Type Encoding/Decoding rules: from C to Protocol Buffer

Shen Liu

(Version 1.1, last modified 06/01/2016)

Abstract

Type conversion between C and protocol buffer is an important issue in our project. When the separation is done and the RPC tool begins working, we must automatically restore all the arguments for each RPC function in the receiver process, which means the function parameter types need to be transmitted between two processes fully and exactly. Unfortunately, the type system that protocol buffer supports is quite weak. To make our project automatically run in the end, we have to design a type conversion protocol to let protocol buffer automatically convert some advanced C types(e.g. pointer) into protocol buffer types.

1 Background

We use gRPC, which is fully based on google protocol buffer, to deal with RPC issues in our project. In gRPC, a C type must be packed to protocol buffer "`message`" type in a .proto file(IDL file) for further transmission. For example, if you have a C function `int foo(int x)` which needs to be called remotely, then in your .proto file, the argument type `int` can be packed in protocol buffer as follows:

```
message M{                          message M{
  int64 x=1;        or                int32 x=1
}                                   }
```

Next, protocol buffer will automatically generate a group of read/write APIs for each message. Here is an API for x's value assignment:

```
void set_x(::google::protobuf::int32 value);
```

and an API for getting the value of x:

```
inline ::google::protobuf::int32 M::x() const {
 return x_;
}
```

Here is a more complex C-protobuf type conversion sample:

```
typedef struct{                    message Circle{
  int x;                             message Point{
  int y;                               int64 x=1;
}Point;                                int64 y=2;
                          --->       }
typedef struct{                    double radius=1;
  Point center;                    }
  double radius;
}Circle;
```

Our project can automatically finish this conversion for all scalar types and simple composite types as "Circle". However, when parameter types become more and more complex, especially for those structures with multi-level pointers, generating a correct .proto file automatically as before will be a real challenge. To achieve this goal, a possible way is designing a type-conversion protocol to make our project work more intelligently. Simply speaking, for any C type input, first we use such a protocol to convert it into an integer array(encoding), and then construct the "message" type in .proto. On the receiver side, we do array parsing to restore the original C types(decoding) instead of parsing the complex .proto file.

2 Type system and encoding/decoding rules

In this draft we only use a small subset of C type system to show how the encoding/decoding idea works. Here is how our toy type system looks like:

```
Type t := int | float | t*
          | struct{t1;t2;...;tn}
```

Any type in this type system will be encoded as a list of integers(`int[] lst`), and the first element(see the following table) in this list denotes what C type this list stands for.

lst[0]	type
0	int
1	float
2	pointer
3	struct

Once we have such a type system, we can easily construct the encoding/decoding rules as follows:

```
Definition encode: type t -> int[]
  match t with
  | int   => [0]
  | float => [1]
```

```
  | t*    => [2]::encode(t)   // the list whose head is 2 and tail is e
  | struct(t1;...;tn) =>
    3::n::encode(t1)::encode(t2)::...::encode(tn)
  end.



Definition decode int[] lst =
  match lst[0] with
    | 0 => (int, tail(lst))  // tail(lst) is the list lst without the h
    | 1 => (float, tail(lst))
    | 2 =>
      let (t1, l1) = decode (tail(lst)) in (t1*, l1)
    | 3 =>
      let n = lst[1] in
      let (t1,l1) = decode (tail(tail(lst)) in
      let (t2,l2) = decode l1 in
       ...
      let (tn,ln) = decode l_{n-1} in
      (struct{t1;t2;...;tn}, ln)
  end
```