1. Tick 中发送网络包，不同职业有护甲，根据计算恢复速度，计算下一次恢复时间进行比较。 GetA() && NextReceverTime > NowTime
2. 宝箱固定数量道具，目标分别进行随机，有几率获取道具 循环变量相同
3. 杀怪收集道具，计算道具时，原来是计算道具时返回static List ,再从返回值里面将内容塞给具体收集列表，后优化改为直接将收集列表传入到计算函数，外层未修改，导致回来返回后死循环
4. 多人开一个宝箱，道具已经初始化列表，在计算具体哪个人分配哪个的时候（个人可见，可拾取），可能存在失败情况，而有安全指针释放（析构），所以被free，未置空，后面宝箱消失时会销毁列表中剩余道具。后续有其他角色使用到这个指针，宝箱再次free，并置空，导致其他人在存档的时候出现空指针，宕机。
5. 读档时候由于 在逻辑层 使用了 lua\_tostring，其返回的是一个指向字符串的内部拷贝指针。且不能修改，只要指针对应值还在栈内，lua会保证这个指针一直有效，当一个C函数返回后，lua会清理它的栈，因此在上层调用指针会出现错误。
   1. 永远不要将指向lua字符串的指针保存到访问他们的外部函数中

Lord: apid 该服务提供跨平台语言消息通信服务

反射在序列化中引用：

|  |
| --- |
| 当然，还可以使用 protobuf 来序列化.  test.proto  syntax = "proto2";package example;  message Message {  required uint64 id = 1;  required uint64 size = 2;  required string data = 3;  }  func ProtoEncodeDecode() {  m1 := &example.Message{  Id: proto.Uint64(4),  Size: proto.Uint64(1024),  Data: proto.String("proto"),  }  buf, err := proto.Marshal(m1)  if err != nil {  log.Fatal("proto marshal error:", err)  }  var m2 example.Message  if err = proto.Unmarshal(buf, &m2); err != nil {  log.Fatal("proto unmarshal error:", err)  }  fmt.Println(m2.GetId(), m2.GetSize(), m2.GetData())  } |
| Golang 序列化的方式：  Binary  Gob  JSON  Protobuf  一. Binary  // OKtype Message struct {  Id uint64  Size uint64  }  // Wrong/\*type Message struct {  Id int  Size int  Data string  }\*/  func BinaryRW() {  m1 := Message{1, 1024}  buf := new(bytes.Buffer)  if err := binary.Write(buf, binary.LittleEndian, m1); err != nil {  log.Fatal("binary write error:", err)  }  var m2 Message  if err := binary.Read(buf, binary.LittleEndian, &m2); err != nil {  log.Fatal("binary read error:", err)  }  }  注意： 如果字段中有不确定大小的类型，如 int，slice，string 等，则会报错。  binary write error:binary.Write: invalid type main.Message  下面是binary.Write的函数说明：  // Data must be a fixed-size value or a slice of fixed-size values.func Write(w io.Writer, order ByteOrder, data interface{}) error {  1  2  解决办法：  int 换成 int32 等固定大小的类型  slice 换成类似 [8]byte 这种固定大小  选择其他序列化方式 |

|  |
| --- |
| 1.luaL\_ref  原型：  int luaL\_ref (lua\_State \*L, int t);  该方法对栈顶对象创建一个引用，放在table的index里，这里的index就是参数t，记住这个t，在用rawgeti是会用到。引用是唯一的  2.lua\_rawgeti  原型：  [html] [view plain](https://blog.csdn.net/yuxikuo_1/article/details/52185370" \o "view plain) [copy](https://blog.csdn.net/yuxikuo_1/article/details/52185370" \o "copy)  void lua\_rawgeti (lua\_State \*L, int index, int n);  把t[n]的值压栈，index就是luaL\_ref中的t，n就是key，比如一些方法，这是一个直接访问，不会触发元方法  lua\_State.NewTable()  ref = lua\_State.luaL\_ref( index )  lua\_State.lua\_rawgeti(index，ref )  以上是伪代码，大意就是先创建Table，设置引用ref，在直接取得table |
| http://www.mamicode.com/info-detail-202127.html |

**1.1 Multiple Threads**

     Lua中的线程本质上是一个协同程序。我们可以认为一个协同程序就是一个线程加上一个良好的接口，或者我们可以认为一个线程就是带一个低等级API的协同程序。

     从C API角度来看，认为一个线程就是一个栈是挺有用的。而从实现过程角度来看，线程确实是一个栈。每个栈保存所有未执行的调用、每次调用的参数和局部变量等信息，换句话来说一个栈拥有一个线程要准备再次运行的所有信息。因此，多线程就是意味着多个独立的栈。

     实际上看，我们之前讨论的各个Lua-C 的API函数，都是在一个特定的栈上操作的，这个从一开始就知道了。但是我们又是如何知道我们要操作具体的哪个栈呢？会不会操作错了？这里关键点就在于我们每次调用函数时的第一个参数类型lua\_State ,这个参数不仅代表一个lua state，也同时包括在这个state中的thread。

     无论何时创建了一个Lua state，Lua都会自动在这个state中创建一个新的线程，然后返回一个代表这个线程的lua\_State。这个主线程是从不会被回收的(这个？？是否指的除去调用lua\_newthread额外创建的这些线程，即创建lua state自动创建的这个线程)。当你用lua\_close 关闭了这个state的时候，线程会随同state一起释放。

     可以在一个state中通过调用lua\_newthread 来创建另外的thread：

lua\_State \*lua\_newthread(lua\_State \*L);

     上面这个函数返回一个代表这个新线程的lua\_State 型指针，并且同时将这个新的线程作为thread类型推进栈中，如：

L1 = lua\_newthread(L);

     经过上面这步，我们就有了两个线程了，且两个线程都实际上在同一个lua state内部。每个线程都有自己独立的栈，新的线程L1有个空的栈；而L线程则将新的线程存储在其自身栈的栈顶。如：

e.g.

printf("%d\n",lua\_gettop(L1)); --0 新的线程的栈是一个空的栈

printf("%s\n",luaL\_typename(L,-1)); --thread 代表在栈顶是一个线程，即新的那个线程

     除了主线程之外，其余的这些线程都是受garbage collection管理的对象，和Lua其余的这些对象一样。当创建了一个新的线程，以thread类型将这个线程推进至栈中，保证了这个线程不会被回收掉。我们应该确保不使用那些不确定还是否在栈中的线程。每次调用Lua的API都可能会触发回收那些不在栈中的线程，即便这个线程还在使用中，如：

lua\_State \*L1 = lua\_newthread(L);

lua\_pop(L,1); /\* L1 is garbage\*/

lua\_pushstring(L1,"helloe"); /\* danger \*/

     调用lua\_pushstring的时候可能会触发garbage collector回收掉L1，即便此时仍然在使用这个线程。为了避免遇到不可预期的错误，应该始终对我们在使用中的线程保持引用，for instance in the stack of an anchored thread or in the registry.

     新创建的线程，可以和使用主线程一样使用。可以从该线程的栈中推进和推出元素，可以使用栈来调用函数等等。这里介绍一个函数lua\_xmove(L1,L,1),这个函数的作用是做同一个state中的栈的数据移动，如lua\_xmove(F，T，n)，就是从栈F中推出一个 元素，然后推进至栈T中。做这个操作我们甚至都不需要创建额外新的线程，只需要在主线程中处理就可以了。在这里使用多线程的前提是实现协同程序，以便我们可以延缓或者恢复某个线程的执行。恢复执行某个函数,在这里就需要函数:lua\_resume 来支持：

int lua\_resume(lua\_State \*L,lua\_State \*from ,int nary);

     如我们使用lua\_pcall 一样，如果我们想运行一个协同程序，我们可以使用lua\_resume函数：将要调用的函数,参数推进至栈中，函数参数narg代表传递的参数的数量。(from 参数指的是执行这个调用的线程所在的state)。主要的行为类似于lua\_pcall,但还是有三个不同的地方：1、lua\_resume没有代表返回值数量的参数，函数总是会返回所有的返回结果；2、没有给消息句柄用的参数，错误不会在栈中触发，因此可以在错误之后检测栈(~~~没看懂)；3、如果在运行的这个程序被挂起了，lua\_resume 返回一个特殊代码：LUA\_YIELD ，且将在state中的线程保留下来留给之后重新调用。

     协同程序也能调用C函数，这些C函数也能回调其他的Lua函数。

Boost.Asio

https://mmoaay.gitbooks.io/boost-asio-cpp-network-programming-chinese/content/Chapter2.html