**Protobuf学习及使用文档 V3.0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **日期** | **版本** | **状态** | **作者** | **说明** |
| 2022-4-15 | V1.0 |  | 曾日璇 | 创建文档，完成介绍及例程搭建、分析部分 |
| 2022-4-18 | V1.2 |  | 曾日璇 | 补充.proto文件编译、API分析 |
| 2022-4-18 | V1.3 |  | 曾日璇 | 新增repeated类型数据说明 |
| 2022-5-5 | V1.5 |  | 曾日璇 | 新增any类型数据说明 |
| 2022-5-6 | V1.6 |  | 曾日璇 | 新增genproto.sh脚本编译说明 |
| 2022-5-17 | V2.0 |  | 曾日璇 | 新增windows下qt工程编译 |
| 2022-5-18 | V2.1 |  | 曾日璇 | 打包工程上传git |
| 2022-5-30 | V2.5 |  | 曾日璇 | 新增msvc编译静态库的方式  补充msvc编译方式.pro文件的注意事项  Msvc工程上传Git |
| 2022-6-4 | V3.0 |  | 曾日璇 | 新增使用cmakelists管理工程  Qt使用cmake编译  生成.sln使用VS编译  Cmakelists工程一起打包上传到git |

**目录**

[一、Protobuf简介 3](#_Toc105316195)

[1.1 protobuf定义及表述 3](#_Toc105316196)

[2.2为何使用protobuf 3](#_Toc105316197)

[二、protobuf例程搭建 4](#_Toc105316198)

[2.1引入第三方库 4](#_Toc105316199)

[2.2创建protobuf 4](#_Toc105316200)

[2.3添加路径 5](#_Toc105316201)

[三、例程分析及演示 6](#_Toc105316202)

[3.1主程序调用.protubuf 6](#_Toc105316203)

[3.2例程演示 7](#_Toc105316204)

[3.3 genproto.sh脚本编译说明 7](#_Toc105316205)

[3.4主要API分析 8](#_Toc105316206)

[四、其他protobuf类型测试 9](#_Toc105316207)

[4.1 repeated类型 9](#_Toc105316208)

[4.2 Any类型 9](#_Toc105316209)

[五、Windows环境QT工程编译protobuf 13](#_Toc105316210)

[5.1环境确认 13](#_Toc105316211)

[5.2编译protobuf静态库 14](#_Toc105316212)

[5.3 生成proto.pb.cpp和.h 18](#_Toc105316213)

[5.4创建QT工程 19](#_Toc105316214)

[5.5打包QT\_protobuf工程 23](#_Toc105316215)

[六、cmakelists管理工程 24](#_Toc105316216)

[6.1 QT编译cmake工程 24](#_Toc105316217)

[6.2 VS编译cmake工程 25](#_Toc105316218)

## 一、Protobuf简介

### 1.1 protobuf定义及表述

Protocol Buffers是一种语言无关、平台无关、可扩展的序列化结构数据的方法，它可用于（数据）通信协议、数据存储等。

它是一种灵活，高效，自动化机制的结构数据序列化方法－可类比 XML，但是比 XML 更小（3~10倍）、更快（20~100倍）、更为简单。

你可以定义数据的结构，然后使用特殊生成的源代码轻松的在各种数据流中使用各种语言进行编写和读取结构数据。你甚至可以更新数据结构，而不破坏由旧数据结构编译的已部署程序。

简单来讲，Protobuf 是结构数据序列化方法，其具有以下特点：

1、语言无关、平台无关。即 Protobuf支持Java、C++、Python 等多种语言，支持多个平台；

2、高效。即比 XML 更小（3~10倍）、更快（20~100倍）、更为简单；

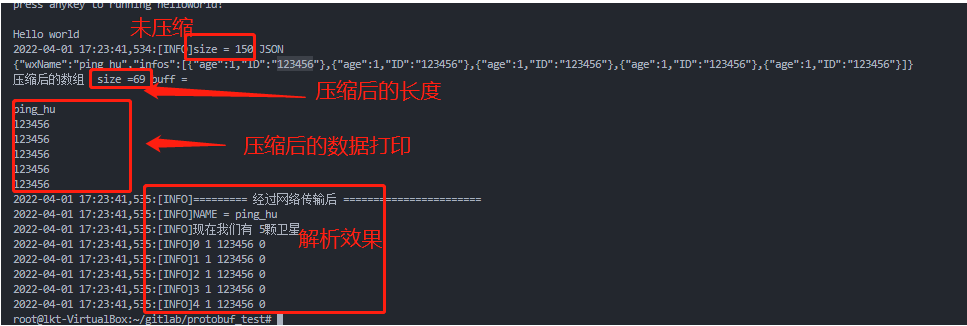
3、扩展性、兼容性好。更新数据结构同时，而不影响和破坏原有的程序。

**补充：**

**对于此处序列化的解释**：将**结构数据**或**对象**转换成**能够被存储和传输（例如网络传输）的格式**，同时应当要保证这个序列化结果在之后（可能在另一个计算环境中）能够被重建回原来的结构数据或对象。

### 2.2为何使用protobuf

一个项目执行过程中，有大量的数据需要在各个环节进行传输，太大的信号量会导致传输速率降低。我们使用protobuf最主要的目的是对每个环节传输的信号量进行压缩。简单的例子如下：



这里传输5颗卫星信息为例，在protobuf压缩前信号长度为150，它的json文件信息在下一行完整打印。当使用protobuf压缩后，信号长度压缩为69，json文件变为压缩后的打印数据进行传输。当接收端接收信息后，再通过protobuf解析，得到解析效果与原数据一致。

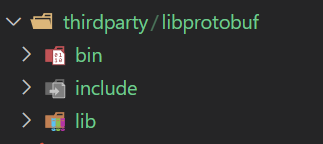
信号量越大、数据结构越统一，压缩效果越明显。

## 二、protobuf例程搭建

此处以helloword例程为基础，通过引入protobuf的第三方库，调用protobuf的Api来搭建protobuf例程，实现数据传输的压缩和解析。

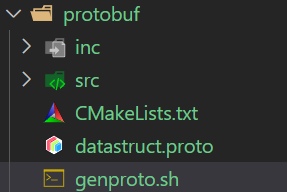
### 2.1引入第三方库

将protobuf第三方库（thirdparty）添加到工程目录，方便我们直接调用API，编译protobuf。库的内容如图所示：



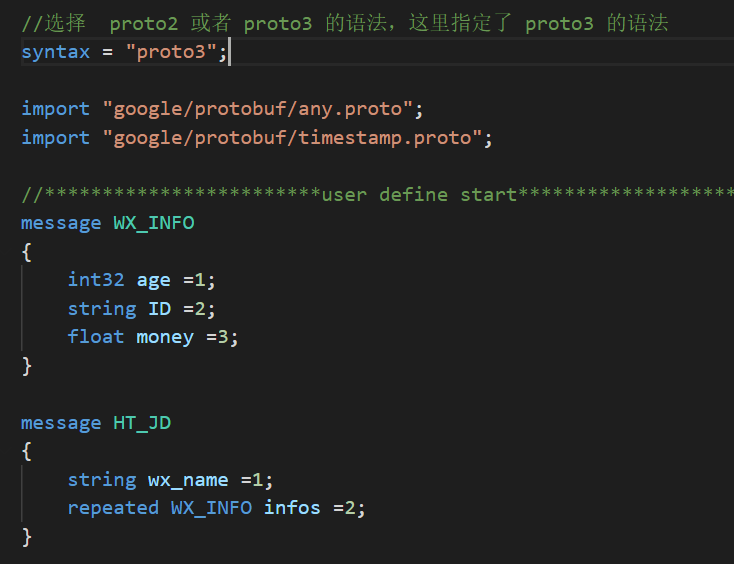
### 2.2创建protobuf

创建protobuf文件夹用于存放.proto文件、sh脚本和编译生成的源文件、头文件，如图所示：



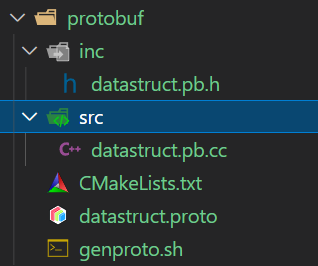
其中datastruct.protobuf文件是protobuf类型文件，用于存放定义的protobuf消息，genproto.sh脚本用于根据protobuf文件生成对应的源文件（.pb.cc）和头文件（.pb.h)，为之后主程序调用protobuf消息时使用。

1.在datastruct.protobuf文件中定义消息如下：



说明：.proto文件固定结构体类型为message，对于每个message结构体，各个结构体成员均从1开始排序。

2.执行 sh genproto.sh ，根据.proto文件生成的对应的源文件（.pb.cc）和头文件（.pb.h)分别自动存放在src和inc文件夹下：



每个.proto生成一个.h和.cc文件，其中包含文件中描述的每种message 类型对应的类。

### 2.3添加路径

引入第三方库和protobuf文件后在cmakelists中添加路径，将protobuf的头文件和源文件包含进来。添加路径如下：

include\_directories ( ${CMAKE\_PREFIX\_PATH}/libprotobuf/include)

include\_directories ( ./protobuf/inc)

aux\_source\_directory ( ./protobuf/src DIR\_USR\_SOURCE )

set\_property (TARGET protobuf PROPERTY IMPORTED\_LOCATION ${CMAKE\_PREFIX\_PATH}/libprotobuf/lib/libprotobuf.a)

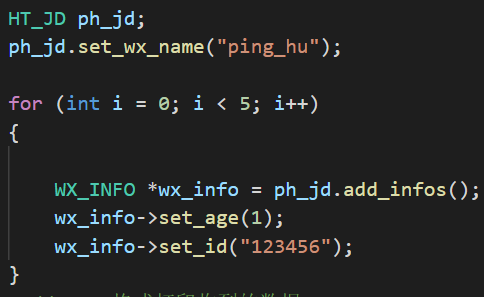
target\_link\_libraries ( ${CMAKE\_PROJECT\_NAME} pthread protobuf )

## 三、例程分析及演示

### 3.1主程序调用.protubuf

#### 3.1.1成员赋值

由于执行了sh genproto.sh，项目中能识别处相关类型的API。在包含相关protobuf路径和头文件之后，我们在main函数中调用.proto文件定义的消息，并给它赋值。如图所示：



定义成员ph\_jd为HT\_JD的结构体成员（HT\_JD结构体在.proto下定义），.set\_wx\_name是2.2步骤中生成的protobuf文件的API，用于给wx\_name赋值，此处赋值“ping\_hu”

在.proto文件中WX\_INFO这一结构体作为repeated类型嵌套在HT\_JD结构体下，用来存放N个相同类型的内容。wx\_info通过for循环存放五颗卫星的信息。

#### 3.1.2打印输出



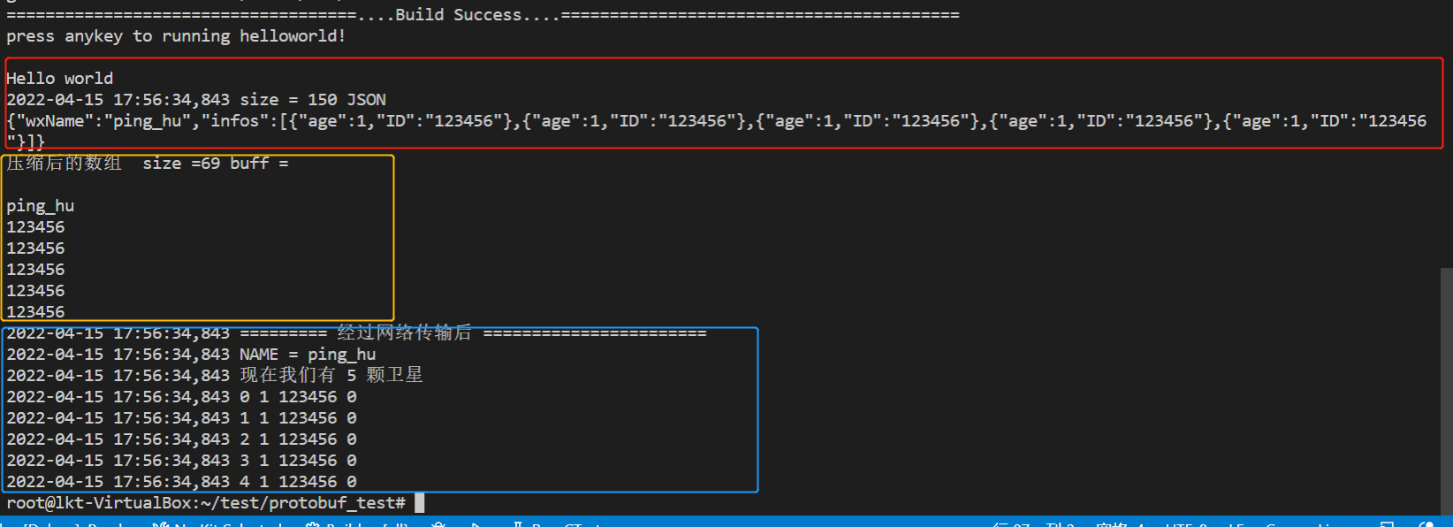
如图所示，新建了一个HT\_JD的对象为xxx\_jd用来存放解压缩后的数据，从数组中解析出消息，传给xxxx\_jd成员。通过for循环依次输出5个repeated类型的wx\_info数据。

### 3.2例程演示

1.由于更改了环境变量及相关路径，每次编译程序前先配置环境变量：source env\_linux.sh

2. 清理肯残留的编译文件：./build\_project.sh clean

3.编译程序：./build\_project.sh

编译结果：

如图所示，编译结果主要分为三部分，分别与代码一一对应。

第一部分为原信息，通过MessageToJsonString这个API将protobuf的信息转化为json格式并打印出来，json内容包括了5个卫星的信息，此时信息长度为150；

第二部分为protobuf压缩后的内容，经过protobuf压缩信息长度缩小为69，此时发送的可见信息被简化；

第三部分为信息在网络传输后，接收端接收信息，并用相同的protobuf通过ParseFromArray这一API将压缩后的信息解析出来，打印出的内容如上图所示，与此前卫星信息一一对应。

### 3.3 genproto.sh脚本编译说明

genproto.sh脚本内容如下：

../thirdparty/libprotobuf/bin/protoc --experimental\_allow\_proto3\_optional ./datastruct.proto --cpp\_out=../protobuf

mv \*.h ./inc/

mv \*.cc ./src/

代码说明：

**../thirdparty/libprotobuf/bin/protoc：**定位到该该目录下的protoc，是protoc是protobuf提供的编译器，使用protoc命令进行编译；

**--experimental\_allow\_proto3\_optional：**开启proto3的optional标签，使用proto3需要添加该选项

**./datastruct.proto --cpp\_out=../protobuf：**根据datastruct.proto文件内容编译成c++文件，存放在当前路径protobuf文件夹下

**mv \*.h ./inc/ mv \*.cc ./src/：**

分别将.h文件移动到inc文件夹，将.cc文件移动到src文件夹。

### 3.4主要API分析

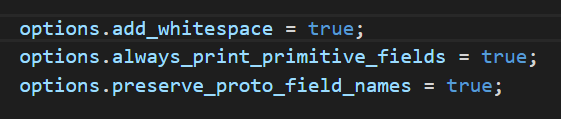
在protobuf传输的压缩和解压缩过程中，用到了以下API：

**JsonPrintOptions：**

定义json输出内容的格式传递给options，option是proto3的可选项特，性，对消息进行扩展

google::protobuf::util::JsonPrintOptions options

具体的格式设置可以更改属性的true或false实现：

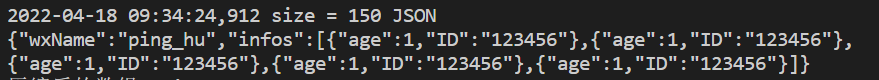


**MessageToJsonString：**

将例子protobuf消息转化为json格式，具体格式为options

MessageToJsonString(ph\_jd, &json\_string, options);

以json格式打印的信息如下所示：



**ByteSizeLong：**

在.pb.cc中自动生成，用于计算类成员的字节长度

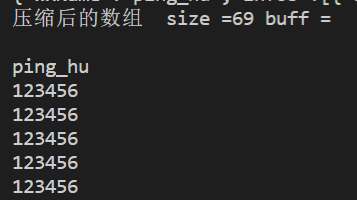
int message\_len = ph\_jd.ByteSizeLong();

**SerializeToArray：**

将消息序列化（压缩）成数组，放在cmd\_buffer数组中

ph\_jd.SerializeToArray(cmd\_buffer, message\_len);

序列化压缩后的信息如下所示，



**ParseFromArray：**

从数组中解析出消息，传给xxxx\_jd成员

xxxx\_jd.ParseFromArray(cmd\_buffer, message\_len);

[ProtoBuf序列化解析\_pocher的博客-CSDN博客\_protobuf序列化](https://blog.csdn.net/bingxuesiyang/article/details/119515335)

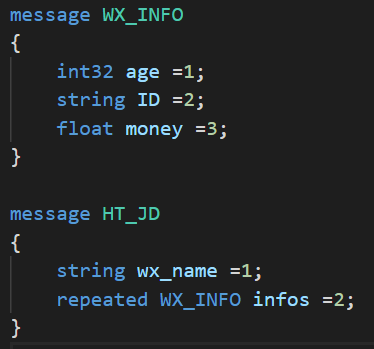
## 四、其他protobuf类型测试

### 4.1 repeated类型

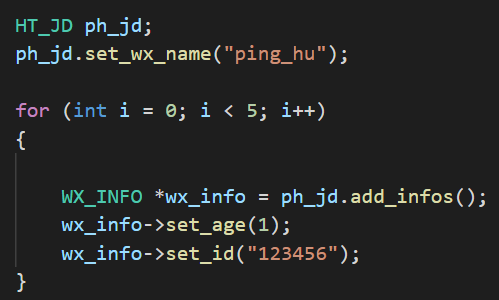
repeated是protobuf中的一种限定修饰符，从字面意思看有“重复”的意思，实际上它就是用来指定某一个字段可以存放同一个类型的多个数据（当然也可以是0个或者1个），相当于C++中的vector。

repeated: 该字段可以在格式良好的消息中重复任意多次（包括零）。其中重复值的顺序会被保留。

由下图.proto定义内容所示，WX\_INFO类型作为repeated被包含在HT\_JD类型中，WX\_INFO是从属于HD\_JD的，对卫星信息的赋值使用了repeated，用于赋值同一类型的多个卫星信息。



定义好数据类型后，就是在主程序中调用api进行赋值。



WX\_INFO \*wx\_info = ph\_jd.add\_infos();

该语句用于对wx\_info赋值。首先说明ph\_jd是HT\_JD类型的结构体，wx\_info是WX\_INFO类型的结构体，因此wx\_info从属于ph\_jd的infos成员，通过.add.infos这个API对wx\_info进行调用，wx\_info的结构体成员用箭头->符号进行赋值。

语句中使用for循环，每循环一次新增一个WX\_INFO，一共5个。

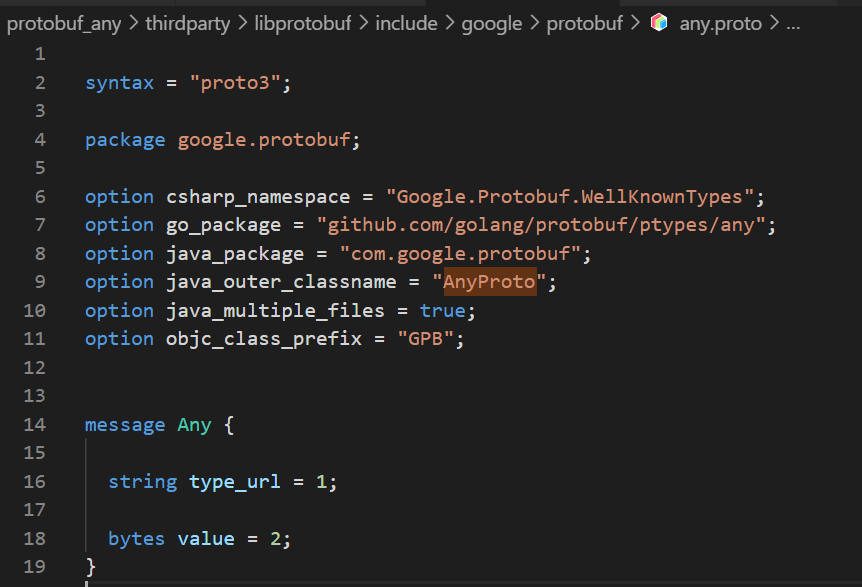
[protobuf入门教程(四)：repeated限定修饰符\_ \_protobuf repeated](https://blog.csdn.net/tennysonsky/article/details/73921025)

### 4.2 Any类型

message 定义好后进行数据传输，但总会遇到某些数据结构进行组合的操作，采用默认的定义 message 方式，造成代码量的激增。为了解决这个问题 protobuf 提供类型 Any 解决 gRPC[[1]](#footnote-1)中泛型的处理方式。

protobuf中的Any类型与C++中的泛型概念类似，可以定义为任意的类型。在序列化的时候可以通过**PackFrom()**方法将任意的数据类型打包为Any类型，反序列化的时候通过**UnpackTo()**把Any类型还原为原始类型。

any.proto的内容很简单，如下图所示：



去掉注释之后，也就一个message，type\_url（指定类型）用来存储类型描述信息，value （内存空间)用来存储序列化（C++是SerializeToString函数）之后的字符串。

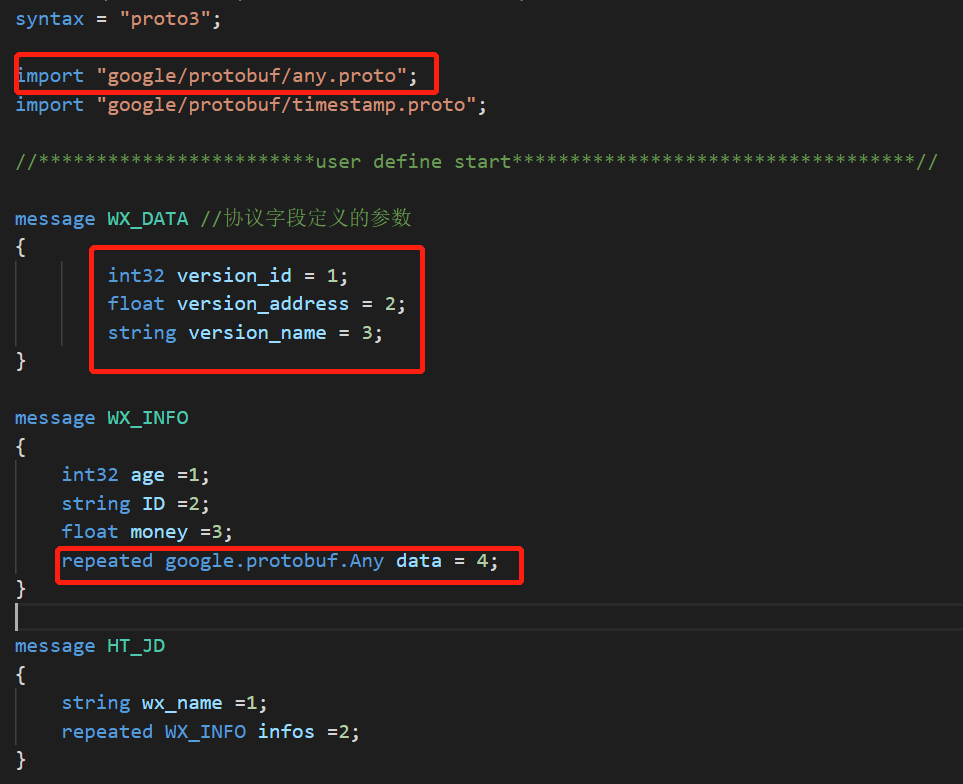
#### 4.2.1 .proto文件中的配置

要使用Any类型必须导入该proto文件，在开头处添加以下代码

import "google/protobuf/any.proto";

在message WX\_INFO中新增Any类型，调用Any的固定语法为：

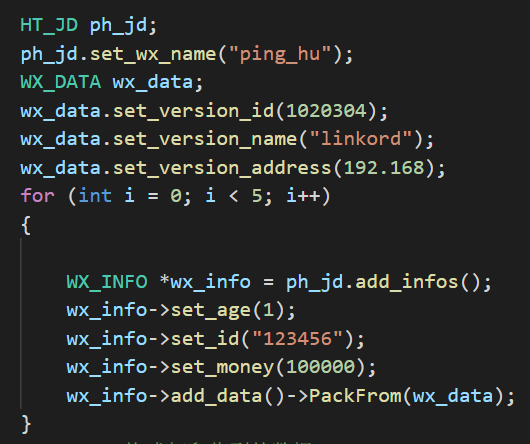
repeated google.protobuf.Any data



新建一个message WX\_DATA可以看到里面配置了 int32、float、string三种数据类型，用来测试any对泛型数据的处理。

#### 4.2.2 主程序中的配置

相比于第三部分中的main函数的内容，测试Any类型时在main函数中补充了代码，如下所示：

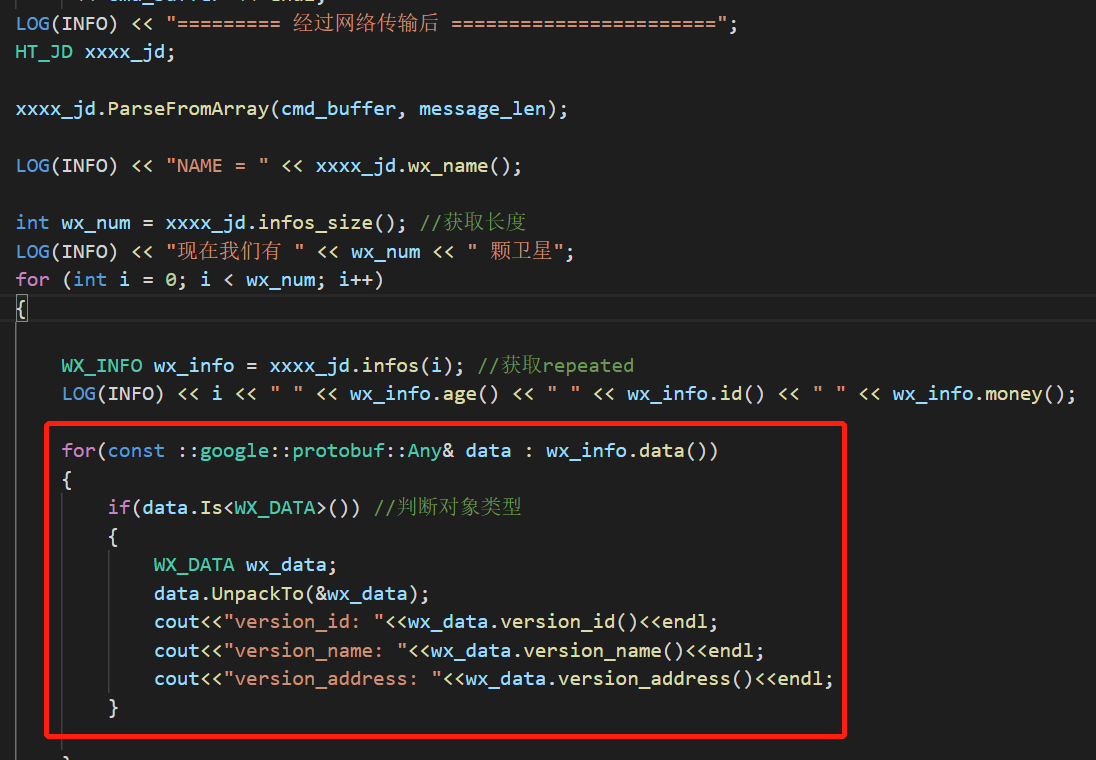


wx\_data是新的变量，并对其中的三个成员version\_id、version\_address、version\_name进行赋值，它们三个分别对应的数据类型为int32、float、string。其中

wx\_info->add\_data()->PackFrom(wx\_data);

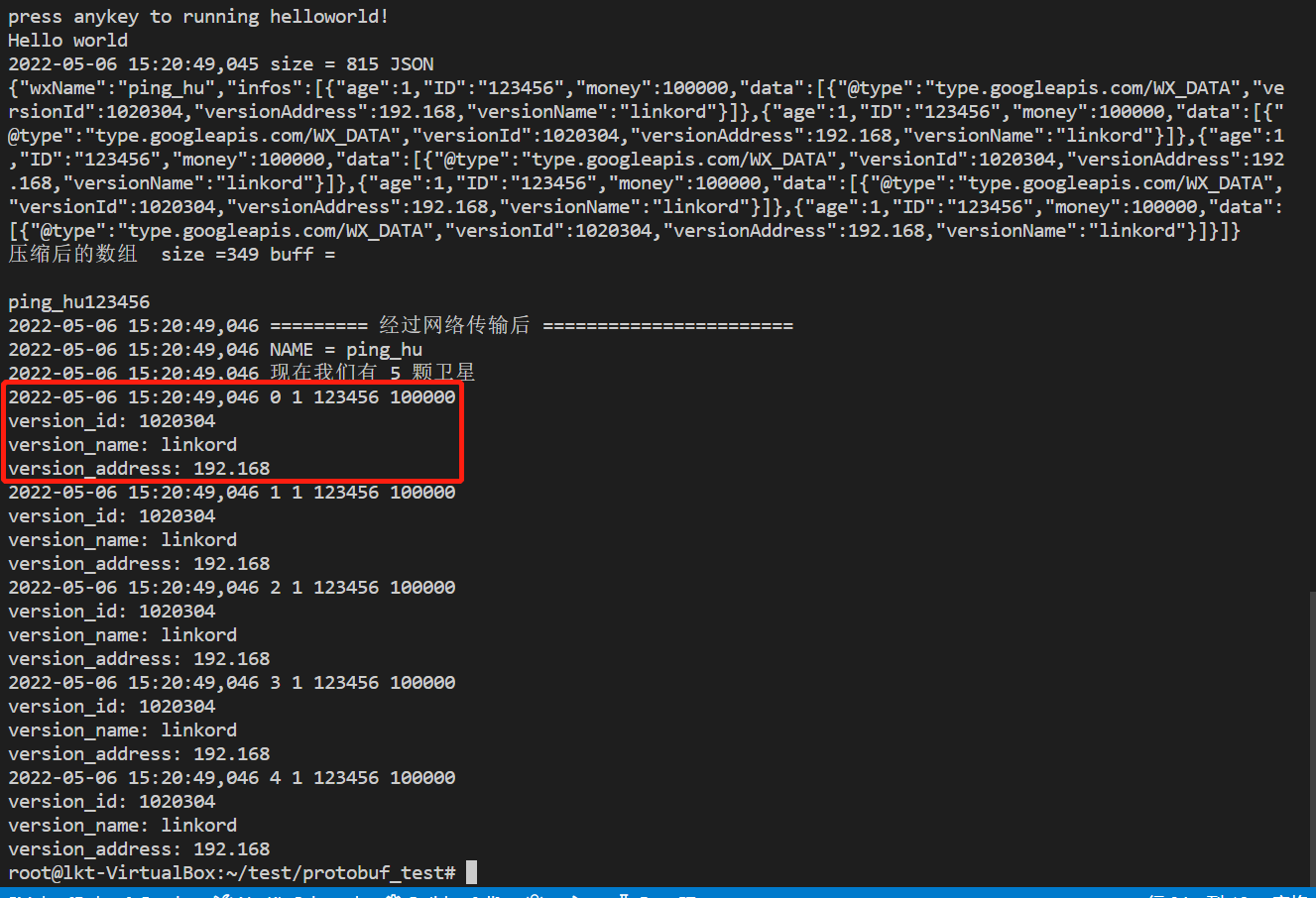
是序列化的时候通过**PackFrom()**方法将任意的数据类型打包Any类型的固定格式。wx\_data作为wx\_info的any类型打包到pack中。

在打印输出的代码，补充了以下框选出的部分



上述代码是将Any对象从pack中恢复到具体对象。

执行结果如下所示，每个卫星都打印出了相关信息，三个类型的any数据都完整打印出来：



## 五、Windows环境QT工程编译protobuf

5.1和5.2为编译protobuf库，如果已熟知过程，这里打包了第三方库可以直接使用。

### 5.1环境确认

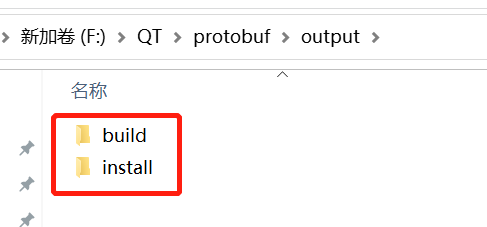
本次用的protobuf库版本为protobuf-cpp-3.20.1

[ReleaseProtocolBuffers v3.20.1 · GitHub](https://github.com/protocolbuffers/protobuf/releases/tag/v3.20.1)

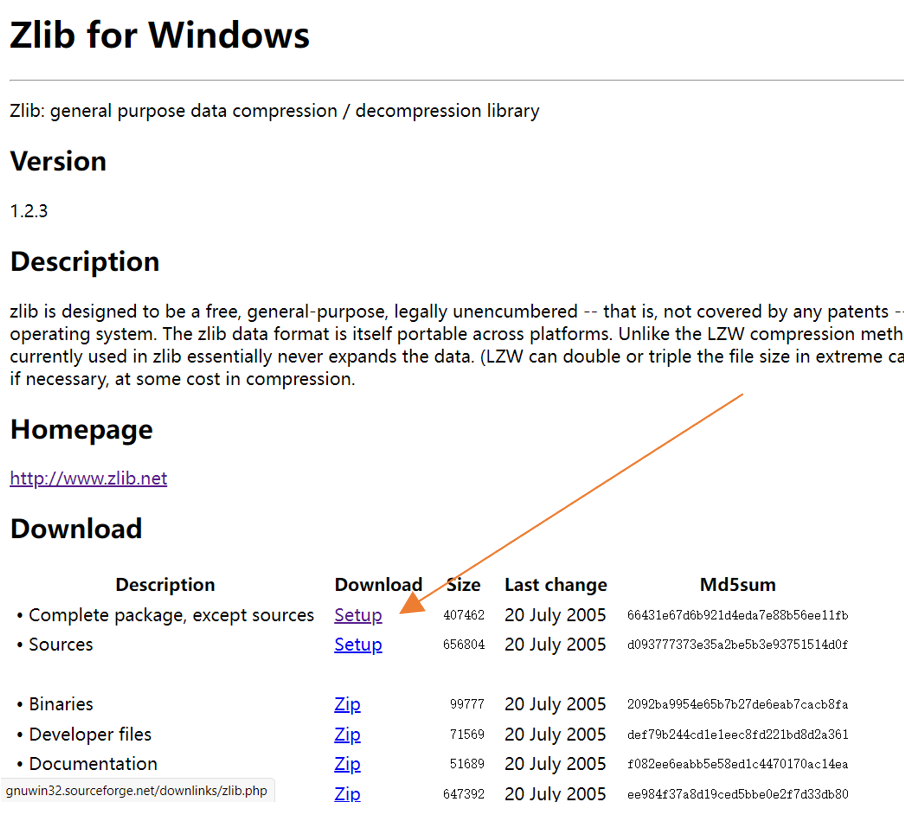


QT版本为QT6.2.4,如果QT版本过低，会导致编译前版本过低而无法编译protobuf库。

我自己新建了output文件夹用于存放之后编译输出的文件，build存放cmake编译protobuf库的输出文件，install存放编译生成的第三方protobuf静态库。



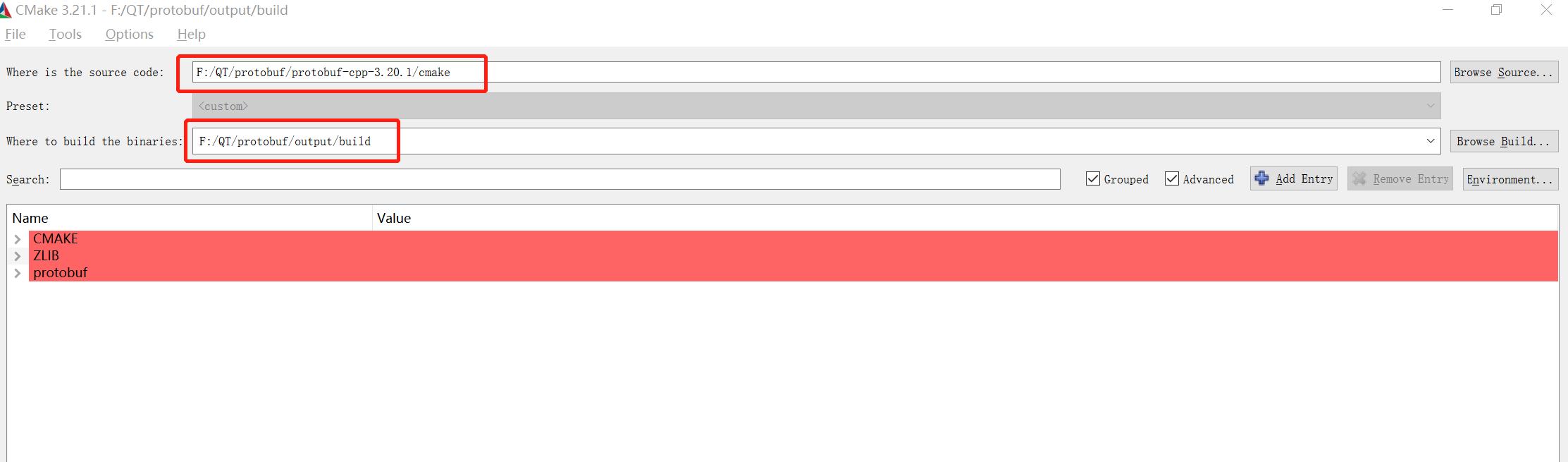
下载zlib：[Zlib for Windows (sourceforge.net)](http://gnuwin32.sourceforge.net/packages/zlib.htm)



### 5.2编译protobuf静态库

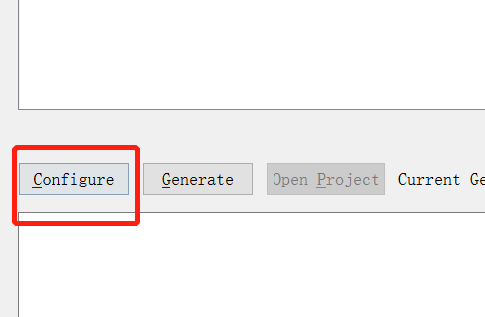
下面介绍两种方式编译生成protobuf静态库，mingw编译器生成和msvc编译器生成。两种不同的编译器生成的静态库要对应工程的编译器使用。推荐使用

打开QT6.2.4\Tools\CMake\_64\bin目录下的cmake-gui.exe



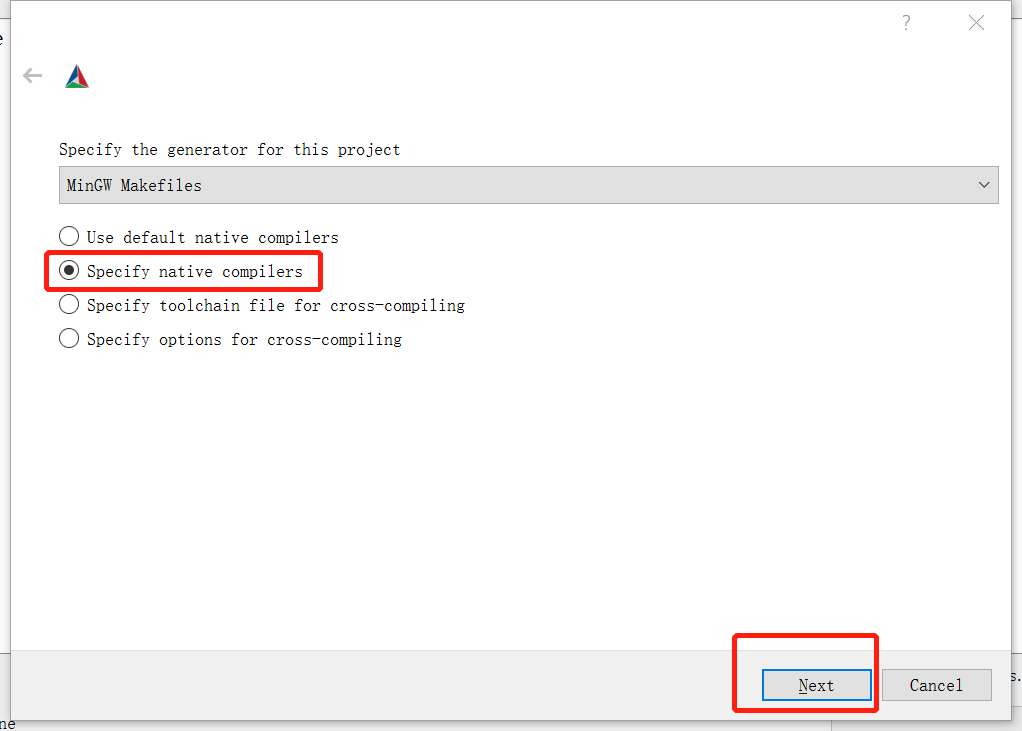
如上图所示，将下载的protobuf库下protobuf-cpp-3.20.1/cmake作为目标文件夹，我自己创建output/build作为输出文件夹。

点击左下角configure，设置配置

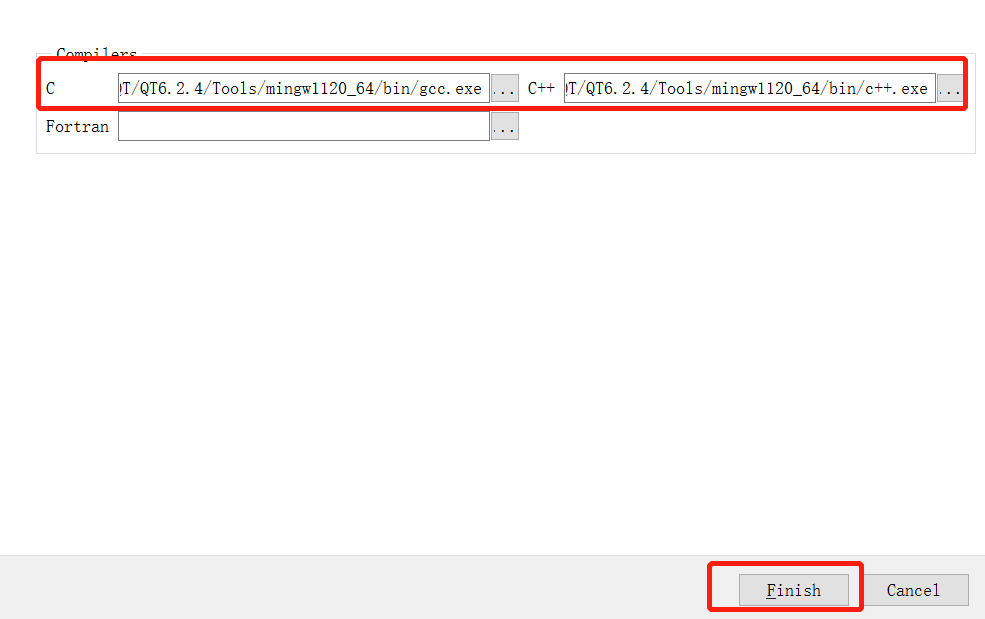


#### 5.2.1 mingw编译静态库

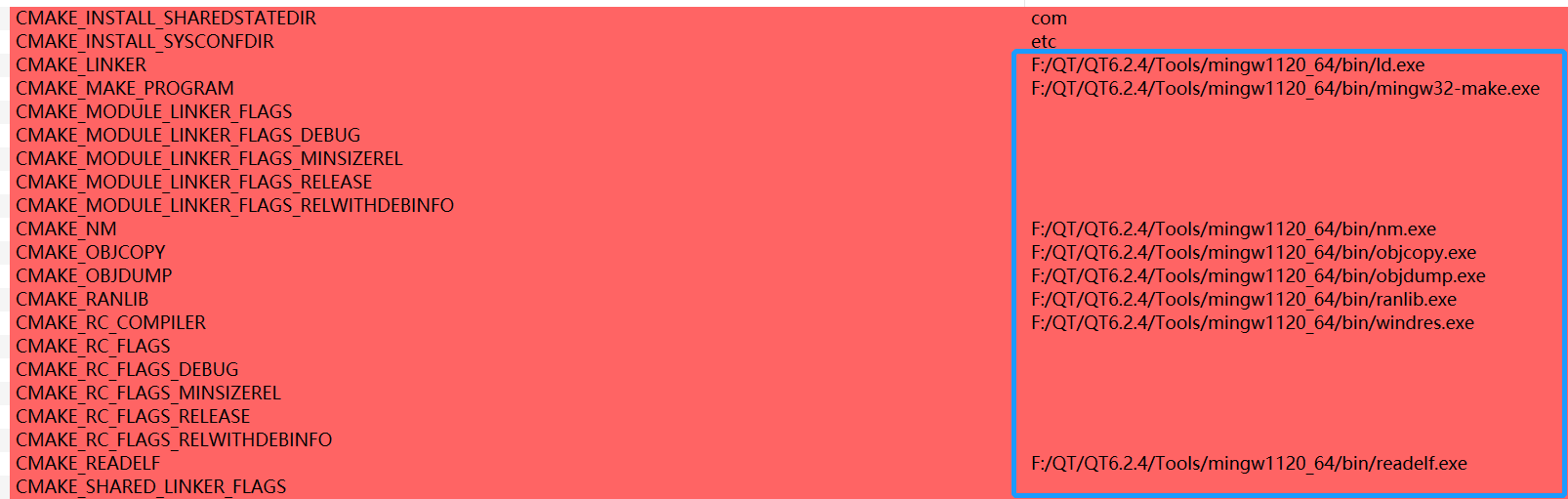
选择第二个指定编译器：



将c/c++编译器设置为QT自带编译器，Fortran不管，参考下图：



Finish后，检查cmake里面的编译器，确保都是QT路径下自带的：



这一选项路径，我设置为自己创建的install目录，存放之后编译生成的protobuf静态库：

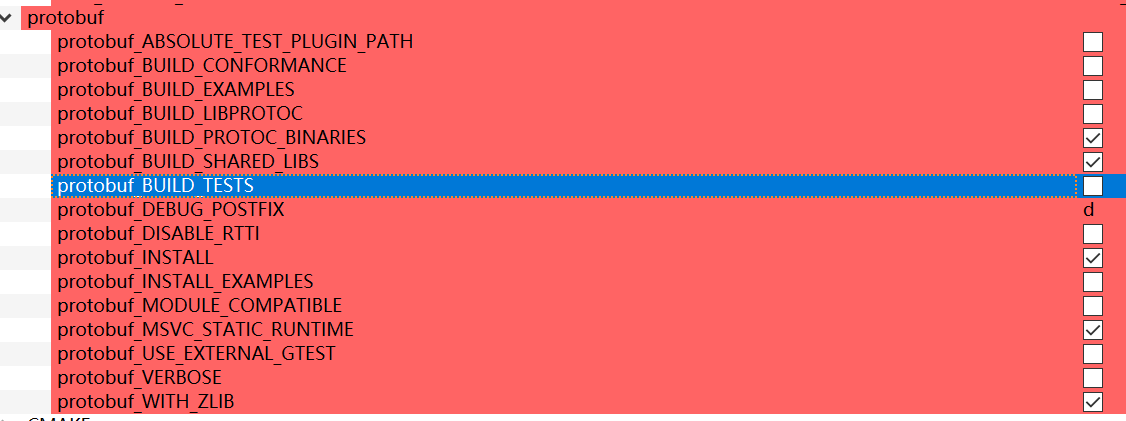


打开ZLIB设置栏，配置zlib的头文件路径和库文件：

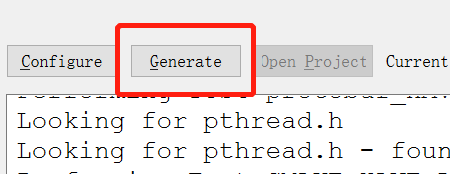


配置protobuf，切记关闭TESTS选项，勾选参考如下：

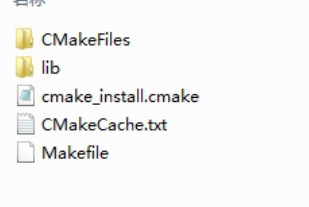
（此处不需要编译动态库，取消勾选SHARED\_LIBS）



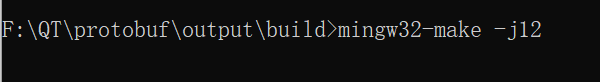
设置完毕后，点击左下角generate：



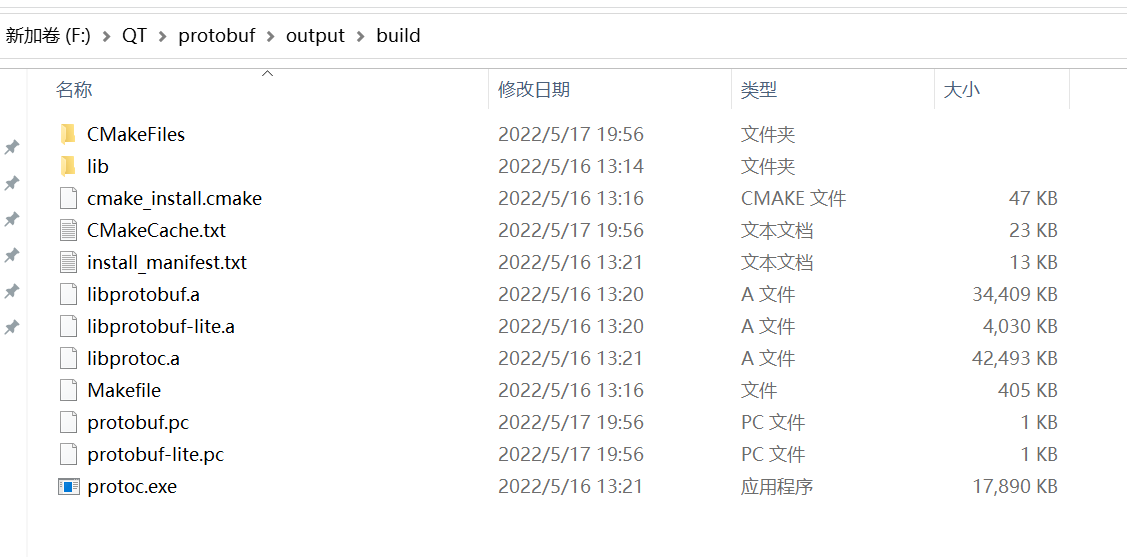
结束后，编译文件输出在我的build目录下：



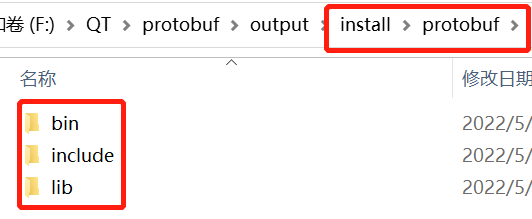
打开cmd，进入makefile所在文件夹，输入mingw32-make.exe -j12命令，开始编译源代码。



编译结束后，打开build文件夹可以看到生成的库文件和protoc.exe如下图所示。

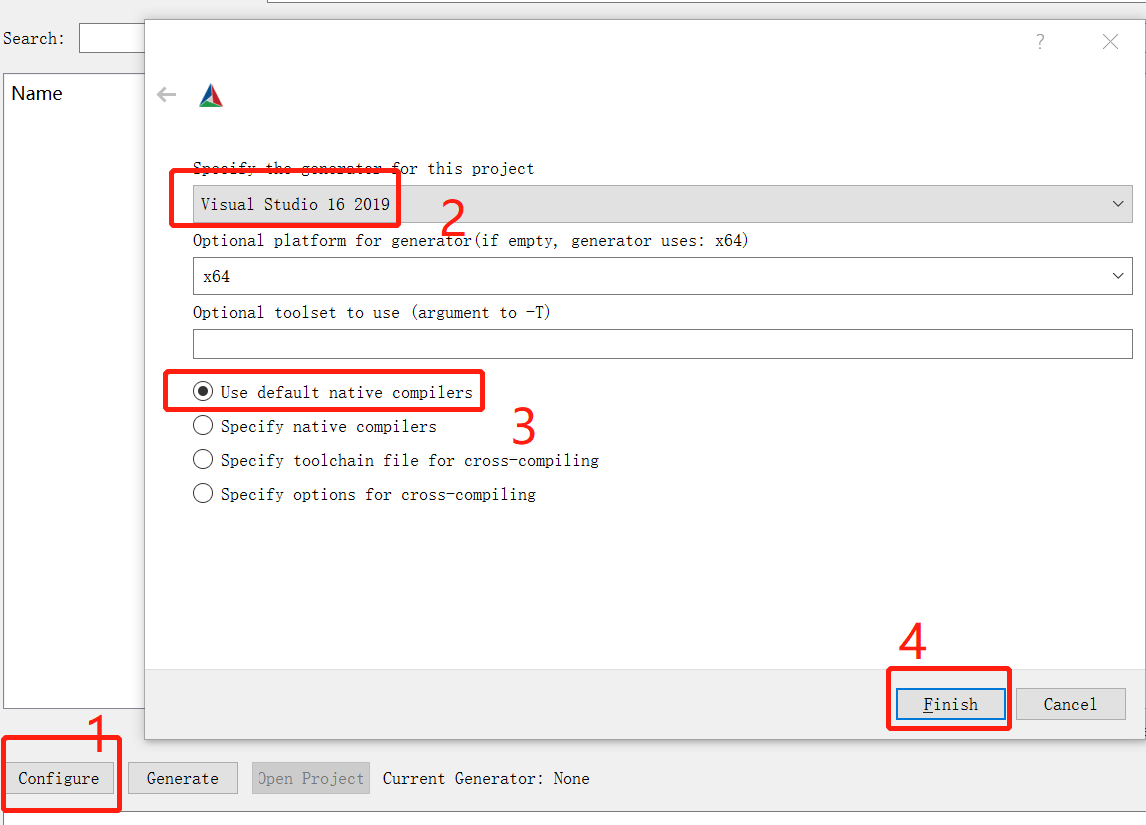


编译生成第三方静态库，在cmd中makefile所在目录下，输入指令mingw32-make.exe install。编译结束后，打开自己建的install目录下可以看到生成的第三方库：

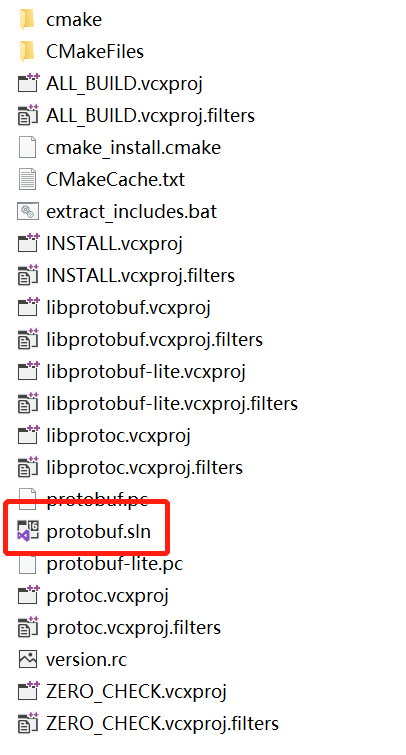


#### 5.2.2 msvc编译静态库

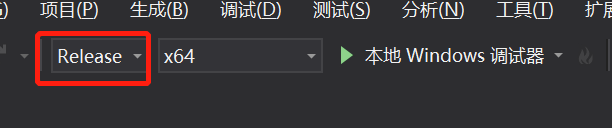
点击congifure后选择默认编译器



之后的配置参考mingw编译中的方式，设置完毕后，点击左下角generate，结束后，编译文件输出在我的build目录下：



打开.sln文件进入VS解决方案。由于我们要生成的是release静态库，将上方编译方式改为release：

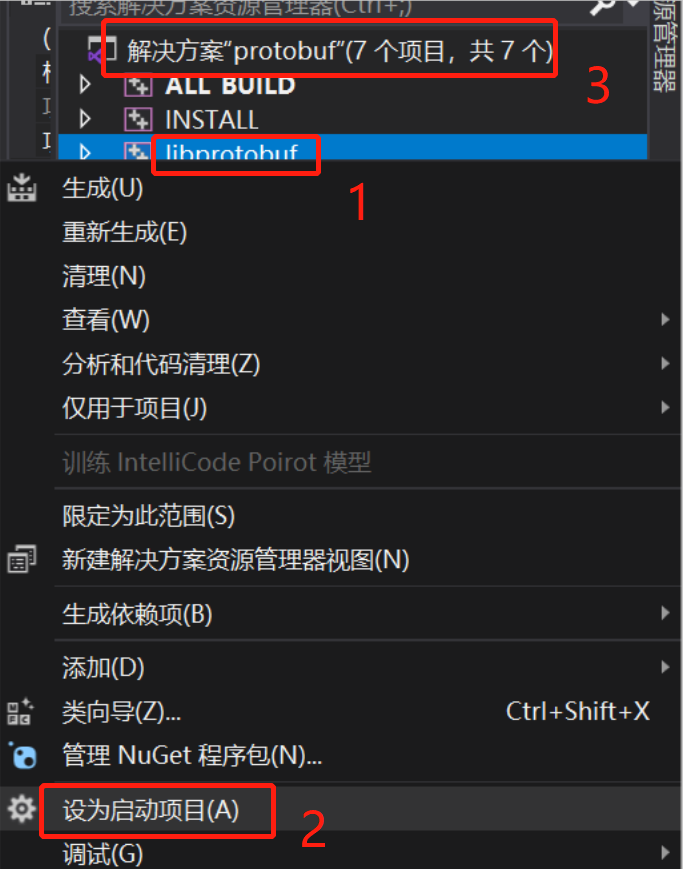


按照下图一次操作：

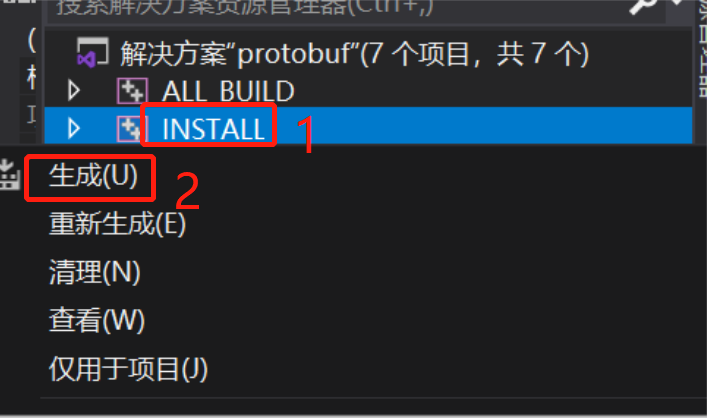
1.右击解决方案资源管理器中的libprotobuf

2.设置为启动项

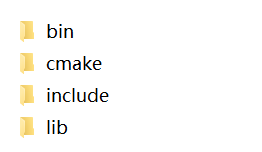
3.右击解决方案->生成解决方案



编译完成后，若提示非执行程序的错误，关掉即可，因为编译的库不是执行文件。编译的结果会显示跳过一项内容，此时右键点击INSTALL，点击生成：



就会在原先设置的install目录下生成静态库。



### 5.3 生成proto.pb.cpp和.h

为了方便编译，在刚才生成的proto.exe同级目录下新建test.proto文件，proto代码如下：

syntax = "proto3";

message Person {

   int32 id = 1;

   string name = 2;

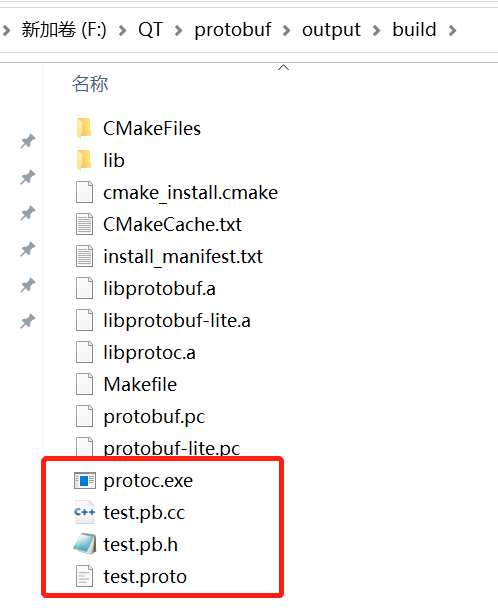
   string email = 3;

}

在cmd中proto.exe和test.proto所在目录下执行protobuf编译执行：

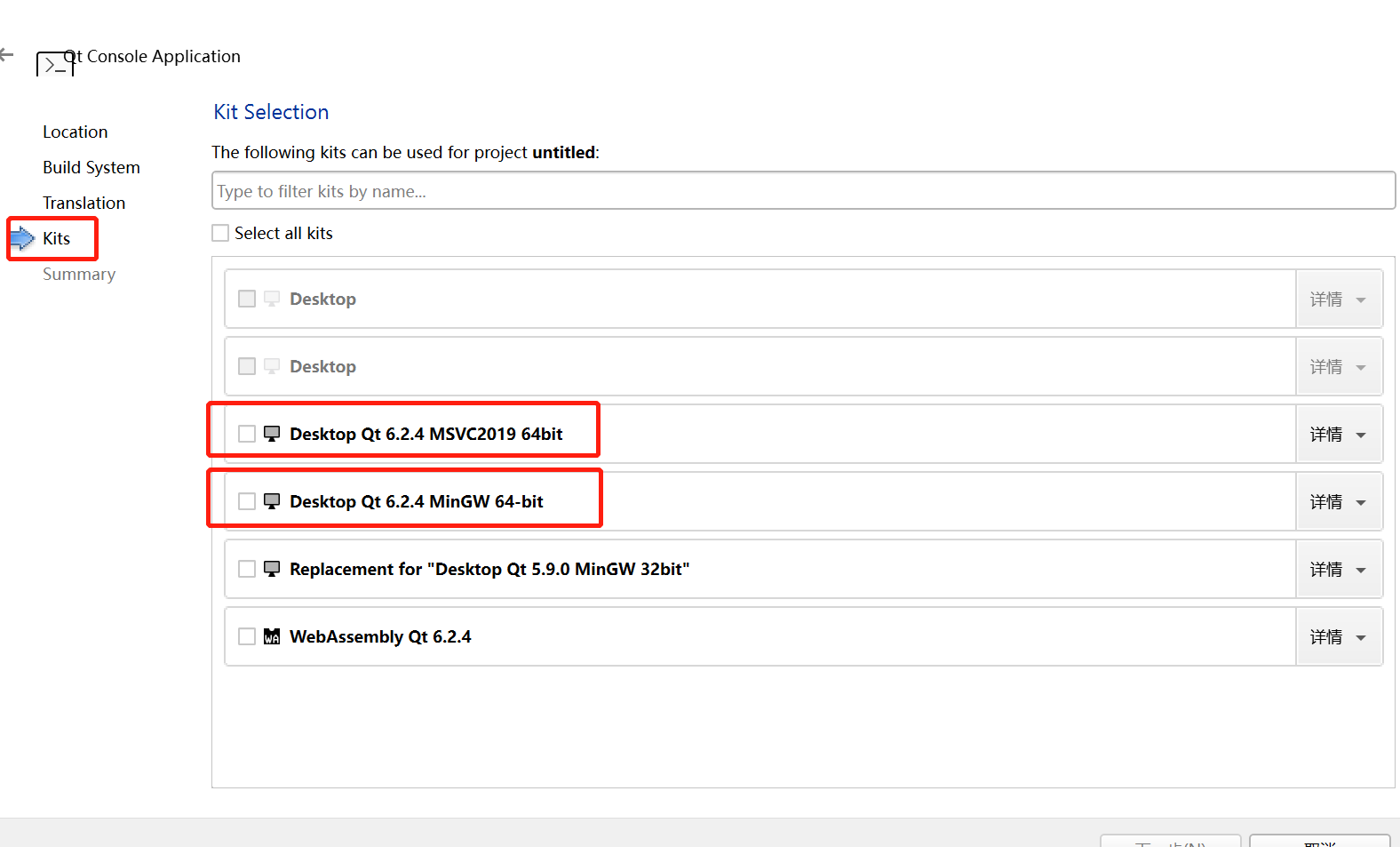
protoc.exe --cpp\_out=./ test.proto

可以看到此时目录下生成.pb.cpp和.h两个文件：

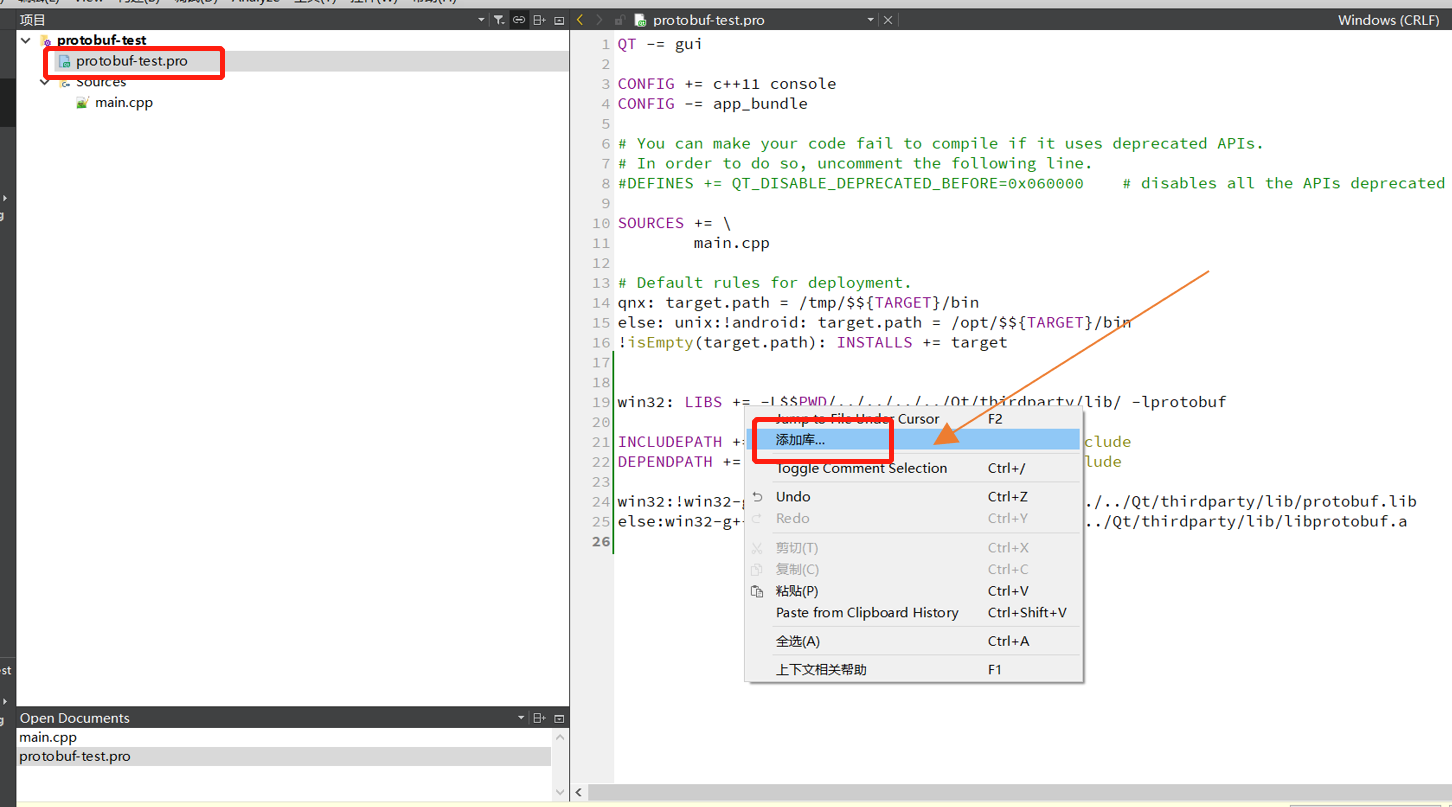


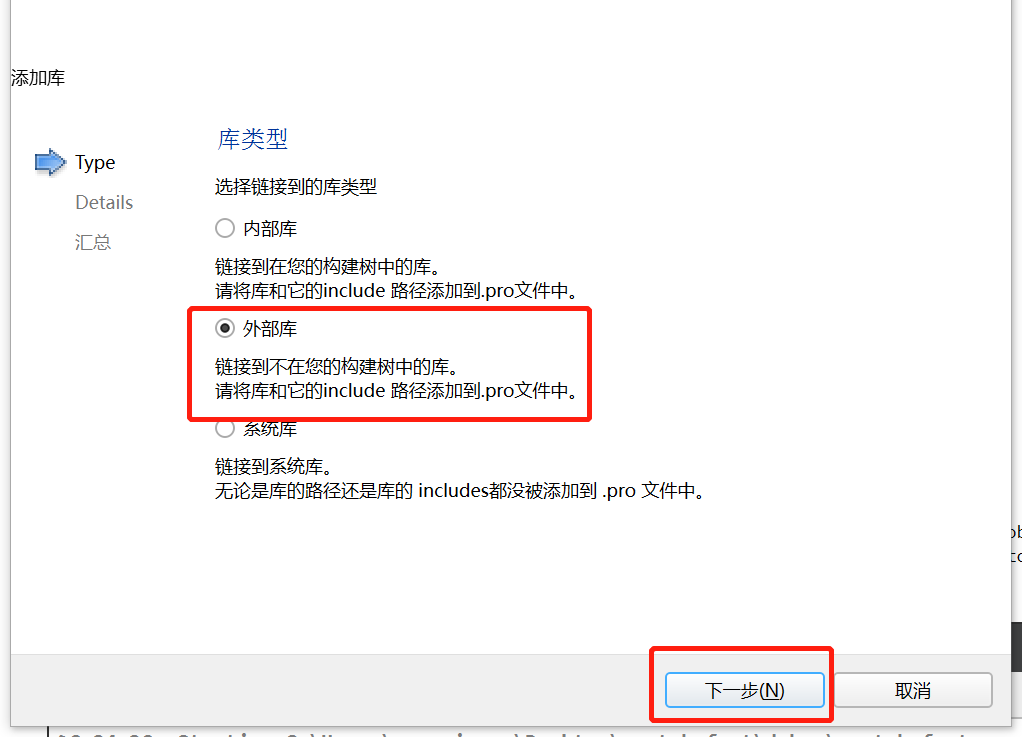
### 5.4创建QT工程

新建QT Console Application，注意下图编译器选项，如果是mingw编译的protobuf库选择mingw编译器，由msvc编译的protobuf库选择msvc编译器：

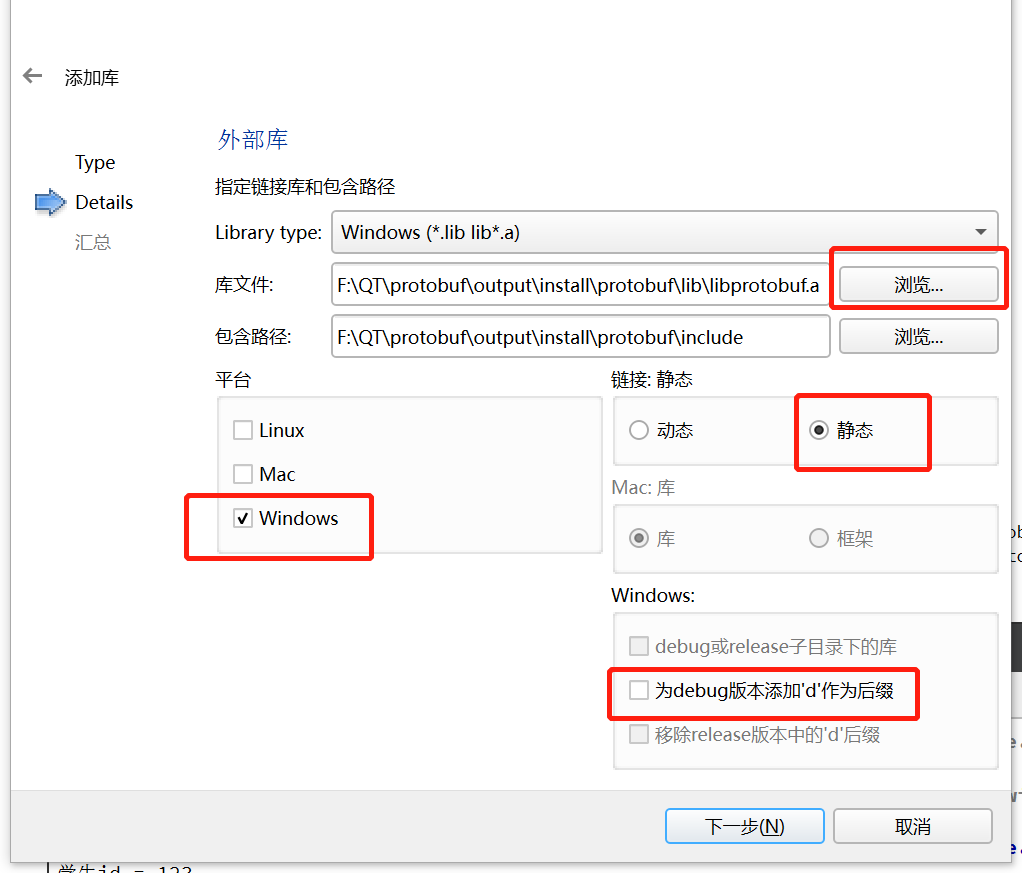


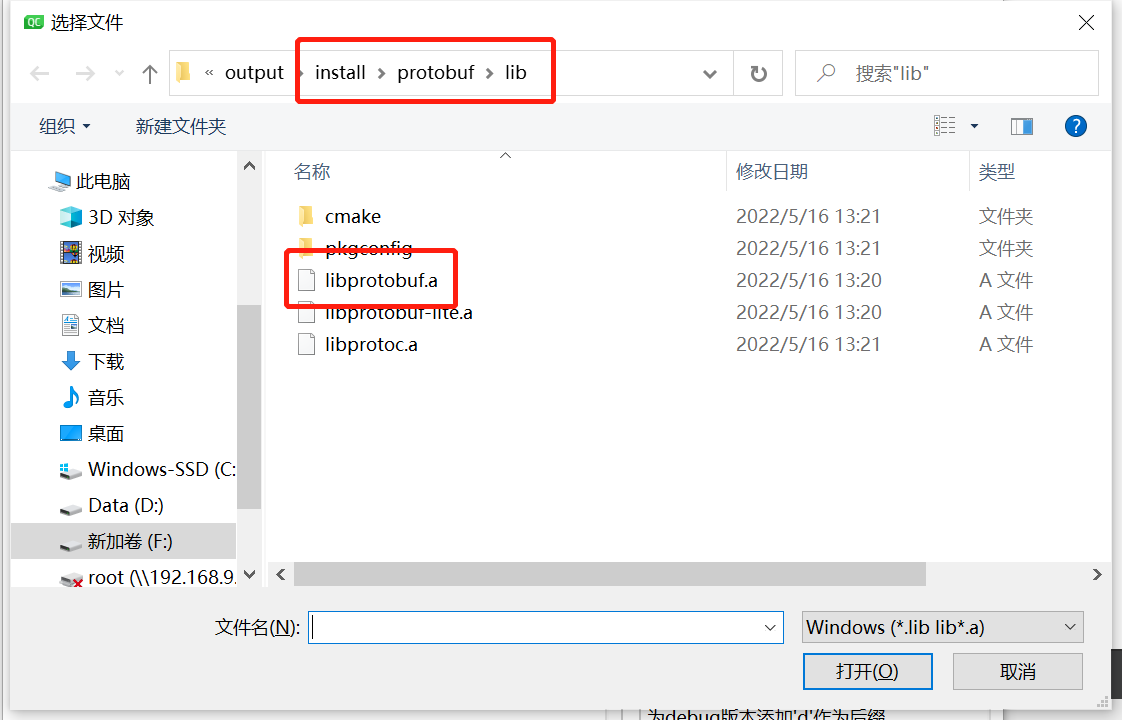
在.pro文件下添加库：



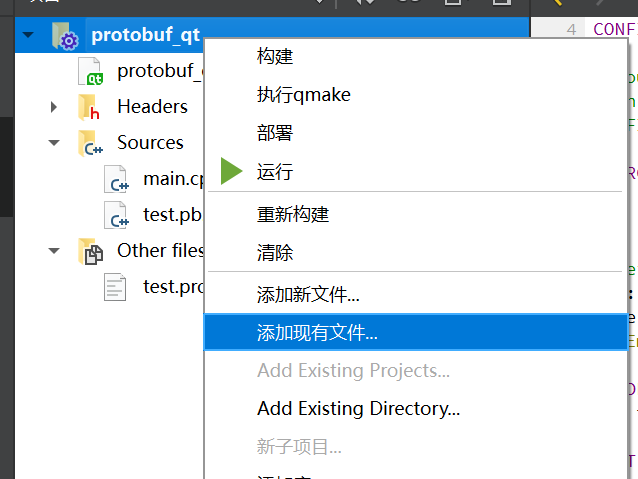


所包含的库文件为上面生成在install/protobuf/lib中的libprotobuf.a文件，配置如下所示：

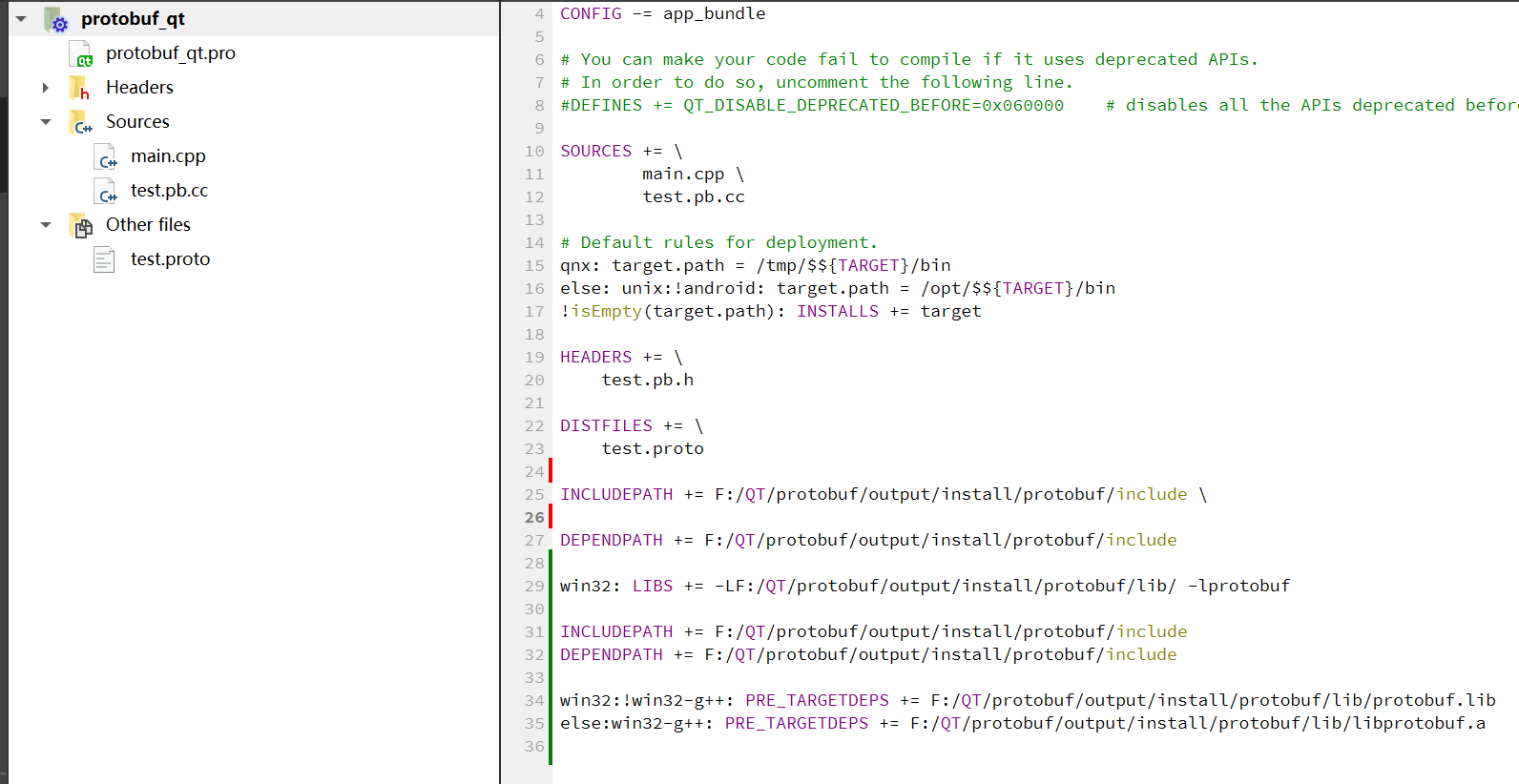




将上面生成的test.proto, .pb.cpp, .pb.h放到QT工程目录下，选择添加现有文件:



此时QT工程目录和.pro代码如下所示：



注：mscv方式编译的protobuf库需要导入库前，在.pro文件中加入以下代码

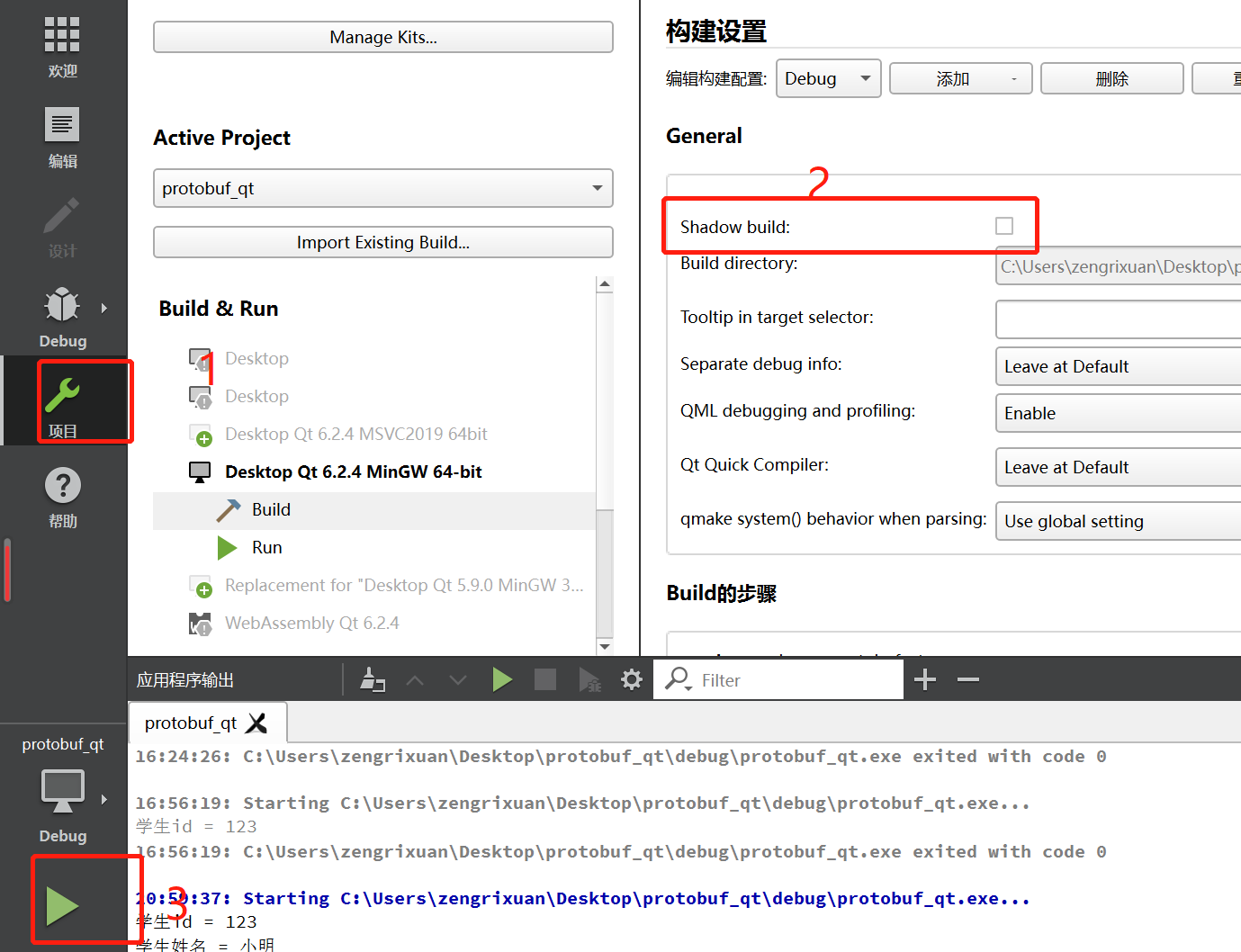
QMAKE\_CFLAGS\_RELEASE += -MT

QMAKE\_CXXFLAGS\_RELEASE += -MT

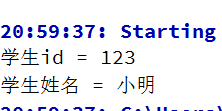
protobuf库是通过Visual Studio 2019编译而成的，而且编译时的运行时库被设置为MT。但是被调用的静态库通过QT编译时，运行时库被默认为MD。要避免这个错误，只能在编译时，将两者的运行时库设置成一样。

Qt5.6.0库是Qt源码通过Visual Studio 2013编译而成的，而且编译时的运行时库被设置为MDd。但是被调用的静态库通过Visual Studio 2013编译时，运行时库被设置为MTd。要避免这个错误，只能在编译时，将两者的运行时库设置成一样。

取消勾选shadow build后开始编译：



编译结果如下：



### 5.5打包QT\_protobuf工程

将msvc编译器编译的release版本protobuf第三方库和.proto打包到qt的工程中。新建thirdparty文件夹存放protobuf第三方库。新建spec文件夹存放.proto文件和bat脚本，双击脚本就会生成cpp和h文件，后续proto内容可以直接更改。

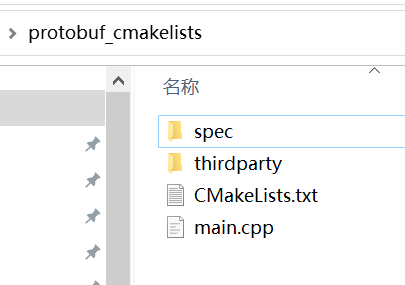
工程git链接：[Zeng Rixuan / qt\_protobuf · GitLab (linkortech.com)](http://git.linkortech.com:10020/Zengrixuan/qt_protobuf)

已打包好的工程直接克隆就可以release方式运行。

## 六、cmakelists管理工程

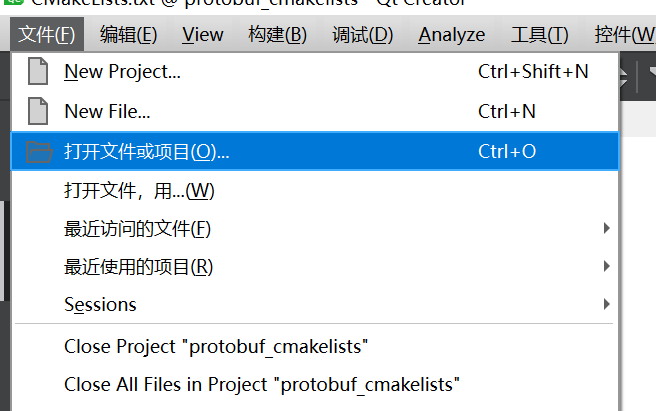
借助cmakelists可以简单有效地管理工程，qt6版本可以直接打开cmake项目进行编译，也可以通过cmd指令make生成.sln解决方案用VS打开编译。编辑好cmakelists，将头文件、源文件、protobuf库打包存放在指定路径下。

Cmake工程文件目录如下，spec文件夹存放protobuf生成的头文件和源文件，thirdparty存放protobuf的动态库。具体cmakelists代码请见工程文件，部分路径需要根据实际情况更改。

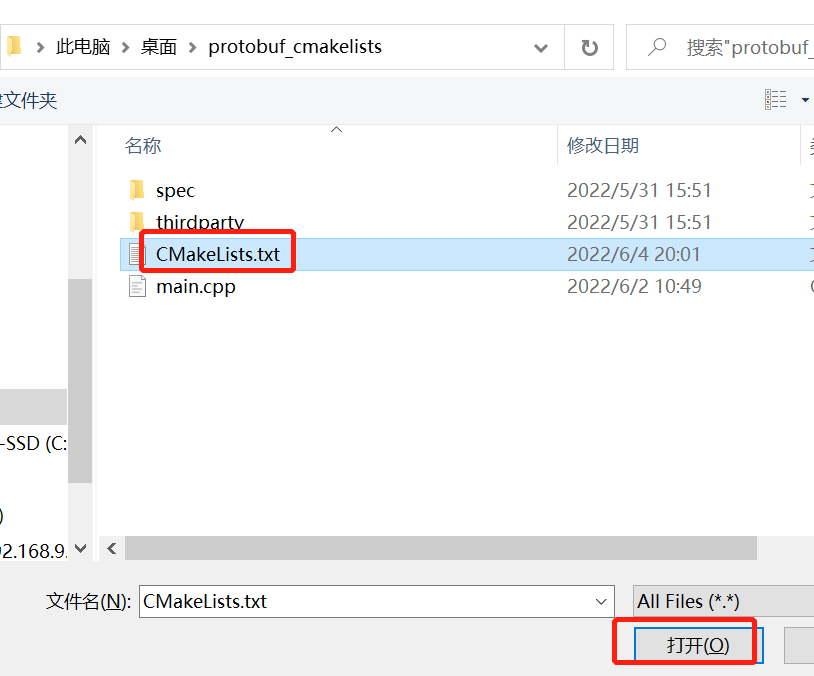


### 6.1 QT编译cmake工程

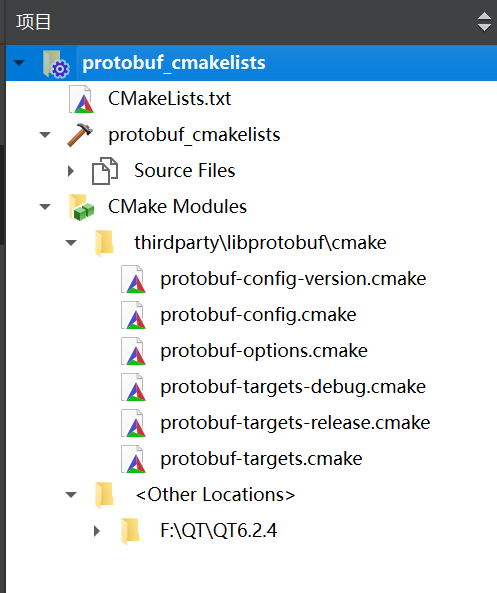
QT页面右上角“文件”，选择打开文件或项目，



选择并打开cmakelists文件：



如果cmakelists代码中的路径都正确，打开后QT会自动识别并导入cmakelists代码中指定的路径中的文件：

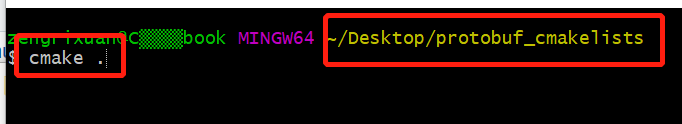


直接编译即可：

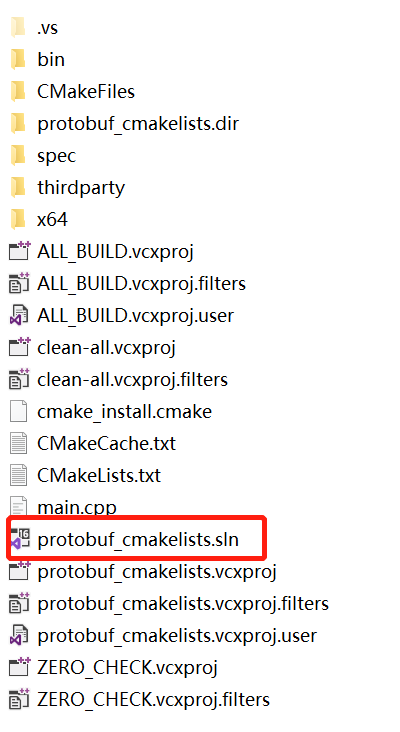


### 6.2 VS编译cmake工程

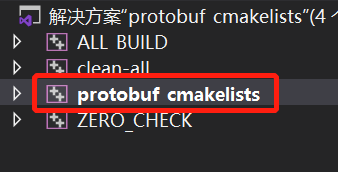
将cmd定位到在cmakelists所在目录下，输入命令 cmake . ：



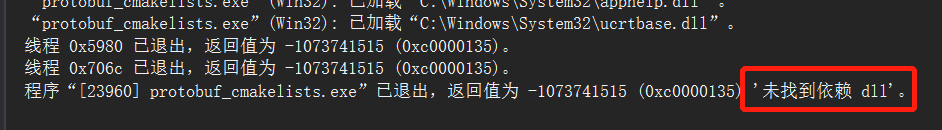
编译结束后，工程目录生成的文件如图所示：



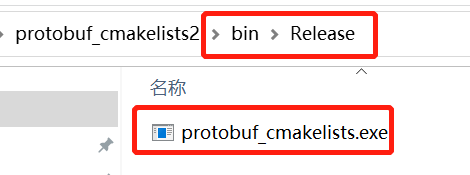
双击.sln解决方案，会启动VS对项目进行管理，进入VS后右键protobuf——cmakelists工程，设置为启动项目，如何编译：



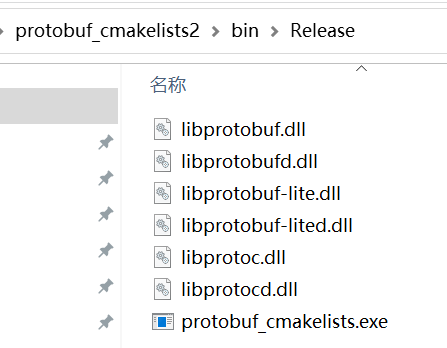
编译之后提示报错：



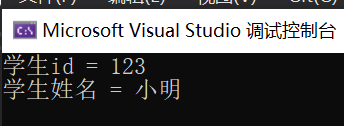
此时找到.exe可执行程序，在bin/release文件夹下：



将protobuf库中的.dll文件(thirdparty\libprotobuf\bin) 拷贝到.exe同级目录下:



此时回到VS页面重新编译，编译结果如下：



建议QT和VS编译cmake两种方法都尝试一下。显然，使用cmake管理工程会方便很多。

完结撒花！

1. RPC：远程过程调用，gRPC是由 google开发的一个高性能、通用的开源RPC框架。gRPC使用Protocol Buffers作为序列化协议 [什么是 gRPC ？\_](https://blog.csdn.net/kevin_tech/article/details/120681720) [↑](#footnote-ref-1)