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

> 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）；

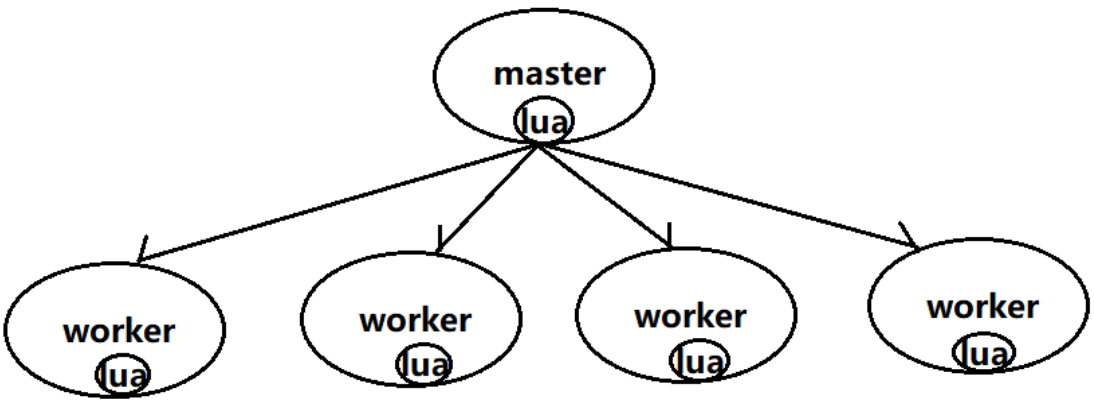
与其他语言差异

- 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 `~=`；

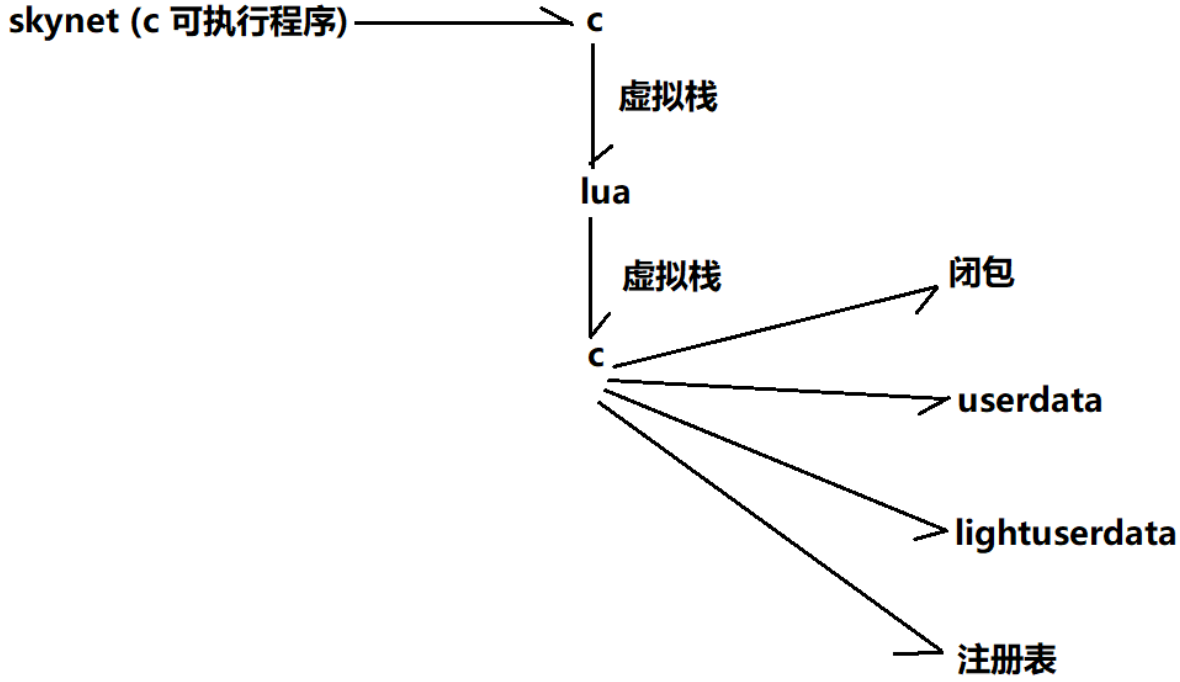
lua/c 接口编程

> 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)

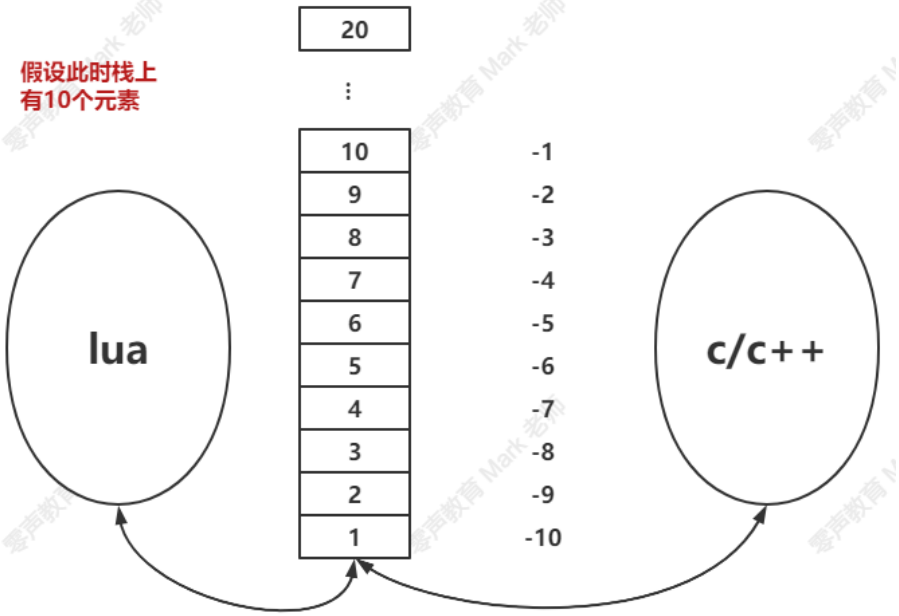


skynet中调用层次



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；

skynet 网络封装

> skynet 采用 reactor 网络编程模型；reactor 网络模型是一种异步事件模型；

连接建立

> 客户端与服务端建立连接处理；
>
> - 建立成功的标识是 listenfd，有读事件触发；
>
> 服务端与第三方服务建立连接；
>
> - 建立成功的标识是 connect 返回的fd，有可写事件触发；

连接断开

> skynet 网络层支持半关闭状态；

读写端都关闭

```
// 事件：1. EPOLLHUP 读写段都关闭
e[i].eof = (flag & EPOLLHUP) != 0;
// 处理：直接关闭并向 actor 返回事件 SOCKET_CLOSE
int halfclose = halfclose_read(s);
force_close(ss, s, &l, result);
if (!halfclose) { // 如果前面因为关闭读端已经发送 SOCKET_CLOSE，在这里避免重复 SOCKET_CLOSE
    return SOCKET_CLOSE;
}
```

检测读端关闭

```
int n = (int)read(s->fd, buffer, sz);
// 事件：2. 读端关闭 注意：EPOLLRDHUP 也可以检测，但是这个 read = 0 更为及时；因为事件处理先处理读事件，再处理异常事件
if (n == 0) {
    if (s->closing) { // 如果该连接的 socket 已经关闭
        // Rare case : if s->closing is true, reading event is disable, and SOCKET_CLOSE is raised.
        if (nomore_sending_data(s)) {
            force_close(ss,s,l,result);
        }
        return -1;
    }
    int t = ATOM_LOAD(&s->type);
    if (t == SOCKET_TYPE_HALFCLOSE_READ) { // 如果已经处理过读端关闭
        // Rare case : Already shutdown read.
        return -1;
    }
    if (t == SOCKET_TYPE_HALFCLOSE_WRITE) { // 如果之前已经处理过写端关闭，则直接 close
        // Remote shutdown read (write error) before.
        force_close(ss,s,l,result);
    } else { // 如果之前没有处理过，则只处理读端关闭
        close_read(ss, s, result);
    }
    return SOCKET_CLOSE;
}
```

> - read = 0，能检测对端写关闭，本地读关闭；

检测写端关闭

```
for (;;) {
    ssize_t sz = write(s->fd, tmp->ptr, tmp->sz);
    if (sz < 0) {
        switch(errno) {
            case EINTR:
                continue;
            case AGAIN_WOULDBLOCK:
                return -1;
        }
        // sz < 0 && errno = EPIPE,  fd is connected to a pipe or socket whose reading end is closed.
        // 在这里的处理是只要sz < 0，且不是被中断打断以及写缓冲满的情况下，直接关闭本地写端
        return close_write(ss, s, l, result);
    }
    ...
}
```

> - write < 0 && errno == EPIPE，能检测对端读端关闭，本地写端关闭；

消息到达

> 单线程读；
>
> 读策略：
>
> `int n = read(fd, buf, sz);`
>
> sz 初始值为 64；根据从网络接收数据情况，动态调整 sz 的大小；

```
// socket_server.c  [forward_message_tcp]
int sz = s->p.size;
char * buffer = MALLOC(sz);
int n = (int)read(s->fd, buffer, sz);
...
if (n == sz) {
    s->p.size *= 2;
    return SOCKET_MORE;
} else if (sz > MIN_READ_BUFFER && n*2 < sz) {
    s->p.size /= 2;
}
return SOCKET_DATA;
```

消息发送完毕

> 多线程写；同一个 fd 可以在不同的 actor 中发送数据；skynet 底层通过加锁确保数据正确发送到 socket 的写缓冲区；

```
// 注意此时在 work 线程中
int socket_server_send(struct socket_server *ss, struct socket_sendbuffer *buf) {
    int id = buf->id;
    struct socket * s = &ss->slot[HASH_ID(id)];
    if (socket_invalid(s, id) || s->closing) {
```

```
        free_buffer(ss, buf);
        return -1;
    }

    struct socket_lock l;
    socket_lock_init(s, &l);
    // 确保能在fd上写数据，连接可用状态，检测 socket 线程是否在对该fd操作；
    if (can_direct_write(s,id) && socket_trylock(&l)) {
        // may be we can send directly, double check
        // 双检测
        if (can_direct_write(s,id)) {
            // send directly
            struct send_object so;
            send_object_init_from_sendbuffer(ss, &so, buf);
            ssize_t n;
            if (s->protocol == PROTOCOL_TCP) {
                // 尝试在 work 线程直接写，如果n > 0，则写成功
                n = write(s->fd, so.buffer, so.sz);
            } else {
                union sockaddr_all sa;
                socklen_t sasz = udp_socket_address(s, s->p.udp_address,
&sa);
                if (sasz == 0) {
                    skynet_error(NULL, "socket-server : set udp (%d) address
first.", id);
                    socket_unlock(&l);
                    so.free_func((void *)buf->buffer);
                    return -1;
                }
                n = sendto(s->fd, so.buffer, so.sz, 0, &sa.s, sasz);
            }
            if (n<0) {
                // ignore error, let socket thread try again
                // 如果失败，不要在 work 线程中处理异常，所有网络异常在 socket 线程中
处理
                n = 0;
            }
            stat_write(ss,s,n);
            if (n == so.sz) {
                // write done
                socket_unlock(&l);
                so.free_func((void *)buf->buffer);
                return 0;
            }
            // write failed, put buffer into s->dw_* , and let socket thread
send it. see send_buffer()
            s->dw_buffer = clone_buffer(buf, &s->dw_size);
            s->dw_offset = n;

            socket_unlock(&l);

            inc_sending_ref(s, id);

            struct request_package request;
            request.u.send.id = id;
            request.u.send.sz = 0;
            request.u.send.buffer = NULL;

```

```
            // let socket thread enable write event
            // 如果写失败，可能写缓冲区满，或被中断打断，直接交由 socket 线程去重试；
            // 注意，这里通过 pipe 来与 socket 线程通信
            send_request(ss, &request, 'W', sizeof(request.u.send));

            return 0;
        }
        socket_unlock(&l);
    }

    inc_sending_ref(s, id);

    struct request_package request;
    request.u.send.id = id;
    request.u.send.buffer = clone_buffer(buf, &request.u.send.sz);

    send_request(ss, &request, 'D', sizeof(request.u.send));
    return 0;
}
```