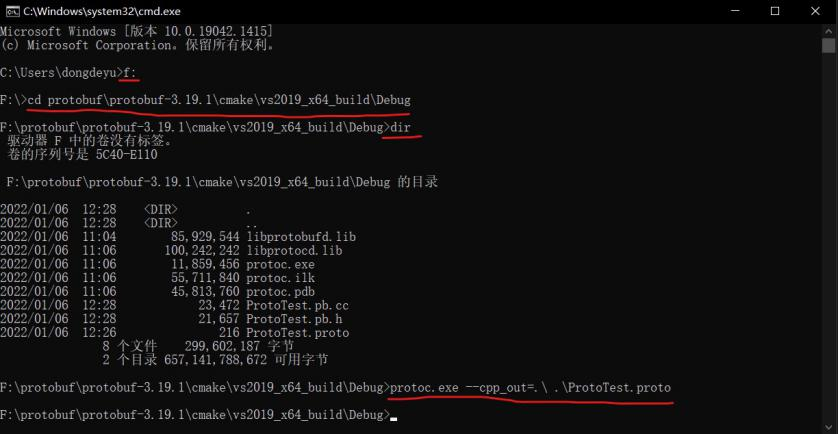


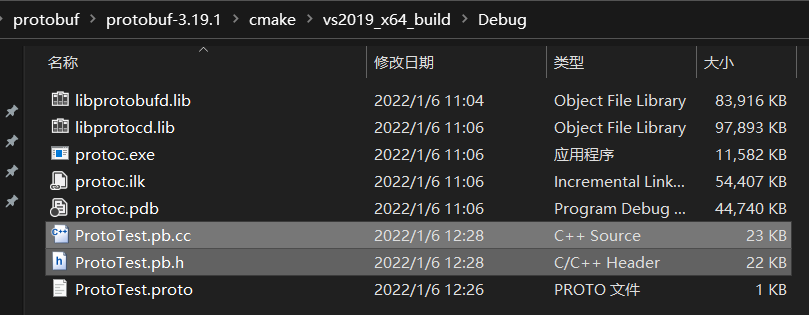
使用windows键+r，调出运行，输入cmd，调出控制台。输入 盘符(c|d|e|f): ，回车切换到proto文件的盘符，使用dir命令查看当前目录下的文件目录列表，使用cd命令跳转到proto所在的目录，调用protoc.exe将当前目录下的proto文件编译生成cpp接口文件：

**protoc.exe [-I=$(ProtoFileDir)] --cpp\_out=$(OutputDir) {$(ProtoFile) ......}**

如果protoc.exe所在目录被加入环境变量，也可在任意目录直接使用命令生成接口文件：

**protoc [-I=$(ProtoFileDir)] --cpp\_out=$(OutputDir) {$(ProtoFile) ......}**

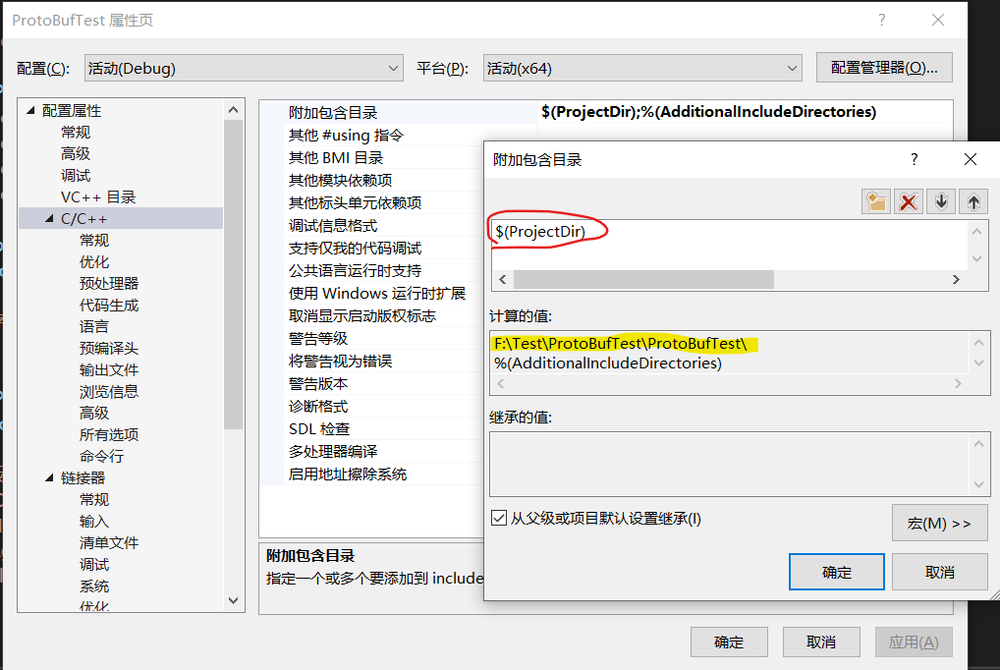




PS：

编译生成.proto文件的CPP接口文件可以使用上传项目中的批处理程序protoc.bat，快速生成所有的.proto文件到一个统一的文件夹。只需要在protoc.bat所在目录创建一个ProtoBuf文件夹，并把所有需要编译的.proto文件放在ProtoBuf文件夹内，运行protoc.bat，就会在protoc.bat所在目录下的Common文件夹内生成编译好的CPP接口文件（没有Common文件夹会自动创建Common文件夹）。

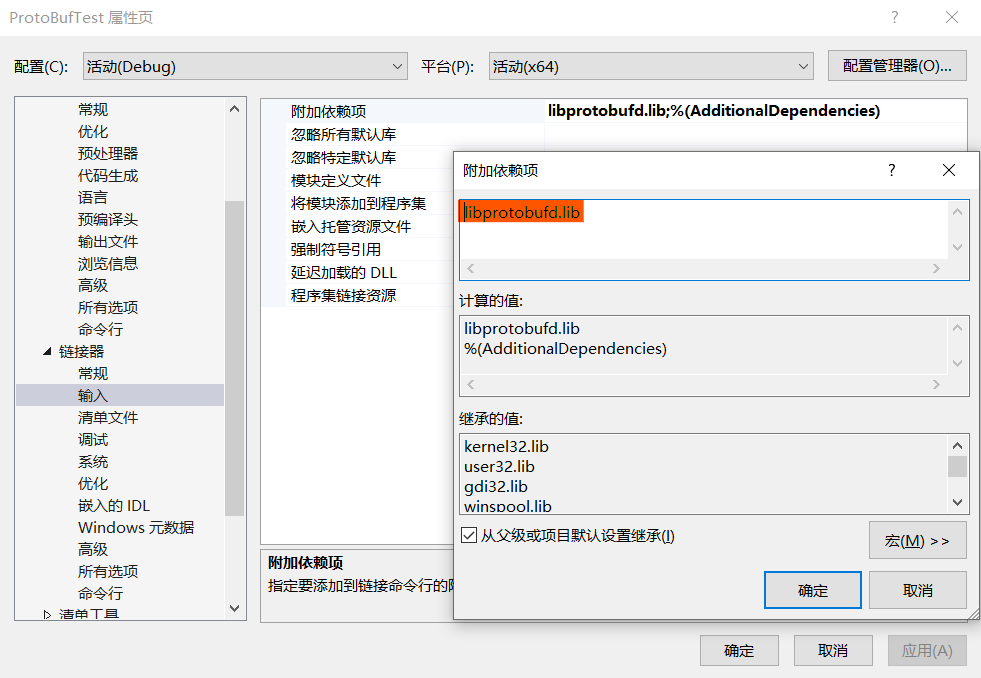
将生成的接口文件移动到项目目录中，并在项目附加包含目录中添加项目目录$(ProjectDir)：



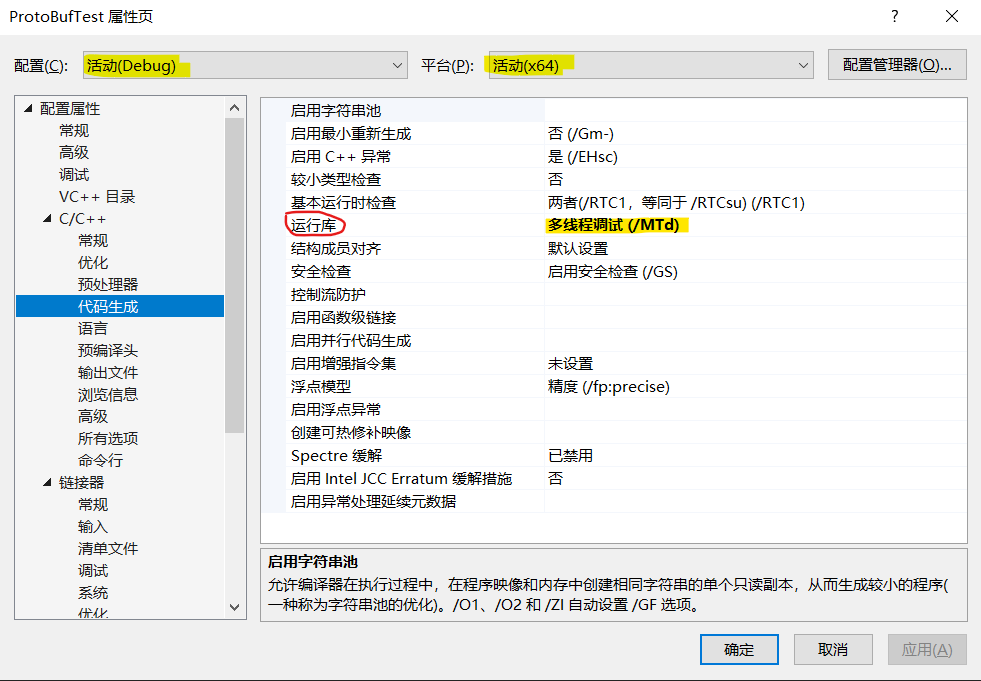
将之前编译生成的libprotobufd.lib（Relearse下是libprotobuf.lib）所在的目录添加到项目属性的附加库目录中去：



将项目使用的附加库目录中的libprotobufd.lib（Relearse下是libprotobuf.lib）添加到附加依赖项中：

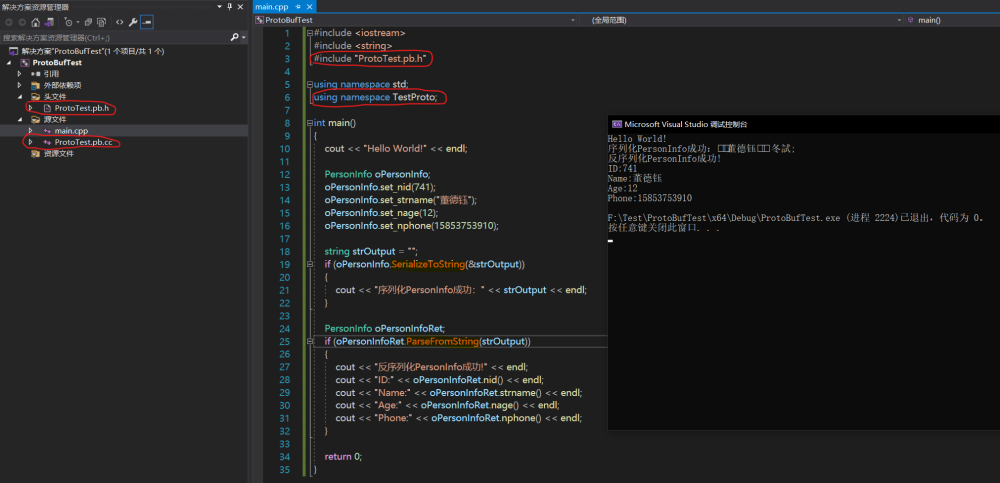


因为使用的是lib静态库，所以还要检查一下运行库是否是lib库的运行模式，Debug是（/MTd），Release是（/MT）：



PS：不同模式（Debug、Release）和平台（x64、x86）各自使用自己的一套配置，配置不通用，所以要看好平台和模式再修改项目属性。

最后将生成的接口文件添加到项目头文件、源文件目录，再include接口文件，并且using之前在proto文件中定义的 包名 命名空间，就可以使用生成的message类型创建对象，调用接口填充对象、序列化、反序列化：



### 单一不重复数据序列化与反序列化测试

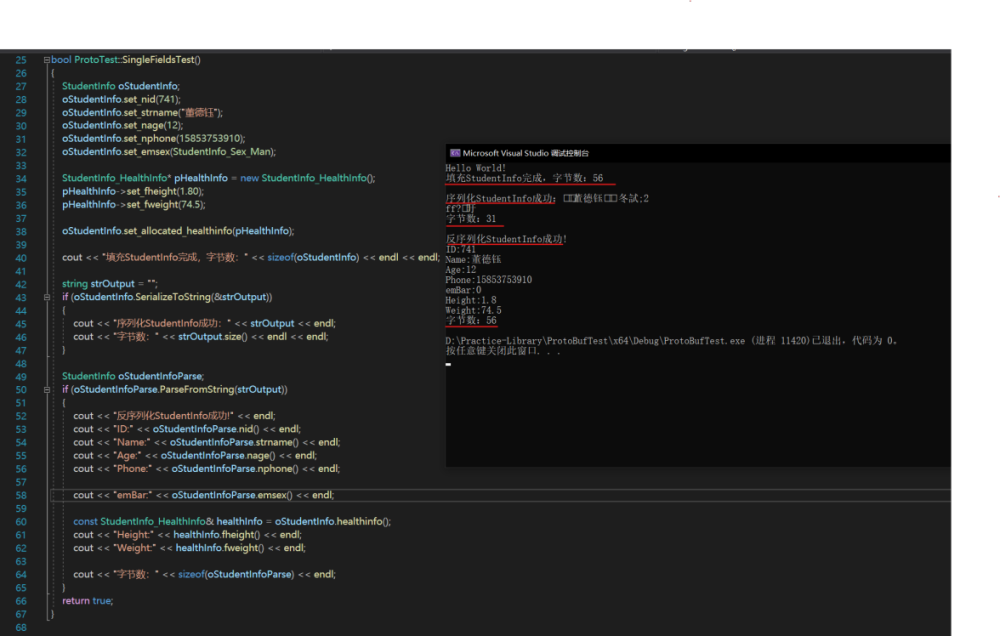
定义的ProtoTest.proto如下：



使用protoc.exe编译生成的C++接口文件：

protoc.exe --cpp\_out=.\ .\ProtoTest.proto

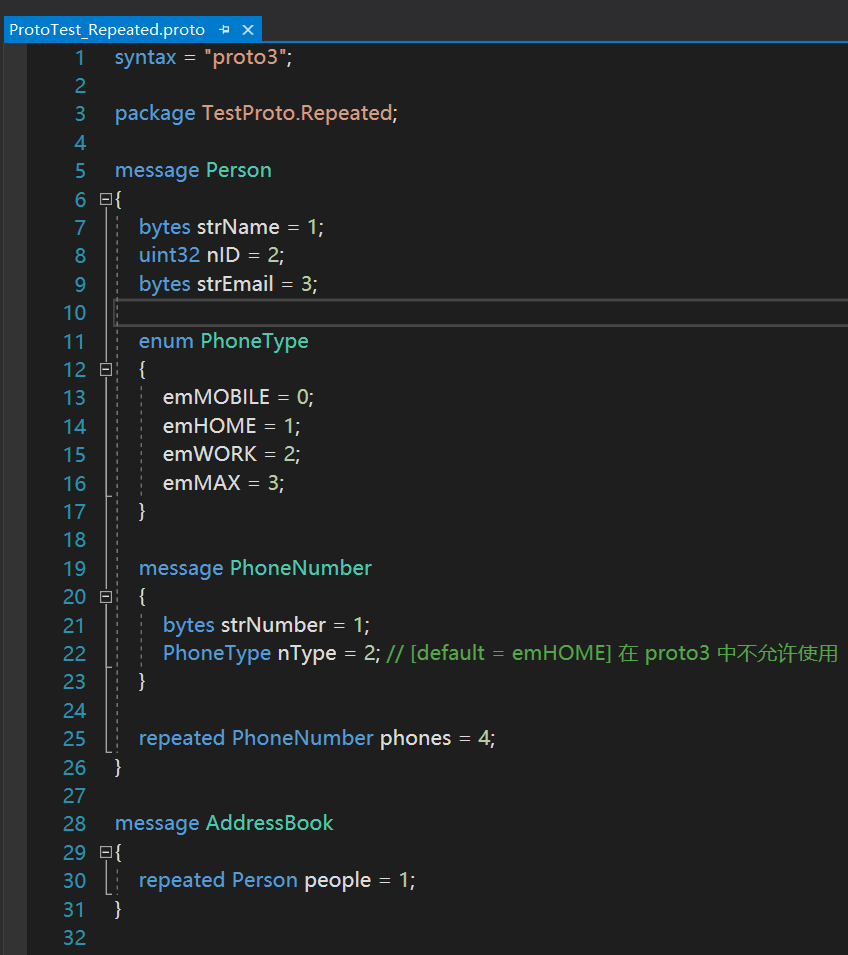
将生成的ProtoTest.pb.h和ProtoTest.pb.cc复制到项目目录（在附加包含目录中有项目目录$(ProjectDir)的情况下），在编译环境中将这两个文件添加到源文件和头文件，编写测试代码，运行如下：



根据上图运行输出可以看到经过本次序列化，**字节数缩小了近50%**。

### 复合数据序列化和反序列化测试

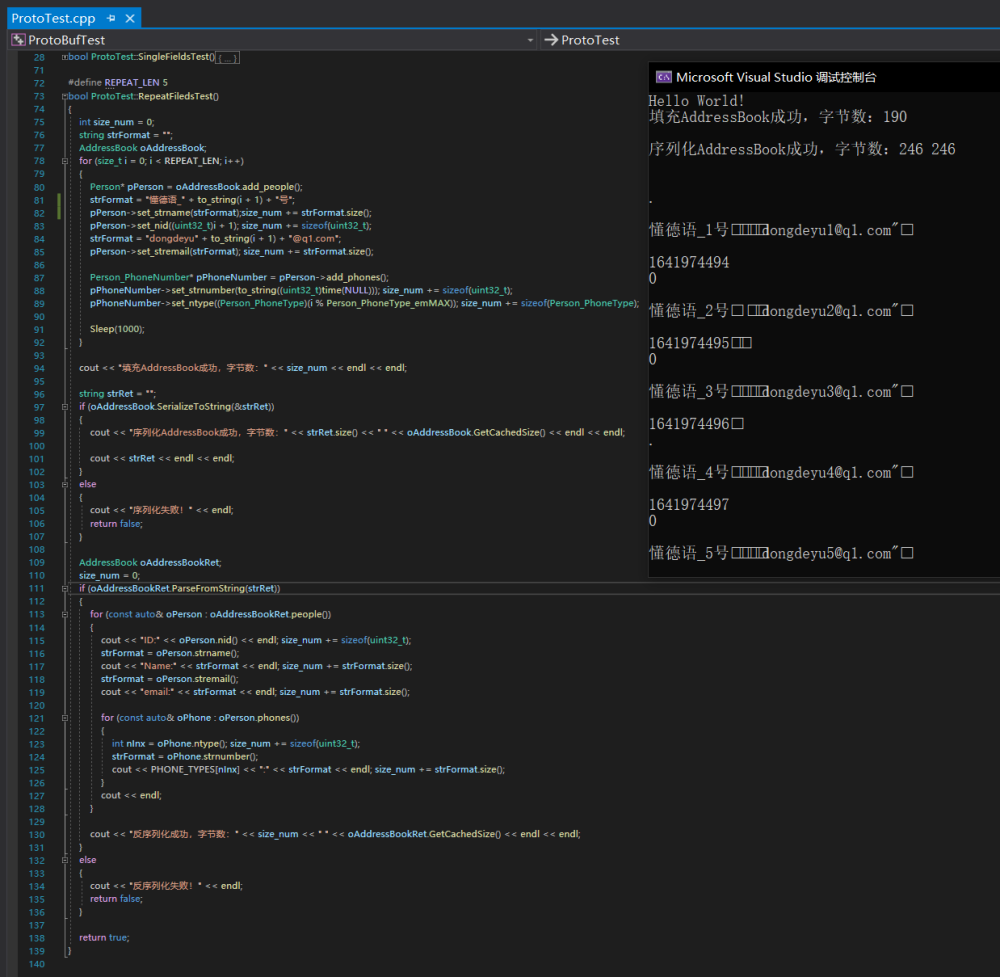
定义ProtoTest\_Repeated.proto如下：

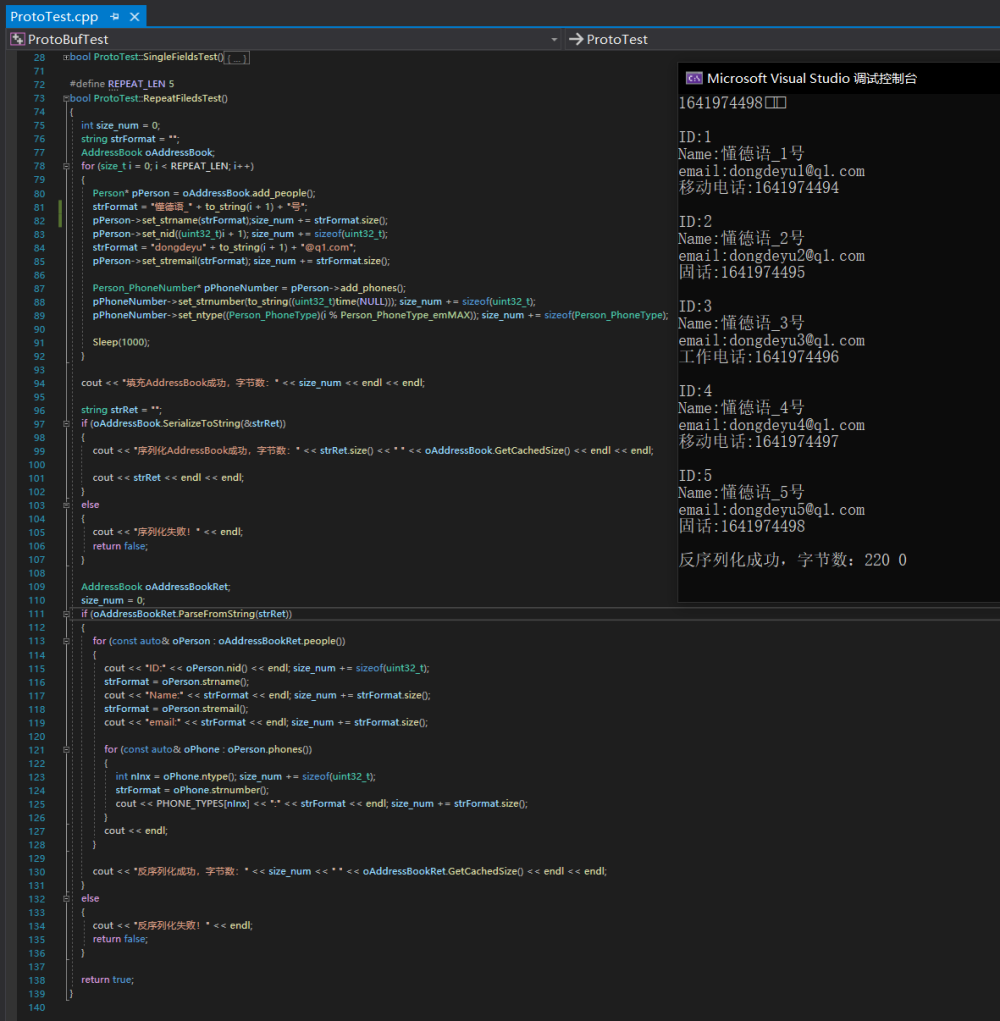


使用protoc.exe编译生成C++接口文件：

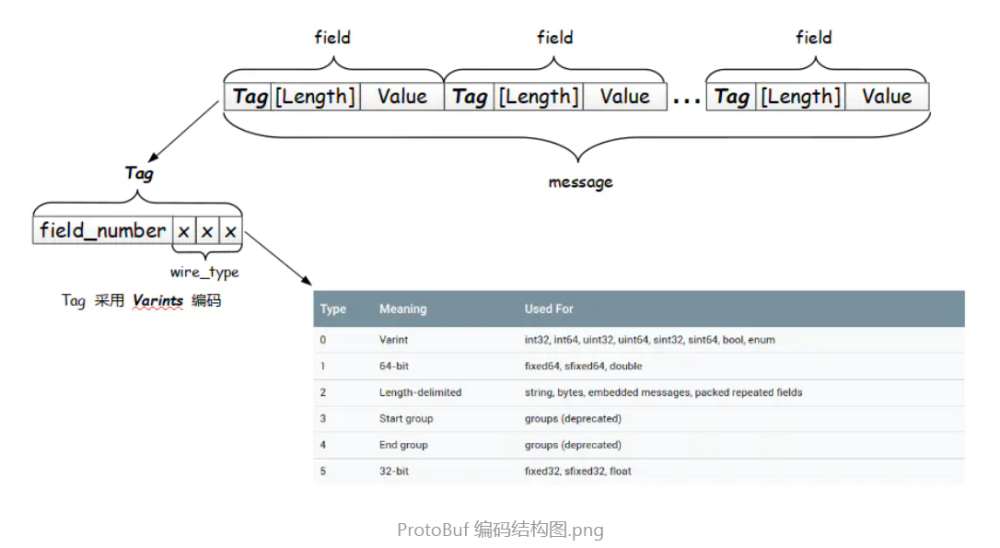
protoc.exe --cpp\_out=.\ .\ProtoTest\_Repeated.proto

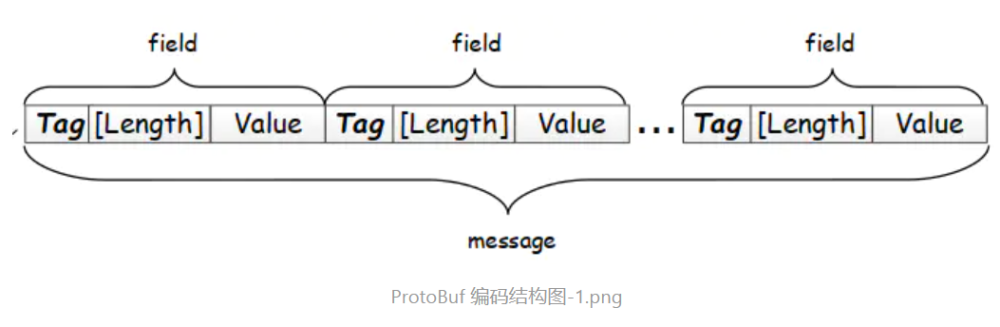
编写如下测试函数并运行：

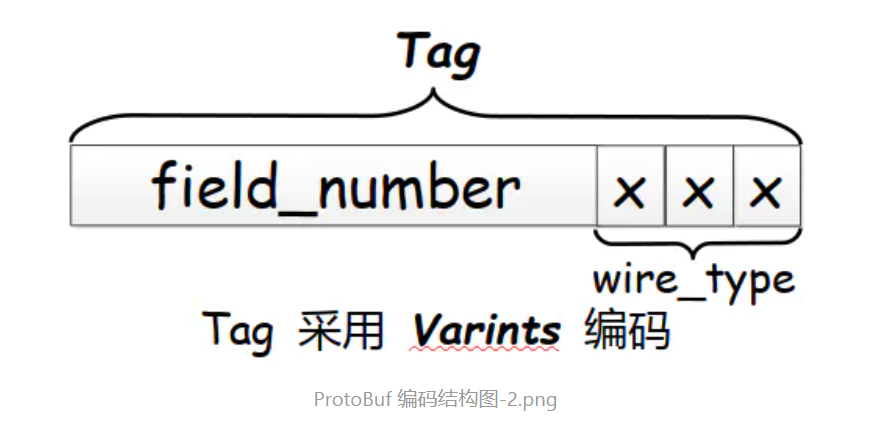




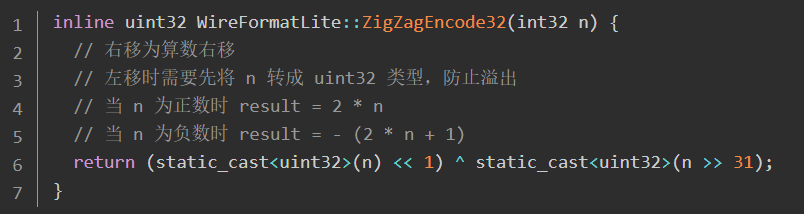
## ProtoBuf的编码原理











**详见**

<https://www.jianshu.com/p/73c9ed3a4877>

## 测试项目和批处理程序

[📎protobuf-cpp-3.19.1.zip](https://q1doc.yuque.com/attachments/yuque/0/2022/zip/12677111/1642408407090-43a57514-76ba-46b3-b658-1bf6850248ab.zip)

[📎ProtoBufTest.zip](https://q1doc.yuque.com/attachments/yuque/0/2022/zip/12677111/1642408304601-c8c554ec-fc7b-4efd-94e5-6ab6ce301b52.zip)

[📎protoc批处理程序.zip](https://q1doc.yuque.com/attachments/yuque/0/2022/zip/12677111/1642408284440-11a6f4fe-6266-4bb6-993a-9e6c66df2111.zip)