pbLearning

author: Zilin Xu
date: 2024.6.11

create project structure

run following command

```
mkdir -p pbLearning/proto
cd pbLearning
touch go.mod go.sum main.go proto/person.proto
```

edit `person.proto`

```
syntax="proto3";

option go_package = "pbLearning/proto";

message Person {
    string name = 1;
    int32 age = 2;
}
```

generate `person.pb.go`

```
protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=. --go-grpc_opt=paths=source_relative proto/person.proto
```

edit `main.go`

```
package main

import (
    "fmt"
    "log"

    pb "pbLearning/proto"

    "google.golang.org/protobuf/proto"
)

func main() {
    fmt.Println("hello")
```

```
    bruce := &pb.Person{
        Name: "bruce",
        Age:  23,
    }

    data, err := proto.Marshal(bruce)
    if err != nil {
        log.Fatal("Marshalling error:", err)
    }
    fmt.Println(data)

    newBruce := &pb.Person{}
    err = proto.Unmarshal(data, newBruce)
    if err != nil {
        log.Fatal("Unmarshalling error:", err)
    }
    fmt.Println(newBruce)
}
```

run `go run main.go`

```
➜  pbLearning git:(main) ✗ go run main.go
hello
[10 5 98 114 117 99 101 16 23]
name:"bruce"  age:23
```

在你的`main.go`程序中，`proto.Marshal`函数将`Person`结构体序列化为字节数组。这种序列化过程将结构体的字段转换为一系列字节，以便在网络上传输或存储在文件中。输出的字节数组是序列化后的二进制数据。

你的代码中序列化`Person`结构体的结果：

```
bruce := &pb.Person{
    Name: "bruce",
    Age:  23,
}

data, err := proto.Marshal(bruce)
```

生成了以下字节数组：

```
[10 5 98 114 117 99 101 16 23]
```

让我们逐个字节解释这些数据的含义：

1. `10`:

- 这是字段号1（Name）的标记（tag），并且字段类型为length-delimited（长度定界的）。10的二进制表示是`00001010`，其中前3位`000`表示类型（长度定界的，类型码为2），后4位`1010`表示字段号1。

2. `5`:
   - 这是`Name`字段的长度，表示接下来的5个字节是`Name`字段的值。
3. `98 114 117 99 101`:
   - 这些是`Name`字段的值，对应ASCII字符`b`、`r`、`u`、`c`、`e`。分别是字节`98`、`114`、`117`、`99`和`101`。
4. `16`:
   - 这是字段号2（Age）的标记，表示类型为varint（变长整数）。16的二进制表示是`00010000`，其中前3位`000`表示类型（varint，类型码为0），后4位`1000`表示字段号2。
5. `23`:
   - 这是`Age`字段的值23。对于varint类型，23本身就是它的值，没有额外的长度字节。

将这些字节组合在一起，形成了完整的序列化结果：

```
10          // field 1, length-delimited
5           // length of the name field
98 114 117 99 101  // "bruce"
16          // field 2, varint
23          // age: 23
```

这解释了字节数组`[10 5 98 114 117 99 101 16 23]`表示的`Person`结构体。这个数组可以通过`proto.Unmarshal`反序列化回原始的`Person`结构体。

update `.pb` functions

```
syntax="proto3";

option go_package = "pbLearning/proto";

message Person {
    string name = 1;
    int32 age = 2;
    Socialfollowers socialfollowers = 3;
}

message Socialfollowers {
    int32 youtube = 1;
    int32 twitter = 2;
}
```

run command to update `.pb.go`

```
protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=. --go-grpc_opt=paths=source_relative proto/person.proto
```

update `main.go`

```
    bruce := &pb.Person{
        Name: "bruce",
        Age:  23,
        Socialfollowers: &pb.Socialfollowers{
            Youtube: 100,
            Twitter: 200,
        },
    }

    fmt.Println(newBruce.Socialfollowers.GetTwitter())
    fmt.Println(newBruce.Socialfollowers.GetYoutube())
```

run the program

```
➜  pbLearning git:(main) ✗ go run main.go
hello
[10 5 98 114 117 99 101 16 23 26 5 8 100 16 200 1]
23
bruce
200
100
```

使用Protocol Buffers（简称protobuf或pb）有许多优势，特别是在分布式系统、网络通信和存储结构化数据时。以下是protobuf的主要优点及其应用场景：

1. 高效的序列化和反序列化

- **紧凑的二进制格式**：protobuf使用二进制格式，比JSON或XML等文本格式更紧凑，节省带宽和存储空间。
- **速度快**：序列化和反序列化速度非常快，适用于高性能要求的应用程序。

2. 跨语言支持

- protobuf支持多种编程语言，如Go、Java、Python、C++、Ruby等。这使得不同语言编写的服务和应用程序可以轻松互操作。

3. 明确的结构

- **定义明确**：通过`.proto`文件定义数据结构，生成代码具有强类型支持，减少了错误。
- **向后兼容**：可以添加新的字段而不破坏现有字段的解析，使得系统更易于演进。

4. 适用的场景

- **分布式系统**：服务之间需要高效地传递消息。
- **网络通信**：客户端和服务器之间的数据交换。
- **持久化存储**：高效地存储和检索结构化数据。

实际例子

假设你有一个微服务架构，多个服务需要相互通信。使用protobuf，可以定义一个数据结构，并在不同服务间共享。举例来说：

1. **定义数据结构**：

   `person.proto`：

   ```
   syntax = "proto3";
   option go_package = "pbLearning/proto";

   message Person {
       string name = 1;
       int32 age = 2;
   }
   ```

2. **生成代码**：

   运行`protoc`命令生成不同语言的代码。

3. **服务间通信**：

   使用生成的代码在服务间传递数据。比如，一个服务序列化数据：

   ```
   data, err := proto.Marshal(person)
   ```

   另一个服务反序列化数据：

   ```
   var person pb.Person
   err := proto.Unmarshal(data, &person)
   ```

为什么使用protobuf?

- **高效传输**：减少了网络传输的负担，适合带宽有限的场景。
- **强类型支持**：减少了数据格式错误的发生。
- **灵活性和扩展性**：可以在不破坏现有数据结构的情况下添加新字段。

总的来说，protobuf在需要高效、可扩展和跨语言的数据传输和存储场景中非常有用。它帮助开发者定义清晰的数据结构，并确保这些数据结构在不同系统和语言之间的一致性和互操作性。

Uber Protobuf Style Guide v2 是 Uber 发布的一套关于如何编写 Protocol Buffers 文件的规范。它旨在提高代码的可读性、一致性和可维护性。以下是该指南的一些关键点和中文解释：

1. 文件命名和结构

- **文件名**：所有的 proto 文件应使用小写字母和下划线命名（例如：`user_profile.proto`）。
- **包结构**：proto 文件应位于其对应的包目录下。例如，包 `uber.foo.bar` 的 proto 文件应位于 `uber/foo/bar` 目录中。

2. 语法版本

- **语法版本**：使用 `proto3` 语法版本。文件开头应该包含 `syntax = "proto3";`。

3. 包和选项

- **包声明**：必须使用包声明，格式为 `package <package_name>;`，包名应使用小写字母和点分隔。
- **go_package 选项**：对于生成的 Go 代码，应使用 `option go_package = "<import_path>";`。

4. 消息定义

- **消息命名**：消息类型名应使用大写字母开头的驼峰命名法（例如：`UserProfile`）。
- **字段命名**：字段名应使用小写字母和下划线命名法（例如：`user_id`）。
- **字段排序**：字段应按重要性或逻辑顺序排列，而不是随机顺序。

5. 枚举

- **枚举命名**：枚举类型名应使用大写字母开头的驼峰命名法，枚举值应使用全大写字母和下划线命名法（例如：`USER_STATUS_ACTIVE`）。
- **零值枚举**：第一个枚举值必须是零值，以表示默认状态。

6. 文档注释

- **注释**：所有的消息、字段、枚举及其值都应有注释。注释应使用 `//` 或 `/** */` 形式，简洁明了地解释其含义和用途。

7. 服务定义

- **服务命名**：服务名应使用大写字母开头的驼峰命名法（例如：`UserService`）。
- **方法命名**：方法名应使用大写字母开头的驼峰命名法（例如：`GetUserProfile`）。

8. 版本控制

- **版本控制**：为了兼容性和版本控制，建议在包名中包含版本信息（例如：`package uber.foo.v1;`）。

9. 其他规范

- **避免使用保留字**：避免使用 Proto 和目标语言中的保留字。
- **常量定义**：尽量避免在 proto 文件中定义常量，使用枚举代替。

示例

```
syntax = "proto3";

package uber.foo.v1;

option go_package = "github.com/uber/foo/v1";

// 用户状态枚举
enum UserStatus {
    USER_STATUS_UNKNOWN = 0; // 默认未知状态
    USER_STATUS_ACTIVE = 1;  // 用户激活状态
    USER_STATUS_INACTIVE = 2; // 用户非激活状态
}

// 用户资料消息
message UserProfile {
    string user_id = 1; // 用户唯一标识符
    string name = 2;    // 用户姓名
    UserStatus status = 3; // 用户状态
}

// 用户服务定义
service UserService {
    // 获取用户资料
    rpc GetUserProfile(GetUserProfileRequest) returns
(GetUserProfileResponse);
}

// 获取用户资料请求
message GetUserProfileRequest {
    string user_id = 1; // 用户唯一标识符
}

// 获取用户资料响应
message GetUserProfileResponse {
    UserProfile profile = 1; // 用户资料
}
```