Lua源码剖析（一）

很早就想读lua的源码，也曾很多次浏览过大概。不过我一直没有深入去读，一是想自己在读lua源码之前，仅凭自己对lua使用的理解自己先实现一个简单的lua子集，二是我觉得自己实现过lua的子集之后也能帮助自己更容易的理解lua源码。前段时间，花了几个月的业余时间，实现了一个简单粗糙的lua子集（https://github.com/airtrack/luna）之后，我觉得现在可以开始读lua的源码了。

从lua.c的`main`函数开始，lua.c是一个stand-alone的解释器，编译完就是一个交互式命令行解释器，输入一段lua代码，然后执行并返回结果，也可以执行一个lua文件。

`main:`

```
/* call 'pmain' in protected mode */
lua_pushcfunction(L, &pmain);
lua_pushinteger(L, argc);  /* 1st argument */
lua_pushlightuserdata(L, argv); /* 2nd argument */
status = lua_pcall(L, 2, 1, 0);
result = lua_toboolean(L, -1);  /* get result */
```

`main`函数创建了`lua_State`之后就按照调用C导出给lua函数的方式调用了`pmain`函数。`pmain`函数中通过lua栈获取到命令行的argc和

argv参数之后，对参数进行分析后，主要可以分为两个分支，一个处理交互命令行，一个处理文件。`dotty`出来交互命令行，`handle_script`处理lua文件。

`handle_script:`

```
status = luaL_loadfile(L, fname);
lua_insert(L, -(narg+1));
if (status == LUA_OK)
    status = docall(L, narg, LUA_MULTRET);
else
    lua_pop(L, narg);
```

在`handle_script`中先loadfile，然后`docall`。

loadfile会产生一个什么东西在栈上呢？写过lua的程序的人估计都会了解到下面这段lua代码：

```
local f = load(filename)
f()
```

load会将文件chunk编译成一个function，然后我们就可以对它调用。如果我们详细看lua文档的话，这个函数可以带有upvalues，也就是这个函数其实是一个闭包（closure）。按照我自己实现的那个粗糙的lua子集的方式的话，每个运行时期的可调用的lua函数都是闭包。

```
#define luaL_loadfile(L,f)     luaL_loadfilex(L,f,NULL)
```

`luaL_loadfilex:`

```
if (filename == NULL) {
  lua_pushliteral(L, "=stdin");
  lf.f = stdin;
}
else {
  lua_pushfstring(L, "@%s", filename);
  lf.f = fopen(filename, "r");
  if (lf.f == NULL) return errfile(L, "open", fnameindex);
}
if (skipcomment(&lf, &c))  /* read initial portion */
  lf.buff[lf.n++] = '\n';  /* add line to correct line numbers */
if (c == LUA_SIGNATURE[0] && filename) {  /* binary file? */
  lf.f = freopen(filename, "rb", lf.f);  /* reopen in binary mode */
  if (lf.f == NULL) return errfile(L, "reopen", fnameindex);
  skipcomment(&lf, &c);  /* re-read initial portion */
}
if (c != EOF)
  lf.buff[lf.n++] = c;  /* 'c' is the first character of the stream */
status = lua_load(L, getF, &lf, lua_tostring(L, -1), mode);
```

`luaL_loadfile` 是一个宏，实际是 `luaL_loadfilex` 函数，在 `luaL_loadfilex` 函数中，我们发现是通过调用 `lua_load` 函数实现， `lua_load` 的函数原型是：

```
LUA_API int lua_load (lua_State *L, lua_Reader reader,
```

```
        void *data, const char *chunkname, const char *mode);
```

定义在lapi.c中，它接受一个 `lua_Reader` 的函数并把data作为这个reader的参数。在 `luaL_loadfilex` 函数中传给 `lua_load` 作为reader是一个static函数 `getF` ， `getF` 通过 `fread` 读取文件。

`lua_load:`

```
ZIO z;
int status;
lua_lock(L);
if (!chunkname) chunkname = "?";
luaZ_init(L, &z, reader, data);
status = luaD_protectedparser(L, &z, chunkname, mode);
```

在函数 `lua_load` 中，又将 `lua_Reader` 和data通过 `luaZ_init` 函数把数据绑定到ZIO的结构中，ZIO是buffered streams。之后调用 `luaD_protectedparser` ，此函数定义在ldo.c中，在这个函数中，我们发现它使用了构造 `lua_Reader` 和data的方式构造了调用函数 `f_parser` 和它的数据SParser，并将它们传给 `luaD_pcall` ， `luaD_pcall` 的功能是在protected模式下用 `SParser` 数据调用 `f_parser` 函数，因此我们只需追踪 `f_parser` 函数即可。

`luaD_protectedparser:`

```
status = luaD_pcall(L, f_parser, &p, savestack(L, L->top), L->errfunc);
```

`f_parser:`

```
if (c == LUA_SIGNATURE[0]) {
  checkmode(L, p->mode, "binary");
  cl = luaU_undump(L, p->z, &p->buff, p->name);
}
else {
  checkmode(L, p->mode, "text");
  cl = luaY_parser(L, p->z, &p->buff, &p->dyd, p->name, c);
}
```

`f_parser` 通过数据头的signature来判断读取的数据是binary还是text的，如果是binary的数据，则调用 `luaU_undump` 来读取预编译好的lua chunks，如果是text数据，则调用 `luaY_parser` 来parse lua代码。我们发现 `luaU_undump` 和 `luaY_parser` 函数的返回值都是 `Closure *` 类型，这个刚好就和我们前面预计的一样，一个chunk load之后返回一个闭包。

进入 `luaY_parser` 函数后，就调用了一个static的 `mainfunc` 开始parse lua代码。

仔细回顾上面看过的函数，我们会发现每个C文件的导出函数都会使用lua开头，如果没有lua开头的函数都是static函数。并且我们会发现lua后的大写前缀可以标识这个函数所属的文件：

`luaL_loadfile` `luaL_loadfilex` L应该是library的意思，属于 `lauxlib`

`luaD_protectedparser` `luaD_pcall` D是do的意思，属于 `ldo`

`luaU_undump` U 是undump的意思，属于 `lundump`

`luaY_parser` Y 是代表yacc的意思，lua的parser最早是用过yacc生成的，后来改成手写，名字也保留下来，属于 `lparser`

其它的lua函数也都有这个规律。