# gRPC 编程指南

## gRPC 介绍

gRPC 是谷歌开源的高性能 RPC 框架。RPC 也即远程方法调用，对于 RPC client 来说，它可以调用远程 server 上的某个方法，看起来就像是在调用本地方法一样。区别就在于，通过 RPC 调用远程方法时，数据经过序列化之后会通过网络发送给远程 server，远程 server 执行方法之后，同样会将返回结果序列化之后发送回 client。在分布式系统中，gRPC 可以用来解耦程序的逻辑，不同组件之间通过 gRPC 进行通信。  
　　gRPC 使用 Protobuf 作为它的数据序列化的工具，Protobuf 会将数据序列化成二进制的数据流。与 JSON 这类文本形式的数据相比，二进制数据显得更加紧凑和便于解析，在网络传输中，二进制数据由于体积更小，传输也更快。另一方面，gRPC 也是跨多种编程语言的，譬如说，一个 Java 的 client 可以与一个 C++ 的 server 通信。

## 在 Linux 安装 gRPC

　　在 Ubuntu 16.04 中，通过下面的步骤就可以安装好 gRPC 和 Protobuf。

|  |
| --- |
| sudo apt-get install build-essential autoconf libtool libgflags-dev libgtest-dev clang libc++-dev pkg-config unzip  git clone -b $(curl -L http://grpc.io/release) https://github.com/grpc/grpc  cd grpc  git submodule update --init  make  sudo make install  cd third\_party/protobuf  sudo ./autogen.sh  sudo ./configure  make  sudo make install |

## 构建服务端程序

　　在创建 gRPC 服务(service)之前，首先需要提供这个服务的接口。Protobuf 除了作为数据序列化工具之外，还可以用来为服务定义接口。例如，下面我们为 Company 服务定义接口：

|  |
| --- |
| syntax = "proto3";  package company;  service Company {  rpc AddEmployee(Employee) returns (EmployeeID) {} // 提交员工信息  rpc ListEmployees(AgeRange) returns (stream Employee) {} // 查询员工信息  }  message Employee {  string name = 1;  int32 age = 2;  }  message EmployeeID {  int32 id = 1;  }  message AgeRange {  int32 low = 1;  int32 high = 2;  } |

　　我们为 Company 服务定义了两个方法，AddEmployee()用于提交员工信息，而ListEmployees()则用于根据年龄查询员工信息。注意到ListEmployees()方法的返回值类型是stream Employee，这表示这个方法会返回多个Employee消息。  
　　执行下面的命令可以自动生成 ProtoBuf 编解码的代码，以及与 Company 服务相关的代码：

|  |
| --- |
| $ tree  ├── cpp  └── protos  └── company.proto  $ protoc -I protos --grpc\_out=cpp --plugin=protoc-gen-grpc=`which grpc\_cpp\_plugin` protos/company.proto  $ protoc -I protos --cpp\_out=cpp protos/company.proto  $ tree  ├── cpp  │   ├── company.grpc.pb.cc  │   ├── company.grpc.pb.h  │   ├── company.pb.cc  │   └── company.pb.h  └── protos  └── company.proto |

　　在company.grpc.ph.h文件里面，已经定义好了Company::Service这个抽象基类，我们可以继承这个基类，并提供方法的具体实现。下面我们创建一个company\_server.cc文件，并定义CompanyImpl这个具体类，同时提供方法的具体实现。值得注意的是，我们需要保证CompanyImpl提供的方法都是线程安全的，因为这些方法允许被多个 client 同时调用。

|  |
| --- |
| // file company\_server.cc  #include "company.grpc.pb.h"  #include <iostream>  #include <unordered\_map>  #include <mutex>  #include <grpc/grpc.h>  #include <grpc++/server.h>  #include <grpc++/server\_builder.h>  #include <grpc++/server\_context.h>  #include <grpc++/security/server\_credentials.h>  using company::Company;  using company::AgeRange;  using company::Employee;  using company::EmployeeID;  class CompanyImpl final : public Company::Service  {  public:  CompanyImpl()  : nextID\_(0) { }  grpc::Status AddEmployee(grpc::ServerContext \*context, const Employee \*request,  EmployeeID \*response) override  {  std::lock\_guard<std::mutex> guard(mtx\_);  employees\_[nextID\_] = \*request;  response->set\_id(nextID\_);  ++nextID\_;  return grpc::Status::OK;  }  grpc::Status ListEmployees(grpc::ServerContext \*context, const AgeRange \*request,  grpc::ServerWriter<Employee> \*writer) override  {  auto low = request->low();  auto high = request->high();  std::lock\_guard<std::mutex> guard(mtx\_);  for (auto &entry : employees\_)  {  auto employee = entry.second;  if (employee.age() >= low && employee.age() <= high)  {  writer->Write(employee); // 调用 Write 写入一个 Employee 消息给client  }  }  return grpc::Status::OK;  }  private:  int32\_t nextID\_;  std::mutex mtx\_;  std::unordered\_map<int32\_t, Employee> employees\_;  };  int main(int argc, char \*argv[])  {  std::string addr = "0.0.0.0:5000";  CompanyImpl service;  grpc::ServerBuilder builder;  builder.AddListeningPort(addr, grpc::InsecureServerCredentials());  builder.RegisterService(&service);  auto server = builder.BuildAndStart();  std::cout << "Server listening on " << addr << std::endl;  server->Wait();  return 0;  } |

## 构建客户端程序

　　client 的代码相对简单很多，为了让 client 可以调用 server 提供的方法，首先需要创建一个 stub：

|  |
| --- |
| // 第二个参数表示不开启 SSL  auto channel = grpc::CreateChannel("localhost:5000", grpc::InsecureChannelCredentials());  auto stub = Company::NewStub(channel); |

　　client 通过这个 stub 就可以直接调用 server 提供的方法了。下面我们创建一个company\_client.cc文件，用来对 server 进行测试：

|  |
| --- |
| #include "company.grpc.pb.h"  #include <iostream>  #include <memory>  #include <string>  #include <grpc/grpc.h>  #include <grpc++/channel.h>  #include <grpc++/client\_context.h>  #include <grpc++/create\_channel.h>  #include <grpc++/security/credentials.h>  using company::Company;  using company::Employee;  using company::EmployeeID;  using company::AgeRange;  class CompanyClient  {  public:  CompanyClient(std::shared\_ptr<grpc::Channel> channel)  : stub\_(Company::NewStub(channel)) { }  void AddEmployee(const std::string &name, int32\_t age)  {  Employee employee;  employee.set\_name(name);  employee.set\_age(age);  EmployeeID id;  grpc::ClientContext context;  stub\_->AddEmployee(&context, employee, &id);  std::cout << "AddEmployee() - new id: " << id.id() << std::endl;  }  void ListEmployeesByAge(int32\_t low, int32\_t high)  {  AgeRange range;  range.set\_low(low);  range.set\_high(high);  grpc::ClientContext context;  auto reader = stub\_->ListEmployees(&context, range);  Employee employee;  while (reader->Read(&employee))  {  std::cout << "Employee: name = " << employee.name() << ", age = " << employee.age() << std::endl;  }  }  private:  std::unique\_ptr<Company::Stub> stub\_;  };  int main(int argc, char \*argv[])  {  auto channel = grpc::CreateChannel("localhost:5000", grpc::InsecureChannelCredentials());  CompanyClient client(channel);  client.AddEmployee("hello", 10);  client.AddEmployee("world", 20);  client.ListEmployeesByAge(0, 100);  return 0;  } |

　　编译好 server 和 client 程序之后就可以运行了：

|  |  |
| --- | --- |
| 1  2 | $ clang++ -std=c++11 -o server -lgrpc++ -lprotobuf -lpthread -lgrpc++\_reflection company.pb.cc company.grpc.pb.cc company\_server.cc  $ clang++ -std=c++11 -o client -lgrpc++ -lprotobuf -lpthread -lgrpc++\_reflection company.pb.cc company.grpc.pb.cc company\_client.cc |