grpc for c++使用案例

腾讯课堂 零声教育 C/C++Linux服务器开发/高级架构【零声教育】-学习视频教程-腾讯课堂 (qq.com)

手把手写rpc范例流程：

1. 编写proto文件
2. 根据proto文件生成对应的.cc和.h文件
3. server程序继承Service
4. client程序使用stub
5. 编译server和client程序。

1 proto文件编辑

IM.Login.proto

```
syntax = "proto3";
package IM.Login;

// 定义服务
service ImLogin {
  // 定义服务函数
  rpc Regist (IMRegistReq) returns (IMRegistRes) {}
  rpc Login (IMLoginReq) returns (IMLoginRes){}
}

// 注册账号
message IMRegistReq{
    string user_name = 1; // 用户名
    string password = 2;  // 密码
}

message IMRegistRes{
    string user_name = 1;
    uint32 user_id = 2;
    uint32 result_code = 3;    // 返回0的时候注册正常
}

// 登录账号
message IMLoginReq{
    string user_name = 1;
    string password = 2;
}

message IMLoginRes{
    uint32 user_id = 1;
    uint32 result_code = 2;  // 返回0的时候正确
}
```

第二行：包的名字 package Player; 是 Player

第五行：grpc服务的名称 service ImLogin 是 ImLogin

后面的请求响应结构，无论名字是大写还是小写，最终调用的时候都是小写。

2 生成C++代码

1、生成protobuf（反）序列化代码

在 .proto 文件中定义了数据结构，这些数据结构是面向开发者和业务程序的，并不面向存储和传输。当需要把这些数据进行存储或传输时，就需要将这些结构数据进行序列化、反序列化以及读写。通过 protoc 这个编译器，ProtoBuf 将会为我们提供相应的接口代码。可通过如下命令生成相应的接口代码：

```
// $SRC_DIR: .proto 所在的源目录
// --cpp_out: 生成 c++ 代码
// $DST_DIR: 生成代码的目标目录
// xxx.proto: 要针对哪个 proto 文件生成接口代码

protoc -I=$SRC_DIR --cpp_out=$DST_DIR  $SRC_DIR/xxx.proto

// 如：
protoc -I ./ --cpp_out=. simple.proto
```

- -I 是 --proto_path 的缩写形式，$SRC_DIR 指定在解析导入指令时查找 .proto 文件的目录。如果省略，则使用当前目录。可以通过多次传递 --proto_path 选项来指定多个导入目录，他们将按顺序搜索。
- 执行下面命令后，将在$DST_DIR目录下生成 xxx.pb.h 和 http://xxx.ph.cc文件

2、生成服务框架代码

```
// 执行下面命令后，将在当前目录下生成 simple.grpc.pb.h 和 simple.grpc.pb.cc 文件
protoc -I ./ --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin`
simple.proto
// 上面的 `which grpc_cpp_plugin` 也可以替换为 grpc_cpp_plugin 程序的路径（如果不在系统
PATH下）
// 生成的代码需要依赖第一步生成序列化代码，所以在使用的时候必须都要有（生成的时候不依赖）
```

这里我们生成对应要使用的文件：

```
protoc --cpp_out=. IM.Login.proto

protoc --cpp_out=. --grpc_out=. --plugin=protoc-gen-
grpc=/usr/local/bin/grpc_cpp_plugin IM.Login.proto

protoc -I . --cpp_out=. --grpc_out=. --plugin=protoc-gen-grpc=`which
grpc_cpp_plugin`  *.proto
```

生成C++代码IM.Login.pb.h和IM.Login.pb.cc， IM.Login.grpc.pb.h和IM.Login.grpc.pb.cc。

3 grpc server端

命名空间

记住首先一定要开放命名空间

```
// 命名空间
// grcp
using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;
// 自己proto文件的命名空间
using IM::Login::ImLogin;
using IM::Login::IMRegistReq;
using IM::Login::IMRegistRes;
using IM::Login::IMLoginReq;
using IM::Login::IMLoginRes;
```

重写服务

定义服务端的类，继承 proto 文件定义的 grpc 服务 ImLogin，重写 grpc 服务定义的方法

```
class IMLoginServiceImpl : public ImLogin::Service {
    // 注册
    virtual  Status Regist(ServerContext* context, const IMRegistReq* request,
IMRegistRes* response) override {
        std::cout << "Regist user_name: " << request->user_name() << std::endl;

        response->set_user_name(request->user_name());
        response->set_user_id(10);
        response->set_result_code(0);

        return Status::OK;
    }

    // 登录
    virtual  Status Login(ServerContext* context, const IMLoginReq* request,
IMLoginRes* response) override {
        std::cout << "Login user_name: " << request->user_name() << std::endl;
        response->set_user_id(10);
        response->set_result_code(0);
        return Status::OK;
    }

};
```

启动服务

```
std::string server_address("0.0.0.0:50051");
  /* 定义重写的服务类 */
  ImLoginServiceImpl service;

  /* 创建工厂类 */
  ServerBuilder builder;
  /* 监听端口和地址 */
  builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());

  builder.AddChannelArgument(GRPC_ARG_KEEPALIVE_TIME_MS, 5000);
  builder.AddChannelArgument(GRPC_ARG_KEEPALIVE_TIMEOUT_MS, 10000);
  builder.AddChannelArgument(GRPC_ARG_KEEPALIVE_PERMIT_WITHOUT_CALLS, 1);
  /* 注册服务 */
  builder.RegisterService(&service);
  /** 创建和启动一个RPC服务器*/
  std::unique_ptr<Server> server(builder.BuildAndStart());
  std::cout << "Server listening on " << server_address << std::endl;
  /* 进入服务事件循环 */
  server->Wait();
```

完整代码

```
#include <iostream>
#include <string>

// grpc头文件
#include <grpcpp/ext/proto_server_reflection_plugin.h>
#include <grpcpp/grpcpp.h>
#include <grpcpp/health_check_service_interface.h>

// 包含我们自己proto文件生成的.h
#include "IM.Login.pb.h"
#include "IM.Login.grpc.pb.h"

// 命名空间
// grcp
using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;
// 自己proto文件的命名空间
using IM::Login::ImLogin;
using IM::Login::IMRegistReq;
using IM::Login::IMRegistRes;
using IM::Login::IMLoginReq;
using IM::Login::IMLoginRes;


class IMLoginServiceImpl : public ImLogin::Service {
    // 注册
    virtual  Status Regist(ServerContext* context, const IMRegistReq* request,
IMRegistRes* response) override {
        std::cout << "Regist user_name: " << request->user_name() << std::endl;
```

```
        response->set_user_name(request->user_name());
        response->set_user_id(10);
        response->set_result_code(0);

        return Status::OK;
    }

    // 登录
    virtual  Status Login(ServerContext* context, const IMLoginReq* request,
IMLoginRes* response) override {
        std::cout << "Login user_name: " << request->user_name() << std::endl;
        response->set_user_id(10);
        response->set_result_code(0);
        return Status::OK;
    }

};


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

    // 创建一个服务类
    IMLoginServiceImpl service;

    ServerBuilder builder;

    builder.AddListeningPort(server_addr, grpc::InsecureServerCredentials());
    builder.AddChannelArgument(GRPC_ARG_KEEPALIVE_TIME_MS, 5000);
    builder.AddChannelArgument(GRPC_ARG_KEEPALIVE_TIMEOUT_MS, 10000);
    builder.AddChannelArgument(GRPC_ARG_KEEPALIVE_PERMIT_WITHOUT_CALLS, 1);
    builder.RegisterService(&service);

    //创建/启动
    std::unique_ptr<Server> server(builder.BuildAndStart());
    std::cout << "Server listening on " << server_addr << std::endl;
    // 进入服务循环
    server->Wait();
}

// 怎么编译?
// 手动编译
// 通过cmake的方式
int main(int argc, const char** argv)
{
    RunServer();
    return 0;
}
```

4 grpc client端

命名空间

记住首先一定要开放命名空间

```
// 命名空间
// grcp
using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;
// 自己proto文件的命名空间
using IM::Login::ImLogin;
using IM::Login::IMRegistReq;
using IM::Login::IMRegistRes;
using IM::Login::IMLoginReq;
using IM::Login::IMLoginRes;
```

定义客户端

定义客户端的类，实现两个方法用来发送grpc请求以及接收grpc响应

```
class ImLoginClient
{
public:
    ImLoginClient(std::shared_ptr<Channel> channel)
        : stub_(ImLogin::NewStub(channel)) {}
    std::string Regist(const std::string &user)
    {
        IMRegistReq request;
        request.set_name(user);
        request.set_test("test");
        IMRegistRes reply;
        ClientContext context;
        Status status = stub_->Regist(&context, request, &reply);
        if (status.ok())
        {
            return reply.message();
        }
        else
        {
            std::cout << status.error_code() << ": " << status.error_message()<<
std::endl;
            return "RPC failed";
        }
    }

std::string Test(const std::string &user)
{
    ClientContext context;
    IMLoginReq req;
    IMLoginRes res;
```

```
    req.set_sessionid(10);
    req.set_elestmname("BStar");

    Status status = stub_->Login(&context, req, &res);

    if (status.ok())
    {
        std::cout << "result:" << res.result() << std::endl;
        return res.extension();
    }
    else
    {
        std::cout << status.error_code() << ": " << status.error_message()<<
std::endl;
        return "RPC failed";
    }
}
private:
    std::unique_ptr<ImLogin::Stub> stub_;
};
```

完整代码

```
#include <iostream>
#include <memory>
#include <string>
// /usr/local/include/grpcpp/grpcpp.h
#include <grpcpp/grpcpp.h>

// 包含我们自己proto文件生成的.h
#include "IM.Login.pb.h"
#include "IM.Login.grpc.pb.h"

// 命名空间
// grcp
using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;
// 自己proto文件的命名空间
using IM::Login::ImLogin;
using IM::Login::IMRegistReq;
using IM::Login::IMRegistRes;
using IM::Login::IMLoginReq;
using IM::Login::IMLoginRes;


class ImLoginClient
{
public:
    ImLoginClient(std::shared_ptr<Channel> channel)
        :stub_(ImLogin::NewStub(channel)) {

    }
    void Regist(const std::string &user_name, const std::string &password) {
        IMRegistReq request;
```

```
        request.set_user_name(user_name);
        request.set_password(password);

        IMRegistRes response;
        ClientContext context;
        std::cout <<  "-> Regist req" << std::endl;
        Status status = stub_->Regist(&context, request, &response);
        if(status.ok()) {
            std::cout <<  "user_name:" << response.user_name() << ", user_id:"
<< response.user_id() << std::endl;
        } else {
            std::cout <<  "user_name:" << response.user_name() << "Regist
failed: " << response.result_code()<< std::endl;
        }
    }

     void Login(const std::string &user_name, const std::string &password) {
        IMLoginReq request;
        request.set_user_name(user_name);
        request.set_password(password);

        IMLoginRes response;
        ClientContext context;
        std::cout <<  "-> Login req" << std::endl;
        Status status = stub_->Login(&context, request, &response);
        if(status.ok()) {
            std::cout <<  "user_id:" << response.user_id() << " login ok" <<
std::endl;
        } else {
            std::cout <<  "user_name:" << request.user_name() << "Login failed:
" << response.result_code()<< std::endl;
        }
    }

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

// 照葫芦画瓢
int main()
{
    // 服务器的地址
    std::string server_addr = "localhost:50051";
    ImLoginClient im_login_client(
        grpc::CreateChannel(server_addr, grpc::InsecureChannelCredentials())
    );
    std::string user_name = "darren";
    std::string password = "123456";
    im_login_client.Regist(user_name, password);
    im_login_client.Login(user_name, password);

    return 0;
}
```

5 异步grcp server端

server_async.cc

```
/*
 *
 * Copyright 2015 gRPC authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 */

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

#include "IM.Login.grpc.pb.h"
#include "IM.Login.pb.h"

#include <grpc/support/log.h>
#include <grpcpp/grpcpp.h>

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

// 自己proto文件的命名空间
using IM::Login::ImLogin;
using IM::Login::IMLoginReq;
using IM::Login::IMLoginRes;
using IM::Login::IMRegistReq;
using IM::Login::IMRegistRes;

class ServerImpl final {
 public:
    ~ServerImpl() {
        server_->Shutdown();
        // Always shutdown the completion queue after the server.
        cq_->Shutdown();
    }

    // There is no shutdown handling in this code.
    void Run() {  // 启动
        std::string server_address("0.0.0.0:50051");
```

```
        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 *asynchronous* service.
        builder.RegisterService(&service_);
        // Get hold of the completion queue used for the asynchronous
communication
        // with the gRPC runtime.
        cq_ = builder.AddCompletionQueue();
        // Finally assemble the server.
        server_ = builder.BuildAndStart();
        std::cout << "Server listening on " << server_address << std::endl;

        // Proceed to the server's main loop.
        HandleRpcs();
    }

 private:
    // Class encompasing the state and logic needed to serve a request.
    class CallData {
    public:
        // Take in the "service" instance (in this case representing an
asynchronous
        // server) and the completion queue "cq" used for asynchronous
communication
        // with the gRPC runtime.
        CallData(ImLogin::AsyncService* service, ServerCompletionQueue* cq)
            : service_(service), cq_(cq), status_(CREATE) {
            std::cout << "CallData constructing, this: " << this
                    << std::endl;  // darren 增加
            // Invoke the serving logic right away.
            Proceed();
        }
        virtual ~CallData(){}
        virtual void Proceed() {
            // std::cout << "CallData Prceed" << std::endl//;
            return;
        }
        // 通用的
        // The means of communication with the gRPC runtime for an asynchronous
        // server.
        ImLogin::AsyncService* service_;
        // The producer-consumer queue where for asynchronous server
notifications.
        ServerCompletionQueue* cq_;
        // Context for the rpc, allowing to tweak aspects of it such as the use
        // of compression, authentication, as well as to send metadata back to
the
        // client.
        ServerContext ctx_;
        // 有差异的
        // What we get from the client.
        // IMRegistReq request_;
        // // What we send back to the client.
        // IMRegistRes reply_;
```

```
        // // The means to get back to the client.
        // ServerAsyncResponseWriter<IMRegistRes> responder_;

        // Let's implement a tiny state machine with the following states.
        enum CallStatus { CREATE, PROCESS, FINISH };
        CallStatus status_;  // The current serving state.
    };

    class RegistCallData : public CallData {
    public:
        RegistCallData(ImLogin::AsyncService* service, ServerCompletionQueue*
cq)
            :CallData(service, cq), responder_(&ctx_) {
            Proceed();
        }
        ~RegistCallData() {}
        void Proceed() override {
            // std::cout << "RegistCallData Prceed" << std::endl//;
            std::cout << "this: " << this
                    << " RegistCallData Proceed(), status: " << status_
                    << std::endl;   // darren 增加
            if (status_ == CREATE) {  // 0
                std::cout << "this: " << this << " RegistCallData Proceed(),
status: "
                            << "CREATE" << std::endl;
                // Make this instance progress to the PROCESS state.
                status_ = PROCESS;

                // As part of the initial CREATE state, we *request* that the
system
                // start processing SayHello requests. In this request, "this"
acts are
                // the tag uniquely identifying the request (so that different
CallData
                // instances can serve different requests concurrently), in this
case
                // the memory address of this CallData instance.

                service_->RequestRegist(&ctx_, &request_, &responder_, cq_, cq_,
this);

            } else if (status_ == PROCESS) {  // 1
                std::cout << "this: " << this << " RegistCallData Proceed(),
status: "
                            << "PROCESS" << std::endl;
                // Spawn a new CallData instance to serve new clients while we
process
                // the one for this CallData. The instance will deallocate itself
as
                // part of its FINISH state.
                new RegistCallData(service_, cq_);  // 1. 创建处理逻辑

                reply_.set_user_name(request_.user_name());
                reply_.set_user_id(10);
                reply_.set_result_code(0);
```

```
                // And we are done! Let the gRPC runtime know we've finished,
using the
                // memory address of this instance as the uniquely identifying
tag for
                // the event.
                status_ = FINISH;
                responder_.Finish(reply_, Status::OK, this);
            } else {
                std::cout << "this: " << this << " RegistCallData Proceed(),
status: "
                          << "FINISH" << std::endl;
                GPR_ASSERT(status_ == FINISH);
                // Once in the FINISH state, deallocate ourselves
(RegistCallData).
                delete this;
            }
        }
    private:
        IMRegistReq request_;
        IMRegistRes reply_;
        ServerAsyncResponseWriter<IMRegistRes> responder_;
    };

    class LoginCallData : public CallData {
    public:
        LoginCallData(ImLogin::AsyncService* service, ServerCompletionQueue* cq)
            :CallData(service, cq), responder_(&ctx_) {
            Proceed();
        }
        ~LoginCallData() {}
        void Proceed() override {
            // std::cout << "LoginCallData Prceed" << std::endl//;
            std::cout << "this: " << this
                    << " LoginCallData Proceed(), status: " << status_
                    << std::endl;   // darren 增加
            if (status_ == CREATE) {  // 0
                std::cout << "this: " << this << " LoginCallData Proceed(),
status: "
                          << "CREATE" << std::endl;
                // Make this instance progress to the PROCESS state.
                status_ = PROCESS;

                // As part of the initial CREATE state, we *request* that the
system
                // start processing SayHello requests. In this request, "this"
acts are
                // the tag uniquely identifying the request (so that different
CallData
                // instances can serve different requests concurrently), in this
case
                // the memory address of this CallData instance.

                service_->RequestLogin(&ctx_, &request_, &responder_, cq_, cq_,
this);

            } else if (status_ == PROCESS) {  // 1
                std::cout << "this: " << this << " LoginCallData Proceed(),
status: "
```

```
                            << "PROCESS" << std::endl;
                // Spawn a new CallData instance to serve new clients while we
process
                // the one for this CallData. The instance will deallocate itself
as
                // part of its FINISH state.
                new LoginCallData(service_, cq_);  // 1. 创建处理逻辑

                reply_.set_user_id(10);
                reply_.set_result_code(0);

                // And we are done! Let the gRPC runtime know we've finished,
using the
                // memory address of this instance as the uniquely identifying
tag for
                // the event.
                status_ = FINISH;
                responder_.Finish(reply_, Status::OK, this);
            } else {
                std::cout << "this: " << this << " LoginCallData Proceed(),
status: "
                            << "FINISH" << std::endl;
                GPR_ASSERT(status_ == FINISH);
                // Once in the FINISH state, deallocate ourselves
(LoginCallData).
                delete this;
            }
        }
    private:
        IMLoginReq request_;
        IMLoginRes reply_;
        ServerAsyncResponseWriter<IMLoginRes> responder_;
    };
    // This can be run in multiple threads if needed.
    void HandleRpcs() {  // 可以运行在多线程
        // Spawn a new CallData instance to serve new clients.
        new RegistCallData(&service_, cq_.get());  //
        new LoginCallData(&service_, cq_.get());
        void* tag;                              // uniquely identifies a
request.
        bool ok;
        while (true) {
            // Block waiting to read the next event from the completion queue.
The
            // event is uniquely identified by its tag, which in this case is
the
            // memory address of a CallData instance.
            // The return value of Next should always be checked. This return
value
            // tells us whether there is any kind of event or cq_ is shutting
down.
            std::cout << "before cq_->Next "
                      << std::endl;  // 1. 等待消息事件  darren 增加
            GPR_ASSERT(cq_->Next(&tag, &ok));
            std::cout << "after cq_->Next " << std::endl;  // darren 增加
            GPR_ASSERT(ok);
            std::cout << "before static_cast" << std::endl;  // darren 增加
            static_cast<CallData*>(tag)->Proceed();
```

```
            std::cout << "after static_cast" << std::endl;  // darren 增加
        }
    }

    std::unique_ptr<ServerCompletionQueue> cq_;
    ImLogin::AsyncService service_;
    std::unique_ptr<Server> server_;
};

int main(int argc, char** argv) {
  ServerImpl server;
  server.Run();

  return 0;
}
```