一、为什么会有ProtoBuf?

1.1 序列化的概念

序列化和反序列化：

1. 序列化：把对象转换为字节序列的过程称为对象的序列化。
2. 反序列化：把字节序列恢复为对象的过程 称为对象的反序列化。

什么情况下需要序列化：

1. 存储数据：当你想把内存中的对象状态保存到一个文件中或者存到数据库中时。
2. 网络传输：网络直接传输数据，但是无法直接传输对象，所以要在传输前序列化将各种数据转换为二进制序列，传输完成后反序列化将二进制序列转换成对象。

如何实现序列化：

为了实现序列化功能就随之而来产生了如xml、json、 protobuf等解决方法。XML 可读性强但标签冗余、解析慢；JSON 简洁通用、跨语言友好，却无类型校验、处理大数据效率一般；Protobuf 凭借二进制格式，兼具超高压缩率、极快解析速度与强类型约束，是高性能跨语言通信的最优选择，仅需预定义协议文件，调试成本略高于前两者。

二、 ProtoBuf概述

2.1 ProtoBuf是什么？

ProtoBuf也叫Protocol Buffer是google 的一种数据交换的格式，它独立于语言，独立于平台。google 提供了多种语言的实现：java、c#、c++、go 和 python 等，每一种实现都包含了相应语言的编译器以及库文件。

由于它是一种二进制的格式，比使用 xml 、json进行数据交换快许多。可以把它用于分布式应用之间的数据通信或者异构环境下的数据交换。作为一种效率和兼容性都很优秀的二进制数据传输格式，可以用于诸如网络传输、配置文件、数据存储等诸多领域。

简单来讲， ProtoBuf（全称为 Protocol Buffer）是让结构数据序列化的方法，其具有以下特点：

- 语言无关、平台无关：即 ProtoBuf 支持 Java、C++、Python 等多种语言，支持多个平台。
- 高效：即比 XML 更小、更快、更为简单。
- 扩展性、兼容性好：你可以更新数据结构，而不影响和破坏原有的旧程序。

2.2 ProtoBuf的工作流程是什么样的？

在学习protobuf之前，我们需要搞清楚使用protbuf进行数据的序列化主要有哪几个步骤：

1. 确定数据格式，数据可简单可复杂，比如：

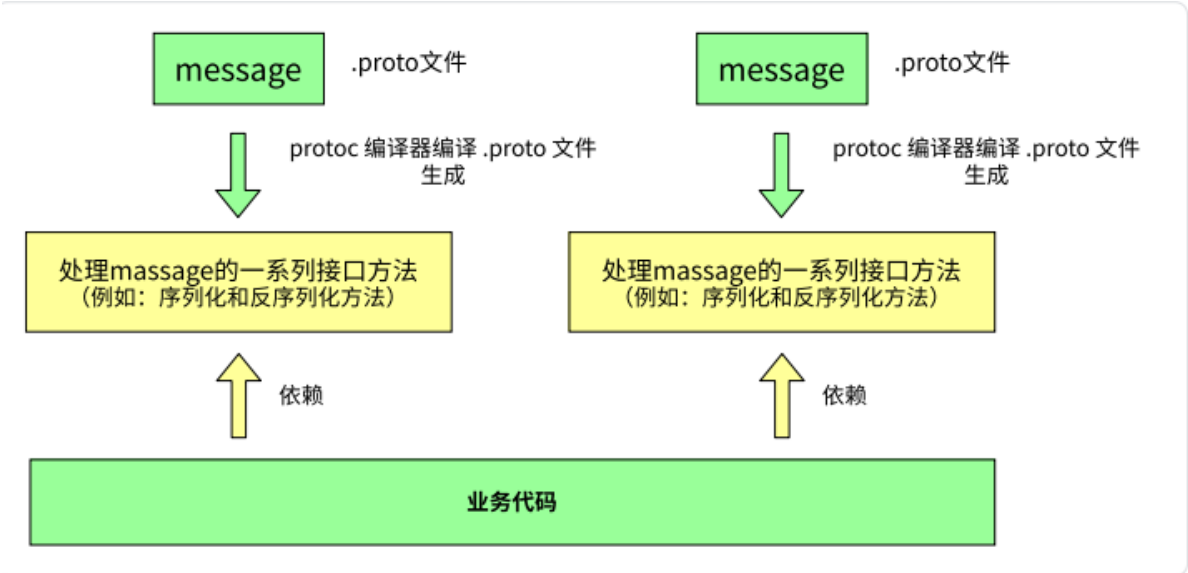
```
// 要序列化的数据
// 第一种: 单一数据类型
int number;

// 第二种: 复合数据类型
struct Person
{
    int id;
    string name;
    string sex;
    int age;
};
```

2. 创建一个新的文件, 文件名随意指定, 文件后缀为 .proto。
3. 根据protobuf的语法, 编辑.proto文件,目的是为了定义结构对象（message）及属性内容。
4. 使用 protoc 编译器编译 .proto 文件，生成一系列接口代码，存放在新生成头文件和源文件中使用 。

- 源文件: xxx.pb.cc –> xxx对应的名字和 .proto文件名相同
- 头文件: xxx.pb.h –> xxx对应的名字和 .proto文件名相同

5. 需要将生成的c++文件添加到项目中, 实现对 .proto 文件中定义的字段进行设置和获取，和对 message 对象进行序列化和反序列化。

protobuf的工作流程如下图所示：



2.3 ProtoBuf基础语法

基本使用

假设我定义了这样一个结构体，现在要基于这个结构体完成数据的序列化，结构体原型如下：

```
struct Person
{
    int id;
    string name;
    string sex; // man woman
    int age;
};
```

接下来，我们需要新建一个文件，给它起个名字，后缀指定为 .proto，在文件的第一行需要指定Protobuf的版本号，有两个版本Protobuf 2 和 Protobuf 3，此处我们使用的是版本3。

```
syntax="proto3";
```

接着需要定义一个消息体，其格式如下：

```
message 名字
{
    // 类中的成员, 格式
    数据类型 成员名字 = 1;
    数据类型 成员名字 = 2;
    数据类型 成员名字 = 3;
        ......
        ......
}
```

- message后面的名字就是生成的类的名字，自己指定一个合适的名字即可;

- 等号后面的编号要从1开始，每个成员都有一个唯一的编号，不能重复，一般连续编号即可;

基于上面的语法，上面结构体对应的.proto文件的内容可以写成这样：

```
// Person.proto
syntax = "proto3";

// 在该文件中对要序列化的结构体进行描述
message Person
{
    int32 id = 1;
    bytes name = 2;
    bytes sex = 3;
    int32 age = 4;
}
```

下表展示了定义于消息体中的标量数据类型，以及编译 .proto 文件之后自动生成的类中与之对应的

字段类型。在这里展示了与 C++ 语言对应的类型。

.proto Type	Notes	C++ Type
double		double
float		float
int32	使用变长编码[1]。负数的编码效率较低——若字段可能为负值，应使用 sint32 代替。	int32
int64	使用变长编码[1]。负数的编码效率较低——若字段可能为负值，应使用 sint64 代替。	int64
uint32	使用变长编码[1]。	uint32
uint64	使用变长编码[1]。	uint64
sint32	使用变长编码[1]。符号整型。负值的编码效率高于常规的 int32 类型。	int32
sint64	使用变长编码[1]。符号整型。负值的编码效率高于常规的 int64 类型。	int64
fixed32	定长 4 字节。若值常大于2^28 则会比 uint32 更高效	uint32
fixed64	定长 8 字节。若值常大于2^56 则会比 uint64 更高效。	uint64
sfixed32	定长 4 字节。	int32
sfixed64	定长 8字节。	int64
bool		bool
string	包含 UTF-8 和 ASCII 编码的字符串，长度不能超过2^32 。	string
bytes	可包含任意的字节序列但长度不能超过 2^32 。	string

> [1] 变长编码是指：经过protobuf 编码后，原本4字节或8字节的数可能会被变为其他字节数。

.proto文件编辑好之后就可以使用protoc工具将其转换为C++文件了。

```
$ protoc -I path .proto文件 --cpp_out=输出路径(存储生成的c++文件)
```

在 protoc 命令中，-I 参数后面可以跟随一个或多个路径，用于告诉编译器在哪些路径下查找导入的文件或依赖的文件，使用绝对路径或相对路径都是没问题的。

如果有多个路径，可以使用多个 -I 参数或在一个参数中使用冒号（:）分隔不同的路径。如果只有一个路径-I 参数可以省略。

例如，protoc -I path1 -I path2 或 protoc -I path1:path2 都表示告诉编译器在 path1 和 path2 路径下查找导入的文件或依赖的文件。

我们可以在.proto文件所在目录执行protoc命令，并生成到当前目录：

```
$ protoc ./Person.proto --cpp_out=.
# 或者使用 -I 参数
$ protoc -I ./ Person.proto --cpp_out=.
```

repeated限定修饰词

在使用Protocol Buffers（Protobuf）中，可以使用repeated关键字作为限定修饰符来表示一个字段可以有多个值，即重复出现的字段。repeated关键字可以用于以下数据类型：基本数据类型、枚举类型和自定义消息类型

比如要序列化的数据中有一个数组，结构如下：

```
// 要序列化的数据
struct Person
{
    int id;
    string name[10];
    string sex;
    int age;
};
```

Protobuf 的消息体就可以写成下面的样子:

```
message Person
{
    int32 id = 1;
    repeated bytes name = 2;
    bytes sex = 3;
    int32 age = 4;
}
```

使用repeated关键字定义的字段在Protobuf序列化和反序列化时会被当作一个集合或数组来处理。这个name可以作为一个动态数组来使用。

枚举

在 Protocol Buffers 中，枚举类型用于定义一组特定的取值。下面定义一个枚举，并将其添加到Person中：

```
// 要序列化的数据
// 枚举
enum Color
{
    Red = 5,    // 可以不给初始值, 默认为0
    Green,
    Yellow,
    Blue
};

// 要序列化的数据
struct Person
{
    int id;
    string name[10];
    string sex;
    int age;
    // 枚举类型
    Color color;
};
```

以下是如何在 Protobuf 中定义和使用枚举类型，其语法如下：

```
enum 名字
{
    元素名 = 0;    // 枚举中第一个原素的值必须为0
    元素名 = 数值;
}
```

枚举元素之间使用分号间隔 ;，并且需要注意一点proto3 中的第一个枚举值必须为0，第一个元素以外的元素值可以随意指定。

上面例子中的数据在.proto文件中可以写成如下格式:

```
// 定义枚举类型
enum Color
{
    Red = 0;
    Green = 3;     // 第一个元素以外的元素值可以随意指定
    Yellow = 6;
    Blue = 9;
}
// 在该文件中对要序列化的结构体进行描述
message Person
{
    int32 id = 1;
    repeated bytes name = 2;
    bytes sex = 3;
    int32 age = 4;
    // 枚举类型
    Color color = 5;
}
```

proto文件的导入

在 Protocol Buffers 中，可以使用import语句在当前.proto中导入其它的.proto文件。这样就可以在一个.proto文件中引用并使用其它文件中定义的消息类型和枚举类型。

语法格式如下：

```
import "要使用的proto文件的名字";
```

假设现在我有一个proto文件Address.proto，里边记录了地址信息：

```
syntax = "proto3";
// 地址信息
message Address
{
    bytes addr = 1;
    bytes number = 2;
}
```

我现需要在上面定义的Person.proto中添加这个人的地址信息，因此就需要用到Address.proto中定义的消息体：

```
syntax = "proto3";
// 使用另外一个proto文件中的数类型, 需要导入这个文件
import "Address.proto";

// 在该文件中对要序列化的结构体进行描述
// 定义枚举类型
enum Color
{
    Red = 0;
    Green = 3;      // 第一个元素以外的元素值可以随意指定
    Yellow = 6;
    Blue = 9;
}
// 在该文件中对要序列化的结构体进行描述
message Person
{
    int32 id = 1;
    repeated bytes name = 2;
    bytes sex = 3;
    int32 age = 4;
    // 枚举类型
    Color color = 5;
    // 添加地址信息, 使用的是外部proto文件中定义的数据类型
    Address addr = 6;
}
```

- import语句中指定的文件路径可以是相对路径或绝对路径。如果文件在相同的目录中，只需指定文件名即可。
- 导入的文件将会在编译时与当前文件一起被编译。
- 导入的文件也可以继续导入其他文件，形成一个文件依赖的层次结构。

包（package)

在 Protobuf 中，可以使用package关键字来定义一个消息所属的包（package）。包是用于组织和命名消息类型的一种机制，类似于命名空间的概念

在一个.proto文件中，可以通过在顶层使用package关键字来定义包：

```
syntax = "proto3";
package mypackage;
message MyMessage
{
  // ...
}
```

在这个示例中，我们使用package关键字将MyMessage消息类型定义在名为mypackage的包中。包名作为一个标识符来命名，可以使用任何有效的标识符，按惯例使用小写字母和下划线。

使用包可以避免不同.proto文件中的消息类型名称冲突，同时也可以更好地组织和管理大型项目中的消息定义。可以将消息类型的名称定义在特定的包中，并使用限定名来引用这些类型。

下面有两个proto文件，分别给他们添加一个package：

- proto文件- Address.proto

```
syntax = "proto3";
// 添加命名空间 myAddress
package myAddress;

// 地址信息, 这个Address类属于命名空间: myAddress
message Address
{
    bytes addr = 1;
    bytes number = 2;
}
```

- proto文件 - Person.proto

```
syntax = "proto3";
// 使用另外一个proto文件中的数类型, 需要导入这个文件
import "Address.proto";
// 指定命名空间 ErBing
package ErBing;

// 以下的类 Person 和枚举 Color 都属于命名空间 ErBing
// 在该文件中对要序列化的结构体进行描述
// 定义枚举类型
enum Color
{
    Red = 0;
    Green = 3;      // 第一个元素以外的元素值可以随意指定
    Yellow = 6;
    Blue = 9;
}
// 在该文件中对要序列化的结构体进行描述
message Person
{
    int32 id = 1;
    repeated bytes name = 2;
    bytes sex = 3;
    int32 age = 4;
    // 枚举类型
    Color color = 5;
    // 添加地址信息, 使用的是外部proto文件中定义的数据类型
    // 如果这个外边类型属于某个命名空间, 语法格式:
```

```
    // 命名空间的名字.类名 变量名=编号;
    myAddress.Address addr = 6;
}
```

2.4 序列化与反序列化

**.ph.h头文件

通过protoc 命令对.proto文件的转换，得到的头文件中有一个类，这个类的名字和.proto文件中message关键字后边指定的名字相同，.proto文件中message消息体的成员就是生成的类的私有成员。

那么如何访问生成的类的私有成员呢？ 可以调用生成的类提供的公共成员函数，这些函数有如下规律：

- 清空(初始化) 私有成员的值: clear_变量名()
- 获取类私有成员的值: 变量名()
- 给私有成员进行值的设置: set_变量名(参数)
- 得到类私有成员的地址, 通过这块地址读/写当前私有成员变量的值: mutable_变量名()
- 如果这个变量是数组类型:
    - 数组中元素的个数: 变量名_size()
    - 添加一块内存, 存储新的元素数据: add 变量名() 、add 变量名(参数)

序列化

序列化是指将数据结构或对象转换为可以在储存或传输中使用的二进制格式的过程。在计算机科学中，序列化通常用于将内存中的对象持久化存储到磁盘上，或者在分布式系统中进行数据传输和通信。

Protobuf 中为我们提供了相关的用于数据序列化的 API，如下所示：

```
// 头文件目录: google\protobuf\message_lite.h
// --- 将序列化的数据 数据保存到内存中
```

```
// 将类对象中的数据序列化为字符串, c++ 风格的字符串, 参数是一个传出参数
bool SerializeToString(std::string* output) const;
// 将类对象中的数据序列化为字符串, c 风格的字符串, 参数 data 是一个传出参数
bool SerializeToArray(void* data, int size) const;

// ------ 写磁盘文件, 只需要调用这个函数, 数据自动被写入到磁盘文件中
// -- 需要提供流对象/文件描述符关联一个磁盘文件
// 将数据序列化写入到磁盘文件中, c++ 风格
// ostream 子类 ofstream -> 写文件
bool SerializeToOstream(std::ostream* output) const;
// 将数据序列化写入到磁盘文件中, c 风格
bool SerializeToFileDescriptor(int file_descriptor) const;
```

反序列化

反序列化是指将序列化后的二进制数据重新转换为原始的数据结构或对象的过程。通过反序列化，我们可以将之前序列化的数据重新还原为其原始的形式，以便进行数据的读取、操作和处理。

Protobuf 中为我们提供了相关的用于数据反序列化的 API，如下所示：

```
// 头文件目录: google\protobuf\message_lite.h
bool ParseFromString(const std::string& data) ;
bool ParseFromArray(const void* data, int size);
// istream -> 子类 ifstream -> 读操作
// wo ri
// w->写 o: ofstream , r->读 i: ifstream
bool ParseFromIstream(std::istream* input);
bool ParseFromFileDescriptor(int file_descriptor);
```

2.5 示例程序

.proto文件

1. Address.proto

```
syntax = "proto3";
package myAddress;
message Address
{
  int32 num = 1;
  bytes addr = 2;
}
```

2. Person.proto

```
syntax = "proto3";
import "Address.proto";
package Erbing;
enum Color
{
    Red = 0;
    Green = 5;
    Yellow = 6;
    Blue = 9;
}

message Person
{
    int32 id = 1;
    repeated bytes name = 2;
    bytes sex = 3;
    int32 age = 4;
    myAddress.Address addr = 5;
    Color color = 6;
}
```

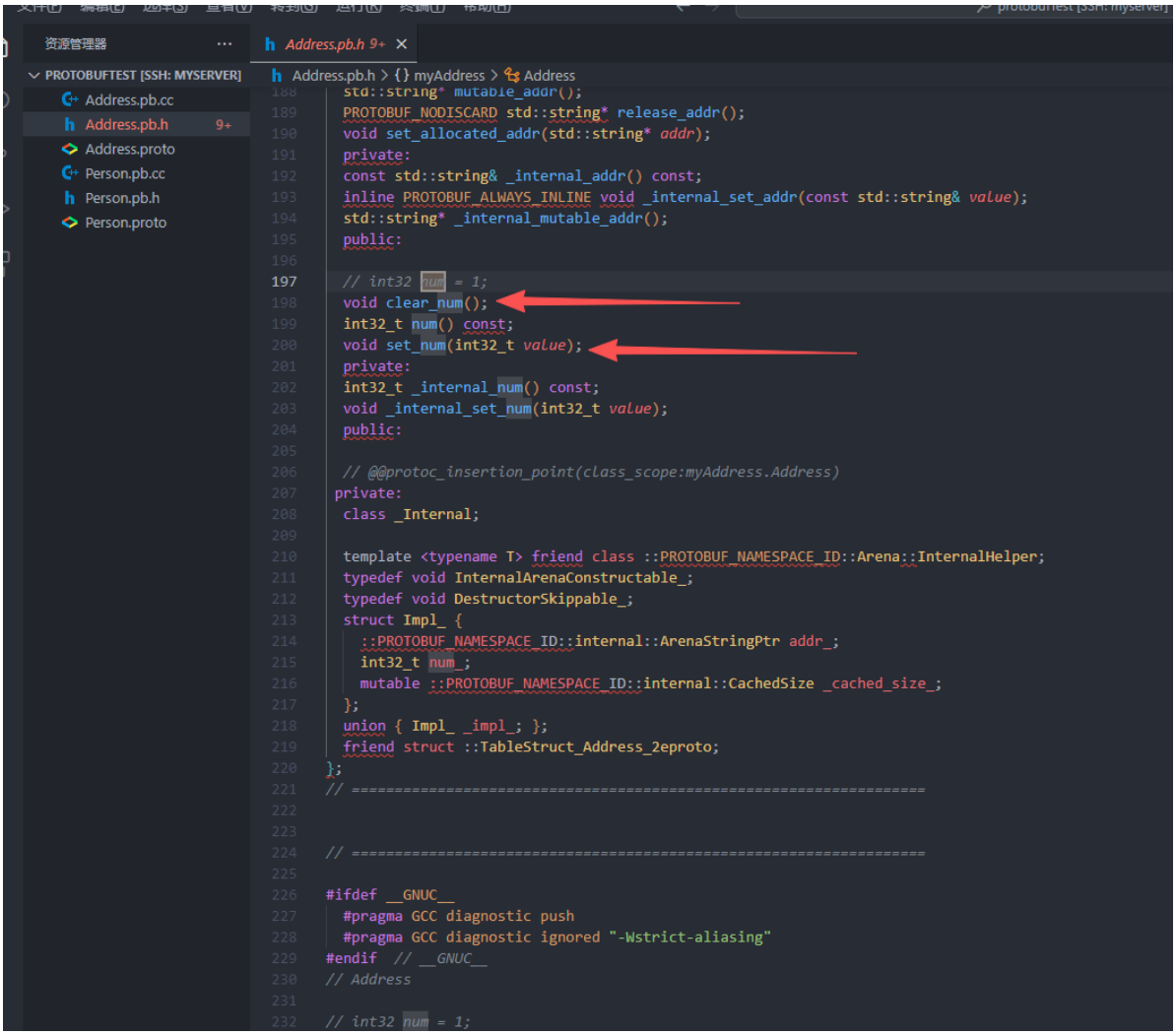
进行编译

```
[chen@lavm-ou94mqwtye protobufTest]$ ls
Address.proto  Person.proto
[chen@lavm-ou94mqwtye protobufTest]$ protoc -I ./ Person.proto Address.proto --cpp_out=./
[chen@lavm-ou94mqwtye protobufTest]$ ls
Address.pb.cc  Address.pb.h  Address.proto  Person.pb.cc  Person.pb.h  Person.proto
[chen@lavm-ou94mqwtye protobufTest]$
```

编译命令

编辑器会针对于每个 .proto 文件生成 .h 和 .cc 文件，分别用来存放类的声明与类的实现。

Address.ph.h部分代码如图所示：



序列化与反序列化的使用

1. 创建测试代码MyTest.h

```
class MyTest
{
public:
    void test();
};
```

2. 创建测试代码MyTest.cpp

```
#include "MyTest.h"
```

```cpp
#include "Person.pb.h"

//using namespace myAddress;
//using namespace Erbing;

void MyTest::test()
{
    // 序列化
    Erbing::Person p;
    p.set_id(10);
    p.set_age(32);
    p.set_sex("man");

    p.add_name();
    p.set_name(0,"路飞");
    p.add_name("艾斯");
    p.add_name("萨博");
    p.mutable_addr()->set_addr("北京市长安区天安门");
    p.mutable_addr()->set_num(1001);
    p.set_color(Erbing::Color::Blue);

    // 序列化对象 p, 最终得到一个字符串
    std::string output;
    p.SerializeToString(&output);

    // 反序列化数据
    Erbing::Person pp;
    pp.ParseFromString(output);
    std::cout << pp.id()  << ", " << pp.sex() << ", " << pp.age() << std::endl;
    std::cout << pp.addr().addr() << ", " << pp.addr().num() << std::endl;
    int size = pp.name_size();
    for(int i=0; i<size; ++i)
    {
        std::cout << pp.name(i) << std::endl;
    }
    std::cout << pp.color() << std::endl;
}
```

3. main.cpp

```cpp
#include "MyTest.h"
int main(int argc, char* argv[])
{
    MyTest t;
    t.test();
    return 0;
}
```

4. 代码书写完成后，编译 main.cpp，生成可执行程序 TestProtoBuf

```
g++ main.cpp MyTest.cpp Person.pb.cc Address.pb.cc -o TestProtoBuf -std=c++11 -lprotobuf
```

5. 执行生成的可执行程序即可观察到测试结果