

RPC远程调用

重点内容

1. 单机要求

- 需要事先约定调用的语义(接口语法)
- 需要网络传输
- 需要约定网络传输中的内容格式

2. 分布式要求

- 注册发现服务
- 负载均衡

3. gRPC实践

- 同步调用
- 异步调用

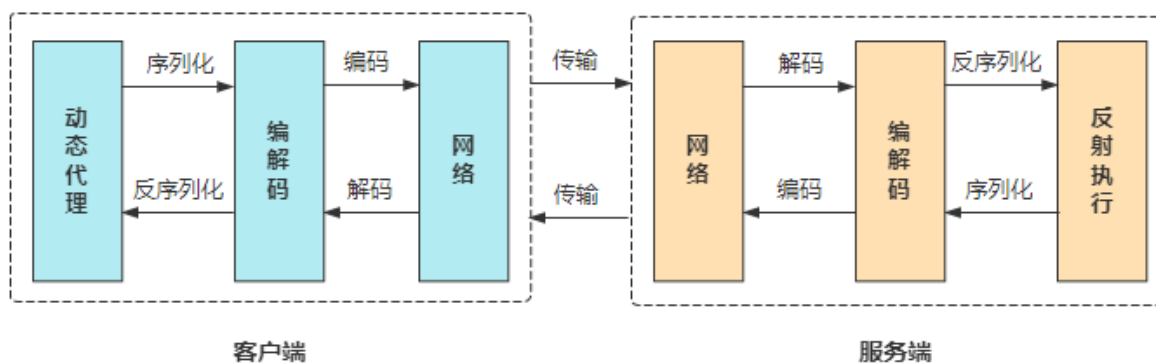
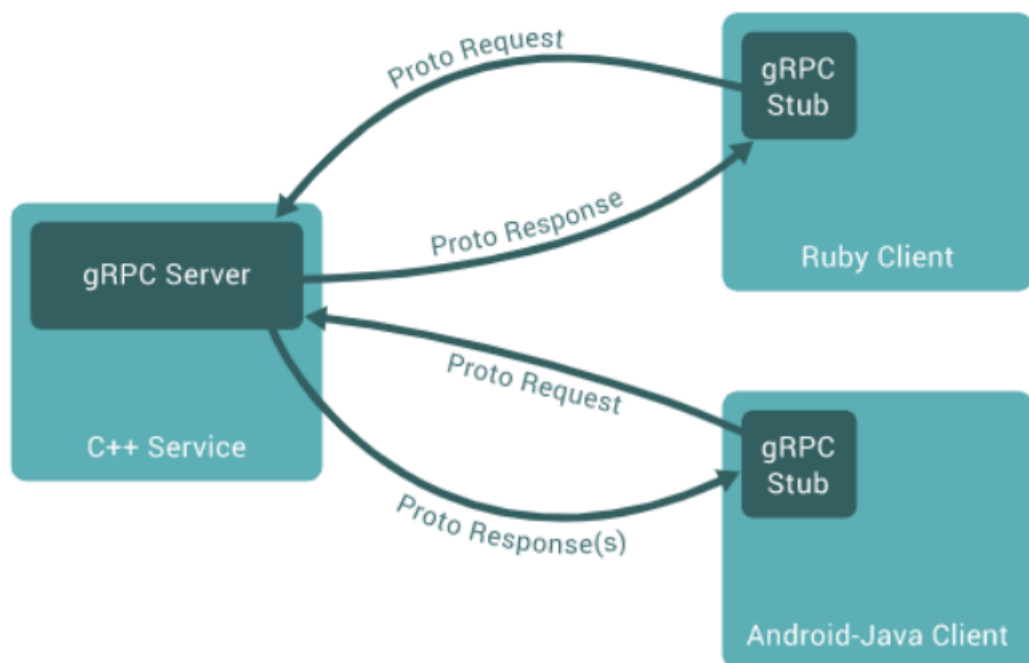
1. gRPC原理

grpc: <https://grpc.io/docs/what-is-grpc/faq/>

1.1 什么是RPC

RPC 即远程过程调用协议 (Remote Procedure Call Protocol) , 可以让我们像调用本地对象一样发起远程调用。RPC 凭借其强大的治理功能, 成为解决分布式系统通信问题的一大利器。

gRPC是一个现代的、高性能、开源的和语言无关的通用 RPC 框架, 基于 HTTP2 协议设计, 序列化使用 PB(Protocol Buffer), PB 是一种语言无关的高性能序列化框架, 基于 HTTP2+PB 保证了的高性能。



1.2 gRPC的一些特性

1. gRPC基于服务的思想：定义一个服务，描述这个方法以及入参出参，服务器端有这个服务的具体实现，客户端保有一个存根，提供与服务端相同的服务
2. gRPC默认采用protocol buffer作为IDL(Interface Description Lanage)接口描述语言,服务之间通信的数据序列化和反序列化也是基于protocol buffer的，因为protocol buffer的特殊性，所以gRPC框架是跨语言的通信框架(与编程语言无关性)，也就是说用Java开发的基于gRPC的服务，可以用GoLang编程语言调用
3. gRPC同时支持同步调用和异步调用，同步RPC调用时会一直阻塞直到服务端处理完成返回结果，异步RPC是客户端调用服务端时不等待服务端处理完成返回，而是服务端处理完成后主动回调客户端告诉客户端处理完成
4. gRPC是基于http2协议实现的，http2协议提供了很多新的特性，并且在性能上也比http1提搞了许多，所以gRPC的性能是非常好的
5. gRPC并没有直接实现负载均衡和服务发现的功能，但是已经提供了自己的设计思路。已经为命名解析和负载均衡提供了接口
6. 基于http2协议的特性：gRPC允许定义如下四类服务方法
 1. 一元RPC：客户端发送一次请求，等待服务端响应结构，会话结束，就像一次普通的函数调用这样简单
 2. 服务端流式RPC：客户端发起一起请求，服务端会返回一个流，客户端会从流中读取一系列消息，直到没有结果为止
 3. 客户端流式RPC：客户端提供一个数据流并写入消息发给服务端，一旦客户端发送完毕，就等待服务器读取这些消息并返回应答

4. 双向流式RPC：客户端和服务端都有一个数据流，都可以通过各自的流进行读写数据，这两个流是相互独立的，客户端和服务端都可以按其希望的任意顺序独写

1.3 gRPC支持的编程语言

C++, Java(包括对Android的支持), Objective-C(对于iOS), Python, Ruby, Go, C#, Node.js都在GA中, 并遵循语义版本控制。

1.4 gRPC的使用场景

低延迟, 高度可扩展的分布式系统

开发与云服务器通信的客户端

设计一个准确, 高效, 且与语言无关的新协议时

分层设计, 以实现扩展, 例如。身份验证, 负载均衡, 日志记录和监控等

1.5 谁在使用gRPC

谷歌长期以来一直在gRPC中使用很多基础技术和概念。目前正在谷歌的几个云产品和谷歌面向外部的API中使用。Square, Netflix, CoreOS, Docker, CockroachDB, Cisco, Juniper Networks以及许多其他组织和个人也在使用它。

1.6 gRPC设计之初的动机和原则

1. **自由, 开放**: 让所有人, 所有平台都能使用, 其实就是开源, 跨平台, 跨语言
2. **协议可插拔**: 不同的服务可能需要使用不同的消息通信类型和编码机制, 例如, JSON、XML和Thrift, 所以协议应允许可插拔机制, 还有负载均衡, 服务发现, 日志, 监控等都支持可插拔机制
3. **阻塞和非阻塞**: 支持客户端和服务端交换的消息序列的异步和同步处理。这对于在某些平台上扩展和处理至关重要
4. **取消和超时**: 一次RPC操作可能是持久并且昂贵的, 应该允许客户端设置取消RPC通信和对这次通信加上一个超时时间
5. **拒绝**: 必须允许服务器通过在继续处理请求的同时拒绝新请求的到来并优雅地关闭。
6. **流处理**: 存储系统依靠流和流控制来表达大型数据集, 其他服务, 如语音到文本或股票行情, 依赖于流来表示与时间相关的消息序列
7. **流控制**: 计算能力和网络容量在客户端和服务端之间通常是不平衡的。流控制允许更好的缓冲区管理, 以及过度活跃的对等体提供对DOS的保护。
8. **元数据交换**: 认证或跟踪等常见的跨领域问题依赖于不属于服务声明接口的数据交换。依赖于他们将这些特性演进到服务, 暴露API来提供能力。
9. **标准化状态码**: 客户端通常以有限的方式响应API调用返回的错误。应约束状态码名称空间, 以使这些错误处理决策更加清晰。如果需要更丰富的特定领域的状态, 则可以使用元数据交换机制来提供该状态。
10. **互通性**: 报文协议(Wire Protocol)必须遵循普通互联网基础框架

信息来源 <https://grpc.io/faq/>

2 数据封装和数据传输问题

2.1 网络传输中的内容封装数据体积问题

早期的RPCJSON的方式，目前的RPC基本上都采用类似Protobuf的二进制序列化方式。

其差别在于：**json的设计是给人看的，protobuf则是利于机器。**

JSON

优点：在body中用JSON对内容进行编码，极易跨语言，不需要约定特定的复杂编码格式和Stub文件。在版本兼容性上非常友好，扩展也很容易。

缺点：JSON难以表达复杂的参数类型，如结构体等；数据冗余和低压缩率使得传输性能差。

Protobuf

gRPC对此的解决方案是丢弃json、xml这种传统策略，使用 Protocol Buffer，是Google开发的一种跨语言、跨平台、可扩展的用于序列化数据协议。

```
// xxxx.proto
service Test {
    rpc HowRpcDefine (Request) returns (Response) ; // 定义一个RPC方法
}
message Request {
    //类型 | 字段名字 | 标号
    int64    user_id  = 1;
    string   name     = 2;
}
message Response {
    repeated int64 ids = 1; // repeated 表示数组
    Value info = 2;      // 可嵌套对象
    map<int, Value> values = 3; // 可输出map映射
}
message Value {
    bool is_man = 1;
    int age = 2;
}
```

以上是一个使用样例，包含方法定义、入参、出参。可以看出有几个明确的特点：

- 有明确的类型，支持的类型有多种
- 每个field会有名字
- 每个field有一个**数字标号**，一般按顺序排列(下文编解码会用到这个点)
- 能表达数组、map映射等类型
- 通过嵌套message可以表达复杂的对象
- 方法、参数的定义落到一个.proto 文件中，**依赖双方需要同时持有这个文件，并依此进行编解码**

protobuf作为一个以跨语言为目标的序列化方案，protobuf能做到多种语言以同一份proto文件作为约定，不用A语言写一份，B语言写一份，各个依赖的服务将proto文件原样拷贝一份即可。

但.proto文件并不是代码，不能执行，要想直接跨语言是不行的，必须得有对应语言的中间代码才行，中间代码要有以下能力：

- 将message转成对象，例如C++里是class，golang里是struct，需要各自表达后，才能被理解
- 需要有进行编解码的代码，能解码内容为自己语言的对象、能将对象编码为对应的数据

更多protobuf的讲解请参考 课程《应用层协议设计protobuf》。

2.2 网络传输效率问题

grpc采用HTTP2.0，相对于HTTP1.0 在 更快的传输 和 更低的成本 两个目标上做了改进。有以下几个基本点：

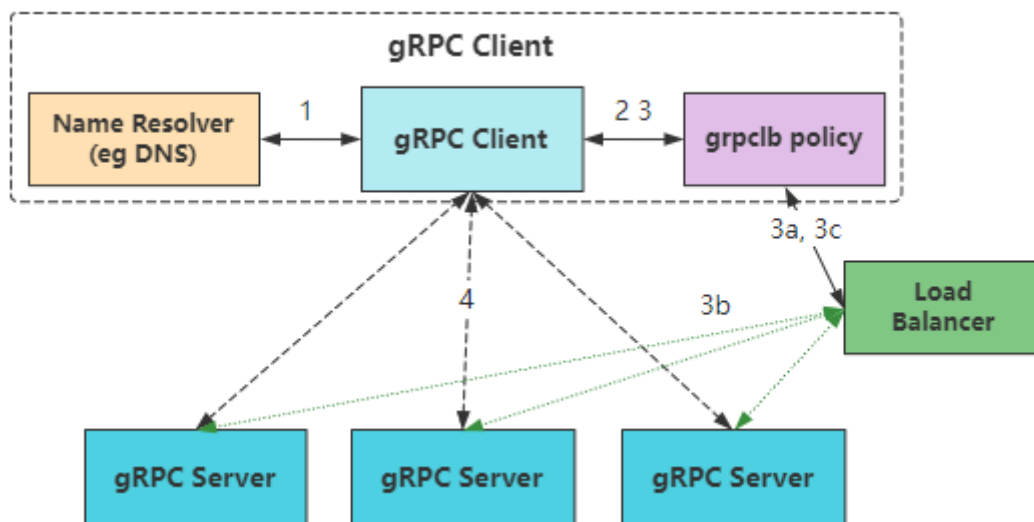
- HTTP2 未改变HTTP的语义(如GET/POST等)，只是在传输上做了优化
- 引入帧、流的概念，在TCP连接中，可以区分出多个request/response
- 一个域名只会有一个TCP连接，借助帧、流可以实现多路复用，降低资源消耗
- 引入二进制编码，降低header带来的空间占用

HTTP1.0核心问题在于：在同一个TCP连接中，没办法区分response是属于哪个请求，一旦多个请求返回的文本内容混在一起，则没法区分数据归属于哪个请求，所以请求只能一个个串行排队发送。这直接导致了TCP资源的闲置。

HTTP2为了解决这个问题，提出了 流 的概念，每一次请求对应一个流，有一个唯一ID，用来区分不同的请求。基于流的概念，进一步提出了 帧，一个请求的数据会被分成多个帧，方便进行数据分割传输，每个帧都唯一属于某一个流ID，将帧按照流ID进行分组，即可分离出不同的请求。这样同一个TCP连接中就可以同时并发多个请求，不同请求的帧数据可穿插在一起，根据流ID分组即可。**这样直接解决了HTTP1.0的核心痛点，通过这种复用TCP连接的方式，不用再同时建多个连接，提升了TCP的利用率。**

3 RPC注册发现

gRPC开源组件官方并未直接提供服务注册与发现的功能实现，但其设计文档已提供实现的思路，并在不同语言的gRPC代码API中已提供了命名解析和负载均衡接口供扩展。



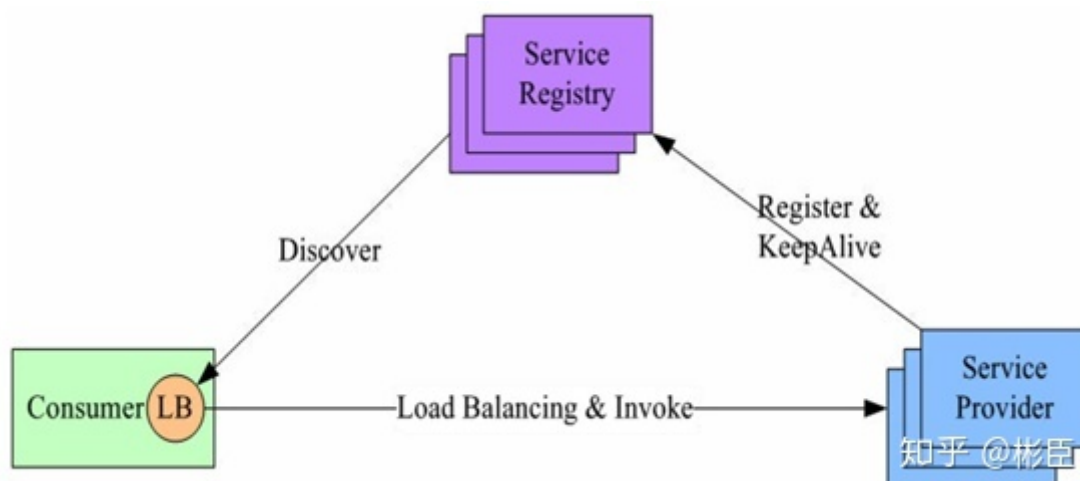
其实现基本原理：

1. 服务启动后gRPC客户端向命名服务器发出名称解析请求，名称将解析为一个或多个IP地址，每个IP地址标示它是**服务器地址**还是**负载均衡器地址**，以及标示要使用那个客户端负载均衡策略或服务配置。
2. 客户端实例化负载均衡策略，如果解析返回的地址是负载均衡器地址，则客户端将使用grpc_lb策略，否则客户端使用服务配置请求的负载均衡策略。
3. 负载均衡策略为每个服务器地址创建一个子通道（channel）。

4. 当有RPC请求时，负载均衡策略决定那个子通道即 gRPC 服务器将接收请求，当可用服务器为空时客户端的请求将被阻塞。

根据gRPC官方提供的设计思路，基于进程内LB方案（阿里开源的服务框架 Dubbo 也是采用类似机制），结合分布式一致的组件（如Zookeeper、Consul、Etc），可找到gRPC服务发现和负载均衡的可行解决方案。

PS: 进程内LB (Balancing-aware Client) 方案

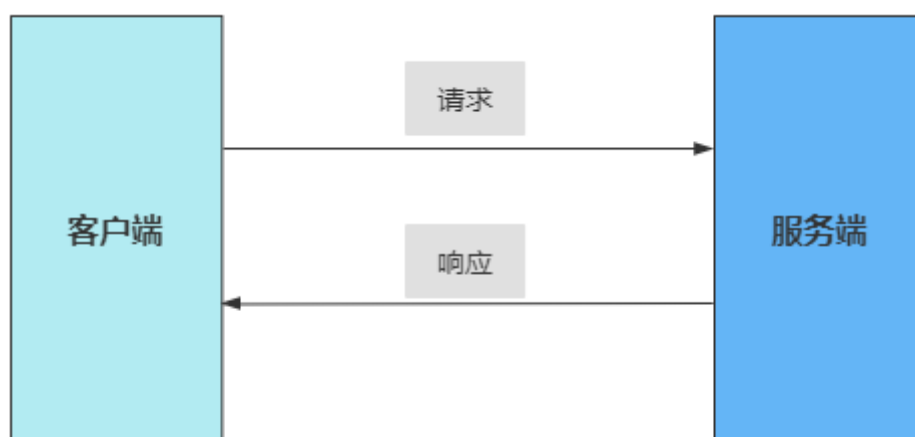


此方案将LB的功能集成到服务消费方进程里，也被称为软负载或者客户端负载方案。服务提供方启动时，首先将服务地址注册到服务注册表，同时定期报心跳到服务注册表以表明服务的存活状态，相当于健康检查，服务消费方要访问某个服务时，它通过内置的LB组件向服务注册表查询，同时缓存并定期刷新目标服务地址列表，然后以某种负载均衡策略(Round Robin, Random, etc) 选择一个目标服务地址，最后向目标服务发起请求。LB和服务发现能力被分散到每一个服务消费者的进程内部，同时服务消费方和服务提供方之间是直接调用，没有额外开销，性能比较好。

4 gRPC 4种模式

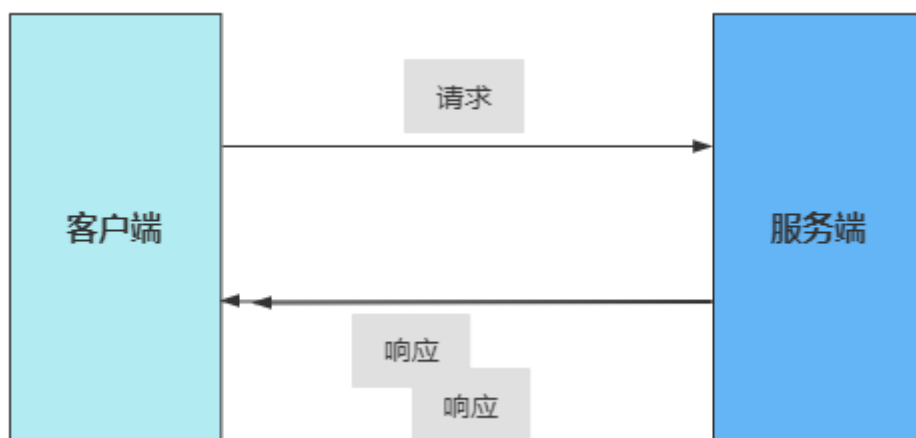
1. 一元RPC模式

一元 RPC 模式也被称为**简单 RPC 模式**。在该模式中，当客户端调用服务器端的远程方法时，客户端发送请求至服务器端并获得一个响应，与响应一起发送的还有状态细节以及 trailer 元数据。



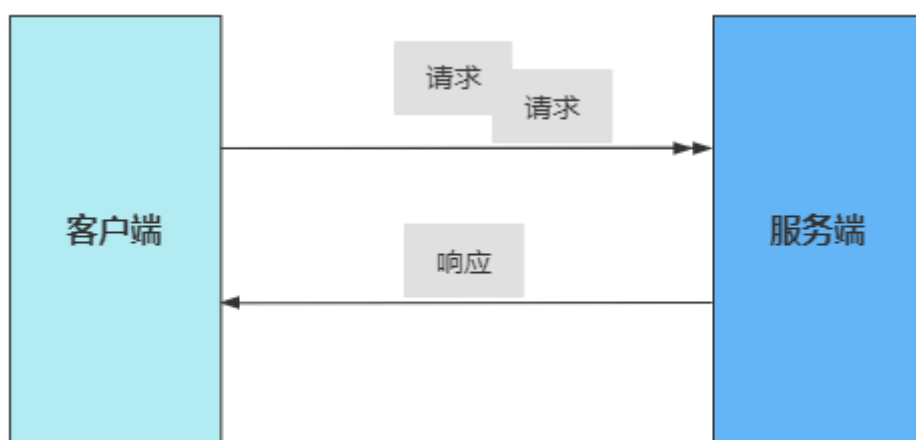
2. 服务器端流RPC模式

在一元 RPC 模式中，gRPC 服务器端和 gRPC 客户端在通信时始终只有一个请求和一个响应。在服务器端流 RPC 模式中，服务器端在接收到客户端的请求消息后，会发回一个响应的序列。这种多个响应所组成的序列也被称为“流”。在将所有的服务器端响应发送完毕之后，服务器端会以 trailer 元数据的形式将其状态发送给客户端，从而标记流的结束。



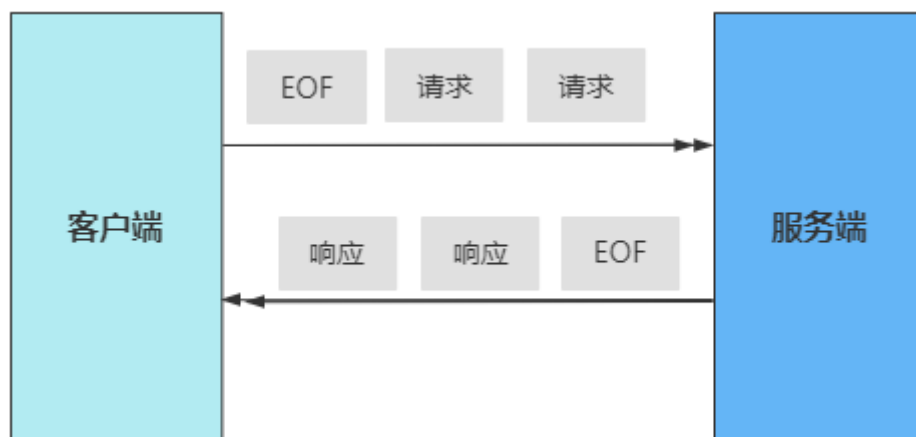
3. 客户端流RPC模式

在客户端流 RPC 模式中，客户端会发送多个请求给服务器端，而不再是单个请求。服务器端则会发送一个响应给客户端。但是，服务器端不一定要等到从客户端接收到所有消息后才发送响应。基于这样的逻辑，我们可以在接收到流中的一条消息或几条消息之后就发送响应，也可以在读取完流中的所有消息之后再发送响应。



4. 双向流RPC模式

在双向流 RPC 模式中，客户端以消息流的形式发送请求到服务器端，服务器端也以消息流的形式进行响应。调用必须由客户端发起，但在此之后，通信完全基于 gRPC 客户端和服务端的应用程序逻辑。



5 GRPC异步同步

具体参考：examples/cpp/helloworld

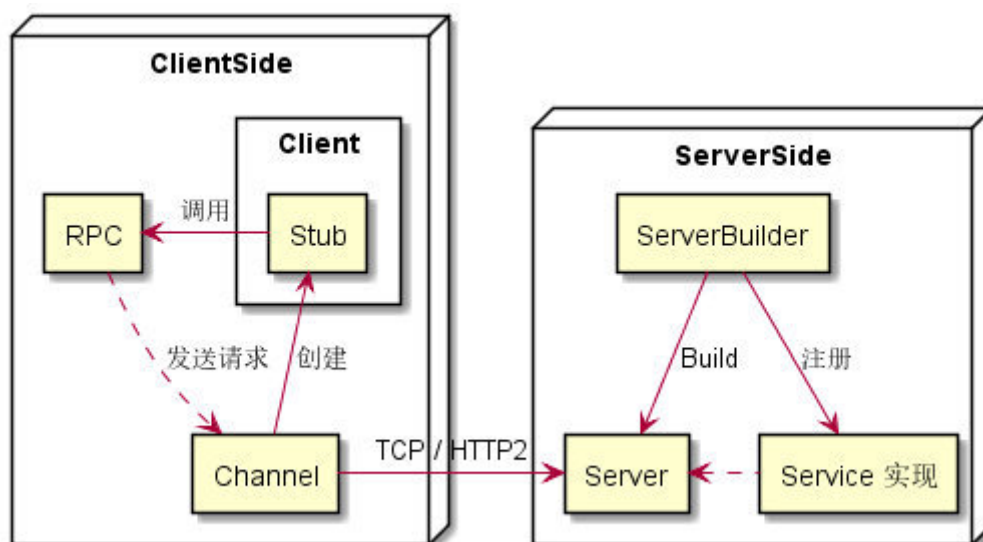
代码分析见课堂讲解。

如果一步步把RPC集成到自己项目，见课堂讲解。

[Quick start | C++ | gRPC](#)

<https://grpc.io/docs/languages/cpp/quickstart/>

5.1 基本概念概览



上图中列出了 gRPC 基础概念及其关系图。其中包括：**Service(定义)**、**RPC**、**API**、**Client**、**Stub**、**Channel**、**Server**、**Service(实现)**、**ServiceBuilder** 等。

接下来，以官方提供的 `example/helloworld` 为例进行说明。

5.1.1 proto文件

helloworld.proto

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.examples.helloworld";
option java_outer_classname = "HelloWorldProto";
option objc_class_prefix = "HLW";

package helloworld;

// The greeting service definition.
service Greeter {
    // Sends a greeting
    rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
    string name = 1;
}

// The response message containing the greetings
message HelloReply {
    string message = 1;
}
```

5.1.2 client

`class GreeterClient` 是 **Client**, 是对 **Stub** 封装; 通过 **Stub** 可以真正的调用 RPC 请求。

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

    std::string SayHello(const std::string& user) {
    ...
private:
    std::unique_ptr<Greeter::Stub> stub_;
};
```

Channel 提供一个与特定 gRPC server 的主机和端口建立的连接。

Stub 就是在 **Channel** 的基础上创建而成的。

```
target_str = "localhost:50051";
auto channel =
    grpc::CreateChannel(target_str, grpc::InsecureChannelCredentials());
GreeterClient greeter(channel);
std::string user("world");
std::string reply = greeter.SayHello(user);
```

5.1.3 server

Server 端需要实现对应的 RPC，所有的 RPC 组成了 **Service**：

```
class GreeterServiceImpl final : public Greeter::Service {
    Status SayHello(ServerContext* context, const HelloRequest* request,
                    HelloReply* reply) override {
        std::string prefix("Hello ");
        reply->set_message(prefix + request->name());
        return Status::OK;
    }
};
```

Server 的创建需要一个 **Builder**，添加上监听的地址和端口，**注册**上该端口上绑定的服务，最后构建出 Server 并启动：

```
ServerBuilder builder;
builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
builder.RegisterService(&service);
std::unique_ptr<Server> server(builder.BuildAndStart());
```

RPC 和 API 的区别：RPC (Remote Procedure Call) 是一次远程过程调用的整个动作，而 API (Application Programming Interface) 是不同语言在实现 RPC 中的具体接口。**一个 RPC 可能对应多种 API，比如同步的、异步的、回调的。**一次 RPC 是对某个 API 的一次调用，比如：

```
std::unique_ptr<ClientAsyncResponseReader<HelloReply> > rpc(
    stub->PrepareAsyncSayHello(&context, request, &cq));
```

不管是哪种类型 RPC，都是由 Client 发起请求。

5.2 异步相关概念

不管是 Client 还是 Server，异步 gRPC 都是利用 [CompletionQueue](#) API 进行异步操作。基本的流程：

- 绑定一个 `CompletionQueue` 到一个 RPC 调用
- 利用唯一的 `void* Tag` 进行读写
- 调用 `CompletionQueue::Next()` 等待操作完成，完成后通过唯一的 Tag 来判断对应什么请求/返回进行后续操作

官方文档 [Asynchronous-API tutorial](#) 中有上边的介绍，并介绍了异步 client 和 server 的解释，对应这 `greeter_async_client.cc` 和 `greeter_async_server.cc` 两个文件。

Client 看文档可以理解，但 Server 的代码复杂，文档和注释中的解释并不是很好理解，接下来会多做一些解释。

5.2.1. 异步 Client

`greeter_async_client.cc` 中是异步 Client 的 Demo，其中只有一次请求，逻辑简单。

- 创建 `CompletionQueue`
- 创建 RPC (既 `ClientAsyncResponseReader<HelloReply>`)，这里有两种方式：

- `stub_>PrepareAsyncSayHello()` + `rpc->StartCall()`
- `stub_>AsyncSayHello()`
- 调用 `rpc->Finish()` 设置请求消息 reply 和唯一的 tag 关联, 将请求发送出去
- 使用 `cq.Next()` 等待 Completion Queue 返回响应消息体, 通过 tag 关联对应的请求

5.2.2. 异步 Server

`RequestSayHello()` 这个函数没有任何的说明。只说是: "we request that the system start processing SayHello requests." 也没有说跟 `cq_>Next(&tag, &ok);` 的关系。我这里通过加上一些日志打印, 来更清晰的展示 Server 的逻辑:

```

1  Server.listening.on.0.0.0.0:50051
2  =====
3  CallData.constructing..this:.0x2e0fcd0
4  [0x2e0fcd0].Proceed(),..Status:.0
5  =====
6  before.cq_>Next
7  after.cq_>Next,.tag:.0x2e0fcd0
8  before.static_cast
9  [0x2e0fcd0].Proceed(),..Status:.1
10 CallData.constructing..this:.0x2e195d0
11 [0x2e195d0].Proceed(),..Status:.0
12 after.static_cast
13 -----
14 before.cq_>Next
15 after.cq_>Next,.tag:.0x2e0fcd0
16 before.static_cast
17 [0x2e0fcd0].Proceed(),..Status:.2
18 after.static_cast
19 =====

```

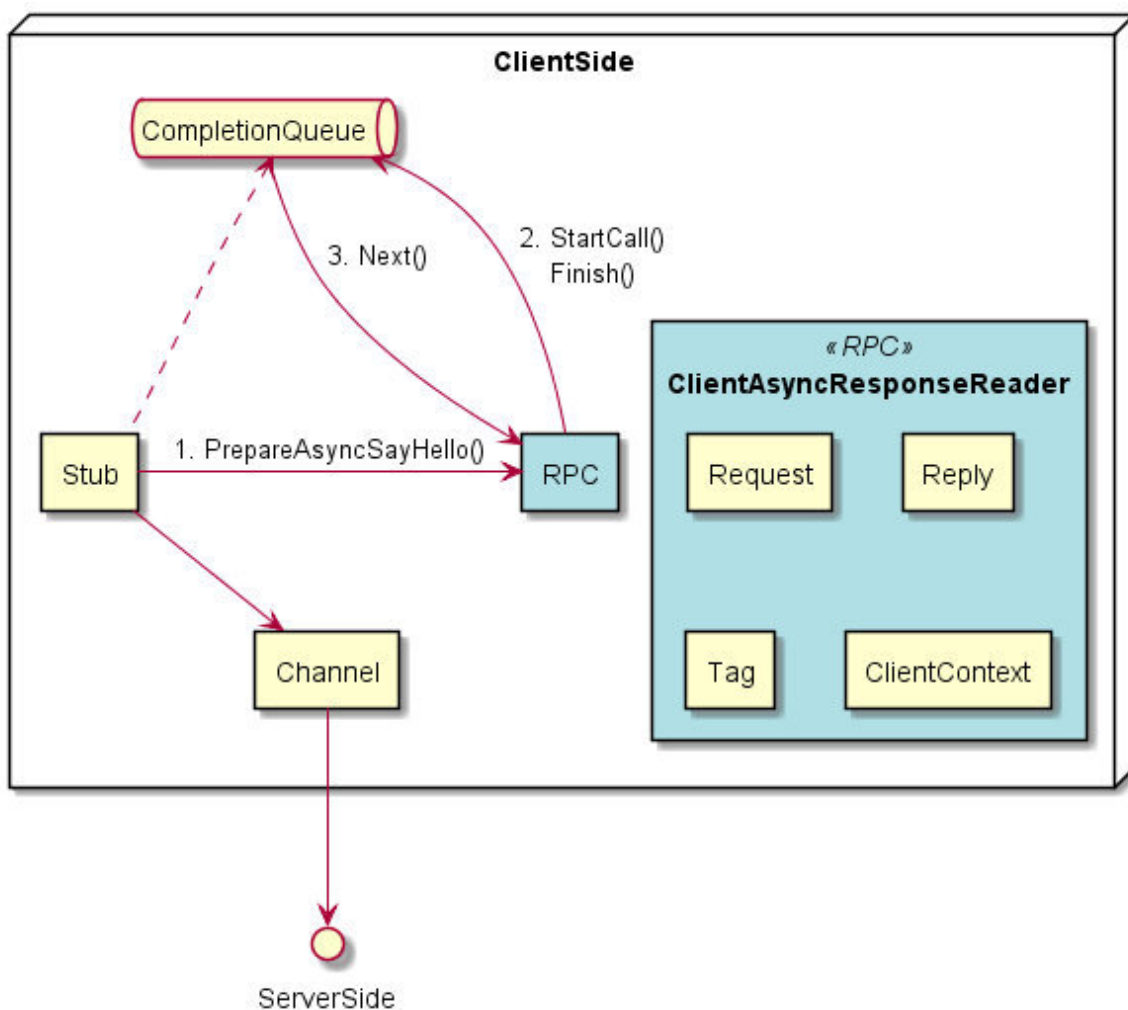
上边绿色的部分为创建的第一个 CallData 对象地址, 橙色的为第二个 CallData 的地址。

- **创建一个 CallData**, 初始构造列表中将状态设置为 **CREATE**
- 构造函数中, 调用 `Process()` 成员函数, 调用 `service_>RequestSayHello()` 后, 状态变更为 **PROCESS**:
 - 传入 `ServerContext ctx_`
 - 传入 `HelloRequest request_`
 - 传入 `ServerAsyncResponseWriter<HelloReply> responder_`
 - 传入 `ServerCompletionQueue* cq_`
 - 将对象自身的地址作为 `tag` 传入
 - 该动作, 能将事件加入事件循环, 可以在 **CompletionQueue** 中等待
- 收到请求, `cq->Next()` 的阻塞结束并返回, 得到 `tag`, 既上次传入的 CallData 对象地址
- 调用 tag 对应 CallData 对象的 `Proceed()`, 此时状态为 **Process**
 - 创建新的 CallData 对象以接收新请求

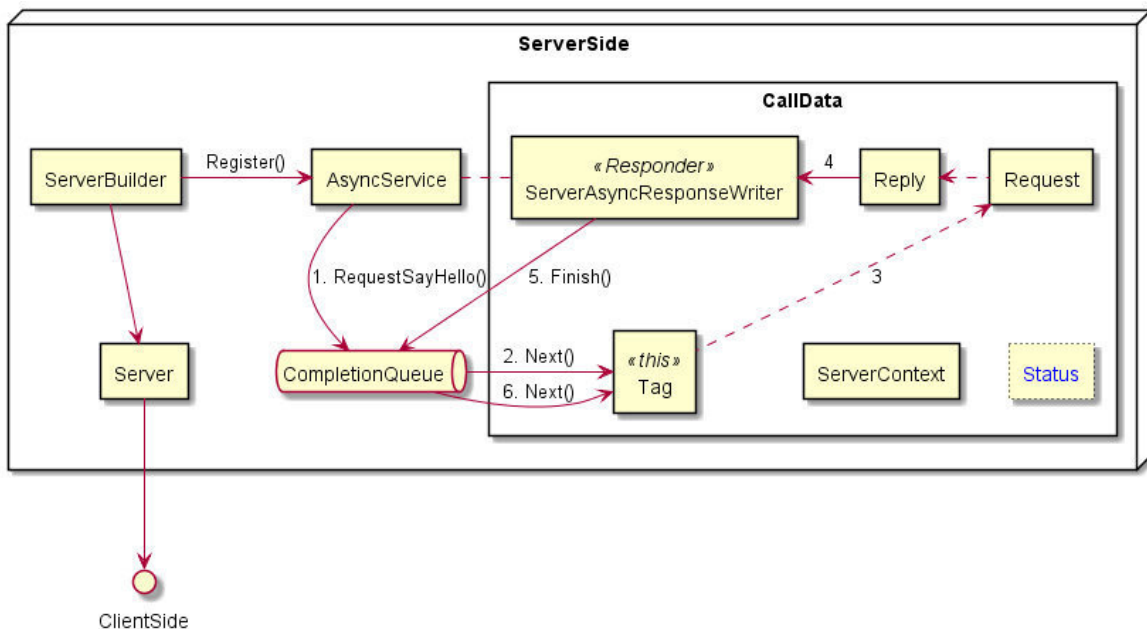
- 处理消息体并设置 `reply_`
 - 将状态设置为 **FINISH**
 - 调用 `responder_.Finish()` 将返回发送给客户端
 - 该动作，能将事件加入到事件循环，可以在 **CompletionQueue** 中等待
- 发送完毕，`cq->Next()` 的阻塞结束并返回，得到 **tag**。现实中，如果发送有异常应当有其他相关的处理
- 调用 tag 对应 `CallData` 对象的 `Proceed()`，此时状态为 **FINISH**，`delete this` 清理自己，一条消息处理完成。

5.2.3. 关系图

将上边的异步 Client 和异步 Server 的逻辑通过关系图进行展示。右侧 RPC 为创建的对象中的内存容，左侧使用相同颜色的小块进行代替。



以下 `CallData` 并非 gRPC 中的概念，而是异步 Server 在实现过程中为了方便进行的封装，其中的 `Status` 也是在异步调用过程中自定义的、用于转移的状态。



5.2.4. 异步 Client 2

在 `example/cpp/helloworld` 中还有另外一个异步 Client，对应文件名为 `greeter_async_client2.cc`。这个例子中使用了两个线程去分别进行发送请求和处理返回，一个线程批量发出 100 个 SayHello 的请求，另外一个不断的通过 `cq_.Next()` 来等待返回。

无论是 Client 还是 Server，在以异步方式进行处理时，都要预先分配好一定的内存/对象，以存储异步的请求或返回。

5.2.5. 回调方式的异步调用

在 `example/cpp/helloworld` 中，还提供了 callback 相关的 Client 和 Server。

使用回调方式简介明了，结构上与同步方式相差不多，但是并发有本质的区别。可以通过文件对比，来查看其中的差异。

```
cd examples/cpp/helloworld/
vimdiff greeter_callback_client.cc greeter_client.cc
vimdiff greeter_callback_server.cc greeter_server.cc
```

其实，回调方式的异步调用属于实验性质的，不建议直接在生产环境使用，这里也只做简单的介绍：

Notice: This API is EXPERIMENTAL and may be changed or removed at any time.

5.2.5.1 回调 Client

发送单个请求，在调用 `SayHello` 时，除了传入 Request、Reply 的地址之外，还需要传入一个接收 Status 的回调函数。

例子中只有一个请求，因此在 `SayHello` 之后，就直接通过 `condition_variable` 的 `wait` 函数等待回调结束，然后进行后续处理。这样其实不能进行并发，跟同步请求差别不大。如果要进行大规模的并发，还是需要使用额外的对象进行封装一下。

```

stub->async()->SayHello(&context, &request, &reply,
                        [&mu, &cv, &done, &status](Status s) {
                            status = std::move(s);
                            std::lock_guard<std::mutex> lock(mu);
                            done = true;
                            cv.notify_one();
                        });

```

上边函数调用函数声明如下，很明显这是实验性（experimental）的接口：

```

void Greeter::Stub::experimental_async::SayHello(
    ::grpc::ClientContext* context, const ::helloworld::HelloRequest* request,
    ::helloworld::HelloReply* response, std::function<void(::grpc::Status)> f);

```

5.2.5.2 回调 Server

与同步 Server 不同的是：

- 服务的实现是继承 `Greeter::CallbackService`
- `SayHello` 返回的不是状态，而是 `ServerUnaryReactor` 指针
- 通过 `CallbackServerContext` 获得 reactor
- 调用 reactor 的 `Finish` 函数处理返回状态

5.3 流相关概念

可以按照 Client 和 Server 一次发送/返回的是单个消息还是多个消息，将 gRPC 分为：

- Unary RPC（一元）
- Server streaming RPC
- Client streaming RPC
- Bidirectional streaming RPC

见：[grpc/examples/cpp/route_guide](#) 范例。

对应的proto文件

```

syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.examples.routeguide";
option java_outer_classname = "RouteGuideProto";
option objc_class_prefix = "RTG";

package routeguide;

// Interface exported by the server. 路线向导
service RouteGuide {
    // A simple RPC.
    //
    // Obtains the feature at a given position.
    //
    // A feature with an empty name is returned if there's no feature at the given
    // position. 定义最简单的RPC服务
    rpc GetFeature(Point) returns (Feature) {}

```

```

// A server-to-client streaming RPC. 带有 stream 的RPC服务, 返回是stream
//
// Obtains the Features available within the given Rectangle. Results are
// streamed rather than returned at once (e.g. in a response message with a
// repeated field), as the rectangle may cover a large area and contain a
// huge number of features.
rpc ListFeatures(Rectangle) returns (stream Feature) {}

// A client-to-server streaming RPC. 带有 stream 的RPC服务, 客户端是stream
//
// Accepts a stream of Points on a route being traversed, returning a
// RouteSummary when traversal is completed.
rpc RecordRoute(stream Point) returns (RouteSummary) {}

// A Bidirectional streaming RPC. 2端都是stream的RPC服务
//
// Accepts a stream of RouteNotes sent while a route is being traversed,
// while receiving other RouteNotes (e.g. from other users).
rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
}

// Points are represented as latitude-longitude pairs in the E7 representation
// (degrees multiplied by 10**7 and rounded to the nearest integer).
// Latitudes should be in the range +/- 90 degrees and longitude should be in
// the range +/- 180 degrees (inclusive).
// 经纬度坐标
message Point {
  int32 latitude = 1; // 纬度
  int32 longitude = 2; // 经度
}

// 经纬度矩形, 表示为两个对角的矩形
// A latitude-longitude rectangle, represented as two diagonally opposite
// points "lo" and "hi".
message Rectangle {
  // One corner of the rectangle.
  Point lo = 1;

  // The other corner of the rectangle.
  Point hi = 2;
}

// A feature names something at a given point.
// 特征名, 对于给定的坐标, 客户端发送给定的特征名返回坐标
// If a feature could not be named, the name is empty.
message Feature {
  // The name of the feature. 特征名
  string name = 1;

  // The point where the feature is detected. 坐标
  Point location = 2;
}

// A RouteNote is a message sent while at a given point.
// 路线注释
message RouteNote {
  // The location from which the message is sent.

```



```

Point location = 1;

// The message to be sent.
string message = 2;
}

// A RouteSummary is received in response to a RecordRoute rpc.
// 路线指南总结
// It contains the number of individual points received, the number of
// detected features, and the total distance covered as the cumulative sum of
// the distance between each point.
message RouteSummary {
    // The number of points received.
    int32 point_count = 1;

    // The number of known features passed while traversing the route.
    int32 feature_count = 2;

    // The distance covered in metres.
    int32 distance = 3; // 距离

    // The duration of the traversal in seconds.
    int32 elapsed_time = 4;
}

```

5.3.1 Server 对 RPC 的实现

Server 需要实现 proto 中定义的 RPC，每种 RPC 的实现都需要将 ServerContext 作为参数输入。

如果是一元 (Unary) RPC 调用，则像调用普通函数一样。将 Request 和 Reply 的对象地址作为参数传入，函数中将根据 Request 的内容，在 Reply 的地址上写上对应的返回内容。

```

// rpc GetFeature(Point) returns (Feature) {}
Status GetFeature(ServerContext* context, const Point* point, Feature* feature);

```

如果涉及到流，则会用 Reader 或/和 Writer 作为参数，读取流内容。如 ServerStream 模式下，只有 Server 端产生流，这时对应的 Server 返回内容，需要使用作为参数传入的 `ServerWriter`。这类似于以 'w' 打开一个文件，持续的往里写内容，直到没有内容可写关闭。

```

// rpc ListFeatures(Rectangle) returns (stream Feature) {}
Status ListFeatures(ServerContext* context,
                    const routeguide::Rectangle* rectangle,
                    ServerWriter<Feature>* writer);

```

另一方面，Client 来的流，Server 需要使用一个 ServerReader 来接收。这类似于打开一个文件，读其中的内容，直到读到 EOF 为止类似。


```
// rpc RecordRoute(stream Point) returns (RouteSummary) {}
Status RecordRoute(ServerContext* context, ServerReader<Point>* reader,
                    RouteSummary* summary);
```

如果 Client 和 Server 都使用流，也就是 Bidirectional-Stream 模式下，输入参数除了 ServerContext 之外，只有一个 `ServerReaderWriter` 指针。通过该指针，既能读 Client 来的流，又能写 Server 产生的流。

例子中，Server 不断地从 stream 中读，读到了就将对应的写过写到 stream 中，直到客户端告知结束；Server 处理完所有数据之后，直接返回状态码即可。

```
// rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
Status RouteChat(ServerContext* context,
                  ServerReaderWriter<RouteNote, RouteNote>* stream);
```

5.3.2 Client 对 RPC 的调用

Client 在调用一元 (Unary) RPC 时，像调用普通函数一样，除了传入 ClientContext 之外，将 Request 和 Response 的地址，返回的是 RPC 状态：

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

Client 在调用 ServerStream RPC 时，不会得到状态，而是返回一个 ClientReader 的指针：

```
// rpc ListFeatures(Rectangle) returns (stream Feature) {}
unique_ptr<ClientReader<Feature>> ListFeatures(ClientContext* context,
                                              const Rectangle& request);
```

Reader 通过不断的 `Read()`，来不断的读取流，结束时 `Read()` 会返回 `false`；通过调用 `Finish()` 来读取返回状态。

调用 ClientStream RPC 时，则会返回一个 ClientWriter 指针：

```
// rpc RecordRoute(stream Point) returns (RouteSummary) {}
unique_ptr<ClientWriter<Point>> RecordRoute(ClientContext* context,
                                           RouteSummary* response);
```

Writer 会不断的调用 `write()` 函数将流中的消息发出；发送完成后调用 `writeDone()` 来说明发送完毕；调用 `Finish()` 来等待对端发送状态。

而双向流的 RPC 时，会返回 ClientReaderWriter，：

```
// rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
unique_ptr<ClientReaderWriter<RouteNote, RouteNote>> RouteChat(
    ClientContext* context);
```

前面说明了 Reader 和 Writer 读取和发送完成的函数调用。因为 RPC 都是 Client 请求而后 Server 响应，双向流也是要 Client 先发送完自己流，才有 Server 才可能结束 RPC。所以对于双向流的结束过程是：

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

示例中创建了单独的一个线程去发送请求流，**在主线程中读返回流，实现了一定程度上的并发。**

5.3.3 流是会结束的

并不似长连接，建立上之后就一直保持，有消息的时候发送。（是否有通过建立一个流 RPC 建立推送机制？）

- Client 发送流，是通过 `writer->WritesDone()` 函数结束流
- Server 发送流，是通过结束 RPC 函数并返回状态码的方式来结束流
- 流接受者，都是通过 `Reader->Read()` 返回的 bool 型状态，来判断流是否结束

Server 并没有像 Client 一样调用 `WriteDone()`，而是在消息之后，将 status code、可选的 status message、可选的 trailing metadata 追加进行发送，这就意味着流结束了。

6 参考和扩展阅读

RPC 远程过程调用 专栏：https://blog.csdn.net/wohu1104/category_11323760.html

RPC源码分析：<https://cloud.tencent.com/developer/article/1416302>

gRPC 基础概念详解：<https://zhuanlan.zhihu.com/p/389328756>