Lua提供高级的require函数来加载运行库。粗略的说require和dofile完成同样的功能但有两点不同：

1.require会搜索目录加载文件;

2. require会判断是否文件已经加载避免重复加载同一文件。

由于上述特征，require在Lua中是加载库的更好的函数。

**(一) require**

　　require使用的路径和普通我们看到的路径还有些区别，我们一般见到的路径都是一个目录列表。require的路径是一个模式列表，每一个模式指明一种由虚文件名（require的参数）转成实文件名的方法。更明确地说，每一个模式是一个包含可选的问号的文件名。匹配的时候Lua会首先将问号用虚文件名替换，然后看是否有这样的文件存在。如果不存在继续用同样的方法用第二个模式匹配。例如，路径如下：

?;?.lua;c:\windows\?;/usr/local/lua/?/?.lua

调用require "test"时会试着打开这些文件：

test

test.lua

c:\windows\test

/usr/local/lua/test/test.lua

require关注的问题只有分号（模式之间的分隔符）和问号，其他的信息（目录分隔符，文件扩展名）在路径中定义。

　　为了确定路径，Lua首先检查全局变量LUA\_PATH是否为一个字符串，如果是则认为这个串就是路径；否则require检查环境变量LUA\_PATH的值，如果两个都失败；require使用固定的路径（典型的"?;?.lua"）

　　require的另一个功能是避免重复加载同一个文件两次。Lua保留一张所有已经加载的文件的列表（使用table保存）。如果一个加载的文件在表中存在, 则require简单的返回；表中保留加载的文件的虚名，而不是实文件名。所以如果你使用不同的虚文件名require同一个文件两次，将会加载两次该文件。比如require "foo"和require "foo.lua"，路径为"?;?.lua"将会加载foo.lua两次。我们也可以通过全局变量\_LOADED访问文件名列表，这样我们就可以判断文件是否被加载过；同样我们也可以使用一点小技巧让require加载一个文件两次。比如，require "foo"之后\_LOADED["foo"]将不为nil，我们可以将其赋值为nil，require "foo.lua"将会再次加载该文件。

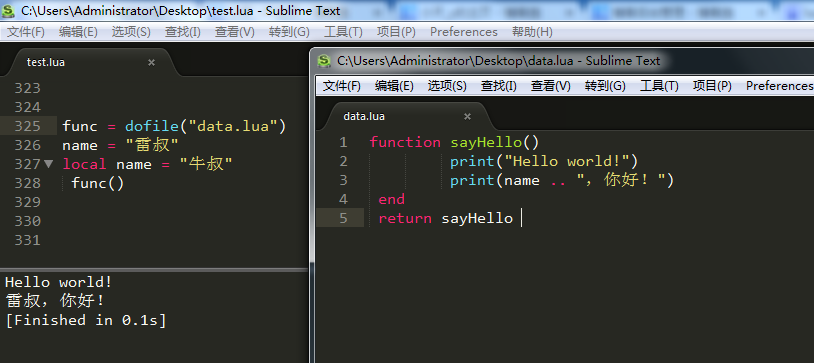
一个路径中的模式也可以不包含问号而只是一个固定的路径，比如：

?;?.lua;/usr/local/default.lua

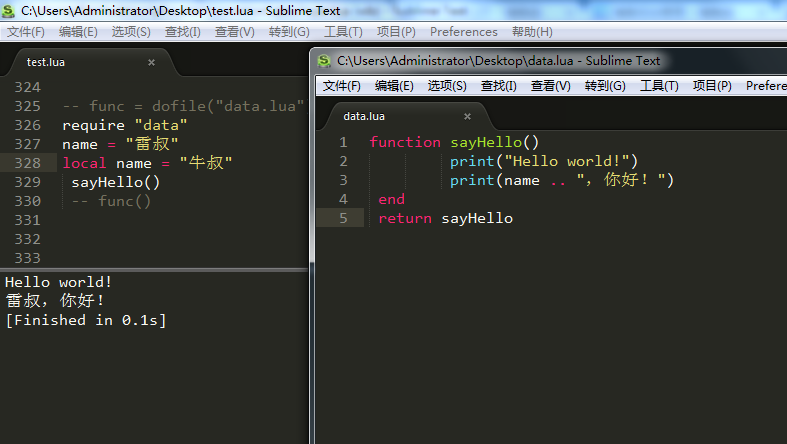
　　这种情况下，require没有匹配的时候就会使用这个固定的文件（当然这个固定的路径必须放在模式列表的最后才有意义）。在require运行一个chunk以前，它定义了一个全局变量\_REQUIREDNAME用来保存被required的虚文件的文件名。我们可以通过使用这个技巧扩展require的功能。举个极端的例子，我们可以把路径设为"/usr/local/lua/newrequire.lua"，这样以后每次调用require都会运行newrequire.lua，这种情况下可以通过使用\_REQUIREDNAME的值去实际加载required的文件。

**(二) dofile**

　　我们知道一个lua文件是作为一个代码块（chunk）存在的，其实质就是一个函数，那么最简单的，我在一个外部lua文件中写一段代码，然后在主lua文件中用dofile调用，外部文件的代码块就会执行了。



外部lua文件在编译时并没有涉及词法域：



　　形式上很类似C语言的#include<...>，在其他地方定义的函数，经这么引入文件之后就可以调用了。不过lua并不是定义和实现分离的语言，这样是把整个定义部分都加载进来了。加载过程大致上是: lua先加载这个外部文件，然后运行它。实际上这段外部代码是可以有返回值的，它的返回值就是require的返回值。这里我们什么返回值都没有，执行这个外部代码的结果就是定义了这么个全局函数。注意是全局函数，虽然通常我们直接定义的函数 都是全局函数所以都没怎么注意过，要是非要定义个局部函数在主程序块里可就看不到了。另外和前面的情形一样，外部代码块只认识“雷叔”，“牛叔”是谁它根本不知道。  
        可以在外部文件里定义一堆函数，然后全都加到全局环境下。不过全局的东西用起来要小心，有一个原则是对全局的“污染”越小越好。那么自然就引入了“模块” 的概念。在lua中，模块由万能的table来充当。最自然的想法就是定义一个table，然后把要定义的函数放在这个table里，最后返回这个 table就行了。

**(三) lua中的require机制**  
    为了方便代码管理，通常会把lua代码分成不同的模块，然后在通过require函数把它们加载进来。现在看看lua的require的处理流程。

**1、require机制相关的数据和函数**  
　　package.path : 保存加载外部模块(lua中"模块"和"文件"这两个概念的分界比较含糊，因为这个值在不同的时刻会扮演不同的角色)的搜索路径，这种路径是"模板式的路径"，它里面会包含可替代符号"?", 这个符号会被替换，然后lua查找这个文件是否存在，如果存在就会调用其中特定的接口。典型的值为:

"./?.lua;./?.lc;/usr/local/?/init.lua"

如果lua代码中调用:require("hello.world"), 那么lua会依次查找：

./hello/world.lua -- 这里"hello.world"变成了"hello/world",并替换了模型"./?.lua"

./hello/world.lc

.....

　　(这种处理方式和python类似，只不过不需要\_\_init\_\_.py,也有调用python中的\_\_init\_\_.py) package.path在虚拟机启动的时候设置，如果存在环境变量LUA\_PATH，那么就用该环境变量作为它的值，并把这个环境变量中的";;"替换为luaconf.h中定义的默认值，如果不存在该变量就直接使用luaconf.h定义的默认值.      
    package.cpath:作用和packag.path一样,但它是用于加载第三方c库的。它的初始值可以通过环境变量LUA\_CPATH来设置;  
    package.loadlib(libname, func):相当与手工打开c库libname, 并导出函数func返回，loadlib其实是ll\_loadlib;  
      
**2.require的处理流程：**

require(modelname)

require(在lua中它是ll\_require函数)的查找顺序如下：  
       a. 首先在package.loaded查找modelname,如果该模块已经存在，就直接返回它的值;  
       b. 在package.preload查找modelname, 如果preload存在，那么就把它作为loader，调用loader(L);  
       c. 根据package.path的模式查找lua库modelname，这个库是通过module函数定义的，对于顶层的lua库，文件名和库名是一样的而且不需要调用显式地在lua文件中调用module函数(在ll\_require函数中可以看到处理方式)，也就是说lua会根据lua文件直接完成一个loader的初始化过程;  
       d. 根据package.cpath查找c库，这个库是符合lua的一些规范的(export具有一定特征的函数接口)，lua先已动态的方式加载该c库，然后在库中查找并调用相应名字的接口，例如:luaopen\_hello\_world;  
       e. 以第一个"."为分割，将模块名划分为:(main, sub)的形式，根据package.cpath查找main，如果存在，就加载该库并查询相应的接口:luaopen\_main\_sub，例如：先查找hello库，并查询luaopen\_hello\_world接口  
       f. 得到loder后，用modname作为唯一的参数调用该loader函数。当然参数是通过lua的栈传递的，所以loader的原型必须符合lua的规范:int LUA\_FUNC(lua\_State \*L)  
           
      ll\_require会将这个loader的返回值赋给package.loaded[modelname],如果loader不返回值同时package.loaded[modelname]不存在时, ll\_require就会把package.loaded[modelname]设为true。最后ll\_reuqire把package.loaded[modelname]返回给调用者。  
  
**3.module的处理流程**

module(name, cb1, cb2, ...)

　　a. 如果package.loaded[name]是一个table，那么就把这个table作为一个mod  
　　b. 如果全局变量name是一个table，就把这个全局变量作为一个mod  
　　c. 创建table:t = {[name]=package.loaded[name], ["\_NAME"]=name, ["\_M"]=t, ["\_PACKAGE"]=\*name\*(删除了最后的".XXXX"部分)}. 如果name是一个以点分割的串，那么得到的mod类似这个样子：

hello.world -- {["hello"]={["world"]={XXXXXXX}}}

　　d. 依次调用cbs：

cb1(mod), cb2(mod),...

　　e. 将当前模块的环境设置为mod,同时把package.loaded[name] = mod