Lua C API

```
void lua_createtable (lua_State *L, int narr, int nrec)
```

- 操作说明：
  创建一个新的table，并把它放在栈顶。 narr 和 nrec 分别指定该 table 的 array 部分和 hash 部分的预分配元素数量
- 返回值：无
- 栈影响： 栈高+1，栈顶元素是新 table
- 注： `#define lua_newtable(L) lua_createtable(L, 0, 0)` 常用这个

```
void lua_getfield (lua_State *L, int index, const char *k)
```

- 操作说明:

  ```
  arr = Stack[index]    // arr 是 table
  Stack.push( arr[k] )
  ```

  取表中键为 k 的元素，这里的表是由 index 指向的栈上的一个表
- 返回值: 无
- 栈影响：栈高+1，栈顶元素是(Stack[index])[k]
- 注：该操作将触发 __index 元方法

```
void lua_setfield (lua_State *L, int index, const char *k)
```

- 操作说明：

  ```
  arr = Stack[index]
  arr[k] = Stack.top()
  Stack.pop()
  ```

  给表中键为 k 的元素赋值 value (value就是栈顶元素)， 这里的表是由 index 指向的栈

上的一个表

- 返回值： 无
- 栈影响：高度-1，被弹出的是value
- 注： 该操作将触发 __newindex 元方法

```
void lua_gettable (lua_State *L, int index)
```

- 操作说明:

  ```
  ele  = Stack[index]
  key = Stack.top()
  Stack.pop()
  value = ele[key]
  Stack.push(value)
  ```

  根据 index 指定取到相应的表; 取栈顶元素为 key， 并弹出栈; 获取表中 key 的值压入栈顶。

- 返回值：无
- 栈影响：
  高度不变, 但是发生了一次弹出和压入的操作, 弹出的是key, 压入的是value
- 注：将触发 __index 元方法

```
void lua_settable (lua_State *L, int index)
```

- 操作说明:

  ```
  ele    = Stack[index]
  value  = Stack.top()
  Stack.pop()
  key    = Stack.top()
  Stack.pop()
  ele[key] = value
  ```

  根据index指定取到相应的表; 取栈顶元素做value，弹出之; 再取当前栈顶元素做key，亦弹出之; 然后将表的键为key的元素赋值为value

- 返回值: 无

- 栈影响：高度-2, 第一次弹出value, 第二次弹出key
- 注： 该操作将触发 __newindex 元方法

```
void lua_rawget (lua_State *L, int index)
```

- 操作说明：和 lua_gettable 操作一样,但是不触发相应的元方法
- 意义：TODO

```
void lua_rawgeti(lua_State *L, int index, int n)
```

- 操作说明

  ```
  ele = Stack[index]
  value = ele[n]
  Stack.push(value)
  ```

- 返回值：无
- 栈影响：栈高+1， 栈顶新增元素就是 value
- 注：不触发相应的元方法

```
void lua_rawset (lua_State *L, int index)
```

- 操作说明：和 lua_settable 操作一样，但是不触发相应的原方法

```
void lua_rawseti (lua_State *L, int index, int n)
```

- 操作说明:

  ```
  ele = Stack[index]
  value = Stack.top()
  Stack.pop()
  ele[n] = value
  ```

- 返回值：无
- 栈影响：-1, 栈顶将value弹出
- 注：不触发相应的元方法

```
void lua_pushvalue (lua_State *L, int index)
```

- 操作说明:

```
value = Stack[index]
Stack.push(value)
```

- 返回值：无
- 栈影响：栈高 +1

```
int luaL_newmetatable (lua_State *L, const char *tname)
```

- 操作说明:

1. 在注册表中查找tname，如果已经注册，就返回0，否者继续， 并平栈

```
lua_getfield(L, LUA_REGISTRYINDEX, tname)
if (!lua_isnil(L, -1))
    return 0;
lua_pop(L, 1);
```

2. 创建一个表，并注册，返回1

```
lua_newtable(L)
lua_pushvalue(L, -1)
lua_setfield(L, LUA_REGISTRYINDEX, tname)
return 1
```

相当于在注册表中有 {tname:栈顶的table} 的键值对。

- 返回值：已注册，就返回0；未注册,创建表后，返回1
- 栈影响：栈高 +1，栈顶元素是在注册表中注册过的新表

```
void *lua_newuserdata (lua_State *L, size_t size)
```

- 操作说明

  该函数分配一块由size指定大小的内存块, 并放在栈顶

- 返回值：新分配的块的地址

- 栈影响：栈高+1 ， 栈顶是userdata

- 注：userdata用来在lua中表示c中的值. 一个完整的userdata有自己的元表, 在垃圾回收时, 可以调用它的元表的__gc方法

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n)
```

- 操作说明：

  向栈上压一个C闭包。当一个c函数被创建时, 可以绑定几个值在它上面, 从而形成一个闭包. 在任何时刻调用这个c函数时, 都可以访问这几个绑定值.

  绑定的方法: 先一次压入要绑定的n个值到栈上, 然后调用lua_pushcclosure(L, fn, n)这样就形成的一个c闭包

- 返回值：无

- 栈影响：栈高 –(n - 1) , 一共弹出n个元素(及那些绑定的值), 压入一个cclosure

- 注

  ```
  #define lua_pushcfunction(L, f) lua_pushcclosure(L, f, 0)
  #define lua_register(L, n, f) (lua_pushcfunction(L, f), lua_setglobal(L, n))
  ```

  这个是比较常用的, 以n为lua中的key压入一个0个绑定值的cclosure.

```
void lua_call(lua_State* L, int nargs, int nresults)
```

- 操作说明

  ```
  argn = Stack.push()
  ... // 一共压入nargs个参数
  ```

```
arg2 = Stack.pop()
arg3 = Stack.pop()
func = Stack.pop() // 函数本身也弹出
res1， res2， ..., resj = func(arg1, arg2, ..., argn)
Stack.push(res1)
Stack.push(res2)
… // 压入nresults个返回值
Stack.push(resj)
```

- 返回值：无
- 栈影响：

  调用结束后，栈高度增加 nresults – (1 + nargs)，如果将 nresults 参数设置为 LUA_MULTRET，那么lua返回几个值，栈上就压入几个值，否者强制压入 nresults 个值，不足的是空值，多余的抛弃掉
- 注： 这个函数是有危险的，如果在其中发生了错误，会直接退出程序

```
int lua_pcall(lua_State* L, int nargs, int nresults, int errfunc)
```

- 操作说明：

  参数, 行为和 lua_call 都一样, 如果在调用中没有发生任何错误, lua_pcall == lua_call; 但是如果有错误发生时, lua_pcall会捕获它
  errfunc指出了 Stack 上的一个元素, 这个元素应该是一个函数, 当发生错误的时候

  ```
  ef = Stack[errfunc]
  value = ef(errmsg)
  Stack.push(value)
  ```

  也就是说, 在错误的时候, errfunc指定的错误处理函数会被调用, 该处理函数的返回值被压到栈上.

  默认情况下, 可以给errfunc传值 0, 实际的效果是指定了这样一个函数做出错处理 function defaulterr(errmsg) return errmsg end.
- 注

  本函数有返回值 LUA_ERRRUN运行时错误 LUA_ERRMEM内存分配错误[注意, 这种错会导致lua调用不了错误处理函数] LUA_ERRERR运行错误处理函数时出错了, 写程序的时候必须检查返回值:)

  强烈推荐该函数, 不过事实上大家也都用的这个函数:)

TODO

```
void lua_register (lua_State *L, const char *name, lua_CFunction f)
```

- 注
  #define lua_register(L,n,f) \
  (lua_pushcfunction(L, f), lua_setglobal(L, n))

```
void lua_setglobal (lua_State *L, const char *name);
```

```
void lua_getglobal (lua_State *L, const char *name);
```

- Pushes onto the stack the value of the global name.

```
void lua_getglobal (lua_State *L, const char *name);
```

- Pushes onto the stack the value of the global name.

```
void luaL_register (lua_State *L,
                const char *libname,
                const luaL_Reg *l);
```

Opens a library.

```
When called with libname equal to NULL, it simply registers all functions in t
he list l (see luaL_Reg) into the table on the top of the stack.

When called with a non-null libname, luaL_register creates a new table t, sets
it as the value of the global variable libname, sets it as the value of packag
e.loaded[libname], and registers on it all functions in the list l. If there i
s a table in package.loaded[libname] or in variable libname, reuses this table
instead of creating a new one.

In any case the function leaves the table on the top of the stack.
```

练习：

```
 int luaopen_lua_map(lua_State *l)
{
    const struct luaL_Reg _obj[] = {
       {"new", lua_map::new_object},
       {"__gc", lua_map::finalize},
       //{"insert",lua_map::insert},
       {"__newindex",lua_map::insert},
       {"erase",lua_map::erase},
       //{"size", lua_map::size},
       {"__len", lua_map::size},
       {"show", lua_map::show},
       {"stack_test",lua_map::tc_stack},
       {"finalize", lua_map::finalize},
       {NULL, NULL}
    };

    //[{}]
    luaL_newmetatable(l, lua_map::type_id);
    //[{"new":lua.....}]
    luaL_register(l, NULL, _obj);

    //{"new":lua.....}
    //{"new":lua.....}
    lua_pushvalue(l, -1);

    //{"new":lua.....}
    //{"new":lua....,__index:{"new":lua.....}

    //{"new":lua....,__index:{"new":lua.....}}
    lua_setfield(l, -2, "__index");


    //{}
    //{"new":lua....,index:{"new":lua.....}}
    lua_newtable(l);

    //{"new":lua....,index:self}
    //{}
    //{"new":lua....,index:self}]
    lua_pushvalue(l, -2);

    //{"new":lua....,index:self}
    //{lua_map:{"new":lua....,index:self},"new":lua....,index:self}
    //{"new":lua....,index:self}

    //[{lua_map:{"new":lua....,index:self},"new":lua....,index:self}
    //{"new":lua....,index:self}]
    lua_setfield(l, -2, "lua_map");

    //[{lua_map:{"new":lua....,index:self},"new":lua....,index:self}
```

```
    //{lua_map:{"new":lua....,index:self},"new":lua....,index:self}]
    lua_pushvalue(l, -1);

    //{map:{lua_map:{"new":lua....,index:self},"new":lua....,index:self}
    lua_setglobal(l, "map");
    return 1;
}
```