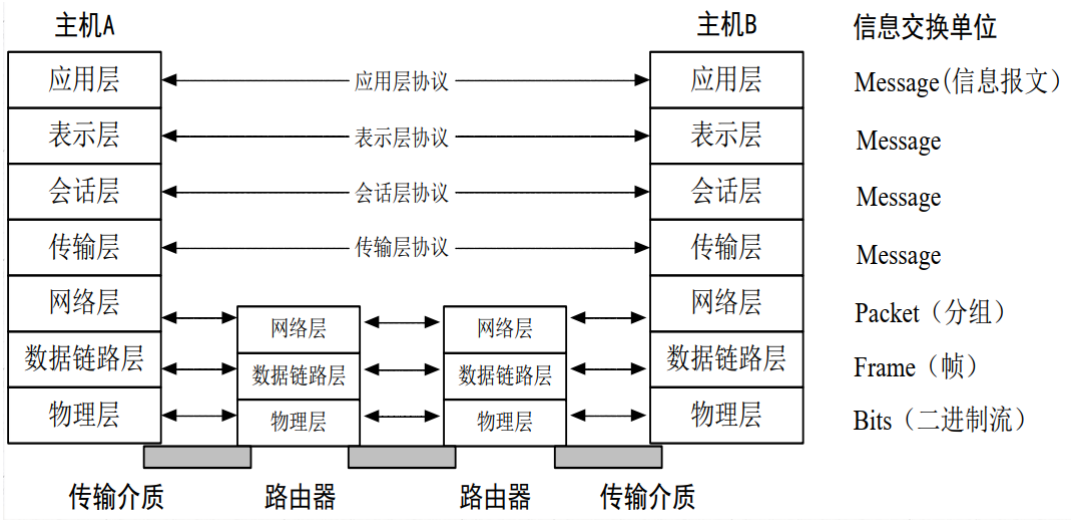
通信基础

DragonAura 2023/7/15

HTTP

TCP/IP

1983 年，国际标准化组织提出了 OSI 参考模型



> - 物理层：传输二进制位流
> - 数据链路层：保证信息帧的正确传输
> - 网络层：报文分组、路由选择
> - 传输层：保证端到端的信息正确传输
> - 会话层：建立、管理、结束主机间的会话
> - 表示层：数据压缩解压缩、加密解密、文件格式转换等
> - 应用层：为应用程序提供网络服务

OSI 模型过于复杂，因此没有被真正实现过。如今，我们实际使用的是 TCP/IP 模型。它把应用层、表示层、会话层合并为应用层，把网络层改为互联网络层，把数据链路层和物理层合并为链路层，这就形成了我们今天使用的 TCP/IP 模型。

IP

IP 是互联网协议，其任务为根据地址来传送数据，是网络层的主要协议。互联网由不同的物理网络互联而成，不同网络之间要实现通信，就要有相应的地址标识，这个地址标识称之为 IP 地址。

IP 地址分为 IPV4 和 IPV6。IPV4 用 32 位二进制表示，物理上限至多只能表示 $2^{32} = 4.3 \times 10^9$ 个地址（实际由于地址的分类、规则问题，可用的 IPV4 地址更少）。因此，诞生了 IPV6 地址。IPV6 用 32 个 16 进制数，即 128 位二进制表示，至多可以表示 $2^{128} = 3.4 \times 10^{38}$ 个地址，因此足以给地球上的每一粒沙子都编一个 IPV6 地址。基于这种好处，世界各地正在积极部署 IPV6。

> IPV4 到 IPV6 的转换过程中面临着重重问题，这一转换过程依然需要漫长的时间。

形如 "202.108.249.134" 这样的地址为 IPV4 地址，形如 "2031:0000:130F:0000:0000:09C0:876A:130B" 这样的地址为 IPV6 地址。

可以看到，这样的地址与自然语言相去甚远，不利于记忆，因此就产生了域名。

形如 "www.baidu.com" 就称为域名，域名能使人方便快捷地使用互联网，但是在互联网中只有 IP 地址才能被计算机识别，因此产生了 DNS，用于从域名到 IP 地址建立映射，此处不再赘述。

TCP

TCP 是传输控制协议，是传输层的主要通信协议。根据上面的介绍，我们会发现，传输层下的网络层只是提供了端到端的连接，并没有保证接收端收到的消息和发送端发送的消息是一致的。而实际信号的传输过程中，经常会出现各种错误，导致数据包丢失等情况。为此，TCP 诞生了——TCP 通过三次握手、计算校验、超时重传等各种方式，来确保接收端收到的消息与发送端是一致的。

> 此处不再赘述 TCP 的工作原理，感兴趣的同学可以自行查阅相关资料。

端口（Port）

TCP 协议保证了信息的正确传输，但是对于更上面一层的应用层而言，仅仅做到信息传输是不够的——想象一个情景，你打开了某个游戏，同时还在观看视频。假设它们不加区分地通过 TCP 传输，那么每个接收的包中都要对自己的内容做对应的表示，告诉应用层"我是游戏""我是视频"来互相区分，这无疑是难以标准化的，同时也占用了宝贵的信道资源。

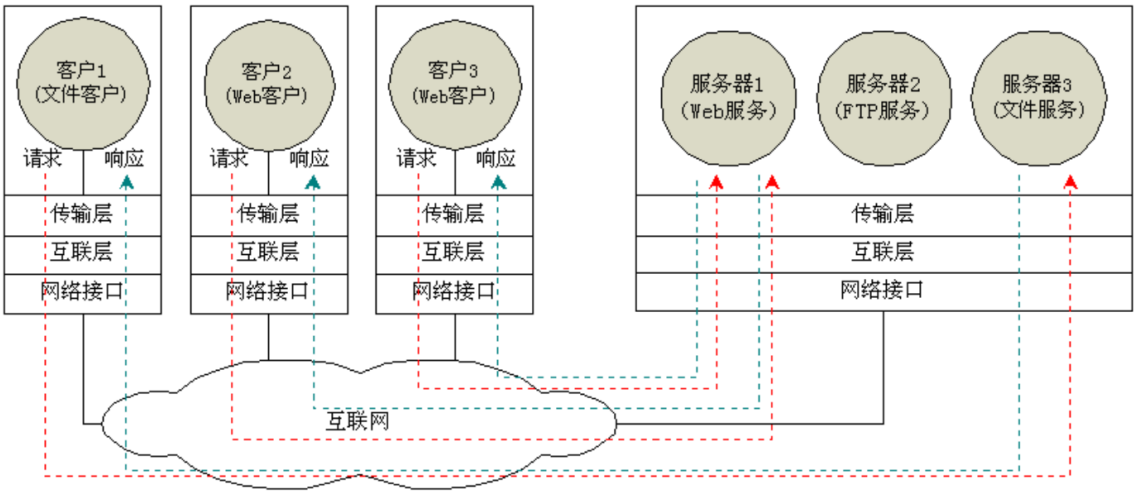
因此，我们通过端口，来将各种网络流量转移到其最终点，也即实际应用层中的应用，这样就在一定程度上解决了上面的问题。如今，我们有 65535 个不同的端口，其中有少部分被作为公认的端口号，例如 HTTP 默认使用 80 端口，HTTPS 默认使用 443 端口等。

HTTP

HTTP 基于 TCP/IP 来传输数据。它有几种特点：

- 基于 Server-Client 模式：

  Server-Client 模式是一种经典的通信模型，客户端向服务端提出请求，服务端根据客户端的请求，向客户端返回结果、提供对应的服务，具体的模式如下图所示。



一台主机可同时运行多个服务器程序，服务器程序需要并发地处理多个客户的请求

  Server-Client 模式的优势在于它提高了网络的运行效率，同时充分发挥了服务端和客户端各自的优势。

- 连接：限制每次连接只处理一个请求，服务器处理完客户的请求，并收到客户的应答后，即断开连接，采用这种方式可以节省传输时间。
- 媒体独立：只要客户端和服务器知道如何处理的数据内容，任何类型的数据都可以通过 HTTP 发送
- 无状态：对于事务处理没有记忆能力

HTTP 的传输格式

请求格式

- 请求行
  - 请求方法：
    - `GET`：获取一个资源
      - `POST`：创建一个资源
        - 登录：向科协网站后端服务器（`https://api.eesast.com/users/login`）发送账号密码，后端返回给客户端一个 `token`，然后每次操作，把 `token` 发送给服务器进行操作
    - `PUT`：更改资源
    - `DELETE`：删除资源
    - `HEADER`
    - ……
  - URL、协议版本…
- 请求头：一系列的键值对
  - `Content-Type`：请求体的格式
  - `Content-Length`：请求的总大小
  - `User-Agent`：使用的是什么客户端
  - `Authorization`：认证信息（例如登录产生的 `token`）
- 参数：（键值对）
- 请求体：一堆字节
  - 请求体到底是什么，通常用 `Content-Type` 指明，例如 JSON 的 `Content-Type` 是 `application/json`

响应格式

- 状态码
  - `2` 开头的：代表请求成功
    - `200`：请求成功
    - `204`：请求成功，并且 Server 没有返回任何内容
  - `4` 开头：客户端错误
    - `403`：Forbidden
    - `404`：Not Found
  - `5` 开头：服务器内部错误
- 响应体：消息报头（`Content-Type`，`Content-Length`…）、响应正文

URL 格式

`scheme://host:port/resource_path?param1=val1&param2=val2#id`

- `scheme`：协议，例如 `http`
- `host`：主机名，例如 `www.baidu.com`

- `port`：端口，不提供的话会使用默认设置（如 `http` 默认使用 `80`）
- `/resource_path`：资源路径，例如 `bilibili.com/video/BVxxxx`
- `?param1=val1&param2=val2`：查询字符串，用于发送一些特定的参数，例如 `uth4.tsinghua.edu.cn/succeed_wired.php?ac_id=1&username=xxx&ip=yyy`
- `#id`：片段标识符，用于指定网页上的一个特定部分，浏览器会自动滚动到带有该id的元素。

小结

由于时间所限，还有很多内容我们没有涉及，例如

- TCP 的具体技术原理
- TCP 与 UDP 的区别与联系
- ...

略去上述内容不会对我们的后续学习产生太大影响，感兴趣的同学可以自行查阅相关文档了解。

Protobuf

序列化协议

实际传输中，我们会面临各种问题，例如：

- 要传输的数据量很大，但其实有效的数据却不多

  > 例如，传输下面这样一个数组：
  >
  > ```
  > 1 long long arr[5] = {1, 2, 3, 4, 1000000000000};
  > ```

- 要传输的的数据类型非常复杂，难以传递

  > 例如，传输下面的类：
  >
  > ```
  > 1 struct Bar
  > 2 {
  > 3  int integer;
  > 4  std::string str;
  > 5  float flt[100];
  > 6 };
  > ```

那么我们如何正确而高效地进行这种传递呢？在发送端，我们需要使用一定的规则，将对象转换为一串字节数组，这就是序列化；在接收端，我们再以同样的规则将字节数组还原，这就是反序列化。

常见的序列化协议有 XML、JSON 和 Protobuf。

- XML（eXtensible Markup Language，可扩展标记语言）使用标签 `<xx>` 和 `</xx>` 来区隔不同的数据。
- JSON（JavaScript Object Notation，JavaScript 对象简谱）使用 JavaScript 构造对象的方法来存储、传输数据。
- Protobuf（Protocol Buffers）是 Google 公司开源跨平台的序列化数据结构的协议。

我们通过一个例子来说明三者的差异：定义

```
struct Helloworld
{
    int id;
    std::string name;
};
```

使用 XML 序列化：

```
<helloworld>
   <id>101</id>
   <name>hello</name>
</helloworld>
```

使用 JSON 序列化：

```
{
  "id": 101,
  "name": "hello"
}
```

使用 Protobuf 序列化：

```
08 65 12 06 48 65 6C 6C 6F 77
```

	XML	JSON	Protobuf
数据存储格式	文本	文本	二进制
可读性	好	较好	差
存储空间	大	较大	小
序列化/反序列速度	慢	慢	快
侧重点	数据结构化	数据结构化	数据序列化

本次课程，我们重点介绍 Protobuf 的用法，因为通信中 Protobuf 的高度序列化是极其有利于通信的。

Protobuf 的安装

在 Protobuf 一节中，我们统一采用 C++ 来演示操作。

protobuf 可以通过以下方式安装（参考自Protobuf C++ Installation）

```
$ sudo apt-get install autoconf automake libtool curl make g++ unzip
  # 安装所需要的工具包
$ git clone https://github.com/protocolbuffers/protobuf.git
  # 若网络不佳，可以将指令换为 git clone
https://gitee.com/mirrors/protobuf_source.git ./protobuf
$ cd protobuf
  # (optional) git submodule update --init --recursive
$ git checkout 3.20.x # 根据版本需求选择不同的分支
$ ./autogen.sh
$ ./configure
$ make -j$(nproc)
$ sudo make install
$ sudo ldconfig
```

以上操作会将`protoc`可执行文件（后续教程会介绍其使用方法）以及与 protobuf 相关的头文件、库安装至本机。在终端输入`protoc`，若输出提示信息，则表示安装成功。

定义第一个 proto 文件

首先，我们需要在一个 .proto 文件中将被序列化的数据结构进行定义

```
// import "other_protos.proto"; // 如果需要引用其它的protobuf文件，可以使用import
语句。

syntax = "proto3"; // 指定protobuf遵循的语法格式是proto2还是proto3。在本教程和之后的
开发中，我们都使用proto3语法格式。
package student; // 包名声明。如在本例中，proto文件生成的类都会被放在namespace
student中，这一举措的意义在于防止命名冲突

enum Sex // 自定义枚举类型
{
    MALE = 0;
    FEMALE = 1;
}

message Course // protobuf中，使用message定义数据结构，类似于C中的结构体
{
    int32 credit = 1;
    string name = 2;
}

message StudentInfo
{
    // 变量声明格式 <限定修饰符> <数据类型> <变量名>=id
    int32 age = 1;
    string name = 2;
    Sex sex = 3;
    repeated Course courses = 4; // repeated表示重复（数组），本例也表明message可
以嵌套message
}
```

protobuf 语法标准

protobuf 有两套语法标准：proto2 和 proto3，两套语法不完全兼容。我们可以使用`syntax`关键字指定 protobuf 遵循的语法标准。

package

为了防止命名冲突，protobuf 文件中可以声明包名（package）。具体效果将在后续章节介绍。

编号

消息定义中的每个字段都有一个唯一的编号，从 1 开始。这些字段号用于识别你在二进制格式消息中的信息。

一个常见的约定是，我们会将经常使用的字段编号为 1-15，不常用的字段编号为 16 以上的数字，因为 1-15 的编号编码仅需要 1 byte，这样可以减小字节流的体积。

数据类型

Protobuf 中常见的基础数据类型与若干编程语言的对应关系如下：

proto Type	C++ Type	Python Type	C# Type
double	double	float	double
float	float	float	float
int32	int32	int	int
int64	int64	int/long	long
uint32	uint32	int/long	uint
uint64	uint64	int/long	ulong
sint32	int32	int	int
sint64	int64	int/long	long
fixed32	uint32	int/long	uint
fixed64	uint64	int/long	ulong
sfixed32	int32	int	int
sfixed64	int64	int/long	long
bool	bool	bool	bool
string	string	str/unicode	string
bytes	string	str (Python 2) bytes (Python 3)	ByteString

更多语言的对应关系参看Protobuf scalar types。

此外，Protobuf 还支持使用`enum`关键字定义枚举类型。每个枚举定义都必须包含一个映射到 0 的常量作为枚举的默认值。

为了尽可能多地压缩数据，Protobuf 对各数据类型地默认值做了以下处理：

- `numeric types`: 0
- `bool`: false
- `string`: 空字符串
- `byte`: 空字节
- `enum`: 第一个定义的枚举值（0）
- `message`: 取决于目标编程语言

`repeated`

`repeated`关键字可以定义重复多次的信息（即数组），其顺序是有序的。

命名法

为了便于阅读，protobuf 规定了一系列命名法：

- message、enum 采用大驼峰命名法，如`message StudentInfo`。
- 字段采用下划线分割法，且全部小写，如`string student_name`。
- 枚举值采用下划线分割法，且全部大写，如`FIRST_VALUE`。

Protobuf 高级语法

protobuf 中还有一些高级语法：

`oneof`

如果你有一个信息，它可能包含若干种字段，并且最多只有一个字段会同时被设置（回忆 C/C++中的联合体`union`），你可以使用`oneof`字段来节省空间。

oneof 块中可以定义除了 map 字段（后续会讲到）和 repeated 字段外的所有类型字段。

```
syntax = "proto3";
package oneof_demo

message MessageA
{
    string name_a = 1;
}

message MessageOneof
{
    oneof test_oneof
    {
        string name = 1;
        MessageA message_a = 2;
    }
}
```

`map`

`map`字段可以定义关联映射类型（类似于 Python 中的字典`dict()`）。

`map`字段的定义方式如下：`map<key_type, value_type> map_field = N;`。其中，`key_value`可以为整数类型或字符串类型，`value_type`为除`map`类型的任意类型。

```
syntax = "proto3";
package map_demo

message StudentInfo
{
    map<int32,string> id_name_pairs = 1;
}
```

除此之外，protobuf 中还有很多高阶语法：

- `Any`
- 保留字段（Reserved Values）
- 嵌套类型（Nested Types）
- ...

此处由于篇幅所限，我们不做过多展开。

使用 proto 文件进行序列化和反序列化

生成目标语言文件

编写好的 protobuf 文件不能直接应用于工程中，我们需要使用`protoc`工具生成对应的文件（以 C++为例）：

```
protoc --help # 查看使用方法
protoc Example.proto --cpp_out=. # 在当前目录下生成.cpp文件和.h文件
```

在目标语言中编写代码

以 C++ 为例（其他语言类似，可以自行在文档中查看），例程如下：

```
#include <iostream>
#include <fstream>
#include <vector>
#include <google/protobuf/message.h>  // for protobuf

#include "Example.pb.h"  // for protobuf source file

int main()
{
    // 可以看到，protobuf文件中的信息都被封装在namespace student中，这是之前protobuf中的`package`语法所规定的。

    // 1. 如何实例化一个proto文件中定义的类
    student::StudentInfo student1;

    // 2. 如何设置类的各个属性
    // a. 添加单一字段：使用set_<xxx>()语句
    student1.set_age(18);
    student1.set_name("Alice");
    student1.set_sex(student::Sex::FEMALE);

    // b. 添加repeated字段：使用add_<xxx>()语句
```

```
    student::Course* course1 = student1.add_courses();
    course1->set_name("calculus");
    course1->set_credit(5);

    student::Course* course2 = student1.add_courses();
    course2->set_name("Fundamentals of Electronic Circuits and System");
    course2->set_credit(2);

    // 3. 如何使用类的各个属性：使用<xxx>()语句
    std::cout << "----------------student info----------------" << std::endl
              << "age: " << student1.age() << std::endl
              << "name: " << student1.name() << std::endl
              << "sex (0:male, 1:female): " << (int)student1.sex() << std::endl
              << "courses: " << std::endl;
    for (int i = 0; i < student1.courses_size(); i++)
    {
        std::cout << " " << i << ". "
                  << "name: " << student1.courses(i).name() << "  "
                  << "credit: " << student1.courses(i).credit() << std::endl;
    }
    std::cout << "--------------------------------------------" << std::endl;

    // 4. 序列化
    std::cout << "serialize to file." << std::endl;
    std::fstream output("./output", std::ios::out | std::ios::binary);
    student1.SerializeToOstream(&output);  // 序列化为流
    output.close();

    std::cout << "serialize to array." << std::endl;
    size_t size = student1.ByteSizeLong();
    unsigned char* data = new unsigned char[size];
    student1.SerializeToArray(data, student1.ByteSizeLong());  // 序列化为数组

    // 5. 反序列化和debug
    std::cout << "deserialize from array." << std::endl;
    student::StudentInfo studentInfoFromArray;
    std::cout << std::endl;
    studentInfoFromArray.ParseFromArray(data, size);
    std::cout << studentInfoFromArray.DebugString() << std::endl;  // 输出字符串化的信息

    // 此处有大坑：需要开着 -pthread，否则会报错
}
```

此处需要链接 protobuf 的库，本例中采用 CMake 处理依赖关系。

小结

由于篇幅所限，我们仍然有许多内容没有展开：

- protobuf 编码之 varint/zigzag protobuf 为什么可以获得如此高效的编码效果？这涉及到其底层算法——varint 和 zigzag 算法。

- proto2 语法和 proto3 语法的区别。
- ...

略去上述内容不会对我们的教学产生太大影响，感兴趣的同学可以参考Protobuf 官方文档学习更多知识。

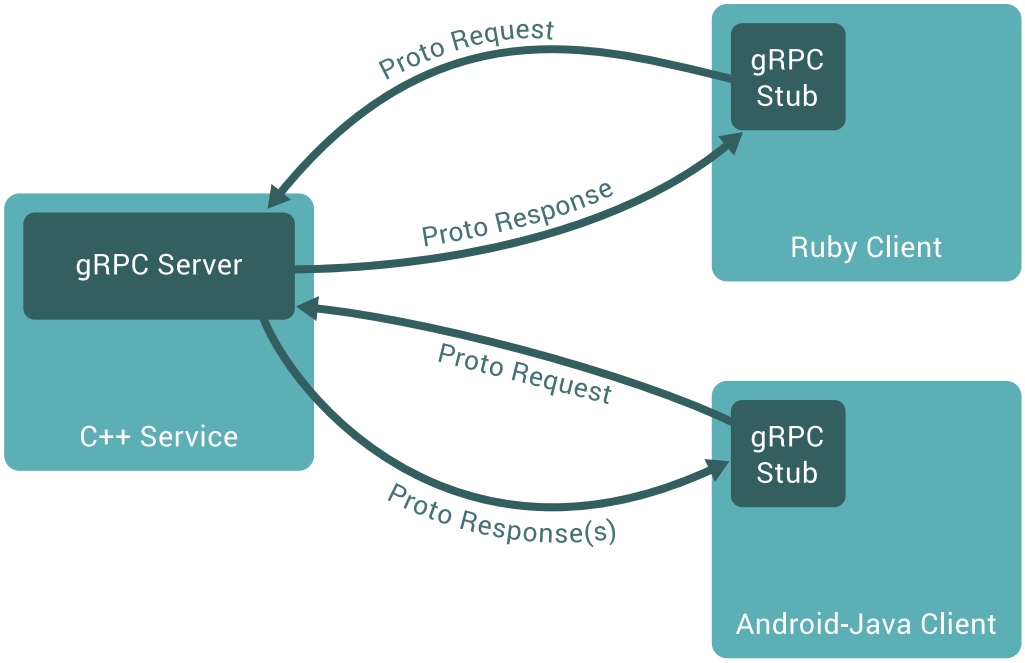
gRPC

gRPC 概览

gRPC 的全称是 gRPC Remote Procedure Calls。其中“Remote Procedure Calls”翻译为“远程过程调用”。“远程过程调用”指的是客户端（Client）可以像调用本地对象一样直接调用服务端（Server）应用的方法。具体过程如下：

1. 定义若干服务（Service），指定其能够被远程调用的方法（包含参数和返回类型）。这些定义都写在`.proto`文件里。
2. 在服务端（Server）实现这个接口（内部处理逻辑），并运行 gRPC 服务器，来处理客户端的调用。
3. 在客户端（Client）建立一个存根（stub），提供与服务端相同的方法。

gRPC官方文档提供了下面这样一张示例图，生动地展示了gRPC的架构。



这样一来，用户在使用 gRPC 构建的应用程序时，不需要关心调用方法的内部逻辑（被封装在 Server 中），只需要调用 Client 端提供的方法向 Server 端提供请求，等待 Server 端返回结果即可——看上去就和在 Client 端本地调用方法一样。

gRPC 有诸多优点：

- 速度快：gRPC 使用 protobuf 进行 Server/Client 之间数据的序列化和反序列化，保证了通信的高效。
- 跨语言：构建 Server 端和 Client 端程序的源语言无需一致。
- 跨平台：Server 端和 Client 端的平台无需一致。

此外，gRPC 采取 HTTP 协议传输信息。

gRPC 安装

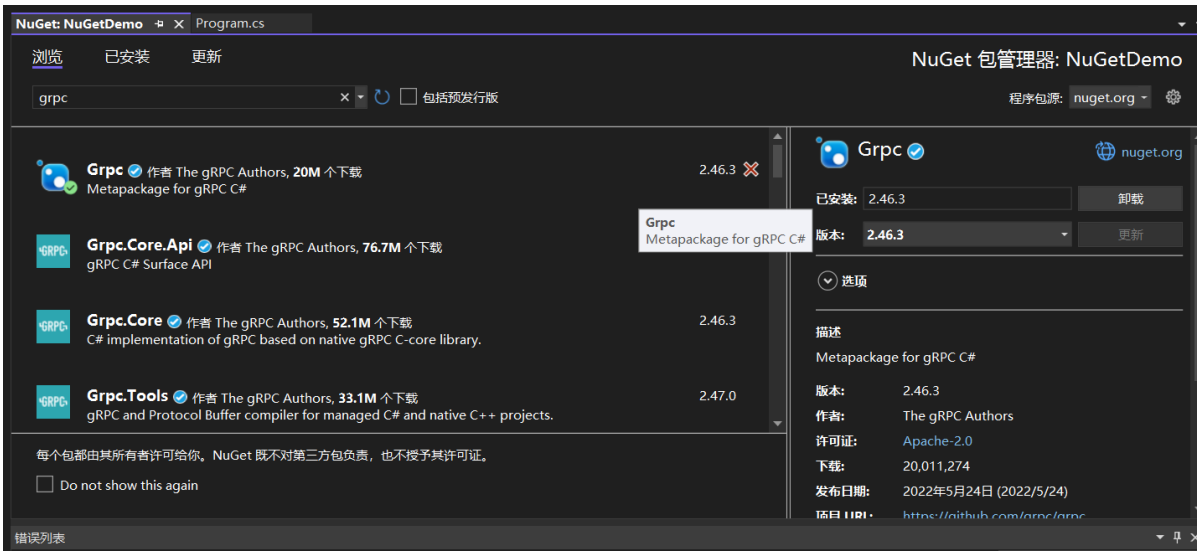
C++

由于 C++ 安装 gRPC 过于困难，此处我们略过不讲。感兴趣的同学，可以自行查阅文档：

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

C#

Csharp 中，我们可以使用 NuGet 程序包安装 gRPC 库（图中第一项）。



Python

Python 中，我们可以使用 pip 安装 gRPC：

```
pip install grpcio grpcio-tools
```

gRPC 服务

proto 定义

gRPC 使用 Protobuf 作为接口定义语言，定义方式见下例：

```
syntax = "proto3";
package hello;

// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply);
  rpc LotsOfReplies(HelloRequest) returns (stream HelloReply);
  rpc LotsOfGreetings(stream 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 reply = 1;
}
```

使用了 `service` 和 `rpc` 关键字。`service` 关键字后是具体的 RPC 内容，`rpc` 关键字定义的服务就类似于一般编程语言的函数，客户端向服务端发送 `request`，服务端接收到后处理，并返回 `response`，客户端接收。

```
rpc FunctionName (request) returns (response)
```

除去单一 RPC 外，gRPC 还支持流式 RPC，使用 `stream` 关键字即可。这允许服务器和客户端在双方连接不中断的情况下发送**多个**消息。

> 例如，在游戏中，服务端经常要连续给客户端发送游戏当前的信息，而客户端经常要连续给服务端发送自己的操作，这时就比较适合流式服务。

gRPC 实例

接下来，我们以上述的 proto 为例，演示如何使用 gRPC。我们使用 C# 作为服务端，用 Python 作为客户端来进行实际演示。此处为了更好地体现 gRPC 的效果，我们在各个服务中都加入了延时，以模拟实际需求中计算的延时。

服务端

我们在 C# 的项目文件中，加入

```
<ItemGroup>
    <Protobuf Include="Example.proto" GrpcServices="Server"/>
</ItemGroup>
```

首先导入所需的依赖：

```
using Grpc.Core;
using Hello;
```

然后开始实现服务。首先，我们需要继承基类

```
class GreeterService : Greeter.GreeterBase
{
// ...
}
```

然后在基类内部重写各个 `rpc` 方法。对于单一 `rpc` 服务，这一过程是比较自然的。为了允许任务的异步执行，我们在返回值中使用 `Task` 关键字。

```
public override Task<HelloReply> SayHello(HelloRequest request,
Grpc.Core.ServerCallContext context)
{
    Thread.Sleep(1000);
    var reply = new HelloReply
    {
        Reply = $"Hello, {request.Name}!"
    };
    return Task.FromResult(reply);
}
```

对于服务器流式 `rpc` 服务，我们需要使用异步方法 `WriteAsync` 将服务器的响应写入异步流 `IServerStreamWriter` 中。

```
public override async Task LotsOfReplies(HelloRequest request,
Grpc.Core.IServerStreamWriter<HelloReply> responseStream,
Grpc.Core.ServerCallContext context)
{
    for (var i = 0; i < 10; i++)
    {
        var reply = new HelloReply { Reply = $"Hello, {request.Name}! {i}" };
        await Task.Delay(1000);
        await responseStream.WriteAsync(reply);
    }
}
```

对于客户端流式 `rpc` 服务，我们需要使用异步流 `IAsyncStreamReader` 逐个读出请求并进行处理。

```
public override async Task<HelloReply>
LotsOfGreetings(Grpc.Core.IAsyncStreamReader<HelloRequest> requestStream,
Grpc.Core.ServerCallContext context)
{
    var replyStr = "";
    while (await requestStream.MoveNext())
    {
        Console.WriteLine($"Received message {requestStream.Current.Name}");
        replyStr += $"Hello, {requestStream.Current.Name}! ";
    }
    var reply = new HelloReply
    {
        Reply = replyStr
    };
    return reply;
}
```

最后，我们启动服务器。需要注意，我们需要关掉端口复用（`SoReuseport, 0`）

```
try
{
    Grpc.Core.Server server = new Grpc.Core.Server(new[] { new
Grpc.Core.ChannelOption(Grpc.Core.ChannelOptions.SoReuseport, 0) })
    {
        Services = { Greeter.BindService(new GreeterService()) },
```

```
        Ports = {new Grpc.Core.ServerPort("0.0.0.0", 8888,
Grpc.Core.ServerCredentials.Insecure) }
    };
    server.Start();
    Console.WriteLine("Server begins to listen");
    Console.ReadLine();
    Console.WriteLine("Server ends!");
    server.ShutdownAsync().Wait();
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

客户端

我们使用如下代码生成所需要的 python 文件：

```
python -m grpc_tools.protoc -I. --python_out=. --pyi_out=. --
grpc_python_out=. Example.proto
```

首先导入需要的依赖：

```
import Example_pb2
import Example_pb2_grpc
import grpc
import time
```

然后建立对应 stub：

```
channel = grpc.insecure_channel("localhost:8888") #
stub = Example_pb2_grpc.GreeterStub(channel)
```

在调用单一 rpc 服务时，我们直接通过本地建立的 stub 调用对应函数即可：

```
response = stub.SayHello(Example_pb2.HelloRequest(name="MyName!"))
print(response.reply)
```

也可以通过异步的方法调用：

```
response_future =
stub.SayHello.future(Example_pb2.HelloRequest(name="MyName!"))
print(response_future.result().reply)
```

在调用服务器流式 rpc 服务时，由于得到的响应是流式的，我们要遍历得到的响应。值得注意的是，如果现有的流中内容已经全部处理完，而服务端还没有结束服务，那么此处线程会被阻塞，直到收到新的消息，或者服务端结束服务；而如果流中积攒了大量的数据，则会按照接收到的顺序来访问这些数据。

```
replies = stub.LotsOfReplies(Example_pb2.HelloRequest(name="MyName!"))
time.sleep(5)
for reply in replies:
    print(reply.reply)
```

在调用客户端流式 rpc 时，我们需要通过 `yield` 关键字来随时间写入内容（有兴趣的同学可以查阅 Python 的相关文档），代码如下：

```
nameList = ["Alice", "Bob", "Cindy"]

def GenerateName():
    for name in nameList:
        yield Example_pb2.HelloRequest(name=name)
        time.sleep(1)

names = GenerateName()
response = stub.LotsOfGreetings(names)
print(response)
```

参考与荐读

由于时间所限，有很多有趣的内容我们没有涉及：

- RPC 的生命周期
- 在 gRPC 中使用安全认证和通讯协议
- ...

略过上述内容不会对我们的教学产生太大影响，感兴趣的同学可以参考以下文档和资源：

- 计算机网络——自顶向下方法
- Stanford CS144
- gRPC 官方文档