gRPC资料

[protobuf](https://link.zhihu.com/?target=https://github.com/protocolbuffers/protobuf)是一款用C++开发的跨语言、二进制编码的数据序列化协议，以超高的压缩率著称。它和早期的RPC方案一样，需要双方维护一个协议约束文件，以.proto结尾，使用proto命令对文件进行解析，会生成对应的Stub程序，客户端和服务端都需要保存这份Stub程序用来进行编解码。对于这种协议文件导致的升级困难问题，protobuf 3 中定义的字段默认都是可选的(可以不传)，在接口升级时，部分客户端不需要升级自己的Stub程序。

// 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**

作为一个以跨语言为目标的序列化方案，protobuf能做到一份.proto文件走天下，不管什么语言，都能以同一份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golang里是struct，Ruby里是class，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

由于message是自己定义的，而且有特定的类型等，一套通用的编解码代码是不行的(类似json)，特定的proto需要对应的方法，对message编解码，不同的message编解码策略还不一样。

这些代码用手写是不行的，protobuf对此的解决方案是，提供一个统一的protoc工具，这个一个C++”翻译“工具，可以通过proto文件，生成某特定语言的中间代码，实现上面说的两个能力。也就是说，**protobuf通过自动化编译器的方式统一提供了这种能力，避免人肉写**。

// 依赖目录 生成golang中间代码 对应proto文件地址

protoc -I=$SRC\_DIR --go\_out=$DST\_DIR $SRC\_DIR/XXX.proto

protoc -I=$SRC\_DIR --java\_out=$DST\_DIR $SRC\_DIR/XXX.proto // 生成java中间代码

执行结果是对应语言的中间代码，以golang为例，会生成一个xx.pb.go文件，里面就是对应rpc、message的结构体，以及编解码的function

**直观对比**

为什么选择protobuf，而不是普及最广的json作为编码方案？ 可以做一个直观对比，以上文proto中的Response为例，一次输出json的结果是：

"{\"ids\":[123,456],\"info\":{\"is\_man\":true,\"age\":20},\"values\":{\"110\":{\"is\_man\":false,\"age\":18}}}"

所有内容被打包成了一个字符串，里面包含字段名、value，当Reponse很大时，体积消耗很大，浪费主要在三个方面：

* 字段名，例如上面的“ids”、“info”等，如果json体大，则重复会更多
* 数字用字符串表达了，例如123数字变成了“123”，这在编码后体积由一个字节变成三字节
* 类型字符，如[ 、 ]、{ 、}

但如果是protobuf呢？ 输出是一段人眼无法理解的二进制串，里面：

* 去掉了字段名，转而以字段标号替代，通过标号可以在proto中找到字段名
* 没有类型字符等
* 用二进制表达内容，不会将数字转成字符串
* 字段值按顺序依次排列

这使得protobuf的编码结果体积，通常是json编码后的十分之一以下。同时由于排列简单，其解析算法的时空复杂度远小于json，对cpu消耗也小很多。这使得protobuf在大数据量、高频率的数据交互场景下，远胜于json，被大规模分布式RPC场景广泛使用。

protobuf将数据类型做了**分类(Wire Type)**，并提供不同的编解码方式：



T - V 编码举例：

message request {

int64 user\_id = 1; // tagNum = 1, wireType = 0,

}

假设 value为 2, 则编码出的T-V为：

+-----+---+-----------------+

|00001|000|00000010|

+-----+---+-----------------+

tagNum type data

假设 value为 300, 则编码出的T-V为：

第一个字节 第二 第三

+-----+---+-----------------------+

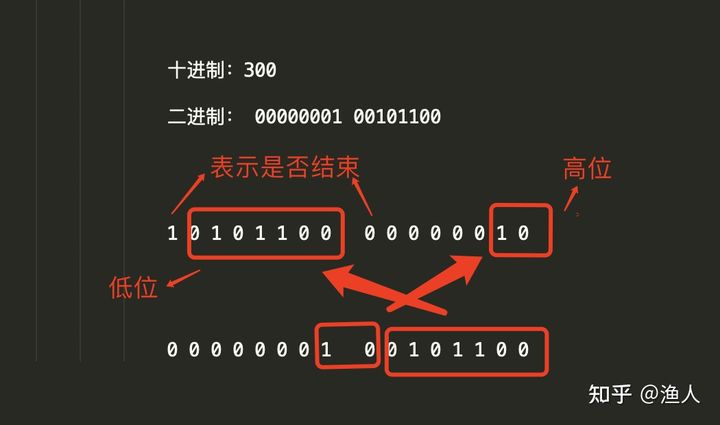
|00001|000| 10101100 00000010| 下个T-V

+-----+---+-----------------------+

tagNum type data

Tag高位=0： 一个byte

data的第一个字节最高位为1，说明下一个字节还要继续读



**T - L - V**

T - L - V 就是在上面的基础上增加了length，用来表达变长的内容：



由于是变长，例如数组、嵌套对象，有多个value，此时就无法通过最高位是否是1，来表示该字段是否解析完毕，必须要在value前增加一个length，其他都和T-V一样。

接下来我们学习两个点：

* 数组如何表达
* 嵌套对象如何表达

数组的表达其实比较简单，就是同一个T不断的重复(tagNum和wireType不变)，解析对应的V就行，然后组成一个数组：



嵌套对象稍微复杂点，每个value都能找到一个message去套，逐层解就行了：

来源于[4]

嵌套对象编码举例：

message request {

User user = 1; // tagNum = 1, wireType = 2,

}

message User {

int64 user\_id = 1; // tagNum = 1

}

假设 request = { user\_id: 2}, 则编码出的T-L-V为：

Tag length value

Tag value

+---------+--------+---------+---------

|00001010 |00000010|00001000|00000010|

1<<3 | 2 2 byte 1<<3 | 0 2

通过解request 知道第一个字段是User，再拿到第一个字段的value去解User，

知道User第一个字段是int64，解析出data为2。 一个嵌套对象即解析完毕

stream模式，其实就是gRPC从协议层支持了，在一次长请求中，分批地处理小量数据，达到多次请求的效果，像流水一样可以延绵不绝，直到某一方终止

服务端：

gRPC 服务在 Startup.cs 中使用 AddGrpc 进行配置。

选项 默认值 描述  
MaxSendMessageSize null 可以从服务器发送的最大消息大小（以字节为单位）。 尝试发送超过配置的最大消息大小的消息会导致异常。 设置为 null时，消息的大小不受限制。  
MaxReceiveMessageSize 4 MB 可以由服务器接收的最大消息大小（以字节为单位）。 如果服务器收到的消息超过此限制，则会引发异常。 增大此值可使服务器接收更大的消息，但可能会对内存消耗产生负面影响。 设置为 null时，消息的大小不受限制。  
EnableDetailedErrors false 如果为 true，则当服务方法中引发异常时，会将详细异常消息返回到客户端。 默认值为 false。 将 EnableDetailedErrors 设置为 true 可能会泄漏敏感信息。  
CompressionProviders gzip 用于压缩和解压缩消息的压缩提供程序的集合。 可以创建自定义压缩提供程序并将其添加到集合中。 默认已配置提供程序支持 gzip 压缩。  
ResponseCompressionAlgorithm null 压缩算法用于压缩从服务器发送的消息。 该算法必须与 CompressionProviders 中的压缩提供程序匹配。 若要使算法可压缩响应，客户端必须通过在 grpc-accept-encoding 标头中进行发送来指示它支持算法。  
ResponseCompressionLevel null 用于压缩从服务器发送的消息的压缩级别。  
Interceptors None 随每个 gRPC 调用一起运行的侦听器的集合。 侦听器按注册顺序运行。 全局配置的侦听器在为单个服务配置的侦听器之前运行。 有关 gRPC 侦听器的详细信息，请参阅 gRPC 侦听器与中间件。  
IgnoreUnknownServices false 如果为 true，则对未知服务和方法的调用不会返回 UNIMPLEMENTED 状态，并且请求会传递到 ASP.NET Core 中的下一个注册中间件。  
　　客户端：

gRPC 客户端配置在 GrpcChannelOptions 中进行设置。 下表描述了用于配置 gRPC 通道的选项

选项 默认值 描述  
HttpHandler 新实例 用于进行 gRPC 调用的 HttpMessageHandler。 可以将客户端设置为配置自定义 HttpClientHandler，或将附加处理程序添加到 gRPC 调用的 HTTP 管道。 如果未指定 HttpMessageHandler，则会通过自动处置为通道创建新 HttpClientHandler 实例。  
HttpClient null 用于进行 gRPC 调用的 HttpClient。 此设置是 HttpHandler 的替代项。  
DisposeHttpClient false 如果设置为 true 且指定了 HttpMessageHandler 或 HttpClient，则在处置 GrpcChannel 时，将分别处置 HttpHandler 或 HttpClient。  
LoggerFactory null 客户端用于记录有关 gRPC 调用的信息的 LoggerFactory。 可以通过依赖项注入来解析或使用 LoggerFactory.Create 来创建 LoggerFactory 实例。 有关配置日志记录的示例，请参阅 .NET 上 gRPC 中的日志记录和诊断。  
MaxSendMessageSize null 可以从客户端发送的最大消息大小（以字节为单位）。 尝试发送超过配置的最大消息大小的消息会导致异常。 设置为 null时，消息的大小不受限制。  
MaxReceiveMessageSize 4 MB 可以由客户端接收的最大消息大小（以字节为单位）。 如果客户端收到的消息超过此限制，则会引发异常。 增大此值可使客户端接收更大的消息，但可能会对内存消耗产生负面影响。 设置为 null时，消息的大小不受限制。  
Credentials null 一个 ChannelCredentials 实例。 凭据用于将身份验证元数据添加到 gRPC 调用。  
CompressionProviders gzip 用于压缩和解压缩消息的压缩提供程序的集合。 可以创建自定义压缩提供程序并将其添加到集合中。 默认已配置提供程序支持 gzip 压缩。  
ThrowOperationCanceledOnCancellation false 如果设置为 true，则在取消调用或超过其截止时间时，客户端将引发 OperationCanceledException。  
MaxRetryAttempts 5 最大重试次数。 该值限制服务配置中指定的任何重试和 hedging 尝试值。单独设置该值不会启用重试。 重试在服务配置中启用，可以使用 ServiceConfig 来启用。 null 值会删除最大重试次数限制。 有关重试的更多详细信息，请参阅“暂时性故障处理与 gRPC 重试”。  
MaxRetryBufferSize 16 MB 在重试或 hedging 调用时，可用于存储发送的消息的最大缓冲区大小（以字节为单位）。 如果超出了缓冲区限制，则不会再进行重试，并且仅保留一个 hedging 调用，其他 hedging 调用将会取消。 此限制将应用于通过通道进行的所有调用。 值 null 移除最大重试缓冲区大小限制。  
MaxRetryBufferPerCallSize 1 MB 在重试或 hedging 调用时，可用于存储发送的消息的最大缓冲区大小（以字节为单位）。 如果超出了缓冲区限制，则不会再进行重试，并且仅保留一个 hedging 调用，其他 hedging 调用将会取消。 此限制将应用于一个调用。 值 null 移除每个调用的最大重试缓冲区大小限制。  
ServiceConfig null gRPC 通道的服务配置。 服务配置可以用于配置 gRPC 重试。