Protobuf使用手册

2016-06-08

c++ cmake protobuf

—— Protobuf协议文件定义 ——

选择版本

- `syntax` 声明可以选择protobuf的编译器版本(v2和v3)
  - `syntax="proto2";` 选择2版本，各个字段必须明确标注编号以确定序列化后二进制数据字段的位置
  - `syntax="proto3";` 选择3版本,没有强制使用字段编号

字段修饰符

- `required`
  - 对于required的字段而言，编号初值是必须要提供的，否则字段的便是未初始化的
  - 对于修饰符为required的字段，序列化的时候必须给予初始化,否则程序运行会异常
- `optional`
  - 对于optional的字段而言，如果未进行初始化，那么一个默认值将赋予该字段编号
  - 也可以指定默认值,如示例所示.
- `repeated`
  - 对于repeated的字段而言，该字段可以重复多个，即每个编码单元可能有多个该字段
  - 在高级语言里面，我们可以通过数组来实现，而在proto定义文件中可以使用repeated来修饰，从而达到相同目的。
  - 当然，出现0次也是包含在内的。

字段类型

proto Type	C++ Type	Notes
int32	int32	有符号32位整型数,非固定长度编码,编码效率比sint32低
int64	int64	有符号64位整型数,非固定长度编码
uint32	uint32	无符号32位整型数,非固定长度编码
uint64	uint64	无符号64位整型数,非固定长度编码
sint32	int32	有符号32位整型数,非固定长度编码,效率较高
sint64	int64	有符号64位整型数,非固定长度编码,效率较高
fixed32	uint32	无符号,固定4字节编码,数据大于2^28时,效率比uint32高
fixed64	uint64	无符号,固定8字节编码,数据大于2^56时,效率比uint32高
sfixed32	int32	有符号,固定4字节编码
sfixed64	int64	有符号,固定8字节编码
bool	bool	布尔值
float	float	浮点数
double	double	浮点数
string	string	必须为UTF-8编码或者7bit ASCII字符串
bytes	string	字节数组

字段类型对应二进制类型

字段类型	二进制类型	二进制编码值
int32,int64,uint32,uint64,sint32,sint64,bool,enum	Varint(可变长度int)	0
fixed64,sfixed64,double	64bit固定长度	1
string,bytes,inner messages(内部嵌套),packaed repeated fields(repeated字段)	Length-delimited	2
groups(deprecated)	Start group	3
groups(deprecated)	Endd group	4
fixed32,sfixed32,float	32bit固定长度	5

数据编码原则

1.Varints编码规则

- protobuf编码基础是Varints,Varints是将一个整数序列化为一个或多个Bytes的方法,越小的整数，使用的Bytes越少
  - 每个byte最高位(msb)是标志位,0表示是最后一个byte,1表示该字段值还有后续byte
  - 每个byte低7位存放数值
  - Varints使用Little Endian(小端)字节序
- 转换示例

Code

```
dec(300)
=> bin(00000001 00101100)
=> Little(00101100 00000001)
=> Encode(10101100 00000010)
```

2.消息编码规则

- message都是以一组或多组key-value对组成,key和value分别采用不同的编码方式
- 序列化时，将message中所有key-value序列化成二进制字节流。反序列化时，解析出所有key-value对，

如果遇到无法识别的类型，则直接跳过。这种机制保证了旧有的编/解码在协议添加新的字段时，依旧可以正常工作

- key由两部分组成，一部分是在定义消息时对字段的编号（field_num），另一部分是字段类型（wire_type，编号最大不超过`536870911`.
- key编码方式`filed_num<<3|wire_type`，编码后的二进制长度是`变长`的
- varint(wire_type=0)编码规则:
  - int32,int64直接按照varint方法来编码,因此-1,-2这种负数由于补码数表示有很多1,所占的byte也比较多
  - sint32,sint64采用`zigzag`方法来避免上述问题
  - 首先采用Zigzag方法，将正数、0和负数映射到无符号数上
  - 再采用varint编码方法
  - `zigzag`映射规则

Code

```
Zigzag(n) = (n<<1)^(n>>31) ,n为sint32时
Zigzag(n) = (n<<1)^(n>>63) ,n为sint64时
```

- 映射值表

Original(原始值)	(编码后的值)EncodeAs
0	0
-1	1
1	2
-2	3
2	4
-3	5
2147483647	4294967294
-2147483648	4294967295

- 64bit(wire_type=1)和32bit(wire_type=5)编码是在key后跟上Little Endian字节的数值
-

string,bytes都属于length-delimited编码,length-delimited(wire_type=2)的编码方式：

key+length+content

- key的编码方式是统一的
- length采用varints编码方式
- content就是由length指定的长度的Bytes
-

完整示例

-

type.proto

Code

```
message IntType {
        optional int32 a_i32 = 1;
        optional int64 b_i64 = 2;
        optional sint32 c_s32 = 3;
        optional sint64 d_s64 = 4;
        optional sfixed32 e_sf32 = 5;
        optional sfixed64 f_sf64 = 6;
}

message UIntType {
        optional uint32 a_u32 = 1;
        optional uint64 b_u64 = 2;
        optional fixed32 c_f32 = 3;
        optional fixed64 d_f64 = 4;
}

message FType{
        optional float a_f = 1;
        optional double b_d = 2;
        optional bool c_b = 3;
}

message BSType{
        optional string a_s = 1;
        repeated bytes b_bs = 2;
}
```

初始值

Code

```
//int
type::IntType it;
it.set_a_i32(-1);
it.set_b_i64(-1);
it.set_c_s32(-1);
it.set_d_s64(-1);
it.set_e_sf32(-1);
it.set_f_sf64(-1);

//uint
type::UIntType ut;
ut.set_a_u32(1);
ut.set_b_u64(2);
ut.set_c_f32(1);
ut.set_d_f64(2);

//float
type::FType ft;
ft.set_a_f(0.1);
ft.set_b_d(0.2);
ft.set_c_b(false);

//bytes
type::BSType bt;
bt.set_a_s("hello,world.");
```

```
26  bt.add_b_bs("h");
27  bt.add_b_bs("0x1");
28  bt.add_b_bs("d");
29  bt.add_b_bs("ILV");
30  bt.add_b_bs("0xf");
```

二进制表示

IntType

类型	key	EncodedAs key	value	EncodedAs value	EncodedAs String	Notes
:-:	:-:	:————:	:—:	:————-:	:—:	:—:
int32	1	`1<<3 OR 0 = 00001000`	-1	11111111(9 times) 00000001	00001000 11111111(9times) 00000001	varints编码
int64	2	`2<<3 OR 0 = 00010000`	-1	11111111(9 times) 00000001	00010000 11111111(9times) 00000001	varints编码
sint32	3	`3<<3 OR 0 = 00011000`	-1	00000001	00011000 00000001	无符号变换
sint32	4	`4<<3 OR 0 = 00100000`	-1	00000001	00100000 00000001	无符号变换
sfixed32	5	`5<<3 OR 5 = 00101101`	-1	11111111(4 times)	00011000 11111111 11111111 11111111 11111111	固定4byte长度
sfixed32	6	`6<<3 OR 1 = 00110001`	-1	11111111(8 times)	00011000 11111111 11111111 11111111 11111111 11111111 11111111 11111111	固定8byte长度

UIntType

类型	key	EncodedAs key	value	EncodedAs value	EncodedAs String	Notes
:-:	:-:	:————:	:—:	:————-:	:—:	:—:
uint32	1	`1<<3 OR 0 = 00001000`	1	00000001	00001000 00000001	无符号
uint64	2	`2<<3 OR 0 = 00010000`	2	00000010	00010000 00000010	无符号
fixed32	3	`3<<3 OR 5 = 00011101`	1	00000001 00000000(3 times)	00011101 00000001 00000000(3 times)	固定4byte长度
fixed64	4	`4<<3 OR 1 = 00100001`	2	00000010 00000000(7 times)	00011101 00000001 00000000(7 times)	固定8byte长度

FType

类型	key	EncodedAs key	value	EncodedAs value	EncodedAs String	Notes
:-:	:-:	:————:	:—:	:————-:	:—:	:—:
float	1	`1<<3 OR 5 = 00001101`	0.1	11001101 11001100 11001100 00111101	00001101 11001101 11001100 11001100 00111101	IEEE浮点数,4byte
double	2	`2<<3 OR 1 = 00010001`	0.2	10011010 10011001 10011001 10011001 10011001 10011001 11001001 00111111	00010001 10011010 10011001 10011001 10011001 10011001 10011001 11001001 00111111	IEEE浮点数,8byte
bool	3	`3<<3 OR 0 = 00011000`	false	00000000	00011000 00000000	固定1byte长度

BSType

类型	key	EncodedAs key	value	EncodedAs value	EncodedAs String	Notes
:-:	:-:	:————:	:—:	:————-:	:—:	:—:
string	1	`1<<3 OR 2 = 00001010`	“hello,world.”	00001100(length) 01101000 01100101 01101100 01101100 01101111 00101100 01110111 01101111 01110010 01101100 01100100 00101110(ASCII)	length+content编码
bytes	2	`2<<3 OR 2 = 00010010`	“h”,”0x1”,”d”,”ILV”,”0xf”	00000001 01101000 00010010 00000011 00110000 01111000 00110001 00010010 00000001 01100100 00010010 00000011 01001001 01001100 01010110 00010010 00000011 00110000 01111000 01100110	符号分割

3.编解码字段顺序

- 编解码与字段顺序无关,由key-value机制就能保证
- 对于未知的字段，编码的时候会把它写在序列化完的已知字段后面

嵌套与引用

1.嵌套定义

- `message` 定义中可以嵌套定义 `message`，`enum`
- 嵌套定义的单元外部可见，引用路径由外到内逐层引用

2.引用

- 定义 `package` 相当于C++的 `namespace`,外部引用时需要package名
- 引用外部message定义，只需要 `import` 外部proto文件即可使用
- 引用使用相对路径,生成文件时protobuf可以自动处理

完整示例

user.proto

Code

```
1  //user.proto
2
3  syntax="proto2“;
4  package user;
5
6  message User{
7      required sfixed64 uid = 1;
8
9      enum PhoneType{
10         NONE = 1;
11         HOME = 2;
12         MOBILE = 3;
13     }
14
15     message PhoneNumber {
16         required string number = 1;
17         optional PhoneType type = 2 [default = MOBILE];
18     }
19
20     repeated PhoneNumber phone = 2;
21
22     optional sfixed32 age = 3;
23 }
24
25 message Admin {
26     required sfixed64 uid = 1;
```

```
27      optional User.PhoneNumber phone = 2;
28  }
```

- room.proto

Code

```
1   //room.proto
2
3   syntax="proto2";
4   package room;
5
6   import "relative_dir/user.proto";
7
8   message Room {
9       repeated sfixed64 rid = 1;
10      repeated user.User user = 2;
11      optional double profit = 3;
12  }
```

—— Protobuf编译使用 ——

- 由协议生成C++文件: `protoc --cpp_out=${DIR} proto-file...`
- 多个协议文件依赖关系: `protoc --cpp_out=${DIR} base-proto-file deliver-proto-file...`
- 依赖问题按相对路径处理import即可解决

cmake 自动化编译

Code

```
FIND_PACKAGE(Protobuf REQUIRED)
LINK_LIBRARIES(
    protobuf
)

FILE(GLOB ProtoFiles RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} *.proto)
PROTOBUF_GENERATE_CPP(PROTO_SRCS PROTO_HDRS ${ProtoFiles} )
```

—— Protobuf序列化与反序列化 ——

1.初始化

- `set_xxx()` 设置 `required`，`optional` 字段值
- `add_xxx()` 添加 `repeated` 字段值
- `set_xxx(int,x)` 设置 `repeated` 中元素的值

2.序列化

- `required` 字段需要初始化,可以通过 `IsInitialized` 来检查是否完成message对象的初始化
- `SerializedAsString()`，`SerializedToString(std::string*)` 序列化为std::string
- `SerializedToArray(void*,int)` 序列化为byte数组
- `SerializedToOstream(ostream*)` 序列化到输出流
- `ByteSize()` 获取二进制字节序的大小，可用于初始化存放容器

3.反序列化

- `ParseFromString(std::string& data)` 从字符串中反序列化
- `ParseFromArray(const void *,int)` 从字节序中反序列化
- `ParseFromIstream(istream*)` 从输入流中反序列化
- `has_xxx()` 用于检查相应字段是否存在数据
- `xxx_size()` 用于确定 `repeated` 字段是否存在，0表示未序列化

4.获取对象

- `xxx()` 返回 `required/optional` 字段的 `const` 值,只读模式，返回 `repeated` 列表的指针,用于修改
- `mutable_xxx()` 返回字段指针,用于修改
- `xxx(int)` 返回 `repeated` 字段列表的元素,只读

—— protobuf实现原理 ——

1.protobuf的cache机制

- protobuf message的clear()操作是存在cache机制的，它并不会释放申请的空间，这导致占用的空间越来越大。
- 如果程序中protobuf message占用的空间变化很大，那么最好每次或定期进行清理。这样可以避免内存不断的上涨。

—— 注意事项 ——

1. 嵌套定义时，被嵌套的结构体被解析成A_B形式,需要获取 `mutable_b()` 指针来初始化该字段,使用栈上的对象会导致程序异常
2. 应用程序中使用protobuf,需要在退出程序时调用 `google::protobuf::ShutdownProtobufLibrary()` 以清理内存,否则会造成内存泄漏

赏

< CMake使用进阶 TCP传输文件和对象序列化 >

Styling with Markdown is supported

Comment

Powered by Gitment

linghutf

林花谢了春红，太匆匆!

主页

历史

标签

文章目录