LUA的unpack函数，

函数unpack可以返回多个值，它传入一个array，然后返回array中的每一个值。

print(unpack{10,20,30}) --> 10 20 30

a,b = unpack{10,20,30} -- a=10, b=20, 30 is discarded

xpcall

xpcall (f, err)

　　功能：与pcall类似，在保护模式下调用函数(即发生的错误将不会反射给调用者)

　　但可指定一个新的错误处理函数句柄

当调用函数成功能返回true,失败时将返回false加err返回的结果

例如 local status, dispatch = xpcall(f, function ( err)

print(err);

end);

LUA与C/C++交互

在C和Lua之间通信关键内容在于一个虚拟的栈。几乎所有的API调用都是对栈上的值进行操作，所有C与Lua之间的数据交换也都通过这个栈来完成

小DEMO

#include<stdio.h>

#include<lua.h>

#include<lauxlib.h>

#include<lualib.h>

intmain (void)

{

charbuff[256];

interror;

lua\_State \*L = lua\_open(); /\* opens Lua \*/

luaopen\_base(L); /\* opens the basic library \*/

luaopen\_table(L); /\* opens the table library \*/

luaopen\_io(L); /\* opens the I/O library \*/

luaopen\_string(L); /\* opens the string lib. \*/

luaopen\_math(L); /\* opens the math lib. \*/

while(fgets(buff, sizeof(buff), stdin) != NULL) {

error = luaL\_loadbuffer(L, buff, strlen(buff), "line") || lua\_pcall(L, 0, 0, 0);

if(error) {

fprintf(stderr, "%s", lua\_tostring(L, -1));

lua\_pop(L, 1);/\* pop error message from the stack \*/

}

}

lua\_close(L);

return0;

}

代码分析

函数lua\_open 创建一个新的LUA环境（或state），这个环境并不包括预定义的函数，甚至是print。

可以通过luaopen\_io(L); /\* opens the I/O library \*/自己去引入这些函数，有需要才引入，保持苗条

对于用户输入

的每一行，C程序首先调用luaL\_loadbuffer编译这些Lua代码

如果没有错误，这个调用返回零并把编译之后的chunk压入栈

C程序调用lua\_pcall，它将会把chunk 从栈中弹出并在保护模式下运行它.

在有错误的情况下，这两个函数都将一条错误消息压入栈；我们可以用lua\_tostring来得到这条信息、输出它，用lua\_pop将它从栈中删除。

LUA与C 变量交互问题

当在Lua和C之间交换数据时我们面临着两个问题：动态与静态类型系统的不匹配

和自动与手动内存管理的不一致

LUA API用一个抽象的栈在Lua与C之间交换值

**Protocol Buffers 简介**

简单地说，这个东东干的事儿其实和**XML**差不多，也就是把某种数据结构的信息，以某种格式保存起来。主要用于数据存储、传输协议格式等场合。

先说时间开销：XML格式化（序列化）的开销倒还好；但是XML解析（反序列化）的开销就不敢恭维啦。俺之前经常碰到一些时间性能很敏感的场合，由于不堪忍受XML解析的速度，弃之如敝履。

再来看空间开销：熟悉XML语法的同学应该知道，XML格式为了有较好的可读性，引入了一些冗余的文本信息。所以空间开销也不是太好（不过这点缺点，俺不常碰到）。

除了性能好，代码生成机制是主要吸引俺的地方，无需自己生成，使用工具即可

先来看一个非常简单的例子。假设你想定义一个“搜索请求”的消息格式，每一个请求含有一个查询字符串、你感兴趣的查询结果所在的页数，以及每一页多少条查询结果。可以采用如下的方式来定义消息类型的.proto文件了：

|  |
| --- |
| message SearchRequest {    required string query = 1;    optional int32 page\_number = 2;    optional int32 result\_per\_page = 3;  } |

正如上述文件格式，在消息定义中，每个字段都有唯一的一个标识符。这些标识符是用来在消息的二进制格式中识别各个字段的，一旦开始使用就不能够再改 变。注：[1,15]之内的标识号在编码的时候会占用一个字节。[16,2047]之内的标识号则占用2个字节。所以应该为那些频繁出现的消息元素保留 [1,15]之内的标识号。切记：要为将来有可能添加的、频繁出现的标识号预留一些标识号。

  required：一个格式良好的消息一定要含有1个这种字段。表示该值是必须要设置的；

optional：消息格式中该字段可以有0个或1个值（不超过1个）。

  repeated：在一个格式良好的消息中，这种字段可以重复任意多次（包括0次）。重复的值的顺序会被保留。表示该值可以重复，相当于java中的List。

由于一些历史原因，基本数值类型的repeated的字段并没有被尽可能地高效编码。在新的代码中，用户应该使用特殊选项[packed=true]来保证更高效的编码。如：

|  |
| --- |
| repeated int32 samples = 4 [packed=true]; |

required是永久性的：在将一个字段标识为required的时候，应该特别小心。如果在某些情况下不想写入或者发送一个required的 字段，将原始该字段修饰符更改为optional可能会遇到问题——旧版本的使用者会认为不含该字段的消息是不完整的，从而可能会无目的的拒绝解析。在这 种情况下，你应该考虑编写特别针对于应用程序的、自定义的消息校验函数。Google的一些工程师得出了一个结论：使用required弊多于利；他们更 愿意使用optional和repeated而不是required。当然，这个观点并不具有普遍性。

对C++来说，编译器会为每个.proto文件生成一个.h文件和一个.cc文件，.proto文件中的每一个消息有一个对应的类。

如上所述，消息描述中的一个元素可以被标记为“可选的”（optional）。一个格式良好的消息可以包含0个或一个optional的元素。当解 析消息时，如果它不包含optional的元素值，那么解析出来的对象中的对应字段就被置为默认值。默认值可以在消息描述文件中指定。例如，要为 SearchRequest消息的result\_per\_page字段指定默认值10，在定义消息格式时如下所示：

|  |
| --- |
| optional int32 result\_per\_page = 3 [default = 10]; |

l  **更新一个消息类型**

如果一个已有的消息格式已无法满足新的需求——如，要在消息中添加一个额外的字段——但是同时旧版本写的代码仍然可用。不用担心！更新消息而不破坏已有代码是非常简单的。在更新时只要记住以下的规则即可。

²  不要更改任何已有的字段的数值标识。

²  所添加的任何字段都必须是optional或repeated的。这就意味着任何使用“旧”的消息格式的代码序列化的消息可以被新的代码所解析，因为它们 不会丢掉任何required的元素。应该为这些元素设置合理的默认值，这样新的代码就能够正确地与老代码生成的消息交互了。类似地，新的代码创建的消息 也能被老的代码解析：老的二进制程序在解析的时候只是简单地将新字段忽略。然而，未知的字段是没有被抛弃的。此后，如果消息被序列化，未知的字段会随之一 起被序列化——所以，如果消息传到了新代码那里，则新的字段仍然可用。注意：对Python来说，对未知字段的保留策略是无效的。

²  非required的字段可以移除——只要它们的标识号在新的消息类型中不再使用（更好的做法可能是重命名那个字段，例如在字段前添加“OBSOLETE\_”前缀，那样的话，使用的.proto文件的用户将来就不会无意中重新使用了那些不该使用的标识号）。

²  一个非required的字段可以转换为一个扩展，反之亦然——只要它的类型和标识号保持不变。

²  int32, uint32, int64, uint64,和bool是全部兼容的，这意味着可以将这些类型中的一个转换为另外一个，而不会破坏向前、 向后的兼容性。如果解析出来的数字与对应的类型不相符，那么结果就像在C++中对它进行了强制类型转换一样（例如，如果把一个64位数字当作int32来 读取，那么它就会被截断为32位的数字）。

²  sint32和sint64是互相兼容的，但是它们与其他整数类型不兼容。

²  string和bytes是兼容的——只要bytes是有效的UTF-8编码。

²  嵌套消息与bytes是兼容的——只要bytes包含该消息的一个编码过的版本。

²  fixed32与sfixed32是兼容的，fixed64与sfixed64是兼容的。

**与C++的相互转换**

1、在.proto文件中定义消息格式

2、使用protobuf编译器

3、使用c++ api来读写消息

运行protoc 来生成c++文件：

protoc -I=$SRC\_DIR --cpp\_out=$DST\_DIR $SRC\_DIR/addressbook.proto

**生成.c/cpp和h文件**