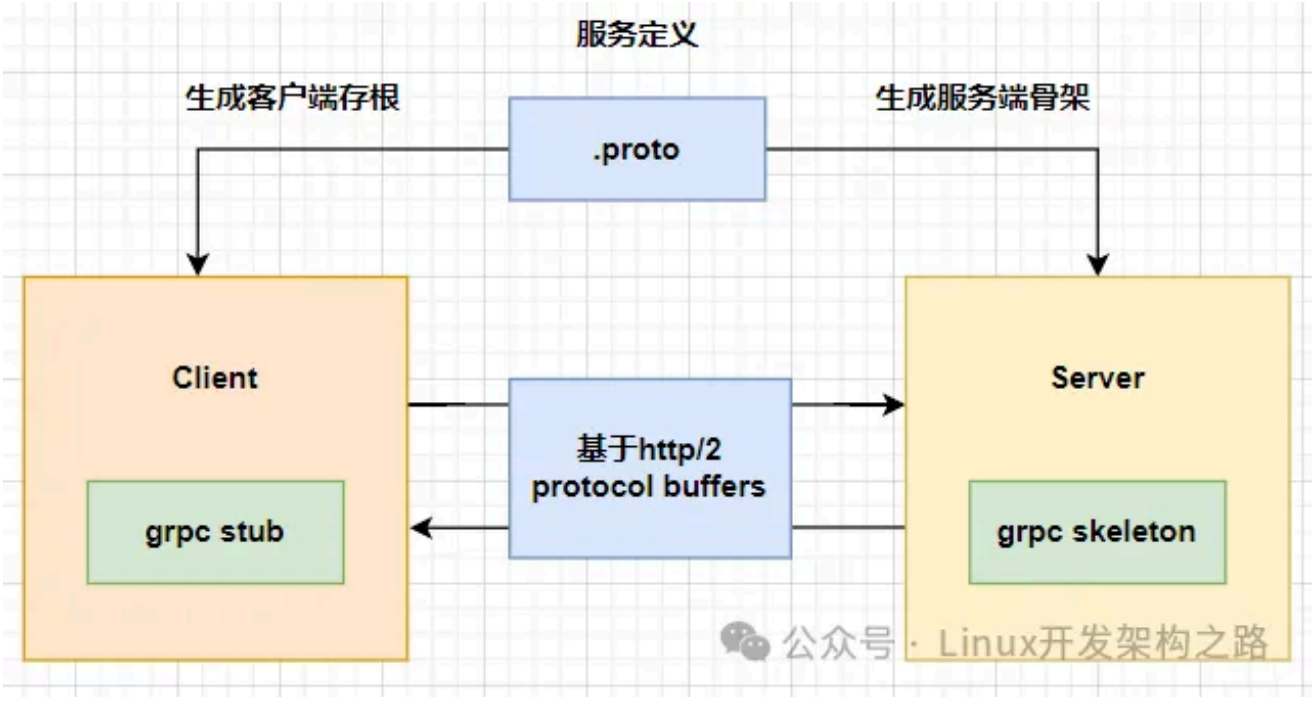
# 深入理解gRPC：C++高性能开源框架

Linux开发架构之路 2024年09月14日 21:09 湖南

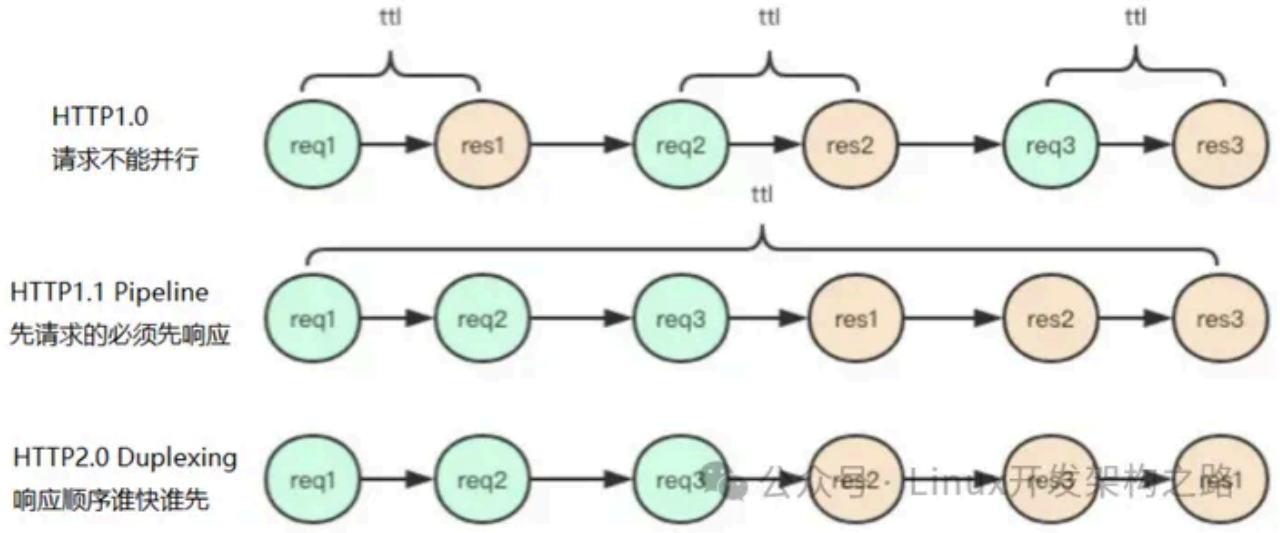
RPC 远程过程调用协议 Remote Procedure Call Protocol，客户端就像调用本地方法一样发起远程调用，用于分布式系统进程间通信。

gRPC 是一个基于 HTTP2 协议设计，语言无关的通用 RPC 框架。借助服务定义，可以生成服务器端骨架（服务器代理）。同时，生成客户端存根（客户端代理）。抽象简化了底层的通信框架，客户端就像调用本地方法那样，远程调用服务接口定义的方法。




附：HTTP 发展

http 1.0
http 1.1：Pipeline，无法分清数据归属，只能串行排队发送请求。
http 2.0：Duplexing，并行发送。每个请求对应一个流，每个请求的数据分为多个帧，数据帧按流 id 分组，分离出不同的请求。




## 1、gRPC 环境搭建

安装 gRPC 1.45.2 版本

安装必要的依赖工具

```
sudo apt-get install autoconf automake libtool
```

### 1.1、安装 cmake

cmake 最低版本 3.15，这里安装 3.23 版本。

```
# 卸载原有的 cmake
sudo apt-get autoremove cmake

# 下载解压 cmake 3.23
wget https://cmake.org/files/v3.23/cmake-3.23.0-linux-x86_64.tar.gz
tar xvzf cmake-3.23.0-linux-x86_64.tar.gz

# 创建软链接
sudo mv cmake-3.23.0-linux-x86_64 /opt/cmake-3.23.0
sudo ln -sf /opt/cmake-3.23.0/bin/*  /usr/bin/

# 测试
cmake -version
```

## 1.2、安装 gcc/gdb

gcc/g++ 版本 6.3，这里安装 7.5

```
# 安装 gcc/g++ 7
sudo apt-get install -y software-properties-common
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt update
sudo apt install g++-7 -y

# 创建软链接
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 60 \
                         --slave /usr/bin/g++ g++ /usr/bin/g++-7
sudo update-alternatives --config gcc

# 测试
gcc -v
g++ -v
```

## 1.3、安装 gRPC

```
# 下载源码
git clone https://github.com/grpc/grpc
# 选择版本 v1.45.2
git tag
git checkout v1.45.2
# 下载第三方依赖
git submodule update --init

# 编译安装: tar -jxvf grpc-v1.45.2.tar.bz2
mkdir -p cmake/build
cd cmake/build
cmake ../..
make
sudo make install
```

## 1.4、protobuf 安装

编译 third_party/protobuf 里面编译安装对应的 protobuf

```
cd third_party/protobuf/
./autogen.sh
./configure --prefix=/usr/local
make

sudo make install
sudo ldconfig  # 使得新安装的动态库能被加载

protoc --version # 3.19.4
```

## 1.5、测试环境

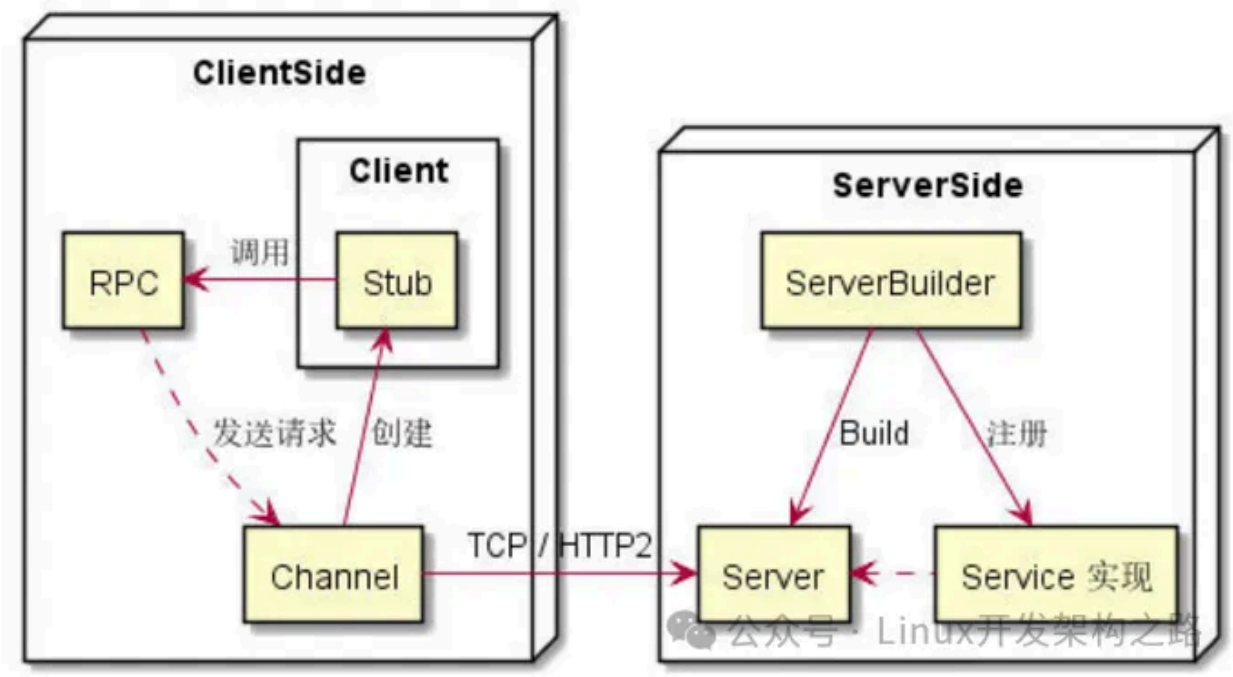
编译 helloworld

```
cd grpc/examples/cpp/helloworld/
mkdir build
cd build/
cmake ..
make登录后复制
```

启动服务和客户端

```
# 启动服务端，监听在50051端口
./greeter_server
Server listening on 0.0.0.0:50051
# 启动客户端，服务端返回Hello world
./greeter_client
Greeter received: Hello world
```

## 2.1、grpc 同步




## 2.1、定义服务

构建 grpc 服务首先要定义服务接口。服务就是可以被远程调用的一组方法。

grpc 使用 pb (protocol buffers) 作为 IDL（接口定义语言，interface definition language），来定义服务接口。pb 是一种语言无关、平台无关、可扩展的结构化数据序列化机制。rpc 服务接口在 .proto 文件中定义，并将 rpc 方法参数和返回类型指定为 pb 消息。可以借助 grpc 插件来根据 pb 文件生成代码。

例：

```
syntax = "proto3";  // 语法
package IM.Login;  // 包名

// 定义服务：远程调用方法，参数 Request，返回值 Reply
// pb 规定只能有一个参数，并只能返回一个值，想传多个，定义消息类型。
service ImLogin {
   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
}

// 注册返回
message  IMRegistRes{
    string user_name = 1;   // 用户名
    uint32 user_id = 2;     // 用户 id
    uint32 result_code = 3; // 返回0，正常注册
}

// rpc 请求
message IMLoginReq{
    string user_name = 1; // 用户名
    string password = 2;  // 密码
}

// rpc 返回
message  IMLoginRes{
    uint32 user_id = 1;
    uint32 result_code = 2; // 返回0的时候注册注册
}
```

生成 C++ 代码

```
# 生成 simple.h 和 simple.cc 文件
protoc -I ./ --cpp_out=. IM.Login.proto

# 生成 simple.grpc.pb.h 和 simple.grpc.pb.cpp 文件，服务框架
protoc -I ./ --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` IM.
protoc --cpp_out=. --grpc_out=. --plugin=protoc-gen-grpc=/usr/local/bin/grpc_c
```

## 2.2、gRPC 服务端

在服务端，需要实现服务定义，实现远程调用方法；并运行 grpc 服务器绑定该服务。具体来说，服务端需要做好两件事：

重载服务：重载服务器基类的远程调用方法，实现 pb 中定义的 rpc。

启动服务：ServerBuilder 工厂类创建并启动 grpc 服务

例：C++ 流程

命名空间：引入 grpc 命名空间和自定义 pb 文件的命名空间
重载服务
启动服务

```
#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>

// 自定义 proto 文件生成的.h
#include "IM.Login.pb.h"
#include "IM.Login.grpc.pb.h"

// 1、命名空间
// grcp 命名空间
using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;
// 自定义 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;

// 2、重写服务
// 1、定义服务端的类：继承 .grpc.pb.h 文件定义的 grpc 服务
// 2、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,
        std::cout << "Regist user_name: " << request->user_name() << std::end
```

```
        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, I
        std::cout << "Login user_name: " << request->user_name() << std::endl
        response->set_user_id(10);
        response->set_result_code(0);
        return Status::OK;
    }

};

// 3、启动 grpc 服务
void RunServer() {
    std::string server_addr("0.0.0.0:50051");

    // 创建一个服务类
    IMLoginServiceImpl service;

    // 创建工厂类
    ServerBuilder builder;

    // 监听端口地址
    builder.AddListeningPort(server_addr, grpc::InsecureServerCredentials());
    // 心跳探活
    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);
    // 多线程：动态调整 epoll 线程数量
    builder.SetSyncServerOption(ServerBuilder::MIN_POLLERS, 4);
    builder.SetSyncServerOption(ServerBuilder::MAX_POLLERS, 8);
    // 注册服务
    builder.RegisterService(&service);

```

```
    // 创建并启动 rpc 服务器
    std::unique_ptr<Server> server(builder.BuildAndStart());
    std::cout << "Server listening on " << server_addr << std::endl;

    // 进入服务事件循环
    server->Wait();
}

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

**需要C/C++ Linux服务器架构师学习资料加qun812855908获取（资料包括C/C++，Linux，golang技术，Nginx，ZeroMQ，MySQL，Redis，fastdfs，MongoDB，ZK，流媒体，CDN，P2P，K8S，Docker，TCP/IP，协程+，DPDK，ffmpeg等），免费分享**


## 2.3、gRPC 客户端

在客户端，由服务定义 pb 生成客户端存根 stub（客户端代理），使用通道 channel 连接特定的 grpc 服务端；stub 在 channel 基础上创建而成，通过 stub 真正调用 rpc 请求。

### 核心代码

```
class ImLoginClient {
public:
    // 使用通道 channel 初始化阻塞式存根 stub
    ImLoginClient(std::shared_ptr<Channel> channel)
    :stub_(ImLogin::NewStub(channel))
    {}

    // 使用阻塞式存根调用远程方法
    void Regist(const std::string &user_name, const std::string &password) {
        // 调用 rpc 接口
        Status status = stub_->Regist(&context, request, &response);
    }

private:
    std::unique_ptr<ImLogin::Stub> stub_;   // 存根，客户端代理
};
```

### 例：C++ 流程

命名空间：引入 grpc 命名空间和自定义 pb 文件的命名空间
定义客户端：实现远程调用的方法。

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

// grpc 头文件
#include <grpcpp/grpcpp.h>

// 自定义 proto 文件生成的.h
#include "IM.Login.pb.h"
#include "IM.Login.grpc.pb.h"

// 命名空间
// grcp 命名空间
using grpc::Channel;
```

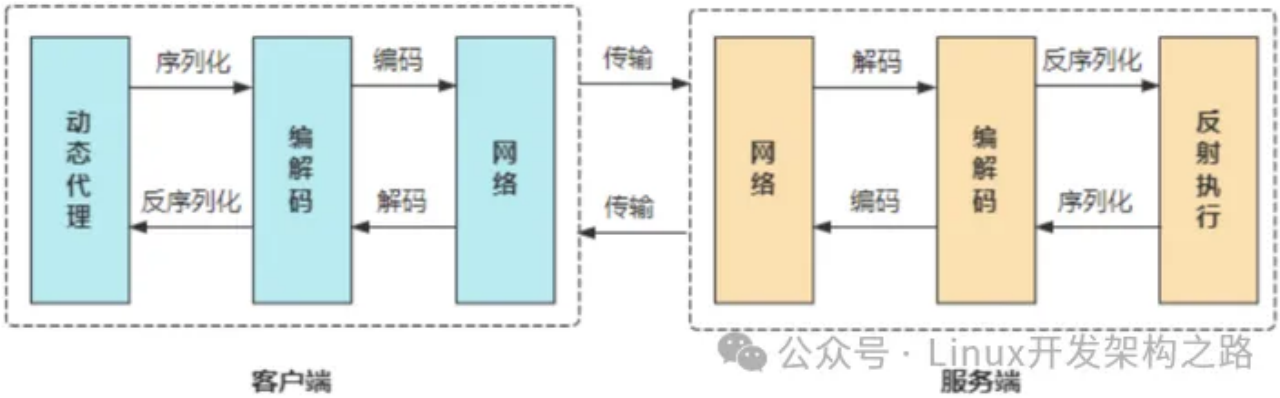
```
using grpc::ClientContext;
using grpc::Status;
// 自定义 proto 文件的命名空间
using IM::Login::ImLogin;
using IM::Login::IMRegistReq;
using IM::Login::IMRegistRes;
using IM::Login::IMLoginReq;
using IM::Login::IMLoginRes;


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;
        // 调用 rpc 接口
        Status status = stub_->Regist(&context, request, &response);
        if(status.ok()) {
            std::cout <<  "user_name:" << response.user_name() << ", user_id:
        }
        else {
            std::cout <<  "user_name:" << response.user_name() << "Regist fai
        }
    }

     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;
        // 调用 rpc 接口
        Status status = stub_->Login(&context, request, &response);
        if(status.ok()) {
            std::cout <<  "user_id:" << response.user_id() << " login ok" <<
        }
        else {
            std::cout <<  "user_name:" << request.user_name() << "Login faile
        }
    }

private:
    std::unique_ptr<ImLogin::Stub> stub_;   // 存根，客户端代理
};

int main()  {
    // 服务器的地址
    std::string server_addr = "localhost:50051";

    // 创建请求通道
    ImLoginClient im_login_client(
        grpc::CreateChannel(server_addr, grpc::InsecureChannelCredentials())
    );

    // 测试
    std::string user_name = "Jim Hacker";
    std::string password = "123456";
    im_login_client.Regist(user_name, password);
    im_login_client.Login(user_name, password);

    return 0;
}
```

## 2.4、消息流

当调用 grpc 服务时，客户端的 grpc 库会使用 pb，并将 rpc 的请求编排 marshal 为 pb 格式，然后将其通过 HTTP/2 进行发送。在服务器端，请求会解排 unmarshal，对

应的过程调用会使用 pb 来执行。




## 3、gRPC stream

grpc 根据消息的数量，将通信模式分为以下四种：

一元 RPC 模式：简单 RPC 模式，请求-响应式 RPC（1请求-1返回）
服务端流 RPC 模式：客户端发送一个请求，服务端回发响应序列（流）
客户端流 RPC 模式：客户端发送请求序列（流），服务端回发一个响应
双向流 RPC 模式：客户端发送请求流，服务器端回发响应流

以官方范例 examples/cpp/route_guide/ 为例：pb 定义的服务如下，stream 关键字来定义流

```
service RouteGuide {
  // A simple RPC.
  rpc GetFeature(Point) returns (Feature) {}
  // A server-to-client streaming RPC.
  rpc ListFeatures(Rectangle) returns (stream Feature) {}
  // A client-to-server streaming RPC.
  rpc RecordRoute(stream Point) returns (RouteSummary) {}
  // A Bidirectional streaming RPC.
  rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
}
```

### 3.1、服务端：RPC 实现

服务端需要实现 pb 中定义的 rpc，每种 rpc 的实现都需要 ServerContext 参数。

其他参数则与 grpc 通信模式有关。

非流模式：Request 请求，Reply 响应。

```
// rpc ListFeatures(Rectangle) returns (stream Feature) {}
Status ListFeatures(ServerContext* context, const routeguide::Rectangle* recta
```

流模式：单向流

ServerReader：读 client 流，通过 `Reader->Read()` 返回的 bool 型状态，判断流的结束。

```
// rpc RecordRoute(stream Point) returns (RouteSummary) {}
Status RecordRoute(ServerContext* context, ServerReader<Point>* reader, RouteS
    // 读取请求
    while (reader->Read(&point)) {
        ...
    }
}
```

ServerWriter：写 server 流，通过结束 rpc 函数并返回状态码的方式结束流

```
// rpc ListFeatures(Rectangle) returns (stream Feature) {}
Status ListFeatures(ServerContext* context, const routeguide::Rectangle* recta
    // 发送响应
    writer->Write(f);
    ...
}
```

流模式：双向流

ServerReaderWriter：只需要 1 个参数

```
// rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
```

```
// 注意线程同步
Status RouteChat(ServerContext* context, ServerReaderWriter<RouteNote, RouteNo
    // 读取数据
    while (stream->Read(&note)) {
        // 写回数据
        stream->Write(n);
    }
}
```

### 3.2、客户端：RPC 调用

客户端均需要传入 ClientContext 参数。

其他参数则与 grpc 通信模式有关。

非流模式：Request 请求，Reply 响应。

```
// rpc GetFeature(Point) returns (Feature) {}
Status GetFeature(ClientContext* context, const Point& request, Feature* respo
```

流模式：单向流

ClientReader：读 server 流，通过 `Reader->Read()` 返回的 bool 型状态，判断流的结束。

```
// rpc ListFeatures(Rectangle) returns (stream Feature) {}
unique_ptr<ClientReader<Feature>> ListFeatures(ClientContext* context, const
    // 创建 reader，读取响应
    // 参数：rpc 的 Context, Request
    std::unique_ptr<ClientReader<Feature> > reader(stub_->ListFeatures(&conte
    // 读取响应
    while (reader->Read(&feature)) {
    ...
    }
    // 等待返回状态
    Status status = reader->Finish();
```

```
    ...
}
```

ClientWriter：写 client 流，流的结束

writer->WritesDone() ： 发送结束
writer->Finish() ： 等待对端返回状态

```
// rpc RecordRoute(stream Point) returns (RouteSummary) {}
void RecordRoute() {
    // 创建 writer
  std::unique_ptr<ClientWriter<Point> > writer(stub_->RecordRoute(&context, &
    // 发送请求
  writer->Write(f.location()))
    // 发送结束
    writer->WritesDone();
    // 等待返回状态
    Status status = writer->Finish();
}
```

流模式：双向流

ClientReaderWriter：对于 rpc 调用，都是 client 请求后 server 响应，即双向流需要 client 先发送完数据，server 才能结束 rpc。流的结束

stream->WriteDone()
stream->Finish()

```
// rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
// client 需要开启发送线程和接收线程
void RouteChat() {
    // 创建 readerwriter，读取写入都是它
    std::shared_ptr<ClientReaderWriter<RouteNote, RouteNote> > stream(
        stub_->RouteChat(&context));

  // 子线程发送请求
    std::thread writer([stream]() {
```

```
        // 发送请求
        stream->Write(note);
        // 发送结束
        stream->WritesDone();
    });
    ...

    // 主线程读取响应
    // 读取响应
    while (stream->Read(&server_note)) {
    }
    writer.join();
    // 等待返回状态
    Status status = stream->Finish();
  ...
}
```

### 3.3、流的结束

这里，总结流的结束方式：

Client 发送流：通过 `Writer->WritesDone()` 结束流
Server 发送流：通过结束 rpc 调用并返回状态码 `status code` 的方式来结束流
读取流：通过 `Reader->Read()` 返回的 bool 型状态，来判断流是否结束

## 4、gRPC 异步

官方文档：Asynchronous-API tutorial

grpc 通过完成队列 CompletionQueue 来进行异步操作，其通用流程为：

绑定完成队列 cq 到 rpc 请求
`void* Tag` 唯一标识请求该 rpc 请求
调用 `cq->Next()` 阻塞读取 cq 队列中的下个 rpc 请求

### 4.1、异步 server

异步 server 的逻辑

创建 CallData 类实例，记录一个 rpc 事件的逻辑和状态。将其加入 cq 队列，并通过将 CallData 实例 this 指针作为 tag 唯一标识该 CallData 实例。
在服务器事件循环中，异步处理 rpc 事件。事件到来时，从 cq 队列取出事件 `cq->Next()`，处理事件 `CallData->Proceed()`，处理后等待对端返回结果 `responder_.Finish` （类型： `ServerAsyncResponseWriter` ）

创建 CallData 类：实现 rpc 请求的逻辑和状态。每个 rpc 请求对应一个 CallData 实例。若要实现不同类型的 rpc 请求，可以构造对应的 CallData 子类，子类继承基类 CallData 的通用部分，并实现自己的差异化部分。

例如：文章第 1 部分的案例

```
class ServerImpl final {
    // 实现 rpc 请求的逻辑和状态
    class CallData {
        public:
        // 创建 CallData 类，
        // 1、绑定 cq 队列到 rpc 调用
        CallData(ImLogin::AsyncService* service, ServerCompletionQueue* cq)
            : service_(service), cq_(cq), status_(CREATE) {
                Proceed();  // 业务逻辑处理
        }

        virtual ~CallData(){}
```

```
        // 虚函数：业务逻辑接口
        virtual void Proceed() {}

        // 基类部分
        // rpc 提供的异步服务
        ImLogin::AsyncService* service_;
        // 完成队列
        ServerCompletionQueue* cq_;
        // rpc 上下文
        ServerContext ctx_;
        // 状态机：描述业务逻辑处理时的状态
        enum CallStatus { CREATE, PROCESS, FINISH };
        // 当前 rpc 服务的状态
        CallStatus status_;
    };

    // rpc：注册服务
    class RegistCallData : public CallData {
      ...
        // 实现注册 rpc 服务的业务逻辑过程处理
        void Proceed() override {...}

        // 子类成员
        IMRegistReq request_;
        IMRegistRes reply_;
        ServerAsyncResponseWriter<IMRegistRes> responder_;
    };

  // rpc：登录服务
    class LoginCallData : public CallData {
        ...
        void Proceed() override {...}

        IMLoginReq request_;
        IMLoginRes reply_;
        ServerAsyncResponseWriter<IMLoginRes> responder_;
    };
  ...
```

```
};
```

以官方范例 examples/cpp/helloworld 为例，完整代码如下：

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

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

#include "examples/protos/helloworld.grpc.pb.h"

using grpc::Server;
using grpc::ServerAsyncResponseWriter;
using grpc::ServerBuilder;
using grpc::ServerCompletionQueue;
using grpc::ServerContext;
using grpc::Status;
using helloworld::Greeter;
using helloworld::HelloReply;
using helloworld::HelloRequest;

class ServerImpl final {
    public:
    ~ServerImpl() {
        server_->Shutdown();
        cq_->Shutdown();
    }

    void Run() {
        std::string server_address("0.0.0.0:50051");
        // 创建工厂类
        ServerBuilder builder;
        // 监听端口地址，不验证
        builder.AddListeningPort(server_address, grpc::InsecureServerCredent
        // 注册服务
        builder.RegisterService(&service_);
```

```
        // 创建完成队列 cq：把要监听的 rpc 对象放入到队列
        cq_ = builder.AddCompletionQueue();
        // 启动服务
        server_ = builder.BuildAndStart();
        std::cout << "Server listening on " << server_address << std::endl;

        // 启动服务器事件循环：处理 rpc 请求
        HandleRpcs();
    }

    private:
    // 实现 rpc 请求的逻辑和状态
    class CallData {
        public:
        // 创建 CallData 类
        // 1、绑定 cq 队列到 rpc 调用
        CallData(Greeter::AsyncService* service, ServerCompletionQueue* cq)
            : service_(service), cq_(cq), responder_(&ctx_), status_(CREATE)
                // 调用业务逻辑处理
                Proceed();
            }

        // 业务逻辑过程处理函数：状态机
        void Proceed() {
            // 创建状态：把 CallData 实例放入 cq 队列后进入该状态
            if (status_ == CREATE) {
                // 该 CallData 实例状态推进到 PROCESS
                status_ = PROCESS;

                // 处理 rpc 请求：CallData 实例的 this 指针作为唯一标识该 rpc 请
                service_->RequestSayHello(&ctx_, &request_, &responder_, cq_
            }
            // 处理状态
            else if (status_ == PROCESS) {
                // 创建一个新的 calldata 实例，用于处理新的 rpc 请求
                new CallData(service_, cq_);

                // 业务逻辑处理
                std::string prefix("Hello ");
                reply_.set_message(prefix + request_.name());
```

```
                // 业务逻辑处理结束
                // 该 calldata 实例状态推进到 FINISH，并将会在 FINISH 状态中释放
                status_ = FINISH;
                // 2、等待对端返回状态：this 指针作为 tag 唯一标识 calldata 实例
                responder_.Finish(reply_, Status::OK, this);
            }
            else {
                GPR_ASSERT(status_ == FINISH);
                // 释放 calldata 内存，即本次 rpc 请求的资源
                delete this;
            }
        }

        private:
        // rpc 提供的异步服务
        Greeter::AsyncService* service_;
        // 完成队列
        ServerCompletionQueue* cq_;
        // rpc 上下文
        ServerContext ctx_;

        // What we get from the client.
        HelloRequest request_;
        // What we send back to the client.
        HelloReply reply_;

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

        // 状态机：描述业务逻辑处理时的状态
        enum CallStatus { CREATE, PROCESS, FINISH };
        // 当前 rpc 服务的状态
        CallStatus status_;
    };

    // 服务器事件循环：处理 rpc 请求，可运行在多线程
    void HandleRpcs() {
        // 创建 calldata 类维护 rpc 请求的逻辑和状态
        new CallData(&service_, cq_.get());
```

```
        // 每个 calldata 请求的唯一标识，指向上面 new calldata 类的地址
        void* tag;
        bool ok;
        while (true) {

            // 3、阻塞读取 cq 队列中的下个 rpc 请求
            // 通过返回值判断是否有请求到来还是 cq 队列正在关闭
            GPR_ASSERT(cq_->Next(&tag, &ok));
            GPR_ASSERT(ok);
            // 处理业务，可以自定义 proceed
            // 改进：扔给线程池去做异步处理
            static_cast<CallData*>(tag)->Proceed();
        }
    }

    // 完成队列
    std::unique_ptr<ServerCompletionQueue> cq_;
    // rpc 异步服务
    Greeter::AsyncService service_;
    // rpc 服务器
    std::unique_ptr<Server> server_;
};

int main(int argc, char** argv) {
    ServerImpl server;
    server.Run();
    return 0;
}
```

## 4.2、异步 client

异步 client 的逻辑

绑定 CompletionQueue 到 rpc 请求。
调用 rpc_.Finish 等待对端返回状态
调用 cq->Next() 阻塞读取 cq 队列中的下个 rpc 事件

以官方范例 examples/cpp/helloworld 为例，完整代码如下

```cpp
#include <iostream>
#include <memory>
#include <string>

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

#include "examples/protos/helloworld.grpc.pb.h"

using grpc::Channel;
using grpc::ClientAsyncResponseReader;
using grpc::ClientContext;
using grpc::CompletionQueue;
using grpc::Status;
using helloworld::Greeter;
using helloworld::HelloReply;
using helloworld::HelloRequest;

class GreeterClient {
 public:
  explicit GreeterClient(std::shared_ptr<Channel> channel)
      : stub_(Greeter::NewStub(channel)) {}

  std::string SayHello(const std::string& user) {
    HelloRequest request;
    request.set_name(user);

    HelloReply reply;
    ClientContext context;
    CompletionQueue cq;
    Status status;

    // 1、绑定 cq 到 rpc 请求
    std::unique_ptr<ClientAsyncResponseReader<HelloReply> > rpc(
        stub_->PrepareAsyncSayHello(&context, request, &cq));

    // 初始化 rpc 调用
    rpc->StartCall();

```

```
    // 2、等待对端返回状态
    rpc->Finish(&reply, &status, (void*)1);

    // 3、阻塞读取 cq 队列中的下个 rpc 事件
    void* got_tag;
    bool ok = false;
    GPR_ASSERT(cq.Next(&got_tag, &ok));
    GPR_ASSERT(got_tag == (void*)1);
    GPR_ASSERT(ok);

    if (status.ok()) {
      return reply.message();
    } else {
      return "RPC failed";
    }
  }

 private:

  std::unique_ptr<Greeter::Stub> stub_;
};

int main(int argc, char** argv) {
  GreeterClient greeter(grpc::CreateChannel( "localhost:50051", grpc::Insecur
  std::string user("world");
  std::string reply = greeter.SayHello(user);
  std::cout << "Greeter received: " << reply << std::endl;

  return 0;
}
```

## 往期精彩回顾

- **linux内核开发是程序员的新风口？那linux内核该如何学习？**
- **DPDK入门指南，DPDK学习路线总结**
- **存储开发怎么样？如何学习C++存储开发技术？**
- **你离腾讯T8还有多远？最全最详细的C/C++后端开发技术栈**
- **音视频/流媒体开发工程师工作内容主要是在做什么**