



麦洛科菲
MallocFree.Com

周哥教 IT 视频课配套课件 结合视频课程学习效果更佳
www.zhougejiaoit.com

周哥教 IT gRPC 详讲

视频课地址: <https://ke.qq.com/course/4334973?tuin=a71606>

目录

周哥教 IT gRPC 详讲.....	1
RPC 与 gRPC.....	1
Linux C++开发环境: Win+VSCode+Docker.....	2
安装中遇到的问题解决: Docker.....	3
VSCode 安装插件, 连接 docker.....	3
自动化构建工具: CMake(1).....	4
Ubuntu:20.04+GCC9 gRPC 安装.....	6
ProtoBuf(1).....	8
ProtoBuf(2) demo.....	9
gRPC 入门例子 helloworld 剖析.....	11
gRPC Add Demo.....	12
Protobuf 类型大全.....	16
ProtoBuf 编写.....	17
protobuf 中的坑.....	23
protobuf VS json.....	24
gRPC 四种模式.....	24
gRPC 超时设置.....	32
异步 GRPC (1)	32
同步与异步.....	32
异步 GRPC (2) :客户端异步.....	33
其他例子.....	33
gRPC vs. Restful API.....	34

RPC 与 gRPC

RPC, 即远程过程调用 (Remote Procedure Call), 作为一种计算机通信协议, 允许运行于一台计算机的程序调用另一台计算机的子程序 (函数)。RPC 允许跨机器、跨语言调用计算机程序方法。比如用 go 语言写了个程序实现了乘法运算 `Multiple()`, 并把 go 程序部署在阿里云服务器上面, 现在我有一个部署在腾讯云上面的 go 或 php 项目, 需要调用 golang 的 `Multiple()`方法获取计算结果, 跨机器调用 go 方法的过程就是 RPC 调用。

gRPC:Google 推出的基于 protobuf 实现的 RPC

周哥教 IT www.zhougejiaoit.com

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

A high-performance, open-source universal RPC framework

gRPC 是一个现代的、开源的、高性能远程过程调用(RPC)框架，可以在任何平台运行。gRPC 使客户端和服务端应用程序能够透明地进行通信，并简化了连接系统的构建。gRPC 支持的语言包括 C++、Ruby、Python、Java、Go 等。

gRPC 在客户端与服务端通信之中应用非常广泛，事实上的标准。服务端开发必备技能。

Linux C++开发环境：Win+VSCode+Docker

安装 docker: <https://www.docker.com/get-started> 内存 8G, 16G

- 更新 linux 内核包, Windows Subsystem for Linux
- 打开 VT 虚拟化: `bcdedit /set hypervisorlaunchtype auto`
- 设置镜像: `"registry-mirrors": ["https://hub-mirrors.c.163.com", "https://mirror.baidubce.com", "https://9cpn8tt6.mirror.aliyuncs.com"]`
- 解决 WSL2 中 Vmmem 内存占用过大问题: `%UserProfile%\.wslconfig wsl --shutdown bash`

`docker info`

`docker version`

`docker search ubuntu:20.04`

`docker pull xxx`

`docker images`

创建并启动容器:

```
docker run -it -v /D/MallocFree2.0/zgjit-lesson/2grpc/demo:/grpcdemo -d -p 50051:50051 --name=grpcdev_new 5fc36d410b62
```

`adduser zyr`

`su zyr`

`vim /etc/sudoers`

`zyr ALL=(ALL:ALL) ALL`

`sudo apt-get update`

`sudo apt-get install build-essential`

`gcc --version`

`sudo apt-get install git`

`sudo apt-get install gdb`

启动容器: `docker exec -it -u zyr 5446e04b5bce /bin/bash`

`docker ps`

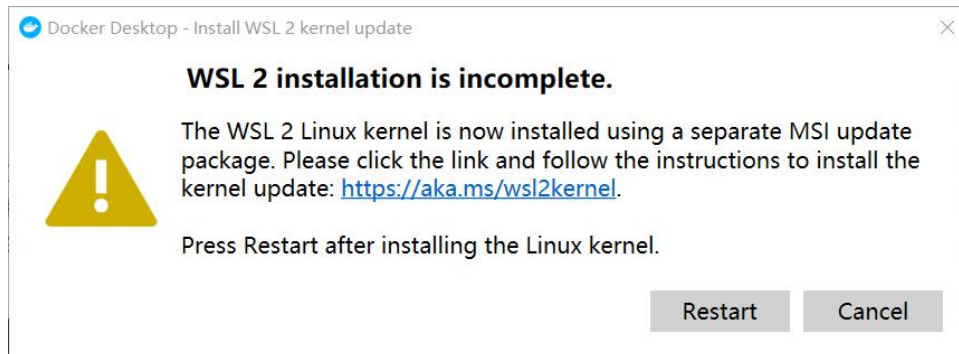
`docker commit container_id imagename:v1.0`

安装中遇到的问题解决：Docker

<https://docs.microsoft.com/zh-cn/windows/wsl/install-manual#step-4---download-the-linux-kernel-update-package>

步骤 4 - 下载 Linux 内核更新包

wsl --set-default-version 2



步骤 4 - 下载 Linux 内核更新包

1. 下载最新包:

- [适用于 x64 计算机的 WSL2 Linux 内核更新包](#)

① 备注

如果使用的是 ARM64 计算机，请下载 [ARM64 包](#)。如果不确定自己计算机的类型，请打开命令提示符或 PowerShell，并输入：`systeminfo | find "System Type"`。Caveat: 在非英文版 Windows 上，你可能必须修改搜索文本，对“System Type”字符串进行翻译。你可能还需要对引号进行转义来用于 find 命令。例如，在德语版中使用 `systeminfo | find "Systemtyp"`。

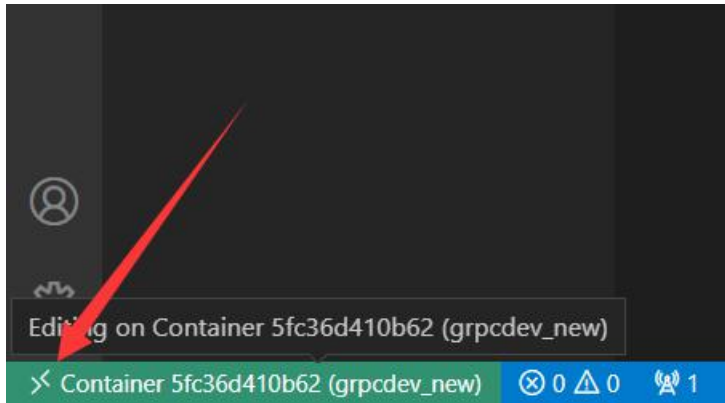
2. 运行上一步中下载的更新包。（双击以运行 - 系统将提示你提供提升的权限，选择“是”以批准此安装。）

VSCode 安装插件，连接 docker

安装 VSCode 及相关插件：<https://code.visualstudio.com/>

Remote-Containers

C/C++ 插件



自动化构建工具：CMake(1)

cmake 是 makefile 的上层工具，产生可移植的 makefile，并简化自己动手写 makefile 时的巨大工作量。makefile 通常依赖于你当前的编译平台，而且编写 makefile 的工作量比较大，解决依赖关系时也容易出错。对于大多数项目，应当考虑使用更自动化一些的 cmake 来生成 makefile，而不是动手编写 makefile

<https://cmake.org/>

```
tar -xvf cmake-3.22.0-rc2.tar.gz
```

```
cd cmake-3.22.0-rc2
```

```
./bootstrap
```

```
make
```

```
make install
```

cmake 会默认安装在 /usr/local/bin 下面

```
修改 ~/.bashrc，添加 export PATH="$PATH:/usr/local/bin"
```

```
source ~/.bashrc
```

```
sudo apt-get install cmake
```

```
cmake -version
```

```
wget -q -O cmake-linux.sh https://github.com/Kitware/CMake/releases/download/v3.19.6/cmake-3.19.6-Linux-x86_64.sh
```

```
sh cmake-linux.sh -- --skip-license --prefix=$MY_INSTALL_DIR
```

```
rm cmake-linux.sh
```

```
#include <iostream>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    std::cout << "Hello CMake!" << std::endl;
```

```
    return 0;
```



麦洛科菲
MallocFree.Com

周哥教 IT 视频课配套课件 结合视频课程学习效果更佳
www.zhougejiaoit.com

```
}

#CMakeLists.txt
#Set the minimum version of CMake that can be used
# To find the cmake version run
# $ cmake --version
cmake_minimum_required(VERSION 3.5)

# Set the project name
project (hello)

# Add an executable
add_executable(hello main.cpp)

mkdir build && cd build && cmake ../ && make

CMake(2)
PROJECT(main)
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)
ADD_SUBDIRECTORY( src )
AUX_SOURCE_DIRECTORY(. DIR_SRCS)
ADD_EXECUTABLE(main ${DIR_SRCS} )
TARGET_LINK_LIBRARIES( main Test protobuf pthread )

AUX_SOURCE_DIRECTORY(. DIR_TEST_SRCS)
file(GLOB DIR_TEST_SRCS proto/*.cc)
file(GLOB_RECURSE DIR_TEST_SRCS ${PROJECT_SOURCE_DIR}/src/*.cc)
ADD_LIBRARY ( Test ${DIR_TEST_SRCS})

set(CMAKE_CXX_COMPILER /usr/local/bin/g++-10)
set(CMAKE_CXX_COMPILER clang++)
set(CMAKE_CXX_STANDARD 17)
string(APPEND CMAKE_CXX_FLAGS " -Wall -Wextra -Winline -Wshadow -Werror")
string(APPEND CMAKE_CXX_FLAGS " -Wno-switch -Wno-sign-compare")

set(LIB_NAME mytestlib)
include_directories(src/thirdparty)

SET(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb")
SET(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")
ccmake .
```

周哥教 IT www.zhougejiaoit.com

CMAKE_BUILD_TYPE 设置为 Debug



Test protobuf pthread

Ubuntu:20.04+GCC9 gRPC 安装

```
sudo apt-get install pkg-config (安装如果慢, 请换 ubuntu 源)
sudo apt-get install autoconf automake libtool make g++ unzip
sudo apt-get install libgflags-dev libgtest-dev
sudo apt-get install clang libc++-dev
sudo apt-get install golang
sudo apt-get install openssl libssl-dev
sudo apt-get install zlib1g zlib1g-dev
sudo apt-get install libc-ares-dev libc-ares2
sudo apt-get install cmake
```

```
git clone https://gitee.com/githubplus/grpc.git https://github.com/grpc/grpc.git
```

```
cd grpc
```

```
git tag
```

```
git checkout v1.20.0
```

```
修改.gitmodules, 替换为 gitee.com/githubplus
```

```
git submodule sync
```

```
git submodule update --init
```

```
cd third_party/protobuf/
```

```
修改.gitmodules
```

```
git submodule update --init --recursive //确保克隆子模块, 更新第三方源码
```

```
sudo ./autogen.sh //生成配置脚本
```

```
sudo ./configure //生成 Makefile 文件
```

```
sudo make
```

```
sudo make install
```

```
sudo ldconfig // 更新共享库缓存 Set disable_coredump false >> /etc/sudo.conf
which protoc // 查看软件的安装位置
protoc --version //检查是否安装成功
cd ../../
将 src/core/lib/gpr/log_linux.cc 、 src/core/lib/gpr/log_posix.cc 、
src/core/lib/iomgr/ev_epollex_linux.cc 这几个文件中的
gettid()改为 sys_gettid()
```

修改 third_party/boringssl/crypto/x509/x509_test.cc Test 为 Test1

```
cmake -DgRPC_INSTALL=ON -DgRPC_ZLIB_PROVIDER=package
-DgRPC_CARES_PROVIDER=package -DgRPC_PROTOBUF_PROVIDER=package
-DgRPC_SSL_PROVIDER=package .
```

若报错:

```
CMake Error at cmake/cares.cmake:34 (find_package):
Could not find a package configuration file provided by "c-ares" with any
of the following names:
```

```
c-aresConfig.cmake
c-ares-config.cmake
```

```
Add the installation prefix of "c-ares" to CMAKE_PREFIX_PATH or set
"c-ares_DIR" to a directory containing one of the above files. If "c-ares"
provides a separate development package or SDK, be sure it has been
installed.
```

```
Call Stack (most recent call first):
```

```
CMakeLists.txt:116 (include)
```

```
-- Configuring incomplete, errors occurred!
```

则安装 cares:

```
cd grpc/third_party/cares/cares
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make install
```

然后开始编译 grpc:

```
sudo make
sudo make install
```

```
cd examples/cpp/helloworld
```

```
cmake .
```



周哥教 IT 视频课配套课件 结合视频课程学习效果更佳
www.zhougejiaoit.com

若编译报错:

CMake Error at CMakeLists.txt:65 (find_package):

Could not find a package configuration file provided by "Protobuf" with any of the following names:

ProtobufConfig.cmake

protobuf-config.cmake

Add the installation prefix of "Protobuf" to CMAKE_PREFIX_PATH or set

"Protobuf_DIR" to a directory containing one of the above files. If

"Protobuf" provides a separate development package or SDK, be sure it has been installed.

请执行:

1.cd grpc/third_party/protobuf/cmake/

2.mkdir build

3.cd build/

4.cmake ..

5.make && make install

then the proobuf.cmake config will be installed in cmake's dir.

然后继续编译 helloworld:

make

```
-- Using protobuf
-- Found ZLIB: /usr/local/lib/libz.so (found version "1.2.11")
-- Found OpenSSL: /usr/lib/x86_64-linux-gnu/libcrypto.so (found version "1.1.1f")
-- Using gRPC 1.20.0
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zyr/develop/grpc/examples/cpp/helloworld
zyr@5446e04b5bce:~/develop/grpc/examples/cpp/helloworld$ make
 5%] Generating helloworld.pb.cc, helloworld.pb.h, helloworld.grpc.pb.cc, helloworld.grpc.pb.h
10%] Building CXX object CMakeFiles/greeter_client.dir/greeter_client.cc.o
15%] Building CXX object CMakeFiles/greeter_client.dir/helloworld.pb.cc.o
20%] Building CXX object CMakeFiles/greeter_client.dir/helloworld.grpc.pb.cc.o
25%] Linking CXX executable greeter_client
25%] Built target greeter_client
30%] Building CXX object CMakeFiles/greeter_server.dir/greeter_server.cc.o
35%] Building CXX object CMakeFiles/greeter_server.dir/helloworld.pb.cc.o
40%] Building CXX object CMakeFiles/greeter_server.dir/helloworld.grpc.pb.cc.o
45%] Linking CXX executable greeter_server
50%] Built target greeter_server
55%] Building CXX object CMakeFiles/greeter_async_client.dir/greeter_async_client.cc.o
60%] Building CXX object CMakeFiles/greeter_async_client.dir/helloworld.pb.cc.o
65%] Building CXX object CMakeFiles/greeter_async_client.dir/helloworld.grpc.pb.cc.o
70%] Linking CXX executable greeter_async_client
75%] Built target greeter_async_client
80%] Building CXX object CMakeFiles/greeter_async_server.dir/greeter_async_server.cc.o
85%] Building CXX object CMakeFiles/greeter_async_server.dir/helloworld.pb.cc.o
90%] Building CXX object CMakeFiles/greeter_async_server.dir/helloworld.grpc.pb.cc.o
95%] Linking CXX executable greeter_async_server
100%] Built target greeter_async_server
```

ProtoBuf(1)

Protobuf 是一种轻便高效的结构化数据存储格式，可以用于结构化数据序列化。它很适合

周哥教 IT www.zhougejiaoit.com

做数据存储或 RPC 数据交换格式。可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。目前提供了 C++、Java、Python、Go 等语言的 API。

23, "tom" 写入文件

```
struct _Data{
    int age;
    char name[128];
}Data;

message Data{
    int age = 1;
    string name = 2;
}
{"age":23, "name":"tom"}
```

ProtoBuf(2) demo

23, "tom"

```
struct _Person{
    int age;
    char name[128];
}Person;

message Person{
    int32 age = 1;
    string name = 2;
}

{"age":23, "name":"tom"}
```

```
protoc -I ./ --cpp_out=./ ./person.proto -->person.pb.h, person.pb.cc
```

```
syntax = "proto3";
```

```
package zgjit;
```

```
message Person {
    uint32 age = 1;
    string name = 2;
}
```

```
#include <iostream>
#include <fstream>

#include "person.pb.h"

int SerializePerson() {

    zgjit::Person person;
    person.set_age(23);
    person.set_name("tom");

    std::fstream output("./person.dat", std::ios::out | std::ios::trunc | std::ios::binary);

    if (!person.SerializeToOstream(&output)) {
        std::cout << "Failed to write person.dat." << std::endl;
        return -1;
    }

    return 0;
}

int DeserializePerson () {
    zgjit::Person person;

    std::fstream input("./person.dat", std::ios::in | std::ios::binary);
    if (person.ParseFromIstream(&input)) {
        std::cout<<"name:"<<person.name()<<" "<<"age:"<< person.age()<<std::endl;
        return 0;

    } else {
        std::cout << "Failed to parse person.dat" << std::endl;
        return -1;
    }
}

int main(void) {

    if(SerializePerson() == 0) {
        std::cout<<"SerializePerson() succeed"<<std::endl;
        if(DeserializePerson() == 0) {
```

```
std::cout<<"DeserializePerson() succeed"<<std::endl;
    }
}

return 0;

}

cmake_minimum_required(VERSION 2.8)

project(protobufDemo)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")

set(protobuf_MODULE_COMPATIBLE TRUE)
find_package(Protobuf CONFIG REQUIRED)
message(STATUS "Using protobuf ${protobuf_VERSION}")

set(_PROTOBUF_LIBPROTOBUF protobuf::libprotobuf)

set(ps_proto_srcs "${CMAKE_CURRENT_BINARY_DIR}/person.pb.cc")
set(ps_proto_hdrs "${CMAKE_CURRENT_BINARY_DIR}/person.pb.h")

# Include generated *.pb.h files
include_directories("${CMAKE_CURRENT_BINARY_DIR}")

add_executable(protobufDemo "main.cc"
    ${ps_proto_srcs})
target_link_libraries(protobufDemo
    ${_PROTOBUF_LIBPROTOBUF})
```

gRPC 入门例子 helloworld 剖析

- 1, 功能: world-->hello world
- 2, proto: grpc/examples/protos :helloworld.proto
protoc
.grpc.pb.h, .grpc.pb.cc (Service, Stub)

.pb.h, .pb.cc

3, 代码: grpc/examples/cpp/helloworld:

client:GreeterClient: server:ip:port->channel->stub->sayHello

server:GreeterServiceImpl:Service

收发数据

4, 编译与 demo: CMakeLists.txt demo

gRPC Add Demo

1, 定义 protobuf 文件:request 结构, response 结构, rpc 接口

2, 编译 protobuf 文件: protoc

protoc -I ./protos --cpp_out=./ ./protos/add.proto

protoc -I ./protos --cpp_out=./ --grpc_out=./

--plugin=protoc-gen-grpc=/usr/local/bin/grpc_cpp_plugin ./protos/add.proto

3, 编写服务端

4, 编写客户端

5, 编译 (cmake) CMakeLists.txt 演示

```
syntax = "proto3";
```

```
package myadd;
```

```
service AddService {  
    rpc DoAdd(AddRequest) returns(AddResponse) {}  
}
```

```
message AddRequest {  
    sint32 x = 1;  
    sint32 y = 2;  
}
```

```
message AddResponse {  
    sint32 sum = 1;  
  
    int32 err_code = 2;  
    string err_msg = 3;  
}
```

```
#include <iostream>
```

```
#include <memory>
```

```
#include <string>
```

```
#include <grpcpp/grpcpp.h>

#include "add.grpc.pb.h"

using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;

using myadd::AddRequest;
using myadd::AddResponse;
using myadd::AddService;

class AddClient {
public:
    AddClient(std::shared_ptr<Channel> channel)
        : stub_(AddService::NewStub(channel)) {}

    // Assembles the client's payload, sends it and presents the response back
    // from the server.
    int DoAdd(int a, int b, int &sum) {
        // Data we are sending to the server.
        AddRequest request;
        request.set_x(a);
        request.set_y(b);

        // Container for the data we expect from the server.
        AddResponse response;

        // Context for the client. It could be used to convey extra information to
        // the server and/or tweak certain RPC behaviors.
        ClientContext context;

        gpr_timespec ts;
        ts.tv_sec = 6;
        ts.tv_nsec = 0;//纳秒
        ts.clock_type = GPR_TIMESPAN;
        context.set_deadline(ts);

        // The actual RPC.
        Status status = stub_->DoAdd(&context, request, &response);
```

```
std::cout<<"err_code:"<<response.err_code()<<"
err_msg:"<<response.err_msg()<<std::endl;

// Act upon its status.
if (status.ok()) {
    sum = response.sum();
    return response.err_code();
} else {
    std::cout<<status.error_code()<<":"<<status.error_message()<<std::endl;
    sum = 0;
    return -1;
}
}

private:
    std::unique_ptr<AddService::Stub> stub_;
};

int main(int argc, char** argv) {
    // Instantiate the client. It requires a channel, out of which the actual RPCs
    // are created. This channel models a connection to an endpoint (in this case,
    // localhost at port 50051). We indicate that the channel isn't authenticated
    // (use of InsecureChannelCredentials()).
    AddClient adder(grpc::CreateChannel(
        "localhost:50051", grpc::InsecureChannelCredentials()));
    int sum = 0;
    int res = adder.DoAdd(3, 5, sum);
    if(res==0) {
        std::cout<<"DoAdd succeeded, the sum is "<< sum <<std::endl;
    } else {
        std::cout<<"DoAdd failed"<<std::endl;
    }

    return 0;
}
#include <iostream>
#include <memory>
#include <string>
#include <thread>
```

```
#include <grpcpp/grpcpp.h>

#include "add.grpc.pb.h"

using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;

using myadd::AddRequest;
using myadd::AddResponse;
using myadd::AddService;

class AddServiceImpl final : public AddService::Service {
    Status DoAdd(ServerContext* context, const AddRequest* request,
                AddResponse* response) override {

        response->set_sum(request->x() + request->y());
        response->set_err_code(0);
        response->set_err_msg("success");

        std::this_thread::sleep_for(std::chrono::milliseconds(5000));
        return Status::OK;
    }
};

void RunServer() {
    std::string server_address("0.0.0.0:50051");
    AddServiceImpl service;

    ServerBuilder builder;
    // Listen on the given address without any authentication mechanism.
    builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
    // Register "service" as the instance through which we'll communicate with
    // clients. In this case it corresponds to an *synchronous* service.
    builder.RegisterService(&service);
    // Finally assemble the server.
    std::unique_ptr<Server> server(builder.BuildAndStart());
    std::cout << "Server listening on " << server_address << std::endl;

    // Wait for the server to shutdown. Note that some other thread must be
```

```
// responsible for shutting down the server for this call to ever return.
server->Wait();
}

int main(int argc, char** argv) {
    RunServer();

    return 0;
}
```

Protobuf 类型大全

各种基本类型

结构体

枚举 enum

数组

Map

```
syntax="proto3";
package mallocfree;
option java_package = "com.example.foo.mallocfree";
option go_package = "./;mallocfree"
```

```
message Data{
    int32 age = 1;
    string name = 2;
}
```

第一行指定了 proto3 语法：如果没有指定这个，编译器会使用 proto2

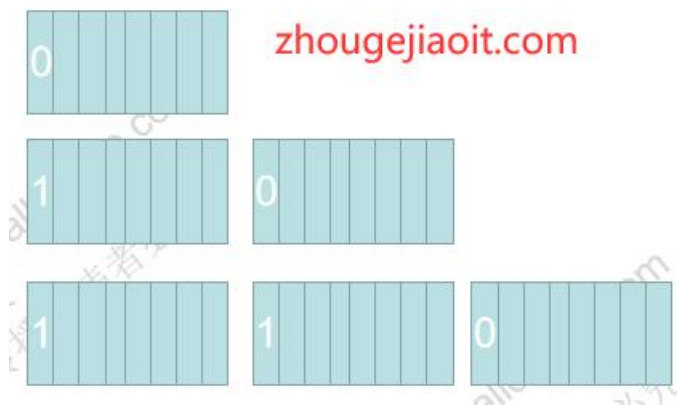
最小的标识号可以从 1 开始，最大到 $2^{29} - 1$, or 536,870,911。不可以使用其中的 [19000–19999]

每个字段都有唯一的一个数字标识符。这些标识符是用来在消息的二进制格式中识别各个字段的，一旦开始使用就不能够再改变。注：[1,15] 之内的标识号在编码的时候会占用一个字节。[16,2047] 之内的标识号则占用 2 个字节。所以应该为那些频繁出现的信息元素保留 [1,15] 之内的标识号。切记：要为将来有可能添加的、频繁出现的标识号预留一些标识号。

```
int32    可变长编码  fixed32
int64                                fixed64
uint32
uint64
sint32  有符号可变长编码  sfixed32
sint64                                sfixed64
```

bool
float
double
string
bytes

sint32/sint64, 与 int32/int64 在用于表示 负数 是有区别的。如果使用 int32 表示一个负数, 结果会占用 10 个字节, 实际上用了一个很大无符号数来表示。如果使用 sint32, 会使用 ZigZag 编码方式来提高效率(正负数的交错方式来表达, 当数值的绝对值小的时候, 会比使用 int32 的方式节省很多字节。例如-1 编码为 1, 1 编码为 2, -2 编码为 3)。



ProtoBuf 编写

```
syntax = "proto3";  
import "phone.proto";  
package humanface;  
  
service HumanFaceService {  
    // 上传人脸数据  
    rpc UploadFaceData (UploadFaceRequest) returns (UploadFaceResponse) {  
    }  
}  
  
message FaceLocation {  
    float x = 1;  
    float y = 2;  
    float z = 3;  
}  
  
message FaceInfo {  
    string faceid = 1;  
    string sex = 2;  
    FaceLocation facelocation = 3;  
}
```

```
}
message UploadFaceRequest {
    repeated FaceInfo faceinfoArray = 1;
    Os os = 2;
}
enum Os {
    Android = 0;
    iOS = 1;
}
message UploadFaceResponse {
    int32 err_code = 1;
    string err_msg = 2;
}

#include <iostream>
#include <memory>
#include <string>

#include <grpcpp/grpcpp.h>

#include "humanface.grpc.pb.h"

using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;

using humanface::UploadFaceRequest;
using humanface::UploadFaceResponse;
using humanface::HumanFaceService;

class FaceClient {
public:
    FaceClient(std::shared_ptr<Channel> channel)
        : stub_(HumanFaceService::NewStub(channel)) {}

    // Assembles the client's payload, sends it and presents the response back
    // from the server.
    int DoUploadFaceData() {
        // Data we are sending to the server.
        UploadFaceRequest request;
```

```
humanface::Os os = humanface::Os::Android;
request.set_os(os);

for(int i=0; i<5; i++) {
    humanface::FaceInfo *faceinfo = request.add_faceinfoarray();

    char buf[16] = {0};
    snprintf(buf, sizeof(buf), "%d", 1000+i);
    faceinfo->set_faceid(buf);

    faceinfo->set_sex("male");

    humanface::FaceLocation *facelocation = new humanface::FaceLocation();
    facelocation->set_x(1.0 + i);
    facelocation->set_y(2.0 + i);
    facelocation->set_z(3.0 + i);
    faceinfo->set_allocated_facelocation(facelocation);
    //delete facelocation;
}

// Container for the data we expect from the server.
UploadFaceResponse response;

// Context for the client. It could be used to convey extra information to
// the server and/or tweak certain RPC behaviors.
ClientContext context;

// The actual RPC.
Status status = stub_->UploadFaceData(&context, request, &response);

std::cout<<"err_code:"<<response.err_code()<<"
err_msg:"<<response.err_msg()<<std::endl;

// Act upon its status.
if (status.ok()) {
    return response.err_code();
} else {
    return -1;
}
}
```

```
private:
    std::unique_ptr<HumanFaceService::Stub> stub_;
};

int main(int argc, char** argv) {
    // Instantiate the client. It requires a channel, out of which the actual RPCs
    // are created. This channel models a connection to an endpoint (in this case,
    // localhost at port 50051). We indicate that the channel isn't authenticated
    // (use of InsecureChannelCredentials()).
    FaceClient faceClient(grpc::CreateChannel(
        "localhost:50051", grpc::InsecureChannelCredentials()));

    int res = faceClient.DoUploadFaceData();
    if(res==0) {
        std::cout<<"DoUploadFaceData succeeded"<<std::endl;
    } else {
        std::cout<<"DoUploadFaceData failed"<<std::endl;
    }

    return 0;
}

#include <iostream>
#include <memory>
#include <string>

#include <grpcpp/grpcpp.h>

#include "humanface.grpc.pb.h"

using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;

using humanface::UploadFaceRequest;
using humanface::UploadFaceResponse;
using humanface::HumanFaceService;
```

```
class HumanFaceServiceImpl final : public HumanFaceService::Service {
    Status UploadFaceData(ServerContext* context, const UploadFaceRequest* request,
        UploadFaceResponse* response) override {

        std::cout<<"client request in"<<std::endl;
        std::cout<<std::endl;

        switch(request->os()) {
            case humanface::Os::Android:
                std::cout<<"Os: Android"<<std::endl;
                break;
            case humanface::Os::iOS:
                std::cout<<"Os: iOS"<<std::endl;
                break;
            default:
                std::cout<<"Unknown os"<<std::endl;
        }

        std::cout<<std::endl;

        for(int i = 0; i < request->faceinfoarray_size(); i++) {
            humanface::FaceInfo faceinfo = std::move(request->faceinfoarray()[i]);
            std::cout<<"faceid:"<<faceinfo.faceid().c_str()<<"
sex:"<<faceinfo.sex().c_str()<<std::endl;
            humanface::FaceLocation facelocation = faceinfo.facelocation();
            std::cout<<"face    location:"<<"x:"<<facelocation.x()<<"    y:"<<facelocation.y()<<"
z:"<<facelocation.z()<<std::endl;
            std::cout<<std::endl;
        }

        response->set_err_code(0);
        response->set_err_msg("success");

        std::cout<<"client request finished"<<std::endl;

        return Status::OK;
    }
};

void RunServer() {
    std::string server_address("0.0.0.0:50051");
```

```
HumanFaceServiceImpl service;

ServerBuilder builder;
// Listen on the given address without any authentication mechanism.
builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
// Register "service" as the instance through which we'll communicate with
// clients. In this case it corresponds to an *synchronous* service.
builder.RegisterService(&service);
// Finally assemble the server.
std::unique_ptr<Server> server(builder.BuildAndStart());
std::cout << "Server listening on " << server_address << std::endl;

// Wait for the server to shutdown. Note that some other thread must be
// responsible for shutting down the server for this call to ever return.
server->Wait();
}

int main(int argc, char** argv) {
    RunServer();

    return 0;
}
```

Protobuf: 简单, 快, , 二进制存储无法编辑, 可使用一些工具实现 protobuf 与 json 转化。
json2pb

1, 使用 bytes 而不是 string

protobuf 的 bytes 和 string 都能表示字符串都对应 C++ 中的 std::string, 但是 string 类型会对字符串做 utf8 格式校验, 而 bytes 不会, 因此使用 bytes 的编解码效率更高。bytes 适用于任意的二进制字节序列

2, 使用 optional 而不是 required 字段。

optional 字段是可选的, 存在与否不影响 proto 对象的序列化和反序列化, 可向后和向前兼容, 以后增加新的字段, 或弃用旧字段都不需要修改代码。

required 字段要求字段必须存在, 否则会导致 proto 解析失败。将来随着业务的发展可能会成为负担。无法兼容

proto3 里, 默认即为 optional

较复杂 protobuf 的 gRPC 应用

```
syntax = "proto3";
```

```
package humanface;
```

```
service HumanFaceService {
    // 上传人脸数据
    rpc UploadFaceData (UploadFaceRequest) returns (UploadFaceResponse) {
    }
}

message FaceLocation {
    float x = 1;
    float y = 2;
    float z = 3;
}

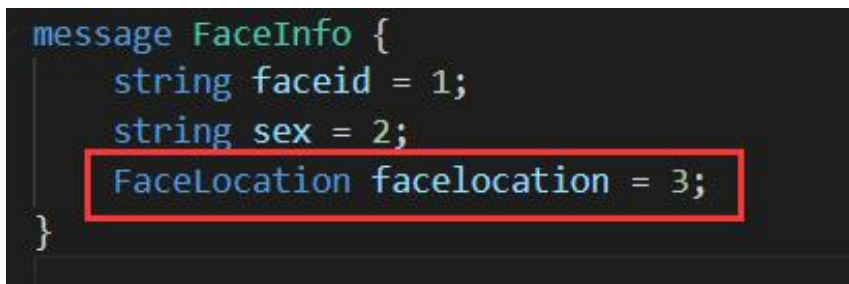
message FaceInfo {
    string faceid = 1;
    string sex = 2;
    FaceLocation facelocation = 3;
}

enum Os {
    Android = 0;
    iOS = 1;
}

message UploadFaceRequest {
    repeated FaceInfo faceinfoArray = 1;
    Os os = 2;
}

message UploadFaceResponse {
    int32 err_code = 1;
    string err_msg = 2;
}
```

protobuf 中的坑



```
message FaceInfo {
    string faceid = 1;
    string sex = 2;
    FaceLocation facelocation = 3;
}
```

```
UploadFaceRequest request;

humanface::Os os = humanface::Os::Android;
request.set_os(os);

for(int i=0; i<5; i++) {
    humanface::FaceInfo *faceinfo = request.add_faceinfoarray();

    char buf[16] = {0};
    snprintf(buf, sizeof(buf), "%d", 1000+i);
    faceinfo->set_faceid(buf);

    faceinfo->set_sex("male");

    humanface::FaceLocation *facelocation = new humanface::FaceLocation();
    facelocation->set_x(1.0 + i);
    facelocation->set_y(2.0 + i);
    facelocation->set_z(3.0 + i);
    faceinfo->set_allocated_facelocation(facelocation);
    //delete facelocation;
}
```

protobuf VS json

protobuf 优点:

- 1、性能好/效率高:序列化快,存储空间小,JSON 是文本的格式,整数和浮点数应该更占空间而且更费时(1000," 1000 ").jackson, fastjson 效率区别大
- 2、有代码生成机制,自带了一个编译器 protoc,只需要用它进行编译,可以编译成 JAVA、python、C++, GO 代码
- 3、支持向后兼容和向前兼容
- 4、支持多种编程语言

protobuf 缺点:

- 1、二进制格式导致可读性差
- 2、缺乏自描述,协议内容必须配合结构体解析
- 3、通用性差,仍然并不是一个传输标准。比 json 和 XML,通用性还是没那么好

gRPC 四种模式

- 1、简单模式 (Simple RPC) 一元模式(Unary RPC)

客户端发起请求,一直等待,直到服务端响应

- 2、客户端数据流模式 (Client-side streaming RPC)

客户端持续向服务端发送数据流，在发送结束后，由服务端返回一个响应。

客户端 `write`，服务端 `read`

案例：（多个）监控摄像头的的数据，实时、持续上传到服务器

3、服务端数据流模式（Server-side streaming RPC）

客户端发起一次请求，服务端可以连续返回数据流。

服务端 `write`，客户端 `read`

案例：客户端向服务端发送一个股票代码，服务端持续返回该股票的实时数据。

4、双向数据流模式（Bidirectional streaming RPC）

客户端和服务端都可以向对方发送数据流。

案例：聊天机器人。

```
syntax = "proto3";
package transferfile;
service TransferFile {
    //Client-side streaming RPC
    //write-->read
    rpc UploadFile(stream FileChunk) returns (UploadReply) {}

    //Server-side streaming RPC
    //read<--write
    rpc DownloadFile(DownloadRequest) returns (stream FileChunk) {}

    //Bidirectional streaming RPC
    //write-->read
    //read<--write
    rpc Talk(stream TalkContent) returns (stream TalkContent) {}
}
message FileChunk {
    bytes buffer = 1;
}
message UploadReply {
    int32 length = 1;
}
message DownloadRequest {
    string filename = 1;
}

message TalkContent {
```

```
        string content = 1;
    }

#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <thread>
#include <chrono>

#include <grpc/grpc.h>
#include <grpc++/channel.h>
#include <grpc++/client_context.h>
#include <grpc++/create_channel.h>
#include <grpc++/security/credentials.h>

#include "filetransfer.grpc.pb.h"
using grpc::Channel;
using grpc::ClientContext;
using grpc::ClientWriter;
using grpc::Status;
using transferfile::FileChunk;
using transferfile::UploadReply;
using transferfile::DownloadRequest;
using transferfile::TransferFile;
using transferfile::TalkContent;
#define CHUNK_SIZE 1024 * 1024
class TransferFileClient
{
public:
    TransferFileClient(std::shared_ptr<Channel> channel) :
        stub_(TransferFile::NewStub(channel));
    void UploadFile(std::string filename)
    {
        FileChunk chunk;
        char data[CHUNK_SIZE];
        UploadReply reply;
        ClientContext context;

        std::ifstream infile;
```

```
infile.open(filename, std::ifstream::in | std::ifstream::binary);

std::unique_ptr<ClientWriter<FileChunk>> writer(stub_->UploadFile(&context,
&reply));
while (!infile.eof()) {
    infile.read(data, CHUNK_SIZE);
    chunk.set_buffer(data, infile.gcount());
    if (!writer->Write(chunk)) {
        break;
    }
}
writer->WritesDone();
Status status = writer->Finish();
if (status.ok()) {
    std::cout<<"file transfered success, sent:"<<reply.length()<<" bytes"<<std::endl;
} else {
    std::cout << "TransferFile rpc failed." << std::endl;
    std::cout<<"err code:"<< status.error_code()<<std::endl;
}
}

void DownloadFile(std::string file)
{
    DownloadRequest request;
    request.set_filename(file);

    std::string filename = "./zyr-download.jpg";
    FileChunk chunk;
    std::ofstream outfile;
    size_t filelen = 0;
    outfile.open(filename, std::ofstream::out | std::ofstream::trunc |
std::ofstream::binary);
    grpc::ClientContext context;

    std::unique_ptr<grpc::ClientReader<FileChunk>>
reader(stub_->DownloadFile(&context, request));
    while(reader->Read(&chunk)) {
        const char *data = chunk.buffer().c_str();
        outfile.write(data, chunk.buffer().length());
        filelen += chunk.buffer().length();
    }
}
```

```
}
outfile.close();

grpc::Status status = reader->Finish();
if (status.ok()) {
    std::cout<<"file download success, received:"<<filelen<<" bytes"<<std::endl;
} else {
    std::cout << "TransferFile rpc failed." << std::endl;
    std::cout<<"err code:"<< status.error_code()<<std::endl;
}

}

void Talk(const std::vector<std::string> & talks) {
    grpc::ClientContext context;
    std::shared_ptr<grpc::ClientReaderWriter<TalkContent, TalkContent>>
stream(stub_->Talk(&context));

    std::thread worker([stream, &talks]() {
        TalkContent content;
        for (const auto & talk : talks) {
            content.set_content(talk);
            if (!stream->Write(content)) {
                break;
            }
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    });
    stream->WritesDone();
});

TalkContent talkContent;
while (stream->Read(&talkContent)) {
    std::cout <<talkContent.content()<< std::endl;;
}

worker.join();
grpc::Status status = stream->Finish();
if(status.ok()) {
    std::cout<<"talk finished success"<<std::endl;
} else {
    std::cout << "talk rpc failed." << std::endl;
}
```

```
        std::cout<<"err code:"<< status.error_code()<<std::endl;
    }
}
private:
    std::unique_ptr<TransferFile::Stub> stub_;
};

int main(int argc, char** argv){

    TransferFileClient transferFile(grpc::CreateChannel("localhost:50052",
        grpc::InsecureChannelCredentials()));
    std::cout<<"Press enter key to start upload file"<<std::endl;
    std::cin.ignore();
    transferFile.UploadFile("./zyr1.jpg");

    std::cout<<"Press enter key to start download file"<<std::endl;
    std::cin.ignore();
    transferFile.DownloadFile("./zyr2.jpg");

    std::cout<<"Press enter key to start talk"<<std::endl;
    std::cin.ignore();
    std::vector<std::string> talksVec = {"my name is:tom", "my name is:susan",
        "my name is:lily", "my name is:jacob", "my name is:david"};
    transferFile.Talk(talksVec);

    return 0;

}

#include <iostream>
#include <fstream>
#include <string>
#include <grpc/grpc.h>
#include <grpc++/server.h>
#include <grpc++/server_builder.h>
#include <grpc++/server_context.h>
#include <grpc++/security/server_credentials.h>
#include "filetransfer.grpc.pb.h"

#define CHUNK_SIZE 1024 * 1024
```

```
using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::ServerReader;
using grpc::Status;

using transferfile::FileChunk;
using transferfile::TalkContent;
using transferfile::UploadReply;
using transferfile::TransferFile;
#define CHUNK_SIZE 1024 * 1024

class TransferFileImpl final : public TransferFile::Service {
public:
    Status UploadFile(ServerContext* context, ServerReader<FileChunk>* reader, UploadReply*
reply) {
        FileChunk chunk;
        const char *filename = "./zyr2.jpg";
        std::ofstream outfile;
        const char *data;
        outfile.open(filename, std::ofstream::out | std::ofstream::trunc |
std::ofstream::binary);
        while (reader->Read(&chunk)) {
            std::cout<<"file transferring from client"<<std::endl;
            data = chunk.buffer().c_str();
            outfile.write(data, chunk.buffer().length());
        }
        long pos = outfile.tellp();
        reply->set_length(pos);
        outfile.close();
        return Status::OK;
    }

    Status DownloadFile(grpc::ServerContext * context,
                        const transferfile::DownloadRequest * request,
                        grpc::ServerWriter<FileChunk> * writer) override {
        std::string filename = request->filename();
        std::ifstream infile;
        char data[CHUNK_SIZE];
```

```
FileChunk chunk;

infile.open(filename, std::ifstream::in | std::ifstream::binary);
while (!infile.eof()) {
    std::cout<<"file transferring to client"<<std::endl;
    infile.read(data, CHUNK_SIZE);
    chunk.set_buffer(data, infile.gcount());
    if (!writer->Write(chunk)) {
        // Broken stream.
        break;
    }
}
return grpc::Status::OK;
}

Status Talk(grpc::ServerContext * context,
            grpc::ServerReaderWriter<TalkContent, TalkContent>
* stream) override {
    TalkContent talkContent;
    while(stream->Read(&talkContent)) {
        auto msg = talkContent.content();
        std::cout<<msg<<std::endl;
        size_t pos = msg.find(':');
        if(pos!=std::string::npos) {
            std::string name = msg.substr(pos+1);
            talkContent.set_content("nice to meet u, " + name);
            stream->Write(talkContent);
        }
    }
    return grpc::Status::OK;
}

};

void RunServer() {
    std::string server_address("0.0.0.0:50052");
    TransferFileImpl service;
    ServerBuilder builder;
    builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
```

```
builder.RegisterService(&service);
std::unique_ptr<Server> server(builder.BuildAndStart());
std::cout << "Server listening on " << server_address << std::endl;
server->Wait();
}

int main(int argc, char** argv) {
    RunServer();
    return 0;
}
```

gRPC 超时设置

```
// 设置同步超时时间
grpc::ClientContext context;
gpr_timespec ts;
ts.tv_sec = 5;
ts.tv_nsec = 0;//纳秒
ts.clock_type = GPR_TIMESPAN;
context.set_deadline(ts);

Status status = stub_->rpcFunc(&context, request, reply);
std::cout << status.error_code() << ": " << status.error_message()
    << std::endl;

DEADLINE_EXCEEDED = 4
```

异步超时？

异步 GRPC (1)

同步与异步

```
ReadFile,read/write
receive/send
Status status = stub_->SayHello(&context, request, reply);
为什么要异步
导弹发射，一直盯着和发射之后不用管。谁的战斗力更强？
```

异步 GRPC (2) :客户端异步

gRPC 的演示样例 greeter_async_client.cc 不是一个真正的异步客户端，它虽然使用了异步请求，却是阻塞式等待应答，结果变为了一个同步调用。

异步 GRPC (3) :服务端异步

调用 RequestSayHello 开启处理流程（相当于注册了一个处理器）-》然后异步请求调用到达-》根据 RequestSayHello 参数加入一个队列-》从队列中取出数据处理-》调用 responder.Finish 发送-》发送状态又会入队列-》清理这次的处理流程。

通过 CallData 对象，封装了一个异步请求的所以动作；

gRPC 异步超时设置

```
Status status;
std::unique_ptr<ClientAsyncResponseReader<HelloReply> > rpc(
    stub_->AsyncSayHello(&context, request, &cq));

    rpc->Finish(&reply, &status, (void*)1);
    void* got_tag;
    bool ok = false;

gpr_timespec time;
time.tv_sec = 2;//设置 2 秒超时
time.tv_nsec = 0;
time.clock_type = GPR_TIMESPAN;
CompletionQueue::NextStatus nextStatus =cq.AsyncNext(&got_tag, &ok, time);
if(nextStatus==GOT_EVENT){
    GPR_ASSERT(got_tag == (void*)1);
    GPR_ASSERT(ok);
}else if(nextStatus==TIMEOUT){
//超时
}
```

其他例子

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

Go

Java

Python 等

gRPC vs. Restful API

Restful API(基于 http, 提供 get, post, put, delete, C/C++中主要是基于 CURL 实现)
gRPC 和 Restful API 都提供了一套通信机制, 用于 server/client 模型通信, 而且它们都使用 http 作为底层的传输协议(严格地说, gRPC 使用的 http2.0, 而 restful api 则不一定)。

不过 gRPC 特有优势:

- 1, gRPC 可以通过 `protobuf` 来定义接口, 从而可以有更加严格的接口约束条件。我们不希望客户端给我们传任意数据, 尤其是考虑到安全性的因素, 通常需要对接口进行更加严格的约束。这时 gRPC 就可以通过 `protobuf` 来提供严格的接口约束。
- 2, 另外, 通过 `protobuf` 可以将数据序列化为二进制编码, 这会大幅减少需要传输的数据量, 从而大幅提高性能。
- 3, gRPC 可以方便地支持流式通信(理论上通过 http2.0 就可以使用 `streaming` 模式, 但是通常 web 服务的 `restful api` 似乎很少这么用, 通常的流式数据应用如视频流, 一般都会使用专门的协议如 HLS, RTMP 等, 这些不是通常的 web 服务而是有专门的服务器应用。)