零声教育 Mark 老师 QQ: 2548898954

lua 编程

lua 数据类型

> `boolean`, `number`, `string`, `nil`, `function`, `table`, `userdata`, `lightuserdata`, `thread`;
>
> `boolean` 为 `true`、`false`；其中 `false` 可以解决 `table` 作为 array 时，元素为 nil 时造成 `table` 取长度未定义的行为；
>
> `number` 为 `integer` 和 `double` 的总称；
>
> `string` 常量字符串；这样 lua 中字符串比较只需要进行地址比较就行了；
>
> `nil` 通常表示**未定义**或者**不存在**两种语义；
>
> `function` 函数；与其他语言不同的是，lua 中 `function` 为第一类型；注意 lua 中的匿名函数，lua 文件可视为一个匿名函数；加载 lua 文件，可视为执行该匿名函数；
>
> `table` 表；lua 中唯一的数据结构；既可以表示 hashtable 也可表示为 array；配合元表可以定制表复杂的功能（如实现面对对象编程中的类以及相应继承的功能）；
>
> `userdata` 完全用户数据；指向一块内存的指针，通过为 `userdata` 设置元表，lua 层可以使用该 `userdata` 提供的功能；`userdata` 为 lua 补充了数据结构，解决了 lua 数据结构单一的问题；可以在 c 中实现复杂的数据结构，生成库继而导出给 lua 使用；注意：`userdata` 指向的内存需要由 lua 创建，同时 `userdata` 的销毁也交由 lua gc 来自动回收；

> `lightuserdata` 轻量用户数据；也是指向一块内存的指针，但是该内存由 c 创建，同时它的销毁也由 c 来完成；不能为它创建元表，轻量用户数据只有类型元表；通常用于 lua 想使用 c 的结构，但是不能让 lua 来释放的结构；在游戏客户端中用的比较多；
>
> `thread` 线程；lua 中的协程和虚拟机都是 `thread` 类型；

元表

> 元表可以修改一个**值**在面对一个**未知操作**时的行为。
>
> 常用的有：
>
> `__index`：索引 `table[key]`。 当 `table` 不是表或是表 `table` 中不存在 `key` 这个键时，这个事件被触发。 此时，会读出 `table` 相应的元方法。
>
> `__newindex`：索引赋值 `table[key] = value` 。 和索引事件类似，它发生在 `table` 不是表或是表 `table` 中不存在 `key` 这个键的时候。 此时，会读出 `table` 相应的元方法。
>
> `__gc`：元表中用一个以字符串 "`__gc`" 为索引的域，那么就标记了这个对象需要触发终结器；

注意

- 只有 `table` 和 `userdata` 对象有独自的元表，其他类型只有类型元表；
- 只有 `table` 可以在 lua 中**修改设置**元表；
- `userdata` 只能在 c 中**修改设置**元表，lua 中不能**修改** `userdata` 元表；

协程

> skynet 最小的运行的单元；
>
> 一段独立的执行线程；
>
> 一个 lua 虚拟机中可以有多个协程，但同时只能有一个协程在运行；

闭包

表现

- 函数内部可以访问函数外部的变量；
- lua 文件是一个匿名函数；

  > lua 内部函数可以访问文件中函数体外的变量；

实现

- C 函数以及绑定在 C 函数上的上值（upvalues）；

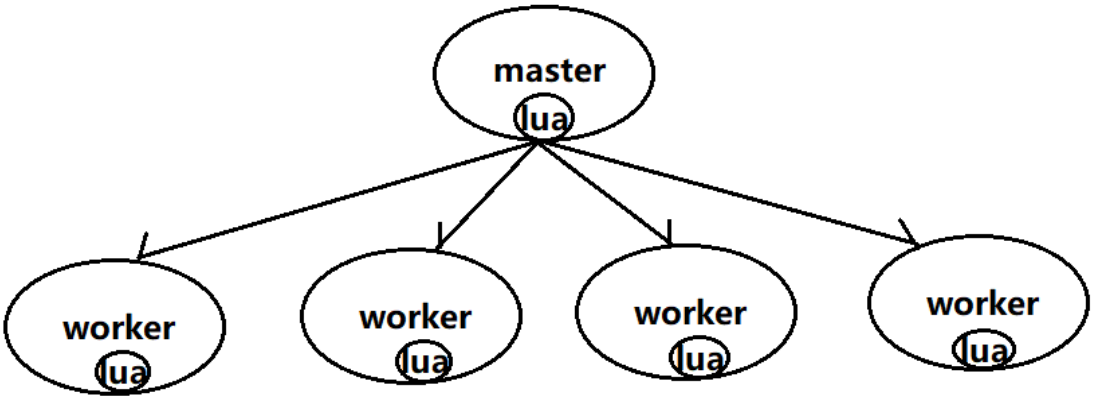
与其他语言差异

- 没有入口函数
- 索引从 1 开始
- 闭包
- 多返回值
- 函数是第一类型
- 尾递归，不占用栈空间
- 条件表达式：`nil` 或者 `false` 为假；非 `nil` 为真；
- 多元运算：`A and B or C` 其中 A、B、C 均为表达式；类似于 c/c++ 中的 `A ? B : C`；差异在于条件表达式的差异；
- 非运算符：是 `~` 而不是 `!`；所以不等于为 `~=`；

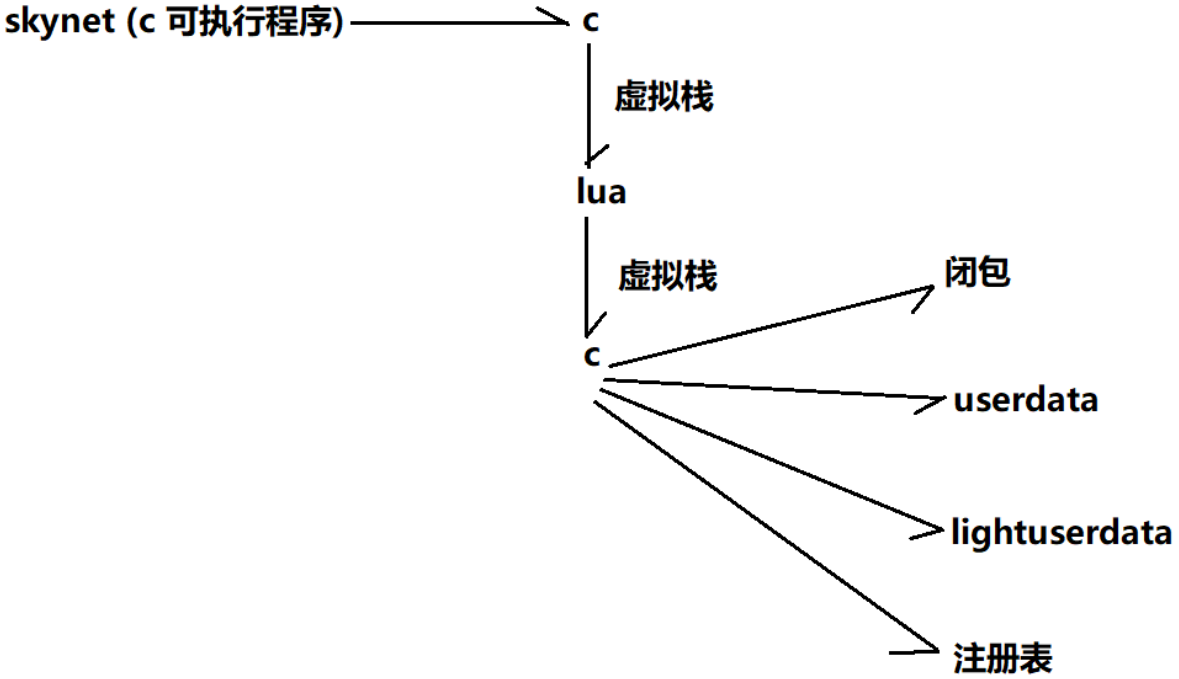
lua/c 接口编程

> skynet、openresty 都是深度使用 lua 语言的典范；学习 lua 不仅仅要学习基本用法，还要学会使用 c 与 lua 交互，这样才学会了 lua 作为胶水语言的精髓；

openresty(nginx + lua)



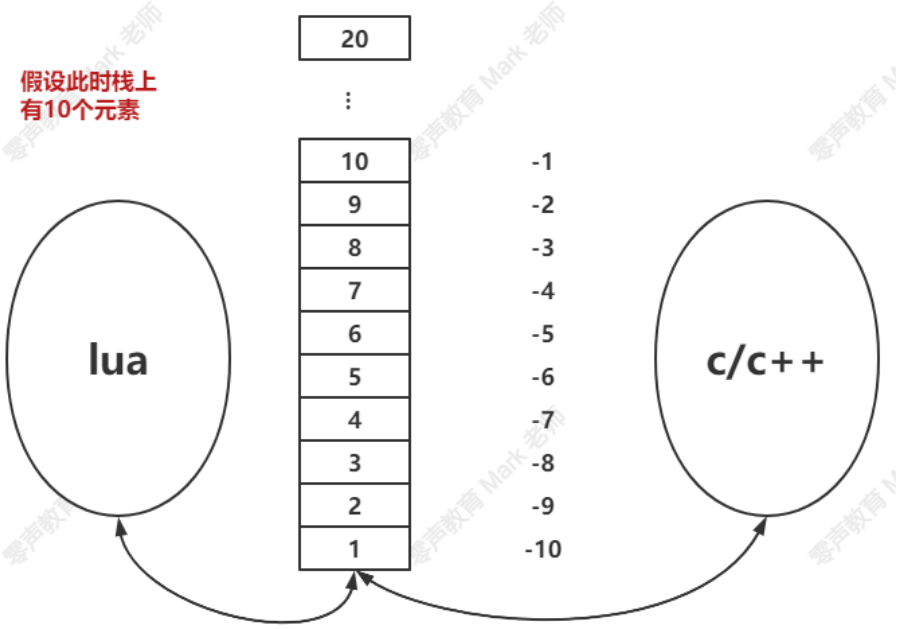
skynet中调用层次



虚拟栈

- 栈中只能存放 lua 类型的值，如果想用 c 的类型存储在栈中，需要将 c 类型转换为 lua 类型；

- lua 调用 c 的函数都得到一个**新**的栈，独立于之前的栈；
- c 调用 lua，每一个协程都有一个栈；
- c 创建虚拟机时，伴随创建了一个主协程，默认创建一个虚拟栈；
- 无论何时 Lua 调用 C， 它都只保证至少有 `LUA_MINSTACK` 这么多的堆栈空间可以使用。 `LUA_MINSTACK` 一般被定义为 **20**，因此，只要你不是不断的把数据压栈， 通常你不用关心堆栈大小。



C 闭包

- 通过 `lua_pushcclosure` 用来创建 C 闭包；
- 通过 `lua_upvalueindex` 伪索引来获取上值（lua 值）；
- 可以为多个导出函数（c 导出函数给 lua 使用）共享上值，这样可以少传递一个参数；

注册表

> 可以用来在多个 c 库中共享 lua 数据（包括 `userdata` 和 `lightuserdata`）；

- 一张预定义的表，用来保存任何 c 代码想保存的 lua 值；
- 使用 `LUA_REGISTRYINDEX` 来索引；

- 例子：获取`skynet_context`；

`userdata`

> `userdata` 是指向一块内存的指针，该内存由 lua 来创建，通过 `void *lua_newuserdatauv(lua_State *L, size_t sz, int nuvalue)` 这个函数来创建；注意：这块内存大小必须是固定的，不能动态增加，但是这块内存中的指针指向的数据可以动态增加；还有就是 `userdata` 可以绑定若干个 lua 值（又称 *uservalue*）（在 lua 5.3 中只能绑定一个 lua 值，lua 5.4 可以绑定多个）；`userdata` 与 *uservalue* 的关系是引用关系，也就是 *uservalue* 的生命周期与 `userdata` 的生命周期一致，`userdata` gc 时，*uservalue* 也会被释放；通常这个特性可以用来绑定一个 lua `table` 结构，因为 c 中没有 hash 结构，辅助 lua `table` 结构实现复杂的功能；也可以用来实现延迟 gc，如果某个 `userdata` 希望晚点 gc，在 `userdata` 的 `__gc` 元表中生成一个临时的 `userdata`，然后将那个希望晚点 gc 的 `userdata` 绑定在这个临时 `userdata` 的 `uservalue` 上；
>
> `int lua_getiuservalue (lua_State *L, int idx, int n)` 来获取绑定在 `userdata` 上的 *uservalue*；
>
> `int lua_setiuservalue (lua_State *L, int idx, int n)` 来设置 `userdata` 上的 *uservalue*；

`lightuserdata`

> 轻量用户数据也是指向一块内存的指针，但是该内存由 c 来创建和销毁；通常这块内存的生命周期由 c 宿主语言来控制；可以将 `lightuserdata` 绑定在注册表中，让多个 lua 库共享该数据；在 skynet 中，`lightuserdata` 可以指向同一块数据，在多个 Actor 中传递这个 `lightuserdata`，然后分别为这个 `lightuserdata` 创建一个 `userdata`；在 `userdata` 中的 `__gc` 来释放这个 `lightuserdata`；注意：为了避免这块内存多次释

> 放，需要为这块内存加上引用计数；同时 skynet 中 actor 是多线程环境下运行，所以需要为该 `lightuserdata` 加上锁；这个锁必须是自旋锁或者原子操作，因为 actor 调度是自旋锁，必须使用比它更小的粒度的锁；如果 `lightuserdata` 操作粒度过大，应该改成只在一个 actor 中加载，其他 actor 通过消息来共享数据；

简单游戏实现

游戏介绍

> 目的：掌握 actor 模型开发思路；
>
> 游戏：猜数字的游戏；
>
> 条件：满3人开始游戏，游戏开始后不能退出，直到这个游戏结束；
>
> 规则：系统当中会随机 1-100 之间的数字，参与游戏的玩家依次猜测规定范围内的数字；如果猜测正确那么该玩家就输了，如果猜测错误，游戏继续；直到有玩家猜测成功，游戏结束，该玩家失败；

设计原则

> 简单可用，持续优化，而不是一开始就过度优化；

接口设计

> skynet 中，从 actor 底层看是通过消息进行通信；从 actor 应用层看是通过 api 来进行通信；
>
> 接口隔离原则：不应该强迫客户依赖于他们不用的方法；从安全封装的角度出发，只暴露客户需要的接口；服务间不依赖彼此的实现；

- agent

  > `login`：实现登录功能；断线重连
  >
  > `ready`：准备，转发到大厅，加入匹配队列；
  >
  > `guess`：猜测数字，转发到房间；
  >
  > `help`：列出所有操作说明；
  >
  > `quit`：退出；

- hall

  > `ready`：加入匹配队列；
  >
  > `offline`：用户掉线，需要从匹配队列移除用户；

- room

  > `start`：初始化房间；
  >
  > `online`：用户上线，如果用户在游戏中，告知游戏进度；
  >
  > `offline`：用户下线，通知房间内其他用户；
  >
  > `guess`：猜测数字，推动游戏进程；

游戏演示

- 客户端

  > telnet 127.0.0.1 8888

- 服务端

  > 先启动 redis，然后启动 skynet
  >
  > ```
  > redis-server redis.conf
  > ./skynet/skynet config.game
  > ```

- 如何优化

1. agent 服务不要实时创建，可以采用预先创建；用户验证通过后再分配 agent 地址，避免无效分配；
2. 创建 gate 服务：登陆验证、流程验证、心跳检测、验证成功之后再分配一个 agent；
3. 如果 agent 功能比较简单，那么可以创建固定数量的 agent；
4. 如果 room 功能比较简单，那么可以创建固定数量的 room；
5. 如果是万人同时在线游戏，agent、room 需要预先分配，长时间运行会让服务内存膨胀，同时也会造成 lua gc 负担会加重；
6. 重启服务策略，创建同样数量的 agent 服务组，新进来的玩家，分配到新的服务组；而旧的玩家在旧的服务组操作结束后，就淘汰该玩家，直到旧的服务组没有玩家，这时旧服务组退出；保证旧的服务组只处理旧的任务，新连接进来的用户在新的服务组进行工作；